# Structuration et contrôle de l'information

*Nous revenons dans ce chapitre à la notion d'information, abordée à une plus grande échelle. La première idée que nous présentons est celle de la structuration d'informations. La seconde est celle de la communication de l'information sur les réseaux qui demande de nouveaux algorithmes qui permettent la manipulation de grandes quantités : des algorithmes de compression, dont nous avons déjà vu un exemple au chapitre « Algorithmique », des algorithmes de correction d'erreur et des algorithmes de chiffrement. La différence entre les algorithmes de correction d'erreur et de chiffrement fait apparaître deux problématiques transversales à l'informatique : la protection contre les erreurs involontaires et la protection contre les agents malveillants.*

## Cours

### Structures de données de base

**Définir des données structurées** Moi, c'est « Boulanger, Paul, boulanger à Saint Paul ». Enchanté, « Jean Pierre, de Grande Pierre, tailleur de Pierre ». Comprenne qui pourra. Extraire du discours les informations données peut rapidement devenir hasardeux et finira par être voué à l'échec. Ici, toutes les ambiguïtés seraient levées si nous présentions les choses de manière structurée, par exemple sous la forme :

```
{
prénom = "Paul"
patronyme = "Boulanger"
métier = "boulanger"
lieu = "06570"
}
```

Chaque paramètre est maintenant explicité. On pourra même, pour exprimer le lieu, disposer de la liste des 36 000 villes et villages de France – du point de vue informatique, c'est une liste de taille moyenne – et lever les ambiguïtés dues au fait que dix villages ont le nom « Saint-Paul » ou que ce nom peut s'écrire « St. Paul », « Saint Paul », etc. en utilisant le code postal, au lieu du nom de la commune.

Quelles sont les « bonnes pratiques » pour définir des données structurées ? Voici trois éléments de réponse.

– *Maximiser l'atomicité.* La structure des données doit avoir une décomposition maximale. Par exemple, il ne faut pas spécifier nom = "Jean Pierre" mais bien {prénom = "Jean" patronyme = "Pierre"} afin de lever toute ambiguïté et pouvoir accéder à chaque « atome » d'information. En fait, dès qu'une information, par exemple une adresse postale, nous semble composite, elle doit être décomposée en éléments plus petits.

– *Maximiser la factorisation*. C'est-à-dire éviter la redondance des données : si une donnée se retrouve plusieurs fois à différents endroits, il faut l'extraire, la coder à une place spécifique et créer des références des endroits initiaux vers sa place canonique. Par exemple, le code postal référence de manière unique les communes de France, il est préférable de l'utiliser plutôt que « Saint-Paul-de-Vence, dans les Alpes-Maritimes ». De plus, ces informations ne concernent pas directement notre Paul Boulanger, mais tous les habitants de Saint Paul. Ici, le code postal est une *référence* qui permet de *pointer* de la structure qui décrit Paul Boulanger vers une autre liste, celle des communes de France, où toutes les informations relatives à Saint Paul sont codées une seule fois. Cela permet d'économiser du temps et de la place, mais aussi d'éviter des incohérences de données.

– *Minimiser les sous-structures de données*. C'est-à-dire éviter les arborescences complexes, lorsque les données peuvent être représentées sous forme de liste uniforme des données. Ainsi, nous aurions pu représenter l'information de la manière suivante :

```
{
 nom = {
  prénom = "Paul "
  patronyme = "Boulanger "
 }
…
}
```

renforçant l'idée que prénom et patronyme sont les deux sous-composantes de la composante nom. Cela est plus rigoureux, mais guère indispensable et rend plus complexe l'accès aux données.

L'utilisation de telles *structures logiques* répond à deux objectifs.

– *Maximiser les descriptions génériques*, c'est-à-dire les représentations indépendantes du format de fichier ou de la syntaxe du système qui les exploite. Selon la façon dont les données vont être utilisées, les algorithmes pourront les restructurer à leur façon, mais ici nous les codons de la manière la plus proche du sens que nous leur donnons.

– *Maximiser la description statique des objets*, c'est-à-dire préciser, autant que possible, les propriétés – caractéristiques, paramètres, options – d'un programme comme un ensemble statique de données, plutôt que de glisser ces données dans le code de ce programme.

Nous aurions pu aussi représenter l'information par un algorithme de la forme : `si (prénom == "Paul") alors nom = "Boulanger"`… mais l'information aurait été noyée dans le code du programme. Ce qui est proposé ici est de séparer au maximum données et programme.

**Quelques données structurées essentielles** Les formulaires que nous trouvons sur le Web nous permettent de définir de tels « n-uplets à champ nommés », ou enregistrements, que nous avons introduits dans le deuxième chapitre et que nous retrouverons dans le septième chapitre.

*Données numériques et quantités physiques*. Nous avons vu comment coder un nombre. Mais la représentation d'une quantité physique, de toutes les valeurs numériques en fait, ne se limite pas à une valeur. Si Paul Boulanger mesure 1,80 m, cette donnée numérique appartient au type des `tailles_humaines` et se spécifie correctement par rien moins que… six paramètres !

```
{
 taille_humaine = {
  valeur_minimale : float = 0
  valeur_maximale : float = 2.7
  précision : float = 0.01
  valeur_par_défaut : float = 1.7
  unité : identifier = "m"
 }
}
```

à savoir, son identificateur – `taille_humaine` –, ses valeurs minimales et maximales, sa précision, une valeur par défaut et l'unité physique qui est un

identificateur, indispensable pour manipuler des quantités avec des unités éventuellement différentes.

Il est essentiel de toujours définir ces paramètres pour manipuler correctement l'information numérique en question. Les algorithmes qui vont estimer ces données numériques ou les utiliser ont besoin de savoir, par exemple, entre quelles valeurs et à quelle précision travailler. Il faut aussi constater que ces spécifications sont naturelles et accessibles : une règle d'écolier mesure une longueur entre 0 et 20 cm à une précision de 1 mm ; un point sur une image numérique est situé horizontalement entre le bord gauche et droit de l'image à une précision de un pixel et une valeur par défaut égale au milieu de l'image ; toute vitesse se situe à plus ou moins la vitesse de la lumière avec une valeur par défaut égale à zéro, etc. C'est donc un simple exercice de rigueur que d'expliciter ces informations, lors du codage de données numériques.

*L'exemple de la date et de l'heure*. Juste avant la tragédie du `01-09-11` le « bug de l'an 2000 » a fait couler beaucoup d'encre… De quoi s'agissait-il ? D'une mauvaise standardisation du codage de la date. Que signifie `01/09/11` ? L'Anglo-Saxon lira le 9 janvier 2011 – ou 1911, suivant son siècle. Le Français, le 1er septembre de la même année. Les anciens logiciels, le 11 septembre 2001, ou 1901. Le codage des bases de données sera `2001-09-11 : année/mois/jours`, c'est la norme ISO-8601. C'est le bon ordre pour aller de la plus grande unité à la plus petite, donc pour que l'ordre lexicographique de la chaîne de caractères corresponde à l'ordre des dates. On fera bien attention de ne pas omettre les zéros, à ne pas écrire `2001/9/11`, bref à respecter l'expression régulière `date : [0-9][0-9][0-9][0-9]/[0-9][0-9]/[0-9][0-9]`. On proposera aussi à l'utilisateur un calendrier graphique à cocher plutôt que le laisser s'embrouiller dans les codages chiffrés des mois ; il pourra aussi situer les jours de la semaine.

Dans le système d'exploitation *Unix*, le temps est codé par la différence, mesurée en secondes, entre l'heure actuelle et 0 heure, 1er janvier 1970, heure de Greenwitch, codée sur 64 bits. Dans beaucoup de logiciels, ce codage est fait en millisecondes avec la même origine. Ce qui permet de coder $10^9$ années environ, et une précision de 1 ms bien évidemment. S'il avait été codé sur 32 bits, ce nombre aurait eu des valeurs minimales et maximales… assez limitées – amusons-nous à le calculer. Cette précision de la milliseconde est générique : tandis qu'un ordinateur effectue quelques milliards d'opérations élémentaires par seconde, on ne peut lui demander d'avoir une précision absolue d'une nanoseconde.

