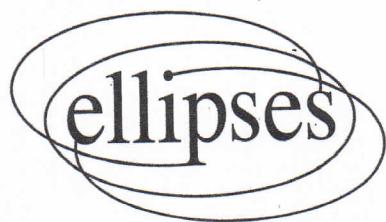


Références sciences

L'essentiel de l'informatique en prépa

Exemples, synthèses et exercices corrigés
en Python et SQL

Frantz Barrault



Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr



ISBN 9782340-011496

©Ellipses Édition Marketing S.A., 2016
32, rue Bargue 75740 Paris cedex 15



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 une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Avant-propos

À qui s'adresse ce livre ?

Ce livre est issu d'un cours distribué à des élèves de CPGE scientifique. Il leur est donc naturellement destiné. Cet ouvrage s'adresse aussi à des élèves de terminale S motivés par des études supérieures scientifiques, à leurs enseignants s'ils désirent illustrer l'algorithmique de façon concrète. Enfin, ce livre s'adresse à toute personne désirant comprendre comment fonctionne un ordinateur et apprendre à programmer.

Que contient ce livre ?

Les notions abordées sont celles du programme « Informatique pour tous » des classes préparatoires scientifiques. Cet ouvrage contient donc des informations culturelles sur l'informatique et des bases pour la programmation dans les langages Python et SQL. Ce sont deux langages de programmation relativement simples à aborder, y compris pour le néophyte. Ce livre devrait servir d'aide-mémoire détaillé et d'outil de référence à poser à côté du clavier.

Comment utiliser ce livre ?

Ceux pour qui Python n'évoque qu'un serpent

Vous apprendrez alors que Python est également un langage de programmation à partir du chapitre 4 par lequel vous pouvez commencer, même si la lecture des trois premiers chapitres permet de mieux comprendre comment fonctionne un ordinateur et en particulier comment il conçoit un nombre. Lisez attentivement, reprenez les exemples, entraînez-vous sur des exercices, c'est la clé du progrès et de la maîtrise du Python.

Si Python ou le SQL vous sont déjà familiers

Alors le sommaire et la table des matières, volontairement très complets, vous permettront de trouver très rapidement l'information recherchée. Les exercices s'adressent aussi bien aux étudiants de première que de seconde année et les sujets peuvent servir d'entraînements aux concours. Je me permets d'insister sur l'importance de passer du temps à écrire des scripts Python ou SQL.

Doit-on lire ce livre dans l'ordre ?

La réponse est non. Ce livre est divisé en quatre parties. Seule la troisième partie est dépendante de la précédente. Les quatre parties sont réparties ainsi :

- Les chapitres 1 à 3 sont une introduction à l'informatique. Ils seront lus de préférence dans cet ordre même s'ils sont relativement indépendants. Le chapitre 3 est plus technique.
- Les chapitres 4 à 10 traitent des bases de la programmation en Python. Il est quasi nécessaire de les lire et de les travailler dans cet ordre.
- Les chapitres 11 à 13 donnent des bases sur la programmation scientifique en Python.
- Les chapitres 14 à 16 concernent les bases de données et un langage associé, le SQL. Ces chapitres, à lire dans cet ordre, sont indépendants du reste de l'ouvrage.

Documents numériques accompagnant cet ouvrage

Des fichiers accompagnant certains exercices ainsi que des compléments à cet ouvrage sont disponibles sur le site suivant :

<http://frantzbarrault.wordpress.com/fichiers-etudiant/>

Tous ces documents sont téléchargeables afin de faciliter l'apprentissage.

Remerciements

Je tiens à remercier pour leur patient travail de relecture, pour les échanges fructueux et constructifs, pour leur conseils, leurs idées et leurs connaissances mes amis et collègues suivants, dont l'ordre d'apparition est alphabétique et non pas tiré au sort par la fonction `shuffle` du module `numpy.random` : Thomas BARUCHEL, Emmanuel CHASTEL, Philippe FICHOU, Frédérique HUTEAU, Aurélien MONTEILLET, Aude RIVIERE et Émilie VIGIER. Merci également à Yvon LE CORRE et Vincent ABHERVÉ pour leur amitié, leur aide et leur disponibilité et enfin à Jean-Michel FERRARD pour la cordialité de nos échanges.

Je remercie les éditions Ellipses ainsi que Paul DE LABOULAYE pour m'avoir accordé leur confiance.

Merci à Ghislaine qui m'a encouragé et soutenu de bout en bout dans ce projet.

Merci enfin à Maxence et Raphaël, jeunes chevaliers Jedi, qui trop souvent trouvaient leur papa sur une autre planète.

Sommaire

1 Brève histoire de l'informatique	7
1.1 La mécanisation des calculs	7
1.2 Avancées logiques et premiers « ordinateurs »	7
Exercices	9
2 Architecture matérielle et logicielle	11
2.1 Le boîtier	11
2.2 Quelques ports fréquents	12
2.2.1 Port Com1	12
2.2.2 Port PS/2	12
2.2.3 Port VGA	13
2.2.4 Port DVI	13
2.2.5 Port HDMI	13
2.2.6 Port USB	13
2.2.7 Port Réseau RJ45	14
2.3 La carte mère	14
2.3.1 Le chipset	15
2.3.2 BIOS et CMOS	15
2.3.3 Le microprocesseur	16
2.4 Les mémoires	18
2.4.1 La mémoire vive	18
2.4.2 Les mémoires mortes	18
2.4.3 Les mémoires de masse	18
2.5 Les BUS	19
2.6 Les systèmes d'exploitation	19
Exercices	20
3 Représentation des nombres	21
3.1 Le binaire	21
3.1.1 Introduction	21
3.1.2 Notation	22
3.1.3 Systèmes de numération pondérée en base 10, 2, 8, 16 et 64	22
3.2 Représentation des nombres dans un ordinateur	24
3.3 Représentation des entiers naturels	25
3.4 Représentation des entiers relatifs	25

3.5	Représentation des nombres à virgule	27
3.5.1	Représentation d'un nombre à virgule flottante suivant la norme IEEE754	27
3.5.2	Quelques nombres particuliers	28
3.6	Les limites des représentations des nombres	29
3.6.1	Il y a un nombre fini de nombres représentables	29
3.6.2	Des nombres non représentables	29
3.6.3	Les problèmes d'arrondis	29
	Exercices	30
4	Premiers pas avec Python	33
4.1	Environnements de travail et liens utiles	33
4.2	Python comme calculatrice	34
4.2.1	Premiers calculs	34
4.2.2	Une petite astuce et une remarque	35
4.3	Affectation, modification et initialisation des variables	35
4.3.1	Affectation, lecture et modification	35
4.3.2	Variables : quelques trucs et astuces	36
4.3.3	Le nom des variables	37
4.4	Importer un module	38
4.4.1	Comment importer un module	38
4.4.2	Obtenir de l'aide sur un module ou une fonction	38
4.4.3	Des modules à connaître	39
	Exercices	43
5	Les différents types de données	45
5.1	Les types numériques	45
5.1.1	Les entiers	45
5.1.2	Les nombres flottants	45
5.1.3	Les nombres complexes	46
5.1.4	Les nombres décimaux	46
5.1.5	Les entiers dans différentes bases	46
5.2	Les chaînes des caractères	46
5.3	Des chaînes aux nombres et des nombres aux chaînes	47
5.4	Un dernier type d'objets, les booléens	47
5.4.1	Notion de booléens	47
5.4.2	Opérations liées aux booléens	47
5.5	Les types de données en algorithmique	49
5.6	Égalité structurelle et égalité physique	49
	Exercices	50
6	Initiation à la programmation en Python	51
6.1	Entrée au clavier (<code>input</code>) et affichage à l'écran (<code>print</code>)	51
6.1.1	La fonction <code>input</code>	51
6.1.2	La fonction <code>print</code>	52
6.1.3	<code>print, input et algorithmique</code>	53

6.2	Blocs d'instructions et indentations	53
6.3	Structures conditionnelles	54
6.3.1	Forme minimale avec <code>if</code>	54
6.3.2	Forme complète avec <code>if, elif et else</code>	54
6.3.3	Une autre utilisation de <code>if</code>	55
6.4	Boucles inconditionnelles et intervalles d'entiers	55
6.4.1	Les boucles <code>for</code>	55
6.4.2	La fonction <code>range</code>	56
6.4.3	Petit complément avec <code>continue</code> et <code>break</code>	57
6.5	Boucles conditionnelles (boucles <code>while</code>)	58
6.5.1	Exemples et pratique	58
6.5.2	Résumé, remarques et mise en garde	59
	Exercices	59
7	Les fonctions en Python	61
7.1	Le cas particulier des fonctions <code>lambda</code>	61
7.2	Les fonctions	62
7.2.1	Les fonctions déjà rencontrées	62
7.2.2	Créer ses propres fonctions	62
7.2.3	Renvoyer une valeur avec <code>return</code>	63
7.2.4	Les paramètres optionnels d'une fonction	63
7.2.5	Portée des variables	64
7.2.6	Les exceptions	65
7.2.7	Fonctions récursives	66
	Exercices	67
8	Les listes et les tuples	69
8.1	Création de listes	69
8.2	Accès aux éléments d'une liste	70
8.3	Opérations classiques sur les listes	70
8.3.1	Modifier, ajouter, supprimer, ordonner des éléments	71
8.3.2	Parcours d'une liste avec <code>for</code>	71
8.3.3	Trouver un élément particulier dans une liste	72
8.3.4	Concaténation de listes	72
8.3.5	Listes en compréhension	72
8.3.6	Des chaînes aux listes	72
8.4	Les tuples	73
	Exercices	73
9	Les chaînes de caractères	75
9.1	Les chaînes, une séquence comme les listes	75
9.2	Parcourir, modifier, ajouter, retirer des caractères	76
9.2.1	Parcours d'une chaîne	76
9.2.2	Caractère non mutable d'une chaîne	76
9.2.3	Des méthodes particulières pour les chaînes de caractères	76
	Exercices	78

