



Modulo.

Une introduction à l'informatique

Groupe de travail DGEP, EPFL, HEP-VD, UNIL

17 juillet 2023



Table des matières

1	Représentation de l'information	1
1.0	Introduction	1
1.0.1	Alphabets anciens et traditionnels	2
1.0.2	Le carré de Polybe	3
1.0.3	Le télégraphe de Chappe	3
1.0.4	Le Morse	4
1.0.5	Le binaire	7
1.1	Les entiers	9
1.1.1	Les systèmes de numération	9
1.1.2	Représentation des entiers négatifs	13
1.2	Les caractères	17
1.2.1	Principe	17
1.2.2	Table ASCII	18
1.2.3	Standard UTF	19
1.2.4	Exercices	21
1.3	Les images	23
1.3.1	Les images matricielles	23
1.3.2	Représentation d'une image en noir et blanc	24
1.3.3	Représentation d'une image en niveaux de gris	25
1.3.4	Représentation d'une image en couleurs	26
1.3.5	Les images vectorielles	29
1.3.6	Bonus	30
1.3.7	Exercices	30
1.4	Le son	33
1.4.1	Numérisation	34
1.4.2	Échantillonnage	35
1.4.3	Quantification	36
1.4.4	Encodage	37
1.4.5	Reconstruction	38
1.5	Redondance	39

1.5.1	Somme de contrôle (checksum)	39
1.5.2	Fonction de hachage	41
1.5.3	Disques RAID	42
1.5.4	Cloud computing	42
1.6	Conclusion	43
2	Algorithmique I	47
2.0	Introduction	47
2.0.1	Quoi?	47
2.0.2	Pourquoi?	47
2.0.3	Comment?	48
2.0.4	Objectifs d'apprentissage	48
2.1	Les algorithmes	49
2.1.1	Résolution d'un problème par étapes	49
2.1.2	Les ingrédients d'un algorithme	53
2.1.3	Exercices	56
2.2	Trie, recherche et trouve	59
2.2.1	Algorithmes de tri	59
2.2.2	Tri par insertion	60
2.2.3	Tri par sélection	60
2.2.4	Tri à bulles	61
2.2.5	Comparaison d'algorithmes	64
2.2.6	Exercices	65
2.3	Des algorithmes aux programmes	67
2.3.1	Exercices	70
2.4	Conclusion	73
2.4.1	Les algorithmes et nous	73
2.4.2	Focus sur l'automatisation	73
3	Architecture des ordinateurs	75
3.0	Introduction	75
3.0.1	De quoi sont faits les nombres binaires?	75
3.0.2	Électricité et nombres binaires	77
3.0.3	Le transistor	78
3.0.4	Des transistors aux systèmes logiques	80
3.1	Portes logiques	81
3.1.1	Exemple suivi : addition de deux nombres	81
3.1.2	Portes logiques	82
3.1.3	Porte ET	82
3.1.4	Porte OU	84
3.1.5	Porte NON	84
3.1.6	Combinaisons de portes	85
3.2	Additionneur	95
3.2.1	Additionneur complet	96
3.2.2	Chaînage d'additionneurs	97
3.3	ALU	103
3.3.1	Sélection de l'opération	103

3.3.2	Une ALU à 4 bits	109
3.4	Mémoire	115
3.4.1	Le verrou SR	115
3.4.2	La bascule D	117
3.4.3	Addition en plusieurs étapes	120
3.4.4	Récapitulatif	123
3.5	CPU	125
3.5.1	Horloge et accès mémoire	127
3.5.2	L'unité arithmétique et logique	129
3.5.3	Processeur à noyau unique	131
3.5.4	Processeur à double cœur	131
3.5.5	Les processeurs quadricœur et autres processeurs à cœurs multiples	132
3.5.6	Le pipeline	134
3.6	Architecture générale	137
3.6.1	La mémoire	137
3.6.2	Le CPU (Central Processing Unit)	141
3.6.3	Les entrées-sorties	143
3.6.4	Les bus	144
3.6.5	Autres composants matériels	146
3.7	Conclusion	149
3.8	TP Portes logiques	155
3.8.1	Transmission d'un signal	155
3.8.2	Commutateur/poussoir	155
3.8.3	Feu de circulation	156
3.8.4	Lampe clignotante	156
3.8.5	Affichage à 7 segments	157
3.8.6	Affichage à 2 chiffres	158
3.8.7	Affichage à 16 segments	158
3.8.8	Porte NON	159
3.8.9	Afficher 0 et 1	159
3.8.10	Alterner 0 et 1	160
3.8.11	Porte OU	160
3.8.12	Décodeur de clavier	161
3.8.13	Clavier numérique	161
3.8.14	Porte ET	162
3.8.15	Décodeur binaire	162
3.8.16	Décodeur de dé	163
3.8.17	Décoder 0 à 3	164
3.9	TP Additionneur	167
3.9.1	Clock et fréquence	167
3.9.2	Porte OU-X	168
3.9.3	Construire un OU-X	168
3.9.4	Détecteur de parité	169
3.9.5	Multiples commutateurs	170
3.9.6	Addition binaire	170
3.9.7	Additionneur complet	171
3.9.8	Additionneur 4 bits	172

3.9.9	Incrémenter (<i>i</i> ++)	173
3.9.10	Décrémenter (<i>i</i> --)	174
3.9.11	Changer de signe (- <i>i</i>)	174
3.9.12	Soustraction (<i>a</i> - <i>b</i>)	175
3.9.13	Inversion commutée	176
3.9.14	Négation commutée	177
3.9.15	Soustraction commutée	177
3.9.16	Les fanions (flags)	178
3.10	TP ALU	179
3.10.1	Sélectionneur	179
3.10.2	Multiplexeur	179
3.10.3	Sélection d'opérations	180
3.10.4	ALU	181
3.10.5	Addition signée	182
3.10.6	Addition 8 bits	182
3.10.7	Carry et Overflow (V)	183
3.10.8	Multiplier 1x1 bit	184
3.10.9	Multiplier par 2, 4, 8	185
3.10.10	Multiplier 1x4 bit	185
3.10.11	Multiplier 4x4 bits	186
3.10.12	Diviser par 2, 4 et 8	187
3.10.13	Registre	188
3.10.14	Accumulateur	188
3.10.15	Incrémenter/décrémenter	189
3.10.16	Comparer	190
3.11	TP Mémoire	193
3.11.1	Cellule de mémoire	193
3.11.2	Verrou SR	194
3.11.3	Bascule D	194
3.11.4	Registre 4 bits	195
3.11.5	Décodeur de touche	195
3.11.6	Décodeur pour 8 touches	196
3.11.7	Compteur 4 bits	197
3.11.8	Compteur 8 bits	197
3.11.9	Compteur avec remise	198
3.11.10	Rouler un dé	199
3.11.11	Registre à décalage	200
3.11.12	RAM (mémoire vive)	200
3.11.13	RAM avec bits aléatoires	201
3.11.14	RAM avec binaire	202
3.11.15	RAM avec image	202
3.11.16	RAM avec ASCII	203
3.12	TP CPU	205
3.12.1	Intel 4004	205
3.12.2	Langage assembleur	206
3.12.3	Le langage machine	207
3.12.4	Mémoire de programme	207

3.12.5	Le jeu d'instructions	208
3.12.6	Décoder une instruction	209
3.12.7	Bus 4 bits	209
3.12.8	Charger imméd. (LDM)	210
3.12.9	Charger depuis reg (LDR)	211
3.12.10	Choix entre LDR/LDM	212
3.12.11	Choix entre ADD/SUB	213
3.12.12	Program counter (PC)	214
3.12.13	Le saut (jump)	215
3.12.14	La pile (stack)	216
3.12.15	Projet final	217
3.12.16	Nand Game	218
3.13	Objectifs	219
3.14	Personnages-clés	221
4	Algorithmique II	223
4.0	Introduction	223
4.0.1	Quoi ?	223
4.0.2	Pourquoi ?	224
4.0.3	Comment?	224
4.0.4	Objectifs	224
4.1	Terminaison et complexité	225
4.1.1	Principe de terminaison	225
4.1.2	Principe de complexité	227
4.2	Algorithmes de recherche	229
4.2.1	Recherche linéaire	229
4.2.2	Recherche binaire	235
4.2.3	Exercices	241
4.3	Algorithmes heuristiques	243
4.3.1	Complexité exponentielle	243
4.3.2	Exercices	247
4.4	Algorithmes de tri [niveau avancé]	249
4.4.1	Tri par sélection	249
4.4.2	Tri rapide	251
4.4.3	Exercices	256
4.5	Récursivité [niveau avancé]	259
4.5.1	Tri par fusion	259
4.5.2	Focus sur la récursivité	261
4.5.3	Exercices	267
4.6	Conclusion	271
5	Réseaux	273
5.0	Introduction	273
5.0.1	Origine d'Internet	273
5.0.2	Structure d'internet	274
5.0.3	Fonctionnement d'Internet	275
5.0.4	Organisation du chapitre	277

5.1	Adressage	279
5.1.1	Les noms de domaine	279
5.1.2	Les adresses IP	280
5.1.3	Système de noms de domaine	284
5.2	Paquets et protocoles	287
5.2.1	Les paquets	287
5.2.2	Le protocole IP	288
5.2.3	Le protocole TCP	289
5.2.4	Le protocole UDP	292
5.3	Routage	295
5.3.1	Les routeurs	295
5.3.2	Les tables de routage	296
5.3.3	Le routage dynamique	297
5.4	World Wide Web	301
5.4.1	Historique	301
5.4.2	Les technologies du Web	301
5.4.3	Les évolutions du Web	304
5.5	Interopérabilité	307
5.5.1	Un modèle en couches	307
5.5.2	Des protocoles ouverts	309
5.5.3	La neutralité d'Internet	309
5.5.4	L'universalité d'Internet en question	310
5.6	Conclusion	311
6	Histoire de l'informatique	313
6.0	Histoire de l'informatique	313
6.0.1	Préinformatique	314
6.0.2	Histoire moderne de l'informatique	318
6.0.3	Académisation et apparition de la science informatique	320
6.0.4	Apparition de la micro-informatique : Small is beautifull	323
6.0.5	Internet	326
6.0.6	Le logiciel libre	327
6.0.7	Le World Wide Web	328
6.0.8	Ubiquité	328
6.0.9	Futur	329

Représentation de l'information

1.0 Introduction

Dans ce chapitre on s'intéresse à la manière dont un *ordinateur* représente l'information afin de pouvoir la traiter automatiquement.

Le saviez-vous ?

Le mot **informatique** est la concaténation de «information» et «automatique».

En informatique, l'information est un élément de connaissance (texte, image, son, ...) susceptible d'être *numérisé*, *stocké* et/ou *transmis* à l'aide d'un support et d'un mode de codification normalisé.

Une des questions centrales de ce chapitre est d'identifier les caractéristiques de la transformation appliquée au réel donnant une représentation suffisamment précise pour permettre aux ordinateurs de la *traiter* de manière fiable.

Mais avant de découvrir le code choisi pour représenter l'information à l'intérieur d'un ordinateur, passons en revue un certain nombre d'alphabets et de systèmes de représentation de l'information qui ont été utilisés au cours de l'histoire.

1.0.1 Alphabets anciens et traditionnels

Depuis qu'elle existe, l'espèce humaine a créé de nombreux alphabets, ainsi que de nombreux *systèmes de communication*. Depuis les sumériens¹⁴ qui utilisaient des *pictogrammes* et l'*écriture cunéiforme*¹⁵, en passant par les égyptiens et leurs *hiéroglyphes*¹⁶, les chinois et leurs *idéogrammes*¹⁷ pour arriver aux symboles de nos alphabets actuels, l'espèce humaine n'a eu de cesse de mettre au point des systèmes pour représenter l'information et la *transmettre*.

<p>Sumérien</p>	<p>Égyptien</p>
<p>Chinois</p>	<p>Synoptique</p>

Différentes représentations de la même information

- Nombres en chiffres classiques, romains, lettres
- Mot en différentes langues, morse, idéogrammes
- Symboles danger, stop, paix

On trouve des exemples célèbres et bien documentés de *systèmes de communication* depuis l'Antiquité grecque.

14. <https://fr.wikipedia.org/wiki/Sum%C3%A9rien>

15. <https://fr.wikipedia.org/wiki/Cun%C3%A9iforme>

16. https://fr.wikipedia.org/wiki/%C3%89criture_hi%C3%A9roglyphique_%C3%A9gyptienne

17. https://fr.wikipedia.org/wiki/Caract%C3%A8res_chinois

1.0.2 Le carré de Polybe

Utilisé en Grèce Antique pour transmettre des messages entre cités voisines, ce système utilisait des torches enflammées en guise de signaux.

Cinq torches «à gauche», cinq torches «à droite», étaient séparées par un espace suffisamment grand pour être identifiables à longue distance. Une torche pouvait être soit allumée, soit éteinte. Le nombre de torches allumées à gauche, de 1 à 5, représentait la ligne d'un tableau de décodage, le nombre de torches allumées à droite représentait la colonne de ce même tableau.

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I,J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

FIG. 1.1 – Le codage de la lettre «s» dans le carré de Polybe est quatre torches à gauche, trois torches à droite.

Remarque

Dans l'exemple ci-dessus, on utilise les lettres de l'alphabet, mais il est plus probable qu'à l'époque les cités voisines utilisaient des expressions toutes faites dans chacune des cases, comme «l'ennemi est à nos portes» ou «envoyez-nous de l'aide».

1.0.3 Le télégraphe de Chappe

Grâce à l'invention du [télescope](#)¹⁸ au XVIIe siècle, les distances avec lesquelles les villes pouvaient communiquer entre elles ont largement augmenté. L'information a commencé à circuler à une vitesse étonnante. Des messages pouvaient être transmis sur une longue distance par l'intermédiaire de relais espacés d'une dizaine de kilomètres et situés sur des hauteurs.

[Claude Chappe](#)¹⁹, inventeur français, développe en 1794 un *télégraphe* capable de relier des villes entre elles sur plusieurs dizaines de kilomètres grâce à un système de bras mobiles, qui ressemblent aux signaux que pourrait faire un être humain sur le tarmac d'un aéroport.

18. <https://fr.wikipedia.org/wiki/T%C3%A9lescope>

19. https://fr.wikipedia.org/wiki/Claude_Chappe

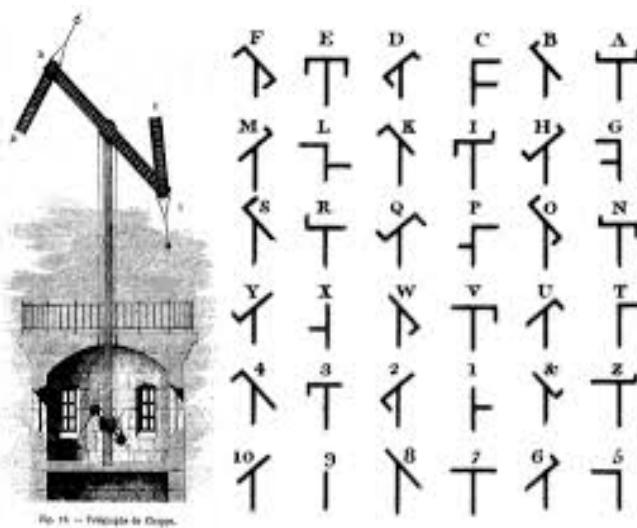


FIG. 1.2 – Le télégraphe de Chappe émet des signaux ressemblant aux bras d'un être humain.

Le saviez-vous ?

Le piratage du télégraphe Chappe²⁰ est un détournement du réseau de télégraphie optique français entrepris par deux hommes d'affaires bordelais, Louis et François Blanc, entre 1834 et 1836, afin de connaître avant tout le monde le prix de certaines actions à la Bourse de Paris.

Le piratage a été rendu possible par la corruption d'un agent télégraphique de Tours, qui ajoutait discrètement le prix actuel des actions aux messages envoyés par l'État.

20. https://fr.wikipedia.org/wiki/Piratage_du_t%C3%A9l%C3%A9graphe_Chappe

1.0.4 Le Morse

Grâce à la découverte de l'électricité au début du XIXe siècle, et les améliorations techniques faites pour la capturer et la transmettre, on a pu utiliser le réseau électrique pour envoyer des messages. En 1832, naît le **code Morse**²¹, qui s'impose rapidement comme un standard de communication.

Bien sûr, le Morse peut être utilisé aussi avec des signaux lumineux, ou sonores, mais la plupart du temps il est utilisé sur les lignes électriques qui se développent à l'époque.

Vous trouverez [ici](#)²² un traducteur du langage naturel vers le Morse.

Micro-activité

Amusez-vous avec votre assistant vocal en lui demandant par exemple : «Salut Siri. Quel est le code Morse pour *j'ai envie de dormir* ?».

21. https://fr.wikipedia.org/wiki/Code_Morse_international

22. <https://morsedecoder.com/>

Code morse international

1. Un tiret est égal à trois points.
2. L'espacement entre deux éléments d'une même lettre est égal à un point.
3. L'espacement entre deux lettres est égal à trois points.
4. L'espacement entre deux mots est égal à sept points.

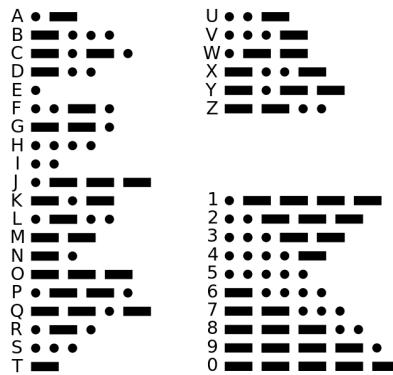


FIG. 1.3 – Le code Morse est le système de représentation de l’information qui se rapproche le plus du langage binaire de l’informatique moderne.

Si vous observez le [code Morse](#)²³, vous remarquerez que les signaux utilisés pour représenter les lettres ne suivent pas simplement l’ordre de l’alphabet, puisqu’il est plus économique de coder les lettres les plus fréquentes avec les codes les plus courts.

Le saviez-vous ?

À l’époque où les transmissions télégraphiques en code Morse sont payées à l’unité d’information, donc la lettre, des codex spécifiques sont développés par les utilisateurs pour utiliser le moins de caractères possibles. C’est exactement la même situation qui s’est produite avec l’arrivée des [SMS](#)²⁴ dans les années 1990, où les utilisateurs payaient au caractère. Aujourd’hui, même s’il est rare de payer à l’unité d’information, ce genre de raccourcis existent encore, mais surtout pour un avantage de vitesse.

23. https://fr.wikipedia.org/wiki/Code_Morse_international

CW Abbreviations

73	best regards	MSG	message
88	love and kisses	NR	number
ABT	about	NW	now
AGN	again	OM	old man
ANT	antenna	OP	operator
BK	break	R	are, received, roger
CPY	copy	RCVR	receiver
CQ	general call to any station	RIG	equipment
CUL	see you later	RST	readability, strength, tone report
CU	see you	SIGS	signal
DE	from	STN	station
DX	distance, rare station	TEMP	temperature
ES	and	TKS	thanks
FB	fine business	TNX	thanks
FER	for	UR	you are
FRE	frequency	U	you
Q			
GA	good afternoon, go ahead	WL	well
GE	good evening	WT	watt
GM	good morning	WX	weather
HR	here	XCVR	transceiver
HW	how	XMTR	transmitter
K	go ahead	XYL	wife

Le désavantage de ces codex d'abréviations est leur faible degré de standardisation. Comment savoir quel codex est utilisé ? Et surtout : comment faire pour que tout le monde s'accorde sur le codex ?

La réponse à cette question est l'apport le plus essentiel de l'introduction du code binaire, et des standards de représentation de l'information qui l'ont suivi : un langage pour les contrôler tous.

24. https://fr.wikipedia.org/wiki/Short_Message_Service

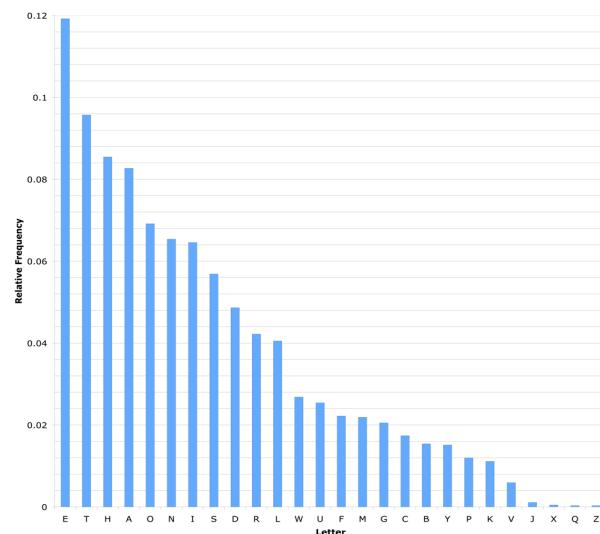


FIG. 1.4 – Ceci est une représentation de la fréquence moyenne de distribution des lettres dans la langue anglaise.

1.0.5 Le binaire

À partir du moment où le Morse²⁵ a été inventé comme système de *codage* et de *transmission* de l'information par l'électricité, il ne manquait plus que quelques éléments pour commencer à construire les *ordinateurs*.

Une pièce technologique, qui permettrait de *transmettre* pour ainsi dire cette information : le *transistor* (voir *architecture des ordinateurs*).

Un *code* plus élaboré que le Morse pour pouvoir représenter tous les types d'informations possibles à partir d'une alternative entre deux états : courant ou pas courant ; allumé ou éteint ; vrai ou faux ; 1 ou 0.

Ce *code* est le *code binaire*. Il permet, en utilisant uniquement des 0 et des 1, de représenter n'importe quel type d'information : des chiffres, du texte, des images, du son, des vidéos, etc.

Question 1

Pourquoi la lettre «e», en Morse, est-elle représentée par un seul point ?

1. Parce que c'est la lettre la plus utilisée en anglais.
2. Par hasard.
3. Parce que c'est la lettre la plus rare en anglais.
4. Parce que c'était la lettre préférée de l'inventeur du Morse.

Réponse: 1

Question 2

Que signifie informatique ?

1. Information + quantique.
2. Information + technique.
3. Information + automatique.
4. Information + pratique

Réponse: 3

25. https://fr.wikipedia.org/wiki/Code_Morse_international

1.1 Les entiers

La plupart des civilisations humaines utilise le système décimal. Pourquoi ? Tout simplement parce que nous avons 10 doigts !

L'ordinateur, lui, n'a pas de doigts mais utilise l'électricité. Par conséquent, il ne connaît que deux types d'informations : il y a du courant, il n'y a pas de courant ; allumé, éteint ; vrai, faux ; 1 ou 0.

On dit qu'il travaille dans un système binaire, ou en base deux.

1.1.1 Les systèmes de numération

Le système décimal

Ce système est composé de 10 symboles qui sont les chiffres :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ainsi, tout nombre écrit dans la base 10 est composé de ces chiffres.

La valeur de chaque chiffre dépend alors du chiffre lui-même et de sa place. Ainsi, le 3 de 1934 et celui de 3008 n'ont pas la même valeur : Le premier vaut 30, alors que le second vaut 3000. On parle alors de **représentation positionnelle en base 10**.

Dans ce système, pour connaître la valeur de chaque chiffre qui compose un nombre, il faut décomposer ce nombre pour identifier chaque chiffre et son coefficient, c'est la **forme canonique**.

Décomposition du nombre 3528 :

- 8 unités
- 2 dizaines
- 5 centaines
- 3 milliers

Sa forme canonique est : $3 \cdot 10^3 + 5 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0$

On peut alors vérifier que le nombre 3528 est bien dans la base 10, car tous ces chiffres appartiennent à la base 10. Les nombres de la base 10 ou du système décimal sont des nombres décimaux.

Le système binaire

Le système binaire, ou numération positionnelle en base 2, est représenté à l'aide d'uniquement 2 symboles : 0 et 1. Cette représentation et la représentation décimale sont deux représentations, parmi d'autres, d'un même concept.

Un élément binaire se nomme un *bit* et un ensemble de *bits* peut représenter un entier en utilisant le même principe que pour le système décimal.

La valeur de chaque chiffre dépend toujours de sa place qui représente, cette fois, une puissance de 2.

La forme canonique du nombre binaire $1101_{(2)}$ est : $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Le saviez-vous ?