*L'exemple des identificateurs universels de ressources*. Lorsque nous tapons par exemple : `http://science-info-lycee.fr`, nous identifions une res-

source disponible sur Internet, ou dans notre ordinateur, de manière universelle. Il se trouve que chaque fragment de cette chaîne de caractères a un sens précis, il correspond à une structure de données qui caractérise exactement la ressource que nous cherchons et la façon dont nous voulons y accéder.

Avec les ingrédients donnés ici, nous verrons en exercice comment comprendre chaque détail de cette spécification.

*L'exemple des documents textuels*. Quelle est la différence entre une chaîne de caractères brute et un texte ? Dans un texte, il y a une structure – des enluminures, des paragraphes, des références… – et des métadonnées – son titre, son auteur, la langue dans laquelle il a été écrit… – et tous ces éléments peuvent être spécifiés grâce aux constructions que nous avons évoquées ici.

Les métadonnées, le titre du texte, son auteur, ses destinataires, etc., rendent le texte référençable. C'est, par exemple, la structure des courriels : ils ont un sujet – c'est le titre –, un expéditeur – c'est l'auteur, sauf dans quelques cas particuliers –, etc. Un document aura en plus des mots-clés – ou *tags* – qui sont des mots précis – pour le présent chapitre, ce sera sûrement « informatique », « information »… – permettant d'indexer le texte, c'est-à-dire de le retrouver dans une base de données ou une bibliothèque numérique qui contient un grand nombre de documents. Si c'est un document multimédia, les auteurs auront probablement des rôles divers : rédacteur, illustrateur, éditeur, relecteur…

Trois ingrédients vont faire de ce texte plus qu'une chaîne de caractères.

Le texte lui-même est structuré de manière hiérarchique en chapitres, sections, sous-sections et paragraphes, par exemple quelque chose comme

```
{
 titre = "Introduction"
 {
  titre = "Avant-propos"
  …
}
{
 titre = "Présentation"
 …
}
 …
}
```

où l'on voit que l'avant-propos et la présentation sont, ici, des sous-sections de l'introduction. C'est cette structure qui permet de naviguer dans le texte sans être obligé de le lire du premier au dernier caractère à chaque accès !

Des portions de texte sont enluminées, c'est-à-dire mises en italique ou entre guillemets, etc.

Surtout, les textes numériques peuvent avoir des liens hypertexte, c'est-à-dire que nous pouvons cliquer pour accéder à un autre contenu : consulter la définition d'un mot si nous ne le connaissions pas. L'encyclopédie Wikipédia en est l'exemple paradigmatique. Ces liens donnent aussi accès à du matériel supplémentaire au texte – une illustration, une parenthèse vers un élément connexe mais hors du fil du discours, etc. Sur papier, ce sont des notes de bas de page ou des références vers un glossaire ou une bibliographie.

Ces trois ingrédients – structure, enluminure, liens – font du texte, qu'il soit numérique ou sur papier, un document bien plus riche qu'un simple discours linéique. Que le texte soit numérique enrichit évidemment les implémentations possibles, mais il est bon de noter que cet enrichissement n'est pas lié à la numérisation : le livre papier code les mêmes ingrédients.

**Des données structurées aux structures de données** Ce qui reste à découvrir, c'est que, sous-jacentes à ces données structurées, l'informatique est dotée de structures de données très riches pour représenter la structure de ces données… Elles se rangent dans deux grandes catégories :

*Les collections*. Les collections sont des suites de données : des listes. Selon la façon dont nous accédons à ces objets, nous parlerons de tableaux où les $N$ objets sont numérotés de 0 à $N-1$ et nous pouvons accéder en lecture ou écriture au $i$-ème élément directement : si le tableau est de nom `tab`, son $i$-ème élément se note souvent `tab[i]`. Nous appelons *tableau* une telle structure de longueur fixe, décidée lors de sa construction. Des tableaux de longueur ajustable au fur et à mesure que nous y ajoutons des éléments sont appelés *vecteurs*. Les piles sont des vecteurs auxquels nous pouvons ajouter les éléments à un seul bout et retirer uniquement le dernier élément, comme une pile d'assiettes : se doter de telles restrictions permet de bien cadrer à quelle fonction une telle structure correspond, donc à en spécifier l'usage. Cela permet aussi d'optimiser son implémentation.

*Les tables de valeurs*. Les tables sont des listes qui associent à une valeur d'entrée une valeur de sortie. Par exemple, un dictionnaire français-anglais qui a des mots français en entrée et leurs correspondants anglais – supposés uniques ici – en sortie. Ou des tables de paramètres avec des noms de paramètres en entrée et des valeurs en sortie. Contrairement au tableau, l'index – appelé « clé » dans ce cas – n'est pas forcément un entier de 0 à $N-1$, mais peut être une chaîne de caractères ou un objet numérique quelconque. Tandis que les valeurs d'une collection sont implicitement ordonnées du premier au dernier, l'ordre de ces paires de valeurs d'entrées/sorties n'a aucune signification.

Ce qui est remarquable ici, c'est que ces quelques éléments décrivent qualitativement la très grande majorité des structures de données informatiques.

Les données du Web sont décrites en utilisant le langage XML, dont le langage HTML est la spécialisation pour les pages web, à quelques nuances syntaxiques près. Cette spécification combine tout simplement collections d'objets et tables de valeurs. La structure logique d'un document XML est une spécification minimale, mais très puissante pour décrire des informations symboliques – l'idée n'est pas nouvelle, mais elle prend en XML une forme assez aboutie et est devenue un standard, ce qui est essentiel pour partager les données. XML, qui date de 1998 pour la 1re édition, est un langage de représentation d'information dit semi-structuré, c'est-à-dire qu'il permet de gérer facilement l'intégration de données à partir de schémas différents, ainsi que les données manquantes – valeurs nulles.

Ce noyau de spécifications XML comprend, entre autres choses :

XML eXtensible Mark-up
le langage lui-même, utilisé pour la représentation de données et documents structurés.
XSLT eXtensible Style Language Transformation
pour définir comment traduire une structure en une autre structure.
XML Schema pour définir le type des données, leur structure et vérifier leur validité.
XHTML eXtensible HyperText Mark-up Language
pour les documents du Web que nous connaissons.
SVG Scalable Vector Graphics
utilisé pour la représentation graphique en 2D.
XUL eXtensible User interface Language utilisé pour spécifier une interface utilisateur graphique (menus, boutons, etc.).
MathML mathématique Mark-up Language, utilisé pour représenter les formules mathématiques.
DocBook utilisé pour spécifier les métadonnées d'un document.

Toutes les informations symboliques des objets numériques sont entièrement spécifiées à travers de tels standards. C'est le reflet numérique des objets du monde réel.

## Compression de l'information

La compression de données ou codage de source est l'opération informatique qui consiste à transformer une suite de bits *A* en une suite de bits *B* plus

courte, contenant les mêmes informations, en utilisant un algorithme particulier. Il s'agit d'une opération de codage. La décompression est l'opération inverse de la compression. C'est un couple de mécanismes omniprésent dans les systèmes numériques.

Avec un algorithme de compression sans perte, la suite de bits obtenue après les opérations successives de compression et de décompression est strictement identique à l'originale. Ces algorithmes s'appliquent à tous les objets numériques, sans tenir compte de leur contenu. Avec un algorithme de compression avec perte, la suite de bits obtenue après les opérations de compression et de décompression est différente de l'originale, mais l'information reste sensiblement la même, ces algorithmes sont donc liés au contenu – images, son et vidéo.

Rappelons qu'il n'existe pas de technique de compression de données sans perte universelle, qui pourrait compresser n'importe quel fichier. Par exemple, on vérifie en pratique qu'un fichier qui est déjà le résultat d'une compression se compresse mal, voire grossit par application du compresseur. Mais, dans la pratique, les mots, messages ou fichiers que l'on souhaite compresser ne sont pas quelconques et choisis aléatoirement parmi tous les mots, messages ou fichiers possibles, ils vérifient certaines propriétés que les compresseurs utilisent.

Compresser des données n'est pas si facile : les premières idées, comme celle consistant à utiliser un codage par répétition, où une suite de bits ou de caractères identiques est remplacée par un couple formé du nombre d'occurrences et bit ou caractère répété, mènent à des méthodes qui ne sont pas efficaces en pratique.

