SCIENCES SUP

Cours et exercices avec solutions

Master • Écoles d'ingénieurs

THÉORIE DES CODES

Compression, cryptage, correction

Jean-Guillaume Dumas
Jean-Louis Roch
Éric Tannier
Sébastien Varrette

DUNOD

THÉORIE DES CODES

Compression, cryptage, correction

Jean-Guillaume Dumas
Maître de conférences à l'université Grenoble 1

Jean-Louis Roch
Maître de conférences à l'ENSIMAG

Éric Tannier
Chercheur de l'INRIA Rhône-Alpes à l'université Lyon 1

Sébastien Varrette
Doctorant à l'université du Luxembourg

DUNOD

Illustration de couverture :
Antenne-relais de transmission sur le mont Pisgah dans les Appalaches.
Photographie de Jean-Guillaume Dumas.

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 2007
ISBN 9-78-210-050692-7

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

Préambule

Cet ouvrage a été initié au printemps 2000, avec la création du Département ENSIMAG-ENSERG Télécommunications à l'Institut National Polytechnique de Grenoble et la construction d'un cours général de première année (niveau Licence troisième année dans le schéma LMD) d'introduction aux codes et à leurs applications.
Édité initialement sous forme de polycopié, il a évolué en devenant un support de cours de référence pour différents cours des universités de Grenoble, à la fois à l'Institut National Polytechnique de Grenoble (INPG) et à l'Université Joseph Fourier (UJF).
Nous remercions nos collègues qui ont participé à ces enseignements et ont contribué par leurs commentaires à améliorer notre support de cours : Gilles Debunne, Yves Denneulin, Dominique Duval, Grégory Mounié et Karim Samaké.

Grenoble, Lyon, Luxembourg, le 19 décembre 2006.
Jean-Guillaume Dumas, Jean-Louis Roch, Éric Tannier, Sébastien Varrette.

Sommaire

Introduction

Ce livre a pour objet la transmission automatique d'informations numériques, et se focalisera sur la structure de l'information, indépendamment du type de support de transmission. L'information peut être de n'importe quel type pourvu qu'on puisse en donner une représentation numérique : textes, images, sons, vidéos par exemple. La transmission de ces types de données est omniprésente dans la technologie, et spécialement dans les télécommunications. Il est donc nécessaire de s'appuyer sur de bonnes bases théoriques pour que cette transmission soit fiable, ce dernier terme se voyant donner plusieurs significations qui sont les objectifs qui nous guideront tout au long de l'ouvrage.
Les canaux de transmission peuvent également être de tous types (réseaux de câbles ou d'ondes), via éventuellement un support de stockage. Nous ne considérons pas les problèmes physiques que posent la transmission, qui font l'objet de la théorie dite « du signal ». La théorie des codes, elle, traite de la forme de l'information elle-même quand elle doit être transmise, ou stockée.

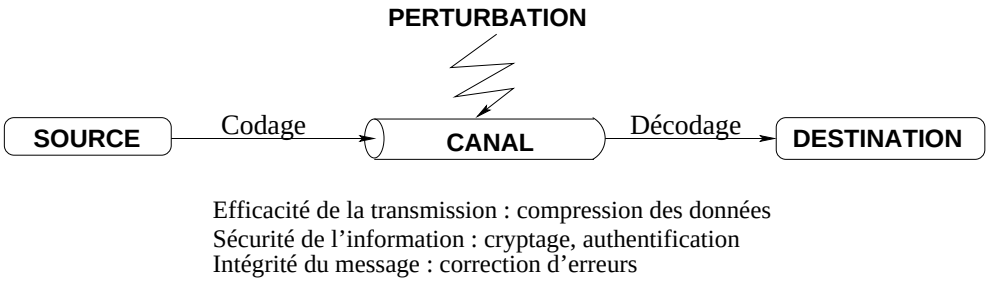


Fig. 1: Schéma fondamental du codage.

La communication d'une information commence par sa formulation par un émetteur, se poursuit par un transit via un canal, et se termine par la reconstitution du message par le destinataire. Parfois l'émetteur et le destinataire sont confondus, s'il s'agit d'un processeur ou d'une personne qui enregistre des données dans un registre, une mémoire ou un disque puis les lit ultérieurement.

L'information doit être reçue par son destinataire dans son intégralité, en sécurité, et le plus rapidement possible. Or quel que soit le canal considéré, avec des variations selon le support, on ne peut jamais le supposer « sûr », dans plusieurs sens du terme : il y a toujours des erreurs au cours des transmissions, et le message est susceptible d'être lu, voire altéré, par des tiers plus ou moins bien intentionnés. Il s'agit alors pour l'émetteur d'envoyer ses messages sous une forme qui permette au destinataire de les reconstituer en tenant compte d'éventuelles altérations au cours du transfert ou de la confidentialité de certaines informations, tout en minimisant la taille des données. Ces contraintes sont les points de départs de plusieurs champs d'étude en mathématiques et en informatique, souvent développés indépendamment les uns des autres bien qu'ils traitent des mêmes objets. Le but de ce livre est de les rassembler en un seul volume afin de présenter une seule théorie, dont l'objet est la forme donnée à l'information au cours de sa transmission, c'est-à-dire une théorie générale des *codes*.

En 1948, Claude Shannon posait la première pierre de ce qu'il a appelé une « théorie mathématique de l'information ». On raconte que sa théorie est née de réflexions sur la langue (l'anglais, en l'occurrence) : Shannon s'amusait à masquer une proportion variable des textes qu'il lisait, et à les reconstituer à partir de la partie visible. En enlevant quelques mots, il pouvait reconstituer le sens de la phrase de façon certaine. C'est que ces mots cachés étaient *redondants*, ils n'apportaient rien au sens du message. Si il en enlevait trop, il lui était impossible de reconstituer le message avec certitude. Shannon mit alors au point une théorie qui serait à même de calculer la « quantité d'information » dans n'importe quel type de message, ce qui revient à déterminer son taux de redondance.
Réduire cette redondance est ce qu'aujourd'hui nous appelons la *compression*, ou « théorie de l'information », par fidélité historique, et bien que ce titre puisse convenir à l'ouvrage entier, qui dépasse largement cette théorie. Dans un souci d'efficacité, qui relève de l'optimisation de l'espace de stockage, et plus encore de la rapidité de transmission, il s'agit de rendre le message le plus court possible, en ne gardant que ce qui est indispensable à sa compréhension, ou mieux, en le reformulant sans redondance.
La confidentialité dont on veut entourer la transmission d'une information est une préoccupation beaucoup plus ancienne que celle de l'efficacité. Partant du principe que les routes (comme les canaux numériques) ne sont pas sûres, et qu'un message peut être intercepté au cours de sa transmission, il faut transformer le texte, le décorréler de sa signification, en ne laissant qu'à ses destinataires la clef de son déchiffrement. Les histoires de codes secrets, et des batailles entre inventeurs de codes et briseurs de codes (qui cherchent à connaître la signification d'un message sans en avoir la clef) parsèment l'his-

toire des sociétés et des états. Shannon contribua également à ce domaine en apportant pour la première fois, en 1949, une preuve théorique de confidentialité. Une discipline scientifique est aujourd'hui consacrée aux codes secrets, la *cryptologie*. Non seulement les méthodes actuelles garantissent le *secret* sur la signification d'un message, mais elles permettent aussi de *signer* des documents ou d'*identifier* un émetteur.
Tous les canaux permettant de transmettre une information numérique peuvent être soumis à des perturbations qui sont susceptibles d'altérer une partie des messages, et donc d'en modifier le sens. Si les informations sont envoyées sans redondance, la plus petite altération peut en effet entraîner des fausses interprétations à l'arrivée. Dans le cas d'un texte en langue naturelle, la plupart des erreurs n'altéreront pas la perception du lecteur, car la redondance lui permettra de reconstituer le message original. Shannon, lui encore, a montré un résultat sensationnel en 1948 : même sur des canaux dont le taux d'erreur est très important, il est possible d'ajouter suffisamment de redondance pour que le message soit reçu dans son intégralité. Mais sa démonstration n'est pas constructive et ce théorème motive toujours la construction de méthodes incluant de façon ordonnée et optimisée de la redondance afin que le destinataire puisse soit s'apercevoir que le message a été altéré (*codes détecteurs*), soit corriger lui-même les erreurs éventuelles (*codes correcteurs*). Toutes ces méthodes sont paramétrables, c'est-à-dire adaptables au type de support considéré, et à son taux d'erreur.

Ce sont ces trois préoccupations — l'*efficacité*, la *sécurité*, l'*intégrité* — qui retiennent l'attention des concepteurs de méthodes de transmission de l'information. L'intérêt de les présenter ensemble repose sur leur objet commun, le *code*, qui structure l'information dans tous les supports technologiques actuels, et sur l'utilité de disposer dans un seul volume de toutes les façons de manipuler l'information avant de l'envoyer ou de la recevoir.
Le socle sur lequel s'appuie la théorie des codes est issu de l'algèbre linéaire, de l'arithmétique, des probabilités, de l'algorithmique et de la combinatoire. Les modèles mathématiques et les premiers développements algorithmiques qui structurent la notion de code sont introduits dans le premier chapitre de cet ouvrage. La présentation de ces modèles est assortie d'introductions des mathématiques utiles à leur manipulation, ainsi que de notions générales sur l'efficacité des méthodes de calcul, dont nous ferons un usage intensif tout au long de l'ouvrage. Il sera utile pour lire ce chapitre de disposer d'un bagage théorique de base en algèbre linéaire, ce qui rend ce livre accessible après une licence scientifique. Certains éléments dépassent les connaissances classiques des étudiants non mathématiciens, ils sont donc présenté en détails ici. Le lecteur se rendra rapidement compte de leur réelle efficacité pratique, le plus souvent au moment même de leur introduction. Pour clarifier encore l'utilité

et les dépendances entre les notions introduites, la figure 2 les représente.

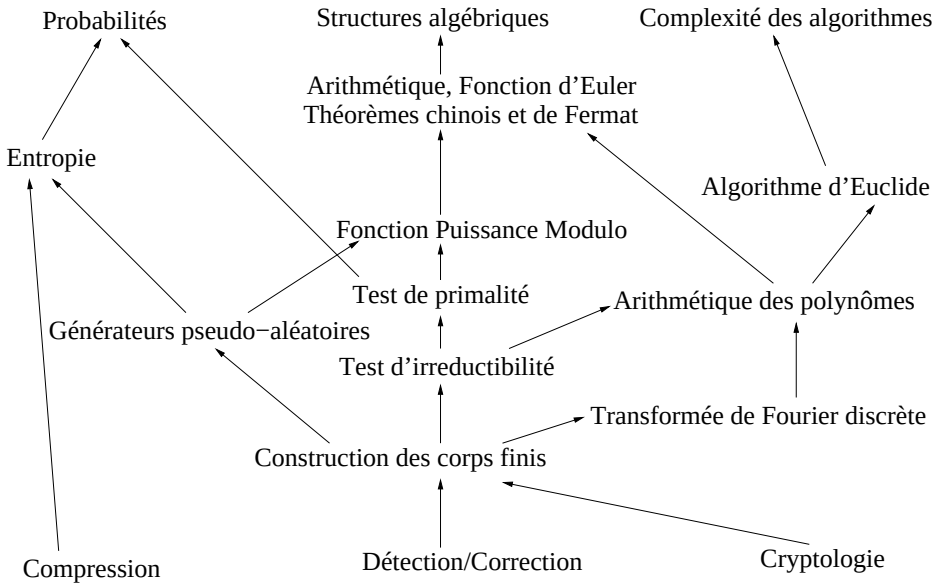


FIG. 2: Schéma des notions introduites dans le premier chapitre, avec leurs dépendances.

Cette figure permettra aussi au lecteur pressé ou intéressé par une partie de l'ouvrage seulement de retrouver son chemin, la lecture linéaire n'étant pas obligatoire. Même si ce premier chapitre est conçu pour introduire la théorie des codes, il peut servir de boîte à outils de référence lors de la lecture des chapitres suivants. Ceux-ci reprennent séparément la compression, la cryptographie, et les codes détecteurs et correcteurs d'erreur. Ils présentent les résultats théoriques fondamentaux et les algorithmes qui en découlent. Chacun de ces trois chapitres est illustré par des exemples d'utilisation pratique dans le contexte des télécommunications. Nous nous sommes efforcés de présenter à la fois les théories classiques du codage et les développements les plus récents sur le sujet, dans la mesure où un tel livre d'introduction le permettait.
En plus de réunir des théories mathématiques qui partagent le même objet, le credo de cet ouvrage est algorithmique. Ici, les propriétés mathématiques des fonctions servent à rendre efficaces leurs calculs. Les méthodes de calcul sont toujours données de façon détaillée, et immédiatement implémentables en n'importe quel langage de programmation. L'efficacité des méthodes est toujours calculée et discutée, les implémentations existantes comparées. Nos sciences sont à la fois les héritières des mathématiques grecques, qui faisaient de l'esthétique de leurs énoncés leur principale qualité, et des mathématiques

orientales, mues plus souvent par l'utilité et le calcul. C'est encore ce qui peut unir toutes les théories des codes, et c'est un de leurs grands mérites, de faire appel à des mathématiques rigoureuses et appréciées des esthètes pour construire des méthodes efficaces appliquées dans la communication courante. Ce livre est ainsi au confluent de ces fleuves, il attirera les férus de technologie autant que les amateurs de théorie des nombres, sans compter ceux que les histoires de déchiffrement de langages obscurs, de machines qui corrigent leurs propres erreurs et de codes secrets font toujours rêver.

Chapitre 1

Théorie des codes

On appellera un *code* une méthode de transformation qui convertit la représentation d'une information en une autre. Cette définition est suffisamment large pour accueillir plusieurs objets mathématiques (fonctions, algorithmes, transformées), dont nous ferons usage tout au long de ce texte. Le mot *code* servira aussi à désigner le produit de ces transformations, c'est-à-dire l'information codée et sa structure.
Mais pour commencer à se retrouver dans ces définitions et comprendre ce qu'est vraiment le code, voici un exemple simple mélangeant les technologies et les époques.

1.1 De Jules César à la télécopie

Pour introduire toutes les propriétés importantes des codes, en voici un construit à partir d'exemples réellement utilisés, actuellement ou par le passé. Supposons que nous voulions envoyer une information par fax tout en assurant le secret de la transmission. Voici ce que pourrait être un code qui réaliserait cette opération.

1.1.1 La source : de l'image à la suite de pixels

La transmission par fax permet de faire transiter des images par un canal téléphonique. Ce qu'on demandera à ce code sera d'assurer les transformations nécessaires pour obtenir à l'arrivée une image semblable à l'originale, rapidement et secrètement, dans un format qui puisse transiter par le canal.
Le premier procédé de transformation sera assuré par le scanner de l'appareil. Il consiste à lire l'image et à la transformer en une suite de pixels, qu'on peut se représenter comme des petits carrés soit noirs, soit blancs. C'est un code,

selon la définition que nous en avons donnée, et nous l'écrirons comme un algorithme, dans un format que nous adopterons tout au long de cet ouvrage.

Algorithme de codage

Entrées Une image
Sorties Une suite de pixels

L'entrée de l'algorithme est aussi appelée la *source*, et la sortie le *code*[1].
La méthode adoptée sera la suivante : la largeur de l'image source est divisée en 1728 parties égales ; la longueur est ensuite divisée en lignes, de façon à former des carrés de même taille. Ces carrés seront les pixels.
Chaque pixel se voit attribuer la couleur noire si à cet endroit l'image est suffisamment foncée, et la couleur blanche si l'image est claire. C'est la première partie de notre code.

1.1.2 La compression du message

Pour formaliser les messages sources et les codes, on définit le langage dans lequel ils sont exprimés. On appelle *alphabet* un ensemble fini $V = \{v_1, \ldots, v_k\}$ d'éléments appelés *caractères*. Le *cardinal* d'un ensemble fini V est son nombre d'éléments, noté $|V|$.
Une suite de caractères de V est une *chaîne*. On note V^* l'ensemble des chaînes sur V, et V^+ l'ensemble des chaînes de longueur non nulle. Comme l'alphabet du code et l'alphabet de la source peuvent différer, on parle d'*alphabet source* et d'*alphabet du code*. Par exemple, pour la méthode de compression que nous allons voir maintenant, l'alphabet source est le résultat du scanner, à savoir $\{pixel\ blanc, pixel\ noir\}$, et l'alphabet du code sera l'ensemble de bits $\{0, 1\}$. On pourrait envoyer la suite de pixels sous forme de bits numériques, puisqu'ils sont immédiatement traduisibles en 0 (pour un pixel blanc) et 1 (pour un pixel noir).
Mais on peut alors appliquer quelques principes simples à cette suite de 0 et de 1, afin de la compresser. Si on imagine que la page à envoyer ne comporte que quelques lignes de texte, et qu'une bonne partie est blanche, ce qui arrive très souvent dans les fax, alors n'y a-t-il pas de meilleur moyen de transmettre une suite de, mettons, 10000 zéros, que d'employer effectivement les 10000 zéros ? Bien sûr, il y a un meilleur moyen, et nous venons d'ailleurs de l'employer, car nous avons évoqué cette longue suite sans l'écrire explicitement. Il s'agit simplement d'indiquer que le code est composé de 10000 zéros plutôt que de

[1] Le mot *code* est très accueillant. Il désigne aussi bien l'ensemble des transformations que leur résultat, ainsi que, souvent, un programme informatique. Cette apparente confusion permet en fait de saisir les liens entre divers procédés mathématiques et informatiques, et c'est pourquoi nous conservons le terme dans ses multiples acceptions.

les écrire tous. Le code du fax réalise ce principe de compression ligne par ligne, c'est à dire par blocs de 1728 pixels. Pour une ligne, on peut décrire précisément l'algorithme de codage par l'algorithme 1.

Algorithme 1 Codage fax simplifié.

Entrées Une suite de 1728 pixels, noté M
Sorties Le message M compressé
  Soit n un compteur de pixels consécutifs, initialisé à 0
  Pour un indice i parcourant la ligne de pixels (de 1 à 1728) **Faire**
    Si le pixel d'indice i est le même que le pixel d'indice $i-1$ **Alors**
      $n \leftarrow n+1$
    Sinon
      Écrire n et la couleur du dernier pixel dans M
      $n \leftarrow 0$
    Fin Si
  Fin Pour

Ainsi, par exemple, une ligne entièrement blanche ne sera plus codée par une suite de 1728 zéros, mais par le code "1728 0", soit, dans un format numérique qui pourra être envoyé par le canal, "11011000000 0", puisque c'est la représentation de 1728 en binaire. On a donc remplacé un message de 1728 bits en un message de 12 bits, ce qui est nettement plus court. C'est donc un principe dit de *compression* des messages, appelé *RLE* pour *Run-Length Encoding*, et que nous détaillerons dans le chapitre 2.3.1. Nous nous sommes servis d'un caractère d'espacement dans notre représentation (entre 1728 et 0), pour une meilleure visibilité, mais ce caractère ne fait pas partie de l'alphabet du code. En pratique, il faudra donc le supprimer. Nous verrons les contraintes que cela implique un peu plus tard dans ce chapitre.

Exercice 1.1. *Une image pixelisée destinée à être faxée contient beaucoup de paires de pixels consécutifs "01".*
– *Que pensez-vous du code fax présenté plus haut ?*
– *Proposez un code plus efficace.* *Solution page 279.*

1.1.3 La détection d'erreurs

Tous nos lecteurs qui ont déjà utilisé un canal téléphonique ne seront pas étonnés d'apprendre que sa fiabilité n'est pas infinie. Tout message est susceptible d'être altéré, et il n'est pas impossible que si "11011000000 0" est envoyé sur le canal numérique, "11010000000 0" (altération d'un bit) ou "1101000000 0" (destruction d'un bit) soit reçu. Habituellement, les canaux téléphoniques ont un taux d'erreur variant de 10^{-4} à 10^{-7}, selon leur nature, ce qui veut dire

qu'ils peuvent commettre en moyenne une erreur tous les 10000 bits. C'est loin d'être négligeable, quand on envoie de longs messages, et cela peut évidemment en altérer le sens. Pour une image, si le 1728 envoyé est devenu 1664 à cause de l'altération d'un seul bit, il va se produire un décalage de l'image à l'arrivée qui rendra le résultat inexploitable.
Le code du fax permet de détecter de telles erreurs de transmission. En cas d'erreur détectée sur une ligne, ceci permet de redemander la transmission de la même ligne pour avoir une confirmation, et comme il est très peu probable qu'une erreur se produise deux fois exactement au même endroit, le message pourra être corrigé.
Le principe de la détection d'erreur dans le code du fax est le suivant : on fait précéder et suivre dans le message chaque ligne par une suite identifiable. Même si ce ne sont pas exactement ces nombres-là qui sont utilisés en pratique, mettons, pour expliquer le principe, que "0" est ajouté au début de chaque ligne, et "1729" est ajouté à la fin de chaque ligne.
On peut alors, pour chaque ligne reçue, vérifier qu'elle est de la forme "0 n_1 b_1 ... n_k b_k 1729", où n_i est un nombre entier qui donne le nombre de bits consécutifs, et b_i la couleur de ces bits. En particulier, on doit avoir $n_1 + \cdots + n_k = 1728$, et les b_i doivent être alternés. Ainsi, on repère facilement l'altération ou la perte d'un bit, dès que ce format n'est pas respecté.
Les principes de détection et correction d'erreur, étudiés plus en détail au chapitre 4, sont tous basés sur ce même principe : ajouter de l'information pour vérifier la cohérence du message reçu.

1.1.4 Le chiffrement

Supposons maintenant, après avoir compressé le message et apporté un format qui permette la détection d'erreurs, que nous voulions garder le message secret pour toute personne excepté son destinataire. Le canal téléphonique, comme la plupart des canaux, ne permet pas le secret en lui-même. Tout message qui y transite peut être facilement lu par un tiers. La mise en œuvre du secret consiste à transformer le message, le mettre sous une forme incompréhensible, et à le retransformer dans sa forme originelle à l'arrivée.
C'est une technique employée par les hommes depuis qu'ils communiquent. Dans les codes secrets employés dans l'antiquité, le secret résidait dans la technique mise en œuvre, qu'on appelle aujourd'hui *algorithme de chiffrement*. Ainsi, les historiens ont retrouvé des messages codés par les services de Jules César. Les messages étaient des textes, et l'algorithme substituait chaque lettre du message initial M par celle située 3 positions plus loin dans l'alphabet. Pour les deux dernières lettres de l'alphabet, il fallait renvoyer aux deux premières. Par exemple le mot $TROYEN$ devenait $WURBHQ$. Ainsi, le texte n'avait

plus de signification immédiate. C'est ce qu'on appelle un principe de *substitution mono-alphabétique*, car chaque lettre est remplacée par une autre, toujours la même, dans le message.
Si César avait voulu envoyer un fax, il aurait adapté son code au format numérique, ce qui aurait donné la fonction $f(x) = x + 3 \mod n$ pour tous les nombres envoyés sur le canal. Le nombre n est la taille de l'alphabet utilisé, ici par exemple 1730, puisqu'aucun nombre supérieur ne peut théoriquement être employé.
Ces fonctions de codage et décodage furent ensuite paramétrées par une simple clef K, un nombre entier choisi secrètement entre les communiquants. Cela revient à construire une fonction $f_K(x) = x + K \mod n$. Les spartiates utilisaient eux un algorithme de chiffrement totalement différent, le chiffrement par transposition : la scytale était un bâton sur lequel était enroulé un parchemin. Le message était écrit sur le parchemin enroulé mais le long du bâton et non le long du parchemin. Ainsi les lettres successives du message apparaissait sur une circonvolution différente du parchemin. Pour le déchiffrer, le destinataire devait posséder un bâton d'un diamètre identique à celui utilisé pour l'encodage. D'autres systèmes cryptographiques, plus élaborés, virent alors le jour (chiffrements affines $f_{a,b}(x) = a.x + b \mod n$ - étudiés dans les exercices 1.2 et 3.1 ; chiffrements par substitution où chaque lettre est remplacée par un symbole d'un autre alphabet comme le code morse etc...).

Exercice 1.2. *Marc-Antoine intercepte un message envoyé par César, crypté par chiffrement affine. Sans connaître la clef (a, b), comment décode-t-il le message ?* *Solution page 279.*

1.1.5 Le décodage

Nous avons vu dans ce qui précède toutes les techniques pour coder un message et en assurer l'efficacité de transmission et le secret. Cette description ne serait pas complète sans un mot pour présenter les principes de l'opération complémentaire, le *décodage*.

L'attaque et le déchiffrement

Le message arrive crypté. Il s'agit de le décoder. Pour qui ne possède pas la clef ou la méthode de chiffrement, on parle d'*attaque* sur le code, ou de *cassage* de code. La discipline qui consiste à inventer des méthodes d'attaque pour briser les codes existants ou pour construire des codes plus résistants est la *cryptanalyse*. Pour le destinataire du message qui connaît la clef, il s'agit de *déchiffrement*. On parlera de l'attaque bientôt. Pour l'instant, la méthode de déchiffrement est très simple et consiste à appliquer la fonction

inverse de la fonction de codage, éventuellement paramétrée par une clef, soit $f_K^{-1}(x) = x - K \mod n$.
Le message est alors déchiffré.Il reste à appliquer toutes les autres transformations en sens inverse. Tout d'abord, vérifier le format de chaque ligne pour détecter d'éventuelles erreurs (en cas d'erreur, redemander une émission), puis appliquer un algorithme de décodage, qui, étant donnée une suite de nombres, renverra la suite de pixels initiale. C'est ce qui est formalisé pour une ligne dans l'algorithme 2.

Algorithme 2 Décodage fax.

Entrées Une suite de nombres, sous la forme "0 n_1 b_1 ... n_k b_k 1729"
Sorties La suite des 1728 pixels correspondants
  Pour un indice i de 1 à k **Faire**
    Dessiner n_i pixels de couleur b_i
  Fin Pour

La décompression et les pertes

En appliquant cet algorithme pour toutes les lignes, on obtient une suite de pixels, qui est la même que celle de départ, en laquelle nous avions transformé l'image. Il ne reste plus qu'à restituer l'image. Ou plutôt une image semblable à l'image de départ. En effet, la seule information dont nous disposons est la valeur des pixels, et la méthode consiste donc à imprimer sur une feuille la suite de carrés noirs ou blancs correspondants. L'image de départ sera dégradée, puisque les carrés d'origine n'étaient pas entièrement noirs ou blancs, ce qui donne toujours aux fax un aspect peu esthétique, mais l'essentiel de l'information est conservé.
Quand, comme ici, l'information initiale n'est pas entièrement restituée, mais arrive sous une forme s'en rapprochant, on parle de codage *avec perte*. On utilise souvent des codes qui permettent un certain niveau de perte, quand on estime que l'information importante n'est pas altérée. C'est souvent le cas pour les informations audiovisuelles (voir la section 2.5).

1.1.6 Les défauts de ce code

Nous avons entièrement décrit un procédé de codage et son pendant, le décodage, en respectant les principales contraintes que nous rencontrerons en théorie des codes (l'efficacité de la compression, la détection d'erreur, le secret). Mais les principes utilisés dans ce code-là ne sont guère utilisables en pratique dans les communications numériques courantes, pour plusieurs raisons :

La compression : le principe RLE tel qu'il est appliqué ici a plusieurs défauts. D'abord, si le message est constitué de pixels blancs et noirs alternés, la taille du message « compressé » sera plus grande que la taille du message original. Le principe n'a donc aucune garantie de compression. De plus, inclure les bits b_i est quasiment inutile puisqu'ils sont alternés et que la valeur du premier suffirait à déterminer tous les autres. Le code du fax les élimine. Il devient alors une suite de nombres $n_1 \dots n_k$. Mais cette suite, codée en bits (chaque nombre a sa traduction binaire) est plus difficile à décoder. En effet, en recevant une suite de bits "1001011010010", comme savoir s'il s'agit d'un nombre n_i, ou de plusieurs nombres concaténés ? On peut par exemple coder tous les nombres sur le même nombre de bits pour résoudre ce problème, mais le principe de compression n'est alors pas optimal. Il oblige à coder "2" par "00000000010", ce qui augmente la taille du message. Nous verrons dans la suite de ce chapitre comme s'assurer de la déchiffrabilité, et dans le chapitre suivant comment en déduire de bons principes de compression.

La détection d'erreurs : ce principe oblige à demander un renvoi de l'information à chaque erreur détectée, alors qu'on pourra concevoir des codes qui corrigent automatiquement les erreurs du canal (chapitre 4). De plus, aucune garantie théorique ne vient assurer que ce codage est bien adapté au taux d'erreur des canaux téléphoniques. On ne sait pas quel est le taux d'erreur détecté par ce code, ni s'il pourrait accomplir la même performance en ajoutant moins d'information dans les messages. L'efficacité de la transmission peut dépendre de la qualité des principes de détection et de correction d'erreurs.

Le secret : le code de César est cassable par un débutant en cryptographie. Pour tous les principes de *substitution mono-alphabétique*, il est facile de décoder, même sans connaître la clef. Une cryptanalyse simple consiste à étudier la fréquence d'apparition des lettres au sein du texte chiffré et d'en déduire la correspondance avec les lettres du texte clair. Le tableau 1.1 présente par exemple la répartition statistique des lettres dans ce manuscrit écrit en LaTeX (les codes de mise en forme LaTeX sont pris en compte en sus des mots du texte). Comme ce manuscrit est assez long, on peut supposer que ces fréquences sont assez représentatives d'un texte scientifique français écrit en LaTeX. Il est bien sûr possible d'avoir des tables de fréquences représentatives de textes littéraires français, scientifiques anglais, etc.

Exercice 1.3. *Scipion trouve un papier sur lequel se trouve le message chiffré suivant :* `HJXFW FZWFNY JYJ HTSYJSY IJ ATZX!`
Aide Scipion à déchiffrer ce message. *Solution page 279.*

E	14.90 %	P	3.20 %
I	7.82 %	F	1.82 %
T	7.52 %	B	1.60 %
N	7.49 %	G	1.59 %
R	6.91 %	X	1.47 %
S	6.76 %	H	1.20 %
A	6.28 %	Q	0.97 %
O	5.88 %	V	0.89 %
L	4.86 %	Y	0.51 %
D	4.75 %	K	0.37 %
U	4.72 %	W	0.16 %
C	4.71 %	J	0.15 %
M	3.33 %	Z	0.14 %

TAB. 1.1: Répartition des lettres dans ce manuscrit LaTeX.

1.1.7 Ordres de grandeur et complexité des algorithmes

Nous avons décrit un code et exposé ses qualités et ses défauts. Il est une qualité cruciale des codes que nous n'avons pas encore évoquée, celle de la rapidité à coder et décoder. Elle dépend de la vitesse des ordinateurs, et surtout de la complexité des algorithmes.

Taille des nombres

Nous considérons des nombres et leurs tailles, soit en chiffres décimaux, soit en bits. Ainsi, un nombre m possédera $\lceil \log_{10}(m) \rceil$ chiffres et $\lceil \log_2(m) \rceil$ bits. Pour fixer les idées, 128 bits font 39 chiffres décimaux ; 512 bits font 155 chiffres et 1024 bits sont représentables par 309 chiffres.

La vitesse des ordinateurs

Aujourd'hui n'importe quel PC est cadencé à au moins 1 GHz, c'est-à-dire qu'il effectue 1 milliard (10^9) d'opérations par seconde. À titre de comparaison, la vitesse la plus fantastique de l'univers est celle de la lumière : 300000 km/s = 3.10^8 m/s. Il ne faut que dix milliardièmes de seconde à la lumière pour traverser une pièce de 3 mètres ! Eh bien, pendant ce temps là, votre ordinateur a donc effectué 10 opérations ! ! ! On peut donc dire que *les ordinateurs actuels calculent à la vitesse de la lumière.*

Taille et âge de l'univers

Cette vitesse de calcul est véritablement astronomique ; pourtant la taille des nombres que l'on manipule reste énorme. En effet, rien que pour compter jusqu'à un nombre de 39 chiffres il faut énumérer 10^{39} nombres. Pour se convaincre à quel point c'est énorme, calculons l'âge de l'univers en secondes :

$$\text{âge Univers} \simeq 15 \text{ milliards années} \times 365.25 \times 24 \times 60 \times 60 \approx 5.10^{17} \text{secondes.}$$

Ainsi, un ordinateur cadencé à 1 GHz mettrait plus de deux millions de fois l'âge de l'univers pour simplement compter jusqu'à un nombre de « seulement » 39 chiffres ! Quant à un nombre de 155 chiffres (ce qu'on utilise couramment en cryptographie), c'est tout simplement le carré du nombre d'électrons contenus dans l'univers. En effet, notre univers contiendrait environ 3.10^{12} galaxies chacune renfermant grosso modo 200 milliards d'étoiles. Comme notre soleil pèse à la louche 2.10^{30} kg et qu'un électron ne pèse que $0,83.10^{-30}$ kg, on obtient :

$$\text{Univers} = (2.10^{30}/0,83.10^{-30}) * 200.10^{9} * 3.10^{12} \approx 10^{84} \text{ électrons.}$$

Ce sont pourtant des nombres que nous aurons à manipuler. Leur taille sera l'un des défis algorithmiques que nous aurons à relever, en même temps qu'une assurance de secret de nos messages, si pour les décoder sans en posséder la clef, il faut des millions d'années de calcul en mobilisant le milliard d'ordinateurs disponibles sur la Terre dont les plus rapides. Nous allons voir dans la suite comment construire des algorithmes capables de travailler avec de tels nombres entiers.

Complexité des algorithmes

Malgré la puissance des ordinateurs, un algorithme mal conçu pourra se révéler inexploitable. La *complexité* d'un algorithme est une mesure du temps que mettra un ordinateur, sur lequel on implémentera l'algorithme, pour terminer son calcul. La vitesse des ordinateurs augmentant constamment, une bonne mesure pour estimer ce temps est de compter le nombre d'opérations arithmétiques que nécessite l'algorithme. En général, il s'agit donc de compter les quatre opérations classiques (addition, soustraction, multiplication, division, mais parfois également des opérations binaires comme les décalages de bits) et seulement celles-ci, par souci de simplification. Bien sûr, ce nombre d'opérations dépend de la taille des nombres qu'on fournit en entrée, c'est pourquoi l'efficacité est toujours une fonction de la taille de l'entrée de l'algorithme. Pour pouvoir calculer cette taille dans tous les cas, on suppose que l'entrée est une suite de bits, et la taille de l'entrée est la longueur de cette suite. Ce

qui suppose que pour calculer l'efficacité d'un algorithme, on doit concevoir un code qui transforme son entrée en suite de bits. C'est une opération souvent assez simple. Par exemple, pour coder un entier a, il faut de l'ordre de $\log_2(a)$ bits (on les écrit simplement en base 2). La taille d'un entier est donc son logarithme en base 2. La taille d'une suite de pixels noirs et blancs est la longueur de cette suite.
Comme il n'est souvent pas possible de compter exactement toutes les opérations réalisées lors de l'exécution d'un programme, on encadre la complexité d'un algorithme par des majorants et minorants asymptotiques en utilisant la notation dite de Landau. Si $k(n)$ est le nombre d'opérations arithmétiques effectuées par l'ordinateur qui exécute l'algorithme sur une entrée de taille n, et f une fonction quelconque, on dit que $k(n) = O(f(n))$ s'il existe une constante c telle que pour tout n assez grand, $k(n) \leqslant c \times f(n)$. On peut alors dire que la complexité de l'algorithme est au plus de l'ordre de $f(n)$.

Exercice 1.4. *Quelle est la complexité du code fax, présenté plus haut ?*
Solution page 279.

Comme la notation de Landau permet de donner la complexité à une constante près, il est par exemple possible d'écrire $O(\log(n))$ sans donner la base du logarithme, puisque la base ne modifiera la fonction que d'une constante (le logarithme de la base).
En pratique tous les algorithmes devront présenter une complexité quasi-linéaire $O(n)$, où n est la taille du message source, pour pouvoir prétendre à une efficacité « temps-réel », c'est-à-dire pour pouvoir être utilisé dans une transmission où un temps d'attente long n'est pas acceptable (téléphonie, audiovisuel...).
Une partie de la cryptographie repose sur l'hypothèse qu'il existe des problèmes pour lesquels on ne connaît aucun algorithme d'une telle complexité. Tous les algorithmes connus nécessitent des temps de calcul astronomiques qui rendent l'approche du problème impossible.
Dans ce livre, nous considérons qu'un problème est impossible à résoudre (on emploie simplement parfois l'euphémisme « difficile ») si on ne connaît pas d'algorithme qui le résolvent en un temps humainement raisonnable, soit par exemple si sa résolution nécessiterait plus de 10^{50} opération.

1.2 Codage par flot et probabilités

Afin d'obtenir des méthodes efficaces, c'est-à-dire linéaires en fonction de la taille du message, on peut considérer le message comme un *flot* de bits, c'est à dire une succession potentiellement infinie de caractères, qu'on codera un par un. Cette technique est notamment utilisée en cryptographie, et permet

d'obtenir des messages dits *inconditionnellement sûrs*, *i.e.* pour lesquels la connaissance du message chiffré n'apporte aucune information sur le message clair. On parle alors de *chiffrement parfait*. Ainsi, la seule attaque possible est la recherche exhaustive de clef secrète. On utilise également le modèle du codage par flot pour construire des principes de détection et correction d'erreur (voir les codes convolutifs dans le chapitre 4).
Le code dit de Vernam, ou code à *clef jetable* - en Anglais *one-time-pad* - est un exemple de code par flot à visée cryptographique dont on peut prouver la sécurité inconditionnelle, moyennant l'introduction de quelques bases de probabilités et de théorie de l'information.

1.2.1 Le code de Vernam

En 1917, pendant la première guerre mondiale, la compagnie américaine AT&T avait chargé le scientifique Gilbert Vernam d'inventer une méthode de chiffrement que les Allemands ne pourraient pas casser. Le chiffrement jetable que celui-ci a conçu est le seul code connu à l'heure actuelle comme mathématiquement prouvé sûr.
Le système à clef jetable est un système cryptographique dit à *clef secrète*, c'est-à-dire que le secret réside dans un paramètre des fonctions de codage et de décodage connu uniquement de l'émetteur et du destinataire. C'est aussi le cas du chiffrement de César, dans lequel le paramètre est la taille du décalage des lettres ou des nombres.
Dans un système à clef jetable, cette clef n'est utilisée qu'une fois. Le secret repose sur le fait qu'une clef est associée à un seul message et est de même longueur que celui-ci. Le décodage s'effectue grâce à la propriété que pour tous messages M et clefs K de même longueur, on a :

$$(M \oplus K) \oplus K = M$$

où $\oplus$ (qu'on note aussi xor) désigne l'opération logique « ou exclusif » bit à bit. Il s'agit donc d'une addition modulo 2 de chaque bit. C'est cette utilisation du ou exclusif qui a été brevetée par Vernam 1919. Pour envoyer un message M de n bits, il faut avoir une clef K secrète de n bits. Le message chiffré $\tilde{M}$ est donné par $\tilde{M} = M \oplus K$. Pour déchiffrer, il suffit de calculer $\tilde{M} \oplus K$.

Exercice 1.5. *Pourquoi faut-il jeter la clef après l'avoir utilisée,* i.e. *changer de clef pour chaque nouveau message ?* *Solution page 279.*

Sous réserve que la clef K ait bien été générée de façon totalement aléatoire, indépendamment de M et qu'elle n'ait jamais été utilisée auparavant, un observateur tiers n'obtient aucune information sur le message clair s'il intercepte le message chiffré (hormis la taille de M). C'est Joseph Mauborgne, capitaine

au service secret de l'armée américaine qui le premier a proposé dans les années 1920 que la clef soit générée aléatoirement et c'est ensuite, Claude Shannon qui a prouvé la sécurité inconditionnelle de ce code, comme nous le verrons section 1.2.4.

Exercice 1.6. *Construire un protocole, à base de clefs jetables, permettant à un utilisateur de se connecter depuis un ordinateur quelconque sur internet à un serveur sécurisé. Le mot de passe doit être crypté pour ne pas circuler de façon visible sur internet ni être capturé par la machine utilisée.*

Solution page 280.

Évidemment, il reste à formaliser ce que veut dire générer un nombre aléatoire, donner des bons moyens de le faire et, pour prouver que le système est sûr, préciser ce que veut dire « obtenir de l'information » sur le message clair. Pour cela, nous introduisons maintenant les principes fondamentaux de la théorie de l'information, qui servent également de base aux principes de compression des messages.

1.2.2 Un peu de probabilités

Dans un système cryptographique, si on utilise une clef générée au hasard, toute déviance par rapport au « vrai » hasard sera un angle d'attaque pour la cryptanalyse. Le hasard a aussi sa part dans les méthodes de compression, car dès qu'il y a un ordre visible, une redondance, une organisation dans un message, non seulement les casseurs de codes vont s'en servir, mais les créateurs de codes vont y voir un moyen d'exprimer le message de façon plus dense. Mais qu'appelle-t-on une déviance par rapport au hasard, et plus simplement qu'appelle-t-on le hasard ? Par exemple, si les numéros "1 2 3 4 5 6" tombent au Loto, on doutera beaucoup qu'ils aient été vraiment générés au hasard, bien que *stricto sensu* cette combinaison ait autant de chance d'être générée que n'importe quelle autre. Si nous n'irons pas bien loin dans la philosophie, cette section apportera quelques moyens mathématiques d'aborder le hasard et ses effets, puis de créer quelque chose qui se rapprocherait du hasard.

Évènements et mesure de probabilité

Un *évènement* est le résultat possible d'une expérience aléatoire. Par exemple, si l'expérience est un jet de dé à six faces, l'obtention du nombre 6 est un évènement. Les opérateurs sur les ensembles ($\cup$, $\cap$, $\setminus$) sont utilisés pour les évènements (ils signifient *ou*, *et*, *sauf*).
On note Ω l'ensemble de tous les évènements possibles pour une expérience donnée.

Une mesure de probabilité P est une application définie sur Ω, à valeur dans $[0,1]$, qui vérifie :

1. $P(\Omega) = 1$ et $P(\emptyset) = 0$;
2. quels que soient A, B des évènements disjoints ($A \cap B = \emptyset$), $P(A \cup B) = P(A) + P(B)$.

Si l'ensemble des évènements est un ensemble discret ou fini, on parle de *probabilité discrète*.
Par exemple, si l'expérience aléatoire est le jet d'un dé à six faces non pipé, l'ensemble des évènements est $\{1,2,3,4,5,6\}$, et la probabilité d'occurrence de chacun $1/6$.
L'ensemble des valeurs prises par la fonction de probabilité est la *distribution des probabilités*, ou *loi de probabilité*. Une distribution est dite *uniforme* si tous les évènements ont une même probabilité d'occurrence.
Le lemme dit de Gibbs est un résultat sur les distributions discrètes qui nous sera utile plusieurs fois dans cet ouvrage.

Lemme 1 (de Gibbs). *Soient $(p_1, \ldots, p_n)$, $(q_1, \ldots, q_n)$ deux lois de probabilité discrètes.*

$$\sum_{i=1}^{n} p_i * \log \frac{q_i}{p_i} \leqslant 0 \ .$$

Preuve. On sait que pour tout $x \in \mathbb{R}_*^+$, $\ln(x) \leqslant x - 1$. Donc

$$\sum_{i=1}^{n} p_i * \ln \frac{q_i}{p_i} \leqslant \sum_{i=1}^{n} p_i * (\frac{q_i}{p_i} - 1) \ .$$

Soit puisque $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i = 1$, alors $\sum_{i=1}^{n} p_i * (\frac{q_i}{p_i} - 1) = \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i = 0$. Par suite, $\sum_{i=1}^{n} p_i * \log \frac{q_i}{p_i} \leqslant 0$. □

Probabilités conditionnelles et Formule de Bayes

On dit que deux évènements sont *indépendants* si $P(A \cap B) = P(A)P(B)$. On appelle *probabilité conditionnelle* de l'évènement A par rapport à l'évènement B, la probabilité que A se produise, sachant que B s'est déjà produit. Elle est notée $P(A|B)$ et définie par :

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Par récurrence, on montre facilement que pour un ensemble d'évènements $A_1, \ldots, A_n$,

$$P(A_1 \cap \ldots \cap A_n) = P(A_1|A_2 \cap \ldots \cap A_n)P(A_2|A_3 \cap \ldots \cap A_n) \ldots P(A_{n-1}|A_n)P(A_n)$$

La formule dite de Bayes permet de calculer pour un ensemble d'évènements $A_1, \ldots, A_n, B$ les probabilités $P(A_k|B)$ en fonction des $P(B|A_k)$.

$$P(A_k|B) = \frac{P(A_k \cap B)}{P(B)} = \frac{P(B|A_k)P(A_k)}{\sum_i P(B|A_i)P(A_i)}$$

Exercice 1.7. *On propose le code secret suivant, permettant de coder deux caractères a et b avec trois clefs différentes k_1, k_2 et k_3 : si la clef est k_1 alors $a \to 1$ et $b \to 2$, si la clef est k_2 alors $a \to 2$ et $b \to 3$, sinon $a \to 3$ et $b \to 4$.*
On suppose en outre que l'on a des connaissances a priori *sur le message M envoyé et la clef K utilisée : $P(M = a) = 1/4$; $P(M = b) = 3/4$ et $P(K = k_1) = 1/2$; $P(K = k_2) = P(K = k_3) = 1/4$.*
Quelles sont les probabilités d'obtenir les chiffres 1, 2 ou 3 ? Quelles sont les probabilités conditionnelles que le message soit a ou b sachant la valeur du chiffre ? Peut-on dire intuitivement si ce code secret est un « chiffrement parfait » ? *Solution page 280.*

Ces brefs rappels étant faits sur la théorie de base permettant d'étudier des évènements aléatoires, voyons ce que l'aléatoire signifie pour des ordinateurs.

1.2.3 Entropie

Source d'information

On appelle S l'alphabet du message source. Un message est alors un élément de S^+. Pour tout message, on peut calculer les fréquences d'apparition de chaque élément de l'alphabet, et construire ainsi une distribution de probabilités sur S. Une *source d'information* est constituée du couple $\mathcal{S} = (S, \mathcal{P})$ où $S = (s_1, \ldots, s_n)$ est l'alphabet source et $\mathcal{P} = (p_1, \ldots, p_n)$ est une distribution de probabilités sur S, c'est-à-dire que p_i est la probabilité d'occurrence de s_i dans une émission. On peut construire une source d'information à partir de n'importe quel message, en construisant la distribution de probabilités à partir de la fréquence des caractères dans le message.
La source d'information $\mathcal{S} = (S, \mathcal{P})$ est dite *sans mémoire* lorsque les évènements (occurrences d'un symbole dans une émission) sont indépendants et que leur probabilité reste stable au cours de l'émission (la source est *stationnaire*). La source est dite *markovienne* si les probabilités d'occurrence des caractères dépendent des caractères émis précédemment. Dans le cas d'un seul prédécesseur, $\mathcal{P} = \{p_{ij}\}$, où p_{ij} est la probabilité d'occurrence de s_i sachant que s_j vient d'être émis. On a alors $p_i = \sum_j p_{ij}$.
Par exemple, un texte en français est une source, dont l'alphabet est l'ensemble des lettres latines et les probabilités d'occurrence sont les fréquences

d'apparition de chaque caractère. Comme les probabilités dépendent dans ce cas fortement des caractères qui viennent d'être émis (un U est beaucoup plus probable après un Q qu'après un autre U), le modèle markovien sera plus adapté.
Une image induit aussi une source, dont les caractères de l'alphabet sont des niveaux de couleurs. Un son est une source dont l'alphabet est un ensemble de fréquences et d'intensités.
Une source $\mathcal{S}$ est dite *sans redondance* si sa distribution de probabilités est uniforme. Ce qui n'est évidemment pas le cas pour les messages courants, où certaines lettres ou certains mots sont beaucoup plus employés que d'autres. Ce sera l'angle d'attaque des méthodes de compression mais aussi des pirates qui chercheraient à lire un message sans y être autorisés.

Entropie d'une source

L'entropie est une notion fondamentale pour la manipulation d'un code. C'est en effet une mesure, à la fois de la quantité d'information qu'on peut attribuer à une source (ce qui sera utile pour la compression des messages), et du degré d'ordre et de redondance d'un message, ce qui est une information cruciale pour la cryptographie.
L'*entropie* d'une source $\mathcal{S} = (S, \mathcal{P})$, $S = (s_1, \ldots, s_n)$, $\mathcal{P} = (p_1, \ldots, p_n)$ est :

$$H(\mathcal{S}) = H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2(p_i) \ = \sum_{i=1}^{n} p_i \log_2(\frac{1}{p_i}) \ .$$

On désigne par extension l'entropie d'un message comme l'entropie de la source induite par ce message, la distribution de probabilités étant calculée à partir des fréquences d'apparition des caractères dans le message.

Propriété 1. *Soit $\mathcal{S} = (S, \mathcal{P})$ une source.*

$$0 \leqslant H(\mathcal{S}) \leqslant \log_2 n \ .$$

Preuve. Appliquons le lemme de Gibbs à la distribution $(q_1, \ldots, q_n) = (\frac{1}{n}, \ldots, \frac{1}{n})$, on obtient $H(\mathcal{S}) \leqslant \log_2 n$ quelle que soit la source S. Enfin, la positivité est évidente puisque les probabilités p_i sont inférieures à 1. □

Remarquons que pour une distribution uniforme, l'entropie atteint donc son maximum. Elle baisse quand on s'éloigne de la distribution. C'est pour cette raison qu'on l'appelle « mesure du désordre », en supposant que le plus grand désordre est atteint par la distribution uniforme.

Exercice 1.8. *Quelle est l'entropie d'une source qui émet un caractère* 1 *avec une probabilité* 0.1 *et le caractère* 0 *avec une probabilité* 0.9 *? Pourquoi une faible entropie est-elle un bon augure pour la compression ? Solution page 280.*

Entropies conjointe et conditionnelle

On étend facilement la définition de l'entropie à plusieurs sources. Soient $\mathcal{S}_1 = (\mathcal{S}_1, P_1)$ et $\mathcal{S}_2 = (\mathcal{S}_2, P_2)$ deux sources sans mémoire, dont les évènements ne sont pas forcément indépendants. On note $\mathcal{S}_1 = (s_{11}, \ldots, s_{1n})$, $P_1 = (p_i)_{i=1..n}$, $\mathcal{S}_2 = (s_{21}, \ldots, s_{2m})$ et $P_2 = (p_j)_{j=1..m}$; puis $p_{i,j} = P(\mathcal{S}_1 = s_{1i} \cap \mathcal{S}_2 = s_{2j})$ la probabilité d'occurrence conjointe de s_{1i} et s_{2j} et $p_{i|j} = P(\mathcal{S}_1 = s_{1i} | \mathcal{S}_2 = s_{2j})$ la probabilité d'occurrence conditionnelle de s_{1i} et s_{2j}.
On appelle l'*entropie conjointe* de $\mathcal{S}_1$ et $\mathcal{S}_2$ la quantité

$$H(\mathcal{S}_1, \mathcal{S}_2) = -\sum_{i=1}^{n} \sum_{j=1}^{m} p_{i,j} \log_2(p_{i,j})$$

Par exemple, si les sources $\mathcal{S}_1$ et $\mathcal{S}_2$ sont indépendantes, alors $p_{i,j} = p_i p_j$ pour tous i, j et donc dans cas on montre facilement que $H(\mathcal{S}_1, \mathcal{S}_2) = H(\mathcal{S}_1) + H(\mathcal{S}_2)$. Au contraire, il peut arriver, si les évènements de $\mathcal{S}_1$ et $\mathcal{S}_2$ ne sont pas indépendants, qu'on veuille connaître la quantité d'information contenue dans une source, connaissant un évènement de l'autre. On calcule alors l'*entropie conditionnelle* de $\mathcal{S}_1$ relativement à la valeur de $\mathcal{S}_2$, donnée par

$$H(\mathcal{S}_1 | \mathcal{S}_2 = s_{2j}) = -\sum_{i=1}^{n} p_{i|j} \log_2(p_{i|j})$$

Enfin, on étend cette notion à une entropie conditionnelle de $\mathcal{S}_1$ connaissant $\mathcal{S}_2$, qui est la quantité d'information restant dans $\mathcal{S}_1$ si la loi de $\mathcal{S}_2$ est connue :

$$H(\mathcal{S}_1 | \mathcal{S}_2) = \sum_{j=1}^{m} p_j H(\mathcal{S}_1 | \mathcal{S}_2 = s_{2j}) = \sum_{i,j} p_{i,j} \log_2 \left(\frac{p_j}{p_{i,j}} \right)$$

Cette notion est cruciale en cryptographie. En effet, il est très important que tous les messages cryptés aient une entropie forte, pour ne pas que les traces d'organisation dans un message donnent des informations sur la manière dont il a été crypté. Mais il est aussi important que l'entropie reste forte si on arrive à connaître des informations dépendantes, par exemple même la connaissance d'un message et de son cryptage ne devrait pas donner d'information sur la clef utilisée.
On a alors les relations simples mais importantes suivantes :

$$H(\mathcal{S}_1) \geqslant H(\mathcal{S}_1 | \mathcal{S}_2)$$

avec égalité si et seulement si $\mathcal{S}_1$ et $\mathcal{S}_2$ sont indépendantes ; et encore :

$$H(\mathcal{S}_1, \mathcal{S}_2) = H(\mathcal{S}_2) + H(\mathcal{S}_1 | \mathcal{S}_2)$$

Mais l'entropie d'une source sans mémoire à elle seule ne capture pourtant pas tout l'ordre ou le désordre contenu dans un message. Par exemple, les messages "1 2 3 4 5 6 1 2 3 4 5 6" et "3 1 4 6 4 6 2 1 3 5 2 5" ont la même entropie, et pourtant le premier est suffisamment ordonné pour qu'on puisse le décrire par une formule, du type "1…6 1…6", ce qui n'est sans doute pas le cas du second. Pour prendre en compte ce type d'organisation, on fait appel aux extensions de sources.

Extension d'une source

Soit une source $\mathcal{S}$ sans mémoire. La $k^{ième}$ *extension* $\mathcal{S}^k$ de $\mathcal{S}$ est le doublet $(S^k, \mathcal{P}^k)$, où S^k est l'ensemble des mots de longueur k sur S, et $\mathcal{P}^k$ est la distribution de probabilité ainsi définie : pour un mot $s = s_{i_1} \dots s_{i_k} \in S^k$, alors $P^k(s) = P(s_{i_1} \dots s_{i_k}) = p_{i_1} \dots p_{i_k}$.

Exemple : $S = (s_1, s_2)$, $\mathcal{P} = (\frac{1}{4}, \frac{3}{4})$ alors $S^2 = (s_1s_1, s_1s_2, s_2s_1, s_2s_2)$ et $P^2 = (\frac{1}{16}, \frac{3}{16}, \frac{3}{16}, \frac{9}{16})$.
Si $\mathcal{S}$ est une source markovienne, on définit de la même façon S^k, et pour un mot $s = s_{i_1} \dots s_{i_k} \in S^k$, alors $P^k(s) = p_{i_1} p_{i_2 i_1} \dots p_{i_k i_{k-1}}$.

Propriété 2. *Soient $\mathcal{S}$ une source, et $\mathcal{S}^k$ sa $k^{ième}$ extension.*

$$H(\mathcal{S}^k) = kH(\mathcal{S})\,.$$

Autrement dit, cette propriété exprime que la quantité d'information d'une source étendue à k caractères est exactement k fois celle de la source originelle. Cela semble tout à fait naturel.
Cependant, il faut distinguer une source d'un message (un fichier par exemple). Tout d'abord, à partir d'un fichier il est possible de dénombrer les occurrences des caractères et de donner une entropie correspondant à la source qui aurait les caractéristiques probabilistes identiques. Cela permet d'utiliser la théorie de l'information pour compresser des fichiers, nous verrons cela plus en détails dans la section 2. Ensuite, dans le cas d'un message, il est possible de regrouper les caractères du message k par k. Attention, là nous ne sommes plus du tout dans le cadre de la propriété 2 ! Au contraire, l'entropie de la source correspondant à cette « extension de message » est alors forcément inférieure à l'entropie correspondant au message initial.

Exemple : Les messages "1 2 3 4 5 6 1 2 3 4 5 6" et "3 1 4 6 4 6 2 1 3 5 2 5" correspondent à la même entropie en prenant la première extension de la source : 6 caractères de probabilité un sixième chacun, donnent une entropie de $\sum_{i=1}^{6} \frac{1}{6} \log_2(6) = \log_2(6) \approx 2.585$. Avec la deuxième extension de la source

$(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6})$, on obtient 36 groupes possibles chacun de probabilité $\frac{1}{36}$ et d'entropie conforme à la propriété 2 : $\log_2(36) = \log_2(6^2) = 2\log_2(6)$.
Toutefois, en regroupant les messages par exemple par blocs de deux caractères, on obtient :
– (12)(34)(56)(12)(34)(56) : trois couples distincts de probabilité un tiers chacun correspondent à une entropie de $\log_2(3) \approx 1.585$.
– De même (31)(46)(46)(21)(35)(25) donnent 5 couples distincts d'entropie $\frac{1}{6}\log_2(6) + \frac{2}{6}\log_2(\frac{6}{2}) + 3\frac{1}{6}\log_2(6) \approx 2.252$.
Dans les deux cas l'entropie obtenue est bien inférieure à deux fois celle du message initial. Nous précisons ceci dans la propriété 3.

Propriété 3. *Soient $\mathcal{M}$ un message de taille n et $\mathcal{S}_{\mathcal{M}^k}$ la source dont les probabilités correspondent aux occurrences des k-uplets consécutifs de M. Alors*

$$H(\mathcal{S}_{\mathcal{M}^k}) \leqslant log_2\left(\left\lceil \frac{n}{k} \right\rceil\right).$$

Preuve. Il y a $\lceil \frac{n}{k} \rceil$ k-uplets dans le message M. En outre l'entropie est maximale pour le plus grand nombre de k-uplets distincts possibles d'occurrences égales. Dans ce cas la source équivalente serait d'au plus $\lceil \frac{n}{k} \rceil$ caractères tous distincts et de probabilité d'occurrence $\frac{1}{\lceil \frac{n}{k} \rceil}$. Ce qui donne une entropie de $log_2\left(\lceil \frac{n}{k} \rceil\right)$. □

Ceci nous amène au problème du hasard et de sa génération. Une suite de nombres générée au hasard devra répondre à des critères forts, et en particulier avoir une haute entropie. On ne serait pas satisfait de la suite “1 2 3 4 5 6 1 2 3 4 5 6”, qui présente une forme d'organisation aisément détectable. On le serait plus si on rencontrait “3 1 4 6 4 6 2 1 3 5 2 5”, qui a une entropie supérieure, mais parfois les modes d'organisation et les biais sont plus subtils, et pourtant ils sont de bons angles d'attaque pour les casseurs de codes.

1.2.4 Chiffrement parfait

Nous disposons maintenant de l'attirail théorique pour préciser ce que nous entendons par chiffrement inconditionnellement sûr, ou chiffrement parfait. Un chiffrement est *parfait* si le message chiffré C ne fournit aucune information sur le message initial M ; en terme d'entropie, puisque c'est la mesure de la quantité d'information que nous avons adoptée, on aura :

$$H(M|C) = H(M)$$

Exercice 1.9 (Preuve de la perfection du code de Vernam).

1. *Démontrer que pour un code secret où la clef K est générée aléatoirement pour chaque message M on a $H(M|K) = H(M)$.*

2. *En utilisant les entropies conditionnelles et la définition du code de Vernam, démontrer que dans un code de Vernam ($C = M \oplus K$), on a toujours $H(M,K,C) = H(M,C) = H(M,K) = H(C,K)$; en déduire des relations entre les entropies conditionnelles de M, C et K.*
3. *Démontrer que le chiffrement à clef jetable de Vernam est un chiffrement parfait.*

Solution page 280.

1.2.5 Mise en pratique du chiffrement parfait ?

On dispose d'une méthode de cryptage dont on possède la preuve de la sécurité. C'est d'ailleurs la seule méthode connue pour laquelle on ait prouvé son caractère parfait.
Mais pour se servir de cette méthode en pratique, il faut savoir générer des clefs aléatoires, et ce problème est loin d'être trivial. D'autre part, comme la clef n'est pas réutilisable, les protocoles d'échange des clefs entre émetteurs et destinataires restent problématiques.
Nous allons voir dans la section suivante comment construire des codes qui réutilisent une clef unique et rendent le protocole d'échange moins fastidieux. Sans être parfaits, ces codes tendent vers cette propriété. La première idée sera de découper le message en blocs et de chiffrer chaque bloc séparément. La deuxième idée est de générer des nombres aléatoires de bonne qualité, ou plus précisément des nombres pseudo-aléatoires. Cela permet de réaliser un chiffrement bit à bit (par flot) comme illustré sur la figure 1.1.

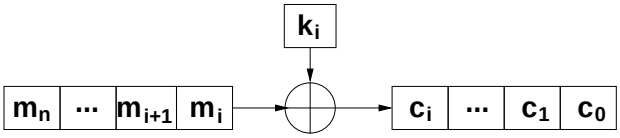


FIG. 1.1: Chiffrement bit à bit (par flot).

1.3 Codage par blocs, algèbre et arithmétique

Le chiffrement de Vernam est aujourd'hui le seul algorithme cryptographique à clef secrète prouvé inconditionnellement sûr. Ainsi tous les autres systèmes sont théoriquement cassables.
Pour ces systèmes, on utilise des chiffrements *pratiquement sûrs* : la connaissance du message chiffré (ou de certains couples message clair/message chiffré)

ne permet de retrouver ni la clef secrète ni le message clair *en un temps humainement raisonnable* (voir les ordres de grandeur et les limites des machines à la section 1.1.7).
On peut décider par exemple, pour éviter des protocoles d'échange de clef trop fréquents, de choisir une clef unique, et de la réutiliser. Ceci impose le découpage des messages sources en blocs d'une certaine taille, fonction de la taille de la clef. Le chiffrement par blocs est un standard qui sera également largement utilisé pour la détection et la correction d'erreurs.
C'est aussi le principe de l'un des codes les plus célèbres, le code ASCII (pour « *American Standard Code for Information Interchange* »), pour la représentation numérique des lettres et signes de l'alphabet latin. Dans le code ASCII, l'alphabet source est l'alphabet latin, et l'alphabet du code est $V = \{0, 1\}$. Les *mots de code* sont tous les mots de longueur 8 sur V :

$$C = \{00000000,\ 00000001,\ \ldots,\ 11111111\}\ .$$

Chacun des $2^8 = 256$ caractères (majuscules, minuscules, caractères spéciaux, caractères de contrôle) est représenté par un mot de longueur 8 sur V suivant une fonction de codage dont un extrait est donné par la table 1.2 ci-dessous.

A	01000001	J	01001010	S	01010011
B	01000010	K	01001011	T	01010100
C	01000011	L	01001100	U	01010101
D	01000100	M	01001101	V	01010110
E	01000101	N	01001110	W	01010111
F	01000110	O	01001111	X	01011000
G	01000111	P	01010000	Y	01011001
H	01001000	Q	01010001	Z	01011010
I	01001001	R	01010010	espace	00100000

Tab. 1.2: Un extrait du code ASCII.

Par exemple, le codage ASCII du message : *UNE CLEF*, est la chaîne :
01010101010011100100010100100000010000110100110001000101010001 10.

1.3.1 Blocs et modes de chaînage

On peut coder chaque bloc d'un message par un même algorithme de façon indépendante pour chaque bloc. C'est ce qui est appelé le mode de codage ECB, pour « *Electronic Code Book* ». Plus généralement, cette indépendance de codage entre les blocs n'est pas requise et les différentes façons de combiner les blocs sont appelés *modes de chiffrement*.

Le mode ECB : *Electronic Code Book*

Dans ce mode, le message M est découpé en blocs m_i de taille fixe.

Chaque bloc est alors crypté séparément par une fonction E_k, paramétrée par une clef k comme sur la figure 1.2.

Ainsi un bloc de message donné m_i sera toujours codé de la même manière. Ce mode de chiffrement est le plus simple mais ne présente donc aucune sécurité et n'est normalement jamais utilisé en cryptographie.

$$c_i = E_k(m_i) \tag{1.1}$$

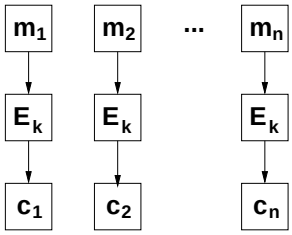


FIG. 1.2: Mode de chiffrement par blocs ECB.

Le mode CBC : *Cipher Bloc Chaining*

Le mode CBC a été introduit pour qu'un bloc ne soit pas codé de la même manière s'il est rencontré dans deux messages différents. Il faut ajouter une valeur initiale C_0 (ou IV pour « *initial value* »), aléatoire par exemple. Chaque bloc est d'abord modifié par XOR avec le bloc crypté précédent avant d'être lui même crypté conformément à la figure 1.3 par :

$$c_i = E_k(m_i \oplus c_{i-1}) \tag{1.2}$$

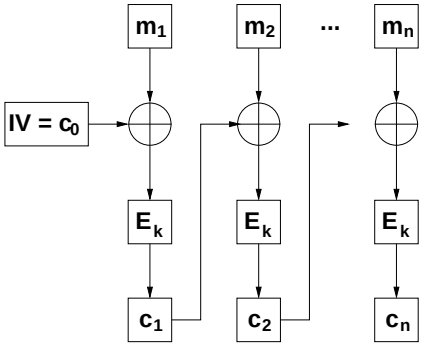


FIG. 1.3: Mode de chiffrement par blocs CBC.

C'est le mode de chiffrement le plus utilisé. Le déchiffrement nécessite l'inverse de la fonction de codage $D_k = E_k^{-1}$ pour déchiffrer : $m_i = c_{i-1} \oplus D_k(c_i)$.

Le mode CFB : *Cipher FeedBack*

Pour ne pas avoir besoin de la fonction inverse pour décrypter, il est possible de faire un XOR après le cryptage, c'est le principe du mode CFB, comme on peut le voir sur la figure 1.4.

$$c_i = m_i \oplus E_k(c_{i-1}) \tag{1.3}$$

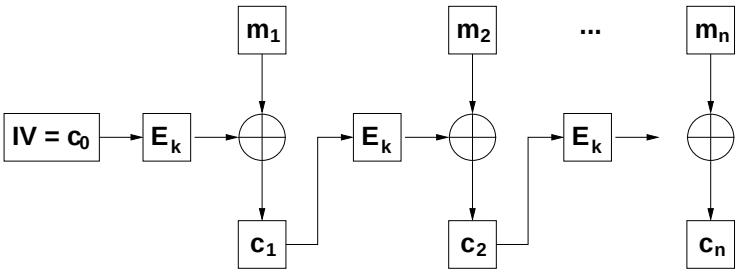


FIG. 1.4: Mode de chiffrement par blocs CFB.

L'intérêt de ce mode est que le déchiffrement ne nécessite pas l'implémentation de la fonction D_k : $m_i = c_i \oplus \ E_k(c_{i-1})$. Ce mode est donc moins sûr que le CBC et est utilisé par exemple pour les cryptages réseaux.

Le mode OFB : *Output FeedBack*

Une variante du mode précédent permet d'avoir un codage et un décodage totalement symétrique, c'est le mode OFB suivant la figure 1.5.

$$z_0 = c_0 \ ; \ z_i = E_k(z_{i-1}) \ ; \ c_i = m_i \oplus z_i \tag{1.4}$$

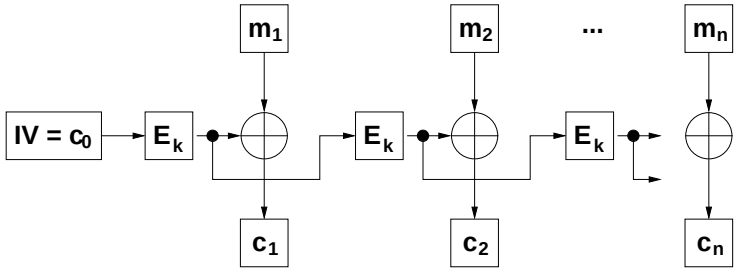


FIG. 1.5: Mode de chiffrement par blocs OFB.

Ce qui se déchiffre par : $z_i = E_k(z_{i-1})$; $m_i = c_i \oplus z_i$. Ce mode est utile dans les satellites pour lesquels minimiser le nombre de circuits embarqués est crucial.

Le mode CTR : *Counter-mode encryption*

Ce mode est également totalement symétrique, mais en outre facilement parallélisable. Il fait intervenir le chiffrement d'un compteur de valeur initiale T :

$$c_i = m_i \oplus E_k(T+i) \tag{1.5}$$

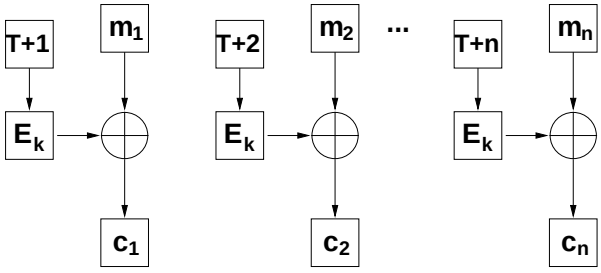


FIG. 1.6: Mode de chiffrement par blocs CTR.

Le déchiffrement est identique : $m_i = c_i \oplus E_k(T+i)$. L'intérêt d'un tel mode est que les différents calculs sont indépendants, comme pour le mode ECB, mais qu'un même bloc n'est *a priori* jamais codé de la même façon.

Exercice 1.10. *Un message M est découpé en n blocs $M = M_1, \ldots, M_n$ qui sont cryptés dans un schéma par blocs en $C = C_1, \ldots C_n$. Bob reçoit les blocs C_i, mais malheureusement, il ne sait pas qu'un et un seul des blocs a été transmis incorrectement (par exemple, des 1 ont été changés en 0 et vice versa durant la transmission du bloc C_1).*

Montrer que le nombre de blocs du message mal décryptés par Bob est égal à 1 si l'un des modes ECB, OFB ou CTR a été utilisé et égal à 2 si CBC ou CFB a été utilisé. *Solution page 281.*

1.3.2 Structures algébriques des mots de codes

L'élaboration des codes par blocs nécessite de pouvoir faire des opérations et des calculs sur les blocs. Par exemple, l'opération $\oplus$ sur un bloc de bits est l'addition modulo 2 de deux vecteurs de bits. D'autre part, les fonctions de codage doivent être inversibles, ce qui nécessitera des structures où l'on peut facilement calculer l'inverse d'un bloc. Pour pouvoir faire ces calculs sur des bases algébriques solides, rappelons les structures fondamentales.

Groupes

Un *groupe* $(G, *)$ est un ensemble muni d'un opérateur binaire interne vérifiant les propriétés suivantes :

1. $*$ est associative : pour tous $a, b, c \in G, a * (b * c) = (a * b) * c$.
2. Il existe un élément neutre $e \in G$, tel que pour tout $a \in G$, on trouve $a * e = e * a = a$.
3. Tout élément a un inverse : pour tout $a \in G$, il existe $a^{-1} \in G$ tel que $a * a^{-1} = a^{-1} * a = e$.

De plus, si la loi est commutative ($a * b = b * a$ pour tous $a, b \in G$), alors G est dit *abélien*. On dit qu'un sous-ensemble H de G est un *sous-groupe* de G lorsque les restrictions des opérations de G confèrent à H une structure de groupe. Pour un élément a d'un groupe G, on note a^n la répétition de la loi $*$, $a * \cdots * a$, portant sur n termes égaux à a pour tout $n \in \mathbb{N}^*$.
Si un élément $g \in G$ est tel que pour tout $a \in G$, il existe $i \in \mathbb{Z}$, tel que $a = g^i$, alors g est un *générateur* du groupe $(G, *)$ ou encore une *racine primitive*. Un groupe est dit *cyclique* s'il possède un générateur. Par exemple, pour un entier n fixé, l'ensemble des entiers $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$, muni de la loi d'addition modulo n est un groupe cyclique généré par 1 ; si $n = 7$, et si on choisit pour loi de composition la multiplication modulo 7, l'ensemble $\{1, \ldots, 6\}$ est un groupe cyclique généré par 3, car $1 = 3^0$, $2 = 3^2 = 9$, $3 = 3^1$, $4 = 3^4$, $5 = 3^5$, $6 = 3^3$.
Soient un groupe $(G, *)$ et $a \in G$. L'ensemble $\{a^i, i \in \mathbb{N}\}$ est un sous-groupe de G, noté $< a >$ ou G_a. Si ce sous-groupe est fini, son cardinal est l'*ordre* de a. Si G est fini, le cardinal de tout sous-groupe de G divise le cardinal de G.

Propriété 4. *(Lagrange) Dans un groupe fini abélien $(G, *, e)$ de cardinal n, pour tout $x \in G$: $x^n = e$.*

Preuve. Soit a un élément du groupe G, alors a est inversible. Donc, l'application $f_a : x \mapsto a \times x$ définie de G dans G est une bijection. Donc $Im(f_a) = G$; d'où $\prod_{y \in Im(f_a)} y = \prod_{x \in G} x$. Or $\prod_{y \in Im(f_a)} y = \prod_{x \in G} a \times x = a^n \prod_{x \in G} x$ (commutativité de $\times$). Ainsi $a^n \prod_{x \in G} x = \prod_{x \in G} x$, d'où $a^n = e$. □

Anneaux

Un *anneau* $(A, +, \times)$ est un ensemble muni de deux opérateurs binaires internes vérifiant les propriétés suivantes :

1. $(A, +)$ est un groupe abélien.
2. $\times$ est associative : pour tous $a, b, c \in A$, $a \times (b \times c) = (a \times b) \times c$.
3. $\times$ est distributive sur $+$: pour tous $a, b, c \in A$, $a \times (b + c) = (a \times b) + (a \times c)$ et $(b + c) \times a = (b \times a) + (c \times a)$.

Si de plus $\times$ possède un neutre dans A, A est dit *unitaire*. Si de plus $\times$ est commutative, A est dit *commutatif*. Tous les éléments de A ont un *opposé*, c'est leur inverse pour la loi $+$. Ils n'ont cependant pas forcément un inverse pour la loi $\times$. L'*ensemble des inversibles* pour la loi $\times$ est souvent noté A^*.
Pour un élément a d'un anneau A, on note $n \cdot a$ (ou plus simplement na) la somme $a + \cdots + a$ portant sur n termes égaux à a pour tout $n \in \mathbb{N}^*$.
Si l'ensemble $\{k \in \mathbb{N}^* \ : \ k \cdot 1 = 0\}$ n'est pas vide, le plus petit élément de cet ensemble est appelé la *caractéristique* de l'anneau. Dans le cas contraire, on dit que l'anneau est de caractéristique 0. Par exemple, $(\mathbb{Z}, +, \times)$ est un anneau unitaire commutatif de caractéristique 0.
Deux anneaux $(A, +_A, \times_A)$ et $(B, +_B, \times_B)$ sont *isomorphes* lorsqu'il existe une bijection $f \ : \ A \longrightarrow B$ vérifiant pour tous x et y dans A :

$$f(x +_A y) = f(x) +_B f(y) \quad \text{et} \quad f(x \times_A y) = f(x) \times_B f(y). \tag{1.6}$$

Si E est un ensemble quelconque et $(A, +, \times)$ un anneau tel qu'il existe une bijection f de E sur A, alors E peut être muni d'une structure d'anneau :

$$x +_E y = f^{-1}\left(f(x) + f(y)\right) \quad \text{et} \quad x \times_E y = f^{-1}\left(f(x) \times f(y)\right). \tag{1.7}$$

L'anneau $(E, +_E, \times_E)$ ainsi défini est évidemment isomorphe à A. Lorsque deux anneaux sont isomorphes, on peut identifier l'un à l'autre. C'est ce que nous ferons le plus souvent.
Un anneau commutatif est *intègre* si il ne possède pas de diviseur de zéro, autrement dit si pour deux éléments x et y vérifiant $xy = 0$ alors forcément l'un deux au moins est nul. Un *idéal* I est un sous-groupe d'un anneau A pour la loi $+$ qui est « absorbant »pour la loi $\times$: pour $g \in I$, le produit $a \times g$ reste dans I pour n'importe quel élément a de l'anneau A. Pour tout $x \in A$ la partie $Ax = \{ax; a \in A\}$ est un idéal de A appelé *idéal engendré* par x. Un idéal I de A est dit *principal* s'il existe un générateur x (tel que $I = Ax$). Un anneau est *principal* si et seulement si tout idéal y est principal.

Corps

Un *corps* $(A, +, \times)$ est un ensemble muni de deux opérateurs binaires internes vérifiant les propriétés suivantes :

1. $(A, +, \times)$ est un anneau unitaire.
2. $(A \setminus \{0\}, \times)$ est un groupe.

Si $(A, +, \times)$ est commutatif, c'est donc un *corps commutatif*; l'inverse (ou opposé) de x par la loi $+$ est noté $-x$; l'inverse de x par la loi $\times$ est noté x^{-1}. La *caractéristique* d'un corps est sa caractéristique en tant qu'anneau. Par exemple, $(\mathbb{Q}, +, \times)$ est un corps commutatif de caractéristique 0.

Puisque tous les anneaux et corps qui nous intéressent dans ce cours sont commutatifs, nous dirons désormais anneau (respectivement corps) pour désigner un anneau unitaire commutatif (respectivement un corps commutatif).
Deux corps sont *isomorphes* lorsqu'ils sont isomorphes en tant qu'anneaux.
On dit qu'un sous ensemble W d'un corps V est un *sous-corps* de V lorsque les restrictions des opérations de V à W confèrent à W une structure de corps.

Espaces vectoriels

Un ensemble $\mathbb{E}$ est un *espace vectoriel* sur un corps V s'il est muni d'une loi de composition interne $+$ et d'une loi externe ".", telles que :

1. $(\mathbb{E}, +)$ est groupe commutatif.
2. Pour tout $u \in \mathbb{E}$, alors $1_V.u = u$.
3. Pour tous $\lambda, \mu \in V$ et $u \in \mathbb{E}$, alors $\lambda.(\mu.u) = (\lambda \times \mu).u$.
4. Pour tous $\lambda, \mu \in V$ et $u \in \mathbb{E}$, alors $\lambda.u + \mu.u = (\lambda + \mu).u$.
5. Pour tous $\lambda \in V$ et $u, v \in \mathbb{E}$, alors $\lambda.(u + v) = \lambda.u + \lambda.v$.

Un élément d'un espace vectoriel est appelé un *vecteur*, et les éléments du corps V sont des *scalaires*. L'ensemble $\{0, 1\}$ muni des opérations d'addition et de multiplication est un corps commutatif noté $\mathbb{F}_2$. L'ensemble des tableaux de bits de taille n peut donc être muni d'une structure d'espace vectoriel. L'ensemble des mots de code est alors $\mathbb{F}_2^n$.
Selon la structure choisie, on peut manipuler les mots de code par des additions et des multiplications, entières ou vectorielles. Toutes ces structures sont très générales, et classiques en algèbre. Une particularité des codes est qu'ils constituent des ensembles finis. Les groupes et corps finis ont des propriétés additionnelles que nous utiliserons intensément au fils de cet ouvrage.
On note $\mathbb{Z}$ l'ensemble des entiers, $\mathbb{Q}$ le corps des rationnels, $\mathbb{R}$ le corps des réels, $\mathbb{N}$ l'ensemble des entiers positifs ou nuls, $\mathbb{Z}_n$ l'ensemble des entiers positifs ou nuls et strictement plus petits que n, pour $n \in \mathbb{N} \setminus \{0, 1\}$. L'ensemble $\mathbb{Z}_n$ muni de l'addition et de la multiplication modulo n est un anneau noté $\mathbb{Z}/n\mathbb{Z}$, qui sera très utilisé en théorie des codes. On dit qu'un anneau est *euclidien* s'il est muni de la division euclidienne, c'est-à-dire que pour tout couple d'éléments a et b ($a \geqslant b$) de cet ensemble, il existe q et r tels que $a = bq + r$ et $|r| < |b|$. Les nombres q et r sont respectivement le quotient et le reste[2] de la division euclidienne, et notés $q = a \text{ div } b$ et $r = a \mod b$ (pour a *modulo* b). Or tout anneau euclidien est principal. Ceci implique l'existence d'un *plus grand commun diviseur* (pgcd) pour tout couple d'éléments (a, b). Ce pgcd est le générateur de l'idéal $Aa + Ab$.

[2]le reste r peut *a priori* être négatif, mais dans ce cas $s = r + |b|$ vérifie également $a = (q \pm 1)b + s$ et $s = |s| < |b|$; dans la suite, nous prenons donc toujours r positif.

Si p est un nombre premier, l'anneau $\mathbb{Z}/p\mathbb{Z}$ est un corps de caractéristique p. En effet, le théorème classique de Bézout (voir page 48) nous apprend que quels que soient deux entiers a et b, il existe des entiers x et y tels que

$$ax + by = \text{pgcd}(a, b).$$

Si p est premier et a est un élément non nul de $\mathbb{Z}_p$, cette identité appliquée à a et p donne $ax + bp = 1$, soit $ax = 1 \mod p$, donc a est inversible et x est son inverse. On note $\mathbb{F}_p$ ce corps.
Le corps des nombres rationnels $\mathbb{Q}$ et les corps $\mathbb{F}_p$, sont appelés *corps premiers*.

Algèbre linéaire

Des rudiments d'algèbre linaire sont nécessaires à la lecture d'une grande partie de ce livre. Sans prétention explicative, nous définissions ici les concepts utiles et introduisons les notations principales.
Un ensemble de vecteur $x_1, \ldots, x_n$ est un ensemble *indépendant* si pour tous scalaires $\lambda_1, \ldots, \lambda_n$, $\sum_{i=1}^{n} \lambda_i x_i = 0$ implique $\lambda_1 = \cdots = \lambda_n = 0$.
La *dimension* d'un espace vectoriel V, notée $\dim(V)$ est le cardinal du plus grand ensemble de vecteurs indépendants de cet espace.
Par exemple, si V est un corps, V^n est un espace de dimension n car les vecteurs $(1, 0, \ldots, 0)$, $(0, 1, 0, \ldots, 0)$, $\ldots$, $(0, \ldots, 0, 1)$. sont indépendants.
Une *application linéaire* est une application d'une structure dans une autre, qui préserve les lois de composition. C'est-à-dire, si $(A, +)$ et $(B, \times)$ sont deux groupes, une application f est dite linéaire si pour tous x, y de a, $f(x + y) = f(x) \times f(y)$. Si A et B sont des espaces vectoriels, on demande également que la loi de composition externe soit préservée, à savoir $f(\lambda.x) = \lambda.f(x)$.
L'*image* d'une application linéaire f d'un espace vectoriel $\mathbb{E}$ sur un espace vectoriel $\mathbb{F}$, notée $\text{Im}(f)$, est l'ensemble des vecteurs $y \in \mathbb{F}$ tels qu'il existe $x \in \mathbb{E}$, $f(x) = y$.
Le *noyau* d'une application linéaire f d'un espace vectoriel $\mathbb{E}$ sur un espace vectoriel $\mathbb{F}$, noté $\text{Ker}(f)$, est l'ensemble des vecteurs $x \in \mathbb{E}$ tels que $f(x) = 0$.
Il est facile de vérifier que $Ker(f)$ et $Im(f)$ sont des espaces vectoriels.
Si $\text{Im}(f)$ est de dimension finie, celle-ci est appelée le *rang* de l'application linéaire et est noté $\text{Rg}(f) = \dim(\text{Im}(f))$. Si en outre, $\mathbb{E}$ est également de dimension finie alors on a $\dim(\text{Ker}(f)) + \dim(\text{Im}(f)) = \dim(\mathbb{E})$.
Une *matrice* M de taille (m, n) est un élément de l'espace vectoriel $V^{m \times n}$, représenté par un tableau de m lignes de taille n, ou n colonnes de taille m. L'élément de la ligne i et e la colonne j de la matrice est noté $M_{i,j}$. La multiplication d'un vecteur x de taille n par une telle matrice donne un vecteur y de taille m vérifiant $y_i = \sum_{k=1}^{n} M_{i,k} x_k$ pour i de 1 à m ; cette multiplication est notée $y = Mx$.

À toute matrice M correspond une application linéaire f de V^m dans V^n, définie par $f(x) = Mx$. Réciproquement, toute application linéaire peut être écrite ainsi sous forme matricielle. Les procédés de codage que nous verrons tout au long de ce livre, et principalement au chapitre 4, font largement appel aux applications linéaires, ce qui permet de montrer rigoureusement les propriétés de ces fonctions.

1.3.3 Codage bijectif d'un bloc

Maintenant que nous disposons de structures dans lesquelles nos blocs peuvent être additionnés, multipliés, divisés (selon la division euclidienne), et posséder un inverse, nous donnons ici des exemples fondamentaux de calculs qu'il est possible de faire dans des ensembles qui possèdent une bonne structure algébrique. Les blocs étant de tailles finies, les ensembles manipulés dans cette section sont finis.

Inverse modulaire : algorithme d'Euclide

Le théorème de Bézout garantit l'existence des coefficients et donc de l'inverse d'un nombre modulo un nombre premier, et c'est l'*algorithme d'Euclide* qui permet de les calculer efficacement.
Dans sa version fondamentale, l'algorithme d'Euclide calcule le pgcd (*plus grand commun diviseur*) de deux nombres entiers. Le principe est : en supposant que $a > b$,

$$\text{pgcd}(a, b) = \text{pgcd}(a - b, b) = \text{pgcd}(a - 2b, b) = \cdots = \text{pgcd}(a \mod b, b)$$

où $a \mod b$ est le reste de la division euclidienne de a par b. En effet, si a et b ont un diviseur commun d alors $a - b, a - 2b, \ldots$ sont aussi divisibles par d. Un principe récursif se dessine :

Algorithme 3 PGCD : algorithme d'Euclide.

Entrées Deux entiers a et b, $a \geqslant b$.
Sorties $\text{pgcd}(a, b)$
  Si $b = 0$ **Alors**
    Renvoyer a ;
  Sinon
    Calculer récursivement $\text{pgcd}(b, a \mod b)$ et renvoyer le résultat ;
  Fin Si

Exemple : Déterminons le pgcd de 522 et 453. On calcule successivement :

$$\begin{aligned}
&\text{pgcd}(522, 453)\\
=\ &\text{pgcd}(453, 522 \mod 453 = 69)\\
=\ &\text{pgcd}(69, 453 \mod 69 = 39)\\
=\ &\text{pgcd}(39, 69 \mod 39 = 30)\\
=\ &\text{pgcd}(30, 39 \mod 30 = 9)\\
=\ &\text{pgcd}(9, 30 \mod 9 = 3)\\
=\ &\text{pgcd}(3, 9 \mod 3 = 0)\\
=\ &3.
\end{aligned}$$

Le pgcd de 522 et 453 est égal à 3.

Algorithme d'Euclide étendu. La version dite « étendue » de l'algorithme d'Euclide, celle que nous emploierons très souvent dans cet ouvrage, permet, en plus du calcul du pgcd de deux nombres, de trouver les coefficients de Bézout. Il est étendu aussi parce qu'on veut le rendre un peu plus générique en lui donnant la possibilité de l'appliquer non seulement à des ensembles de nombres mais aussi à n'importe quel anneau euclidien. Ce sera le cas des polynômes, comme nous le verrons dans les sections suivantes.
Le principe de l'algorithme est d'itérer la fonction G suivante :

$$G\ :\ \begin{bmatrix} a \\ b \end{bmatrix} \longmapsto \begin{bmatrix} 0 & 1 \\ 1 & -(a \text{ div } b) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}.$$

Exemple : Pour trouver x et y tel que $x \times 522 + y \times 453 = \text{pgcd}(522, 453)$, on écrit les matrices correspondant à l'itération avec la fonction G, on a :

$$\begin{aligned}
\begin{bmatrix} 3 \\ 0 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -6 \end{bmatrix} \\
&\quad \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 522 \\ 453 \end{bmatrix} \\
&= \begin{bmatrix} 46 & -53 \\ -151 & 174 \end{bmatrix} \begin{bmatrix} 522 \\ 453 \end{bmatrix}
\end{aligned}$$

On obtient ainsi $-151 \times 522 + 174 \times 453 = 3$. D'où $d = 3$, $x = -151$ et $y = 174$.
On donne une version de l'algorithme d'Euclide étendu qui réalise ce calcul en ne stockant que la première ligne de G. Il affecte les variables x, y et d de façon à vérifier en sortie : $d = \text{pgcd}(a, b)$ et $ax + by = d$.
Pour une résolution « à la main », on pourra calculer récursivement les équations (E_i) suivantes (en s'arrêtant lorsque $r_{i+1} = 0$) :

$$\begin{array}{lrcrcl}
(E_0): & 1 \times a & + & 0 \times b & = & a \\
(E_1): & 0 \times a & + & 1 \times b & = & b \\
(E_{i+1}) = (E_{i-1}) - q_i(E_i) & u_i \times a & + & v_i \times b & = & r_i
\end{array}$$

Algorithme 4 PGCD : algorithme d'Euclide étendu.

Entrées Deux éléments a et b d'un ensemble euclidien, $a \geqslant b$.
Sorties $d = \text{pgcd}(a, b)$ et x,y tels que $ax + by = d$
  Si $b = 0$ **Alors**
    Renvoyer $d = a$, $x = 1$, $y = 1$;
  Sinon
    Appeler récursivement Euclide étendu sur b, $a \mod b$;
    Soient d, x, y les éléments du résultat.
    Calculer $u = x - (a \text{ div } b) * y$;
    Renvoyer d, y, u ;
  Fin Si

Un exemple de résolution par cette méthode est donné en solution de l'exercice suivant.

Exercice 1.11 (Algorithme d'Euclide étendu). *On demande de trouver les coefficients de Bézout pour les nombres entiers suivants :*
– $(a, b) = (50, 17)$
– $(a, b) = (280, 11)$
– $(a, b) = (50, 35)$ *Solution page 281.*

Nous donnons maintenant la preuve que cet algorithme est correct, cela fournit également une preuve constructive du théorème de Bézout.

Théorème 1 (Bézout). *Soient a et b deux entiers relatifs non nuls et d leur* pgcd. *Il existe deux entiers relatifs x et y tels que $|x| < |b|$ et $|y| < |a|$ vérifiant l'égalité de Bézout $ax + by = d$ et l'algorithme d'Euclide étendu est correct.*

Preuve. Tout d'abord, montrons que la suite des restes est toujours divisible par $d = \text{pgcd}(a, b)$: par récurrence si $r_{j-2} = kd$ et $r_{j-1} = hd$ alors $r_j = r_{j-2} - q_j r_{j-1} = d(k - q_j h)$ et donc $\text{pgcd}(a, b) = \text{pgcd}(r_{j-1}, r_j)$. Ensuite, la suite des restes positifs r_j est donc strictement décroissante et minorée par 0 donc converge. Cela prouve que l'algorithme termine.
En outre, la suite converge forcément vers 0, puisque sinon on peut toujours continuer à diviser le reste. Il existe donc un indice i tel que $r_{i-1} = q_{i+1} r_i + 0$. Dans ce cas, $\text{pgcd}(r_{i-1}, r_i) = r_i$ et la remarque précédente indique donc que r_i est bien le pgcd recherché.
Il reste à montrer que les coefficients conviennent. Nous procédons par récurrence. Clairement le cas initial $a \mod b = 0$ convient et l'algorithme est correct dans cas. Ensuite, notons $r = a \mod b$ et $q = a \text{ div } b$, alors $a = bq + r$. Par récurrence, avec les notations internes de l'algorithme 4 on a que $d = xb + yr$ avec $|x| < b$ et $|y| < r$. Cette relation induit immédiatement

que $d = ya + (x - qy)b$ avec $|y| < r < b$ et $|x - qy| \leqslant |x| + q|y| < b + qr = a$. Ce qui prouve que l'algorithme est correct. □

Exercice 1.12 (Calcul modulaire). *L'algorithme d'Euclide Étendu permet aussi de résoudre des équations linéaires modulaires. Donner une méthode qui résout les équations suivantes :*

1. $17x = 10 \mod 50$
2. $35x = 10 \mod 50$
3. $35y = 11 \mod 50$

Solution page 281.

Complexité de l'algorithme d'Euclide. À chaque étape, le plus grand des nombres est au moins divisé par deux, donc sa taille diminue d'au moins 1 bit. Le nombre d'étapes est par conséquent majoré par $O(\log_2(a) + \log_2(b))$. À chaque étape, on fait aussi un calcul de reste de division euclidienne. Les algorithmes appris à l'école primaire pour effectuer une division euclidienne ont un coût de $O(\log_2^2(a))$. Le coût total est finalement majoré par $O(\log_2^3(a))$, soit $O(n^3)$ si n est la taille des données. Une étude plus fine de l'algorithme peut toutefois préciser cette complexité. En effet, le coût de l'algorithme d'Euclide est plutôt de l'ordre de $O(\log_2^2(a))$! La preuve est technique, mais l'idée est assez simple : soit il y a effectivement de l'ordre de $O(\log_2(a))$ étapes, mais alors chaque quotient est très petit et alors chaque division et multiplication peut se faire en seulement $O(\log_2(a))$ opérations, soit les quotients sont grands et chaque division et multiplication doit se faire en $O(\log_2^2(a))$ opérations, mais alors le nombre d'étapes est constant.

Exercice 1.13. *Traduisez l'algorithme d'Euclide étendu dans votre langage de programmation préféré.* *Solution (en C++) page 282.*

L'indicatrice d'Euler et le théorème de Fermat

Soit $n \geqslant 2$ un entier. On note $\mathbb{Z}_n^*$ l'ensemble des entiers strictement positifs plus petits que n et premiers avec n :

$$\mathbb{Z}_n^* = \{x \in \mathbb{N} : 1 \leqslant x < n \text{ et } \operatorname{pgcd}(x, n) = 1\}.$$

Le cardinal de $\mathbb{Z}_n^*$ est noté $\varphi(n)$. La fonction φ est appelée *indicatrice d'Euler*. Par exemple, $\varphi(8) = 4$. De plus, si p est un nombre premier, $\varphi(p) = p - 1$. Une formule plus générale fait l'objet de l'exercice 1.16.
Dans $\mathbb{Z}_n^*$, tout élément x a un inverse : en effet, comme x est premier avec n, l'identité de Bézout assure l'existence de deux entiers de signes opposés u et

v ($1 \leqslant |u| < n$ et $1 \leqslant |v| < x$), tels que :

$$u.x + v.n = 1$$

On a alors $u.x = 1 \mod n$, *i.e.* $u = x^{-1} \mod n$. On appelle u l'*inverse de x modulo n*. Le calcul de u se fait grâce à l'algorithme d'Euclide étendu.

Théorème 2 (Euler). *Soit a un élément quelconque de $\mathbb{Z}_n^*$. On a : $a^{\varphi(n)} = 1 \mod n$.*

Preuve. Soit a un élément quelconque de $\mathbb{Z}_n^*$. L'ensemble des produits des éléments de $\mathbb{Z}_n^*$ par a, $G_a = \{y = ax \mod n \text{ pour } x \in \mathbb{Z}_n^*\}$ est égal à $\mathbb{Z}_n^*$. En effet, quel que soit $y \in \mathbb{Z}_n^*$, on peut poser $x = a^{-1}y$ puisque a est inversible et réciproquement si a et x sont inversibles modulo n alors leur produit l'est aussi ($(ax)^{-1} = x^{-1}a^{-1} \mod n$). Ainsi, comme ces deux ensembles sont égaux, les produits respectifs de tous leurs éléments sont aussi égaux modulo n :

$$\prod_{x \in \mathbb{Z}_n^*} x = \prod_{y \in G_a} y = \prod_{x \in \mathbb{Z}_n^*} ax \mod n$$

Or, comme la multiplication est commutative dans $\mathbb{Z}_n^*$, on peut alors sortir les a du produit, et comme il y a $\varphi(n)$ éléments dans $\mathbb{Z}_n^*$ on obtient la formule suivante modulo n :

$$\prod_{x \in \mathbb{Z}_n^*} x = a^{\varphi(n)} \prod_{x \in \mathbb{Z}_n^*} x \mod n$$

La conclusion vient alors du fait que les éléments de $\mathbb{Z}_n^*$ étant inversibles, leur produit l'est aussi et donc, en simplifiant, on obtient $1 = a^{\varphi(n)} \mod n$. □

Le théorème de Fermat se déduit directement du théorème d'Euler dans le cas où n est un nombre premier.

Théorème 3 (Fermat). *Si p est premier, alors tout $a \in \mathbb{Z}_p$ vérifie : $a^p = a \mod p$.*

Preuve. Si a est inversible, alors le théorème d'Euler nous donne $a^{p-1} = 1 \mod p$. En remultipliant par a on obtient la relation désirée. Le seul non-inversible de $\mathbb{Z}_p$ si p est premier est 0. Dans ce cas, on a immédiatement $0^p = 0 \mod p$. □

Le théorème chinois, originellement formulé par le mathématicien chinois Qin Jiu-Shao au XIII$^{\text{ième}}$ siècle, permet de combiner plusieurs congruences modulo des nombres premiers entre eux, pour obtenir une congruence modulo le produit de ces nombres.

Théorème 4 (Théorème chinois). *Soient $n_1, \ldots, n_k$ des entiers positifs premiers entre eux deux à deux, et $N = \prod n_i$. Alors, pour tout ensemble de nombres entiers $a_1, \ldots, a_k$, il existe une unique solution $x \leqslant N$ au système d'équations modulaires $\{x = a_i \mod m_i$, pour $i = 1..k\}$. En posant $N_i = \frac{N}{n_i}$, cette unique solution est donnée par :*

$$x = \sum_{i=1}^{k} a_i N_i N_i^{-1 \mod n_i} \mod \prod_{i=1}^{k} n_i$$

Preuve. Nous procédons en deux temps : nous montrons d'abord l'existence de x puis l'unicité. Comme les n_i sont premiers entre eux, N_i et n_i sont premiers entre eux. Le théorème d'Euclide nous assure alors l'existence de l'inverse de N_i modulo n_i, notée $y_i = N_i^{-1 \mod n_i}$. Nous posons ensuite $x = \sum_{i=1}^{n} a_i y_i N_i \mod N$ et il est facile de vérifier que x ainsi défini convient ! En effet, pout tout i, $x = a_i y_i N_i \mod n_i$ puisque n_i divise tous les N_j avec $j \neq i$. Mais alors la définition de y_i donne $x = a_i.1 = a_i \mod n_i$.
Il nous reste à prouver l'unicité de x. Supposons qu'il en existe deux, x_1 et x_2. Alors $x_2 - x_1 = 0 \mod n_1$ et $x_2 - x_1 = 0 \mod n_2$. Donc $x_2 - x_1 = k_1 n_1 = k_2 n_2$ pour certains k_1 et k_2. Donc n_1 divise $k_2 n_2$. Or n_1 et n_2 sont premiers entre eux, donc n_1 doit diviser k_2 ; ainsi $x_2 - x_1 = 0 \mod n_1 n_2$. En procédant par récurrence, comme n_{i+1} est premier avec le produit $n_1 n_2 \ldots n_i$, on en déduit que $x_2 - x_1 = 0 \mod N$, ou encore que $x_2 = x_1 \mod N$, ce qui montre bien l'unicité de la solution. □

Exercice 1.14. *Trouver tous les x entiers tels que $x = 4 \mod 5$ et $x = 5 \mod 11$. En déduire l'inverse de 49 modulo 55.* *Solution page 282.*

Exercice 1.15. *Trouver tous les x entiers dont les restes par 2, 3, 4, 5, et 6 sont respectivement 1, 2, 3, 4, 5.* *Solution page 282.*

Exercice 1.16 (Une formule pour l'indicatrice d'Euler).

1. *On sait que si p est premier, $\varphi(p) = p - 1$. Calculer $\varphi(n)$ si $n = p^k$ avec p premier et $k \in \mathbb{N}^*$.*
2. *Montrer que φ est* multiplicative, i.e. *si m et n sont premiers entre eux, alors $\varphi(mn) = \varphi(m)\varphi(n)$.*
3. *En déduire la formule générale la formule d'Euler, en utilisant la décomposition d'un nombre en facteurs premiers.*

Solution page 282.

L'exponentiation modulaire et le logarithme discret

L'exponentiation modulaire est une fonction de codage dont le principe est très largement utilisé dans les méthodes modernes de chiffrement.

$$\begin{array}{rccl} E_a : & \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ & b & \longrightarrow & a^b \mod n \end{array}$$

Elle est associée à sa fonction de décodage. Pour c dans $\mathbb{Z}_n$, le plus petit entier positif b tel que $a^b = c \mod n$ est appelé le *logarithme discret* (ou encore l'*index*) en base a de b modulo n ; on note $b = \log_a c \mod n$.

$$\begin{array}{rccl} D_a : & \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ & c & \longrightarrow & \log_a c \mod n \end{array}$$

La fonction de codage est facile à calculer. La méthode est appelée *élévation récursive au carré*. Elle consiste en la décomposition de b en carrés successifs. Par exemple :

$$a^{11} = a \times a^{10} = a \times (a^5)^2 = a \times (a \times a^4)^2 = a \times (a \times ((a)^2)^2)^2$$

Avec ce principe, le calcul de a^{11} ne nécessite que 5 multiplications : 3 élévations au carré et 2 multiplications.
De manière générale, la complexité de l'algorithme 5 est $O(\log_2 n)$ multiplications modulo n.

Algorithme 5 Puissance Modulaire.

Entrées Trois entiers $a \neq 0$, b et $n \geqslant 2$.
Sorties $a^b \mod n$

```
Si b = 0 Alors
  Renvoyer 1 ;
Sinon
  Calculer récursivement la puissance modulaire a^⌊b/2⌋ mod n
  Soit d le résultat
  Calculer d = d * d mod n ;
  Si b impair Alors
    Calculer d = d * a mod n ;
  Fin Si
  Renvoyer d ;
Fin Si
```

En effet, à chaque appel, l'exposant b est divisé par 2. Il y a donc au plus $\log_2 b$ appels récursifs. Lors de chaque appel, on effectue au plus 2 multiplications :

une élévation au carré et éventuellement une multiplication par a. Ces opérations sont faites modulo n, donc sur des nombres de $\log_2 n$ chiffres. Même en utilisant les algorithmes de multiplication naïfs (ceux vus à l'école primaire), le coût d'une telle multiplication est $O(\log_2^2 n)$.
Le coût final de l'algorithme est donc $O(\log_2 b \log_2^2 n)$. Ce coût est raisonnable par rapport à $O(\log_2 n)$ qui est le temps nécessaire pour la lecture de a ou l'écriture du résultat.

Exercice 1.17 (Calculs d'inverse).

1. *Proposer un algorithme de calcul de l'inverse dans $\mathbb{Z}/n\mathbb{Z}$ fondé sur le théorème d'Euler.*
   Application : calculer (rapidement) $22^{-1} \mod 63$ *et* $5^{2001} \mod 24$. *On pourra utiliser :* $22^2 \mod 63 = 43; 22^4 \mod 63 = 22$.
2. *Donner trois algorithmes différents pour calculer l'inverse de y modulo $N = p_1^{\delta_1}.p_2^{\delta_2}.\ldots.p_k^{\delta_k}$, où les p_i sont des entiers premiers distincts.*

Solution page 283.