Le *bit* vient de la terminologie anglo-saxonne de *binary digit*. Un ensemble de 8 bits est appelé un **octet**. Un *kilo-octet* (ko) correspond à 10^3 octets soit 1000 octets, donc 8000 bits. Attention à ne pas confondre les préfixes binaires (2^{10} , 2^{20} , 2^{30} , etc.) et les préfixes décimaux (10^3 , 10^6 , 10^9 , etc.). On appelle *kibioctet*, pour kilo binaire, une quantité de $2^{10} = 1024$ octets. On peut remarquer que cette notation, quoique rigoureuse, peine à s'imposer dans le vocabulaire courant des ingénieurs eux-mêmes...

Compter en binaire

On compte en binaire de la même manière que l'on compte en base 10 en ajoutant 1 aux unités (position la plus à droite). Lorsqu'on arrive au dernier chiffre (i.e. 9 en base 10 et 1 en base 2), on le remet à 0 et on augmente de 1 le chiffre à sa gauche.

On répète ces opérations pour tous les chiffres, quelle que soit leur position. Ainsi, en base 10 :

0; -; 1; -; 2; -; 3; -; ...; -; 9; -; 10; -; 11; -; ...; -; 99; -; 100; -; 101; -; ...

En binaire, on obtient : 0; -; 1; -; 10; -; 11; -; 100; -; 101; -; 110; -; 111; -; 1000; -; ...

Micro-activité

Comptez jusqu'à 40 en binaire. Que pouvez vous observer au sujet de la parité des nombres binaires ? Pourquoi ?

Conversion du système binaire vers le système décimal

La conversion d'un nombre binaire en nombre en base 10 se fait aisément grâce à la forme canonique.

En effet, il suffit de calculer le résultatat de la somme pondérée par les puissances de 2.

Conversion du nombre 10101

$$10101_{(2)} = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21_{(10)}$$

Le *tableau* ci-dessous permet de convertir un octet en nombre décimal.

Coefficient	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Valeur	128	64	32	16	8	4	2	1
Bit	0	0	1	0	1	0	1	0

L'exemple utilisé ici est l'octet $(00101010)_{(2)}$ dont la valeur décimale est : $00101010_{(2)} = 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 42_{(10)}$

Important

L'utilisation d'un tableau de conversion nécessite d'écrire le nombre binaire de droite à gauche car le bit de poids faible ($= 2^0$) se trouve à droite, de la même façon que le chiffre de poids faible (=l'unité) se trouve à droite en représentation décimale.

Micro-activité

Donnez la conversion en base 10 des nombres binaires suivants :

- 10101101
- 01110010
- 1111
- 1111111
- 1
- 10
- 100
- 1000
- 10000
- 100000
- 1000000
- 10000000

Conversion du système décimal vers le système binaire

L'opération de conversion du système décimal vers le système binaire est moins directe. Cependant, à l'aide d'un tableau de conversion et des instructions suivantes, il est possible d'obtenir la représentation binaire de n'importe quel entier positif.

Tableau de conversion

Coefficient	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Valeur	1024	512	256	128	64	32	16	8	4	2	1
Bit											

Instructions de conversion d'un entier du système décimal vers le système binaire

1. Déterminer le coefficient **maximum** dont la valeur est plus petite que l'entier à convertir.
2. Le bit associé à ce coefficient est 1.
3. Soustraire la valeur de ce coefficient à l'entier à convertir pour obtenir le nouveau nombre à convertir.
4. Recommencer à l'étape 1 tant que le nombre est différent de 0.
5. En commençant par le plus grand coefficient utilisé, le nombre binaire correspondant est composé de la suite des bits où des 0 représentent les coefficients non utilisés.

Par exemple, la conversion du nombre 666 en base 10 vers le binaire s'obtient avec les étapes suivantes :

$$666 = 512 + 154 \quad (1.1)$$

$$154 = 128 + 26 \quad (1.2)$$

$$26 = 16 + 10 \quad (1.3)$$

$$10 = 8 + 2 \quad (1.4)$$

$$2 = 2 + 0 \quad (1.5)$$

Valeur	1024	512	256	128	64	32	16	8	4	2	1
Bit		1	0	1	0	0	1	1	0	1	0

Résultat : $(666_{(10)} = 1010011010_{(2)})$

Micro-activité

Donnez la conversion binaire des nombres décimaux suivants :

- 97
- 123
- 256
- 1000
- 511

Pour aller plus loin

Pouvez-vous penser à une autre façon de convertir un entier du système décimal en binaire ?

Le saviez-vous ?

Le 4 juin 1996, le premier vol de la fusée Ariane 5 a explosé 40 secondes après l'allumage. La fusée et son chargement avaient coûté 500 millions de dollars. La commission d'enquête a rendu son rapport au bout de deux semaines. Il s'agissait d'une erreur de programmation dans le système inertiel de référence. À un moment donné, un nombre codé en virgule flottante sur 64 bits (qui représentait la vitesse horizontale de la fusée par rapport à la plate-forme de tir) était converti en un entier sur 16 bits. Malheureusement, le nombre en question était plus grand que 32767 (le plus grand entier que l'on peut coder en tant qu'entier signé sur 16 bits) et la conversion a été incorrecte, induisant un changement de trajectoire fatal.

1.1.2 Représentation des entiers négatifs

Après la représentation des entiers naturels (\mathbb{N}), on souhaite évidemment pouvoir représenter les entiers négatifs afin d'avoir une représentation des entiers relatifs (\mathbb{Z}). Un entier relatif est un entier naturel auquel on a ajouté un signe positif ou négatif indiquant sa position **relative** par rapport à 0.

Bit de signe

La première idée pour représenter un entier relatif est d'utiliser un bit pour représenter le signe. En effet, un bit peut uniquement prendre deux valeurs, 0 ou 1, comme le signe, + ou -. Les bits restants étant utilisés pour représenter un entier positif. Considérons l'encodage sur un octet (8 bits), nous réservons le bit de poids fort (la puissance de 2 la plus grande) pour le signe : 0 indique un nombre positif et 1 un nombre négatif.

Avec cette approche, un entier relatif est représenté par sa valeur absolue (entier naturel) auquel on associe un signe. Le domaine couvert se trouve donc divisé par deux, un bit étant utilisé pour le signe. Ainsi, avec un octet, 7 bits sont utilisés pour encoder la valeur absolue, soit $[0, 127]$, ce qui permet de représenter les entiers relatifs dans l'intervalle $[-127, 127]$.

La représentation de -21 est :

signe	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	1	0	1

Bien que cette approche soit simple et semble convenir, elle pose deux problèmes majeurs :

- le premier est d'avoir deux représentations pour zéro (+0 et -0),
- le second apparaît lorsque l'on additionne un nombre et son opposé. Le résultat de cette opération devrait être 0, mais, sans rentrer dans le détail de l'arithmétique binaire qui sera abordée plus tard, l'addition bit à bit avec cette représentation donne $(-2|x|)$. En effet, l'addition des bits de signe donne toujours 1 et le reste des bits représente l'addition de deux fois la valeur absolue.

Pour remédier ces problèmes, une autre représentation des entiers relatifs a été mise au point, il s'agit de la représentation en complément à deux.

Complément à deux

La représentation en complément à deux reprend l'idée du signe encodé par le bit de poids fort et conserve la représentation des entiers positifs. Donc tous les entiers positifs ont une représentation en binaire qui commence par un 0 et le plus grand entier représenté sur un octet est 127.

Les entiers négatifs sont représentés grâce au complément à deux dont voici le principe :

1. Écrire la valeur absolue du nombre en binaire (le bit de poids fort est égal à 0).
2. Inverser tous les bits, les 0 deviennent des 1 et vice-versa. Le résultat obtenu s'appelle le complément à 1 du nombre de départ.
3. On ajoute 1, la dernière retenue est ignorée le cas échéant.

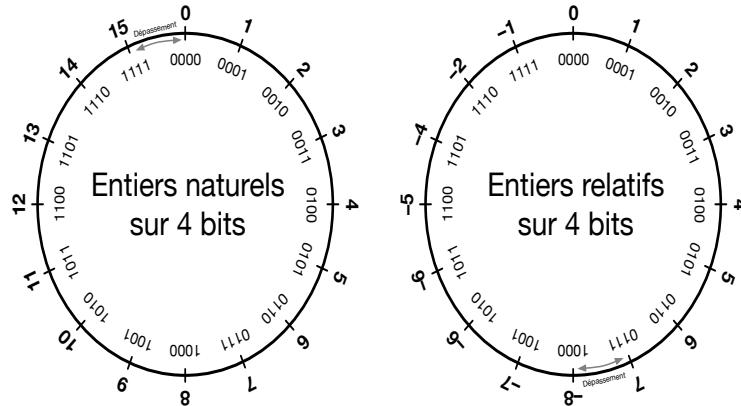
La représentation de -21 en complément à 2 :

1. Valeur absolue en binaire : 00010101

2. Complément à 1 : 11101010

3. Ajouter 1 : 11101011

La représentation de -21 est 11101011, qui additionné à 21, soit 00010101 donne bien zéro : 00000000.



Représentation des entiers avec 4 bits. La figure ci-dessus illustre la différence du domaine couvert avec 4 bits pour la représentation des entiers naturels ou des entiers relatifs. Ainsi, avec 4 bits le domaine couvert pour les entiers naturels est : [0, 15], et pour les entiers relatif : [-8, 7].

À retenir

Puisque le nombre d'entiers relatifs représentés est forcément pair et que le 0 en fait partie, il y a une asymétrie entre les nombres positifs et négatifs représentés. Par exemple, avec 4 bits on peut représenter -8, **mais pas 8**

Quel est le domaine couvert en utilisant la représentation en complément à deux sur un octet ?

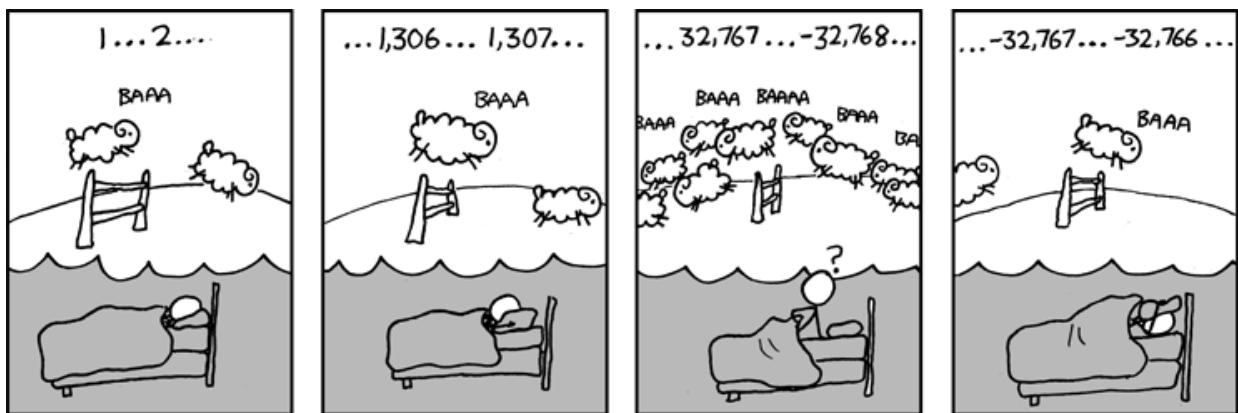
[−128, 127]

Micro-activité

Encodez les entiers relatifs suivants sur un octet :

- -99
- -1
- 127
- -128

Vous pouvez maintenant comprendre ce comic d'un robot comptant des moutons pour s'endormir... d'ailleurs, combien de bits utilise-t-il pour compter??



Source : [xkcd](https://xkcd.com/571/)²⁶

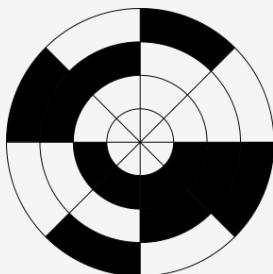
Pour aller plus loin

Le code binaire réfléchi, ou code Gray, est un type de codage binaire ne modifiant qu'un seul bit à la fois quand un nombre est augmenté d'une unité. Ce type de codage est utilisé pour les codeurs rotatifs absolu calibrés et initialisés une seule fois **qui doivent conserver leur valeur** lors de l'arrêt de l'appareil, comme les compteurs kilométriques des automobiles (à la différence du compteur journalier qui peut être remis à zéro par l'utilisateur).

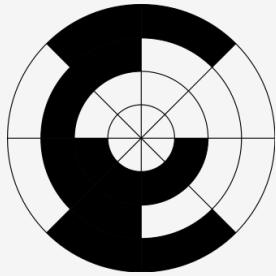
Voici le début de la table de correspondance décimal-binaire-Gray sur quatre bits (huit premières valeurs) :

Codage décimal	Codage binaire	Codage binaire réfléchi
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

Chaque encodage peut être représenté sur un disque divisé en huit secteurs identiques :



26. <https://xkcd.com/571/>



- 1 - Etablir la correspondance entre la table binaire et la figure de l'encodage binaire, puis entre la table Gray et la figure correspondante.
- 2 - En comprenant la construction des huit valeurs données dans la table de correspondance (correspondant donc aux huit secteurs des disques), tenter d'écrire la table complète sur quatre bits.
- 3 - Combien de secteurs angulaires les disques vont-ils comprendre pour un codage sur quatre bits ? Représenter alors le disque binaire puis Gray sur quatre bits.

1.2 Les caractères

Toute l'information est représentée dans un ordinateur par des nombres encodés sous forme binaire par des 0 et des 1. Se pose alors la question de la représentation des caractères, ne serait-ce que parce que la communication entre les utilisateurs et les ordinateurs s'opère essentiellement sous forme textuelle.

1.2.1 Principe

La solution est simple : on associe chaque caractère à un code binaire.

Caractère	Décimal	Hexadécimal	Binaire
A	65	0x41	01000001
B	66	0x42	01000010
C	67	0x43	01000011
...
Z	90	0x5A	01011010

Chaque caractère frappé sur le clavier est représenté par le code correspondant dans ce tableau.

Chacun des caractères de la phrase que vous lisez (qu'on nomme **chaîne de caractères**) a ainsi été stocké, transmis et manipulé par l'ordinateur sous la forme d'une séquence de 0 et 1.

Lorsqu'il s'agit de représenter ce texte à l'écran ou à l'impression, les logiciels utilisent la table dans l'autre sens pour trouver le caractère correspondant au nombre binaire.

En plus des lettres, les caractères qui représentent les chiffres sont eux-mêmes listés dans la table de conversion. Contre-intuitivement, la valeur binaire du caractère représentant un chiffre ne correspond pas au chiffre lui-même.

Caractère	Décimal	Hexadécimal	Binaire
0	48	0x30	00110000
1	49	0x31	00110001
...
9	57	0x39	00111001

Ces tables donnent également une représentation des caractères de ponctuation et des symboles mathématiques, ainsi que des caractères non-imprimables comme le retour à la ligne.

En réalité, il n'existe pas une table de conversion unique, mais des dizaines de tables de conversion. Certaines tables ont été proposées à l'origine par des constructeurs d'ordinateurs ou des éditeurs de systèmes d'exploitation.

1.2.2 Table ASCII

Le code américain normalisé pour l'échange d'information ASCII (pour American Standard Code for Information Interchange) est apparu dans les années 1960. Malgré sa large acceptation, avec ses **7 bits par caractère**, cette table avait pour principal défaut de ne pas prendre en compte les caractères qui n'existent pas dans la langue anglaise, ne serait-ce que les lettres accentuées.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	'
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Tab. 1 La table de représentation des caractères ASCII

Des tables multiples, mutuellement incompatibles, ont alors émergé : une table pour les européens, une autre pour les Japonais et ainsi de suite.

Progressivement, notamment avec l'émergence du Web au cours des années 1990, l'augmentation de l'interconnexion des ordinateurs personnels a amené au début des années 2000 à la mise en place d'une énorme table intégrant le contenu de toutes les tables existantes, via le standard UTF.

1.2.3 Standard UTF

Le [standard Unicode](#)²⁷ UTF (Universal Character Set Transformation Format) s'est imposé pour l'échange, car il permet d'agréger sur 8 bits, 16 bits ou 32 bits par caractère la totalité des caractères utilisés dans toutes les langues humaines... et même extraterrestres, puisque le [Klingon](#)²⁸ est également intégré.

Les caractères liés à l'édition des partitions de musique ou les émojis sont également intégrés.

Variantes

Pour éviter de consommer 32 bits par caractère, des variantes plus compactes ont été mises à disposition.

La plus connue – des européens, puisqu'elle regroupe les caractères qui nous concernent – est la [table UTF-8](#)²⁹. Elle se concentre sur les premiers 8 bits de la table UTF complète. Par sa nature, UTF-8 est d'un usage très répandu sur internet et dans les systèmes échangeant de l'information. Il s'agit également du codage le plus utilisé dans les systèmes de logiciels libres pour gérer le plus simplement possible des textes et leurs traductions dans tous les systèmes d'écritures et alphabets du monde. Les navigateurs internet d'aujourd'hui utilisent le codage UTF-8 et les concepteurs de sites prennent en compte cette même norme ; c'est pourquoi il y a de moins en moins de problèmes de *compatibilité*. Toutefois, toutes ces différentes normes et leurs incompatibilités sont la cause des problèmes que l'on rencontre parfois avec les caractères accentués. Il est donc préférable pour la rédaction de courriels à l'étranger, de n'utiliser que des caractères non accentués.

UTF-8 est donc un encodage des caractères basé sur UNICODE, de longueur variable qui utilise de 1 à 4 octets par symbole.

Comment s'opère le codage sur plusieurs octets ?

En UTF-8, chaque point de code de 0 à 127 est stocké dans un seul octet. Seuls les points de code 128 et supérieurs sont stockés en utilisant 2, 3 ou 4 octets. Chaque octet commence alors par quelques bits qui indiquent s'il s'agit d'un point de code à un octet, d'un point de code à plusieurs octets ou de la continuation d'un point de code à plusieurs octets :

0xxx xxxx : c'est un code US-ASCII à un seul octet (permettant donc d'encoder les 127 premiers caractères).

Les points de code multi-octets commencent chacun par quelques bits à 1 du premier octet en partant de la gauche, suivis d'un bit à 0, et qui vont dire si l'on doit lire l'octet suivant, ou les deux ou les trois suivants, pour comprendre l'encodage global. Par exemple, si l'octet le plus à gauche s'écrit :

110x xxxx : cela indique que le message global est encodé sur $1+1=2$ octets, et donc qu'un deuxième octet suit.

1110 xxxx : cela indique que le message global est encodé sur $1+1+1=3$ octets, et donc qu'un deuxième puis un troisième octet suivent.

1111 0xxx : cela indique que le message global est encodé sur 4 octets, et donc qu'un deuxième puis un troisième puis un quatrième octet suivent.

27. <https://home.unicode.org/>

28. <https://www.kli.org/about-klingon/klingon-history/>

29. <https://www.utf8-chartable.de/>

Enfin, les octets qui suivent ces codes de démarrage sont tous de la forme : 10xx xxxx. Les bits représentés par le caractère «x» représentent ce que l'on appelle la *charge utile*, c'est à dire l'encodage du caractère proprement dit.

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1110xxxx 10xxxxxx 10xxxxxx	3 octets codant 12 à 16 bits
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits

Tab. 2 Définition du nombre d'octets utilisés

Puisqu'on peut dire quel type d'octet on regarde à partir des premiers bits du premier octet à gauche, alors même si quelque chose est altéré quelque part, la séquence entière n'est pas perdue : ce codage est appelé *codage auto-synchronisant*.

Codage UTF-8 en détail

Le premier octet en partant de la droite sert lui à encoder les caractères ASCII, donnant ainsi au jeu de caractères une **totale compatibilité avec ASCII**.

Chaque caractère non ASCII (c'est à dire dont le point de code - ici le codage décimal - est supérieur à 127) se code nécessairement sur plusieurs octets, entre 2 et 4 octets ; les bits de poids fort du premier octet en partant de la gauche forment, en partant de la gauche également, une suite de 1 de longueur égale au nombre total d'octets utilisés pour coder le caractère ; les octets suivants auront 10 comme bits de poids fort comme on vient de l'écrire.

Reprendons la table ASCII de la figure 8 et la [table UTF-8](#)³⁰ : on observe que le signe ~ par exemple est sur la table ASCII à l'adresse décimale 126 (01111110 en binaire), et sera donc à la même adresse sur la table UTF-8. Même chose pour le caractère suivant, qui est le caractère de contrôle *del* qui se trouve à l'adresse 127 (01111111 en binaire). En revanche, le caractère suivant, qui est également un caractère de contrôle, bien évidemment n'apparaît plus sur la table ASCII ; sur la [table UTF-8](#)³¹, l'adresse décimale est 194 128.

Si l'on prend à présent, par exemple, le caractère «æ», on lit sur la [table UTF-8](#) : 195 166, soit, en binaire : 11000011 10100110.

On constate bien le passage du codage sur deux octets. L'adresse décimale 195 du premier octet correspond à la valeur binaire 11000011. On retrouve la suite de deux “1” en début de ce premier octet en partant de la gauche, indiquant ce codage total sur deux octets ; il reste 000011 pour la charge utile du premier octet du codage UTF-8. L'adresse décimale de 166 est 10100110 et commence donc bien par 10 comme bits de poids fort ; la charge utile du deuxième octet du codage UTF-8 est donc 100110. L'encodage binaire UTF-8 global s'écrit donc, en concaténant les deux charges utiles : 000011100110, ce qui correspond à 230 en décimal, valeur qu'on peut vérifier sur cette autre [table UTF-8](#)³² indiquant également le codage décimal.

30. <https://www.utf8-chartable.de/unicode-utf8-table.pl?number=512&utf8=dec>

31. <https://www.utf8-chartable.de/unicode-utf8-table.pl?number=512&utf8=dec>

32. https://kellykjones.tripod.com/webtools/ascii_utf8_table.html

Caractère	Numéro du caractère	Codage binaire UTF-8
A	65	0100001
é	233	11000011 10101001
€	8364	11100010 10000010 10101100
ƒ	119070	11110000 10011101 10000100 10011110

Tab. 3 Définition du nombre d'octets utilisés

Par exemple le caractère « € » (euro) est le 8365e caractère du répertoire Unicode ; son index, ou point de code, est donc 8364, il se code en UTF-8 sur 3 octets : 226, 130, et 172 exprimé en décimal (11100010 10000010 10101100 exprimé en binaire).

Type	Caractère	Point de code (hexadécimal)	Valeur scalaire		Codage UTF-8	
			décimal	binnaire	binnaire	hexadécimal
Contrôle	[NUL]	U+0000	0	0	00000000	00
	[US]	U+001F	31	1·1111	00011111	1F
Texte	[SP]	U+0020	32	10·0000	00100000	20
	A	U+0041	65	100·0001	01000001	41
	~	U+007E	126	111·1110	01111110	7E
Contrôle	[DEL]	U+007F	127	111·1111	01111111	7F
	[PAD]	U+0080	128	000 1000·0000	11000010 10000000	C2 80
	[APC]	U+009F	159	000 1001·1111	11000010 10011111	C2 9F
Texte	[NBSP]	U+00A0	160	000 1010·0000	11000010 10100000	C2 A0
	é	U+00E9	233	000 1110·1001	11000011 10101001	C3 A9
	ƒ	U+07FF	2047	111 1111·1111	11011111 10111111	DF BF
	ℳ	U+0800	2048	1000 0000·0000	11100000 10100000 10000000	E0 A0 80
	€	U+20AC	8364	10·0000 1010·1100	11100010 10000010 10101100	E2 82 AC
	ƒ	U+D7FF	55 295	1101·0111 1111·1111	11101101 10011111 10111111	ED 9F BF

Tab. 4 Extrait de la table de représentation UTF-8

1.2.4 Exercices

Exercice 1 – Utilisation de la table ASCII

- 1 - À l'aide de la table ASCII, codez en binaire la phrase suivante «L'an qui vient !».
- 2 - Voici maintenant une exclamation codée en binaire : 01000010 01110010 01100001 01110110 01101111 00100001. Retrouvez cette exclamation !
- 3 - Peut-on coder en binaire la phrase «Un âne est-il passé par là ?» à l'aide de la table ASCII (justifiez la réponse) ?

Exercice 2 – Activité codage et internet

Ouvrez un navigateur Internet (Firefox, ...). Dans la barre d'outils, on peut voir à «Affichage», «Encodage des caractères» que c'est le format UTF-8 qui est sélectionné par défaut.

1 - Changez la sélection UTF-8 et choisissez à présent Europe Centrale (Windows). De petits caractères désagréables apparaissent. Que s'est-il passé ?