10 Les fichiers	81
10.1 Pourquoi utiliser des fichiers ?	81
10.2 Notion de répertoire courant	81
10.2.1 Connaître et modifier le répertoire courant	81
10.2.2 Chemin relatif et chemin absolu	82
10.3 Travailler avec un fichier	82
10.3.1 Ouverture et fermeture d'un fichier	83
10.3.2 Lire le contenu d'un fichier	83
10.3.3 Écrire dans un fichier	84
Exercices	84
11 Le module NumPy	87
11.1 Introduction	87
11.2 Importation du module	87
11.3 Création d'un tableau	88
11.3.1 Création d'un tableau avec la fonction array	88
11.3.2 Création de tableaux particuliers	88
11.4 Dimensions d'un tableau	89
11.5 Lecture et écriture de valeurs dans un ndarray	89
11.6 Les fonctions arange, linspace et reshape	90
11.6.1 La fonction arange	90
11.6.2 La fonction linspace	91
11.6.3 La fonction reshape	91
11.7 Un sous-module pour les tableaux à valeurs pseudo-aléatoires	92
11.8 Les fonctions avec NumPy	93
11.8.1 Notion de fonction universelle	93
11.8.2 Opérations arithmétiques	93
11.8.3 Fonctions universelles classiques	94
11.9 Calcul matriciel	94
11.9.1 Combinaisons linéaires	94
11.9.2 Trace	94
11.9.3 Transposition	94
11.9.4 Produit matriciel et produit scalaire	95
11.9.5 Déterminant, inversion et résolution de systèmes	95
11.9.6 Normes matricielles et vectorielles	96
11.9.7 Valeurs propres et vecteurs propres	96
Exercices	96
12 Calcul scientifique avec SciPy	97
12.1 Résolution d'équations	97
12.1.1 Brève introduction	97
12.1.2 La méthode de la dichotomie	97
12.1.3 La méthode de Newton	99
12.1.4 Résolution d'une équation avec NumPy et scipy.optimize	100
12.2 Résolution d'équations différentielles	101
12.2.1 Introduction	101

Sommaire

12.2.2 La méthode d'Euler	101
12.2.3 Coder la méthode d'Euler « à la main »	102
12.2.4 Résolution des équations différentielles avec SciPy	103
12.2.5 Équations différentielles d'ordre deux avec odeint	104
12.3 Résolution d'un système linéaire	106
12.3.1 Méthode de Gauss	106
12.3.2 Pourquoi échanger deux lignes ? La recherche d'un bon pivot	108
12.3.3 La version matricielle du pivot de Gauss	108
12.3.4 Pourquoi les matrices de permutations sont-elles « rares » ?	110
12.3.5 Complexité de la méthode de Gauss	110
12.3.6 Le déterminant et la méthode de Cramer	111
12.4 Calcul intégral	111
Exercices	112
13 Le module matplotlib	115
13.1 Introduction	115
13.2 Importation du module	115
13.3 Les fonctions plot et show pour afficher des courbes	115
13.4 Plusieurs graphes dans une figure	115
13.5 Exemples d'autres fonctionnalités	117
13.5.1 Un titre	117
13.5.2 Des flèches et des annotations	117
13.5.3 Une légende	118
13.5.4 Un quadrillage	119
13.5.5 Les ticks et les axes	119
13.5.6 Modifier les dimensions	120
13.5.7 Des couleurs	120
Exercices	120
14 Introduction aux bases de données	121
14.1 Introduction	121
14.2 Vocabulaire	122
14.3 Les types de données en SQL	124
14.4 En pratique	124
Exercices	124
15 Les bases du langage SQL	125
15.1 Brève introduction à l'algèbre relationnelle	125
15.2 Sélection des données	126
15.2.1 La requête SELECT	126
15.2.2 La clause WHERE	128
15.2.3 La valeur NULL	129
15.2.4 Définir plusieurs conditions avec des booléens	130
15.2.5 Recherche entre deux valeurs avec BETWEEN	131
15.2.6 Tester la présence d'une valeur dans une liste avec IN	132
15.2.7 Le tri des données avec ORDER BY	132

15.2.8 Éliminer les doublons et restreindre le nombre de résultats	133
15.2.9 Rechercher dans une chaîne de caractères avec LIKE	134
15.3 Les fonctions d'agrégation et de regroupement	135
15.3.1 Introduction	135
15.3.2 Les fonctions d'agrégation	136
15.3.3 Regroupements avec GROUP BY	137
15.3.4 Rajouter une condition avec HAVING	138
15.4 Compléments d'algèbre relationnelle	139
15.4.1 Clé primaire	139
15.4.2 Opérations ensemblistes	139
Exercices	141
16 Jointures, produit cartésien et sous-requêtes	143
16.1 Produit cartésien	143
16.2 Jointure	144
16.3 Sous-requêtes	147
16.3.1 Renommage d'une partie d'une table	147
16.3.2 Création d'une liste utilisée avec le prédictat IN	147
16.3.3 Sous-requêtes renvoyant une seule valeur	148
Exercices	148
17 Sujets d'étude	149
18 Solutions des exercices et sujets	153
Annexe A Installer Python et SQL	177
Annexe B Spyder	179
Bibliographie	181
Index	183

Chapitre 1

Brève histoire de l'informatique

1.1 La mécanisation des calculs

Plongez-vous un instant dans la peau d'un riche propriétaire de trois troupeaux de chèvres et moutons en Mésopotamie il y a cinq mille ans. L'un de vos bergers, analphabète, part estiver six mois. Comment être certain qu'il reviendra avec le même nombre d'animaux (sans en vendre quelques-uns) ? En modelant en argile une sphère creuse contenant des jetons de différentes formes qui permettent de dénombrer le troupeau. Codage, mémoire et premiers algorithmes (car les sumériens savaient calculer) sont déjà nés !

Les **abaques** (tables sur lesquelles étaient posées des petites pierres) ont été utilisés par les civilisations européennes, indiennes, chinoises et mexicaines. Les plus anciens datent d'environ -500 av. J.-C. Rappelons que le mot **calcul** vient du latin *calculus* qui signifie caillou. Créé à la même époque, le **boulier** est toujours utilisé en Chine et au Japon.

Une étape historique a été la construction de la *pascaline* au XVII^e siècle, machine inventée par **Blaise Pascal** qui effectue les quatre opérations arithmétiques classiques. Une première machine à calculer **programmable** (technologie inspirée de celle des métiers à tisser) est inventée en 1834, mais jamais réalisée à son époque par **Charles Babbage**. Les programmes, écrits sur des cartes perforées, technique utilisée jusqu'au milieu des années 1980, sont inventés par la mathématicienne **Ada Lovelace**, qualifiée de « première programmeuse au monde ». Dès le début du XX^e siècle, des entreprises naissantes comme IBM ou Bull, ont créé des machines (mécaniques) enregistreuses, additionneuses et multiplicatrices. Elles furent vendues par milliers aux entreprises et administrations. À la fin du XIX^e siècle, l'électricité permet de motoriser ces calculateurs. C'est le début de l'électromécanique. La microélectromécanique n'a cependant émergé que dans les années 1970. L'ordinateur personnel a connu son essor dix ans plus tard.

1.2 Avancées logiques et premiers « ordinateurs »

Parallèlement à ces avancées technologiques, des idées ont, elles aussi, contribué aux progrès scientifiques. Les **algorithmes** les plus anciens sont attestés par des tables datant

de l'époque d'Hammurabi (env. -1750 av. J.-C.) en Mésopotamie. De nombreux autres algorithmes furent décrits par la suite. C'est toujours le cas aujourd'hui : la compression des images en jpeg s'appuie sur des algorithmes datant de la fin des années 1990.

En logique (domaine des mathématiques lié à l'informatique), **George Boole** démontre que tout processus logique est décomposable en une suite d'opérations logiques (ET, OU, NON) appliquées sur deux états (VRAI-FAUX). Des questions plus profondes comme celle consistant à savoir si un mécanisme permet d'affirmer si une proposition est vraie ou fausse furent soulevées par **David Hilbert** dès 1928. Les logiciens **Kurt Gödel** et **Alan Turing** ont aussi participé à des avancées dans ce domaine de la logique.

La *machine de Turing* (1936) est un modèle abstrait permettant de mettre en œuvre n'importe quel algorithme. C'est en quelque sorte le modèle abstrait d'un ordinateur où la technologie permet de répondre à des questions algorithmiques.

De nombreux calculateurs programmables, dont les plus connus sont ceux appelés bombes (à cause du bruit engendré par leur fonctionnement) et dont le but était d'aider à déchiffrer les messages allemands durant la seconde guerre mondiale, sont construits entre 1936 et 1956. Alan Turing a participé à la conception de ces bombes.

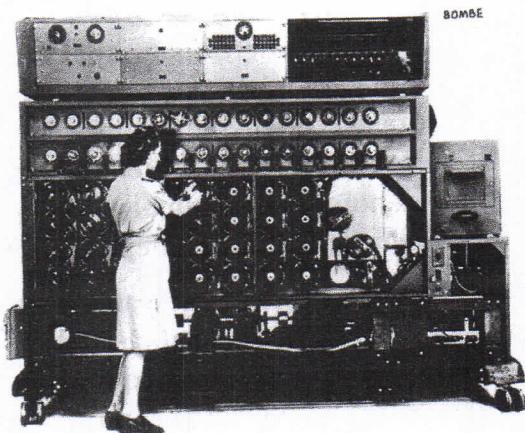


Fig. 1.1 – US Navy Bombe (1942), assez semblable aux bombes anglaises ; Alan Turing participa à leur conception. Archive de la NSA.

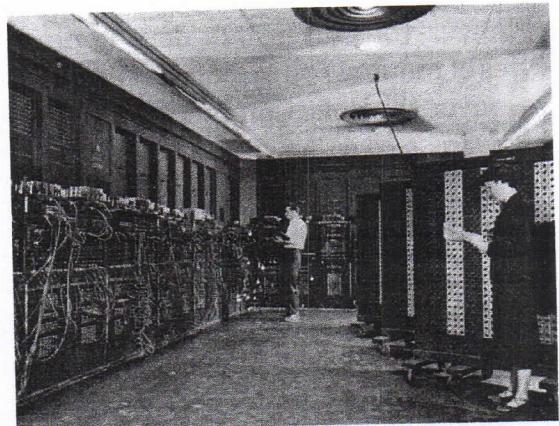


Fig. 1.2 – L'ENIAC (env. 1950), le premier ordinateur entièrement électronique, programmable mais incapable d'enregistrer les programmes.

En 1948, **Claude E. Shannon**, précurseur de la théorie de l'information, popularise l'utilisation du mot **bit** comme mesure élémentaire de l'information numérique, information qui devient mesurable et réductible à **deux signaux** élémentaires (notés 0 et 1).