Le problème du logarithme discret (qu'on note DLP, pour *Discrete Logarithm Problem*) est le calcul inverse de la puissance modulaire. Mais si la puissance modulaire est un calcul réalisable en temps raisonnable comme nous venons de le voir, il n'en est pas de même pour le logarithme discret. On tirera de cette dissymétrie des principes fondamentaux pour la cryptographie.
Le résultat suivant est connu sous le nom de théorème du logarithme discret. Un *générateur* de l'ensemble $\mathbb{Z}_n^*$ est un nombre g tel que $\{g^i,\ i \in \mathbb{N}\} = \mathbb{Z}_n^*$.

Théorème 5 (logarithme discret). *Si g est un générateur de $\mathbb{Z}_n^*$, alors pour tous $x, y \in \mathbb{N}$: $g^x = g^y \mod n$ si et seulement si $x = y \mod \varphi(n)$.*

Preuve. ($\Leftarrow$) Si $x = y \mod \varphi(n)$, on a $x = y + k * \varphi(n)$. Or $g^{\varphi(n)} = 1 \mod n$, d'où $g^x = g^y \mod n$.
($\Rightarrow$) Comme la séquence des puissances de g est périodique de période $\varphi(n)$, alors $g^x = g^y \mod n \Longrightarrow x = y \mod \varphi(n)$. □

Mais cela ne permet pas le calcul du logarithme discret avec une complexité raisonnable. Étant donné y, il est difficile de calculer x tel que $g^x = y$. La seule méthode simple consiste en l'énumération exhaustive de tous les x possibles, et prend un temps $O(n)$. Aucun algorithme polynomial en $\log_2 n$ (la taille de l'entrée) n'est connu pour ce problème.
Ainsi, si $n = 10^{100}$, le calcul de la puissance modulaire demande moins de 10^8 opérations, soit moins de 1 seconde sur un PC. Par contre, l'énumération exhaustive pour calculer le logarithme discret demande 10^{100} opérations, ce qui, nous l'avons vu, est impensable en un temps raisonnable!!!

En pratique, on peut, pour tenter de résoudre le logarithme discret, appliquer des principes similaires aux algorithmes de factorisation. Ce sont des méthodes qui trouvent les diviseurs des nombres composés, que nous verrons en fin de chapitre.
Ce type de fonctions, qu'on peut calculer rapidement dans un sens et pas dans l'autre, est crucial pour la théorie des codes, en particulier pour la cryptographie à clef publique.

Fonctions à Sens Unique

Dans des systèmes cryptographiques dits à clef publique, le « système de codage » doit être connu, tandis que le système de décodage doit être inconnu. Dans cet exemple de codage par exponentiation modulaire et décodage par logarithme discret, le fait que la fonction de codage E soit connue et la fonction de décodage D soit inconnue semble a priori contradictoire : si l'on connaît E, on connaît forcément D puisque $D = E^{-1}$.
En fait, en remplaçant « inconnu » par « extrêmement long à calculer sur un ordinateur » (*i.e.* plusieurs années par exemple), les fonctions E et D d'un système de cryptographie à clef publique doivent vérifier :
– $D = E^{-1}$ pour garantir $D(E(M)) = M$;
– il est facile (i.e. très rapide) de calculer $\tilde{M} = E(M)$ à partir de M ;
– il est difficile (i.e. très long) de retrouver M à partir de $\tilde{M}$.
Autrement dit, il faut trouver une fonction E de chiffrement qui soit rapide à calculer mais longue à inverser. On parle de *fonction à sens unique* (parfois abrégé par FSU). C'est une notion tout à fait cruciale en cryptographie, sur laquelle sont fondés tous les codes modernes. Le principe est illustré dans la figure 1.7.

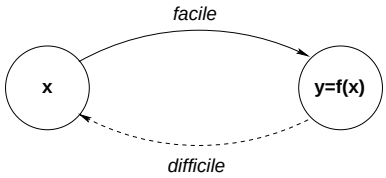


FIG. 1.7: Principe d'une fonction à sens unique.

Le paramétrage par une clef ajoutera la facilité à décoder en possédant la clef, et la difficulté à décoder sans la clef.
De bonnes FSU sont des fonctions telles que la recherche de x à partir de $F(x)$ soit un problème mathématique réputé difficile.

L'intérêt de calculer en arithmétique modulaire est double. D'une part, les calculs « modulo n » sont très rapides : ils ont un coût $O(\log^2 n)$ avec les algorithmes les plus naïfs. D'autre part, les itérations d'une fonction F même simple calculées en arithmétique modulo n tendent à avoir un comportement aléatoire. Ce type de calcul est d'ailleurs utilisé dans la plupart des générateurs de nombres aléatoires, comme nous le verrons à la section 1.3.6. Connaissant F et n grand, il apparaît difficile de résoudre l'équation : trouver x tel que $F(x) = a \mod n$, donc d'inverser la fonction F.

1.3.4 Construction des corps premiers et corps finis

Nous avons mentionné que nous chercherons le plus possible à donner à nos codes des structures de corps, afin de faciliter les opérations. Nous avons maintenant une première façon de générer un bon code : le corps premier. Il suffit de choisir un nombre premier p, et de munir l'ensemble $\{0, \ldots, p-1\}$ de l'addition et de la multiplication modulo p. Mais « choisir un nombre premier » n'est pas une tâche évidente. C'est un domaine d'étude à part entière, dont nous donnons un aperçu afin de ne laisser aucun vide algorithmique dans nos méthodes de codage.

Tests de primalité et génération de nombres premiers

Si l'on ne connaît pas d'algorithme pour factoriser un entier n en temps polynomial $\log_2^{O(1)} n$, il est revanche possible de générer un nombre premier p rapidement. En théorie des codes, il est très utile de savoir générer des nombres premiers, à la fois pour construire des codes structurés comme des corps, et donc aisément manipulables pour la correction d'erreurs, ou encore pour fabriquer des systèmes de cryptographie sûrs. Pour cela, on utilise des tests de primalité, c'est-à-dire des algorithmes qui déterminent si un nombre donné est premier ou non. En prenant un grand nombre impair n, et en lui appliquant le test, on peut, s'il est composé, recommencer avec $n+2$ jusqu'à trouver un nombre premier. Le nombre de nombres premiers inférieurs à n est asymptotiquement $n/\ln(n)$. On en déduit que partant de n impair, en moyenne $O(\ln(n))$ itérations suffisent pour trouver un nombre premier en ajoutant 2 à n à chaque itération.

Le test de primalité le plus utilisé est un test proposé par Miller, et rendu efficace en pratique par Rabin. Pour cela, le test de Miller-Rabin est un algorithme qui détermine si un nombre donné est probablement premier. La réponse donnée par le calcul est donc seulement probabiliste, c'est-à-dire que celle-ci peut être erronée. Néanmoins, s'il l'on répète ce test suffisamment de fois et qu'il donne constamment la même réponse, la probabilité d'erreur devient de plus en plus faible et finalement négligeable.

Test de Miller-Rabin. Soit n un nombre impair et soient s et t tels que $n-1 = t2^s$ avec t impair. Pour un entier $a < n$ quelconque, on a

$$a^{(n-1)} - 1 = a^{t2^s} - 1 = (a^t - 1)(a^t + 1)(a^{2t} + 1) \dots (a^{(2^{s-1})t} + 1).$$

Si n est premier, d'après le théorème de Fermat, $a^{(n-1)} - 1 = 0 \mod n$; donc
– soit $a^t - 1 = 0 \mod n$;
– soit $a^{t2^i} + 1 = 0 \mod n$ avec $0 \leqslant i < s$.
Le test de composition de Miller-Rabin est basé sur cette propriété.

Algorithme 6 Test de primalité de Miller-Rabin.

Entrées Un entier $n \geqslant 4$.
Sorties Soit n est composé, soit il est probablement premier.
Soient s et t tels que $n-1 = t2^s$
Soit a un nombre entier aléatoire entre 0 et $n-1$.
Soit $q \leftarrow a^t \mod n$
Si $q \neq 1$ ou $q \neq n-1$ **Alors**
Renvoyer « n est composé »
Fin Si
Appliquer s fois $q \leftarrow q * q \mod n$, et à chaque étape
Si $q \neq n-1$ **Alors**
Renvoyer « n est composé »
Fin Si
Si on n'a pas trouvé de preuve que n est composé, **Alors**
Renvoyer « n est probablement premier ».
Fin Si

On dit que a réussit le test de composition de Miller-Rabin pour n si $a^t - 1 \neq 0 \mod n$ et si $a^{t2^i} + 1 \neq 0 \mod n$ pour tout $i = 0, \dots, s-1$. En effet, dans ce cas, $a^{(n-1)} - 1 \neq 0 \mod n$; en utilisant le théorème de Fermat, on en déduit que n n'est pas premier (il est composé).
Si n est impair et non premier, moins de $(n-1)/4$ nombres a échouent au test de composition de Miller-Rabin. En choisissant a au hasard dans $\{1, \dots, n-1\}$, la probabilité d'échouer est donc inférieure à $\frac{1}{4}$.

Ce test peut être utilisé efficacement pour fabriquer un nombre premier avec une probabilité inférieure à 4^{-k} de se tromper. On procède comme suit :

1. On tire au hasard un nombre impair n ;
2. On tire au hasard k nombres a_i distincts, $1 < a_i < n$. On applique le test de composition Miller-Rabin pour chaque entier a_i.

3. Si aucun a_i ne réussit le test de composition, on en déduit que n est premier ; la probabilité de se tromper est inférieure à 4^{-k} ;
4. Sinon, on recommence en remplaçant n par $n + 2$.

La complexité d'un test de Miller-Rabin est similaire à celle de l'élévation à la puissance modulo, soit $O(\log_2^3 n)$; et $O(k \log_2^3 n)$ si l'on veut une probabilité d'erreur de l'ordre de 4^{-k}. Le coût moyen de l'algorithme de génération de nombre premier est donc majoré par $O(k \log^4 n)$. En effet, puisqu'il y a environ $\frac{n}{\ln(n)}$ nombres premiers inférieurs à n, il faudra en moyenne de l'ordre de $\ln(n)$ essais pour trouver un nombre premier.
En pratique, avec ce test il est facile de générer un nombre premier de 1000 chiffres décimaux avec une probabilité d'erreur arbitrairement faible.
En outre, il est possible de rendre déterministe l'algorithme de Miller-Rabin en testant suffisamment de nombres a. Par exemple, Burgess a prouvé que tester tous les nombres a jusqu'à seulement $n^{0.134}$ était suffisant pour obtenir une certitude. Toutefois, le test devient alors exponentiel en la taille de n. Finalement, un théorème de 1990 montre qu'il suffit de tester les $2 \log_2 n$ premiers entiers, en admettant l'Hypothèse de Riemann Généralisée. Les études théoriques montrent donc que ce test est très efficace et sûr.

Test AKS. Mentionnons enfin pour être complets un autre test de primalité très récent, le test AKS, dû à Agrawal, Kayal et Saxena, qui ont montré en 2002 qu'il existe un algorithme déterministe polynomial pour tester si un nombre est premier ou non, sans le recours de l'hypothèse de Riemann. Malgré ce résultat théorique important, on préférera en pratique les algorithmes probabilistes, nettement plus performants à ce jour.
L'idée est voisine de celle de Miller-Rabin : si n est premier, alors pour tout a

$$(X - a)^n = (X^n - a) \mod n.$$

L'algorithme AKS vérifie cette égalité pour un certain nombre de témoins a, en développant explicitement $(X - a)^n$. Pour rendre le test polynomial, il faut réduire la taille des polynômes (ce qui est fait en effectuant le test modulo $(X^r - 1)$ pour r vérifiant certaines propriétés[3]) et il faut suffisamment de témoins a, mais seulement de l'ordre de $log^{O(1)}(n)$.

On sait construire des grands corps premiers. Mais ce ne sont pas les seuls corps finis qui existent. Pour construire un corps fini de n'importe quelle taille (y compris si cette taille n'est pas un nombre premier), pourvu qu'il en existe un, nous aurons besoin d'introduire l'anneau des polynômes sur un corps quelconque.

[3] r doit être premier avec n, le plus grand facteur premier q de r doit vérifier $q \geqslant 4\sqrt{r} \log n$, et enfin il faut que $n^{\frac{r-1}{q}} \neq 1 \mod r$.

Arithmétique des polynômes

Soit V un corps. L'anneau des polynômes à coefficients dans V est

$$V[X] = \left\{ P = \sum_{i=0}^{d} a_i X^i / d \in \mathbb{N}, (a_0, \dots a_d) \in V^{d+1} \right\}.$$

On appelle *degré* du polynôme P, et on note $d = \deg(P)$ le plus grand entier naturel d tel que $a_d \neq 0$. Par convention, le degré du polynôme nul est noté $\deg(0) = -\infty$. Un polynôme est *unitaire* si son coefficient de plus haut degré est l'élément neutre pour la multiplication de V.
Dans cet anneau, on peut définir une division euclidienne : pour tous $A, B \in V[X]$, il existe $Q, R \in V[X]$ uniques avec $\deg(R) < \deg(B)$ tels que :

$$A = B.Q + R \ .$$

Le polynôme $Q = A$ div B est le *quotient* dans la division euclidienne de A par B ; le *reste* R est aussi noté $A \mod B$.
L'anneau $V[X]$ est donc un anneau euclidien.
La notion de pgcd est définie ; l'algorithme d'Euclide appliqué à deux polynômes non nuls A et B fournit un polynôme de degré maximal (unique si on le choisit unitaire) qui divise à la fois A et B. Par ailleurs, l'identité de Bézout est valide. Autrement dit, si A et B sont deux polynômes dans $V[X]$ et si $D \in V[X]$ est leur pgcd, alors il existe deux polynômes S et T dans $V[X]$ tels que

$$A.S + B.T = D \ .$$

L'*algorithme d'Euclide étendu* permet de calculer effectivement deux polynômes S et T dont les degrés respectifs sont strictement inférieurs à ceux de A div D et B div D.
Deux polynômes A et B sont dits *premiers entre eux* s'ils admettent le polynôme 1 comme pgcd ; autrement dit, A et B n'admettent aucun facteur commun de degré non nul. On dit qu'un polynôme P de $V[X]$ est un *polynôme irréductible* sur V lorsque P est premier avec tout polynôme de degré inférieur à $\deg(P)$.
Tout comme un entier admet une décomposition en facteurs premiers, tout polynôme de degré non nul admet une décomposition unique en produit de puissances de facteurs irréductibles sur V ; on dit que $V[X]$ est un *anneau factoriel*. Autrement dit, on peut décomposer tout polynôme de degré non nul A de $V[X]$ sous la forme

$$A = P_1^{d_1} \dots P_k^{d_k}$$

où les d_i sont des entiers non nuls et les polynômes P_i sont irréductibles sur V. Si A est unitaire, les facteurs P_i peuvent être choisis unitaires : la décomposition est alors unique à une permutation d'indices près.

Un élément α de V est une *racine* de $A \in V[X]$ si $A(\alpha) = 0$.
Si α est une racine de A, alors $(X - \alpha)$ divise A. Soit B le polynôme tel que $A = (X-\alpha).B$. On dit que α est racine *simple* de A si α n'est pas racine de B, i.e. $B(\alpha) \neq 0$. Sinon, dans le cas où $B(\alpha) = 0$, on dit que α est racine *multiple* de A.

Exemple : Dans $\mathbb{F}_2[X]$ le polynôme $X^3 - 1$ se décompose en :

$$X^3 - 1 = (X - 1) \cdot (X^2 + X + 1) \ .$$

On vérifie aisément que $X^2 + X + 1$ est irréductible (les seuls polynômes de $\mathbb{F}_2[X]$ de degré non nul et inférieur à 2 sont X et $X - 1$ et aucun d'entre eux ne divise $X^2 + X + 1$).

L'anneau $V[X]/P$ et les corps finis

Soit $(V, +, \times)$ un corps et soit P un polynôme de degré $d \geqslant 1$. On note $V[X]/P$ l'ensemble des polynômes de degré inférieur strictement à d muni des opérations arithmétiques d'addition et de multiplication modulo P, à savoir, pour tous polynômes A, B de $V[X]$, avec $\deg(A) < d$ et $\deg(B) < d$:

$$\begin{aligned} A +_{V[X]/P} B &= (A +_{V[X]} B) \mod P \\ A \times_{V[X]/P} B &= (A \times_{V[X]} B) \mod P \end{aligned}$$

C'est un anneau commutatif unitaire. qui admet les polynômes 0 et 1 comme éléments neutres respectivement pour les lois + et $\times$. Cet anneau est appelé *anneau quotient* de $V[X]$ par P.
Si P est un polynôme irréductible, alors $V[X]/P$ est un corps. En effet, si Q est un polynôme non nul de degré inférieur à $\deg P$, Q est P sont premiers entre eux et l'identité de Bézout s'écrit : $AQ + BP = 1$, soit $AQ = 1 \mod P$, autrement dit, Q est inversible dans l'anneau quotient $V[X]/P$.

Exemple : Sur le corps $V = \mathbb{F}_2$:
Si $P = (X + 1)(X^2 + X + 1)$ (non irréductible), l'anneau $V[X]/P$ est :

$$V[X]/P = \left\{0, 1, X, 1 + X, X^2, 1 + X^2, X + X^2, 1 + X + X^2\right\} \ .$$

Cet anneau n'est pas un corps car $(1+X)(1+X+X^2) = 0$ montre que $1+X$ n'est pas inversible. Au contraire, si l'on prend $P = X^2 + X + 1$ (irréductible), l'anneau $V[X]/P$ est : $V[X]/P = \{0, 1, X, 1 + X\}$. Cet anneau est un corps puisque $X(X + 1) = 1$.
On dispose donc d'une classe de corps finis plus étendue que les corps premiers, puisque ce dernier exemple nous fournit un corps à 4 éléments, qui n'est pas un corps premier.

Les corps finis sont appelés *corps de Galois*. Ils sont notés $\mathbb{F}_q$, où q est le cardinal du corps. La propriété suivante nous permet de caractériser tous les corps finis et de justifier la notation $\mathbb{F}_q$ pour « le » corps fini de cardinal q.

Propriété 5. *Deux corps finis de même cardinal sont isomorphes.*

Pour construire un corps fini, il faut donc disposer d'un polynôme irréductible d'un degré fixé. Comme pour la génération de nombres premiers, c'est là un sujet à part entière, dont nous donnons un aperçu.

Polynômes irréductibles

Pour fabriquer des corps finis, nous allons avoir besoin de polynômes irréductibles, comme nous avions besoin des nombres premiers pour construire les corps premiers. De la même façon que pour les tests de primalité des nombres vus à la section 1.3.4, nous commençons par donner un test permettant de reconnaître les polynômes irréductibles.
Un premier test facile à réaliser est de s'assurer que le polynôme est sans carré, c'est-à-dire qu'il ne contient pas comme diviseur le carré d'un autre polynôme. Cela se fait sur un corps fini comme dans tout autre corps en examinant sa dérivée terme à terme.

Propriété 6. *Un polynôme P est sans carré si et seulement si* $\text{pgcd}(P, P') = 1$

Preuve. Si P est divisible par un carré alors $P = g^2 h$ pour certains polynômes h et g. Il s'en suit que $P' = 2g'gh + g^2h' = g(2g'h + gh')$ et donc g divise le pgcd de P et P'. Réciproquement, si $g = \text{pgcd}(P, P')$, prenons un facteur irréductible γ de g de degré au moins 1. Alors $P' = \gamma f$ et $P = \gamma\lambda$ avec f et λ deux polynômes. En dérivant P on obtient $\gamma f = \gamma'\lambda + \gamma\lambda'$, ou encore $\gamma(f - \lambda') = \gamma'\lambda$. Le polynôme γ étant irréductible et γ' de degré strictement inférieur au degré de γ, γ et γ' sont premiers entre eux. Alors γ divise forcément λ et donc γ^2 divise P. □

Ensuite, le principe du test d'irréductibilité est donné par la propriété suivante.

Proposition 1. *Pour p premier et $d \geqslant 1$, dans $\mathbb{F}_p[X]$, le polynôme $X^{p^d} - X$ est le produit de tous les polynômes unitaires irréductibles dont le degré divise d.*

Pour prouver cette proposition, nous avons tout d'abord besoin du lemme suivant :

Lemme 2. *$r > 0$ divise $d > 0$ si et seulement si $p^r - 1$ divise $p^d - 1$.*

Preuve. Si r divise d, alors $p^d = (p^r)^k = 1 \mod p^r - 1$. Réciproquement, on a $p^d - 1 = 0 \mod p^r - 1$. Supposons que $d = qr + s$, avec $s < r$, on a alors $p^d - 1 = p^{qr}p^s - 1 = p^s - 1 \mod p^r - 1$. Or, $0 \leqslant s < r$, on obtient donc $p^s - 1 = 0$ sur les entiers, ce qui implique que $s = 0$. Donc r divise d. □

Preuve. [de la proposition 1] Soit P irréductible de degré r divisant d. Alors $V = \mathbb{F}_p[X]/P$ est un corps. Dans V, l'ordre de tout élément divise le cardinal du groupe de ses inversibles, à savoir $p^r - 1$ (voir la section 1.3.2). On applique cette propriété à $X \in V$, ce qui nous certifie que $X^{p^r-1} = 1$. Or, $p^r - 1$ divise $p^d - 1$ grâce au lemme, donc $X^{p^d-1} = 1 \mod P$, et P divise donc $X^{p^d-1} - 1$. Réciproquement, soit P un diviseur irréductible de $X^{p^d} - X$ de degré r. Si $r = 1$, r divise forcément d. Sinon, $X^{p^d-1} = 1 \mod P$. Or, $\mathbb{F}_p[X]/P$ est un corps de cardinal p^r et donc $X^{p^r-1} = 1$ dans ce corps par le théorème d'Euler, ou encore, $X^{p^r-1} = 1 \mod P$. Posons $p^d - 1 = (p^r - 1)q + s$, avec $0 \leqslant s < p^r - 1$. On a donc $X^s = 1 \mod P$. Or, s est inférieur au degré r de P, donc $X^s = 1$, ce qui n'est possible que si $s = 0$. Le lemme permet de conclure que r divise bien s. Il reste à montrer qu'aucun carré ne divise $X^{p^d} - X$. En effet, son polynôme dérivé est $p^d X^{p^d-1} - 1 = -1 \mod P$ et le polynôme -1 est premier avec tout autre polynôme. □

Les facteurs de $X^{p^d} - X$ sont donc tous les polynômes irréductibles unitaires de degré divisant d. Si un polynôme de degré d n'a aucun facteur commun avec $X^{p^i} - X$ pour $1 \leqslant i \leqslant d/2$, il est irréductible. On peut construire à partir de cette propriété le test dit de Ben-Or (voir algorithme 7).

Algorithme 7 Test d'irréductibilité de Ben-Or.

Entrées Un polynôme, $P \in \mathbb{F}_p[X]$.
Sorties « P est réductible » ou « P est irréductible ».

```
Soit P' le polynôme dérivé de P.
Si pgcd (P, P') ≠ 1 Alors
    Renvoyer « P est réductible ».
Fin Si
Soit Q ← X
Pour i ← 1 à deg(P)/2 Faire
    Q ← Q^p mod P
    Si pgcd(Q − X, P) ≠ 1 Alors
        Renvoyer « P est réductible » (arrêt de l'algorithme).
    Fin Si
Fin Pour
Renvoyer « P est irréductible ».
```

Ainsi, en utilisant ce test, il est possible de tirer des polynômes au hasard et d'avoir une bonne chance de tomber sur un polynôme irréductible. On note

$m_r(p)$ le nombre de polynômes unitaires irréductibles de degré r dans $\mathbb{F}_p[X]$. Comme $X^{p^r} - X$ est le produit de tous les polynômes irréductibles de degré divisant r, on obtient

$$\frac{1}{r}(p^r - p^{\lfloor \frac{r}{2} \rfloor + 1}) \leqslant m_r(p) \leqslant \frac{1}{r}p^r \tag{1.8}$$

En effet, p^r est le degré de $X^{p^r} - X$, donc $p^r = \sum_{d|r} d m_d(p) \geqslant r m_r(p)$. Donc $m_r(p) \leqslant p^r/r$. D'un autre côté, $p^r = \sum_{d|r} d m_d(p) \geqslant r m_r(p)$ implique $p^r - r m_r \leqslant \sum_{d|r} p^d \leqslant \sum_{d \leqslant \lfloor r/2 \rfloor} p^d$. Cette dernière est une série géométrique qui vaut $\frac{p^{\lfloor r/2 \rfloor + 1} - 1}{p-1} < p^{\lfloor r/2 \rfloor + 1}$. Finalement, $\frac{1}{r}(p^r - p^{\lfloor \frac{r}{2} \rfloor + 1}) \leqslant m_r(p)$.

Cela montre que parmi les polynômes de degré r, environ un sur r est irréductible. Pour construire un polynôme irréductible, on peut donc en première approche choisir un polynôme au hasard, tester son irréductibilité, et recommencer jusqu'à tomber sur un polynôme irréductible. En moyenne il faudra r tirages pour trouver un polynôme adéquat. Cependant, pour faciliter les calculs avec ces polynômes, il est intéressant d'obtenir des polynômes creux, c'est-à-dire avec très peu de coefficients non nuls. Dans ce cas, la recherche systématique peut s'avérer plus efficace en pratique.
Nous proposons l'algorithme hybride 8 produisant de préférence un polynôme irréductible creux, et un polynôme irréductible au hasard à l'aide du test de Ben-Or, si le précédent s'avère trop difficile à calculer. Il est fondé sur l'idée de prendre des polynômes du type $X^r + g(X)$ avec $g(X)$ au hasard de faible degré par rapport à r.

Algorithme 8 Recherche d'un polynôme irréductible creux.

Entrées Un corps fini $\mathbb{F}_p$, un entier $r > 0$.
Sorties Un polynôme irréductible de $\mathbb{F}_p[X]$, de degré r.

```
Pour d = 2 à r − 1 Faire
  Pour tout a, b ∈ F_q, a ≠ 0 Faire
    Si (X^r + bX^d + a) est irréductible Alors
      Renvoyer X^r + bX^d + a
    Fin Si
  Fin Pour
Fin Pour
Répéter
  Sélectionner P, unitaire de degré r, au hasard dans F_q[X].
Jusqu'à ce que P soit irréductible
Renvoyer P.
```

Constructions des corps finis

Nous disposons de tous les éléments pour construire un corps fini de taille p^n, où p est un nombre premier. La méthode de construction des corps finis est la preuve du résultat suivant :

Théorème 6. *Pour tout nombre premier p et tout entier $d > 0$, il existe un corps K à p^d éléments et ce corps est unique à un isomorphisme près.*

Preuve. Soit p un nombre premier et $\mathbb{F}_p[X]$ l'anneau des polynômes à coefficients dans $\mathbb{F}_p$. D'après l'équation (1.8), il existe au moins un polynôme P irréductible de degré d dans $\mathbb{F}_p[X]$. L'anneau quotient $V = \mathbb{F}_p[X]/P$ est alors un corps.
Puisque P est de degré d et que $|\mathbb{F}_p| = p$, alors il y a p^d restes possibles et par conséquent $|V| = p^d$.
D'après la proposition proposition 5 de la page 60, tout corps de cardinal p^d est isomorphe à V. □

Remarque 1. *L'isomorphisme entre $V[X]/P$ et $V^{\deg(P)}$ confère à $V^{\deg(P)}$ une structure de corps.*

En effet, à tout vecteur $u = [u_0, \ldots, u_{d-1}]$ de l'espace vectoriel V^d, on peut associer de manière bijective le polynôme $\psi(u) = \sum_{i=0}^{d-1} u_i X^i$ et en plus, on a la propriété suivante :

$$\text{Pour tous } u, v \in V^d,\ \lambda \in V,\quad \psi(u + \lambda \cdot v) = \psi(u) + \lambda \cdot \psi(v) \quad .$$

Ainsi, ψ est un isomorphisme entre V^d et $\psi(V^d) = V[X]/P$. Ceci confère à V^d une structure de corps dans laquelle la multiplication est définie par :

$$\text{Pour tous } u, v \in V^d, \qquad u \cdot v = \psi^{-1}(\psi(u) \cdot \psi(v)) \ .$$

On peut ainsi utiliser une structure de corps avec les vecteurs de $V^{\deg(P)}$.

Exercice 1.18. *Soit K un corps fini de cardinal $q > 0$. À l'aide de l'application $\Psi : \mathbb{Z} \to K$, définie par :*

$$\textit{Pour tout } n \in \mathbb{Z},\ \Psi(n) = \underbrace{1_K + 1_K + \ldots + 1_K}_{n \textit{ fois}} = n.1_K,$$

montrer qu'il existe un unique nombre premier p, dit caractéristique de K, tel que pour tout $x \in K$, $px = 0$. *Solution page 284.*

Exercice 1.19. *Suite de l'exercice précédent*
En déduire que le cardinal de K est une puissance de p, en se servant du fait que K est un espace vectoriel sur ses sous-corps. Indication : exhiber un sous-corps de K isomorphe à $\mathbb{F}_p$. *Solution page 284.*

Exercice 1.20 (Construction du corps $\mathbb{F}_4$).

1. *Donner une condition nécessaire et suffisante pour qu'un polynôme dans $\mathbb{F}_2[X]$ de degré $2 \leqslant n \leqslant 3$ soit irréductible. En déduire tous les polynômes irréductibles de degré 2 et 3.*
2. *En déduire les polynômes irréductibles de degré 4.*
3. *Soit $\mathbb{F}_4 = \{e_0, e_1, e_2, e_3\}$, avec la convention e_0 élément neutre pour l'addition et e_1 élément neutre pour la multiplication. En utilisant la première question, expliciter les tables d'opérations ($+$, $\times$, inverse) dans $\mathbb{F}_4$.*

Solution page 284.

1.3.5 Implémentations des corps finis

Opérations sur les polynômes

Une construction classique de l'arithmétique dans un corps fini est donc, pour un nombre premier p, d'implémenter $\mathbb{F}_p$, de chercher un polynôme irréductible P dans $\mathbb{F}_p[X]$ de degré d, puis de représenter les éléments de $\mathbb{F}_q = \mathbb{F}_p[X]/P$ par des polynômes, ou des vecteurs, et d'implémenter les opérations arithmétiques comme des opérations modulo p et P.

Exemple : Construction du corps $\mathbb{F}_{16}$.
Il existe un corps à 16 éléments, puisque $16 = 2^4$. Pour construire le corps $\mathbb{F}_{16}$, on cherche d'abord un polynôme P irréductible de degré 4 dans $\mathbb{F}_2[X]$, puis on établit les règles de calcul dans $\mathbb{F}_2[X]/P$.

– *Détermination de P.*
Le polynôme irréductible P s'écrit $P = X^4 + aX^3 + bX^2 + cX + 1$ avec a, b et c dans $\mathbb{F}_2$. Pour déterminer P, examinons les différentes valeurs possibles pour le triplet (a, b, c). On ne peut avoir $(a, b, c) \in \{(0, 1, 1), (1, 0, 1), (1, 1, 0), (0, 0, 0)\}$ puisque dans chacun de ces cas, 1 est racine de P. Le triplet (a, b, c) est donc à chercher dans $\{(0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$. Les seuls polynômes irréductibles sur $\mathbb{F}_2$ de degré au plus égal à deux sont X, $1 + X$ et $X^2 + X + 1$. Pour chacun des quatre cas restants, il suffit de chercher le pgcd de P et $(1 + X)(X^2 + X + 1)$. Le calcul (par l'algorithme d'Euclide par exemple) de ces pgcd montre que les seules valeurs de (a, b, c) pour lesquelles P est irréductible sont $(0, 0, 1)$, $(1, 0, 0)$ et $(1, 1, 1)$. Donc, $P = X^4 + X^3 + 1$ est un choix possible de P. Faisons ce choix.

– *Opérations sur les polynômes.*
Les éléments du corps sont alors $X, X^2, X^3, 1 + X^3, 1 + X + X^3, 1 + X + X^2 + X^3, 1 + X + X^2, X + X^2 + X^3, 1 + X^2,\ X + X^3, 1 + X^2 + X^3, 1 +$

$X, X + X^2, X^2 + X^3, 1$. Les opérations se font alors modulo P et modulo 2. Par exemple, $(X^2)(X + X^3) = 1 + X$.

Utilisation des générateurs

Il existe d'autres possibilités pour implémenter les opérations dans les corps finis, avec lesquelles la multiplication pourra être réalisée beaucoup plus rapidement.
L'idée consiste à utiliser la propriété des corps finis selon laquelle le groupe multiplicatif des inversibles d'un corps fini est cyclique, c'est-à-dire qu'il existe au moins un générateur et que le corps est généré par les puissances de ce générateur. Ainsi, si g est un générateur du groupe multiplicatif d'un corps fini $\mathbb{F}_q$, tous les inversibles peuvent s'écrire g^i.
On peut choisir d'implémenter chaque élément inversible g^i simplement par son indice i, et zéro par un indice spécial. Cette construction où l'on représente les éléments par leurs logarithmes est appelée *représentation exponentielle* ou *représentation cyclique* ou encore *construction de Zech*. Les opérations arithmétiques classiques sont alors grandement simplifiées, en utilisant la proposition suivante :

Proposition 2. *Soient $\mathbb{F}_q$ un corps fini et g un générateur de F_q^*. Alors $g^{q-1} = 1_{\mathbb{F}_q}$. En outre, si la caractéristique de $\mathbb{F}_q$ est impaire, alors $g^{\frac{q-1}{2}} = -1_{\mathbb{F}_q}$ et dans le cas contraire, $1_{F_{2^n}} = -1_{\mathbb{F}_{2^n}}$.*

Preuve. L'ordre de g divise $q-1$ donc $g^{q-1} = 1_{\mathbb{F}_q}$. Si le corps est de caractéristique 2, alors, comme dans $\mathbb{F}_2[X]$, $1 = -1$. Sinon $\frac{q-1}{2} \in \mathbb{Z}$ et donc $g^{\frac{q-1}{2}} \in \mathbb{F}_q$. Or comme on est dans un corps, l'équation $X^2 = 1$ possède au plus deux racines, 1 et -1. Or g est un générateur donc l'ordre de g est $q-1$ et non pas $\frac{q-1}{2}$. La seule possibilité restante est $g^{\frac{q-1}{2}} = -1$. □

Cela donne le codage suivant pour un élément $x \in \mathbb{F}_q$ si $\mathbb{F}_q$ est généré par g :

$$\begin{cases} 0 & \text{si } x = 0 \\ q-1 & \text{si } x = 1 \\ i & \text{si } x = g^i \end{cases}$$

En particulier, dans notre codage, notons $\overline{q} = q - 1$ le représentant de $1_{\mathbb{F}_q}$. Nous notons aussi i_{-1} l'indice de $-1_{\mathbb{F}_q}$; il vaut $\frac{q-1}{2}$ si $\mathbb{F}_q$ est de caractéristique impaire et $q-1$ sinon. Ce qui donne la possibilité d'écrire simplement les opérations arithmétiques.

– La multiplication et la division d'inversibles sont respectivement une addition et une soustraction d'indices modulo $q-1$.

– La négation est donc simplement l'identité en caractéristique 2 et l'addition de $\frac{q-1}{2}$ modulo $q-1$ en caractéristique impaire.
– L'addition est l'opération la plus complexe. Il faut l'implémenter en utilisant les autres opérations, par exemple de cette manière : si g^i et g^j sont deux éléments non nuls d'un corps fini, $g^i + g^j = g^i(1 + g^{j-i})$. En construisant une table, $t\_plus1[]$, de taille q, des successeurs de chaque élément du corps, l'addition est implémentée par une soustraction d'indice, un accès à une table et une addition d'indices.

Opération	Éléments	Indices	Coût		
			+/-	Tests	Accès
Multiplication	$g^i * g^j$	$i + j\ (-\overline{q})$	1.5	1	0
Division	g^i / g^j	$i - j\ (+\overline{q})$	1.5	1	0
Négation	$-g^i$	$i - i_{-1}\ (+\overline{q})$	1.5	1	0
Addition	$g^i + g^j$	$k = j - i\ (+\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3	2	1
Soustraction	$g^i - g^j$	$k = j - i + i_{-1}\ (\pm\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3.75	2.75	1

TAB. 1.3: Opérations sur les inversibles avec générateur en caractéristique impaire.

Dans le tableau 1.3, nous montrons le calcul de ces opérations sur les indices, en considérant une seule table de taille q, celle des successeurs. Nous nous intéressons ici à la complexité du calcul en utilisant le moins de mémoire possible, en considérant des éléments aléatoires. Nous indiquons le coût des calculs en nombre moyen d'additions et de soustractions (+/-), en nombre de tests et en nombre d'accès dans une table.

Racines primitives

Pour mettre en œuvre cette implémentation, et de la même façon que nous avons eu besoin de générateurs de nombres premiers pour construire les corps premiers, nous devons maintenant donner un moyen de trouver des générateurs des corps finis.

Générateurs des corps premiers. Un générateur du groupe des inversibles de $\mathbb{Z}/n\mathbb{Z}$ est appelé *racine primitive* de n. La plus petite racine primitive de m est notée $\chi(m)$.
Si p est un nombre premier, alors il possède toujours exactement $\varphi(p-1)$ racines primitives. Il faut maintenant en calculer au moins une. On utilise

pour cela le test suivant, qui teste si l'ordre d'un élément pris au hasard est bien $p-1$. La principale difficulté est de factoriser $p-1$, au moins partiellement, nous verrons comment faire cela en section 1.4.3.

Algorithme 9 Test Racine Primitive.

Entrées Un entier premier $p > 0$.
Entrées Un entier $a > 0$.
Sorties Oui, si a est une racine primitive de p ; Non dans le cas contraire.
  Pour tout q, premier et divisant $p-1$, **Faire** {Factorisation de $p-1$}
    Si $a^{\frac{p-1}{q}} = 1 \mod p$ **Alors**
      Renvoyer « Non ».
    Fin Si
  Fin Pour
  Renvoyer « Oui ».

Théorème 7. *L'algorithme Test Racine Primitive est correct.*

Preuve. On utilise le résultat de la section 1.3.2 : soit un entier a, d'ordre k modulo p, alors $a^h = 1 \mod p$ si et seulement si $k|h$.
On en déduit que si l'ordre de a est plus petit que $p-1$, comme il doit diviser $p-1$, nécessairement l'une des valeurs $\frac{p-1}{q}$ sera un multiple de l'ordre de a. Dans le cas contraire, la seule valeur possible pour l'ordre de a est $p-1$. □

Une première méthode pour trouver une racine primitive est alors d'essayer un à un tous les entiers plus petits que p, qui ne soient ni 1, ni -1, ni une puissance sur les entiers, et de trouver ainsi la plus petite racine primitive de p. De nombreux résultats théoriques existent prouvant qu'en général, il ne faut pas trop d'essais pour la trouver, de l'ordre de

$$\chi(p) = O\big(r^4(\log(r)+1)^4 \log^2(p)\big)$$

avec r le nombre de facteurs premiers distincts de p
Une autre méthode est de tirer aléatoirement des entiers plus petits que p et de tester si ceux-ci sont une racine primitive ou non. Étant donné qu'il y a $\varphi(p-1)$ racines primitives, la probabilité d'en trouver une est de $\frac{\varphi(p-1)}{p-1}$ et donc l'espérance du nombre de tirages pour tomber sur une racine primitive est de $\frac{p-1}{\varphi(p-1)}$. Ce qui nous donne une meilleure chance que la force brute (essayer toutes les possibilités).

Générateurs des corps finis. Nous savons maintenant trouver un générateur pour un corps premier. Nous nous intéressons maintenant aux corps finis :

les F_{p^k}. Pour les construire, rappelons qu'il faut d'abord construire F_p, puis un polynôme de degré k irréductible sur ce corps. Il s'agit maintenant de trouver un *polynôme générateur* de ce corps pour pouvoir coder les éléments non plus par des polynômes, mais par leurs indices. Le codage et les opérations sont alors les *mêmes* que ceux des corps premiers.
Nous utilisons à nouveau un algorithme probabiliste. Nous montrons tout d'abord un algorithme testant si un polynôme est générateur dans $\mathbb{F}_p[X]$. Cet algorithme est similaire à celui développé pour les racines primitives dans $\mathbb{F}_p$.

Algorithme 10 Test Polynôme Générateur.

Entrées Un polynôme $A \in \mathbb{F}_p[X]$.
Entrées Un polynôme F irréductible de degré d dans $\mathbb{F}_p[X]$.
Sorties Oui, si A est générateur du corps $\mathbb{F}_p[X]/F$; Non dans le cas contraire.
  Pour tout q, premier et divisant $p^d - 1$ **Faire** {Factorisation de $p^d - 1$}
    Si $A^{\frac{p^d-1}{q}} = 1 \mod F$ **Alors** {Calcul récursif par carrés}
      Renvoyer « Non ».
    Fin Si
  Fin Pour
  Renvoyer « Oui ».

Un algorithme cherchant au hasard un générateur, une fois le corps construit, est alors aisé. En outre, on peut restreindre l'ensemble de recherche à des polynômes de faible degré ($O(\log(n))$). Toutefois, toujours par souci d'obtenir des polynômes creux, nous montrons qu'il est possible de trouver rapidement, sur un corps premier, un polynôme irréductible pour lequel X est une racine primitive. Un tel polynôme est appelé *X-Irréductible*, ou *primitif*. En pratique, pour les corps finis de taille comprise entre 4 et 2^{32}, il est possible de montrer que plus d'un polynôme irréductible sur 12 est X-Irréductible ! Un algorithme recherchant un polynôme X-irréductible au hasard nécessite ainsi moins de 12 essais en moyenne. L'algorithme pour trouver un polynôme irréductible ayant X comme générateur est donc une simple modification de l'algorithme 8 ne sélectionnant pas le polynôme irréductible trouvé si l'algorithme 10 répond que X n'est pas un générateur.

Exemple : Reprenons l'exemple du corps $\mathbb{F}_{16}$, construit avec le polynôme générateur $X^4 + X^3 + 1$.
L'algorithme 10 testé avec X répond que X est un générateur. On peut donc mener les calculs avec les puissances de X.
Identification des éléments de $\mathbb{F}_{16}$ et règles pour les opérations.
Le calcul des puissances successives de X donne

$X^1 = X \mod P$; $X^2 = X^2 \mod P$; $X^3 = X^3 \mod P$; $X^4 = 1 + X^3 \mod P$; $X^5 = 1 + X + X^3 \mod P$; $X^6 = 1 + X + X^2 + X^3 \mod P$; $X^7 = 1 + X + X^2 \mod P$; $X^8 = X + X^2 + X^3 \mod P$; $X^9 = 1 + X^2 \mod P$; $X^{10} = X + X^3 \mod P$; $X^{11} = 1 + X^2 + X^3 \mod P$; $X^{12} = 1 + X \mod P$; $X^{13} = X + X^2 \mod P$; $X^{14} = X^2 + X^3 \mod P$; $X^{15} = 1 \mod P$.

Il en résulte que X est un élément générateur du groupe multiplicatif $\mathbb{F}_{16}$. Donc :

$\mathbb{F}_{16} = \{0, 1, X, X^2, X^3, X^4, X^5, X^6, X^7, X^8, X^9, X^{10}, X^{11}, X^{12}, X^{13}, X^{14}\}$.

Avec $\mathbb{F}_{16}$ donné sous cette forme, la multiplication et le calcul de l'inverse dans $\mathbb{F}_{16}$ se font plus aisément. L'addition s'en trouve aussi simplifiée si l'on tient compte du fait que $X^k + X^t = X^t(1 + X^{k-t})$ pour tout k et t dans $\{1, \ldots, 14\}$ tels que $k > t$.

1.3.6 Générateurs pseudo-aléatoires

La génération de nombres au hasard est employée très souvent dans toutes les méthodes que nous venons de voir, et souvent encore dans ce livre. En particulier, générer des nombres aléatoirement est une condition de la perfection du code à clef jetable de Vernam (voir page 29). Il est temps maintenant de s'attaquer à ce problème qui mérite un petit développement.

Les définitions du hasard sont cruciales pour la théorie des codes. En effet, tout message qui présente un mode d'organisation descriptible par une formule plus courte que le message lui-même est un angle d'attaque pour la compression, aussi bien que pour le cassage des codes. On a donc tout intérêt à se parer d'une théorie solide concernant le hasard et l'aléa pour concevoir des codes efficaces et sûr.

La production d'un évènement vraiment aléatoire est insoluble par des ordinateurs, qui ne répondent par définition qu'à des procédures déterminées et prévisibles. Pour obtenir des valeurs qui seraient la production du « vrai » hasard (même si cette notion n'est pas tout à fait absolue, et fait référence à ce qu'on peut en observer et en prédire), il faut faire appel à des phénomènes physiques réputés imprévisibles, comme le bruit thermique, la description du mouvement brownien des électrons dans une résistance.

Mais il est possible que cette production ne soit appelée hasard que parce que nous sommes incapables, étant donné l'état de nos connaissance, d'en expliquer le fonctionnement, et seule les théories probabilistes permettent de l'appréhender. Avec les ordinateurs, on tente de procéder de même pour générer des nombres aléatoires. On applique des procédures qui rendent imprévisible le résultat en pratique. C'est ce qu'on appelle les générateurs pseudo-aléatoires.

La production de nombres aléatoires est une tâche très complexe, qui a solli-

cité depuis plusieurs dizaines d'années, à la fois les concepteurs de machines (production matérielle « hardware », comme le bruit thermique) que de logiciels (production logicielle « software », comme nous allons en voir quelques exemples). Si on n'accorde pas assez d'attention à ce problème crucial, des méthodes efficaces (nous en verrons quelques-unes) existent pour repérer toutes les composantes non aléatoires dans une séquence supposée l'être, reconstituer ainsi la méthode qui l'a produite, et casser une clef générée au hasard. Si la machine qui génère la combinaison gagnante du loto n'obéit pas à un bon processus de génération du hasard, on pourra aussi prédire la prochaine combinaison gagnante.

On génère souvent une séquence de nombres pseudo-aléatoires en calculant chaque nombre à partir du précédent (ce qui bien sûr rend le processus complètement déterministe et donc élimine tout hasard), d'une façon suffisamment compliquée pour qu'en examinant la séquence sans connaître la méthode, on puisse croire au hasard.

Un générateur doit vérifier certaines propriétés pour être qualifié de pseudo-aléatoire. Tous les nombres générés doivent être indépendants les uns des autres, avoir une grosse entropie, et aucune règle ne doit pouvoir être reconstituée à partir de la suite des nombres générés. Il y a plusieurs façons de tester si un générateur est acceptable. D'abord, lui faire passer des tests statistiques pour vérifier que la distribution qu'il produit ne diffère pas significativement de celle qui est attendue par un modèle théorique du hasard. Ou alors, on peut aussi employer des principes de complexité algorithmique : il est possible de prouver qu'en temps raisonnable, aucun algorithme ne pourra prédire le fonctionnement du générateur.

Par exemple, un générateur peut être construit sur le modèle de la suite de Fibonacci, en produisant les nombres $x_n = x_{n-1} + x_{n-2} \mod m$, m étant l'entier maximum pouvant être produit. Cette méthode a l'avantage d'être très simple à implémenter, très rapide à exécuter, et le modulo permet un comportement difficilement prévisible. Mais ce générateur, comme la plupart des générateurs simples et classiques, a des défauts, et il est possible de reconstituer leur comportement à partir d'analyses statistiques.

Les propriétés requises pour un générateur pseudo-aléatoire ressemblent fortement aux propriétés qu'on attend d'un message crypté. En effet, il doit être impossible en recevant un message, ou un nombre, de reconstituer la manière dont il a été produit, sans en connaître la clef. C'est pourquoi certaines méthodes de génération de nombres aléatoires ressemblent aux méthodes de cryptographie, ou les utilisent.

Les générateurs congruentiels

On appelle *générateur congruentiel linéaire* un générateur qui suit le principe suivant : si x_i, $i \in \mathbb{N}$ est la suite de nombres aléatoires générés, on calcule x_i à partir de son prédécesseur : $x_i = ax_{i-1} + b \mod m$, où m est un grand nombre, et $a, b \in \mathbb{Z}_m$. Le générateur est dit multiplicatif si $b = 0$.
Une telle séquence est toujours périodique. il faudra choisir a, b, m de façon à ce que la période soit suffisamment grande. Par exemple, pour $m = 10, x_0 = a = b = 7$, la période est 4, et ce générateur n'est pas du tout satisfaisant.
La période maximale est évidemment m. Il existe un résultat décrivant tous les générateurs de période m avec $b = 0$:

Théorème 8. *Le générateur congruentiel linéaire défini par $a, b = 0, m, x_0$ est de période m si et seulement si x_0 est premier avec m et a est une racine primitive de m.*

On choisit habituellement pour m le plus grand nombre premier descriptible par une machine (on a vu comment générer ce nombre dans cette section à la page 55).
Évidemment, une grande période n'est pas un critère suffisant pour les générateurs aléatoires, en témoignent les choix $a = 1, b = 1$.
L'exercice 1.32, page 94 est une approche des méthodes d'attaque des générateurs congruentiels linéaires.

Les registres à décalage linéaire

Une généralisation des générateurs congruentiels linéaires consiste à ne plus utiliser seulement la précédente valeur pour fabriquer l'élément suivant de la séquence, mais plusieurs des précédentes valeurs, c'est-à-dire que x_n est calculé par des combinaisons linéaires de $x_{n-1}, \ldots x_{n-k}$. Autrement dit :

$$x_n = (a_1 x_{n-1} + \ldots + a_k x_{n-k}) \mod m$$

Ces générateurs sont particulièrement intéressants si m est un nombre premier car leur période maximale est alors $m^k - 1$. Ainsi il est possible, même avec un petit modulo, d'avoir de très grandes périodes.
Par exemple, pour générer des suites aléatoires de bits, on choisit $m = 2$. Dans ce cas, les opérations peuvent être réalisées très rapidement sur machine par des « ou exclusifs » (xor) pour l'addition modulo 2 et par des décalages des bits x_i pour générer les bits suivants. Il existe même des puces spécialisées réalisant les opérations nécessaires. On parle alors de *registres à décalage linéaire*, abrégé par *LFSR* (de l'anglais *Linear Feedback Shift Registers*). La figure 1.8 en résume le fonctionnement.

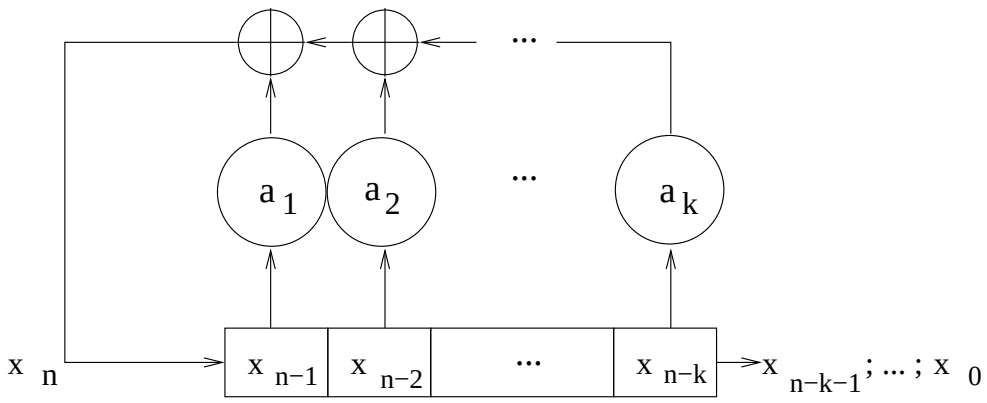


FIG. 1.8: Schéma de fonctionnement d'un LFSR.

Pour certains calculs, il est intéressant d'écrire LFSR par une forme polynomiale : soit $\Pi(X) = X^k - a_1 X^{k-1} - \ldots - a_k$.
Ainsi, $LFSR_\Pi(x_0, \ldots, x_{k-1})$ désigne la séquence infinie des x_i linéairement générée par le polynôme Π avec pour k valeurs initiales les $x_0, \ldots, x_{k-1}$.

Exercice 1.21. *Produire les 8 premières valeurs générées par le registre à décalage linéaire* $LFSR_{X^4+X^3+X^2+1}(0,1,1,0)$. *Solution page 285.*

Enfin, nous avons l'équivalent du théorème 8 des générateurs congruentiels linéaires où la racine primitive est remplacée par un polynôme primitif.

Théorème 9. *Pour un polynôme* Π *de degré* k*, le* $LFSR_\Pi$ *modulo un nombre premier* p *est de période maximale* $p^k - 1$ *si et seulement si* Π *est primitif dans* $\mathbb{F}_p$.

Ces générateurs sont très rapides et possèdent en outre une très grande période. Nous verrons cependant en section 1.4.3 que l'algorithme de Berlekamp-Massey permet de prévoir les bits suivants sans connaître le polynôme générateur, pourvu que $2k$ valeurs successives aient été interceptées.
Ces générateurs sont utilisés en pratique pour générer rapidement des bits avec de bonnes propriétés statistiques mais doivent être combinés à d'autres générateurs pour être cryptographiquement sûrs.

Exemple : Sécurisation du protocole *Bluetooth*
Bluetooth est une technologie radio courte distance destinée à simplifier les connexions entre les appareils électroniques. Elle a été conçue dans le but de remplacer les câbles entre les ordinateurs et leurs périphériques comme les

imprimantes, les scanners, les souris, les téléphones portables, les PC de poche ou encore les appareils photos numériques.

La sécurisation de ce protocole utilise un schéma de type chiffrement de Vernam (voir page 1.2.1) mais avec un générateur pseudo-aléatoire à base de $LFSR$: l'algorithme de chiffrement utilise quatre LFSR de longueurs respectives 25, 31, 33 et 39, pour un total de $25 + 31 + 33 + 39 = 128$ bits. Les 128 bits de la valeur initiale représentent la clef secrète du chiffrement *Bluetooth*. La figure 1.9 indique le schéma de fonctionnement de ce chiffrement.

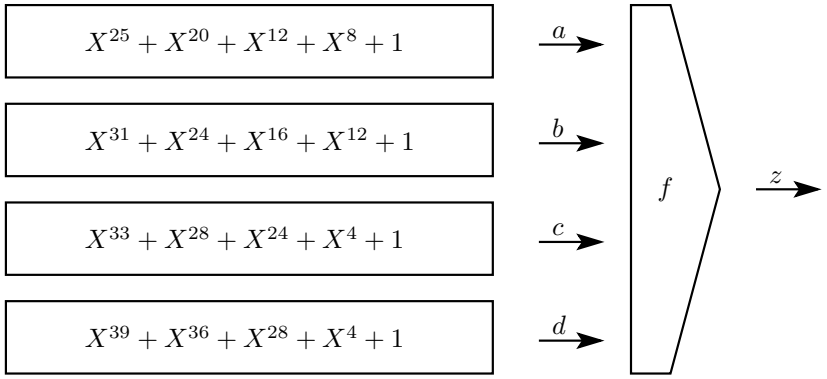


FIG. 1.9: Chiffrement *Bluetooth*.

Nous voyons que les quatre polynômes utilisés sont :
- $X^{39} + X^{36} + X^{28} + X^4 + 1$;
- $X^{33} + X^{28} + X^{24} + X^4 + 1$;
- $X^{31} + X^{24} + X^{16} + X^{12} + 1$;
- $X^{25} + X^{20} + X^{12} + X^8 + 1$.

Ces quatre polynômes sont primitifs modulo 2 pour une période totale de $\text{ppcm}(2^{39} - 1, 2^{33} - 1, 2^{31} - 1, 2^{25} - 1) = 7 \cdot 23 \cdot 31 \cdot 79 \cdot 89 \cdot 601 \cdot 1801 \cdot 8191 \cdot 121369 \cdot 599479 \cdot 2147483647 \approx 2^{125}$.

Les 4 bits $\{a; b; c; d\}$ produits par ces 4 LFSR successifs sont ensuite combinés par une fonction discrète non-linéaire $f$qui produit le prochain bit de sortie z_t, à partir de son seul état initial ($\{cl_{-1}; ch_{-1}; cl_0; ch_0\} = IV \in \{0,1\}^4$) et des valeurs successives des LFSR, par l'algorithme suivant :

1. $z_t = a_t \oplus b_t \oplus c_t \oplus d_t \oplus cl_t$;
2. $s_{t+1} = \lfloor \frac{a_t + b_t + c_t + d_t + 2ch_t + cl_t}{2} \rfloor \in [0, 3]$, codé sur deux bits (sl_{t+1}, sh_{t+1}) ;
3. $cl_{t+1} = sl_{t+1} \oplus cl_t \oplus cl_{t-1} \oplus ch_{t-1}$;
4. $ch_{t+1} = sh_{t+1} \oplus ch_t \oplus cl_{t-1}$.

Générateurs cryptographiquement sûrs

On peut utiliser le principe de la fonction à sens unique, c'est-à-dire une fonction facile à calculer, mais difficile à inverser (temps de calcul trop important) pour décider de la qualité d'un générateur.
Si, pour un générateur donné, et étant donnée une séquence produite, il est possible, sans connaître la méthode, de prédire avec une bonne probabilité (significativement supérieure à la probabilité de la distribution attendue), et en un temps de calcul raisonnable, le nombre suivant dans la séquence, alors il ne peut pas être considéré comme aléatoire.
Si on peut prouver qu'il n'existe aucun algorithme efficace qui permettra cette prédiction, alors le générateur est dit *cryptographique*, ou *cryptographiquement sûr*.
Par exemple, le générateur dit de Blum et Micali fonctionne de la façon suivante :
- Générer un grand nombre premier p.
- Soit α une racine primitive de p (un générateur du groupe des inversibles de $\mathbb{F}_p$).
- Soit f la fonction d'exponentiation modulaire $f(x) = \alpha^x \mod p$.
- Soit B la fonction à valeurs dans $\{0, 1\}$:
  - $B(x) = 1$ si $0 \leqslant log_\alpha x \leqslant (p-1)/2$;
  - $B(x) = 0$ si $log_\alpha x > (p-1)/2$.

La séquence de bits pseudo-aléatoire $b_1 \dots b_k$ est alors calculée suivant la séquence $x_0 x_1 \dots x_k$, où x_0 est un élément non nul quelconque de $\mathbb{F}_p$, $x_i \leftarrow f(x_{i-1})$ pour $i > 0$. On a alors $b_i = B(x_i)$.
Cette fonction est facile à calculer : on a vu dans les paragraphes précédents comment générer un nombre premier, trouver une racine primitive et calculer une exponentiation modulaire. Enfin, au moment de calculer B, on dispose de la valeur de $log_\alpha x$, car on vient de calculer $f(x) = \alpha^x \mod p$. Par contre, en ne disposant pas de la suite $x_0 x_1 \dots x_k$, on peut prouver que trouver $B(x_i)$ est aussi difficile que de calculer le logarithme discret. C'est donc un problème difficile, car on ne connaît aucun algorithme efficace pour le logarithme discret, et le générateur est cryptographiquement sûr.

Quelques tests statistiques

Les méthodes précédentes étaient fondées sur la réputation de difficulté algorithmique de certains problèmes, qui rendaient impossible la prédiction du comportement des générateurs. Pour mesurer la qualité d'un générateur, on peut aussi examiner les séquences produites, et tester si elles s'écartent de ce qu'on attend d'un générateur vraiment aléatoire. C'est une tâche difficile, car les critères sont nombreux et pas forcément triviaux.

Les statistiques nous fournissent ici l'outil adéquat pour ces tests. Par exemple, le test du χ^2 (khi-deux) permet de mesurer la déviance par rapport à une loi discrète uniforme attendue.
Pour tout caractère v_i de l'alphabet V, on a une probabilité attendue p_i, et un effectif observé e_i sur la séquence générée, de taille n. Les fréquences attendues ne seront jamais exactement égales aux fréquences observées, il faut donc décider à partir de quel degré de divergence le générateur n'est plus considéré comme aléatoire.
Il faut garder à l'esprit qu'avec un générateur aléatoire, toute séquence est *a priori* possible, même celles dont la distribution est complètement différente de celle attendue, puisque précisément le générateur est aléatoire. Ces séquences seront seulement très peu probable. Si on les obtient, on considérera donc comme très peu probable qu'elles aient été générées par un bon générateur (même si ce n'est pas impossible). Voici comment fonctionne le test du χ^2.
On mesure l'écart entre la distribution attendue et observée avec le paramètre :

$$K = \sum_{i=1}^{n} \frac{(e_i - np_i)^2}{np_i}$$

Il reste à décider quelles sont les valeurs acceptables pour le paramètre K. Elles sont données par des tables dites du χ^2, dont nous donnons un extrait ici dans le tableau 1.4.

	$p = 0,75$	$p = 0,95$	$p = 0,99$
9	11,39	16,92	21,67
10	12,55	18,31	23,21
11	13,70	19,68	24,72
12	14,85	21,03	26,22
15	18,25	25,00	30,58
20	23,83	31,41	37,57
30	34,80	43,77	50,89

TAB. 1.4: Extrait de table du χ^2.

Dans ce tableau, la première colonne donne le nombre de « degrés de liberté ». On choisit ce nombre égal à $|V| - 1$. Soit, pour un alphabet de taille 21, la ligne notée 20. La première ligne donne la probabilité que la valeur K soit inférieure à la valeur du tableau. Par exemple, la probabilité que K dépasse 24,72 pour un alphabet de taille 12 est 0,01.

Exercice 1.22 (Test du χ^2). *Un générateur pseudo-aléatoire censé générer des nombres entre 0 et 10 selon une loi uniforme donne la séquence : 0 0 5 2*

3 6 4 2 0 2 3 8 9 5 1 2 2 3 4 1 2. Faites le test du χ^2 sur cette séquence. Que pensez-vous de ce générateur ? Que pensez-vous ce ce test ? Solution page 285.

Il est évident qu'un test de ce type, s'il est utile, et s'il peut être suffisant pour rejeter un générateur, n'est pas suffisant pour l'accepter. Par exemple, il ne permettra pas de rejeter la séquence 1234561234561234 56, alors qu'un œil même peu exercé y verra la régularité (bien qu'il faille se méfier de l'impression de régularité qu'on peut soi-même avoir en regardant une séquence, souvent largement biaisée par de fausses intuitions sur ce qu'est vraiment l'aléatoire). On peut par exemple renforcer ce test en l'appliquant aux extensions de la source induite par le message. Il existe de nombreux tests statistiques qui renforcent la confiance qu'on pourra avoir en un générateur. Il est important de noter que chaque test permet de rejeter un générateur, mais seul l'ensemble de tous les tests permettra de l'accepter, et encore, sans rigueur mathématique. Il n'est pas garanti qu'ayant passé x tests, le générateur ne se révèle fragile au $x + 1^{\text{ième}}$.

1.4 Décoder, déchiffrer, attaquer

Pour terminer ce chapitre d'introduction, nous allons développer des méthodes de codage en adoptant le point de vue de l'opération inverse, le décodage. Nous avons déjà vu à plusieurs reprises que le décodage n'est pas une entreprise triviale consistant à inverser les fonctions de codage, et ce pour plusieurs raisons :

- si, comme dans le codage fax détaillé en début de chapitre, le message codé est une suite de bits, reconstituer sans ambiguïté le message source en redécoupant la suite selon les blocs dont elle est constituée requiert une forme particulière du code ;
- le décodage exact n'est parfois même pas totalement possible, si le codage n'inclut pas toute l'information de départ, lors d'une compression par exemple ; on a vu que l'image fax perdait en qualité ; il existe de nombreuses méthodes de codage « avec pertes », qui rendent codage et décodage dissymétriques ;
- on a vu que le principe des fonctions à sens unique rend le calcul de la fonction de codage et celui de la fonction de décodage totalement différents ; il se peut qu'il existe un algorithme efficace pour l'une et pas pour l'autre.

Nous allons développer tous ces points, en employant le mot *décodage* comme un terme général permettant de retrouver la source à partir d'un message codé, *déchiffrement* pour le décodage cryptographique, *cassage* ou *attaque* pour le déchiffrement non autorisé, c'est-à-dire ne disposant pas des informations dont ne jouit que le destinataire.

1.4.1 Décoder sans ambiguïté

La première qualité que doit avoir un code est de pouvoir être décodé. C'est une évidence, mais pas forcément un problème trivial.
Supposons que le code est une fonction bijective, qui transforme le message composé par l'émetteur en message qui est transmis par le canal. Pour un message source $a_1...a_n$, chaîne sur un alphabet source quelconque, et pour un alphabet de code V, appelons f la fonction de codage. On a alors le message codé $c_1...c_n = f(a_1)...f(a_n)$, avec $c_i \in V^+$ pour tout i. Le *code*, en tant qu'ensemble des mots de codes, est alors l'image de la fonction de codage f. Le fait que f soit bijective ne suffit cependant pas pour que le message puisse être décodé sans ambiguïté par le récepteur.
Prenons l'exemple du codage des lettres de l'alphabet $S = \{A, \ldots, Z\}$ par les entiers $C = \{0, \ldots, 25\}$ écrits en base 10 :

$$f(A) = 0, f(B) = 1, \ldots, f(J) = 9, f(K) = 10, f(L) = 11, \ldots, f(Z) = 25\,.$$

Le mot de code 1209 peut alors correspondre à différents messages : par exemple, BUJ ou MAJ ou BCAJ.
Il est donc nécessaire d'ajouter des contraintes sur le code pour qu'un message quelconque puisse être déchiffré sans ambiguïté. Un code C sur un alphabet V est dit *non ambigu* (on dit parfois *uniquement déchiffrable*) si, pour tout $x = x_1 \ldots x_n \in V^+$, il existe au plus une séquence $c = c_1 \ldots c_m \in C^+$ telle que

$$c_1 \ldots c_m = x_1 \ldots x_n\,.$$

La propriété suivante est une simple reformulation :

Propriété 7. *Un code C sur un alphabet V est non ambigu si et seulement si pour toutes séquences $c = c_1 \ldots c_n$ et $d = d_1 \ldots d_m$ de C^+ :*

$$c = d \Longrightarrow (n = m \text{ et } c_i = d_i \text{ pour tout } i = 1, \ldots, n)\,.$$

Exemple : Sur l'alphabet $V = \{0, 1\}$,
– le code $C = \{0, 01, 001\}$ n'est pas uniquement déchiffrable.
– le code $C = \{01, 10\}$ est uniquement déchiffrable.
– le code $C = \{0, 10, 110\}$ est uniquement déchiffrable.
La contrainte de déchiffrabilité implique une longueur minimale aux mots de code. Le théorème de Kraft donne une condition nécessaire et suffisante sur la longueur des mots de code pour assurer l'existence d'un code uniquement déchiffrable.

Théorème 10 (Kraft). *Il existe un code uniquement déchiffrable sur un alphabet V dont les n mots sont de longueur $l_1, \ldots, l_n$ si et seulement si*

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leqslant 1 \,.$$

Preuve. ($\Rightarrow$) Soit C un code uniquement déchiffrable, d'arité q (le vocabulaire du code comporte q caractères). Soit m la longueur du plus long mot de C. Pour $1 \leqslant k \leqslant m$, soit r_k le nombre de mots de longueur k. Nous développons l'expression suivante, pour un entier quelconque u, avec $u \geqslant 1$:

$$\left(\sum_{i=1}^{n} \frac{1}{q^{l_i}} \right)^u = \left(\sum_{k=1}^{m} \frac{r_k}{q^k} \right)^u \,.$$

Une fois développée, chaque terme de cette somme est de la forme $\frac{r_{i_1} \ldots r_{i_u}}{q^{i_1+\ldots+i_u}}$. Et en regroupant pour chaque valeur $s = i_1 + \ldots + i_u$, on obtient les termes $\sum_{i_1+\ldots+i_u=s} \frac{r_{i_1} \ldots r_{i_u}}{q^s}$. Soit $N(s) = \sum_{i_1+\ldots+i_u=s} r_{i_1} \ldots r_{i_u}$. L'expression initiale s'écrit : $(\sum_{i=1}^{n} \frac{1}{q^{l_i}})^u = \sum_{s=u}^{mu} \frac{N(s)}{q^s}$, N(s) est le nombre de combinaisons de mots de C de longueur totale s. Comme C est uniquement déchiffrable, deux combinaisons de mots de C ne peuvent être égales au même mot sur l'alphabet de C. Comme C est d'arité q, et que $N(s)$ est inférieur au nombre total de messages de longueur s sur cet alphabet, alors $N(s) \leqslant q^s$. Cela donne

$$\left(\sum_{i=1}^{n} \frac{1}{q^{l_i}} \right)^u \leqslant mu - u + 1 \leqslant mu \,.$$

Et donc $\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leqslant (mu)^{1/u}$, puis $\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leqslant 1$ quand u tend vers l'infini.
($\Leftarrow$) La réciproque est une conséquence du théorème de McMillan, que nous verrons plus loin dans ce chapitre. □

Propriété du préfixe

On dit qu'un code C sur un alphabet V a la *propriété du préfixe* (on dit parfois qu'il est *instantané* , ou *irréductible*) si et seulement si pour tout couple de mots de code distincts (c_1, c_2), c_2 n'est pas un préfixe de c_1.

Exemple : $a = 101000$, $b = 01$, $c = 1010$: b n'est pas un préfixe de a mais c est un préfixe de a.

Grâce à la propriété du préfixe, on peut déchiffrer les mots d'un tel code dès la fin de la réception du mot (instantanéité), ce qui n'est pas toujours le cas

pour les codes uniquement déchiffrables : par exemple, si $V = 0, 01, 11$, et si on reçoit le message $m = 001111111111\ldots$, il faut attendre l'occurrence suivante d'un 0 pour pouvoir déchiffrer le second mot (0 ou 01 ?).

Propriété 8. *Tout code possédant la propriété du préfixe est uniquement déchiffrable.*

Preuve. Soit un code C sur V qui n'est pas uniquement déchiffrable. Alors il existe une chaîne $a \in V^n$ telle que $a = c_1 \ldots c_l = d_1 \ldots d_k$, les c_i et d_i étant des mots de C et $c_i \neq d_i$ pour au moins un i. Choisissons le plus petit i tel que $c_i \neq d_i$ (pour tout $j < i$, $c_j = d_j$). Alors $l(c_i) \neq l(d_i)$, sinon, vu le choix de i, on aurait $c_i = d_i$, ce qui est en contradiction avec la définition de i. Si $l(c_i) < l(d_i)$, c_i est un préfixe de d_i et dans le cas contraire d_i est un préfixe de c_i. C n'a donc pas la propriété du préfixe. □ La réciproque est fausse : le code $C = \{0, 01\}$ est uniquement déchiffrable, mais ne possède pas la propriété du préfixe. La propriété suivante est évidente, mais assure la déchiffrabilité d'une catégorie de code très usitée.

Propriété 9. *Tout code dont tous les mots sont de même longueur possède la propriété du préfixe.*

Exercice 1.23. *Soit S la source d'alphabet $\{a, b, c, d\}$ et de probabilités :*

U	$s_1 = a$	$s_2 = b$	$s_3 = c$	$s_4 = d$
$P(U)$	0.5	0.25	0.125	0.125

On code S avec le code suivant :

a	b	c	d
0	10	110	111

1. *Coder adbccab. Décoder 1001101010.*
2. *Est-ce un code instantané ?*
3. *Calculer l'entropie de la source.*

Solution page 285.

Exercice 1.24. *On veut construire un code compresseur binaire sur une source $\mathcal{S} = (S, \mathcal{P})$ (supposée infinie) où $S = (0, 1)$ est l'alphabet source et $\mathcal{P} = (P(0) = p, P(1) = 1 - p)$ est la loi de probabilité de $\mathcal{S}$.*
On propose le code suivant : on compte le nombre d'occurrences de "0" avant l'apparition d'un "1". Les deux règles de codage sont :
- *Une chaîne de quatre "0" consécutifs (sans "1") est codée par 0.*
- *Si moins de quatre "0" apparaissent avant un symbole "1", on code la chaîne par le mot de code "1e_1e_2", où e_1e_2 est la représentation binaire du nombre de "0" apparus avant le "1".*

Par exemple, l'apparition de quatre zéros consécutifs "0000" est codée par "0", tandis que l'occurrence "001" est codé par "110" car deux "0" sont apparus avant le "1" et que "10" est la représentation binaire de deux.

1. *Expliciter les 5 mots de code. Le code possède-t-il la propriété du préfixe ?*
2. *Sachant que la probabilité d'apparition de deux symboles successifs s_1 et s_2 est, dans notre cas où l'on suppose la source sans mémoire, $p(s_1) * p(s_2)$, calculer la probabilité d'occurrence dans $\mathcal{S}$ d'une chaîne de k "0" suivis d'un "1".*
3. *Pour chaque mot du code, calculer le nombre de bits de code nécessaires par bit de source. En déduire le* taux de compression *de ce code, c'est à dire la* longueur moyenne par bit de source.

Solution page 285.

L'arbre de Huffman

L'arbre de Huffman est un objet permettant de représenter facilement tous les codes qui ont la propriété du préfixe, et cette représentation facilite grandement leur manipulation. On donne ici les définitions dans le cas binaire, mais elles sont généralisables pour des codes d'arité quelconque.
On appelle *arbre de Huffman* un arbre binaire tel que tout sous-arbre a soit 0 soit 2 fils (il est localement complet). Dans ce dernier cas, on assigne le symbole "1" à l'arête reliant la racine locale au fils gauche et "0" au fils droit.

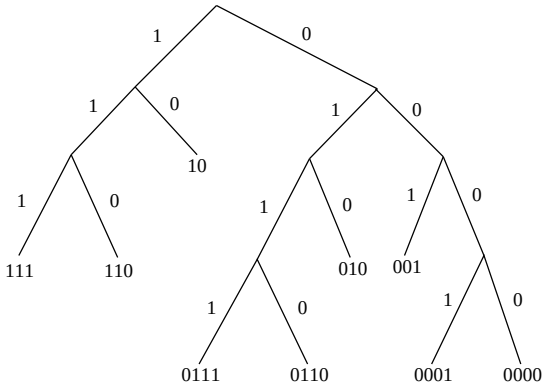


FIG. 1.10: Exemple d'arbre de Huffman.

À chaque feuille d'un arbre de Huffman, on peut associer un mot de $\{0,1\}^+$: c'est la chaîne des symboles marquant les arêtes d'un chemin depuis la racine jusqu'à la feuille.

Le maximum sur l'ensemble des longueurs des mots d'un arbre de Huffman est appelé *hauteur* de cet arbre. On appelle *code de Huffman* l'ensemble des mots correspondant aux chemins d'un arbre de Huffman ; la hauteur de cet arbre est appelée aussi *hauteur* du code C.

Exemple : le code correspondant à l'arbre de la figure 1.10 est :

$$\{111, 110, 10, 0111, 0110, 010, 001, 0001, 0000\}.$$

Représentation des codes instantanés

L'introduction des arbres de Huffman est justifiée par les deux propriétés suivantes, qui assurent que tous les codes instantanés pourront être manipulés avec de tels arbres.

Propriété 10. *Un code de Huffman possède la propriété du préfixe.*

Preuve. Si un mot de code c_1 est un préfixe de c_2, alors le chemin représentant c_1 dans l'arbre de Huffman est inclus dans le chemin représentant c_2. Comme c_1 et c_2 sont, par définition, associés à des feuilles de l'arbre, $c_1 = c_2$. Il n'existe donc pas deux mots différents dont l'un est le préfixe de l'autre, et le code de Huffman a la propriété du préfixe. □

Propriété 11. *Tout code qui possède la propriété du préfixe est contenu dans un code de Huffman.*

Preuve. Soit un arbre de Huffman complet (toutes les feuilles sont à distance constante de la racine) de hauteur l, la longueur du plus long mot de C. Chaque mot c_i de C est associé à un chemin depuis la racine jusqu'à un nœud. On peut alors élaguer le sous-arbre dont ce nœud est racine (tous les mots pouvant être représentés dans les nœuds de ce sous-arbre ont c_i pour préfixe). Tous les mots de C sont toujours dans les nœuds de l'arbre résultant. On peut effectuer la même opération pour tous les mots. On a finalement un arbre de Huffman contenant tous les mots de C. □

Théorème de McMillan

On a vu que les arbres de Huffman permettaient de représenter tous les codes instantanés. Mais ils ne permettent pas de représenter tous les codes uniquement déchiffrables. Le théorème de McMillan certifie que les codes uniquement déchiffrables (non ambigus) n'ayant pas la propriété du préfixe ne sont pas indispensables. En effet, il existe toujours un autre code qui a la propriété du préfixe avec les mêmes longueurs de mots. Ne pas considérer les codes ambigus ne fait donc rien perdre en compression.

Théorème 11 (McMillan). *Sur un alphabet V, il existe un code qui possède la propriété du préfixe dont les mots $\{c_1, \ldots, c_n\}$ sont de longueur $l_1, \ldots, l_n$ si et seulement si*

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leqslant 1\,.$$

Exercice 1.25. *À l'aide de la représentation des codes instantanés par des arbres de Huffman, donner une preuve du théorème de McMillan.*

Solution page 285.

Corollaire 1. *S'il existe un code uniquement déchiffrable dont les mots sont de longueur $l_1, \ldots, l_n$, alors il existe un code instantané de mêmes longueurs de mots.*

C'est une conséquence des théorèmes de Kraft et McMillan. Les codes déchiffrables qui ne possèdent pas la propriété du préfixe ne produisent pas de code aux mots plus courts que les codes instantanés, auxquels on peut donc se restreindre pour la compression d'information (leurs propriétés les rendent plus maniables).

1.4.2 Les codes non injectifs

Tous les codes n'assurent pas ainsi le décodage non ambigu, ni même bijectif. Il peut arriver, pour des raisons diverses, que des fonctions de codage ne codent qu'une *empreinte* d'un message, ou alors seulement l'information jugée suffisante aux besoins de la transmission. Les empreintes sont par exemple utilisées pour la détection d'erreur : en envoyant un message et son empreinte, le destinataire peut vérifier en recalculant l'empreinte que l'empreinte reçue est bien l'empreinte du message reçu, et ainsi vérifier que la transmission n'a pas altéré le tout.
La compression avec perte est utilisée par exemple pour les images ou les sons. Ne sont codées que les informations qui permettront de reconstituer non pas le message entier, mais l'image ou le son avec des différences qui ne seront pas perceptibles.

L'empreinte de vérification

Le principe le plus simple de la détection d'erreur est un exemple de calcul d'empreinte. On a vu un exemple de codage de ce type pour le code du fax, bien que le code ajouté à chaque ligne ne dépendait pas du contenu de la ligne. Cela n'assurait d'ailleurs qu'une efficacité de détection très limitée.
Le premier principe d'ajout d'une empreinte pour la détection d'erreurs est l'ajout d'un simple *bit de parité* aux blocs codés. Pour le mot $m = s_1 \ldots s_k$,

le bit de parité vaut $b(m) = (\sum_{i=1}^{k} s_i) \mod 2$. Il est évident que cette dernière égalité devient fausse lorsque des bits en nombre impair dans l'ensemble « message+empreinte » changent de valeurs. Ainsi, l'ajout d'un bit de parité permet de détecter une erreur portant sur un nombre impair de bits. Nous verrons plus en détails ce mécanisme dans le chapitre 4 en particulier sur la figure 4.2 page 220.

Les fonctions de hachage

La fonction de hachage suit le même principe, mais code une empreinte plus évoluée, car elle est censée identifier le message. C'est la définition d'un résumé du message qui permettra, non de le reconstituer, mais de l'identifier si on dispose au préalable d'une table de correspondance. Cela fonctionne comme une empreinte digitale qui ne permet pas de reconstituer d'autres caractéristiques d'un individu, mais qui permet de l'identifier.
Les fonctions de hachage sont particulièrement utiles en cryptographie. Elles permettent notamment de diminuer la quantité d'information à crypter. Si l'image de x par la fonction de hachage est appelée l'*empreinte* de x, on peut par exemple ne crypter que l'empreinte. D'autre part, elles permettent de mettre au point des protocoles de signature électronique et d'authentification des messages (voir la section 3.5.2), ainsi que de vérifier l'intégrité d'un document, au même titre que le bit de parité, qui lui-même est une fonction de hachage particulière.

Formellement, une *fonction de hachage* $H : \{0,1\}^* \longrightarrow \{0,1\}^n$ est une application qui transforme une chaîne de taille quelconque en une chaîne de taille fixe n, comme illustré sur la figure 1.11.

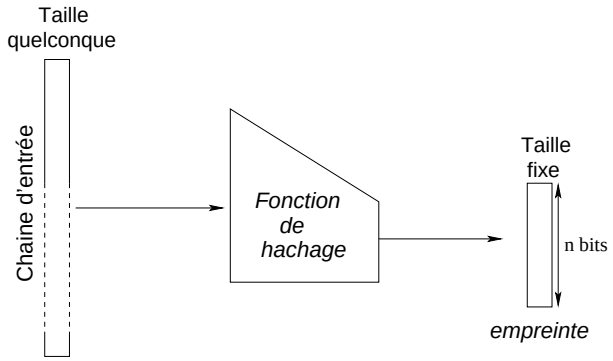


FIG. 1.11: Principe d'une fonction de hachage.

On parle de *collision* entre x et x' lorsque

$$\begin{cases} x \neq x' \\ H(x) = H(x') \end{cases}$$

Dans la mesure ou l'entrée d'une fonction de hachage peut avoir une taille quelconque (en particulier $> n$), les collisions sont inévitables. Si y est tel que $y = H(x)$, alors x est appelée *antécédent* ou *préimage* de y (on rappelle que y est l'*empreinte* de x).
Une des contraintes de base dans l'élaboration d'une fonction de hachage est l'efficacité : une empreinte doit être facile à calculer. En outre, les fonctions de hachage ont une propriété naturelle de compression.
D'autres propriétés peuvent être souhaitées :
- **Résistance à la préimage** : étant donné y, on ne peut pas trouver en temps raisonnable un x tel que $y = H(x)$.
- **Résistance à la seconde préimage** : étant donné x, on ne peut pas trouver en temps raisonnable $x' \neq x$ tel que $H(x) = H(x')$;
- **Résistance aux collisions** : on ne peut pas trouver en temps raisonnable x et x' tels que $H(x) = H(x')$;

Une fonction de hachage *à sens unique* est une fonction de hachage qui vérifie les propriétés de résistance à la préimage et à la seconde préimage.

Exercice 1.26 (Sécurité des fonctions de hachage). *Montrer, par contraposée, que la résistance aux collisions implique la résistance à la seconde préimage qui implique la résistance à la préimage.* *Solution page 286.*

Exercice 1.27 (Une mauvaise fonction de hachage). *Soit $f : \mathbb{F}_2^m \to \mathbb{F}_2^m$ une fonction quelconque. On propose comme fonction de hachage à itérer $g : \mathbb{F}_2^{2m} \to \mathbb{F}_2^m$, la fonction telle que pour un x de $2m$ bits, découpé en deux blocs x_h et x_l, on ait $g(x) = g(x_h||x_l) = f(x_h \oplus x_l)$. Montrer que g n'est pas résistante à la seconde préimage.* *Solution page 287.*

Les fonctions de hachage peuvent être utilisées pour :
- les codes de détection de manipulation (MDC pour *Manipulation Detection Code*) qui permettent de s'assurer de l'intégrité d'un message, à l'instar des bits de parité ;
- les codes d'authentification de messages (MAC pour *Message Authentification Code*) qui gèrent à la fois l'intégrité et l'authentification de la source d'une donnée.

Nous verrons plusieurs exemples de telles utilisations au chapitre 3.

Construction de fonction de hachage de Merkle-Damgård. L'une des constructions les plus connues de fonction de hachage fait intervenir une fonction de compression

$$h : \{0,1\}^b \times \{0,1\}^n \longrightarrow \{0,1\}^n$$

Une telle fonction est illustrée dans la figure 1.12.

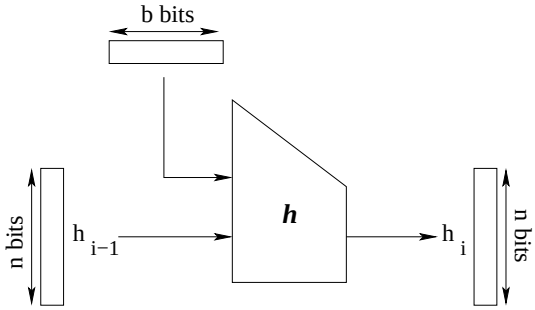


FIG. 1.12: Fonction de compression d'une fonction de hachage.

Le message M est découpé en blocs de b bits $M_1, \ldots, M_k$ (il faut éventuellement rajouter des bits fictifs afin que la taille de M divise exactement b).
On itère la fonction de compression h selon le modèle présenté dans la figure 1.13.

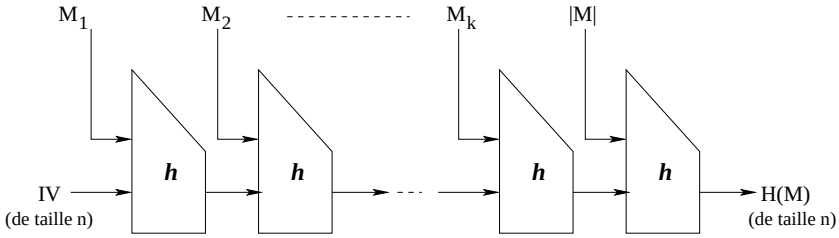


FIG. 1.13: Construction de Merkle-Damgård.

IV (Initial Value) est une chaîne (de taille n) fixée par l'algorithme ou l'implémentation. Un théorème prouvé indépendamment par Ralph Merkle et Ivan Damgård permet de circonscrire les propriétés théoriques d'une telle construction :

Théorème 12 (Merkle-Damgård). *Si h est résistante aux collisions, alors H l'est aussi.*

C'est ce résultat qui fait de Merkle-Damgård la construction la plus utilisée dans les calculs d'empreintes.

Exercice 1.28. *Prouver le théorème de Merkle-Damgård par la contraposée.*
Solution page 287.

Exercice 1.29 (Construction par composition). *Soit $f : \mathbb{F}_2^{2m} \to \mathbb{F}_2^m$ une fonction de hachage. Soit maintenant une deuxième fonction de hachage définie par $h : \mathbb{F}_2^{4m} \to \mathbb{F}_2^m$ telle que pour $x_1, x_2 \in \mathbb{F}_2^{2m}$, alors $h(x_1||x_2) = f(f(x_1)||f(x_2))$, où $||$ désigne l'opération de concaténation.*

1. *Montrez que si f est résistante aux collisions alors h l'est également.*
2. *Quel est l'inconvénient de cette construction ?*

Solution page 287.

Il reste à expliciter la construction des fonctions de compression h résistantes aux collisions. Par exemple, la construction de Davies-Meyer (voir figure 1.14) définit $h_i = E_{M_i}(h_{i-1}) \oplus h_{i-1}$, E_{M_i} étant une fonction de cryptographie à clef secrète par blocs.

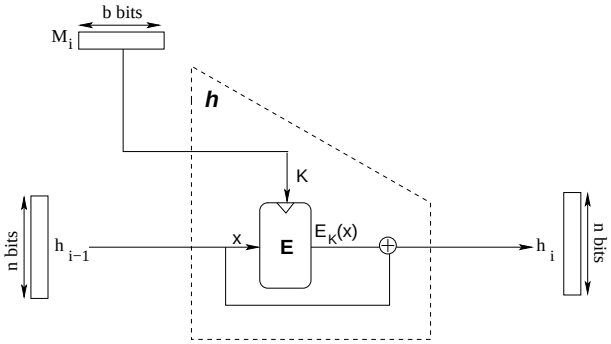


FIG. 1.14: Construction de Davies-Meyer.

Mais une attaque de la résistance à la préimage fut conçue en 1999 par Drew Dean qui exploitait la définition de point fixe sur cette construction. Aussi les fonctions de compression utilisant cette construction sont moins robustes.

La construction de Miyaguchi-Preneel (voir figure 1.15) est une amélioration de la construction précédente, particulièrement robuste d'un point de vue cryptographique.
La fonction g est une fonction d'adaptation à la taille de clef de la fonction de chiffrement E. On a ainsi $h_i = E_{g(h_{i-1})}(M_i) \oplus h_{i-1} \oplus M_i$.

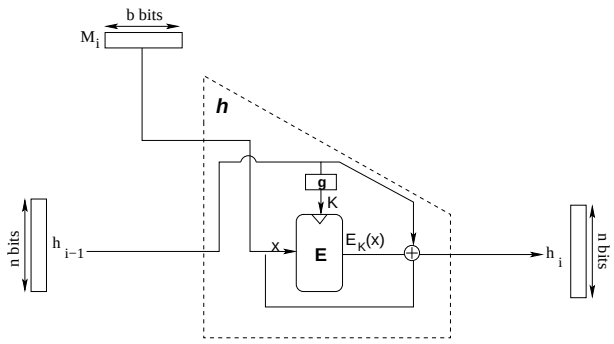


FIG. 1.15: Construction de Miyaguchi-Preneel.

Transformations avec perte

D'autres procédés de transformation aboutiront à un message codé avec perte, et un décodage intégral impossible. Par exemple, le fax n'est qu'une mauvaise copie de l'original, mais on estime qu'elle remplit son devoir de faire passer l'information contenue dans la feuille. On peut aussi coder une image ou un son sans en garder toutes les informations, pourvu que la perte ne soit pas visible à l'œil ou à l'oreille.
Les informations numériques sont par essence des informations discrètes. On a vu que des données continues ou analogiques, comme par exemple les sons ou les images, pouvaient facilement être numérisées. Pour les sons, on peut coder à chaque instant une fréquence et une amplitude. Pour les images, on les décompose en pixels et on peut coder une couleur pour chaque pixel.
Il se peut pourtant que cette numérisation naturelle ne soit pas le meilleur modèle pour bien des codes. Souvent, pour les images par exemple les couleurs de deux pixels voisins ne sont pas indépendantes, et il sera plus judicieux de coder un ensemble de pixels comme une fonction que chacun indépendamment comme une valeur. On code alors par blocs de pixels, par une fonction périodique ou presque périodique.
Il arrive donc que le codage agisse sur des fonctions plutôt que sur des entités discrètes ou numériques, et c'est le principe de cette section. Le codage sera alors une fonction particulière, qui s'applique elle-même sur des fonctions, et qu'on appellera alors plutôt *transformée*, ou *transformation*.

Transformée de Fourier et transformée discrète

Supposons qu'un message, ou une partie de ce message, puisse s'exprimer comme une fonction périodique h, variant avec le temps noté t, de période

2π. C'est le cas par exemple des sons. Puisqu'on fait l'hypothèse que h est périodique, le même message peut s'exprimer comme une amplitude H, variant en fonction d'une fréquence f.
La transformée de Fourier est un procédé de codage permettant de passer d'une représentation à une autre. Comme tout procédé de codage, il est associé à son inverse, le décodage, et est exprimé par les formules suivantes.

Transformée de Fourier

$$\text{Codage : } H(f) = \int_{-\infty}^{\infty} h(t)e^{2\pi ift}dt$$

$$\text{Décodage : } h(t) = \int_{-\infty}^{\infty} H(f)e^{2\pi ift}dt$$

Pour un son, si le codage naturel et immédiat est plutôt $h(t)$, souvent, $H(f)$, qui code exactement la même information puisque le codage est réversible, est beaucoup plus économique, puisqu'il tire parti de la périodicité de h. La transformée est donc directement efficace pour la compression.
La transformée de Fourier dite discrète suit le même principe, mais avec des fonctions discrètes. Elle nous sera évidemment très utile, puisque par essence, nos messages numériques doivent être codés par des informations discrètes. Supposons maintenant que nos fonctions $h(t)$ et $H(f)$ soient des fonctions discrètes, c'est-à-dire des vecteurs $h_0, \ldots, h_{n-1}$ et $H_0 \ldots H_{n-1}$ de variables discrètes. On exprime alors ainsi la transformation en notant $\omega = e^{-\frac{2i\pi}{n}}$ une racine $n^{\text{ième}}$ de l'unité. ω vérifie par exemple $\sum_{k=0}^{n-1} \omega^k = \sum_{k=0}^{n-1} \left(e^{-\frac{2i\pi}{n}}\right)^k = \frac{e^{-\frac{2in\pi}{n}}-1}{e^{-\frac{2i\pi}{n}}-1} = 0$.

Transformée de Fourier Discrète (DFT)

$$\text{Codage : } H_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} h_j \omega^{kj}$$

$$\text{Décodage : } h_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} H_k \omega^{-kj}$$

ou encore, si $h(X) = \sum_{j=0}^{n-1} h_j X^j$ alors

$$DFT(h) = [H_0, \ldots, H_{n-1}] = \frac{1}{\sqrt{n}}[h(\omega^0), \ldots, h(\omega^{n-1})] \tag{1.9}$$

Le décodage est correct car

$$h_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} h_i \omega^{ki} \omega^{-kj} = \frac{1}{n} \sum_{i=0}^{n-1} h_i \sum_{k=0}^{n-1} (\omega^{i-j})^k = \frac{1}{n} \sum_{i=0}^{n-1} h_i \times \begin{cases} 0 & \text{si } i \neq j \\ n & \text{sinon} \end{cases}$$

La transformée discrète en cosinus (DCT) dérive directement de la transformée de Fourier discrète, pour une fonction h discrète, mais au lieu d'être une fonction du temps (ce qui est par exemple un bon modèle pour un son), est une fonction de l'espace (ce qui permet de coder une image). c'est donc une fonction à deux variable discrètes $h(x, y)$. C'est par exemple la couleur d'un pixel aux coordonnées x et y. De la même façon, il est possible de représenter la même information différemment, par une fonction H de deux variable discrètes i, j, représentant une analyse *spectrale* de l'image.

Transformée en Cosinus Discrète (DCT)

$$\text{Codage} : H(i,j) = \frac{1}{n} \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} h(x,y) \cos(\frac{(2x+1)i\pi}{2n}) \cos(\frac{(2y+1)j\pi}{2n})$$

$$\text{Décodage} : h(x,y) = \frac{1}{n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} H(i,j) \cos(\frac{(2x+1)i\pi}{2n}) \cos(\frac{(2y+1)j\pi}{2n})$$

La DCT est également un bon principe de compression pour les images, ainsi que pour tout signal périodique, ou quasi-périodique.
Toutes ces transformations sont réversibles. Elles peuvent déjà se révéler de bons procédés de compression en elles-mêmes, mais leur intérêt est aussi dans la facilité à choisir des informations importantes et à ne garder que celles-là. En effet, il est possible lors d'un tel codage de ne garder que certains coefficients de la DFT ou DCT, ce qui réduit la taille de l'information à coder, et ne change pas forcément le résultat audiovisuel obtenu. Nous allons détailler ces principes dans le chapitre 2.

Algorithme de la DFT

On peut écrire la transformation DFT sous forme matricielle

$$H = \frac{1}{\sqrt{n}} \Omega h$$

où $\Omega_{k,j} = \omega^{kj}$. La transformation inverse s'écrit alors $(\Omega^{-1})_{k,j} = \frac{1}{\sqrt{n}} \omega^{-kj}$.
Un algorithme immédiat pour le calcul de la transformée utilise le produit matrice-vecteur et a donc une complexité $O(n^2)$.

Un algorithme de type « diviser pour régner » diminue la complexité, ce qui sera extrêmement important pour le codage. Le principe de « diviser pour régner » est de couper le problème en sous-problèmes équivalents de taille inférieure. Ici, on sépare l'expression de la transformée en deux parties. En supposant que n est pair, $m = \frac{n}{2}$

$$\begin{aligned} H_k &= \sum_{j=0}^{m-1} h_j \omega^{kj} + \sum_{j=m}^{n-1} h_j \omega^{kj} \\ &= \sum_{j=0}^{m-1} h_j \omega^{kj} + \sum_{j=0}^{n-m+1} h_{j+m} \omega^{k(j+m)} \\ &= \sum_{j=0}^{m-1} h_j \omega^{kj} + \sum_{j=0}^{m-1} h_{j+m} \omega^{kj} \omega^{km} \\ &= \sum_{j=0}^{m-1} (h_j + \omega^{km} h_{j+m}) \omega^{kj} \end{aligned}$$

Or, comme ω est racine $n^{\text{ième}}$ de l'unité, ω^{km} vaut 1 ou -1 selon la parité de k.
- Si k est pair, alors on définit le vecteur

$$\tilde{h}^p = (h_0 + h_m, \ldots, h_{m-1} + h_{n-1})$$

et les coefficients pairs de la transformée H de h sont les coefficients de la transformée $\tilde{H}^p$ de $\tilde{h}^p$, qui a une taille de la moitié de h.
- De même, si k est impair, on définit le vecteur

$$\tilde{h}^i = (h_0 - h_m, \ldots, h_{m-1} - h_{n-1})$$

et les coefficients impairs de la transformée H de h sont les coefficients de la transformée $\tilde{H}^i$ de $\tilde{h}^i$, qui a une taille de la moitié de h.
On obtient l'algorithme 11. La complexité de cet algorithme est $C(n) = 2C(n/2) + n$, soit $C(n) = O(n \log n)$. C'est une complexité quasiment linéaire, et donc un gain important par rapport à l'algorithme immédiat.

DFT et racines $n^{\text{ièmes}}$ de l'unité dans un corps fini

On considère dans un corps $\mathbb{F}_q$ le polynôme $X^n - 1$, avec $n < q$. Une *racine $n^{ième}$ de l'unité* dans $\mathbb{F}_q$ est une racine simple, s'il y en a, du polynôme $X^n - 1$. L'*ordre* d'une racine de l'unité γ est le plus petit entier o tel que $\gamma^o = 1$.

Algorithme 11 Transformée de Fourier Discrète Rapide.

Entrées Le vecteur h dont la taille est une puissance de 2
Sorties Sa transformée le vecteur H
  Si h est de taille 1 **Alors**
    retourner h
  Sinon
    Calculer les vecteurs $\tilde{h}^i$ et $\tilde{h}^p$.
    Calculer récursivement $\tilde{H}^i$ et $\tilde{H}^p$, les DFT de $\tilde{h}^i$ et $\tilde{h}^p$.
    Les coefficients pairs de H sont les coefficients de $\tilde{H}^p$ et les coefficients impairs de H sont les coefficients de $\tilde{H}^i$.
  Fin Si

Puisque γ est une racine de $X^n - 1$, clairement $o < n$. En outre o divise n, car en posant $n = ob + r$ on obtient $\gamma^r = 1$ et donc $r = 0$.
Une *racine primitive $n^{ième}$ de l'unité* est une racine $n^{\text{ième}}$ de l'unité d'ordre n. C'est cette dernière notion qui est essentielle au fonctionnement de la DFT. Pour le calcul de la DFT dans le corps $\mathbb{C}$, nous avons utilisé une racine $n^{\text{ième}}$ particulière, $e^{-\frac{2i\pi}{n}}$, qui est primitive dans $\mathbb{C}$.
Autant dans $\mathbb{C}$ des racines primitives $n^{\text{ième}}$ sont disponibles pour tout n, autant dans un corps fini on n'est même pas assuré de leur existence. En effet, dans $\mathbb{F}_q$, une racine primitive $(q-1)^{\text{ième}}$ de l'unité est ce que nous avons simplement appelé racine primitive dans la section 1.3.5. De la même façon que pour ces dernières, le théorème suivant nous permet de savoir dans quels corps il est possible d'effectuer des calculs rapides pour des vecteurs de taille n donnée :

Théorème 13. *Soient q une puissance d'un nombre premier et n un entier premier avec q. Le corps fini $\mathbb{F}_q$ contient une racine primitive $n^{ième}$ de l'unité si et seulement si n divise $q - 1$.*

Démonstration. Si a est une racine primitive $n^{\text{ième}}$ alors $a^n = 1$ et n est l'ordre de a. Mais comme a est un élément du corps à q éléments, son ordre divise forcément $q - 1$. Réciproquement, on utilise un générateur g du corps (une racine primitive $(q-1)^{\text{ième}}$) dont on est assuré de l'existence par l'algorithme de la section 1.3.5. Ainsi, si $q - 1 = kn$ alors g^k est une racine primitive $n^{\text{ième}}$ de l'unité. □

On dit qu'un corps *supporte la DFT à l'ordre n* s'il existe des racines primitives $n^{\text{ième}}$ de l'unité dans ce corps. Les corps supportant la DFT pour n des puissances de 2 sont bien entendu particulièrement intéressants pour appliquer l'algorithme rapide diviser pour régner. Par exemple, le corps à 786433 éléments permet de multiplier des polynômes de degré jusqu'à 2^{18} par l'algorithme rapide car $786433 - 1 = 2^{18} \cdot 3$.

Il nous reste à calculer de telles racines primitives $n^{\text{ième}}$ de l'unité. Il est possible d'utiliser un générateur, ou plutôt directement une variante de l'algorithme de la section 1.3.5 : tirer au hasard une racine $n^{\text{ième}}$ (une racine du polynôme $X^n - 1$ dans $\mathbb{F}_q$) et tester si son ordre est bien n. Le corollaire suivant nous donne la probabilité de bien tomber : $\varphi(n)$ chances sur n.

Corollaire 2. *Si un corps fini possède une racine primitive $n^{ième}$ de l'unité alors il en possède exactement $\varphi(n)$.*

Démonstration. On note $q - 1 = kn$. Soit g un générateur du corps. Alors g^k est une racine primitive $n^{\text{ième}}$, de même que tous les g^{tk} pour t entre 1 et $n - 1$ premier avec n. Tous ces g^{tk} sont distincts sans quoi g ne serait pas un générateur ; et ce sont les seuls car g^{tk} avec t non premier avec n est d'ordre strictement plus petit que n. Enfin les racines primitives $n^{\text{ième}}$ s'écrivent forcément sous la forme g^{tk} : si g^u est une racine primitive $n^{\text{ième}}$, alors $g^{un} = 1$ et donc $un = t(q-1)$ car g est un générateur. Cela prouve que u est de la forme tk. □

Exercice 1.30. *Trouver une racine primitive $6^{ième}$ de l'unité modulo 31.*
Solution page 287.

Il est noter qu'il y a une solution pour travailler néanmoins avec des racines primitives $n^{\text{ième}}$ dans un corps qui n'en contient pas. À la façon de $\mathbb{C}$ par rapport à $\mathbb{R}$, on peut toujours se placer dans un corps contenant $\mathbb{F}_q$ et dans lequel le polynôme $X^n - 1$ se factorise complètement en polynômes de degré 1. Ce corps est une *extension* du corps $\mathbb{F}_q$ et est appelé le *corps de scindage* de $X^n - 1$.

Exercice 1.31. *Trouver une racine primitive $4^{ième}$ de l'unité dans un corps de caractéristique 31.*
Solution page 288.

Produit rapide de polynômes par la DFT

De la même façon que pour la transformée de Fourier discrète, le produit de deux polynômes, que nous utilisons fréquemment en théorie des codes (en témoignent toutes les constructions à base de polynômes dans ce chapitre), a un algorithme immédiat de complexité $O(n^2)$.
La transformée de Fourier discrète et l'algorithme de calcul que nous venons de voir permettent de faire ce calcul en temps $O(n \log(n))$.
Étant donnés deux polynômes $P = a_0 + a_1 X + \cdots + a_m X^m$ et $Q = b_0 + b_1 X + \cdots + b_n X^n$, on note $A = DFT(P)$ et $B = DFT(Q)$ les transformées de Fourier discrètes respectives des vecteurs $a = a_0 \dots a_m 0 \dots 0$ et $b = b_0 \dots b_n 0 \dots 0$, où les coefficients des polynômes sont étendus par des zéros jusqu'au degré

$n+m$ (degré du produit). La définition de la transformée nous donne alors immédiatement chacun des coefficients comme $A_k = \sum_{i=0}^{n+m} a_i \omega^{ki} = P(\omega^k)$ et $B_k = Q(\omega_k)$. En multipliant simplement ces deux scalaires on obtient alors, simplement en utilisant l'arithmétique des polynômes, la valeur du produit PQ évalué en ω^k : $C_k = A_k B_k = P(\omega^k)Q(\omega^k) = (PQ)(\omega^k)$; autrement dit $C = DFT(PQ)$.
Cette propriété permet de construire l'algorithme 12 calculant le produit de deux polynômes en temps seulement $O(n \log(n))$.