2 - Utilisez toujours le navigateur web, et allez dans «Affichage», «Source». On lit alors l'entête de la page *html* visitée. Où se situe l'information relative à l'encodage ?

3 - On peut aussi dans «Affichage», «Codage», sélectionner Grec (ISO) et se rendre compte en lisant le texte, que le «à» a été remplacé par un «L» à l'envers dit *Gamma*.

Exercice 3 – Coder en UTF-8

Le symbole Ø correspond à la valeur décimale 8709.

1 - Convertissez cette valeur en binaire.

2 - Combien d'octets doit-on utiliser en UTF-8 pour coder ce nombre convenablement (les moitiés d'octet sont interdites) ?

3 - Donnez le codage UTF-8 correspondant.

Micro-activité – Hexadécimal

Nous avons vu au cours du chapitre précédent deux systèmes de numération, décimal et binaire. Il existe également un troisième système de numération très utilisé, le système hexadécimal, visible par ailleurs sur les tables. Le système binaire permet d'exprimer n'importe quel nombre en base 2 (soit 0, soit 1), le système décimal en base 10 (de 0 à 9) - c'est notre mode de représentation usuel. Le système hexadécimal permet d'exprimer n'importe quel nombre en base 16 : de 0 à 9... puis les lettres A, B, C, D, E, F.

1 - Selon vous, comment s'expriment les nombres décimaux 6, 8, 11, 14 et 16 en hexadécimal ?

2 - Exprimer 34 puis 128 en hexadécimal.

3 - A quels nombres décimaux correspondent les nombres hexadécimaux 80, puis 9A ?

4 - En prenant la valeur décimale 154, essayez de décrire une méthode permettant de passer du système décimal au système hexadécimal.

5 - En reprenant la valeur hexadécimale 9A, essayez de décrire une méthode permettant de passer du système hexadécimal au système décimal.

1.3 Les images

1.3.1 Les images matricielles

Depuis des siècles les humains gardent des traces de leur environnement sous forme d'images. Plus le temps passe, plus ces traces sont fidèles. On découvre par exemple la perspective autour du XVe siècle, les progrès en optique et en chimie permettent ensuite la création de la camera obscura et de la photographie argentique. Enfin l'informatique se développe permettant l'invention de la photographie numérique.

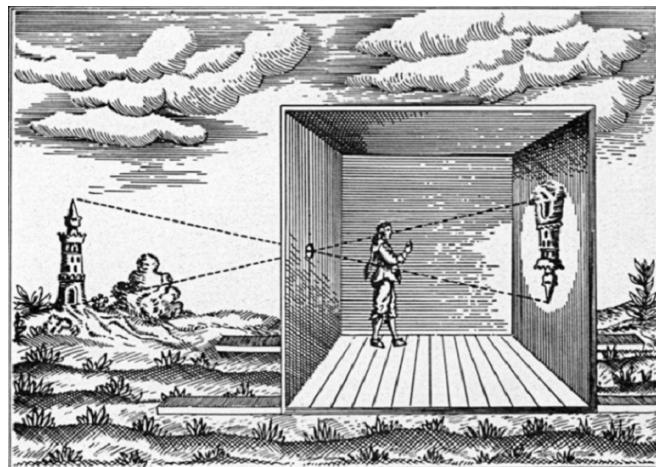


FIG. 1.5 – Principe de fonctionnement de la camera obscura.



FIG. 1.6 – Une caméra obscura.

De la camera obscura à la caméra numérique

Comment fonctionne une caméra numérique ? Une caméra numérique fonctionne en fait d'une manière très similaire à la caméra obscura et aux appareils photographiques analogiques d'un point de vue optique. Imaginez une chambre noire pourvue d'un trou sur l'une de ses parois. La lumière venant de l'extérieur vient se projeter sur le mur opposé.

Dans un appareil analogique, la paroi illuminée est recouverte d'une pellicule chimique photosensible qui permet de capturer l'image.

La différence est que dans un appareil photo numérique cette paroi, le capteur photographique, est recouverte d'une grille de capteurs électroniques photosensibles (photosites) produisant de l'électricité quand ils reçoivent de la lumière. Chaque photosite est recouvert d'un filtre coloré ne laissant passer que les rayons d'une seule couleur (grille de Bayer) : le rouge, le vert ou le bleu. Les filtres sont répartis par carré de quatre : deux verts, un rouge et un bleu. La tension électrique produite par chaque photosite est convertie numériquement et transmise au processeur de l'appareil photo.

L'image numérique ne sera alors rien d'autre que la collection des mesures de tous les capteurs à un temps précis. Comme ces mesures sont organisées sous forme de tableau (grille), on parle souvent d'images matricielles. Plus le nombre de capteurs est grand, plus la définition de cette image le sera aussi.

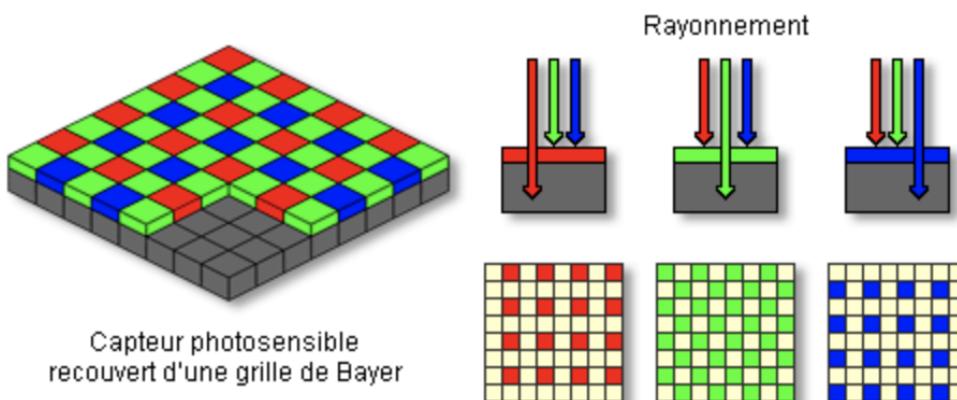


FIG. 1.7 – Principe de la capture numérique d'une image.

1.3.2 Représentation d'une image en noir et blanc

Un bit est l'unité minimale d'information qu'un ordinateur manipule : 1 ou 0, allumé ou éteint. L'image la plus simple qu'un ordinateur puisse afficher est constituée uniquement de noir et blanc. Ainsi, un pixel pourrait être à l'état soit «noir», soit «blanc».

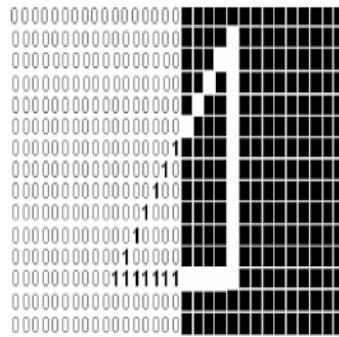


FIG. 1.8 – Tous les pixels marqués d'un 1 s'affichent en blanc, tous ceux marqués d'un 0 s'affichent en noir. Ceci nous permet de construire des images simples, dessinées seulement en noir et blanc.

Un *pixel*, de l'anglais “picture element”, est le composant minimal d'une image. C'est à dire que c'est le plus petit élément avec lequel on construit une image sur un écran d'ordinateur. Dans notre exemple minimaliste, chaque pixel peut être soit noir, soit blanc, ce qui nous permet de construire une image.

1.3.3 Représentation d'une image en niveaux de gris

Dans ce type d'image seul le niveau de l'intensité est codé sur un octet (256 valeurs). Par convention, la valeur 0 représente le noir (intensité lumineuse nulle) et la valeur 255 le blanc (intensité lumineuse maximale) :

0	8	16	32	56	72	90	104	112	128
136	144	160	176	192	208	224	244	248	255

FIG. 1.9 – Niveaux de gris, codage sur 8 bits.

En général, les images sont représentées sous forme de tableau numérique, aussi appelé format *matriciel*. Une image en niveau de gris sera ainsi représentée par un tableau de valeurs correspondant à la *luminance* de chaque pixel. Les valeurs de luminance sont des nombres allant de 0 (noir) à 255 (blanc). Pour encoder une image en niveaux de gris, chaque pixel nécessite donc 8 bits.

Pour accéder à un pixel particulier, il faut indiquer à quelle ligne et à quelle colonne de l'image ce pixel se trouve. Le pixel (0,0) correspondra normalement au pixel de la première ligne et de la première colonne.

Le saviez-vous ?

Ce mode de fonctionnement est similaire à celui des tableurs pour lesquels il est possible d'accéder à la valeur d'une case en utilisant sa référence. On pourrait d'ailleurs utiliser le formatage conditionnel pour transformer un tableau de valeurs dans un tableur en image matricielle.

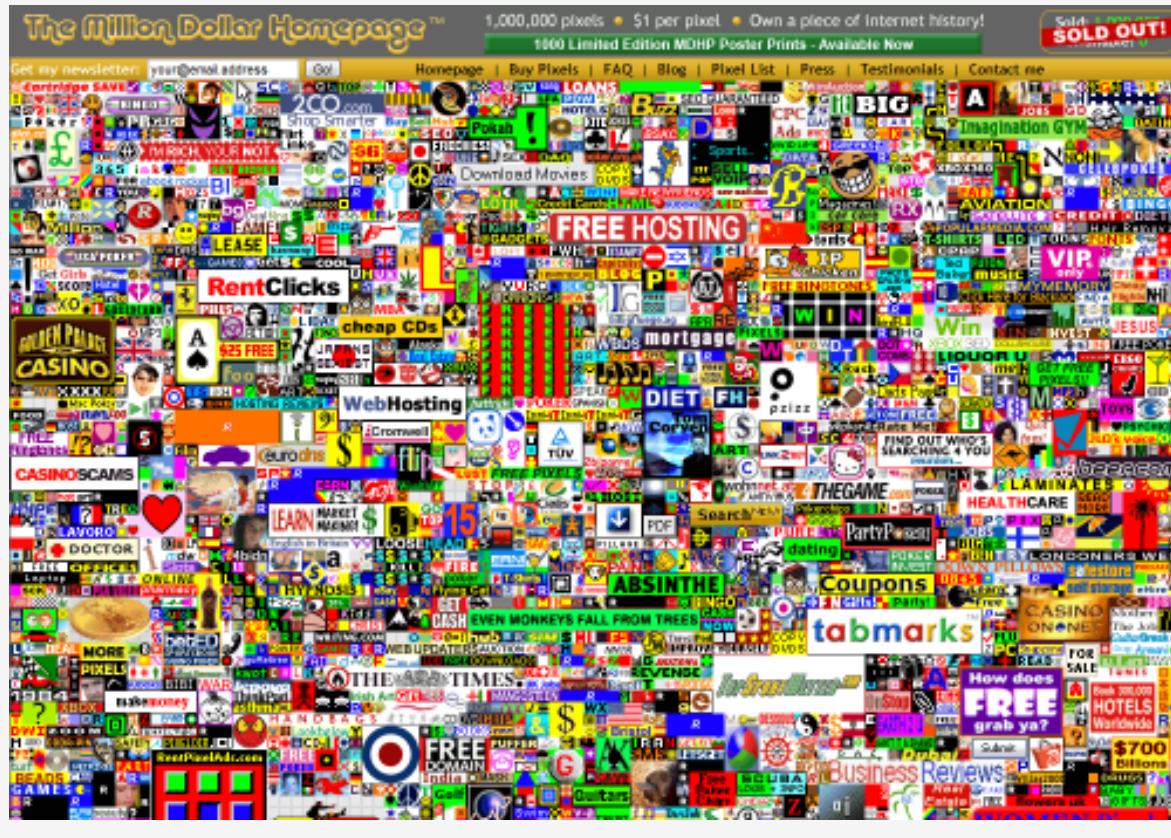


FIG. 1.10 – Image monochrome, pixels et luminance.

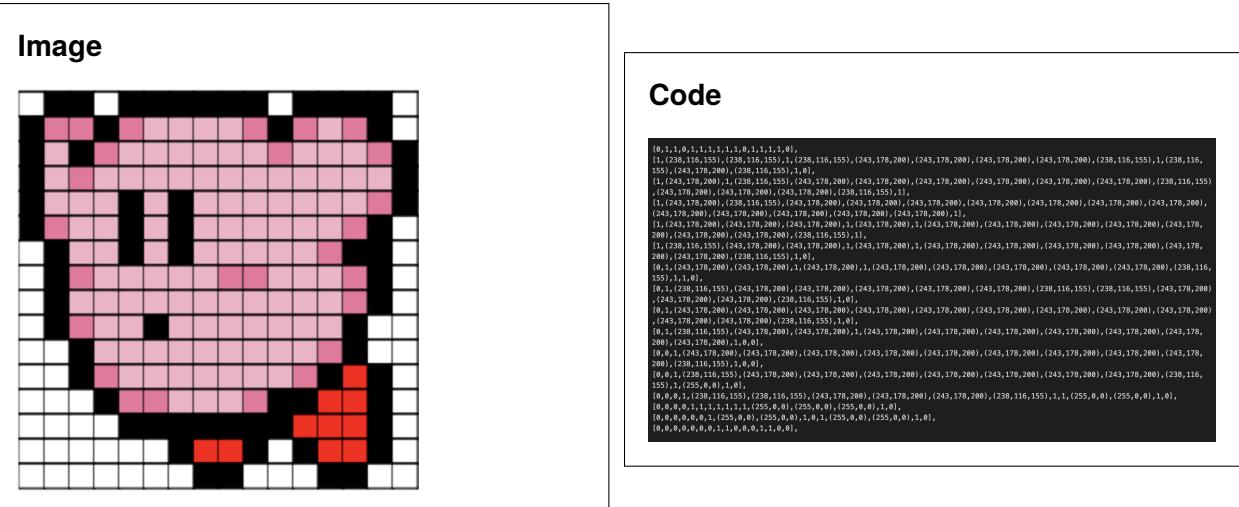
1.3.4 Représentation d'une image en couleurs

Le saviez-vous ?

The Million Dollar Homepage³³ est un site web conçu en 2005 par Alex Tew, un étudiant anglais, dans le but de financer ses études supérieures. La page d'accueil est une grille de 1000 × 1000 pixels. Chaque pixel était vendu 1\$ en tant qu'espace publicitaire. Ils ont tous été vendus...



33. https://fr.wikipedia.org/wiki/The_Million_Dollar_Homepage



En peinture, pour obtenir toutes les couleurs de l'arc-en-ciel, on utilise un mélange de magenta, de cyan et de jaune, qui vont chacune absorber une partie de la lumière; c'est ce que l'on appelle la *synthèse soustractive*: en ajoutant du pigment à une surface, une partie du spectre lumineux est soustraite.

Pour faire la même chose sur un écran, on utilisera également trois couleurs, mais celles-ci seront le rouge, le vert et le bleu (couleurs primaires). Cela correspond à la *synthèse additive* : en allumant une LED rouge par exemple, on ajoute de la lumière sur la partie du spectre lumineux correspondant.

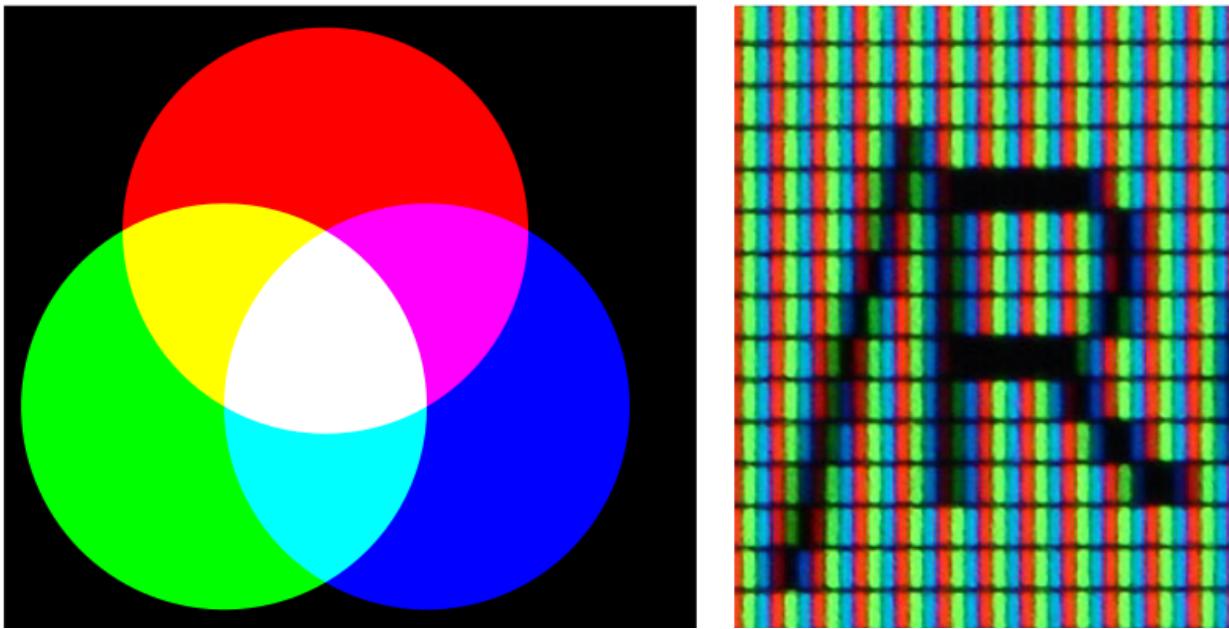


FIG. 1.11 – Système additif et écran au microscope.

Chaque couleur est donc représentée comme un mélange de ces trois couleurs et donc sous forme de trois entiers (triplet). Comme pour les images en niveaux de gris, ces entiers sont généralement représentés sur 8 bits ; les valeurs de luminance sont chacune déclarées comme un nombre allant de 0 (intensité nulle) à 255 (intensité maximale). Pour représenter une image en couleurs il faut donc 8 bits pour le niveau de rouge, 8 bits pour le niveau de vert, et 8 bits pour le niveau de bleu, soit 24 bits.

Dans cette animation³⁴ vous pouvez zoomer sur chacun des pixels qui constituent l'image totale. Chaque pixel possède trois valeurs allant de 0 à 255. RGB signifie en anglais Red, Green, Blue.

Dans cette autre animation³⁵ vous pouvez jouer avec la valeur de Rouge, Vert, Bleu, pour créer une couleur finale. L'outil vous permet d'abord de jouer avec des couleurs codées en 24 bits, puis en 8 bits, ce qui illustre bien la précision qu'on arrive à atteindre avec 24 bits.

Les formats matriciels sont Portable Network Graphics (.png), Joint Photographic Experts Group (.jpeg), Tagged Image File Format (.tiff), BITMAP (.bmp), Graphics Interchange Format (.gif) pour citer les plus courants.

Définition et résolution

On appelle *définition* le nombre de points (pixel) constituant l'image, c'est-à-dire sa « dimension informatique » (le nombre de colonnes de l'image que multiplie son nombre de lignes). Une image possédant 640 pixels en largeur et 480 en hauteur aura une définition de 640 pixels par 480, notée 640x480 soit 307200 pixels.

La *résolution*, terme souvent confondu avec la *définition*, détermine en revanche le nombre de points par unité de longueur, exprimé en points par pouce (PPP, en anglais DPI pour Dots Per Inch), un pouce représentant 2.54 cm. La résolution permet ainsi d'établir le rapport entre le nombre de pixels d'une image et la taille réelle de sa représentation sur un support physique. Une résolution de 300 dpi signifie donc 300 colonnes et 300 rangées de pixels sur un pouce carré, ce qui donne donc 90000 pixels sur un pouce carré. La résolution de référence de 72 dpi nous donne un pixel de 1/72 (un pouce divisé par 72) soit 0.353 mm, correspondant à un point pica (unité typographique anglo saxonne).

Les dimensions d'une image sont donc définies par :

- largeur = nombre de colonnes / résolution,
- hauteur = nombre de lignes / résolution.

Compression

La plupart de ces formats utilisent des algorithmes de compression, afin de réduire la taille de l'image sur les mémoires de masse de l'ordinateur (disque durs, ...).

On définit alors le taux de compression par : $(1 - (\text{taille du fichier image})) / (\text{taille de l'image en mémoire})$

La compression peut être réalisée avec ou sans perte :

- sans perte : l'image compressée est parfaitement identique à l'originale,
- avec perte : l'image est plus ou moins dégradée, selon le taux de compression souhaité.

34. <https://www.csfieldguide.org.nz/en/interactives/pixel-viewer/>

35. <https://csfieldguide.org.nz/en/interactives/colour-matcher/>

1.3.5 Les images vectorielles

Pour reproduire une image sur une feuille, on peut la diviser en grille et définir un niveau de gris pour chaque case, mais on peut aussi tout simplement dessiner une figure, par exemple un trait d'un millimètre d'épaisseur allant d'un point A à un point B de l'image. De la même manière, en informatique, il est possible de représenter des images sous forme de grilles de pixels, comme nous l'avons vu, mais il est en effet également possible de définir une image comme une collection d'objets graphiques élémentaires (un segment, un carré, une ellipse...) sur un espace plan : c'est le principe des images vectorielles.

L'image vectorielle est dépourvue de matrice. Elle est en fait créée à partir d'équations mathématiques. Cette image numérique est composée d'objets géométriques individuels, des *primitives géométriques* (segments de droite, arcs de cercle, polygones, etc.), définies chacunes par différents attributs (forme, position, couleur, remplissage, visibilité, etc.) et auxquels on peut appliquer différentes transformations (rotations, écrasement, mise à l'échelle, inclinaison, effet miroir, symétrie, translation, et bien d'autres ...).

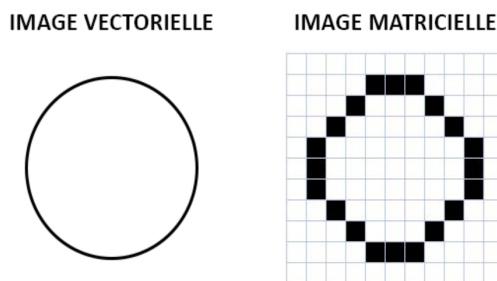


FIG. 1.12 – Un même cercle en représentation matricielle et vectorielle.

À l'inverse de l'image matricielle composée de pixels, l'image vectorielle peut être **redimensionnée** sans pour autant perdre en qualité. Elle est contenue dans un **fichier beaucoup plus léger** qu'une image pixelisée, indépendamment de sa taille et de sa définition. En revanche, chaque forme d'une image vectorielle est remplie d'une seule couleur dite solide ou d'un dégradé de couleurs. Elle reste donc **limitée en termes de réalisme**, et donc inutilisable en photographie par exemple. De plus une image vectorielle ne peut être **créée qu'à partir d'un logiciel dédié**, et n'est pas reconnue par les navigateurs internet.

Les formats vectoriels les plus courants sont Postscript (.ps) et Encapsulé Postscript (.eps), Adobe Illustrator (AI), Portable Document Format (PDF), WMF (format Windows).

Micro-activité

Saisissez le texte suivant dans un éditeur de texte et enregistrer le sous forme de fichier .svg. Il vous sera ensuite normalement possible d'ouvrir ce fichier avec un logiciel pour afficher les images.

```
<svg width="100" height="100" version="1.1" xmlns="http://www.w3.org/2000/svg">
<circle cx="50" cy="50" r="40" stroke="black" stroke-width="2" fill="red"/>
</svg>
```

Modifier le fichier afin de dessiner quatre carrés différents.

Pour aller plus loin

Identifiez et listez les avantages et les inconvénients du format vectoriel en comparaison avec le système matriciel.

1.3.6 Bonus

Une œuvre d'art numérique signée Andreas Gysin ...³⁶

1.3.7 Exercices

Exercice 1 – Définition

Quelle est la définition d'une feuille scannée de largeur 6,5 pouces, de hauteur 9 pouces en 400 dpi ?

Exercice 2 – Carte graphique

1 - Calculer, pour chaque définition d'image et chaque couleur, la taille mémoire nécessaire à l'affichage.

Définition de l'image	Noir et blanc	256 couleurs	65000 couleurs	True color
320x200				
640x480				
800x600				
1024x768				

Exercice 3 – Compression

1. Une image de couleur a pour format : 360×270 . Elle est enregistrée en bitmap 8 bits. Quelle est sa taille sur le disque dur (détaillez les calculs) ?
2. Une image noir et blanc de format 1024×1024 est enregistrée en JPG. Le taux de compression est de 50%. Quelle est sa taille sur le disque dur (détaillez les calculs) ?