Voici un exemple de **codage de l'information** issu du code ASCII (*American Standard Code for Information Interchange*, voir le tableau 9.1 page 78) :

lettre « A » \longleftrightarrow code 65 \longleftrightarrow bits 1000001 \longleftrightarrow signaux électriques

Fig. 1.3 – Codage d'une lettre en signaux électriques

À partir de 1948 apparaissent les premières machines à **architecture de Von Neumann**. Ce modèle d'architecture utilise une structure unique de stockage pour les données et les instructions ; un ordinateur peut alors modifier les instructions, effectuer des boucles, ce

1.2. Exercices

que les programmes venant de cartes perforées ne permettaient pas. Tous les ordinateurs actuels possèdent une architecture issue de celle décrite par **John Von Neumann**.

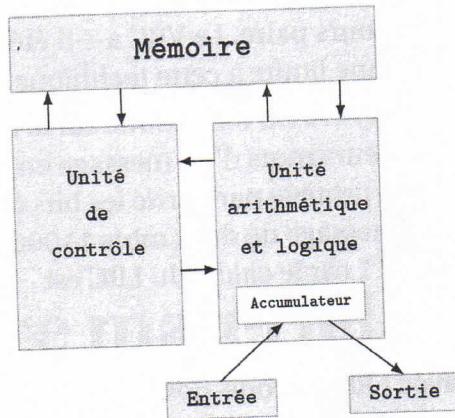


Fig. 1.4 – Architecture de Von Neumann

L'architecture de Von Neumann décompose l'ordinateur en quatre parties distinctes :

- l'unité arithmétique et logique qui effectue les opérations de base ;
- l'unité de contrôle, chargée du « séquençage » des opérations ;
- la mémoire qui contient à la fois les données et le programme qui indiquera à l'unité de contrôle quels sont les calculs à effectuer sur ces données. La mémoire se divise en mémoire volatile (programmes et données en cours de fonctionnement) et mémoire permanente (programmes et données de base de la machine) ;
- les dispositifs d'entrée-sortie (périphériques) pour communiquer avec l'extérieur.

EXERCICES

EXERCICE 1.1 ! Code 2 parmi 5 POSTNET et clés de contrôle

Le code 2 parmi 5 POSTNET est un code-barres symbolisant le code-postal américain à 5 chiffres et utilisé par le service postal des États-Unis pour l'aiguillage du courrier. Ce code représente chaque chiffre de 0 à 9 par des *mots* de 5 bits (ensuite convertis en barres de deux hauteurs), chaque mot contenant exactement 2 bits égaux à 1. Voici ce code :

Chiffre	0	1	2	3	4	5	6	7	8	9	<i>n</i>
Code	11000	00011	00101	00110	01001	01010	01100	10001	10010	10100	<i>c_n</i>

1. Sachant qu'un code-barres commence et finit toujours par une grande barre, lire la valeur du code-barres suivant :



2. Le *chekdigit*, ou *clé* de contrôle, est un entier rajouté à un code-barres afin de contrôler la validité de celui-ci ; une fois la clé rajoutée, la somme des chiffres codés doit être un multiple de 10. Le code-barres précédent possède-t-il une clé ? Si la réponse est non, quelle clé rajouter (à droite) à ce code-barres pour qu'il possède une clé de contrôle ?

3. Le code 374420 possède-t-il une clé de contrôle correcte ?
4. Le VRC (*Vertical Redundancy Check*) est une technique de contrôle de transfert du code : à chaque mot on ajoute un nouveau bit appelé *bit de parité* de façon à ce que la somme des bits du mot soit toujours paire. Le VRC a-t-il été appliqué au code 2 parmi 5 ? Trouver un défaut au VRC et une limite à cette technique.
5. Le LRC (*Longitudinal Redundancy Check*) fonctionne sur le même principe mais il protège l'ensemble des bits de plusieurs mots d'un message en ajoutant un mot de même longueur dans lequel chaque bit protège par parité les bits de même rang des mots du message. Par exemple pour un message de deux mots 11000 et 10011 alors le message transmis est 11000 10011 01011 car le calcul du LRC est

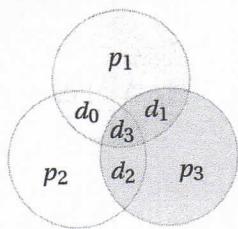
$$\begin{array}{r}
 11000 \\
 10011 \\
 \hline
 01011
 \end{array}$$

Quel est le code POSTNET avec LRC du code-postal 11052 (Port Washington, NY) ?

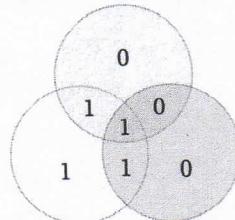
EXERCICE 1.2 !! Un code correcteur, le code de Hamming (7,4)

Le Code de Hamming (7,4) est un code qui permet de corriger toutes les erreurs de transmission de mots binaires de 7 bits. Sur ces 7 bits, 4 contiennent les données (ou information, les d_i) et 3 (les p_j) sont des bits de contrôle. \mathcal{H} est l'ensemble des mots de ce code. Par exemple 0010111 et 1101001 sont deux mots codés de \mathcal{H} .

Le mot $d_3d_2d_1d_0$ d'un message sera codé par $p_1p_2d_0p_3d_1d_2d_3$ avec $p_1 = d_0 \oplus d_1 \oplus d_3$, $p_2 = d_0 \oplus d_2 \oplus d_3$ et $p_3 = d_1 \oplus d_2 \oplus d_3$, le symbole \oplus désignant l'addition modulo 2. Cette addition vérifie $0 \oplus 0 = 1 \oplus 1 = 0$ et $0 \oplus 1 = 1 \oplus 0 = 1$ (voir chap. 3 pour plus de détails).



Représentation des quatre bits de données et des trois bits de parité. Cas général à gauche, un exemple à droite, celui du message 1101 codé par 0110011.



1. Décoder les mots 1000011 et 1111110 puis coder les mots 1010 et 0011.
2. Deux questions uniquement pour ceux qui connaissent déjà Python.
 - a. Écrire en Python une fonction `code_h` qui prend en argument un entier mot de quatre digits et renvoie son code de Hamming (7,4) de sept digits.
 - b. Écrire en Python une fonction `decode_h` qui prend en argument un mot codé `mcode` de sept digits (de type chaîne) et renvoie le mot de quatre digits décodé.
3. Combien l'ensemble \mathcal{H} contient-il d'éléments ?
4. La **distance de Hamming** $d_h(i, j)$ entre deux mots i et j de codes respectifs c_i et c_j est le nombre de bits (notée $d'_h(c_i, c_j)$) de même rang (ou position) différents entre ces deux codes. Ainsi $d_h(0000, 0011) = d'_h(0000000, 0111100) = 4$. Calculer toutes les distances entre 1010, 1110, 1100 et 0001.
5. Plus généralement, la distance de Hamming $d_{\mathcal{H}}$ est le minimum de l'ensemble des entiers $\{d'_h(c_i, c_j), c_i \in \mathcal{H}, c_j \in \mathcal{H}\}$. Déterminer \mathcal{H} puis prouver que $d_{\mathcal{H}} = 3$.
6. Le mot 1100100 a été reçu. Est-il correct ? Sinon, en admettant qu'il n'y a qu'une erreur, quel mot avait été transmis ?

Chapitre 2

Architecture matérielle et logicielle

L'objectif de ce chapitre est de découvrir l'architecture matérielle et logique des ordinateurs individuels, les PC (*Personal Computer*). Il existe bien d'autres formes d'ordinateurs dont ceux de l'informatique embarquée par exemple.

On ne parlera pas dans ce chapitre de la souris, du clavier ou des écrans qui sont des périphériques et sans lesquels un ordinateur peut fonctionner malgré tout (même si l'intérêt de ce mode de fonctionnement vous en paraît réduit).

2.1 Le boîtier

Le boîtier (souvent une tour) ne fait que protéger un ordinateur. Cependant ses dimensions correspondent à des normes, en particulier celles des ouvertures, qui permettent déjà d'observer un grand nombre de pièces.

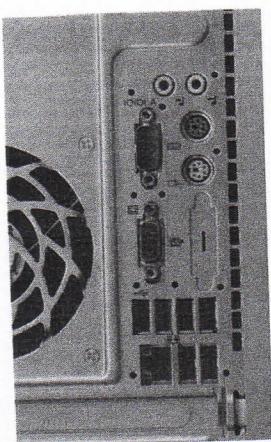


Fig. 2.1 – Arrière d'un boîtier et ses nombreuses entrées/sorties.

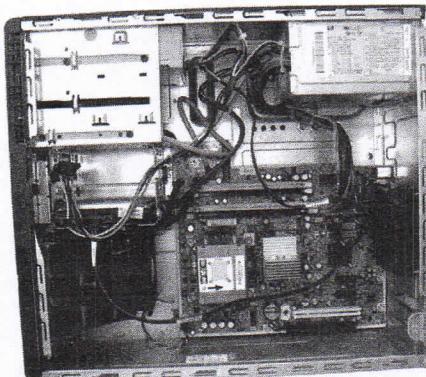


Fig. 2.2 – À l'intérieur du boîtier (bien visibles, les fils d'alimentation et de données).

Sur la face avant sont généralement présents :

- le bouton de démarrage/arrêt ;
- un lecteur (ou graveur) CD/DVD (périphérique d'entrée/sortie) ;
- un ou des ports USB (périphérique d'entrée/sortie) ;
- Éventuellement d'autres lecteurs (cartes SSD, ...).

On trouve sur la face arrière (détails de ports en 2.2) :

- l'alimentation et son radiateur associé ;
- une sortie VGA (sortie vidéo, de plus en plus rare et remplacée par la sortie suivante) ;
- une sortie HDMI (souvent sortie de la carte vidéo) ou DisplayPort (ou les deux) ;
- des ports PS/2 (clavier, souris) ;
- un port Série COM1 (souvent associé à la norme RS232, de plus en plus rare) ;
- des ports USB ;
- un port RJ45 (pour internet en particulier, présent sur la carte mère ou une carte réseau).

2.2 Quelques ports fréquents

2.2.1 Port Com1

Historiquement, le port série **Com1** est le premier port de communication utilisant une transmission série (norme RS232). Son débit est au maximum de 19 200 bps (bits/s). Il a été très longtemps utilisé pour sa simplicité de configuration, de pilotage et sa robustesse. Il tend à disparaître des PC. On l'utilisait pour relier une souris ou un modem.



Fig. 2.3 – Prise mâle Port COM1.