Algorithme 12 Produit rapide de deux polynômes.

Entrées Deux polynômes $P = a_0 + a_1 X + \cdots + a_m X^m$ et $Q = b_0 + b_1 X + \cdots + b_n X^n$.

Sorties Le polynôme produit PQ.

On étend les polynômes par des zéros jusqu'au degré $m+n$ (degré du produit).

Calculer $DFT(P)$ et $DFT(Q)$ les transformées de Fourier discrètes des vecteurs de coefficients de P et Q.

Par multiplication terme à terme, calculer le vecteur $DFT(P).DFT(Q)$.

Calculer la transformée inverse pour obtenir $PQ = DFT^{-1}(DFT(P).DFT(Q))$.

La complexité est bien $O(n \log n)$, puisque le produit terme à terme est linéaire, et que la transformée, aussi bien que son inverse, sont de complexité $O(n \log n)$.

1.4.3 Cryptanalyse

Nous avons exploré certaines dissymétries entre le codage et le décodage. L'une d'entre elles, la plus importante sans doute en cryptographie, est celle qui différencie le déchiffrement (par le destinataire) du cassage (par un tiers). Nous consacrons un petit développement à des techniques d'attaques qui se basent sur des faiblesses de codes trop rapidement conçus.
Les codes cryptographiques utilisent des générateurs pseudo-aléatoires pour la génération de clef secrète, les fonctions de hachage pour l'authentification, et les fonctions à sens unique pour les techniques à clef publique. Nous exposons séparément des attaques connues de chacune de ces étapes. Connaître les techniques d'attaques est indispensable pour générer des codes qui puissent leur résister.

Attaque des générateurs congruentiels linéaires

Les générateurs congruentiels linéaires ont été abordés en section 1.3.6. Un nombre aléatoire x_i est généré en fonction du nombre précédent généré x_{i-1},

par la formule $x_i = ax_{i-1} + b \mod m$.

Exercice 1.32 (Attaque des générateurs congruentiels linéaires).
- *Si m est premier, quelle est la période maximale d'un générateur congruentiel linéaire ? En particulier, Fishman et Moore ont étudié en 1986 les générateurs modulo $2^{31} - 1 = 2147483647$ et ont déterminé que celui pour $a = 950706376$ est de période maximale et présente de bonnes propriétés statistiques. Que pouvez-vous dire de 950706376 ?*
- *Pour $m = p^e$ avec p premier impair, quelle est la période maximale d'un générateur congruentiel linéaire ?*
  Enfin, il est possible de démontrerque si $\lambda(m)$ est la période maximale, alors $\lambda(2^e) = 2^{e-2}$ pour $e > 2$ et que $\lambda(m) = ppcm(\lambda(p_1^{e_1}) \dots \lambda(p_k^{e_k}))$ si $m = p_1^{e_1} \dots p_k^{e_k}$ avec $p_1 \dots p_k$ premiers distincts.
- *En supposant que m est connu, comment récupérer a et b ?*
- *Maintenant m est inconnu et l'on supposera tout d'abord que $b = 0$. Comment casser le générateur ?* Indication : on pourra étudier $x_{n+1} - x_n$. *Que se passe-t-il si $b \neq 0$?*
- *Quel entier est le suivant dans la liste : 577,114,910,666,107 ?*

Solution page 288.

Algorithme de Berlekamp-Massey pour la synthèse de registres à décalage linéaire

L'algorithme de Berlekamp-Massey permet de détecter, pour une séquence infinie (S_i), $i \in \mathbb{N}$ d'éléments d'un corps $\mathbb{F}$, la propriété qu'à partir d'un certain rang, ses éléments sont une combinaison linéaire des précédents éléments. C'est ce qu'on appelle une *suite récurrente linéaire*.
Cet algorithme est très utile en théorie des codes, notamment pour cryptanalyser des générateurs aléatoires et des clefs cryptographiques, ou encore pour corriger les erreurs d'un code correcteur cyclique (voir §4.4.4). En particulier il permet de retrouver le polynôme générateur d'un LFSR (voir §1.3.6) en ne connaissant que les premiers termes de la séquence générée par ce LFSR.
Il s'agit, pour la suite (S_i), $i \in \mathbb{N}$, de trouver les $\Pi_0 \dots \Pi_d \in \mathbb{F}$ telles que $\Pi_0 S_t = S_{t-1}\Pi_1 + S_{t-2}\Pi_2 + ... + S_{t-d}\Pi_d$ pour tout $t \geqslant d$. Si l'on utilise ces constantes pour définir un polynôme, $\Pi(X) = \Pi_0 X^d - \Pi_1 X^{d-1} - \Pi_2 X^{d-2} - ... - \Pi_d$, ce polynôme est appelé un polynôme annulateur de la séquence. L'ensemble des polynômes annulateurs est un idéal de l'ensemble $\mathbb{F}[X]$ des polynômes sur $\mathbb{F}$. Comme cet ensemble est un anneau principal, il existe un polynôme annulateur unitaire de degré minimal, appelé polynôme minimal de la séquence.
Comment trouver ce polynôme à partir des seuls coefficients de la séquence ? Si l'on connaît le degré d de ce polynôme, il faut poser d équations linéaires correspondant à la propriété de récurrence linéaire pour $2d$ coefficients et résoudre

le système linéaire antisymétrique ainsi produit :

$$\begin{bmatrix} S_0 & S_1 & \dots & S_{d-1} \\ S_1 & S_2 & \dots & S_d \\ \vdots & \iddots & \iddots & \vdots \\ S_{d-1} & S_d & \dots & S_{2d-1} \end{bmatrix} \cdot \begin{bmatrix} \Pi_d \\ \Pi_{d-1} \\ \vdots \\ \Pi_1 \end{bmatrix} = \begin{bmatrix} S_d \\ S_{d+1} \\ \vdots \\ S_{2d} \end{bmatrix} \tag{1.10}$$

Il reste à déterminer le degré du polynôme minimal. Une première idée est d'essayer à la suite chaque degré possible par degrés croissants et tester chaque polynôme produit sur la suite de la séquence pour voir s'il est annulateur. Si la suite est réellement infinie, cela peut ne jamais se terminer.
Si, au contraire, la suite est finie, on voit sur le système que le degré maximal du polynôme minimal est la moitié du nombre d'éléments de la suite. Cet algorithme nécessite de résoudre plusieurs systèmes linéaires à la suite. En pratique, il est possible de tirer parti de la structure antisymétrique du système pour le résoudre directement, tout en ajoutant des éléments de la séquence au fur et à mesure. Cela donne l'algorithme de Berlekamp-Masseysuivant de complexité simplement $O(d^2)$.

Algorithme 13 Algorithme de Berlekamp-Massey.

Entrées S_0, ..., S_n une séquence d'éléments d'un corps K.
Sorties Le polynôme minimal de la séquence.

$b \leftarrow 1$; $e \leftarrow 1$; $L \leftarrow 0$; $\Pi \leftarrow 1 \in \mathbb{F}[X]$; $\psi \leftarrow 1 \in \mathbb{F}[X]$
Pour k de 0 à n **Faire**
  $\delta \leftarrow S_k + \sum_{i=1}^{L} \Pi_i S_{k-i}$
  Si $(\delta = 0)$ **Alors**
    $e \leftarrow e + 1$
  Sinon, si $2L > k$ **Alors**
    $\Pi \leftarrow \Pi - \frac{\delta}{b} X^e \psi$
    $e \leftarrow e + 1$
  Sinon
    $\Pi \leftarrow \Pi - \frac{\delta}{b} X^e \psi$
    $\psi \leftarrow \Pi$
    $L \leftarrow k + 1 - L; b \leftarrow \delta; e \leftarrow 1$
  Fin Si
  Si $e > TerminaisonAnticipee$ **Alors**
    Arrêter l'algorithme
  Fin Si
  Retourner $\Pi(X)$
Fin Pour

L'idée de l'algorithme est de calculer explicitement les coefficients du polynôme. La mise à jour de Π se fait donc en deux étapes. L'astuce du test $2L > k$ permet de faire chacune de ces deux étapes alternativement. Déroulons l'algorithme sur les trois premiers termes de la suite. Le degré du polynôme minimal augmente donc d'au plus un tous les ajouts de deux coefficients de la séquence. Les δ sont appelés défauts (discrepancy en anglais). Le premier défaut est le premier terme de la suite, $\delta_0 = S_0$ et $\Pi(X)$ devient $1 - S_0X$. Les défauts correspondent à la valeur que prend le polynôme sur la suite de la séquence. Si le défaut est nul, alors le polynôme que l'on avait est annulateur d'une partie supplémentaire de la séquence. Le deuxième défaut est donc $1 - S_0X$ appliqué à la séquence S_0, S_1, soit $\delta_1 = S_1 - S_0^2$. La mise à jour de Π est donc $\Pi - \frac{\delta_1}{\delta_0} X\psi = (1 - S_0X) - \frac{S_1 - S_0^2}{S_0} X$, soit $\Pi = 1 - \frac{S_1}{S_0} X$ qui est bien annulateur de la séquence S_0, S_1. Le troisième défaut vaut ensuite $\delta_2 = S_2 - \frac{S_1^2}{S_0}$ et les deux polynômes Π et ψ valent alors respectivement $1 - \frac{S_1}{S_0} X - \frac{\delta_2}{S_0} X^2$ et $1 - \frac{S_1}{S_0} X$. Ainsi, Π est annulateur de S_0, S_1, S_2 et ψ annulateur de S_0, S_1. Ainsi, comme la multiplication par X de ces annulateurs revient à décaler de 1 leur application sur la séquence initiale, on obtient $\delta_3 = \Pi[S_1, S_2, S_3]$ et $\delta_2 = \psi[S_1, S_2] = (X\psi)[S_1, S_2, S_3]$. Il est alors possible de combiner ces deux polynômes pour annuler également la séquence S_1, S_2, S_3, par $\Pi - \frac{\delta_3}{\delta_2} X\psi$, ce qui est précisément ce que réalise l'algorithme. Par la suite, tant que les défauts suivants sont nuls, cela veut dire que le polynôme déjà obtenu est annulateur de la suite de la séquence. On peut alors continuer jusqu'à être sûr d'obtenir le polynôme minimal des $n+1$ termes de la séquence ou terminer l'algorithme de manière anticipée sans vérifier les derniers défauts.

Pour la complexité, la boucle termine après $2d + TerminaisonAnticipee$ et au plus $n + 1$ itérations. Dans chaque itération, le calcul du défaut nécessite $2\frac{k}{2}$ opérations, la mise à jour du polynôme $2\frac{k}{2}$ opérations également, soit un total de $(2d + TA)(2d + TA + 1)$ opérations.

Il est possible d'utiliser un algorithme rapide pour encore réduire cette complexité, au moins asymptotiquement. L'idée est de voir la séquence également comme un polynôme. Alors le polynôme minimal de la séquence possède la propriété que le produit $\Pi(X)(S_0 + S_1X + S_2X^2 + \ldots)$ n'a qu'un nombre fini de termes non nuls, les termes de degré au plus d. Il est possible de réécrire ceci de la façon suivante :

$$\Pi(x) \cdot (S_0 + S_1X + ... + S_{2n-1}X^{2n-1}) - Q(x) \cdot X^{2n} = R(x) \qquad (1.11)$$

Ainsi, on voit que calculer Π, Q et R peut se faire par l'algorithme d'Euclide arrêté au milieu, c'est-à-dire dès que le degré de R est plus petit que n. La complexité du calcul est alors celle de l'algorithme d'Euclide, en $O(dlog(d))$. En pratique, toutefois, l'algorithme de Berlekamp-Massey reste plus efficace

pour des valeurs de d jusqu'à des dizaines de milliers.

Le paradoxe des anniversaires

Le paradoxe des anniversaire est un résultat de probabilités, appelé paradoxe parce qu'il semble contraire à la première intuition qu'on pourrait avoir. Il est utilisé dans plusieurs méthode d'attaque en cryptanalyse. Il apprend également à se méfier des intuitions en ce qui concerne les probabilités.
Il y a 365 jours dans l'année, et pourtant dans un groupe de plus de 23 personnes il y a plus d'une chance sur deux pour qu'au moins deux d'entre elles aient leur anniversaire le même jour !
En effet, prenons une population de k personnes, sachant que le nombre de jours dans l'année est n, le nombre de combinaisons de k anniversaires différents est $A_n^k = \frac{n!}{n-k!}$. Donc la probabilité que toutes les personnes aient des anniversaires différents est $\frac{A_n^k}{n^k}$. Donc la probabilité pour que deux personnes au moins aient leur anniversaire le même jour est

$$1 - \frac{A_n^k}{n^k}.$$

Ainsi, en considérant 365 jours, cette probabilité est environ d'une chance sur 10 dans un groupe de 9 personnes, de plus d'une chance sur deux dans un groupe de 23 personnes et de 99.4% dans un groupe de 60 personnes. Plus généralement, nous avons le théorème suivant :

Théorème 14. *Dans un ensemble de $\lceil 1.18\sqrt{n} \rceil$ éléments choisis aléatoirement parmi n possibilités, la probabilité de collision est supérieure à 50%.*

Preuve. Nous avons vu que le nombre de collisions sur un espace de taille $n = 2^m$, avec k tirages est $1 - \frac{A_n^k}{n^k}$. Il faut estimer cette probabilité : $1 - \frac{A_n^k}{n^k} = 1 - (1 - \frac{1}{n})(1 - \frac{2}{n}) \dots (1 - \frac{k-1}{n})$. Or, $1 - x < e^{-x}$, pour x positif, donc

$$1 - \frac{A_n^k}{n^k} > 1 - \prod_{i=1}^{k-1} e^{-\frac{i}{n}} = 1 - e^{-\frac{k(k-1)}{2n}}$$

Alors, pour que cette probabilité vaille α, il faut que $k(k-1) = 2n\ln(1-\alpha)$, soit, comme k est positif, $k = \frac{1}{2} + \sqrt{n}\sqrt{\frac{1}{4n} - 2\ln(1-\alpha)}$. Ainsi, pour $\alpha = 0.5$, $k \approx 1.18\sqrt{n}$ (on retrouve pour $n = 365$, $k \approx 22.5$). □

Ce type de probabilités de collision, qu'on penserait plus faible si on se laissait faire par son intuition, permet de construire des attaques contre des systèmes dont la même intuition minimiserait les chances de réussir.

Attaque de Yuval sur les fonctions de hachage

La résistance à la collision des fonctions de hachage peut se mesurer : il faut déterminer la probabilité d'obtenir des collisions, ce qui ressemble fort à la probabilité des collisions des dates d'anniversaires (les dates d'anniversaire jouent le rôle des empreintes des individus).
C'est pourquoi on appelle l'attaque des fonctions de hachage de Yuval l'attaque par anniversaires. Il s'agit de faire passer un message corrompu $\tilde{M}$ à la place d'un message légitime M, de façon à ce que la corruption ne soit pas détectable par une fonction de hachage h. On cherche alors M' et $\tilde{M}'$, tels que $h(M') = h(\tilde{M}')$. On pourra alors par exemple changer frauduleusement M en $\tilde{M}$, ou envoyer M et prétendre avoir envoyé $\tilde{M}$, ce contre quoi devrait protéger h !

Algorithme 14 Attaque des anniversaires de Yuval.

Entrées $h : \{0,1\}^* \longrightarrow \{0,1\}^m$ une fonction de hachage.
M légitime, $\tilde{M}$ frauduleux, une fonction de hachage h, sur m bits.
Sorties $M' \approx M$ et $\tilde{M}' \approx \tilde{M}$ tels que $h(M') = h(\tilde{M}')$.
Générer $t = 2^{\frac{m}{2}} = \sqrt{2^m}$ modifications mineures de M, notées M'.
Pour tout t, calculer $h(M')$.
Générer des $\tilde{M}'$, modifications mineures de $\tilde{M}$, jusqu'à collision avec un M'.

En conséquence du théorème 14, l'espérance du nombre de tirages de $\tilde{M}'$ dans l'attaque de Yuval est $O(t) = O(\sqrt{2^m})$.
Si on utilise l'attaque de Yuval pour envoyer M', puis le répudier plus tard en soutenant que $\tilde{M}'$ avait en fait été envoyé, on a plus d'une chance sur deux de réussir en $\sqrt{2^m}$ essais. Ceci démontre que la force brute peut être efficace si une fonction de hachage n'est pas protégée contre la collision.
Est-ce que cette attaque est vraiment réalisable ? Un calcul simple permet de s'en convaincre : pour une empreinte numérique sur 128 bits, il faudra effectuer de l'ordre de $O(2^{64})$ essais, ce qui est réalisable sur des machines grand public aujourd'hui : un ordinateur tournant à 3 GHz effectue $3 * 10^9 * 24 * 3600 \approx 2^{48}$ opérations par jour, donc il faudrait un peu plus de deux mois sur les 1000 PC d'une entreprise pour trouver une collision.
Mais si l'on utilise des fonctions de hachage sur 160 bits, le coût est multiplié par $2^{16} = 65536$, ce qui reste pour l'instant inatteignable.

Factorisation des nombres composés

On sait relativement facilement reconnaître un nombre premier ou un nombre composé. Mais savoir par quels nombres il est composé est un problème apparemment nettement plus difficile. C'est le problème de la factorisation.

Bien qu'il puisse s'exprimer de manière relativement simple, il n'a pas, jusqu'à présent, de solution vraiment efficace (le célèbre crible d'Ératosthène, par exemple, est inutilisable pour des nombres de plus de *10* chiffres).
La difficulté de ce problème et l'efficacité des méthodes d'attaque sont très importantes, car beaucoup de fonctions à sens unique reposent sur la difficulté de la factorisation, ou alors sur la difficulté d'un autre problème équivalent, comme le logarithme discret. Chercher de bons algorithmes de factorisation est donc quasiment en soi une méthode de cryptanalyse.
De nombreux algorithmes très différents existent, le but de cette section n'est pas de les énumérer tous mais plutôt de donner une idée de ceux qui sont les plus efficaces pour différentes tailles de nombres à factoriser.

Le Rho de Pollard (Nombres de quelques chiffres). La première catégorie de nombres à factoriser est celle des « nombres composés de tous les jours », c'est-à-dire les nombres qui ont moins de *20* chiffres. Un algorithme très efficace est celui de Pollard.L'algorithme ne nécessite que quelques lignes de code (une quarantaine seulement au total) et est très simple à programmer. Soit le nombre m composé, donc on veut trouver les facteurs. Il faut tout d'abord calculer une suite du type $u_{k+1} = f(u_k) \mod m$ de grande période, de sorte que les u_k soient distincts le plus longtemps possible.

0	1	2	...	p	p+1	p+2	...	kp	kp+1	kp+2	...	m-1
		u_i	u_l		u_k			u_h		u_j		

TAB. 1.5: Distribution des multiples de p modulo m.

Ensuite, l'idée est que si p est un facteur de m, les u_k distincts modulo m le seront moins souvent modulo p (voir Tableau 1.5). Dans ce cas, si $u_i = u_j \mod p$ alors le pgcd de m et de $u_i - u_j$ vaut kp et est donc un facteur non trivial de m. Si les u_i sont bien tous distincts, le calcul se termine alors en au plus p étapes. Une première version de l'algorithme consiste à produire des u_i et à chaque nouvel élément de calculer les pgcd avec tous les précédents u_k. Cette version présente deux défauts : tout d'abord il faut stocker de l'ordre de p éléments ; en outre, il faut faire j^2 calculs de pgcd si i et $j > i$ sont les plus petits indices tels que $u_i = u_j \mod p$. La deuxième astuce de Pollard est d'utiliser la détection de cycle de Floyd.Il s'agit de stocker uniquement les u_k tels que k soit une puissance de 2. En effet, puisque les u_k sont générés par une fonction, si $u_i = u_j$, alors pour tout $h \geqslant 0, u_{i+h} = u_{j+h}$ et un cycle se crée modulo p, même si il n'est pas visible directement.
En ne stockant que des puissances de 2, le cycle sera donc détecté seulement pour $u_{2^a} = u_{2^a+j-i}$ avec $2^{a-1} < i \leqslant 2^a$, comme on en voit l'illustration sur la figure 1.16.

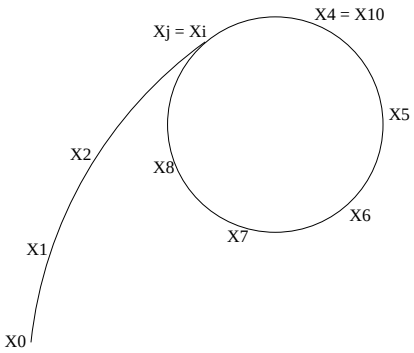


FIG. 1.16: Détection de cycle de Floyd, sous forme de rho.

Or, $2^a + j - i < 2i + j - i = i + j < 2j$. Donc on effectue au plus le double d'étapes nécessaires. Un seul pgcd est calculé à chaque étape et un seul élément supplémentaire est stocké au long de l'algorithme. Cela donne l'algorithme extrêmement simple donné dans l'algorithme 15.

Algorithme 15 Factorisation de Pollard.

Entrées Un entier m, composé.
Sorties p, un facteur non trivial de m.

```
p ← 1 ;
Générer aléatoirement y ;
k ← 0 ;
Tant que (p ← 1) Faire
  Si k est une puissance de 2 Alors
    x ← y ;
  Fin Si
  y ← f(y) mod m ;
  p ← pgcd(y − x, m) ;
  Incrémenter k ;
Fin Tant que
Renvoyer p.
```

Si on prend $f(u) = u^2 + 1$, de sorte que les u_i soient bien tous distincts modulo m par exemple, il faudra au pire $2p$ itérations si p est le plus petit facteur de m, mais beaucoup moins en général :

Théorème 15. *L'algorithme Rho de Pollard a plus d'une chance sur deux de se terminer en $O(\sqrt{p})$ étapes.*

Preuve. La preuve est encore du type de celle du célèbre paradoxe des anni-

versaires. Si k valeurs distinctes u_i sont tirées au hasard, il y a A_p^k combinaisons qui ne donnent pas de collisions entre les u_i sur un total de p^k. Pour que la probabilité de trouver un facteur soit plus grande que 1/2 il faut donc d'après le théorème 14 que $k > 0.5 + 1.18\sqrt{p}$. □

En pratique, cet algorithme factorise en quelques secondes les nombres de 1 à environ 25 chiffres (avec des facteurs de 12 ou 13 chiffres, cela donne environ 10 millions d'itérations !) et devient très rapidement inutilisable pour des facteurs au-delà de 15 chiffres.

Courbes elliptiques (Nombres de quelques dizaines de chiffres). Pour aller plus loin, la méthode des courbes elliptiques de Pollard et Lenstra est une solution. Celle-ci utilise des courbes elliptiques (courbes du type $y^2 = x^3 + ax + b$) dont l'étude dépasse le cadre de ce livre. Néanmoins, sa programmation reste simple (environ 150 lignes de code) et nous pouvons donner quelques idées des propriétés de cet algorithme : il est conjecturé que pour factoriser un entier m, de plus petit facteur p, cet algorithme nécessite un nombre moyen d'opérations de l'ordre de

$$O\left((\ln m)^2 e^{\sqrt{2 \ln p \ln(\ln p)}}\right)$$

En pratique, cet algorithme factorise en quelques secondes les nombres de 25 à 40 chiffres (avec deux facteurs de tailles semblables). En outre, si l'on a de la chance et si l'on choisit une courbe elliptique particulière, celle-ci peut permettre de factoriser très rapidement : le projet ECMNET (Elliptic Curve Method on the Net) consiste à fournir une implémentation de cet algorithme, disponible sur internet Ce projet a permis de factoriser des nombres avec des facteurs jusqu'à 58 chiffres Le problème est que les bonnes courbes elliptiques varient pour chaque nombre à factoriser et qu'il n'existe pas encore de moyen de trouver la bonne courbe pour un nombre donné. Toutefois, cette rapidité de calcul lorsque l'on a une « bonne » courbe elliptique est à l'origine du programme de factorisation sur internet : en effet, de nombreux internautes peuvent récupérer le programme ECM et le lancer pour qu'il essaye différentes courbes elliptiques. Ainsi, un nombre très important de courbes elliptiques peuvent être explorées dans le même temps et accélérer potentiellement la recherche de facteurs premiers.

Cribles de corps de nombre (Le champion du monde). Enfin, le champion actuel pour la factorisation des clefs RSA (produits de deux grands nombres premiers, voir la section 3.4.2) est l'algorithme *Number Field Sieve*, crible de corps de nombre ou parfois crible algébrique en français, qui, pour factoriser un nombre m composé de deux facteurs de tailles respectives similaires,

semble nécessiter un nombre moyen d'opérations de l'ordre de

$$O\left(e^{\sqrt[3]{(7.11112)\ln(m)\ln(\ln(m))^2}}\right)$$

Un corps de nombre est la généralisation des extensions de corps au corps des rationnels (on travaille dans un corps infini $\mathbb{Q}[X]/P$). Le crible de corps de nombre est une généralisation du crible quadratique pour lequel le corps considéré est l'ensemble des entiers modulo m. Par souci de simplification, nous ne presentons que l'idée de ce dernier.
Il s'agit de trouver des couples de nombres tels que leurs carrés soient congrus modulo m : $x^2 = y^2 \mod m$. Alors, $x^2 - y^2 = (x-y)(x+y)$ est un multiple de m et, avec de la chance, l'un de ces deux nombres permet donc de décomposer m.

Exemple : Cherchons à factoriser 7429. Nous calculons au hasard certains carrés : $87^2 = 7429 + 140$ et $88^2 = 7429 + 315$. Or 140 et 315 sont petits par rapport à 7429 et donc plus faciles à factoriser, par exemple avec la méthode de Pollard. On obtient $140 = 2^2 \cdot 5 \cdot 7$ et $315 = 3^2 \cdot 5 \cdot 7$ et l'on peut donc écrire $(227)^2 = (87 \cdot 88)^2 = (2 \cdot 3 \cdot 5 \cdot 7)^2 = (210)^2 \mod 7429$. Nous avons donc une relation du type $x^2 = y^2 \mod 7429$ qui nous donne un facteur de 7429 : $(227 - 210) = 17$, et $7429 = 17 \cdot 347$. Toute la difficulté de l'algorithme consiste à trouver de tels entiers x et y. Pour cela, il faut calculer des carrés de nombres et conserver ceux dont les restes modulo m sont suffisamment petits pour être factorisés par une autre méthode. Ensuite, il faut trouver une combinaison linéaire de ces carrés qui donne un autre carré : dans une matrice mettons en colonne les exposant et en ligne les carrés comme dans la table 1.6.

	Exposant de 2	Exposant de 3	de 5	de 7	de ...
83^2	2	3	1	0	...
87^2	2	0	1	1	...
88^2	0	2	1	1	...
$\vdots$					

Tab. 1.6: Crible quadratique pour 7429.

D'après cette table, $(87 * 88)^2$ est un carré si et seulement si la ligne de 87^2 ajoutée à la ligne de 88^2 ne donne que des exposants pairs. Autrement dit, si M est la matrice des exposants, il faut trouver un vecteur binaire x tel que xM est pair, ou encore trouver une solution modulo 2 au système linéaire associé : x tel que $xM = 0 \mod 2$.

Si l'idée de base est relativement simple, la programmation est un peu plus délicate que pour les précédents algorithmes mais cet algorithme détient les records actuels. Il permit en particulier la factorisation en 2005 d'une clé RSA de 200 chiffres (665 bits). Le temps de calcul nécessaire pour cette dernière factorisation a été gigantesque : plus d'un an et demi de calcul sur plus de 80 machines !

Nombres premiers robustes

La cryptographie RSA (voir le chapitre 3) est fondée sur l'utilisation de deux nombres premiers p et q, et de la difficulté à factoriser leur produit m. Afin de résister aux méthodes de factorisations diverses (dont certaines sont présentées dans les exercices du chapitre 3, page 174), les nombres premiers utilisés doivent vérifier un certain nombre de propriétés :

- Pour résister à la factorisation par courbes elliptiques, p et q doivent être de tailles similaires et suffisamment grands. Par exemple, pour travailler sur 1024 bits ils devraient tous les deux être de taille environ 512 bits.
- $p - q$ doit être relativement grand, sinon il suffit d'essayer pour p ou q les entiers proches de $\sqrt{m}$ (attaque par racines carrées de Fermat).
- Pour résister aux algorithmes $p - 1$ et $p + 1$ de Pollard (qui exploitent la factorisation de $p - 1$ et $p + 1$ si elle est faisable), p et q doivent être des nombres premiers *robustes*, c'est à dire qu'ils doivent chacun vérifier :
  - $p - 1$ possède un grand facteur, noté r.
  - $p + 1$ possède un grand facteur.
  - $r - 1$ possède un grand facteur.

L'algorithme 16 dit de Gordon permet de générer des nombres premiers robustes.

Algorithme 16 Algorithme de Gordon.

Entrées Un nombre de bits b.
Sorties Un nombre premier robuste d'au moins $2b + 1$ bits.
Générer deux nombres premiers s et t de b bits.
Chercher r premier de la forme $2kt + 1$.
Calculer $l \leftarrow (2s^{r-2} \mod r)s - 1$.
Retourner p le plus petit premier de la forme $l + 2hrs$.

Exercice 1.33. *Montrer que la sortie de l'algorithme de Gordon est un nombre premier robuste.* *Solution page 288.*

Résolution du logarithme discret

Après l'exponentiation modulaire, l'autre grande classe de fonctions à sens unique repose sur le problème du logarithme discret.
Soit G un groupe de taille n, possédant un générateur. On pourra considérer par exemple le groupe des inversibles modulo un nombre premier p, $\mathbb{Z}_p^*$.
Le problème consiste, étant donné une racine primitive g et un élément b de G, à trouver le logarithme discret de b en base g, c'est-à-dire trouver x tel que $b = g^x$.
L'algorithme naïf pour résoudre ce problème est d'essayer un à un tous les x possibles jusqu'à trouver le bon. La complexité au pire et en moyenne est en $O(n)$, donc exponentielle par rapport à la taille de n.
Les meilleurs algorithmes connus à ce jour pour résoudre ce problème sont de complexité $O(\sqrt{n})$ dans le cas général. Ces algorithmes sont la plupart du temps dérivés des algorithmes de factorisation : nous allons voir que des variantes du rho de Pollard - $O(\sqrt{n})$ -, et des calculs d'index - $O(n^{1/3})$ - sont applicables dans certains groupes. Mais les complexités sont élevées au carré par rapport à la factorisation. En effet, si l'on travaille modulo un nombre composé de deux nombres premiers $n = pq$, les algorithmes de factorisation ont une complexité qui dépend du plus petit facteur premier : $O(p^{1/3}) = O(n^{1/6})$. C'est pourquoi le logarithme discret permet de travailler sur des nombres deux fois plus petits avec la même sécurité.

Pas de bébé, pas de géant. Cette méthode, développée par Shanks, se déroule en deux phases : la phase des bébés, tests de g^x en g^{x+1} pour tous les x sur un certain intervalle, et la phase des géants, sauts de $g^{x\lfloor\sqrt{n}\rfloor}$ en $g^{(x+1)\lfloor\sqrt{n}\rfloor}$.
L'idée est de décomposer x en deux morceaux $x = i\lfloor\sqrt{n}\rfloor + j$, avec i et j compris entre 1 et $\lceil\sqrt{n}\rceil$. Ainsi, on peut écrire $b = g^x = (g^{\lfloor\sqrt{n}\rfloor})^i g^j$, ou encore $b(g^{-\lfloor\sqrt{n}\rfloor})^i = g^j$. Il faut alors calculer tous les g^j possibles (pas de bébé), et tous les $b(g^{-\lfloor\sqrt{n}\rfloor})^i$ possibles pour essayer de voir s'il n'y en a pas deux qui correspondent (pas de géant). Si fabriquer toutes ces valeurs ne nécessite que $2\sqrt{n}$ multiplications, chercher les correspondances par une méthode naïve nécessite de tester $\sqrt{n}\sqrt{n}$ possibilités dans le pire cas.
L'astuce qui fait diminuer cette complexité consiste à trier d'abord les g^j en ordre croissant (complexité $O(\sqrt{n}\log(\sqrt{n}))$) pour pouvoir faire ensuite les comparaisons par recherche dichotomique en seulement $\sqrt{n}\log_2(\sqrt{n})$ tests !
La complexité en temps est donc améliorée, malheureusement la complexité en espace est telle que cet algorithme est impraticable : il faut stocker les $\sqrt{n}$ entiers. Même pour un nombre d'opérations abordable (autour de $n = 2^{128}$ aujourd'hui) l'espace mémoire nécessaire est alors de l'ordre de plusieurs milliards de Gigaoctets.

Le retour du rho de Pollard. L'algorithme rho de Polard va nous permettre de modifier la méthode pas de bébé, pas de géant pour y introduire la détection de cycles de Floyd. Celle-ci permet de conserver la complexité en temps $O(\sqrt{n}\log(n))$, tout en réduisant drastiquement la complexité en mémoire à seulement $O(\log(n))$ octets. Par rapport à l'algorithme de factorisation, il faut modifier la fonction génératrice de la séquence de la façon suivante : Construire trois sous ensembles S_1, S_2, S_3 de G de tailles semblables (par exemple dans $\mathbb{F}_p$, on peut prendre $S_1 = \{u = 1 \mod 3\}$, $S_2 = \{u = 2 \mod 3\}$ et $S_3 = \{u = 0 \mod 3\}$). On définit alors la fonction génératrice f par

$$u_{k+1} = f(u_k) = \begin{cases} bu_k & \text{si } u_k \in S_1 \\ u_k^2 & \text{si } u_k \in S_2 \\ gu_k & \text{si } u_k \in S_3 \end{cases}$$

Ainsi, tout élément de la suite peut s'écrire sous la forme $u_k = g^{i_k} b^{j_k}$ pour un certain i_k et un certain j_k. Or, comme pour la factorisation par la méthode rho, les u_k sont répartis modulo p et donc une collision $u_k = u_l$ arrive en moyenne après $\sqrt{p}$ tirages, même si $j_k \neq j_l$ et $i_k \neq i_l$. La fonction f assure, toujours comme pour la factorisation, que cette collision va être reproduite constamment après k et donc il est possible de trouver y tel que $u_y = u_{2y}$ grâce à l'algorithme de Floyd en une complexité mémoire de seulement quelques entiers de taille $O(\log(n))$.
Nous avons alors $u_k = g^{i_k} b^{j_k} = g^{i_l} b^{j_l} = u_l$ et en même temps $j_k \neq j_l$. Dans ce cas on obtient $b^{j_k - j_l} = g^{i_l - i_k}$, ce qui veut dire, puisque $b = g^x$, que dans l'espace des indices on obtient directement x :

$$x = (i_l - i_k) \cdot (j_k - j_l)^{-1} \mod n$$

Attention, dans $\mathbb{Z}_p^*$ par exemple, $n = p-1$ et $j_k - j_l$, bien que non nul, n'est pas forcément inversible. Si ce n'est pas le cas, il faudra recommencer l'algorithme.

Calcul d'index de Coppersmith. Tout comme nous venons de modifier l'algorithme de Pollard pour l'adapter au calcul du logarithme discret, il est possible de modifier les algorithmes de crible. Nous montrons l'idée de l'algorithme sur un exemple.

Exemple : Comment trouver x, tel que $17 = 11^x \mod 1009$? On sait que 1009 est premier et que 11 est une racine primitive , soit $Z_{1009} = \{11^i$, pour $i = 1 \ldots \varphi(1009) = 1008\}$. L'idée consiste à tirer au hasard des valeurs de i de sorte que $v_i = 11^i \mod 1009$ soit facilement factorisable (*i.e.* composé de petits facteurs premiers).

En pratique, on se donne une base de facteurs premiers, par exemple $B = \{2, 3, 5, 7, 11\}$. Ensuite, pour chaque nombre premier $p_j \in B$, on divise v_i par la plus grande puissance de p_j divisant v_i. Si à la fin de ce processus, on obtient la valeur 1, v_i sera décomposable selon les facteurs de B.
Après quelques tirages aléatoires, 104, 308, 553 et 708 ont été retenus :

$$\begin{aligned} 11^{104} &= 363 = 3 \cdot 11^2 && \mod 1009 \\ 11^{308} &= 240 = 2^4 \cdot 3 \cdot 5 && \mod 1009 \\ 11^{553} &= 660 = 2^2 \cdot 3 \cdot 5 \cdot 11 && \mod 1009 \\ 11^{708} &= 1000 = 2^3 \cdot 5^3 && \mod 1009 \end{aligned}$$

En passant au logarithme, on obtient un système linéaire dans l'espace des exposants, dont les inconnues sont les logarithmes discrets de 2, 3 et 5 :

$$\begin{aligned} 104 &= \log_{11}(3) + 2 && \mod 1008 \\ 308 &= 4\log_{11}(2) + \log_{11}(3) + \log_{11}(5) && \mod 1008 \\ 553 &= 2\log_{11}(2) + \log_{11}(3) + \log_{11}(5) + 1 && \mod 1008 \\ 708 &= 3\log_{11}(2) + 3\log_{11}(5) && \mod 1008 \end{aligned}$$

Cela donne $\log_{11}(3) = 102 \mod 1008$ ou encore $11^{102} = 3 \mod 1009$, puis $\log_{11}(2) = (308-553+1)/2 = 886 \mod 1008$ et $\log_{11}(5) = 308-4*886-102 = 694 \mod 1008$.
Il reste alors à trouver un nombre de la forme $17 \cdot 11^y$ dont le reste se factorise dans la base $B = \{2, 3, 5, 7, 11\}$. Toujours avec la même méthode, on trouve par exemple après quelques essais aléatoires :$17 \cdot 11^{218} = 2 \cdot 3 \cdot 5 \cdot 11 \mod 1009$, et donc $x = 886 + 102 + 694 + 1 - 218 = 457 \mod 1008$ vérifie bien $17 = 11^{457} \mod 1009$.
Les algorithmes d'index ainsi obtenus sont les plus performants dans certains corps particuliers. Le record est détenu par une équipe française qui, en novembre 2005, réussit à calculer un logarithme discret dans le corps $\mathbb{F}_{2^{613}}$) en seulement 17 jours sur les 64 processeurs du supercalculateur Teranova de Bull.

Nous disposons à la fin de ce chapitre d'un arsenal suffisant pour commencer à nous spécialiser. Nous savons construire les objets dont nous nous servirons tout au long de ce livre, et nous avons décrit les algorithmes courants qui font les fondements de la théorie des codes. Chacun des chapitres suivants y fera référence, selon les besoins des objectifs spécifiques auquel il s'attaque : compression, cryptage ou correction.

Chapitre 2

Théorie de l'information et compression

Nous avons vu dans le chapitre précédent quelques principes de codes pour la compression : le code du fax, les codes instantanés, le principe de l'entropie... Le but de ce chapitre est de faire une étude plus poussée et plus exhaustive de ces principes, et de donner des exemple de codes compresseurs couramment utilisés en informatique.
L'objectif est une optimisation du temps de transit de chaque message, et de l'espace de stockage. À cette fin, il faut construire des codes qui optimisent la taille des messages. Nous supposons ici que le canal n'est pas soumis aux perturbations (on parle de codage sans bruit), la gestion des erreurs est étudiée au dernier chapitre. Nous allons construire des techniques d'encodage permettant de choisir des codes efficaces ainsi qu'une théorie importante permettant de quantifier l'information contenue dans un message, de calculer la taille minimale d'un schéma de codage, et de connaître ainsi la « valeur » d'un code donné. Nous nous intéressons principalement à la compression de données sans perte, c'est-à-dire que la compression suivie d'une décompression ne modifie pas le fichier initial. Au contraire, en fin de chapitre nous introduirons quelques techniques pour compresser des images ou du son, qui peuvent autoriser des pertes sur la qualité visuelle ou sonore.

Exercice 2.1 (De l'impossibilité de compresser sans perte TOUS les fichiers).

1. *Combien y-a-t-il de fichiers distincts de taille exactement N bits ?*
2. *Combien y-a-t-il de fichiers distincts de taille strictement inférieure à N bits ?*
3. *Conclure.*

Solution page 289.

2.1 Théorie de l'information

La théorie de l'information donne le cadre mathématique aux codes compresseurs. On rappelle qu'un *alphabet* est un ensemble fini qui sert à former les messages contenant l'information à coder ou l'information codée. L'ensemble des lettres du message source est l'*alphabet source* et l'ensemble des lettres du code l'*alphabet du code*. Par exemple, l'alphabet latin est l'ensemble des lettres que nous utilisons pour écrire ce texte, ou $\{0, 1\}$ est l'alphabet qui sert à écrire les messages qui doivent transiter dans la plupart des canaux numériques. L'ensemble des séquences sur un alphabet V est noté V^+, et l'image de l'alphabet source par la fonction de codage est un sous-ensemble de V^+ appelé l'ensemble des *mots de code*, à qui on donne parfois le nom de *code*, tout simplement, surtout en théorie de l'information.
Un *code* C sur un alphabet V est alors un sous-ensemble fini de V^+. Le code est constitué des briques de base à partir desquelles les messages transmis seront construits. Un élément m de C est appelé *mot de code*. Sa longueur est notée $l(m)$. L'*arité* du code est le cardinal de V. Un code d'arité 2 est dit *binaire*.
Par exemple, $C = \{0, 10, 110\}$ est un code binaire d'arité 2, sur l'alphabet $V = \{0, 1\}$.

2.1.1 Longueur moyenne d'un code

Dans toute cette partie, dans un souci de simplicité et de part leur importance pratique en télécommunications, on s'intéresse plus particulièrement aux codes binaires. La plupart des résultats sont cependant généralisables à des codes quelconques.
Tous les mots de code n'étant pas toujours de la même longueur, on utilise une mesure dépendant des fréquences d'apparition pour estimer la longueur des messages qui coderont une source. On rappelle qu'une *source d'information* est constituée d'un alphabet S et d'une distribution de probabilités $\mathcal{P}$ sur S. Pour un symbole s_i d'une source $\mathcal{S} = (S, \mathcal{P})$, $P(s_i)$ est la probabilité d'occurrence de s_i, et $l(m)$ désigne la longueur d'un mot m (de source ou de code).
Soient $\mathcal{S} = (S, \mathcal{P})$ où $S = \{s_1, \ldots, s_n\}$, et C un code de $\mathcal{S}$, dont la fonction de codage est f (C est l'image de S par f). La *longueur moyenne* du code C est :

$$l(C) = \sum_{i=1}^{n} l(f(s_i))P(s_i)$$

Exemple : $S = \{a, b, c, d\}$, $\mathcal{P} = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$, $V = \{0, 1\}$.
Si $C = \{f(a) = 00, f(b) = 01, f(c) = 10, f(d) = 11\}$, la longueur moyenne du schéma est 2.

Si $C = \{f(a) = 0, f(b) = 10, f(c) = 110, f(d) = 1110\}$, la longueur moyenne du schéma est : $1 * \frac{1}{2} + 2 * \frac{1}{4} + 3 * \frac{1}{8} + 4 * \frac{1}{8} = 1.875$.
Nous utilisons la longueur moyenne d'un schéma de codage pour mesurer son efficacité.

2.1.2 L'entropie comme mesure de la quantité d'information

Nous arrivons aux notions fondamentales de la théorie de l'information. Soit une source $\mathcal{S} = (S, \mathcal{P})$. Nous ne connaissons de cette source qu'une distribution de probabilité, et nous cherchons à mesurer quantitativement à quel point nous ignorons le comportement de $\mathcal{S}$. Par exemple, cette *incertitude* est plus grande si le nombre de symboles dans S est plus grand. Elle est faible si la probabilité d'occurrence d'un symbole est proche de 1, et plus forte en cas d'équiprobabilité.
L'entropie d'une source sera utilisée comme mesure de la quantité moyenne d'information contenue dans cette source.
Par exemple, imaginons un jet de dé dont la valeur ne nous est communiquée que par des comparaisons à un nombre que nous pouvons choisir, combien faut-il de questions pour déterminer quelle est la valeur obtenue ? En procédant par dichotomie, il suffit de $3 = \lceil log_2(6) \rceil$ questions. Supposons maintenant que le dé est pipé, le 1 sort une fois sur deux et les cinq autres une fois sur 10. Si la première question est : « est-ce 1 ? » dans la moitié des cas une question aura suffit, dans l'autre moitié il faudra 3 questions supplémentaires. Ainsi le nombre moyen de questions nécessaires sera $\frac{1}{2} * 1 + \frac{1}{2} * 4 = -\frac{1}{2} log_2(\frac{1}{2}) - 5 * \frac{1}{10} \lceil log_2(\frac{1}{10}) \rceil = 2.5$. En fait, il est encore possible d'affiner ce résultat en remarquant qu'il n'est pas toujours nécessaire de poser 3 questions pour déterminer lequel de $2, 3, 4, 5, 6$ a été tiré : si la dichotomie sépare ces 5 possibilités en deux groupes $2, 3$ et $4, 5, 6$, deux questions supplémentaires seulement seront nécessaires pour trouver 2 ou 3, et seuls 5 et 6 ne pourront être départagés qu'en trois questions. Sur un grand nombre de tirage, il est encore possible de faire mieux si les questions ne séparent pas toujours de la même façon, par exemple en $2, 3, 5$ et $4, 6$, pour que deux questions soient alternativement nécessaires, etc. Poussé à l'infini, ce raisonnement implique que la moyenne des questions nécessaires pour les 5 possibilités vaut $log_2(10)$!
La quantité d'information contenue dans ce jet de dé (facilement généralisable à toute source) est ainsi définie intuitivement par le nombre moyen de questions.
Il est ainsi naturel de choisir l'entropie pour mesure, à savoir, pour une source $(S, \mathcal{P})$, $\mathcal{P} = (p_1, \ldots, p_n)$,

$$H(\mathcal{S}) = \sum_{i=1}^{n} p_i \log_2(\frac{1}{p_i}) \ .$$

C'est une mesure de *l'incertitude* liée à une loi de probabilités, ce qui est toujours illustré par l'exemple du dé : On considère la variable aléatoire (source) issue du jet d'un dé à n faces. Nous avons vu qu'il y a plus d'incertitude dans le résultat de ce jet si le dé est normal que si le dé est biaisé. Ce qui se traduit par : pour tous $p_1, \ldots, p_n$, $H(p_1, \ldots, p_n) \leqslant H(\frac{1}{n}, \ldots, \frac{1}{n}) = \log_2 n$, comme nous l'a appris la propriété 1 du chapitre 1.

2.1.3 Théorème de Shannon

Ce théorème fondamental de la théorie de l'information est connu sous le nom de *théorème de Shannon* ou *théorème du codage sans bruit.*
Nous commençons par énoncer le théorème dans le cas d'une source sans mémoire.

Théorème 16. *Soit une source $\mathcal{S}$ sans mémoire d'entropie H. Tout code uniquement déchiffrable de $\mathcal{S}$ sur un alphabet V de taille q (i.e. $q = |V|$), de longueur moyenne l, vérifie :*

$$l \geqslant \frac{H}{\log_2 q}\,.$$

De plus, il existe un code uniquement déchiffrable de $\mathcal{S}$ sur un alphabet de taille q, de longueur moyenne l, qui vérifie :

$$l < \frac{H}{\log_2 q} + 1\,.$$

Preuve. *Première partie* : Soit $C = (c_1, \ldots, c_n)$ un code de $\mathcal{S}$ uniquement déchiffrable sur un alphabet de taille q, et $(l_1, \ldots, l_n)$ les longueurs des mots de C. Soit $K = \sum_{i=1}^{n} \frac{1}{q^{l_i}}$, $K \leqslant 1$ d'après le théorème de Kraft (voir à la page 78). Soient $(q_1, \ldots, q_n)$ tels que $q_i = \frac{q^{-l_i}}{K}$ pour tout $i = 1, \ldots, n$. On a $q_i \in [0, 1]$ pour tout i, et $\sum_{i=1}^{n} q_i = 1$, donc $(q_1, \ldots, q_n)$ est une distribution de probabilités. Le lemme de Gibbs (voir page 31) s'applique, et on obtient

$$\sum_{i=1}^{n} p_i \log_2 \frac{q^{-l_i}}{K p_i} \leqslant 0\,;\ \text{autrement dit}\ \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \leqslant \sum_{i=1}^{n} p_i l_i \log_2 q + \log_2 K.$$

Or, comme $\log_2 K \leqslant 0$, on a donc $H(\mathcal{S}) \leqslant l * \log_2 q$; d'où le résultat.
Seconde partie : Soient $l_i = \lceil \log_q \frac{1}{p_i} \rceil$. Comme $\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leqslant 1$ (car $q^{l_i} \geqslant \frac{1}{p_i}$), il existe un code de $\mathcal{S}$ sur un alphabet de taille q, uniquement déchiffrable, avec des longueurs de mots égales à $(l_1, \ldots, l_n)$ et donc de longueur moyenne $l = \sum_{i=1}^{n} p_i l_i$. La propriété de partie entière supérieure nous donne ensuite $\log_q \frac{1}{p_i} + 1 > l_i$ et, par suite, $\sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} > \sum_{i=1}^{n} p_i l_i \log_2 q - \log_2 q$. Ce qui s'écrit $H(\mathcal{S}) > \log_2 q\,((\sum_{i=1}^{n} p_i l_i) - 1) \geqslant \log_2 q(l-1)$ et prouve le théorème. □

On en déduit le théorème pour la $k^{\text{ième}}$ extension de $\mathcal{S}$:

Théorème 17. *Soit une source $\mathcal{S}$ sans mémoire d'entropie H. Tout code uniquement déchiffrable de $\mathcal{S}^k$ sur un alphabet de taille q, de longueur moyenne l_k, vérifie :*

$$\frac{l_k}{k} \geqslant \frac{H}{\log_2 q}\,.$$

De plus, il existe un code uniquement déchiffrable de $\mathcal{S}^k$ sur un alphabet de taille q, de longueur moyenne l_k, qui vérifie :

$$\frac{l_k}{k} < \frac{H}{\log_2 q} + \frac{1}{k}\,.$$

Preuve. La preuve de ce théorème est immédiate d'après la propriété 2 de la page 35 selon laquelle $H(\mathcal{S}^k) = k * H(\mathcal{S})$. □

Pour une source stationnaire quelconque, le théorème peut s'énoncer :

Théorème 18. *Pour toute source stationnaire d'entropie H, il existe un procédé de codage uniquement déchiffrable sur un alphabet de taille q, et de longueur moyenne l, aussi proche que l'on veut de sa borne inférieure $H/\log_2(q)$.*

En théorie, il est donc possible de trouver un code s'approchant indéfiniment de l'entropie. En pratique pourtant, si le procédé de codage consiste à coder les mots d'une extension de la source, on est limité évidemment par le nombre de ces mots ($|\mathcal{S}^k| = |\mathcal{S}|^k$, ce qui peut représenter un très grand nombre de mots). Nous allons voir dans la suite plusieurs procédés de codage et leur relation avec cette borne théorique.

2.2 Codage statistique

Les codages statistiques utilisent la fréquence de chaque caractère de la source pour la compression et, en codant les caractères les plus fréquents par des mots plus courts, se positionnent proches de l'entropie.

2.2.1 Algorithme de Huffman

Cette méthode permet de trouver le meilleur schéma d'encodage d'une source sans mémoire $\mathcal{S} = (S, \mathcal{P})$. L'alphabet de codage est V, de taille q. Il est nécessaire à l'optimalité du résultat de vérifier que $q-1$ divise $|S|-1$ (afin d'obtenir un arbre localement complet). Dans le cas contraire, il est facile de rajouter des symboles à S, de probabilités d'occurrence nulle, jusqu'à ce que $q-1$ divise $|S|-1$. Les mots de codes associés (les plus longs) ne seront pas utilisés.

Algorithme 17 Description de l'algorithme de Huffman.

On construit avec l'alphabet source S un ensemble de nœuds isolés auxquels on associe les probabilités de $\mathcal{P}$.

FIG. 2.1 : Algorithme de Huffman : départ.

Soient $p_{i_1}, \ldots, p_{i_q}$ les q symboles de plus faibles probabilités. On construit un arbre (sur le modèle des arbres de Huffman), dont la racine est un nouveau nœud et auquel on associe la probabilité $p_{i_1} + \ldots + p_{i_q}$, et dont les branches sont incidentes aux nœuds $p_{i_1}, \ldots, p_{i_q}$. La figure 2.2 montre un exemple de cette opération pour $q = 2$.

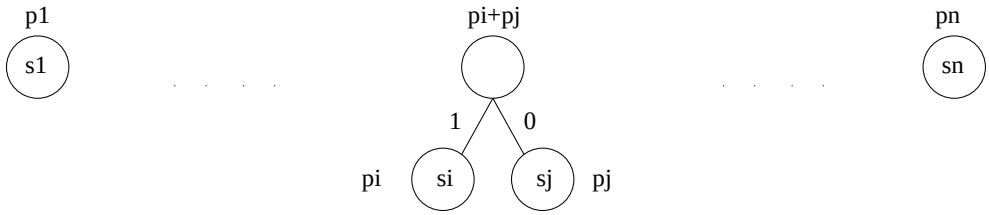


FIG. 2.2 : Algorithme de Huffman : première étape ($q = 2$).

On recommence ensuite avec les q plus petites valeurs parmi les nœuds du plus haut niveau (les racines), jusqu'à n'obtenir qu'un arbre (à chaque itération, il y a $q-1$ éléments en moins parmi les nœuds de plus haut niveau), dont les mots de S sont les feuilles, et dont les mots de code associés dans le schéma ainsi construit sont les mots correspondant aux chemins de la racine aux feuilles.

Exemple : Soit la source à coder sur $V = \{0, 1\}$

Symbole	Probabilité
a	0,35
b	0,10
c	0,19
d	0,25
e	0,06
f	0,05

Les étapes successives de l'algorithme sont décrites par la figure 2.3.
Le code de Huffman construit est alors :

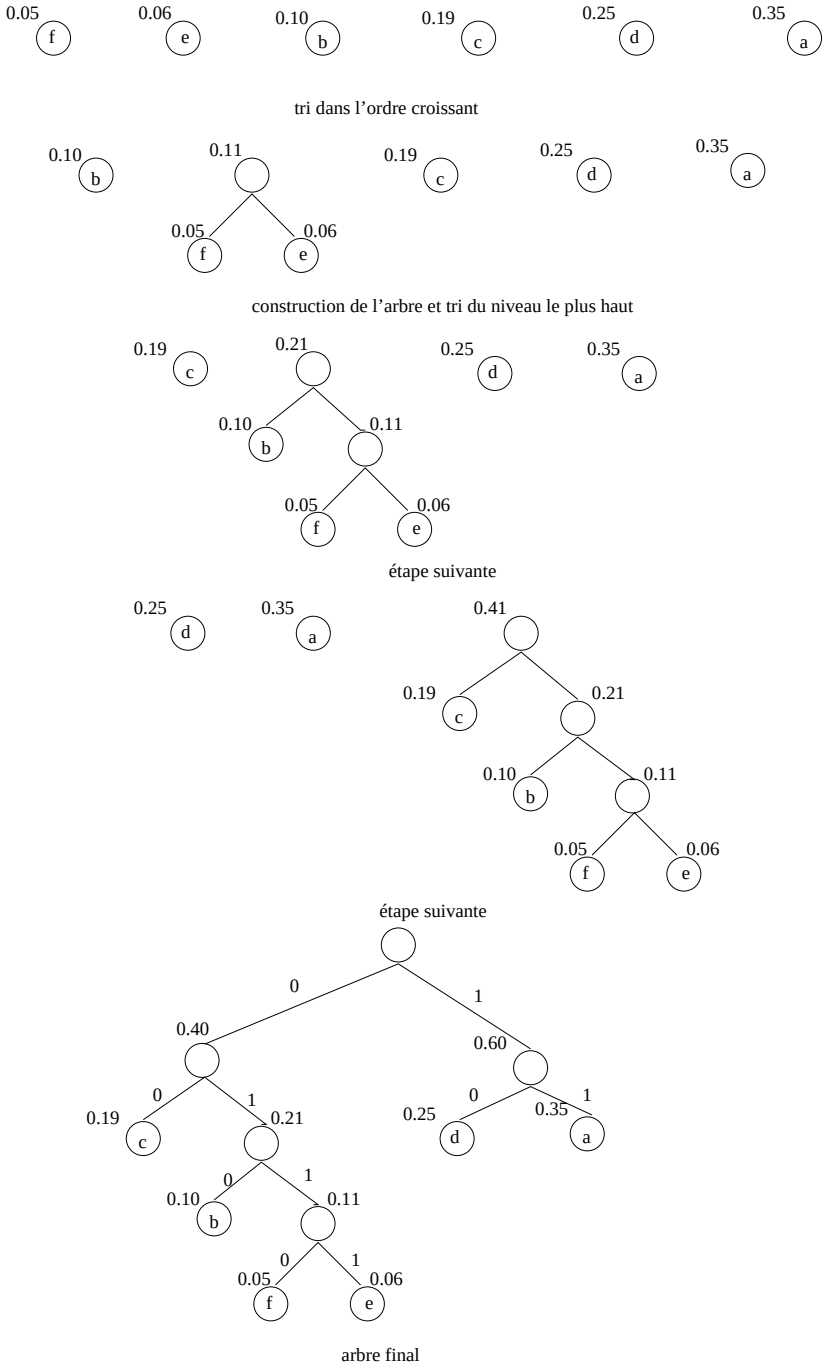


FIG. 2.3: Exemple de construction d'un code de Huffman.

Symbole	Mot de code
a	11
b	010
c	00
d	10
e	0111
f	0110

Exercice 2.2. *Cet exercice introduit des éléments théoriques sur la valeur du code généré par l'algorithme de Huffman. Soit la source* $\mathcal{S} = (S, \mathcal{P})$*, où* $S = (0, 1)$*, et* $P(0) = 0.99$ *et* $P(1) = 0.01$*.*

1. *Calculer l'entropie de* $\mathcal{S}$*.*
2. *Donner le code généré par l'algorithme de Huffman sur la troisième extension* $\mathcal{S}^3$*. Quel est son taux de compression ?*
3. *Que pouvez-vous dire de l'optimalité de l'algorithme de Huffman en comparant les taux de compression obtenus avec ceux de l'exercice 1.24, page 79 ? Cela est-il conforme au théorème de Shannon ?*

Solution page 289.

Exercice 2.3 (Pile ou Face pour jouer au 421). *On désire jouer au lancé de dé, avec pour unique moyen une pièce de monnaie. On va donc chercher à coder un dé non pipé à 6 faces avec une pièce non pipée à deux faces.*

1. *Quelle est l'entropie d'un dé ?*
2. *Proposer un algorithme de codage.*
3. *Calculer la longueur moyenne de ce codage.*
4. *Ce codage est-il optimal ?*

Solution page 289.

L'algorithme de Huffman est optimal

Théorème 19. *Le code issu de l'algorithme de Huffman est optimal parmi tous les codes instantanés de* $\mathcal{S}$ *sur* V*.*

Preuve. On suppose dans la preuve pour alléger les notations que $q = 2$, mais on peut à toutes les étapes généraliser automatiquement.
On sait qu'un code instantané peut être représenté par un arbre de Huffman. Soit A l'arbre représentant un code optimal, et H l'arbre représentant le code issu de l'algorithme de Huffman.

Remarquons que dans A, il n'existe pas de nœud avec un seul fils dont les feuilles contiennent des mots du code (en effet, on remplace un tel nœud par son fils et on obtient un meilleur code).

Remarquons maintenant que dans A, si pour des mots c_1, c_2, les probabilités d'apparition respectives p_1, p_2 satisfont $p_1 < p_2$, alors les hauteurs respectives des feuilles représentant c_1, c_2 : l_1, l_2 satisfont $l_1 \geqslant l_2$ (en effet, dans le cas contraire, on remplace c_1 par c_2 dans l'arbre et on obtient un code meilleur). On peut donc supposer que A représente un code optimal pour lequel les deux mots de plus faibles probabilités sont deux feuilles "sœurs" (elles ont le même père).

On raisonne maintenant par récurrence sur le nombre de feuilles n dans A. Pour $n = 1$, le résultat est évident. Pour $n \geqslant 2$ quelconque, on considère les deux feuilles sœurs correspondant aux mots c_1, c_2 de plus faibles probabilités d'apparition p_1, p_2 dans A. D'après le principe de construction de Huffman, c_1 et c_2 sont des feuilles sœurs dans H. On définit alors $H' = H \setminus \{c_1, c_2\}$. C'est un codage de Huffman pour le code $C' = C \setminus \{c_1, c_2\} \cup \{c\}$, c étant un mot de probabilité d'apparition $p_1 + p_2$. H' représente par le principe de récurrence le meilleur code instantané sur C', donc de longueur moyenne inférieure à $A' = A \setminus \{c_1, c_2\}$. Donc, et d'après les premières remarques, la longueur moyenne de H est inférieure à la longueur moyenne de A. □

Il faut bien comprendre la signification de ce théorème : il ne dit pas que l'algorithme de Huffman est le meilleur pour coder une information dans tous les cas ; mais qu'en fixant pour modèle une source $\mathcal{S}$ sans mémoire sur un alphabet V, il n'y a pas de code plus efficace qui a la propriété du préfixe. Dans la pratique (voir à la fin de ce chapitre), les codes jouent sur le modèle en choisissant un modèle de source adapté. En particulier, on peut obtenir des codes plus efficaces à partir des extensions de la source, comme on peut le voir sur l'exemple suivant :
Soit $\mathcal{S} = (S, \mathcal{P})$, $S = (s_1, s_2)$, $\mathcal{P} = (1/4, 3/4)$. Un codage de Huffman pour $\mathcal{S}$ donne évidemment $s_1 \to 0$ et $s_2 \to 1$, et sa longueur moyenne est 1.
Un codage de Huffman pour $\mathcal{S}^2 = (S^2, \mathcal{P}^2)$ donne :

$$\begin{aligned} s_1 s_1 &\to 010 \\ s_1 s_2 &\to 011 \\ s_2 s_1 &\to 00 \\ s_2 s_2 &\to 1 \end{aligned}$$

et sa longueur moyenne est $3 * \frac{1}{16} + 3 * \frac{3}{16} + 2 * \frac{3}{16} + \frac{9}{16} = \frac{27}{16} = 1,6875$
La longueur moyenne de ce code est donc $l = 1,6875$, et en comparaison avec le code sur $\mathcal{S}$ (les mots de $\mathcal{S}^2$ sont de longueur 2), $l = 1,6875/2 = 0,84375$, ce

qui est meilleur que le code sur la source originelle.
Nous pouvons encore améliorer ce codage en examinant la source $\mathcal{S}^3$. Il est aussi possible d'affiner le codage par une meilleure modélisation de la source : souvent, l'occurrence d'un symbole n'est pas indépendante des symboles précédemment émis par une source (dans le cas d'un texte, par exemple). Dans ce cas, les probabilités d'occurrence sont conditionnelles et il existe des modèles (le modèle de Markov, en particulier) qui permettent un meilleur codage. Mais ces procédés ne conduisent pas à des améliorations infinies. L'entropie reste un seuil pour la longueur moyenne, en-deçà duquel on ne peut pas trouver de code.

Exercice 2.4. *On cherche à coder des jets successifs (en nombre supposé infini) d'un dé faussé. Les symboles de la source sont notés (1,2,3,4,5,6), et suivent la loi de probabilité d'apparition (0.12,0.15,0.16,0.17,0.18,0.22).*

1. *Quelle est l'entropie de cette source ?*
2. *Proposer un code ternaire (sur un alphabet à trois chiffres) issu de l'algorithme de Huffman pour cette source. Quelle est sa longueur moyenne ? Quelle longueur moyenne minimale peut-on espérer pour un tel code ternaire ?*
3. *Même question pour un code binaire.*

Solution page 290.

Exercice 2.5. *(Suite de l'exercice précédent) L'algorithme de Huffman construit un code où des mots de source de longueur fixes sont codés par des mots de code de longueur variable. L'organisation de la mémoire où l'on stocke le code impose parfois une longueur fixe aux mots de code, et on code alors des séquences de longueur variable des chiffres de la source.*
Les codes dits de Tunstall *sont les codes optimaux qui suivent ce principe. En voici la méthode de construction dans le cas d'un alphabet binaire. Si la longueur choisie des mots de code est k, il faut trouver les 2^k séquences de chiffres de source qu'on va choisir de coder. Au départ, l'ensemble des candidats est l'ensemble des mots de longueur 1 (ici, {1,2,3,4,5,6}). Soit dans cet ensemble le mot le plus probable (ici, 6). On construit toutes les séquences réalisables en rajoutant un chiffre à ce mot, et on remplace ce mot par ces séquences dans l'ensemble des candidats (ici, on obtient alors {1,2,3,4,5,61,62,63,64,65,66}). On recommence alors cette opération tant que la taille de l'ensemble des candidats n'est pas strictement supérieure à 2^k (on arrête avant d'avoir dépassé cette valeur). On code alors toutes ces séquences par des mots de longueur k.*

1. *Sur un alphabet binaire, construire un code de Tunstall pour le dé, pour des longueurs de mots de code $k = 4$.*
2. *Comment est codée la séquence "6664" par Tunstall ? par Huffman ?*

3. *Pour chaque mot de code, calculer le nombre de bits de code nécessaires par caractère de source. En déduire la longueur moyenne* par bit de source *du code de Tunstall*

Solution page 290.

2.2.2 Codage arithmétique

Le codage arithmétique est une méthode statistique souvent meilleure que le codage de Huffman ! Or nous avons vu que Huffman était optimal, quelle optimisation est donc possible ? En codage arithmétique, chaque caractère peut être codé sur un nombre non entier de bits : des chaînes entières de caractères plus ou moins longues sont encodées sur un seul entier ou réel machine. Par exemple, si un caractère apparaît avec une probabilité de 0.9, la taille optimale du codage de ce caractère pourrait être de $\frac{H}{\log_2 Q} = \frac{9/10\log_2(10/9)+1/10*\log_2(10/1)}{\log_2 10}$ soit environ 0.14 bits alors qu'un algorithme de type Huffman utiliserait sûrement 1 bit entier. Nous allons donc voir comment le codage arithmétique permet de faire mieux.

Arithmétique flottante

L'idée du codage arithmétique est d'encoder les caractères par intervalles. La sortie d'un code arithmétique est un simple réel entre 0 et 1 construit de la manière suivante : à chaque symbole on associe une portion de l'intervalle $[0, 1[$ qui a pour taille sa probabilité d'occurrence. L'ordre d'association n'importe pas pourvu que soit le même pour le codage et le décodage. Par exemple, une source et les intervalles associés sont donnés dans le tableau suivant.

Symbole	a	b	c	d	e	f
Probabilité	0,1	0,1	0,1	0,2	0,4	0,1
Intervalle	[0,0.1[	[0.1,0.2[	[0.2,0.3[	[0.3,0.5[	[0.5,0.9[	[0.9,1[

Un message sera codé par un nombre choisi dans un intervalle dont les bornes contiennent l'information du message. A chaque caractère du message, on affine l'intervalle, en attribuant la portion qui correspond au caractère. Par exemple, si l'intervalle courant est $[0.15, 0.19[$ et qu'on code le caractère b, on attribue la portion $[0.1, 0.2[$ relative à $[0.15, 0.19[$, soit $[0.15 + (0.19 - 0.15) * 0.1 = 0.154, 0.15 + (0.19 - 0.15) * 0.2 = 0, 158[$.
Si le message à coder est "bebecafdead", cela donne l'intervalle réel de la table 2.1 : tous les réels entre 0.15633504384 et 0.15633504640 correspondent à la chaîne "bebecafdead", choisissons par exemple "0.15633504500". Le programme de codage est donc très simple et peut s'écrire schématiquement par l'algorithme 18.

Symbole	Borne inférieure	Borne supérieure
b	0.1	0.2
e	0.15	0.19
b	0.154	0.158
e	0.1560	0.1576
c	0.15632	0.15648
a	0.156320	0.156336
f	0.1563344	0.1563360
d	0.15633488	0.15633520
e	0.156335040	0.156335168
a	0.156335040	0.1563350528
d	0.15633504384	0.15633504640

TAB. 2.1: Codage arithmétique de "bebecafdead".

Algorithme 18 Codage arithmétique.

Soit borneInf $\leftarrow 0.0$
Soit borneSup $\leftarrow 1.0$
Tant que il y a des symboles à coder **Faire**
  C $\leftarrow$ symbole à coder
  Soient x, y les bornes de l'intervalle correspondant à C dans la table 2.1
  taille $\leftarrow$ borneSup $-$ borneInf ;
  borneSup $\leftarrow$ borneInf + taille $*y$
  borneInf $\leftarrow$ borneInf + taille $*x$
Fin Tant que
Retourner borneSup

Pour le décodage, il suffit alors de repérer dans quel intervalle se trouve "0.15633504500" : c'est $[0.1, 0.2[$, donc la première lettre est un 'b'. Il faut ensuite se ramener au prochain intervalle, on retire la valeur inférieure et on divise par la taille de l'intervalle de 'b', à savoir $(0.15633504500 - 0.1)/0.1 = 0.5633504500$. Ce nouveau nombre nous indique que la valeur suivante est un 'e'. La suite du décodage est indiqué table 2.2, et le programme est décrit dans l'algorithme 19.

En résumé, l'encodage réduit progressivement l'intervalle proportionnellement aux probabilités des caractères. Le décodage fait l'inverse en augmentant cet intervalle.

Réel	Intervalle	Symbole	Taille
0.15633504500	[0.1,0.2[	b	0,1
0.5633504500	[0.5,0.9[	e	0,4
0.158376125	[0.1,0.2[	b	0,1
0.58376125	[0.5,0.9[	e	0,4
0.209403125	[0.2,0.3[	c	0,1
0.09403125	[0.0,0.1[	a	0,1
0.9403125	[0.9,1.0[	f	0,1
0.403125	[0.3,0.5[	d	0.2
0.515625	[0.5,0.9[	e	0,4
0.0390625	[0.0,0.1[	a	0,1
0.390625	[0.3,0.5[	d	0.2

TAB. 2.2: Décodage arithmétique de “0.15633504500”.

Algorithme 19 Décodage arithmétique.

Soit r le nombre en entrée à décoder
Tant que r $\neq 0.0$ **Faire**
  Soit C le symbole dont l'intervalle correspondant dans la table 2.1 contient r
  Afficher C
  Soient a, b les bornes de l'intervalle correspondant à C dans la table 2.1
  $taille \leftarrow b - a$
  $r \leftarrow r - a$
  $r \leftarrow r/taille$
Fin Tant que

Arithmétique entière

Le codage précédent est dépendant de la taille de la mantisse dans la représentation informatique des réels et n'est donc pas forcément très portable. Il n'est donc pas utilisé tel quel avec de l'arithmétique flottante. Les décimales sont plutôt produites une à une, et adapté au nombre de bits du mot machine. Une arithmétique entière plutôt que flottante est alors plus naturelle : au lieu d'un intervalle [0,1[flottant, c'est un intervalle du type [00000,99999] entier qui est utilisé. Ensuite, le codage est le même. Par exemple, une première apparition de 'b' (avec les fréquences de la table 2.1) donne [10000,19999], puis 'e' donnerait [15000,18999].
On peut alors remarquer que dès que le chiffre le plus significatif est identique dans les deux bornes de l'intervalle, il ne change plus. Il peut donc être

affiché en sortie et supprimé des nombres entiers représentant les bornes, ce qui permet de ne manipuler que des nombres relativement petits pour les bornes. Pour le message "bebecafdead", la première occurrence de 'b' donne [10000,19999], on affiche '1' que l'on retire et l'intervalle devient [00000,99999]. Le 'e' donne ensuite [50000,89999], et la suite est dans le tableau 2.3. Le résultat est 156335043840.

Symbole	Borne inférieure	Borne supérieure	Sortie
b	10000	19999	
shift 1	00000	99999	1
e	50000	89999	
b	54000	57999	
shift 5	40000	79999	5
e	60000	75999	
c	63200	64799	
shift 6	32000	47999	6
a	32000	33599	
shift 3	20000	35999	3
f	34400	35999	
shift 3	44000	59999	3
d	48800	51999	
e	50400	51679	
shift 5	04000	16799	5
a	04000	05279	
shift 0	40000	52799	0
d	43840	46399	
			43840

TAB. 2.3: Codage arithmétique entier de "bebecafdead".

Le décodage suit quasiment la même procédure en ne lisant qu'un nombre fixe de chiffres à la fois : à chaque étape, on trouve l'intervalle (et donc le caractère) contenant l'entier courant. Ensuite, si le chiffre de poids fort est identique pour l'entier courant et les bornes, on décale le tout comme dans le tableau 2.4.

En pratique

Ce schéma d'encodage rencontre un problème si les deux chiffres de poids fort ne deviennent pas égaux au fur et à mesure du codage. La taille des entiers qui bornent l'intervalle augmente, mais ne peut augmenter indéfiniment à cause des limitations de toute machine. On est confronté à ce problème si l'on obtient un intervalle de type $[59992, 60007]$. Après quelques itérations supplémentaires, l'intervalle converge vers $[59999, 60000]$, et plus rien n'est affiché !

Décalage	Entier	Borne inférieure	Borne supérieure	Sortie
	15633	$>$ 10000	$<$ 19999	b
1	56335	00000	99999	
	56335	$>$ 50000	$<$ 89999	e
	56335	$>$ 54000	$<$ 57999	b
5	63350	40000	79999	
	63350	$>$ 60000	$<$ 75999	e
	63350	$>$ 63200	$<$ 64799	c
6	33504	32000	47999	
	33504	$>$ 32000	$<$ 33599	a
3	35043	20000	35999	
	35043	$>$ 34400	$<$ 35999	f
3	50438	44000	59999	
	50438	$>$ 48800	$<$ 51999	d
	50438	$>$ 50400	$<$ 51679	e
5	04384	04000	16799	
	04384	$>$ 04000	$<$ 05279	a
0	43840	40000	52799	
	43840	$>$ 43840	$<$ 46399	d

TAB. 2.4: Décodage arithmétique entier de “156335043840”.

Pour pallier ce problème, il faut, outre comparer les chiffres de poids fort, comparer également les suivants si les poids forts diffèrent seulement de 1. Alors, si les suivants sont 9 et 0, il faut décaler ceux-ci et conserver en mémoire que le décalage est au niveau du deuxième chiffre de poids fort. Par exemple, $[59123, 60456]$ est décalé en $[51230, 64569]_1$; l'indice 1 indiquant qu'on a décalé le deuxième chiffre. Ensuite, lorsque les chiffres de poids fort deviennent égaux (après k décalages de 0 et de 9), il faudra afficher ce chiffre suivi de k zéros ou k neufs. En format décimal ou hexadécimal, il faudrait stocker un bit supplémentaire indiquant si l'on a convergé vers le haut ou le bas ; en format binaire, cette information se déduit directement.

Exercice 2.6. *Avec les probabilités* $('a' : 4/10;' b' : 2/10;' c' : 4/10)$*, le codage de la chaîne “bbbbba” est donné ci-dessous :*

Symbole	*Intervalle*	*Décalage*	*Sortie*
b	$[40000; 59999]$		
b	$[48000; 51999]$		
b	$[49600; 50399]$	*shift 0 et 9 :* $[46000; 53999]_1$	
b	$[49200; 50799]_1$	*shift 0 et 9 :* $[42000; 57999]_2$	
b	$[48400; 51599]_2$		
a	$[48400; 49679]_2$	*shift 4 :* $[84000; 96799]$	*499, puis 84000*

Avec les mêmes probabilités, décoder la chaîne 49991680. *Solution page 291.*

2.2.3 Codes adaptatifs

Le codage présenté par arbre de Huffman (avec paramétrage de l'extension de source) ou le codage arithmétique sont des codages statistiques qui fonctionnent sur le modèle de source introduit au chapitre 1, à savoir sur la connaissance a priori de la fréquence d'apparition des caractères dans le fichier. Ces codages nécessitent en outre une table de correspondances (ou un arbre) entre les mots de source et les mots de code afin de pouvoir décompresser. Cette table peut devenir très grande dans le cas d'extensions de sources.

Exercice 2.7. *Soit un alphabet sur 4 caractères ('a','b','c','d'), de loi de probabilité d'apparition (1/2,1/6,1/6,1/6).*

1. *Donner la table de correspondances de l'algorithme de Huffman statique pour cette source.*
2. *En supposant que les caractères a, b, c et d sont écrits en ASCII (8 bits), quel espace mémoire minimal est nécessaire pour stocker cette table ?*
3. *Quel serait l'espace mémoire nécessaire pour stocker la table de correspondances de l'extension de source à 3 caractères ?*
4. *Quelle est la taille du fichier ASCII contenant la séquence "aaa aaa aaa bcd bcd bcd" non compressée ?*
5. *Compresser cette séquence par Huffman statique. Donner la taille totale du fichier compressé. Est-il intéressant d'utiliser une extension de source ?*

Solution page 291.

En pratique, il existe des variantes dynamiques (à la volée) qui permettent de s'affranchir à la fois du pré-calcul des fréquences et des tables de correspondances. Ce sont ces variantes qui sont les plus utilisées dans les utilitaires de compression. Elles utilisent la fréquence des caractères, mais la calculent au fur et à mesure de l'occurrence des symboles.

Algorithme de Huffman dynamique – `pack`

L'algorithme de Huffman dynamique permet de compresser un flot à la volée en faisant une seule lecture de l'entrée ; à la différence de l'algorithme statique d'Huffman, il évite de faire deux parcours d'un fichier d'entrée (un pour le calcul des fréquences, l'autre pour le codage). La table des fréquences est élaborée au fur et à mesure de la lecture du fichier ; ainsi l'arbre de Huffman est modifié à chaque fois qu'on lit un nouveau caractère.
La commande « pack » de Unix implémente cet algorithme dynamique.

Compression dynamique

La compression est décrite par l'algorithme 20. On suppose qu'on doit coder un fichier de symboles binaires, lus à la volée par blocs de k bits (k est souvent un paramètre) ; on appelle donc « caractère » un tel bloc. À l'initialisation, on définit un caractère symbolique (noté @ dans la suite) et codé initialement par un symbole prédéfini (par exemple comme un 257ème caractère virtuel du code ASCII). Lors du codage, à chaque fois que l'on rencontre un nouveau caractère pas encore rencontré, on le code sur la sortie par le code de @ suivi des k bits du nouveau caractère. Le nouveau caractère est alors entré dans l'arbre de Huffman.
Pour construire l'arbre de Huffman et le mettre à jour, on compte le nombre d'occurrences de chaque caractère et le nombre de caractères déjà lus ; on connaît donc, à chaque nouvelle lecture, la fréquence de chaque caractère depuis le début du fichier jusqu'au caractère courant ; les fréquences sont donc calculées dynamiquement.
Après avoir écrit un code (soit celui de @, soit celui d'un caractère déjà rencontré, soit les k bits non compressés d'un nouveau caractère), on incrémente de 1 le nombre d'occurrences du caractère écrit. En prenant en compte les modifications de fréquence, on met à jour l'arbre de Huffman à chaque itération.

Algorithme 20 Algorithme de Huffman dynamique : compression.

Soit $nb(c)$, le nombre d'occurrence d'un caractère c
Initialiser l'arbre de Huffman (AH) avec le caractère @, $nb(@) \leftarrow 1$
Tant que on n'est pas à la fin de la source **Faire**
  Lire un caractère c de la source
  Si c'est la première occurrence de c **Alors**
    $nb(c) \leftarrow 0$
    $nb(@) \leftarrow nb(@) + 1$
    Afficher en sortie le code de @ dans AH suivi de c.
  Sinon
    Afficher le code de c dans AH
  Fin Si
  $nb(c) \leftarrow nb(c) + 1$
  Mettre à jour AH avec les nouvelles fréquences
Fin Tant que

L'arbre existe donc pour la compression (et la décompression) mais n'a pas besoin d'être envoyé au décodeur.
Enfin, il y a plusieurs choix pour le nombre d'occurrences de @ : dans l'algorithme 20 c'est le nombre de caractères distincts (cela permet d'avoir peu de

bits pour @ au début), il est aussi possible de lui attribuer par exemple une valeur constante très proche de zéro (dans ce cas le nombre de bits pour @ évolue comme la profondeur de l'arbre de Huffman, en laissant aux caractères très fréquents les codes les plus courts).

Décompression dynamique

Elle est décrite par l'algorithme 21. À l'initialisation, le décodeur connaît un seul code, celui de @ (par exemple 0). Il lit alors 0 qui est le code associé à @. Il en déduit que les k bits suivants contiennent un nouveau caractère. Il recopie sur sa sortie ces k bits et met à jour l'arbre de Huffman, qui contient déjà @, avec ce nouveau caractère.
Il faut bien noter que le codeur et le décodeur maintiennent chacun leur propre arbre de Huffman, mais utilisent tous les deux le même algorithme pour le mettre à jour à partir des occurrences (fréquences) des caractères déjà lus. Ainsi, les arbres de Huffman calculés séparément par le codeur et le décodeur sont exactement les mêmes.

Algorithme 21 Algorithme de Huffman dynamique : décompression.

Soit $nb(c)$, le nombre d'occurrence d'un caractère c
Initialiser l'arbre de Huffman (AH) avec le caractère @, $nb(@) \leftarrow 1$
Tant que on n'est pas à la fin du message codé **Faire**
  Lire un mot de code c du message (jusqu'à une feuille de AH)
  Si $c = @$ **Alors**
    $nb(@) \leftarrow nb(@) + 1$
    Lire dans c les k bits du message et les afficher en sortie.
    $nb(c) \leftarrow 0$
  Sinon
    Afficher le caractère correspondant à c dans AH
  Fin Si
  $nb(c) \leftarrow nb(c) + 1$
  Mettre à jour AH avec les nouvelles fréquences
Fin Tant que

Puis, le décodeur lit le code suivant et le décode via son arbre de Huffman. S'il s'agit du code de @, il lit les k bits correspondants à un nouveau caractère, les écrit sur la sortie et ajoute le nouveau caractère à son arbre de Huffman (le nouveau caractère est désormais associé à un code). Sinon, il s'agit du code d'un caractère déjà rencontré ; via son arbre de Huffman, il trouve les k bits du caractère associé au code, et les écrit sur la sortie. Il incrémente alors de 1 le nombre d'occurrences du caractère qu'il vient d'écrire (et de @ si c'est un

nouveau caractère) et met à jour l'arbre de Huffman.
Cette méthode dynamique est un peu moins efficace que la méthode statique pour estimer les fréquences. Il est probable que le message codé sera donc un peu plus long. Mais elle évite le stockage de l'arbre et de la table des fréquences, ce qui rend le résultat final souvent plus court. Ceci explique son utilisation en pratique dans des utilitaires courants.

Exercice 2.8. *Nous reprenons la séquence "aaa aaa aaa bcd bcd bcd".*

1. *Quel serait le codage de Huffman dynamique de cette séquence ?*
2. *Quel serait le codage de Huffman dynamique de l'extension à 3 caractères de cette séquence ?*

Solution page 291.

Codage arithmétique adaptatif

L'idée générale du codage adaptatif qui vient d'être développée pour l'algorithme de Huffman, est la suivante :
– à chaque étape le code courant correspond au code statique que l'on aurait obtenu en utilisant les occurrences déjà connues comme fréquences.
– après chaque étape, l'encodeur met à jour sa table d'occurrence avec le nouveau caractère qu'il a reçu et crée un nouveau code statique correspondant. Ceci doit être fait toujours de la même manière afin que le décodeur puisse faire de même à son tour.

Cette idée s'implémente très bien également dans le cadre du codage arithmétique : le code est construit essentiellement comme le code arithmétique, sauf que la distribution de probabilités est calculée au vol par l'encodeur sur les symboles qui ont déjà été traités. Le décodeur peut faire les mêmes calculs et reste ainsi synchronisé. Le codeur arithmétique adaptatif travaille en arithmétique flottante : à la première itération, l'intervalle $[0, 1[$ est divisé en segments de longueurs égales. Chaque fois qu'un symbole est reçu, l'algorithme calcule la nouvelle distribution de probabilités et le segment du symbole reçu est lui-même re-divisé en nouveaux segments. Cependant, ces nouveaux segments sont maintenant de longueurs correspondantes aux probabilités, mises à jour, des symboles. Plus le nombre de symboles reçus est grand, plus la probabilité calculée sera proche de la probabilité réelle. Comme pour Huffman dynamique, il n'y a aucun surcoût dû à l'envoi préliminaire de la table des fréquences et pour des sources variables l'encodeur est capable de s'adapter dynamiquement aux variations de probabilités.
Enfin, outre la compression souvent meilleure par un codage arithmétique, il est à noter que si les implémentations de Huffman statique sont plus rapide que celles du codage arithmétique statique, c'est en général l'inverse dans le cas adaptatif.

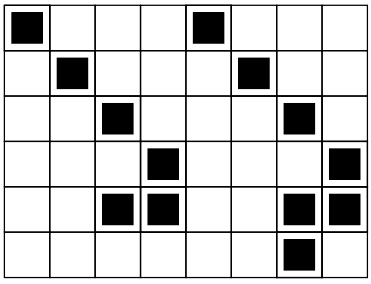
2.3 Heuristiques de réduction d'entropie

En pratique, l'algorithme de Huffman (ou ses variantes) est utilisé en conjonction avec d'autres procédés de codage. Ces autres procédés ne sont pas toujours optimaux en théorie mais ils font des hypothèses raisonnables sur la forme des fichiers à compresser (ce sont des modèles de source) pour diminuer l'entropie ou accepter une destruction d'information supposée sans conséquence pour l'utilisation des données.
Nous donnons ici trois exemples de réduction d'entropie. Le principe est de transformer un message en un autre, par une transformation réversible, de façon à ce que le nouveau message ait une entropie plus faible, et soit donc compressible plus efficacement. Il s'agit donc d'un codage préalable à la compression, chargé de réduire l'entropie.

2.3.1 RLE – Run Length Encoding

Le codage statistique tire parti des caractères apparaissant souvent, mais absolument pas de leur position dans le texte. Si un même caractère apparaît souvent répété plusieurs fois d'affilée, il peut être utile de coder simplement le nombre de fois où il apparaît. Par exemple, pour transmettre une page par fax, le codage statistique codera le 0 par un petit mot de code, mais chaque 0 sera écrit, ce qui le rendra beaucoup moins efficace que le code du fax présenté au chapitre 1, spécialement pour les pages contenant de grandes parties blanches. Le codage RLE est une extension de ce codage fax. Il lit chaque caractère à la suite mais, lorsque au moins 3 caractères identiques successifs sont rencontrés, il affiche plutôt un code spécial de répétition suivi du caractère à répéter et du nombre de répétitions. Par exemple, avec @ comme caractère de répétition, la chaîne "aabbbcdddddd@ee" est codée par "@a2@b3c@d5@@1@e2" (il est nécessaire d'employer une petite astuce si le caractère spécial est rencontré pour rendre le code instantané).

Exercice 2.9. *1. Quel serait le codage RLE de l'image suivante ?*



2. Y-a-t-il un gain ?

3. *Quel serait le codage sur blocs de 5 bits en 16 niveaux de gris ? Et le gain ? Peut-on compresser plus avec ce système ?*
4. *Et en 255 niveaux de gris (avec au plus 16 pixels identiques consécutifs) ?*
5. *Et en colonnes ?*

Solution page 292.

Pour ne pas avoir le problème du caractère spécial de répétition qui se code forcément comme une répétition de taille 1, les modems utilisent une variante de RLE appelée MNP5. L'idée est que lorsque 3 ou plus caractères identiques sont rencontrés, ces trois caractères sont affichés suivis d'un compteur indiquant le nombre d'apparitions supplémentaires du caractère. Il faut bien sûr convenir d'un nombre de bits fixe à attribuer à ce compteur. Si le compteur dépasse son maximum, la suite sera codée par plusieurs blocs des 3 mêmes caractères suivi d'un compteur. Par exemple, la chaîne "aabbbcddddd" est codée par "aabbb0cddd2". Dans ce cas, si une chaîne de n octets contient m répétitions de longueur moyenne L, le gain de compression est $\frac{n-m(L-4)}{n}$ si le compteur est sur un octet. Ce type de codage est donc très utile par exemple pour des images noir et blanc ou des pixels de même couleur sont très souvent accolés. Enfin, il est clair qu'une compression statistique peut être effectuée ultérieurement sur le résultat d'un RLE.

Le code du fax (suite et fin)

Revenons au code du fax, avec lequel nous avons commencé notre exposé de théorie des codes (chapitre 1), pour le compléter et donner son principe de compression en entier.

Rappelons que l'information scannée est un ensemble de pixels noirs ou blancs qui subissent un RLE par lignes de 1728 caractères. Les pixels noirs et blancs sont alternés, le message à coder est donc une suite de nombres entre 1 et 1728. Ils seront codés par leur traduction en bits, selon le principe suivant. Un code uniquement déchiffrable contient le codage de tous les nombres entre 1 et 63, puis quelques nombres supérieurs, de façon à ce que pour tout nombre supérieur à n, il existe un grand nombre m dans la table, tel que $0 \leqslant n - m \leqslant 64$. n répétitions sont alors représentées par deux codes successifs : m répétitions, puis $n - m$ répétitions. Des extraits de tables montrant les nombres codant les pixels blancs sont reproduits sur la figure 2.4. Les nombres compris entre 1 et 64 sont appelés « Terminating White Codes » et les nombres supérieurs « Make Up White Codes ». Les mêmes tables existent pour les noirs, avec différents codes, le tout formant un code uniquement déchiffrable.

Terminating White Codes		Make Up White Codes	
Code	Run	Code	Run
00110101	0	11011	64
000111	1	10010	128
0111	2	010111	192
1000	3	0110111	256
1011	4	00110110	320
...	..	...	...
00110010	61	010011010	1600
00110011	62	011000	1664
00110100	63	010011011	1728

FIG. 2.4: Code du fax.

2.3.2 Move-to-Front

Un autre pré-calcul est possible lorsque que l'on transforme un caractère ASCII en sa valeur entre 0 et 255 : en modifiant à la volée l'ordre de la table. Par exemple, dès qu'un caractère apparaît dans la source, il est d'abord codé par sa valeur, puis il passe en début de liste, et sera dorénavant codé par 0, tous les autres caractères étant décalés d'une unité. Ce « Move-to-front » permet d'avoir plus de codes proches de 0 que de 255. Ainsi, l'entropie est modifiée. Par exemple, la chaîne "aaaaffff" peut être modélisée par une source d'entropie 1, si la table est (a,b,c,d,e,f). Elle est alors codée par "00005555". L'entropie du code (considéré lui-même comme source) est 1. C'est aussi l'entropie de la source. Mais le code d'un Move-to-front sera "00005000", d'entropie $H = 7/8\log_2(8/7)+log_2(8)/8 \approx 0.55$. Le code lui-même est alors compressible, c'est ce qu'on appelle la réduction d'entropie. Dans ces conditions, on peut coder beaucoup plus efficacement le message obtenu, et donc le message de départ. Une variante de cette méthode peut être de déplacer le caractère dans la liste d'un nombre fixe de caractères vers le début, au lieu de le placer en première position. Ainsi, les caractères les plus fréquents vont se retrouver vers le début de la liste. Cette idée préfigure les codages adaptatifs et les codages par dictionnaires des sections suivantes.

Exercice 2.10. *Soit A l'alphabet sur 8 symboles suivant : A = ('a', 'b', 'c', 'd', 'm', 'n', 'o', 'p').*

1. *On associe à chaque symbole un numéro entre 0 et 7, selon leur position alphabétique. Quels nombres représentent "abcddcbamnopponm" et "abcdmnopabcdmnop" ?*

2. *Codez les deux chaînes précédentes en utilisant la technique « Move-to-Front ». Que constatez-vous ?*
3. *Combien de bits sont nécessaires pour coder les deux premiers nombres, par Huffman par exemple ?*
4. *Combien de bits sont nécessaires pour coder les deux nombres obtenus après « Move-to-Front » ? Comparer les entropies.*
5. *Que donne Huffman étendu à deux caractères sur le dernier nombre ? Quelle est alors la taille de la table des fréquences ?*
6. *Comment pourrait-on qualifier l'algorithme composé d'un « Move-to-Front » suivi d'un code statistique ?*
7. *Le « Move-ahead-k » est une variation du « Move-to-Front » où le caractère est seulement avancé de k positions au lieu du sommet de la pile. Encoder les deux chaînes précédentes avec $k = 1$ puis $k = 2$ et comparer les entropies.*

Solution page 292.

2.3.3 BWT : Transformation de Burrows-Wheeler

L'idée de l'heuristique de Michael Burrows et David Wheeler est de trier les caractères d'une chaîne afin que le Move-to-front et le RLE soient les plus efficaces possible. Le problème est bien sûr qu'il est impossible de retrouver la chaîne initiale à partir d'un tri de cette chaîne ! L'astuce est donc de trier la chaîne, mais d'envoyer plutôt une chaîne intermédiaire, de meilleure entropie que la chaîne initiale, et qui permette cependant de retrouver la chaîne initiale. Prenons par exemple, la chaîne "COMPRESSE". Il s'agit de créer tous les décalages possibles comme sur la figure 2.5, puis de trier les lignes dans l'ordre alphabétique et lexicographique. La première colonne **F** de la nouvelle matrice des décalages est donc la chaîne triée de toutes les lettres du mot source. La dernière colonne s'appelle **L**. Seules la première et la dernière colonne sont écrites, car elles sont seules importantes pour le code. Sur la figure et dans le texte nous mettons des indices pour les caractères qui se répètent ; ces indices permettent de simplifier la vision de la transformation mais n'interviennent pas dans l'algorithme de décodage : en effet l'ordre des caractères est forcément conservé entre **L** et **F**.

Bien sûr, pour calculer **F** (la première colonne) et **L** (la dernière colonne), il n'est pas nécessaire de stocker toute la matrice des décalages, un simple pointeur se déplaçant dans la chaîne est suffisant. Cette première phase est donc assez simple. Mais si seule la colonne **F** est envoyée, comment effectuer la réciproque ?

												F		**L**
D_0	C	O	M	P	R	E_1	S_1	S_2	E_2		D_0	C		E_2
D_1	O	M	P	R	E	S	S	E	C		D_8	E_2		S_2
D_2	M	P	R	E	S	S	E	C	O		D_5	E_1		R
D_3	P	R	E	S	S	E	C	O	M	tri	D_2	M		O
D_4	R	E	S	S	E	C	O	M	P	$\longrightarrow$	D_1	O	...	C
D_5	E_1	S	S	E	C	O	M	P	R		D_3	P		M
D_6	S_1	S	E	C	O	M	P	R	E_1		D_4	R		P
D_7	S_2	E	C	O	M	P	R	E	S_1		D_7	S_2		S_1
D_8	E_2	C	O	M	P	R	E	S	S_2		D_6	S_1		E_1

FIG. 2.5: BWT sur COMPRESSE.

La solution est alors d'envoyer la chaîne **L** au lieu de la chaîne **F** : si **L** n'est pas triée, elle est cependant issue du tri d'une chaîne quasiment semblable, simplement décalée d'une lettre. On peut donc espérer qu'elle conserve des propriétés issues du tri et que l'entropie en sera réduite. La magie du décodage vient de ce que la connaissance de cette chaîne, conjuguée à celle de l'**index primaire** (le numéro de ligne contenant le premier caractère de la chaîne initiale, sur l'exemple 2.5, c'est 4 – en numérotant de 0 à 8) permet de récupérer la chaîne initiale. Il suffit de trier L pour récupérer **F**. Ensuite, on calcule un vecteur de transformation **H** contenant la correspondance des indices entre **L** et **F**, c'est-à-dire l'indice dans **L** de chaque caractère pris dans l'ordre trié de **F**. Pour l'exemple 2.5, cela donne $H = \{4, 0, 8, 5, 3, 6, 2, 1, 7\}$, car C est en position 4, l'index primaire, dans **L**, puis E_2 est en position 0 dans **L**, puis E_1 est en position 8, etc. Ensuite, il faut se rendre compte qu'en plaçant **L** avant **F** en colonnes, dans chaque ligne deux lettres se suivant doivent se suivre dans la chaîne initiale : en effet, par décalage, la dernière lettre devient la première et la première devient la deuxième. Ceci se traduit également par le fait que pour tout j, $L[H[j]] = F[j]$. Il ne reste plus alors qu'à suivre cet enchaînement de lettres deux à deux pour retrouver la chaîne initiale, comme dans l'algorithme 22.

Algorithme 22 Réciproque de la BWT.

On dispose de la chaîne **L** et de l'index primaire *index*.
F ← tri de **L** ;
Calculer le vecteur de transformation **H** tel que $L[H[j]] = F[j]$ pour tout j.
Pour i de 0 à Taille de **L** **Faire**
  Afficher **L**[*index*]
  index ← **H**[*index*]
Fin Pour

Exercice 2.11 (Analyse du code BWT).

1. *La dernière colonne* **L** *de la matrice triée de la transformation BWT contient des concentrations de caractères identiques, c'est pourquoi* **L** *se compresse bien. Toutefois, la première colonne* **F** *se compresserait encore mieux puisqu'elle est totalement triée. Pourquoi sélectionner* **L** *plutôt que* **F** *comme code ?*
2. *Soit la chaîne S="sssssssssh". Calculer* **L** *et sa compression Move-to-front.*
3. *Implémentation pratique : BWT est efficace sur de longues chaînes S de taille n. En pratique il est donc impensable de stocker toute la matrice $n \times n$ des permutations. En fait il faut seulement trier ces permutations et pour les trier, il suffit d'être capable de comparer deux permutations.*
   (a) *Donner un indice qui soit capable de désigner et différencier les permutations de la chaîne de départ.*
   (b) *Construire un algorithme* `comparer_permutations`*, calculant une fonction booléenne qui a pour entrée une chaîne S et deux entiers i et j. Cet algorithme doit décider si la chaîne décalée i fois est avant la chaîne décalée j fois, dans l'ordre obtenu en triant toutes les permutations par ordre alphabétique.*
   (c) *Conclure sur l'espace mémoire nécessaire pour calculer la BWT.*
   (d) *Comment calculer* **L** *et l'index primaire à partir du tableau des permutations ?*

Solution page 293.

En sortie de la transformation BWT, on obtient donc en général une chaîne d'entropie plus faible, bien adaptée à un Move-to-front suivi d'un RLE. L'utilitaire de compression `bzip2` détaillé sur la figure 2.6 utilise cette suite de réductions d'entropie avant d'effectuer un codage de Huffman. Nous verrons section 2.4.2 que cette technique est parmi les plus efficaces actuellement.

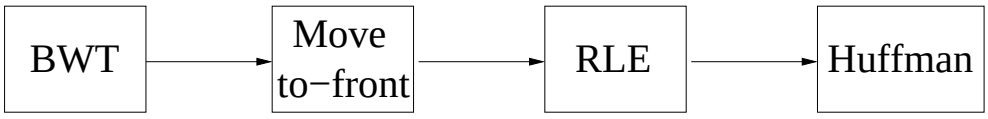


FIG. 2.6: bzip2.

2.4 Codes compresseurs usuels

Comme nous venons de le voir pour L'utilitaire de compression `bzip2` les implémentations usuelles combinent plusieurs algorithmes de compression. Cela permet de tirer parti des avantages de chacun et de minimiser les cas de mauvaise compression. Parmi les techniques les plus répandues, les techniques par dictionnaire de Lempel et Ziv, permettent de compter des groupes entiers de caractères.

2.4.1 Algorithme de Lempel-Ziv et variantes `gzip`

L'algorithme inventé par Lempel et Ziv est un algorithme de compression par dictionnaire (ou encore par substitution de facteurs) ; il consiste à remplacer une séquence de caractères (appelée facteur) par un code plus court qui est l'indice de ce facteur dans un dictionnaire. L'idée est de faire de la réduction d'entropie, non plus par caractère, mais par mots entiers. Par exemple, dans la phrase « un gars bon est un bon gars », on peut associer dynamiquement des numéros aux mots déjà rencontrés, et le codage devient alors : "un,gars,bon,est,1,3,2". Il est à noter que, comme pour les algorithmes dynamiques, les méthodes par dictionnaire ne nécessitent qu'une seule lecture du fichier. Il existe plusieurs variantes de cet algorithme ; parmi celles-ci, LZ77 et LZ78 sont libres de droits.

LZ77. (publié par Lempel et Ziv en 1977) est le premier algorithme de type dictionnaire. Alternative efficace aux algorithmes de Huffman, il a relancé les recherches en compression. LZ77 est basé sur une fenêtre qui coulisse sur le texte de gauche à droite. Cette fenêtre est divisée en deux parties : la première partie constitue le dictionnaire ; la seconde partie (tampon de lecture) rencontre le texte en premier. Initialement, la fenêtre est située de façon à ce que le tampon de lecture soit positionné sur le texte, et que le dictionnaire n'y soit pas. À chaque itération, l'algorithme cherche dans le dictionnaire le plus long facteur qui se répète au début du tampon de lecture ; ce facteur est codé par le triplet (i, j, c) où :

- i est la distance entre le début du tampon et la position de la répétition dans le dictionnaire ;
- j est la longueur de la répétition ;
- c est le premier caractère du tampon différent du caractère correspondant dans le dictionnaire.

Après avoir codé cette répétition, la fenêtre coulisse de $j + 1$ caractères vers la droite. Dans le cas où aucune répétition n'est trouvée dans le dictionnaire, le caractère c qui a provoqué la différence est codé alors $(0, 0, c)$.

Exercice 2.12 (Codage d'une répétition par LZ77).

1. *Coder la séquence "abcdefabcdefabcdefabcdef" avec LZ77.*
2. *Coder cette séquence avec LZ77 et une taille de fenêtre de recherche tenant sur 3 bits.*

Solution page 294.

Une des implémentations les plus connues de LZ77, est la bibliothèque LZMA (pour Lempel-Ziv-Markov chain-Algorithm) qui combine donc une compression LZ77 avec un dictionnaire de taille variable (adaptatif) suivi par un codage arithmétique entier.

LZ78. est une amélioration qui consiste à remplacer la fenêtre coulissante par un pointeur qui suit le texte et un dictionnaire indépendant, dans lequel on cherche les facteurs situés au niveau du pointeur. Supposons que, lors du codage, on lit la chaîne sc où s est une chaîne qui est à l'index n dans le dictionnaire et c est un caractère tel que la chaîne sc n'est pas dans le dictionnaire. On écrit alors sur la sortie le couple (n, c). Puis, le caractère c concaténé au facteur numéro n nous donne un nouveau facteur qui est ajouté au dictionnaire pour être utilisé comme référence dans la suite du texte. Seules deux informations sont codées au lieu de trois avec LZ77.

LZW. (Lempel-Ziv-Welch) est une autre variante de Lempel-Ziv proposée par Terry Welch. LZW a été brevetée par Unisys. Il consiste à ne coder que l'indice n dans le dictionnaire ; de plus le fichier est lu bit par bit. Enfin, il est nécessaire d'avoir un dictionnaire initial (par exemple, la table ASCII). La compression, décrite par l'algorithme 23, est alors immédiate : on remplace chaque groupe de caractère déjà connu par son code et on ajoute au dictionnaire un nouveau groupe formé par ce groupe et le prochain caractère.