36. https://play.erdfgcvb.xyz/#/src/demos/doom_flame_full_color

Exercice 4 – Appareil photo

L'appareil numérique FinePix2400Z (Fujifilm) permet la prise de vue avec trois définitions : a) 640x480 pixels ; b) 1280x960 pixels ; c) 1600x1200 pixels.

Calculez la taille de l'image non-compressée pour chaque définition.

Exercice 5 – Pixelisation

Une image numérique de définition 1024×768 mesure 30 cm de large et 20 cm de haut.

1. Déterminez les dimensions des pixels.
2. On a une photographie de 10 cm sur 5 cm que l'on scanne avec une résolution de 300 ppi. Quelle sera alors la taille de l'image (en nombre de pixels) ?
3. Soit une image 15×9 cm, définie en RVB, que l'on scanne en 72, 300 et 1200 ppi. Quels seront les poids des images, pour une profondeur de 16 bits par couleur ?

1.4 Le son

Un son est une histoire d'énergie et de vibrations. Un son émerge quand des molécules subissent une pression initiale, ce qui va les amener à avancer et entraîner ce mouvement sur les molécules devant immédiatement voisines en leur transmettant une grande partie de cette énergie. Suite à ce nouveau mouvement, elles repartent en arrière pour retrouver leur position d'équilibre ayant transmis cette énergie initiale aux molécules voisines qui à leur tour vont se comporter de la même manière.



Vidéo 1: <https://www.youtube-nocookie.com/embed/kW9nwkrfGFw>

Toutes ces «tranches» de molécules vont donc osciller successivement, formant une onde qui va se déplacer au sein du milieu matériel : air, eau, caoutchouc par exemple. C'est ce que l'on peut observer lorsqu'un projectile heurte une flaue d'eau : à partir du point d'impact, se forme progressivement une onde circulaire qui s'étend et se propage à la surface de l'eau.



Vidéo 2: <https://www.youtube-nocookie.com/embed/Yi3LW5riHfc>

Le son est donc une **vibration mécanique**, nécessitant un **milieu matériel** : s'agissant des sons que nous entendons tous les jours, le milieu matériel est bien évidemment l'air ambiant.

On appelle **fréquence** du son, la vitesse avec laquelle ces molécules vibrent. Plus la vibration des molécules est rapide, plus le son est aigu : on parle de fréquence élevée. Inversement, plus la vibration est lente, plus basse est la fréquence. Une corde de guitare détendue vibre moins vite que sa voisine très tendue, elle va produire un son plus grave avec une oscillation bien plus lente.

Le niveau sonore correspond lui à la hauteur de l'oscillation : on parle d'**amplitude**.

Ce phénomène physique d'oscillation des molécules dans l'air est capté par notre oreille en mettant en vibration nos organes qui vont convertir cette pression reçue en signaux électriques transmis au cerveau. Votre musique préférée est donc une addition de sons avec des fréquences et amplitudes différentes qui vont vous faire vibrer au sens propre... comme au figuré !

Entre phénomène physique et organe sensoriel, le son physique (on parle également de son **analogique**) va être un ensemble d'oscillations, de vibrations, définies par des fréquences et des amplitudes.



Vidéo 3: <https://www.youtube-nocookie.com/embed/XFyT1bsSnHI>

Chaque «son élémentaire» peut ainsi être assimilé à une courbe comme celle décrite dans la vidéo : on parle de courbe sinusoïdale, ou encore de sinusoïde. Les sons ou la musique que vous écoutez n'est autre qu'une somme de ces courbes «convenablement» arrangées.

La question est de savoir comment ramener ces oscillations sinusoïdales combinées ensemble en un ensemble de 0 et 1 pour être stockées numériquement dans un ordinateur, comme les nombres, images et caractères.

Le saviez-vous ?

Les casques à conduction osseuse transmettent les vibrations directement à l'os temporal du crâne : la cochlée qui est nichée dans cet os va vibrer et transmettre les informations électriques au cerveau, comme le ferait un signal passant par le tympan et le marteau.

Le saviez-vous ?

Vous rappelez-vous l'explosion de l'étoile de la mort dans Star Wars ? Eh bien un son pareil ne peut exister dans l'espace : il n'y a pas assez de molécules à agiter, l'énergie transmise par l'explosion ne peut pas se propager de la sorte.

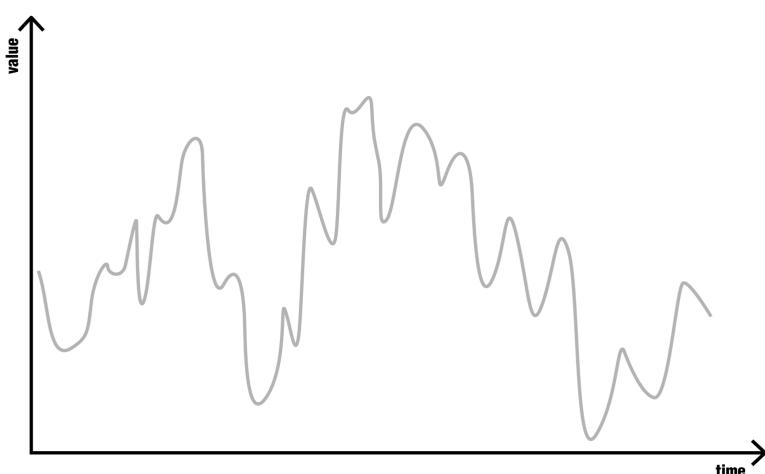
1.4.1 Numérisation

La conversion d'une grandeur physique analogique continue – température, vitesse du vent, position d'une girouette, etc. – en données numériques digitales est appelée **numérisation**. Elle est réalisée en trois étapes : un **échantillonnage**, une **quantification** puis un **encodage**.

Le processus de numérisation engendre une quantité d'information (des bits) qui vise à représenter, aussi précisément que nécessaire, la grandeur physique sous une forme manipulable par les ordinateurs.

Il s'agit donc d'un compromis entre la qualité de la représentation et les coûts engendrés par un fichier plus grand, qui prend plus de place de stockage, plus de temps à copier, à transmettre sur un réseau et/ou nécessite une puissance de calcul plus importante pour la numérisation (conversion analogique/digitale) et pour la **reconstruction** (conversion digitale/analogique).

Ci-après, un signal continu sera numérisé, mettant en évidence le rôle et les effets des différents paramètres de la numérisation. Il s'agira pour l'exemple de l'intensité sonore telle qu'elle peut être capturée par un microphone.



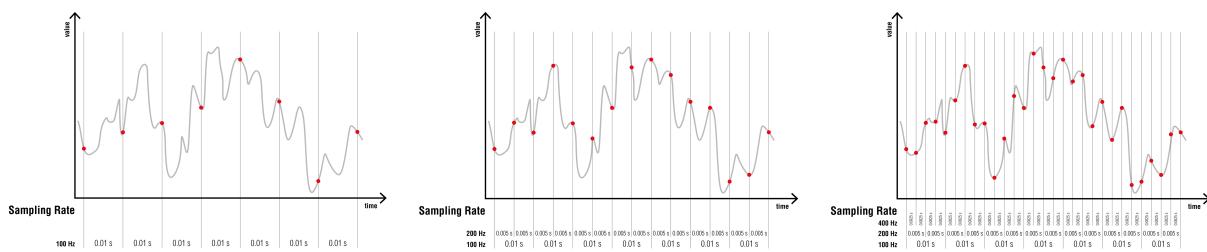
Le saviez-vous ?

Les sons, tels que perçus par notre ouïe, résultent de la vibration de l'air, prenant la forme d'oscillations cycliques de la pression.

1.4.2 Échantillonnage

L'intervalle temporel entre deux mesures est appelé la période d'échantillonnage. La **fréquence d'échantillonnage** (sampling rate) est le nombre de mesures prises par seconde, exprimée en Hz.

Les limites pratiques d'un échantillonnage sont fixées par la fréquence de Nyquist, qui, de façon très simplifiée, indique que l'information découlant d'un processus dont la fréquence est supérieure à la moitié de la fréquence d'échantillonnage sera perdue lors de la numérisation. Il ne sera donc jamais possible d'avoir une représentation complète d'un processus complexe, tout au mieux une représentation suffisante. Comme toute activité d'ingénierie, la solution retenue résulte d'une pesée d'intérêts et non d'une évidence pointant vers une solution unique.



Sachant que l'oreille humaine ne perçoit globalement que les fréquences comprises entre 20 et 20000 Hz, une fréquence d'échantillonnage supérieure à 40 kHz permettra de restituer l'ensemble de l'information physiologiquement perceptible par l'oreille humaine.

Le saviez-vous ?

La fréquence d'échantillonnage de 44.1 kHz a été retenue dans les années 1970 pour permettre l'utilisation des bandes vidéo pour stocker les enregistrements numériques. Ces bandes représentaient le meilleur rapport volume de stockage/prix pour l'époque.

C'est la raison pour laquelle l'échantillonnage de la musique en qualité "CD" est réalisé à 44.1 kHz, en prenant en compte une petite marge pour l'utilisation de filtres passe-bas lors de l'enregistrement.

Une fréquence d'échantillonnage inférieure génère un son dont la qualité est dégradée, à commencer par la précision des sons les plus aigus, aboutissant à une numérisation qui rappelle le son des anciens téléphones analogiques dont les fréquences transmises étaient limitées à 3.4 kHz pour des raisons techniques.

D'ailleurs, les premiers téléphones mobiles ont par la suite répliqué ce niveau de qualité sonore comme base pour leur échantillonnage numérique (norme G.711). Selon la norme utilisée, les téléphones mobiles actuels transmettent quant à eux les fréquences jusqu'à 7 kHz (normes G.722.2 et suivantes).

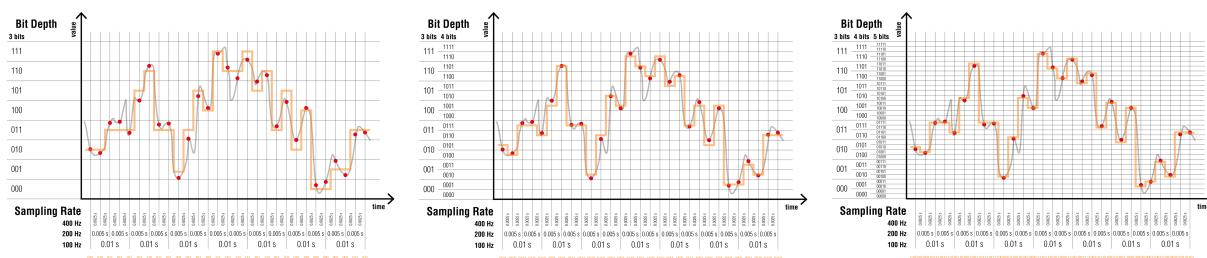
Une fréquence d'échantillonnage supérieure génère une plus grande quantité d'information, sans ajouter de valeur qualitative pour la très grande majorité des auditeurs.

Le choix de la fréquence d'échantillonnage résulte donc d'une délicate balance entre coûts (taille des données) et bénéfices (qualité de la numérisation).

1.4.3 Quantification

La quantification d'une valeur échantillonnée requiert de déterminer la **précision** de chaque échantillon, ce qui détermine le volume de données générées. Cela découle de la *représentation des entiers* par les ordinateurs.

La précision de l'encodage est donnée par la **profondeur de l'échantillonnage** (bit depth) exprimée en bits (binary digits). Comme pour l'échantillonnage, plus la profondeur de l'échantillonnage est importante, plus la quantité d'information générée est importante.



Lorsque l'ensemble de la plage des valeurs possibles est utilisé pour l'encodage, la profondeur de l'échantillonnage définit la **plage dynamique** disponible. Elle est définie entre la valeur encodée la plus petite (0, par exemple) et la valeur encodée la plus élevée ($2^n - 1$ pour une valeur encodée sur n bits, par exemple). Elle correspond également à une idée de précision ou de discrimination des échantillons.

Ici encore, l'oreille humaine ne peut percevoir ni les intensités les plus faibles (inférieures au bruit émis par l'individu lui-même) ni les intensités au-delà du seuil de douleur.

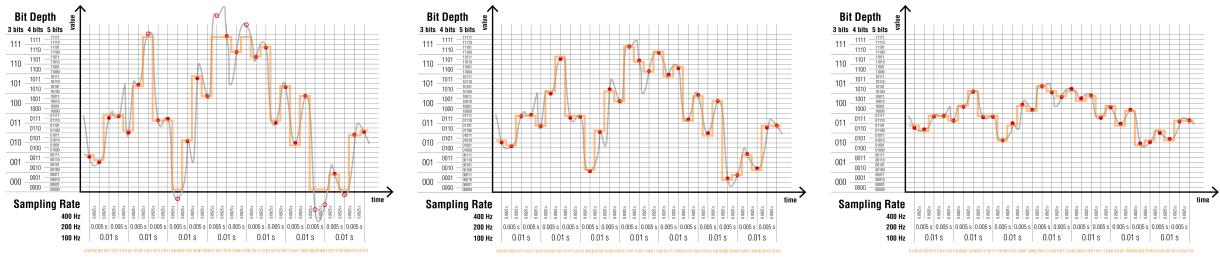
Une précision minimale (environ 8 bits) est ainsi nécessaire pour restituer agréablement un enregistrement respectant les subtilités de l'expression orale (entre voix posée et criée, par exemple).

Au-delà de 16 bits, une profondeur d'échantillonnage supérieure engendre une plage dynamique qui n'a pas d'application pertinente pour la restitution des sons pour la plupart des humains, au coût d'une plus grande quantité d'information collectée.

À l'inverse, il est nécessaire de gérer correctement la plage d'amplitude dans laquelle la numérisation du signal se déroule. Cela s'opère en agissant sur le paramètre de **gain** du signal.

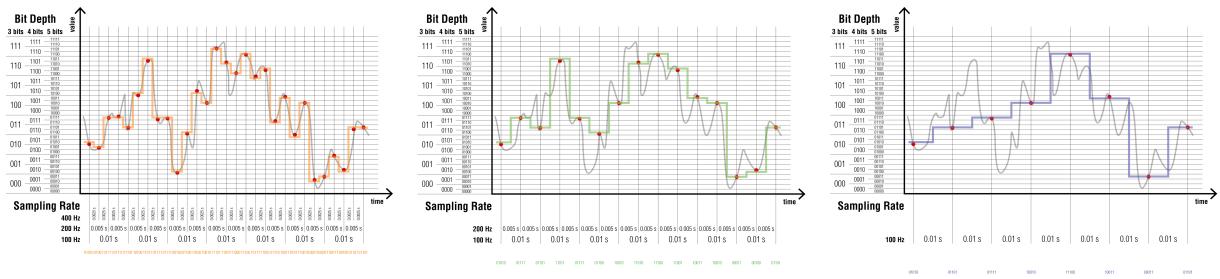
La **distorsion** découle d'un signal dont l'amplitude dépasse les capacités d'encodage du système. Dans ces conditions, un ajustement du gain d'entrée est nécessaire pour rester au plus proche des limites du système, sans les franchir.

La numérisation d'un signal dont l'amplitude serait par trop réduite débouche au contraire sur un encodage qui contient moins d'information, ce qui limite les opérations réalisables numériquement par la suite sans détériorer la qualité du signal.



On notera finalement que l'échantillonnage et la quantification travaillent ensemble pour définir la qualité du signal numérisé. Ces deux paramètres ne sont pas complètement indépendants. Leur choix est réalisé en fonction du résultat escompté et de ce que l'on cherche à réaliser avec le signal numérisé.

Pour l'intensité sonore par exemple, une fréquence d'échantillonnage insuffisante ne peut pas être compensée par une profondeur d'échantillonnage supérieure. La qualité du résultat n'est pas améliorée.



Les dispositifs électroniques dont la fonction est l'échantillonnage et la quantification des signaux sont appelés des convertisseurs analogique-numérique (CAN) ou **analog to digital converter** (ADC), en anglais.

1.4.4 Encodage

L'encodage de l'information numérisée se fait dans des formats de fichiers spécifiques aux applications.

Dans l'absolu, on pourrait imaginer un format universel de stockage de 0 et de 1. En connaissant la profondeur de l'échantillonnage, il serait aisément possible de reconstruire un signal. Toutefois, cela pose plusieurs problèmes.

Cela ne donnerait aucune indication sur l'interprétation qu'il faut faire de ce signal (est-ce un son ? une image ? la variation de la vitesse du vent ?) ou même les bornes de ce signal (entre 0 et $2^n - 1$? entre $-2^{(n-1)}$ et $2^{(n-1)} - 1$?).

De plus, la quantité de mémoire nécessaire pour stocker et pour manipuler les données serait maximisée. Or, il est possible de construire des formats de fichiers qui exploitent les propriétés spécifiques au signal numérisé pour simplifier dans un deuxième temps le résultat de la numérisation avant de l'enregistrer. Cela débouche sur des fichiers qui sacrifient une partie de la qualité du signal numérisé en échange d'un gain sur la taille des fichiers générés. L'usage de la mémoire est ainsi économisé, mais, en contre-partie, un plus grand nombre de calculs est nécessaire pour manipuler les signaux.

C'est ainsi que des fichiers optimisés différents sont disponibles pour stocker des fichiers d'images (JPG), de vidéo (MP4), de son (MP3), ou de toute autre application. La plupart recourent pour cela à des compressions destructives au-cours desquelles une partie de l'information est abandonnée car, dans le contexte particulier, elles ne sont pas jugées indispensables.

Par exemple, la reproduction exacte des nuances de bleu du ciel importe peu pour un film d'action. Pourtant, ces mêmes nuances sont essentielles pour la reproduction d'un tableau de Monnet...

1.4.5 Reconstruction

On appelle **reconstruction** le processus qui transforme un signal numérisé en une variation continue d'une grandeur physique.

Les dispositifs électroniques dont la fonction est la reconstruction des signaux sont appelés des convertisseurs numérique-analogique (CNA) ou **digital to analog converter** (DAC), en anglais. La sortie du convertisseur est généralement une tension électrique proportionnelle à l'intensité du signal.

1.5 Redondance

Pourrait-on construire un véhicule qui ne tombe jamais en panne ?

Si tous les ingénieurs cherchent à y parvenir, l'aviation commerciale est un domaine dans lequel les résultats sont les plus probants. Les [statistiques](#)³⁷ montrent que, depuis une décennie, seuls 2 à 4 accidents mortels par millions de vols sont enregistrés.

Ce résultat est atteint au prix d'une maintenance extrêmement stricte, une formation exigeante et continue du personnel de bord, une analyse méticuleuse de chaque incident, un contrôle permanent du respect des recommandations tant par les constructeurs que par les opérateurs, et enfin un soin particulier apporté à la redondance des systèmes critiques.

La **redondance** est une technique qui consiste à dupliquer les fonctions critiques d'un système pour augmenter sa fiabilité ou ses performances. Évidemment, la redondance des systèmes a un coût : en complexité, en masse et volume, et en maintenance. En effet, paradoxalement, comme chaque composant a une probabilité de panne (fût-elle faible), plus il y a de composants, plus il y a de pannes. Toutefois, moins leurs conséquences sont graves. On parle alors d'un **système robuste ou résilient**, puisqu'il n'est pas mis en péril par la première panne.

La redondance se retrouve à tous les niveaux des systèmes informatiques, qu'ils soient embarqués dans un avion ou non. Ces redondances ont toujours un coût qui se mesure dans ce cas par une **augmentation** de la **quantité de données** et/ou du **temps de traitement** requis. Ils contribuent en échange à la sécurité des données, en augmentant l'**intégrité** (cohérence) et/ou la **disponibilité** de l'information.

1.5.1 Somme de contrôle (checksum)

Pour s'assurer qu'une information est reçue **intégralement** (sans omissions) et **parfaitement** (sans erreurs), un expéditeur pourrait naïvement l'envoyer deux fois, par deux moyens indépendants, dans un SMS et sur une carte postale par exemple. Le destinataire devrait alors renvoyer une confirmation... par deux moyens indépendants ? !

On convient aisément que cette solution serait atrocement dispendieuse. Une étude attentive montre que, de plus, elle ne permet pas de déterminer laquelle des deux copies est la bonne version lorsqu'elles diffèrent, ce qui indique qu'il y a un problème, mais ne propose pas de solution pour le résoudre.

Pourtant, aucun signal n'étant parfait, l'augmentation de la vitesse de transmission débouche nécessairement sur une augmentation des erreurs, notamment des pertes d'information et des mutations.

Les systèmes informatiques, dès lors qu'ils communiquent continuellement et abondamment, sont particulièrement sensibles à ce problème. Lorsqu'il s'agit d'assurer l'intégrité des informations transmises, des solutions plus élégantes que la proposition naïve présentée plus haut ont été élaborées pour effectuer un contrôle de qualité par redondance.

Les **checksums** par exemple sont une brève représentation d'un bloc d'information plus grand, des sortes d'empreintes numériques. Bien qu'elles soient transmises avec le bloc d'information qu'elles représentent, leur petite taille relative ne surcharge pas démesurément la transmission. En choisissant une représentation qui se calcule et se contrôle facilement, ces checksums n'imposent pas non plus un temps de traitement beaucoup plus long pour les créer et les vérifier.

37. <https://www.icao.int/safety/iStars/Pages/Accident-Statistics.aspx>

Idéalement, les checksums de deux blocs d'information sont très différentes même lorsque les différences entre les blocs sont infimes. Cela simplifie en effet la détection des erreurs.

La forme la plus simple est utilisée depuis la nuit des temps informatiques : le **bit de parité**. Il permet de contrôler par redondance une erreur sur la transmission d'un paquet de 7 bits, en utilisant 1 bit supplémentaire.

Dans l'exemple qui suit, on donne la valeur 0 au bit de parité lorsque la somme des bits de la valeur à transmettre est paire, et la valeur 1 lorsque cette somme est impaire. On notera que, de cette façon, la somme des 8 bits est toujours paire.

Le bit de parité est habituellement placé à la position de poids le plus faible, ce qui permet de contrôler directement la valeur transmise.

Valeur à transmettre	Somme des bits	Bit de parité	Valeur transmise
0000000	0	0	00000000
0000001	1	1	00000011
0000010	1	1	00000101
0000011	2	0	00000110
0000100	1	1	00001001
0000101	2	0	00001010
0000110	2	0	00001100
0000111	3	1	00001111
...	
1111101	6	0	11111010
1111110	6	0	11111100
1111111	7	1	11111111

On notera que, pour un coût de taille modeste (un huitième des bits transmis) et un calcul rapide à réaliser (une somme et une comparaison), des erreurs de transmission ponctuelles — celles qui ne portent que sur un nombre de positions impair — sont immédiatement détectables. Cela inclut les erreurs qui porteraient sur le bit de parité lui-même.

Le saviez-vous ?

Les contrôles de bit de parité peuvent être intégrés aux composants électroniques.

La mémoire vive RAM (*Random Access Memory*) de type ECC (*Error-Correcting Code*) s'appuie sur des bits de contrôle pour détecter, voire corriger, les erreurs de stockage qui pourraient affecter les données ou le code des logiciels en cours d'exécution.

Cette fonction supplémentaire explique leur coût plus élevé.

1.5.2 Fonction de hachage

L'exemple de bit de parité permet grossièrement de contrôler une communication caractère par caractère. Une forme plus élaborée du même concept prend la forme du **hachage** de l'information qui peut alors s'appliquer à des quantités d'information beaucoup plus importantes, de type et de longueurs variables.

Prenons par exemple un texte représenté par les valeurs décimales de l'encodage ASCII et construisons une empreinte digitale sur la base de calculs simples :

h	a	c	h	a	g	e
104	97	99	104	97	103	101

La somme de toutes ces valeurs totalise 705 ($= 104 + 97 + 99 + 104 + 97 + 103 + 101$), soit 0x2C1 en hexadécimal.

La somme des produits de chaque valeur par l'index de sa position totalise quant à elle 2821 ($= 1 \times 104 + 2 \times 97 + 3 \times 99 + 4 \times 104 + 5 \times 97 + 6 \times 103 + 7 \times 101$), soit 0xB05 en hexadécimal.

On peut décider de limiter ces deux totaux à leur deux dernières positions hexadécimales, (C1 et 05, respectivement), ce qui permet de construire une empreinte digitale (**digest** ou **hash**) C105 indépendante de la longueur du texte.