**Codage par dictionnaire** Le codage par dictionnaire est une classe d'algorithmes de compression sans perte qui fonctionne par la recherche d'une correspondance entre le contenu à comprimer et un ensemble de chaînes binaires contenues dans une structure de données appelée le « dictionnaire ». Lorsque le compresseur trouve une telle correspondance, il substitue à la chaîne binaire reconnue la référence dans ce dictionnaire.

Pour décrire en détail un tel algorithme, de type Lempel-Ziv-Welch, faisons les conventions suivantes :

– nous découpons la chaîne binaire en octets ;

– nous considérons un dictionnaire qui associe à chaque chaîne d'octets un numéro ;

– nous initialisons le dictionnaire avec les chaînes à un octet, le numéro de chaque chaîne étant la chaîne elle-même : 00000000 a pour numéro 000000000, etc. ;

– nous codons initialement les numéros sur 8 bits, puis, lorsque cela ne suffira plus, sur 9 bits, puis 10 bits, etc.

À mesure que le compresseur examine le texte, chaque fois qu'une chaîne déjà rencontrée est lue, la chaîne la plus longue déjà rencontrée est déterminée, et le numéro correspondant à cette chaîne avec le caractère concaténé – le caractère suivant du flux entrant – est enregistré dans le dictionnaire. Le numéro pour la partie la plus longue de la chaîne de caractères rencontrée est envoyé en sortie et le dernier caractère est utilisé comme base pour la chaîne suivante. Ce mécanisme permet donc de substituer aux chaînes leur numéro dans le dictionnaire. Le dictionnaire est ainsi construit dynamiquement d'après les motifs rencontrés.

L'algorithme de décompression reconstruit le dictionnaire à partir du texte compressé en entrée, puisqu'il dispose d'un numéro déjà connu et du caractère de base pour la chaîne suivante pour fabriquer le nouveau numéro. Cette opération se fait avec un temps de retard et il faut tenir compte du cas où le numéro de la chaîne suivante correspond au nouveau numéro non encore entré, mais cela se détecte et se gère sans problème. Il va donc aussi automatiquement augmenter le nombre de bits du codage du numéro selon la taille du dictionnaire reconstitué.

Les variantes de ces méthodes permettent de disposer d'un algorithme de compression performant et d'un algorithme de décompression rapide. Ils sont utilisés dans les compressions usuelles – fichiers « zip », images au format « gif », etc.

**Codage entropique** Nous avons déjà vu, dans le troisième chapitre, un algorithme de codage entropique : l'algorithme de Huffman. De manière plus générale, le codage entropique utilise des statistiques sur le message source, avec un code à longueur variable, qui attribue les mots de code les plus courts aux symboles de source les plus fréquents. La limite de cette possibilité de compression est l'entropie définie dans le premier chapitre.

Un algorithme plus sophistiqué et plus courant est l'algorithme de Huffman adaptatif : au fur et à mesure de l'arrivée des symboles, on compte leurs occurrences, puis on les associe sous forme d'un arbre où chaque feuille correspond à un symbole. Cet algorithme est adaptatif, car l'arbre est construit de manière dynamique au fur et à mesure de la compression du flux. La compression est toujours adaptée aux données, sans calcul statistique avant la compression. Il ne faut donc pas transmettre ou stocker la table des fréquences des symboles, puisque cette dernière se reconstruit à la décompression. L'algorithme est de plus capable de coder sur des flux de données, sans avoir besoin de connaître les symboles à venir pour déterminer le codage.

**Codage et mesure en information** Ces mécanismes de compression ont un double usage pratique lors de transmission de données et lors d'archivage sur des supports. Ils peuvent être combinés, à ces fins, avec les mécanismes d'encryptage ou de codes correcteurs décrits ci-après.

Les notions définies dans le premier chapitre permettent à la fois de comprendre les limites de ces méthodes et de les utiliser à plus haut niveau.

D'une part, on peut montrer que, pour une source de données d'entropie $H$, la longueur moyenne $L$ d'un mot de code obtenu par codage de Huffman en utilisant des blocs de $n$ symboles vérifie :

$$H \leq L < H + \frac{1}{n}$$

donc que l'information au sens de Shannon est bien une borne maximale de compression et que le codage de Huffman peut donner une très bonne approximation de cette borne : il est optimal.

Lorsque Huffman a inventé son algorithme, il avait connaissance des travaux les plus théoriques sur l'information de Shannon et il avait la volonté de construire un procédé de compression optimal. Proposer un tel algorithme n'avait donc pas uniquement un but pratique, mais aussi le but, plus théorique, d'atteindre la borne de Shannon.

Par ailleurs, ces algorithmes de compression permettent d'approximer le contenu en information d'un message au sens de Kolmogorov. En appliquant divers mécanismes de compression, on peut évaluer la longueur du programme correspondant qui pourra générer le message – c'est la longueur de l'algorithme de décompression ajoutée à la longueur du message compressé – et des méthodes assez sophistiquées permettent alors d'interpoler une approximation du contenu en information lui-même. Très récemment, il a été montré que la complexité de Kolmogorov est approchée par la taille d'un fichier compressé, et que l'on peut approcher la profondeur de Bennett en considérant le temps de décompression des fichiers compressés.

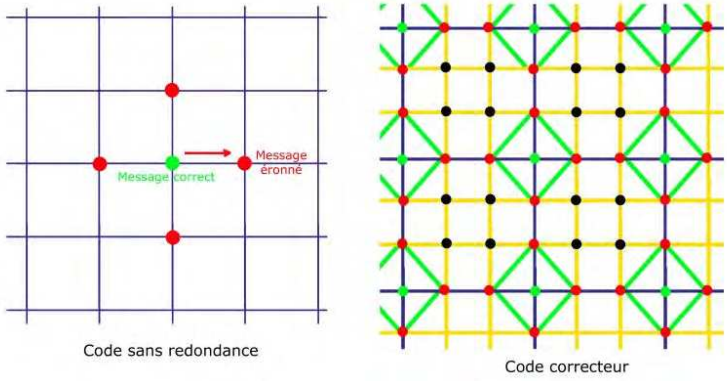
## Codes correcteurs d'erreurs

Si la transmission n'est pas parfaite, certains bits seront erronés. Il faut alors ajouter à la suite binaire des bits supplémentaires pour espérer détecter, voire corriger, les possibles erreurs. L'exemple le le le plus plas plus simple sumple simple est de transmettre trois fois chaque élément. Si les trois éléments reçus sont identiques, la transmission a été effectuée sans erreur. Si l'un diffère des deux autres, il y a une erreur que l'on corrige en choisissant la valeur confirmée deux fois. Si deux erreurs se sont produites, cela ne sera ni détecté ni corrigé.

On peut aussi transmettre un bit de contrôle, par exemple 0 ou 1 selon qu'il y a eu un nombre pair ou impair de 1 dans les 8 bits précédents.

Les codes les plus efficaces sont plus compliqués et reposent par exemple sur l'utilisation de propriétés de polynômes sur un corps fini. Dans tous les cas, il faut ajouter de la redondance à l'information pour détecter et/ou corriger les erreurs. La problématique au niveau industriel est double. Dans le cas de la transmission de données, par exemple sur Internet, le rôle du code correcteur peut se limiter à la détection des erreurs et la correction est alors réalisée par une nouvelle demande de transmission du message. Pour d'autres situations, l'objectif est la correction d'erreurs, sans nouvelle demande possible de transmission. Avec deux types de situations : la correction de petites erreurs relativement fréquentes mais isolées, par exemple lors de transmissions, ou moins fréquentes mais beaucoup plus volumineuses, par exemple lors de rayures ou d'impuretés sur un DVD. Dans ce dernier cas, les codes utilisés corrigent jusqu'à 4096 bits consécutifs, soit une rayure de plus d'un millimètre de large. Dans un premier cas, les erreurs induisent une modification des données, où certains bits passent de la valeur 0 à la valeur 1 et vice versa. Dans un second cas, les erreurs provoquent des pertes d'informations aussi appelées effacements.

Le principe qu'une information supplémentaire est nécessaire pour, soit détecter l'erreur, soit la corriger, s'illustre très bien de manière géométrique :




Si une unique erreur se produit, alors le message transmis correspond à un point rouge. Si la redondance a été habilement construite, alors il n'existe qu'un point licite en vert proche du point rouge reçu. Si d'autres erreurs se produisent et que le message transmis correspond à un point noir, l'erreur est détectée sans être corrigée.