2.2.2 Port PS/2

PS/2 : *Personal System/2*, appelé aussi port mini-Din. Port de communication de taille réduite ayant succédé au PS/1 (Din, le même en plus encombrant) permettant la connexion du clavier ou de la souris mais démocratisé en 1995 suite à son intégration sur les cartes mères de type ATX. Il a également été supplanté par le standard USB depuis quelques années et actuellement de plus en plus par le Bluetooth.

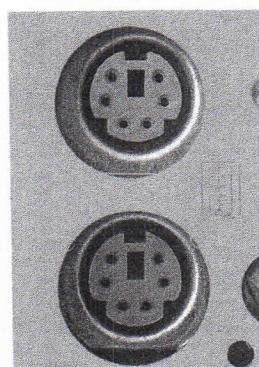


Fig. 2.4 – Deux ports femelles PS/2.

2.2.3 Port VGA

VGA : *Video Graphics Array*. Ce port est de type analogique. Il était souvent utilisé pour relier un PC à une sortie vidéo (écran, vidéo projecteur, ...). Il tend à disparaître au profit de l'HDMI.



Fig. 2.5 – Prise VGA femelle.

2.2.4 Port DVI

DVI : *Digital Visual Interface*. Ce port est de type numérique non HD. Il apporte une amélioration en terme de réduction du bruit par rapport au connecteur VGA analogique. Il est intéressant pour les dispositifs d'affichage tels que les écrans LCD et plasma.

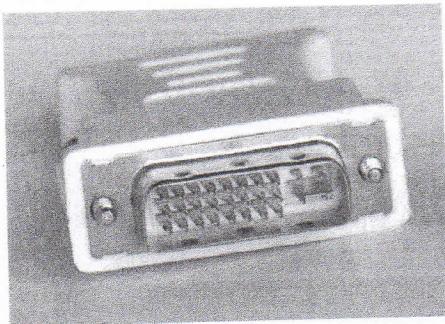


Fig. 2.6 – Prise DVI (convertisseur DVI/VGA)

2.2.5 Port HDMI

Le port **HDMI** (*High Definition Multimedia Interface*) est de type numérique et permet la transmission de signaux au format HD. Il existe différents niveaux de norme HDMI. Il remplace peu à peu le port DVI pour relier des écrans, vidéoprojecteurs (sorties) ou encore des lecteurs Blu-ray (entrées).

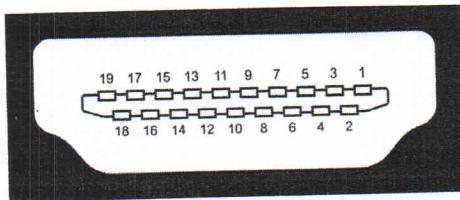


Fig. 2.7 – Prise HDMI de type A et ses 19 broches.

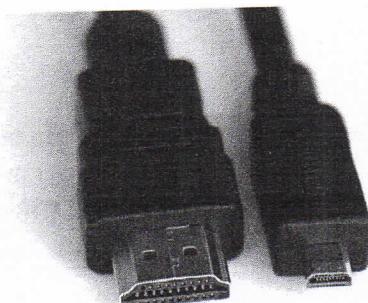


Fig. 2.8 – Différents types de prises HDMI.

2.2.6 Port USB

USB : *Universal Serial Bus*. Cette norme de communication série est apparue en 1996. Le protocole série associé a révolutionné la liaison PC-périphériques, uniformisant beau-

coup de modes de communication. C'est le **bus de communication** le plus fréquent. Ses nouveaux concurrents sont les protocoles sans fil Bluetooth et WiFi.

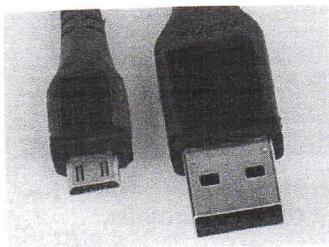


Fig. 2.9 – Différents types de prises USB.

Version	Année	Débit
USB1.0	1996	0,19 Mo/s
USB1.1	1998	1,5 Mo/s
USB2.0	2000	60 Mo/s
USB3.0	2008	600 Mo/s
USB3.1	2014	1,2 Go/s

Fig. 2.10 – Évolution des débits des normes USB.



Fig. 2.11 – Symbole de la norme USB.

2.2.7 Port Réseau RJ45

RJ45 (RJ pour *Registered Jack*). Ce port permet la connexion filaire réseau ethernet (dénommé LAN pour *Local Area Network*) par câble. Différents protocoles existent, spécifiant différents débits.

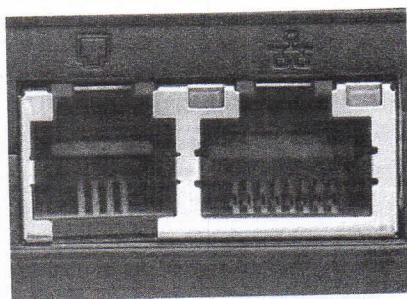


Fig. 2.12 – Prises Réseau RJ45 à droite (RJ11 à gauche, plus lente).

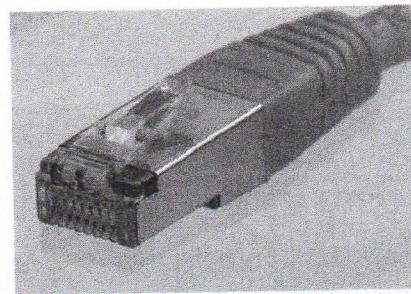


Fig. 2.13 – Câble Réseau RJ45.

2.3 La carte mère

La **carte mère** (*mother board*) est l'élément central de l'architecture d'un PC. À la carte mère sont reliés :

- le microprocesseur;
- la mémoire vive appelée RAM (*Random Access Memory*);
- le disque dur (HDD, *Hard Disk Drive*);
- l'horloge interne;
- le BIOS;
- la mémoire CMOS;
- un chipset;
- des bus de communication (bus de données, bus d'adresses, ...);

2.3. La carte mère

- un ensemble de contrôleurs d'entrée/sortie aux rôles divers (communications avec le disque dur, les ports d'entrée/sortie du type USB, capteurs de température, ...);
- d'éventuelles autres cartes (carte vidéo, carte son, ...).

Au cœur d'un PC, la carte mère doit assurer la communication entre les divers constituants de la machine. Voici l'architecture standard :

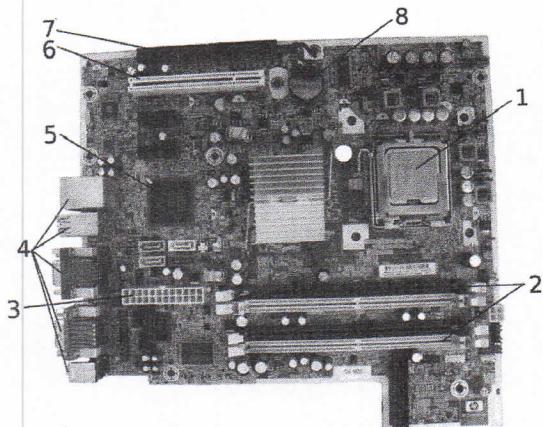


Fig. 2.14 – Carte mère.

- 1 - CPU sur son socket;
- 2 - Slots pour la RAM ;
- 3 - Connecteurs d'alimentation de la carte ;
- 4 - Ports d'entrées/sorties (son, USB, série, ...);
- 5 - Un des processeur/microcontrôleur de la carte ;
- 6 - Port PCI ;
- 7 - Port PCI Express ;
- 8 - Pile du BIOS ;

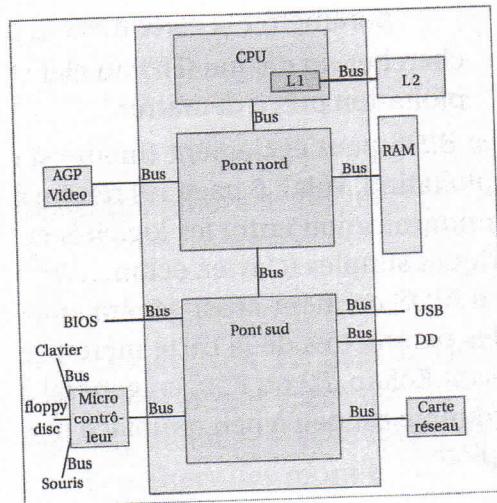


Fig. 2.15 – Schéma du rôle d'une carte mère.

Pont nord (*North Bridge*) et pont Sud (*South Bridge*) : éléments du chipset (voir ci-dessous), ce sont des contrôleurs qui gèrent les bus de communications à des vitesses de transmission différentes.

2.3.1 Le chipset

Présent sur la carte mère, le **chipset** est l'élément le plus important de celle-ci. Constitué de deux entités, le **North Bridge** et le **South Bridge**, il gère les échanges (via les bus de données et d'adresses) avec les mémoires « rapides » pour le premier et avec les mémoires plus « lentes » pour le second. L'horloge interne de l'ordinateur permet de gérer les vitesses de transmission de ces données.

2.3.2 BIOS et CMOS

BIOS

Le **BIOS** (*Basic Input/Output System*) est un petit programme. Il est situé sur la carte mère de l'ordinateur dans une puce de type ROM. Le BIOS est le premier programme chargé en mémoire dès que l'ordinateur est mis sous tension. Il assure plusieurs fonctions :

- le POST : un ensemble de tests qu'effectue le BIOS avant de démarrer le système d'exploitation :
 - vérifier que la carte mère fonctionne correctement (barrettes RAM, contrôleurs de ports série, parallèle, IDE, ...),
 - vérifier que certains périphériques connectés à la carte mère fonctionnent bien (clavier, carte graphique, disques dur, lecteur de CD-Rom, ...),
 - paramétriser la carte mère (à partir des informations stockées dans les CMOS) ;
 - chercher un disque (HD ou clef USB ou CDROM) sur lequel il y a un système d'exploitation prêt à démarrer.

Le BIOS peut également (même si ce n'est pas le cas pour la majorité des systèmes d'exploitation, voir 2.6 page 19) rendre des services au système d'exploitation en assurant la communication entre les logiciels et les périphériques, mais seulement pour les périphériques simples (clavier, écran, ...).

Le BIOS contient aussi généralement un programme, le **setup** qui permet de modifier les paramètres de la carte mère (c'est le programme auquel vous pouvez accéder en pressant Echap, F2 ou F10, en général à la mise sous tension de l'ordinateur). Ce dernier programme est peu à peu remplacé par l'**UEFI** qui présente une interface graphique plus complète.