Si le texte venait à être modifié, ne serait-ce que très légèrement, l'empreinte numérique ainsi définie serait affectée :

h	a	c	h	a	h	e
104	97	99	104	97	104	101

En effet, la somme des valeurs totalise alors 706 ($= 104 + 97 + 99 + 104 + 97 + 104 + 101$), soit 0x2C2 en hexadécimal, alors que la somme des produits totalise 2827 ($= 1 \times 104 + 2 \times 97 + 3 \times 99 + 4 \times 104 + 5 \times 97 + 6 \times 104 + 7 \times 101$) soit 0xB0B, ce qui donne un hash de C20B au lieu de C105 précédemment, alors qu'un seul bit diffère entre les deux messages.

On notera que les mots ‘hat’ et ‘fer’ débouchent sur la même empreinte (3D86), par exemple.

On notera que la suppression d'une lettre au texte (“hachage” => C105) ne change pas la longueur de cette simple empreinte mais sa valeur (“hachag” => 5C42) et que cette **fonction de hachage** est aussi sensible à la casse (“Hachage” => A1E5).

Le saviez-vous ?

Même si le hachage d'une information est à dessein relativement rapide en soi, des contraintes artificielles provoquant délibérément la multiplication de ces hachages peuvent être imposées lors de l'ajout des blocs dans une **blockchain**. Cela constitue la preuve de travail (*proof-of-Work*, PoW) des cryptomonnaies que l'on nomme communément **minage des cryptomonnaies**.

On notera qu'une empreinte numérique est une simplification de l'information hachée. Il est dès lors envisageable de trouver deux informations, de longueurs possiblement différentes, dont les empreintes sont identiques. En contrepartie, il n'est en principe pas envisageable de reconstruire le texte d'origine sur la base de la seule empreinte.

Toutefois, grâce à leurs propriétés (déterministes et rapides), des **fonctions de hachage** plus complexes (**SHA**, **MD5**...) trouvent des applications dans de nombreux contextes : authenticité (signatures numériques), intégrité (erreurs de transfert, stockage, blockchains...), identification (fichiers, connexions réseau...), authentification (stockage et vérification des mots de passe)...

1.5.3 Disques RAID

Les pannes de disques durs sont très communes et entraînent des pertes de données aux conséquences parfois irrécupérables.

La mise en place de sauvegardes automatiques régulières sur des supports distincts et, de préférence, délocalisés (en soi une forme de redondance sur le stockage de l'information) représente un début de réponse. Toutefois, si le support utilisé pour le stockage n'est lui-même pas résilient, la sécurité de ces sauvegardes n'est pas assurée.

Une solution technique a été proposée dès les années 1980 basée sur la disponibilité de grappes de disques durs relativement bon marché (*Redundant Array of Independent Disks*, RAID). Il est alors possible de créer (entre autres) des disques logiques de taille T (RAID 1), formés d'une grappe de n disques physiques de taille T, sur chacun desquels est stockée une copie complète des données. À chaque écriture, le système maintient automatiquement l'ensemble des n copies, ce qui permet de récupérer l'intégralité de l'information, même si $n - 1$ disques sont endommagés.

Ici encore, en exploitant le principe des bits de parité décrits précédemment, il est par exemple possible de construire une grappe de 3 disques (RAID 5) de taille T capable de stocker $2 \times T$ données. La part de stockage perdue (un disque sur trois) y est utilisée de telle sorte que, lorsqu'un des trois disques est perdu – n'importe lequel des trois ! –, aucune information n'est réellement perdue. Mieux, si le disque défectueux est remplacé, son contenu peut être reconstruit automatiquement et la résilience de la grappe rétablie. En outre, les vitesses d'écriture et de lecture sur ces trois disques en grappe est également accélérée.

Ce type d'infrastructure, malgré son coût plus élevé, est à la base des systèmes critiques qui ne peuvent se permettre de perdre des informations, ce qui pourrait inclure, à terme, les copies de sécurité des postes personnels, quand les données traitées sont sensibles.

1.5.4 Cloud computing

Les systèmes informatiques récents sont distribués à l'échelle d'Internet, tant pour leurs parties matérielles que logicielles. On parle de systèmes *cloud* ou **informatique en nuage**.

On trouve ainsi des systèmes de stockage de fichiers distribués sur plusieurs ordinateurs, voire dans plusieurs fermes de stockage. Cette configuration augmente considérablement la sécurité des données en contribuant à leur **intégrité** et à leur **disponibilité**.

1.6 Conclusion

Trois éléments clés sous entendent notre description du monde : la matière, l'énergie et l'information. Transmettre l'information est sans aucun doute la révolution de ce 20ème siècle achevé et l'enjeu majeur de ce début du 21ème.

La notion d'information est porteuse de sens. Elle s'appuie sur des *données* et un *canal de transmission* ; elle est véhiculée entre un *émetteur* et un *récepteur*.

Dès lors se pose la problématique de la *fidélité* de la représentation, et de l'adaptation au canal de transmission.



FIG. 1.13 – Publicité «iconique» pour la société française Pathé Marconi, du nom d'Émile Pathé (1860-1937), leader du disque phonographique dès le xixe siècle, et de Guglielmo Marconi (1874-1937), pionnier de la radio et prix Nobel de physique en 1909

La mission de la phase d'encodage est d'assurer au mieux cela. Le codage binaire d'entiers ou de caractères s'effectue sans perte, étant effectué entre deux espaces *discrets* ou discontinus : en effet, l'espace des entiers ou des caractères se parcourt par sauts successifs d'une valeur à une autre et trouve une correspondance parfaite avec le codage binaire. Chaque entier distinct va pouvoir trouver sa représentation parfaite comme nous l'avons vu au premier chapitre via son codage binaire et sa représentation décimale associée. De même, chaque caractère trouve son équivalent binaire via la table de représentation ASCII ou mieux UTF présentées au chapitre deux. Représenter un grand nombre entier, ou une palette plus large de caractères ne dépend dès lors que de la capacité de la machine et du nombre de bits de codage : 4, 8, 16, 32, 64 bits...

Le problème est tout autre dès lors qu'on s'intéresse à la représentation machine des images ou du son, abordée aux chapitres trois et quatre.

Le dialogue entre *discret* et *continu* est caractéristique de la science notamment depuis le début du XXème siècle, en mathématiques, physique, chimie, biologie et... en informatique.



FIG. 1.14 – Günther Uecker, Spirale I 1997^{page 44, 38}

Ce que l'oeil perçoit est en réalité une information physique *continue*, tout comme un son. Dès lors, la représentation d'une image dans l'espace *fini* de la machine - l'ordinateur - ne pourra être que partielle ; la puissance de l'ordinateur va définir la précision avec laquelle l'information va pouvoir être transmise, sa *fidélité*.

La chaîne complète émetteur- canal de transmission- récepteur participe de cette fidélité, et tout se joue alors dans l'ajustement entre cette chaîne et la puissance de l'encodage, c'est à dire la capacité de la machine : la complexité de la représentation matricielle pour l'image (pixellisation), la finesse de discréétisation pour le son (échantillonnage).

Représenter l'information à travers ce que comprend la machine (c'est à dire le mode binaire) assure de pouvoir étendre ses possibilités de transmission : aujourd'hui nous pouvons écrire un texte avec un logiciel de traitement de texte, dessiner une image avec un logiciel ou la capturer avec un appareil photo numérique, enregistrer un son, puis transmettre ces «objets numériques» à une multitude de personnes, à l'autre bout du monde, sans altération significative de l'objet initial. Cette information peut être stockée, modifiée, complétée et s'intégrer à nouveau dans ce flux mondial : c'est toute la puissance et la révolution apportée par la technologie numérique, partagée aujourd'hui par plus de 80% des pays de la planète³⁹, et 60% des humains⁴⁰.

Dès lors, on peut s'interroger sur les évolutions de cette technologie dans le sens d'un accroissement de ces performances de représentation, c'est à dire principalement la fidélité, la rapidité et l'accessibilité. L'accroissement des performances des machines liées à l'adoption des transistors, l'arrivée des

38. <https://www.echosciences-grenoble.fr/articles/l-artiste-günther-uecker-avec-ses-tableaux-de-clous-rencontre-le-discret-et-le-continu-de-la-s>

39. <http://www.smartaddict.fr/ces-regions-sans-internet/>

40. <https://www.suricats-consulting.com/fresque-du-numérique/?cn-reloaded=1>

microprocesseurs puis à la miniaturisation suit une croissance exponentielle depuis sa prédiction par Gordon E. Moore en 1965 : c'est la fameuse [loi de Moore](#)⁴¹. Les ordinateurs sont devenus de plus en plus petits, de moins en moins coûteux et de plus en plus rapides et puissants.

Les performances calculatoires ont impacté les mathématiques elles-mêmes, science des nombres.

Les outils de traduction parviennent à des niveaux inimaginables il y a encore vingt ans.

La synthèse vocale atteint aujourd'hui une telle qualité qu'il devient difficile pour l'oreille humaine de distinguer une voix humaine de celle d'un robot.

Pour représenter le réel, si l'on peut imaginer des marges de progression encore possibles en terme de fidélité, c'est sans doute sur l'accessibilité, la diffusion, la consommation des ressources et l'impact sur le climat que se situent les enjeux du développement numérique.

41. https://fr.wikipedia.org/wiki/Loi_de_Moore

Algorithmique I

2.0 Introduction

2.0.1 Quoi ?

Nous avons tous entendu parler des algorithmes dans les médias. Normal, c'est le mot à la mode et que tout le monde utilise sans vraiment le comprendre. Ils sont partout, ils font toutes sortes de choses, même nous manipuler. Pourquoi en parle-t-on de la même manière que des extraterrestres ? Dans ce cours, nous allons tenter de revenir sur terre, parce que les algorithmes ce n'est pas si compliqué que ça. On apprendra à les définir, à les faire marcher et surtout à reconnaître la différence entre un programme et un algorithme, ainsi qu'entre un « bon » et un « mauvais » algorithme.

2.0.2 Pourquoi ?

Les algorithmes existent depuis des millénaires. On doit le nom d'algorithme à Al-Khwârizmî, mathématicien perse né en l'an 780 dont les ouvrages ont contribué à la popularisation des chiffres arabes en Europe, ainsi que la classification de plusieurs algorithmes connus à ce moment. D'ailleurs l'algorithme le plus connu, l'algorithme d'Euclide, date environ de l'an 300 av J.-C. et permet de calculer le plus grand diviseur commun de deux nombres. Si Euclide a bien laissé des traces écrites de cet algorithme, il est vraisemblable qu'il ait puisé cette connaissance auprès de disciples de Pythagore lui-même.

Les algorithmes sont devenus très populaires aujourd'hui grâce à la machine qui a permis de les automatiser. Que ce soit dans votre smartphone, sur un ordinateur ou dans un système embarqué, ils permettent de résoudre une quantité de problèmes, facilement et avec une rapidité impressionnante.

2.0.3 Comment ?

Dans un premier temps nous allons nous intéresser à la notion même d’algorithme : qu’est-ce qui caractérise un algorithme et comment le faire exécuter par une machine ? Nous allons voir que pour un problème donné il existe de nombreuses solutions, mais que toutes ces solutions ne sont pas de *bonnes* solutions, selon le contexte dans lequel on tente de résoudre le problème.

2.0.4 Objectifs d’apprentissage

À la fin de ce chapitre, vous saurez ce qu’est un algorithme et vous serez capable de transcrire des algorithmes en programmes. Vous saurez résoudre des problèmes, en décomposant leur solution en étapes à suivre. Vous verrez également que pour un même problème, on peut avoir plusieurs solutions avec des propriétés, avantages et désavantages différents.

- Se familiariser avec la notion d’algorithme.
- Savoir résoudre des problèmes, en décomposant leur solution en étapes à suivre.
- Savoir que pour un même problème, on peut avoir plusieurs solutions avec différents propriétés, avantages et désavantages.
- Être capable de transcrire un algorithme dans un programme.

Bienvenue dans le monde fascinant des algorithmes.

2.1 Les algorithmes

La première question que l'on va se poser est la suivante : qu'est-ce qu'un *algorithme* ? Est-ce la même chose qu'un programme informatique, ou s'agit-il d'autre chose ?

Un algorithme est en quelque sorte « une recette » que l'on peut suivre pour **résoudre un problème**. De nos jours, il existe énormément de problèmes que les algorithmes nous permettent de résoudre. Il existe des algorithmes pour calculer le trajet le plus rapide entre deux lieux ; d'autres algorithmes ont été imaginés pour détecter les visages dans nos photos ; une demande sur un moteur de recherche est analysée par de nombreux algorithmes afin de nous aider à mieux définir ce que l'on cherche ou afin de nous proposer des contenus publicitaires adaptés.

Ce n'est pas l'algorithme qui est exécuté sur une machine pour nous donner une solution concrète pour tous ces problèmes. *L'algorithme n'est donc pas un programme*. L'algorithme décrit plutôt un « mode d'emploi », qui permet de réfléchir à un problème de manière générale et ensuite de créer un *programme*. C'est le programme qui sera exécuté par un système informatique pour concrètement résoudre le problème. En d'autres mots, l'algorithme décrit l'idée humaine derrière la solution d'un problème, alors que c'est le programme qui permet à une machine de trouver une solution numérique dans des cas précis.

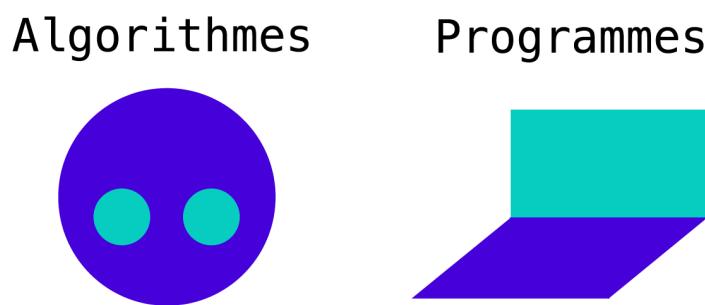


FIG. 2.1 – Différence entre un algorithme et un programme. Un algorithme doit être compréhensible par un humain, alors qu'un programme est écrit de façon à ce qu'il soit compréhensible par une machine.

2.1.1 Résolution d'un problème par étapes

Un mode d'emploi, ou une recette, décrit les **étapes** à suivre pour arriver à une solution. Dans le cas d'une recette de cuisine, la préparation des ingrédients, leur cuisson et leur présentation sont différentes étapes que l'on peut suivre pour réaliser un plat. Prenons un cas précis : *faire une omelette*. Pour chaque étape de la préparation de l'omelette, il faut prévoir une marche à suivre suffisamment détaillée, afin que la personne qui suit la recette arrive au résultat souhaité. Dans le cas de l'omelette, les opérations pourraient être (voir la figure suivante) :

1. Casser les œufs dans un bol.
2. Mélanger les œufs jusqu'à obtenir un mélange homogène.
3. Cuire le mélange d'œufs dans une poêle à température moyenne.
4. Lorsque cuite, glisser l'omelette dans une assiette.

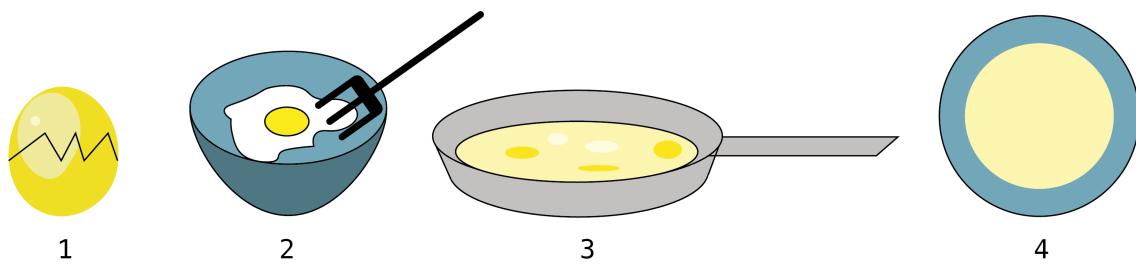


FIG. 2.2 – Un algorithme est un peu comme une recette de cuisine. Cet exemple illustre les opérations à suivre pour la réalisation d'une omelette.

Dans le cas de la recette d'une omelette, nous avons décomposé la marche à suivre en étapes à réaliser dans un certain ordre. Il en est de même pour un algorithme. Pour résoudre un problème, il faut d'abord **décomposer le problème en sous-problèmes** que l'on *sait résoudre*. La solution de chaque sous-problème donne lieu à une étape qu'il faudra exécuter pour arriver à un résultat. Voici les sous-problèmes que certaines étapes ci-dessus permettent de résoudre. Afin d'extraire le contenu comestible de l'œuf, il faut casser les œufs. Pour que l'omelette ait une jolie couleur uniforme, il faut mélanger le jaune et le blanc d'œuf. Cette étape ne serait pas du tout pertinente si le problème que l'on essaie de résoudre est la préparation d'un œuf au plat. *L'algorithme décrit donc toutes les opérations qu'il faut effectuer pour arriver à ce résultat.* Nous allons ainsi définir l'algorithme comme **une suite d'opérations qui permettent de résoudre un problème**.

Le langage utilisé pour écrire un programme doit être extrêmement précis, sans quoi une machine ne pourrait pas le comprendre. Nous avons vu qu'un algorithme n'a pas besoin d'être compris par une machine, mais seulement par les humains. Ainsi, le langage que l'on va utiliser pour exprimer un algorithme sera plus libre que celui utilisé pour coder un programme. Ce langage **peut varier d'une personne à l'autre** et se rapproche dans notre cas de la langue française, comme le montre cet exemple :

```

Liste Nombres      # la variable Nombres contient une liste de nombres
n ← longueur(Nombres) # la variable n contient le nombre d'éléments dans Nombres
i ← 1              # la variable i contient 1 pour commencer
Résultat ← 0        # la variable Résultat contient 0 pour commencer

Répéter Pour i ← 1 à n # i prend la valeur de 1, puis 2, puis 3, jusqu'à n
    Résultat ← Résultat + Nombres[i]
    # Résultat est incrémenté de l'i-ème élément de Nombres
Fin Répéter          # quand i vaut n l'algorithme se termine

Retourner Résultat  # la solution se trouve dans Résultat

```

Dans cet algorithme on mentionne le terme *variable*. Pour rappel, les variables associent un nom (ou un identifiant) à une valeur. Par exemple, ci-dessus on va utiliser une variable que l'on va appeler *i* et qui va stocker pour commencer la valeur 1. Le terme variable prend tout son sens dans l'opération *Répéter*, lorsque *i* contient à tour de rôle des valeurs allant de 1 à *n*, car à ce moment-là la valeur stockée dans *i* **varie**.

Pour mieux vous représenter une variable, imaginez un grand meuble avec des tiroirs (voir la figure suivante). Les variables sont les tiroirs. Chaque tiroir comporte une étiquette, c'est le nom de la variable, et c'est grâce à ce nom que l'on sait quel tiroir ouvrir et quelle valeur utiliser. Le tiroir est petit et ne peut contenir qu'une valeur. Donc `i` peut valoir 1 ou 2, mais pas 1 et 2 à la fois. Par contre `i` pourrait contenir une liste qui contient les valeurs [1, 2]. Cependant, `i` ne peut contenir qu'une seule liste à la fois et pas par exemple deux listes [1, 2] et [3, 4].



FIG. 2.3 – Une variable est un tiroir avec une étiquette. Cela peut être utile de voir la variable comme un tiroir qui permet de stocker une valeur (contenu du tiroir) sous un nom (étiquette du tiroir). Attention, le tiroir est petit et ne peut contenir qu'une chose (valeur) à la fois. Deux tiroirs différents ne peuvent porter la même étiquette.

Lorsque l'on dit que `i ← 1`, ou que `i = 1` en Python, cela veut tout simplement dire que la variable `i` vaut maintenant 1. Cette opération signifie que l'on va prendre le tiroir avec étiquette `i` dans la commode (s'il n'existe pas encore on va noter `i` sur l'étiquette d'un tiroir disponible) et on va mettre la valeur 1 dedans. Ce qui se trouvait dans le tiroir avant la valeur 1 ne s'y trouve plus, on dit que *la valeur précédente est écrasée*. A chaque fois que nous utilisons `i` dans l'algorithme ou dans le code, nous faisons référence à la valeur stockée dans le tiroir.

Exercice 1 – Algorithme mystère

Lisez bien l'algorithme présenté ci-dessus. Quel problème cet algorithme permet-il de résoudre ? Il est plus facile de répondre à cette question, si l'on imagine que la liste `Nombres` contient par exemple les nombres 4, 5 et 6 (correspond à [4, 5, 6] en Python).

Solution 1 – Algorithme mystère

Pour répondre à cette problématique il faut se poser la question suivante : que contient la variable **Résultat** à la fin de l'algorithme ?

Pour commencer, la variable **Résultat** vaut 0. En effet, l'opération **Résultat** \leftarrow 0 initialise **Résultat** à 0. Initialiser une variable veut dire qu'on lui assigne une toute première valeur (une valeur initiale). Dans le cas de **Nombres** qui contiendrait les nombres 4, 5 et 6, après le premier passage dans la boucle **Répéter**, **Résultat** vaut 4. En effet, pour commencer **i** vaut 1 et donc **Nombres[i]** vaut **Nombres[1]**. **Nombres[1]** correspond au premier élément de la liste **Nombres** et vaut 4. L'opération **Résultat** \leftarrow **Résultat** + **Nombres[i]**, additionne alors 0 et 4 (**Résultat** + **Nombres[i]**) et l'opérateur \leftarrow stocke cette valeur 4 dans la variable **Résultat**.

Au deuxième passage dans la boucle, **i** vaut 2. On additionne à nouveau **Résultat**, qui maintenant vaut 4, au 2ème élément de **Nombres**, qui vaut 5. Après ce deuxième passage de la boucle, **Résultat** contient 9 (4 + 5). Finalement, au troisième et dernier passage de la boucle, on additionne cette nouvelle valeur de **Résultat** (ou 9) avec le 3ème élément de **Nombres**, qui vaut 6. Il s'agit du dernier passage de la boucle, parce que lors de ce passage de la boucle **i** atteint la longueur de la liste **Nombres** (ou 3). À la fin de l'algorithme, **Résultat** vaut ainsi 15.

Il est plus facile de se représenter ces valeurs sous forme de tableau :

Passage dans la boucle	i	Nombres[i]	Résultat
avant	1	4	0
1	1	4	4
2	2	5	9
3	3	6	15

Cet algorithme permet de calculer la somme des nombres contenus dans une liste (ici la liste **Nombres**).

Pour comprendre ce que fait l'algorithme ci-dessus, il faut se mettre à la place de la machine. On parle de *simuler* un algorithme, de faire comme si l'algorithme s'exécutait sur une machine. Pour que ce soit plus concret, on peut imaginer des valeurs fictives pour les variables telles que **Nombres**. Dans la vie réelle, **Nombres** pourra contenir tous les nombres possibles, mais cela ne nous aide pas à comprendre. On imagine alors des nombres précis que **Nombres** pourrait contenir, comme par exemple 4, 5 et 6. Lorsqu'on exécute les opérations de l'algorithme l'une après l'autre, avec des valeurs concrètes, on comprend mieux quel effet ces opérations ont sur les valeurs contenues dans les variables. La simulation de l'algorithme nous permet de saisir les **calculs** réalisés par cet algorithme, ici une simple somme.

Exercice 2 – Machine mystère

Quel objet du quotidien (autre que la calculatrice) fait des additions et utilise cet algorithme pour résoudre un problème ?

Il y a-t-il des avantages à automatiser cette tâche, à demander à une machine de le faire à la place d'un humain ?

Il y a-t-il des désavantages à automatiser cette tâche ?

Solution 2 – Machine mystère

Une caisse enregistreuse ! La caisse enregistreuse calcule la somme des prix des produits contenus dans un panier (une liste de courses) et nous donne le prix total à payer. Il s'agit d'un exemple parmi d'autres.

Au niveau des avantages, la caisse enregistreuse fait bien moins d'erreurs qu'un humain, elle ne se fatigue pas, elle ne se plaint pas et elle est bien plus rapide.

Au niveau des désavantages, l'automatisation est en général énergivore (avec une empreinte environnementale significative) et provoque une certaine « obsolescence des humains » en les remplaçant dans leur travail pour un moindre coût financier.

« Chaque étape d'un algorithme doit être définie précisément » (Knuth, 2011). En effet, si on ne décompose pas suffisamment la solution du problème, on peut se retrouver face à une recette inutile, par exemple : prendre des œufs et cuire l'omelette. Cette recette ne nous dit pas vraiment comment procéder pour arriver à faire une omelette...