Les points noirs consomment de la place inutilement si l'objectif est d'accepter le message même partiellement erroné. Un code est dit *parfait* quand il existe une taille des boules centrées sur les mots de code qui forment une partition de l'espace. On corrige alors certaines erreurs, mais on ne détecte pas d'erreurs non correctibles.

Un code correcteur propose donc une géométrie où les messages licites sont éloignés le plus possible les uns des autres. Les boules centrées sur les bons codes, si elles ne s'intersectent pas, permettent de retrouver le bon message, correspondant à son centre. Une perturbation, tant qu'elle reste suffisamment petite pour ne pas faire sortir le code de sa boule, est correctible.

Un bel exemple est le code de Hamming (7,4) qui, à travers un message de sept bits, transfère quatre bits de données et trois bits de *parité* qui permettent de corriger toute erreur portant sur un bit unique. C'est-à-dire que si, sur les sept bits transmis, l'un est altéré – un 0 devient un 1 ou l'inverse —, alors il existe un calcul algébrique permettant de corriger l'erreur. La génération du code est donnée par la matrice :

| | $d_0$ | $d_1$ | $d_2$ | $d_3$ |
|---|---|---|---|---|
| $c_1$ | 1 | 0 | 0 | 0 |
| $c_2$ | 0 | 1 | 0 | 0 |
| $c_3$ | 0 | 0 | 1 | 0 |
| $c_4$ | 0 | 0 | 0 | 1 |
| $c_5$ | 1 | 1 | 0 | 1 |
| $c_6$ | 1 | 0 | 1 | 1 |
| $c_7$ | 0 | 1 | 1 | 1 |

qui indique comment calculer les sept bits du message codé à partir des quatre bits du message original : $c_1 = d_0$, $c_2 = d_1$, $c_3 = d_2$, $c_4 = d_3$, $c_5 = d_0 + d_1 + d_3$, $c_6 = d_0 + d_2 + d_3$, $c_7 = d_1 + d_2 + d_3$. Ces opérations s'effectuent en arithmétique modulo deux, où l'addition et la multiplication sont définies par les tables suivantes :

| A | B | A + B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| A | B | A . B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

L'addition correspond à la fonction logique « ou exclusif » et la multiplication correspond à la fonction logique « et ». La somme d'une suite de valeurs est égale à 0 si la somme est paire dans les entiers et 1 sinon.

À la réception, un calcul matriciel dual défini par la matrice :

| | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ |
|---|---|---|---|---|---|---|---|
| $e_0$ | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| $e_1$ | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| $e_2$ | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

permet de calculer les bits $e_0e_1e_2$ d'erreur qui valent 000 en cas de transmission correcte et donne *l'indice du bit erroné* en cas d'une erreur de transmission, qui peut alors être corrigé pour retrouver la valeur correcte.

Mathématiquement, ce code est structuré comme un sous-espace vectoriel sur un corps fini, ici le corps à deux éléments. Nous pourrions généraliser à des corps de base $d$, $d$ étant une puissance de nombre premier. L'espace vectoriel du code est de dimension $n = 7$ dans notre cas et permet de coder des mots de taille $k = 4$ avec une distance minimale de $\delta = 3$ entre les mots du code, permettant de corriger des erreurs sur $\delta/2 = 1$ bit. On peut définir des codes linéaires pour d'autres dimensions $[n, k, \delta]$ et il est possible de vérifier qu'il faut $n - k \geq \delta - 1$. Si la borne est atteinte, nous sommes dans un cas de distance séparable maximale, ce qui n'est pas tout à fait le cas pour le code de Hamming (7,4).

### Fonctions de hachage et condensats

Quand les erreurs sont peu probables, on se contente de les détecter, sans essayer de les corriger. Par exemple, lorsqu'on télécharge un gros fichier, la probabilité d'erreur est faible, mais on veut pouvoir détecter qu'il y a eu une erreur, afin de recommencer le téléchargement. Pour cela, on peut utiliser un condensat, c'est-à-dire un résumé du fichier que l'on pourra comparer à un résumé de référence.

En fait, le condensat est le résultat d'une fonction, appelée fonction de hachage, qui, à partir d'une donnée fournie en entrée, calcule une valeur servant à identifier rapidement, bien qu'incomplètement, la donnée initiale. Le

résultat d'une fonction de hachage peut être appelé selon le contexte « somme de contrôle », « empreinte », « hash », « résumé de message », « condensé », « condensat » ou encore « empreinte cryptographique ».

Les fonctions de hachage doivent être rapides : calculer le condensat d'une donnée ne doit coûter qu'un temps négligeable. De plus, elles doivent éviter autant que possible les collisions : ce sont des états dans lesquels des données différentes ont un condensat identique.

En général, on veut un condensat bien plus petit que la donnée à traiter. Ainsi, on veut télécharger le DVD d'une installation Linux d'environ 4 Go et on vérifie que le téléchargement s'est bien déroulé avec un condensat de 256 à 1024 octets. Si deux fichiers ont des condensats différents, ils sont différents, et on demande également que deux données différentes, mais proches, aient des condensats différents. Sinon, l'utilisation comme code détecteur d'erreur serait peu efficace. On demande donc souvent que deux données différant sur un seul bit aient des condensats différents.

Un premier exemple de condensat est le MD5 (*Message Digest 5*) inventé par Ronald Rivest en 1991. C'est un algorithme basé sur les opérations binaires ou exclusif, et, ou, non, l'addition, modulo 32 bits, et la rotation de bits, ainsi qu'un certain nombre de constantes. Depuis, cet algorithme a été étudié et on a découvert des collisions complètes. N'étant plus sûr, MD5 devrait être mis de côté au profit de fonctions plus robustes. L'algorithme SHA-1 (*Secure Hash Algorithm*) produit un condensat de 160 bits. Il utilise les mêmes briques de base que MD5 avec quelques opérations supplémentaires. Il a également été cassé dans les années 2000. Le condensat actuellement conseillé est son successeur SHA-256 dont le condensat est sur 256 bits.

## Codage et cryptage

Dans de nombreuses situations, nous ne souhaitons pas voir nos données personnelles – numéro de carte de crédit, photos, lettres, etc. – exposées au grand jour. Cela demande de crypter ces informations. Cette idée est loin d'être nouvelle, puisque Jules César souhaitait déjà envoyer à ses armées des ordres, sous la forme de messages écrits, sans, bien entendu, que ceux-ci soient connus de ses ennemis. L'accès aux messages étant facile – il suffisait de capturer les messagers –, il fallait les crypter.

### Codage de César et chiffrement par substitution

La méthode utilisée par César était extrêmement simple : elle consistait à associer à chaque lettre une autre lettre – par exemple la suivante dans l'ordre alphabétique. Pour César, c'était la lettre suivant celle suivant la suivante. Cela

revient à considérer les lettres comme des entiers entre 0 et 25 et à considérer une addition de 3 modulo 26. Ainsi, le message « je veux un bon cryptage » devenait « mh zhxa xq erq fubwdjh », ce qui n'est pas évident à traduire, sauf si l'on connaît le code – il suffit alors de soustraire 3 modulo 26. Cette méthode était efficace, mais son efficacité tenait beaucoup au fait que très peu de gens savaient lire. Elle souffre, en fait, de nombreuses faiblesses : la clé – c'est-à-dire la valeur permettant de coder et décoder les messages – est toujours la même, il suffit donc de la récupérer une seule fois pour pouvoir traduire tous les messages futurs. De plus, en comparant les messages codé et décodé, on en déduit facilement la clé. Enfin, la clé est 3 qui correspond à la lettre C, initiale de César, ce qui aide à confirmer que 3 est la bonne clé. Une version plus récente est le ROT13 qui ajoute 13 modulo 26 et qui permet de crypter légèrement des informations – par exemple, pour ne pas révéler la solution d'un jeu ou la fin d'un film. Il est surtout utilisé dans les forums, news et groupes.