CMOS

Le **CMOS** (*Complementary Metal Oxide Semiconductor*) est un type de puce capable de stocker des informations et de les conserver même quand l'ordinateur est hors tension. Leur contenu est maintenu par un faible courant électrique fourni par une pile. Ces mémoires peuvent être modifiées souvent sans dommage. Le BIOS vient lire des informations dans cette mémoire au démarrage de l'ordinateur. Il y stocke également la date et l'heure et vient régulièrement les mettre à jour. Les mémoires CMOS sont plus lentes que celles utilisées pour le fonctionnement courant de l'ordinateur. Les CMOS ont l'avantage de consommer peu de courant par rapport aux autres types de mémoires.

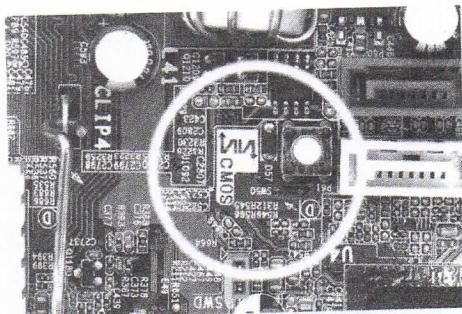


Fig. 2.16 – Bouton de réinitialisation du BIOS.

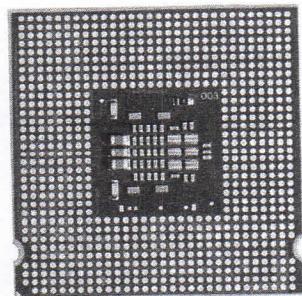


Fig. 2.17 – Microprocesseur (CPU).

2.3.3 Le microprocesseur

Le **microprocesseur** (*CPU* en anglais : *Central Processing Unit*) est monté sur son **socket** (réceptacle) et est équipé d'un radiateur-ventilateur pour assurer son refroidissement.

2.3. La carte mère

ment, le radiateur étant relié au CPU par une pâte thermique. Le processeur est la principale **unité de traitement** de l'ordinateur : il exécute les programmes stockés en mémoire, il charge, décode puis exécute les instructions. Le microprocesseur est structuré en plusieurs parties.

Architecture logique d'un microprocesseur

Un microprocesseur est essentiellement constitué :

- d'une UAL (unité arithmétique et logique) ;
- d'une unité de commandes (ordonne et décide des opérations à effectuer) ;
- de registres (mémoires particulièrement rapides) ;
- d'entrées/sorties.

Cette architecture est dite **architecture de Von Neumann**, voir 1.2 page 8. Les calculs sont effectués par l'UAL (elle-même divisée en plusieurs unités spécialisées). L'UAL et l'unité de contrôle effectuent de nombreux échanges de données avec la RAM, via des bus de données et d'adresses. Il existe d'autres types d'architectures (dont celle de Harvard, utilisée dans des microprocesseurs spécifiques).

Un processeur exécute des instructions, l'ordre des instructions est géré par le **compteur de programme** (PC, *Program Counter*, à ne pas confondre avec *Personal Computer*). Chaque instruction est chargée, décodée, des calculs sont effectués pour connaître les adresses des opérandes (adresses mémoires dans les registres ou la RAM), les opérandes sont chargés, l'instruction est exécutée, les résultats envoyés vers la bonne adresse mémoire. Ensuite, le compteur de programme exécute l'instruction suivante.

Précisons que l'UAL fait partie de l'unité de traitement, qui contient aussi généralement une unité de virgule flottante (FPU, pour *Floating Point Unit*) qui est spécialisée pour les calculs sur les flottants, un registre d'état et d'autres registres, tous spécialisés.

Enfin, d'un point de vue physique, l'UAL est constituée de circuits électroniques logiques eux-mêmes construits à l'aide de **transistors**. Le nombre de transistors d'un microprocesseur d'un PC familial était d'environ un million en 1990, il est de près de dix milliards en 2016.



Fig. 2.18 – Premier transistor, 1947 (Bell Labs).

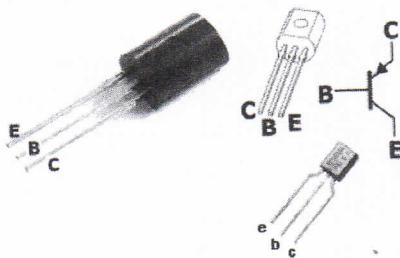


Fig. 2.19 – Transistors et schémas de transistor.

2.4 Les mémoires

2.4.1 La mémoire vive

La **mémoire vive** (ou **RAM**, *Random Access Memory*, mémoire à accès aléatoire) de l'ordinateur se présente sous forme de barrettes. Elle est qualifiée de mémoire vive par opposition à la mémoire morte (ou **ROM**, *Read Only Memory*) car c'est une mémoire qui :

- est effaçable ;
- disparaît en l'absence d'alimentation ;
- est d'accès particulièrement rapide.

En plus des barrettes, on la trouve aussi dans certaines cartes (vidéo, son) et enfin et surtout sous plusieurs formes au sein du microprocesseur (mémoires caches L1, L2 et L3, registres du microprocesseur).

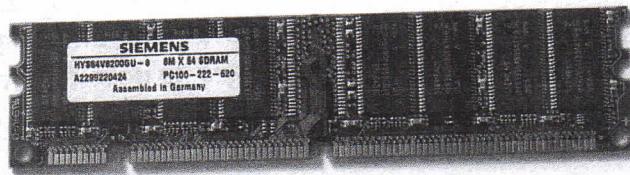


Fig. 2.20 – Barrette de RAM

2.4.2 Les mémoires mortes

Les mémoires mortes ou **ROM** (*Read-Only Memory*) sont des mémoires qui perdurent malgré l'absence d'alimentation. Elles sont d'accès plus lent que la mémoire RAM. Les types de mémoires mortes ont évolué, les vraies ROM sont rares. Il existe maintenant par exemple des **EPROM** (*Erasable Programmable Read-Only Memory*) à effacement de mémoire par rayonnement ultraviolet et surtout des **EEPROM** (*Electrically Erasable Programmable Read Only Memory*), effaçables électriquement et programmables (cartes CP, MS ou SD utilisées entre autre pour les cartes mémoires des appareils photographiques numériques et disques SSD, *Solid-state-drives* qui remplacent peu à peu les disques durs). Les « vraies » ROM sont présentes sous différentes formes :

- des puces (les CMOS, qui contiennent le BIOS, voir 2.3.2) ;
- des disques (CDROM, DVD, BLU-RAY, ...).

2.4.3 Les mémoires de masse

Les mémoires de masse sont :

- les disques durs DD (disque dur) ou HD *Hard Disk* ou HDD *Hard Disk Drive* à mémoire magnétique ou encore les disques durs SSD (*Solid State Drive*) à mémoire flash (plus récents) ;

- les cartes flash des APN (appareils photographiques numériques) ;
- les clefs USB ;
- les cartes perforées (de vos ancêtres).

2.5 Les BUS

En informatique, le terme **bus** désigne le matériel (souvent des fils) qui transporte l'information et permet ainsi la communication entre différents composants. Il existe deux grands types de bus :

- le **bus série** : il comporte plusieurs fils dont : le fil de masse (référence de potentiel), le fil de données, le fil d'horloge ; les données sont transmises en série (bit par bit). Les souris utilisent un bus série ;
- le **bus parallèle** : il comporte un fil de masse, un fil d'horloge, et n fils de données pour un bus n bits ; les données sont transmises en parallèle. Le bus entre le microprocesseur et la RAM est un bus parallèle.

Le débit d'un bus série peut être supérieur à celui d'un bus parallèle, c'est son intérêt principal.

2.6 Les systèmes d'exploitation

Un système d'exploitation (ou OS pour *Operating System*) est un programme qui prend la main juste après le BIOS

au démarrage d'un ordinateur. Le système d'exploitation a de très nombreux rôles, certains visibles et connus, d'autres moins. Il gère en particulier :

- l'interface graphique ;
- l'ordre de priorité des programmes ;
- l'utilisation du microprocesseur.

Le rôle essentiel d'un système d'exploitation est d'être une interface entre d'une part le matériel dont les ports, la mémoire, et le microprocesseur ont leur propre langage et d'autre part les logiciels (dits **logiciels applicatifs**) de l'utilisateur qui ne tiennent pas compte, grâce aux OS, des particularités du matériel. Les logiciels sont ainsi écrits pour fonctionner sur un système d'exploitation qui, lui, rend possible ce fonctionnement avec le matériel de l'ordinateur en gérant les communications avec les périphériques, le microprocesseur et les zones de stockage de la mémoire. Il serait, pour donner un exemple simple, extrêmement compliqué et désagréable si chaque logiciel devait avoir une version adaptée à chacun des ordinateurs pour lesquels il est destiné.

Le système d'exploitation offre, outre cette plus grande simplicité d'exploitation des capacités de l'ordinateur, ses propres logiciels (manipulations de fichiers, gestion des utilisateurs, gestion des interfaces graphiques utilisateurs, ...). Enfin, il est important de savoir que les OS non seulement détectent et gèrent les erreurs mais aussi effectuent un contrôle permanent de l'utilisation des ressources.

Bien que des dizaines de systèmes d'exploitation différents existent, deux familles d'OS sont essentiellement présentes : ceux de la famille UNIX dont GNU/Linux (qui est libre et

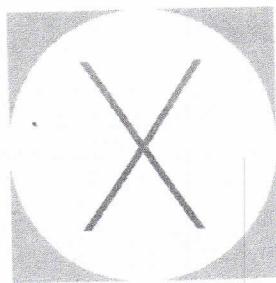


Fig. 2.21 – Logo de Mac OS X.

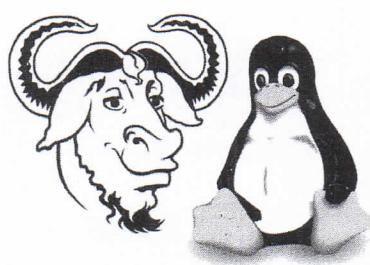


Fig. 2.22 – Logo de GNU/Linux.

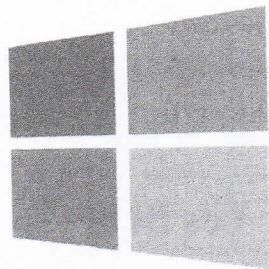


Fig. 2.23 – Logo de Microsoft Windows.

gratuit) ou OS X (qui est propriétaire et payant) et ceux de la famille Windows (propriétaire et payant).