Algorithme 23 LZW : compression.

```
Soit chaine ← ∅ la chaîne courante
Tant que on n'est pas à la fin de la source Faire
    Lire un caractère c de la source
    Si chaine|c est dans le dictionnaire Alors
        chaine ← chaine|c
    Sinon
        Afficher le code de chaîne
        Ajouter chaine|c au Dictionnaire
        chaine ← c
    Fin Si
Fin Tant que
```

La décompression est un plus complexe, car il faut traiter un cas particulier où la même chaîne est reproduite deux fois de suite. L'algorithme 24 décrit ce fonctionnement : on part du même dictionnaire initial que l'on augmente au fur et à mesure de la même manière que pour la compression, mais avec un décalage car il faut connaitre le prochain caractère. Ce décalage induit un problème si la même chaîne est reproduite deux fois de suite. En effet, dans ce cas le nouveau code n'est pas encore connu à la décompression. Néanmoins, comme c'est le seul cas où l'on recontre ce type de problème, on sait que c'est le même début de chaîne qui se présente et l'on peut donc deviner la valeur du groupe.

Algorithme 24 LZW : décompression.

Lire un octet *prec* du message codé
Tant que on n'est pas à la fin du message codé **Faire**
  Lire un octet *cour* du message codé
  Si *cour* est dans le dictionnaire **Alors**
    chaine ← la traduction de *cour* dans le dictionnaire
  Sinon
    chaine ← la traduction de *prec* dans le dictionnaire
    chaine ← *chaine*|*c*
  Fin Si
  Afficher *chaine*
  Soit *c* le premier caractère de *chaine*
  Soit *mot* la traduction de *prec* dans le dictionnaire
  Ajouter *mot*|*c* au dictionnaire
  prec ← *cour*
Fin Tant que

L'exercice suivant illustre ce principe.

Exercice 2.13. *En utilisant la table ASCII hexadécimale (7bits) ci-dessous comme dictionnaire initial, compresser et décompresser la chaîne "BLEBLBLBA" avec LZW.*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Décompresser le résultat sans le traitement du cas particulier, que constatez-vous ? *Solution page 294.*

Enfin, plusieurs variantes ont été proposées, par exemple LZMW où 'chaîne + prochain mot' est ajouté au dictionnaire au lieu de seulement 'chaîne + prochain caractère' ou encore LZAP dans laquelle tous les préfixes de 'chaîne + prochain mot' sont ajoutés au dictionnaire.

La commande `compress` de Unix implémente LZ77. L'utilitaire `gzip` est basé sur une variante ; il utilise deux arbres de Huffman dynamiques : un pour les chaînes de caractères, l'autre pour les distances entre occurrences. Les distances entre deux occurrences d'une chaîne sont bornées : lorsque la distance devient trop grande, on réinitialise la compression en redémarrant une compression à partir du caractère courant, indépendamment de la compression déjà effectuée jusqu'à ce caractère.
Cette particularité fait qu'il n'y a pas de différence structurelle entre un seul fichier compressé par `gzip` et une séquence de fichiers compressés. Ainsi, soient `f1.gz` et `f2.gz` les résultats associés à la compression de deux fichiers sources `f1` et `f2`, que l'on concatène dans un fichier `f3.gz`. En décompressant avec `gunzip f3.gz`, on obtient en résultat un fichier dont le contenu est identique à la concaténation de `f1` et `f2`.

Exercice 2.14 (gzip).

1. *Encoder la chaîne "abbarbarbbabbarbaa" par LZ77.*
2. *Découper le codage précédent en trois chaînes (distances, longueurs et caractères), et appliquer* `gzip`*, le codage du @ de Huffman dynamique étant* fixé *à un seul bit,* `1`*. En outre, dans* `gzip`*, le premier @ n'est pas omis.*
3. *On suppose que la chaîne précédente est en deux morceaux, "abbarbarb" et "babbarbaa", dans deux fichiers distincts "f1" et "f2". Donner le contenu de "f1.gz" et "f2.gz", les compressions de "f1" et "f2" par* `gzip`*.*
4. *On concatène les deux fichiers "f1.gz" et "f2.gz" dans un fichier "f3.gz" (par exemple par la commande unix* `cat f1.gz f2.gz > f3.gz`*, puis l'on décompresse le fichier "f3.gz" par* `gunzip`*. Quel est le résultat ? En déduire l'utilité de ne pas omettre le premier @.*
5. *Comment* `gzip` *peut-il être paramétré par un niveau de compression (par ex.* `gzip -1 xxx.txt'` *ou* `gzip -9 xxx.txt'`*) ?*

Solution page 294.

2.4.2 Comparaison des algorithmes de compression

Nous comparons dans le tableau 2.5 les taux de compression et les vitesses de codage et décodage des utilitaires classiques sous unix/linux sur l'exemple d'un fichier de courriels (contenant donc texte, images, fichiers binaires).

Algorithme	Fichier compressé	Taux	codage	décodage
7-Zip-4.42 (LZMA+ ?)	5.96 Mo	62.57%	23.93s	6.27s
RAR-3.51 (?)	6.20 Mo	61.07%	14.51s	0.46s
rzip-2.1 -9 (LZ77+Go)	6.20 Mo	61.09%	9.26s	2.94s
ppmd-9.1 (Prédictif)	6.26 Mo	60.71%	11.26s	12.57s
bzip2-1.0.3 (BWT)	6.69 Mo	57.96%	7.41s	3.16s
gzip-1.3.5 -9 (LZ77)	7.68 Mo	51.77%	2.00s	0.34s
gzip-1.3.5 -2 (LZ77)	8.28 Mo	47.99%	1.14s	0.34s
WinZip-9.0 (LZW+ ?)	7.72 Mo	51.55%	5s	5s
compress (LZW)	9.34 Mo	41.31%	1.11s	0.32s
lzop-1.01 -9 (LZW+ ?)	9.35 Mo	41.32%	6.73s	0.09s
lzop-1.01 -2 (LZW+ ?)	10.74 Mo	32.54%	0.45s	0.09s
pack (Huffman)	11.11 Mo	30.21%	0.33s	0.27s

Tab. 2.5: Comparaison de la compression d'un fichier de courriels (15.92 Mo de texte, images, exécutables, etc.) par différents algorithmes, sur un PIV 1.5GHz.

Les programmes `gzip` et `compress` utilisent des variantes de Lempel-Ziv et sont décrits à la section précédente. `bzip2` utilise des réductions d'entropies comme la BWT avant de faire un codage entropique et est décrit section 2.3.3. `7-Zip` utilise la variante LZMA de LZ77, associée à un codage arithmétique, et un certain nombre d'heuristiques supplémentaires suivant le type de fichier en entrée. `WinZip` utilise également plusieurs variantes suivant le type de fichier pour finir par un codage LZW. `rzip` est spécialisé pour les gros fichiers et utilise donc LZ77 avec une très large fenêtre. `ppmd` utilise un modèle probabiliste de Markov pour prévoir le prochain caractère grâce à la connaissance des caractères qui le précèdent immédiatement. `pack`, enfin, implémente un codage de Huffman simple.

En conclusion le comportement pratique reflète la théorie : LZ77 compresse mieux que LZW, mais plus lentement. Les algorithmes par dictionnaires compressent mieux que les codages entropiques simples, mais moins bien qu'un codage entropique avec une bonne heuristique de réduction d'entropie. Les logiciels généraux essayent en général plusieurs méthodes pour trouver la meilleure compression. Si le taux de compression est donc amélioré, le temps de compression en souffre fortement en général.

2.4.3 Formats GIF et PNG pour la compression d'images

Les formats de données compacts usuels tiennent compte de la nature des données à coder. Ils sont différents s'il s'agit d'un son, d'une image ou d'un texte. Parmi les formats d'images, le format GIF (*Graphic Interchange Format*) est un format de fichier graphique bitmap proposé par la société Compuserve. C'est un format de compression pour une image pixellisée, c'est-à-dire décrite comme une suite de points (pixels) contenus dans un tableau ; chaque pixel a une valeur qui décrit sa couleur.

Le principe de compression est en deux étapes. Tout d'abord, les couleurs pour les pixels (initialement, il y a 16,8 millions de couleur codées sur 24 bits RGB) sont limitées à une palette contenant de 2 à 256 couleurs (2, 4, 8, 16, 32, 64, 128 ou 256 qui est le défaut). La couleur de chaque pixel est donc approchée par la couleur la plus proche figurant dans la palette. Tout en gardant un nombre important de couleurs différentes avec une palette de 256, ceci permet d'avoir un facteur 3 de compression. Puis, la séquence de couleurs des pixels est compressée par l'algorithme de compression dynamique LZW (Lempel-Ziv-Welch). Il existe deux versions de ce format de fichier développées respectivement en 1987 et 1989 :

– GIF 87a permet un affichage progressif (par entrelacement) et la possibilité d'avoir des images animées (les GIFs animés) en stockant plusieurs images au sein du même fichier.
– GIF 89a permet en plus de définir une couleur transparente dans la palette et de préciser le délai pour les animations.

Comme l'algorithme de décompression LZW a été breveté par Unisys, tous les éditeurs de logiciel manipulant des images GIF auraient pu payer une redevance à Unisys. C'est une des raisons pour lesquelles le format PNG est de plus en plus plébiscité, au détriment du format GIF. Le format PNG est un format analogue ; mais l'algorithme de compression utilisé est LZ77.

2.5 La compression avec perte

2.5.1 Dégradation de l'information

Nous avons vu que les compresseurs sans perte ne permettent pas de compresser au-delà du seuil de l'entropie, et que les heuristiques de réduction de l'entropie permettaient d'abaisser ce seuil en changeant le modèle de la source. Mais aucune de ces méthodes ne tenait compte du type de fichier à compresser, et de son utilisation.

Prenons l'exemple d'un image numérique. Les compresseurs sans perte transmettront le fichier sans utiliser l'information qu'il s'agit d'un image, et le fichier

pourra toujours être restitué dans son intégralité. Mais il peut exister des formats beaucoup plus efficaces pour lesquels, si une partie de l'information est perdue, la différence ne soit jamais *visible*. Les fichiers ne pourront donc pas être restitués dans leur état initial, mais leur état permettra de visualiser des images d'aussi bonne qualité. Ce qui veut dire que la qualité qui était codée dans le fichier de départ était superflue, puisqu'elle n'induit aucune différence visuelle.

Les compresseurs avec perte sont donc spécifiques du type de données à transmettre ou à stocker (images, son et vidéo pour la plupart), et utilisent cette information en codant le résultat visuel du fichier plutôt que le fichier lui-même.

Ce type de compression est obligatoire pour les formats vidéo, par exemple, où la taille des données et les calculs sont très importants.

Les compresseurs avec perte vont permettre de dépasser le seuil de l'entropie et obtenir des taux de compression très importants. Ils tiennent compte non plus seulement de l'information brute, mais aussi de la manière dont elle est perçu par des êtres humains, ce qui nécessite une modélisation de la persistance rétinienne ou auditive par exemple.

On pourra mesurer l'efficacité de ces compresseurs avec le taux de compression, mais la qualité des images produites n'est évaluée que grâce à l'expérience des utilisateurs. Nous présentons donc les formats standards, largement éprouvés.

En outre, pour les informations audiovisuelles, un autre critère important viendra se greffer au taux de compression : la rapidité de la décompression. En particulier pour les sons ou les vidéos, la visualisation nécessite des calculs extrêmement rapides pour lire les fichiers compressés. La complexité algorithmique des décompresseurs est donc une contrainte aussi importante que la taille des messages compressés lorsqu'on traite des informations audiovisuelles.

2.5.2 Transformation des informations audiovisuelles

Les formats compacts de stockage d'images, de sons ou de vidéo utilisent toujours des sources étendues. Ils codent non pas pixel par pixel, mais prennent en compte un pixel et son entourage, pour observer des paramètres globaux, comme l'apparition des contours, les régularités, ou l'intensité des variations. Ces données seront traitées sous la forme de signaux pseudo-périodiques. Par exemple, si un morceau d'une image est un dégradé doux, la fréquence du signal sera faible, tandis que la fréquence d'une partie dont les couleurs, les textures varient beaucoup sera élevée. Cela permettra un codage très efficace des zones de basse fréquence. En outre, une suppression pure et simple des zones de très haute fréquence sera souvent invisible ou inaudible, donc parfaitement acceptable.

2.5.3 Le format JPEG

Le codage au format JPEG (pour *Joint Photographic Experts Group*) compresse des images fixes avec perte d'information. L'algorithme de codage est complexe, et se déroule en plusieurs étapes. Le principe de base est que les couleurs des pixels voisins dans une image diffèrent peu en général. De plus, une image est un signal : plutôt que les valeurs des pixels, on calcule les fréquences (transformée de Fourier DFT ou transformée en cosinus discrète DCT dans le cas de JPEG).

En effet, on peut considérer qu'une image est une matrice de couleurs des pixels ; et il se trouve que les images numériques sont principalement composées de basses fréquences DCT quand on regarde la luminance des couleurs (dans le codage d'une image en mode luminance / chrominance, YUV par exemple) plutôt qu'en mode rouge / vert / bleu (RGB). Il existe plusieurs formules pour passer du mode RGB au mode YUV selon les modèles physiques de la lumière. Une formule classique pour trouver la luminance est celle-ci : $Y = 0.299 * R + 0.587 * G + 0.114 * B$.

FIG. 2.7: Image 8×8 en rouge, vert et bleu sur fond blanc (ici en niveaux de gris).

L'idée est donc d'appliquer la DCT de la section 1.4.2 sur la matrice des luminances. Or, la DCT va transformer cette matrice en une matrice contenant les hautes fréquences en haut à droite et les basses fréquences en bas à gauche. Ainsi, la luminance de l'image 2.7 est transformée par DCT de $L \rightarrow D$ de la façon suivante :

$$L = \begin{bmatrix} 76 & 76 & 76 & 255 & 255 & 255 & 255 & 255 \\ 255 & 76 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 76 & 255 & 150 & 150 & 255 & 255 & 255 \\ 255 & 76 & 255 & 150 & 255 & 150 & 255 & 255 \\ 255 & 255 & 255 & 150 & 255 & 150 & 29 & 29 \\ 255 & 255 & 255 & 150 & 150 & 255 & 29 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 29 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 29 & 29 \end{bmatrix}$$

$$D = \begin{bmatrix} 1631 & 71 & -83 & 49 & 94 & -36 & 201 & 32 \\ 6 & -348 & 49 & 2 & 26 & 185 & -60 & 86 \\ 39 & -81 & -123 & 27 & 28 & 0 & -43 & -130 \\ -49 & 3 & -10 & -25 & 19 & -126 & -33 & -92 \\ -101 & 43 & -111 & 55 & -4 & -3 & -120 & 50 \\ 25 & -166 & 62 & -50 & 50 & -2 & -9 & -7 \\ -64 & -34 & 54 & 11 & -69 & 0 & 26 & -54 \\ -48 & 73 & -48 & 28 & -27 & -33 & -23 & -29 \end{bmatrix}$$

Enfin, l'œil humain donne plus d'importance à la luminosité qu'aux couleurs. En ne gardant que les premiers termes de la décomposition en luminance DCT (les plus importants), on perd un peu d'information mais l'image reste visible. Cette opération est appelée quantification et c'est la seule étape de tout le codage JPEG qui soit avec perte d'information. L'opération consiste à arrondir la valeur $C(i,j) = \frac{DCT(i,j)}{Q(i,j)}$ où Q est une matrice de quantification

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

$$C = \begin{bmatrix} 102 & 6 & -8 & 3 & 4 & -1 & 4 & 1 \\ 1 & -29 & 4 & 0 & 1 & 3 & -1 & 2 \\ 3 & -6 & -8 & 1 & 1 & 0 & -1 & -2 \\ -4 & 0 & 0 & -1 & 0 & -1 & 0 & -1 \\ -6 & 2 & -3 & 1 & 0 & 0 & -1 & 1 \\ 1 & -5 & 1 & -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & -1 & 0 & 0 & -1 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

On voit sur l'exemple qu'après quantification, beaucoup de valeurs sont identiques et proches de zéro. Le choix de la matrice de quantification respectivement avec valeurs proches de 1 ou loin de 1 permet d'augmenter ou de diminuer le niveau de détails.

L'image est finalement codée comme une suite de nombres (une valeur quantifiée de DCT suivie du nombre de pixels ayant cette valeur qui sont consécutifs selon le balayage en zigzag la figure 2.8). Les faibles valeurs obtenues permettent d'espérer que l'entropie obtenue après ce RLE a été considérablement réduite. L'algorithme JPEG se termine alors par un codage statistique, par exemple Huffman.

Enfin, il est à noter que le standard JPEG, ou encore DCT-JPEG, tend à être remplacé par le nouveau standard JPEG-2000 où la transformation en cosinus et la quantification sont remplacées par une transformée en ondelettes dans laquelle ne sont conservés que certains niveaux. Le nombre de niveaux ou bandes conservés influe directement sur le taux de compression désiré.

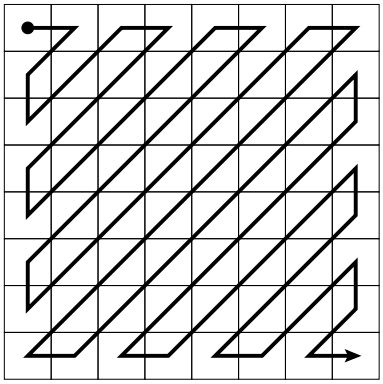


FIG. 2.8: Balayage en zigzag de JPEG.

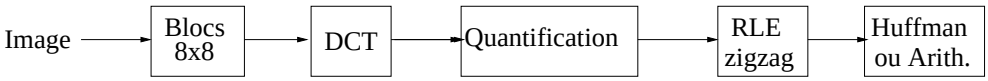


FIG. 2.9: Compression JPEG.

2.5.4 Le format MPEG

Le format MPEG (pour *Motion Picture Experts Group*) définit la compression d'images animées. L'algorithme de codage utilise JPEG pour coder une image mais prend en compte le fait que deux images consécutives dans une séquence vidéo sont très voisines. Une des particularités des normes MPEG est de chercher à compenser le mouvement (un zoom par exemple etc..) d'une image à la suivante. Une sortie MPEG-1 contient 4 sortes d'images : des images au format I (JPEG), des images P codées par différences avec l'image précédente (par compensation du mouvement deux images consécutives sont très voisines), des images bidirectionnelles B codées par différence avec l'image précédente et la suivante (en cas de différences moins importantes avec une image future qu'avec une image précédente), enfin des images basse résolution utilisées pour l'avance rapide sur un magnétoscope. En pratique, avec les images P et B, il suffit d'une nouvelle image complète I toutes les 10 à 15 images.

Un film vidéo étant composé d'images et de son, il faut également compresser le son. Le groupe MPEG a défini trois formats : les MPEG-1 Audio Layer I, II et III, le troisième étant le fameux MP3. Le MPEG-1 est la combinaison de compressions d'images et de son, associées à une synchronisation par marquage temporel et une horloge de référence du système comme indiqué sur la figure 2.10.

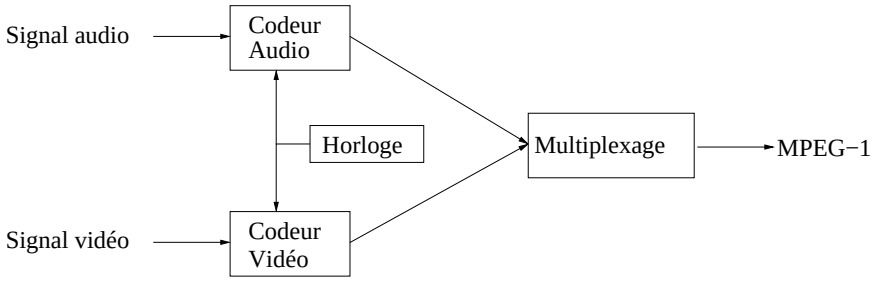


FIG. 2.10: Compression MPEG-1.

Depuis 1992, le format MPEG à évolué passant de MPEG-1 (320×240 pixels et un débit de 1.5 Mbits/s), à MPEG-2 en 1996 (avec quatre résolutions de 352×288 à 1920×1152 pixels plus un flux de sous-titres et de langues multiplexés, celui-ci est l'actuel format des DVD) ou encore à MPEG-4 en 1999 (format des vidéos conférences et du DivX où la vidéo est devenue une scène orientée objet).
Enfin, le son est décomposé en trois couches, MP1, MP2 et MP3, suivant les taux de compression (et donc les pertes) désirés. En pratique, les taux de compression par rapport à l'analogique sont d'un facteur 4 pour le MP1, de 6 à 8 pour le MP2, utilisé par exemple dans les vidéos CD et de 10 à 12 pour le MP3.
Nous donnons une idée de la transformation MP3 sur la figure 2.11 : le son analogique est d'abord filtré et modifié par des transformées de Fourier (avec perte) et simultanément en cosinus (DCT).

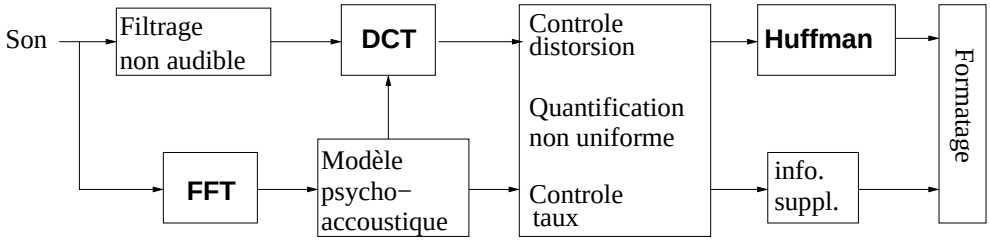


FIG. 2.11: Compression du son MP3.

Ensuite, la combinaison de ces deux transformations permet de faire une quantification et de contrôler le résultat obtenu pour ne conserver que la partie audible principale. Au final, une compression statistique est bien sûr appliquée, pour réduire le fichier au niveau de son entropie finale.

Chapitre 3

Cryptologie

Il s'agit maintenant d'étendre le codage au cas où on souhaite cacher une information. On doit supposer que les routes ne sont pas contrôlées, et que le message transmis est lisible par tous. Mais seul le destinataire doit pouvoir reconstituer le message d'origine. Les applications sont évidemment très nombreuses, qu'elles soient commerciales ou militaires.
Littéralement « science du secret », la cryptologie comporte deux composantes complémentaires que sont la *cryptographie* (qui consiste à étudier et concevoir des procédés de chiffrement des informations) et la *cryptanalyse* (qui analyse des textes chiffrés pour retrouver les informations dissimulées). À noter qu'il ne faut pas confondre la cryptographie avec la *stéganographie* qui, elle, consiste à dissimuler l'existence même de l'information secrète (par l'utilisation d'une encre sympathique par exemple). Nous allons systématiser les approches vues au chapitre 1, et pour cela, reprendre depuis le début les principes et protocoles associés à la cryptologie.

3.1 Principes généraux et terminologie

3.1.1 Terminologie

L'objectif fondamental de la cryptographie est de permettre à deux personnes, appelées traditionnellement *Alice* et *Bob* de communiquer à travers un canal peu sûr de telle sorte qu'un opposant, *Oscar*, qui a accès aux informations qui circulent sur le canal de communication, ne puisse pas comprendre ce qui est échangé. Le canal peut être par exemple une ligne téléphonique ou tout autre réseau de communication.
L'information qu'Alice souhaite transmettre à Bob est appelée *texte clair*. Il peut s'agir d'un texte en français, d'une donnée numérique ou de n'importe quoi d'autre, de structure arbitraire. Le codage, c'est-à-dire le processus de

transformation d'un message M de manière à le rendre incompréhensible est appelé *chiffrement*. On génère ainsi un *message chiffré* C obtenu à partir d'une *fonction de chiffrement* E par $C = E(M)$. Le processus de reconstruction du message clair à partir du message chiffré est appelé *déchiffrement* et utilise une *fonction de déchiffrement* D. Il est donc requis que : $D(C) = D(E(M)) = M$. Donc E est injective (ou encore un chiffré possède au plus un message source associé) et D est surjective (un chiffré possède toujours un message source associé).

Un *algorithme cryptographique* est un algorithme qui calcule la valeur des fonctions mathématiques utilisées pour le chiffrement et le déchiffrement. En pratique, la sécurité des messages est toujours assurée par ce qu'on appelle des *clefs*. Ce sont des paramètres des fonctions E et D, qu'on notera K_e et K_d, qui peuvent prendre l'une des valeurs d'un ensemble appelé *espace des clefs*. On aboutit ainsi à la relation fondamentale de la cryptographie (illustrée dans la figure 3.1) :

$$\begin{cases} E_{K_e}(M) = C \\ D_{K_d}(C) = M \end{cases} \tag{3.1}$$

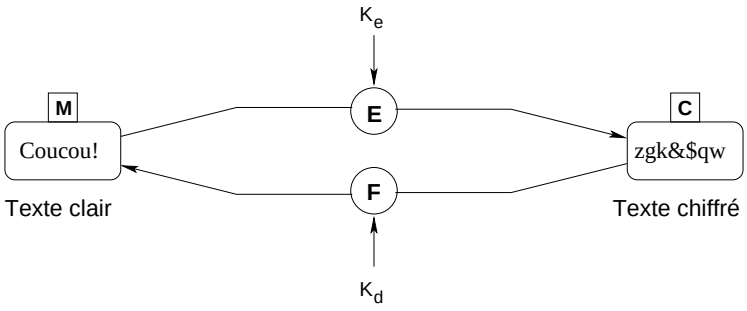


FIG. 3.1: Relation fondamentale de la cryptographie.

Le type de relation qui unit les clefs K_e et K_d utilisées dans le chiffrement et le déchiffrement permet de définir deux grandes catégories de systèmes cryptographiques :

- les systèmes à clef secrète d'une part : la clef est un secret partagé entre émetteur et destinataire ; ces systèmes sont décrits dans la section 3.3 ;
- les systèmes à clef publique d'autre part : aucune information secrète n'est partagée *a priori* par plusieurs protagonistes ; ces systèmes sont décrits dans la section 3.4.

3.1.2 À quoi sert la cryptographie ?

Les communications échangées entre Alice et Bob sont sujettes à un certain nombre de menaces (elles sont développées un peu plus loin, voir la section 3.2.2). La cryptographie apporte un certain nombre de fonctionnalités permettant de pallier ces menaces, résumées dans le sigle **CAIN**, pour Confidentialité, Authentification, Intégrité, Non-répudiation :

1. **Confidentialité** des informations stockées ou manipulées par le biais des algorithmes de chiffrement. La confidentialité consiste à *empêcher l'accès* aux informations qui transitent à ceux qui n'en sont pas les destinataires. Ils peuvent lire les messages cryptés transmis sur le canal mais ne peuvent pas les déchiffrer.
2. **Authentification** des protagonistes d'une communication (ou plus généralement de ressources). Il faut pouvoir détecter une usurpation d'identité. Par exemple, Alice peut s'identifier en prouvant à Bob qu'elle connaît un secret S qu'elle est la seule à pouvoir connaître.
3. **Intégrité** des informations stockées ou manipulées. Il s'agit de vérifier que le message n'a pas subi d'*altérations* lors de son parcours (voir figure 3.2). Cette vérification ne se place pas au même niveau que celle que nous verrons au chapitre 4. Elle concerne plutôt une modification volontaire et malicieuse e l'information provoquée par un tiers lors du transfert sur le canal. Ces modifications sont en générales masquées par le tiers pour être difficilement détectables. Sur la figure 3.2, par exemple, le contrôle d'intégrité sur un message M se fait grâce à une fonction f telle qu'il doit être très difficile de trouver deux messages M_1 et M_2 ayant la même image A par f.

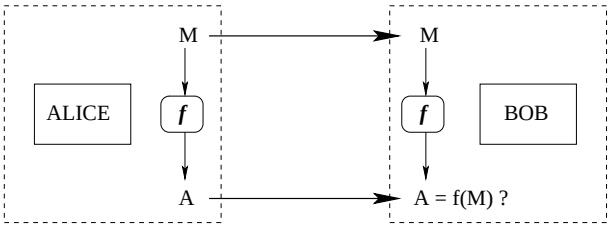


FIG. 3.2: Principe d'un algorithme de contrôle d'intégrité.

4. **Non-répudiation** des informations. C'est une protection des protagonistes d'un échange entre eux, et non plus contre un tiers. Si Alice envoie un message M, elle ne doit pas pouvoir prétendre ensuite devant Bob qu'elle ne l'a pas fait, ou alors qu'elle a envoyé M' et que le message a

été mal compris. On associe pour cela des algorithmes de type signature. Cet aspect sera plus détaillé à la section 3.5.

Jusqu'à une période récente, le chiffrement ne tenait compte que de la confidentialité, et seules les méthodes à clefs secrètes étaient développées. Les trente dernières années ont vu émerger l'étude de nouvelles tendances :

– l'authentification devient aussi voire plus importante que le secret. C'est particulièrement vrai dans le commerce électronique : il faut pouvoir prouver que la commande vient bien de la personne à qui la livraison est destinée pour éviter les contestations ;
– une partie de la clef doit être publique, afin de ne pas provoquer une explosion du nombre de clefs nécessaires pour communiquer avec un grand nombre de personnes.

Un dernier critère est primordial, c'est la rapidité des calculs de chiffrement et déchiffrement, et la taille des messages chiffrés. Les opérations portant sur de grandes quantités de données, le critère d'efficacité est très important afin de pouvoir chiffrer « à la volée » des flux audio ou vidéo par exemple en utilisant au minimum la bande passante.

Un système de chiffrement idéal devrait résoudre tous ces problèmes simultanément : utiliser des clefs publiques, assurer le secret, l'authentification et l'intégrité, le tout le plus rapidement possible. Malheureusement, il n'existe pas encore de technique unique qui satisfasse tous ces critères. Les systèmes conventionnels comme le DES (voir la section 3.3.3) sont efficaces mais utilisent des clefs privées ; les systèmes à clef publique peuvent assurer l'authentification mais sont inefficaces pour le chiffrement de grandes quantités de données car trop coûteux. Cette complémentarité a motivé le développement de protocoles cryptographiques hybrides, à l'instar de PGP (*Pretty Good Privacy*) (section 3.6.6), qui sont basés à la fois sur des clefs publiques et des clefs secrètes.

3.2 Attaques sur les systèmes cryptographiques

Pour construire un bon système cryptographique, il faut étudier les différents types d'attaques que peut employer Oscar pour comprendre les messages échangés. Une bonne connaissance de tous les types de menaces qui pèsent sur le secret permettra de le garantir.

3.2.1 Principes de Kerckhoffs

Longtemps, la sécurité d'un système cryptographique a reposé sur le secret qui entoure les algorithmes utilisés dans ce système. On citera comme exemple le chiffrement de César (voir la section 1.1) ou, plus récemment, le code ADFVGX

utilisé durant la Première Guerre Mondiale par les forces allemandes. La sécurité est illusoire car tôt ou tard, les détails de l'algorithme seront connus et sa faiblesse éventuelle pourra alors être exploitée. D'autre part, les systèmes publics sont souvent meilleurs car ils font l'objet d'attaques continuelles, et sont donc soumis à rude sélection. Un système cryptographique dont les mécanismes internes sont librement diffusés et qui résiste aux attaques continues de tous les cryptanalystes pourra être considéré comme sûr.
Le premier à avoir formalisé ce principe est Auguste Kerckhoffs en 1883 dans l'article « La cryptographie militaire » paru dans le « Journal des Sciences Militaires ». Son article comporte en réalité six principes, connus depuis sous le nom de « *principes de Kerckhoffs* ». On en résumera ici que trois, les plus utiles aujourd'hui :

1. La sécurité repose sur le secret de la clef et non sur le secret de l'algorithme. Ce principe est notamment utilisé au niveau des cartes bleues et dans le chiffrement des images et du son sur Canal+ ;
2. Le déchiffrement sans la clef doit être impossible (en temps raisonnable) ;
3. Trouver la clef à partir du clair et du chiffré est impossible (en temps raisonnable).

Ainsi, **toute attaque doit être envisagée en supposant que l'attaquant connaît tous les détails du cryptosystème**. Bien que connues depuis longtemps, on déplore malgré tout un certain nombre d'industriels qui continuent d'ignorer (volontairement ou non) ces règles. Parmi les exemples récents les plus médiatiques, on citera le cas des algorithmes de chiffrement A5/0 et A5/1 utilisés sur GSM et surtout le logiciel de protection anti-copie de DVD CSS (Content Scrambling System) introduit en 1996 qui s'avéra être contourné en quelques semaines en dépit du secret entouré autour de l'algorithme de chiffrement utilisé.

3.2.2 Les grands types de menaces

Attaques passives/actives

On distingue déjà les *attaques passives*, où Oscar se contente d'écouter les messages échangés entre Alice et Bob, et les *attaques actives*, dans lesquelles Oscar peut modifier le message au cours de sa transmission. Le premier type menace la confidentialité des informations, tandis que le second peut entraîner l'altération des informations ou des usurpations d'identité.

Cryptanalyse et attaques sur un chiffrement

On suppose généralement qu'Oscar connaît le système cryptographique utilisé (selon les principes de Kerckhoffs). On distingue les niveaux d'attaques

possibles :

Texte chiffré connu. Oscar ne connaît que le message chiffré C.

Texte clair connu. Oscar dispose à la fois d'un texte clair M et de sa correspondance chiffrée C.

Texte clair choisi. Oscar peut choisir un texte clair M et obtenir le texte chiffré associé C.

Texte chiffré choisi. Oscar peut choisir un texte chiffré C et obtenir le texte déchiffré associé M.

Dans tous les cas, garantir la confidentialité des communications entre Alice et Bob signifie qu'Oscar ne peut pas :
- trouver M à partir de $E(M)$; le système de chiffrement doit être résistant aux attaques sur le message codé,
- trouver la méthode de déchiffrement D à partir d'une séquence $\{\mathrm{E}(M_i)\}$ pour une séquence quelconque de messages clairs $\{M_1,M_2,M_3,\dots\}$, le système doit être sûr vis-à-vis des attaques avec du texte en clair.

Les fonctions de chiffrement sont paramétrées par des clefs. Pour « casser » un algorithme de chiffrement, on cherche le plus souvent à découvrir la valeur de ces clefs. On peut par exemple dire qu'une clef est bonne si son entropie (voir la définition au chapitre 1) est élevée car cela signifie qu'elle ne contient pas de motifs répétés qui donnent des informations sur sa forme.
Parallèlement aux niveaux d'attaques définis précédemment, il existe un certain nombre d'algorithmes généraux qu'Oscar va pouvoir utiliser :

Attaque par force brute. Elle consiste en l'énumération de toutes les valeurs possibles de la clef, c'est-à-dire à une exploration complète de l'espace des clefs. La complexité de cette méthode apparaît immédiatement : une clef de 64 bits oblige à essayer 2^{64} combinaisons différentes.

Pour une clef de 64 bits, il existe $1.844 * 10^{19}$ combinaisons différentes, sur un ordinateur essayant un milliard de clefs par seconde il faudra 584 ans pour être sûr de trouver la clef[1].

Attaque par séquences connues. Ce type d'attaque consiste à supposer connue une certaine partie du message en clair (par exemple les entêtes standards dans le cas d'un message transmis par courrier électronique) et de partir de cette connaissance pour essayer de deviner la clef.

Cette attaque peut réussir si l'algorithme de chiffrement laisse apparaître les mêmes régularités que le message original.

Attaque par séquences forcées Cette méthode, basée sur la précédente, consiste à faire chiffrer par la victime un bloc dont l'attaquant connaît le contenu.

[1] ou un an avec 584 ordinateurs tournant en parallèle.

Attaque par analyse différentielle Cette attaque utilise les faibles différences existant entre des messages successifs (par exemple des identifiant (*logs*) de serveur) pour essayer de deviner la clef.

3.3 Système cryptographique à clef secrète

3.3.1 Principe du chiffrement à clef secrète

En reprenant les notations de l'équation (3.1), le chiffrement à clef secrète repose sur le principe $K_e = K_d = K$. Autrement dit, Alice et Bob conviennent secrètement d'une clef secrète K qui est donc utilisée à la fois pour le chiffrement et le déchiffrement. Ils conviennent également d'un algorithme cryptographique de chiffrement et déchiffrement. Le principe général de l'utilisation d'un algorithme de chiffrement à clef secrète est illustré dans la figure 3.3. On utilise souvent l'**analogie au coffre-fort** pour caractériser les systèmes cryptographiques à clef secrète : seul celui qui possède la clef (Alice et Bob a priori) est capable d'ouvrir le coffre. Oscar, qui ne la possède pas, devra forcer le coffre s'il veut accéder à son contenu. Évidemment, si Oscar parvient par un moyen quelconque à obtenir cette clef, il pourra déchiffrer tous les messages échangés entre Alice et Bob.

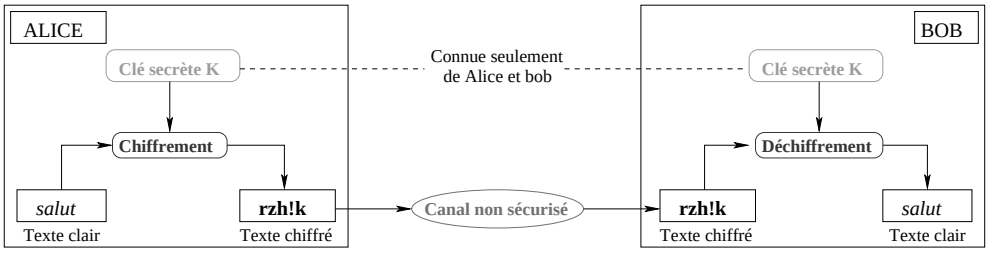


FIG. 3.3: Principe du chiffrement à clef secrète.

Notons qu'on emploie également l'expression « chiffrement symétrique » pour mentionner un chiffrement à clef secrète (par opposition aux chiffrements asymétriques). De tels systèmes ont l'avantage principal d'être efficaces en terme de temps de calcul, tant pour le chiffrement que pour le déchiffrement. En revanche, la faiblesse de ce système vient du secret absolu qui doit entourer la clef K. Il s'agit historiquement du premier type de chiffrement utilisé. Plusieurs exemples de ce type (le chiffrement de César et le chiffrement parfait de Vernam) ont été développés au chapitre 1. Voyons d'autres exemples de ce type sous la forme d'un exercice.

Exercice 3.1 (Chiffrement affine). *On a vu en page 19 le cas du chiffrement de César défini sur un alphabet $\mathcal{A}$ de n caractères. Il s'agit d'un cas particulier du chiffrement affine. Étant donnée une bijection entre $\mathcal{A}$ et $\mathbb{Z}_n$, et notant $\mathbb{Z}_n^*$ l'ensemble des inversibles de $\mathbb{Z}_n$, on peut écrire la fonction de chiffrement affine d'un message $M \in \mathbb{Z}_n$ à partir d'une clef $K = (a, b) \in \mathbb{Z}_n^* \times \mathbb{Z}_n$:*

$$\begin{array}{rccl} E_{(a,b)} : & \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ & x & \longrightarrow & E_{(a,b)}(x) = ax + b \mod n \end{array}$$

1. *Expliciter $|\mathbb{Z}_{26}^*|$ et $\mathbb{Z}_{26}^*$.*
2. *Toujours pour $n = 26$, et en utilisant la clef $K = (15, 7)$ (qu'on justifiera), donner le chiffrement de $x \in \{0, 1, 2\}$.*
3. *Expliciter la fonction de déchiffrement affine $D_{(a,b)}$.*
4. *Si le texte chiffré par le chiffrement affine semble incompréhensible, il ne modifie pas pour autant la fréquence d'apparition des différentes lettres qui composent le texte. La répartition statistique des lettres dans un texte écrit en français, ainsi que la bijection lettre par lettre qui sera utilisée dans la suite, est fournie dans le tableau 3.1.*

A → 0	8.11 %	J → 9	0.18 %	S → 18	8.87 %
B → 1	0.81 %	K → 10	0.02 %	T → 19	7.44 %
C → 2	3.38 %	L → 11	5.99 %	U → 20	5.23 %
D → 3	4.28 %	M → 12	2.29 %	V → 21	1.28 %
E → 4	17.69 %	N → 13	7.68 %	W → 22	0.06 %
F → 5	1.13 %	O → 14	5.20 %	X → 23	0.53 %
G → 6	1.19 %	P → 15	2.92 %	Y → 24	0.26 %
H → 7	0.74 %	Q → 16	0.83 %	Z → 25	0.12 %
I → 8	7.24 %	R → 17	6.43 %		

TAB. 3.1: Fréquence d'apparition des lettres en français.

Alice envoie à Bob le message texte suivant où par convention, les caractères qui ne sont pas dans $\mathcal{A}$ ne sont pas chiffrés. Vous avez intercepté ce message :

```
mcahbo isbfock, ekb kp cbfbo vobixo,
   hopcah op esp foi kp rbsmcuo.
 mcahbo bopcbl, vcb j'slokb cjjoixo
  jka haph c vok vboe io jcpucuo :
 "xo ! fspdskb, mspeaokb lk isbfock,
yko nske ohoe dsja ! yko nske mo eomfjoz fock !
      ecpe mophab, ea nshbo bcmcuo
      eo bcvvsbho à nshbo vjkmcuo,
 nske ohoe jo vxopat loe xshoe lo ioe fsae."
```

On compte les occurrences de chaque lettre de l'alphabet dans le texte chiffré. On obtient ainsi le tableau 3.2.

Dans le texte chiffré		Référence (texte français)	
o → 14	18.11%	e → 4	17.69%
b → 1	9.05%	s → 18	8.87%
c → 2	7.82%	a → 0	8.11%
e → 4	7.41%	⋮	⋮
⋮	⋮		

TAB. 3.2: Analyse de fréquence du texte chiffré et comparaison aux références.

(a) À partir de ce résultat, déduire la clef utilisée par Alice.

(b) En déduire la clef de déchiffrement et le message clair.

Solution page 296.

3.3.2 Classes de chiffrements symétriques

Comme on l'a vu au chapitre 1, sections 1.2 et 1.3, on distingue deux types de codes qui se traduisent en autant de types de chiffrements.

Les chiffrement symétriques par flot

Ceux-ci effectuent un traitement à la volée pour se rapprocher du modèle de Vernam (présenté section 1.2.1). Leur utilisation repose sur un générateur de nombres pseudo-aléatoires (1.3.6) et un mécanisme de substitution bit-à-bit rapide comme l'opération 'ou exclusif' (XOR $\oplus$) utilisée dans Vernam. Ce principe est illustré dans la figure 3.4.

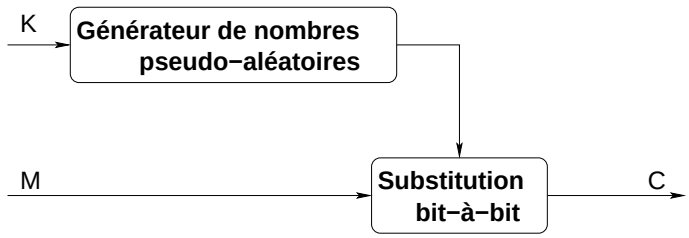


FIG. 3.4: Chiffrement symétrique par flot (Stream cypher).

On peut tout de suite remarquer en conséquence des principes de Kerckhoffs que la sécurité d'un algorithme de chiffrement symétrique par flot repose sur la qualité du générateur de clef.
Parmi les exemples les plus connus (pas forcément les plus sûrs) de chiffrement par flot, on citera LFSR (voir le fonctionnement du générateur page 71), RC4

(notamment utilisé dans le protocole SSL pour protéger les communications Internet ou encore dans le protocole WEP utilisé pour sécuriser les connexions WiFi), Py, E0 (utilisé dans les communications par Bluetooth) et A5/3 (utilisé dans les communication par GSM).

Les chiffrements symétriques par blocs

Ceux-ci découpent le message M de n bits en s blocs de $r = \frac{n}{s}$ bits (on ajuste initialement la taille du message en ajoutant des caractères sans signification afin que sa taille soit un multiple de r). Un algorithme de chiffrement par blocs opère sur des blocs de r bits, pour produire en général un bloc de r bits afin d'assurer la bijectivité du code. La combinaison avec un mode de chiffrement (ECB, CBC, CFB, OFB ou CTR - voir la section 1.3.1) décrit complètement le chiffrement du message M comme illustré dans la figure 3.5.

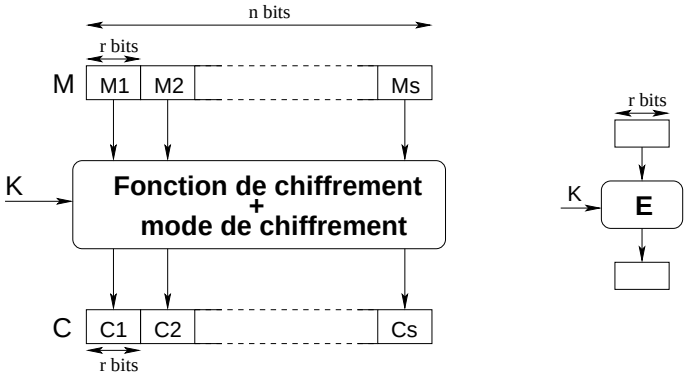


FIG. 3.5: Chiffrement symétrique par bloc (Bloc cypher).

Les systèmes classiques comme les DES et AES décrits plus loin utilisent le chiffrement par blocs.

Chiffrement inconditionnellement sûr

On rappelle qu'un chiffrement est dit *inconditionnellement sûr* ou *parfait* si la connaissance du message chiffré n'apporte aucune information sur le message clair. Ainsi, la seule attaque possible sera la recherche exhaustive de la clef secrète. À l'heure actuelle, le seul chiffrement prouvé inconditionnellement sûr est le chiffrement de Vernam, qui utilise une clef aussi longue que le texte clair (voir la section 1.2.1), **sous réserve que** la clef secrète soit totalement aléatoire et utilisée une seule fois. En particulier, tous les autres systèmes sont théoriquement cassables.

On considère donc aujourd'hui des algorithmes de chiffrement *pratiquement* sûrs dans lesquels un message chiffré ne permet de retrouver ni la clef secrète ni le message clair *en un temps humainement raisonnable*, ce qui permet d'utiliser des clefs de plus petite taille. Nous allons maintenant présenter les deux standards les plus importants pour les chiffrements symétriques par blocs.

3.3.3 Le système DES (*Data Encryption Standard*)

Présentation exhaustive du système DES

Ce système de chiffrement à clef secrète est le plus connu et fut proposé comme standard de chiffrement par le NIST en 1977. Il fonctionne par blocs de texte clair de 64 bits en utilisant une clef K de 56 bits[2]. Il permet d'obtenir des blocs de texte chiffré de 64 bits. L'algorithme se déroule en trois étapes :

1. Soit x un bloc de texte clair de 64 bits. On lui applique une permutation initiale IP fixée pour obtenir une chaîne x_0. On a donc : $x_0 = IP(x) = L_0R_0$ où L_0 contient les 32 premiers bits de x_0 et R_0 les 32 restants.
2. On effectue 16 itérations (qu'on appelle *tours de chiffrement* ou encore *rondes* par analogie sonore avec le terme anglais "round") d'une certaine fonction f dépendant de la clef K. La fonction f sera détaillée plus loin. On calcule $L_iR_i, 1 \leqslant i \leqslant 16$ suivant la règle :

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \end{cases} \tag{3.2}$$

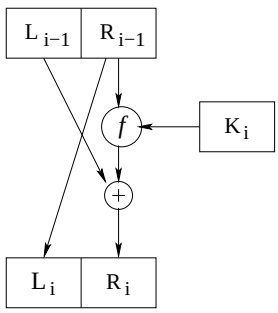


FIG. 3.6: Un tour de DES.

La figure 3.6 présente un tour de chiffrement. La fonction f est une fonction à deux variables, l'une de 32 bits (correspondant à R_{i-1} à la i^{eme} itération) et l'autre de 48 bits (il s'agit de K_i). Les éléments K_i sont obtenus par ce qu'on appelle la *diversification* des bits de la clef initiale K, développée un peu plus bas.

3. La permutation inverse IP^{-1} est appliquée à $R_{16}L_{16}$ pour obtenir un bloc de texte chiffré $y = IP^{-1}(R_{16}L_{16})$ (à noter l'inversion de L_{16} et de R_{16}).

Le même algorithme, avec la même clef mais en réalisant les tours de chiffrement dans l'ordre inverse, est utilisé pour déchiffrer.
La sûreté du DES vient de la fonction f. Cette fonction prend deux arguments :

[2]La clef compte en fait 64 bits mais parmi ceux-ci 1 bit sur 8 est utilisé comme contrôle de parité, voir section 4.1.2.

– une chaîne A de 32 bits (la partie droite du bloc à chiffrer) ;
– une chaîne J de 48 bits (une clef diversifiée).

Le calcul de $f(A, J)$ se déroule en plusieurs étapes, illustrées dans la figure 3.7 :

1. A est augmentée en une chaîne de 48 bits par une *fonction d'expansion* E.
2. $B = E(A) \oplus J$ est calculé et découpé en 8 sous-chaînes consécutives de 6 bits chacune : $B = B_1.B_2 \dots B_8$. Chacune des sous-chaînes B_i est ensuite passée en entrée à une boîte de substitution S_i (les fameuses boîtes-S ou SBox) pour fournir un bloc de 4 bits C_i en sortie.
3. La chaîne de 32 bits $C = C_1.C_2 \dots C_8$ est ensuite réordonnée suivant une permutation fixée P. Le résultat $P(C)$ définit $f(A, J)$.

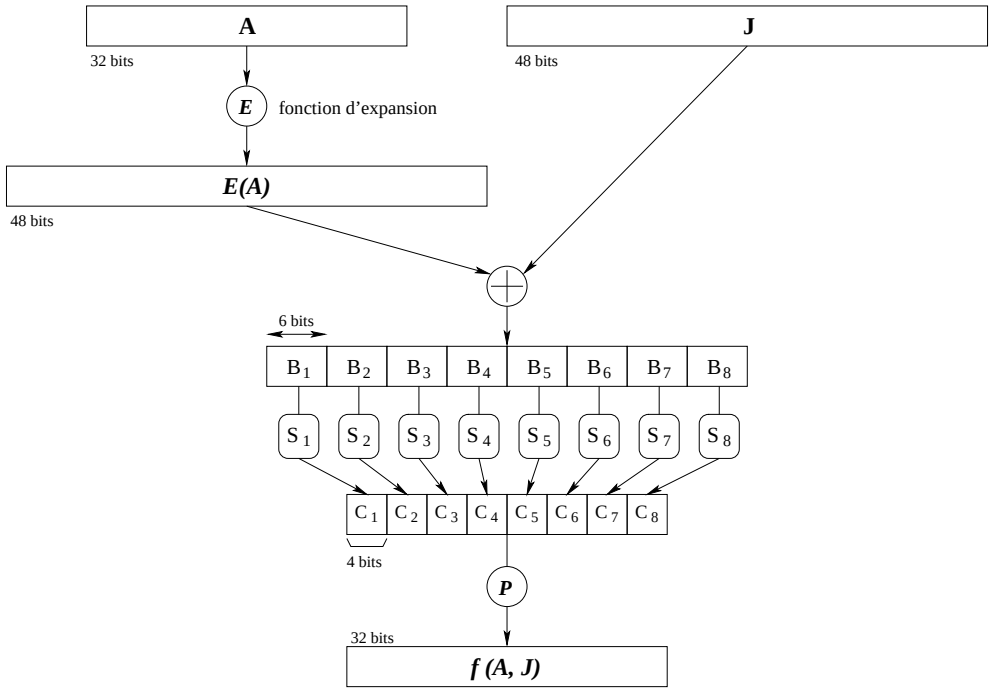


FIG. 3.7: La fonction f de DES.

Exercice 3.2 (Déchiffrement du DES). *Montrer qu'il n'est pas nécessaire d'inverser f pour inverser un tour de DES et en déduire l'algorithme de déchiffrement du système DES.* *Solution page 297.*

Diversification de la clef dans DES

On a vu que le second argument de la fonction f pour le tour i est une clef K_i extraite de la clef initiale K. Ce paragraphe détaille l'algorithme utilisé pour obtenir les 16 sous-clefs $\{K_i\}_{1 \leqslant i \leqslant 16}$.

1. La clef K de 64 bits est réordonnée dans une permutation initiale PC-1 qui supprime les bits de parités (ceux situés en position 8,16,...,64). On note PC-1$(K) = C_0.D_0$, où C_0 est composé des 28 premiers bits de PC-1(K) et D_0 des 28 restants.
2. C_i, D_i et K_i $(1 \leqslant i \leqslant 16)$ sont calculés récursivement de la façon suivante :

$$\begin{cases} C_i = LS_i(C_{i-1}) \\ D_i = LS_i(D_{i-1}) \\ K_i = \text{PC-2}(C_i.D_i) \end{cases} \tag{3.3}$$

   LS_i est une rotation circulaire vers la gauche d'une ou deux positions selon la valeur de i et PC-2 une autre permutation des bits.

Toutes ces étapes sont illustrées dans la figure 3.8.

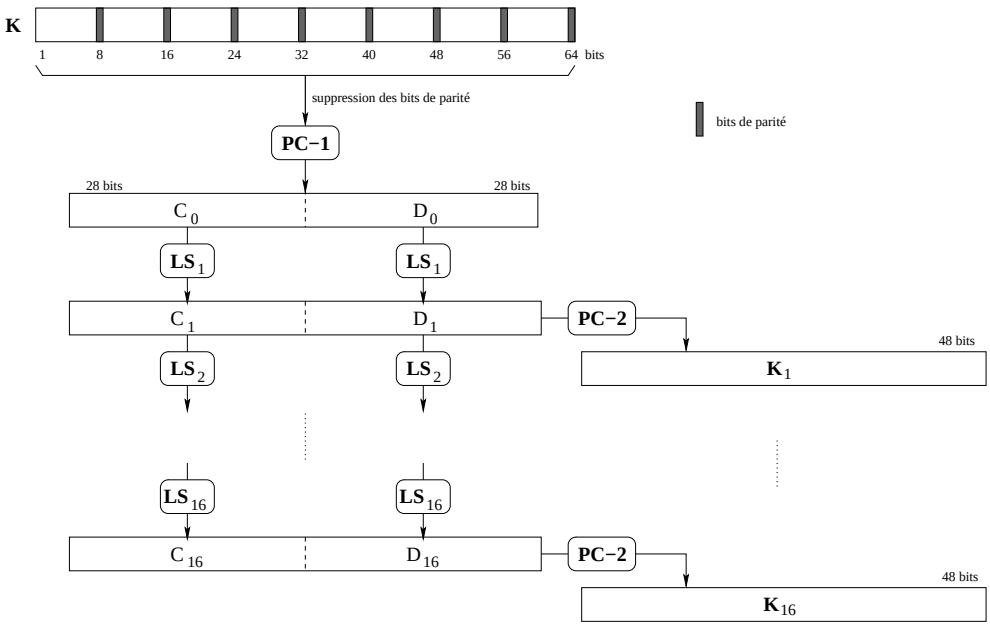


FIG. 3.8: La diversification de clef dans DES.

Avantages et applications de DES

Après 16 tours, le résultat de DES est statistiquement « plat », c'est-à-dire que les caractéristiques générales du message source (la fréquence des caractères, le nombre d'espaces, ...) seront indétectables. De plus il dispose d'une caractéristique très importante pour éviter les attaques par analyse différentielle : une légère modification de la clef ou du texte à chiffrer provoque des changements importants dans le texte chiffré.
Le gros avantage du DES est qu'il repose, tant pour le chiffrement que pour le déchiffrement, sur des opérations facilement implémentables au niveau matériel, il est donc possible d'obtenir des vitesses de chiffrement très élevées, de l'ordre de 40Mo/s il y a une dizaine d'années, avec du matériel spécifique. Aujourd'hui, avec une puce spécifique bas de gamme (de l'ordre de 60 euros), on arrive à des débits de l'ordre de 190 Mo/s.
Le système DES est actuellement utilisé pour chiffrer les paiements par cartes de crédit (par exemple *UEPS* pour *Universal Electronic Payment System*), des protocoles d'authentification dans les réseaux tels que *Kerberos* et la messagerie électronique avec par exemple *PEM* pour *Privacy-Enhanced Mail*. Voir le livre de Bruce Schneier (en bibliographie) pour plus de détails.

Exercice 3.3 (Confidentialité de D.E.S).

1. *Montrer qu'un chiffrement parfait (le chiffré ne donne aucune information sur le message, ou encore, en terme d'entropie, $H(M|C) = H(M)$; voir la section 1.2.4) vérifie $|K| \geqslant |M|$.*
2. *Discuter de la confidentialité parfaite du DES pour un bloc.*

Solution page 297.

Cryptanalyse de DES

Dès sa parution, de nombreux efforts de recherche furent investis dans la cryptanalyse de DES, d'abord sur un nombre réduit de tours puis sur la version complète. Des avancées significatives (notamment dans les techniques utilisées) apparurent au début des années 90. On peut aujourd'hui distinguer quatre méthodes de cryptanalyse de DES :
– Recherche exhaustive. Cette méthode peut évidemment être appliquée quel que soit le système.
– Précalcul exhaustif. Il s'agit de stocker le résultat du chiffrement de DES sur un texte choisi pour toutes les clefs K possibles. Ainsi, en obtenant un chiffré de ce texte, on peut facilement remonter à la clef utilisée (en coût constant). Cela suppose qu'on soit capable de stocker 2^{56} blocs de 64 bits, ce qui est aujourd'hui possible avec les capacités actuelles des disques durs.

– **Cryptanalyse différentielle.** Cette attaque marqua une avancée significative dans la cryptanalyse de DES. Il s'agit d'étudier des différences de chiffrement entre des textes similaires pour sélectionner un sous-ensemble de clefs probables.
– **Cryptanalyse linéaire.** Ce fut une autre étape majeure dans la cryptanalyse générale des chiffrements par blocs. Il s'agit d'utiliser des relations linéaires sur certains bits du message chiffré et du message clair pour interpoler des bits de la clef.

Le tableau 3.3 résume les complexités des attaques sur DES.

Méthode d'attaque	**Texte connu**	**Texte choisi**	**Stockage**	**Calculs**
Recherche exhaustive	1			2^{55}
Précalcul exhaustif		1	2^{56}	1 tableau
Crypta. linéaire	2^{47} puis 2^{36}		Textes	2^{47} puis 2^{36}
Crypta. différentielle	2^{55}	2^{47}	Textes	2^{47}

Tab. 3.3: Complexité des cryptanalyses sur DES.

Avec les progrès réalisés dans le domaine de la cryptanalyse et la progression de la puissance de calcul des processeurs, une étude fut demandée en 1996 pour estimer le coût et les performances des attaques sur DES pour un budget donné. Le résultat de cette étude est fourni dans le tableau 3.4.

Attaquant	**Budget**	**Outil**	**Clef 56 bits**
Hacker	300 €	Logiciel	38 ans
PME	7500 €	Circuit	18 mois
Grande Entreprise	225 K€	Circuit ASIC	19 j. 3 h
Multinationale	7,5 M€	ASIC	6 min
Gouvernement	225 M€	ASIC	12 s

Tab. 3.4: Coût et performance des attaques sur DES en 1996.

Aujourd'hui, le DES avec sa longueur de clef fixe de 56 bits n'est plus considéré comme pratiquement sûr. Un petit calcul simple peut en donner une idée : il y a 2^{56} clefs possibles c'est-à-dire moins de $10^{17} = 10^8 10^9$; en supposant que l'on dispose de 1000 PC tournant à $1GHz = 10^9 Hz$, il faut 10^5 secondes, soit 30 heures pour donc effectuer $1000 * 10^9 * 10^5 = 10^{17}$ opérations ; le cassage par force brute n'est donc pas impossible en un temps raisonnable !

Pour améliorer la sécurité de DES, une première solution consistait à combiner deux clefs. On obtient ainsi le système "double DES" pour lequel $C = E_2(E_1(M))$ et $M = D_1(D_2(C))$. Il apparut rapidement que le cassage effectif

de ce système n'était qu'environ 100 fois plus difficile que le cassage de DES (et non 2^{56} fois plus difficile comme on aurait pu s'y attendre en prenant deux clefs). En effet, l'attaque "Meet-in-the-middle" utilise un tri rapide pour casser le double DES à peine plus difficilement que le simple DES :

1. Effectuer les 2^{56} cryptages possibles X_i d'un message M.
2. Trier ces X_i avec un tri rapide en $O(n\log(n))$ étapes, soit approximativement 56×2^{56} opérations.
3. Effectuer tous les décryptages Y_j possibles de C par une seule clef j : , $Y_j = DES_j^{-1}(C)$; comparer au fur et à mesure les Y_j avec les X_i (ces derniers étant triés ceci ne nécessite qu'au pire $\log(n)$ étapes par Y_j en procédant par dichotomie).
4. Quand $X_{i_0} = Y_{j_0}$, les clefs recherchées sont i_0 et j_0.

La taille équivalente de clef d'un double DES est donc aux alentours de 64 bits seulement.
On proposa donc l'utilisation de "triple DES" avec cette fois-ci une clef effective de 112 bits mais le temps de calcul est triplé. Cela peut être réalisé avec 3 clefs et un cryptage $C = E_3(E_2(E_1(M)))$, ou $DES - EEE$; dans ce cas l'attaque précédente donne une clef équivalente plutôt aux alentours de 120 bits. On préfère donc souvent utiliser seulement deux clefs et un cryptage $C = E_1(D_2(E_1(M)))$, ou $DES-EDE$, qui donne une sécurité de 112 bits mais permet d'être compatible avec le cryptage DES simple en prenant $K_1 = K_2$.
Finalement, un nouveau standard plus rapide a pris sa place depuis 2000. Il s'agit de l'AES, *Advanced Encryption Standard*.

3.3.4 Le nouveau standard AES (Rijndael)

Rijndael est le chiffrement à clef secrète qui a été retenu par le NIST (*National Institute of Standards and Technology*) comme le nouveau standard américain de chiffrement (AES : *Advanced Encryption Standard*).Ce système a été choisi après une compétition organisée par le NIST. Des cinq finalistes (Rijndael, Serpent, Twofish, RC6, Mars), c'est Rijndael qui s'est avéré le plus résistant et est donc devenu le nouveau standard américain, remplaçant le DES. C'est un code par blocs encodant 128 bits avec des clefs de 128, 192 ou 256 bits. Il est important de noter que Rijndael était parmi les propositions les plus rapides, comme on peut le voir dans la table 3.5 issue du rapport NESSIE-D21.
Toutes les opérations d'AES sont des opérations sur des octets considérés comme des éléments du corps fini à 2^8 éléments, $\mathbb{F}_{256}$. On rappelle que pour p premier, $\mathbb{F}_{p^m}$ est isomorphe à $\mathbb{F}_p[X]/g(X)$, où $g(X)$ est un polynôme irréductible sur $\mathbb{F}_p[X]$ de degré m (voir la section 1.3.4).

Algorithme	Taille de clef	Cycles/octet	
		Cryptage	Décryptage
IDEA	128	56	56
Khazad	128	40	41
Misty1	128	47	47
Safer++	128	152	168
CS-Cipher	128	156	140
Hierocrypt-L1	128	34	34
Nush	128	48	42
DES	56	59	59
3-DES	168	154	155
kasumi	128	75	74
RC5	64	19	19
Skipjack	80	114	120
Camellia	256	47	47
RC6	256	18	17
Safer++	256	69	89
Anubis	256	48	48
Grand cru	128	1250	1518
Hierocrypt-3	260	69	86
Nush	256	23	20
Q	256	60	123
SC2000	256	43	46
Rijndael	256	34	35
Serpent	256	68	80
Mars	256	31	30
Twofish	256	29	25

TAB. 3.5: Vitesses comparées de quelques méthodes de chiffrements par blocs.

Pour AES, $p = 2$ et $m = 8$ et $g(X) = X^8 + X^4 + X^3 + X + 1$ qui est bien un polynôme irréductible sur $\mathbb{F}_2[X]$. Ainsi, l'octet $b_7b_6b_5b_4b_3b_2b_1b_0$ sera vu comme le polynôme $b_7X^7 + b_6X^6 + b_5X^5 + b_4X^4 + b_3X^3 + b_2X^2 + b_1X + b_0$. Par extension, le polynôme $g(X)$ s'écrira `0x11B`, qui est l'octet correspondant en notation hexadécimale.

– L'addition de deux polynômes est l'addition terme à terme des coefficients (l'addition dans $F_2 = \mathbb{Z}/_{2\mathbb{Z}}$) et correspond à l'opération ou exclusif (XOR $\oplus$) sur les octets.
– La multiplication est une multiplication de polynômes modulo $g(X)$.

Exercice 3.4 (Opérations sur $\mathbb{F}_{256}$).

1. *On considère les octets $a = 0x57$ et $b = 0x83$ (en notation héxadécimale, ou encore notés $a = [57]$ et $b = [83]$ si il n'y a pas d'ambiguïté) vus comme des éléments de $\mathbb{F}_{256}$.*

(a) *Calculer* $a + b$

(b) *Calculer* $a \times b$

2. *On utilise cette fois une notation polynomiale. Soit* $a(X) \in \mathbb{F}_{256}$.

   (a) *Donner un algorithme permettant de calculer l'élément de* $\mathbb{F}_{256}$ *correspondant à* $X.a(X)$.

   (b) *En déduire un algorithme calculant l'élément de* $\mathbb{F}_{256}$ $X^i a(X)$.

3. *Soit* $w(X)$ *un générateur de* $\mathbb{F}_{256}$. *Dans ce cas, tout élément non nul de* $\mathbb{F}_{256}$ *se représente de manière unique par* $w(X)^i \mod g(X)$, $0 \leqslant i < 255$. *En déduire un moyen efficace d'effectuer la multiplication de deux éléments dans* $\mathbb{F}_{256}$. Remarque *: Pour AES,* $w(X) = X + 1 =$ `0x03`.

Solution page 298.

Des blocs d'octets sont organisés sous forme matricielle, selon un modèle illustré dans la figure 3.9. Cette matrice aura nécessairement 4 lignes et un nombre de colonnes fonction de la taille choisie du bloc. On note N_b le nombre de colonnes dans une matrice pour écrire un bloc d'octets, et N_k ce même nombre pour une clef.

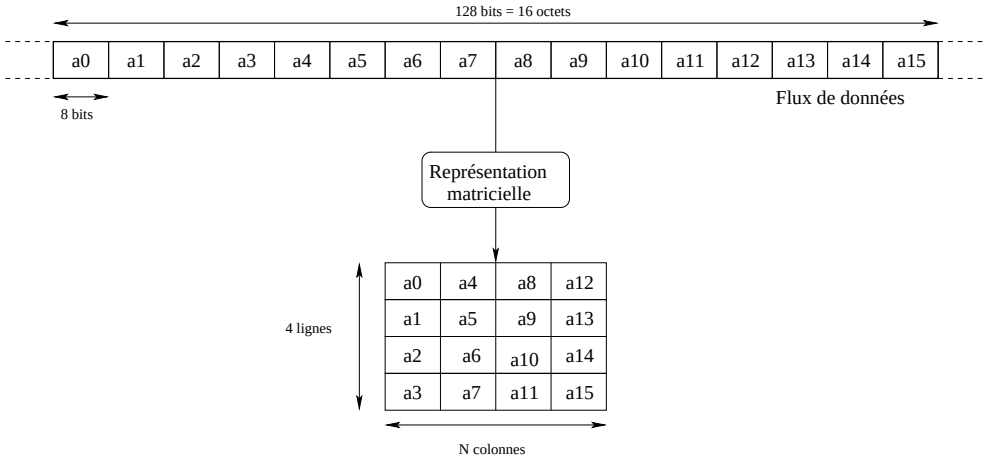


FIG. 3.9: Représentation matricielle d'un bloc de 16 octets.

Par exemple, pour les flux d'entrée/sortie qui, dans AES, correspondent à des séquences de 16 octets, on obtiendra des matrices de 4 lignes et $N_b = 4$ colonnes. De même :
– la matrice associée à une clef de 128 bits aura 4 lignes et $N_k = 4$ colonnes ;
– avec une clef de 192 bits, la matrice aura 4 lignes et $N_k = 6$ colonnes ;
– pour une clef de 256 bits, la matrice aura 4 lignes et $N_k = 8$ colonnes ;

Pour être tout à fait exact, l'algorithme AES n'est pas *stricto sensu* celui de Rijndael. Ce dernier supporte des tailles de blocs plus nombreux qu'AES. AES **fixe la taille de blocs à 128 bits** ($N_b = 4$), alors que l'algorithme de Rijndael utilise des **clefs de 128, 192 ou 256 bits**.

Principe de l'algorithme

Comme DES, AES exécute une séquence de tours qui seront détaillés dans la suite. On note N_r le nombre de tours qui doivent être effectués. Ce nombre dépend des valeurs de N_b et de N_k. Les différentes configurations possibles sont détaillées dans le tableau 3.6.

	N_k	N_b	N_r
`AES-128`	4	4	10
`AES-192`	6	4	12
`AES-256`	8	4	14

TAB. 3.6: Détail des configurations possibles.

Comme on l'a vu précédemment, AES opère sur des blocs vus comme une matrice $4 \times N_b$ d'éléments de $\mathbb{F}_{256}$. Le chiffrement AES consiste en une addition initiale de clef(`AddRoundKey`), suivie par $N_r - 1$ tours, chacun divisé en quatre étapes :

1. `SubBytes` est une substitution non-linéaire lors de laquelle chaque octet est remplacé par un autre octet choisi dans une table particulière (une *Boîte-S*).
2. `ShiftRows` est une étape de transposition où chaque élément de la matrice est décalé cycliquement à gauche d'un certain nombre de colonnes.
3. `MixColumns` effectue un produit matriciel en opérant sur chaque colonne de la matrice, vue comme un vecteur.
4. `AddRoundKey` combine par addition chaque octet avec l'octet correspondant dans une clef de tour obtenue par diversification de la clef de chiffrement.

Enfin, un tour final `FinalRound` est appliqué (il correspond à un tour dans lequel l'étape `MixColumns` est omise). La clef de tour pour l'étape i sera notée `RoundKeys[i]`, et `RoundKeys[0]` référencera un des paramètres de l'addition initiale de clef. La dérivation de la clef de chiffrement K dans le tableau `RoundKeys[]` est notée `KeyExpansion` et sera détaillée plus loin. L'algorithme 25 décrit le chiffrement AES.

Algorithme 25 Chiffrement AES.

Entrées Une matrice `State` correspondant au bloc clair, une clef K
Sorties Une matrice `State` correspondant au bloc chiffré

```
KeyExpansion(K, RoundKeys)
AddRoundKey(State, RoundKeys[0]) ; // Addition initiale
Pour r = 1 à N_r − 1 Faire
    SubBytes(State) ;
    ShiftRows(State) ;
    MixColumns(State) ;
    AddRoundKey(State, RoundKeys[r]) ;
Fin Pour
// Tour final
SubBytes(State) ;
ShiftRows(State) ;
AddRoundKey(State, RoundKeys[N_r]) ;
```

Étape `SubBytes(State)`. Cette étape correspond à la seule transformation non-linéaire de l'algorithme. Dans cette étape, chaque élément de la matrice `State` est permuté selon une table de substitution inversible notée `SBox`. La figure 3.10 illustre par exemple la transformation de l'élément $a_{2,2}$ en l'élément $b_{2,2} = S[a_{2,2}]$.

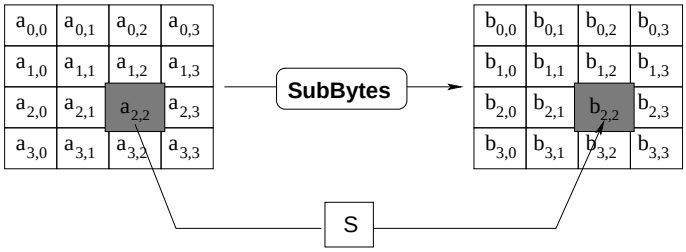


FIG. 3.10: Étape `SubBytes` dans AES.

Remarque : la table `SBox` dérive de la fonction inverse $t : a \longrightarrow a^{-1}$ sur $\mathbb{F}_{256}$. Cette fonction est connue pour ses bonnes propriétés de non-linéarité (0 n'ayant pas d'inverse, il est envoyé sur 0 pour cette étape). Afin d'éviter des attaques basées sur de simples propriétés algébriques, la boîte-S est construite en combinant cette fonction inverse avec une transformation affine inversible f. On a donc :

$$\texttt{SBox}[a] = f\left(t(a)\right), \text{ pour tout } a \in \mathbb{F}_{256}$$

Les concepteurs ont également fait en sorte que cette boîte-S n'admette pas

de point fixe, ni de point fixe opposé :

$$\texttt{SBox}[a] + a \neq \texttt{00}, \quad \text{pour tout } a \in \mathbb{F}_{256}$$
$$\texttt{SBox}[a] + a \neq \texttt{FF}, \quad \text{pour tout } a \in \mathbb{F}_{256}$$

Enfin, la fonction affine f est définie par :

$$b = f(a) \Longleftrightarrow \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Exercice 3.5 (`InvSubBytes`). *Détailler l'opération inverse de cette étape.*
Solution page 299.

Exercice 3.6 (Opérations autour de `SubBytes`). *On pose $a = 0x11 \in \mathbb{F}_{256}$ (notation hexadécimale) avec les conventions de représentation utilisées dans AES. Calculer* $\texttt{SBox}[a]$.
Solution page 299.

Étape `ShiftRows(State)`. Cette étape opère sur les lignes de la matrice `State` et effectue pour chaque élément d'une ligne un décalage cyclique de n éléments vers la gauche. Le nombre d'octets décalés dépend de la ligne considérée. La ligne i est décalée de C_i éléments, si bien que l'élément en position j de la ligne i est déplacé en position $(j - C_i) \mod N_b$. Les valeurs de C_i dépendent de la valeur de N_b et sont détaillées dans la table 3.7.

N_b	C_0	C_1	C_2	C_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

TAB. 3.7: `ShiftRows` : décalage des lignes en fonction de N_b dans l'algorithme Rijndael.

Évidemment, cette table n'est fournie qu'à titre indicatif dans la mesure où AES fixe la taille de bloc à $N_b = 4$. L'étape `ShiftRows` est illustrée dans la figure 3.11.

Exercice 3.7 (`InvShiftrows`). *Détailler l'opération inverse de cette étape.*
Solution page 300.

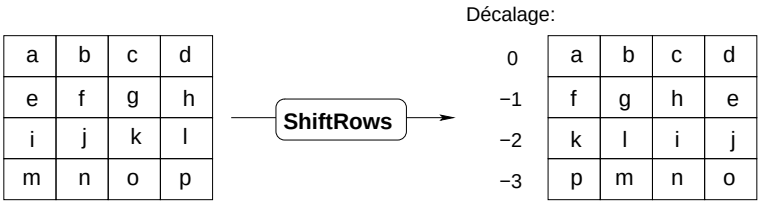


FIG. 3.11: Opération `ShiftRows` dans AES.

Étape `MixColumns(State).` La transformation `MixColumns` opère sur les colonnes c de la matrice `State` en les considérant chacune comme un polynôme $a(X)$ de degré 3 à coefficients dans $\mathbb{F}_{256}$. Elle consiste à effectuer pour chaque colonne une multiplication par $c(X) = \texttt{03}X^3 + X^2 + X + \texttt{02}$, modulo le polynôme $X^4 + 1$ (les coefficients des polynômes sont des éléments de $\mathbb{F}_{256}$ notés comme des nombres hexadécimaux). Dans `MixColumns`, on réalise donc l'opération : $(\texttt{03}X^3 + X^2 + X + \texttt{02}) \times a(X) \mod (X^4 + 1)$ Matriciellement, cette opération (illustrée dans la figure 3.12) s'écrit :

$$b(X) = c(X) \times a(X) \mod (X^4+1) \Longleftrightarrow \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

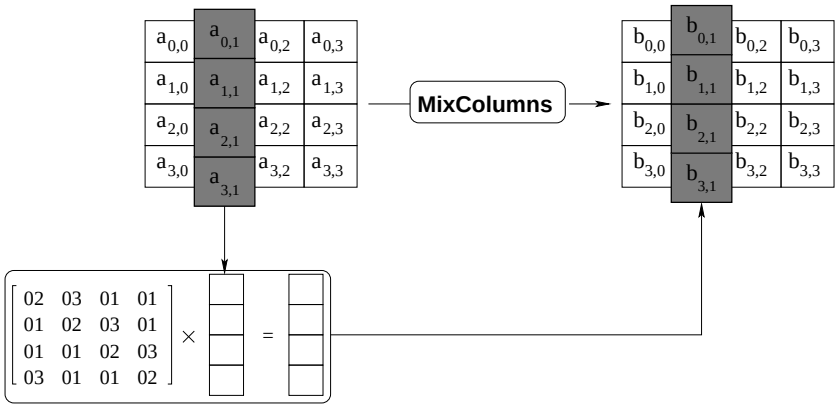


FIG. 3.12: Opération `MixColumns` dans AES.

En outre, comme $c(X)$ et X^4+1 sont premiers entre eux, $c(X)$ est bien inversible modulo X^4+1 et la transformation MixColumn est également inversible, ce qui permet de décrypter.

Exercice 3.8 (Retour sur $\mathbb{F}_{256}$ et `InvMixColumns`). *On reprend la représentation des éléments de $\mathbb{F}_{256}$ utilisée dans AES. Les éléments sont écrits sous forme de deux chiffres hexadécimaux encadrés. Par exemple, $X^7+X^6+X^2+1$ s'écrit en binaire* 11000101 *et* $[C5]$ *en hexadécimal.*

1. *Calculer* $[0B]+[A2]$, $-[03]$, $[FD]-[F0]$, $[FD]+[F0]$, $[23]+[45]$.
2. *Quel est le résultat de la division euclidienne de $g(X)$ par $X+1$ sur $\mathbb{F}_2[X]$?*
3. *En déduire l'inverse de* $[03]$ *dans* $\mathbb{F}_{256}$.
4. *Donner un algorithme de multiplication par X en binaire dans $\mathbb{F}_{256}$. En déduire les valeurs binaires de X^8, X^9, X^{10}, X^{11}, X^{12}, X^{13} et X^{14}. Qu'est-ce que la multiplication par* $[02]$ *?*
5. *Que vaut $(a+b)^2$ modulo 2 ? En déduire l'expression de $(a_1+a_2+\ldots+a_n)^2$ modulo 2.*
6. *Déduire des deux questions précédentes la valeur de* $[F6]^2$.
7. *Dans l'AES, l'étape MixColumn est réalisée par la multiplication d'une colonne par le polynôme $c(Y)=[03]Y^3+Y^2+Y+[02]$, le tout modulo le polynôme $M=Y^4+[01]$. Calculer le polynôme Q tel que $M=cQ+R$ avec $R(Y)=[A4]Y^2+[A5]Y+[A5]$.*
8. *En supposant connus deux polynômes U et V tels que $Uc+VR=1$, donner une relation de Bézout entre c et M.*
9. *Application : $U=[A0]Y+[FE]$ et $V=[1D]Y^2+[1C]Y+[1D]$ sont donnés, expliciter la fonction réciproque de MixColumn dans l'AES. Cette fonction sera notée `InvMixColumns` dans la suite.* On pourra se servir des résultats suivants : $[F6][1D]=[0B]$, $[F6][1C]=[FD]$, $[52][1D]=[F0]$, $[52][1C]=[A2]$, $[52][F6]=[C7]$.

Solution page 300.

Étape `AddRoundKey(State,Ki)`. Il s'agit d'une simple addition des deux matrices `State` et `Ki`. L'addition opérant sur des éléments de $\mathbb{F}_{256}$, c'est une opération ou exclusif $\oplus$ bit à bit sur les octets.

Exercice 3.9 (`InvAddRoundKey`). *Détailler l'opération inverse de cette étape.*
Solution page 300.

La diversification de la clef dans AES

Cette étape, notée `KeyExpansion`, permet de diversifier la clef de chiffrement `K` (de $4N_k$ octets) dans une clef étendue `W` de $4N_b(N_r+1)$ octets. On disposera ainsi de N_r+1 clefs de tours (chacune de $4Nb$ octets - voir figure 3.13).

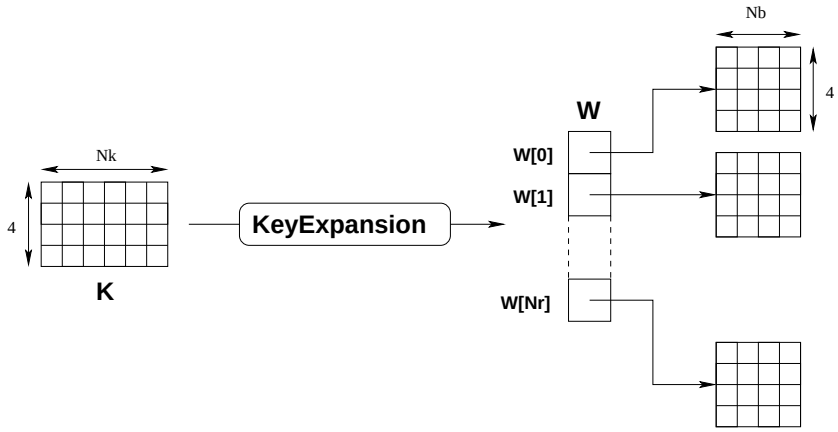


FIG. 3.13: Opération `KeyExpansion` dans AES.

Les tableaux `K` et `W` peuvent être vus comme une succession de colonnes, chacune constituée de 4 octets, comme l'illustre la figure 3.14. Dans la suite, on notera `c[i]` (respectivement `k[i]`) la $(i+1)^{\text{ème}}$ colonne de `W` (respectivement de `K`).

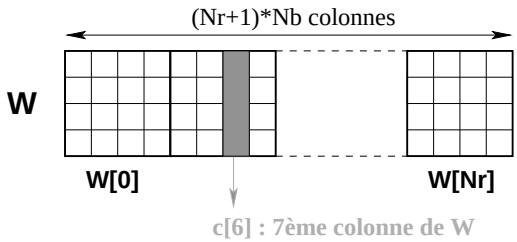


FIG. 3.14: Vision de la clef étendue `W` comme une succession de colonnes.

L'algorithme utilisé pour la diversification de clef diffère légèrement selon que $N_k \leqslant 6$ ou $N_k > 6$. Dans tous les cas, les N_k premières colonnes de `K` sont recopiées sans modifications aux N_k premières colonnes de `W`. Les colonnes suivantes sont définies récursivement à partir des colonnes précédentes. `KeyExpansion` utilise notamment les deux fonctions et le tableau de constantes suivants :

– `SubWord` qui est une fonction prenant en entrée un mot de 4 octets et applique la boîte-S `SBox` sur chacun des octets.
– La fonction `RotWord` qui prend en entrée un mot de 4 octets $a = [a_0, a_1, a_2, a_3]$ et effectue une permutation circulaire de façon à renvoyer le mot $[a_1, a_2, a_3, a_0]$.

– Le tableau de constantes de tours `Rcon[i]`, indépendant de N_k, qui est défini récursivement par :

$$\texttt{Rcon}[i] = [x^{i-1}, \texttt{00}, \texttt{00}, \texttt{00}], \forall i \geqslant 1$$

L'algorithme 26 résume la diversification des clefs dans AES.

Algorithme 26 Diversification des clefs dans AES.

Entrées Une clef K de $4N_k$ octets.
Sorties Une clef étendue W de $4N_b(N_r + 1)$ octets

```
Pour i = 0 à N_k − 1 Faire
  W[i] = K[i]
Fin Pour
Pour i = N_k à N_b(N_r + 1) − 1 Faire
  tmp = W[i-1]
  Si i mod N_k = 0 Alors
    tmp = SubWord(RotWord(tmp)) + Rcon[i/Nk]
  Sinon, si (N_k > 6) ET (i mod N_k = 4) Alors
    tmp = SubWord(tmp) ;
  Fin Si
  W[i] = W[i-N_k] + tmp
Fin Pour
```

Exercice 3.10 (Déchiffrement A.E.S). *Détailler l'algorithme de déchiffrement du système AES.* *Solution page 300.*

Exercice 3.11 (Tailles de blocs et collisions). *On rappelle que les différents modes de chiffrement ont été introduits section 1.3.1.*

1. *Mode ECB : Décrire un scénario illustrant la faiblesse de ce mode.*
2. *Mode CBC :*
   (a) *En supposant que l'on chiffre un disque dur de 32Go de données avec Triple-DES en mode CBC, quel est le nombre de blocs cryptés et quelle est la probabilité d'obtenir une collision entre deux blocs cryptés Y_i et Y_j ?*
   (b) *Comment exploiter cette faille ?*
   (c) *Qu'en est-il avec l'AES ? Conclure sur l'importance de la taille des blocs.*

Solution page 301.

Sécurité de AES

La « S-box »a été construite pour être résistante à la cryptanalyse. En particulier, elle ne possède pas de point fixe $S(a) = a$, ni de point opposé $S(a) = \bar{a}$, ni de point fixe inverse $S(a) = S^{-1}(a)$. Ensuite, l'opérateur ShiftRows permet de diffuser largement les données en séparant les octets originellement consécutifs. Enfin, combinée avec MixColumn, elle permet qu'après plusieurs tours chaque bit de sortie dépende de tous les bits en entrée. Par ailleurs, MixColumn est un code linéaire de distance maximale (voir section 4.4.1).
Il n'y a actuellement aucune attaque significative sur AES. Il convient cependant de tempérer notre propos dans la mesure où ce standard est encore relativement récent et n'est donc soumis à la cryptanalyse mondiale que depuis un peu plus de 7 ans.

3.4 Système cryptographique à clef publique

3.4.1 Motivations et principes généraux

Nous avons vus que les systèmes cryptographiques à clef secrète peuvent être pratiquement sûrs et efficaces en termes de temps de calcul. Néanmoins, dès le milieu des années 1970, de nouvelles interrogations furent soulevées :
– Avant d'utiliser un système de chiffrement à clef secrète, comment convenir d'une clef ?
– Comment établir une communication sécurisée entre deux entités sans aucun échange préalable de clef ?

C'est pour répondre à ces questions qu'en 1976, Diffie et Hellman posèrent les bases des systèmes cryptographiques à clef publique, par **analogie avec une boîte aux lettres** dont Bob est le seul à posséder la clef :
– toute personne peut envoyer du courrier à Bob ;
– seul Bob peut lire le courrier déposé dans sa boîte aux lettres.

Pour comparaison, un système de chiffrement à clef secrète était vu comme un coffre-fort dont la clef est partagée par Alice et Bob.
Dans le cadre d'un tel système et toujours pour reprendre les notations de l'équation (3.1), on a cette fois $K_e \neq K_d$. Pour être plus précis, K_e est une clef publique qui est publiée dans une sorte d'annuaire (en fait sous la forme d'un certificat : voir la section 3.6.5) de telle sorte que n'importe qui pourra récupérer cette clef, en tester l'origine et chiffrer un message avec. K_d est une clef privée et secrète que Bob garde pour lui et qui lui servira à déchiffrer. Comme les deux clefs sont distinctes, le chiffrement à clef publique est également appelé *chiffrement asymétrique*.

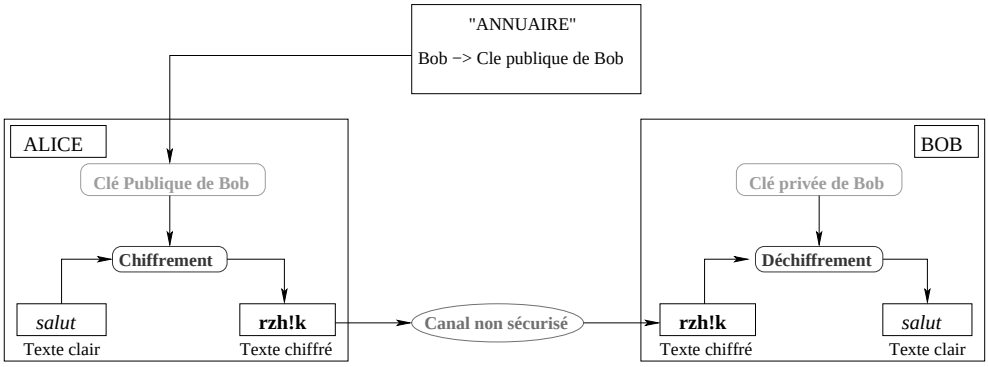


FIG. 3.15: Principe du chiffrement à clef publique.

Peut-on caractériser un tel système de chiffrement à l'aide des outils de la théorie de l'information ? Dans ce cadre, la mesure du secret consiste à calculer l'information restant à trouver sur la clef privée, sachant que la clef publique est connue :$H(K_d|K_e)$. Par ailleurs, un algorithme asymétrique doit vérifier $D_{K_d} = E_{K_e}^{-1}$ où les fonctions D et E sont parfaitement connues. Cela induit que la clef K_d est parfaitement déterminée dès que la clef publique K_e est également connue : il suffit d'inverser la fonction E_{K_e}. En théorie du moins, cela implique que $H(K_d|K_e) = 0$ et donc qu'un système asymétrique ne présente aucun secret. En pratique, les deux clefs K_e et K_d sont donc liées mais choisies de telle sorte qu'il soit trop difficile de calculer la valeur de la clef K_d à partir de K_e.
On utilise pour cela le principe des *fonctions à sens unique*, introduit à la section 1.3.3. Le secret d'un cryptosystème à clef publique ne peut donc pas être caractérisé par la théorie de l'information de Shannon ; ce secret ne vient pas de l'incertitude sur la clef privée K_d mais sur la difficulté intrinsèque à calculer K_d à partir de la seule connaissance de la clef K_e et de C. L'outil mathématique permettant de caractériser cette difficulté est la théorie de la *complexité algorithmique*.
L'exemple le plus connu et le plus utilisé d'algorithme de chiffrement à clef publique, principalement en raison de sa simplicité, est le système RSA, du nom de ses concepteurs, Rivest, Shamir et Adleman. La difficulté de RSA est basée sur la difficulté de factoriser un nombre entier.

3.4.2 Chiffrement RSA

En combinant le théorème d'Euler et le théorème chinois (page 49), on peut obtenir une relation de congruence commune à tous les éléments de $\mathbb{Z}_n$ lorsque

n est le produit de deux nombres premiers. C'est un résultat de Rivest, Shamir et Adleman, de 1978, à l'origine de la cryptographie à clef publique RSA.

Théorème 20 (Théorème RSA). *Soit $n = pq$ un produit de deux nombres premiers. Soit a un élément quelconque de $\mathbb{Z}_n$. Quel que soit l'entier positif k, $a^{k\varphi(n)+1} = a \mod n$.*

Preuve. Il y a deux cas : si a est inversible dans $\mathbb{Z}_n$, alors on applique directement le théorème d'Euler et donc $a^{\varphi(n)} = 1 \mod n$. En élevant à la puissance k et en remultipliant par a, on obtient la relation désirée.
Dans le deuxième cas, si a est nul, la relation modulo n est immédiate, et sinon on utilise deux fois le théorème de Fermat modulo chacun des deux nombres premiers p et q.
En effet, si a est non nul modulo p premier, alors $a^{p-1} = 1 \mod p$. En élevant à la puissance $k(q-1)$ et en remultipliant par a, on obtient $a^{k\varphi(n)+1} = a \mod p$.
Si au contraire $a = 0 \mod p$, on a également forcément $a^{k\varphi(n)+1} = 0 = a \mod p$. De la même manière on obtient une relation similaire modulo q : $a^{k\varphi(n)+1} = a \mod q$.
Comme p et q sont premiers entre eux, il suffit maintenant d'appliquer l'unicité du théorème chinois, pour avoir $a^{k\varphi(n)+1} = a \mod pq$. □

Nous en déduisons le système de codage RSA, dans lequel un utilisateur crée son couple (clef publique, clef privée) en utilisant la procédure suivante :

1. Choisir deux grands nombres premiers p et q. Il faut que p et q contiennent au moins 150 chiffres décimaux chacun.
2. Calculer $n = pq$
3. Choisir un petit entier e premier avec $\varphi(n) = (p-1)(q-1)$
4. Calculer d, l'inverse de e par la multiplication modulo $\varphi(n)$.
5. Publier la paire $K_e = (e, n)$ comme sa clef publique RSA.
6. Garder secrète la paire $K_d = (d, n)$ qui est sa clef privée RSA.

On a alors :

Chiffrement RSA : $E(M) = M^e \mod n$

Déchiffrement RSA : $D(\tilde{M}) = \tilde{M}^d \mod n$

On vérifie aisément que, pour $0 \leqslant M < n$, on a : $D_{K_d}(E_{K_e}(M)) = M$. En effet, $D_{K_d}(E_{K_e}(M)) = M^{ed} \mod n = M^{1+\alpha\varphi(n)} \mod n = M$, d'après le théorème RSA.

Exercice 3.12 (Chiffrement RSA).

1. *Déterminer la clé publique et la clé privée pour $p = 47$ et $q = 59$. On prendra $e = 17$ (et on justifiera la possibilité de ce choix).*

2. *Chiffrer la lettre B en système ASCII (66) avec la clé publique et vérifier que la clé privée permet bien de retrouver le message initial.*

Solution page 302.

Exercice 3.13 (Chiffrement/Déchiffrement RSA). *On considère la clef publique RSA* $K_e = (e, n) = (11, 319)$.

1. *Quel est le chiffrement avec cette clé du message* $M = 100$ *?*
2. *Calculer* d *la clé privée correspondant à la clé publique* e.
3. *Déchiffrer le message* $C = 133$. *On pourra se servir du résultat :* $133^{25} = 133 \pmod{319}$.
4. *Le message codé* 625 *peut-il résulter d'un codage avec la clé publique ? Même question avec la clé privée.*

Solution page 302.

Exercice 3.14 (Chiffrement de Vigenère et RSA). *On considère un système de chiffrement opérant sur l'alphabet* $\{A, B, C, \ldots, Z, \_\}$ *dont chaque symbole est désigné par un nombre compris entre 0 et 26 :*

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Étant donnés une clé $K = K_0 K_1 \ldots K_k$ *et un message clair* $M = M_0 M_1 \ldots M_k$, *le chiffré* $C = C_0 C_1 \ldots C_k$ *est donné par :*

$$\text{Pour tout } i \in [0, k], \quad C_i = M_i + K_i \mod 27$$

C'est ce qu'on appelle le chiffrement de Vigenère. *Voici un texte français chiffré qu'Oscar a intercepté pour vous :*

```
HWQIO QVPIF TDIHT Y_WAF NGY_F COMVI CGEVZ CVIAF JDFZK
YLYHG YGEHR SHMMX CVHBF AJYKN ZIXHP ZHEQY YJRHT YWMUK
YKPBY YGEHA G_DY_ YWDTF MHFZK YZYHX CISVI CHIVZ
```

1. *Fournir une méthode de cryptanalyse d'un chiffrement de Vigenère en commençant par déterminer la taille de la clef.*
2. *Oscar et Eve travaillent aussi sur la cryptanalyse de ce texte qu'ils ont également intercepté. Oscar envoie à Eve la longueur de la clef qu'il a réussit à déterminer. Pour cela, il utilise la clef publique RSA de Eve* $(n, e) = (35, 5)$. *Ivan intercepte le message chiffré RSA de la longueur de la clef : il obtient* 10. *Quelle est la longueur de la clef ?*
3. *Eve a réussi à déchiffrer la deuxième et la troisième clef de chiffrement. Elle les envoie à Oscar en utilisant la clef publique RSA de ce dernier* $(n, e) = (65, 7)$. *Ivan intercepte ainsi le chiffré de* K_1 *(il obtient 48) et*

de K_2 (4). Quelles étaient les valeurs de K_1 et de K_2 ?
Note : *on pourra utiliser les résultats suivants :*
$48^2 = 29 \mod 65$ *et* $48^5 = 3 \mod 65$

4. *Maintenant que vous disposez de la longueur de la clef, déchiffrez le texte. Préciser la clef utilisée pour le chiffrement* ***et*** *le déchiffrement (sous forme de chaîne de caractères). La répartition (en %) des symboles utilisés dans un texte en français est résumée dans le tableau 3.8.*

_	18,35%
E	14,86%
S	6,97%
A	6,40%
N	6,23%
...	...

TAB. 3.8: Répartition fréquentielle des symboles dans un texte français.

Solution page 302.

Efficacité et robustesse de RSA.

Grâce aux algorithmes vus dans les sections précédentes :
- Il est facile de générer des grands nombres premiers, tout au moins en acceptant un taux d'erreur (voir le test de Miller-Rabin, section 1.3.4, page 56). Dans le cas de RSA, l'erreur n'est pas trop grave : en effet, si l'on commet une erreur en croyant que p et q sont premiers, le destinataire se rendra rapidement compte que les nombres ne sont pas premiers : soit la clef d n'est pas inversible, soit certains blocs du message décrypté sont incompréhensibles. Dans ce cas, on peut procéder à un changement de système RSA (recalcul de p et q).
- le calcul du couple (e, d) est extrêmement facile : il suffit d'appliquer l'algorithme d'Euclide étendu ;
- enfin chiffrement et déchiffrement sont réalisés par exponentiation modulaire. Nous avons vu que cette exponentiation pouvait être réalisée assez efficacement.

La sécurité fournie par RSA repose essentiellement sur la difficulté à factoriser de grands entiers. En effet, si un attaquant peut factoriser le nombre $n = pq$ de la clef publique, il peut alors déduire directement $\varphi(n) = (p-1)(q-1)$ et donc calculer la clef privée K_d à partir de la clef publique K_e par l'algorithme d'Euclide étendu. Donc, si l'on dispose d'un algorithme rapide pour factoriser de grands entiers, casser RSA devient facile aussi.

Après 20 ans de recherche, aucun moyen plus efficace que la factorisation de n n'a été publié pour casser RSA. Cependant, la réciproque : « si factoriser de grands entiers est dur alors casser un message crypté par RSA est dur »n'a pas été prouvée. Autrement dit, on ne sait pas aujourd'hui si casser RSA est aussi difficile que factoriser n.

On peut cependant montrer l'équivalence entre "déterminer d à partir de (n, e)" et "factoriser n". Un sens est trivial : connaissant p et q, on peut facilement calculer d puisque c'est ce qui est fait dans le cadre de la génération de clef. Réciproquement, calculer p et q à partir de (n, e, d) peut être obtenu :
- par un algorithme déterministe si e est petit ($e \leqslant \log n$ typiquement - voir exercice 3.15) ;
- par un algorithme probabiliste (une variante de l'algorithme de Miller-Rabin) dans le cas général (exercice 3.16).

Exercice 3.15 (Factorisation de n à partir de n,e et d pour e petit). *On suppose disposer dans un système RSA de (n, e, d) avec e "petit".*

1. *Montrer qu'il existe $k \in \mathbb{Z}$, tel que $ed - 1 = k \mod n - (p + q) + 1$.*
2. *On suppose p et q différents de 2 et 3. Montrer que $k \leqslant 2e$.*
3. *Proposer alors un algorithme permettant de factoriser n.*

Solution page 304.

Exercice 3.16 (Factorisation de n à partir de n, e et d). *On suppose disposer dans un système RSA de (n, e, d). Soit $s = \max\{v \in \mathbb{N}$, tel que 2^v divise $ed - 1\}$ (Autrement dit : $ed - 1 = t2^s$).*

1. *Montrer qu'une variante de Miller-Rabin permet de factoriser n,* i.e *de déterminer p et q. On précisera l'algorithme qui effectue un seul tirage et renvoie soit les facteurs de n, soit une erreur.*
2. *Combien d'appels à cet algorithme sont en moyenne nécessaires pour aboutir à la factorisation de n ?*

Solution page 305.

Ainsi, il est aussi difficile de casser RSA en cherchant à calculer la clef privée d à partir de la clef publique (e, n) que de factoriser n (un problème réputé difficile si p et q sont très grands). C'est sur ce point que réside la confiance portée à RSA. Les limites actuelles de factorisation concernent des nombres de 200 chiffres environ (au moment ou ce livre est rédigé, le record actuel[3] fut établi par Bahr, Boehm, Franke et Kleinjung en mai 2005 dans le cadre du challenge RSA200). On considère donc aujourd'hui un modulo n de 2048 voire seulement 1024 bits comme sûr.

[3] voir `http://www.loria.fr/~zimmerma/records/factor.html` pour les derniers records de factorisation.

Attaques sur RSA

La génération des clefs présentée plus haut est simplifiée, et il convient en pratique de prendre quelques précautions. Ces précautions résultent d'attaques qui exploitent telle ou telle relation sur les paramètres d'un chiffrement RSA. Certaines attaques seront présentées dans la suite sous forme d'exercices.
Les recommandations concernant l'implémentation de systèmes cryptographiques basés sur RSA (générations des clefs etc...) sont résumées dans la norme PKCS#1. Cette norme est régulièrement mise à jour (en fonction des avancées dans la cryptanalyse de RSA).

Exercice 3.17 (Attaque RSA par exposant commun). *William, Jack et Averell ont respectivement les clefs RSA publiques* $(n_W, 3)$, $(n_J, 3)$ *et* $(n_A, 3)$. *Joe envoie en secret à chacun d'eux le même message* x *avec* $0 \leqslant x < \min(n_W, n_J, n_A)$.
Montrer que Lucky Luke, qui voit passer sur le réseau $c_W = x^3 \mod n_W$, $c_J = x^3 \mod n_J$ *et* $c_A = x^3 \mod n_A$ *peut facilement calculer* x.
Indication. *On rappelle (ou on admettra !) que pour* a *et* k *entier, la méthode de Newton-Raphson (faire converger la suite* $u_{i+1} = u_i(1 - \frac{1}{k}) - \frac{a}{k u_i^{k-1}}$*) permet de calculer très rapidement* $\lfloor a^{1/k} \rfloor$, *en temps* $\mathcal{O}(\log^2 a)$. *Solution page 307.*

Exercice 3.18 (Attaque RSA par module commun). *Une implémentation de RSA donne à deux personnes (Alice et Bob) le même nombre* n *(produit de deux nombres premiers) mais des clefs* (e_A, d_A) *et* (e_B, d_B) *différentes. On suppose de plus que* e_A *et* e_B *sont premiers entre eux (ce qui est le plus général). Supposons alors que Alice et Bob chiffrent un même message* M *et que Oscar intercepte les deux messages* $c_A = M^{e_A} \mod n_A$ *et* $c_B = M^{e_B} \mod n_B$ *qu'il sait être deux chiffrements du même message* M.
Montrer qu'Oscar peut alors très facilement découvrir le message M.
Solution page 307.

Exercice 3.19 (Attaque RSA par texte chiffré bien choisi). *Eve intercepte le message* c *chiffré envoyé par Bob à Alice :* $c = m^{e_A} \mod n_A$. *Pour déchiffrer* c, *Eve procède comme suit :*

1. *Eve choisit un entier* $0 < r < n_A$ *au hasard et calcule* $x := r^{e_A} \mod n_A$;
2. *Eve calcule* $y = x.c \mod n_A$;
3. *Eve demande à Alice de signer* y *avec sa clef privée ; Alice renvoie à Eve* $u = y^{d_A} \mod n_A$.