De nos jours, un tel algorithme ne résisterait pas longtemps. Si l'on suppose qu'une lettre est toujours cryptée en une même autre lettre, il suffit d'observer, dans un message crypté, la fréquence des lettres pour retrouver le code. En français, la lettre « e » est la plus fréquente. Il faut donc, en considérant suffisamment de messages cryptés, trouver la lettre la plus fréquente pour lui associer la lettre « e ». On continue ensuite avec les autres lettres en fonction de leur fréquence moyenne en français et dans les messages cryptés. Cette méthode simple fonctionne très bien et permet de casser les codes par substitution assez facilement.

### Cryptages symétriques

Dans les grandes catégories d'algorithmes de cryptage, on trouve les cryptages symétriques : c'est la même clé qui sert à chiffrer et à déchiffrer un message. Dans ce cas, il est primordial de garder cette clé secrète et qu'elle soit aléatoire. Une clé qui serait la date de naissance ou le nom du chien de l'expéditeur peut se deviner facilement. Un exemple simple de cryptage symétrique est le suivant. On considère l'opérateur binaire ou exclusif, noté $\oplus$ dont la table de vérité est

| A | B | A$\oplus$B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

et qui correspond à l'addition modulo 2, vue précédemment.

Revenons au chiffrement d'un message $M$ de taille $n$. Il nous faut une clé secrète et aléatoire $C$ également de taille $n$. On fait alors un chiffrement bit à bit : chaque bit de $M$ subit un ou exclusif avec le chiffre de même rang de la clé $C$. On notera également, par abus de langage, $\oplus$ le ou exclusif sur les suites de bits. Pour déchiffrer le message crypté $M' = M \oplus C$, on procède exactement comme pour le chiffrement et on refait un ou exclusif bit à bit avec la clé. On a alors

$$M'' = M' \oplus C = (M \oplus C) \oplus C = M \oplus (C \oplus C) = M \oplus 0 = M.$$

En effet, $\oplus$ est associatif et, quelle que soit la clé, un ou exclusif avec elle-même donne la suite de zéros de taille $n$, qui est l'élément neutre de $\oplus$. En le cryptant une seconde fois, on a donc décrypté le message. Bien sûr, cette technique est loin d'être parfaite, car elle demande de fabriquer une clé aléatoire de la même longueur que le message, et surtout que cette clé soit partagée entre l'expéditeur et le destinataire du message.

### Partager un secret : la méthode de Diffie-Hellman

Traditionnellement, en cryptographie, les deux personnes qui veulent communiquer de façon secrète s'appellent Alice et Bob. Leur canal de communication – courrier, Internet… – peut être écouté par Ève. Dans ces conditions, comment Alice et Bob peuvent-ils échanger une clé qui pourra servir à crypter une future conversation ?

L'algorithme de Diffie-Hellman permet de résoudre ce problème. Alice et Bob se mettent d'accord publiquement sur un nombre premier $p$ et un générateur $g$ du groupe $\mathbb{Z}/p\mathbb{Z}$. Alice choisit secrètement un nombre $a$ et Bob secrètement un nombre $b$. Alice calcule $g^a$ modulo $p$ et l'envoie à Bob. De même, Bob calcule $g^b$ et l'envoie à Alice. Une fois ces échanges publics effectués, Alice calcule $(g^b)^a = g^{ab}$ et Bob calcule $(g^a)^b = g^{ab}$. Ils ont donc à leur disposition la même valeur $g^{ab}$ qu'ils pourront ensuite utiliser comme clé.

Quant à Ève, elle a en sa possession $p$, $g$, $g^a$ et $g^b$, valeurs qu'elle a écoutées. Pour que Ève retrouve $g^{ab}$ à partir de ces valeurs, elle doit élever $g^a$ à la puissance $b$ ou $g^b$ à la puissance $a$. Mais déduire $a$ – resp. $b$ – de $g^a$ – resp. $g^b$ –, problème du logarithme discret, est très difficile : nous ne connaissons pas d'algorithme pour le faire rapidement. Ève est donc dans l'impossibilité pratique de calculer la valeur de $g^{ab}$ sans utiliser un temps démesuré.

Mais ce protocole est vulnérable à l'attaque de l'« homme du milieu », Malo, qui suppose un attaquant capable de lire et de modifier tous les messages échangés entre Alice et Bob. Ainsi, si Malo peut modifier les messages,

il peut remplacer $g^a$ par $g^{a'}$. Dans ce cas, Bob calcule $g^{a'b}$ et utilisera cette clé. De même, Malo remplace $g^b$ par $g^{b'}$ et Alice utilisera $g^{ab'}$. Au lieu de communiquer secrètement, Alice et Bob communiqueront, sans le savoir, via Malo, qui pourra décoder et modifier tous les messages supposés cryptés.

### Cryptages asymétriques et RSA

Pour éviter de devoir ainsi partager une clé secrète, on utilise souvent des chiffrements asymétriques – ou à clé publique. L'idée ici est d'avoir deux clés : une clé publique, qui sert à crypter, et une clé privée, qui sert à décrypter. Une métaphore est de mettre à disposition de tout le monde des portes blindées sans poignées et de donner à chacun la poignée de l'une de ces portes. Chacun peut alors facilement mettre un message derrière une porte et la claquer, c'est-à-dire crypter le message avec la clé publique. Désormais, seule la personne qui a la poignée de cette porte peut l'ouvrir et récupérer le message : seule la personne ayant la clé privée correspondante est capable de décrypter le message. Cryptage et décryptage sont ici deux algorithmes distincts.

La cryptographie asymétrique est basée sur l'existence d'une fonction dite à sens unique, pour transformer un message en message codé. Il faut que cette fonction soit simple à appliquer à un message quelconque, mais qu'il soit difficile de retrouver le message original à partir du message codé.

Le principe peut se voir comme un coffre à deux serrures. Lorsque l'une des deux serrures est fermée, la seule façon d'ouvrir la boîte est d'utiliser l'autre serrure. La clé d'une des deux serrures est publique, c'est-à-dire que tout le monde peut l'obtenir, tandis que l'autre est privée, et seule une personne la possède. Par exemple, Bob décide d'envoyer un message secret à Alice qui possède une telle boîte à deux serrures. Dans ce cas, Bob met dans la boîte d'Alice son message et referme la boîte avec la clé publique. Seule Alice pourra ouvrir la boîte, puisqu'elle est seule à posséder la clé privée. Elle seule pourra ainsi lire le message de Bob. Si Ève intercepte le message, elle ne pourra le déchiffrer sans avoir la clé privée d'Alice.

Par ailleurs, la même technique permet de signer un message : Alice le met dans la boîte qu'elle referme à l'aide de sa clé privée. Tous les destinataires pourront alors ouvrir le message et seront certains que celui-ci provient d'Alice, car elle est la seule à posséder la clé privée.

Le but de la cryptologie asymétrique est donc de construire un « coffre à deux serrures » virtuel. Un exemple est la méthode RSA, inventée par Ron Rivest, Adi Shamir et Len Adleman en 1977.

– Avant de recevoir des messages cryptés, Alice doit créer des clés. Pour cela, elle choisit deux nombres premiers $p$ et $q$, un nombre $e$ premier avec $(p-1)(q-1)$ et deux nombres $d$ et $m$ tels que $e \times d + m \times (p-1)(q-1) = 1$ – ces deux derniers nombres se calculent avec l'algorithme de Bezout. Elle calcule enfin le nombre $n = p \times q$.

– Alice rend publics les nombres $n$ et $e$, par exemple en les publiant dans un annuaire, sur un site web ou bien en les communiquant directement à l'expéditeur, Bob. Elle conserve secrètement les nombres $p$, $q$ et $d$, et peut même détruire $p$ et $q$, qui ne serviront plus à rien. Ainsi, la clé publique est constituée par les nombres $e$ et $n$ tandis que la clé privée est $d$.

– Bob, qui veut transmettre une information secrète à Alice, transforme son information en un nombre entier $A$ inférieur à $n$, ou en plusieurs, s'il est trop long.

– Bob élève ensuite son message $A$ à la puissance $e$ modulo $n$. Il envoie ce nombre, que nous notons $B$, de façon non protégée.

– Alice élève alors le nombre $B$ que Bob vient de lui envoyer à la puissance $d$, modulo $n$, qui est sa clé secrète, et obtient le message original que Bob lui a envoyé, car, d'après le petit théorème de Fermat, $B^d = A^{ed} = A$, modulo $n$.