La politique commerciale très offensive de Microsoft fait qu'en 2014, 90 % des PC sont équipés de systèmes Windows, Mac Os ayant environ 7 % des parts du marché et environ 1,5 % des PC fonctionnent avec GNU/Linux. Par contre, sur les smartphones (et dans une moindre mesure sur les tablettes), le système libre Android utilisant le noyau Linux est très largement majoritaire (plus de 80 % du marché). La répartition des parts des OS sur les tablettes mobiles est proche de celui de smartphones : Android est très largement majoritaire et la part d'iOS d'Apple est à la baisse (moins de 20 %).

Enfin, Linux équipe la majorité des serveurs dans le monde. Actuellement 90 % des serveurs sont installés avec Linux, pourcentage en progression.

EXERCICES

EXERCICE 2.1 ● Architecture matérielle

1. Quelle est l'utilité de la pile de la carte mère ?
2. Qu'est-ce qu'un double cœur ?
3. ● La RAM peut se schématiser sous forme de cases empilées, chaque case ayant une adresse et contenant un mot d'un octet. Un composant (Chipset, CPU, ...) ayant un bus d'adresse de 32 bits, quelle est sa capacité maximale adressable (en octets) ?

EXERCICE 2.2 ● Systèmes d'exploitation et architectures

1. Comment ouvrir une console ou un terminal ? Quelle commande permet d'obtenir la liste des répertoires ?
2. Comment se déplacer dans les répertoires ?
3. Comment observer sous Linux, Mac OS et Windows, l'utilisation du CPU, de la RAM, ainsi que la liste des processus ? Comment arrêter un processus ?

Chapitre 3

Représentation des nombres

Un ordinateur a une manière propre de manipuler les nombres qui est très différente de celle de la vie courante. On va expliquer dans ce chapitre :

- ce qu'est le système de numération binaire ;
- comment sont représentés les nombres entiers, naturels puis relatifs ;
- comment sont représentés les nombres non-entiers ;
- quelques conséquences de ces représentations.

3.1 Le binaire

3.1.1 Introduction

Le système **binaire** est aussi appelé **base 2**, c'est un système d'écriture des nombres comportant exactement deux symboles.

On utilise ce système en informatique car il n'y a besoin que de deux valeurs qui correspondent, en électronique, à deux niveaux de tension différents. D'autres systèmes auraient pu exister (système ternaire, décimal) mais les complications électroniques rencontrées sont trop nombreuses pour qu'ils soient mis en place à l'heure actuelle.

Le système binaire, contrairement au système décimal qui utilise dix chiffres (0, 1, ..., 8 et 9), n'utilise que **deux** chiffres (ou symboles ou encore **digits**) pour représenter les nombres : **0** et **1**.

C'est un **système de position** comme le système décimal, c'est-à-dire que la position des chiffres donne leur valeur (multiple de la base) ; par exemple en base 10, le 3 de 231 a pour valeur 30 alors que celui de 321 vaut 300 ; de même en base 2 les nombres 10 et 01 ne représentent pas les mêmes nombres. *A contrario*, dans les écritures XI et IX, le symbole X a la même valeur, dix.

Voici des exemples d'entiers écrits en base 2, les mêmes nombres écrits en base 10, puis en base 16 où les chiffres (ou caractères ou **digits**) utilisés sont 0, 1, 2, ..., 9, A, B, C, D, E, F.

Le même nombre écrit ...		
... en base 10	... en base 2	... en base 16
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
:	:	:
9	1001	9
10	1010	A
11	1011	B
12	1100	C
:	:	:
165	10100101	A5

3.1.2 Notation

Par convention, on notera en indice et entre parenthèses la base dans laquelle est écrite un nombre. Par exemple

$$11_{(10)}$$

est le nombre onze écrit en base 10 et

$$11_{(2)}$$

est le nombre « un-un » en base 2 (et égal à 3 en base 10, voir le tableau ci-dessus). On a donc, à titre d'exemple (voir tableau ci-dessus)

$$7_{(10)} + 3_{(10)} = 111_{(2)} + 11_{(2)} = 10_{(10)} = 1010_{(2)} = A_{(16)}.$$

3.1.3 Systèmes de numération pondérée en base 10, 2, 8, 16 et 64

Écriture d'un nombre en base 10

Tout nombre N en base 10 s'écrit sous la forme

$$[c_n \dots c_2 c_1 c_0, c_{-1} c_{-2} \dots c_{-m}]_{(10)}$$

où c_n, \dots, c_{-m} sont les *chiffres* (ne pas confondre nombre et chiffre) constituant ce nombre N . Par exemple si

$$N = 9876,25$$

alors $c_3 = 9, c_2 = 8, c_1 = 7, c_0 = 6, c_{-1} = 2$ et $c_{-2} = 5$.

Au rang i correspond le chiffre (ou digit) c_i de poids 10^i en base 10. Ainsi la **valeur décimale** de N est donnée par :

$$N = c_n \times 10^n + \dots + c_2 \times 10^2 + c_1 \times 10^1 + c_0 \times 10^0 + c_{-1} \times 10^{-1} + \dots + c_{-m} \times 10^{-m}$$

et ici

$$9876,25_{(10)} = 9 \times 10^3 + 8 \times 10^2 + 7 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}.$$

Écriture d'un nombre en base 2

Il en est exactement de même en base 2. Si un nombre N s'écrit

$$b_n \cdots b_1 b_0, b_{-1} \cdots b_{-m}$$

où pour tout i , $b_i \in \{0; 1\}$, alors la valeur décimale de N est donnée par

$$N_{(10)} = b_n \times 2^n + \cdots + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + \cdots + b_{-m} \times 2^{-m}.$$

Par exemple si

$$N = 10101_01_{(2)}$$

alors sa valeur décimale se calcule ainsi :

$$\begin{aligned}
 N &= [1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}]_{(10)} \\
 &= [16 + 4 + 1 + 0 + 0,25]_{(10)} \\
 &= 21,25_{(10)}
 \end{aligned}$$

c'est-à-dire qu'on a les égalités $N = 10101,01_{(2)} = 21,25_{(10)}$.

Du décimal au binaire

On peut démontrer que pour passer de l'écriture décimale à l'écriture binaire d'un entier N on peut effectuer les **divisions euclidiennes successives** de N puis des quotients successifs **par 2**, jusqu'à ce que le quotient devienne 0. L'écriture binaire de N est alors la suite des restes dans l'**ordre inverse** de leur obtention.

Exemples de 1005 et 23 :

$$\begin{array}{r}
 1005 \quad | \quad 2 \\
 1 \quad | \quad 502 \quad | \quad 2 \\
 \quad \quad 0 \quad | \quad 251 \quad | \quad 2 \\
 \quad \quad \quad 1 \quad | \quad 125 \quad | \quad 2 \\
 \quad \quad \quad \quad 1 \quad | \quad 62 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad 0 \quad | \quad 31 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad 1 \quad | \quad 15 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad \quad 1 \quad | \quad 7 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad \quad \quad 1 \quad | \quad 3 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad \quad \quad \quad 1 \quad | \quad 1 \quad | \quad 2 \\
 \quad 1 \quad | \quad 0
 \end{array}$$

← restes des divisions dans cet ordre

$$\text{d'où } 1005_{(10)} = 1111101101_{(2)}$$

Écriture d'un nombre en base 8

De même il existe la base 8, ou **octale**, dont les symboles sont 0, 1, 2, 3, 4, 5, 6 et 7, dans laquelle un nombre O s'écrit sous la forme

$$o_n \cdots o_2 o_1 o_0, o_{-1} \cdots o_{-m}.$$

Écriture d'un nombre en base 16

De même en base 16, ou base **hexadécimale**, dont les symboles sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F, un nombre H écrit sous la forme

$$h_n \cdots h_2 h_1 h_0, h_{-1} \cdots h_{-m}$$

où $h_i \in \{0, \dots, 9, A, \dots, F\}$ et $A_{(16)} = 10_{(10)}$, $B_{(16)} = 11_{(10)}, \dots$

La valeur décimale de H est :

$$H = h_n \times 16^n + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0 + h_{-1} \times 16^{-1} + \cdots + h_{-m} \times 16^{-m}$$

donc, par exemple, si $H = A3B_{(16)}$ alors

$$H = [10 \times 16^2 + 3 \times 16^1 + 11 \times 16^0]_{(10)} = 2619_{(10)}.$$

Par exemple vous trouverez facilement sur votre box internet une clé de sécurité écrite en hexadécimal.

Du binaire à l'hexadécimal et inversement

Tout vient de l'égalité $2^4 = 16$. Elle implique que tous les nombres de quatre chiffres en base 2 s'écrivent à l'aide d'un seul chiffre en base 16 et réciproquement. On passe ainsi très facilement d'une base à l'autre.

Voici deux exemples :

$$1010\ 1101\ 0111_{(2)} = [\underbrace{1010}_{A_{(16)}}\ \underbrace{1101}_{D_{(16)}}\ \underbrace{0111}_{7_{(16)}}]_{(2)} = AD7_{(16)}$$

$$3A4_{(16)} = 0011\ 1010\ 0100_{(2)} \quad \text{car} \quad 3_{(16)} = 0011_{(2)}, A_{(16)} = 1010_{(2)} \text{ et } 4_{(16)} = 0100_{(2)}$$

Écriture d'un nombre en base 64

De même est utilisée la base 64, appelée **base64** (surprenant, non ?), dont les symboles sont $A, B, \dots, Z, a, b, \dots, z, 0, 1, \dots, 9, +,$ et $/$. Ce codage est particulièrement utilisé sur l'Internet pour la transmission des pièces jointes aux messages (mails).

3.2 Représentation des nombres dans un ordinateur

Comme toute information dans un ordinateur, les nombres (dont les entiers) sont **codés à l'aide de bits** (*binary digit*). Ces bits sont, sauf exception, regroupés par huit et un groupe de huit bits est appelé un **octet** (*byte* en anglais, de symbole **B**, à ne pas confondre avec **b** pour bits car $1B = 8b$).

Les ordinateurs utilisent différentes méthodes pour représenter (ou coder) les nombres en fonction de leur type : les « entiers courts » sont généralement codés sur deux octets ; les « entiers longs » sont codés sur 4 ou 8 octets (32 ou 64 bits) dont les **entiers naturels** et

les **entiers relatifs** (codés différemment) ; des nombres à virgule (nombres **flottants**). Tous ces types de nombres ont des représentations différentes.