Montrer que Eve peut alors facilement découvrir le message m *émis par Bob (on calculera* $u.r^{-1} \mod n_A$*). Moralité ?* *Solution page 307.*

Exercice 3.20 (Attaque factorielle ou p-1 de Pollard). *Soit p et q les deux nombres premiers secrets utilisés pour la construction du modulo $n = pq$ d'une clef publique RSA. L'entier $p-1$ est pair ; soit B un entier tel que $B!$ (factorielle de B) est multiple de $p-1$. Autrement dit, il existe un entier μ tel que : $\mu(p-1) = B!$*

1. *Soit p_1 un facteur premier de $(p-1)$. Montrer que $p_1 \leqslant B$.*
2. *Soit a un entier $< n$. Montrer que $a^{B!} = 1 \mod p$.*
3. *Soit $A = a^{B!} \mod n$; en déduire que $(A-1)$ est un multiple de p.*
4. *Soit k un entier ; quel est le coût du calcul de $a^k \mod n$?*
5. *En déduire une borne sur le coût du calcul de $A = a^{B!} \mod n$ en fonction de B et $\log n$.*
6. *Si (p-1) est un diviseur de $B!$ avec B petit, alors un attaquant qui connaît n mais pas p ni q peut faire l'attaque suivante : il présuppose un majorant C petit de B ($C \geqslant B$) puis il calcule :*

$$A = 2^{C!} \mod n \qquad et \qquad G = \mathrm{pgcd}(A-1, n).$$

   Montrer que si G est différent de 1 et n alors l'attaquant a forcément cassé RSA avec n comme modulo.
7. *On suppose que $p-1$ est un diviseur de $B!$ avec $B = \alpha \log n$; on suppose de plus α petit (par exemple $\alpha \leqslant 1000$).*
   Montrer que cette attaque permet de casser RSA : on donnera un majorant du coût de cette attaque.
8. *Quelle est la parade ?*

Solution page 307.

3.4.3 Chiffrement El Gamal

Si RSA est basé sur le problème de la factorisation d'entiers, le chiffrement El Gamal exploite le problème du logarithme discret (DLP) abordé dans la section 1.3.3. La génération des clefs s'opère de la façon suivante :

- Soit p un nombre premier tel que le problème du logarithme discret est difficile dans $\mathbb{Z}_p^*$.
- Soit $g \in \mathbb{Z}_p^*$ une racine primitive.
- Soit s un nombre et $\beta = g^s$.
- La clef publique est alors le triplet $K_e = (p, g, \beta)$.
- La clef privée correspond à $K_d = s$.

Chiffrement El Gamal : Soit M le texte clair à chiffrer et $k \in \mathbb{Z}_{p-1}$ un nombre aléatoire secret.

$$E_{K_e}(M) = (y_1, y_2) \text{ avec } \begin{cases} y_1 = g^k \mod p \\ y_2 = M.\beta^k \mod p \end{cases}$$

Déchiffrement El Gamal : Pour $y_1, y_2 \in \mathbb{Z}_p^*$, on définit :

$$D_{K_d}(y_1, y_2) = y_2.(y_1^s)^{-1}$$

En effet, $y_2.(y_1^s)^{-1} = M.\beta^k.(g^{k.s})^{-1} = M.g^{s.k}.(g^{k.s})^{-1} = M.$

Il est donc possible de coder facilement un message par exponentiation modulaire. Le déchiffrement sans la clef privée est équivalent au logarithme discret. Mais en possédant la clef, une exponentiation modulaire, suivie de l'algorithme d'Euclide étendu pour trouver l'inverse d'un nombre dans $\mathbb{Z}$ est suffisant. Le principal intérêt de ce chiffrement est que pour un niveau de sécurité fixé, il nécessite de travailler avec de plus petits entiers, et donc plus rapidement que le chiffrement RSA. En effet, la complexité des attaques sur RSA est en $O(\sqrt[3]{(p)}) = O(\sqrt[6]{(n)})$, avec p un facteur du modulo n, alors qu'elle est au mieux en $O(\sqrt[3]{(p)})$, ou p est le modulo, pour le logarithme discret. La taille du modulo nécessaire pour RSA est donc au moins deux fois plus importante. En outre, de même que DLP peut s'exprimer dans tout groupe, on peut étendre El Gamal sur d'autres groupes où le problème du logarithme discret est encore plus difficile. C'est le cas notamment pour les groupes de points d'une courbe elliptique dans lesquels les attaques par calcul d'index ne fonctionnent pas. L'étude de ces groupes particuliers dépasse néanmoins le cadre de cet ouvrage.

3.5 Authentification, Intégrité, Non-répudiation

En plus du secret, on demande aujourd'hui aux codes cryptographiques d'assurer une fonction d'authentification des messages. Le destinataire doit pouvoir être certain que le message a été émis par l'émetteur qui prétend l'avoir émis. Pour cela, on assure aussi une fonction d'intégrité, à savoir la vérification que le message arrive bien tel qu'il a été émis. On utilise pour cela les fonctions de hachage. La section 1.4.2 page 83 a permis d'introduire ces fonctions. On fournit maintenant quelques analyses plus spécifiques, notamment sur les fonctions de hachage les plus connues. On introduira également la construction de *codes d'authentification*, ou MAC pour *Message Authentification Code*, qui permettent de vérifier à la fois l'intégrité et l'authentification de la source d'une donnée. Enfin, le concept de signature électronique sera présenté avec les principaux schémas de signature utilisés aujourd'hui.

3.5.1 Fonctions de hachage cryptographiques

Les fonctions de hachage ont déjà été introduites au chapitre 1, page 83. La figure 3.16 dresse un rapide historique des principales fonctions de hachages en explicitant les dérivation de construction.

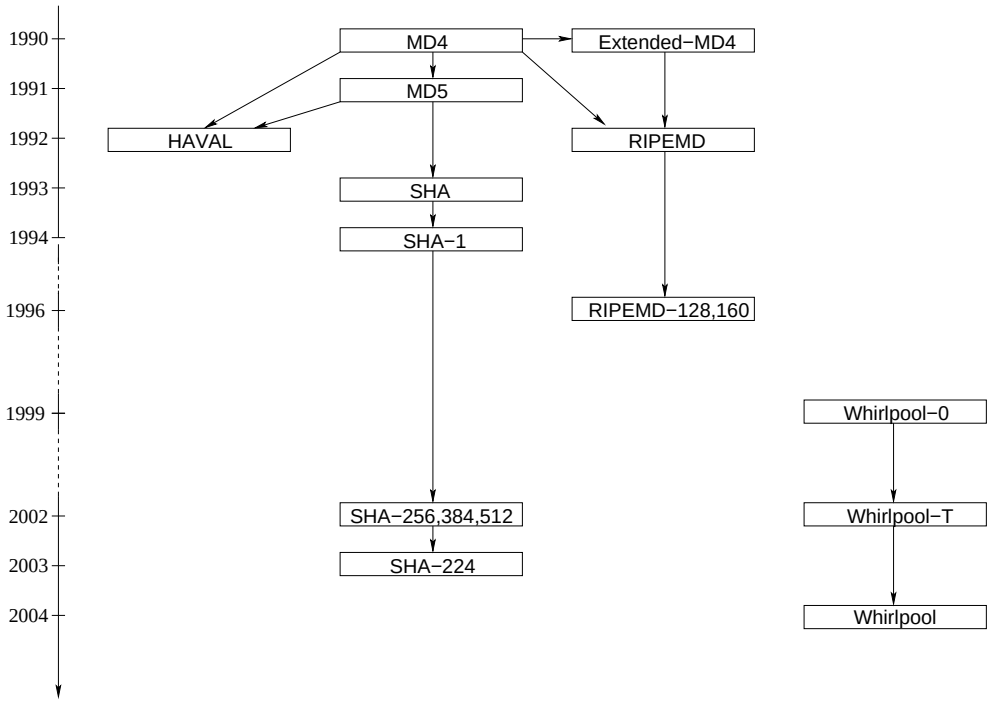


FIG. 3.16: Historique des principales fonctions de hachages.

Les fonctions de hachage cryptographiques les plus connues sont MD5, SHA-1, SHA-256 et Whirlpool. Elles sont toutes sans clefs et non prédictibles (les entrées et les sorties sont pratiquement indépendantes). La propriété de résistance aux collisions (page 84) est particulièrement recherchée pour assurer des schémas de signature ou de chiffrement cryptographiquement sûr. Ainsi, une fonction de hachage sera considérée comme *cassée* lorsqu'il existera un algorithme permettant de trouver des collisions pour cette fonction avec une complexité meilleure que l'attaque de Yuval (en $\mathcal{O}(2^{\frac{n}{2}})$ calculs d'empreintes successifs où n est la taille d'empreinte - voir la section 1.4.3).

Exercice 3.21 (Construction d'une fonction de compression résistante aux collisions). *On se propose de construire une fonction de compression*

$$h : \{0,1\}^{2m-2} \longrightarrow \{0,1\}^m$$

résistante aux collisions. On utilise pour cela une fonction à sens unique (ici, le logarithme discret pour $m = 1024$ bits $= 64$ octets typiquement). Une construction de type Merkle-Damgård fournira alors une fonction de hachage résistante aux collisions. On procède comme suit :

1. *On choisit p et q deux nombres premiers tels que $q = \frac{p-1}{2}$. Autrement dit, $p = 2q+1$. On suppose que p comporte m bits (ainsi q en comportera $m-1$).*
2. *On considère alors les groupes multiplicatifs $\mathbb{Z}_p^*$ et $\mathbb{Z}_q^*$. Soient α et β deux générateurs de $\mathbb{Z}_p^*$ ($\alpha \neq \beta$). Autrement dit :*

$$\mathbb{Z}_p^* = \{\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{p-2}\} = \{\beta^0, \beta^1, \beta^2, \ldots, \beta^{p-2}\}$$

   On pose $\lambda = \log_\alpha(\beta)$ (donc $\alpha^\lambda = \beta \mod p$). On suppose que λ n'est pas connu et extrêmement coûteux à calculer.
3. *On suppose α, β et p connus publiquement et on définit :*

$$\begin{array}{rccl} h: & \mathbb{Z}_q \times \mathbb{Z}_q & \longrightarrow & \mathbb{Z}_p \\ & (x_1, x_2) & \longrightarrow & \alpha^{x_1}\beta^{x_2} \mod p \end{array}$$

a. Application numérique : pour $p = 83$, $q = 41$, $\alpha = 15$ et $\beta = 22$, calculer l'empreinte de $(12, 34)$. Sur quel calcul repose la résistance aux collisions de la fonction h ?

Pour montrer que h est résistante aux collisions, on raisonne par l'absurde :

– *On suppose disposer d'une collision. Autrement dit, on suppose disposer de $x = (x_1, x_2) \in \mathbb{Z}_q^2$ et $y = (y_1, y_2) \in \mathbb{Z}_q^2$ avec $x \neq y$ tel que $h(x) = h(y)$*
– *On montre qu'on peut alors facilement calculer λ.*

Dans toute la suite, on pose $d = \text{pgcd}(y_2 - x_2, p-1)$.

b. Quels sont les diviseurs de $p-1$? En déduire que $d \in \{1, 2, q, p-1\}$.

c. Montrer que $-q < y_2 - x_2 < q$ et en déduire que $d \neq q$.

d. Montrer que $\alpha^{x_1 - y_1} = \beta^{y_2 - x_2} \mod p$.

e. On suppose que $d = p-1$. Montrer que $x = y$. En déduire que $d \neq p-1$.

f. On suppose que $d = 1$. Montrer que $\lambda = (x_1 - y_1)(y_2 - x_2)^{-1} \mod p-1$.

g. On suppose que $d = 2 = \text{pgcd}(y_2 - x_2, 2q)$. On en déduit donc $\text{pgcd}(y_2 - x_2, q) = 1$ et on pose $u = (y_2 - x_2)^{-1} \mod q$.

   1. *Montrer que $\beta^q = -1 \mod p$. En déduire : $\beta^{u(y_2 - x_2)} = \pm\beta \mod p$.*
   2. *Montrer que soit $\lambda = u(x_1 - y_1) \mod p-1$ soit $\lambda = u(x_1 - y_1) + q \mod p-1$.*

h. Conclure en donnant un algorithme qui prend en entrée une collision $x = (x_1, x_2) \in \mathbb{Z}_q^2$ et $y = (y_1, y_2) \in \mathbb{Z}_q^2$ et renvoie λ. Majorer le coût de cet algorithme et en déduire que h est résistante aux collisions.

Solution page 308.

On détaille maintenant les fonctions de hachage les plus connues avec notamment un état des lieu sur leur cryptanalyse.

MD5

MD5 fut proposé par Rivest en 1991. On s'intéresse ici à la fonction de compression utilisée dans MD5. Celle-ci utilise des blocs B de $b = 512$ bits pour produire une empreinte de $n = 128$ bits (en reprenant les notations de la figure 1.12 page 85).
Chaque bloc B est d'abord découpé en 16 sous-blocs $\{B_i\}_{0 \leqslant i \leqslant 15}$ de 32 bits chacun. MD5 utilise également 64 constantes $\{K_i\}_{0 \leqslant i \leqslant 63}$ fixées. L'algorithme travaille avec un état sur 128 bits qui est lui-même divisé en 4 mots de 32 bits : A, B, C et D (initialisés au début avec des constantes). 64 tours sont alors effectuées (et fournissent en sortie l'empreinte résultat de la fonction de compression).
Le détail d'un tour MD5 est donné dans la figure 3.17. Les tours sont divisés en 4 paquets de 16 sous-tours où la fonction F prendra les valeurs successives suivantes :

1. $F = (B \texttt{ AND } C) \texttt{ OR } (\bar{B} \texttt{ AND } D)$
2. $F = (D \texttt{ AND } B) \texttt{ OR } (\bar{D} \texttt{ AND } C)$
3. $F = B \oplus C \oplus D$
4. $F = C \oplus (B \texttt{ OR } \bar{D}))$

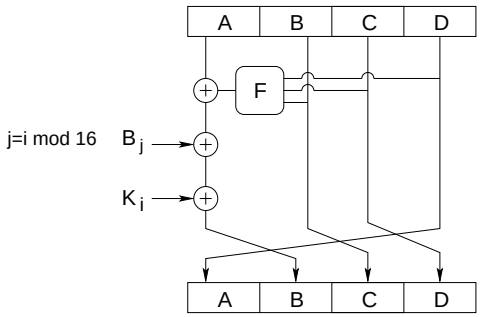


FIG. 3.17: Le i-ème tour dans MD5 ($0 \leqslant i \leqslant 63$).

Sécurité de MD5. La taille d'empreinte dans MD5 est de 128 bits. La recherche de collisions par force brute (attaque de Yuval) a donc une complexité de 2^{64} calculs d'empreintes. Pourtant, plusieurs algorithmes ont été proposés récemment pour trouver des collisions avec MD5 en seulement 2^{42} opérations, voire 2^{30}. C'est pourquoi **MD5 n'est plus considéré comme sûr aujourd'hui**.

SHA-1

SHA-1 fut publié en 1995 par le NIST sous la forme d'une norme. La fonction de compression utilisée dans SHA-1 utilise des blocs B de $b = 512$ bits pour produire une empreinte de $n = 160$ bits (en reprenant les notations de la figure 1.12 page 85). Comme pour MD5, le bloc B est d'abord découpé en 16 sous-blocs $\{B_i\}_{0 \leqslant i \leqslant 15}$ de 32 bits chacun, qui sont ensuite étendus en 80 nouveaux blocs $\{W_i\}_{0 \leqslant i \leqslant 79}$. En outre, 4 constantes $\{K_i\}_{1 \leqslant i \leqslant 3}$ sont fixées.
L'algorithme travaille donc sur un état de 160 bits subdivisés en 5 mots de 32 bits chacun : A, B, C, D et E (initialisés au début avec des constantes). 64 tours sont effectuées (et fournissent en sortie l'empreinte résultat de la fonction de compression). Le détail d'un tour SHA-1 est fourni dans la figure 3.18.

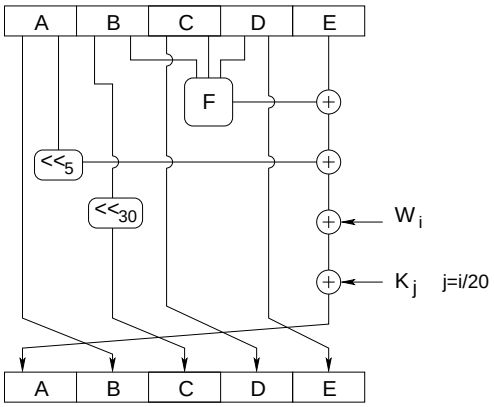


FIG. 3.18: Le i-ème tour dans SHA-1 ($0 \leqslant i \leqslant 79$).

Les tours sont divisées en 4 paquets de 16 sous-tours où la fonction F prendra les valeurs successives suivantes :

1. $F = (B \texttt{ AND } C) \texttt{ OR } (\bar{B} \texttt{ AND } D)$
2. $F = B \oplus C \oplus D$
3. $F = (B \texttt{ AND } C) \oplus (B \texttt{ AND } D) \oplus (C \texttt{ AND } D)$
4. $F = B \oplus C \oplus D$

Sécurité de SHA-1. La taille d'empreinte dans SHA-1 est de 160 bits. La recherche de collisions par force brute (attaque de Yuval) a donc une complexité en 2^{80} calculs d'empreintes. Récemment, une technique similaire à celle utilisée contre MD5 a permis d'obtenir des collisions en seulement $\mathcal{O}(2^{63})$ calculs d'empreinte. On considère donc que **SHA-1 est cassé**, même si l'attaque

reste difficile. De part sa popularité, cette fonction de hachage est encore largement utilisée (notamment au niveau des schémas de signature).

SHA-256

SHA-256 est une amélioration plus robuste (et supportant une taille d'empreinte plus grande) que SHA-1 publiée en 2000. Elle se base sur un chiffrement à clef secrète par bloc (SHACAL-2) et utilise des blocs B de $b = 512$ bits pour produire une empreinte de $n = 256$ bits (cf figure 1.12 page 85). Nous n'entrerons pas ici dans les détails du fonctionnement interne de SHA-256.

Sécurité de SHA-256. La taille d'empreinte dans SHA-256 est de 256 bits. La recherche de collisions par force brute (attaque de Yuval) nécessite 2^{128} calculs d'empreintes. À l'heure actuelle, il n'y a **pas d'attaque efficace connue contre SHA-256** qui est donc considérée comme encore sûre.

Whirlpool

Whirlpool est l'une des fonctions de hachage recommandée par le projet NESSIE (en 2004), dont la fonction de hachage est basée sur une construction de Miyaguchi-Preneel (évoquée à la section 1.4.2). Le chiffrement à clef secrète par blocs sous-jacent utilisé est W, une variante de Rijndael (on rappelle que Rijndael est le nom du chiffrement à clef secrète à la base du standard AES - voir page 158). Le tableau 3.9 résume les principales différences entre les deux fonctions de chiffrement.

	Rijndael	**W**
Taille Bloc	128, 160, 192, 224 ou 256	512
Nb tours	10, 11, 12, 13 ou 14	10
Polynôme de réduction sur $\mathbb{F}_{256}$	$x^8 + x^4 + x^3 + x + 1$ (**0x11B**)	$x^8 + x^4 + x^3 + x^2 + 1$ (**0x11D**)
Origine de la S-Box	$t : a \longrightarrow a^{-1}$ sur $\mathbb{F}_{256}$	structure récursive
Origine des constantes de tour	polynômes x^i sur $\mathbb{F}_{256}$	entrées successives de la S-Box

TAB. 3.9: Principales différences entre W et Rijndael.

La fonction de compression de Whirlpool comporte 10 tours et utilise des blocs B de $b = 512$ bits pour produire une empreinte de $n = 512$ bits (toujours en reprenant les notations de la figure 1.12 page 85). Nous n'entrerons pas ici dans les détails de son implémentation.

Remarque : les noms choisis pour les fonctions de hachage ou de chiffrement ont souvent une origine obscure. À titre indicatif, Whirlpool doit son nom à la galaxie Whirlpool qui fut la première dont on reconnut la structure en spirale.

Sécurité de Whirlpool. Une recherche de collisions sur Whirlpool nécessite 2^{256} calculs d'empreintes. À l'heure actuelle, il n'y a **pas d'attaque efficace connue contre Whirlpool** si bien qu'elle est encore considérée comme sûre.

État des lieux sur la résistance aux collisions pour les principales fonctions de hachage

Comme on l'a vu, l'un des principaux critère de robustesse d'une fonction de hachage est sa résistance aux collisions : elle sera considérée comme sûre s'il n'existe pas d'attaque permettant de trouver des collisions avec une complexité meilleure que celle de l'attaque de Yuval. Le tableau 3.10 résume les résistances aux collisions pour les fonctions de hachage les plus significatives. Il s'agit d'un survol réalisé au moment où cet ouvrage est rédigé et est donc susceptible d'évoluer avec les cryptanalyses futures.

Fonction	**Empreinte**	**Attaque anniversaire**	**Résistance aux collisions**	**Complexité de l'attaque**
MD5	128 bits	$\mathcal{O}(2^{64})$	Cassée	$\mathcal{O}(2^{30})$
SHA-1	160 bits	$\mathcal{O}(2^{80})$	Cassée	$\mathcal{O}(2^{63})$
HAVAL	256 bits	$\mathcal{O}(2^{128})$	Cassée (2004)	$\mathcal{O}(2^{10})$
SHA-256	256 bits	$\mathcal{O}(2^{128})$	**Sûre**	
Whirlpool	512 bits	$\mathcal{O}(2^{256})$	**Sûre**	

TAB. 3.10: Résistance aux collisions pour les fonctions de hachage les plus connues.

Il convient également de noter que la résistance aux collisions n'est pas toujours suffisante. On dénombre ainsi d'autres propriétés sur les fonctions de hachage qui peuvent être requises. On citera par exemple :

- la *résistance aux collisions proches* : il doit être difficile de trouver des collisions (x, x') pour lesquelles la distance de Hamming (définie au chapitre 4) entre les empreintes $d(H(x), H(x'))$ est faible.
- la *résistance aux pseudo-collisions* dans laquelle on autorise la modification du vecteur d'initialisation IV de la fonction de hachage pour trouver des collisions.
- enfin, le *caractère pseudo-aléatoire* où il s'agit de rendre difficile la distinction entre H et une fonction aléatoire.

3.5.2 La signature électronique

Contrôle d'intégrité par les fonctions de hachage

L'intégrité des données sur un canal non sûr peut être assurée par des codes correcteurs d'erreur (voir chapitre 4) mais aussi par des fonctions de hachage. On distingue pour cela trois méthodes :

1. Utiliser un canal sécurisé pour transmettre l'empreinte de façon sûre, le message étant transmis sur le canal non sûr. On est alors capable de vérifier la correspondance entre l'empreinte du message reçu et celle réceptionnée sur le canal fiable. Cette configuration est illustrée dans la figure 3.19.

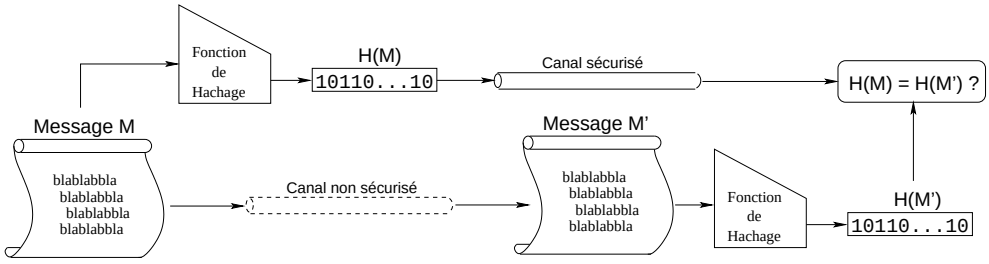


FIG. 3.19: Vérification d'intégrité par des fonctions de hachage en disposant d'un canal sécurisé.

2. Utiliser une fonction de chiffrement E (et la fonction de déchiffrement D associée) pour chiffrer à la fois le message et l'empreinte avant de l'envoyer sur le canal non sûr. Cette configuration est illustrée dans la figure 3.20.

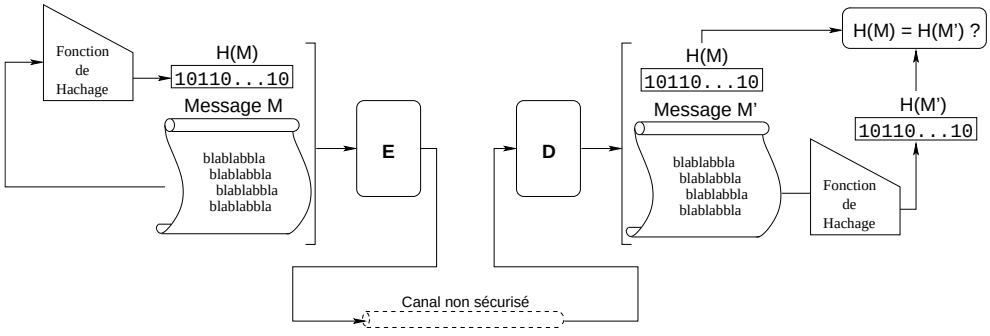


FIG. 3.20: Vérification d'intégrité par des fonctions de hachage à partir d'une fonction de chiffrement.

3. Utiliser un code d'authentification. Cette approche fait l'objet du prochain paragraphe.

Codes d'authentification, ou MAC

On peut utiliser les fonctions de hachage pour faire à la fois de l'authentification et du contrôle d'intégrité de message. On parle alors de MAC, pour *Message Authentification Code* : c'est une fonction de hachage à sens unique paramétrée avec une clef secrète K (on notera du coup $H_K(x)$ le calcul de l'empreinte de x) qui respecte la propriété additionnelle suivante :

$$\text{Pour tout } K \text{ inconnu et pur tout ensemble de paires } \begin{cases} (x_1, H_K(x_1)) \\ (x_2, H_K(x_2)) \\ \vdots \\ (x_i, H_K(x_i)) \end{cases}$$

il est impossible (en temps raisonnable) de calculer une autre paire $(x, H_K(x))$ valide pour $x \notin \{x_1, x_2, ..., x_i\}$.
Pour résumer, seul celui qui possède la clef K peut calculer et donc vérifier une empreinte :
– Alice et Bob partagent une clef K.
– Alice envoie à Bob un message M et $r = H_K(M)$.
– Bob vérifie que le r reçu est bien égal à $H_K(M')$ pour M' reçu.
Cette configuration est illustrée dans la figure 3.21. Les MAC permettent de garantir à la fois l'intégrité du message transmis (tous les possesseurs de K peuvent vérifier que M n'a pas été modifié puisqu'il correspond à l'empreinte r) mais aussi l'authentification de la source de donnée : sous réserve que Alice et Bob soient les seuls à partager la clef K, ils sont les seuls à pouvoir générer l'empreinte $H_K(M)$.

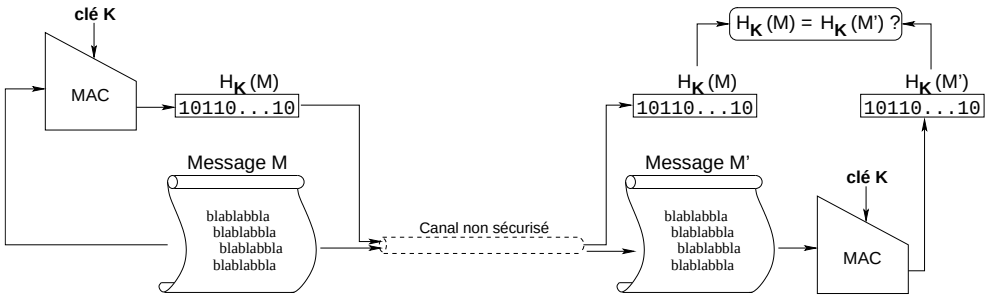


FIG. 3.21: Principe d'utilisation d'un MAC.

Remarque : si un MAC fournit l'authentification et l'intégrité, il ne garantit évidemment pas la confidentialité du message. Il faudra pour cela le coupler avec d'autres techniques. En pratique, on cherche à rendre le cassage du MAC aussi difficile que le cassage de la fonction de hachage elle-même.

Construction d'un MAC. Il y a principalement deux façons de réaliser des MAC :

– À partir d'un chiffrement à clef secrète par blocs utilisé en mode CBC ou CFB (l'empreinte est alors le dernier bloc chiffré, chiffré une fois encore en mode CBC ou CFB). Cette opération (pour le mode CBC) est illustrée dans la figure 3.22.

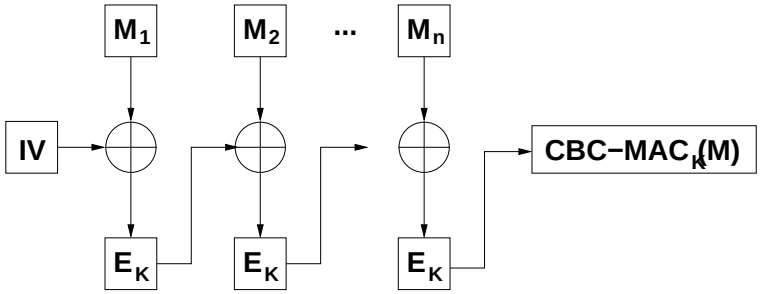


FIG. 3.22: Code d'authentification par chiffrement à clef secrète.

Sur un Pentium III cadencé à 500 MHz, une implémentation logicielle du CBC-MAC-AES (c'est-à-dire un CBC-MAC où les chiffrements E_k sont réalisés par l'algorithme Rijndael, standardisé sous le nom AES) peut produire des signatures de message à la vitesse de 234 Mbits/seconde. La même implémentation de AES seul tourne à une vitesse de 275 Mbits/seconde. Le coût de calcul de l'empreinte est donc ainsi très voisin de celui du cryptage.

– À partir d'une fonction de hachage à sens unique, l'idée est alors d'intégrer la clef K dans le calcul d'empreinte. Plusieurs solutions sont possibles mais toutes ne conviennent pas. On veillera ainsi à **ne pas utiliser** (sur un modèle de Merkle-Damgård pour la fonction de hachage) les construction suivantes :
  – $H_K(M) = H(K||M)$
  – $H_K(M) = H(M)$ avec $IV = K$
  – $H_K(M) = H(M||K)$

Exercice 3.22. *Expliquer pourquoi ces trois constructions ne permettent pas de fournir des MAC valides.* *Solution page 309.*

On préférera donc les constructions suivantes, avec deux clefs K_1 et K_2 :
– Enveloppe : $H(K_1||M||K_2)$
– NMAC : $H_{K_1}(H_{K_2}(M))$
– Hybride : $H_{K_1}(x||K_2)$

Les signatures électroniques

Les signatures manuscrites ont longtemps été utilisées pour prouver l'identité de leur auteur ou du moins l'accord du signataire avec le contenu du document. Avec des documents numériques, il faut inventer des protocoles qui remplacent l'empreinte manuelle par un équivalent. On remarquera déjà que la signature électronique devra dépendre du signataire **et** du document. Autrement, un simple copier/coller (particulièrement facile avec un document numérique) permettrait d'obtenir une signature valide.
Ainsi, une signature électronique devra être :
– *authentique* : elle convainc le destinataire que le signataire a délibérément signé un document ;
– *infalsifiable* ;
– *non réutilisable* : elle est attachée à un document donné et ne pourra pas être utilisée sur un document différent ;
– *inaltérable* : toute modification du document doit être détectable ;
– *non reniable* : le signataire ne peut répudier le document signé.
Pour reprendre les fonctionnalités offertes par la cryptographie (3.1.2), on voit que les signatures électroniques fournissent l'intégrité, l'authentification mais aussi la non-répudiation.
Comme pour les algorithmes de chiffrement, on distingue deux classes de signatures :

1. les signatures symétriques qui utilisent un cryptosystème à clef secrète. Dans ce cas, il convient de disposer soit d'un arbitre, soit d'une clef secrète qui sera utilisée dans un MAC en mode CBC ou CFB comme cela a été présenté dans la section précédente.
2. les signatures asymétriques qui utilisent un cryptosystème à clef publique et une fonction de hachage publique (sans clef) comme SHA-256 ou Whirlpool.

Le principe général de cette seconde approche est détaillé dans la figure 3.23 (Cette figure n'explicite pas les algorithmes de signature et de vérification utilisés ; les plus connus seront détaillés dans la suite). On préfère en effet signer l'empreinte d'un document qui a une taille fixe suffisamment petite pour être utilisée efficacement par un cryptosystème à clef publique. Ainsi, lorsque Alice signe le document M, elle utilise $h_M = H(M)$ l'empreinte de M, sa clef secrète K_d et enfin la fonction de déchiffrement D. Elle génère ainsi la

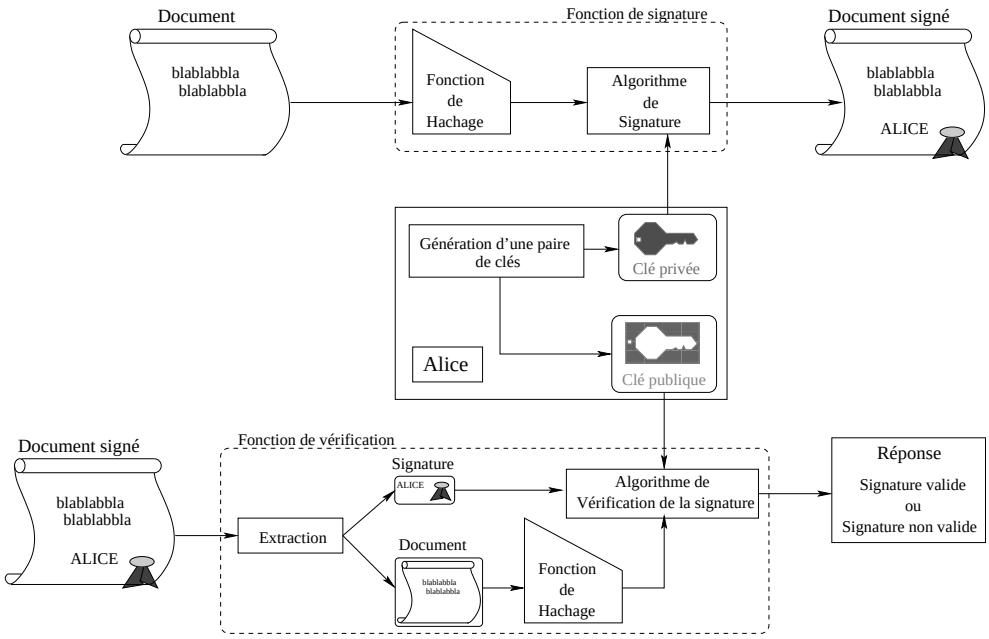


FIG. 3.23: Principe de la signature à clef publique et de sa vérification.

signature $s(M) = D_{K_d}(h_M)$ (on notera que seule Alice a pu générer $s(M)$). Le document signé consiste alors en la paire $[M, s(M)]$.
Ce document peut être vérifié par Bob à partir de la clef publique K_e d'Alice et la fonction de chiffrement E (également publique). Il lui suffit de calculer l'empreinte du document h_M et de vérifier si $E_{K_e}(s(M)) = h_M$.
On détaille maintenant les deux schémas de signature les plus courants.

Signature RSA

Il est en effet possible d'utiliser le système RSA pour définir une signature digitale. Il suffit pour cela d'appliquer RSA à l'envers.
La génération des paramètres est identique à celle de RSA. Prenons donc $K_e = (e, n)$ la clef publique d'Alice, et d son exposant secret.
Lorsque Alice souhaite signer un document M, il lui suffit d'effectuer les opérations suivantes :
- elle calcule l'empreinte $h_M = H(M)$. On supposera $0 \leqslant h_M < n$.
- elle génère la signature $s(M) = (h_M)^d \mod n$ (encore une fois, on aura remarqué qu'elle est la seule à pouvoir générer cette signature étant la seule à posséder d).
- Le document signé est alors la paire $[M, s(M)]$ et est envoyé à Bob.

Ainsi, Bob reçoit un document signé $[\tilde{M}, s(M)]$ d'Alice. Ce document est potentiellement altéré voire illégitime. Bob récupère alors la clef publique d'Alice (n, e) (la section 3.6.5 est consacrée aux protocoles d'échanges de clefs). Il calcule alors $\tilde{h_M} = H(\tilde{M})$ et vérifie l'identité

$$s(M)^e = \tilde{h_M} \mod n$$

En effet : $s(M)^e = (h_M)^{e.d} \mod n = h_M \mod n = h_M$ et si le document est authentique : $h_M = \tilde{h_M}$.
La sécurité de cette signature est donc celle du cryptosystème RSA. On notera quand même que cette présentation reste simplifiée et en l'état sujette à des attaques.

Standard DSS - *Digital Signature Standard*

Ce schéma de signature est basé sur le problème du logarithme discret.Le système comprend des paramètres partagés q, p et g ainsi qu'une clef publique y et une clef privée x qui sont générés de la façon suivante :

1. Choisir q premier de 160 bits.
2. Trouver $p = kq + 1$ premier de 512 à 1024 bits.
3. Tant que $a = 1$, choisir g et calculer $a \leftarrow g^{\frac{p-1}{q}} \mod p$. *($g$ est alors d'ordre au plus q).*
4. Choisir x de 160 bits.
5. $y \leftarrow g^x \mod p$.

Une fois les paramètres construits, la signature se fait de la façon suivante :

1. Alice choisit k aléatoire inférieur à q.
2. Alice calcule et envoie la signature de son message m à l'aide d'une fonction de hachage h (pour le DSS, il s'agit de SHA-1) :

$$r = (g^k \mod p) \mod q \quad et \quad s = (k^{-1}[h(m) + xr]) \mod q$$

3. Bob vérifie la signature si et seulement si $v = r$ pour :
   - $w = s^{-1} \mod q$;
   - $u_1 = h(m)w \mod q$;
   - $u_2 = rw \mod q$;
   - $v = ([g^{u_1} y^{u_2}] \mod p) \mod q$;

Théorème 21. *La vérification est correcte.*

Preuve. Le paramètre s est construit de sorte que $k = (s^{-1}[h(m) + xr]) \mod q$. Or $g^{u_1}y^{u_2} = g^{u_1+xu_2}$ donc, d'après la donnée de u_1 et u_2, et comme $g^q = 1 \mod p$ par construction, on a donc $g^{u_1}y^{u_2} = g^{h(m)w+xrw}$. Or, comme $w = s^{-1}$, la remarque de début de preuve nous donne $g^{u_1}y^{u_2} = g^k$ ou encore $v = r \mod q$. □

Exercice 3.23. *Parallèlement au DSS développé aux États-Unis, les Russes avaient développé leur propre signature : GOST. Les paramètres p, q, g, x et y sont les mêmes sauf que q est premier de 254 à 256 bits et p premier entre 509 et 512 bits (ou entre 1020 et 1024 bits). Seule la partie s de la signature change :*

1. *Alice calcule et envoie*

$$r = (g^k \mod p) \mod q \quad et \quad s = (xr + kh(M)) \mod q$$

2. *Bob vérifie la signature si et seulement si $u = r$ pour :*
   - $v = h(M)^{q-2} \mod q$.
   - $z_1 = sv \mod q$.
   - $z_2 = ((q - r)v) \mod q$.
   - $u = ((g^{z_1}y^{z_2}) \mod p) \mod q$.

Justifier la vérification. *Solution page 309.*

3.6 Protocoles usuels de gestion de clefs

Chaque étape d'un processus d'échange de messages cryptés est cruciale, parce que susceptible d'être la cible d'attaques. En particulier, si les systèmes à clef secrète sont de bons systèmes cryptographiques, leur faiblesse réside en bonne partie dans le processus d'échange de clefs. Des protocoles ont été inventés pour assurer la sécurité de ces échanges.

3.6.1 Génération de bits cryptographiquement sûre

Nous avons vu différents générateurs pseudo-aléatoires en section 1.3.6, ainsi que différentes attaques permettant de prédire le comportement de certains de ces générateurs en sections 1.4.3 et 1.4.3. Un exemple de générateur sûr est celui de Blum, Blum et Shub : en prenant p et q premiers assez grands congrus à 3 modulo 4 et en posant x_0 le carré d'un entier aléatoire modulo $n = pq$, ce générateur calcule la suite des carrés successifs $x_{i+1} = x_i^2 \mod n$. Or, il s'avère que découvrir le bit de poids faible des x_i suite du bit de poids faible de ces x_i est aussi difficile que casser la factorisation. Plus généralement, il est possible d'utiliser RSA pour fabriquer des séquences pseudo-aléatoires cryptographiquement sûres, comme dans l'exercice suivant :

Exercice 3.24 (Générateur de bits pseudo-aléatoires RSA). *Soient n et e le modulo et l'exposant d'un code RSA. Soit $k = \mathcal{O}(\log(\log n))$ un entier. Alors, connaissant $c = x^e \mod n$, découvrir les k premiers bits de x est aussi difficile que de découvrir la clef secrète associée à n et e.*
Décrire une procédure prenant k en entrée et générant une suite de l bits pseudo-aléatoire et cryptographiquement sûre. *Solution page 309.*

3.6.2 Protocole d'échange de clef secrète de Diffie-Hellman et attaque *Man-in-the-middle*

L'utilisation d'une fonction à sens unique comme l'exponentiation modulaire permet le partage de clef.
Supposons donc qu'Alice et Bob souhaitent partager une clef secrète K. Ils conviennent d'abord d'un entier premier p et d'un générateur g de $\mathbb{Z}_p^*$. On peut alors définir le protocole suivant en deux étapes leur permettant de construire K en secret :

1. (a) Seule, Alice choisit un nombre $a \in \mathbb{Z}_p^*$ secret et calcule
$$A = g^a \mod p$$
Elle envoie A à Bob.
   (b) Symétriquement, Bob choisit un nombre $b \in \mathbb{Z}_p^*$ secret et calcule
$$B = g^b \mod p$$
Il envoie B à Alice.
2. (a) Alice calcule seule $K = B^a \mod p$.
   (b) Symétriquement, Bob calcule de son côté $A^b \mod p$.

À la fin, Alice et Bob partagent la même clef secrète $K = g^{a.b} \mod p$ sans l'avoir jamais communiquée directement.

Exercice 3.25. *On suppose qu'Oscar voit passer A et B. Expliquer pourquoi Oscar ne peut alors pas facilement en déduire K.* *Solution page 309.*

Exercice 3.26. *Alice et Bob conviennent des paramètres suivants : $p = 541$ et $g = 2$. Alice génère le nombre secret $a = 292$. De son coté, Bob génère le nombre $b = 426$. Quelle est la clef secrète résultant du protocole d'échange de Diffie-Hellman ?* *Solution page 309.*

L'attaque la plus classique des protocoles d'échange consiste pour Oscar à couper la ligne communication entre Alice et Bob avant qu'ils ne commencent le partage de la clef (émission de A et B). C'est l'attaque dite *« Man-in-the-middle »*.

L'idée générale est qu'Oscar se fait passer pour Bob auprès d'Alice et simultanément pour Alice auprès de Bob. Ainsi, Oscar peut alors lire toutes les communications entre Alice et Bob avec la clef qu'ils pensent avoir construite en secret.
En pratique Oscar fabrique a' et b'. Il intercepte ensuite A et B puis fabrique $K_A = A^{b'}$ et $K_B = B^{a'}$. Il envoie ensuite $g^{b'}$ à Alice et $g^{a'}$ à Bob. L'évolution de l'attaque est illustrée dans la table suivante :

Alice		Oscar		Bob
génère a		génère b' et a'		génère b
$A = g^a \mod p$		$B' = g^{b'} \mod p$		$B = g^b \mod p$
		$A' = g^{a'} \mod p$		
	$\xrightarrow{A}$		$\xleftarrow{B}$	
	$\xleftarrow{B'}$		$\xrightarrow{A'}$	
(connaît $[a, A, B']$)				(connaît $[b, A', B]$)
Clef secrète :		Clefs secrètes :		Clef secrète :
$K_A = B'^a \mod p$		K_A et K_B		$K_B = A'^b \mod p$

Tab. 3.11: Attaque Man-in-the-middle dans le protocole d'échange de clef de Diffie-Hellman.

Ensuite, Oscar doit décoder à la volée les messages envoyés par Alice ou Bob, puis les recoder avec l'autre clef de la manière décrite dans le tableau 3.12.

Alice		Oscar		Bob
$C = E_{K_A}(M)$				
	$\xrightarrow{C}$			
		$M = E_{K_A}(C)$		
		$\Gamma = E_{K_B}(M)$		
			$\xrightarrow{\Gamma}$	
				$M = E_{K_B}(C)$

Tab. 3.12: Envoi de message intercepté par le Man-in-the-middle.

Ainsi, Alice et Bob croient s'envoyer des messages secrets, alors que Oscar les intercepte et les décrypte au milieu ! Ainsi, Oscar est non seulement capable d'obtenir en clair tous les échanges entre Alice et Bob, mais en outre il peut modifier ces échanges. En effet, rien de l'empêche, au milieu, de remplacer le message envoyé par Alice par une information de son cru.
Dans la suite, nous décrivons différents protocoles à clef secrète et à clef publique développés pour parer cette attaque.

3.6.3 Kerberos : un distributeur de clefs secrètes

Pour se mettre d'accord sur les fonctions de chiffrement et déchiffrement E et D, la solution la plus sûre est qu'Alice et Bob se rencontrent physiquement. Le protocole de Diffie-Hellman permet d'éviter cette rencontre physique, mais sa faiblesse est qu'un adversaire actif placé entre Alice et Bob peut intercepter toutes les communications.
Une solution pour éviter de trop nombreux échanges, tout en se prémunissant contre l'attaque *« Man-in-the-middle »*, est d'introduire une tierce personne.
Développé par le *Massashusetts Institute of Technilogy*, Kerberos est un système d'authentification sécurisé à tierce personne de confiance (ou TA pour *Trusted Authority*) conçu pour les réseaux TCP/IP. Il ne s'agit pas en revanche d'un système d'autorisation d'accès aux ressources, bien que des extensions existent en ce sens. La distribution MIT de Kerberos est libre.
Ce système partage avec chaque entité U du réseau une clef secrète K_U (un mot de passe dans le cas d'un utilisateur) et la connaissance de cette clef tient lieu de preuve d'identité. L'authentification est négociée par le biais d'un tiers de confiance : le centre de distribution des clefs, *« Key Distribution Center » (KDC)*. Kerberos regroupe un certain nombre de fonctionnalités :
– l'authentification est sécurisée ;
– il n'y a pas de transmission de mot de passe sur le réseau ;
– l'authentification est unique : l'utilisateur n'entre son mot de passe qu'une seule fois (pour une période de temps donnée, 1 jour par exemple) pour se connecter plusieurs fois aux services distant (c'est un principe appelé « *Single Sign On* »en anglais).
– l'authentification est gérée de façon centralisée ;
– c'est un standard supporté par de nombreux systèmes d'exploitation (en particulier *Windows*®).

Kerberos identifie les utilisateurs du système (une personne physique, un serveur ou un service) par un triplet <*nom, rôle, domaine*> (ou *domaine* identifie le domaine d'administration associé à au moins un serveur Kerberos), plus précisément sous la forme d'une chaîne `nom/rôle@domaine`. Ainsi, une personne physique sera identifiée par la chaîne **login/staff@DOMAIN**. Dans le cadre d'un service, on préférera référencer le serveur qui fournit le service : on identifiera ce dernier par la chaîne **service/hostname@DOMAIN**.

Présentation générale du protocole Kerberos

Kerberos est fondé sur l'utilisation de *tickets* qui serviront à convaincre une entité de l'identité d'une autre entité. Il crée également des *clefs de session* qui sont données à deux participants et qui servent à chiffrer les données entre ces deux participants, suivant les étapes de la figure 3.24.

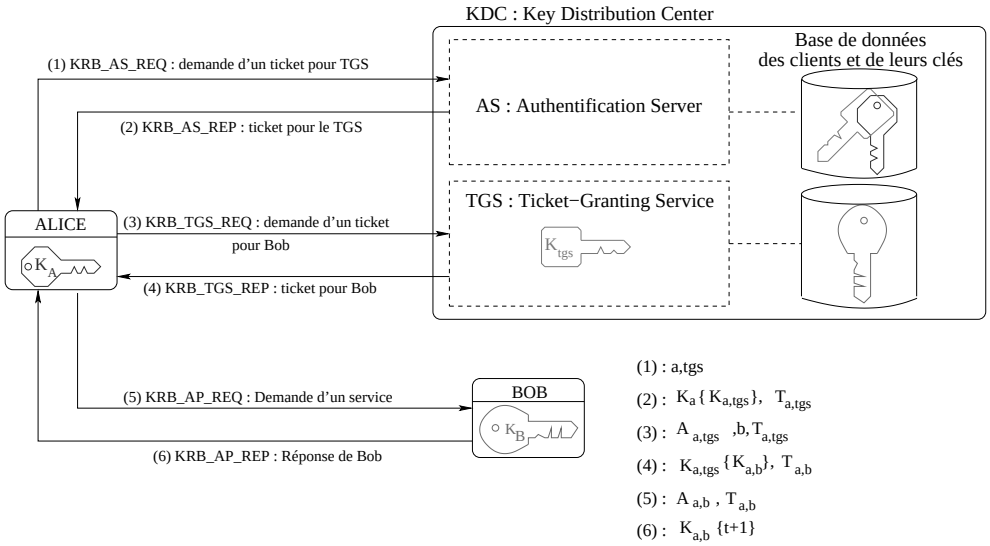


FIG. 3.24: Les étapes d'authentification Kerberos.

On distingue deux types d'accréditations :
– les **tickets** qui servent à donner au futur destinataire, (Bob ou le TGS, *Ticket-Granting Service*) de manière sûre, l'identité de l'expéditeur (Alice) à qui le ticket a été fourni. Il contient également des informations que le destinataire peut utiliser pour s'assurer que l'expéditeur qui utilise le ticket est bien celui à qui le ticket a été délivré.
– les **authentifiants** qui sont des accréditations supplémentaires présentées avec le ticket.

Un ticket prendra la forme suivante :

$$T_{a,s} = s, E_{K_s}(id_a, t, t_{end}, K_{a,s})$$

Il contient donc le nom du service s qu'Alice souhaite utiliser (TGS ou Bob) ainsi qu'une liste d'informations chiffrées avec la clef secrète du service (qu'il sera donc le seul à pouvoir déchiffrer), plus précisément :
– l'identité d'Alice id_a,
– la date de la demande t,
– la date de fin de validité du ticket t_{end},
– enfin (et surtout) une clef de session $K_{a,s}$ qui sera utilisée d'une part pour l'authentifiant (voir ci-après) et d'autre part pour chiffrer les futures communications entre Alice et le service s.

Bien qu'Alice ne puisse pas déchiffrer le ticket (il est chiffré avec la clef secrète K_s), elle peut néanmoins le donner chiffré à s.

Parallèlement aux tickets, Kerberos utilise donc des authentifiants qui revêtent la forme suivante :

$$A_{a,s} = E_{K_{a,s}}(id_a, t)$$

Alice l'engendre chaque fois qu'elle veut utiliser un service s (TGS ou Bob). Contrairement au ticket qu'Alice peut utiliser plusieurs fois pour accéder au service jusqu'à l'expiration du ticket, un authentifiant est unique et ne peut être utilisé qu'une seule fois. On notera cependant que dans la mesure où Alice possède la clef de session $K_{a,s}$, elle sera capable d'en générer autant de fois que nécessaire.

Détail des messages Kerberos

La figure 3.24 présente les différents messages qui sont échangés. Pour résumer :
- Les messages 1 et 2 permettent l'obtention du premier ticket qu'Alice devra présenter ensuite au TGS à chaque fois qu'elle souhaite contacter un destinataire.
- Les messages 3 et 4 permettent l'obtention d'un ticket de service qu'Alice devra présenter à Bob pour une demande de service.
- Les messages 5 et 6 correspondent à la demande de service qu'Alice formule à Bob et la réponse de ce dernier. Cette étape permet comme on va le voir de garantir l'authentification mutuelle d'Alice et de Bob et de leur fournir une clef de session qui leur permettra de chiffrer leurs futurs messages. C'est en ce sens qu'il faut prendre la notion de service.

On détaille maintenant le contenu explicite de ces messages :

1. KRB_AS_REQ $= [a\,;\,tgs]$: ce message sert simplement d'introduction à Alice. Elle y précise son nom et quel TGS elle souhaite rencontrer[4].
2. KRB_AS_REP $= [E_{K_a}(K_{a,tgs})\,;\,T_{a,tgs}]$: le serveur d'authentification[5] cherche le client dans sa base de données. S'il le trouve, il engendre une clef de session $K_{a,tgs}$ qui devra être utilisée entre Alice et le TGS. Cette clef est d'une part chiffrée avec la clef secrète K_a d'Alice[6] : c'est la première partie du message ($E_{K_a}(K_{a,tgs})$). Ensuite, il crée un ticket $T_{a,tgs}$ pour Alice afin que celle-ci puisse s'authentifier auprès du TGS. Comme on l'a déjà vu, ce ticket est chiffré avec la clef secrète K_{tgs} du TGS. Alice ne pourra pas le déchiffrer mais pourra le présenter tel quel à chaque requête au TGS. Dans ce cas particulier, le ticket est appelé TGT. Il est important de noter que seule la véritable Alice est capable de récupérer la clef de session $K_{a,tgs}$ (elle est la seule à posséder la clef

[4] Il peut y en avoir plusieurs.
[5] AS sur la figure 3.24.
[6] S'il s'agit d'un utilisateur humain, K_a correspond au hachage de son mot de passe.

secrète K_a). Ainsi, Alice dispose maintenant de la clef de session $K_{a,tgs}$ et du TGT $T_{a,tgs}$.

3. KRB_TGS_REQ = $[A_{a,tgs}\,;\,b\,;\,T_{a,tgs}]$: Alice doit maintenant obtenir un nouveau ticket pour chaque Bob qu'elle souhaite contacter. Pour cela, Alice contacte le TGS en lui fournissant d'une part le ticket TGT $T_{a,tgs}$ qu'elle possède déjà, et un authentifiant $A_{a,tgs}$ d'autre part (en plus du nom du serveur qu'elle souhaite contacter). L'authentifiant possède des informations formatées vérifiables à partir du ticket par le TGS et comme ces informations sont chiffrées avec la clef de session $K_{a,tgs}$, cela prouve au moins qu'Alice la connaît et donc l'authentifie (d'où le nom d'authentifiant donné à $A_{a,tgs}$).
4. KRB_TGS_REP = $[E_{K_{a,tgs}}(K_{a,b})\,;\,T_{a,b}]$: grâce à sa clef secrète K_{tgs}, le TGS déchiffre le ticket, récupère la clef de session $K_{a,tgs}$ et peut ainsi déchiffrer l'authentifiant $A_{a,tgs}$. Il compare le contenu de l'authentifiant avec les informations contenues dans le ticket et, si tout concorde (Alice est authentifiée), il peut engendrer une clef de session $K_{a,b}$ (qui sera utilisée entre Alice et Bob) qu'il chiffre avec la clef de session $K_{a,tgs}$ et un nouveau ticket $T_{a,b}$ qu'Alice devra présenter à Bob. Après réception de ce message et déchiffrement, Alice dispose donc en plus de $K_{a,tgs}$ et de $T_{a,tgs}$ (qu'elle conserve jusqu'à expiration du ticket pour dialoguer avec TGS) d'une nouvelle clef de session $K_{a,b}$ et d'un nouveau ticket $T_{a,b}$ qu'elle pourra utiliser avec Bob.
5. KRB_AP_REQ = $[A_{a,b}\,;\,T_{a,b}\,]$: maintenant, Alice est prête à s'authentifier auprès de Bob ; cela s'effectue de la même manière qu'entre Alice et le TGS[7] (cf message KRB_TGS_REQ).
6. KRB_AP_REP = $[E_{K_{a,b}}(t+1)]$: Bob doit maintenant s'authentifier en prouvant qu'il a pu déchiffrer le ticket $T_{a,b}$ et donc qu'il possède la clef de session $K_{a,b}$. Il faut qu'il renvoie une information vérifiable par Alice chiffrée avec cette clef. En vérifiant cela, Alice est maintenant sûre de l'identité de Bob et dispose d'une clef de session $K_{a,b}$ utilisable pour chiffrer les communications entre Alice et Bob.

Les faiblesses de Kerberos

– *Attaque par répétition* : bien que les datations soient supposées éviter cela, les messages peuvent être rejoués pendant la durée de vie des tickets (environ 8 heures par défaut).
– *Services de datation* : les authentifiants dépendent du fait que toutes les horloges du réseau soient plus ou moins synchronisées. Si l'on peut tromper

[7] Bob n'est rien d'autre qu'un serveur particulier.

un ordinateur quant à l'heure réelle, alors les anciens authentifiants peuvent être rejoués. La plupart des protocoles de maintien du temps en réseau ne sont pas sûrs, ce qui peut donc être un sérieux défaut.

– *Paris de mots de passe* : il s'agirait pour un intrus de collectionner les premières moitiés du message KRB_AS_REP ($E_{K_a}(K_{a,tgs})$) pour prévoir la valeur de K_a (en général, $K_a = H(Password)$). En pariant sur le mot de passe $\tilde{P}$, l'intrus peut calculer $\tilde{K}_a$, déchiffrer et obtenir $\tilde{K}_{a,tgs}$ et vérifier la pertinence de son choix en déchiffrant l'authentifiant $A_{a,tgs} = E_{K_{a,tgs}}(id_a, t)$ dont il connaît le contenu (il connaît au moins id_a).
– *Usurpation de login* : (ou *« spoofing »* en anglais) on peut envisager une attaque où tous les logiciels Kerberos clients sont remplacés par une version qui non seulement réalise le protocole Kerberos mais enregistre également les mots de passe.

Enfin, le KDC possède les clefs de tous les utilisateurs et de tous les serveurs. Si cela permet de réduire le nombre total de clefs secrètes (tous les utilisateurs devraient sinon avoir une clef pour chaque serveur et vice-versa), cela induit que si le KDC est attaqué avec succès, toutes les clefs doivent être changées. Les architectures à clefs publiques permettent de pallier ce dernier problème.

Exercice 3.27 (Attaque du protocole de Needham-Schroeder). *Alice et Bob veulent s'échanger une clef de session et Ivan partage une clef secrète I_1 avec Alice et une clef secrète I_2 avec Bob. Le protocole de Needham et Schroeder est le suivant :*

1. *Alice envoie $(Alice||Bob||Random_1)$ à Ivan.*
2. *Ivan crée une clef de session K et envoie $E_{I_1}(Random_1||Bob||K||E_{I_2}(K||Alice))$ à Alice.*
3. *Alice envoie $E_{I_2}(K||Alice)$ à Bob.*
4. *Bob envoie $E_K(Random_2)$ à Alice.*
5. *Alice envoie $E_K(Random_2 - 1)$ à Bob.*

a. *À quoi sert $Random_1$?*

b. *À quoi sert $Random_2$ et pourquoi Alice renvoie-t-elle $Random_2 - 1$?*

c. *Un problème de sécurité avec ce protocole est que les vieilles clefs de session ont de la valeur. Si Eve est capable d'écouter les messages échangés et a pu obtenir une ancienne clef de session, indiquer comment elle peut convaincre Bob qu'elle est Alice.*

d. *Que faudrait-il ajouter aux messages pour éviter cet écueil ? Est-ce suffisant et à quoi ressemble le protocole ainsi obtenu ?*

e. *Dans le système obtenu, un utilisateur n'a pas besoin de s'authentifier auprès du KDC chaque fois qu'il désire accéder à un service. Pourquoi ? Quelle est la faiblesse de cette méthode ?*

f. *Dans ce système, Ivan possède une liste des clefs symétriques (ou des empreintes des mots de passe) de tous ses clients. Que pourrait-il se passer en cas de vol de cette liste ? Comment se prémunir contre un tel danger ?*

Solution page 309.

Pour pallier le problème du vol des clefs, une solution serait de préférer un système à clef publique pour l'authentification. Ce système est proposé dans la section suivante.

3.6.4 Authentification à clef publique

Supposons que A veuille envoyer un message secret M à B. A doit alors avoir accès à E_B, la fonction de transformation publique de B. A va chiffrer M, $C = E_B(M)$ et envoyer le résultat C à B.
À la réception, B va utiliser sa fonction privée de transformation D_B pour déchiffrer. Si la transmission de A est espionnée, l'intrus ne pourra pas déchiffrer C car la fonction D_B est secrète donc la confidentialité est assurée. Par contre, comme E_B est publique, B n'a aucun moyen de connaître l'identité de l'envoyeur. De même le message envoyé par A peut être altéré. Les propriétés d'authentification et d'intégrité ne sont donc pas assurées. Pour qu'elles le soient, les transformations doivent satisfaire la propriété $E(D(M)) = M$. En effet, supposons que A veuille envoyer un message authentifié M à B. Cela signifie que B doit pouvoir vérifier que le message a bien été envoyé par A et qu'il n'a pas été altéré en chemin. Pour cela A va utiliser sa transformation privée D_A, calculer $M' = D_A(M)$ et envoyer M' à B. B peut alors utiliser la transformation publique E_A pour calculer $E_A(M') = E_A(D_A(M)) = M$.
En supposant que M représente un texte « valide » (au sens du protocole utilisé), B est sûr que le message a été envoyé par A et n'a pas subi d'altérations pendant le transport. Cela vient de la nature unidirectionnelle de E_A : si un attaquant pouvait, à partir d'un message M, trouver M' tel que $E_A(M') = M$ cela signifierait qu'il peut calculer l'inverse de la fonction E_A ce qui est une contradiction. Par contre dans ce cas le secret n'est pas assuré car tout le monde peut accéder à E_A et donc déchiffrer le message.
L'exercice suivant contient le principe de l'authentification à clef publique. Il est assez simple pour que vous puissiez le découvrir à ce stade, en fonction des contraintes qui viennent d'être décrites.

Exercice 3.28 (Principe de l'authentification à clef publique). *En utilisant les fonctions E_A, D_A, E_B et D_B trouver un couple de fonctions de chiffrement et déchiffrement satisfaisant à la fois le secret et l'authentification.*

Solution page 310.

Exercice 3.29 (Cryptographie RSA et authentification). *Un professeur envoie ses notes au secrétariat de l'école par mail. La clef publique du professeur est* $(e_P, n_P) = (3, 55)$*, celle du secrétariat* $(e_S, n_S) = (3, 33)$*.*

1. *Déterminer la clé privée du professeur et du secrétariat de l'école.*
2. *Pour assurer la confidentialité de ses messages, le professeur chiffre les notes avec la clef RSA du secrétariat. Quel message chiffré correspond à la note 12 ?*
3. *Pour assurer l'authenticité de ses messages, le professeur signe chaque note avec sa clé privée et chiffre le résultat avec la clef RSA du secrétariat. Le secrétariat reçoit ainsi le message 23. Quelle est la note correspondante ?*

Solution page 311.

3.6.5 Infrastructures à clefs publiques : PGP et PKI X

Une architecture PKI (pour *Public Key Infrastructure*) est un ensemble d'infrastructures permettant de réaliser effectivement des échanges sécurisés. En effet, une fois définis des algorithmes complexes utilisant par exemple des clefs publiques, le premier problème pratique qui se pose est « comment rattacher une clef publique à son propriétaire ? ». L'idée des PKI est d'abord de ne pas distribuer des clefs mais plutôt des certificats numériques contenant ces clefs ainsi que des données d'identité (état civil, adresse, adresse mail pour une personne, nom de domaine ou adresse IP pour un serveur...). Ensuite, les PKI sont des structures précises assurant en particulier la création et la gestion de ces certificats.

Exercice 3.30 (Numéros de carte bleue jetables). *Une banque possède un mécanisme de numéros jetables de carte bleue. Un utilisateur a reçu par courrier classique un code pour générer de tels numéros sur le site internet de sa banque.*

1. *Quand l'utilisateur demande de générer un numéro jetable, quel est le type d'authentification que cela entraîne ?*
2. *En déduire une attaque permettant de récupérer le code.*
3. *Comment la banque peut-elle en fait contrer cette attaque ?*

Solution page 311.

Principe général

Pour s'authentifier, une entité cherche à prouver qu'elle possède une information secrète qui ne peut être connue que par elle seule. Dans le cas d'une

PKI, une entité (par exemple Alice) génère un couple clef publique/clef privée. Alice conserve sa clef privée (soit dans un répertoire accessible par elle seule ou mieux sur une carte à puce ou une clef USB). Ensuite, il s'agit de rendre sa clef publique disponible pour toute personne désireuse d'effectuer une transaction avec elle.

Pour cela, Alice va utiliser un *tiers de confiance* (un CA : voir paragraphe suivant) pour créer un *certificat*. Un certificat est un document numérique qui contient au moins les coordonnées personnelles d'Alice, ainsi que sa clef publique. Ce document est signé par le CA. Cette signature sert à certifier l'origine du certificat ainsi que son intégrité.

La figure 3.25 illustre les étapes de la création d'un certificat. Il reste encore à savoir comment vérifier l'authenticité du tiers de confiance. En pratique, ce tiers possède également un certificat signé par un autre tiers etc... Le dernier de cette *chaîne de certificats* devra alors se signer lui-même. Nous parlerons alors de certificat auto-signé ou certificat racine. Pour augmenter la confiance qu'un utilisateur pourra placer en eux, ces certificats racines devront faire l'objet de certifications croisées avec d'autres CA.

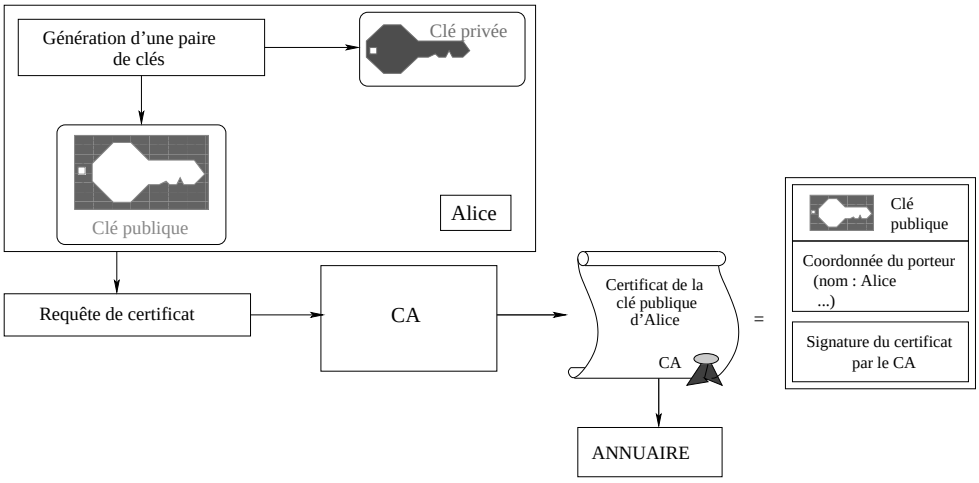


FIG. 3.25: Principe de la création des certificats.

Une PKI donc est un ensemble de technologies, organisations, procédures et pratiques qui supporte l'implémentation et l'exploitation de certificats basés sur la cryptographie à clef publique. C'est donc un ensemble de systèmes fournissant des services relatifs aux certificats numériques. Elle fournit les moyens d'utiliser ces certificats pour l'authentification entre entités. Il existe un standard pour ce mécanisme qui sera exposé dans la figure 3.31.

Les éléments de l'infrastructure

Une PKI doit d'abord fournir un certain nombre de fonctions dont les principales sont les suivantes : émettre des certificats à des entités préalablement authentifiées ; révoquer des certificats, les maintenir ; établir, publier et respecter des pratiques de certification pour établir un espace de confiance ; rendre les certificats publics par le biais de services d'annuaires ; éventuellement, gérer les clefs et fournir des services d'archivage. Dans ce dernier cas les services à rendre aux utilisateurs sont de trois types :

1. *Gestion des clefs.* Une architecture peut assurer la gestion de la création, la distribution, le stockage, l'utilisation, le recouvrement, l'archivage et la destruction des clefs (couples publiques/privés, ou clef secrète de chiffrement symétrique).
2. *Mise en commun des clefs.* Cela est réalisé par exemple par le protocole de Diffie et Hellman de la section 3.6.2.
3. *Transport des clefs*, par exemple par RSA comme dans la section 3.6.4.

Par exemple, la procédure d'émission est normalisée et doit suivre les différentes étapes de la figure 3.26.

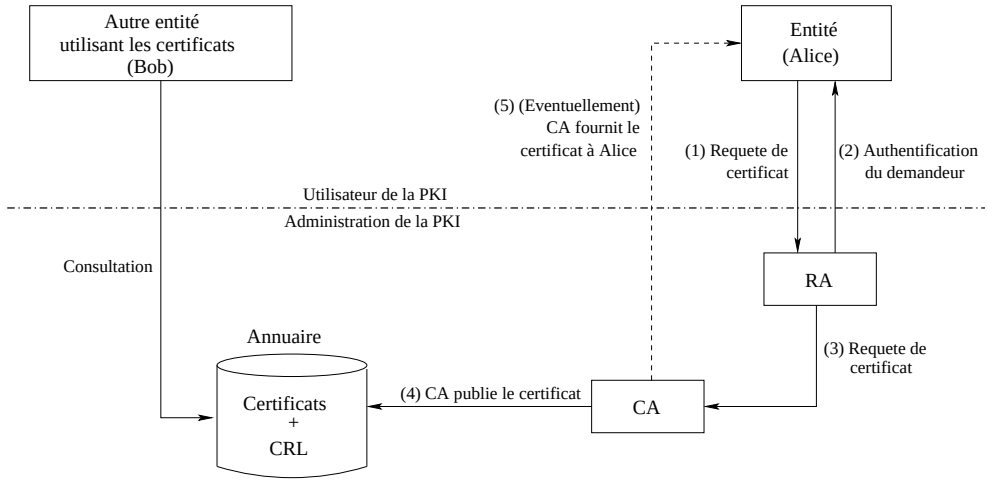


FIG. 3.26: Émission d'un certificat.

Les acteurs d'une PKI. On distingue différentes entités :

– Le *détenteur d'un certificat* : c'est une entité qui possède une clef privée et est le sujet d'un certificat numérique contenant la clef publique correspondante.

Il peut s'agir d'une personne physique, d'un serveur Web, d'un équipement réseau etc...

- L'*utilisateur d'un certificat* : celui-ci récupère le certificat et utilise la clef publique qu'il contient dans sa transaction avec le détenteur du certificat.
- L' *Autorité de Certification* (CA, *Certification Authority*) : c'est un ensemble de ressources (logicielles et/ou matérielles) et de personnels défini par son nom et sa clef publique qui :
  - génère des certificats ;
  - émet et maintient les informations sur les Listes de Révocation des Certificats (CRL, *Certification Revocation List*) ;
  - publie les certificats non encore expirés ;
  - maintient les archives concernant les certificats expirés et révoqués.

  C'est également l'entité juridique et morale d'une PKI.
- L' *Autorité d'enregistrement* (RA, *Registration Authority*) : c'est l'intermédiaire entre le détenteur de la clef et le CA. Il vérifie les requêtes des utilisateurs et les transmet au CA (le niveau de vérification dépend de la politique de sécurité mise en œuvre). Chaque CA a une liste de RA accrédités. Un RA est connu d'un CA par son nom et sa clef publique, ce dernier vérifiant les informations fournies par un RA par le biais de sa signature.
- L'*émetteur de CRL* : l'émission des listes de révocation peut être déléguée hors du CA à une entité spécialisée.
- le *Dépôt* ou *Annuaire* (Repository) qui se charge quand à lui de :
  - distribuer les certificats et les CRL.
  - accepter les certificats et les CRL d'autres CA et les rend disponibles aux utilisateurs.

  Il est connu par son adresse et son protocole d'accès.
- L'*Archive* se charge du stockage sur le long terme des informations pour le compte d'un CA. Cela permet de régler les litiges en sachant quel certificat était valable à telle époque.

De cette organisation il ressort que l'élément central d'une PKI est l'autorité de Certification. Une question légitime est donc : « Pourquoi faut-il des autorités d'enregistrement ? ». Deux types de facteurs interviennent :

1. Des éléments techniques.
   - Des algorithmes efficaces sont nécessaires à la création et à la gestion des clefs. Un RA peut fournir ces implémentations, logicielles ou matérielles, aux utilisateurs qui n'en disposent pas.
   - Les utilisateurs ne sont pas forcément capables de publier leurs certificats.
   - Le RA peut émettre une révocation *signée* même si un utilisateur a perdu toutes ses clefs (le RA est lui sensé ne pas perdre les siennes !).

   Le RA est donc un intermédiaire sûr.

2. Une organisation simplifiée.
   - Il est moins cher d'équiper un RA que tous les utilisateurs.
   - Le nombre de CA nécessaires est réduit par le regroupement de fonctions simples.
   - La proximité avec les utilisateurs est augmentée.
   - Souvent des structures *préexistantes* peuvent faire office de RA.

Cependant, séparer le CA et le RA est en général moins sûr que de centraliser tout dans un CA : un utilisateur et un RA peuvent en effet se mettre d'accord, *mais* le RA transmet ensuite au CA. Le CA, ayant la décision finale, peut falsifier les informations. La sécurité reste donc dans la confiance en le CA.

Les certificats électroniques

Il existe plusieurs normes pour les PKI, la plupart en cours d'évolution. Des exemples d'infrastructures à clef publiques faisant actuellement l'objet d'une normalisation à l'IETF (*The Internet Engineering Task Force*) sont PKIX (*Public Key Infrastructure X.509*), SPKI (*Simple Public Key Infrastructure*) ou encore SDSI (*Simple Distributed Security Infrastructure*). Chaque infrastructure définit souvent son propre format de certificat. Dans la suite, nous présentons le principal format utilisé : la norme X.509. La figure 3.27 donne l'allure d'un certificat au format X.509.

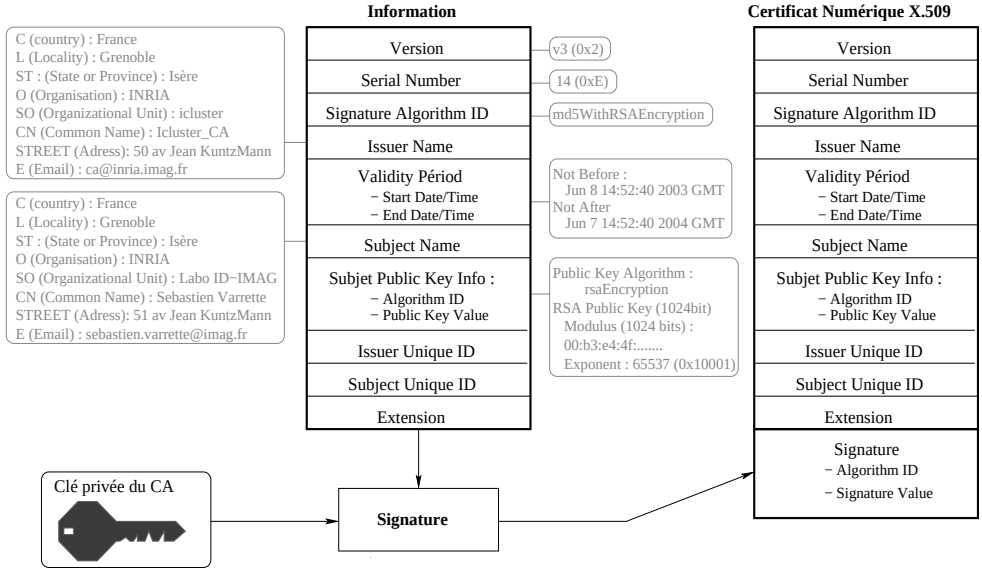


FIG. 3.27: Génération et contenu d'un certificat X.509.

Cette figure illustre par des exemples la valeur des principaux champs qui composent ce certificat. On notera sur les exemples de « Distinguished Name » (DN) (champs « Issuer name » et « Subject Name ») la liste des sous-champs qui composent ces DN. Ils permettent de localiser très précisément les entités associées.
Une fois qu'un certificat a été émis et déposé dans un annuaire, les autres utilisateurs peuvent venir interroger l'annuaire pour récupérer puis vérifier des certificats, comme par exemple sur la figure 3.28, ou encore consulter les CRL.

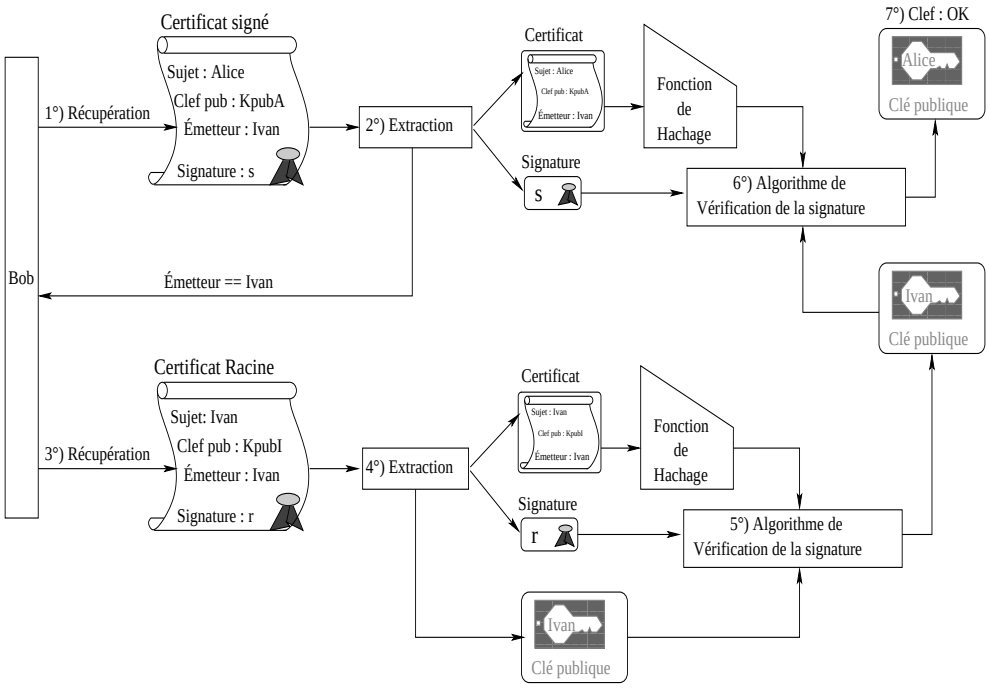


Fig. 3.28: Vérification de certificat et extraction de clef publique.

La norme X.509 a connu plusieurs versions successives motivées par des retours d'expériences (v1 : 1988, v2 : 1993, v3 : 1996). Elle précise également le format des Listes de Révocations de certificats. Celui-ci est très similaire à celui des certificats : un numéro de version, le type des algorithmes utilisés, le nom de l'émetteur, les dates de validité, la liste de numéros de certificats invalide, et la signature du CA.
Les certificats et les CRL X-509 sont écrits dans un langage de spécifications, ASN.1 (*Abstract Syntax Notation*). Parmi les types ASN.1 principaux utilisés pour les certificats X-509, on peut trouver :

– « OID » (*Object Identifier*) : des nombres séparés par des points. Le département de la défense des USA est par exemple désigné par {1.3.6}.
– *« Algorithm Identifier »* : un nom descriptif d'algorithme. Par exemple, 1.2.840.113549.3.7 correspond à du $DES - EDE3 - CBC$; 1.2.840.113549.1.1.4 à $md5withRSAEncryption$; 1.2.840.10045.1 à une signature par courbe elliptique avec un hachage SHA-1 $ecdsawithSha1$ etc...
– *« Directory String »* : information en texte.
– *« Distinguished Names », « General names », « Time »* : par exemple les dates sont définies par « Time :=CHOICE { UTCTime, GeneralizedTime } », où « UTCTime » est valide entre 1950 et 2049 et le « GeneralizedTime » après 2050.

Le modèle PKIX

Le groupe de travail PKIX (*PKI for X.509 certificates*) de l'IETF a été établi à l'automne 1995 dans le but de développer les standards Internet nécessaires à une infrastructure reposant sur les certificats X-509.

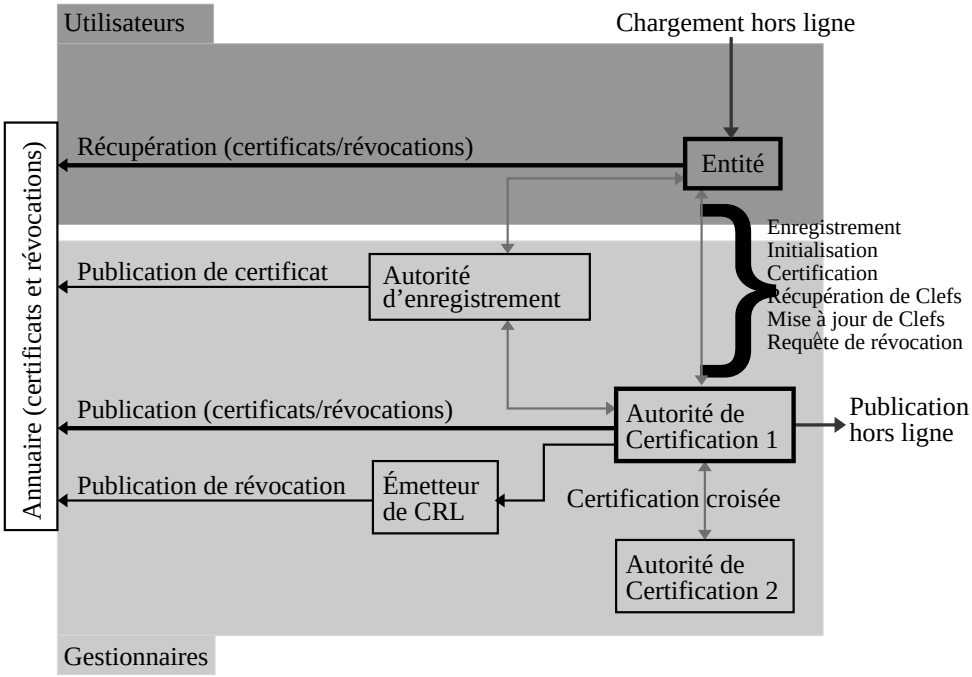


FIG. 3.29: Le modèle PKI pour les certificats X-509.

La première norme produite, le RFC-2459, décrit la version 3 des certificats

X-509 et la version 2 des CRL. Le RFC-2587 sur le stockage des CRL, le RFC-3039 sur la politique de certification et le RFC-2527 sur le cadre pratique de certifications ont suivi. Enfin, le protocole de gestion des certificats, (RFC-2510), le statut des certificats en-ligne, (RFC-2560), le format de demande de gestion de certificat (RFC 2511), la datation des certificats (RFC-3161), les messages de gestion de certificats (RFC-2797) et l'utilisation de FTP et HTTP pour le transport des opérations de PKI (RFC-2585) sont les autres standards mis au point par le groupe. Au total, l'ensemble des fonctions d'administration d'une infrastructure à clef publique a été défini et la figure 3.29 récapitule les structures retenues par le groupe.

Exercice 3.31 (OpenSSL - exercice à réaliser sur machine). *OpenSSL est un ensemble d'utilitaires de cryptographie implémentant SSL v2/v3 (Secure Sockets Layer) et TLS v1 (Transport Layer Security) et les standards qu'ils requièrent (en particulier ceux des PKI).*

1. Création d'un fichier de caractères aléatoires *: Sous Linux (depuis la version 1.3.30), il existe un fichier, réservoir d'entropie, (*`/dev/urandom`*, crée à partir de différentes sources aléatoires hardware). À l'aide de* `dd`*, comment créer un fichier aléatoire* `.rand` *d'au moins 16ko, avec 16 blocs distincts ?*
2. Mise en place d'un CA[8] *: Créer un répertoire demoCA et des sous répertoires private, certs, crl et newcerts. Entrer un numéro de série dans le fichier demoCA/serial, et créer un fichier demoCA/index.txt. Définir les droits d'accès à tous ces répertoires et fichiers.*
   (a) *Avec* `openssl genrsa`*, comment créer une paire de clefs de 4096 bits en utilisant votre fichier aléatoire ? Cette clef sera stockée dans un fichier* `demoCA/private/cakey.pem` *et cryptée sur le disque par AES-256 bits.*
   (b) *Avec* `openssl req`*, comment créer un nouveau certificat racine auto-signé, à l'aide de la clef précédente ? Ce certificat devra être au format X509, stocké dans le fichier* `demoCA/cacert.pem` *et valable 365 jours.*
3. Certificat utilisateur.
   – *Avec* `openssl`*, comment créer une paire de clefs RSA de 1024 bits ?*
   – *Comment faire une requête de certificat, avec* `openssl req`*, en utilisant cette clef. Cette requête devra contenir également la clef et être valable 2 mois.*
   – *Avec* `openssl ca`*, comment demander au CA de répondre à votre requête ? Cette requête devra créer un fichier certificat* `newcert.pem` *et spécifier la politique de sécurité par défaut* `policy_anything`*.*

[8] Tous les noms suivants peuvent être modifiés dans le fichier `openssl.cnf`.

4. Application à l'envoi de courriels sécurisés.
Des clients de courriel (par exemple Thunderbird) utilisent plutôt le format PKCS12 pour les certificats au lieu du format X509. Comment ajouter un certificat créé ci-dessus à un client de ce type ? Que se passe-t-il si la clef privée associée à ce certificat est perdue ? En particulier :

(a) Peut-on toujours envoyer des courriels chiffrés ? En recevoir ?

(b) Peut-on toujours signer des courriels qu'on envoie ? Vérifier les signatures des courriels reçus ?

Solution page 311.

La fonction principale d'une PKI est d'utiliser des certificats pour authentifier des personnes, sans tiers et sans risque de subir une attaque de type *Man-in-the-middle*. En particulier l'enregistrement initial d'une entité avec la création et la signature du certificat associé est régi par le mécanisme suivant de la figure 3.30.

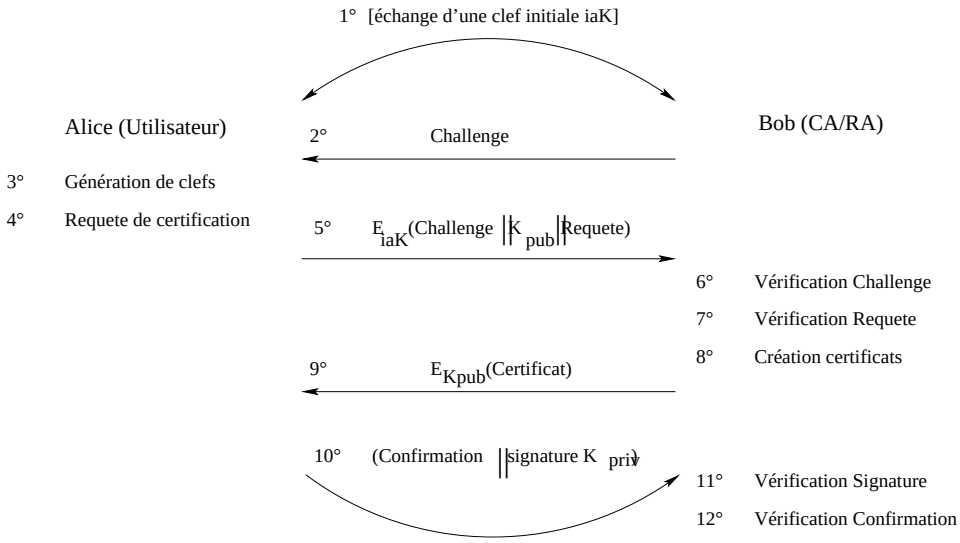


FIG. 3.30: Enregistrement authentifié.

Dans ce processus, chaque vérification de certificat est réalisée par extraction des informations concernant l'émetteur et la vérification des signatures grâce à la clef publique de celui-ci, comme décrit sur la figure 3.28. Une fois cet enregistrement réalisé, les authentifications ultérieures sont alors plus simples. La figure 3.31 donne les différentes étapes d'authentification mutuelle (l'authentification unilatérale ne contient pas le dernier message) du protocole FIPS-196.

1. Alice envoie à Bob une demande d'authentification
2. Bob génère un nombre aléatoire R_b qu'il retient et crée ainsi le jeton $TokenBA_1$ par :
$$TokenBA_1 = R_b$$
Bob l'envoie à Alice avec éventuellement un jeton d'identification.
3. Alice génère un nombre aléatoire R_a qu'elle retient. Ensuite, elle génère un jeton d'authentification $TokenAB$ de la forme :
$$TokenAB = R_a||R_b||B||Sign_a(R_a||R_b||B)$$
Le jeton contient le message et la signature avec la clef privée. Alice l'envoie à Bob avec son certificat (qui contient donc sa clef publique qui permettra de vérifier la signature) et éventuellement un identificateur de jeton (en fait, même l'envoi du certificat est optionnel)
4. Bob commence par vérifier le certificat d'Alice (voir figure 3.28). Ensuite, il peut vérifier les informations contenues dans $TokenAB$ à l'aide de la clef publique d'Alice qui est contenue dans le certificat. Après vérification de la signature d'Alice, celle-ci est authentifiée auprès de Bob. Il lui reste à s'authentifier en générant lui aussi un jeton d'authentification $TokenBA_2$ par :
$$TokenBA_2 = R_b||R_a||A||Sign_b(R_b||R_a||A)$$
Bob l'envoie à Alice avec son certificat et éventuellement avec un identificateur de jeton (même remarque que précédemment)
5. Alice procède de la même manière pour authentifier Bob. À la fin, il y a donc bien authentification mutuelle.

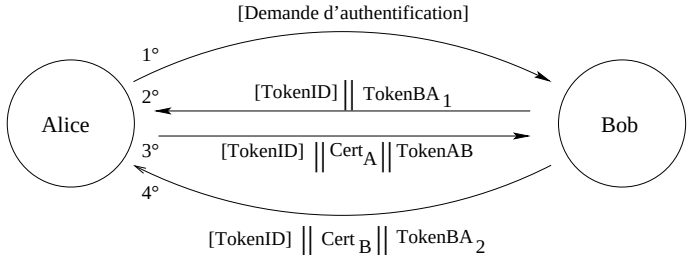


FIG. 3.31: Authentification par cryptographie à clef publique.

Une fois les deux parties authentifiées, elles peuvent alors utiliser leurs clefs publiques/privées pour crypter ou signer des messages, sans crainte d'une attaque Man-in-the-middle.

Exercice 3.32 (Certificats électroniques).

1. *Décrire les éléments essentiels d'un certificat numérique. Que comporte-t-il et pourquoi ?*
2. *Expliquer l'intérêt des certificats numériques.*
3. *Qu'est-ce qu'une politique de certification ?*

Solution page 312.

Architectures non hiérarchiques, PGP

À la différence de PKIX, PGP a été conçu pour permettre la messagerie sécurisée. Né avant la prolifération des CA accessibles par internet, PGP a dû résoudre le problème de la délégation de la confiance dans un modèle non-hiérarchique. Au lieu de se reposer sur différents CA, PGP fait en sorte que chaque utilisateur soit son propre CA.
Avoir l'équivalent d'un CA par utilisateur est impossible ! PGP a alors ajouté la notion de confiance : si un utilisateur sait que le certificat de l'utilisateur B est valide alors il peut signer le certificat de B. PGP autorise alors un nombre illimité d'utilisateurs à signer un même certificat, ainsi un véritable réseau de connaissances se crée et permet des confiances mutuelles.

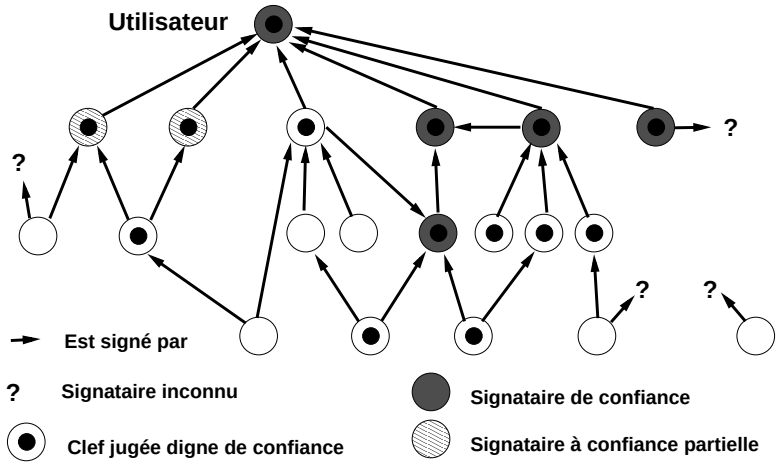


FIG. 3.32: Un exemple du modèle de confiance de PGP.

Dans ce modèle, la difficulté est de retrouver la chaîne de confiance qui lie deux utilisateurs. La solution élégante proposée par PGP est la confiance partielle : une signature sur un certificat ne donne qu'une fraction de la confiance en ce certificat et il faut alors plusieurs signataires de confiance pour déterminer un

certificat. L'autre difficulté de ce système est de bien faire la différence entre accorder sa confiance au certificat d'un utilisateur et accorder sa confiance à un tiers pour signer d'autres certificats. Un seul tiers indélicat ou simplement négligent peut détruire le réseau de confiance.
Enfin, il est à noter que les dernières versions du modèle de confiance de PGP introduisent une entité « méta » qui joue finalement également le rôle d'un CA racine afin d'émuler le fonctionnement hiérarchique.

Exercice 3.33. *Dans un modèle d'échange PGP, vous recevez un message signé par un certain Oscar. Le certificat d'Oscar est signé par Bob, dont le certificat est signé par Alice, en qui vous avez totalement confiance pour signer des certificats. Que pensez-vous du message reçu ?* *Solution page 312.*

Défauts des PKI

Les PKI ne sont pas la panacée, quelques défauts peuvent être décrits :

1. Plusieurs personnes peuvent avoir un même *« Distinguished Name »*. Ce risque est réduit par l'accumulation d'informations (pays, localité, région, adresse, courriel, IP, DNS, etc.). Cependant, une question demeure, qui certifie ces informations complémentaires ?
2. Que faut-il exactement comprendre lorsqu'un CA prétend être digne de confiance ? Ce problème est réduit par les certifications croisées, mais il faut que *tous* les enregistrements soient bien initiés *hors ligne* (*« out-of-band »* en anglais), c'est-à-dire sur un canal différent de celui de la communication courante, ou par un moyen non électronique.
3. La politique de sécurité de l'ensemble des CA est donc celle du plus faible !
4. Enfin, il manque une autorité de datation.

Exercice 3.34. *Carl M. Ellison et Bruce Schneier ont écrit un article « Tout ce que l'on ne vous dit pas au sujet des PKI ». Dans cet article, ils présentent 10 risques liés à l'utilisation des PKI. Commenter les quatre risques suivants :*

1. *Risque n°1 : « Who do we trust, and for what ? » (À qui faisons-nous confiance et pour quoi ?). Plus précisément, la question est de savoir qui a donné à un CA l'autorité de signer des certificats et quelle est la notion de confiance. Commenter pour les scénarios : PKI au sein d'une seule entreprise ; PKI entre une entreprise et ses clients/filiales ; PKI gérée par une entreprise spécialisée (VeriSign, RSA Security, etc.) pour une entreprise tiers.*
2. *Risque n°2 : « Who is using my key ? » (Qui utilise ma clef). Plus précisément, comment protéger les clefs secrètes ?*

3. *Risque n°4 : « Which John Robinson is he ? » (Quel Jean Martin est-ce ?). Plus précisément, comment traiter les différents Jean Martin connus par un CA ?*
4. *Risque n°9 : « How secure are the certificate practices ? » (Quelle est la sécurité des pratiques de certification ?). Par exemple :*
   (a) *Sur quels critères établir des durées de validité de clefs ?*
   (b) *Comment établir les longueurs de clefs, de hachage ?*
   (c) *Faut-il faire des CRL ?*
   (d) *Plus généralement comment savoir quelles sont les pratiques de certification d'une autorité ?*

Solution page 312.

En conclusion, il est nécessaire de bien définir la politique de sécurité et, au final, les défauts des PKI sont quasiment tous résolus si une PKI est restreinte à une seule entité administrative (e.g. une seule entreprise) et restent raisonnables lorsque quelques-unes de ces entités interagissent.

3.6.6 Un utilitaire de sécurisation de canal, SSH

Malgré la relative simplicité des fonctions D et E dans le cadre de RSA ou de El Gamal, elles ne sont pas aussi rapides à calculer que leurs équivalents pour le DES : l'exponentiation modulaire est une opération plus coûteuse en temps que des permutations ou des recherches dans une table lorsque p est grand. Cela rend plus compliquée la conception d'un composant électronique destiné à chiffrer et déchiffrer rapidement en utilisant cette approche. Un algorithme de type RSA est environ 10,000 fois plus lent qu'un DES.
La solution pour un chiffrement efficace réside dans l'utilisation conjointe de systèmes de chiffrement symétrique et asymétrique. Ainsi, le système PGP chiffre un message avec le protocole suivant :

1. Le texte source est chiffré avec une clef K de type 3-DES.
2. La clef K est chiffrée selon le principe RSA avec la clef publique du destinataire.
3. Envoi du message composé de la clef K chiffrée par RSA et d'un texte chiffré selon un algorithme de type 3-DES.

Le récepteur effectuera alors les opérations de déchiffrement suivantes :

1. Déchiffrement de la clef grâce à la clef privée D en utilisant RSA.
2. Déchiffrement du texte en utilisant DES ou IDEA avec la clef K.

Pour l'authentification le principe est un peu différent :

– Génération d'un code qui identifie, par un nombre de taille fixe, le texte, par exemple une fonction de hachage.
– Chiffrement de ce code en utilisant RSA avec la clef privée de l'émetteur du message.

Cela permet au correspondant de vérifier la validité du message et l'identité de l'émetteur en utilisant la clef publique de l'expéditeur.

Exercice 3.35. *Un groupe de n personnes souhaite utiliser un système cryptographique pour s'échanger deux à deux des informations confidentielles. Les informations échangées entre deux membres du groupe ne devront pas pouvoir être lues par un autre membre.*

1. *Quel est le nombre minimum de clefs symétriques nécessaires ?*
2. *Quel est le nombre minimum de clefs asymétriques nécessaires ?*
3. *Donner les raisons pour lesquelles ce groupe utilise finalement un système hybride pour le chiffrement ?*

Solution page 313.

Comme application du système PGP, citons le programme SSH utilisé en réseau pour se connecter sur une machine tierce, faire des transferts de fichiers et lancer des applications.

SSH, un protocole hybride

SSH est un protocole sécurisé de connexion à distance par des canaux non sûrs dans un modèle client-serveur.
Il se compose de trois parties :

1. une couche transport, qui permet l'authentification du serveur, la confidentialité et l'intégrité,
2. une couche d'authentification de l'utilisateur,
3. un protocole de connexion qui permet le passage de plusieurs canaux logiques.

De plus, ce protocole garantit : la confidentialité des données par un chiffrement fort ; l'intégrité des communications ; l'authentification des entités ; le contrôle de privilèges ; et la mise en place d'un canal sécurisé pour la suite des communications.
Comme PGP, il utilise un protocole à clef publique pour échanger des clefs secrètes utilisées par les algorithmes de chiffrement plus rapides utilisés pour la connexion elle-même. Commençons par la version SSH-1.

Authentification du serveur

Chaque serveur possède sa propre clef publique qui l'identifie. Le problème est de diffuser cette clef aux clients. Il y a deux stratégies :
– Le client maintient une base de données de clefs publiques des serveurs auxquels il se connecte,
– Il existe un tiers de confiance capable de diffuser de façon sûre les clefs.
En pratique, pour l'instant, l'algorithme consiste à ce que le serveur envoie sa clef publique, et lors de la première connexion, le client la stocke dans sa base (actuellement dans le fichier `~/.ssh/known_hosts`). Le client compare cette clef avec les clefs reçues ultérieurement. Néanmoins, cela autorise quand même une attaque *man-in-the-middle*, même si elle devient plus complexe. Pour ajouter un peu de sécurité, une signature de la clef est calculée et peut être facilement échangée par d'autres moyens, téléphone par exemple.

Établissement d'une connexion sécurisée

Elle se déroule en plusieurs phases et permet au serveur de fournir de façon sécurisée une clef de session K qui sera utilisée par un algorithme de chiffrement à clef secrète :

1. *Le client contacte le serveur.* Il s'agit simplement pour le client d'émettre une demande de connexion sur le port TCP du serveur (le port 22).
2. *Le client et le serveur s'échangent les versions du protocole SSH qu'ils supportent.* Ces protocoles sont représentés par une chaîne ASCII (ex : `SSH-1.5-1.2.27`).
3. *Le serveur et le client passent à une communication par paquets formatés.* Le format des paquets est explicité dans la figure 3.33.

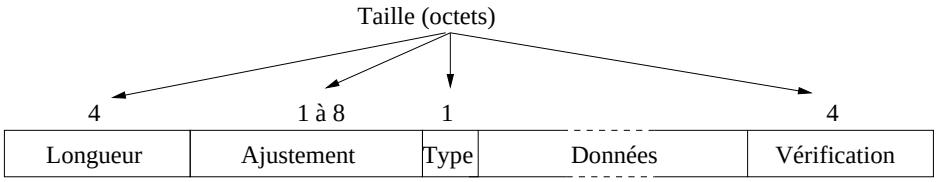


FIG. 3.33: Format d'un paquet SSH.

Voici la signification des différents champs de ce paquet :
– **Longueur** : il s'agit de la taille du paquet (sans compter les champs « Longueur » et « Ajustement ») codée sous forme d'un entier de 32 bits.

- **Ajustement** : il s'agit de données aléatoires dont la longueur varie de 1 à 8 octets ; plus précisément, la longueur est donnée par la formule : $8 - (\text{Longueur} \mod 8)$. Ce champ permet de rendre les attaques à texte clair choisi difficilement exploitables.
- **Type** : spécifie le type de paquet.
- **Données** : les données transmises.
- **Vérification** : il s'agit de quatre octets calculés par codes de redondance cycliques (CRC), voir la section 4.2 pour la construction d'un CRC.

Ce sont les champs « Ajustement », « Type », « Données » et « Vérification » qui seront chiffrés dès qu'une clef de session aura été établie.

4. *Le serveur s'identifie au client et fournit les paramètres de session.* Le serveur envoie les informations suivantes au client (toujours en « clair ») :
   - une *clef hôte* (publique) H (qui est LA clef publique du serveur) ;
   - une *clef serveur* (publique) S (une paire de clef RSA est régénérée toutes les heures) ;
   - une séquence C de 8 octets aléatoires (octets de contrôle) : le client devra inclure ces octets dans la prochaine réponse sous peine de rejet ;
   - la liste des méthodes de chiffrement, de compression et d'authentification supportées.

5. *Le client envoie au serveur une clef secrète de session.* Le client choisit une méthode M de chiffrement supportée par les deux parties (IDEA en mode CFB, DES en mode CBC, et RC4 par exemple pour SSH-1 ; 3-DES et AES en mode CBC pour les versions plus récentes), puis génère aléatoirement une clef de session K et envoie $[M, C, E_H(E_S(K))]$. Ainsi, seul le serveur sera capable de déchiffrer K, car seul le serveur connaît la clef privée associée à H. Le deuxième cryptage par S assure que le message n'a pas été rejoué puisque S n'est valide que pendant une heure.

   Le client et le serveur calculent également chacun de leur côté un identificateur de session (Id_S) qui identifie de manière unique la session et qui sera utilisé plus tard.

6. *Chaque entité met en place le chiffrement et complète l'authentification du serveur.* Après avoir envoyé sa clef de session, le client devra attendre un message de confirmation du serveur (chiffré à l'aide de la clef de session K), qui achèvera d'authentifier le serveur auprès du client (seul le serveur attendu est capable de déchiffrer la clef de session).

7. *La connexion sécurisée est établie* : le client et le serveur disposent d'une clef secrète de session qui leur permet de chiffrer et de déchiffrer leurs messages.

Authentification du client

Une fois qu'une connexion sécurisée a été mise en place, le client peut s'authentifier au serveur. Cette authentification peut s'effectuer par utilisation du nom de l'hôte (utilisant le fichier `~/.rhosts`, méthode peu satisfaisante en termes de sécurité), par mot de passe, ou par clef publique.