Bob transforme son message en fermant la serrure publique : en élevant le message, éventuellement découpé en plusieurs morceaux, à la puissance $e$ modulo $n$. Une fois cette opération effectuée, le message n'est plus compréhensible. La seule façon de retourner au message initial est de posséder la clé privée $d$. Pour obtenir le message original et ouvrir la serrure privée, il s'agit simplement d'élever le message obtenu à la puissance $d$ modulo $n$.

### Signature

Parfois, on ne souhaite pas crypter le message, mais on souhaite s'authentifier, c'est-à-dire garantir que c'est bien Alice qui a envoyé le message et non pas Ève. Par exemple, la date d'un rendez-vous n'a pas forcément besoin d'être secrète, mais Bob veut être sûr d'avoir rendez-vous avec Alice et pas avec Ève. L'algorithme précédent garantit en prime l'identité de l'envoyeur, mais il est assez coûteux, on veut donc signer un message sans le crypter complètement. Une solution simple est de crypter le condensat du message avec l'une des méthodes précédentes.

Ainsi, Alice envoie son message et le condensat de ce message crypté avec une clé symétrique qu'elle partage avec Bob ou cryptée avec sa clé privée. Bob récupère le message et calcule son condensat. Il vérifie alors à l'aide de la clé commune ou de la clé publique d'Alice que le condensat crypté est bien le condensat du message qu'il a reçu. Non seulement cela authentifie le message

comme venant d'Alice, mais cela authentifie également le message, puisque si Ève avait modifié le message, cela aurait également modifié le condensat et Bob se serait rendu compte de la supercherie. Bob peut donc être sûr de l'auteur et du contenu du message. Cette technique est très peu coûteuse : en effet, calculer le condensat est rapide et le crypter également, puisque c'est un message court.

## Protection des données et persistance de l'information

Discutons rapidement quelques aspects moins techniques de la structuration et du contrôle de l'information.

**Persistance de l'information** Lorsque nous stockons une donnée dans un fichier ou dans une base de données, nous comptons sur le fait qu'elle reste accessible sans limite.

Dans le cas d'un fichier, la gestion de la persistance des données est donc confiée au système d'exploitation, mais la préservation de la structure du fichier reste aux risques et périls de l'utilisateur – un logiciel mal conçu peut faire une erreur en ajoutant une donnée à un fichier et détruire l'ensemble des données de ce fichier, devenu illisible.

Dans le cas d'une base de données, on veut éviter ce type de problèmes. Le système de gestion de la base de données doit pouvoir redémarrer dans un état cohérent : celui dans lequel il était avant le début de cette opération. Pour cela, on doit prévoir, avant chaque opération, une sauvegarde des données, à laquelle il est possible de revenir en cas de problème. Pour se protéger, en même temps, des risques de disparition des données due par exemple à un incendie – accidentel ou intentionnel – ou une inondation des locaux dans lesquels se trouve l'ordinateur sur lequel ces données sont stockées, on effectue souvent cette sauvegarde, par exemple sur une machine distante.

**Hypermnésie du Net** Mais cette persistance de l'information a aussi un aspect moins désirable. Une fois qu'une information est stockée sur un disque dur, sauvegardé régulièrement, ou sur une page web publique – et donc immédiatement archivée par plusieurs moteurs de recherche, comme Google –, il devient impossible de l'effacer, car il restera toujours de nombreuses copies, ici ou là.

Recopier un bit d'information coûte en effet très peu d'énergie. En théorie, il semble que la borne inférieure soit $kT\ln(2)$ soit de l'ordre de $10^{-21}$ joules. En pratique, l'énergie dissipée par le basculement d'un transistor est de l'ordre de $10^{-14}$ joules, c'est beaucoup plus, mais cela reste très petit devant, par exemple, les 9.81 joules qu'il faut pour monter un kilogramme d'un mètre. Il ne faut donc pas s'étonner que de telles copies soient fréquentes.

Il faut être conscient du fait que, lorsqu'une photo ou un texte est publié sur la page d'un réseau social, elle peut être consultée non seulement par beaucoup de gens, mais il est presque impossible de l'enlever. Cette photo ou ce texte sera peut-être encore accessible dans cent ans.

Quand un site recopie des informations depuis un autre site, une « règle de bonne conduite » devrait être de supprimer l'information, quand elle est supprimée sur le site original, de manière à limiter la persistance de l'information. Certains imaginent que la loi devrait garantir un « droit à l'oubli », en interdisant de conserver des informations au-delà d'un certain délai, ou, au moins, en encadrant la manière dont elle est conservée.

**Protection de la vie privée** Bien que les outils juridiques de protection de la vie privée existent depuis plus de trente ans, il n'est pas toujours facile de faire table rase de son passé numérique. L'enjeu est de taille, d'autant que l'exploitation de ces données personnelles représente un potentiel inestimable sur lequel reposent de nombreux modèles économiques sur le Net. Des solutions techniques et juridiques existent, mais aucune n'est parfaite.

Les solutions techniques sont fondées soit sur la destruction des données, soit sur une anonymisation irréversible : mode de navigation privé, possibilité pour l'utilisateur de définir lui-même la durée de conservation de certaines informations…

Du côté des solutions juridiques, la loi Informatique et Liberté nous protège contre l'utilisation abusive de bases de données puisque, selon la loi, les données « sont collectées pour des finalités déterminées, explicites et légitimes, et ne sont pas traitées ultérieurement de manière incompatible avec ces finalités » et « sont conservées sous une forme permettant l'identification des personnes concernées pendant une durée qui n'excède pas la durée nécessaire aux finalités pour lesquelles elles sont collectées et traitées ».

Comme toute loi, celle-ci nous protège, mais elle nous contraint aussi, par exemple, à déclarer à la CNIL nos propres bases de données.

**Non-rivalité de l'information** L'information a cette propriété que la consommation du bien par un agent n'empêche pas sa consommation par un autre – les agents ne sont pas rivaux pour la consommation du bien. Par exemple, le fait que quelqu'un écoute la radio n'empêche pas les autres de le faire. De même, sur le Web, il y a non-exclusion : tous les agents ont librement accès au bien.

La manière de diffuser des biens non rivaux est différente de la manière de diffuser des biens rivaux. Par exemple, la gratuité pour les biens rivaux entraîne souvent la pénurie, mais ce n'est pas le cas pour des biens non rivaux. Quand des biens rivaux deviennent non rivaux et gratuits, logiciels, œuvres d'art, etc.,

il faut souvent repenser entièrement la manière dont leurs producteurs sont rémunérés.

## Exercices corrigés et commentés

### Exercice 1. Comprendre ce que l'on tape dans la barre de navigation

En utilisant l'entrée de Wikipédia
`http://fr.wikipedia.org/wiki/Uniform_Resource_Locator`
rendre explicite chaque élément d'un URL (*Uniform Ressource Locator*, Localisateur uniforme de ressource) qui permet d'accéder à une ressource sur Internet. Illustrer avec l'exemple l'URL ci-dessous.

```
http://me:maux2passe@mabanque.fr/compte/retrait
?montant=10000&beneficiaire=toi
```

**Correction.** *Méthode*. Il se trouve que l'on ne peut pas toujours tout savoir. On ne peut même pas toujours savoir où tout trouver.

En posant un tel type d'exercice, on invite les élèves à aller à la recherche de connaissances qui leur permettent de comprendre ce qu'ils tapent ou cliquent. Dans ce cas, chacun tape quotidiennement de telles chaînes de caractères, dont on voit confusément qu'elles doivent avoir un sens caché – on y reconnaît probablement `.fr` pour France, on devine que `http` doit être un standard, etc. Il suffit de communiquer un URL à une personne qui connaît le codage sous-jacent et à une autre qui ne le connaît pas pour mesurer la différence : la première a juste besoin de mémoriser les éléments particuliers à l'URL, la seconde doit le noter caractère par caractère, avec l'angoisse de se tromper.

En posant un tel type d'exercice, on invite l'élève à aller à la recherche de connaissances qu'il ne connaît certes pas encore, mais qu'il ne sait pas non plus forcément comment chercher. Pour ce qui concerne l'informatique et les mathématiques qui y sont liées, la démarche est le plus souvent simplifiée par Wikipédia. Le premier geste n'est plus d'aller chercher sur Google, mais de demander à Wikipédia. La version française est bien fournie, la version anglaise est parfois plus riche. Les liens internes et externes de bas de page y sont précieux.