Ces différentes représentations sont présentes côté à côté : cela permet d'optimiser la gestion de la mémoire. Chaque type possède ses avantages et inconvénients. Détaillons trois représentations : celle des entiers naturels, des entiers relatifs et des flottants.

3.3 Représentation des entiers naturels

Les nombres entiers naturels ($0, 1, 2, \dots$) sont représentés en binaire dans un ordinateur et appelés entiers **non signés** (*unsigned*). Ils sont codés sur 8, 16, 32 ou 64 bits, chaque bit valant 0 ou 1. Ces bits sont regroupés par huit (**octets**).

Sur un octet on peut représenter 2^8 soit 256 nombres de 0 à $1111\ 1111_{(2)}$. Remarquons que $1111\ 1111_{(2)} = 255_{(10)} = [2^8 - 1]_{(10)}$. Sur quatre octets, soit 32 bits, on peut représenter $2^{32} = 4294967296$ entiers, de 0 à $111111111111111111111111_{(2)} = [2^{32} - 1]_{(10)}$.

Par exemple, le nombre $240_{(10)} = 1111\ 0000_{(2)}$ est représenté par

1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---

sur *un* octet et sur *deux* octets par

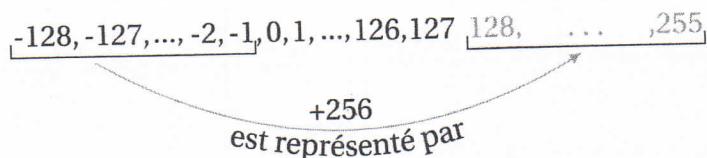
0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

3.4 Représentation des entiers relatifs

Pour représenter les entiers relatifs, il faut étendre la représentation des entiers positifs aux négatifs. Il existe pour cela plusieurs méthodes. Celle qui est utilisée au sein des ordinateurs est nommée **complément à deux**. Expliquons-la sur un octet pour simplifier (intelligent, le lecteur, adaptera la représentation sur plusieurs octets).

Sur un octet (huit bits), il y a 2^8 , soit 256 entiers relatifs distincts qui peuvent être représentés. La moitié est utilisée pour représenter les entiers positifs, l'autre moitié pour les négatifs. Il y aura donc 128 entiers positifs représentables, les nombres de 0 à 127 et 128 entiers négatifs représentables, les nombres de -128 à -1. On représente :

- les 128 entiers positifs (de 0 à $127 = 2^7 - 1$) par *le même* entier (en binaire) ;
- les 128 entiers strictement négatifs x (donc avec $-128 = -2^7 \leq x \leq -1$) par l'entier positif $2^8 + x = 256 + x$, écrit en binaire (255 en binaire représente -1, ...).



Remarquons que, comme on a alors l'encadrement $128 \leq 256 + x \leq 255$, les entiers de 128 à 255 représenteront les 128 entiers négatifs x de -128 à -1.

Le tableau page suivante donne la représentation d'entiers relatifs de -128 à +127 sur huit bits avec la méthode du complément à deux. Par exemple $1111\ 1110$ représente -2 et $1000\ 0000$ représente -128.

Entier relatif x	Entier non signé	Représentation sur 8 bits de x
-128	128	1000 0000
-127	129	1000 0001
-126	130	1000 0010
⋮	⋮	⋮
-3	253	1111 1101
-2	254	1111 1110
-1	255	1111 1111
0	0	0000 0000
1	1	0000 0001
⋮	⋮	⋮
126	126	0111 1110
127	127	0111 1111

TABLE 3.1 – Représentation sur 8 bits de 256 entiers relatifs par la méthode du complément à deux

L'encadré suivant explique comment obtenir autrement et de façon simple cette représentation pour les entiers négatifs, puisque pour les positifs, il n'y a qu'à les écrire en binaire. Sur 8 bits, 12 est représenté par 0000 1100, voyons comment représenter -12 sur 8 bits.

Complément à deux

Méthode : pour obtenir le complément à deux d'un entier négatif en binaire sur n bits, il suffit de :

- coder sa valeur absolue en binaire sur n bits ;
- inverser tous les bits ;
- ajouter 1 (addition binaire) à la ligne précédente.

Exemple : calculons le complément à deux de -12 sur 8 bits :

on code $ -12 $ soit 12 en binaire :	0 0 0 0 1 1 0 0
on inverse les bits :	1 1 1 1 0 0 ¹ 1 ¹ 1
ajout de 1 :	+ 0 0 0 0 0 0 1
Résultat (complément à deux) :	= 1 1 1 1 0 1 0 0

Conclusion : le complément à deux de -12 en binaire sur 8 bits est 1111 0100.

Remarques :

1. Reprenons l'addition binaire ci-dessus colonne par colonne.

Dernière colonne : $1_{(2)} + 1_{(2)} = 10_{(2)}$ donc on pose 0 et on retient 1.

Avant-dernière colonne $1_{(2)}(\text{retenue}) + 1_{(2)} + 0_{(2)} = 10_{(2)}$ donc on pose 0 et on retient 1.

Colonne précédente : $1_{(2)}(\text{retenue}) + 0_{(2)} + 0_{(2)} = 1_{(2)}$, donc on pose 1. Ainsi de suite.

2. Les nombres positifs sont finalement codés sous la forme $0b_1b_2b_3b_4b_5b_6b_7$ avec $b_i \in \{0, 1\}$, les nombres négatifs de la forme $1b'_1b'_2b'_3b'_4b'_5b'_6b'_7$ avec $b'_i \in \{0, 1\}$, donc le premier bit donne le signe de l'entier.

3.5 Représentation des nombres à virgule

3.5.1 Représentation d'un nombre à virgule flottante suivant la norme IEEE754

Les nombres autres que les entiers naturels et les entiers relatifs ont une représentation appelée **représentation en virgule flottante**.

Cette représentation est proche de la notation scientifique en base 10 : rappelons sur un exemple que l'écriture scientifique en base 10 de 1234,56 est

+1,23456 × 10³

écriture qui présente un signe, un nombre décimal x avec $1 \leq x < 10$ et un exposant (la puissance de 10). « 1,234 56 » est la **mantisso** et « 23 456 » la **pseudo-mantisso**.

Les nombres à virgule flottante de nos ordinateurs sont représentés selon le standard EEE 754 [9]. Ce standard décrit deux types de représentations en virgules flottantes (très proches), le type **simple précision** sur 32 bits et le type **double précision** sur 64 bits. Tous deux s'écrivent sous la forme

$$\pm m \times 2^e,$$

ù apparaît un **signe** \pm , une **mantisse** m avec $1 \leq m < 2$ et un **exposant** e . Le nombre le bits de la mantisse et celui de l'exposant sont normalisés (norme IEEE754). Décrivons cette norme en simple précision :

- 1 bit est réservé pour le signe ;
 - 8 bits sont réservés pour l'exposant (11 en double-précision) ;
 - 23 bits sont réservés pour la pseudo-mantisse (52 en double-précision).

En Python il est possible de savoir si l'on travaille en simple ou double précision. En effet, le module `sys` permet d'accéder aux propriétés élémentaires des nombres à virgule flottante sur le système utilisé, notamment le nombre de bits de la mantisse. Voici comment procéder en Python.

```
>>> from sys import *      # importation du module sys (tout le module)
>>> sys.float_info.mant_dig # nombre de bits de la mantisse
53                         #  $53 = 52 + 1$ , le  $+1$  va être expliqué ci-dessous
                           # l'ordinateur travaille donc en double précision
```

Voici un exemple d'un nombre N à virgule flottante représenté en simple précision.

Diagram illustrating the floating-point representation:

0	10000001	11100000000000000000000000000000
signe (1 bit)	exposant (8 bits)	pseudo-mantisse (23 bits)

et le même représenté en double précision :

The diagram illustrates the structure of a floating-point number. It consists of three main fields: 'signe' (sign), 'exposant' (exponent), and 'pseudo-mantisse' (mantissa). The 'signe' field is a single bit at the left end. The 'exposant' field is a group of 11 bits immediately following the signe. The 'pseudo-mantisse' field is a group of 52 bits starting after the exposant. The entire sequence of bits is enclosed in a bracket above, representing the binary value.

On calcule la valeur décimale d'un flottant en utilisant la formule qui suit :

$$(-1)^{\text{signe}} \times 1.[\text{pseudo-mantisse}] \times 2^{\text{exposant-décalage}}$$

Reprendons l'exemple du nombre N précédent représenté en simple précision par

0	10000001	11100000000000000000000000000000
---	----------	----------------------------------

- le premier bit est 0 donc le calcul commence par $(-1)^0 = +1$. Le nombre est positif;
 - la pseudo-mantisse est 111000000000000000000000 donc il faut la faire précéder de « 1. » pour obtenir la mantisse, ce qui explique le +1 du code précédent page 27 (on rajoute un chiffre). On obtient donc le nombre
$$1,111\ 000\ 000\ 000\ 000\ 000\ 000\ 00_2 = [1 + 2^{-1} + 2^{-2} + 2^{-3}]_{(10)} = 1,875_{(10)}$$
 - enfin avec l'exposant $10000001_2 = 129_{(10)}$ et en tenant compte du *décalage* égal à 127^1 , on obtient donc $2^{129-127} = 2^2$.

Finalement

$$N = +1,875 \times 2^2 = 7,5$$

Le décalage permet de coder des « petits » nombres. En effet, pour un exposant plus petit que le décalage, le nombre *exposant-décalage* est négatif, et la puissance de 2 correspondante sera un nombre très « petit ».

Autre exemple : écrivons en virgule flottante sur 32 bits le nombre $-1234,25_{(10)}$. Transformons l'écriture de ce nombre :

$$\begin{aligned}
 -1234,25_{(10)} &= -10011010010,01_{(2)} \\
 &= -1,001101001001_{(2)} \times [2^{10}]_{(10)} \\
 &= -1,001101001001_{(2)} \times [2^{137-127}]_{(10)} \\
 &= (-1)^1 \times 1,001101001001_{(2)} \times [2_{(10)}]^{10001001_{(2)} - 1111111_{(2)}},
 \end{aligned}$$

d'où la représentation :

1	10001001	001101001001000000000000
---	----------	--------------------------

3.5.2 Quelques nombres particuliers

Quelques représentations particulières sont réservées pour des nombres particuliers.
Signalons seulement (sur 32 bits)

qui représente 0 et qui est le plus simple des nombres dénormalisés ; les nombres dénormalisés sont des nombres dont le codage ne respecte pas les règles décrites ci-dessus et qui permettent entre autre d'étendre la plage des nombres représentables.