Dans ce dernier cas, le client génère à l'aide de la commande `ssh-keygen` les fichiers `~/.ssh/identity` (qui contiendra la clef secrète RSA d et qui est protégée par une « phrase de passe ») et `~/.ssh/identity.pub` (qui contiendra la clef publique RSA (e, n)). Le contenu de `identity.pub` est ajouté sur le serveur au fichier `~/.ssh/authorized_keys`.

Une fois cette étape effectuée, le client envoie une requête pour une authentification par clef publique. Le modulo n est passé en paramètre et sert d'identificateur. Le serveur peut rejeter la requête s'il ne permet pas l'authentification de cette clef (pas d'entrée associée dans le fichier `authorized_keys` par exemple). Sinon, il génère aléatoirement un nombre C de 256 bits et envoie au client $C^e \mod n$ (chiffrement RSA avec la clef publique du client).

Puis à l'aide de sa clef privée D, le client déchiffre C, le concatène avec l'identificateur de session Id_S, calcule l'empreinte du tout avec MD5 et envoie cette dernière au serveur en réponse. Le serveur effectue la même opération de son côté et vérifie la concordance du résultat avec ce qu'il reçoit. Cette dernière vérification achève l'authentification du client.

Sécurité des algorithmes, intégrité et compression des données

Les algorithmes de cryptage, intégrité et compression sont choisis par les deux parties. Les algorithmes utilisés sont bien connus et bien établis. Néanmoins cela autorise le changement d'algorithme si une faille apparaissait.

Comme on l'a vu, le contrôle d'intégrité se fait par un simple code de redondance cyclique (section 4.2), ce qui n'est pas efficace contre certaines attaques (par exemple l'attaque par insertion de Futoransky et Kargieman).Cette faiblesse est en partie à l'origine du changement de version, et de la création de SSH-2. Dans SSH-2, l'intégrité de chaque paquet est vérifiée en ajoutant des bits de signature (code d'authentification MAC, voir la section 3.5.2). Ils sont calculés à partir du numéro de séquence du paquet (qui ne fait pas partie du paquet) et de son contenu non chiffré.

Enfin, les données peuvent être compressées par exemple à l'aide de l'utilitaire `gzip` ce qui permet de limiter la bande passante utilisée pour la transmission des paquets.

Différences majeures entre SSH-1 et SSH-2

La nouvelle version du protocole SSH, SSH-2 fournit : un plus grand choix dans la négociation d'algorithmes entre le client et le serveur (choix du hachage etc...) ; la gestion des certificats X.509 pour les clefs publiques ; une plus grande flexibilité dans l'authentification (incluant une authentification partielle) ; un meilleur test d'intégrité par chiffrement ; le remplacement périodique de la clef de session toutes les heures ; les clefs secrètes sont désormais de 128 bits, les clefs publiques de 1024 bits ; plusieurs méthodes pour l'échange de clefs, en particulier le protocole de Diffie-Hellman ...

Exercice 3.36 (OpenSSH sous Linux). *La commande* `ssh` *sous Linux lance un programme pour se connecter sur une machine distante et pour exécuter des commandes sur cette machine, en suivant le protocole SSH. Dans le répertoire* `.ssh`*, à la racine du répertoire utilisateur, ce programme maintient un fichier* `known_hosts` *des serveurs sur lesquels l'utilisateur s'est connecté. La première connexion est considérée comme sûre et les suivantes sont authentifiées par la clef publique stockée dans le fichier* `.ssh/known_hosts`*.*

1. *Comment rendre cette authentification du serveur sûre ?*
2. *Plusieurs méthodes d'authentification du client sont alors possibles (clef publique, mot de passe, clavier) ; comment générer une paire clef publique / clef privée de type DSA sur 2048 bits avec l'utilitaire* `ssh-keygen` *?*
3. *Ajouter la clef publique ainsi générée au fichier* `authorized_keys` *dans le répertoire* `.ssh` *d'un compte tiers ; connectez-vous sur ce compte en utilisant* `ssh -v`*. Que constatez-vous ?*
4. *Détruire le fichier* `known_hosts`*. Se reconnecter sur une autre machine. À quel message doit-on s'attendre ?*
5. *Modifier le fichier* `known_hosts` *en changeant une clef publique. À quel message doit-on s'attendre ?*

Solution page 313.

3.6.7 Projets de standardisation et recommandations

Nous conclurons ce chapitre en mentionnant l'existence de projets de standardisation à travers le monde qui ont pour objectif de fournir des recommandations à un instant donné sur l'utilisation de tel ou tel algorithme. L'existence de ces projets est particulièrement intéressante pour les industriels et les néophytes qui peuvent alors choisir les algorithmes les plus sûrs et les plus adaptés à leurs besoins sans pour autant s'investir dans leur assimilation. Ces projets servent également à la communauté des chercheurs du domaine puisqu'une

analyse rigoureuse y est pratiquée sur les algorithmes que des équipes proposent.
On dénombre trois principaux projets/organismes. On donne pour chacun quelques exemples des recommandations pratiquées :

1. Le NIST (*National Institute of Standards and Technology*) aux États-Unis qui, par exemple, recommande depuis 2000 l'utilisation de AES-Rijndael (blocs de 128 bits et clef de 128/192/256 bits) comme algorithme de chiffrement à clé secrète par bloc.
2. Le KICS (*Korean Information and Communication Standards*) en Corée qui recommande depuis 2001 les algorithmes de chiffrement à clef secrète par bloc SEED et ARIA (tous deux avec des blocs et clefs de 128 bits).
3. Nous finirons par le projet européen NESSIE (*New European Schemes for Signatures, Integrity and Encryption*)

Ce dernier projet est particulièrement intéressant car les plus grands laboratoires de cryptographie d'Europe y ont participé et ont permis de dégager les algorithmes les plus intéressants en les classant par type. Ainsi, NESSIE recommande depuis 2003 :

– au niveau des chiffrement symétriques par blocs : MISTY1 (Blocs de 64 bits ; clef de 128 bits) ; AES-Rijndael ; Camellia (Blocs de 128 bits ; Clef de 128/192/256 bits) ; et SHACAL-2 (Blocs de 256 bits ; Clef de 512 bits).
– pour les fonctions de hachage à sens unique, Whirlpool ; SHA-256 ; SHA-384 et SHA-512 (on notera la clairvoyance des auteurs qui avaient anticipé la cryptanalyse de SHA-1).

Chapitre 4

Détection et correction d'erreurs

Dans les chapitres précédents, on supposait toutes les communications « sans bruit », c'est-à-dire qu'un message émis était systématiquement reçu tel quel, ou du moins les altérations étaient dues à un tiers mal intentionné qui savait les masquer. En réalité, il faut aussi tenir compte des altérations possibles du message dues au canal de transmission.
On appelle *taux d'erreur* la probabilité qu'un bit transmis par le canal soit différent du bit émis, c'est-à-dire le nombre de bits erronés reçus par rapport au nombre de bits émis. Ce taux d'erreur dépend de la nature de la ligne de transmission. Le tableau 4.1 donne quelques ordres de grandeur du taux d'erreur selon le canal utilisé. Dans le cas d'une communication téléphonique, qu'elle soit locale ou internationale, le taux d'erreurs dépend du nombre de répéteurs, du type de support (câble, satellite, etc.) et il peut atteindre 10^{-4} (jusqu'à une erreur pour 10000 bits transférés). Les supports de stockage contiennent eux aussi des erreurs. Par exemple, un fichier numérique stocké sur un CD ou un DVD contient nécessairement des erreurs ayant pour origine la gravure, la lecture par le laser, mais aussi des imperfections sur le support (poussières, rayures, ...). Dans un serveur de stockage distribué, l'information est distribuée sur plusieurs supports ; il est essentiel de retrouver l'ensemble d'une information même si certaines parties sont manquantes ; c'est le cas par exemple lors d'une panne de disque dur dans un système RAID ou de pairs non connectés dans un système de stockage pair-à-pair.
Par rapport à une information analogique, un intérêt fondamental d'une information numérique (donc sous forme de symboles discrets comme des bits) est qu'il existe des formats de représentation des données qui permettent de retrouver l'information initiale lorsque le taux d'erreur est borné. Développée au XX$^{\text{ième}}$ siècle à partir d'un article fondateur de Shannon, la théorie du co-

ligne	taux d'erreur
Disquette	10^{-9} : à 5 Mo/s, 3 bits erronés par minute
CD-ROM optique	10^{-5} : 7ko erronés sur un CD de 700 Mo
DAT audio	10^{-5} : à 48 kHz, deux erreurs par seconde
Mémoires à semi-conducteurs	$< 10^{-9}$
Liaison téléphonique	entre 10^{-4} et 10^{-7}
Télécommande infrarouge	10^{-12}
Communication par fibre optique	10^{-9}
Satellite	10^{-6} (Voyager), 10^{-11} (TDMA)
ADSL	10^{-3} à 10^{-9}
Réseau informatique	10^{-12}

Tab. 4.1: Ordre de grandeur du taux d'erreurs.

dage de canal, souvent dénommée théorie des codes, a pour objet l'étude et la construction de telles représentations – les codes correcteurs – qui soient les plus performantes possibles.
Le but de ce chapitre est d'introduire les principaux codes détecteurs et correcteurs d'erreurs utilisés en pratique : les codes CRC, Reed-Solomon et turbo. Les outils mathématiques sur lesquels s'appuient ces codes (arithmétique des corps finis et polynômes) sont présentés au chapitre 1. On pourra les consulter au préalable, ou en renvoi lors de leur utilisation.

4.1 Principe de la détection et de la correction d'erreurs

4.1.1 Codage par blocs

Soit un canal qui transmet des symboles (les caractères) d'un alphabet V. Pour un canal binaire, chaque symbole est un bit : $V = \{0, 1\}$. Soit $m \in V^+$ un message à transmettre, sous forme d'un flot (on suppose le message source préalablement encodé sous forme d'une séquence de symboles de V, voir chapitre 1).
Le *codage par blocs* de taille k consiste à découper le message $m \in V^+$ en blocs M_i de k symboles, en traitant chaque bloc l'un après l'autre.
Lorsqu'une séquence de k symboles $s = [s_1, \ldots, s_k]$ est émise sur le canal, la séquence $s' = [s'_1, \ldots, s'_k]$ reçue par le destinataire peut différer ; on dit qu'il y a eu erreur de transmission. Si $s'_i \neq s_i$, on dit que le symbole i est erroné ; si j symboles sont erronés, on dit qu'il y a eu j erreurs.
Pour se protéger contre ces erreurs en les détectant voire en les corrigeant de manière automatique, les méthodes de *codage par blocs* consistent à ajouter de

la redondance aux k symboles d'information du bloc initial.
Le codage transforme donc le bloc à transmettre $s = [s_1, \ldots, s_k]$ en un bloc de $n = k + r$ symboles, $\phi(s) = [c_1, \ldots, c_k, \ldots, c_n]$ qui comporte r symboles de redondance par rapport aux k symboles du message initial. L'ensemble $C_\phi = \{\phi(s) \ : s \in V^k\}$, image par ϕ de V^k, est appelé un *code(n, k)* ; ses éléments sont appelés *mots de code*. Pour que le décodage permette de reconstruire sans ambiguïté le message source lorsqu'il n'y a pas d'erreur, la fonction de codage ϕ doit bien sûr être injective. Le *rendement* R d'un code(n, k) est le taux de chiffres de source contenus dans un mot de code :

$$R = \frac{k}{n}.$$

Ainsi, la réception d'un mot qui n'est pas dans C_ϕ (i.e. un mot de $\overline{C_\phi} = V^n \backslash C_\phi$) indique une erreur. La détection d'erreur repose alors sur ce test de non-appartenance à C_ϕ. Suite à une détection d'erreurs, le décodeur peut éventuellement procéder à une correction. On distingue deux types de corrections :
– la correction directe, comme dans le schéma ci-dessus : le signal erroné reçu contient suffisamment d'information pour permettre de retrouver le mot émis ;
– la correction par retransmission (ou ARQ, *Automatic Repeat reQuest*) : lorsqu'une erreur est détectée mais ne peut pas être corrigée, le récepteur demande à l'émetteur de retransmettre une nouvelle fois le message source.
Dans les paragraphes suivants, les principes de la détection et de la correction d'erreurs sont introduits avec deux codes très simples.

4.1.2 Un exemple simple de détection par parité

Dans cette section on considère un canal binaire : $V = \{0, 1\}$. On code le mot $m = (s_1, \ldots, s_k)$ par $\phi(m) = (s_1, \ldots, s_k, c_{k+1})$ où $c_{k+1} = (\sum_{i=1}^{k} s_i) \mod 2$ (ce qui s'écrit avec l'opérateur ou-exclusif $c_{k+1} = \oplus_{i=1}^{k} s_i$). On a donc

$$c_{k+1} \oplus \left(\bigoplus_{i=1}^{k} s_i \right) = 0,$$

autrement dit $c_{k+1} = 1$ si et seulement si le nombre de bits nuls dans m est pair. Le test de cette égalité s'appelle *contrôle de parité*. Il est évident que cette égalité devient fausse si, lors de la transmission, un nombre impair de bits dans $\phi(m)$ changent de valeurs comme nous l'avons vu section 1.4.2. Ainsi, l'ajout d'un bit de parité longitudinale permet de détecter une erreur portant sur un nombre impair de bits (voir le tableau 4.2).
Cette technique élémentaire de détection d'erreurs est toujours très utilisée au niveau matériel, en particulier pour détecter les erreurs dans la mémoire. Les

m : mot source (7 bits)	$\phi(m)$ avec bit de parité (8 bits)
0101001	01010011
0001001	00010010
0000000	00000000

TAB. 4.2: Codage avec bit de parité.

codes de parité sont souvent utilisés pour vérifier la correction de nombreux numéros d'identification.

Exercice 4.1 (Numéro de Sécurité Sociale). *Un numéro de sécurité sociale est un nombre de $n = 15$ chiffres : un numéro d'identification K sur $k = 13$ chiffres suivi de la clef C de $r = 2$ chiffres calculée pour que $K + C$ soit un multiple de 97.*

1. *Quel est la clef du numéro de sécurité sociale 2.63.05.38.516.305 ?*
2. *Quel est le rendement de ce code ?*
3. *Combien d'erreurs de chiffres, la clef du numéro de sécurité sociale permet-elle de détecter ?*

Solution page 315.

4.1.3 Un exemple simple de correction directe par parité longitudinale et transversale

Toujours en considérant $V = \{0, 1\}$, on construit un code(n, k_1k_2) binaire comme suit. Comme $k = k_1k_2$, un mot source m peut être représenté dans un tableau bidimensionnel de bits avec k_1 lignes et k_2 colonnes. Le mot de code associé $\phi(m)$ contient $(k_1 + 1)(k_2 + 1)$ bits dont $k_1 + k_2 + 1$ bits de parité : un pour chaque ligne et chaque colonne et un pour l'ensemble des bits (figure 4.3). Ce code permet non seulement de détecter mais aussi de corriger une erreur : il suffit de localiser la ligne et la colonne où s'est produite l'erreur en contrôlant le bit de parité de chaque ligne et chaque colonne (exercice 4.2).

Exercice 4.2 (Code de parité longitudinale et transversale). *On construit un code de parité longitudinale et transversale pour des mots source m de 9 bits.*

1. *Quelle est la longueur n des mots de code $\phi(m)$?*
2. *Montrer que l'on détecte toute erreur portant sur un nombre impair de bits.*

m : mot source de $21 = 7 \times 3$	$\phi(m)$ codé sur 32 bits avec bits de parité	mot reçu (1 erreur)	mot corrigé (1 correction)
0101001	0101001 **1**	01010011	01010011
0001001	0001001 **0**	00011**0**1**0**	00010010
0000000	0000000 **0**	00000000	00000000
	0100001 0	0100**0**001	01000001

TAB. 4.3: Correction d'erreur par bits de parité.

3. *Montrer que l'on détecte jusqu'à trois erreurs de bits.*
4. *Donner deux exemples de mots comportant quatre erreurs : l'un où l'on détecte qu'il y a eu erreur, l'autre où l'on ne peut pas détecter.*
5. *Montrer que l'on peut corriger tout mot reçu comportant au plus une erreur.*
6. *Donner l'exemple d'un mot reçu comportant deux erreurs que l'on ne peut pas corriger.*

Solution page 315.

4.1.4 Schéma de codage et probabilité d'erreur

Plus généralement, avec la fonction injective de codage $\phi : V^k \to V^n$ qui associe à tout mot source un mot du code(n, k), le schéma de codage par bloc C_ϕ est le suivant :

1. Le mot de code $c = \phi(s)$ est calculé et transmis sur le canal ; le récepteur reçoit un mot $c' = [c'_1, \ldots, c'_n]$.
2. Si le mot c' reçu appartient à C_ϕ, le message est décodé en $\tilde{s}$ tel que $\phi(\tilde{s}) = c'$.
3. Sinon, $c' \notin C$ et on détecte qu'il y a eu erreur. La correction procède comme suit :
   (a) s'il existe un entier i et un unique mot de code $c'' \in C_\phi$ ayant au moins $n - i$ bits égaux à c', alors on corrige c' en c'' et le message décodé est $\tilde{s}$ tel que $\phi(\tilde{s}) = c''$.
   (b) sinon, il existe au moins deux mots de code qui ont tous deux i chiffres communs avec c', et aucun mot de code n'a $i - 1$ chiffres communs avec c'. On ne peut donc pas corriger l'erreur et on renvoie un mot fixe $\tilde{s} = m$ de k bits, qui peut correspondre à un signal d'erreur.

Avec ce schéma, il y a erreur si $\tilde{s}$ finalement reçu est différent de s : on note P_{C_ϕ} la probabilité d'une telle erreur.
Dans un souci d'efficacité, le code C_ϕ – et en particulier la fonction de codage ϕ – doit être choisi pour :
– avoir une probabilité d'erreur P_{C_ϕ} aussi petite que désirée ;
– avoir un rendement R le plus grand possible ;
– être le moins coûteux à calculer possible, grâce à des algorithmes de codage (calcul de ϕ) et de décodage (test d'appartenance à C_ϕ, calcul de ϕ^{-1} et correction) les plus efficaces possibles.

Pour un canal donné, le deuxième théorème de Shannon montre qu'il est possible, avec des contraintes sur le choix de k en fonction de n, de construire des codes C_ϕ tels que cette probabilité d'erreur P_{C_ϕ} soit aussi petite que désiré.

4.1.5 Deuxième théorème de Shannon

Shannon est l'auteur d'un théorème important sur la possibilité de construire des codes correcteurs d'erreurs dans des canaux bruités. Nous nous limitons ici au cas d'un canal bruité sans mémoire. Plus précisement le théorème que nous allons voir s'applique au *codage de canal* où chaque mot de source transite par un canal théorique qui transforme ce mot en une parmi M valeurs équiprobables. Autrement dit, la source est discrète uniforme.

Rendement d'un code et efficacité

Pour un canal donné avec ses taux d'erreurs de transmission intrinsèques, le deuxième théorème de Shannon prouve l'existence de codes ayant une probabilité d'erreur P_{C_ϕ} arbitraire, mais dont le rendement est limité en fonction de la fiabilité du canal, plus précisément de sa *capacité*. Autrement dit, même sur un canal perturbé, il est possible de faire de la transmission sans erreur !

Capacité de canal

Soit un canal bruité sans mémoire qui transmet des symboles de l'alphabet $V = \{s_1, \ldots, s_{|V|}\}$. Soit X la variable aléatoire source à l'entrée du canal, dont les réalisations sont des symboles de V. De même, soit Y la variable aléatoire à la sortie du canal dont on suppose que les réalisations sont aussi les symboles de V. Soit la distribution $p_{j|i} = P(Y = s_j | X = s_i)$ des probabilités de transition ; $p_{j|i}$ est la probabilité de recevoir s_j en sortie du canal sachant que s_i a été émis en entrée.
Par exemple, sur un canal sans erreur, $p_{i|i} = 1$ et $p_{j|i} = 0$ pour $j \neq i$. Ainsi, la distribution $(p_{j|i})_{1 \leqslant i,j \leqslant |V|}$ caractérise les probabilités d'erreurs lors de la transmission sur le canal sans mémoire.

On note $p_i = P(X = s_i)$ la probabilité qu'une réalisation de X prenne la valeur s_i. Avec les probabilités de transition, cette distribution (p_i) de la source induit une distribution de probabilités sur la variable Y. Avec l'égalité de Bayes (voir section 1.2.2), on a $P(Y = s_j) = \sum_{i=1}^{|V|} P(X = s_i, Y = s_j) = \sum_{i=1}^{|V|} P(X = s_i) \cdot P(Y = s_j | X = s_i)$; d'où

$$P(Y = s_j) = \sum_{i=1}^{|V|} p_i \cdot p_{j|i}$$

La *capacité du canal* $\mathcal{C}(X, Y)$ est la quantité maximale d'information sur l'entrée effectivement transmise sur le canal, le maximum étant pris sur toutes les distributions d'entrées $(p_i) \in \bar{p}$ vérifiant $\sum_{i=1}^{|V|} p_i = 1$. Or, l'entropie conditionnelle $H(X|Y)$ (voir page 34) représente ce qu'il reste à découvrir de l'entrée X quand on connaît la sortie Y, c'est à dire la perte d'information sur l'entrée. La capacité du canal est donc le maximum de la quantité d'information de l'entrée diminuée de cette perte :

$$\mathcal{C} = \max_{p \in \bar{p}} H(X) - H(X|Y).$$

À l'aide des formules de l'entropie conditionnelle, il est possible de réécrire cette définition également sous les formes suivantes :

$$\mathcal{C} = \max_{p \in \bar{p}} H(Y) + H(X) - H(X, Y) = \max_{p \in \bar{p}} H(Y) - H(Y|X).$$

Les cas extrêmes correspondent donc à :
- $H(Y) = H(Y|X)$: si savoir quelle est l'entrée n'influe pas sur la sortie, alors le canal ne transmet jamais rien de l'entrée ;
- $H(Y|X) = 0$: si la sortie peut apporter exactement la même information que l'entrée, alors le canal est a priori sans erreur.

Le calcul de ce maximum est facile dans le cas où le canal est *symétrique*. Dans ce cas, les probabilités de transition sont indépendantes du symbole s_i considéré : pour tout $1 \leqslant i, j \leqslant |V|$ avec $i \neq j$, on a alors $p_{i|j} = p$ et $p_{i|i} = 1 - p$; la probabilité p est alors appelée la *probabilité d'erreur du canal*. Le maximum est atteint quand les entropies $H(X)$ et $H(Y)$ sont maximales, donc pour une distribution uniforme des probabilités d'entrée $p_i = \frac{1}{|V|}$.
Par exemple, l'exercice 4.3 montre que la capacité d'un canal binaire symétrique BSC (*Binary Symmetric Channel*, pour lequel $|V| = 2$) de probabilité d'erreur p est :

$$\mathcal{C}_{BSC} = 1 + p \log_2(p) + (1 - p) \log_2(1 - p).$$

Exercice 4.3. *Un canal binaire symétrique (BSC) de probabilité d'erreur p est un canal pour lequel $|V| = 2$ et*
– la probabilité que le bit reçu soit différent de l'émis est p : $p_{2|1} = p_{1|2} = p$;
– la probabilité que le bit reçu soit égal à l'émis est $1-p$: $p_{1|1} = p_{2|2} = 1-p$.
Montrer que sa capacité est $\mathcal{C}_{BSC} = 1 + p\log_2(p) + (1-p)\log_2(1-p)$.
En déduire $\mathcal{C}_{BSC} = 0$ si $p = \frac{1}{2}$ mais $0 < \mathcal{C}_{BSC} \leqslant 1$ sinon. *Solution page 316.*

Transmission sans erreur sur un canal de capacité fixée

Grâce à la capacité de canal, le théorème de Shannon donne une condition nécessaire et suffisante pour réaliser une transmission sans erreur sur un canal de capacité $\mathcal{C}$.

Théorème 22 (Deuxième théorème de Shannon). *Soit un canal de capacité $\mathcal{C}$. Soit $\epsilon > 0$ arbitraire. Il existe au moins un code(n,k) de rendement $R = \frac{k}{n}$ et de probabilité d'erreur $P < \epsilon$ si et seulement si*

$$0 \leqslant R < \mathcal{C}.$$

Par exemple, dans le cas d'un canal binaire symétrique de probabilité d'erreur $p = \frac{1}{2}$, on a $\mathcal{C} = 0$. Il est donc impossible de transmettre correctement. Par contre, si $p \neq \frac{1}{2}$, $0 < \mathcal{C} \leqslant 1$; il existe donc toujours un code(n,k) permettant de transmettre sans erreur, mais avec une contrainte sur son rendement. Plus généralement, il est toujours possible de transmettre sans erreur sur un canal de capacité non nulle, mais le rendement du code est limité par la capacité du canal.

Exercice 4.4. *On peut trouver intuitif le fait de pouvoir transmettre sans erreur sur un canal BSC pour lequel la probabilité p de recevoir un bit différent de celui émis est inférieure à $\frac{1}{2}$.*
Justifier sans utiliser le théorème de Shannon que l'on peut alors aussi transmettre sans erreur lorsque $\frac{1}{2} < p \leqslant 1$. *Solution page 316.*

Ce théorème donne une condition nécessaire sur le nombre minimum r de chiffres de redondance à ajouter :

$$r > k\left(\frac{1}{\mathcal{C}} - 1\right). \tag{4.1}$$

Cette borne ne dépend pas du taux d'erreurs corrigées par le code mais de la capacité du canal.
La preuve du théorème de Shannon n'est pas constructive ; le théorème montre l'existence de codes, mais pas comment les construire. De plus, il est possible de faire tendre le rendement vers C en faisant tendre n vers l'infini ; par exemple,

en choisissant comme code un sous-ensemble de mots choisis aléatoirement dans V^n et de même cardinal que V^k. Cependant, pour être pratique, il est nécessaire que le codage et le décodage puisse être réalisés efficacement, avec un coût matériel et temporel raisonnable. Aussi, le code doit vérifier des propriétés additionnelles que ne possède pas a priori un code aléatoire.
C'est pourquoi ce théorème a motivé la recherche de codes efficaces en particulier dans deux directions :
– d'une part pour un taux de correction fixé, construire des codes qui optimisent le rendement ; c'est le cas en particulier des codes par blocs de Reed-Solomon ;
– d'autre part, pour un rendement fixé, construire des codes qui optimisent le taux de correction ; c'est le cas des codes convolutifs et des turbo-codes.
Les turbo-codes hybrides par blocs proposent des compromis entre les deux approches.
Les sections suivantes introduisent les principaux codes utilisés pour une source binaire. Après une modélisation du problème de la correction d'erreurs et la définition de la distance d'un code (section 4.3), les codes cycliques par blocs sont définis et illustrés avec les codes de Reed-Solomon (section 4.4). Les techniques d'entrelacement (section 4.5) permettent d'étendre ces codes pour la correction d'erreurs en rafales ; ils sont illustrés avec l'exemple pratique du code CIRC pour les disques compacts audio, construit à partir d'un code de Reed-Solomon. Enfin, le chapitre conclut avec les codes convolutifs et les turbo-codes qui sont construits par entrelacement.
De manière générale, ces différents codes sont tous associés à des polynômes. Cette représentation par polynôme est d'abord introduite dans la section suivante (4.2) avec les codes détecteurs d'erreur CRC ; ces codes apparaissent dans de très nombreux cas pratiques où l'émetteur peut réémettre un message que le récepteur a mal reçu.

4.2 Détection d'erreurs par parité - codes CRC

Lorsque le récepteur peut demander à l'émetteur de ré-émettre l'information à moindre coût (ARQ, voir la section 4.1.1), il suffit de pouvoir coder et décoder rapidement, en détectant efficacement les erreurs sans les corriger. On s'intéresse donc à la construction d'un *code détecteur* d'erreurs.
Pour être efficaces, les codes détecteurs contiennent dans chaque mot de code les symboles d'information associés. On dit qu'un code(n, k) est *systématique* s'il existe k positions telles que la restriction de tout mot de code $c = \phi(s)$ à ces k positions est égale à s. Le code est dit aussi *séparable* car on peut extraire directement les symboles d'information $s = [s_1, \ldots, s_k]$ du mot de code $c = \phi(s)$. Souvent, ces symboles d'information sont mis dans l'ordre aux

k premières positions. Le principe du codage systématique est alors de calculer les $r = n - k$ symboles de redondance $[c_{k+1}, \ldots, c_n] = \phi_r([s_1, \ldots, s_k])$, où ϕ_r est la restriction de ϕ aux r dernières positions. La généralisation à un code systématique sur k autres positions est directe.
Dans un code systématique, les symboles de redondance sont appelés *symboles de parité* ou *symboles de contrôle*. En effet, la détection d'erreurs se ramène à un *contrôle de parité* (ou *checksum*) comme introduit au paragraphe 4.1.2 On associe r symboles de redondance à chaque mot $[s_1, \ldots s_k]$ source tel que le mot de code $c = [s_1, \ldots s_k, c_{k+1}, \ldots, c_{k+r}]$ vérifie un prédicat $f(c)$ (la parité) avec f facile à calculer.
Après une présentation de quelques contrôles de parité sur les entiers utilisés dans des domaines de la vie courante qui dépassent largement les télécommunications, cette section présente les codes de redondance cyclique (CRC) utilisés dans les réseaux informatiques.

4.2.1 Contrôle de parité sur les entiers : ISBN, EAN, LUHN

Comme en témoigne l'exemple du numéro de Sécurité Sociale (voir exercice 4.1), les codes de contrôle de parité sont utilisés couramment. Dans ces cas pratiques, le prédicat $f(c)$ consiste généralement à tester si une somme des chiffres c_i du mot reçu est multiple d'un entier fixé. Le code ISBN d'identification des livres, les codes barres d'identification des produits (EAN) et leurs extensions, la clef RIB d'un compte bancaire et le numéro d'une carte bancaire en sont d'autres exemples qui sont détaillés ici.

Code barre EAN

Le code EAN-13 (*European Article Numbering*) est un code à barres utilisé dans l'industrie et le commerce pour identifier de manière univoque les produits ou les unités logistiques. Il est composé d'un numéro sur 13 chiffres $c_{12} - -c_{11}c_{10}c_9c_8c_7c_6 - -c_5c_4c_3c_2c_1c_0$ et d'un motif graphique avec des barres noires et blanches verticales pouvant être lues par un scanner optique. Ce code est dérivé du code américain UPC-A (*Universal Product Code*) sur 12 chiffres : les barres sont identiques mais un zéro est ajouté en tête du numéro UPC-A (*i.e.* $c_{13} = 0$) pour former le code EAN-13. Les 12 chiffres de gauche $c_{12}, \ldots, c_1$ identifient le produit ; c_0 est le chiffre de parité calculé comme suit. Soit $a = \sum_{i=1}^{6} c_{2i}$ la somme des chiffres de rang pair et $b = \sum_{i=1}^{6} c_{2i-1}$ celle des chiffres de rang impair. On a alors $c_0 = 10 - (a + 3b) \mod 10$.
Le code barre graphique code le même numéro EAN-13 : chaque colonne de 2,31 mm code un et un seul chiffre. Chaque colonne est divisée en 7 sous colonnes de largeur 0,33 mm mais le codage EAN-13 impose qu'un élément (un chiffre) soit toujours codé par une succession de seulement 4 barres de

largeurs différentes, mais dont l'ensemble a une largeur totale de 7 fois la largeur élémentaire. Par exemple le 0 est codé par la succession d'une bande blanche de largeur $3*0,33 = 0,99$ mm, d'une bande noire de largeur $2*0,33 = 0,66$ mm, d'une bande blanche de largeur $0,33$ mm et d'une bande noire de largeur $0,33$ mm. On peut résumer ce codage par $0 \rightarrow 000\ 11\ 0\ 1$. Les représentations symétriques doivent toujours coder le même chiffre, donc on a aussi $0 \rightarrow 111\ 00\ 1\ 0$; $0 \rightarrow 1\ 0\ 11\ 000$ et $0 \rightarrow 0\ 1\ 00\ 111$.

Exercice 4.5. *Montrer qu'il y a bien* 40 *choix possibles de 4 barres par colonne.* *Solution page 316.*

Une extension du code EAN-13 est le code EAN-128, qui permet de coder non seulement des chiffres mais également des lettres comme sur la figure 4.1 ci-dessous.

FIG. 4.1: Code EAN-128 de la théorie des codes.

Le code EAN-13 permet de détecter des erreurs mais pas de les corriger. Certains codes barres graphiques permettent non seulement de détecter mais aussi de corriger des erreurs ; par exemple, PDF417 (*Portable Data File*) est un code barre bidimensionnel, multi-lignes et de petite taille, qui permet de coder jusqu'à 2725 caractères, grâce à un code de Reed-Solomon (dont on donne les détails un peu plus loin dans ce chapitre).

Code ISBN

Le numéro ISBN (pour *International Standard Book Number*) est un numéro qui identifie tous les livres édités dans le monde. Ce numéro comporte dix chiffres $c_{10}c_9c_8c_7c_6c_5c_4c_3c_2c_1$, structurés en quatre segments $A-B-C-D$ séparés par un tiret. Les neuf premiers chiffres $A-B-C$ identifient le livre : A identifie la communauté linguistique, B le numéro de l'éditeur et C le numéro d'ouvrage chez l'éditeur. La clef de contrôle $D = c_1$ est un symbole de parité qui est soit un chiffre entre 0 et 9, soit la lettre X qui représente 10. Cette clef c_1 est telle que $\sum_{i=1}^{10} i \times c_i$ est un multiple de 11 ; autrement dit, $c_1 = 11 - \left(\sum_{i=2}^{10} i \times c_i\right) \mod 11$.
La norme ISO 2108 en vigueur à partir du 1 janvier 2007 identifie les livres par code barre EAN-13 de 13 chiffres : les trois premiers chiffres valent 978,

les 9 suivants sont les 9 premiers chiffres du code ISBN (ABC) et le dernier est le chiffre de contrôle EAN-13.

Clé RIB

Un numéro RIB de compte bancaire $B - G - N - R$ comporte 23 caractères. B et G sont des nombres de 5 chiffres qui identifient respectivement la banque et le guichet. N est composé de 11 alphanumériques, que l'on convertit en une séquence S_N de 11 chiffres en remplaçant les lettres éventuelles cycliquement comme suit : A et J en 1 ; B,K, S en 2 ; C,L, T en 3 ; D,M, U en 4 ; E,N, V en 5 ; F,O,W en 6 ; G,P,X en 7 ; H,Q,Y en 8 ; I,R,Z en 9. Enfin la clef RIB R est un nombre de deux chiffres qui est tel que, mis bout à bout, B, G, S_N et R forment un nombre n de 23 chiffres multiple de 97. Autrement dit, $R = 97 - ([BGS_N] \times 100) \mod 97$.

Carte bancaire : LUHN-10

Le numéro à 16 chiffres $c_{15}c_{14}c_{13}c_{12} - c_{11}c_{10}c_9c_8 - c_7c_6c_5c_4 - c_3c_2c_1c_0$ d'une carte bancaire est un mot de code de LUHN 10. Pour vérifier la validité du numéro, on multiplie les chiffres d'indice impair par 2 et on soustrait 9 si le nombre obtenu est supérieur à 9. Tous ces chiffres sont additionnés et sommés aux autres chiffres d'indice pair. La somme finale doit être un multiple de 10. En plus du numéro N de carte à 16 chiffres, la validité d'une carte peut être vérifiée à partir des 7 chiffres qui figurent à son dos. Les 4 premiers de ces 7 chiffres ne sont autres que les 4 derniers chiffres du numéro de carte. Les 3 derniers sont le résultat d'un calcul qui fait intervenir, outre N, les 4 chiffres mm/aa de sa date d'expiration.

Exercice 4.6 (Codes ISBN et EAN-13). *Compléter la séquence 210-050-692 pour obtenir un numéro ISBN correct. Quel est son code barre EAN-13 associé ?* *Solution page 317.*

4.2.2 Codes de redondance cyclique (CRC)

Les codes de redondance cyclique, appelés CRC (*Cyclic Redundancy Check*), sont très utilisés dans les réseaux informatiques. Dans ces codes, un mot binaire $u = [u_{m-1} \dots u_0] \in \{0,1\}^m$ est représenté par un polynôme $P_u(X) = \sum_{i=0}^{m-1} u_i X^i$ de $\mathbb{F}_2[X]$. Par exemple, $u = [10110]$ est représenté par le polynôme $P_u = X^4 + X^2 + X$.
Un code CRC est caractérisé par un *polynôme générateur* P_g de degré r : $P_g(X) = X^r + \sum_{i=0}^{r-1} g_i X^i$ dans $\mathbb{F}_2[X]$. Le mot source binaire $s = [s_{k-1} \dots s_0]$ associé au polynôme $P_s(X) = \sum_{i=0}^{k-1} s_i X^i$, est codé par le mot de code binaire

$c = [c_{n-1} \dots c_0] = [s_{k-1} \dots s_0 c_{r-1} \dots c_0]$ où $[c_{r-1} \dots c_0]$ est la représentation du reste de la division euclidienne de $X^r.P_s$ par P_g. La multiplication de P_s par X^r se traduisant par un décalage de bits, le polynôme $P_c(X) = \sum_{i=0}^{n-1} c_i X^i$ vérifie donc

$$P_c = P_s.X^r + (P_s.X^r \mod P_g).$$

Exemple : Avec le polynôme générateur $P_g = X^2 + 1$, le mot $u = [10110]$ est codé en ajoutant les bits de redondance du polynôme $(X^4 + X^2 + X)X^2 \mod X^2 + 1 = X^6 + X^4 + X^3 \mod X^2 + 1 = X$ qui correspond aux bits de redondance [10]. Le mot codé est donc $\phi(u) = [1011010]$.

Codage CRC

Le codage se ramène donc à implémenter le calcul du reste d'une division euclidienne. Cette opération peut être faite en temps $O(n \log n)$ par un algorithme basé sur le produit de polynômes par DFT (section 1.4.2), ou même en $O(n)$ avec l'algorithme standard de division de polynôme dans le cas pratique où le polynôme a un petit nombre de coefficients non nuls ($O(1)$ en théorie). De plus, l'implémentation matérielle de l'algorithme standard est très efficace en utilisant des registres à décalage linéaire (LFSR, section 1.3.6).

Décodage CRC

Pour le décodage, on remarque que dans $\mathbb{F}_2[X]$ on a aussi $P_c = P_s.X^r - (P_s.X^r \mod P_g)$; le polynôme P_c est donc un multiple de P_g. Cette propriété permet un décodage rapide, lui aussi basé sur une division euclidienne de polynômes. À la réception de $c' = [c'_{n-1} \dots c'_0]$, il suffit de calculer le reste $R(X)$ du polynôme $P_{c'}$ par le polynôme P_g. Si ce reste est nul, le mot reçu est un mot de code et l'on ne détecte pas d'erreurs. Sinon, si $R(X) \neq 0$, il y a eu erreur lors de la transmission.

Nombre d'erreurs de bit détectées

Supposons qu'il y a une erreur dans la transmission. Les coefficients du polynôme $P_e = P_{c'} - P_c$ valent 0 sur les bits non erronés et 1 aux bits de c modifiés. L'erreur est détectée si et seulement si P_e n'est pas un multiple de P_g.
On en déduit que si un des multiples de degré au plus $n-1$ de P_g possède j monômes, alors le code CRC généré par g ne peut détecter qu'au plus $j-1$ erreurs. Réciproquement, pour qu'un code CRC détecte toute erreur portant sur au plus j bits, il faut et il suffit que tout polynôme de degré au plus $n-1$ et ayant au plus j monômes ne soit pas multiple de P_g.

Par ailleurs, l'erreur P_e égale à P_g, ou multiple de P_g, ne peut pas être détectée. On en déduit que le nombre d'erreurs de bit détectées est au plus $\omega(P_g)$ où $\omega(P_g)$ est le nombre de coefficients non nuls de P_g.

Propriété 12. *Un code CRC de générateur* $P_g(X) = X^r + \sum_{i=1}^{r-1} g_i X^i + 1$ *détecte toute erreur portant sur* $r-1$ *bits consécutifs ou moins.*

Preuve. Une séquence d'erreurs portant sur au plus $r-1$ bits consécutifs commençant au $i^{\text{ième}}$ bit du mot est associée à un polynôme $P_e = X^i Q$ avec Q de degré inférieur strictement à r ; Q n'est donc pas multiple de P_g. Et comme $g_0 = 1$, P_g est premier avec X ; on en déduit que P_e n'est pas multiple de P_g donc l'erreur est bien détectée. □

Cette propriété est importante ; elle justifie l'utilisation des codes CRC dans les cas où les erreurs peuvent affecter quelques bits consécutifs.

Exercice 4.7 (Condition de détection d'une erreur). *Montrer qu'une condition nécessaire et suffisante pour qu'un code CRC détecte toute erreur portant sur un unique bit est que son polynôme générateur admette au moins deux monômes.* *Solution page 317.*

Exercice 4.8 (Détection d'un nombre impair d'erreurs). *Montrer que si la somme des coefficients du polynôme générateur d'un code CRC est nulle, alors toute erreur portant sur un nombre impair de bits est détectée.* *Solution page 317.*

Exercice 4.9 (Détection de deux erreurs). *Montrer que si le polynôme générateur d'un code CRC admet un facteur irréductible de degré d, alors il détecte tout paquet d'erreurs (c'est-à-dire des erreurs consécutives) portant sur au plus* $d-1$ *bits.* *Solution page 317.*

Exemples de codes CRC

Les codes CRC sont très utilisés dans les réseaux. Outre leurs bonnes propriétés énoncées ci-dessus, ils sont choisis aussi pour l'efficacité du codage et décodage. La table 4.4 donne quelques exemples normalisés ; la normalisation concerne non seulement la valeur du polynôme mais aussi les algorithmes de codage et décodage.

4.3 Distance d'un code

4.3.1 Code correcteur et distance de Hamming

Un code (n, k) est dit *t-détecteur* (resp. *t-correcteur*) s'il permet de détecter (resp. corriger) toute erreur portant sur t chiffres ou moins lors de la transmission d'un mot de code de n chiffres.

Nom	Générateur	Factorisation	Exemples d'utilisation
TCH/FS -HS-EFS	$X^3 + X + 1$	irréductible	GSM transmission de voix
GSM TCH/EFS	$X^8 + X^4 + X^3 + X^2 + 1$	irréductible	GSM pré-codage canal à plein taux
CRC-8	$X^8 + X^7 + X^4 + X^3 + X + 1$	$(X+1)(X^7 + X^3 + 1)$	GSM 3ème génération
CRC-16 X25-CCITT	$X^{16} + X^{12} + X^5 + 1$	$(X+1)(X^{15} + X^{14} + X^{13} + X^{12} + X^4 + X^3 + X^2 + X + 1)$	Protocole X25-CCITT ; contrôle trames PPP FCS-16 (RFC-1662)
CRC-24	$X^{24} + X^{23} + X^{18} + X^{17} + X^{14} + X^{11} + X^{10} + X^7 + X^6 + X^5 + X^4 + X^3 + X + 1$	$(X+1)(X^{23} + X^{17} + X^{13} + X^{12} + X^{11} + X^9 + X^8 + X^7 + X^5 + X^3 + 1)$	communications UHF et satellites (SATCOM) ; messages OpenPGP (RFC-2440)
CRC-24 (3GPP)	$X^{24} + X^{23} + X^6 + X^5 + X + 1$	$(X+1)(X^{23} + X^5 + 1)$	GSM 3ème génération
CRC-32 AUTODIN-II	$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$	irréductible	IEEE-802.3, ATM AAL5, trames PPP FCS-32 (RFC-1662) ; contrôle d'intégrité des fichiers ZIP et RAR ;

TAB. 4.4: Exemples de codes CRC.

Exemples :
- L'ajout d'un bit pour contrôler la parité des 7 bits qui le précèdent est un code systématique par bloc. C'est un code (8,7). Il est 1-détecteur et 0-correcteur avec un rendement de 87,5%.
- Le contrôle de parité longitudinale et transversale sur 21 bits (avec ajout de 11 bits de contrôle) est un code (32,21). Il est 1-correcteur avec un rendement de 65,625%.

Dans toute la suite on considère un code de longueur n ; les mots de code sont donc des éléments de V^n. On appelle *poids de Hamming* de $x = (x_1, \ldots, x_n) \in V^n$, noté $w(x)$, le nombre de composantes non nulles de x :

$$w(x) = |\{i \in \{1, \ldots, n\} / x_i \neq 0\}|.$$

Remarque 2. *Dans le cas où V est binaire ($V = \{0, 1\}$), on note $x \oplus y$ le ou-exclusif bit à bit (ou addition) de deux mots x et y. Le poids de Hamming w satisfait alors l'inégalité triangulaire : $w(x \oplus y) \leqslant w(x) + w(y)$.*

On peut alors obtenir une notion de distance, la *distance de Hamming* entre x et y, notée $d_H(x, y)$, qui est le nombre de composantes pour lesquelles x et y diffèrent, *i.e.*

$$d_H(x, y) = |\{i \in \{1, \ldots, n\} / x_i \neq y_i\}|.$$

Exercice 4.10. *1. Montrer que d_H ainsi définie est une distance sur V^n.*

2. Montrer que si V est binaire, $d_H(x, y) = w(x \oplus y)$.

Solution page 317.

La distance de Hamming permet de caractériser le nombre d'erreurs que peut corriger un code C de longueur n. En effet, soit $m \in C$ un mot de n chiffres émis et soit m' le mot reçu, supposé différent de m ($d_H(m, m') > 0$) :
- pour que C puisse détecter une erreur, il est nécessaire que $m' \notin C$ (sinon, le mot reçu est un mot de C donc considéré comme correct).
- pour pouvoir faire une correction (c'est-à-dire retrouver m à partir de m'), il faut que m soit l'unique mot de C le plus proche de m', soit

$$\text{pour tout } x \in C, \text{ si } x \neq m \text{ alors } d_H(x, m') > d_H(m, m').$$

C'est ce que traduit la propriété suivante.

Propriété 13. *Soit C un code de longueur n. C est t-correcteur si*

$$\textit{pour tout } x \in V^n, |\{c \in C / d_H(x, c) \leqslant t\}| \leqslant 1.$$

Remarque 3. *La correction d'une erreur peut aussi être décrite en considérant les boules $B_t(c)$ de centre $c \in C$ et de rayon t :*

$$B_t(c) = \{x \in V^n / d_H(c,x) \leqslant t\}\,.$$

Lors de la réception de m', on peut corriger m' en m si on a :

$$B_{d_H(m,m')}(m') \cap C = \{m\}\,.$$

La capacité de correction d'un code C est donc liée à la distance minimale entre deux éléments de C.
La *distance minimale*du code C, notée $\delta(C)$, est définie par :

$$\delta(C) = \min_{c_1,c_2 \in C\ ;c_1 \neq c_2} d_H(c_1,c_2)\ . \tag{4.2}$$

La propriété 13 et la remarque 3 sont à la base du théorème suivant qui caractérise un code t-correcteur.

Théorème 23. *Soit C un code de longueur n. Les propriétés suivantes sont équivalentes, et toutes impliquent par conséquent que C est t-correcteur :*
(i) Pour tout $x \in V^n$, $|\{c \in C / d_H(x,c) \leqslant t\}| \leqslant 1$;
(ii) Pour tout $c_1, c_2 \in C$, $c_1 \neq c_2 \Longrightarrow B_t(c_1) \cap B_t(c_2) = \emptyset$;
(iii) Pour tout $c_1, c_2 \in C$, $c_1 \neq c_2 \Longrightarrow d_H(c_1,c_2) > 2t$;
(iv) $\delta(C) \geqslant 2t+1$.

Preuve. $(i) \Longrightarrow (ii)$: Supposons qu'il existe $x \in B_t(c_1) \cap B_t(c_2)$; ainsi c_1 et c_2 sont à une distance $\leqslant t$ de x. D'après (i), cela n'est possible que pour au plus un mot de code ; donc $c_1 = c_2$.
$(ii) \Longrightarrow (iii)$: par l'absurde. Si $d_H(c_1,c_2) \leqslant 2t$; soient $i_1, \ldots, i_d$ avec $d \leqslant 2t$ les indices où les chiffres de c_1 et c_2 diffèrent. Soit x le mot de V^n dont les chiffres sont les mêmes que ceux de c_1 sauf les chiffres en position $i_1, ..., i_{d/2}$ qui sont égaux à ceux de c_2. Alors $d_H(x,c_1) = d/2 \leqslant t$ et donc $x \in B_t(c_1)$. De même, $d_H(x,c_2) = d - (d/2) \leqslant t$ et donc $x \in B_t(c_2)$. D'où $x \in B_t(c_1) \cap B_t(c_2)$ ce qui contredit (ii).
$(iii) \Longrightarrow (iv)$: immédiat.
$(iv) \Longrightarrow (i)$: par la contraposée. Supposons qu'il existe $x \in V^n$, tel que $|\{c \in C / d_H(x,c) \leqslant t\}| \geqslant 2$. Il existe alors $c_1 \neq c_2$ tels que $d_H(x,c_1) \leqslant d_H(x,c_2) \leqslant t$. Et comme d_H, distance de Hamming, vérifie l'inégalité triangulaire, $d_H(c_1,c_2) \leqslant d_H(c_1,x) + d_H(x,c_2) \leqslant 2t$. Donc $\delta(C) \leqslant 2t$. □

Exercice 4.11. *Montrer que si C est t-correcteur alors C est s-détecteur avec $s \geqslant 2t$. Dans quelle cas la réciproque est-elle vraie ?* *Solution page 317.*

Exercice 4.12. *On considère un code C de répétition pure sur $V = \{0, 1\}$, c'est à dire un code (mk, k) où chaque groupe de k bits est simplement répété m fois.*

1. *Quelle est la distance minimale entre deux mots de code ?*
2. *Proposer un code de répétition qui soit 2-correcteur.*
3. *Quel est son rendement ?*

On considère un code de répétition $(n = 5k, k)$ et on désire écrire une procédure de décodage qui retourne en sortie le mot décodé s et un booléen e qui vaut vrai si s peut être considéré comme correct (pas d'erreurs ou des erreurs corrigées) et faux sinon (des erreurs ont été détectées mais pas corrigées).

4. *Combien d'erreurs peuvent être détectées ? Écrire l'algorithme de décodage pour détecter un nombre maximal d'erreurs mais n'en corriger aucune.*
5. *Combien d'erreurs peuvent être corrigées ? Écrire l'algorithme de décodage pour corriger un nombre maximal d'erreurs. Combien d'erreurs sont détectées mais non corrigées ?*
6. *Écrire l'algorithme de décodage pour corriger une seule erreur mais en détecter jusqu'à 3.*

Solution page 318.

Exercice 4.13 (Calcul naïf de la distance d'un code). *Soit C un code (n, k) sur un alphabet V. Écrire une procédure qui calcule le taux de détection et de correction du code ; donner la complexité de l'algorithme en fonction de k et n. L'algorithme est-il efficace ?*

Application : Soit le code binaire

$$C = \{0000000000,\ 0000011111,\ 1111100000,\ 1111111111\}.$$

Que valent n et k pour ce code ? Calculer son rendement, son taux de détection et de correction. *Solution page 318.*

4.3.2 Codes équivalents, étendus et raccourcis

Comme c'est la distance qui caractérise le nombre d'erreurs que peut corriger un code, c'est une caractéristique importante, complémentaire à son rendement. Afin de réaliser des compromis entre rendement et distance, de nombreuses méthodes permettent de modifier un code pour en construire un autre avec des propriétés différentes.

Codes équivalents

Soient $x_1x_2 \dots x_n$ et $y_1y_2 \dots y_n$ deux mots d'un code :
– soit π une permutation de $\{1, \dots, n\}$ des positions ; on a

$$d_H(x_1x_2 \dots x_n, y_1y_2 \dots y_n) = d_H(x_{\pi(1)}x_{\pi(2)} \dots x_{\pi(n)}, y_{\pi(1)}y_{\pi(2)} \dots y_{\pi(n)})\ ;$$

– soient $(\pi_i)_{i=1,\dots,n}$ n permutations des symboles de V (potentiellement distinctes), on a également :

$$d_H(x_1 \dots x_n, y_1 \dots y_n) = d_H(\pi_1(x_1) \dots \pi_n(x_n), \pi_1(y_1) \dots \pi_n(y_n)).$$

Par suite, ni l'ordre choisi des symboles dans V ni celui des positions dans les mots de code ne modifient la distance. Aussi, deux codes sont dits *équivalents* si l'un peut être obtenu à partir de l'autre par permutation π des positions et par permutations $(\pi_i)_{i=1,\dots,n}$ des valeurs des symboles. Deux codes équivalents ont la même distance, donc les mêmes taux de détection et correction.
Par exemple pour un code binaire, en permutant les symboles en première et troisième position et en substituant en deuxième position les 0 par des 1 et les 1 par des 0 dans tous les mots, on obtient un code équivalent.

Propriété 14. *Deux codes équivalents ont le même rendement et la même distance.*

Code étendu

Soit C un code (n, k) ; on appelle code étendu C' de C le code $(n+1, k)$ obtenu en ajoutant un chiffre de contrôle de parité. Par exemple, si une opération d'addition est définie sur V,

$$C' = \left\{ c_1 c_2 \dots c_n c_{n+1} \mid c_1 c_2 \dots c_n \in C, \sum_{i=1}^{n+1} c_i = 0 \right\}.$$

La distance d' du code étendu vérifie trivialement $d \leqslant d' \leqslant d + 1$.

Exercice 4.14 (Extension de code). *Soit C est un code binaire de distance d impaire. Montrer que la distance du code étendu C' est $d + 1$.* *Solution page 320.*

Code poinçonné

Soit C un code (n, k) de distance d ; le *poinçonnage* consiste à supprimer dans le code C tous les symboles à m positions fixées. Le code C' obtenu est dit *code poinçonné* de C ; la distance d' du code vérifie alors $d' \geqslant d - m$.

Code raccourci

Soit C un code de longueur n et de distance d et soient $s \in V$ un symbole et $i \in \{1, \dots, n\}$ une position. On *raccourcit* le code C en ne prenant que ses mots dont le symbole en position i vaut s et en supprimant la position i. Le code C' obtenu est dit *code raccourci* ; il est donc de longueur $n - 1$ et sa distance d' vérifie $d' \geqslant d$.

Exemple : Le code binaire $C = \{0000, 0110, 0011, 1010, 1110\}$ est de longueur 4 et de distance 1. Le code C' raccourci obtenu avec le symbole 0 en position 1 est $C' = \{000, 110, 011\}$; il est de longueur 3 et de distance 2.
Ainsi, raccourcir un code permet d'augmenter le rendement sans diminuer la distance, voire même en l'augmentant comme le montre l'exemple ci-dessus. Aussi, cette opération est fréquemment utilisée dans la construction de codes ou comme outil de preuve.

Exercice 4.15 (Code raccourci). *Soit C un code de longueur n, de cardinal M et de distance d. Montrer que C est le raccourci d'un code de longueur $n+1$, de cardinal $M+2$ et de distance 1.* *Solution page 320.*

Exercice 4.16 (Code de parité transversale et longitudinale(9,4)). *Le contrôle de parité transversale et longitudinale sur 4 bits conduit au code (9,4) suivant : le mot de code associé à la source $b_0b_1b_2b_3$ est :*

$$[b_0, b_1, b_2, b_3, (b_0 + b_1), (b_2 + b_3), (b_0 + b_2), (b_1 + b_3), (b_0 + b_1 + b_2 + b_3)] .$$

On obtient ainsi C= { 000000000, 000101011, 001001101, 001100110, ... }.

1. *Montrer que C est exactement 1-correcteur.*
2. *Donner une configuration comportant 2 erreurs non corrigibles.*
3. *Montrer que C est 3-détecteur mais pas 4-détecteur.*
4. *Quelles sont les caractéristiques du code poinçonné de C à la première position ?*
5. *Quelles sont les caractéristiques du code raccourci de C à la première position ?*

Solution page 320.

4.3.3 Code parfait

Un code t-correcteur permet de corriger toute erreur e de poids $w(e) \leqslant t$; mais il peut subsister des erreurs détectées non corrigées (voir exercice précédent). On dit qu'un code est *parfait* (à ne pas confondre avec les chiffrements parfaits ou inconditionnellement sûrs de la section 1.2.4) lorsque toute erreur détectée est corrigée. Ce paragraphe étudie quelques propriétés des codes parfaits. Un code *t-parfait* est un code t-correcteur dans lequel toute erreur détectée est corrigée. Si C est t-parfait, lorsqu'on reçoit un mot $m' \notin C$, alors il existe un unique mot m de C tel que $d(m, m') \leqslant t$. De plus, comme C est t-correcteur, les boules de rayon t et de centre les mots de C sont deux à deux disjointes ; d'où le théorème suivant.

Théorème 24. *Le code C (n,k) sur V est t-parfait si les boules ayant pour centre les mots de C et pour rayon t forment une partition de V^n, i.e.*

$$\biguplus_{c\in C} B_t(c) = V^n\,.$$

Évidemment, une telle partition n'est possible que pour certaines valeurs de n et k, en fonction du cardinal de V. Le théorème suivant donne des conditions nécessaires pour l'existence de codes t-correcteurs et t-parfaits.

Théorème 25. *Soit $C(n,k)$ un code t-correcteur sur V. Alors*

$$1 + C_n^1(|V|-1) + C_n^2(|V|-1)^2 + \cdots + C_n^t(|V|-1)^t \leqslant |V|^{n-k}. \tag{4.3}$$

Si de plus il y a égalité, alors $C(n,k)$ est t-parfait.

Preuve. La preuve repose sur le calcul du cardinal de $\biguplus_{c\in C} B_t(c)$, les boules étant 2 à 2 disjointes dans le cas d'un code t-correcteur. En effet, le cardinal d'une boule de rayon t de V^n est :

$$|B_t(x)| = 1 + C_n^1(|V|-1) + C_n^2(|V|-1)^2 + \cdots + C_n^t(|V|-1)^t\,.$$

Comme un code $C(n,k)$ possède exactement $|V|^k$ éléments de V^n associées à des boules deux à deux disjointes dont l'union est incluse dans V^n, on a :

$$|V|^k\Big(1 + C_n^1(|V|-1) + C_n^2(|V|-1)^2 + \cdots + C_n^t(|V|-1)^t\Big) \leqslant |V|^n,$$

d'où la première inégalité. S'il y a égalité, alors le cardinal de l'union est égal à $|V|^n$; de plus, comme le code est t-correcteur, les boules sont disjointes. Donc leur union est égale à V^n et le code est parfait. □

Exercice 4.17. *Montrer que tout code binaire 1-correcteur sur des mots de $k = 4$ bits requiert au moins 3 bits de redondance. Montrer que s'il existe un code 1-correcteur avec 3 bits de redondance, alors il est parfait.*

Solution page 321.

4.3.4 Codes de Hamming binaires

Les *codes de Hamming binaires* (1950) sont des codes 1-correcteur à redondance minimale. En effet, soit $x \in \mathbb{F}_2^n$; le poids de Hamming binaire de x est alors donné par $w(x) = |\{i \in \{1,\ldots,n\}/x_i \neq 0\}| = \sum_{i=1}^n x_i$.
Or, dans $\mathbb{F}_2$, l'addition et la soustraction sont identiques ($0 = -0$ et $1 = -1$) ; on a alors les propriétés suivantes.

Propriété 15. *Soient x et y deux éléments quelconques de $\mathbb{F}_2^n$ et $\wedge$ l'opérateur ET bit à bit :*
– $w(x) = d_H(x, 0)$;
– $d_H(x, y) = w(x + y) = w(x) + w(y) - 2w(x \wedge y)$.

Exercice 4.18. *Soit un code binaire (n, k) 1-correcteur.*

1. *Montrer que*
$$k \leqslant 2^r - r - 1. \tag{4.4}$$
2. *En déduire une borne sur le rendement maximal d'un code 1-correcteur dont le nombre de bits de contrôle est 3, puis 4, puis 5, puis 6.*
3. *Existe-t-il un code 1-parfait de longueur $n = 2^m$ où $m \in \mathbb{N}$?*

Solution page 321.

Connaissant k, l'inégalité (4.4) (qui s'écrit $2^r \geqslant n + 1$) permet de déterminer le nombre minimal de bits de contrôle à ajouter pour obtenir un code 1-correcteur. Les codes de Hamming binaires sont donc des codes systématiques qui atteignent cette limite théorique : pour n bits, le nombre de bits de contrôle est $\lfloor \log_2 n \rfloor + 1$ (soit, si $n + 1$ est une puissance de 2, $\log_2(n + 1)$). Ce sont donc des codes parfaits pour $n = 2^r - 1$) ; autrement dit, ce sont des codes $(n, n - \lfloor \log_2 n \rfloor - 1)$.

Description d'un codage pour le code de Hamming binaire $(n, n - \lfloor \log_2 n \rfloor - 1)$. Un mot $c = c_1 \dots c_n \in \{0, 1\}^n$ du code de Hamming binaire est tel que les bits c_i dont l'indice i est une puissance de 2 (2^l avec $l = 0, 1, \dots$) sont des bits de contrôle et les autres sont des bits de données. Le bit de contrôle d'indice $i = 2^l$ est la somme modulo 2 de tous les bits de données c_j dont l'indice j écrit en base 2 a le $(l + 1)^{\text{ième}}$ bit (à partir de la gauche) à 1.
Par exemple, dans un code $(7, 4)$ de Hamming, au message source $[s_0, s_1, s_2, s_3]$ correspond le mot de code $[c_1, c_2, c_3, c_4, c_5, c_6, c_7]$ donné par :

$$\begin{cases} c_3 = s_1 \\ c_5 = s_2 \\ c_6 = s_3 \\ c_7 = s_4 \\ c_1 = c_3 + c_5 + c_7 \\ c_2 = c_3 + c_6 + c_7 \\ c_4 = c_5 + c_6 + c_7 \end{cases}$$

Ainsi le message 1000 est codé 1110000.

Pour assurer la correction dans un code de Hamming, le contrôle de parité est fait de la façon suivante. Tous les bits de contrôle d'indice $i = 2^l$ sont vérifiés ; une erreur est détectée si l'un de ces bits est erroné (parité erronée). Soit alors e la somme des indices des bits de contrôle i qui sont erronés. Si il y a une seule erreur, elle provient alors du bit e.

Théorème 26. *Le code de Hamming* $(n, n - \lfloor log_2 n \rfloor - 1)$ *est un code 1-correcteur qui requiert un nombre de bits de contrôle minimal parmi tous les codes* (n, k) *qui sont 1-correcteur.*
En particulier, le code de Hamming $(2^m - 1, 2^m - 1 - m)$ *est un code 1-parfait.*

La preuve découle directement des propriétés ci-dessus et nous avons en outre immédiatement la propriété suivante :

Propriété 16. *La distance d'un code de Hamming est 3.*

Preuve. Le code est 1-correcteur donc $\delta(C) \geqslant 2t + 1 = 3$. Il suffit donc d'exhiber deux mots distants de 3 : pour n'importe quel mot, en changeant uniquement le premier bit de donnée (c_3), seuls les deux premiers bits de contrôle sont modifiés. □

Exemple de correction avec le code(7,4) binaire de Hamming. En reprenant l'exemple précédent, supposons que l'on reçoive le mot 1111101. Le message source devrait être 1101 mais le contrôle de parité indique que les bits 2 et 4 sont erronés ; la correction d'une erreur unique est alors réalisée en modifiant le bit d'indice 4+2=6. Le mot corrigé est alors 1111111 qui correspond au mot source 1111.

Exercice 4.19. *On considère le code de Hamming (15,11) de distance 3.*

1. *Quel est son rendement ?*
2. *Vérifier que le code de Hamming(15,11) est équivalent au code* $C(15, 4)$ *défini par une fonction* $\phi_C : \mathbb{F}_2^{11} \to \mathbb{F}_2^{15}$ *de la forme* $\phi_C(s) = s \cdot [I_{11} R]$ *où* I_{11} *est la matrice identité* 11×11 *et* R *une matrice de* $\mathcal{M}_{4,11}(\mathbb{F}_2)$ *que l'on explicitera.*
3. *Donner l'algorithme de codage.*
4. *Donner un algorithme qui détecte jusqu'à 2 erreurs.*
5. *Donner un algorithme qui corrige 1 erreur. Que se passe-t-il en cas de 2 erreurs ?*

Solution page 322.

Pour les applications où le taux d'erreurs est faible, un codage de Hamming peut être utilisé. Le Minitel par exemple utilise un code de Hamming (128,120) qui est 1-correcteur : le message est tronçonné en blocs de 15 octets, i.e. 120 bits ; Un 16ème octet contient 8 bits de contrôle qui permet de localiser un bit d'erreur parmi les 128. En outre un 17ème octet, dit de validation et ne contenant que des 0, est utilisé pour détecter des perturbations importantes. Le code final est donc un code binaire 1-correcteur de paramètres (136,120).
Cependant, les développements des codes cycliques et notamment la mise au point de procédures de codage et de décodage particulièrement efficaces ont motivé l'intégration de ces codes correcteurs dans de nombreuses applications où le taux d'erreurs est important et donc la capacité de correction critique. Les paragraphes suivants montrent quelques-uns de ces codes utilisés par exemple pour la lecture de disques compacts et pour la transmission d'images par satellites.

4.4 Codes linéaires et codes cycliques

4.4.1 Codes linéaires et redondance minimale

Nous avons vu qu'un code correcteur (n, k) est caractérisé par une application $\phi : V^k \to V^n$. L'analyse de ϕ dans un contexte quelconque est difficile ; aussi, on se restreint au cas où ϕ est linéaire. Cette restriction offre des avantages en termes de temps de calcul : le codage et décodage seront effectués assez rapidement, en temps polynomial en n. Pour pouvoir étudier ce cas, il faut que V^k et V^n puissent être munis d'une structure vectorielle, donc que V puisse être muni d'une structure de corps $\mathbb{F}$ (avec les restrictions déjà mentionnées sur le cardinal de V, voir les rappels mathématiques au chapitre 1). Dans toute la suite, on considère indifféremment l'alphabet V et le corps fini $\mathbb{F}$ de même cardinal.
Soit C un code (n, k) sur $\mathbb{F}$. On dit que C est un *code linéaire* de longueur n et de dimension k lorsque C est un sous-espace vectoriel de $\mathbb{F}^n$ de dimension k. Puisque deux espaces vectoriels de même dimension sur un même corps sont isomorphes, un code $C(n, k)$ sur $\mathbb{F}$ est un code linéaire si et seulement s'il existe une application linéaire $\phi : \mathbb{F}^k \longrightarrow \mathbb{F}^n$ telle que $\phi(\mathbb{F}^k) = C$.
Pour une base du sous-espace vectoriel C fixée, l'application linéaire ϕ est caractérisée par la donnée de sa matrice G de dimension $k \times n$ à coefficients dans $\mathbb{F}$. Une telle matrice G est appelée *matrice génératrice* du code C.

Remarque 4. *Parce que les mots de code sont souvent représentés sous forme de vecteurs lignes, on a choisi de représenter ϕ par une matrice rectangulaire dans laquelle les vecteurs générateurs de $Im(\phi)$ sont les lignes.*
Pour un même code linéaire C, il y a autant de matrices génératrices qu'il y a

de choix possibles d'une base de C. Mieux, si G est une matrice génératrice de C, alors pour toute matrice carrée inversible A d'ordre k et à éléments dans $\mathbb{F}$, la matrice $G' = A \cdot G$ est une matrice génératrice de C.
Preuve. $Im(G') = \{x^t.A \cdot G/x \in \mathbb{F}^k\} = \{y^t.G/y \in Im((A)^t)\} = \{y^t.G/y \in \mathbb{F}^k\} = C$. □

Exemple : Considérons le code $(4, 3)$ de parité sur $\mathbb{F}_2$. Le mot de code associé au mot d'information $x^t = [x_0, x_1, x_2]$ est alors $b^t = [b_0, b_1, b_2, b_3]$ défini par :

$$\begin{cases} b_0 = x_0 \\ b_1 = x_1 \\ b_2 = x_2 \\ b_3 = x_0 + x_1 + x_2 \mod 2 \end{cases}$$

Ce code est un code linéaire sur $\mathbb{F}_2$ de matrice génératrice :

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

On a alors $b^t = x^t G$.

Théorème 27. *Soit C un code (n, k) linéaire et soit $d = \min_{c \in C\ ; c \neq 0} w(c)$. Alors*

$$\delta(C) = d,$$

i.e. *C est $(d-1)$-détecteur et $\left\lfloor \frac{d-1}{2} \right\rfloor$-correcteur. Le code C est alors dit* code (n, k, d).

Preuve. Comme ϕ est linéaire, $C = Im(\phi)$ est un sous-espace vectoriel de $\mathbb{F}^n$. Donc $0 \in C$ (0 est le vecteur dont les n composantes sont nulles). D'où, pour tout $c \in C$, $\delta(C) \leqslant d(c, 0) = w(c)$, puis $\delta(C) \leqslant \min_{c \in C} w(c)$.
Réciproquement, soient c_1 et c_2 deux éléments de C tels que $\delta(C) = d(c_1, c_2) = w(c_1 - c_2)$. Comme C est un sous-espace vectoriel, $c = c_1 - c_2$ appartient à C ; $\delta(C) \geqslant \min_{c \in C} w(c)$. Finalement $\delta(C) = \min_{c \in C} w(c)$. □

Théorème 28 (Borne de Singleton). *La distance minimale d d'un code (n, k) linéaire est majorée par :*

$$d \leqslant n - k + 1\,.$$

Preuve. Soit C un code linéaire défini par l'application linéaire ϕ. Comme ϕ est injective, rang$(\phi) = k$. D'où dim(C) = dim(Im(ϕ))=k.

Considérons le sous-espace E de $\mathbb{F}^n$ formé par les vecteurs dont les $k-1$ dernières composantes sont nulles : $\dim(E) = n-k+1$.
Ainsi, E et C sont deux sous-espaces vectoriels de $\mathbb{F}^n$ et $\dim(C) + \dim(E) = n+1 > n$. Il existe donc un élément non nul a dans $C \cap E$. Comme $a \in E$, $w(a) \leqslant n-k+1$; comme $a \in C$, $\delta(C) \leqslant w(a) \leqslant n-k+1$. □

La borne de Singleton impose donc à un code(n,k) linéaire de distance d d'avoir au moins $r \geqslant d-1$ chiffres de redondance. Un code qui atteint cette borne $d = r+1$ est dit *MDS* (*maximum distance separable*).

Exemple : Le code de Hamming binaire $(7,4)$ est un code linéaire qui admet comme matrice génératrice :

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}. \tag{4.5}$$

Sa distance étant 3, c'est un code linéaire $(7,4,3)$.
Plus généralement, le code de Hamming binaire de longueur n est un code $(n, n-1-\lfloor log_2 n \rfloor, 3)$ (nous avons vu qu'un tel code est 1-correcteur). Bien que de distance 3, il n'atteint pas la borne de Singleton car $d = 3 < 2 + \lfloor log_2 n \rfloor$ sauf dans le cas $(n,k) = (3,1)$. Pourtant, nous avons vu qu'il était parfait et qu'il n'existait pas de code binaire de distance 3 avec moins de chiffres de redondance lorsque n n'est pas une puissance de 2.

Théorème 29. *Tout code poinçonné d'un code MDS est MDS.*

Preuve. Soit C un code linéaire MDS (n,k) et G une matrice génératrice. Soit C' le code poinçonnéCode !poinçonné de C obtenu en supprimant u positions dans les mots de code. Le code C' est linéaire $(n-u,k)$ car la sous-matrice G' obtenue en supprimant les colonnes de G aux u positions est génératrice de C'. De plus C' est de distance d' supérieure ou égale à $d-u$. Comme C est MDS, $d = n-k+1$; d'où $d' \geqslant (n-u)-k+1$. Donc, $C'(n-u,k)$ atteint la borne de Singleton $n-u-k+1$ et est donc MDS. □

Exercice 4.20 (Un code MDS est systématique). *Soit G la matrice génératrice d'un code MDS et $\phi(x) = x^t G$ le codage associé. Montrer que pour tout mot de code $c = \phi(x)$, il est possible de reconstruire x de manière unique à partir de la donnée de k coefficients quelconques de c.* *Solution page 322.*

Exercice 4.21. *Montrer que tout code raccourci d'un code MDS est MDS.*
Solution page 323.

4.4.2 Codage et décodage des codes linéaires

Le décodage et la correction d'erreur, si le code n'a pas de structure particulière, peut être très long, voire impossible pour des très grands codes. Il consisterait, pour chaque mot reçu, à le comparer avec tous les mots de C et à l'identifier au plus proche. En pratique, la structure linéaire des codes conduit à des algorithmes de codage et décodage assez efficaces lorsque le nombre d'erreurs t corrigées par le code est petit.

Pour l'étude du codage et du décodage du code linéaire C, nous pouvons nous limiter au cas où la matrice génératrice G, s'écrit sous la forme dite *systématique* :

$$G = \left[\begin{array}{c|c} \quad L \quad & R \end{array}\right] \tag{4.6}$$

où L est une matrice carrée $k \times k$ inversible, et R une matrice $k \times r$.
En effet, si G n'est pas sous cette forme, on permute les colonnes de G jusqu'à obtenir L inversible ; ceci est possible puisque $\mathrm{rg}(G) = k$. On obtient alors une matrice génératrice G_σ d'un code linéaire C_σ équivalent à C. Après le codage et la correction à partir de C_σ, on retrouve le message initial en ré-appliquant la matrice de permutation.
On construit alors $G' = L^{-1}G$, soit :

$$G' = \left[\begin{array}{c|c} \quad I_k \quad & T = L^{-1}R \end{array}\right]$$

La matrice G' est également génératrice de C, d'après la remarque 4.
La matrice G' est appelée matrice génératrice *normalisée (ou canonique)* du code C. Tout code linéaire possède une matrice génératrice normalisée.
Cette matrice normalisée est de plus unique. En effet, soit $[I_k|T_1]$ et $[I_k|T_2]$ deux matrices génératrices de C. Pour tout $x \in \mathbb{F}^k$, $[x^t, x^tT_1]$ et $[x^t, x^tT_2]$ sont donc dans C. Comme C est un sous-espace vectoriel, leur différence $[0, x^t(T_1 - T_2)]$ est aussi dans C. Or, le seul élément de C dont les k premières composantes sont nulles est 0. D'où, quel que soit $x \in \mathbb{F}^k$, $x^tT_1 = x^tT_2$, soit $T_1 = T_2$.

Codage par multiplication matrice-vecteur

On déduit de ce qui précède une méthode de codage de C : tout mot source $u \in \mathbb{F}^k$ (vecteur ligne) est codé par $\phi(u) = uG' = [u, u.T]$. Ainsi, on écrit u suivi de uT qui contient les chiffres de redondance.

Décodage par multiplication matrice-vecteur

La méthode de détection d'erreur et de décodage utilise la propriété suivante :

Propriété 17. *Soit H la matrice $r \times n$:* $H = \left[\begin{array}{c|c} T^t & -I_r \end{array}\right]$. *Alors* $x^t = [x_1, \dots x_n] \in C$ *si et seulement si* $Hx = 0$.
H *est appelée* matrice de contrôle *de* C.

Preuve. Soit $x \in C$. Il existe $u \in \mathbb{F}^k$ tel que $x = u[I_k|T]$. D'où $xH^t = u[I_k|T]\begin{bmatrix} T \\ -I_r \end{bmatrix}$. Or, $[I_k|T]\begin{bmatrix} T \\ -I_r \end{bmatrix} = [T - T] = [0]$; par conséquent pour tout x, $Hx = 0$.
Réciproquement, si $Hx = 0$ alors $[x_1, \dots, x_n]\begin{bmatrix} T \\ -I_r \end{bmatrix} = [0, \dots, 0]_r$.
La composante j $(1 \leqslant j \leqslant r)$ donne alors $[x_1, \dots, x_k]\begin{bmatrix} T_{1,j} \\ \vdots \\ T_{k,j} \end{bmatrix} - x_{k+j} = 0$
Soit $x_{k+j} = [x_1, \dots, x_k]\begin{bmatrix} T_{1,j} \\ \vdots \\ T_{k,j} \end{bmatrix}$. D'où $[x_{k+1}, \dots, x_{k+r}] = [x_1, \dots, x_k]T$.
Finalement on a $[x_1, \dots, x_n] = [x_1, \dots, x_k][I_k|T] = [x_1, \dots, x_k]G'$, et $x \in C$. □

Pour détecter une erreur si on reçoit un mot y, il suffit donc de calculer Hy, qu'on appelle le *syndrome d'erreur*. On détecte une erreur si et seulement si $Hy \neq 0$. On cherche alors à calculer le mot émis x à partir de y. Pour cela, on calcule le vecteur d'erreurs $e = y - x$. Ce vecteur e est l'unique élément de $\mathbb{F}^n$ de poids de Hamming $w_H(e)$ minimal tel que $He = Hy$. En effet, si C est t-correcteur, il existe un unique $x \in C$, tel que $d_H(x, y) \leqslant t$. Comme $d_H(x, y) = w_H(x - y)$, on en déduit qu'il existe un unique $e \in \mathbb{F}^n$, tel que $w_H(e) \leqslant t$. De plus $He = Hy - Hx = Hy$ car $Hx = 0$.

On construit alors une table de $|\mathbb{F}|^r$ entrées, dans laquelle chaque entrée i correspond à un élément unique z_i de $\mathrm{Im}(H)$; dans l'entrée i, on stocke le vecteur $e_i \in \mathbb{F}^n$ de poids minimal et tel que $He_i = z_i$.
La correction est alors triviale. Quand on reçoit y, on calcule Hy et l'entrée i de la table tel que $z_i = Hy$. On trouve alors e_i dans la table et on renvoie $x = y - e_i$.

Exemple : Considérons le code de Hamming (7,4) dont la matrice génératrice G est donnée par la relation (4.5) (voir à la page 242). Dans ce cas, on a :

$$L = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \quad ; \quad R = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{et} \quad T = L^{-1}R = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} .$$

La matrice génératrice canonique G' et la matrice de contrôle H sont données par :

$$G' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{et} \quad H = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} .$$

Si l'on reçoit le mot $y = 1111101$, alors le calcul $Hy = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ montre qu'il y a erreur. Pour corriger l'erreur, il suffit (la table à $2^3 = 8$ entrées n'est pas nécessaire dans cet exemple simple) de constater que le vecteur $e = 0000010$ est de poids (de Hamming) minimal et tel que $He = Hy$. Donc la correction de y est $x = y + e = 1111111$.

Exercice 4.22 (Code dual). *Soit C un code linéaire (n, k) sur $\mathbb{F}$. Le* code dual *de C, noté $C^\perp$, est l'ensemble des mots de $\mathbb{F}^n$ orthogonaux à C :*

$$C^\perp = \{x \in \mathbb{F}^n, \text{ tel que pour tout } c \in C \ : x.c^t = 0\}.$$

Montrer que $C^\perp$ est un code linéaire $(n, n-k)$ sur $\mathbb{F}$ engendré par la matrice de contrôle H du code C. *Solution page 323.*

Exercice 4.23 (Code de Golay). *On considère le code de Golay $\mathcal{G}_{12}$ ternaire (i.e. sur $V = \mathbb{F}_3$) de distance 6 et de matrice génératrice $G = [I_6|R]$ avec*

$$R = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 2 & 2 & 1 \\ 1 & 1 & 0 & 1 & 2 & 2 \\ 1 & 2 & 1 & 0 & 1 & 2 \\ 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 0 \end{bmatrix}$$

1. *Donner les caractéristiques de $\mathcal{G}_{12}$ et sa matrice de contrôle.*

2. *On vérifie facilement que si r et s sont deux lignes quelconques de G, alors $r.s = 0$. En déduire que $\mathcal{G}_{12}$ est* auto-dual, i.e. $\mathcal{G}_{12} = \mathcal{G}_{12}^{\perp}$.
3. *Montrer que $\mathcal{G}_{12}$ n'est pas parfait.*
4. *Soit $\mathcal{G}_{11}$ le code obtenu à partir de $\mathcal{G}_{12}$ en supprimant sa dernière composante ; expliciter la matrice de contrôle associée à $\mathcal{G}_{11}$. Quelle est sa distance ?*
5. *Montrer que $\mathcal{G}_{11}$ est un code parfait.*

Solution page 323.

En s'intéressant aux codes linéaires, on a donc nettement amélioré les performances du codage et décodage avec correction d'erreur. Il existe une classe de codes linéaires, appelée classe des codes cycliques, qui améliore encore la facilité du calcul, et la possibilité de l'implémenter simplement sur des circuits électroniques. De plus, grâce aux codes cycliques, on verra aussi des méthodes simples pour construire des codes en garantissant un taux de correction.

4.4.3 Codes cycliques

On appelle *opération de décalage cyclique* l'application linéaire σ de V^n dans lui-même définie par

$$\sigma([u_0, \ldots, u_{n-1}]) = [u_{n-1}, u_0, \ldots, u_{n-2}]\,.$$

L'opération σ est linéaire, et on peut exhiber sa matrice Σ :

$$\Sigma = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 1 \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

Un *code cyclique* est un code linéaire qui est stable pour l'opération de décalage de chiffre. Autrement dit, un code linéaire C de V^n est cyclique lorsque

$$\sigma(C) = C\,.$$

Exemple : Le code de parité $(n, n-1)$ est un code cyclique. En effet, si $c = (c_0, \ldots, c_{n-1})$ est un mot de code, alors $c_{n-1} = \sum_{i=0}^{n-2} c_i \mod 2$. Mais on a alors aussi $c_{n-2} = c_{n-1} + \sum_{i=0}^{n-3} c_i \mod 2$; ainsi, $\sigma(c) = (c_{n-1}, c_0, \ldots, c_{n-2})$ est aussi un mot de code. Le code de parité est donc cyclique. Cela peut se

voir également directement sur les matrices : en effet, on associe au code de parité sa matrice génératrice

$$G = \begin{bmatrix} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \ddots & \vdots & 1 \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & 1 \end{bmatrix}$$

Alors la remarque 4 nous donne que ΣG et G génèrent le même code car Σ est inversible.

Caractérisation : polynôme générateur

Tout élément $U = [u_0, \ldots, u_{n-1}]$ de V^n peut être représenté par le polynôme de degré $n-1$ de $V[X]$ (voir la section 1.3.4 du chapitre 1 page 63)

$$P_U = \sum_{i=0}^{n-1} u_i X^i.$$

Comme $X^n - 1$ est un polynôme de degré n, V^n est isomorphe à $V[X]/(X^n - 1)$. Dans $V[X]/(X^n - 1)$, on a alors

$$P_{\sigma(U)} = X.P_U(X) - u_{n-1}.(X^n - 1) = X.P_U \mod (X^n - 1)$$

Autrement dit, l'opération de décalage σ d'un vecteur revient à multiplier son polynôme associé par X dans $V[X]/(X^n - 1)$.
Cette propriété est à la base du théorème suivant qui donne une caractérisation algébrique d'un code cyclique.

Théorème 30. *Tout code cyclique $C = (n, k)$ admet une matrice génératrice G_C de la forme :*

$$G_C = \begin{bmatrix} m \\ \sigma(m) \\ \cdots \\ \sigma^{k-1}(m) \end{bmatrix}$$

avec $m = [a_0, a_1, \ldots, a_{n-k} = 1, 0, \ldots, 0]$ tel que $g(X) = \sum_{i=0}^{n-k} a_i X^i$ est un diviseur unitaire de $(X^n - 1)$ de degré $r = n - k$.
Le polynôme g est appelé polynôme générateur *du code cyclique.*
Réciproquement, tout diviseur g de $(X^n - 1)$ est polynôme générateur d'un code cyclique.

Preuve. Nous avons tout d'abord besoin du lemme suivant :

Lemme 3. *Soit $C(X)$ l'ensemble des polynômes associés aux mots du code cyclique C. Alors $C(X)$ est un idéal de $V[X]/(X^n-1)$.*

En effet, $C(X)$ est clairement un groupe. Ensuite, pour tout P_U de $C(X)$, $P_{\sigma(U)} = XP_U$ est dans $C(X)$ car C est cyclique. De la même manière, tous les X^iP_U sont également dans C. Enfin, par linéarité toutes les combinaisons linéaires des précédents restent dans C. Autrement dit pour tout $A(X) \in V[X]/(X^n-1)$, $A(X)P_U$ est dans C.
Retour à la preuve du théorème. L'anneau $V[X]/(X^n-1)$ est principal (voir section 1.3.4), donc $C(X)$ est généré par un certain polynôme $g(X)$ de degré minimal r.
Montrons que $r = n-k$. Comme $g(X)$ est un générateur, tous les éléments du code s'écrivent $A(X)g(X)$. Pris modulo X^n-1, $A(X)$ est donc de degré $n-r-1$; il y a donc exactement $|V|^{n-r}$ éléments distincts du code. Ce qui n'est possible que si $n-r=k$.
Il reste à montrer que le polynôme g est un diviseur de X^n-1. On écrit la division euclidienne $X^n-1 = Q(X)g(X)+R(X) = 0 \mod X^n-1$ avec degré(R) <degré(g). Or 0 et $Q(x)g(X)$ sont dans l'idéal $C(X)$, donc par linéarité $R(X)$ aussi. Mais g est générateur de $C(X)$ de degré minimal donc $R(X)=0$ et donc g divise X^n-1.
Enfin, pour la réciproque, on écrit

$$G_C = \begin{bmatrix} g(X) \\ Xg(X) \\ \dots \\ X^{n-r-1}(m) \end{bmatrix}$$

Ce code est linéaire et $(n, n-r)$, donc $k = n-r$. Ensuite, il est cyclique car tout mot de code s'écrit UG_C ou encore $\sum_{i=0}^{n-r-1} u_iX^ig(X)$ et donc $\sigma(UG_C) = \sum_{i=0}^{n-r-1} u_iX^{i+1}g(X) = u_{n-r-1}g(X) + \sum_{i=0}^{n-r-2} u_iX^{i+1}g(X) \mod X^n-1$ est bien une combinaison linéaire des puissances de g et donc un mot du code généré par les puissances de g. □

Cette propriété permet la construction directe de code correcteur par la donnée d'un polynôme diviseur de X^n-1. Pour reconnaître ces diviseurs, il est possible de factoriser X^n-1. Cela se fait en temps raisonnable par exemple par l'algorithme Cantor-Zassenhaus donné en exercice ci-dessous.

Exercice 4.24 (Factorisation probabiliste de Cantor-Zassenhaus de polynômes sur un corps premier). *La complexité (en nombre d'opérations arithmétiques sur un corps premier $\mathbb{Z}_p$) de l'addition de deux polynômes de degré au plus d sera notée $A_d = O(d)$, la complexité de la multiplication ou de la division sera notée M_d,celle de l'algorithme d'Euclide étendu sera notée*

E_d.Le but de cet exercice est d'évaluer la complexité de la factorisation de polynômes sans carrés sur un corps fini.

1. *En adaptant l'algorithme 7 page 61, en déduire un algorithme* `FDD` *(Facteurs de Degrés Distincts) factorisant un polynôme P en $P = P_1 \dots P_k$ où P_i est un produit de polynômes irréductibles de degré i.*
2. *Que vaut k au maximum ? En déduire la complexité de* `FDD`.
3. *Si $p = 2$ comment factoriser P_1 ?*
4. *Dans toute la suite, nous supposons donc que p est impair et $i \geqslant 2$. Il ne reste plus qu'à séparer les facteurs irréductibles des P_i. Montrer que pour tout polynôme $T = \sum a_j X^j$, $X^{p^i} - X$ divise $T^{p^i} - T$ dans $\mathbb{Z}_p$.*
5. *En déduire que pour tout T, les facteurs irréductibles de P_i sont partagés entre T, $T^{\frac{p^i-1}{2}} - 1$ et $T^{\frac{p^i-1}{2}} + 1$.*
6. *En pratique, Cantor et Zassenhaus ont montré que si T est quelconque de degré $< 2i$, il y a environ une chance sur deux que les facteurs de P_i ne soient pas tous dans le même des trois polynômes ci-dessus. En déduire un algorithme de factorisation probabiliste* `SF` *(Séparation de Facteurs). Comment réduire la taille des polynômes et quelle est alors sa complexité en moyenne ?*

Solution page 324.

Le lien entre code cyclique et polynôme générateur est important ; il est à la base des algorithmes efficaces de codage et de décodage qui évitent la multiplication par la matrice génératrice G_C (de coût $O(nk)$) en la remplaçant par une multiplication par le polynôme g, de coût $O(n \log n)$. L'intérêt pratique est en outre que les circuits associés au codage et décodage sont relativement simples à réaliser, par exemple par des registres à décalage linéaire (LFSR).

Opération de codage

Soit C un code cyclique de polynôme générateur g ; soit G une matrice génératrice de C construite à partir de g comme dans le théorème 30. Soit $a = [a_0, \dots, a_{k-1}] \in V^k$ un mot source et $P_a = \sum_{i=0}^{k-1} a_i X^i$ son polynôme associé. Le mot de code associé à a est $\phi(a) = aG$ de polynôme associé P_{aG}. Par l'écriture de G à partir des coefficients de $g(X)$ (théorème 30), on a :

$$\begin{aligned} P_{aG} &= \sum_{i=0}^{k-1} a_i (X^i g(X) \mod X^n - 1) \\ &= [g(X)(\sum_{i=0}^{k-1} a_i X^i)] \mod X^n - 1 \\ &= [g(X).P_a(X)] \mod X^n - 1 \ . \end{aligned}$$

Le codage correspond donc à un produit de polynômes, les degrés des monômes étant pris modulo n : en effet, calculer modulo $X^n - 1$ revient à considérer que $X^n = 1 = X^0$.

Exemple : Soit le code cyclique (7,4) associé au polynôme $g = (1 + X^2 + X^3)$. Soit $a = [a_0, a_1, a_2, a_3]$ un mot source ; le code de ce mot est alors égal à aG ; il est obtenu par multiplication à droite par la matrice G. Le même mot est obtenu en calculant les coefficients du polynôme :

$$(a_0 + a_1X + a_2X^2 + a_3X^3)(1 + X^2 + X^3) \mod X^7 - 1\,.$$

Ainsi, pour $a = [1001]$: $g \cdot P_a \mod X^7 - 1 = (1 + X^2 + X^3)(1 + X^3) = 1 + X^2 + 2.X^3 + X^5 + X^6 = 1 + X^2 + X^5 + X^6$. Donc $\phi(a) = [1010011]$.

Détection d'erreur et opération de décodage

Le codage précédent montre que tout mot de code reçu est un multiple de $g(X)$. Soit $m \in V^n$ un message reçu et soit P_m son polynôme associé. Lors du décodage, on calcule d'abord $P_e = P_m \mod g$;

- si $P_e = 0$: le polynôme reçu est bien un multiple de g : il correspond donc à un mot de code. On retrouve le message émis en calculant le quotient P_m/g.
- sinon, $P_e \neq 0$: le mot reçu n'est pas un mot de code et il y a donc eu des erreurs lors de la transmission.

La détection d'erreur est donc réalisée via le calcul de $P_e = P_m$ modulo g ; P_e est le *syndrome d'erreur*.

Dans le cas où P_e est non nul, on peut procéder à une correction. Une manière brutale est de tabuler tous les mots de V^n en associant à chacun le mot de code le plus proche, qui correspond au message corrigé. La correction se fait alors par lecture de la table.

Le lien entre code cyclique et polynômes fait qu'il existe des algorithmes moins coûteux en place mémoire pour corriger un message erroné. La méthode de Meggitt et ses variantes en sont des exemples.

La détection d'erreurs est donc une division de polynômes, qui se ramène, comme le codage, à une multiplication de polynômes de coût $O(n \log n)$. L'algorithme de Berlekamp-Massey (voir sa description page 94) permet de réaliser la correction pour un code cyclique avec un coût similaire.

Exercice 4.25. *On a montré que tout code cyclique est caractérisé par la donnée d'un polynôme générateur, diviseur unitaire de degré $r = n - k$ de $X^n - 1$. Le code C est donc caractérisé par la donnée de seulement $r = n - k$ coefficients $g_0, \ldots, g_{n-k-1}$. Justifier que réciproquement, la donnée d'un polynôme g diviseur unitaire de degré $n - k$ de $X^n - 1$ définit un unique code cyclique (n, k).* *Solution page 324.*

4.4.4 Codes BCH

Le taux de correction d'un code cyclique est difficile à calculer. Cependant, ce paragraphe présente un théorème qui permet de garantir une minoration de la distance d'un code, et par suite (d'après le théorème 27) une minoration du taux de détection.

Distance garantie

Nous nous plaçons dans un corps fini $\mathbb{F}_q$ et rappelons la notion de racine primitive $n^{\text{ième}}$ de l'unité vue en section 1.4.2 : c'est une racine simple du polynôme $X^n - 1$, d'ordre n (en général l'ordre d'une racine $n^{\text{ième}}$ quelconque de l'unité γ est le plus petit entier o tel que $\gamma^o = 1$ et il vaut donc *au plus* n). Un intérêt des racines primitives $n^{\text{ième}}$ de l'unité est que l'on peut les utiliser pour trouver des codes de distance minimale fixée.

Théorème 31. *Soit C un code cyclique (n, k) de polynôme générateur $g \in \mathbb{F}_q[X]$ avec n premier avec q et soit β une racine primitive $n^{ième}$ de l'unité. S'il existe deux entiers l et s tels que $g(\beta^l) = g(\beta^{l+1}) = \ldots = g(\beta^{l+s-1}) = 0$ alors*

$$\delta(C) \geqslant s + 1.$$

Le code C est donc au moins s-détecteur et $\lfloor s/2 \rfloor$-correcteur.

Preuve. Comme le code est linéaire, il suffit de montrer qu'aucun mot de code n'est de poids de Hamming inférieur à $s + 1$. Par l'absurde, supposons qu'il existe un mot de code $m = m_0 + m_1 X + \ldots m_{n-1} X^{n-1}$ de poids w tel que $1 \leqslant w < s + 1$. Comme m est un multiple de g, m vérifie également $m(\beta^l) = m(\beta^{l+1}) = \ldots = m(\beta^{l+s-1}) = 0$. Cela se réécrit également sous la forme matricielle suivante où l'on n'a gardé que les lignes pour lesquelles $m_{i_k} \neq 0$:

$$\begin{bmatrix} \beta^{li_1} & \ldots & \beta^{li_w} \\ \beta^{(l+1)i_1} & \ldots & \beta^{(l+1)i_w} \\ \vdots & \ddots & \vdots \\ \beta^{(l+w-1)i_1} & \ldots & \beta^{(l+w-1)i_w} \end{bmatrix} \begin{bmatrix} m_{i_1} \\ m_{i_2} \\ \vdots \\ m_{i_w} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Le système linéaire ainsi écrit est carré et n'existe que si $w \geqslant s$ car il y a seulement s racines g de la forme β^{l+i} donnant donc un second membre nul. Comme les m_{i_k} sont non nuls, ce système n'est possible que si la matrice est singulière. Or cette matrice est de type Vandermonde et son déterminant vaut $\beta^{l(i_1+\ldots+i_w)} \prod_{i_j<i_k} (\beta^{i_j} - \beta^{i_k})$. Ce dernier est non nul car β est une racine primitive $n^{\text{ième}}$. Ceci est absurde, et donc w doit être strictement supérieur à s. □

Construction des polynômes générateurs BCH

En application de ce théorème, Bose, Chaudhuri et Hocquenghem ont proposé une méthode de construction de codes cycliques ayant un taux de correction arbitraire.
Nous considérons que n est premier avec q. Ensuite, prenons β une racine primitive $n^{\text{ième}}$ de l'unité. Alors on peut écrire $X^n - 1 = \prod(X - \beta^i)$. On peut également écrire la factorisation $X^n - 1 = P_1(X) \dots P_k(X)$ en polynômes irréductibles de $\mathbb{F}_q[X]$. Or, comme n est premier avec q, toutes les racines de $X^n - 1$ sont distinctes et sont donc les racines d'un et d'un seul des facteurs irréductibles P_i. On note i_k, l'indice du polynôme irréductible dont β^k est racine. Bose, Chaudhuri et Hocquenghem ont alors proposé d'utiliser comme polynôme générateur d'un code le polynôme

$$g(X) = \text{ppcm}\{P_{i_{k+1}}(X), \dots, P_{i_{k+s}}(X)\}.$$

Un tel code est appelé *code BCH.*

Corollaire 3. *Un code BCH est de distance minimale supérieure à $s + 1$ et donc de taux de correction garanti supérieur à $\lfloor s/2 \rfloor$.*

Preuve. Dans une extension, on voit que $g(X) = H(X) \cdot \prod_{i=1}^{s}(X - \beta^{k+i})$ pour un certain polynôme H. Le théorème 31 donne alors la borne inférieure sur la distance. □

Exemple : Pour fabriquer un code 2-correcteur dans $\mathbb{F}_5$, prenons $\mathbb{F}_5[X]$ et par exemple $X^8 - 1$. Alors on calcule, par exemple par l'algorithme de l'exercice 4.24, page 248 :

$$X^8 - 1 = (X^2 + 2)(X^2 + 3)(X + 1)(X - 1)(X - 2)(X - 3) \mod 5.$$

Clairement 1, 2, 3, -1 sont d'ordre plus petit que 5 et ne peuvent donc pas être racines primitives $8^{\text{ième}}$ de l'unité. Dans une extension on note β une racine de $X^2 + 2$. Alors $\beta^2 = -2 = 3$ et donc $\beta^3 = 3\beta$, $\beta^4 = 4$, $\beta^5 = 4\beta$, $\beta^6 = 2$, $\beta^7 = 2\beta$ et $\beta^8 = 6 = 1$. Nous avons donc de la chance, car il existe bien une racine primitive $8^{\text{ième}}$ de l'unité dans $\mathbb{F}_5$. Si cela n'avait pas été le cas, il aurait fallu essayer d'autres polynômes de la forme $X^n - 1$ avec $n \neq 8$. Ensuite, si l'on veut un code 2-correcteur, il faut trouver les polynômes irréductibles ayant β^i comme racine pour i par exemple de 1 à 4. β est racine de $X^2 + 2$, $\beta^2 = 3$ est racine de $X - 2$, $\beta^3 = 3\beta$, vérifiant $(3\beta)^2 = 4 * 3 = 2$, est donc racine de $X^2 + 3$ et enfin, $\beta^4 = 4 = -1$ est racine de $X + 1$. On peut donc prendre $g(X) = (X^2 + 2)(X^2 + 3)(X + 1)(X - 3)$. À noter néanmoins que l'on obtient un code au moins 2-correcteur avec 7 caractères de redondance sur 8, soit un rendement de seulement $0,125$.

Exercice 4.26 (Code BCH plus économique).

1. *Trouver une racine primitive* $16^{ième}$ *de l'unité modulo* 5 *à l'aide de* $X^{16} - 1 = (X^4 + 2)(X^4 + 3)(X^8 - 1) \mod 5$.
2. *Comment construire un code* 2*-correcteur plus économique que celui de l'exemple ci-dessus.*

Solution page 324.

Sur cet exemple, il est clair que l'on peut construire des codes avec un taux de correction choisi ; il faut toutefois pour cela arriver à trouver une racine primitive adéquate.
En pratique, les codes BCH les plus utilisés correspondent à $n = q^t - 1$; un tel code est dit *primitif*. Parmi ceux-ci, les codes BCH ayant $n = q - 1$ sont appelés *codes de Reed-Solomon* et nous allons voir qu'ils sont particulièrement économiques et plus faciles à construire.

Codes BCH optimaux : codes de Reed-Solomon

Tout code BCH primitif a une longueur $n = q^t - 1$ et nécessite par conséquent que n soit premier avec q. Les codes de Reed-Solomon vérifient en outre plus précisément $n = q - 1$. Dans ce cas, il existe toujours des racines primitives $n^{\text{ièmes}}$ de l'unité, ce sont les racines primitives classiques de $\mathbb{F}_q$ (voir section 1.3.5). Nous allons voir qu'il est alors toujours possible de ne choisir que des polynômes irréductibles sur $\mathbb{F}_q[X]$ de degré 1, comme polynômes minimaux de chacun des β^{k+i}. Le polynôme générateur est alors de degré global minimal, et donc le code est de redondance minimale, pour un taux de correction donné.
L'intérêt de ces codes est double. D'une part, ils sont optimaux dans le sens où ils requièrent un nombre de chiffres de redondance minimal pour une capacité de correction fixée. D'autre part, ils sont particulièrement rapides à coder et décoder.
Construisons donc un polynôme générateur de Reed-Solomon. Le polynôme $X^{q-1} - 1$ se factorise très simplement sur $\mathbb{F}_q$: ses racines sont tous les éléments non nuls de $\mathbb{F}_q$.

$$X^{q-1} - 1 = \prod_{\lambda \in \mathbb{F}_q - \{0\}} (X - \lambda)\,.$$

Le groupe multiplicatif $\mathbb{F}_q - \{0\}$ étant cyclique, soit α une racine primitive. Alors $\alpha \in \mathbb{F}_q$ et donc aucune extension de scindage n'est nécessaire pour avoir :

$$X^{q-1} - 1 = \prod_{i=1}^{q-1} (X - \alpha^i)\,.$$

La construction d'un code de Reed-Solomon avec r chiffres de redondance repose sur le choix d'un polynôme générateur de degré r, dont les racines sont consécutives :

$$g(X) = \prod_{i=s}^{s+r-1} (X - \alpha^i)\,.$$

Le code de Reed-Solomon ainsi obtenu est donc un code $(n = q-1, k = n-r)$ avec r arbitraire.
Le taux de correction de ce code est optimal. En effet, en tant que code BCH, la distance minimale δ du code de Reed-Solomon est au moins $\delta \geqslant r+1$. Or, la borne de Singleton (théorème 28) montre que $\delta \leqslant n-k+1 = r+1$; ainsi, la distance est $\delta = r+1$ et atteint la borne de Singleton.
Les codes de Reed-Solomon sont très utilisés en pratique.

Exemple : Le satellite d'exploration de Jupiter *Galileo* utilise le code de Reed-Solomon (255,223) de distance 33. Ce code est 16-correcteur sur le corps $\mathbb{F}_{2^8}$. Si α est une racine primitive de $\mathbb{F}_{2^8}$ (ici c'est une racine quelconque du polynôme primitif $T^8+T^7+T^2+T+1$ utilisé pour engendrer $\mathbb{F}_{2^8}$), le polynôme générateur de degré $r = 32$ du code de Reed-Solomon (255,223,33) choisi est[1] :

$$g = \prod_{j=12}^{43} (X - \alpha^{11j}).$$

Décodage des codes de Reed-Solomon

Le décodage des codes de Reed-Solomon peut se faire très rapidement car calcul du syndrome d'erreur se ramène dans ce cas à un calcul rapide de transformée de Fourier ! En effet, la matrice de contrôle H définie section 4.4.2 d'un code de Reed-Solomon peut s'exprimer facilement comme ceci :

$$H = \left[\begin{array}{ccccc} 1 & \alpha & \alpha^2 & \dots & \alpha^{n-1} \\ \hline 1 & \alpha^2 & (\alpha^2)^2 & \dots & (\alpha^2)^{n-1} \\ \hline \vdots & \vdots & \vdots & & \vdots \\ \hline 1 & \alpha^r & (\alpha^r)^2 & \dots & (\alpha^r)^{n-1} \end{array}\right] = \left[\begin{array}{c} L_\alpha \\ \hline L_{\alpha^2} \\ \hline \dots \\ \hline L_{\alpha^r} \end{array}\right]$$

[1] $g = X^{32} + [96]X^{31} + [E3]X^{30} + [98]X^{29} + [49]X^{28} + [B2]X^{27} + [82]X^{26} + [28]X^{25} + [32]X^{24} + [F1]X^{23} + [64]X^{22} + [C2]X^{21} + [70]X^{20} + [F0]X^{19} + [72]X^{18} + [4C]X^{17} + [76]X^{16} + AFX^{15} + [20]X^{14} + [65]X^{13} + [9F]X^{12} + [6B]X^{11} + [8B]X^{10} + [69]X^9 + [29]X^8 + [57]X^7 + [D3]X^6 + [6A]X^5 + [7F]X^4 + [C4]X^3 + [90]X^2 + [D6]X + [7C]$.