Mais ce n'est pas le seul recours. Le *Site du zéro* `http://www.siteduzero.com` – où tout est expliqué… à partir de zéro – est très riche de contenus plus proches de l'informatique technique, tandis que *Interstice* `http://interstices.info` – où l'informatique devient une science… à portée de clic – est un site précieux de ressources culturelles. Un site de ressources documentaires pour les enseignants `http://science-info-lycee.fr` va compléter ces grands silos de contenus.

Comment passer concrètement à l'appropriation de connaissance ? Peut-être en « remalaxant » l'information. À l'ère où l'on zappe et survole souvent les choses, le moyen de ne pas en rester à « j'ai déjà vu ça quelque part » est sûrement de mettre ses propres mots dessus, faire *son* résumé des faits et données collectées. On pense avec des mots. On peut aussi penser avec des éléments structurés, rédigés sous forme de notes écrites. La seule règle à s'imposer : ne pas utiliser le copier-coller, qui empêche de s'approprier le sens du texte.

*Contenu*. Un URL est une chaîne de caractères structurée combinant les informations nécessaires pour indiquer à un logiciel comment accéder à une ressource Internet et qui se décompose ainsi :

```
URL : $protocole ://$connection(/$chemin($paramètres)?)?
```

– Le *protocole* de communication, `http` pour le Web, spécifie les règles sémantiques et syntaxiques pour initier, paramétrer, effectuer et contrôler un échange automatique de données entre deux systèmes d'information.

– La *connexion* définit quel serveur connecter, la spécification complète étant : `connection : ($login(:$mot-de-passe)?@)?($nom-du-serveur|$adresse-ip)(:$port)?` permet de spécifier un identifiant `login` avec éventuellement un mot de passe `mot-de-passe` – qui est transmis en clair, donc à un niveau de sécurité assez bas –, de spécifier le serveur sous son nom de domaine, le nom du serveur ou son adresse Internet, son adresse IP, voire de préciser le numéro de port TCP/IP à utiliser, au cas où le même serveur possède des services n'utilisant pas le port par défaut pour le protocole de communication.

Le nom du serveur est lui-même structuré :

```
nom-du-serveur : ($nom(.$nom-de-sous-domaine)*.$nom-du-domaine)
```

où les noms de domaine constituent une énumération standardisée :

```
nom-du-domaine : (info|org|com|fr|…)
```

pour les sites d'information `*.info`, de grandes organisations `*.org`, commerciaux `*.com`, français `*.fr`, etc.

– Le *chemin* est de la forme :

```
chemin : (./|(../)+)?($repertoire/)*($nom-de-base($extension)?)?
```

et permet de localiser la ressource au sein du serveur, exactement comme dans un système de fichiers. C'est un chemin à travers un arbre. L'extension est étroitement liée au type de la ressource, par exemple pour un document bureautique `.doc` ou `.odt`, pour une image `.png` ou `.jpg`, etc.

– Les *paramètres* se spécifient de deux manières :

```
paramètres : (#fragment|[?]requête)
```

soit sous forme de `fragment`, l'identificateur faisant référence à un identificateur à l'intérieur du document, soit sous forme de `requête`

```
requête : nom=valeur([&]nom=valeur)*
```

qui est un n-uplets à champs nommés permettant de paramétrer la requête. On peut par exemple référencer une page précise d'un pdf avec une construction de la forme `http://mon-serveur/le-chemin/le-document.pdf#page=11`, ou interroger une base de données, comme cela sera détaillé au chapitre suivant.

Au-delà du protocole `http`, on peut accéder à un fichier local par le protocole `file`, ou via d'autres protocoles comme le `ftp`.

Plus généralement, un URI (*Uniform Ressource Identifier*, Identificateur uniforme de ressource) permet de spécifier une ressource, en utilisant un protocole de communication autre que `http`. Ainsi

```
mailto:prenom.nom@serveur?subject=mon-sujet&body=Bonjour
```

définit un message à envoyer à une adresse électronique. L'accès à des services Internet peut se faire en définissant des URI.

### Exercice 2. Chiffrement par substitution

Chiffrer par le codage de César et en ROT13 la phrase :

**Bob veut envoyer un message secret a Alice.**

### Correction

**Bob veut envoyer un message secret a Alice.**

César Ere yhxw hqyobhu xq phvvdjh vhfuhw d Dolfh.

ROT13 Obo irhg raiblre ha zrffntr frperg n Nyvpr.

### Exercice 3. Chiffrement par ou exclusif

On considère le chiffrement symétrique basé sur l'opérateur ou exclusif. Que penser des clés 0…0 et 1…1 ?

Crypter avec la clé composée de la représentation binaire de π le message « Salut » crypté en ASCII-7 bits.

### Correction

Considérons un message $M$. Le cryptage avec la clé nulle donne le message initial : $M \oplus 0\ldots0 = M$. Le cryptage avec la clé 1…1 donne $M \oplus 1\ldots1 = M$, le complément, bit à bit, de $M$. Ce sont donc deux clés extrêmement faibles.

| Message | S | a | l | u | t |
|---|---|---|---|---|---|
| ASCII | 1010011 | 1100001 | 1101100 | 1110101 | 1110100 |
| Clé : $\pi$ | 1100100 | 1000011 | 1111101 | 0001010 | 1011010 |
| Message crypté | 0110111 | 0100010 | 0010001 | 1111111 | 0101110 |

## Exercices non corrigés

### Exercice 1

Définir un format de donnée symbolique correspondant aux données permettant de s'inscrire sur `facebook` – ou toute autre plateforme du Web. Définir celui correspondant à un rendez-vous dans un agenda. Définir celui correspondant aux métadonnées d'un album de photos de vacances.

### Exercice 2

En analysant les entrées de Wikipédia `http://fr.wikipedia.org/wiki/OpenDocument` et `http://fr.wikipedia.org/wiki/DocBook`, expliquer en deux ou trois lignes comment ces deux standards s'articulent.

### Exercice 3

Prendre un fichier de grande taille et appliquer plusieurs fois un mécanisme de compression, avec des commandes de type :

```
gzip -9 mon-fichier; mv mon-fichier.gz mon-fichier; ls -l mon-fichier
```

et observer que la taille du fichier va, en général, diminuer, puis croître légèrement, avant de se stabiliser. Expérimenter s'il est toujours possible de retrouver exactement le fichier originel en faisant les opérations inverses de décompression.

### Exercice 4. Inexistence d'une méthode de compression universelle

Un compresseur sans perte peut être vu comme une injection de l'ensemble des fichiers dans lui-même. Montrer qu'il est impossible qu'une telle fonction compresse strictement tous les fichiers d'une taille donnée.

Indication : considérer les puissances itérées de cette fonction.

Plus difficile : montrer que si une fonction compresse strictement au moins un fichier, alors il existe un fichier qu'elle allonge strictement.

### Exercice 5

Écrire un code qui construit l'arbre du codage d'Huffman au fur et à mesure de l'arrivée d'un flux de caractères et l'appliquer à un mécanisme de compression-décompression. Pour simplifier la programmation, on pourra « tricher » en utilisant l'arbre du compresseur dans le décompresseur pour valider le reste de l'algorithme. Puis, on pourra coder deux algorithmes indépendants et vérifier que les arbres de codages restent bien identiques.

### Exercice 6

En utilisant l'entrée `http://fr.wikipedia.org/wiki/Indicatif_téléphonique_international` expliquer en quoi nous sommes devant un code préfixe. Si l'on regarde les préfixes des grandes nations émergentes, ce code est-il optimal ?

### Exercice 7

Étude du code de Reed-Solomon. Prenons trois nombres $u$, $v$, $w$ et transmettons $u$, $v$, $w$ avec $s = u + v + w$ et $m = u + 2v + 3w$. À la réception, en supposant que au plus un des nombres est erroné :
– calculer à partir de $s$ la valeur de cette erreur $e$ ;
– déterminer, à partir de $m$ et $e$, lequel des trois nombres est en erreur.

En déduire l'algorithme de correction.

## Questions d'enseignement

À ce stade de notre parcours, nous avons assez de recul pour proposer brièvement quelques clés pédagogiques sur l'enseignement de l'informatique en général.