En lien

Lorsqu'on sauve un fichier dans un ordinateur, il est stocké dans une mémoire. La mémoire d'un ordinateur pourrait être comparée à une grande commode de tiroirs étiquetés. Ainsi, lorsqu'un fichier est stocké en mémoire, la taille du fichier correspond au nombre de tiroirs qu'il occupe. Si c'est un fichier de texte par exemple, on peut imaginer qu'un tiroir contient un caractère simple (un octet). Si c'est une image en couleur, un pixel de cette image occuperait 3 tiroirs (un octet par couleur rouge, vert et bleu).

2.1.2 Les ingrédients d'un algorithme

L'objectif d'un algorithme est de décrire la solution à un problème donné. Concrètement, pour résoudre un problème, l'algorithme va utiliser des **données** qu'il reçoit *en entrée* et va retourner un **résultat** *en sortie*. Le résultat en sortie va être la solution au problème sur la base des calculs effectués sur les données en entrée. Un exemple d'algorithme qui détecte les visages reçoit *en entrée* une image (ce sont les *données*) et

retourne en sortie «oui» ou «non» (c'est le résultat) selon si l'image contient un visage ou pas. Les données en entrée d'un algorithme qui traduit pourraient être le mot à traduire et un dictionnaire. L'algorithme traiterait ces données pour retourner en *sortie* la traduction du mot dans une autre langue.

Entre l'entrée et la sortie, l'algorithme précise les **opérations** qu'il faut exécuter sur les données en entrée. Les opérations que l'on peut demander à un humain sont très différentes de celles que l'on peut demander à une machine. On peut demander à un humain de casser des œufs, mais un ordinateur ne peut pas comprendre et réaliser cette opération. Par contre on peut demander à un ordinateur de se souvenir de milliers de valeurs stockées dans des variables et de comparer les valeurs de toutes ces variables entre elles sans faire d'erreur. Pour résoudre un problème, l'humain cherche une solution sur la base des données à disposition, et la décrit sous la forme d'opérations dans un algorithme. Dans un deuxième temps, ces opérations sont retranscrites en une suite d'instructions élémentaires dans un programme informatique, exécutable par une machine. Dans un troisième temps on vérifie si la solution obtenue est correcte, et si besoin on corrige l'algorithme.

Le dernier ingrédient de l'algorithme, mais tout aussi important, est l'**ordre des opérations**. Dans l'exemple de l'omelette, on ne peut cuire les œufs avant de les avoir cassés, sinon on obtiendrait des œufs durs. De même, l'ordinateur a besoin de recevoir les instructions élémentaires à exécuter dans le bon ordre. Pour résumer, les ingrédients pour concevoir un algorithme sont les suivants :

1. Des données en entrée.
2. Des opérations, dans un ordre précis.
3. Un résultat en sortie.

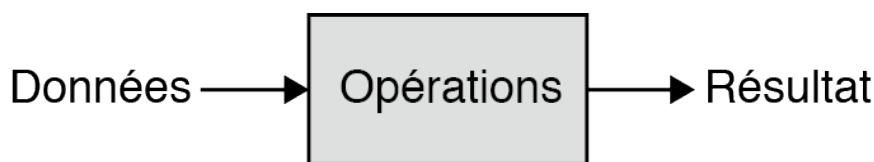


FIG. 2.4 – Schéma des ingrédients d'un algorithme. Un algorithme reçoit des données en entrée, qu'il traite selon des opérations dans un ordre précis, dans le but de produire un résultat en sortie. Ce résultat représente la solution à un problème donné.

Notez que les opérations d'un algorithme doivent être précises et *non ambiguës*. Il doit y avoir une seule interprétation possible de l'algorithme. Une recette de cuisine ne serait pas assez précise pour une machine, par exemple, il faudrait indiquer clairement ce que température moyenne et mélange homogène veulent dire. Les êtres humains peuvent interpréter, deviner et supposer, mais pas les machines (pour l'instant).

Le saviez-vous ? – Jeu d'instructions

Le jeu d'instructions élémentaires dépend du système informatique sur lequel elles s'exécutent. Nous avons vu qu'un algorithme spécifie des opérations à suivre dans un ordre donné afin de résoudre un problème. Ces opérations sont transcris sous la forme d'un programme informatique en instructions élémentaires exécutables par une machine, qui peuvent être très différentes d'une machine à l'autre pour un même algorithme. Ainsi, l'algorithmique permet d'aborder la résolution de problèmes de manière générale, sans se préoccuper des détails d'implémentation sur différents systèmes.

Exercice 3 – Ingrédients de l'algorithme mystère

A quoi correspondent « les ingrédients d'un algorithme » dans l'exemple de la recette de l'omelette ?

Solution 3 – Ingrédients de l'algorithme mystère

Les données en entrée sont les œufs, les opérations sont les étapes 1 à 4 de la recette et finalement le résultat en sortie est l'omelette. On peut considérer le matériel culinaire (bol, fourchette, poêle, spatule) comme du matériel informatique à notre disposition, capable de traiter des données (œufs). En effet, on peut cuire plein d'autres aliments dans une poêle.

Exercice 4 – Échange de deux variables

Écrire un algorithme qui échange les valeurs de deux variables. Par exemple, si la première variable X contient 1 et la deuxième variable Y contient 2, à la fin de l'algorithme X contient 2 et Y contient 1. Pour rappel, une variable peut contenir une seule valeur à la fois.

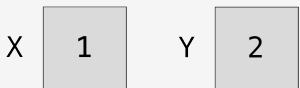
Conseil : cela aide de se mettre à la place de la machine et de représenter le contenu de chaque variable sous la forme d'un tiroir, en la dessinant avec son étiquette et son contenu *après chaque opération de votre algorithme*.

Solution 4 – Échange de deux variables

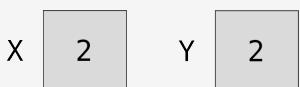
Pour commencer, la variable X contient 1 et la variable Y contient 2. Une solution naïve consisterait à écrire l'algorithme suivant :

```
X ← Y
Y ← X
```

Cet algorithme met la valeur de Y dans X, puis la valeur de X dans Y. Représentons maintenant ces deux variables par des tiroirs étiquetés. Le premier tiroir s'appelle X et contient 1, le deuxième s'appelle Y et contient 2 :



Après la première opération où on met la valeur de Y dans la variable X on se retrouve avec cette situation, où la valeur contenue dans Y écrase la valeur qui était contenue dans X :

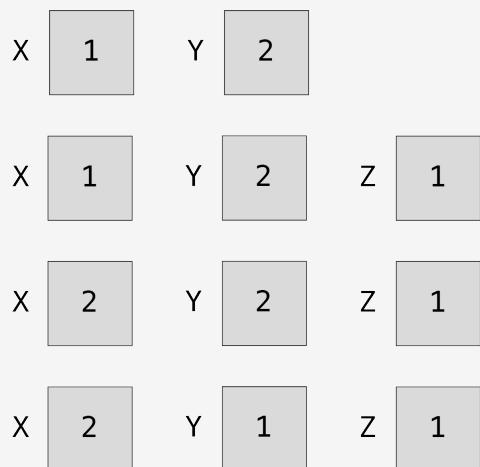


En effet, un tiroir ne peut contenir qu'une seule valeur ! Nous n'avons donc plus accès à la valeur qui était stockée dans la variable X avant d'y mettre celle de Y. Pour remédier à ce problème, il faut penser à utiliser une variable temporaire Z qui permet de se souvenir de la valeur initiale de X.

Un algorithme correct pour échanger les valeurs de deux variables est :

```
Z ← X  
X ← Y  
Y ← Z
```

Si on dessine l'état des variables après chacune de ces opérations dans des tiroirs, voici ce qu'on obtient :



Nous avons donc la confirmation que la solution obtenue résout correctement notre problème d'échange des valeurs de deux variables.

2.1.3 Exercices

Exercice 5 – Forme mystère

L'algorithme suivant contrôle un crayon. Quelle forme dessine-t-il ?

```
Répéter 8 fois:  
    Avance de 2 cm  
    Tourne à droite de 60°
```

Exercice 6 – Nombre minimum

Ecrire un algorithme qui permet de trouver le plus petit nombre d'une liste. Penser à décomposer la solution en différentes étapes.

Appliquer l'algorithme à la liste [3, 6, 2, 8, 1, 9, 7, 5].

L'algorithme trouve-t-il la bonne solution ? Si non, modifier l'algorithme afin qu'il trouve la bonne solution.

Exercice 7 – Le prochain anniversaire

On souhaite déterminer l'élève dont la date d'anniversaire est la plus proche de la date d'aujourd'hui, dans le futur. Ecrire un algorithme (en langage familier) qui permet de trouver cet élève. Penser à décomposer le problème en sous-problèmes.

Comparer la solution trouvée à celle de la personne à côté de vous. Avez-vous procédé de la même manière ? Si non, expliquer vos raisonnements.

Un ordinateur peut-il réaliser les opérations décrites par cet algorithme ?

Exercice 8 – Échange de trois variables

Ecrire un algorithme qui effectue la permutation circulaire des variables X, Y et Z : à la fin de l'algorithme, X contient la valeur de Z, Y la valeur de X et Z la valeur de Y. Pour rappel, une variable ne peut contenir qu'une valeur à la fois.

Conseil : il est très utile de se mettre à la place de la machine et de représenter le contenu de chaque variable sous la forme d'un tiroir, en dessinant le tiroir avec son étiquette et son contenu *après chaque opération de l'algorithme*. Est-ce que votre algorithme donne le résultat attendu ? Si non, modifier l'algorithme pour qu'il résolve le problème correctement.

Exercice 9 – Affectations

Quel est le résultat de la suite des trois affectations suivantes ? On parle d'*affectation* lorsqu'on attribue une valeur à une variable.

```
X ← X + Y
Y ← X - Y
X ← X - Y
```

Vérifier la solution que vous avez trouvée en représentant chaque variable avec une valeur fictive. Suivre les opérations dans l'ordre et dessiner le contenu des variables après chaque étape.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je connais la différence entre un algorithme et un programme.
2. Je sais simuler un algorithme : je représente les valeurs des variables après chaque opération de l'algorithme.
3. Je sais formuler un algorithme : je décompose le problème en sous-problèmes et je décris les opérations qui permettent de résoudre chaque sous-problème.

2.2 Trie, cherche et trouve

Matière à réfléchir – Bibliothèque inutile

Imaginez une bibliothèque scolaire un peu spéciale : les livres n'y sont pas rangés par ordre alphabétique ! Ils sont bien rangés sur des étagères, mais sans aucune logique particulière. Vous entrez dans cette bibliothèque un peu spéciale et vous vous mettez à chercher l'ouvrage *Le Guide du voyageur galactique*.

Pensez-vous pouvoir retrouver ce livre ? Combien de temps cela vous prendra-t-il ?

Y a-t-il des objets chez vous, que vous rangez dans un ordre bien particulier ?

Y a-t-il des objets chez vous, que vous feriez mieux de ranger dans un ordre bien particulier, parce que vous passez beaucoup de temps à les chercher ?

Pour l'instant il faut nous croire sur parole, mais si l'on veut pouvoir trouver une information parmi une avalanche d'informations, il faut que ces informations soient bien rangées. L'exemple de la bibliothèque ci-dessus illustre ce besoin de manière intuitive, mais vous allez pouvoir l'expérimenter concrètement dans le chapitre Algorithmique II.

Saviez-vous que le succès fulgurant de *Google* est surtout dû à sa capacité à bien ranger l'information disponible sur le Web ? Au moment où vous avez besoin d'une information particulière, leurs algorithmes sont capables de la retrouver parce qu'elle est bien rangée. Ce problème qui consiste à ranger les données à un nom, il s'agit du **problème du Tri**. Il est si important qu'il est un des problèmes les plus étudiés en algorithmique.

2.2.1 Algorithmes de tri

Un **algorithme de tri** est un algorithme qui permet de résoudre le problème du tri des données, donc d'organiser les données selon **une relation d'ordre**. Dans la figure ci-dessous, les objets sont organisés soit par la luminosité de leur couleur (ligne du haut), soit par leur taille (lignes du bas), dans un **ordre croissant**.

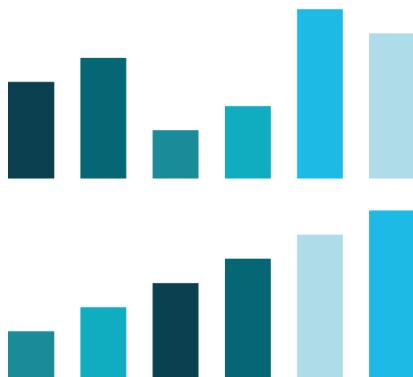


FIG. 2.5 – Problème du tri. Des objets sont triés selon une relation d'ordre, en lien avec une propriété. Sur la ligne du haut, les rectangles sont organisés selon leur couleur (de la plus sombre à la plus claire), alors que sur la ligne du bas, ils sont triés selon leur taille (du plus petit au plus grand).

Exercice 1 – Problème du tri

Trier les rectangles de la ligne du haut de la figure précédente en fonction de leur taille, pour arriver à la disposition de la ligne du bas. Noter toutes les étapes intermédiaires de vos actions et la disposition des rectangles avant d'arriver à la solution finale. Conseil : remplacer les rectangles par un nombre qui représente leur taille.

En lien avec les ingrédients d'un algorithme, déterminer les données en entrée et le résultat en sortie de l'algorithme.

Quels types d'opérations avez-vous effectuées ?

Solution 1 – Problème du tri

Si on remplace les rectangles de la ligne du haut par un nombre qui représente leur taille, on obtient la liste [3, 4, 1, 2, 6, 5]. Le plus important est que l'ordre des nombres conserve l'ordre de la taille des rectangles. Après le tri, si l'algorithme est correct, vous devriez vous retrouver avec la liste [1, 2, 3, 4, 5, 6]. Les opérations et les dispositions intermédiaires exactes dépendent de l'algorithme que vous avez utilisé.

Les données en entrée sont les rectangles sur la ligne du haut : leur taille et l'ordre de leur taille, ici [3, 4, 1, 2, 6, 5]. Le résultat en sortie correspond aux rectangles sur la ligne du bas : l'ordre croissant de leur taille, ici [1, 2, 3, 4, 5, 6].

Les types d'opérations que vous avez effectuées sont des comparaisons de la taille de deux rectangles et des déplacements de rectangles.

Nous allons exposer ici **trois algorithmes de tri simple**, que l'on pourrait utiliser pour trier des objets dans la vie de tous les jours.

2.2.2 Tri par insertion

L'*algorithme du tri par insertion* parcourt la liste d'éléments à trier du deuxième au dernier élément. Pour chaque nouvel élément considéré, il l'insère à l'emplacement correct dans la liste déjà parcourue. A tout moment, la liste d'éléments déjà parcourus (jusqu'à l'élément que l'on considère à un moment donné) est toujours bien triée.

2.2.3 Tri par sélection

L'*algorithme du tri par sélection* commence par rechercher le plus petit élément de la liste et l'échange avec le premier élément de la liste. Il recherche ensuite le plus petit élément de la liste restante (sans le premier plus petit élément). Il sélectionne ainsi le deuxième plus petit élément de la liste et l'échange avec le deuxième élément de la liste. Et ainsi de suite : il recherche le plus petit élément de la liste restante, en excluant les éléments déjà triés, et échange ce plus petit élément avec le premier élément pas encore trié. Il continue de la sorte jusqu'à arriver au dernier élément de la liste.

2.2.4 Tri à bulles

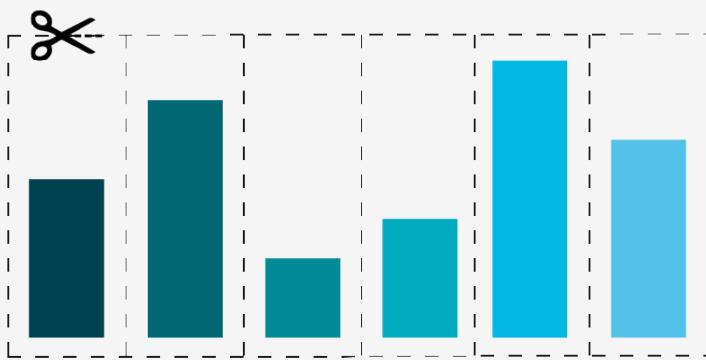
L'**algorithme du tri à bulles** compare les éléments voisins, deux par deux. Il commence par comparer les deux premiers éléments de la liste et les met dans le bon ordre (le plus petit des deux éléments précède le plus grand des deux). Il compare ensuite les deux éléments suivants (le nouveau deuxième et le troisième élément de la liste) et les met dans le bon ordre. Il continue de la sorte jusqu'à la fin de la liste. Après ce premier parcours de la liste, le plus grand élément se retrouve en dernière position de la liste. L'algorithme parcourt à nouveau la liste, en comparant et en déplaçant les éléments voisins deux par deux (en excluant également le dernier élément qui est déjà bien trié). Après le deuxième parcours de la liste, le deuxième plus grand élément se retrouve en avant-dernière position de la liste. L'algorithme parcourt la liste de la sorte, autant de fois qu'elle possède d'éléments, en excluant les éléments bien triés en fin de la liste.

Exercice 2 – Algorithme de tri

Il est fortement recommandé de résoudre cet exercice avant d'avancer dans le chapitre.

Appliquer au moins un des trois algorithmes ci-dessus (tri par insertion, tri par sélection et tri à bulles) pour trier les rectangles de la ligne du haut de la figure **Problème du tri** en fonction de leur taille (le résultat est illustré dans la ligne du bas).

Noter l'ordre des éléments à chaque fois qu'il change. Vous aurez besoin d'une grande feuille de papier. Vous pouvez aussi représenter la taille des rectangles par un nombre, cela permet de gagner de la place. Si cela vous aide, vous pouvez découper les rectangles ci-dessous et les manipuler.



Solution 2 – Algorithme de tri

La solution est donnée dans la suite du chapitre et est illustrée dans la figure **Algorithmes de tri**.

Remarque – Quand on cherche on trouve. Vraiment ?

Vous passez trop de temps à chercher vos affaires ? Pensez à mieux les trier. Le temps perdu à ranger vos affaires sera bien inférieur à celui que vous passerez à les chercher plus tard.

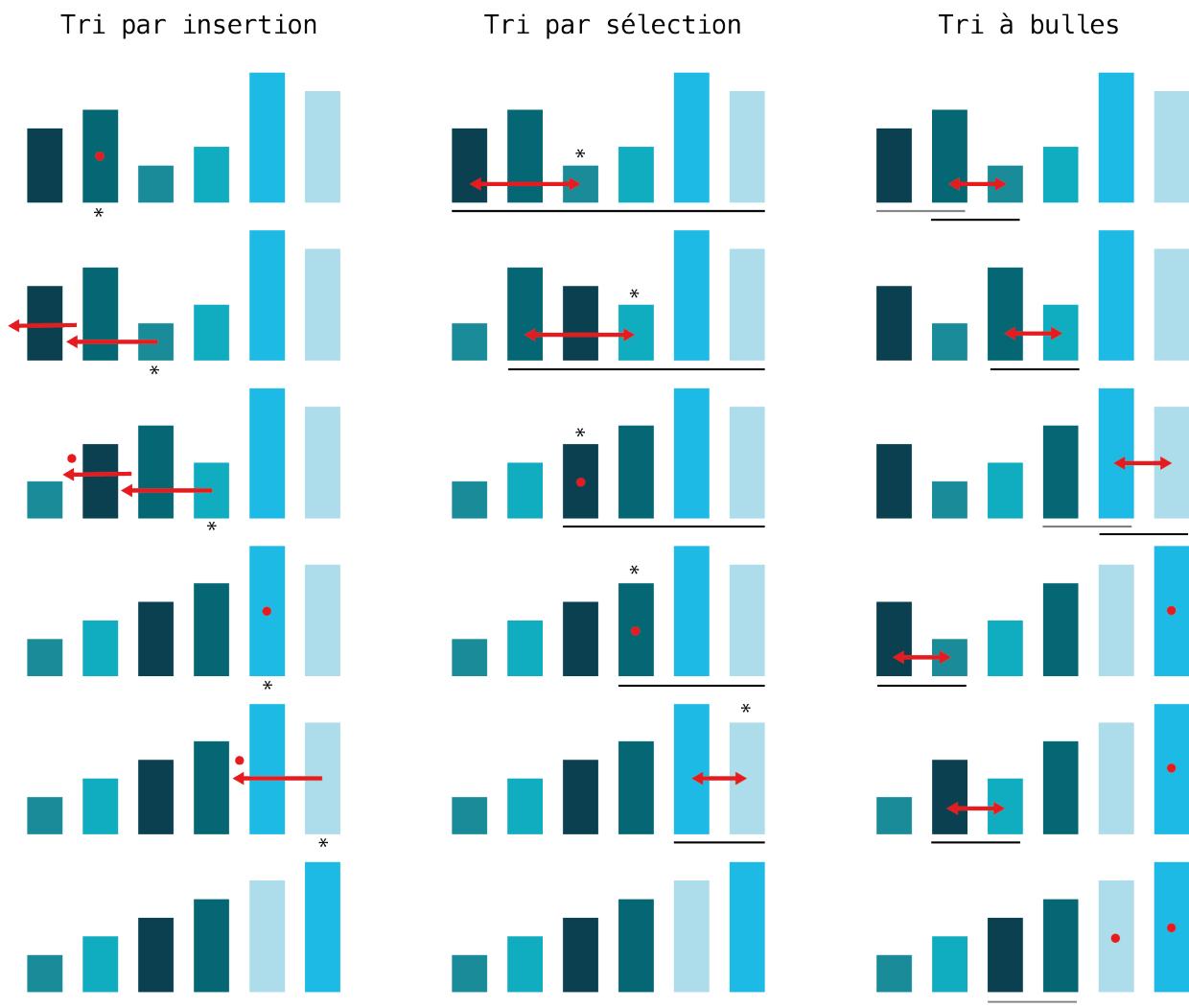


FIG. 2.6 – Algorithmes de tri. Etapes intermédiaires lors de l’application des différents algorithmes de tri. La flèche rouge montre les mouvements des éléments suite à une opération. Si l’élément ne bouge pas, la flèche rouge est remplacée par un point rouge. **A gauche**, le tri par insertion. L’étoile dénote l’élément considéré à un moment donné. **A milieu**, le tri par sélection. L’étoile désigne le plus petit élément de la liste non triée. **A droite**, le tri à bulles. Ici le point rouge signale les éléments triés.

La figure ci-dessus détaille les étapes intermédiaires des trois *algorithmes* de tri vus précédemment. Dans le **tri par insertion** à gauche, on parcourt la liste dans l’ordre, un élément après l’autre (dénoté par une étoile). A chaque étape, on cherche à **insérer** le rectangle considéré à la bonne place dans la liste précédemment triée. La flèche rouge montre la position à laquelle le rectangle sera inséré après comparaison avec l’élément précédent. Si l’élément est déjà bien trié, aucune action n’est requise dans ce cas et la flèche est remplacée par un point rouge. Notez que la liste qui précède le rectangle considéré (celui avec l’étoile) est toujours bien triée.

Dans le **tri par sélection** au milieu, on parcourt la liste pour **sélectionner** son plus petit élément, et on le met à la bonne position. La ligne noire au-dessous des rectangles montre la liste parcourue pour rechercher le plus petit élément. Le plus petit élément de cette liste est désigné par l’étoile. Finalement, la flèche rouge montre les éléments échangés : le premier élément de la liste non triée et le plus petit élément. Ainsi, le

plus petit élément sélectionné (avec étoile) se retrouve à la fin de la liste déjà triée (liste non soulignée). Si l'élément est déjà bien trié et qu'aucune action n'est requise, la flèche bidirectionnelle est remplacée par un point rouge.

Dans **le tri à bulles** à droite, les lignes en dessous des rectangles montrent les éléments voisins qui sont comparés à chaque étape. Lorsque cette ligne est grise, les éléments sont déjà bien ordonnés et aucune action n'est requise. Lorsque la ligne est noire, les éléments ne sont pas dans le bon ordre et doivent être intervertis (flèche rouge). Après un passage complet de la liste, l'élément le plus grand se retrouve en dernière position, il remonte comme une **bulle** (voir la 4e ligne). Le point rouge ici indique les éléments triés. Dans ce cas, la liste est triée après deux parcours complets de la liste.

Notez que même si tous les *algorithmes* arrivent à la même solution finale, ils y arrivent de manière très différente et avec plus ou moins de calculs.

Exercice 3 – Votre algorithme de tri

Rappelez-vous quelle méthode vous avez utilisée pour résoudre le premier exercice. De quel algorithme de tri se rapproche-t-elle le plus ?