La raison est que si G est la matrice génératrice du code, alors GH^T s'exprime comme l'application du polynôme g au différents coefficients de H, les α^i, décalé de X à chaque ligne :

$$\begin{aligned} GH^T &= \left[\begin{array}{c} g(X) \\ \hline Xg(X) \\ \hline \vdots \\ \hline X^{r-1}g(X) \end{array}\right] \cdot \left[\begin{array}{c|c|c|c} L_\alpha^T & L_{\alpha^2}^T & \dots & L_{\alpha^r}^T \end{array}\right] \\ &= \left[\begin{array}{cccc} g(\alpha) & g(\alpha^2) & \dots & g(\alpha^{r-1}) \\ \alpha g(\alpha) & \alpha g(\alpha^2) & \dots & \alpha g(\alpha^{r-1}) \\ \vdots & \vdots & & \vdots \\ \alpha^{r-1}g(\alpha) & \alpha^{r-1}g(\alpha^2) & \dots & \alpha^{r-1}g(\alpha^{r-1}) \end{array}\right] \end{aligned} \tag{4.7}$$

Or, cette matrice GH^T est nulle car les α^i sont les racines du polynôme $g(X)$; en outre, H est de rang r car α est une racine primitive. On a donc bien que H est *la* matrice de contrôle du code.

Ensuite, un mot de code $b = b_0 + b_1X + \dots b_{n-1}X^{n-1}$ est un multiple de g ($b(X) = u(X)g(X)$ où u est le message initial), puisqu'un code de Reed-Solomon est cyclique. L'interprétation duale dans le cas particulier des codes de Reed-Solomon est qu'un mot de code est caractérisé par le fait que les r premiers termes de sa transformée de Fourier sont nuls :

$$bH^T = \sqrt{n}DFT_{1..r}(b).$$

Cela découle de la formule 1.9, page 88 et du fait que $0 = ugH^T = bH^T$ car H est la matrice de contrôle du code.

Nous pouvons alors maintenant décoder. Posons b le mot de code transmis, c le mot reçu et e l'erreur commise, donc telle que $c = b + e$. Puisque b est un mot de code, sa multiplication par la matrice de contrôle doit donner un vecteur nul, et donc les premiers termes de sa transformée de Fourier sont nuls. Cela induit que les premiers termes des transformées de Fourier discrètes de c et e sont égaux : $DFT_{1..r}(c) = DFT_{1..r}(e)$.

Le problème qu'il reste à résoudre est donc de reconstituer e à partir des seuls premiers termes de sa transformée de Fourier. L'astuce est que les coefficients de la DFT d'un mot sont linéairement générés par un polynôme de degré le poids de Hamming de e (le nombre $w(e)$ d'erreurs commises, voir la section 4.3.1) :

Théorème 32 (Blahut). *Les coefficients de la DFT de e sont linéairement générés par un polynôme de degré $w(e)$.*

Preuve. Tout d'abord, complétons la série E associée à la DFT de e : on a $E(X) = \sum_{i=0}^{\infty} E_i X^i$ où les termes E_i pour $i \geqslant n$, valent $E_i = E_{i \bmod n} = \frac{1}{\sqrt{n}} e(\alpha^i) = \frac{1}{\sqrt{n}} e(\alpha^{i \bmod n})$ car α étant une racine primitive, elle est d'ordre $n = q - 1$ dans $\mathbb{F}_q$. Ainsi, $E(X) = \frac{1}{\sqrt{n}} \sum_{i=0}^{\infty} e(\alpha^i) X^i$ ou encore $E(X) = \frac{1}{\sqrt{n}} \sum_{i=0}^{\infty} \left(\sum_{j=0}^{n-1} e_j \alpha^{ij} \right) X^i = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} e_j \sum_{i=0}^{\infty} (\alpha^j X)^i$. Or cette dernière formule inclut le développement en série entière de $\frac{1}{1-\alpha^j X}$. Cette remarque nous permet alors d'écrire

$$E(X) = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} e_j \frac{1}{1-\alpha^j X} = \frac{1}{\sqrt{n}} \sum_{j=0 \ et \ e_j \neq 0}^{n-1} e_j \frac{1}{1-\alpha^j X} = \frac{R(X)}{\Pi(X)}$$

et donc $E(X)$ est le développement en éléments simples d'une fraction rationnelle $\frac{R(X)}{\Pi(X)}$, avec degré$(R) <$ degré(Π) et

$$\Pi(X) = \prod_{j=0 \ et \ e_j \neq 0}^{n-1} 1 - \alpha^j X$$

Donc degré$(\Pi) = w(e)$ et Π est le polynôme minimal générateur de la séquence $E(X)$ calculé par l'algorithme de Berlekamp-Massey, comme on le voit sur l'équation 1.11, page 96. □

De cette preuve constructive, on déduit immédiatement le taux de correction d'un code de Reed-Solomon et l'algorithme de décodage des codes de Reed-Solomon :

Algorithme 27 Décodage des Reed-Solomon.

Entrées c un mot comportant moins de $\frac{r-1}{2}$ erreurs.
Sorties b le mot de code corrigé.

$[E_0, \ldots, E_{r-1}] = DFT_{1..r}(c)$
$\Pi(X) = Berlekamp - Massey(E_{0..r-1})$
$E = LFSR_{\Pi}(E_0, \ldots, E_{r-1})$
$e = DFT^{-1}(E)$
Renvoyer $b = c - e$

Corollaire 4. *Les codes de Reed-Solomon sont $\frac{r-1}{2}$ correcteurs.*

Preuve. Par définition, c'est le taux maximal possible. Ensuite la preuve du théorème précédent montre que dès que $2w(e) < r - 1$ l'algorithme de Berlekamp-Massey aura suffisamment de termes pour trouver un générateur de degré $w(e)$ et compléter le syndrome. □

Corollaire 5. *Les codes de Reed-Solomon sont décodables en temps quasi linéaire.*

Preuve. La complexité du calcul de la DFT ou de son inverse est quasi-linéaire, en $O(n\log(n))$ opérations. De même l'algorithme de synthèse de LFSR de Berlekamp-Massey nécessite $4r^2$ opérations ou $O(r\log(r))$ avec un algorithme d'Euclide rapide arrêté au milieu. Enfin, l'application du registre à décalage linéaire nécessite $2(n-r)w(e) < 2nr$ opérations où encore $O(n\log(n))$ avec une multiplication rapide de polynômes. □

Exercice 4.27 (Correction des effacements). *Lors de la lecture d'un CD-ROM, un spot laser peut renvoyer 3 symboles : '0', '1', '?'. Le symbole '?' correspond à un relief non correct (dû par exemple à une poussière ou une rayure) ; on dit alors qu'il y a* effacement*. Plus généralement, la présence de '?' à la sortie d'un canal indique que le symbole correspondant est manquant. À la différence d'une erreur, un effacement donne la position du bit erroné.*

1. *Montrer qu'un code (n,k) quelconque de distance d permet de corriger $d-1$ effacements.*
2. *On considère un canal binaire sur lequel on veut pouvoir corriger jusqu'à 1% d'effacements (d'octet) en utilisant un code de Reed-Solomon sur $\mathbb{F}_{256}$. Préciser les caractéristiques (n,k,d) de ce code.*
3. *On suppose que le mot reçu comporte des effacements, mais pas d'erreur. Proposer une méthode de correction d'au plus $d-1$ effacements.*
4. *Soit C un code (n,k) de distance $d = 2t+s+1$. En déduire une méthode permettant de corriger simultanément t erreurs et s effacements.*

Solution page 325.

Les codes de Reed-Solomon sont très utilisés pour la correction des erreurs et des effacements dans les systèmes distribués de fichiers, comme les systèmes RAID ou encore les systèmes pair-à-pair ; dans ces systèmes, des ressources de stockage peuvent disparaître. L'utilisation de codes correcteurs permet alors de retrouver, grâce à la redondance, les blocs manquants.

Exemple : Le protocole PAR-2 de stockage d'un fichier sur un système pair-à-pair en est un exemple ; il est basé sur un code de Reed-Solomon sur $\mathbb{F}_{2^{16}}$ avec le polynôme primitif $X^{16}+X^{12}+X^3+X+1$.
Concrètement, PAR-2 permet de reconstruire un fichier lorsque des blocs sont manquants ; il ne corrige donc pas les erreurs mais uniquement des effacements. Le nombre de blocs qui peuvent être reconstruits est paramétrable et noté r ci-dessous.

Soit un fichier source à stocker de taille $2km$ octets ; PAR-2 partitionne ce fichier en k fichiers $D_1, \ldots, D_k$ de même taille égale à m mots de 16 bits. En complément, r fichiers redondants $C_1, \ldots, C_r$, contenant aussi m mots de 16 bits sont ajoutés. En outre, dans leur en-tête, les fichiers C_i contiennent des informations complémentaires, comme les résumés MD5 de tous les fichiers, leur taille, etc ; ces informations sont utilisées pour détecter les fichiers corrompus. Dans la suite, le mot formé par les $n = k + r$ chiffres de 16 bits en position i des fichiers $D_1, \ldots, D_k, C_1, \ldots, C_r$ est considéré comme un mot de code. Il est noté $d_1, \ldots, d_k, c_1, \ldots c_r$ de manière générique indépendamment de sa position i.
Les chiffres de redondance $c_1, \ldots, c_r$ sont calculés par un code de Reed-Solomon (n, k), raccourci du code de Reed-Solomon standard de longueur 65535 sur $\mathbb{F}_{16}$, généré par le polynôme $g(x) = \prod_{i=1}^{r}(X - \alpha^i)$; il est donc de distance $r + 1$.
Pour la lecture, on cherche à récupérer les n fichiers $D_1, \ldots D_k, C_1, \ldots, C_r$ mais on s'arrête dès que l'on en a récupéré k complets et non corrompus (vérification des hashs MD5 et de la taille). Pour reconstruire les d_i, on ne conserve que les k fichiers correspondants en poinçonnant les r autres positions. Soit alors y l'un des m mots reçus. Soit G' la sous-matrice de G construite en supprimant les colonnes des effacements. Alors, G' est de rang k (théorème 29) ; le système $x.G' = y$ admet donc une unique solution x qui est calculée par résolution du système linéaire en G' et qui est le mot corrigé.

4.5 Paquets d'erreurs et entrelacement

4.5.1 Paquets d'erreur

Sur la plupart des canaux, les erreurs ne se produisent pas de manière isolée (ils ne suivent pas une distribution aléatoire et uniforme) mais peuvent atteindre plusieurs symboles consécutifs ; on parle d'erreurs par *paquet*. C'est par exemple le cas des disques compacts (poussières, rayures) ou des liaisons satellites (perturbations électromagnétiques).
La *longueur* d'un paquet est sa taille en nombre de symboles. Par exemple, après émission de la séquence binaire ...10010110011..., la séquence reçue est ...10111011011... ; le mot d'erreur est alors ...00101101000... qui contient 4 erreurs isolées mais qui correspondent à un même paquet de longueur 6.
Le plus souvent, la longueur du paquet est aléatoire et même la longueur moyenne peut varier au cours du temps.

Exercice 4.28 (Code linéaire et détection de paquet d'erreurs). *Soit C un code linéaire (n, k) sur $\mathbb{F}_q$ qui détecte tout paquet de longueur au plus l. Montrer que C ne contient aucun mot qui est un paquet de longueur inférieure ou égale à l. En déduire que $l \leqslant n - k$.* *Solution page 325.*

Exercice 4.29 (Distance d'un code et longueur de paquet). *Sur un canal binaire, on utilise un code sur $\mathbb{F}_{256}$ par blocs $C(n,k)$ de distance d. Quelle est la longueur maximale l d'un paquet d'erreurs binaires corrigé par ce code ?* *Solution page 326.*

Les techniques d'entrelacement et d'entrelacement croisé permettent, à partir d'un code donné, avec un taux de correction donné ; d'augmenter la longueur des paquets d'erreurs corrigés.

4.5.2 Entrelacement

Plutôt que de transmettre les mots de code consécutivement, la technique d'entrelacement consiste à les entrelacer pour que deux chiffres consécutifs d'un même mot n'apparaissent pas consécutivement dans la séquence transmise.
Dans la suite, C est un code (n,k). Il existe différentes méthodes pour transmettre de manière entrelacée un message $M = [a, b, c, ...]$ qui est une séquence de taille quelconque de mots de code. Le principe est le suivant. Supposons que l'on veuille transmettre un message M formé de trois mots de code $[a, b, c]$ de longueur 5. La transmission sans entrelacement de M consiste à envoyer sur le canal la séquence $a_1a_2a_3a_4a_5b_1b_2b_3b_4b_5c_1c_2c_3c_4c_5$. Matriciellement, le message M s'écrit :

$$M = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \end{bmatrix}$$

et, sans entrelacement, cette matrice est transmise ligne par ligne. L'inconvénient est qu'un paquet d'erreurs de longueur 2 peut entraîner deux erreurs dans un même mot de code et donc ne pas être corrigé si le code est seulement 1-correcteur.
Avec entrelacement, la matrice est transmise colonne par colonne ; la séquence correspondante est donc $a_1b_1c_1a_2b_2c_2a_3b_3c_3a_4b_4c_4a_5b_5c_5$. Tout paquet d'erreurs de longueur au plus 3 ne peut alors pas entraîner plus d'une erreur sur un même mot ; grâce à l'entrelacement, il peut donc être corrigé même si le code initial est simplement 1-correcteur.
On généralise l'entrelacement ci-dessus en traitant le message par blocs de p mots ; un tel entrelacement est dit de *profondeur p*. Si le code initial corrige des paquets de longueur l, l'entrelacement de profondeur p permet de corriger des paquets de longueur $p.l$ (exercice 4.30).

Exercice 4.30 (Profondeur d'entrelacement et longueur de paquet corrigé). *Soit C un code (n,k) sur V de distance d qui corrige des paquets de longueur l. Soit p un entier et C_p le code (np, kp) sur V qui est l'ensemble de tous les entrelacements de p mots du code C. Le code C_p est dit* entrelacé

de C de profondeur p.
Montrer que C_p est de distance d mais permet de corriger tout paquet d'erreurs de longueur lp. *Solution page 326.*

4.5.3 Entrelacement avec retard et table d'entrelacement

Le codage étant généralement réalisé à la volée, on ne connaît pas la taille du message M transmis ; M est donc une suite de taille inconnue de mots d'un code (n, k). On réalise alors un *entrelacement avec retard* de M, à la volée. Cet entrelacement est décrit ci-dessous en utilisant la technique classique de *table d'entrelacement* ; la table sert en fait de tampon circulaire par ligne.

À envoyer

a_1	b_1								
		a_2	b_2						
				a_3	b_3				
						a_4	b_4		
								a_5	b_5

Tab. 4.5: Initialisation d'une table d'entrelacement de retard 2 pour des mots de longueur 5.

Une telle table est caractérisée par son retard r. Pour simplifier l'exposé, on suppose que la table a n lignes et $r.n$ colonnes. À l'initialisation, on stocke les r premiers mots (donc nr symboles) dans la table comme suit : le symbole en position i du mot j est stocké en ligne i et colonne $j + (i-1)r$. Autrement dit, les r premiers mots sont stockés dans les r premières colonnes ; puis, pour $i = 1, \ldots, n$, la ligne i est décalée à droite de $(i-1).r$ positions. Le résultat final est donné sur le tableau 4.5.
Après cette initialisation, pour transmettre les différents mots du message, on procède en itérant pour chaque bloc de p mots les trois opérations suivantes :

1. on envoie d'abord en séquence dans le canal les r premières colonnes de la table, l'une après l'autre ;
2. puis on décale circulairement les colonnes à gauche de r positions ; la colonne $r+1$ devient ainsi la première colonne et r nouvelles colonnes sont ajoutées à droite ;
3. enfin, on fait rentrer les r mots suivants du message dans la table de la même façon que pour les r mots précédents : le symbole en position i du mot j est encore stocké en ligne i et colonne $j + (i-1)r$.

La table 4.6 montre l'état de la table après un tour.

À envoyer

c_1	d_1								
a_2	b_2	c_2	d_2						
		a_3	b_3	c_3	d_3				
				a_4	b_4	c_4	d_4		
						a_5	b_5	c_5	d_5

TAB. 4.6: Table d'entrelacement de retard 2 après un tour.

On remarque qu'en régime permanent, chaque symbole transmis correspond à un symbole du message M. Mais ce n'est pas le cas au début et en fin de communication, puisque des cases non initialisées de la table sont transmises. Aussi, lors du décodage, on supprime en début et en fin de la séquence reçue, les positions qui ne correspondent pas à un symbole de source ; par exemple les positions de 2 à n si $r \geqslant 1$.
Grâce à cet entrelacement avec retard, deux caractères consécutifs d'un même mot de code sont distants d'au moins $rn + 1$ positions dans la séquence de symboles transmise. Si le code initial corrige des paquets de longueur l, l'entrelacement avec retard r garantit donc la correction de paquets de longueur $l(rn + 1)$

Remarque 5. *La longueur des paquets corrigés est donc limitée par le retard. Aussi cette technique ne permet-elle pas de corriger des paquets dont la longueur moyenne varie dans le temps. De plus, le choix d'un retard trop grand entraîne un surcoût proportionnel (envoi de caractères inutiles en début et fin de messages), inefficace pour les petits messages. Aussi d'autres techniques d'entrelacement ont été proposées, notamment pour les turbo-codes. Par exemple, les entrelaceurs de type* Golden *sont basés sur un entrelacement dont la profondeur varie en progression arithmétique de raison le nombre d'or.*

4.5.4 Code entrelacé croisé et code CIRC

Les codes entrelacés croisés sont basés sur la composition de deux codes, l'un permettant de détecter les paquets d'erreurs, l'autre de les corriger.
Plus précisément, soient $C_1(n_1, k_1)$ et $C_2(n_2, k_2)$ deux codes de distances respectives d_1 et d_2. Le principe de codage d'un message source M de m symboles est le suivant :

1. on code d'abord avec C_1 le message M, par blocs de k_1 symboles, pour obtenir un message M' de $m_1 = \left\lceil \frac{m}{k_1} \right\rceil$ mots donc $m_1 = n_1 . \left\lceil \frac{m}{k_1} \right\rceil$ symboles ;

2. les mots de C_1 sont entrelacés, par exemple avec un retard ; on obtient un message M'' que l'on considère de même taille m_1 (pour simplifier, on ne compte pas les symboles supplémentaires éventuellement ajoutés lors de l'entrelacement en début et fin de message) ;
3. les symboles de M'' sont alors codés avec C_2, par blocs de k_2 symboles, pour donner un message M''' de $m_2 = n_2 . \left\lceil \frac{m_1}{k_2} \right\rceil$ symboles.

Éventuellement, les mots de M''' peuvent à nouveau être entrelacés, mais ce cas n'est pas considéré dans la suite.
Pour le décodage, le code C_2 est utilisé pour détecter les erreurs, et le code C_1 pour les corriger. Aussi, la capacité de correction du code C_2 n'est pas complètement exploitée, l'accent étant mis sur sa capacité de détection. En pratique, on pose $d_2 = 2.t_2 + u_2 + 1$, avec t_2 assez petit. La capacité maximale de correction de C_2 serait $t_2 + \lfloor \frac{u_2}{2} \rfloor$, mais on n'utilise qu'une correction de t_2. Ainsi, si l'on détecte plus de t_2 erreurs, on ne les corrige pas avec C_2 car cela ferait perdre la capacité de corriger de grands paquets d'erreurs ; mais au contraire on les marque seulement afin de laisser le code C_1 s'en occuper. Plus rigoureusement, le principe de décodage est le suivant :

1. on décode d'abord chaque bloc M_i''' de n_2 symboles de M'''. Deux cas sont distingués. Soit le mot peut être décodé avec au plus t_2 corrections, auquel cas on le corrige et on récupère la séquence M_i'' source de k_2 symboles codée par C_2. Sinon, on détecte la présence de plus de t_2 erreurs, ce qui est le cas si le mot est affecté par un paquet d'erreurs ; dans ce cas, on marque la séquence M_i'' de k_2 symboles comme effacée.
2. on désentrelace les symboles obtenus pour reformer les mots M_i' codés par C_1 ;
3. du fait du désentrelacement, le mot M_i' peut contenir des symboles effacés, mais aucun symbole erroné. Par suite, la distance d_1 de C_1 permet de corriger jusqu'à $d_1 - 1$ symboles effacés dans M_i' pour reconstruire les k_1 symboles source M_i (voir l'exercice 4.27).

Application : code CIRC

Les codes de Reed-Solomon sont souvent utilisés comme codes de base d'un code entrelacé croisé ; on parle alors de code *CIRC* (*Cross-Interleaved Reed-Solomon Code*). Par exemple, le code utilisé pour coder les trames de son sur les disques compacts audio est un code CIRC. Une rayure d'un millimètre détruit environ 3300 bits sur la piste d'un CD ; le code permet de corriger jusqu'à 28224 bits consécutifs sur la piste, ce qui correspond à une longueur d'environ 8,2 mm sur le disque !

Sur un CD audio, l'information numérique source est codée par blocs appelés trames. Une trame correspond à 6 échantillons stéréo de 32 bits (16 bits à gauche et 16 bits à droite), soit 24 octets ou 192 bits de source auxquels il faut ajouter 8 octets de parité et un octet de contrôle. Chacun de ces octets est représenté physiquement par 17 bits : un mot de 14 bits (codage EFM pour *Eight-to-Fourteen Modulation*) suivi de 3 bits de liaison. De plus, 27 bits sont également ajoutés entre chaque trame. Finalement, 192 bits d'information numérique deviennent donc $(24+8+1)\times(14+3)+27 = 588$ bits sur la piste physique (plages lisses ou changements de relief détectés par le laser à l'entrée ou la sortie d'une cuvette gravée sur le CD), ce qui correspond à environ 0,17 mm sur la piste. Le signal numérique est converti en son analogique à une fréquence de 44100 Hz ; 1 s de musique est donc codée par $44100 \times 588/6 = 4321800$ bits sur la piste, soit environ 1,2 mètre.

Le codage CIRC utilisé est construit par entrelacement et croisement de deux codes $C_1(28, 24)$ et $C_2(32, 28)$ de distance 5. Ces deux codes sont de distance 5 sur $\mathbb{F}_{256}$, obtenus à partir de raccourcis d'un code (255, 251) de Reed-Solomon (cf exercice 4.31 page 264). Ils admettent respectivement comme matrices génératrices normalisées $G_1 = [I_{24}, R_1]$ et $G_2 = [I_{28}, R_2]$, où R_1 est une matrice 24×4 et $R_2 = \left[\frac{T}{R_1}\right]$ est une matrice 28×4 (i.e. T est une matrice 4×4).

Le codage est le suivant :

- chaque trame x de 24 octets est codée avec C_1 en un mot y de 28 octets : $y = [x, x.R_1]$. On obtient ainsi une séquence de mots de code de C_1 de 28 octets ;
- les mots de C_1 sont entrelacés avec retard de 4 octets dans une table ; après entrelacement on obtient une séquence de mots de 28 octets, chaque mot y' correspondant à une colonne de la table ;
- chaque mot y' est codé avec C_2 en un mot z de 32 octets : $z = [y', y'.R_2]$.

Ce codage est de rendement $\frac{3}{4}$.

Le décodage procède comme suit (cf exercice 4.32) :

- Chaque bloc z de 32 octets est décodé avec C_2 en un mot y' de 28 octets en essayant de corriger au plus une erreur. Si on détecte plus d'un chiffre erroné, on ne réalise aucune correction : on retourne l'information que le mot est effacé. Dans la suite on note '?' un symbole effacé (qui n'appartient donc pas à $\mathbb{F}_{256}$) ; on retourne donc par exemple le mot y' formé de 28 symboles '?'.
- les blocs y' de 28 symboles sont désentrelacés. On reconstruit ainsi une séquence de blocs y de 28 symboles. Dans un bloc y peuvent figurer des octets de $\mathbb{F}_{256}$ et des '?', chaque '?' correspondant à un effacement suite à une erreur détectée par C_2.
- Chaque bloc y de 28 symboles est décodé avec C_1. De distance 5, C_1 permet

de corriger jusqu'à quatre effacements dans le bloc y. Si on arrive à corriger ces effacements, on retourne les 24 premiers octets du mot corrigé ; sinon un message d'erreur.

Ce décodage permet de corriger jusqu'à 15 trames successives (exercice 4.33), soit $15 \times 32 \times 8 = 3840$ bits de signal ou encore $588 \times 15 = 8820$ bits ce qui correspond à des rayures de 2,5 mm environ.

Par ailleurs, il est possible d'extrapoler le son d'un échantillon manquant si l'on dispose des échantillons précédent et suivant. Aussi, à la sortie du code C_1, avant l'entrelacement avec retard, on effectue une permutation préalable. En sortie de C_1, les 28 octets correspondent à 14 mots de 16 bits : $g_1\ d_1\ g_2\ d_2\ g_3\ d_3\ g_4\ d_4\ g_5\ d_5\ g_6\ d_6\ p_1\ p_2$ où chaque mot g_i et d_i désigne les 16 bits gauche ou droite d'un échantillon stéréo sur 32 bits et p_1, p_2 les 4 octets de redondance. De façon à éloigner deux échantillons sonores de 16 bits consécutifs, on les permute en $g_1\ g_3\ g_5\ d_1\ d_3\ d_5\ p_1\ p_2\ g_2\ d_2\ g_4\ d_4\ g_6\ d_6$. Ainsi, grâce aussi à l'entrelacement avec retard, un paquet d'erreurs portant sur 48 trames de 32 octets peut empêcher de reconstruire un échantillon de 16 bits (g_3 par exemple) mais garantit de toujours disposer en sortie de C_1 de l'échantillon précédent et suivant (g_2 et g_4 pour g_3). Avec cette extrapolation, il est possible de corriger $48 \times 32 \times 8 = 28224$ bits d'information, soit $48 \times 588 = 28224$ bits ou environ 8,2 mm sur la piste.

Un autre entrelacement complémentaire est effectué en sortie de C_2 : on coupe les trames de 32 octets en bloc de 16 octets et on permute les blocs d'indice pair avec ceux d'indice impair. Avec cette permutation, deux erreurs consécutives affectent deux mots de code de C_2 distincts au lieu d'un seul. Comme C_2 corrige au plus une erreur par bloc, cela augmente sa probabilité de correction lors du décodage en évitant certains effacements.

Exercice 4.31 (Construction des codes C_1 et C_2). *Expliciter une construction possible des matrices génératrices G_1 et G_2 à partir d'un code de Reed-Solomon (255,251) sur $\mathbb{F}_{256}$. Justifier que les codes C_1 et C_2 sont de distance 5.* *Solution page 326.*

Exercice 4.32 (Décodage du code CIRC d'un CD audio). *On considère les codes $C_1(28, 24)$ et $C_2(32, 28)$ sur $\mathbb{F}_{256}$ du code CIRC pour les CDs audio. Le but de cet exercice est d'expliciter les étapes du décodage.*

1. *Lorsqu'on reçoit un bloc de 32 octets, expliciter comment effacer un bloc ou corriger au plus une erreur avec le code C_2 en utilisant le calcul du syndrome d'erreur.*
2. *Lors du décodage avec C_1, expliciter comment corriger au plus 4 effacements.*

Solution page 327.

Exercice 4.33 (Taux de correction du code CIRC). *Montrer que le code CIRC précédent permet de corriger 15 trames consécutives.*

Solution page 327.

La technique d'entrelacement série utilisée dans les codes CIRC permet de corriger des grands paquets d'erreurs en mélangeant les symboles provenant d'un nombre donné de blocs différents. Il est aussi possible de réaliser un entrelacement de taille non bornée en remettant en cause la notion de bloc ; les codes convolutifs en sont une illustration.

4.6 Codes convolutifs et turbo-codes

Les codes convolutifs ne fonctionnent plus comme des codes par blocs mais plutôt par flot : ils étendent les entrelacements précédents en codant chaque bit émis par une source par un ensemble de bits, dépendant d'opérations de convolution sur les derniers bits émis. Ils présentent l'avantage d'un décodage très simple et de bonnes performances.

4.6.1 Le codage par convolution

Dans son acceptation la plus générale, le codeur convolutif prend en entrée des blocs de k bits, applique des opérations aux m derniers blocs reçus, via une matrice de $k \times n$ polynômes de $\mathbb{F}_2[X]$ de degré maximum m, et code ainsi chaque bloc par un bloc de n bits.
Les paramètres du code sont donc la taille k des blocs d'entrée, la taille n des blocs de sortie, la mémoire m et les $k \times n$ polynômes.
Pour plus de simplicité dans la présentation, nous allons exposer le fonctionnement d'un code convolutif qui code un à un chaque bit du message source, soit $k = 1$. On dispose alors de n polynômes générateurs de degré maximum m.
La source est considérée comme une suite infinie de bits $(S_i)_{i \geqslant 0}$, et les bits d'indice négatifs sont tous considérés comme nuls pour marquer le début d'un message.
L'ensemble des n polynômes générateurs peut être écrit sous la forme d'une matrice G sur $\mathbb{F}_2$: $G = (g_{ij})$ est de taille $m \times n$.
Pour chaque bit de source d'indice l, on note $S(l)$ le vecteur ligne $S_l \dots S_{l-m}$.
Le bit S_l est alors codé par les n bits

$$C(l) = S(l)G \tag{4.8}$$

Chaque bit j du code correspond donc à la convolution $\sum_{i=0}^{m}(g_{ij}S_{l-i})$, d'où le nom de code convolutifs.

Exemple : Si les polynômes générateurs sont $P_1 = X$ et $P_2 = X + 1$, on a

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

Le message 100110100 sera codé dans ce cas par 110100111001110100 00.
La relation (4.8) donne, pour coder la source $(S_i)_{i \geqslant 0}$ la relation

$$C = SG_\infty$$

où G_∞ est la matrice infinie

$$\begin{pmatrix}
 & & & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\
 & G & & & & & 0 & 0 & 0 & \cdots \\
 & & & & G & & & & & \cdots \\
0 & 0 & 0 & & & & & G & & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \cdots \\
\vdots & \vdots & \vdots & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & 0 & 0 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots
\end{pmatrix}$$

Le code convolutif est donc un code linéaire, cyclique, sur une matrice infinie (pour un message fini de taille N, la matrice est de taille nN).
On peut aussi voir le code comme le produit du message en tant que liste des coefficients d'un polynôme par chacun des polynômes générateurs, suivi d'un entrelacement. On retrouve le principe du code par flot, où le codage d'un message bit par bit à la volée correspond à une opération linéaire sur tout le message.

Exercice 4.34. *Vérifier sur l'exemple précédent que le message codé peut être obtenu par des produits de polynômes.* *Solution page 327.*

4.6.2 Le décodage par plus court chemin

Les codes convolutifs sont des codes correcteurs d'erreurs. Il faut donc pour le décodage supposer que l'on reçoit un message R, qui est le message codé avec un certain nombre d'erreurs. L'algorithme de décodage retrouve le mot de code le plus proche du mot reçu au sens de la distance de Hamming, et donne le message source correspondant.
Le principe est similaire aux principes de programmation dynamique et de recherche de plus courts chemin. L'algorithme de décodage est connu sous le nom d'*algorithme de Viterbi*.

Le diagramme d'état

Le *diagramme d'état* d'un code convolutif est un graphe orienté ayant pour sommets les 2^m mots binaires de taille m, correspondant aux états possibles de l'entrée. On place alors pour tout mot $a_0 \dots a_{m-1}$ deux arcs correspondant aux deux valeurs possibles du bit suivant de l'entrée : l'un vers $a_1 \dots a_{m-1}0$ et l'autre vers $a_1 \dots a_{m-1}1$.
On étiquette alors chaque arc par le code produit (un mot de code de n bits) par l'entrée correspondante.
Tout mot de code correspond à une chaîne parcourant ce graphe, c'est-à-dire un ensemble d'arcs consécutifs, qui peuvent se répéter, et toute chaîne à un mot de code.

Exemple : (Suite de l'exemple précédent). Le diagramme d'état pour le code de polynômes générateurs $P_1 = X$ et $P_2 = X + 1$ est indiqué figure 4.2.

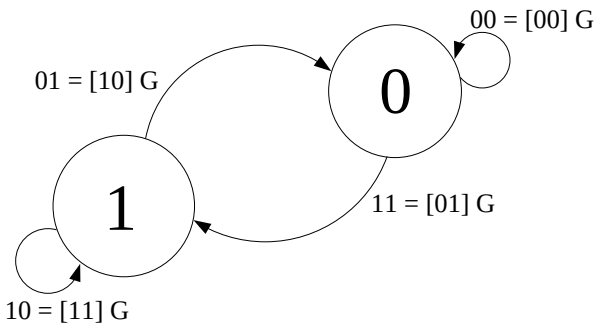


FIG. 4.2: Diagramme d'état du code convolutif de polynômes générateurs $P_1 = X$ et $P_2 = X + 1$.

Le treillis de codage

Le *treillis de codage* suit le principe du diagramme d'état en l'étendant à toutes les étapes d'un codage effectif. C'est-à-dire que le treillis contient pour chaque bit i reçu en entrée, un ensemble de 2^m sommets comme dans le diagramme d'états, et que chaque arc relie un mot $a_0 \dots a_{m-1}$ de l'étape i aux mots $a_1 \dots a_{m-1}0$ et $a_1 \dots a_{m-1}1$ de l'étape $i + 1$.
Tout mot de code correspond à un chemin dans ce graphe, et tout chemin à un mot de code.

Exemple : (Suite de l'exemple précédent). Le treillis du code de polynômes générateurs $P_1 = X$ et $P_2 = X + 1$ est indiqué figure 4.3.

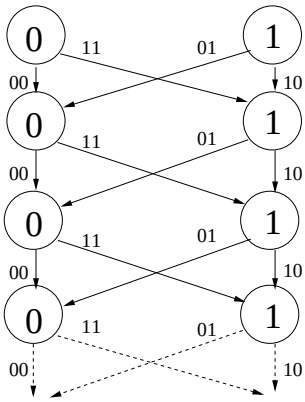


FIG. 4.3: Treillis du code convolutif généré par $P_1 = X$ et $P_2 = X + 1$.

L'algorithme de Viterbi

L'algorithme de Viterbi consiste à retrouver le chemin qui correspond au mot de code le plus proche du mot reçu. C'est un algorithme classique de recherche du plus court chemin dans un graphe orienté sans cycle, par programmation dynamique. Le poids sur les arcs correspond à la distance de Hamming entre les mots reçus et les mots de code correspondant à l'attribut des arcs.

Exercice 4.35. *Donner pour le code de polynômes générateurs* $P_1 = X$ *et* $P_2 = X + 1$ *le mot de code le plus proche de* 10010010011101111010*, et donner le message source associé.* *Solution page 328.*

L'algorithme 28 permet de retrouver le mot de code correct à partir de la totalité du mot de code reçu. En pratique, dans un code convolutif, il est plus intéressant de décoder à la volée, le code pouvant être infini (ou plutôt très grand en réalité). L'idée est qu'une erreur n'a un impact que sur un nombre limité de mots de code : ce nombre est appelé *longueur de contrainte* du code et est égal à km. Ainsi, pour décoder à la volée, il faut appliquer l'algorithme de Viterbi sur $T = km$ mots de code reçus. Il est alors possible de décider que le premier mot de source ne dépendra plus du dernier sommet exploré. On reprend ensuite l'algorithme de Viterbi avec le mot de code reçu suivant qui permet de s'assurer du deuxième mot de source, etc.
La question est alors d'évaluer la capacité de correction d'un code convolutif. On utilise également la notion de distance entre deux mots de code, mais comme les mots de code sont potentiellement infinis, on parle alors de *distance libre* du code. Comme le code est linéaire, la distance libre est le poids minimal d'une séquence non totalement nulle.

Algorithme 28 Algorithme de Viterbi (décodage des codes convolutifs).

Entrées Un mot binaire r de longueur T
Sorties Le mot de code c le plus proche de r selon la distance de Hamming

```
Donner une marque m(v) ← ∞ à tous les sommets v du treillis.
Donner une marque prec(v) ← ∅ à tous les sommets v du treillis.
Pour tous les sommets v à l'étape du début du message reçu, m(v) ← 0.
Pour un indice i parcourant de l'étape 1 à l'étape T Faire
  Pour tous les sommets v du treillis de l'étape i Faire
    Pour tous les arcs uv arrivant au sommet v Faire
      Soit d la distance de Hamming entre le mot de code reçu à l'étape i
      et l'attribut de l'arc uv
      Si d + m(u) < m(v) Alors
        m(v) ← d + m(u)
        prec(v) ← u
      Fin Si
    Fin Pour
  Fin Pour
Fin Pour
Soit v le sommet tel que m(v) est minimum parmi tous les sommets de la
dernière étape.
Pour un indice i de l'étape T à l'étape 0 Faire
  Soit uv l'arc tel que prec(v) = u
  Écrire à gauche du mot précédent le mot de code correspondant à l'attri-
  but de uv.
Fin Pour
```

Une séquence non totalement nulle est obtenue en partant de la séquence $000\ldots$, en passant par des caractères non nuls puis en revenant à une séquence de zéros. Autrement dit le poids minimal d'une séquence non totalement nulle est exactement le poids minimal d'un cycle de transitions partant et revenant en zéro. Pour l'exemple de la figure 4.3 on voit que le cycle le plus court ne reste en 1 qu'une seule fois, pour les arcs $11|01$. La distance libre du code est donc 3.

Comme un code convolutif est linéaire, le théorème 27 s'applique et la distance libre fournit donc le taux de correction d'erreur. Dans notre exemple, le code convolutif est donc 1-correcteur.

Cela veut dire que ce code convolutif peut corriger 1 erreur dans chaque bloc de mots de code de taille la longueur de contrainte.

Codes convolutifs systématiques, rendement et poinçonnage

Nous avons vu qu'un code convolutif fonctionne par flot, ou encore en *série*, en entrelaçant plusieurs bits. Ce fonctionnement induit donc un entrelacement que l'on appelle *entrelacement en série*.
Similairement aux codes par blocs, un code convolutif est dit *systématique* si les bits source apparaissent directement dans les bits de sortie. Autrement dit, la matrice G contient la matrice identité à une permutation des colonnes près. Au contraire, dans un code convolutif non-systématique ou NSC (*Non-Systematic Convolutional code*), tout bit de sortie est la combinaison de plusieurs bits d'entrée.
En sortie d'un codeur convolutif systématique, on a donc une séquence de bits qui est l'alternance de deux séquences : d'une part les bits de l'entrée X, d'autre part les bits Y calculés par le codeur. Le rendement d'un code convolutif est a priori bien moins bon que le rendement d'un code par blocs. Pour augmenter le rendement du code tout en gardant un code systématique qui permet de décoder l'entrée, la *perforation* ou *poinçonnage* consiste à supprimer une partie des bits Y. Par exemple, supposons que le code est de rendement $\frac{1}{3}$; pour chaque bit X_i en entrée, on a alors 3 bits en sortie donc une séquence de la forme $\ldots, X_i, Y_i, Y'_i, X_{i+1}, Y_{i+1}, Y'_{i+1}, \ldots$. Par poinçonnage des bits en positions 2 et 5 modulo 6, on obtient la séquence $\ldots, X_i, Y'_i, X_{i+1}, Y_{i+1}, \ldots$. Le code obtenu est un code convolutif systématique de rendement $\frac{1}{2}$ qui fonctionne comme un code poinçonné.

Codes convolutifs récursifs

Un code convolutif est dit *récursif* si certains bits de sortie du codeur sont réinjectés en entrée. Un code convolutif récursif systématique ou *RSC* peut être construit à partir d'un code systématique en réinjectant dans l'entrée l'une des sorties (figure 4.4, où D est un registre à décalage ou LFSR).

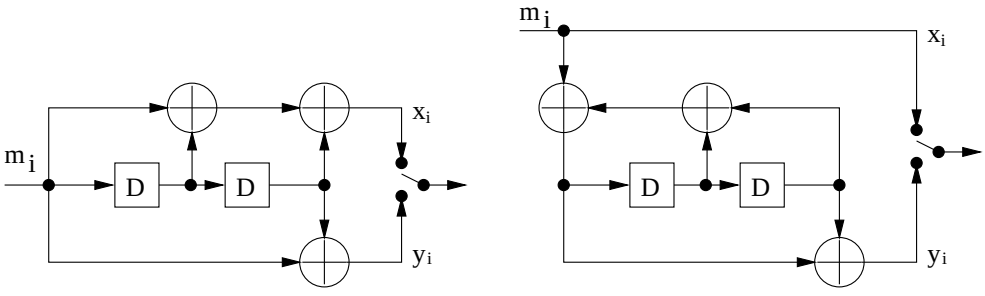


FIG. 4.4: Transformation d'un code convolutif systématique de longueur de contrainte 3 en code systématique récursif RSC.

Les codes convolutifs systématiques récursifs (RSC) sont à la base de la construction des turbo-codes.

4.6.3 Les turbo-codes

Les *turbo-codes* utilisent la composition avec entrelacement de deux codes. Il s'agit souvent de codes systématiques récursifs (RSC) dans lesquels des bits de sortie sont réinjectés en entrée du codeur ; le décodage tire parti de cette propriété, d'où le nom *turbo*.
Le fait de réinjecter des bits de sortie dans les bits d'entrée est un procédé itératif et la correction se fait alors par tour, chaque tour corrigeant de plus en plus d'erreurs. Plus précisément, seules les corrections portant sur peu de bits sont effectuées à chaque tour, le mot corrigé étant alors réinjecté par le turbo. À partir de ce dernier une nouvelle passe de correction est initiée. Comme le mot a déjà été corrigé, les anciennes erreurs peuvent dorénavant porter sur moins de bits et être mieux corrigées. Au niveau théorique, le nombre garanti d'erreurs corrigées dans le pire des cas n'est pas amélioré par rapport à un code par blocs. Au niveau pratique cependant, le nombre d'erreurs mal corrigées ou même non détectées est drastiquement réduit. En outre, si des mots très particuliers peuvent toujours échapper à la correction, le système d'entrelacement permet de répartir les erreurs de manière pseudo-aléatoire, uniformément dans les mots de code. Ainsi, le comportement pratique *en moyenne* des turbo-codes sur des entrées quelconques est très bon. Nous allons maintenant voir plus en détail la technique turbo.

Composition parallèle

À la différence des codes entrelacés croisés qui utilisent un entrelacement série de deux codes, les turbo-codes utilisent un entrelacement généralement parallèle de deux codeurs C et C' (figure 4.5). Le code C est toujours convolutif systématique récursif (RSC), et généralement le code C' aussi. D'ailleurs, en pratique, on a souvent $C' = C$.
Le code C (resp. C') étant systématique, sa sortie est une séquence alternée de deux séquences : d'une part les bits X de la source et d'autre part les bits Y (resp. Y') calculés par le codeur convolutif. Un codeur turbo est réalisé comme suit :
– la source X est mise en entrée du codeur C dont la sortie systématique est aussi la sortie systématique du turbo-codeur ;
– la source X est entrelacée avant d'être mise en entrée du codeur C' dont la sortie est Y' ;
– les sorties Y et Y' sont utilisées pour construire la sortie Z du turbo-code, dans laquelle sont alternés les bits de la source X.

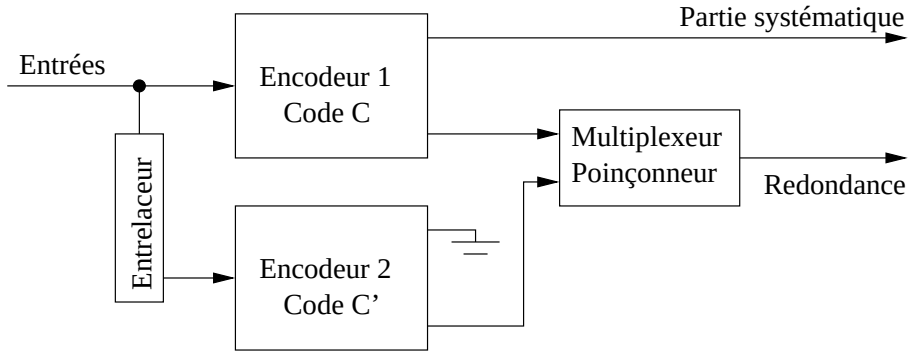


FIG. 4.5: Codeur turbo par composition parallèle et entrelacement de deux codeurs RSC.

Avec des codeurs C et C' de rendement $\frac{1}{2}$, on obtient un turbo-code de rendement $\frac{1}{3}$: à chaque top, trois bits sont générés en sortie $Z = \ldots, X_i, Y_i, Y'_i, \ldots$. Il est possible d'augmenter le rendement en multiplexant les sorties Y et Y' plutôt que de les concaténer, ce qui correspond à un poinçonnage. Par exemple, en multiplexant alternativement Y_i et Y'_i, la sortie devient $Z = \ldots, X_i, Y_i, X_{i+1}, Y'_{i+1}, X_{i+2}, Y_{i+2}, \ldots$; on obtient un turbo-code de rendement $\frac{1}{2}$.

Exercice 4.36 (Turbo-code parallèle et turbo-code série). *Soit C_1 et C_2 deux codes RSC de rendements respectifs R_1 et R_2.*

1. *Quel est le rendement du turbo-code obtenu par composition parallèle de ces deux codes ?*
2. *On peut aussi construire des turbo-codes séries en entrelaçant la sortie de C_1 et en la mettant en entrée de C_2, similairement au codage entrelacé croisé. La sortie du turbo-code série est alors la sortie de C_2. Quel est le rendement du turbo-code série ?*
3. *Comparer les rendements des turbo-codes parallèles et séries lorsque les rendements sont identiques $R_1 = R_2 = r$. Étudier le cas $r = \frac{1}{2}$.*

Solution page 328.

Décodage turbo

D'un point de vue pratique, il est difficile de décoder un turbo-code par calcul de maximum de vraisemblance (algorithme de Viterbi) : à cause de l'entrelacement, la complexité de cet algorithme est prohibitive, en $O(2^L)$ où L est la taille d'une trame et aussi la profondeur de l'entrelacement. Aussi, le décodage d'un turbo code utilise les décodeurs de chacun des deux codes C et

C' le constituant, en utilisant une technique itérative : la sortie de chaque décodeur est réinjectée en entrée de l'autre pour calculer la distribution de probabilités des bits d'entrée. De manière analogue à un moteur turbo, les gaz d'échappement sont réinjectés pour améliorer les performances.
Pour le décodage, les opérations de démultiplexage et de désentrelacement permettent de reconstruire les séquences $\tilde{X}$, $\tilde{Y}$, $\tilde{Y}'$ correspondant respectivement à la sortie systématique X – donc l'entrée –, et aux sorties Y et Y' des deux codeurs, mais éventuellement erronées.

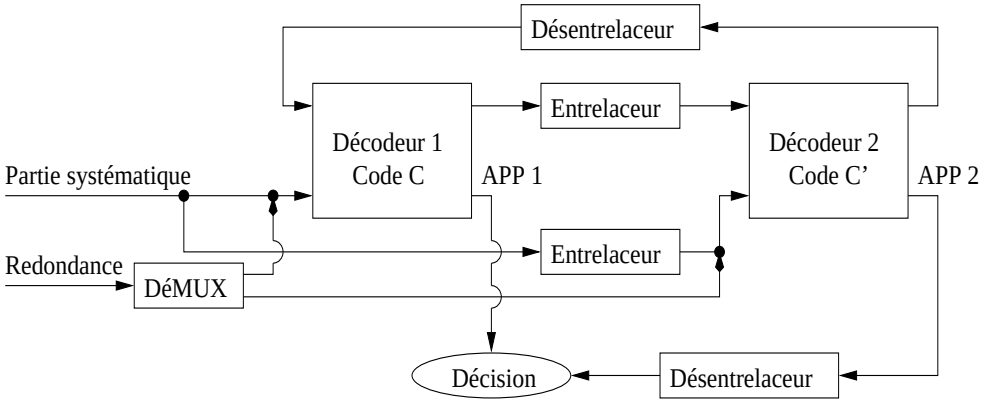


FIG. 4.6: Décodage itératif d'un turbo-code.

La correction est basée sur un mécanisme itératif et utilise les deux décodeurs associés aux codes C et C' (figure 4.6). Chacun des deux décodeurs calcule de son côté la probabilité *a posteriori* (APP) de chaque bit de source. Cette probabilité est utilisée par l'autre décodeur comme une information *a priori* sur la valeur du bit de source. Ainsi, le décodeur de C (resp. C') calcule la probabilité APP à partir de la valeur $\tilde{X}$ de l'entrée systématique du décodeur, la probabilité APP de la source calculée par le décodeur C' (resp. C) et l'entrée $\tilde{Y}$ du décodeur C (resp. $\tilde{Y}'$ de C'). Ainsi, chaque décodeur se base *a priori* sur une distribution de probabilités et non sur une valeur pré-décidée. Pour chaque décodage de bit, cette étape de réinjection/calcul est itérée un nombre fixé de fois, souvent 2. La valeur décidée pour le bit de source en fin d'itération est celle qui a la plus grande probabilité.

Turbo-codes en blocs et turbo-codes hybrides

Il est possible de remplacer les encodeurs convolutifs d'un turbo-code convolutif par des encodeurs par blocs. Si les deux encodeurs sont des codes par bloc (par exemple deux Reed-Solomon) on parle de *turbo-code en blocs*, si un

seul des deux encodeurs est un code par bloc et l'autre de type convolutif, on parle de *turbo-code hybride*. L'intérêt de remplacer des codes convolutifs par des codes en blocs est que le rendement est amélioré. Cependant il faut rendre le code en bloc probabiliste afin de tirer parti de l'effet turbo. Pour cela, au lieu de corriger automatiquement vers le mot de code le plus proche, on modifie l'algorithme de correction pour qu'il marque plutôt tout un ensemble de mots de code proches avec un certain poids (et une probabilité d'être le mot de code recherché). La décision finale se fait ensuite par le raffinement de ces poids au fur et à mesure des itérations turbo.

Performances et applications des turbo-codes

En raison de leurs performances, les turbo-codes ont été intégrés dans de nombreux standards et en particulier pour les communications spatiales. L'ESA (*European Space Agency*) a été la première à les utiliser pour la sonde lunaire Smart-1 (lancement le 27 septembre 2003, orbite autour de la Lune le 15 novembre 2004 et alunissage en septembre 2006).
La NASA les utilise depuis 2003 pour toutes ses sondes spatiales.
Les turbo-codes sont utilisés de manière intensive pour les communications par flot en général : ADSL-2, la téléphonie mobile (UMTS, 3G) ou encore le nouveau standard DVB-S2 de télévision haute définition (HDTV) normalisé en mars 2005.
Parmi les paramètres qui influencent la performance d'un turbo-code, la taille de l'entrelaceur joue un rôle critique ; les performances se dégradent lorsque cette taille diminue et inversement s'améliorent lorsqu'elle augmente. Il y a donc un compromis prix/performance à optimiser selon l'application. Généralement, la taille L varie de 2^{10} à 2^{16}. De plus, l'entrelaceur est souvent construit de manière aléatoire, en évitant les entrelacements de type bloc. Comme pour les autres codes, le rendement joue aussi un rôle important. Pour obtenir un code de rendement supérieur à $\frac{1}{3}$, un poinçonnage est nécessaire qui dégrade les performances de correction. Enfin, le nombre d'itérations à effectuer lors du décodage augmente avec la taille de l'entrelacement, typiquement environ 9 itérations pour un turbo-code parallèle avec un entrelacement de taille $L = 16384$ est un bon compromis performances/complexité.
La longueur de contrainte a moins d'influence que pour les codes convolutifs ; aussi, les turbo-codes utilisés en pratique ont une petite longueur de contrainte, typiquement 3 ou 5.

Compression, cryptage, correction : en guise de conclusion

Le codage des messages dans un alphabet lisible par un canal va optimiser la taille des données transmises, pallier aux erreurs éventuelles dues à la transmission, et assurer la confidentialité des données. Cette section est une petite conclusion pratique au codage dans son ensemble.

Soit M le message émis, sous forme d'une chaîne sur un alphabet V, ou, de façon équivalente, d'un ensemble de nombres sur V. On doit pouvoir reconstituer M à partir du message reçu (ou éventuellement, si le message reconstitué est différent, il doit être accompagné d'un message d'alerte). Supposons que l'émetteur dispose d'un utilitaire de compression $COMP$, de cryptage $CRYPT$, et de correction d'erreurs $CORR$. Le destinataire dispose des utilitaires $CORR^{-1}$, $CRYPT^{-1}$ et $COMP^{-1}$, qui lui permettront de reconstituer le message. Quelle est alors la procédure de codage ?

Le message doit être compressé avant d'être crypté. En effet, s'il contient de la redondance, elle ne sera peut-être pas éliminée par le cryptage, et la présence de redondance dans un message intercepté est un angle d'attaque pour reconstituer la méthode de cryptage (la méthode de compression n'est pas secrète). Pour la même raison, il doit être crypté avant de subir les transformations nécessaires à son intégrité (détection et correction d'erreur). En effet $CORR$ va rajouter de la redondance, organisée et optimisée, qui ne doit pas être un point faible du secret. D'autre part, la préparation à la correction d'erreur est forcément la dernière opération de l'émetteur, car dans les autres cas, une altération sur un message n'est plus corrigée dès sa réception, et peut donc être amplifiée par les premières opérations en sortie de canal (une décompression par exemple). Enfin, le fait de compresser avant tout permet de réduire le temps de calcul des deux autres fonctions. Il est donc indispensable d'effectuer les opérations dans cet ordre.

L'émetteur va donc générer le message

$$CORR(CRYPT(COMP(M)))$$

et l'envoyer à son destinataire. Le message reçu est M' correspondant donc à $CORR(CRYPT(COMP(M)))$, éventuellement altéré au cours de la transmission.

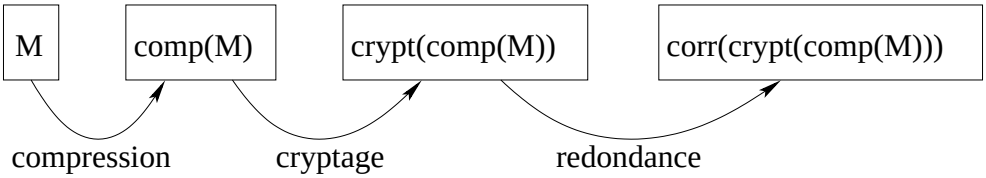


FIG. 69: Codage $CORR(CRYPT(COMP(M)))$ d'un message M.

Le destinataire applique

$$COMP^{-1}(CRYPT^{-1}(CORR^{-1}(M'))) = M.$$

En cas d'altération non corrigée, la détection peut permettre un renvoi automatique du message.

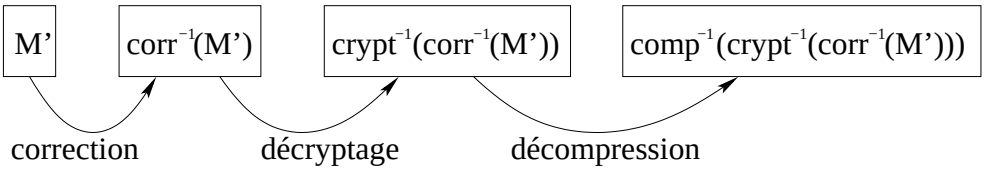


FIG. 70: Décodage d'un message $M' = CORR(CRYPT(COMP(M)))$.

Le message $CORR(CRYPT(COMP(M)))$ ne contient aucune information qui permette de reconstituer M sans disposer des bonnes fonctions, il doit contenir des informations qui permettent de vérifier sa cohérence, et éventuellement de s'auto-corriger, et enfin être le plus petit possible, pour minimiser la durée et le coût de la transmission.
Ce sont toutes ces fonctions que nous avons étudiées séparément dans les chapitres précédents. Le chapitre premier donne sa cohérence au tout, en fournissant une méthode globale d'utilisation des différentes parties, et en présentant un cadre théorique commun.

Exercice C.1 (Le casino). *Dans un casino, on joue au 421. Le casino possède un serveur centralisé qui enregistre tous les lancers de dés effectués sur chaque table de jeu. À chaque table, un croupier transmet -par infrarouge- les séquences de dés au serveur. Le problème est de mettre en place une architecture permettant de réaliser la transmission fiable et sécurisée des informations.*

1. *Construction d'un code de Reed-Solomon adapté.*
   La liaison infrarouge est modélisée par un canal binaire symétrique de probabilité d'erreur 0.001. On désire ici assurer des transmissions fiables sur ce canal.
   (a) *Quelle est la probabilité p d'erreur lorsqu'on envoie un octet ?*
   (b) *Pour remédier aux erreurs dues au canal, lorsqu'on envoie n octets, on veut garantir de corriger jusqu'à $p \times n$ erreurs. Expliquer comment construire un code correcteur de type Reed-Solomon en précisant :*
      i. *le corps de base et la valeur choisie pour n ;*
      ii. *le degré du polynôme générateur et le rendement du code.*
      iii. *le nombre maximal d'erreurs détectées ;*
      iv. *la dimension d'une matrice génératrice du code. Comment écrire cette matrice à partir des coefficients du polynôme générateur ?*
   (c) *Pour $d = 3, .., 10$, les polynômes suivants, à coefficients dans $\mathbb{F}_2$, sont primitifs :*

degré d	Polynôme primitif
3	$1 + \alpha + \alpha^3$
4	$1 + \alpha + \alpha^4$
5	$1 + \alpha^2 + \alpha^5$
6	$1 + \alpha + \alpha^6$
7	$1 + \alpha^3 + \alpha^7$
8	$1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^8$
9	$1 + \alpha^4 + \alpha^9$
10	$1 + \alpha^3 + \alpha^{10}$

TAB. 4.7: Quelques polynômes primitifs de $\mathbb{F}_2$.

      i. *Donner le polynôme utilisé pour implémenter le corps de base et expliquer brièvement comment réaliser les opérations d'addition et de multiplication ;*
      ii. *Donner l'expression du polynôme générateur en fonction de α.*

(d) *Calculer la capacité du canal. Comparer au rendement du code proposé.*

2. *Sécurisation des communications.*
   Comment coder les séquences de dés pour garantir qu'il n'y ait pas de trucages sans consentement d'un croupier ?

3. *Codage des lancers.*
   On suppose les dés du casino non pipés. On cherche à coder les séquences de tirages sur le canal binaire :

   (a) *Quelle est l'entropie d'un dé ?*

   (b) *Proposer un algorithme de codage où les mots de code sont de même longueur.*

   (c) *Calculer la longueur moyenne de ce codage.*

   (d) *Proposer un codage de Huffman.*

   (e) *Ce codage est-il optimal ? Si non, proposer un codage plus performant.*

Solution page 329.

Solution des exercices

Solutions des exercices proposés au chapitre 1

Exercice 1.1, page 21.
- Dans ce cas, le code n'est pas efficace. Le résultat peut même être plus long que si on avait codé simplement les pixels noirs par des 1 et les pixels blancs par des 0, par exemple lorsque l'image consiste entièrement en une suite de "01". Ce n'est sûrement pas très courant pour les fax, mais on apprend ainsi que ce code n'a aucune garantie théorique !
- Pour l'améliorer, on peut par exemple coder un nombre k de "01" consécutifs par "k01", et ainsi étendre les possibilités du code. Le chapitre 2 donne des clefs pour systématiser ce type de principe.

Exercice 1.2, page 23.
Essayer toutes les clefs est une méthode raisonnable, car l'ensemble des clefs possible est limité. On verra dans la section 1.1.7 quelle taille doit avoir l'espace des clefs pour garantir l'impossibilité d'une telle méthode.

Exercice 1.3, page 25.
CÉSAR AURAIT ÉTÉ CONTENT DE VOUS !

Exercice 1.4, page 28.
Le code a une complexité $O(n)$, où n est le nombre de pixels nécessaires pour coder l'image. En effet, l'algorithme consiste en un parcours de tous les pixels, avec un nombre d'opérations constant pour traiter chaque pixel.

Exercice 1.5, page 29.
Les raisons sont liées à deux attaques possibles :

1. En supposant qu'un seul message en clair et son équivalent crypté aient été interceptés, la clef est facilement obtenue par $K = \tilde{M} \oplus M$.
2. En outre, Si $C_1 = M_1 \oplus K$ et $C_2 = M_2 \oplus K$ alors $C_1 \oplus C_2 = M_1 \oplus M_2$ dans lequel la clef n'apparaît plus. Une analyse pourrait alors déterminer des morceaux de M_1 ou M_2.

Exercice 1.6, page 30.
Si on n'a pas confiance dans la machine qu'on utilise, qui peut capturer un mot de passe s'il est tapé en clair, il faut taper ce mot de passe crypté. Il faut alors convenir avec le serveur sécurisé d'une liste de clefs jetables (chacune à 8 caractères par exemple si le mot de passe a 8 caractères) conservée sur le serveur et emportée imprimée sur un papier par l'utilisateur.
À chaque utilisation, il faut alors crypter à la main son mot de passe avec la prochaine clef non encore utilisée, et envoyer le mot de passe crypté au serveur sécurisé, qui déchiffrera et identifiera l'utilisateur. Ainsi le mot de passe ne circule jamais en clair.

Exercice 1.7, page 32.

1. $P(C=1)=1/8$; $P(C=2)=7/16$; $P(C=3)=1/4$; $P(C=4)=3/16$.
2. $P(M=a|C=1)=1$; $P(M=a|C=2)=1/7$; $P(M=a|C=3)=1/4$; $P(M=a|C=4)=0$ $P(M=b|C=1)=0$; $P(M=b|C=2)=6/7$; $P(M=b|C=3)=3/4$; $P(M=b|C=4)=1$
3. On voit que la connaissance du chiffré donne une information sur le message envoyé alors qu'un chiffrement parfait ne devrait donner aucune information.

Exercice 1.8, page 33.
On trouve une entropie de 0,14 (à comparer avec l'entropie de la source équiprobable 0,3. Le 0 sera très courant dans le message, on peut s'attendre à de longues suites de 0. On peut donc choisir de coder efficacement ces longues suites.

Exercice 1.9, page 36.

1. Comme les messages M et les clefs K sont indépendants, on a directement $H(M,K)=H(M)+H(K)$ ou encore $H(M|K)=H(M)$ et $H(K|M)=H(K)$.
2. Ensuite, par sa construction avec les xor, dans un code de Vernam on a : $H(M,K,C)=H(M,C)=H(M,K)=H(C,K)$. En effet, en connaissant M et C on peut récupérer la clef par $K=M\oplus C$, de même connaissant M et K, C n'ajoute aucune information, etc. Or, pour tout X,Y, $H(X,Y)=H(Y)+H(X|Y)=H(X)+H(Y|X)$. Donc on tire directement $H(C|K)=H(M|K)$ ou encore $H(K|C)=H(M|C)$.
3. Si la source de clefs K est totalement aléatoire, elle est donc d'entropie maximale sur les mots de même taille par la propriété 1. Donc $H(K)\geqslant H(C)$, ce qui prouve que $H(K|C)\geqslant H(C|K)$. En combinant les deux

résultats précédents, on obtient donc $H(M|C) \geqslant H(C|K) = H(M|K) = H(M)$. Ce qui prouve que forcément $H(M|C) = H(M)$ et que donc la connaissance du chiffré n'apporte aucune information sur le message.

Exercice 1.10, page 41.

ECB et CTR : les blocs sont indépendants.

OFB : Seul C_1 est faux, donc $M_1 = C_1 + Z_1$ est faux, mais cela n'influe pas $M_2 = C_2 + Z_2$.

CFB : $M_2 = C_2 + E(C_1)$, donc M_2 est faux car C_1 est faux, mais $M_3 = C_3 + E(C_2)$ ne dépend que de C_2 et C_3 qui sont corrects.

CBC : $M_2 = C_1 + D(C_2)$, donc M_2 est faux car C_1 est faux, et $M_3 = C_2 + D(C_3)$ ne dépend que de C_2 et C_3 qui sont corrects.

Exercice 1.11, page 48.

Appliquer l'algorithme d'Euclide étendu :

$$\begin{array}{lrcrcl} (E_0): & 1\times a & + & 0\times b & = & a \\ (E_1): & 0\times a & + & 1\times b & = & b \\ (E_{i+1}) = (E_{i-1}) - q_i(E_i): & u_i\times a & + & v_i\times b & = & r_i \end{array}$$

On ne détaille que le premier couple $(a, b) = (17, 50)$:

(E_0) :	1×50	+	0×17	=	50	
(E_1) :	0×50	+	1×17	=	17	$q_1 = 50/17 = 2$ $r_1 = 50 \bmod 17 = 16$
$(E_2) =$ $(E_0) - 2(E_1)$	1×50	+	$(-2)\times 17$	=	16	$q_2 = 17/16 = 1$ $r_2 = 17 \bmod 16 = 1$
$(E_3) =$ $(E_1) - (E_2)$	$(-1)\times 50$	+	3×17	=	1	$q_3 = 16/1 = 16$ $r_3 = 16 \bmod 1 = 0$

Pour les deux autres couples on obtient : $1 = 51 \times 11 + (-2) \times 280$ et $5 = 3 \times 35 + (-2) \times 50$

Exercice 1.12, page 49.

1. Comme 17 est premier avec 50, 17 est inversible modulo 50 :

   $$x = 17^{-1}.10 = 3.10 = 30 \mod 50$$

   Donc $S = \{30 + 50.k, k \in \mathbb{Z}\}$

   NB : Calcul de l'inverse par Bézout : cf exercice précédent

2. Il existe $k \in \mathbb{Z}$ tel que $35x = 10 + 50.k \iff 7x = 2 + 10k \iff 7x = 2 \mod 10$.

   Comme 7 est premier avec 10, 7 est inversible modulo 10 :

   $$x = 7^{-1}.2 = 3.2 = 6 \mod 10$$

   Donc $S = \{6 + 10.k, k \in \mathbb{Z}\}$.

3. pgcd$(35, 50) = 5$ et 5 ne divise pas 11 donc $S = \emptyset$

Exercice 1.13, page 49.
On utilise la classe `mpz_class` de la version C++ de la bibliothèque GMP[2] qui implémente des entiers à précision arbitraire.

```
#include <gmpxx.h>
typedef mpz_class Entier;

void AEE(const Entier a, const Entier b,
        Entier& d, Entier& x, Entier& y) {
   if ( b==0 )  {
       d = a; x = 1; y = 1 ;
   }
   else {
       AEE( b, a % b, d, x, y ) ;
       Entier tmp = x ;
       x = y ;
       y = tmp - (a / b) * y ;
   }
}
```

Exercice 1.14, page 51.
$x = 4.11.11^{-1} \mod 5 + 5.5.5^{-1} \mod 11 \mod 55 = 44 - 25.2 \mod 55 = -6 \mod 55 = 49 \mod 55$. Donc $y = 49^{-1} \mod 55$ se projette en $4^{-1} \mod 5 = 4 \mod 5$ et en $5^{-1} \mod 11 = 9 \mod 11$. D'où l'on tire la reconstruction

$$y = \left(4.11.11^{-1} \mod 5 + 9.5.5^{-1} \mod 11\right) \mod 55.$$

Soit, $y = 44 - 45.2 \mod 55 = 46 \mod 55 = 9 \mod 55$.

Exercice 1.15, page 51.
On ne considère dans un premier temps que le système (3,4,5) ; on a alors $x = -1 \mod 3$,4 et 5. Donc $x = -1 \mod 60$. On a donc $x = 60.k - 1$.
On vérifie alors les 2 autres contraintes modulo 2 et 6 ; on a $(60k-1) \mod 2 = -1 \mod 2 = 1$ et $(60k - 1) \mod 6 = -1 \mod 6 = 5$. Toutes les contraintes sont vérifiées donc $x = 60k - 1$.

Exercice 1.16, page 51.

1. On utilise le fait que si un nombre divise un produit, alors il divise l'un des deux termes de ce produit. En conséquence, les seuls diviseurs de p^k sont les puissances de p inférieures à k, soient $p, \ldots, p^{k-1}$. On en conclut que $\varphi(p^k) = p^k - p^{k-1} = (p-1)p^{k-1}$.

[2] *Gnu Multiprecision package*, `www.swox.com/gmp`

2. On utilise une bijection entre les ensembles $\mathbb{Z}_m \times \mathbb{Z}_n$ et $\mathbb{Z}_{mn}$: si $x \in \mathbb{Z}mn$, on lui associe les deux nombres $a = x \mod m$ et $b = x \mod n$. Inversement, par le théorème des restes chinois, puisque m et n sont premiers entre eux, si $(a, b) \in \mathbb{Z}_m \times \mathbb{Z}_n$, alors il existe un unique $x \leqslant mn$ tel que $x = a \mod m$ et $x = b \mod n$. Puisque x est premier avec mn si et seulement si a est premier avec m et b est premier avec n, alors les cardinaux de $\mathbb{Z}_m^* \times \mathbb{Z}_n^*$ et $\mathbb{Z}_{mn}^*$ sont égaux.
3. On déduit des questions précédentes que si on a la décomposition $n = \prod p_i^{k_i}$ d'un nombre en produit de facteurs premiers, alors $\varphi(n) = \prod (p_i - 1)p_i^{k_i - 1}$.

Exercice 1.17, page 53.

1. On rappelle que dans un groupe fini commutatif $(G, \times, e)$ de cardinal c, on a : pour tout $x \in G$: $x^c = e$ (propriété 4). Or le cardinal du groupe des inversibles modulo n est $\varphi(n)$.
   Après factorisation de n, on calcule $\varphi(n)$. Il suffit ensuite de calculer $x^{-1} = x^{\varphi(n)-1} mod n$ par élévation récursive au carré.
   – $63 = 3^2 \times 7$: $\varphi(63) = 3 \times 2 \times 6 = 36$. D'où $22^{-1} \mod 63 = 22^{35} \mod 63 = 43 \mod 63$.
   – $\varphi(24) = 2 \times 4 = 8$; donc $x^{8 \times u} = 1 \mod 24$. D'où $5^{2001} = 5 \mod 24$.
2. On pose $n_i = p_i^{\delta_i}$; les n_i sont premiers 2 à 2.
   – Algorithme 1 : par l'algorithme d'Euclide étendu.
   – Algorithme 2 : par le théorème d'Euler, par élévation à la puissance modulaire. D'après Euler, $y^{\varphi(n)} = 1[n]$. D'où, après calcul de $\varphi(n)$, $y^{-1}[n] = y^{\varphi(n)-1}[n]$.
     NB : la factorisation de N permet de calculer $\varphi(N)$.
   – Algorithme 3 : par le théorème chinois, en utilisant le théorème d'Euler pour chaque nombre premier.
     Soit $N = \prod p_i^{\delta_i}$ décomposition de N en facteurs premiers p_i, et soit $n_i = p_i^{\delta_i}$. On peut d'abord pour tout i calculer l'inverse de $y \mod n_i$ par Lagrange : $y_i^{-1} = y_i^{\varphi(n_i)-1} \mod n_i$ avec $\varphi(n_i) = n_i \left(1 - \frac{1}{p_i}\right)$. On calcule donc les résidus de l'inverse modulo les n_i et on remonte l'inverse par le théorème chinois. Soit $N_i = N/n_i$. D'après le théorème chinois,

$$y^{-1}[N] = \left(\sum_{i=1}^{k} y^{-1 \mod n_i} . N_i . N_i^{-1 \mod n_i}\right) \mod N$$

     Par exemple, soit y l'inverse de 49 mod 55. Alors $y = 4^{-1[5]} = -1[5]$ et $y = 5^{-1[11]} = -2[11]$ d'où $y = 9[55]$.

Exercice 1.18, page 63.
On a tout d'abord $\Psi(0) = 0_K, \Psi(1) = 1_K, \Psi(n_1 + n_2) = \Psi(n_1) + \Psi(n_2)$. Ensuite, comme la multiplication d'éléments de K est associative, on obtient

$$\begin{aligned}\Psi(n_1) \times \Psi(n_2) &= \underbrace{(1_K + 1_K + \ldots + 1_K)}_{n_1} \times \underbrace{(1_K + 1_K + \ldots + 1_K)}_{n_2} \\ &= \underbrace{(1_K \times 1_K + \ldots + 1_K \times 1_K)}_{n_1 n_2} = \underbrace{(1_K + \ldots + 1_K)}_{n_1 n_2} = \Psi(n_1 n_2)\end{aligned}$$

et donc Ψ est un homomorphisme d'anneau. Comme K est fini et $\mathbb{Z}$ infini, Ψ est non-injectif, en conséquence, il existe $n \neq 0$ tel que $\Psi(n) = 0_K$ (si $\Psi(j) = \Psi(i)$ pour $j \neq i$ alors $n = |j - i|$ convient).
Si n n'est pas premier, soit $n = n_1 n_2$. On a $\Psi(n_1) \times \Psi(n_2) = 0_K$ donc $\Psi(n_1) = 0_K$ ou $\Psi(n_2) = 0_K$ (K est un corps donc ses éléments non nuls sont inversibles). Donc il existe p premier tel que $\Psi(p) = 0_K$.
Pour l'unicité de p : si p_1 et p_2 sont premiers et $\Psi(p_1) = \Psi(p_2) = 0_K$. alors, d'après Bézout, il existe a, b tels que $ap_1 + bp_2 = 1$ d'où $\Psi(1) = 0_K$, ce qui est absurde.

Exercice 1.19, page 63.
Soit Ψ_p la restriction de Ψ à F_p. Soit $k = \{\Psi_p(0), \Psi_p(1), \ldots, \Psi_p(p-1)\}$; k est isomorphe à F_p (Ψ_p est injectif et k et F_p ont même cardinal) ; donc k est un sous-corps de K. K est un espace vectoriel sur k ; soit m sa dimension. Comme K est fini, m est fini et $|K| = k^m$.

Exercice 1.20, page 63.

1. Condition nécessaire pour que P soit irréductible de degré $n \leqslant 2$: doit être de la forme $X^n + X^0 + \sum_{i=1}^{n-1} a_i X^i$ avec $\sum_{i=1}^{n-1} a_i = 1 \mod 2$.
   Cette condition est suffisante pour $2 \leqslant n \leqslant 3$. D'où
   De degré 2 : $1 + X + X^2$ car n'admet ni 0 ni 1 comme racine.
   De degré 3 : $X^3 + X + 1$ et $X^3 + X^2 + 1$.
2. Tout polynôme vérifiant la condition ci-dessus et qui n'est pas $(1 + X + X^2)^2 = 1 + X^2 + X^4$. D'où :
   $X^4 + X^3 + X^2 + X + 1, X^4 + X^3 + 1, X^4 + X + 1$
3. En utilisant le polynôme $1 + X + X^2$, on a les éléments $e_0 = 0$, $e_1 = 1$, $e_2 = X$, $e_3 = X + 1$. On a alors

$+$	e_0	e_1	e_2	e_3
e_0	e_0	e_1	e_2	e_3
e_1	e_1	e_0	e_3	e_2
e_2	e_2	e_3	e_0	e_1
e_3	e_3	e_2	e_1	e_0

$\times$	e_0	e_1	e_2	e_3
e_0	e_0	e_0	e_0	e_0
e_1	e_0	e_1	e_2	e_3
e_2	e_0	e_2	e_3	e_1
e_3	e_0	e_3	e_1	e_2

inverse	
e_0	–
e_1	e_1
e_2	e_3
e_3	e_2

Exercice 1.21, page 72.
Modulo 2, $1 = -1$, donc $x_4 = x_0+x_2+x_3 = 0+1+0 = 1$; $x_5 = x_1+x_3+x_4 = 1+0+1 = 0$; $x_6 = x_2+x_4+x_5 = 1+1+0 = 0$; $x_7 = x_3+x_5+x_6 = 0+0+0 = 0$; $x_8 = x_4 + x_6 + x_7 = 1 + 0 + 0 = 1$, etc.

Exercice 1.22, page 75.
On a $p_i = 0,1$ pour tout i, et $n = 21$. Puis $e_0 = 3$, $e_1 = 2$, $e_2 = 6$, $e_3 = 3$, $e_4 = 2$, $e_5 = 2$, $e_6 = 1$, $e_7 = 0$, $e_8 = 1$, $e_9 = 1$. Soit

$$K = \frac{0,9^2}{2,1} + \frac{0,1^2}{2,1} + \frac{3,9^2}{2,1} + \frac{0,9^2}{2,1} + \frac{0,1^2}{2,1} + \frac{0,1^2}{2,1} + \frac{1,1^2}{2,1} + \frac{2,1^2}{2,1} + \frac{1,1^2}{2,1} + \frac{1,1^2}{2,1}$$

Donc $K = 13,957$. Comme l'alphabet est de taille 10, le nombre de degrés de liberté est 9. Avec une probabilité 0,25 de se tromper, on peut donc affirmer que la distribution obtenue n'est pas uniforme (la valeur à comparer à K est 11,39). Mais on ne peut pas faire baisser cette probabilité d'erreur, car les autres valeurs du tableau sont supérieures à K. Comme la probabilité d'erreur est forte, il vaut mieux s'abstenir et ne rien conclure de ce test.

Exercice 1.23, page 79.

1. Le mot de code pour adbccab est 011110110110010, et le mot de source pour 1001101010 est bacbb.
2. On vérifie facilement en comparant tous les mots deux à deux qu'aucun n'est préfixe d'un autre. C'est donc un code instantané.
3. L'entropie de la source est $H = 1,75$

Exercice 1.24, page 79.

1. 0 n'est préfixe d'aucun autre mot de code, car ils commencent tous par 1. Tous les autres mots de code sont de la même longueur, ils ne peuvent donc pas être préfixes les uns des autres.

   Autre preuve : on peut dessiner un arbre de Huffman contenant tous les mots de code (voir la section 1.4.1).
2. $P(0\ldots01) = P(0)\ldots P(0)P(1) = p^k(1-p)$
3. Pour les mots de code $(100, 101, 110, 111, 0)$, les rapports (nombres de bits de code par bit de source) sont $(3/1, 3/2, 3/3, 3/4, 1/4)$. Le taux de compression est donc $l = 3(1-p) + 3/2(1-p)p + (1-p)p^2 + 3/4(1-p)p^3 + 1/4p^4 = 3 - 3/2p - 1/2p^2 - 1/4p^3 - 1/2p^4$.

Exercice 1.25, page 82.
($\Rightarrow$) On suppose qu'il existe un code instantané (c'est-à-dire qui vérifie la propriété du préfixe) avec les longueurs de mots $l_1, \ldots, l_n$. On a montré qu'on

peut construire sa représentation par un arbre de Huffman. L'arbre complet initial a pour hauteur l, et $|V|^l$ feuilles.
On compte à chaque opération d'élagage le nombre de feuilles de l'arbre initial complet qu'on enlève ainsi à l'arbre de Huffman. Pour un mot de longueur l_i, c'est le nombre de feuilles de l'arbre complet de hauteur $l - l_i$, soit $|V|^{l-l_i}$ (on suppose enlevée, car non réutilisée, une feuille d'un sous-arbre de hauteur 0). Pour toutes les opérations, on enlève $\sum_{i=1}^{n} |V|^{l-l_i}$ feuilles. Mais on ne peut en enlever au total plus que le nombre initial, soit $\sum_{i=1}^{n} |V|^{l-l_i} \leqslant |V|^l$, d'où : $\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leqslant 1$.
($\Leftarrow$) Réciproquement, on suppose que l'inégalité est satisfaite. On cherche à construire un arbre de Huffman dont les mots de code ont pour longueur $l_1, \ldots, l_n = l$, mots que l'on suppose classés par ordre croissant de leurs longueurs.
On part d'un arbre de hauteur l. On procède ensuite par induction.
Pour pouvoir choisir un nœud dans l'arbre, correspondant à un mot de longueur l_k, il faut qu'on puisse trouver dans l'arbre, un sous-arbre complet de hauteur $l - l_k$. Mais, toutes les opérations d'élagage précédentes consistent à enlever des sous-arbres de hauteur plus grande (correspondant à des mots plus courts). Donc, s'il reste au moins $|V|^{l-l_k}$ feuilles dans l'arbre restant, l'arbre restant contient un sous-arbre, de hauteur $l - l_k$. Montrons qu'il reste effectivement $|V|^{l-l_k}$ feuilles dans l'arbre restant.
Si on a déjà « placé » les mots de longueur $l_1, \ldots, l_{k-1}$, on a alors enlevé $\sum_{i=1}^{k-1} |V|^{l-l_i}$ feuilles de l'arbre initial par les opérations successives d'élagage. Il reste donc $|V|^l(1 - \sum_{i=1}^{k-1} |V|^{-l_i})$ feuilles. Or, on sait d'après l'inégalité de Kraft que :

$$\sum_{i=k}^{n} \frac{1}{|V|^{l_i}} \leqslant 1 - \sum_{i=1}^{k-1} \frac{1}{|V|^{l_i}}$$

soit :

$$|V|^l(1 - \sum_{i=1}^{k-1} |V|^{-l_i}) \geqslant \sum_{i=k}^{n} |V|^{l-l_i} \geqslant |V|^{l-l_k}$$

On peut donc placer le mot de longueur l_k, et en réitérant l'opération, construire l'arbre de Huffman. Le code de Huffman, de longueurs de mots $l_1, \ldots, l_n$, vérifie la propriété du préfixe.
Remarque : ceci prouve l'implication laissée en suspens du théorème de Kraft, un peu plus haut dans ce chapitre.

Exercice 1.26, page 84.

1. Il est facile de trouver des x tel que $y = H(x)$. Donc si l'on connaît x, on calcule $y = H(x)$, puis on trouve un autre x' tel que $y = H(x')$.

2. Il est facile de trouver x' connaissant x, donc on prend un x quelconque au hasard et on génère x'.

Exercice 1.27, page 84.
Prenons par exemple $x'_h = (x_h \oplus x_l)$ et $x'_l = 0$, alors on a $x_h \oplus x_l = x'_h \oplus x'_l$ et a fortiori également après l'application de f.

Exercice 1.28, page 86.
Supposons que H ne soit pas résistante aux collisions. Alors il est facile de trouver M et M' distincts tels que $H(M) = H(M')$. On suppose que M est sur $k \geqslant 1$ bloc de b bits et M' sur $l \geqslant 1$ blocs, avec $k \leqslant l$. Alors on pose $y_1 = h^{k-1}(M)$ et $y'_1 = h^{l-1}(M')$. Trois cas sont alors possibles :

1. $M_1 \dots M_{k-1} = M'_1 \dots M'_{k-1} = u$. Alors, $h(u||M_k) = h(u||M'_k \dots M'_l)$ et on a une collision sur h.
2. Soit $y_1 \neq y'_1$ et alors on a trouvé une collision sur h car $h(y_1) = H(M) = H(M') = h(y'_1)$.
3. Soit $y_1 = y'_1$ et $M_1 \dots M_{k-1} \neq M'_1 \dots M'_{k-1}$. Dans ce dernier cas on réitère le procédé avec $y_i = h^{k-i}(M)$ et $y'_i = h^{l-i}(M')$ jusqu'à obtenir $y_i \neq y'_i$.

Exercice 1.29, page 86.

1. Supposons que h ne l'est pas. Alors on peut trouver y et y' tels que $h(y) = h(y')$. Posons $y = y_1||y_2$ et $y' = y'_1||y'_2$. Alors, soit $x = f(y_1)||f(y_2)$ et $x' = f(y'_1)||f(y'_2)$. On a $f(x) = f(x') = h(y)$. Donc f n'est pas résistante aux collisions.
2. L'inconvénient de cette construction stricte est qu'elle n'est extensible qu'à des messages de taille une puissance de 2.

Exercice 1.30, page 92.
On vérifie tout d'abord que c'est possible : effectivement $31 - 1 = 2 \cdot 3 \cdot 5$ donc 6 divise bien 30. Ensuite, on obtient que $X^6 - 1 = (X^3 - 1)(X^3 + 1) = (X - 1)(X^2 + X + 1)(X + 1)(X^2 - X + 1)$. On cherche alors les racines, si il y en a, de $X^2 + X + 1$ et $X^2 - X + 1$: ce sont respectivement 5, -6, 6 et -5. Nous avons donc six candidats pour être racine primitives $6^{\text{ièmes}}$, et seuls $\varphi(6) = \varphi(2)\varphi(3) = 2$ conviennent : 1 n'est que racine primitive $1^{\text{ième}}$, -1 racine primitive $2^{\text{ième}}$, $5^3 = 5^2 \cdot 5 = -6 \cdot 5 = -30 = 1 \mod 31$ et 5 est donc racine racine primitive $3^{\text{ième}}$, de même que -6. Les deux racines primitives $6^{\text{ièmes}}$ de l'unité modulo 31 sont donc 6 et $-5 = 26$.

Exercice 1.31, page 92.
4 ne divise pas 30, il faut donc se placer dans une extension de $\mathbb{Z}_{31}$.
Factorisons $X^4-1=(X-1)(X+1)(X^2+1)$ et X^2+1 est irréductible modulo 31. On se place alors dans le corps $\mathbb{F}_{31^2} \simeq \mathbb{Z}_{31}[X]/X^2+1$ dans lequel on a bien 4 qui divise $31^2-1=960$.
On sait que les éléments de $\mathbb{Z}_{31}$ ne peuvent pas être racine primitive $4^{\text{ième}}$ et l'on regarde alors une des racines de X^2+1, notée par exemple i que l'on sait être racine $4^{\text{ième}}$. On vérifie que c'est bien une racine primitive : $i^2=-1$, $i^3=-i$ et $i^4=-i^2=1$.

Exercice 1.32, page 94.
– $\varphi(p)$. a est une racine primitive.
– $\varphi(p^e)$.
–
$$\begin{bmatrix} x_n & 1 \\ x_{n+1} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_{n+1} \\ x_{n+2} \end{bmatrix} [m]$$
Donc
$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{x_n - x_{n+1}} \begin{bmatrix} 1 & -1 \\ -x_{n+1} & x_n \end{bmatrix} \begin{bmatrix} x_{n+1} \\ x_{n+2} \end{bmatrix} [m]$$
– $x_{n+1}-x_n = a(x_n-x_{n-1})$ et $x_{n+2}-x_{n+1} = a(x_{n+1}-x_n)$. Donc $(x_{n+1}-x_n)^2 = (x_{n+2}-x_{n+1})(x_n-x_{n-1})$ si a est inversible modulo m. Dans ce cas m est un facteur de la différence de cette égalité et l'on est ramené au cas précédent.
– $p=1009$, $a=238$, $b=12$, $X_0=456$, $X_6=253$.

Exercice 1.33, page 103.
Par définition, $r-1$ a pour facteur t. Si $s \neq r$ alors $s^{r-1}=1 \mod r$ d'après le théorème de Fermat. Donc $m=1 \mod r$, ce qui veut dire que $p-1=0 \mod r$. Ensuite, $m=-1 \mod s$ par définition, donc $p+1=0 \mod s$. Par conséquent, r divise $p-1$, s divise $p+1$ et t divise $r-1$.

Solutions des exercices proposés au chapitre 2

Exercice 2.1, page 107.

1. 2^N fichiers.
2. 1 fichier de taille 0, 2 fichiers de taille 1, 4 fichiers de taille 2, ..., 2^{N-1}. Soit un total de $\frac{2^N-1}{2-1} = 2^N - 1$ fichiers de strictement moins que N bits.
3. Ceci prouve que soit au moins deux fichiers distincts de N bits seront compressés de manière identique, et donc il y aura eu perte. En effet, dans ce cas, après compression il sera impossible de savoir vers lequel des fichiers initiaux décompresser. L'autre solution est que certains fichiers compressés font plus de N bits. Dans ce cas il n'y a pas eu de compression.

Exercice 2.2, page 114.
Le code issu de l'algorithme de Huffman est :

S	C
000	1
001	010
010	011
100	001
011	00011
101	00010
110	00001
111	00000

Sa longueur moyenne est : $l = 1.05$, soit par bit de source $l = 0.35$. C'est supérieur à la longueur moyenne du code de l'exercice 1.24, mais cela ne remet pas en cause l'optimalité de Huffman : il faudrait comparer avec la quatrième extension, qui est d'après l'optimalité de Huffman meilleure que le code construit. L'entropie de $\mathcal{S}$ est $H(\mathcal{S}) = 0.99 log_2 \frac{1}{0.99} + 0.01 log_2 \frac{1}{0.01} = 0.0808$. Le théorème est bien vérifié dans tous les cas.

Exercice 2.3, page 114.
L'idée est de procéder par *rejet* (avec relance de la pièce si nécessaire).
Pour cela, il faut générer 6 évènements de probabilités uniformes avec une pièce en tirant 3 fois à pile ou face et en associant les évènements suivants (Pile = P ou Face = F) à un jet de dé :

Tirages Pile ou Face	PPP	PPF	PFP	PFF	FPP	FPF
Jet de dé	1	2	3	4	5	6

Si on trouve FFP ou FFF : on rejette le tirage (on peut donc s'arrêter après deux jets, dès que l'on a $FF-$) et on relance 3 fois la pièce (tirages indépendants).
Le nombre N moyen de lancers de pièces est donc : $N = (\frac{6}{8}) * 3 + (\frac{2}{8}) * (\frac{6}{8}) * (2+3) + (\frac{2}{8})^2 * (\frac{6}{8}) * (2+2+3) + ...$, ou encore $N = (\frac{6}{8}) * \sum_{k=0}^{\infty} (2k+3)(\frac{2}{8})^k = 3 * (\frac{6}{8}) * \sum_{k=0}^{\infty} (\frac{2}{8})^k + 2 * (\frac{6}{8}) * \sum_{k=0}^{\infty} k(\frac{2}{8})^k = 3 * (\frac{6}{8}) \frac{1}{1-(\frac{2}{8})} + 2 * (\frac{6}{8}) * (\frac{2}{8}) \sum_{j=1}^{\infty} (j+1)(\frac{2}{8})^j$
Or

$$\sum_{j=0}^{\infty} (j+1)x^j = \left(\sum_{j=1}^{\infty} x^k \right)' = (\frac{1}{1-x}\ ' = \frac{1}{(1-x)^2}.$$

D'où $N = 3 * (\frac{6}{8}) * (\frac{8}{6}) + 2 * (\frac{6}{8}) * (\frac{2}{8}) * (\frac{8}{6})^2 = 3 + (\frac{2}{3})$, soit une moyenne de 11 pile ou face pour les trois dés du 421.
L'entropie des trois dés non pipés étant $3 * \left(6 * (\frac{1}{6}) * \log_2(6)\right) \approx 7.755$, ce codage n'est sans doute pas optimal.
En effet, le cas de l'extension de source à 3 dés permet de faire en encodage sur 8 pile ou face, puisque $6^3 = 216 < 256 = 2^8$. Dans ce cas, la moyenne du nombre de tirages nécessaires avec rejets est plus proche de 8.5.

Exercice 2.4, page 116.

1. H(dé)=2.58.
2. sur {0,1,2}, Huffman construit le code (par exemple) : (1-> 10 , 2-> 12 , 3-> 01 , 4-> 00 , 5-> 02 , 6-> 2), de longueur moyenne (fixe) égale à $0.22*1+(1-0.22)*2 = 1.78$ chiffres. C'est le code optimal (par théorème), on ne pourra obtenir mieux (et toujours plus que $2.58/log_2(3) = 1.62$) qu'en codant la source par séquences de chiffres.
3. Sur {0,1}, un code possible est : (1-> 000 , 2-> 001 , 3-> 010 , 4-> 011 , 5-> 10 , 6-> 11), de longueur moyenne 2.6. C'est aussi le code optimal.

Exercice 2.5, page 116.

1. On code les séquences {1,2,3,4,51,52,53,54,55,56,61,62,63,64,65,66} (au nombre de 16) avec les 16 mots de longueur 4.
2. Selon le code choisi (dont ne dépend pas la longueur des séquences), on peut avoir par exemple : Tunstall(66-> 1111 et 64-> 1100) : 11111100 et Huffman : 11|11|11|011.
3. Pour coder les séquences de deux chiffres, on a besoin de 4 bits, soit un rendement de 2, et de même pour les mots de un chiffre, soit un rendement de 4.

   La longueur moyenne par bit de source est donc : $2 * 0.22 + 2 * 0.18 + 4 * 0.60 = 3.2$.

Exercice 2.6, page 121.

Nombre	*Intervalle*	*Sortie*
49991	[40000; 59999]	*b*
49991	[48000; 51999]	*b*
49991	[49600; 50399]	*b*
49916	[46000; 53999]	*shift*
49916	[49200; 50799]	*b*
49168	[42000; 57999]	*shift*
49168	[48400; 51599]	*b*
49168	[48400; 49679]	*a*
91680	[84000; 96799]	*shift*
91680	[91680; 96799]	*c*

Exercice 2.7, page 122.

1. Par exemple : (a->0 ;b->10 ;c->110 ;d->111).
2. a, b, c et d nécessitent donc $4 * 8$ bits pour être écrits et leurs codes nécessitent $1 + 2 + 3 + 3$ bits, soit un total d'au moins 41 bits.
3. Dans le cas de l'extension de source à 3 caractères, il y a $4^3 = 64$ triplets possibles, chacun écrit sur 8 bits et il y a donc également 64 codes nécessitant donc au moins $64 * 6$ bits au total[3]. Soit, à mettre dans le fichier compressé : au moins $64 * 8 + 64 * 6 = 896$ bits.
4. 18 caractères ASCII :$8 * 18 = 144$ bits.
5. Compression par Huffman : $0, 0, 0, 0, 0, 0, 0, 0, 0, 10$, 110, 111, 10, 110, 111, 10, 110, 111, soit $1 * 9 + 3 * (2 + 3 + 3) = 33$ bits, auxquels il faut ajouter 41 bits de table. Pour une extension de source à 3 caractères, la table (ou l'arbre) nécessiterait déjà un espace mémoire plus de 5 fois plus grand que le fichier initial.

Exercice 2.8, page 125.

1. @a| 1| 0| 0| 0| 0| 0| 0| 0| @b| @c| @d, là l'arbre dynamique est a->0 ; @->10 ; b->110 ; c->1110 ; d->1111, la suite de la compression est donc 110| 1110| 1111| 110| 1110 et juste avant le dernier d l'arbre est devenu a->0 ; @->100 ; b->101 ; c->110 ; d->111, et donc le dernier d est codé par 111. La compression totale en ignorant le premier @ est donc : $ASCII(a)$

[3]Pour voir que c'est une borne inférieure il faut considérer un code de 64 mots dont certains font strictement plus de 6 bits. Alors cela veut dire qu'il y a aussi des mots de strictement moins de 6 bits. Prenons un mot par exemple de 8 bits et un mot de 5 bits (par exemple 10101), remplaçons dans le code ces deux mots par les deux mots de 6 bits qui l'allongent directement : 101010 et 101011. Le code obtenu nécessite $8 + 5 - 6 - 6 = 1$ bit de moins dans sa table.

10000000 1$ASCII(b)$ 10$ASCII(c)$ 10$ASCII(d)$ 110 1110 1111 110 1110 111, soit $8+8+1+8+2+8+2+8+3+4+4+3+4+3 = 66$ bits seulement au total (à comparer avec les $33+41 = 74$ bits obtenus par Huffman statique).

2. Sur des groupes de 3 caractères cela donne sur ce cas particulier @aaa| 1| 0| @bcd| 10| 10, soit $ASCII(aaa)101ASCII(bcd)1010$ qui nécessite seulement $3*8+1+1+1+3*8+4 = 55$ bits au total.

Exercice 2.9, page 126.

1. Par exemple, 6801314131413141312222611, où 68 code pour les dimensions de l'image (6×8 pixels), puis chaque chiffre pour le nombre de pixels consécutifs de même couleur (alternance blanc-noir).
2. 25 octets (ou 13 en limitant à des blocs de taille 16) au lieu de 6, plutôt une perte sur les petites images !
3. On note un bloc par un bit collé à quatre autres bits en hexadécimal (par exemple : $0F = [0,1,1,1,1]$). Le premier bit indique si l'on code une couleur ou un run. On obtient donc : 6, 8, 00, (13, 0F), 00, (14, 0F), 00, (13, 0F), 00, (14, 0F), 00, (13, 0F), 00, (14, 0F), 00, (13, 0F), 00, 0F, 0F, 00, 00, 0F, 0F, 00, 00, (16, 0F), 00, 0F. Pour $34*5+2*8 = 186\ bits = 24\ octets$ au lieu de $6*8*4+2*8 = 192\ bits = 26 octets$. On peut également supprimer le bit à 0 dans la couleur suivant un run. On obtient alors : 6, 8, 00, 13F, 00, 14F, 00, 13F, 00, 14F, 00, 13F, 00, 14F, 00, 13F, 00, 0F, 0F, 00, 00, 0F, 0F, 00, 00, 16F, 00, 0F. Ce qui fait $8*4+26*5+2*8 = 178\ bits = 23\ octets$.
4. Le caractère de répétition est FF, on code noir par "00" et blanc par "FE".
   On a donc : 6, 8, 00, (FF, 3, FE), 00, (FF, 4, FE), 00, (FF, 3, FE), 00, (FF, 4, FE), 00, (FF, 3, FE), 00, (FF, 4, FE), 00, (FF, 3, FE), 00, FE, FE, 00, 00, FE, FE, 00, 00, (FF, 6, FE), 00, FE. Soit $8*4+34*8+2*8 = 320\ bits = 40\ octets$ au lieu de $6*8*8+2*8 = 400\ bits = 50\ octets$
5. 6, 8, 00, (FF, 6, FE), 00, (FF, 6, FE), 00, FE, 00, (FF, 4, FE), 00, 00, FE, 00, (FF, 6, FE), 00, (FF, 6, FE), 00, FE, 00, 00, (FF, 3, FE), 00, 00, FE. Soit $6*4+29*8+2*8 = 272\ bits = 34\ octets$ au lieu de 50.

Exercice 2.10, page 128.

1. "0123321045677654" et "0123456701234567".
2. "0123012345670123" et "0123456777777777". La deuxième chaîne a une entropie visiblement réduite.
3. Les fréquences sont égales à 1/8, l'entropie est donc maximale et 3 bits sont nécessaires pour chaque caractère. Il faut donc $3*16/8 = 6$ octets.

4. Pour la première chaîne on obtient une entropie : $H = 4(\frac{3}{16}\log_2(\frac{16}{3})) + 4(\frac{1}{16}\log_2(16)) \approx 1.81 + 1 = 2.81$. Par Huffman le code est donc : 00, 01, 100, 101, 1100, 1101, 1110, 1111, d'où un codage sur seulement $2*3*2+2*3*3+4*1*4 = 46 = 5.75$ octets. La deuxième chaîne s'y prête encore mieux : $H = 7(\frac{1}{16}\log_2(16)) + (\frac{9}{16}\log_2(\frac{16}{9})) = 1.75 + .47 = 2.22$. Huffman donne alors : 00, 01, 100, 101, 1100, 1101, 111, 1, d'où seulement $2+2+3+3+4+4+3+1*9 = 30 = 3.75$ octets !
5. Huffman donne : 000, 001, 010, 011, 1, pour respectivement 12, 34, 56, 67 et 77. Ainsi, $4*3+4*1 = 16 = 2$ octets sont nécessaires, mais avec une table de taille 256 octets au lieu de 8 triplets soit 3 octets.
6. Un « Move-to-Front » suivi d'un code statistique est donc un « code statistique localement adaptatif ».
7. Pour $k = 1$ on obtient "0123220345676644" et "0123456770123457" d'entropies respectives $H = 2.858$ et $H = 2.953$. Pour $k = 2$, cela donne : "0123112345675552" et "0123456777012347" d'où $H = 2.781$ et $H = 2.875$.

Exercice 2.11, page 131.

1. Car le message initial peut être reconstruit à partir de L et de l'index primaire, mais pas à partir de F !!!
2. Ici, S=L="sssssssssh" ! Move-to-front donne C=(1,0,0,0,0,0,0,0,0,1), qui est aussi la sortie de Huffman.
3. Implémentation pratique :
   (a) On peut se servir de la position du premier caractère du message source.
   (b) **Entrées** Une chaîne S (de taille n). Deux entiers i et j.
   Sorties « i est avant » ou « j est avant »
   Tant que on n'a pas de réponse **Faire**
   Si $S[i] < S[j]$ **Alors**
   Répondre « i est avant »
   Sinon, si $S[i] > S[j]$ **Alors**
   Répondre « j est avant »
   Sinon
   $i \leftarrow (i+1) \mod n$ et $j \leftarrow (j+1) \mod n$
   Fin Si
   Fin Tant que
   (c) Il faut pouvoir stocker S le message source, puis T contenant les indices de permutation et enfin L que l'on calcule par $S[T[i]-1]$.
   (d) L'index primaire est celui pour lequel $T[i] = 1$, puisque c'est là que se trouve la ligne 1.

Exercice 2.12, page 132.

1. Il faut faire attention à bien mettre la dernière lettre (ici le f) dans le dernier triplet et donc se limiter à en reprendre seulement 10, et pour le dernier triplet, il faut bien sûr que la distance soit toujours plus longue que la longueur : $(0,0,a)$ $(0,0,b)$ $(0,0,c)$ $(0,0,d)$ $(0,0,e)$ $(0,0,f)$ $(6,6,a)$ $(12,10,f)$.
2. Là, la distance et la longueur doivent tenir sur 3 bits, et sont donc comprises entre 0 et 7. Il faut donc couper le dernier bloc en deux morceaux : $(0,0,a)$ $(0,0,b)$ $(0,0,c)$ $(0,0,d)$ $(0,0,e)$ $(0,0,f)$ $(6,6,a)$ $(6,6,b)$ $(6,3,f)$.

Exercice 2.13, page 134.
La compression de "BLEBLBLBA" par LZW donne donc le résultat suivant :

chaîne		Affichage		Dictionnaire		
B	$\rightsquigarrow$	42		BL	$\leftrightarrow$	80
L	$\rightsquigarrow$	4C		LE	$\leftrightarrow$	81
E	$\rightsquigarrow$	45		EB	$\leftrightarrow$	82
BL	$\rightsquigarrow$	80		BLB	$\leftrightarrow$	83
BLB	$\rightsquigarrow$	83		BLBA	$\leftrightarrow$	84
A	$\rightsquigarrow$	41				

Si l'on décompresse cette sortie sans traiter le cas particulier, on obtient à cause du décalage de mise en place du dictionnaire la sortie suivante :

Code		Affichage		Dictionnaire		
42	$\rightsquigarrow$	B				
4C	$\rightsquigarrow$	L		BL	$\leftrightarrow$	80
45	$\rightsquigarrow$	E		LE	$\leftrightarrow$	81
80	$\rightsquigarrow$	BL		EB	$\leftrightarrow$	82
83	$\rightsquigarrow$	? ? ?				

Puisque que le code 83 n'est pas connu c'est que l'on est précisement dans le cas particulier et que le prochain groupe est donc forcément la répétition du précédent (BL) augmenté de la première lettre du groupe, soit BLB. On voit donc bien ici, que le traitement du cas particulier permet de mettre à jour le dictionnaire une itération plus tôt. Ainsi, toute chaîne répétée deux fois de suite peut être traitée correctement à la décompression.