*L'informatique se prête à une pédagogie participative*, avec un enseignement par mini-projets, orienté vers le travail en groupe. Apprendre à programmer un petit logiciel, c'est donner à l'élève des clés, mais aussi la liberté de s'approprier ces clés et de les mettre en pratique de manières diverses, car il y a plusieurs manières de mettre en œuvre une solution. Ce n'est pas la seule matière qui offre ce levier, mais c'est un cas exemplaire.

*L'informatique favorise l'apprentissage par l'utilisation*, ce qui correspond bien à l'esprit humain – par exemple : découvrir un algorithme avant d'en abstraire la notion sous-jacente. L'enseignement scolaire de l'informatique démarre bien sûr par l'apprentissage des usages de logiciels, mais, dès cette étape, il est important de comprendre les concepts sous-jacents à l'utilisation de ces logiciels.

*Le levier pédagogique est l'apprentissage de la programmation*. Apprendre à programmer permet de faire le lien entre pratique et théorie. Programmer les algorithmes permet non seulement de vérifier que les savoirs sont compris, mais également d'entrevoir toutes les potentialités de l'informatique. Voir François Élie, Bastien Guerry, Dominique Lacroix, Philippe Lucaud, Charlie Nestel, Cécile Picard-Limpens, Thierry Viéville, « Est-il besoin de savoir programmer pour comprendre les fondements de l'informatique ou utiliser les logiciels ? », *Revue de l'EPI*, a1012b.

*L'informatique conduit aussi à un apprentissage de la rigueur*, par un mécanisme spécifique : celui des essais-erreurs avec une machine neutre qui ne donnera un résultat que si tout est correct, mais qui donnera indéfiniment une chance de corriger, de reprendre, de tester à nouveau. La machine est un outil pour apprendre de manière incrémentale, sans jamais porter de jugement de valeur.

*L'informatique permet d'entrevoir l'intérêt des sciences théoriques*. On peut toucher – opérer avec, visualiser… – des objets abstraits.

*L'informatique est un levier pour les autres sciences*, car elle aide à mieux comprendre des notions universelles – par exemple la notion d'information – ou fondamentales – par exemple le calcul mécanique, par opposition à d'autres formes de raisonnement. L'informatique offre aussi la découverte de notions nouvelles, par exemple les suites aléatoires, les définitions récursives, etc.

C'est en apprenant l'informatique le plus tôt possible que l'on tirera le meilleur profit de son rôle transversal à la quasi-totalité des autres disciplines.

Les enjeux de société sont évidents : donner aux futurs citoyens les clés du monde numérique, pour leur permettre de se positionner de manière éclairée sur les choix de société posés par ces nouvelles technologies, par exemple découvrir comment les nouvelles technologies aideront à relever les grands défis qui se posent à nous. Mais ces questions de société sont toujours ancrées dans des questions scientifiques et techniques, dont elles ne doivent pas être séparées.

Les enjeux en termes de formation de l'esprit – en lien avec l'enseignement de philosophie – sont aussi fondamentaux, les éléments de ce chapitre et du premier chapitre le montrent assez.

Ces éléments s'opposent aussi à quelques idées reçues :

*L'informatique n'est pas une science, juste de la technique ?* Ce qui fait marcher les programmes est bien de la théorie : algorithmique, théorie des langages de programmation, etc. Ce sont des logiciens et des physiciens théoriques du siècle dernier qui sont à l'origine de l'informatique. Les questions de sûreté sont exemplaires : c'est de l'étude scientifique des logiciels qu'émergent aujourd'hui les méthodes permettant de concevoir des logiciels sûrs.

*Pas besoin d'apprendre l'informatique, cela s'apprend tout seul ?* On dit souvent : ce sont les enfants qui apprennent l'informatique aux enseignants ! Cette double idée reçue est la conséquence d'une confusion entre l'apprentissage des usages et l'apprentissage des fondements. Cette idée reçue néglige aussi le fait que, depuis quarante ans, l'informatique s'est stratifiée et complexifiée : on ne peut plus y bricoler. Le mythe de l'auto-apprentissage se brise devant la nécessité d'apprendre au plus grand nombre des savoirs et des pratiques qui doivent être intégrés à l'échelle d'une société entière. Le risque de mal apprendre et de devoir passer des heures à se corriger, les risques liés aux mauvaises méthodes – perte de données, logiciels non fiables… – deviennent majeurs : l'enseignement de l'informatique en tant que matière rigoureuse est une nécessité.

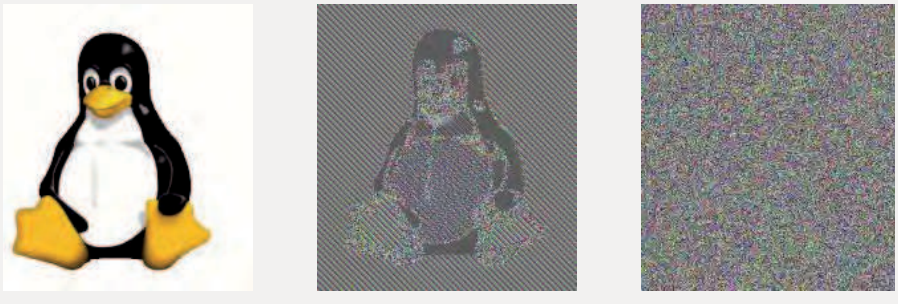
Et nous ne pouvons pas ne pas noter aussi :

*L'informatique n'est pas très féminine*. C'est bien la mixité des genres qui aide à créer la mixité et l'ouverture des idées… et l'informatique a quelques grands noms féminins : Ada Lovelace – premières notions de programmes et de traitement symboliques –, Emmy Noether – processus algébriques permettant de mécaniser des calculs –, Grace Hopper – langage COBOL, mémoires tampons –, Rose Dieng – ontologies informatiques pour le Web sémantique –, même si la situation reste déséquilibrée en France, contrairement à d'autres pays, comme la Thaïlande, où 65 % des étudiants et des professeurs en informatique sont des étudiantes et des professeures.

## Compléments

Une question que nous n'avons pas abordée est celle du codage de messages longs. En général, les techniques sont adaptées au codage de blocs de quelques centaines/milliers de bits. Mais on peut vouloir crypter un fichier plus long, comme une vidéo. Dans ce cas, la méthode naturelle est celle du dictionnaire de codes : ECB (*Electronic codebook*). Elle consiste à découper le message en blocs, à chiffrer chaque bloc avec la méthode et la clé choisies, à recoller les morceaux chiffrés et à envoyer le tout. Le receveur pourra alors redécouper en blocs, les décrypter et recoller les bouts obtenus. Ce cryptage par bloc est en fait facilement attaquable : deux blocs avec le même contenu seront chiffrés de la même manière. On peut donc tirer des informations à partir du texte chiffré en cherchant les séquences identiques. On obtient dès lors un « dictionnaire de codes » avec les correspondances entre le clair et le chiffré.

Voici un exemple frappant en image issu de Wikipédia. L'image de gauche a été cryptée en utilisant la méthode ECB décrite précédemment pour obtenir l'image du milieu. Une autre méthode a été utilisée pour obtenir l'image de droite.



*Cryptage d'une image par la méthode du dictionnaire de codes (ECB) ou par une autre méthode.*

Pour plus de détails et la description d'autres méthodes, on pourra se référer à Mode d'opération (cryptographie).

`http://fr.wikipedia.org/wiki/Mode_d'opération_(cryptographie)`

## Pour aller plus loin

Sur le cryptage : François Cayre, Cryptographie, du chiffre et des lettres
`http://interstices.info/crypto-chiffre-lettres`
)i(nterstices, 2009.

Sur les codes secrets : Mathieu Cunche, À l'attaque des codes secrets
`http://interstices.info/codes-secrets`
)i(nterstices, 2011.

Sur le cryptage RSA : Jonathan Touboul, Nombres premiers et cryptologie l'algorithme RSA
`http://interstices.info/rsa`
)i(nterstices, 2007.

Sur la didactique de l'informatique en général, Werner Hartmann, Michael Näf, Raimond Reichert, *Informatikunterricht planen und durchführen*, Planifier et réaliser un cours d'informatique, Springer Verlag, 2006, dont une traduction française a été publiée en octobre 2011, est une référence.