Solution 3 – Votre algorithme de tri

Cela dépend de votre solution du premier exercice. Vous avez probablement utilisé la méthode du tri par sélection ou du tri à bulles.

Le saviez-vous ? – Tri stupide

Il existe un algorithme, **Tri de Bogo** (ou *Bogosort*), aussi nommé le *tri lent* ou encore le *tri stupide*. C'est un tri qui génère différentes permutations des éléments de la liste et s'arrête lorsque la configuration obtenue par hasard est triée. A votre avis, combien d'opérations prend cet algorithme en moyenne ?

Exercice 4 – Opérations

Pour chaque algorithme de tri, compter le nombre de **comparaisons** de la taille de deux rectangles, ainsi que le nombre de **déplacements** (le nombre de fois que deux rectangles échangent leur place).

Imaginons que ce qui prend le plus de temps est une **comparaison**. Dans ce cas précis, quel algorithme de tri parmi les trois algorithmes présentés est le plus lent ?

Imaginons que ce qui prend le plus de temps est un **déplacement**. Dans ce cas précis, quel algorithme de tri est le plus lent ? Quel algorithme est le plus rapide ?

Solution 4 – Opérations

Le décompte des opérations effectuées, en se référant à la figure **Algorithmes de tri** est comme suit :

Tri par insertion : 9 comparaisons deux par deux (flèches et points rouges) et 5 déplacements deux par deux (flèches rouges). Notez que pour insérer un élément en première position, il faut tout d'abord l'échanger avec l'élément juste devant, puis avec l'élément avant, et ainsi de suite jusqu'à arriver à la première position.

Tri par sélection : 15 comparaisons deux par deux (lignes en dessous) et 3 déplacements deux par deux (flèches rouges).

Tri à bulles : 9 comparaisons deux par deux (lignes en dessous) et 5 déplacements deux par deux (flèches rouges).

Si ce qui prend beaucoup de temps est la comparaison de la taille de deux rectangles, il ne faudrait pas utiliser le tri par sélection, car il comporte le plus grand nombre de comparaisons et il serait le plus lent. Si c'est le déplacement de deux rectangles qui coûte beaucoup de temps, cette fois-ci le tri par sélection serait le plus rapide (avec 3 rectangles qui échangent leur position). Donc, selon l'implémentation sur la machine, le tri par sélection serait le plus lent ou le plus rapide des trois algorithmes.

Ces résultats sont valables pour cette configuration en particulier. Si on trie un autre tableau, la performance des trois algorithmes pourrait changer. Le choix du meilleur algorithme dépend donc de l'implémentation et de la situation initiale. Notez finalement qu'il existe des algorithmes de tri bien plus rapides que les trois algorithmes considérés ici.

2.2.5 Comparaison d'algorithmes

Toutes les recettes de cuisine ne se valent pas, de la même manière, un *algorithme* peut aussi être **plus approprié** qu'un autre algorithme pour résoudre le même problème. Il existe des dizaines d'*algorithmes* qui trient avec des approches différentes (nous en verrons encore quelques-uns). Certains algorithmes sont plus rapides, d'autres plus économies en mémoire ou encore plus simples à coder. Ainsi, selon la situation, il faut choisir le « bon » *algorithme*.

La qualité d'un *algorithme* dépend de la propriété que l'on souhaite optimiser (maximiser ou minimiser). Cela pourrait être de maximiser la **vitesse d'exécution** (mesurée par le nombre d'*instructions* élémentaires exécutées), de minimiser la place occupée en **mémoire**, de minimiser la **consommation d'énergie** ou de maximiser la **précision de la solution**. L'*algorithme* utilisé devrait être choisi en fonction de ce qui est important.

La vitesse d'un algorithme dépend également des données en entrée. Selon la configuration initiale des données en *entrée* (correspond à la ligne du haut de la figure **Algorithmes de tri**), un *algorithme* « rapide » peut devenir « lent » et *vice versa*. Il faut savoir que les *algorithmes* vus jusqu'ici sont tous des *algorithmes* lents (nous verrons un *algorithme* de tri rapide ultérieurement).

Le saviez-vous ? – Tri trop lent

Pour trier 1 million d'éléments, selon l'algorithme choisi, cela peut prendre de 20 millions à 1 billion d'opérations. Si chaque opération prenait 1 microseconde ($10^{-6}s$) à s'exécuter, il faudrait 20 secondes pour trier 1 million d'éléments si l'algorithme est efficace. Par contre, pour un des algorithmes ci-dessus, cela pourrait prendre 11 jours !

Pour aller plus loin

Imaginer que les quatre éléments d'une liste sont triés dans le sens inverse de ce que l'on souhaite (ils sont placés du plus grand au plus petit). Trier la liste selon les trois algorithmes de tri vus précédemment : le tri par insertion, le tri par sélection et le tri à bulles.

Dans cette configuration précise, quel algorithme est le plus rapide (présente le moins d'étapes intermédiaires) ?

Et quel algorithme est le plus lent ?

2.2.6 Exercices**Exercice 5 – L'algorithme de votre journée**

Réfléchir à votre journée : y a-t-il des actions qui se retrouvent chaque jour ouvrable ? Arrivez-vous à esquisser un algorithme que vous suivez sans que vous en ayez conscience ?

Exercice 6 – Trois algorithmes de tri

Trier la liste [2, 5, 3, 4, 7, 1, 6] en utilisant les trois algorithmes de tri vus dans le cours. Représenter l'état de la liste après chaque étape.

Exercice 7 – Vérificateur de tri

Ecrire un algorithme qui vérifie si une liste est triée.

Que prend l'algorithme en entrée et que retourne-t-il en sortie ?

Demander ensuite à un autre élève de suivre les opérations décrites par votre algorithme. Est-ce que votre algorithme est correct ?

Comparer vos algorithmes. Sont-ils différents ?

Exercice 8 – Mondrian

Analyser les œuvres cubistes de Piet Mondrian. Trouver un algorithme qui permet de créer une œuvre qui pourrait être attribuée à Mondrian.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais qu'il existe plusieurs manières différentes de résoudre un problème.
2. Je sais qu'il faut choisir le meilleur algorithme en fonction de critères objectifs : vitesse de l'algorithme, qualité de la solution, espace utilisé en mémoire ou encore consommation d'énergie.
3. Je sais appliquer les trois algorithmes de tri vus dans le cours.

2.3 Des algorithmes aux programmes

Matière à réfléchir – Lieu mystère

Pensez à un lieu connu, qui se trouve à proximité. Ecrivez les étapes à suivre pour s'y rendre, sans mentionner le lieu. Vous ne pouvez utiliser que les instructions : **avancer**, **tourner à gauche** et **tourner à droite**.

Demandez à vos camarades de classe de suivre ces instructions. Sont-ils arrivés à deviner à quel lieu ils se sont rendus ?

Si non : essayez de comprendre à quel moment ils se sont perdus. Adaptez votre algorithme en fonction.

Si oui : reformulez vos instructions en utilisant les mots-clés **si (if)**, **sinon (else)** ou **tant que (while)**.

[Optionnel] Imaginez que votre camarade peut uniquement **avancer de 1m tout droit** et **tourner de 15 degrés**. Reformulez votre solution en utilisant en plus le mot clé **répéter (for)**.

Une fois que l'on a déterminé le meilleur *algorithme* à utiliser, pour l'automatiser, il faut le retranscrire dans un *programme* qu'une machine peut comprendre. Nous allons détailler ce processus pour l'algorithme du **tri par sélection**.

Cet algorithme consiste à parcourir la liste à trier plusieurs fois. A chaque *itération*, on sélectionne le plus petit élément et on l'échange avec le premier élément de la liste non triée. Comment pourrait-on traduire ceci en Python ? Comment représenter les rectangles dans un langage de programmation ?

Tout d'abord, il faut représenter la taille des rectangles par des nombres. On peut par exemple représenter l'ordre des rectangles de la première ligne de la figure **Trier** en fonction de leur taille, dans une liste nommée `rect` :

```
rect = [3, 4, 1, 2, 6, 5]
```

On doit ensuite **parcourir la liste** pour trouver le plus petit élément de la liste, qui correspond au rectangle le plus court. Nous allons commencer par **déclarer une variable**, nommée `indice_min`, qui va se souvenir de la position du plus petit élément de la liste (équivalent à l'indice de l'élément à l'intérieur de la liste). Pour commencer, nous supposons que le plus petit élément de la liste est le premier élément, et nous initialisons la variable nommée `indice_min` à `0`.

```
# initialise une variable qui va se souvenir du plus petit rectangle de la liste
indice_min = 0
```

Nous allons ensuite parcourir la liste à partir du deuxième élément. Pour chaque nouvel élément, nous allons tester s'il est plus petit ou plus grand que le plus petit élément connu jusqu'alors. Si le nouvel élément est plus petit que l'élément désigné par `indice_min`, c'est l'indice du nouvel élément qui sera stocké dans `indice_min` à la place de l'ancien :

```
# for permet de parcourir la liste rect
for i in range(1,len(rect)): # len(rect) donne la longueur de la liste rect

    # identifie l'indice du plus petit élément de la liste
    if rect[i] < rect[indice_min] :
        indice_min = i
```

Pour faire plus simple, nous pouvons également utiliser la *fonction Python* `min()` qui retourne directement le plus petit élément d'une liste. Nous avons aussi besoin de la fonction `index()` afin d'accéder à la position (ou l'indice) du plus petit élément.

```
# identifie l'indice du plus petit élément de la liste
indice_min = rect.index(min(rect))
```

Grâce à ces fonctions Python préexistantes, nous avons remplacé les 3 lignes du code au-dessus par une seule ligne de code. Après cette opération, `indice_min` contient l'indice du plus petit élément de la liste. On doit à ce stade, échanger cet élément et le premier élément. Comme nous avons pu le voir avant, pour échanger les valeurs de deux variables, nous avons besoin d'une **variable temporaire**. En effet, si on met la valeur du plus petit élément directement à la position 0, nous perdons la valeur contenue à la position 0 à ce moment-là. Il faut donc la stocker temporairement dans une autre variable :

```
# échange le plus petit élément avec le premier élément
rect_temp = rect[0]
rect[0] = rect[indice_min]
rect[indice_min] = rect_temp
```

Là encore, Python permet d'écrire ces trois lignes de manière beaucoup plus compacte. En affectant les deux variables simultanément, c'est Python qui se charge de créer la variable temporaire :

```
# échange le plus petit élément avec le premier élément
rect[0], rect[indice_min] = rect[indice_min], rect[0]
```

On doit ensuite refaire exactement les mêmes opérations (parcourir à nouveau la liste pour trouver le plus petit élément et échanger sa position), mais en excluant le premier élément de la liste qui est maintenant bien trié. Donc on va rechercher le plus petit élément de la liste restante, et l'échanger cette fois-ci avec le deuxième élément de la liste (attention, il s'agit de la position 1 et non 2 en Python). On adapte le code précédent :

```
# trouve le plus petit rectangle de la liste rect[1:] (à partir du 2e élément)
indice_min = rect.index(min(rect[1:]))

# échange le plus petit élément avec le deuxième élément
rect[1], rect[indice_min] = rect[indice_min], rect[1]
```

La suite de l'algorithme consiste à nouveau à rechercher le plus petit élément de la liste restante (en excluant cette fois-ci le premier et deuxième élément, qui sont bien triés) et l'échanger avec le troisième élément (premier élément non trié). À nouveau on peut reprendre le même code, mais on incrémente tous les indices de 1 :

```
# trouve le rectangle le plus petit de la liste rect[2:] (à partir du 3e élément)
indice_min = rect.index(min(rect[2:]))  
  
# échange le plus petit élément avec le troisième élément
rect[2], rect[indice_min] = rect[indice_min], rect[2]
```

On détecte un motif qui se répète. On fait toujours les mêmes opérations, mais en commençant à une position différente. On peut réécrire le même code autant de fois que d'éléments dans la liste, mais ce n'est pas optimal si la liste est longue et si on veut pouvoir réutiliser ce code pour une liste de longueur différente. Il vaut mieux remplacer l'indice qui change par une variable que l'on *incrémente* (augmente). Notez que ce code est répété `len(rect)-1` fois et pas autant de fois qu'il y a d'éléments de la liste, car on doit pouvoir comparer et échanger deux éléments.

```
# pour tous les éléments de la liste non triée
for j in range(0,len(rect)-1):  
  
    # trouve le rectangle le plus petit de la liste rect[j:] (à partir de l'élément j)
    indice_min = rect.index(min(rect[j:]))  
  
    # échange le plus petit élément avec le j-ième élément
    rect[j], rect[indice_min] = rect[indice_min], rect[j]
```

Le principal avantage de cette **factorisation** (réécriture) est que maintenant notre code fonctionne pour toutes les longueurs de listes. Nous n'avons plus besoin de savoir à l'avance combien d'éléments sont contenus dans la liste (combien de fois répéter les opérations). Au lieu de répéter le code un nombre prédéterminé de fois, le code s'exécute autant de fois qu'il y a d'éléments dans la liste (moins 1, car on compare toujours 2 éléments).

L'étape suivante consiste à encapsuler tout le code dans une **fonction** qui reçoit la liste comme *paramètre*, afin de le rendre utilisable par d'autres programmes sans avoir à copier-coller le code. Cela permet aussi en cas d'erreur de facilement corriger la fonction, plutôt que de corriger le code partout il a été copié-collé. Pour que la fonction soit utilisable, il ne faut pas oublier de rajouter le `return` qui retourne le résultat.

```
# Tri par sélection
def tri_selection(rect):  
  
    # pour tous les rectangles de la liste non triée
    for j in range(0,len(rect)-1):  
  
        # trouve le rectangle le plus petit de la liste rect[j:]
        indice_min = rect.index(min(rect[j:]))  
  
        # échange le plus petit élément et le j-ième élément
```

(suite sur la page suivante)

(suite de la page précédente)

```
rect[j], rect[indice_min] = rect[indice_min], rect[j]
return(rect)
```

Finalement le terme `rect` n'est pas assez général, car le tri par sélection peut être utilisé pour trier toutes sortes d'éléments et pas seulement des rectangles. Ainsi on peut renommer la *variable* `rect` par le terme plus général `liste`, partout où `rect` apparaît dans le code ci-dessus :

```
# Tri par sélection
def tri_selection(liste) :

    # pour tous les éléments de la liste non triée
    for j in range(0,len(liste)-1):

        # trouve l'élément le plus petit de liste[j:]
        indice_min = liste.index(min(liste[j:])) 

        # échange le plus petit élément et le j-ième élément
        liste[j], liste[indice_min] = liste[indice_min], liste[j]

    return(liste)
```

Pour trier la liste `rect` définie au tout début, il suffit d'appeler la fonction `tri_selection` avec la liste `rect` en *argument*. La fonction `print()` permet d'afficher la liste triée :

```
# trier la liste de rectangles par sélection
rect = [3,4,1,2,6,5]
print(tri_selection(rect))
```

En traduisant les étapes intermédiaires du tri par sélection en des lignes de code, nous avons automatisé l'algorithme. Nous l'avons transcrit en un programme informatique qui peut être exécuté sur une machine.

2.3.1 Exercices

Exercice 1 – Jeu de la devinette

Ecrire le programme suivant : le programme pense à un nombre au hasard. Lorsque vous lui proposez un nombre, il vous dit si « c'est plus » ou si « c'est moins » jusqu'à ce que vous trouvez le bon nombre. Conseil : utiliser le module Python `random`.

Y a-t-il une stratégie gagnante ?

Exercice 2 – Plus petit nombre

Transcrire l'algorithme de l'exercice qui permet de déterminer le plus petit nombre d'une liste, en un programme Python.

Exercice 3 – Programmes de tri

Implémenter le tri à bulles et/ou le tri par insertion vus au cours.

Créer une liste qui contient les valeurs de 1 à n dans un ordre aléatoire, où n prend la valeur 10, par exemple. Vous pouvez utiliser la fonction `shuffle()` du module random.

Pour aller plus loin.

A l'aide du module `time` et de sa fonction `time()`, chronométrier le temps pour le tri d'une liste de 100, 500, 1000, 10000, 20000, 30000, 40000 puis 50000 nombres.

Noter les temps obtenus et les afficher sous forme de courbe dans un tableau. Ce graphique permet de visualiser le temps d'exécution du tri en fonction de la taille de la liste. Que constatez-vous ?

Sur la base de ces mesures, pouvez-vous estimer le temps que prendrait le tri de 100000 éléments ?

Lancer le programme avec 100000 éléments et comparer le temps obtenu avec votre estimation.

Exercice 4 – Tri de Bogo

Coder l'algorithme du tri de Bogo en Python (voir chapitre précédent : Le saviez-vous ?).

Relancer l'algorithme plusieurs fois, en notant le nombre d'itérations nécessaires pour qu'il termine.

A partir de quelle taille de liste cet algorithme est-il inutilisable ?

Exercice 5 – Fibonacci

Ecrire un algorithme qui calcule la suite des nombres de Fibonacci.

Traduire l'algorithme en une fonction Python.

Comparer avec les solutions trouvées par vos camarades de classe.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais lire et appliquer un algorithme, c'est-à-dire que je peux déduire le résultat que me donnera un algorithme à partir d'un jeu de données particulier.
2. Je sais retranscrire un algorithme en un programme, je sais traduire les opérations d'un algorithme en instructions élémentaires if, else, while et for.

2.4 Conclusion

2.4.1 Les algorithmes et nous

Sans le développement de l’algorithmique, de nombreux problèmes n’auraient pas pu être résolus par les ordinateurs dans un temps raisonnable.

L’étude des algorithmes a un effet bénéfique sur notre manière de réfléchir et de résoudre des problèmes dans notre vie quotidienne. L’étude de l’algorithmique permet de structurer notre pensée et de prendre des décisions fondées sur une réflexion argumentée.

Les algorithmes sont omniprésents. « Ils » décident de ce que nous voyons sur les réseaux sociaux, ils influencent nos choix quand nous cherchons une personne qui nous correspond, ils nous suggèrent des livres à lire et des films à regarder, corrigent nos textes, les traduisent ou encore embellissent nos photos en un clic. Ils font la pluie et le beau temps en bourse, décident si un prévenu doit être emprisonné, rédigent des articles de journaux, conduisent des voitures autonomes. Cette liste s’allonge chaque jour...

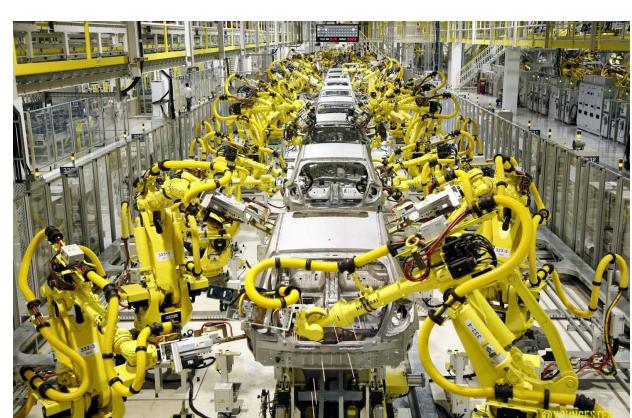
Comprendre le fonctionnement de base des algorithmes permet de mieux appréhender ce qu’il se passe dans toutes ces situations. Nous y reviendrons plus en détail dans la deuxième partie du cours d’algorithmique.

2.4.2 Focus sur l’automatisation

Grâce aux algorithmes, la machine a pu remplacer l’homme dans de nombreuses tâches (comparez les deux images ci-dessous), en allant des robots constructeurs d’automobiles aux pilotes automatiques dans les avions, ou encore aux logiciels de trading. L’automatisation permet aux employés de se concentrer sur des tâches plus valorisantes et permet aux entreprises de réaliser des économies.



Usine du début du siècle dernier. Les machines dans cette usine de métallurgie à Vallorbe dans le canton de Vaud sont au service des ouvriers. Source : <https://wikivaud.ch/metallurgie-vaudoise/>



Usine du début de ce siècle. Les machines dans cette usine de montage Mitsubishi en Chine ont remplacé les ouvriers. Source : <https://www.lemonde.fr/blog/fredericjoignot/2015>

Selon un rapport publié en 2017 par DELL et le Think Tank californien « L’institut du futur », cité par la chasseuse de têtes Isabelle Rouhan dans son livre « Les métiers du futur », **85% des métiers** qui seront exercés en 2030 par les écoliers d’aujourd’hui n’ont pas encore été inventés.

À retenir

Un algorithme est **une suite d'instructions dans un ordre bien précis** qui permet de résoudre un problème. L'algorithme va produire un résultat en sortie, en fonction de données reçues en entrée.

Pour arriver à résoudre un problème, il faut commencer par le **décomposer en sous-problèmes**.

Afin de pouvoir rechercher de manière efficace, **les données doivent impérativement être triées** en utilisant un algorithme de tri.

Il existe de multiples manières de résoudre un problème, par exemple différents algorithmes de tris. Toutes ces manières ne se valent pas. Il faut **choisir l'algorithme en fonction de ce qui doit être optimisé** : le temps de résolution, l'espace de stockage, la consommation d'énergie, la précision de la solution, etc.

L'algorithme n'est pas un programme. Pour être exécuté sur un système informatique, l'algorithme doit être transcrit en un programme, pour résoudre le problème concrètement et de manière automatisée.

Remarque – Souhaiteriez-vous devenir neuro-manageur.euse ou éducateur.rice de robots ?

Extrait. *Intelligence artificielle. Enquête sur ces technologies qui changent nos vies. Les algorithmes vont-ils tuer l'emploi ?,* éd. Flammarion, 2008, pp. 72-73.

Résultat, ce ne sont pas seulement les cols-bleus qui sont touchés, mais également les cols blancs, cadres de professions intermédiaires et même supérieures, comptables, traducteurs ou encore traders, etc. Pour l'essayiste Hakim El Karoui, la robotisation pourrait être à ces derniers ce que la mondialisation a été aux premiers. Goldman Sachs, la star des banques d'affaires new-yorkaises, a provoqué une onde de choc dans la profession lorsqu'un de ses responsables a déclaré début 2017 que son *desk de trading actions*, qui employait 600 traders à son pic en 2000, n'en comptait plus que deux ! « Ces 600 traders, ils occupaient beaucoup d'espace », a-t-il lâché, sûr de son effet, lors d'un colloque à l'université de Harvard. Des bataillons de traders disparus pour cause de bascule vers le trading électronique à haute fréquence, qui représente aujourd'hui 99% des ordres d'achat et de vente chez Goldman Sachs, 75% chez les concurrents et 45% toutes classes d'actifs confondus dans le secteur bancaire. Ce géant de la finance les a avantageusement remplacés par... 200 ingénieurs. Ce sont eux qui pilotent désormais, pour des salaires jusqu'à cinq fois moins importants, des algorithmes programmés pour gagner des sommes certes infinitésimales, mais sur des millions d'opérations quotidiennes en limitant au maximum le risque. Au suivant ?

CHAPITRE 3

Architecture des ordinateurs

3.0 Introduction

Attention

Ce document doit être retravaillé ...

Dans ce chapitre, nous aborderons la question de l'architecture des ordinateurs, c'est-à-dire les multiples couches physiques qui rendent possibles des opérations numériques aussi complexes que celles qu'effectuent à chaque instant nos smartphones.

Comme vous avez pu le voir dans le chapitre lié à la représentation de l'information, tout ce qui apparaît sur votre écran est représenté par l'ordinateur par suite de 0 et de 1. Pour comprendre comment ces 0 et 1 sont traités par l'ordinateur, il faut avoir en tête que les ordinateurs sont construits à partir d'une couche et de multiples niveaux, comme un mille-feuille, dont chacun possède ses propres règles.

Dans ce chapitre, nous nous concentrerons sur les couches de bas niveau, et tenterons de remonter progressivement jusqu'aux couches logicielles.

3.0.1 De quoi sont faits les nombres binaires ?

Les ordinateurs ne comprennent que les nombres binaires. La lettre «A», par exemple, est pour ces derniers une suite de 0 et de 1. Même chose pour une image, une vidéo, une chanson et ainsi de suite. Mais alors comment ces 0 et ces 1 sont-ils stockés et manipulés physiquement par les ordinateurs ? De quelle matière sont-ils faits ? Un indice : que mettez-vous dans votre smartphone pour le faire fonctionner : de l'essence ? Du gaz ? De l'énergie solaire ?

De l'électricité !