Exercice 2.14, page 135.

1. (0, 0, a) ; (0, 0, b) ; (1, 1, a) ; (0, 0, r) ; (3, 3, b) ; (4, 2, b) ; (10, 5, a)
2. "0, 0, 1, 0, 3, 4, 10" devient "1-0, 0, 1-1, 00, 1-3, 1-4, 1-10" par Huffman dynamique . "a, b, a, r, b, b, a" devient "1-a, 1-b, 00, 1-r, 010, 010, 010". D'où le codage gzip : (10, 0, 1a) ; (0, 0, 1b) ; (11, 1, 00) ; (00, 0, 1r) ; (13, 3, 010) ; (14, 2, 010) ; (110, 5, 010).

3. f1.gz : (0, 0, a) ; (0, 0, b) ; (1, 1, a) ; (0, 0, r) ; (3, 3, b),
   et donc (10, 0, 1a) ; (0, 0, 1b) ; (11, 1, 00) ; (00, 0, 1r) ; (13, 3, 010)
   f2.gz : (0, 0, b) ; (0, 0, a) ; (2, 1, b) ; (3, 1, a) ; (0, 0, r) ; (3, 2, a)
   qui devient (10, 0, 1b) ; (00, 0, 1a) ; (12, 1, 00) ; (13, 1, 01) ; (00, 0, 1r) ;
   (011, 2, 010).
4. La première distance pour le premier caractère d'un fichier est forcément 0. Les deux premiers caractères d'un fichier gzip sont donc forcément 10. En outre, comme @ est fixé au bit 1, le début de fichier est le seul endroit où la séquence 10 peut être rencontrée. Donc toute rencontre, dans le fichier, de la séquence 10 sur la chaîne des occurrences est forcément une remise à zéro des Huffman dynamiques. Cette particularité fait qu'il n'y a pas de différence structurelle entre un seul fichier compressé par `gzip` et une séquence de fichiers compressés. En décompressant avec `gunzip f3.gz`, on obtiendra donc bien en résultat un fichier dont le contenu est identique à la concaténation de `f1` et `f2`.
5. En changeant la taille maximale de fenêtre de LZ77. Avec une petite fenêtre il est plus rapide mais la compression est moins bonne.

Solutions des exercices proposés au chapitre 3

Exercice 3.1, page 149.

1. $|\mathbb{Z}_{26}^*| = \varphi(26) = \varphi(13 \times 2) = 12$ et $\mathbb{Z}_{26}^* = \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}$.
2. $15 \in \mathbb{Z}_{26}^*$ donc la clef est valide et on obtient les chiffrements successifs 7, 22 et 11.
3. $$\begin{array}{rccl} D_{(a,b)} : & \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ & y & \longrightarrow & D_{(a,b)}(y) = a^{-1}(y-b) \mod n \end{array}$$
4. (a) Une recherche exhaustive requiert de l'ordre de $\mathcal{O}(|\mathbb{Z}_{26}^*|.|\mathbb{Z}_{26}|)$ tests de clefs. Grâce à l'analyse de fréquence, on est capable de proposer des correspondances pour les lettres les plus courantes. Ainsi, on est quasiment assuré que la lettre 'e' est chiffrée en 'o'. On en déduit l'équation :

$$E_{(a,b)}(\text{'e'}) = \text{'o'} \Longleftrightarrow 4a + b = 14 \mod 26$$

À partir de là, deux méthodes sont possibles :
- On résout l'équation diophantienne $4a + b = 14$ dans $\mathbb{Z}$ et on en déduit les solutions possibles dans $\mathbb{Z}_{26}$. On teste les clefs admissibles (pour lesquelles $a \in \mathbb{Z}_{26}^*$) jusqu'à trouver la bonne. On rappelle que pour résoudre cette équation, on commence par résoudre l'équation homogène $4a + b = 0$ (l'ensemble des solutions est $S_H = \{(-k, 4k), k \in \mathbb{Z}\}$), on trouve ensuite une solution particulière (a_0, b_0) (soit directement (0,14), soit en calculant les coefficients de Bézout par l'algorithme d'Euclide étendu) et on en déduit l'ensemble des solutions dans $\mathbb{Z}$. On trouve ainsi $S = \{(-k, 14 + 4k), k \in \mathbb{Z}\}$. On en déduit les valeurs admissibles de k pour trouver les solutions dans $\mathbb{Z}_{26} : 0 \leqslant a < 26 \Rightarrow -25 \leqslant k \leqslant 0$. Il reste à tester les clefs admissibles correspondantes. On obtient les résultats suivants :
  - $k = -25 \Rightarrow K = (a, b) = (1, 10)$. On déchiffre avec cette clef : `csqxre yirvesa, uar af srvre...` Ce n'est donc pas la bonne clef.
  - $k = -23 \Rightarrow K = (a, b) = (3, 2)$. On déchiffre avec cette nouvelle clef : `maitre corbeau, sur un arbre...` On a trouvé la bonne clef !

  De façon générale, on aura au pire $\mathcal{O}(|\mathbb{Z}_{26}^*|)$ essais à effectuer.
- On extrait une seconde relation probable (pour le chiffrement de 's', la lettre la plus fréquente après le 'e') et on résout le système

correspondant. Compte tenu des résultats de l'analyse statistique, on testera successivement les systèmes

$$(1)\ \begin{cases} e_{(a,b)}(\text{'e'}) = \text{'o'} \\ e_{(a,b)}(\text{'s'}) = \text{'b'} \end{cases} \iff \begin{cases} 4a + b = 14 \mod 26 \\ 18a + b = 1 \mod 26 \end{cases}$$

$$(2)\ \begin{cases} e_{(a,b)}(\text{'e'}) = \text{'o'} \\ e_{(a,b)}(\text{'s'}) = \text{'c'} \end{cases} \iff \begin{cases} 4a + b = 14 \mod 26 \\ 18a + b = 2 \mod 26 \end{cases}$$

$$(3)\ \begin{cases} e_{(a,b)}(\text{'e'}) = \text{'o'} \\ e_{(a,b)}(\text{'s'}) = \text{'e'} \end{cases} \iff \begin{cases} 4a + b = 14 \mod 26 \\ 18a + b = 4 \mod 26 \end{cases}$$

$$\vdots$$

On montre que c'est finalement ce dernier système qui fournit la bonne clef.

La clef de chiffrement utilisée est donc $K = (a, b) = (3, 2)$.

(b) On obtient ainsi le texte déchiffré suivant :

```
maitre corbeau, sur un arbre perche,
   tenait en son bec un fromage.
maitre renard, par l'odeur alleche
 lui tint a peu pres ce langage :
"he ! bonjour, monsieur du corbeau,
que vous etes joli ! que vous me semblez beau !
      sans mentir, si votre ramage
      se rapporte à votre plumage,
vous etes le phenix des hotes de ces bois."
```

Exercice 3.2, page 154.
Les deux blocs L_i et R_i sont obtenus à partir des blocs précédents et de la clef de tour K_i par : $L_i = R_{i-1}$ et $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$. Alors, au déchiffrement, on peut obtenir les deux blocs précédents L_{i-1} et R_{i-1} à partir des trois valeurs L_i, R_i et K_i simplement par $R_{i-1} = L_i$ et $L_{i-1} = R_i \oplus f(L_i, K_i)$. Il n'est donc pas nécessaire d'inverser f. En outre le déchiffrement est bien identique au chiffrement si l'on applique les tours dans l'ordre inverse.

Exercice 3.3, page 156.
- La connaissance de la clef et du crypté permet de décrypter ; autrement dit en général $H(K, C) \geqslant H(M, C)$, ou encore $H(K|C) + H(C) \geqslant H(M|C) + H(C)$. La définition du chiffrement parfait donne $H(K|C) \geqslant H(M)$. Il y a donc forcément au moins autant de K possibles que de M.
- M et C sont codés par des mots de $\{0, 1\}^{64}$ alors que les clefs ne sont que sur 56 bits. Le nombre de clefs n'est donc pas suffisant pour avoir un chiffrement parfait.

Exercice 3.4, page 159.

1. (a) On fournit le résultat pour différentes représentations :
Avec des polynômes : $(X^6 + X^4 + X^2 + X + 1) + (X^7 + X + 1) = X^7 + X^6 + X^4 + X^2$.
En notation binaire : $[01010111] + [10000011] = [11010100]$
En notation hexadécimale : $[57] + [83] = [D4]$

   (b)
$$\begin{aligned}(X^6 + X^4 + X^2 + X + 1)(X^7 + X + 1) = \\ X^{13} + X^{11} + X^9 + X^8 + X^7 + X^7 + \\ X^5 + X^3 + X^2 + X + X^6 + X^4 + X^2 + X + 1 = \\ X^{13} + X^{11} + X^9 + X^8 + X^6 + X^5 + X^4 + X^3 + 1\end{aligned}$$

   Et $(X^{13} + X^{11} + X^9 + X^8 + X^6 + X^5 + X^4 + X^3 + 1) \mod (X^8 + X^4 + X^3 + X + 1) = X^7 + X^6 + 1$.
   Le modulo $g(X)$ assure que le résultat reste un polynôme binaire de degré inférieur à 8 qui peut donc être représenté par un octet. On a ainsi obtenu : $[57] \times [83] = [C1]$.

2. (a) Soit $a \in \mathbb{F}_{256}$. En notation polynomiale, $a(X) = a_7X^7 + \ldots + a_1X + a_0$.
   Ainsi, $X.a(X) = a_7X^8 + \ldots + a_1X^2 + a_0X$. Modulo $g(X)$, deux cas sont possibles :
   - Soit $a_7 = 0$, on obtient directement une expression réduite et donc $X.a(X) = a_6X^7 + \ldots + a_1X^2 + a_0X$.
   - Soit $a_7 = 1$ et dans ce cas : $X.a(X) = X^8 + \ldots + a_1X^2 + a_0X$. En outre, $g(X)$ est forcément nul modulo $g(X)$, ce qui implique que $X^8 = X^4 + X^3 + X + 1 \mod g(X)$ et donc

$$\begin{aligned}X^8 + a_6X^7 + \ldots + a_1X^2 + a_0X = \\ (a_6X^7 + \ldots + a_1X^2 + a_0X) \oplus (X^4 + X^3 + X + 1)\end{aligned}$$

     En notation polynomiale, cette opération consiste donc à réaliser un décalage à gauche suivi éventuellement d'un XOR bits-à-bits avec `{1B}`.

   Pour résumer :

$$X * [a_7a_6a_5a_4a_3a_2a_1a_0] = \begin{cases}[a_6a_5a_4a_3a_2a_1b_00] & \text{si } a_7 = 0 \\ [a_6a_5a_4a_3a_2a_1b_00] \oplus [00011011] & \text{sinon}\end{cases}$$

(b) On itère i fois l'algorithme précédent.

3. $\mathbb{F}_{256} \simeq \{0\} \cup \left\{w(X)^i \mod g(X)\right\}_{0 \leqslant i < 255}$; avec cette représentation (dite cyclique ou exponentielle, ou de Zech), la multiplication de deux éléments a et b non nuls est aisée :

$$\begin{aligned} a(X) &= w(X)^i && (4.9)\\ b(X) &= w(X)^j && (4.10)\\ a(X) \times b(X) &= w(X)^{i+j} \mod 255 && (4.11) \end{aligned}$$

On génère une table `ExpoToPoly` de 256 entrées telle que la $k^{\text{ième}}$ entrée `ExpoToPoly[k]` donne la représentation polynomiale de $w(X)^k$ modulo $g(X)$. (Par convention, on représentera l'élément 0 par $w(X)^{255}$ bien que mathématiquement, $w^{255} = 1$). La table `PolyToExpo` correspond à la table inverse. On utilise ces tables pour effectuer efficacement la multiplication de deux éléments a et b à partir de la relation précédente.

Exercice 3.5, page 163.
`InvSubBytes` consiste à effectuer la même manipulation mais à partir de la Boite-S inverse S^{-1} notée `InvSBox`.
Remarque : Comme la fonction t est son propre inverse, on a :

$$\texttt{SBox}^{-1}[a] = t^{-1}\left(f^{-1}(a)\right) = t\left(f^{-1}(a)\right), \text{ pour tout } a \in \mathbb{F}_{256}$$

La fonction affine inverse f^{-1} est définie par :

$$b = f^{-1}(a) \Longleftrightarrow \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Exercice 3.6, page 163.
Par définition, $\texttt{SBox}[a] = f\left(t(a)\right)$, pour tout $a \in \mathbb{F}_{256}$ avec $t : a \longrightarrow a^{-1}$ sur $\mathbb{F}_{256}$.
Or $a = X + 1$ en notation polynomiale. Donc $t(a) = (1+X)^{-1} = X^2 + X^4 + X^5 + X^7 \mod g(X) = [10110100] = [B4]$ par l'algorithme d'Euclide. Ensuite, on fait l'opération matricielle pour obtenir $f([B4]) = [00011110] + [11000110] = [11011000] = [D8]$.

Exercice 3.7, page 163.
`InvShiftrows` consiste évidemment à effectuer au niveau le la ligne i un décalage cyclique *à droite* de C_i éléments.

Exercice 3.8, page 164.

1. $[0B] + [A2] = [A9]$, $-[03] = [03]$, $[FD] - [F0] = [FD] + [F0] = [0D]$, $[23] + [45] = [66]$.
2. $X^8 + X^4 + X^3 + X + 1 = (X + 1)(X^7 + X^6 + X^5 + X^4 + X^2 + X) + 1$.
3. $X + 1 = [03]$, donc $[03]^{-1} = [F6]$.
4. La multiplication par X revient à décaler le mot binaire vers la gauche. Ensuite, si X^8 est présent, il faut le remplacer par $X^4 + X^3 + X + 1$, et faire la somme. Cela revient à faire un 'ou exclusif' (XOR) avec 00011011. On obtient ainsi, avec $X = [02]$, les monômes suivants : $X^8 = 00011011 = [1B]$, $X^9 = 00110110 = [36]$, $X^{10} = 01101100 = [6C]$, $X^{11} = 11011000 = [D8]$, $X^{12} = 10101011 = [AB]$, $X^{13} = 01001101 = [4D]$ et $X^{14} = 10011010 = [9A]$.
5. $(a_1 + a_2 + \ldots + a_n)^2 = a_1^2 + a_2^2 + \ldots + a_n^2 \mod 2$.
6. $[F6]^2 = X^{14} + X^{12} + X^{10} + X^8 + X^4 + X^2$, soit en additionnant les valeurs binaires : $[F6]^2 = 01010010 = [52]$.
7. Par la division euclidienne, on obtient directement (par exemple en identifiant les coefficients) $Q = [F6]Y + [52]$.
8. $(U - VQ)c + VM = 1$.
9. Le décodage de MixColumn se fait en multipliant par l'inverse de c, $(U - VQ)$, qui vaut $[OB]Y^3 + [0D]Y^2 + [09]Y + [0E]$. Cette étape revient à effectuer le calcul $b(X) = d(X) \times a(X) \mod (X^4 + 1)$ qui s'écrit matriciellement ainsi :

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \texttt{0E} & \texttt{0B} & \texttt{0D} & \texttt{09} \\ \texttt{09} & \texttt{0E} & \texttt{0B} & \texttt{0D} \\ \texttt{0D} & \texttt{09} & \texttt{0E} & \texttt{0B} \\ \texttt{0B} & \texttt{0D} & \texttt{09} & \texttt{0E} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Exercice 3.9, page 165.
C'est exactement la même opération puisque l'addition sur $\mathbb{F}_2$ est son propre inverse !

Exercice 3.10, page 167.
La routine de chiffrement peut être inversée et réordonnée pour produire un algorithme de déchiffrement utilisant les transformations `InvSubBytes`,

Algorithme 29 Déchiffrement AES.

Entrées Une matrice `State` correspondant au bloc chiffré, une clef K
Sorties Une matrice `State` correspondant au bloc clair

```
KeyExpansion(K, RoundKeys)
AddRoundKey(State, RoundKeys[N_r]) ; // Addition initiale
Pour r ← N_r − 1 à 0 Faire
  InvShiftRows(State) ;
  InvSubBytes(State) ;
  AddRoundKey(State, RoundKeys[r]) ;
  InvMixColumns(State) ;
Fin Pour
// FinalRound
InvShiftRows(State) ;
InvSubBytes(State) ;
AddRoundKey(State, RoundKeys[0]) ;
```

`InvShiftRows`, `InvMixColumns`, et `AddRoundKey`. Une modélisation formelle de l'algorithme de déchiffrement est fournie dans l'algorithme 29.
Dans cette version du déchiffrement, la séquence des transformations diffère de celle du chiffrement, le traitement de la clef restant inchangé. Certaines propriétés de l'algorithme de Rijndael permettent d'implémenter une routine de déchiffrement équivalente qui respecte la séquence de transformations de la routine de chiffrement, la structure de celle-ci étant la plus efficace. Cette version équivalente n'est pas demandée ici.

Exercice 3.11, page 167.

1. Un exemple parmi tant d'autres : un fichier contenant les salaires :
   - Le même bloc chiffré que moi : le même salaire
   - Échanger un bloc chiffré avec un autre : échange de salaire
2. (a) Triple DES code sur 64 bits, soit 3 octets, donc $2^{35-3} = 2^{32}$ blocs. probabilité $\approx \frac{N(N-1)}{2H} = \frac{2^{32}(2^{32}-1)}{2*2^{64}} \approx 0.5$, une chance sur deux !
   (b) Si $Y_i = Y_j$, alors $X_i \oplus Y_{i-1} = X_j \oplus Y_{j-1} = S$. Alors $Y_{i-1} \oplus Y_{j-1} = X_i \oplus X_j = X_i \oplus S \oplus X_j \oplus S = Z$ et Z est le XOR de deux parties claires du message. Avec des essais de messages en clair M, on obtient ainsi que $X_i \oplus X_j \oplus M$ est un morceau clair.
   (c) Avec AES codé sur 128 bits, on a seulement $\frac{N(N-1)}{2H} = \frac{2^{32}(2^{32}-1)}{2*2^{128}}$ soit environ 3.10^{-20} chances de collision. Ainsi les deux paramètres, taille de clef et taille de bloc, sont d'importance pour obtenir des codes résistants, le premier contre les attaques par force brute, le deuxième pour éviter le risque de collision.

Exercice 3.12, page 170.

1. $p = 47$ et $q = 59$. On notera que les valeurs proposées pour p et q sont faibles et ne correspondent évidemment pas à des clé réelles.

   - on calcule $n = p * q = 47 * 59 = 2773$.
   - on calcule $\varphi(n) = (p-1)(q-1) = 46 * 58 = 2668$
   - on vérifie que e est bien premier avec $\varphi(n)$ et on calcule son inverse $d = e^{-1} \mod \varphi(n)$. On utilisera pour cela l'algorithme d'Euclide étendu. On obtient ainsi $d = 157$.

   Finalement, la clé publique est $K_e = (e, n) = (17, 2773)$. La clé privée est $K_d = (d, n) = (157, 2773)$.
2. La lettre 'B' correspond à la valeur ASCII 01000010 = 66 (voir la table 1.2 page 38). On pourra chiffrer et déchiffrer cette valeur qui est bien dans l'intervalle $[0, 2772]$.
   - Chiffrement : on calcule $66^{17} \mod 2773$ c'est-à-dire 872.
   - Déchiffrement : on calcule $872^{157} \mod 2773$ et on vérifie qu'on obtient bien 66.

Exercice 3.13, page 171.

1. $M' = 100^{11} \pmod{319} = 265$
2. On doit résoudre $11 * d = 1 \pmod{280}$. On trouve $d = 51$
   - soit en utilisant l'algorithme d'Euclide étendu ;
   - soit en essayant "à la main" car $51 = (280/11) * 2$ donc il suffit de deux essais pour trouver ;
   - soit en utilisant Euler, $d = 11^{-1} \mod 280$ ce qui implique que $d = 11^{\varphi(280)-1} \mod 280 = 11^{\varphi(7\times5\times8)-1} \mod 280 = 11^{(6\times4\times4)-1} = 11^{95} \mod 280$. Soit $d = 11^{64+16+8+4+2+1} = 81.81.121.81.121.11 = 81.11 = 51 \mod 280$.
3. On doit calculer $133^{51} \pmod{319}$. Dans les notes on donne $133^{25} = 133 \pmod{319}$. Le résultat est $133 * 133 * 133 \pmod{319} = 12$.
4. Évidemment non pour les deux car les messages à chiffrer/déchiffrer doivent appartenir à $\mathbb{Z}_n$, c'est-à-dire $\mathbb{Z}_{319}$ dans ce cas.

Exercice 3.14, page 171.

1. Dans un long message, si des séquences d'au moins 3 lettres se répètent il est possible que cela ne soit pas du au hasard. Au contraire, il peut s'agir de mêmes mots de message qui se retrouvent codés par la même partie de la clef. Déterminer la taille de la clef peut alors se faire en calculant le pgcd des distances entre des séquences se répétant dans le message chiffré (on appelle cette analyse le test de Kasiski). Il faut ensuite regrouper les

caractères du chiffré par paquet de cette taille et appliquer une analyse de fréquence pour chaque position.

2. $C = M^5 \mod 35$. On calcule alors :
   $d = 5^{-1} \mod \varphi(7 \times 5) = 5^{-1} \mod 24 = 5$.
   On en déduit $M = 10^5 \mod 35$. Puisque $100 = -5 \mod 35$, on a :
   $10^5 = 25 \times 10 = 5 \times 50 = 5 \times 15 = 75 = 5 \mod 35$.
   Le mot clef comporte donc 5 caractères.

3. On calcule la clef secrète d'Oscar :
   $d = 7^{-1} \mod \varphi(65) = 7^{-1} \mod 48 = 7$.
   Ensuite : $K_1 = 48^7 \mod 65 = 48^2 \times 48^5 = 29 \times 3 = 87 = 22 \mod 65$
   Puisque $4^3 = 64 = -1 \mod 65$, on a :
   $K_2 = 4^7 \mod 65 = (-1)^2 \times 4 = 4 \mod 65$

4. On réordonne le texte par groupe de 5 caractères.

H → J	W → A	Q → M	I → A	O → I
Q → S	V → _	P → L	I → A	F → _
T → V	D → I	I → E	H → _	T → N
Y → _	_ → E	W → S	A → T	F → _
N → P	G → L	Y → U	_ → S	F → _
C → E	O → T	M → I	V → N	I → C
C → E	G → L	E → A	V → N	Z → T
C → E	V → _	I → E	A → T	F → _
J → L	D → I	F → B	Z → R	K → E
Y → _	L → Q	Y → U	H → _	G → A
Y → _	G → L	E → A	H → _	R → L
S → U	H → M	M → I	M → E	X → R
C → E	V → _	H → D	B → U	F → _
A → C	J → O	Y → U	K → C	N → H
Z → A	I → N	X → T	H → _	P → J
Z → A	H → M	E → A	Q → I	Y → S
Y → _	J → O	R → N	H → _	T → N
Y → _	W → A	M → I	U → M	K → E
Y → _	K → P	P → L	B → U	Y → S
Y → _	G → L	E → A	H → _	A → V
G → I	_ → E	D → _	Y → Q	_ → U
Y → _	W → A	D → _	T → L	F → _
M → O	H → M	F → B	Z → R	K → E
Y → _	Z → D	Y → U	H → _	X → R
C → E	I → N	S → O	V → N	I → C
C → E	H → M	I → E	V → N	Z→ T

On effectue une analyse fréquentielle colonne par colonne :

– Colonne 1 : 9 'Y' et 6 'C' : on suppose

$$\begin{cases} Y(24) \longleftarrow \_(26) \\ C(2) \longleftarrow E(4) \end{cases}$$

Chiffrement : décalage de $-2 = 25 \mod 27$ (Z)
Déchiffrement : décalage de $+2$
– Colonne 2 : on a déjà la clef de chiffrement : $K_1 = 22 = -5 \mod 27$. Pour déchiffrer, il convient donc de faire un décalage de $+5$.
– Colonne 3 : on a aussi la clef de chiffrement : $K_2 = 4 \mod 27$. Pour déchiffrer, il doit faire un décalage de $-4 = 23 \mod 27$.
– Colonne 4 : 7 'H' : on suppose $H(7) \longleftarrow \_(26)$.
Chiffrement : décalage de $+8$ (I)
Déchiffrement : décalage de $-8 = 19 \mod 27$
– Colonne 5 : 6 'F' on suppose $F(5) \longleftarrow \_(26)$.
Chiffrement : décalage de $+6$ (G)
Déchiffrement : décalage de $-6 = 21 \mod 27$

Le texte déchiffré est donc :

```
JAMAIS_LA_VIE_N_EST_PLUS_ETINCELANTE_ET_LIBRE_
QU_A_LA_LUMIERE_DU_COUCHANT_
JAMAIS_ON_N_AIME_PLUS_LA_VIE_
QU_A_L_OMBRE_DU_RENONCEMENT
```

On peut en déduire la clef de chiffrement :

$$K_0K_1K_2K_3K_4 = (25, 22, 4, 8, 6) = \text{"ZWEIG"}$$

La clef de déchiffrement est alors :

$$K_0^{-1}K_1^{-1}K_2^{-1}K_3^{-1}K_4^{-1} = (2, 5, 23, 19, 21) = \text{"CFXTV"}$$

Exercice 3.15, page 173.

1. Puisque $\varphi(n)$ divise $ed-1$ et que $\varphi(n) = (p-1)(q-1) = n-(p+q)+1$, il existe $k \in \mathbb{Z}$, tel que $ed - 1 = k \mod n - (p+q) + 1$.
2. On suppose p et q différents de 2 et 3 ; ainsi p et q sont inférieurs à $\frac{n}{4}$, et donc $pq - p - q + 1 \geqslant \frac{n}{2}$.
Finalement, k vérifie :
$$k \leqslant \frac{2e.d}{n}.$$
Comme $d < n$, on a finalement $k \leqslant 2e$.

3. Comme e est petit, on peut énumérer toutes les valeurs possibles entre 1 et $2e$ pour trouver le bon k. Dans ce cas, $n = pq$ et $S_k = n + 1 - \frac{ed-1}{k} = p + q$ sont solutions entières de l'équation du second degré : $X^2 - S_k + n = 0$.
   La méthode est détaillée dans l'algorithme 30.

Algorithme 30 Factorisation de n à partir de (n,e,d) dans RSA (cas e petit).

Entrées (n, e, d).
Sorties Les facteurs p et q tels que $n = pq$
  Pour $k \leftarrow 1$ à $2e$ **Faire**
    $S_k \leftarrow n + 1 - \frac{ed+1}{k}$
    Si S_k est entier **Alors**
      Calculer les 2 racines p et q de l'équation : $X^2 - S_k + n = 0$.
      Si p et q sont entiers **Alors**
        Renvoyer p et q.
      Fin Si
    Fin Si
  Fin Pour

Exercice 3.16, page 173.

1. On sait que $\varphi(n)$ divise $ed - 1$. Il existe $k \in \mathbb{Z}$, tel que $ed - 1 = k\varphi(n)$. En outre : $t = \frac{ed-1}{2^s}$. Soit alors $a \in \mathbb{Z}$ premier avec n.
   Alors $1 = a^{k\varphi(n)} = \left(a^t\right)^{2^s} \mod n$. Ainsi, l'ordre de a^t dans $\mathbb{Z}_n$ est forcément dans $\{2^j \ ; \ 0 \leqslant j \leqslant s\}$. Autrement dit :

$$\text{il existe } i \in [0, s] \ / \begin{cases} a^{2^{i-1}t} \neq \pm 1 \mod n \\ a^{2^i t} = 1 \mod n \end{cases}$$

   Posons $u = a^{2^{i-1}t}$. Nous avons donc trouvé u tel que

$$\begin{cases} u^2 - 1 = (u-1)(u+1) = 0 \mod n \\ (u-1) \neq 0 \mod n \\ (u+1) \neq 0 \mod n \end{cases}$$

   Ainsi $\text{pgcd}(u - 1, n) = \text{pgcd}(a^{2^{i-1}t} - 1, n)$ est un facteur non trivial de n. On obtient l'autre facteur directement par division.

2. L'espérance du nombre de tirages de a en essayant tous les i est 2 (l'algorithme est immédiat !) car si n est composé, au moins la moitié des inversibles ne vérifient pas la relation donnée .

Algorithme 31 Factorisation de n à partir de (n, e, d) dans RSA.

Entrées (n, e, d).
Sorties (Les facteurs p et q tels que $n = pq$) ou ERREUR.
  Soient s et t tels que $ed - 1 = t2^s$ (t impair)
  Soit a un nombre entier aléatoire entre 0 et $n - 1$.
  $\alpha \leftarrow a^t \mod n$
  Si $\alpha = 1$ ou $\alpha = n - 1$ **Alors**
    Renvoyer ERREUR ;
  Fin Si
  Pour $i \leftarrow 1$ à s **Faire**
    tmp $= \alpha^2 \mod n$
    Si $tmp = 1 \mod n$ **Alors**
      Renvoyer $p = \text{pgcd}(\alpha, n)$ et $q = \frac{n}{p}$
    Fin Si
    $\alpha \leftarrow$tmp
  Fin Pour
  Renvoyer ERREUR ;

La preuve est en deux parties : montrer qu'il existe au moins un a vérifiant ces conditions, puis montrer que dans ce cas il y en a au moins la moitié des inversibles. Nous commençons par la deuxième partie, plus facile.

Posons $m = 2^i t$. Supposons qu'il existe un a tel que $a^m = 1$ et $a^{m/2} \neq \pm 1$. Soit $\{b_1, \ldots, b_k\}$ l'ensemble des b_i tels que $b_i^{m/2} = \pm 1$ et $b_i^m = 1$. Alors, forcément $(b_i a)^{m/2} \neq \pm 1$. Donc, s'il existe un tel a, il en existe au moins $\frac{\varphi(n)}{2}$, soit un inversible sur deux.

Il reste à prouver qu'il en existe au moins un. Cela n'est pas toujours le cas pour un composé quelconque, mais nous allons voir que c'est le cas pour un entier n produit de deux nombres premiers distincts. Nous savons que t est impair, donc $(-1)^t = -1 \neq 1 \mod n$. Il existe un a (au pire une certaine puissance de -1) vérifiant $u = a^{m/2} \neq 1 \mod n$ et $a^m = 1 \mod n$. Par le théorème chinois $u^2 = 1 \mod p$ et $u^2 = 1 \mod q$. Ainsi, u doit être congru à 1 ou -1 modulo p et q. Il ne peut pas être égal à 1 modulo les deux nombres premiers sinon il le serait aussi modulo n. Donc, sans perte de généralité, nous supposons qu'il est congru à -1 modulo p. Si alors u est congru à 1 modulo q, nous avons trouvé notre élément.

Sinon, u est congru à -1 modulo n. Mais dans ce dernier cas, nous pouvons construire un autre élément qui peut convenir car alors $m/2$ ne divise ni $p - 1$, ni $q - 1$ (sinon u serait congru à 1 modulo p ou q).

En effet, nous avons $\frac{m}{2} = \frac{ed-1}{2^{s-i+1}} = \frac{k\varphi(n)}{2^{s-i+1}}$, et comme $m/2$ ne divise pas $p-1$, nous pouvons poser $\frac{m}{2} = v\frac{p-1}{2^j}$ avec v impair et j entre 0 et s. Puis, nous prenons g une racine primitive modulo p et nous posons $\alpha = g^{2^j} \mod p$ et $\beta = a \mod q$. Alors $\alpha^{m/2} = 1 \mod p$ par construction et $\beta^{m/2} = a^{m/2} = -1 \mod q$. Il ne reste plus qu'à reconstruire par restes chinois, γ correspondant aux restes respectifs α et β modulo p et q. Ce γ vérifie $\gamma^m = 1 \mod n$ ainsi que $\gamma^{m/2} \neq \pm 1 \mod n$. comme il en existe un, au moins un élément sur deux vérifie bien cette précédente propriété et le cassage de RSA se fait bien avec une espérance de nombre de tirages de 2.

Exercice 3.17, page 174.
Grâce au théorème chinois, Lucky Luke peut calculer facilement – en temps $\mathcal{O}(\log^2(n_A + n_J + n_W))$ – l'entier $C = x^3 = \Psi^{-1}(c_W, c_J, c_A) \mod n_W.n_J.n_A$ i.e. $C = x^3$ puisque $x < \min(n_W, n_J, n_A)$.
Comme x est entier, il suffit ensuite de calculer dans $\mathbb{R}$ $x = C^{1/3}$ par Newton-Raphson pour obtenir x en clair.

Exercice 3.18, page 174.
Oscar connaît (e_A, e_B, n) et (c_A, c_B). Par l'algorithme d'Euclide étendu, il calcule facilement (en temps -presque- linéaire du nombre de bits de n) les coefficients de Bézout r et s tels que $re_A + se_B = 1$.
Oscar peut alors calculer $c_A^r.c_B^s \mod n$ par exponentiation modulaire rapide et obtenir $c_A^r.c_B^s = M^{r.e_A+s.e_B} \mod n = M$.
Moralité : ne jamais utiliser le même n pour un groupe d'utilisateurs.

Exercice 3.19, page 174.
En effet, $u.r^{-1} = y^{d_A}.r^{-1} \mod n_A = c^{d_A}x^{d_A}r^{-1} \mod n_A$. Or $x^{d_A} = r^{e_A d_A} = r \mod n_A$ et donc $u.r^{-1} = c^{d_A}rr^{-1} \mod n_A = c^{d_A} \mod n_A = m$.
Le calcul de $u.r^{-1} \mod n_A$ peut être très facilement fait par Eve en temps $\mathcal{O}(\log^2 n_A)$ et donne le message m.
Moralité : ne jamais signer un message illisible (aléatoire ou chiffré) ; ou alors ne chiffrer que le résumé du message.

Exercice 3.20, page 174.

1. $\mu p_1 k = B!$, donc $p_1 \leqslant B$.
2. p premier, donc le théorème de Fermat donne $a^{p-1} = 1 \mod p$, le tout puissance μ.
3. $(A - a^{B!}) = kn = kpq$, donc $(A-1) - (a^{B!} - 1) = (A-1) - hp = kpq$.
4. $\log(k)\log^2(n)$.

5. $B\log(B)\log^2(n)$.
6. Si $A-1$ est un multiple de n, alors $B!$ est un multiple de $\varphi(n)=(p-1)(q-1)$, donc B est plus grand que tous les facteurs de $p-1$ et $q-1$ d'après la question 1. Or B est petit.
7. la puissance : $\alpha\log^3(n)\log(\alpha\log(n))$, le pgcd $\log^2(n)$.
8. Prendre p et q tels que $p-1$ et $q-1$ ont des gros facteurs (des nombres premiers robustes, voir page 103).

Exercice 3.21, page 177.

a. $h(x_1,x_2)=15^{x_1}22^{x_2} \mod 83$. Ainsi, $h(12,34)=15^{12}22^{34} \mod 83=63$. La résistance aux collisions repose sur la difficulté à trouver λ tel que $15^\lambda=22 \mod 83$.

b. $p-1=2q$ et q est premier. Les diviseurs de $p-1$ sont donc dans l'ensemble $\{1,2,q,2q=p-1\}$. Par définition, d divise $p-1$ donc $d\in\{1,2,q,p-1\}$.

c. $-q<y_2-x_2<q$ car $y_2,x_2\in\mathbb{Z}_q$. On en déduit que y_2-x_2 est premier avec q donc $d\neq q$.

d. $\alpha^{x_1}\beta^{x_2}=h(x)=h(y)=\alpha^{y_1}\beta^{y_2}$ implique que $\alpha^{x_1-y_1}=\beta^{y_2-x_2} \mod p$ car α et β sont inversibles.

e. Si $d=p-1=2q$ alors puisque d divise y_2-x_2, il existe $k\in\mathbb{Z}$ tel que

$$-q<y_2-x_2=2kq<q$$

On en déduit que $k=0$: $y_2=x_2$.
On en déduit que $\alpha^{x_1-y_1}=\beta^{y_2-x_2}=1 \mod p$. Or α est un générateur de $\mathbb{Z}_p^*$, $\alpha^u=1 \mod p \iff u=0 \mod p-1$: il existe $k\in\mathbb{Z}/x_1-y_1=k(p-1)$.
Or, de façon similaire à la question 3, on montre que

$$-(p-1)<-\frac{p-1}{2}\leqslant -q<x_1-y_1<q\leqslant\frac{p-1}{2}<p-1$$

Autrement dit, $k=0$: $x_1=y_1$. Finalement, $x=y$ ce qui est une contradiction.

f. Si $d=1$, on pose $u=(y_2-x_2)^{-1} \mod p-1$. Alors
$\exists k\in\mathbb{Z}/u(y_2-x_2)=1+k(p-1)$. On en déduit

$$\beta^{u(y_2-x_2)}=\beta^{1+k(p-1)}=\beta=\alpha^\lambda=\alpha^{u(x_1-y_1)} \mod p$$

et donc $\lambda=u(x_1-y_1) \mod p-1$.

g. 1. $u=(y_2-x_2)^{-1} \mod q$ alors il existe $k\in\mathbb{Z}/u(y_2-x_2)=1+kq$. On en déduit : $\beta^{u(y_2-x_2)}=\beta^{1+k.q}=\beta\left(\beta^q\right)^k$. Or $q=\frac{p-1}{2}$ et β est un générateur de $\mathbb{Z}_p^*$. Donc $\beta^{p-1}=1 \mod p$ et $\beta^{\frac{p-1}{2}}=\beta^q=-1 \mod p$. Finalement, $\beta^{u(y_2-x_2)}=(-1)^k\beta \mod p=\pm\beta \mod p$

2. On en déduit : $\beta^{u(y_2-x_2)} = \pm\beta = (-1)^\delta\alpha^\lambda = (\alpha^q)^\delta\alpha^\lambda \mod p$ avec $\delta \in \{0,1\}$. Autrement dit : $\alpha^{u(x_1-y_1)-q\delta} = \alpha^\lambda \mod p$.
   - Si $\delta = 0$: $\lambda = u(x_1-y_1) \mod p-1$
   - Si $\delta = 1$: $\lambda = u(x_1-y_1)-q = u(x_1-y_1)+q \mod p-1$ car $q = \frac{p-1}{2}$.

h. L'algorithme est simple :
- Calcul de $d = \text{pgcd}(y_2-x_2, p-1)$ par l'algorithme d'Euclide étendu
- Si $d = 1$, $u = (y_2-x_2)^{-1} \mod p-1$ et $\lambda = u(x_1-y_1) \mod p-1$
- Si $d = 2$, $u = (y_2-x_2)^{-1} \mod q$ et $\lambda = u(x_1-y_1) \mod p-1$. Si $\beta^\lambda = -\beta \mod p$ alors $\lambda = \lambda + q$.

Cet algorithme ne comporte que des opérations élémentaires ou peu coûteuses et le calcul de λ est rapide. On achève ainsi le raisonnement par l'absurde et on en déduit que h est résistante aux collisions.

Exercice 3.22, page 185.
Pour les deux premières constructions, un attaquant (qui ne dispose pourtant pas de la clef K) peut ajouter un nouveau bloc à la fin et obtenir un MAC valide. Pour la dernière construction, une collision sur H fournit un MAC valide. En effet, si $H(x) = H(x')$, alors $H(x||K) = h(H(x)||K) = h(H(x')||K) = H(x'||K)$.

Exercice 3.23, page 188.
$g^{z_1}y^{z_2} = g^{sv}g^{xz_2} = g^{xrv+k}g^{xz_2} = g^{xrv+k}g^{xqv-xrv} = g^k \mod p$.

Exercice 3.24, page 189.
L'idée est d'utiliser des générateurs pseudo-aléatoires simples pour fabriquer n, e et x_0 qui resteront secrets. La procédure utilisée est alors décrite dans l'algorithme 32, page 310.

Exercice 3.25, page 190.
Pour en déduire K, il faudrait que Oscar trouve a à partir de A ou b à partir de B, pour ensuite calculer soit A^b, soit B^a. C'est le problème du logarithme discret, vu en section 1.3.3.

Exercice 3.26, page 190.
- Alice calcule $A = 2^{292} \mod 541 = 69$ qu'elle transmet à Bob.
- Bob calcule $B = 2^{426} \mod 541 = 171$ qu'il transmet à Alice.
- La clef secrète est alors $(171)^{292} \mod 541 = 368 = (69)^{426} \mod 541$.

Exercice 3.27, page 196.

a. *Random*$_1$ garantit à Alice en retour d'étape 2 que la clef ne sera pas réutilisée.

Algorithme 32 Générateur pseudo-aléatoire cryptographique RSA.

Entrées Une taille $b > 1024$.
Entrées Un entier $k < \mathcal{O}(\log(b))$;
Sorties Une séquence de l bits pseudo-aléatoires cryptographiquement sûre.
Générer deux nombres premiers p et q robustes de taille au moins $\frac{b}{2}$ bits (voir page 103) ;
$n = pq$;
Générer un exposant pseudo-aléatoire e tel que $\text{pgcd}(e, (p-1)(q-1)) = 1$;
Générer un entier pseudo-aléatoire s (voir par exemple page 69) ;
$x_0 = s^e \mod n$;
Pour tout i de 1 à $\lceil \frac{l}{k} \rceil$ **Faire**
$x_i = x_{i-1}^e \mod n$;
z_i reçoit les k premiers bits de x_i ;
Fin Pour
Retourner la séquence $z_1||z_2||\ldots||z_l$;

b. $Random_2$ garantit à Bob que l'étape 3 n'est pas rejouée. Alice renvoie $Random_2 - 1$ car le renvoi crypté de $Random_2$ seul serait cette fois-ci identique à l'envoi de Bob.

c. Eve enregistre le message 3. Une fois qu'elle aura K, elle pourra renvoyer ce message et prétendre facilement être Alice en répondant en 5 à l'aide de K.

d. Il faut ajouter une datation (un ticket). C'est à partir de ce constat et de cette modification qu'est né Kerberos !

e. Le ticket Kerberos est valable durant une période de validité donnée (souvent 8 heures) durant laquelle l'utilisateur peut le présenter au TGS afin d'obtenir des tickets de service. Le problème est que Kerberos ne prévoit pas de système de révocation. Tant que la période de validité n'est pas expirée, il est donc possible d'utiliser une clef de session piratée.

f. Le vol des clefs, ou même des empreintes permet de se faire passer pour un client. Il faut donc utiliser un système à clef publique où seul le client possède une information privée.

Exercice 3.28, page 197.
Pour garantir le secret et l'authentification, il faut qu'un domaine commun existe entre A et B, ou plus précisément entre E_A, D_A, E_B et D_B.
A envoie alors $C = E_B(D_A(M))$ à B ; De son coté, B calcule $E_A(D_B(C)) = E_A(D_A(M)) = M$.
Un espion ne peut pas décoder C car il ne dispose pas de D_B et le secret est donc préservé. Si un espion envoie un message M' à la place de C, M' ne

pourra pas être décodé en un message valide M : en effet, pour cela il faudrait connaître D_A. Cela assure l'authenticité.

Exercice 3.29, page 197.

1. Pour le professeur : $\varphi(55) = 40$ et $27 \times 3 = 81 = 1 \mod 40$.
   Pour le secrétariat : $\varphi(33) = 20$ et $7 \times 3 = 21 = 1 \mod 20$.

2. Le professeur envoie $m = 12^3 \mod 33$. Or $12^2 = 12 \mod 33$; donc $m = 12 \mod 33$.
3. Le professeur a calculé $(x^{e_S} \mod n_S)^{d_P} \mod n_P$.
   Le secrétariat reçoit y et calcule $(y^{e_P} \mod n_P)^{d_S} \mod n_S$.
   D'où la note $(23^3 \mod 55)^7 \mod 33 = (12^7 \mod 33) = 12$.

Exercice 3.30, page 198.

1. L'authentification est unilatérale : la banque authentifie le client.
2. Un faux site web !
3. Il faut également que la banque s'authentifie. Cela est fait grâce aux certificats du site Web et aux PKI associées.

En France, un tel système de numéros jetables n'est intéressant que pour la banque : en effet, la véritable sécurité juridique des cartes à puces repose sur la possibilité de répudiation. C'est à la banque de prouver que tel achat a bien été effectué par le détenteur du compte. Le système à numéro jetable effectue un retournement de la preuve juridique puisqu'il est associé à un code secret reçu par courrier. Ainsi c'est à l'utilisateur de prouver qu'un numéro jetable associé à son compte en banque à été généré sans ce code secret.

Exercice 3.31, page 205.

1. `dd if=/dev/urandom of=demoCA/private/.rand bs=1k count=16`
2. `mkdir demoCA ; mkdir demoCA/private ; mkdir demoCA/certs ;`
   `mkdir demoCA/crl ; mkdir demoCA/newcerts ;`
   `echo ‘‘01’’ > demoCA/serial ; touch demoCA/index.txt ;`
   `chmod -R go-rwx demoCA`
   (a) `openssl genrsa -aes256 -out demoCA/private/cakey.pem`
   `-rand demoCA/private/.rand 4096`
   (b) `openssl req -new -x509 -key demoCA/private/cakey.pem`
   `-out ./demoCA/cacert.pem -days 60`
3. `openssl genrsa -aes256 -out luckyluke.key`
   `-rand demoCA/private/.rand 1024`
   `openssl req -new -key luckyluke.key`

```
-keyout newreq.pem -out newreq.pem -days 365
openssl ca -policy policy_anything
-out newcert.pem -infiles newreq.pem
```

4. `openssl pkcs12 -export` permet de convertir des certificats du format X509 au format pkcs12. La ligne de commande est donc :
   ```
   openssl pkcs12 -export -in cacert.pem -out cacert.p12
   -inkey private/cakey.pem
   ```
   (a) Pour envoyer des courriels chiffrés il faut récupérer la clef publique dans le certificat du destinataire. C'est donc toujours possible. Au contraire pour les messages chiffrés reçus il est impossible de les déchiffrer puisque la clef est perdue.

   (b) Pour signer des courriels il faut utiliser la clef privée perdue, c'est donc impossible. Au contraire, il est toujours possible de vérifier une signature puisque pour cela il faut récupérer la clef publique dans le certificat de l'envoyeur.

Exercice 3.32, page 207.

1. L'élément essentiel d'un certificat numérique est l'association d'une identification et d'une clef publique.
2. Cela permet de réaliser des authentifications sans intermédiaire (pourvu qu'une PKI sous-jacente ait été initiée) et sans risque d'attaque Man-in-the-middle. En effet, pour pouvoir se faire passer pour une des parties, un attaquant au milieu devrait avoir un faux certificat d'identité.
3. Une politique de certification dépend d'une politique de sécurité. Elle doit notamment définir la durée de validité des certificats, qui peut-être certifié, etc.

Exercice 3.33, page 209.

Comme vous avez totalement confiance en Alice, vous pouvez être sûr du certificat de Bob. Cependant, vous ne savez rien de la politique de certification de Bob, donc vous ne pouvez pas savoir si le message provient bien d'Oscar ou non.

Exercice 3.34, page 209.

1. Au sein d'une entreprise pas de problème ; entre clients et filiales, il faut de toutes façons faire confiance ; pour les entreprises spécialisées, pas de raison de faire confiance à une entité commerciale.
2. Dans tous les cas : stockage crypté sur disque protégé par une phrase.
3. C'est à différencier les Jean Martin que servent les autres informations d'identité d'un certificat : localité, adresse courriel, etc.

4. (a) En fonction de la durée nécessaire pour casser une clef, de la probabilité de perte/vol d'une clef privée en fonction de la durée et de l'usage.
   (b) Se renseigner sur les algorithmes et les attaques pour définir des clefs difficiles à casser.
   (c) Oui, cela permet de gérer les pertes/vol de clefs.
   (d) Une autorité doit décrire tout ceci dans sa politique de certification.

Exercice 3.35, page 211.

1. Une pour chaque couple de personnes : $\frac{n(n-1)}{2}$.
2. Un couple de clefs publique/privée par personne : n.
3. L'échange de clefs symétriques est délicat (nécessite par exemple un canal confidentiel préexistant) et une clef symétrique n'autorise pas directement la non-répudiation (puisque l'expéditeur et le destinataire possèdent la même clef). Cependant, les systèmes asymétriques restent plus lents.

Exercice 3.36, page 215.

1. Il faudrait utiliser une PKI par exemple.
2. `ssh-keygen -b 2048 -t dsa`
3. La connexion se fait maintenant non plus avec le mot de passe associé au login, mais avec la phrase de passe associée à la clef DSA.
4. Un message déclarant que le serveur sur lequel on se connecte est inconnu et il faudrait donc vérifier si la clef publique proposée est correcte :

```
The authenticity of host 'www.imag.fr' can't be established.
RSA\index{RSA} key fingerprint is
3a:fa:9e:88:7f:13:89:8a:63:ad:18:45:5b:24:68:d6.
Are you sure you want to continue connecting (yes/no)?
```

5. La clef envoyée par le serveur ne va pas correspondre à celle stockée localement. La connexion doit être interdite car des clefs différentes peuvent signifier qu'une attaque de type Man-in-the-middle par exemple est en train d'être essayée :

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@        WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!         @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)!\index{Man-in-the-middle}
It is also possible that the RSA\index{RSA} host key has just been
```

```
changed.
The fingerprint for the RSA\index{RSA} key sent by the remote host
is 13:bb:ae:c1:b2:df:19:2b:8f:ff:e1:b2:75:45:02:5c.
Please contact your system administrator.
Add correct key in .ssh/known_hosts to get rid of this message.
Offending key in .ssh/known_hosts:1
RSA\index{RSA} host key for www.imag.fr has changed
and you have requested strict checking.
Host key verification failed.
```

Solutions des exercices proposés au chapitre 4

Exercice 4.1, page 220.

1. La clef C est telle que $2630538516305 + C$ est multiple de 97 ; donc $C = 97 - 2630538516305 \mod 97 = 64$.
2. $R = \frac{13}{15} \simeq 86,67\%$.
3. Le code permet de détecter une erreur sur un seul chiffre. Soit $n = [K, C]$ l'entier de 15 chiffres associé au numéro de sécurité sociale suivi de la clef. Une erreur sur un seul chiffre c_i d'indice i donne $n' = n + e \cdot 10^i$ avec $e \neq 0$ et $-c_i \leqslant e \leqslant 9 - c_i$. On a alors $n' \mod 97 = e \cdot 10^i \neq 0$ car e et 10 sont premiers avec 97 ; l'erreur est détectée.
   Par contre, une erreur sur deux chiffres n'est pas détectée. Par exemple, deux erreurs de chiffres peuvent transformer le numéro de sécurité sociale valide 2.63.05.38.516.301.68 en le numéro 2.63.05.38.516.398.68 qui est aussi valide.

Exercice 4.2, page 220.

1. Le mot m est écrit en 3 blocs de 3 bits ; $\phi(m)$ contient 16 bits dont 7 bits de parité.
2. Si $\phi(m)$ contient un nombre impair d'erreurs, nécessairement il existe au moins une ligne ou une colonne de 4 bits comportant un nombre impair d'erreurs. Pour ces 4 bits, le quatrième bit est le bit de parité et sera nécessairement différent de la somme binaire des 3 autres bits.
3. Ce code détecte donc 1 et 3 erreurs. Si 2 erreurs surviennent, 2 bits de parité au moins ne seront pas vérifiés et les erreurs seront donc détectées. Ce code détecte donc jusqu'à 3 erreurs de bits.
4. Soit le mot de code $\phi(m) = \begin{bmatrix} 0000 \\ 0000 \\ 0000 \\ 0000 \end{bmatrix}$. Le mot $\begin{bmatrix} 010\mathbf{0} \\ 010\mathbf{0} \\ 010\mathbf{0} \\ \mathbf{0010} \end{bmatrix}$ comporte 4 erreurs par rapport à $\phi(m)$; on détecte que le mot est erroné en vérifiant les bits de parité en gras.
   Le mot $\begin{bmatrix} 1100 \\ 0000 \\ 0000 \\ 1100 \end{bmatrix}$ comporte 4 erreurs par rapport à $\phi(m)$ mais ces erreurs ne peuvent pas être détectées car tous les bits de contrôle sont corrects. Ici, les erreurs ont conduit à un mot de code correct.
5. Soit le mot reçu ne comporte pas d'erreur et on vérifie facilement que tous les bits de parité sont vérifiés. Soit le mot comporte une erreur,

et le bit de parité totale (qui doit correspondre à la somme de tous les autres bits) est nécessairement non vérifié. On distingue trois cas selon le nombre des autres bits de parité non vérifiés.
- Seul le bit de parité de tous les bits n'est pas vérifié : il suffit de le changer pour corriger l'erreur.
- Un autre bit de parité correspondant à la ligne i (ou à la colonne i) n'est pas vérifié : il suffit de changer ce bit de parité pour corriger l'erreur.
- Deux autres bits de parité ne sont pas vérifiés ; nécessairement, l'un correspond à une ligne i et l'autre à une colonne j. Il suffit de changer le bit de source d'indice (i, j) pour corriger l'erreur.

6. Le mot $\begin{bmatrix} \mathbf{1001} \\ 0000 \\ 0000 \\ 0000 \end{bmatrix}$ comporte 2 erreurs par rapport à $\phi(m)$. Seuls les deux bits de parité correspondant à la première et la quatrième colonne ne sont pas vérifiés. Même si l'on suppose qu'il y a eu deux erreurs, on ne peut pas décider sur laquelle des quatre lignes ont eu lieu les erreurs. On ne peut donc pas le corriger sans ambiguïté : la première ligne pourrait par exemple être 0000, ou alors la deuxième, la troisième ou encore la quatrième ligne pourraient chacune être 1001, ou les trois ensemble, etc.

Exercice 4.3, page 223.
Comme le canal est symétrique, $H(Y|X) = H(Y|X = 0) = H(Y|X = 1) = -p\log_2(p) - (1-p)\log_2(1-p)$. Soit p_1 la probabilité d'avoir le bit 0, respectivement $p_2 = 1 - p_1$ la probabilité d'avoir le bit 1, en entrée. Alors la probabilité d'avoir le bit 0 en sortie est $y_1 = p_1 * (1-p) + (1-p_1) * p$, et d'avoir le bit 1 est $y_2 = (1-p_1) * (1-p) + p_1 * p$. Comme $H(Y|X)$ est indépendant de p_1, le maximum correspondra au maximum de $H(Y)$. Or $H(Y)$ est maximale quand les probabilités y_1 et y_2 sont égales, soit quand $y_1 = y_2 = 1/2$ (ce qui arrive par exemple quand $p_1 = 1/2$). On peut alors calculer $H_{max}(Y) = 1$, ce qui donne la capacité du canal $\mathcal{C}_{BSC} = 1 + p\log_2(p) + (1-p)\log_2(1-p)$.

Exercice 4.4, page 224.
Si $p > \frac{1}{2}$, il suffit dès la réception en préambule au décodage de transformer tout bit reçu en son opposé. Cela revient à transformer le canal BSC de probabilité p en un canal BSC de probabilité $q = 1 - p$ avec $q < \frac{1}{2}$.

Exercice 4.5, page 227.
Le nombre de codes différents correspond au choix de 3 séparateurs dans un intervalle discret de 1 à 7, soit 3 parmi 6, le tout multiplié par les deux possibilités : commencer par blanc ou commencer par noir. Au total il y a bien $2 * C_3^6 = 40$ codes différents possibles par colonne, 4 par chiffre.

Exercice 4.6, page 228.
On obtient $c_1 = 11 - \sum_{i=2}^{10} c_i = 11 - (2*2 + 9*3 + 6*4 + 0*5 + 5*6 + *7 + 0*8 + 1*9 + 2*10 \mod 11) = 7$, soit le code ISBN 210-050-692-7.
Le code EAN-13 est 9-782100-50692c_0 avec $c_0 = (10 - (9 + 8 + 7 + 6 + 1 + 6 + 3*(7 + 2 + 4 + 2 + 2 + 0)) \mod 10 = 7$, soit 9-782100-506927, comme vous pouvez le vérifier sur la couverture. Dans le cas particulier de ce livre, les chiffres de parité du code ISBN et du code à barres sont les mêmes !

Exercice 4.7, page 230.
Toute erreur non détectée est associée à un polynôme P_e multiple de P_g. Si l'erreur porte sur un seul bit i, $P_e = X^i$. Si P_g a au moins deux coefficients, il ne peut pas diviser X^i donc l'erreur est détectée.
Réciproquement si P_g a un seul monôme $P_g = X^r$, il ne détecte pas l'erreur $P_e = X^{r+1}$.

Exercice 4.8, page 230.
Comme $g_r = 1$, si $\sum_{i=0}^{r} g_i = 0$, 1 est racine de P_g donc P_g est multiple de $X + 1$. Toute erreur non détectée est associée à un multiple de $X + 1$. Or tout multiple de $X + 1$ dans $\mathbb{F}_2[X]$ a un nombre pair de coefficients non nuls. Toute erreur non détectée doit donc nécessairement porter sur un nombre pair de bits. Donc le code détecte toute erreur portant sur un nombre impair de bits.

Exercice 4.9, page 230.
Pour un paquet d'erreurs, $P_e = X^k + \ldots + X^i = X^k(1 + \ldots + X^{i-k})$. Si $(i - k) < d$ alors le facteur irréductible de degré d ne pourra pas diviser P_e, et donc *a fortiori* P_g ne pourra pas non plus diviser P_e.

Exercice 4.10, page 232.

1. En effet pour tous x, y et z dans V^n on a :
   - $d_H(x, y) \in \mathbb{R}^+$;
   - $d_H(x, y) = 0 \Leftrightarrow x = y$;
   - $d_H(x, y) = d_H(y, x)$;
   - $d_H(x, y) \leqslant d_H(x, z) + d_H(y, z)$.
2. Le bit i de $x \oplus y$ vaut 1 si et seulement si $x_i \neq y_i$, donc $d_H(x, y) = w(x \oplus y)$.

Exercice 4.11, page 233.
Si on reçoit un mot m' avec s erreurs par rapport au mot m émis ; l'erreur sera détectée si et seulement si $m' \notin C$. Or, comme C est t-correcteur, on a, pour tout $x \in C$, $d(m, x) \geqslant 2t + 1$. Donc il n'existe pas de mot de C à une distance $\leqslant 2t$. Ainsi, si $d(m', m) = s \leqslant 2t$ alors $m' \notin C$ et on détecte l'erreur. Donc C est s-détecteur avec $s \geqslant 2t$.
La réciproque est vraie : En effet, si C est s-détecteur et que $s \geqslant 2t$, alors on a $\delta(C) > s$, donc $\delta(C) \geqslant 2t + 1$ et par conséquent C est au moins t-correcteur.

Exercice 4.12, page 233.

1. Deux mots distincts ont au moins un bit de source de distinct ; comme ce bit est répété m fois, $\delta(C) \geqslant m$. De plus soient $\omega_0 = C(0^k) = 0^{mk}$ et $\omega_1 = C(10^{k-1}) = (10^{k-1})^m$; $d_H(\omega_0, \omega_1) = m$, donc $\delta(C) \leqslant m$. Finalement, $\delta(C) = m$.
2. Il faut $\delta(C) \geqslant 2 * 2 + 1 = 5$. Donc $(5, 1)$ convient. $(5k, k)$ aussi.
3. $(5k, k)$ est de rendement $\frac{k}{5k} = 0.2$.
4. La distance est 5 ; l'algorithme 33 permet de détecter 4 erreurs au moins (voire plus si elles concernent différents chiffres de source).

Algorithme 33 Décodage du code de répétition avec détection maximale.

```
erreur ← false
Pour i ← 0 à k − 1 Faire
  s[i] ← lirebit { Initialisation de s}
Fin Pour
Pour i ← k à n − 1 Faire
  Si lirebit ≠ s[i  mod k]  Alors
    erreur ← true
  Fin Si
Fin Pour
```

5. La distance est 5 ; l'algorithme 34 permet de corriger 2 erreurs ; il suffit de calculer pour chaque symbole de source le nombre de 1 associé. Si on a une majorité de 1 (au moins 3), on déduit que le symbole source était 1 ; sinon la source était 0. Ainsi on corrige au plus 2 erreurs pour les 5 bits émis correspondant à un même bit de source.

6. L'algorithme 35 corrige une erreur ; il peut détecter en outre 2 ou 3 erreurs sans les corriger (mais pas 4 !). La distance est 5 ; si on reçoit 4 symboles identiques (4 '0' ou 4 '1'), on suppose qu'il n'y a eu qu'une erreur et on fait la correction. S'il y a moins de 4 symboles identiques on renvoie un signal d'erreur. On corrige toutes les erreurs simples et on détecte les cas de 2 et 3 erreurs correspondant à un même bit de source.

Exercice 4.13, page 234.

On peut calculer la distance minimale entre deux mots de code quelconques. L'algorithme 36, page 320, nécessite $O(\frac{|V|^k \cdot (|V|^k+1)}{2})$ comparaisons. En les énumérant, il y a $|V|^k$ mots qui sont codés avec $n = k+r$ chiffres. Le coût est donc $O(n \cdot |V|^{2k})$, qui est impraticable pour $|V| = 2$ dès que k et n sont plus grands que 15. On suppose donc les mots de C stockés dans un tableau $C[1..|V|^k]$.

Algorithme 34 Décodage du code de répétition avec correction maximale.

```
Pour i ← 0 à k − 1 Faire
  NbrUn[i] ← 0 { Initialisation }
Fin Pour
Pour i ← 0 à n − 1 Faire
  Si lirebit = 1 Alors
    Incrémenter NbrUn[i mod k]
  Fin Si
Fin Pour
Pour i ← 0 à k − 1 Faire
  Si NbrUn[i] ⩾ 3 Alors
    s[i] ← 1
  Sinon
    s[i] ← 0
  Fin Si
Fin Pour
```

Algorithme 35 Décodage du code de répétition avec détection et correction.

```
erreur ← false
Pour i ← 0 à k − 1 Faire
  NbrUn[i] ← 0{ Initialisation }
Fin Pour
Pour i ← 0 à n − 1 Faire
  Si lirebit() = 1 Alors
    Incrémenter NbrUn[i mod k]
  Fin Si
Fin Pour
Pour i ← 0 à k − 1 Faire
  Si NbrUn[i] ⩾ 4 Alors
    s[i] ← 1
  Sinon, si NbrUn[i] ⩽ 1 Alors
    s[i] ← 0
  Sinon
    erreur ← true
  Fin Si
Fin Pour
```

Algorithme 36 Distance d'un code ($|V| = 2$).

```
delta ← +infini
Pour i ← 1 à 2^k Faire
  Pour j ← i + 1 à 2^k Faire
    d_ij ← 0 { calcul de la distance d_ij entre C[i] et C[j] }
    Pour l ← 1 à k + r Faire
      Si C[i][l] ≠ C[j][l] Alors
        d_ij ← d_ij + 1
      Fin Si
    Fin Pour{Mise à de jour de δ = min(d_ij) }
    Si ( d_ij < δ ) Alors
      δ ← d_ij
    Fin Si
  Fin Pour
Fin Pour
TauxDeDetection ← δ − 1 ;
TauxDeCorrection ← (δ−1)/2 ;
```

Chaque mot $C[i]$ est un tableau $[1..k+r]$ de chiffres de V.
Application : $k = log_2(|C|) = 2$; donc $r = 10 - 2 = 8$. Le code C est un code $(10, 2)$. On a $\delta = 5$: C est donc 4-détecteur et 2-correcteur.

Exercice 4.14, page 235.
Soit $x \in C'$: on a alors $\oplus_{i=1}^{n+1} x_i = 0$. Donc tous les mots de C' sont de poids de Hamming pair. Par suite, la distance d' entre deux mots de C' est paire. Comme $d \geqslant d' \geqslant d+1$ et d impaire, on en déduit $d' = d+1$.

Exercice 4.15, page 236.
Comme V est de cardinal au moins 2, il possède deux symboles distincts $a \neq b$. À partir de C, on construit le code C' de longueur $n+1$:

$$C' = \{c_1 \dots c_n a | c_1 \dots c_n \in C\} \cup \{aa \dots ab, ba \dots ab\}.$$

C' est de cardinal $M+2$ et de distance 1 ; de plus C est bien le code raccourci de C' avec les mots de C' se terminant par a.

Exercice 4.16, page 236.

1. Il suffit de calculer $\delta(C)$. Soit $x \neq y$ deux mots de C. Il existe i tel que $0 \leqslant i \leqslant 3$ et $x_i \neq y_i$. Soit e le nombre de ces indices i où les chiffres de x et y diffèrent.

- si $e = 1$ alors le bit en position i apparaît dans trois autres chiffres de x et y, à savoir les bits $(b_i + b_k, b_i + b_l, b_0 + b_1 + b_2 + b_3)$. Donc $d(x, y) \geqslant 4$.
- si $e = 2$ alors x et y ont exactement 2 bits en position i et j $(0 \leqslant i < j \leqslant 3)$ différents. Alors, comme chaque bit b_i apparaît dans deux chiffres de contrôle de la forme $b_i + b_l$ et $b_i + b_m$ et idem pour j, x et y différents au moins pour deux de ces bits. Ainsi $d(x, y) \geqslant 4$.
- Si $e = 3$ alors x et y ont 3 bits différents. Donc x et y diffèrent aussi pour leur dernier bit $(b_0 + b_1 + b_2 + b_3)$. Ainsi $d(x, y) \geqslant 4$.
- Si $e = 4$ alors $d(x, y) \geqslant 4$.

Ainsi $\delta(C) \geqslant 4$. Les deux mots $a = 000000000$ et $b = 110000110$ sont dans C et ont une distance de 4 : finalement $\delta = 4$.

2. En reprenant les deux mots précédents : soit $c = 100000010$. $d(a, c) = d(b, c) = 2$ et aucun mot de code n'est à une distance 1 de c. Donc l'erreur n'est pas corrigible.
3. $\delta = 4 \Longrightarrow C$ est 3-détecteur $(\delta - 1 = 3)$.
   a est obtenu à partir b en changeant 4 bits : donc C n'est pas 4-détecteur.
4. Le code poinçonné est de longueur 8 et de cardinal 16. Comme C est de distance 4, il est de distance $\geqslant 3$. Il contient en particulier les mots $a' = 00000000$ et $b' = 10000110$ poinçonnés de a et b qui sont à distance 3. Sa distance est donc 3.
5. On considère que le code est raccourci en prenant les mots de C qui ont un 0 en première position, i.e. $b_0 = 0$ (le cas $b_0 = 1$ est symétrique). Le code raccourci C' est alors un code $(8, 3)$ de cardinal 8. Sa distance est au moins celle de C, soit 4. Les mots $a = 000000000$ et $c = 001100110$ sont dans C ; donc les mots $a' = 00000000$ et $c' = 01100110$ sont dans C' et à distance 4. Le code raccourci est donc de distance 4.

Exercice 4.17, page 237.
Considérons un code avec r bits de redondance, *i.e.* $(n, 4)$ avec $n = 4 + r$. Pour avoir un code 1-correcteur il est nécessaire que $1 + n \leqslant 2^{n-4}$ (théorème 25), donc $5 \leqslant 2^r - r$. Le nombre minimal r assurant cette condition est $r = 3$. En outre, pour $r = 3$ on a égalité donc un code 1-correcteur $(7, 4)$ est parfait.

Exercice 4.18, page 237.

1. L'inégalité du théorème 25 avec $t = 1$ s'écrit : $1+n \leqslant 2^{n-k}$, soit $1+k+r \leqslant 2^r$.
2. Le rendement du code est $\frac{k}{r+k}$; donc maximiser le rendement à r fixé revient à maximiser k. L'inégalité du théorème 25 permet alors d'avoir une borne sur le plus grand k possible et donc une borne pour le rendement R.

Avec $r = 3$, on a nécessairement $k \leqslant 4$. D'où $R \leqslant \frac{4}{7}$.
Avec $r = 4 : k \leqslant 11$ et $R \leqslant \frac{11}{15}$.
Avec $r = 5 : k \leqslant 26$ et $R \leqslant \frac{26}{31}$.
Avec $r = 6 : k \leqslant 57$ et $R \leqslant \frac{57}{63}$.

3. La longueur d'un code binaire 1-parfait vérifie d'après le théorème 25 (calcul analogue au a.) : $n = 2^r - 1$. Il n'existe donc pas de code 1-parfait de longueur 2^m.

Exercice 4.19, page 239.

1. $\rho = \frac{k}{n} = \frac{11}{15}$.
2. Le code est : $\begin{array}{rcl} b_1 & = & b_3 + b_5 + b_7 + b_9 + b_{11} + b_{13} + b_{15} \\ b_2 & = & b_3 + b_6 + b_7 + b_{10} + b_{11} + b_{14} + b_{15} \\ b_4 & = & b_5 + b_6 + b_7 + b_{12} + b_{13} + b_{14} + b_{15} \\ b_8 & = & b_9 + b_{10} + b_{11} + b_{12} + b_{13} + b_{14} + b_{15} \end{array}$.
   Pour le mettre sous forme canonique, on fait la permutation : $s_1 = b_3$; $s_2 = b_5$; $s_3 = b_6$; $s_4 = b_7$; $s_5 = b_9$; $s_6 = b_{10}$; $s_7 = b_{11}$; $s_8 = b_{12}$; $s_9 = b_{13}$; $s_{10} = b_{14}$; $s_{11} = b_{15}$; $s_{12} = b_1$; $s_{13} = b_2$; $s_{14} = b_4$; $s_{15} = b_8$; d'où la matrice R correspondant au codage ϕ_C :
   $$R = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$
3. On calcule les 4 derniers bits de parité $[s_{12} ; s_{13} ; s_{14} ; s_{15}] = [s_1 ; \ldots ; s_{11}].R$. Le mot codé est alors : $[s_1 ; \ldots ; s_{11} ; s_{12} ; s_{13} ; s_{14} ; s_{15}]$.
4. Soit $[y_1 ; \ldots ; y_{15}]$ le mot reçu ; on calcule $\sigma = [y_1 ; \ldots ; y_{11}].R - [y_{12} ; y_{13} ; y_{14} ; y_{15}]$. On détecte une erreur si et seulement si $\sigma \neq 0$.
5. On calcule comme à la question précédente : $\sigma = [\sigma_1 ; \sigma_2 ; \sigma_3 ; \sigma_4]$. Si $\sigma \neq 0$, soit $j = \sum_{i=1}^{4} 2^i.\sigma_i$. On corrige alors l'erreur en changeant le $j^{\text{ième}}$ bit y_j de y.

Exercice 4.20, page 242.

Soit n la longueur de C. À partir de la donnée de k positions $1 \leqslant i_1 < \ldots < i_k \leqslant n$ quelconques, on construit un code(k, k) poinçonné de C' en supprimant dans tous les mots de code les $n - k$ autres positions. Soit ϕ' la restriction de

ϕ obtenue en ne gardant que les k positions $i_1, \ldots, i_k$; ϕ' est la fonction de codage associée à C'.
En tant que code poinçonné d'un code MDS, C' est MDS (théorème 29). Sa distance est donc $d' = k - k + 1 = 1$. Pour tout $x \neq y \in \mathbb{F}^k$, on a donc $\phi'(x) \neq \phi'(y)$. Par suite, il existe un unique $x \in \mathbb{F}^k$ tel que $\phi'(x)$ coïncide avec c aux k positions $i_1, \ldots, i_k$ choisies.

Exercice 4.21, page 242.
Soit C' le code raccourci $(n-1, k')$ obtenu en prenant dans C tous les mots ayant le symbole s en position i et en supprimant la position i ; on a $k' \geqslant k-1$. D'après 4.3.2, C' est de distance $d' \geqslant d = n - k + 1$; d'où d'après la borne de Singleton $k' \leqslant n - d' \leqslant n - d = k - 1$. On en déduit $k' = k - 1$ et le code raccourci est un code $(n-1, k-1)$. Comme sa distance d' atteint la borne de Singleton $n' - k' + 1 = n - k + 1$, il est MDS.

Exercice 4.22, page 245.
L'orthogonal à un sous-espace vectoriel de dimension k est un sous espace vectoriel de dimension $n - k$. Donc $C^\perp$ est un code linéaire de longueur n et de dimension $n-k$. De plus, d'après la propriété 17, les $(n-k)$ vecteurs lignes de la matrice de contrôle H de C sont orthogonaux à C et indépendants. On en déduit qu'ils forment une base de $C^\perp$, donc que H est génératrice de $C^\perp$.

Exercice 4.23, page 245.

1. Code (12,6,6). On remarque que R est symétrique.
2. On vérifie que si r et s sont deux lignes quelconques de G, alors $r.s = 0$; donc $\mathcal{G}_{12} \subset \mathcal{G}_{12}^\perp$.
   Or $\mathcal{G}_{12}$ et $\mathcal{G}_{12}^\perp$ ont la même dimension (12,6) ; ils sont donc égaux.
3. Il ne peut pas être parfait car il est de distance paire (= 6).
   (En effet : soit x et y 2 mots de code situés à une distance 6 ; on construit facilement un mot de $\mathbb{F}_3^{12}$ à une distance 3 de x et y, donc impossible à corriger de manière unique).
4. $G_{11} = [I_6 | R_5]$ avec
$$R_5 = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 2 & 2 \\ 1 & 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 0 & 1 \\ 1 & 2 & 2 & 1 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{bmatrix}$$
   Comme $\mathcal{G}_{12}$ est de distance 6, $\mathcal{G}_{11}$ est de distance au moins 5. La première ligne est de poids 5 (car la ligne 1ère ligne de R est de poids 4). Or toutes

les lignes sont des éléments du code. On en déduit que le code possède un mot de poids 5. Il est donc de distance 5. Le code est donc (11,6,5).

5. Comme $\mathcal{G}_{11}$ est de distance 5, il est 2-correcteur. Ainsi, les boules de rayon 2 centrées sur les mots de $\mathcal{G}_{11}$ sont disjointes et forment un sous-recouvrement de $\mathbb{F}_3^{11}$, de cardinal $3^6(1 + 2.C_{11}^1 + 2^2.C_{11}^2) = 177147$.
Or $\text{card}(\mathbb{F}_3^{11}) = 3^{11} = 177147$; le sous-recouvrement est en fait un recouvrement et le code est donc parfait.

Exercice 4.24, page 248.

1. Il suffit de récupérer les pgcd différents de 1 dans le calcul du test d'irréductibilité.
2. Si P est réductible, $k \leqslant \frac{d}{2}$. FDD est de donc complexité $\frac{d}{2}(log(p)M_d+E_d)$.
3. Si $p = 2$, les seuls facteurs possibles de P_1 sont X et $X - 1$.
4. $X^{p^i} = X \mod (X^{p^i} - X)$. Donc cette propriété s'étend à tout polynôme T par $T^{p^i} - T = \sum a_j(X^{jp^i} - X^j) = 0 \mod (X^{p^i} - X)$.
5. $T^{p^i} - T = T(T^{\frac{p^i-1}{2}} - 1)(T^{\frac{p^i-1}{2}} + 1)$.
6. On choisit T au hasard de degré $< 2i$ et on calcule $\text{pgcd}(T, P_i) \mod P_i$, puis $\text{pgcd}(T^{\frac{p^i-1}{2}} - 1 \mod P_i, P_i)$ où les puissances sont calculés par exponentiation rapide par carrés modulo P_i à chaque étape. De complexité $2E_d + i\log_2(p)M_d$. En moyenne, une chance sur deux de séparer les facteurs donne une espérance du nombre de tirages aléatoires de T de 2.

Exercice 4.25, page 250.

La donnée des $r = n - k$ coefficients de g définit une matrice G_C unique et donc un code linéaire unique. Or g étant un diviseur de $X^n - 1$, C est stable par σ donc cyclique.

Exercice 4.26, page 252.

1. Les racines de $(X^8 - 1)$ ne conviennent pas car elles sont au plus seulement $8^{\text{ième}}$. Essayons alors γ, racine de $X^4 + 2$, donc telle que $\gamma^4 = 2$. Les seuls ordres possibles sont 16, 8, 4 et 2. Si $\gamma^2 = 1$ alors $\gamma^4 = 1$ ce qui est impossible puisque $\gamma^4 = 2$. γ^4 n'est pas non plus égal à 1. $\gamma^8 = (\gamma^4)^2 = 2^2 = 4 = -1$ est également différent de 1, donc γ est forcément une racine primitive $16^{\text{ième}}$ de l'unité.
2. Regardons de quel polynôme les puissances successives de γ sont des racines :

γ	γ^2	γ^3	γ^4	γ^5	γ^6	γ^7	γ^8
X^4+2	X^2+2	X^4+2	$X+2$	X^4+3	X^2+3	X^4+2	$X+1$
γ^9	γ^{10}	γ^{11}	γ^{12}	γ^{13}	γ^{14}	γ^{15}	$1 = \gamma^{16}$
X^4+2	X^2+2	X^4+2	$X+3$	X^4+3	X^2+3	X^4+2	$X+4$

Pour être 2-correcteur, le degré minimal du polynôme générateur est donc égale à 7. Par exemple $g(X) = (X+2)(X^4+3)(X^2+3)$ possède γ^4, γ^5, γ^6 et γ^7 comme racines, ce qui donne un rendement de $\frac{9}{16} = 0,5625$.

Exercice 4.27, page 257.

1. Soit $x \in C$; après $d-1$ effacements on obtient y avec le caractère '?' en $d-1$ positions. Comme C est de distance d, il existe au moins un mot à distance $d-1$ de y (le mot x).
   Supposons qu'il existe deux mots de code $x_1 \neq x_2$ à distance $d-1$ de y. Alors x_1 et x_2 sont à distance au plus $d-1$ l'un de l'autre car il ne peuvent différer qu'aux positions des effacements. Or ceci est impossible car le code étant de distance d, $d_H(x_1, x_2) \geqslant d$.
   Conclusion : x est l'unique mot de C égal à y aux positions non effacées dans y. Ainsi, x est la seule correction possible : le code permet de corriger jusqu'à $d-1$ effacements.

2. La probabilité qu'un bit ne soit pas effacé est $p = 0.99$. Dans ce cas, si on travaille dans le corps $\mathbb{F}_{2^8}$, la probabilité qu'un chiffre ne soit pas effacé est $p^8 = 0.99^8 > 0.92$; avec 1% de bits effacés, on a donc entre 7% et 8% d'octets effacés.
   La longueur des mots d'un code de Reed-Solomon sur $\mathbb{F}_{256}$ est $n = 255$. Il faut pouvoir corriger $0.08 \times 255 = 20.40 \leqslant 21$ effacements par mot, donc un code de distance 22 : on choisit donc $r = 21$. Soit α une racine primitive de $\mathbb{F}_{256}$; on prend par exemple $g(X) = \prod_{i=1}^{2} 1(X - \alpha^i)$. On a un code $(255, 234, 22)$ sur $\mathbb{F}_{256}$.

3. Dans le mot reçu, on remplace les $s \leqslant d-1$ effacements par s inconnues x_i ; soit x le vecteur ligne obtenu qui doit appartenir au code. Donc, si H est la matrice de contrôle du code, on a $H.x^t = 0$ ce qui s'écrit comme un système linéaire de dimension s à s inconnues, les x_i. La résolution de ce système donne la valeur de x.

4. Soit x un mot reçu qui comporte s effacements et soit x' le mot de longueur $n-s$ obtenu en supprimant dans x les s effacements.
   Soit C' le code poinçonné de C obtenu en supprimant les s positions des effacements. Le code C' est un code linéaire de longueur $(n-s)$ et de distance $d' \geqslant d - s = 2t + 1$. Il permet donc de corriger t erreurs dans le mot x'. Soit $y' \in C'$ le mot corrigé, et y le mot de longueur n obtenu à partir de y en remettant les effacements. En utilisant la question précédente, on corrige les s effacements dans y avec la matrice de contrôle de C. On obtient ainsi l'unique mot de code à distance $s + t$ de x qui permet de le corriger.

Exercice 4.28, page 258.
Soit $w = 0 \dots 0 e_1 \dots e_l 0 \dots 0 \in \mathbb{F}_q^n$ un paquet d'erreur de longueur au plus l. Comme C détecte tout paquet de longueur au plus l, le mot $0 + w$ est détecté erroné. Donc $w \notin C$. On a dim(C)=k (voir par exemple la preuve de la borne de Singleton) et l'ensemble des w est de dimension l. Si ces deux ensembles sont totalement disjoints, sauf pour le mot 0, on a forcément $k + l <$dim$(\mathbb{F}_q^n)$, soit $l \leqslant n - k$.

Exercice 4.29, page 258.
Soit $t = \left\lfloor \frac{d-1}{2} \right\rfloor$; le code étant t-correcteur, il code corrige donc tout paquet qui s'étale sur au plus t octets consécutifs. Or, une erreur sur $8.(t-1)+2$ bits consécutifs peut s'étaler sur $t+1$ octets consécutifs (tous les bits faux sur les $t-1$ octets du milieu et 1 seul bit faux sur les octets extrêmes) et donc ne pas être corrigée. Alors que tout paquet de taille $8.(t-1)+1$ tient forcément sur t octets et sera donc corrigé. D'où $l = 8.\left(\left\lfloor \frac{d-1}{2} \right\rfloor - 1\right) + 1$.

Exercice 4.30, page 259.
Tout mot de C_p est construit à partir de p mots de C et peut être vu comme un p-uplet de mots de C qui est entrelacé . Ainsi, soit deux mots de C_p qui diffèrent ; les deux p-uplets diffèrent au moins en un mot de C donc sont à distance au moins d. Montrons que c'est la distance minimale : il existe deux mots u et v de C de distance d. Soit c un autre mot de C. Soient les deux mots u' et v' égaux à l'entrelacement des deux p-uplets respectifs $(u, c, \dots, c)$ et $(v, c, \dots, c)$; u' et v' diffèrent en d positions exactement ; donc C_p est de distance d.
On a clairement $l \leqslant d$. Soit $v \in C_p$ et e un paquet d'erreur de longueur $p*l$. Le désentrelacé de $v+e$ est un p-uplet de la forme $v_1 + e_1, \dots, v_p + e_p$ avec $v_i \in C$ et e_i paquet d'erreur de longueur au plus l. Comme C corrige les paquets de longueur l, il permet de corriger pour retrouver le p-uplet $(v_1, \dots, v_p)$, qui est associé au mot v C_p. Donc C_p corrige au moins les paquets de longueur pl.

Exercice 4.31, page 264.
Soit α une racine primitive de $\mathbb{F}_{256}$ et $g(X) = \prod_{i=1}^{4}(X - \alpha^i)$ par exemple. Ce polynôme définit un code C de Reed-Solomon (255,251) sur $\mathbb{F}_{256}$, de distance 5. La matrice génératrice de C associée à g a ses k premières colonnes indépendantes. Donc C admet une matrice génératrice normalisée $G = [I_{251}, R_{251\times 4}]$.
Soient R_1 la matrice formée des 24 dernières lignes de R et $R_2 = \left[\frac{T}{R_1}\right]$ celle formée par les 28 dernières lignes.
Soit $G_1 = [I_{24}, R_1]$ (resp. $G_2 = [I_{28}, R_2]$) la sous-matrice 24×28 (resp. 28×32) formée des 24 (resp. 28) dernières lignes et 28 (resp. 32) dernières colonnes.
La matrice G_1 est génératrice du code raccourci C_1 de C, formé à partir des

mots de C dont les 227 premiers chiffres sont nuls. C_1 est donc de distance d_1 supérieure ou égale à C, donc $d_1 \geqslant 5$. De plus, d'après la borne de Singleton, $d_1 \leqslant 255 - 251 + 1 \leqslant 5$. Finalement C_1 est de distance 5.

Exercice 4.32, page 264.

1. Le syndrome d'erreur est $s_2(z) = z.H_2^t$ où H_2 est la matrice de contrôle de C_2.
   - Si $s_2(z) = 0$, il n'y a pas d'erreur : on retourne les 28 premiers octets de z.
   - Si $s_2(z)$ est colinéaire à l'une des lignes de H_2^t : soit $i \in \{1, 32\}$ et $\lambda \in \mathbb{F}_{256}$ tels que $s_2(z) = \lambda.e_i.H_2^t = s_2(\lambda.e_i)$, où e_i est le $i^{\text{ième}}$vecteur ligne canonique. La correction $-\lambda.e_i$, de poids 1, permet de corriger z ; on retourne donc les 28 premiers octets y' du vecteur $z - \lambda.e_i$.
   - sinon, on détecte qu'il y a eu moins deux erreurs. On ne réalise aucune correction : on retourne l'information que le mot est effacé. On rappelle qu'on note '?' un symbole effacé (qui n'appartient donc pas à $\mathbb{F}_{256}$) ; on retourne donc par exemple le mot y' formé de 28 symboles '?'.
2. On utilise le calcul du syndrome d'erreur avec le code C_1. Le décodage est le suivant :
   - si le bloc y ne contient pas '?', ses 24 premiers octets sont retournés ;
   - si il contient au moins cinq '?', l'erreur ne peut pas être corrigé ; un message d'erreur est retourné ;
   - sinon, on calcule les valeurs des u effacements inconnus ($1 \leqslant u \leqslant 4$) en résolvant le système linéaire de dimension $u \times u$ défini par $y.H_1^t = 0$ où H_1 est la matrice de contrôle de C_1. On remplace alors dans y les effacements par leur valeur et on retourne les 24 premiers octets.

Exercice 4.33, page 264.
Tout mot en entrée du décodeur C_1 qui contient moins de 4 effacements est corrigé. L'entrelacement étant de retard 4, cela correspond a au plus 15 colonnes consécutives effacées par C_2. Une colonne est constituée de 32 octets, donc correspond à une trame. Le code permet donc de corriger 15 trames consécutives.

Exercice 4.34, page 266.
Le polynôme formé par le message 100110100 est $P = X^2 + X^4 + X^5 + X^8$. On a $P * P_1 = X^3 + X^5 + X^6 + X^9$, ce qui correspond au mot binaire 1001101000 et $P * P_2 = X^2 + X^3 + X^4 + X^6 + X^8 + X^9$, ce qui correspond au mot binaire 1101011100. Par entrelacement, on obtient 11010011100111010000, ce qui correspond bien au message codé à la volée dans l'exemple 4.6.1.

Exercice 4.35, page 268.
On applique l'algorithme de Viterbi. Les marques sont écrites dans la figure 71. Détaillons le début du décodage : au départ, toutes les marques sont à l'infini. L'étape 0 est composée de zéros, le premier mot de code reçu est 10 qui est à distance 1 de chacun des deux arcs possibles partant de 0, donc les deux marques à l'étape 1 valent 1. Pour l'étape 2 et toutes les suivantes il y a 4 arcs possibles. Le deuxième mot de code reçu est 01, à distance 0 de l'arc 01, d'où la marque 1 sur le bit 0 et à distance 1 de l'arc 11, d'où la marque 2 sur le bit 1, etc.

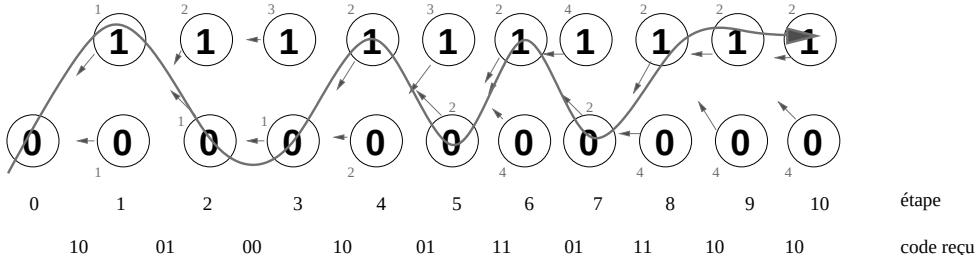


FIG. 71: Algorithme de Viterbi sur le message 10010010011101111010.

Le mot de code le plus proche est donc 11010011011101111010, à distance deux du mot reçu (marque pour le dernier bit de source), et le mot source associé en suivant la dernière phase de l'algorithme est 1001010111.

Exercice 4.36, page 272.

1. Pour k bits en entrée, on a en sortie du turbo code $\frac{k}{R_1}$ bits pour C_1 et $\frac{k}{R_2} - k$ bits pour C_2, puisque l'on ne conserve pas la sortie systématique de C_2. D'où un rendement $R_P = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}-1} = \frac{R_1.R_2}{R_1+R_2-R_1.R_2}$.
2. Le codage étant série, pour k bits en entrée, on a $\frac{k}{R_1}$ bits en sortie de C_1 et en entrée de C_2, donc $\frac{k}{R_1.R_2}$ bits en sortie de C_2. Le rendement est donc $R_S = R_1.R_2$.
3. Si $R_1 = R_2 = r$, les rendements respectifs R_P et R_S des turbo-codes parallèles et séries sont : $R_P = \frac{r}{2-r}$ et $R_S = r^2$.
   Comme $0 < r < 1$ et $(r-1)^2 > 0$, on a $r. < \frac{1}{2-r}$, d'où $R_S < R_P$. La composition série est de rendement inférieure à la composition parallèle. Dans le cas $r = \frac{1}{2}$, on a $R_P = \frac{1}{3}$ et $R_S = \frac{1}{4}$.

Solution de l'exercice du casino

Exercice C.1, page 277.

1. (a) $p_8 = 1 - 0.999^8 = 0.00797$

   (b) i. Comme on envoie des octets, on choisit $\mathbb{F}_{256}$, qui a $q = 256 = 2^8$ éléments, comme corps de base. Un code de Reed-Solomon impose alors de choisir $n = q - 1 = 255$.

   ii. Pour corriger au moins $0.0079n = 2.03$ erreurs, il faut pouvoir corriger 3 erreurs, donc choisir un code de distance $\delta \geqslant 2 * 3 + 1 = 7$. Avec un code de Reed-Solomon, le polynôme générateur est de degré $r = \delta - 1 = 6$.

   iii. Le nombre maximal d'erreurs détectées est 6.

   iv. Soit $g = \sum_{i=0}^{6} g_i X^i$ le polynôme générateur. La matrice génératrice a 248 lignes et 255 colonnes. Elle s'écrit sous la forme

$$\begin{bmatrix} g_0 & g_1 & \dots & g_6 & 0 & \dots & 0 \\ 0 & g_0 & g_1 & \dots & g_6 & \dots & 0 \\ & & & \dots & & & \end{bmatrix}$$

   (c) i. On choisit donc $P = 1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^8$ pour implémenter $\mathbb{F}_{256} = \mathbb{F}_2[\alpha]/P$. Soit $X = \sum_{i=0}^{7} x_i \alpha^i$ avec $x_i \in \{0, 1\}$; X est représenté par l'octet $(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$.
   Soit $Y = \sum_{i=0}^{7} y_i \alpha^i$ avec $y_i \in \{0, 1\}$ un autre élément du corps. $X + Y$ est alors représenté par l'octet $(x_0 + y_0, \dots x_7 + y_7)$.
   Pour $X.Y$, on calcule le produit de polynômes $X.Y \bmod P$; les coefficients de ce polynôme permettent d'obtenir le codage de XY sous forme d'octet.

   ii. Comme P est primitif, α est un élément générateur de $\mathbb{F}_{256}^{\star}$ (une racine primitive $255^{\text{ième}}$ de l'unité). Il suffit donc de choisir comme polynôme générateur $g = \prod_{i=1..6}(X - \alpha^i) \bmod P = X^6 + \sum_{i=0}^{5} g_i X^i$, où chaque g_i est un élément de $\mathbb{F}_{256}$ représenté par un octet (cf question précédente).

   (d) La capacité du canal binaire symétrique de probabilité d'erreur $q = 0.001$ est $C = 1 + q \log_2 q + (1 - q) \log_2(1 - q) = 0.98859$. Le code est de rendement $249/255 = 0.97648$ et le rendement du code ne dépasse pas la capacité du canal ($2^{\text{ième}}$ théorème de Shannon).

2. Le problème est que chaque séquence de dé reçue par le casino puisse être identifiée et associée à un croupier. On utilise donc une signature par le croupier de la séquence lors de la transmission.
   On peut utiliser une signature avec une clef secrète (construite entre le

casino et le croupier). Mais, pour que les joueurs puissent vérifier que la séquence émise est bien celle jouée, il est préférable d'utiliser un système à clef publique. Chaque croupier a une clef privée et une clef publique (créées par le casino par exemple). Le croupier chiffre la séquence avec sa clef privée : ainsi le serveur du casino et les joueurs peuvent vérifier, avec la clef publique du croupier, qu'il a transmis les bonnes informations.
Pour éviter qu'un jet de dé chiffré par un vrai croupier ne soit intercepté puis retransmis plus tard par un faux croupier (qui veut se faire passer pour le vrai) chaque émission est estampillée avec un numéro de lancer (incrémenté de 1 par le croupier) et une date éventuellement.
Enfin, pour minimiser le coût de chiffrement, le croupier peut limiter le chiffrement au résumé de la séquence (SHA-1 ou Whirlpool par exemple).
En conclusion : chaque croupier C a une clef publique Pub_C. Il numérote chaque séquence qu'il envoie dans un en-tête (date + numéro de séquence) et obtient un message m clair. Il calcule ensuite le résumé $r = \text{SHA1}(m)$ de m puis chiffre $D_{Priv_C}(r) = r'$ avec sa clef privée et transmet sur le canal le message (m, r'). Les joueurs et le casino peuvent alors vérifier que le message a bien été émis par le croupier : il leur suffit de vérifier que $\text{SHA1}(m)$ et $E_{Pub_C}(r')$ sont égaux.

3. (a) $H = \log_2(1/6) = 2.58$
   (b) On code les 6 faces par 3 bits : I=000, II=001, III=010, IV=011, V=100, VI=101. Les messages 110 et 111 sont inutilisés.
   (c) $l = 3$. La longueur est toujours supérieure à l'entropie qui est une borne inférieure pour tout codage binaire.
   (d) L'entropie est une borne inférieure. On peut faire un peu mieux que le code précédent et se rapprocher de l'entropie en faisant un codage de Huffman. On code alors V et VI sur 2 bits : V=10 et VI=11. Le codage est de longueur=$4.3/6 + 2.2/6 = 2.67$.
   (e) Il est optimal ; mais on peut encore améliorer en codant plusieurs lancers successifs (extension de source). Ainsi si l'on code 5 lancers successifs, on a $6^5 = 7776$ possibilités. Un code par blocs de même taille peut être réalisé avec $\log_2 7776 = 12.92$ soit 13 bits ; ce code est de longueur moyenne par lancer : $l = 13/5 = 2.6$. Un code de Huffman de ces 5 lancers ne peut être que meilleur (optimalité de l'arbre de Huffman).
   Asymptotiquement, on tend vers $H = 2.58$. Par exemple, avec 22 lancers, on a $\log_2(6^{21}) = 56.8$; donc 22 lancers sont codables par un code de même longueur = 57 bits, pour une longueur moyenne de code par lancer de dé = 57/22=2.59.

Bibliographie

[1] Thierry Autret, Laurent Bellefin et Marie-Laure Oble-Laffaire. *Sécuriser ses échanges électroniques avec une PKI : Solutions techniques et aspects juridiques.* Eyrolles, 2002.

[2] Gildas Avoine, Pascal Junod et Philippe Oechslin. *Sécurité informatique : Exercices corrigés.* Vuibert, 2004.

[3] Michel Barlaud et Claude Labit, éditeurs. *Compression et codage des images et des videos.* Hermes, 2002.

[4] Daniel J. Barrett, Richard E. Silverman et Robert G. Byrnes. *SSH, The Secure Shell : The Definitive Guide.* O'Reilly, seconde édition, 2005.

[5] Ed Bott et Carl Siechert. *Windows XP : Réseaux et sécurité.* Microsoft Press, 2006.

[6] Johannes A. Buchmann. *Introduction à la cryptographie.* Dunod, 2006.

[7] David M. Burton. *Elementary number theory.* International series in Pure and Applied Mathematics. McGraw-Hill, 4ième édition, 1998.

[8] Christophe Cachat et David Carella. *PKI Open source : déploiement et administration.* O'Reilly, 2003.

[9] Gérard Cohen, Jean-Louis Dornstetter et Philippe Godlewski. *Codes correcteurs d'erreur.* Masson, 1992.

[10] James H. Davenport, Yvon Siret et Évelyne Tournier. *Calcul formel : systèmes et algorithmes de manipulations algébriques.* Masson, 1993.

[11] Michel Demazure. *Cours d'algèbre. Primalité, Divisibilité, Codes*, volume XIII de *Nouvelle biliothèque Mathématique.* Cassini, Paris, 1997.

[12] Farid Dowla. *Handbook of radio frequency & wireless technologies.* Newnes, 2003.

[13] Touradj Ebrahimi, Franck Leprévost et Bertrand Warusfel, éditeurs. *Cryptographie et Sécurité des systèmes et réseaux.* Hermes, 2006.

[14] Touradj Ebrahimi, Franck Leprévost et Bertrand Warusfel, éditeurs. *Enjeux de la sécurité multimedia.* Hermes, 2006.

[15] Alain Glavieux, éditeur. *Codage de Canal*. Hermes, 2005.

[16] Claude Gomez, Bruno Salvy et Paul Zimmermann. *Calcul formel : mode d'emploi. Exemples en Maple*. Masson, 1995.

[17] Jean-Paul Guillois. *Techniques de compression des images*. Hermes, 1996.

[18] Aurélien Géron. *WiFi déploiement et sécurité : La QoS et le WPA*. Dunod/01 Informatique, seconde édition, 2006.

[19] Godfrey Harold Hardy et Edward Maitland Wright. *An introduction to the theory of numbers*. Oxford science publications. Clarendon Press, cinquième édition, 1998.

[20] Gilbert Held. *Data and Image Compression : Tools and Techniques*. Wiley, 4$^{\text{ième}}$ édition, 1996.

[21] D. G. Hoffman, D. A. Leonard, C. C. Linder, K. T. Phelps, C. A. Rodger et J. R. Wall. *Coding Theory : The Essentials*. Marcel Dekker, Inc., 1992.

[22] Eric Incerti. *Compression d'image : Algorithmes et standards*. Vuibert, 2003.

[23] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 de *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, troisième édition, 1997.

[24] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.

[25] David Leblanc et Michael Howard. *Écrire du code sécurisé : Techniques et stratégies pour écrire des applications sécurisées*. Collection développeur. Microsoft Press, 2003.

[26] Rudolf Lidl et Harald Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, révisée édition, 1994.

[27] Bruno Martin. *Codage, Cryptologie et applications*. Presses Polytechniques et Universitaires Romandes, 2004.

[28] James Massey. *Applied digital information theory*. ETH Zurich, 1998. www.isi.ee.ethz.ch/education/public.

[29] Alfred J. Menezes, Paul C. van Orschot et Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. www.cacr.math.uwaterloo.ca/hac.

[30] Maurice Mignotte. *Mathématiques pour le calcul formel*. PUF, 1989.

[31] Jean-Marie Monier. *Algèbre - MP*. Dunod, 2005.

[32] Jean-Marie Monier. *Algèbre - MPSI*. Dunod, 2005.

[33] Stéphane Natkin. *Les protocoles de sécurité d'internet.* Sciences Sup. Dunod, 2002.

[34] M. Nelson. *The Data compression book.* MT books, 1991.

[35] Odile Papini et Jacques Wolfmann. *Algèbre discrète et codes correcteurs.* Springer-Verlag, 1995.

[36] Vera Pless. *Introduction to the Theory of Error-Correcting Codes.* John Wiley and Sons, 3ième édition, 1998.

[37] Pascal Plumé. *Compression de données.* Eyrolles, 1993.

[38] Alain Poli et Llorenç Huguet. *Codes Correcteurs : Théorie et applications.* Logique, Mathématiques, Informatique. Masson, 1989.

[39] Alain Reboux. *Devenez un magicien du numérique : Effets graphiques, sonores et cryptographie.* Dunod, 2003.

[40] Steven Roman. *Introduction to Coding and Information Theory.* Springer, 1996.

[41] David Salomon. *Data Compression.* Springer, 1997.

[42] Bruce Schneier. *Secrets et mensonges : sécurité numérique dans un monde en réseau.* Vuibert informatique, 2000.

[43] Bruce Schneier. *Cryptographie appliquée, Algorithmes, protocoles et codes source en C.* Vuibert, Paris 2ième édition, 2001.

[44] Victor Shoup. *A computational introduction to number theory and algebra.* Cambridge University Press, 2005.

[45] Simon Singh. *Histoire des codes secrets.* LGF - Le livre de poche, 2001.

[46] William Stallings. *Sécurité des Réseaux : applications et standards.* Vuibert, 2002.

[47] Douglas Stinson. *Cryptographie : théorie et pratique.* International Thomson Publishing France, Paris, 1996.

[48] Peter Sweeney. *Error Control Coding - An introduction.* Prentice Hall International, 1991.

[49] Jacques Vélu. *Méthodes mathématiques pour l'informatique : Cours et exercices corrigés.* Dunod, quatrième édition, 2005.

[50] Joachim von zur Gathen et Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[51] Ross N. Williams. *Adaptive Data Compression.* Kluwer Academic Publishers, 1991.

Liste des figures, tableaux et algorithmes

Liste des figures

Liste des tableaux

Liste des algorithmes

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

050692 - (I) - (1,2) - OSB 100° - AUT - MLN

Achevé d'imprimer en Belgique sur les presses de SNEL Grafics sa
Z.I. des Hauts-Sarts - Zone 3 – Rue Fond des Fourches 21 – B-4041 Vottem (Herstal)
Tél +32(0)4 344 65 60 - Fax +32(0)4 286 99 61
janvier 2007 — 40840

Dépôt légal : février 2007

SCIENCES SUP

Jean-Guillaume Dumas • Jean-Louis Roch
Éric Tannier • Sébastien Varrette

THÉORIE DES CODES
Compression, cryptage, correction

Ce manuel s'adresse aux étudiants en Master 1 ou 2 de mathématiques appliquées ou d'informatique ainsi qu'aux élèves ingénieurs. Il sera une référence utile pour les enseignants-chercheurs et les sociétés informatiques intervenant dans les télécommunications ou la sécurité.

La transmission d'informations numériques est omniprésente dans la technologie aujourd'hui. Qu'il s'agisse des textes, de sons, d'images, de vidéos, de codes barres ou de numéros de carte bleue, que ces informations transitent par Internet, par satellite ou soient gravées sur un DVD, leur circulation doit répondre à des contraintes fortes : optimisation de la taille des messages transmis pour éviter de surcharger les canaux, palliation des erreurs de canal, secret et authentification des transmissions, rapidité des calculs pour les transformations que nécessite le passage par un canal.

L'originalité de cet ouvrage est de présenter une théorie unifiée des principes mathématiques et informatiques qui fondent ces développements technologiques. L'accent est mis sur une présentation en détail et en profondeur, alliant structures algébriques et développement algorithmique poussé, des protocoles de télécommunication en vigueur actuellement.

Les notions théoriques présentées sont illustrées par 120 exercices corrigés.

JEAN-GUILLAUME DUMAS est maître de conférences à l'université Grenoble 1.

JEAN-LOUIS ROCH est maître de conférences à l'ENSIMAG.

ERIC TANNIER est chercheur de l'INRIA Rhône-Alpes à l'université Lyon 1.

SÉBASTIEN VARRETTE est doctorant à l'université du Luxembourg.

MATHÉMATIQUES
PHYSIQUE
CHIMIE
SCIENCES DE L'INGÉNIEUR
INFORMATIQUE
SCIENCES DE LA VIE
SCIENCES DE LA TERRE

LICENCE MASTER DOCTORAT
1 2 3 4 5 6 7 8

9 782100 506927

6494405
ISBN 978-2-10-050692-7

www.dunod.com

DUNOD