011111111000000000000000000000000000000

représente l'infini (utile pour les calculs dépassant les capacités de ces représentations). Il existe encore d'autres nombres, appelés NaN (*Not a Number*). Leur rôle dépasse les objectifs de ce livre.

$1 - 127 = 2^{8-1} - 1$ en simple précision ; le décalage est de $2^{11-1} - 1 = 1023$ en double précision.

3.6 Les limites des représentations des nombres

3.6.1 Il y a un nombre fini de nombres représentables

Une première conséquence évidente de la représentation des entiers, naturels ou relatifs ou des flottants est que dans ces représentations, qu'elles soient sur 8, 16 (entiers naturels), 32 ou 64 bits il n'y a qu'un nombre fini de nombres représentables possibles. Par exemple, sur 32 bits on peut représenter au plus $2^{32} = 4\,294\,967\,296$ nombres différents.

3.6.2 Des nombres non représentables

Les entiers les plus grands représentables sont, suivant la machine utilisée, $2^{32} - 1$ ou $2^{64} - 1$. Cependant certains langages de programmation, dont Python, permettent de contourner ce problème de façon transparente pour l'utilisateur.

Voici un exemple en Python :

```
>>> 2**80 # calcul de 280
1208925819614629174706176
```

Par contre, pour les flottants, les nombres « trop grands » ne sont pas représentables. On parle de dépassement ou plus couramment d'*overflow*. Voici un *overflow* en Python :

```
>>> 2.0 * 10**(309) # calcul de 2 × 10309
Traceback (most recent call last):
  File "<stdin>", line 1, in <module> # un message d'erreur apparaît
OverflowError: long int too large to convert to float # Overflowerror # le type d'erreur ci-dessous
```

De même des réels très petits ne sont pas représentables (en virgule flottante) :

```
>>> sys.float_info.min # plus petit flottant positif normalisé
2.2250738585072014e-308 # résultat
>>> 10**(-324) # affichage d'un nombre petit 10-324
0.0 # résultat arrondi à zéro
```

On parle ici d'*underflow* ou dépassement par valeurs inférieures.

3.6.3 Les problèmes d'arrondis

Partons des égalités :

$$\begin{aligned}
 \underbrace{2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + 2^{-10} + 2^{-11} + \dots}_{\text{développement périodique infini}} &= (1 + 2^{-1})(2^{-2} + 2^{-6} + 2^{-10} + \dots) \\
 &= 2^{-2}(1 + 2^{-1})(1 + 2^{-4} + 2^{-8} + \dots) \\
 &= \frac{1}{4} \times \left(1 + \frac{1}{2}\right) \times \left(\lim_{n \rightarrow +\infty} \sum_{k=0}^n (2^{-4})^k\right) \\
 &= \frac{1}{4} \times \frac{3}{2} \times \frac{1-0}{1-2^{-4}} \\
 &= \frac{2}{5} \\
 &= 0,4
 \end{aligned}$$

On en déduit que

$$0,01100110011001\cdots_{(2)} = 0,4_{(10)}$$

Le nombre $0,4_{(10)}$, qui a une écriture décimale simple, a une écriture infinie en base 2, donc sa représentation en flottant, suivant la norme IEEE754 qui ne contient qu'un nombre fini de chiffres dans la mantisse, sera une **valeur approchée** (troncature) de sa valeur. De même en réutilisant le résultat précédent (écriture de 0,4), puisque

$$\begin{aligned} 0,1 &= \frac{1}{10} = \frac{1}{4} \times \frac{2}{5} = 2^{-2}(2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + 2^{-10} + 2^{-11} + \dots) \\ &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-10} + 2^{-11} + \dots \end{aligned}$$

on en déduit que 0,1 a un développement binaire illimité.

Remarque : dire qu'un nombre peut s'écrire sous la forme $m \times 2^e$ avec $1 \leq m < 2$ équivaut à pouvoir l'écrire sous la forme $m' \times 2^{e'}$ où m' est un entier. Par exemple, $0,1 \times 2^k$ n'est pas un entier pour tout $k \in \mathbb{N}$, donc 0,1 ne peut s'écrire de façon exacte avec la norme IEEE754.

Ces arrondis de nombres sont la source d'**erreurs d'arrondis**. Exemple :

<code>>>> 0.4 - 0.1</code>	# deux nombres dont la représentation est un arrondi
<code>0.3000000000000004</code>	# résultat avec une accumulation d'arrondis

Ces arrondis rendent **négligeables** certains nombres devant d'autres :

<code>>>> 1 + 2**(-53)</code>	# avec des ordres de grandeur éloignées : 1 et 2^{-53}
<code>1.0</code>	# la puissance de 2 est « absorbée »

Ces arrondis peuvent rendre invalides des règles de calcul (ici la commutativité) :

<code>>>> 1 + 2**(-53) - 1</code>	# un calcul très simple
<code>0.0</code>	# Python calcule de gauche à droite, 2^{-53} est absorbé
<code>>>> 1 - 1 + 2**(-53)</code>	# le même calcul dans un ordre différent
<code>1.1102230246251565e-16</code>	# un résultat non nul

Les conséquences de ces arrondis sont nombreuses et il faudra en tenir compte :

- les divisions par des nombres « petits » peuvent conduire à des divisions par zéro ;
- les **comparaisons d'égalité de deux flottants** sont à proscrire ; mieux vaut tester si leur différence en valeur absolue est suffisamment petite ;
- les erreurs d'arrondis peuvent s'accumuler, il faudra veiller à l'ordre des opérations en particulier en analyse numérique.

EXERCICES

EXERCICE 3.1 ● Les différentes représentations (exercice classique)

1. Donner l'écriture en base 2 de 12, de 57, de 255 et de 7924.
2. Donner l'écriture en base 10 des cinq nombres suivants (écrits en base 2) : 111, 1111, 10101010, 10000000 et 0,111.
3. Déterminer le complément à deux sur 8 bits puis 16 bits de 55, de -5 et de -54.
4. Quel est le plus grand entier naturel représentable sur 32 bits ?

Quels sont le plus grand et le plus petit entier relatif que l'on peut coder sur 32 bits ?
 Quel nombre est représenté en simple précision par

0	10000010	10101000000000000000000000000000	?
---	----------	----------------------------------	---

Représenter en simple précision sur 32 bits les réels 456,671 875, 18,5 et -2,6875.

EXERCICE 3.2 Algorithme et flottants

Voici un algorithme :

ARIABLES x, y : flottants

n : entier

ÉBUT

$\leftarrow 1.0$

$\leftarrow 2.0$

$\leftarrow 0$

ANT_QUE $y-x = 1.0$ FAIRE

$x \leftarrow x*2.0$

$y \leftarrow x+1.0$

$n \leftarrow n+1$

IN_TANT_QUE

FFICHER n

IN

réponse affichée est alors 53.

Que valent les variables dans les trois premiers tours de boucle ? Comment expliquer le fait que la boucle s'arrête ?

Expliquer ce qu'affiche le programme (réponse en lien avec l'architecture ou le codage). Que se passerait-il lors de son exécution si x et y étaient des nombres entiers ?

EXERCICE 3.3 Exactitude de la représentation des flottants

Parmi les cinq nombres suivants, indiquer pour chacun d'eux s'il peut être représenté non de façon exacte selon la norme IEEE754 sur 32 bits :

0,40625	0,50011	0,03035	$1/4 + 1/8 + 3/32$	$1/4 + 1/8 + 1/72$
---------	---------	---------	--------------------	--------------------

EXERCICE 3.4 Logarithme binaire

1 note \log_2 le logarithme binaire (ou logarithme de base 2), $\log_2 : x \mapsto \frac{\ln(x)}{\ln(2)}$ pour $x > 0$.

Soit (e, m) (exposant, mantisse) la représentation en virgule flottante d'un réel positif x avec $1 \leq m < 2$. Exprimer x en fonction de 2 et m et en déduire $\log_2(x)$ en fonction de $\log_2(m)$ et e .

Déterminer $m \in [1; 2[$ tel que le calcul de $\log_2(10)$ se ramène au calcul de $\log_2(m)$.

Déterminer $\log_2(1)$, $\log_2(x)$ en fonction de $\log_2(x^2)$ et $\log_2(x)$ en fonction de $\log_2(\frac{x}{2})$.

L'objectif de cette question est de déterminer une approximation écrite en base 2 du logarithme binaire d'un réel positif x quelconque. La question 1. permet de ramener la question pour un réel x avec $1 \leq x < 2$.

L'algorithme suivant utilise les deux résultats de la question précédente et permet de déterminer une valeur approchée à n chiffres de l'écriture en base 2 de $\log_2(x)$ avec $1 < x < 2$.

```

TANT_QUE ( le nombre de chiffres de l'écriture binaire de  $\log_2(x)$ 
est inférieur à n) FAIRE :
    x ← x*x
    SI x ≥ 2, ajouter 1 à l'écriture binaire, diviser x par 2
    SINON, ajouter 0 à l'écriture binaire
FIN_TANT_QUE

```

Utiliser cet algorithme pour obtenir l'approximation $\log_2(1,25) \approx 0,01010_{(2)}$.

- En déduire : $\log_2(10) \approx 3,301010_{(2)}$.

EXERCICE 3.5 • Interpréter une erreur

Voici un code Python (après le prompt >>>), des commentaires (après le #) et le résultat dessous. Quelle conclusion tirer des deux résultats ?

```

>>> 2.0**1023          # 2.0 exposant 1023
8.98846567431158e+307 # réponse
>>> 2.0**1024          # 2.0 exposant 1024
Traceback (most recent call last):      # réponse (3 lignes de)
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')

```

EXERCICE 3.6 Problèmes posés par les flottants

- Comparer les résultats du même calcul effectué avec **deux types différents**, les types *int* et *float* (utiliser un émulateur via une recherche sur Python online si Python n'est pas encore installé) :

```

>>> 2**100   # Résultat : .....
>>> 2.0**100 # Résultat : .....

```

Quelle conclusion en tirer ?

- Les flottants sont-ils très précis ? Effectuer

```
>>> 0.1+0.2 # Résultat : .....
```

Conclusion ?

- Peut-on comparer des flottants ? Sachant que == compare l'égalité entre deux éléments, effectuer et expliquer les résultats suivants (voir 5.4 page 47 pour plus de détails) :

```

>>> 1+2 == 3      # Résultat : .....
>>> 2+2 == 3      # Résultat : .....
>>> 0.1+0.2 == 0.3 # Résultat : .....

```