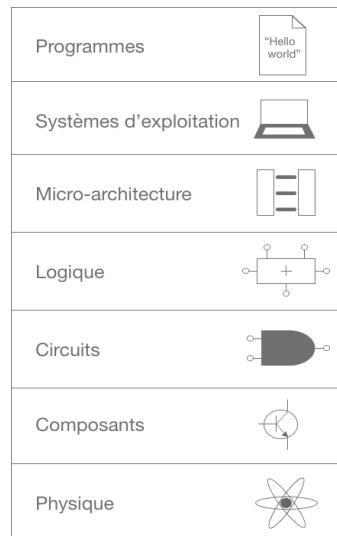


FIG. 3.1 – Les différents niveaux d'abstraction de l'informatique, en partant des électrons, jusqu'aux «programmes»

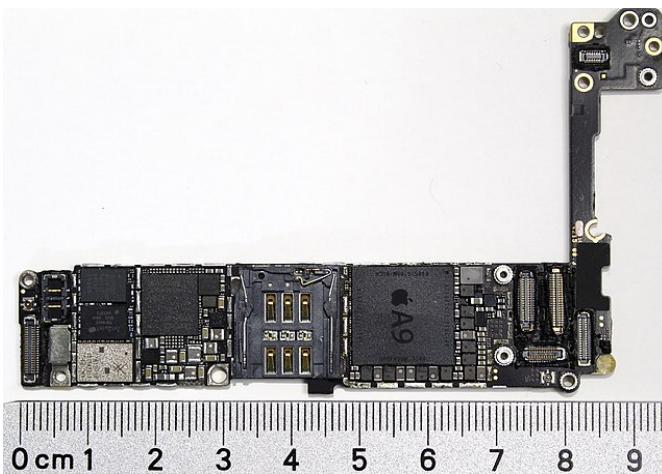


FIG. 3.2 – Vos photos, vos vidéos, vos messages, tout ce que vous consultez sur votre téléphone portable, sont traitées par un processeur similaire au modèle A9 de Apple, commercialisé dans les iPhone SE.



FIG. 3.3 – Vos likes, vos partages, vos vidéos transmises via des applications telles que WhatsApp, Instagram, TikTok, Snapchat, YouTube, sont stockées dans des centres de données aux quatre coins de la planète.

3.0.2 Électricité et nombres binaires

Les nombres binaires, au niveau le plus élémentaire, sont matérialisés par des courants électriques, qui traversent les circuits des ordinateurs. Mais pourquoi avoir choisi des 0 et des 1 comme alphabet ? Quel rapport avec l'électricité ?

En informatique, si nous avons choisi d'utiliser un code binaire, ça n'est pas par hasard. Ce sont les deux signaux les plus élémentaires que l'on puisse transmettre avec l'électricité. Soit le courant passe, soit il ne passe pas. Ouvert ou fermé ; allumé ou éteint ; 1 ou 0.

Le saviez-vous ?

On aurait pu choisir un code possédant plus de deux signaux différents. Par exemple, avec trois signaux, on pourrait coder trois valeurs avec un courant faible, un courant moyen, un courant fort, ou encore mieux : une tension négative, une tension nulle et une tension positive. On appelle cette dernière proposition le ternaire balancé. En fait, cela s'est déjà fait : les soviétiques ont développé en 1958 un ordinateur nommé *Setun*⁴² basé sur ce principe, réputé très fiable et extrêmement performant dans le développement d'applications dans certains domaines. Mais ce projet, pour des raisons politiques, n'a pas reçu le soutien qu'il aurait mérité. D'autre part, il est plus simple de concevoir des circuits électroniques qui ne doivent traiter que deux valeurs.

42. <https://en.wikipedia.org/wiki/Setun>

La grande idée derrière la conception des ordinateurs et de leurs circuits électroniques repose sur l'utilisation de sortes d'«interrupteurs automatiques». Ce composant fonctionne donc comme un interrupteur (en laissant ou non passer le courant sur un fil donné), mais de façon automatique : ce n'est pas un humain qui doit venir commuter l'interrupteur, mais l'interrupteur commute automatiquement en fonction de si oui ou non du courant passe sur un *autre* fil du système. Historiquement, on a réalisé que

si l'on disposait d'un tel composant, on pouvait en assembler plusieurs (en fait, plusieurs milliers) et ainsi construire des systèmes à même de manipuler des données représentées par des 0 et des 1. Nous allons voir comment dans les prochaines sections.

Dans les premiers ordinateurs entre les années 1950 et 1960, ce sont les **tubes à vide**⁴³ qui ont rempli cette fonction. Mais les tubes à vides étaient gros, consommaient beaucoup d'électricité, et avaient une durée de vie limitée : il fallait souvent les changer, un peu comme de vieilles ampoules à incandescence. En utilisant des tubes à vide, on pouvait certes construire des ordinateurs, mais certainement pas ceux que l'on connaît aujourd'hui.



FIG. 3.4 – Différents modèles de tubes à vide. Photographie de Stefan Riepl, 2008, CC BY-SA.

Il a fallu attendre une invention majeure pour permettre aux ordinateurs de se miniaturiser, de consommer moins, d'être plus efficaces et fiables, pour finalement arriver à ce qu'on a appelé dès la fin des années 1970 des **PC** : des *personal computers*, des ordinateurs personnels. L'invention qui a permis cette évolution est le **transistor**.

3.0.3 Le transistor

Le **transistor** est aujourd'hui la brique de base de construction des systèmes informatiques. Il a été développé dans les années 1940 dans les **laboratoires Bell**⁴⁴, aux Etats-Unis. Ce n'est que vers la fin des années 1950 que l'on commence à construire des ordinateurs commerciaux qui utilisent des transistors plutôt que des tubes à vide. Le transistor est à l'origine d'une révolution dans la taille, la fiabilité, et les performances générales des ordinateurs de l'époque.

Le transistor, comme le tube à vide qu'il remplace, fonctionne comme un interrupteur automatique. Il laissera ou non passer du courant entre deux de ses pattes en fonction de ce qui se passe sur sa troisième. On peut aussi le comparer à un robinet d'eau qui peut être ouvert ou fermé, et qu'on peut ouvrir ou fermer automatiquement sans devoir l'activer manuellement.

43. https://fr.wikipedia.org/wiki/Tube_%C3%A9lectronique

44. https://fr.wikipedia.org/wiki/Laboratoires_Bell

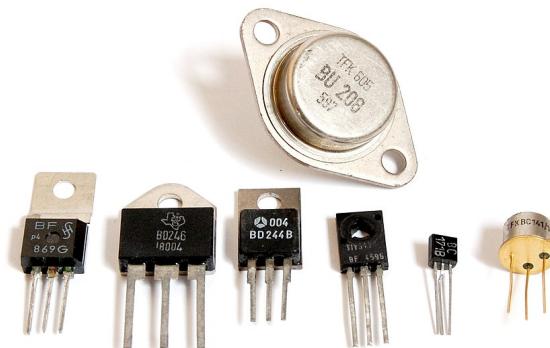
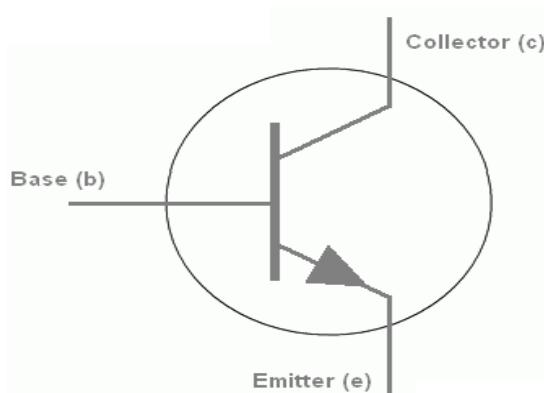


FIG. 3.5 – Différents modèles de transistor. On les reconnaît à leurs trois «pattes» aussi appelées : émetteur, base, collecteur.



© www.petervis.com

FIG. 3.6 – En appliquant un courant qui va de la base à l'émetteur (en rose pâle), on permet au courant de circuler entre le collecteur et l'émetteur (appelés ainsi parce que l'émetteur *émet* des électrons, et le collecteur les *collecte*). Envoyer du courant dans la base, c'est donc *ouvrir* le transistor ; ne plus en envoyer a, inversement, l'effet de *fermer* le transistor.

En appliquant un courant qui va de la base à l'émetteur (en rose pâle), on permet au courant de circuler entre le collecteur et l'émetteur (appelés ainsi parce que l'émetteur *émet* des électrons, et le collecteur les *collecte*). Envoyer du courant dans la base, c'est donc *ouvrir* le transistor ; ne plus en envoyer a, inversement, l'effet de *fermer* le transistor

De par sa capacité à être ouvert ou fermé, le transistor fonctionne comme une brique fondamentale dans la construction de systèmes informatiques permettant de *transmettre*, stocker et *traiter* des nombres binaires.

Pour aller plus loin

Les transistors sont faits avec des matériaux dit «semi-conducteurs». Voici une vidéo qui explique en détail ce qui se passe dans ces semi-conducteurs et qui permet de faire fonctionner un transistor.



Vidéo 4: <https://www.youtube-nocookie.com/embed/33vbFFFn04k>

Des transistors presque invisibles

Chercher à se représenter la taille des transistors utilisés dans les microprocesseurs actuels n'a pas d'intérêt tellement ils sont petits. À titre d'exemple, disons simplement que le microprocesseur Apple A9 en possède six milliards.

Zoom sur un transistor



Vidéo 5: <https://www.youtube-nocookie.com/embed/Fxv3JoS1uY8>

3.0.4 Des transistors aux systèmes logiques

Il reste difficile de concevoir des circuits d'ordinateurs en réfléchissant en termes de transistors. Un transistor seul ne peut représenter ou traiter qu'un bit d'information. Oui ou non, ouvert ou fermé, 1 ou 0.

Dans le chapitre suivant, où nous commençons à voir comment sont conçus les circuits électroniques des ordinateurs, nous parlerons tout d'abord de *portes logiques*. Ce sont des composants qui sont eux-mêmes constitués de plusieurs transistors. Réfléchir en termes de portes logiques permet de véritablement concevoir les circuits des ordinateurs qui vont manipuler les bits d'informations formant nos données.

3.1 Portes logiques

En informatique, les *systèmes logiques* décrivent comment sont connectés les circuits électroniques des ordinateurs afin de leur permettre, d'une part, d'effectuer des calculs et de traiter des données et, d'autre part, d'utiliser leur mémoire de travail, où sont stockées les données qu'ils traitent.

Même si on a l'impression que les ordinateurs peuvent faire toutes sortes de choses, il y a un ensemble limité d'opérations de base que l'électronique d'une machine peut faire. Parmi ces quelques opérations de base, on trouve l'addition, la soustraction, la multiplication ou la division de nombres. La plupart des tâches que l'ordinateur exécute reposent sur ces quelques opérations (ainsi que sur quelques opérations dites *logiques*, qui vont être explicitées).

C'est assez fascinant de se dire que des tâches a priori non mathématiques, comme corriger l'orthographe ou la grammaire d'un texte automatiquement, sont réalisées avec ces opérations de base.

En parallèle à ce qui leur permet de faire des calculs, les ordinateurs disposent et utilisent de la mémoire. Il y en a au cœur des microprocesseurs, les *registres*, ce qu'on appelle la *mémoire vive* — appelée aussi RAM (*Random-Access Memory*). La mémoire servant au stockage de longue durée comme disques durs et autres SSD n'est pas discutée dans cette section. L'étude des systèmes logiques permet de comprendre les principes derrière la gestion de cette mémoire et de voir comment les ordinateurs peuvent y lire et écrire des données entre deux calculs.

3.1.1 Exemple suivи : addition de deux nombres

On s'intéresse à une des opérations arithmétiques les plus simples : l'**addition**. Comment l'ordinateur additionne-t-il deux nombres ? On va définir le cadre de travail et s'intéresser aux circuits électroniques qui vont être à même de réaliser une addition.

Que se passe-t-il pour l'addition de deux nombres entiers ? On va utiliser leur représentation binaire (avec uniquement des 1 et des 0). Pour faire simple, on va chercher à additionner simplement deux bits, disons A et B , où chacun peut valoir soit 0 soit 1. Posons que la somme $S = A + B$. En énumérant tous les cas de figure, on a :

A	B	S
0	0	0
1	0	1
0	1	1
1	1	10

La dernière ligne est intéressante : on sait que $1 + 1 = 2$, mais en *binaire*, on sait aussi que n'existent que des 0 et des 1, et 2 s'écrit ainsi 10 (voir le chapitre *représentation de l'information*). Cela veut dire que, pour traiter tous les cas d'une addition de deux *bits*, on a besoin aussi de deux bits de sortie, et qu'un seul ne suffit pas. En explicitant chaque fois le deuxième bit de sortie, notre tableau devient :

A	B	S
0	0	00
1	0	01
0	1	01
1	1	10

La question est de déterminer comment faire calculer les deux bits de la somme S à partir de A et B à un circuit électronique. Pour cela, on a besoin du concept de *portes logiques*. Ces portes logiques sont elles-mêmes constituées de *transistors*, dont on a parlé en début de chapitre.

Dans un premier temps, on détaille les portes logiques et on s'intéresse à la réalisation des circuits logiques.

Ensuite, on regarde comment, fort de cette connaissance des portes logiques, il est possible de concevoir un circuit qui effectuera l'addition en question.

Finalement, on comprendra comment un ordinateur est capable, avec un circuit logique, de stocker le résultat d'un tel calcul afin qu'il soit réutilisable plus tard.

Les opérations arithmétiques et logiques et l'accès à la mémoire ne suffisent pas à constituer un ordinateur complet. C'est dans le chapitre suivant que sera traité la problématique de l'agencement de ces sous-systèmes afin de constituer une machine capable d'exécuter une suite d'instructions, c'est à dire un programme.

3.1.2 Portes logiques

Les circuits électroniques qui composent un ordinateur sont constitués de composants électroniques comme des *résistances*, des condensateurs, des *transistors*, etc., qui déterminent où va passer le courant électrique et sur quelles parties du circuit règnera quelle *tension*.

Quand on parle de portes et de circuits logiques, on simplifie tout cela. On considérera simplement qu'un segment de circuit électronique où la tension est nulle (0 volt) représente la valeur binaire 0, alors qu'une tension non nulle (par exemple, 3 volts) représente la valeur binaire 1. Ainsi, pour véhiculer deux bits comme A et B dans un circuit, on a besoin de deux «fils».

Les portes logiques sont des composants électroniques (eux-mêmes constitués en général de transistors et résistances) qui ont une ou plusieurs entrées et qui combinent ces entrées pour produire une sortie donnée. La manière dont la sortie est calculée dépend du type de la porte. On se propose à présent d'étudier en détails l'ensemble de ces portes.

3.1.3 Porte ET

Une de ces portes est la porte **ET**. Elle a deux entrées, qu'on appellera X et Y , et une sortie Z . Z sera 1 si et seulement si aussi bien X que Y valent 1. D'où son nom : il faut que X **et** Y soient à 1 pour obtenir un 1 sur la sortie.

En énumérant les quatre possibilités pour les entrées, on peut écrire ce qu'on appelle *table de vérité* pour la porte **ET** :

X	Y	Z
0	0	0
1	0	0
0	1	0
1	1	1

On peut dessiner des diagrammes avec des portes logiques. Ce ne sont pas des diagrammes électroniques, ils cachent une partie de la complexité réelle des circuits. Dans un tel diagramme logique, la porte **ET** est représentée ainsi :



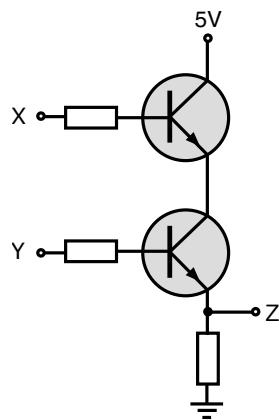
Sur ce schéma logique, les entrées sont à gauche, la sortie à droite et la porte est connectée au milieu. Les circuits sont représentés en noir s'ils véhiculent un «0» et avec une couleur s'ils véhiculent un «1».

Cliquez sur l'entrée X ou l'entrée Y pour changer leurs valeurs et observez le comportement de la sortie Z . Est-ce que cela correspond à la table de vérité ci-dessus ?



Pour aller plus loin

Comment une porte **ET** est-elle elle-même construite ? Cela a déjà été mentionné : avec d'autres composants électroniques plus simples. En simplifiant un peu, on peut considérer qu'une porte **ET** est constituée de deux transistors (avec quelques résistances en plus) :



Ici, les deux transistors sont les composants symbolisés par un cercle. Rappelons qu'ils laissent passer du courant de haut en bas lorsqu'ils détectent un courant sur l'entrée qui vient de la gauche. Ici, comme on a en haut une tension de 5 volts, on aura une tension similaire sur la sortie Z que si à la fois les entrées X et Y sont «actives» — donc lorsque les deux transistors sont «ouverts». Sinon, on aura une tension de 0 volt sur la sortie Z .

3.1.4 Porte OU

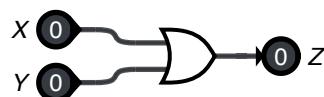
Pour que la sortie de la porte **OU** vaille 1, il suffit que l'une des deux entrées X ou Y vaille 1.

Voici sa table de vérité :

X	Y	Z
0	0	0
1	0	1
0	1	1
1	1	1

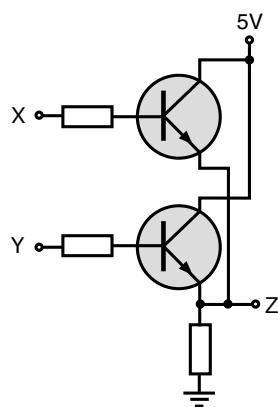
On notera que le **OU** logique est un peu différent du «ou» que l'on utilise en général à l'oral : on voit à la dernière ligne de la table de vérité que la sortie Z vaut également 1 si les deux entrées X et Y valent 1. À l'oral, le «ou» est en général interprété comme *exclusif* : si l'on propose à un ami un bonbon *ou* une glace, on exclut la possibilité qu'il choisisse les deux. Ce n'est pas le cas pour le **OU** logique.

Essayez la porte **OU** :



Pour aller plus loin

Voici comment une porte **OU** peut être construite avec deux transistors :



3.1.5 Porte NON

Cette porte est plus simple : elle n'a qu'une entrée, et sa sortie se contente d'inverser la valeur en entrée. On l'appelle d'ailleurs aussi un *inverseur*.

Voici sa table de vérité :

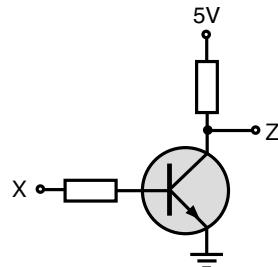
X	Z
0	1
1	0

Essayez l'inverseur :



Pour aller plus loin

Voici comment un inverseur peut être construit avec un transistor :



Ensemble, les portes **ET**, **OU** et **NON** représentent les relations logiques de la *conjonction*, la *disjonction* et la *négation*. Même si on ne les appelle pas ainsi, on utilise tous les jours des relations logiques de conjonction, de disjonction et de négation.

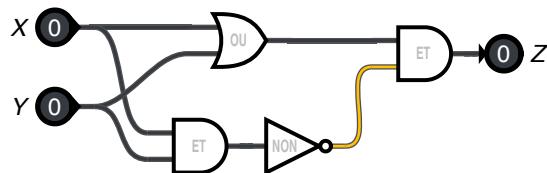
- La **conjonction** est ainsi l'«intersection logique» de deux propositions. Si on dit «je vais à la piscine *s'il fait beau ET que mes amis m'accompagnent*», on utilise la conjonction.
- Au contraire, si on dit «je vais à la piscine *s'il fait beau OU que mes amis m'accompagnent*», on utilise la **disjonction**, qui est comme une sorte de «somme logique» de deux propositions (même si, comme noté plus haut, le «ou», dans le langage courant, est généralement exclusif, contrairement au **OU** logique, qui est inclusif).
- La **négation** est encore plus évidente, puisque la proposition «je ne vais pas à la piscine» est simplement la négation, ou l'inverse, de la proposition «je vais à la piscine».

Ressource complémentaire

Une application pour s'exercer à l'interprétation des conjonctions, disjonctions et négations logiques : [The Boolean Game](#)⁴⁵

3.1.6 Combinaisons de portes

Les portes peuvent être connectées les unes aux autres. Voici par exemple un diagramme logique réalisant la fonction appelée **OU-X**, qui est un «ou exclusif» et dont la sortie *Z* vaut 1 lorsque soit *X*, soit *Y* vaut 1, mais pas les deux en même temps :



Ce circuit contient une porte **OU**, deux portes **ET** et un inverseur, tous interconnectés.

Ce diagramme n'est pas forcément facile à lire — discutons d'abord comment l'interpréter avec papier et crayon pour vérifier s'il effectue bien un **OU-X**.

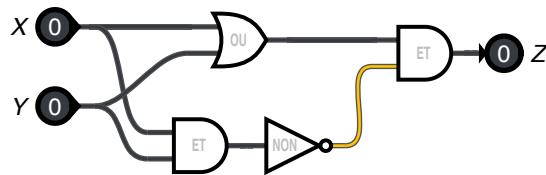
45. <https://booleangame.com/>

Analyse d'un circuit

Pour analyser un circuit logique comme celui présenté ci-dessus, on cherchera à établir sa table de vérité. En l'occurrence, comme pour les portes précédentes, ce circuit a deux entrées : si chaque entrée peut valoir 1 ou 0, on a en tout, de nouveau, quatre configurations possibles à examiner dans le but de remplir la dernière colonne :

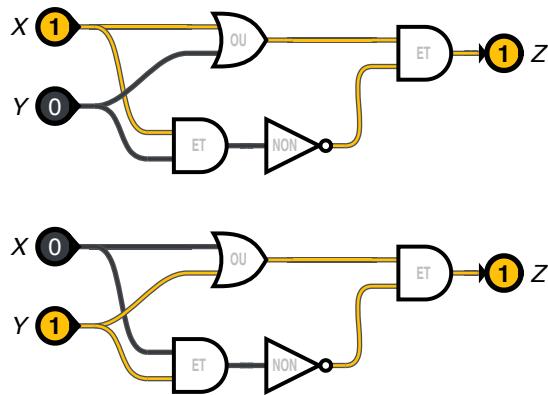
X	Y	Z
0	0	?
1	0	?
0	1	?
1	1	?

Pour remplir chaque ligne, on va changer les entrées selon les valeurs de X et Y et observer l'effet des portes et ainsi voir comment le circuit se comporte. Prenons $X = Y = 0$: c'est le cas représenté par le diagramme fixe ci-dessous. Rappelons qu'un segment noir véhicule un «0», alors qu'un segment coloré véhicule un «1».



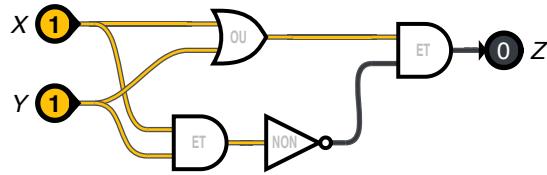
Le résultat intermédiaire des deux portes de gauche sera 0. L'inverseur transforme en 1 la sortie de la porte ET, mais la porte finale, qui est aussi une porte ET, n'obtient qu'un seul 1 en entrée et donc livre une sortie de 0.

Le cas est différent si l'une des deux entrées vaut 1. Voici deux diagrammes fixes, une fois pour $X = 1, Y = 0$ et une fois pour $Y = 1, X = 0$:



Ici, dans les deux cas, la porte OU, en haut, livrera un 1, dont a besoin la porte ET finale de droite pour donner une sortie de 1. La porte ET du bas, elle, continue de livrer un 0.

Mais dans le cas $X = Y = 1$, représenté ici, la situation est différente :



La porte **ET** du bas livre un 1, qui est inversé en 0 avant d'atteindre la porte finale, qui ne peut dès lors elle-même que livrer un 0 comme sortie.

La table de vérité complétée de ce circuit est ainsi :

X	Y	Z
0	0	0
1	0	1
0	1	1
1	1	0

Cette fonction s'appelle «ou exclusif», car pour avoir un 1 de sortie, elle exclut le cas où les deux entrées sont 1 en même temps. Elle est souvent utilisée, au point qu'on la représente en fait dans les diagrammes simplement par le dessin de cette porte, appelée **OU-X**, comme simplification du diagramme ci-dessus :



Exercice 1 – Vérification d'une porte

Vérifiez que la porte **OU-X** se comporte bien comme le circuit ci-dessous réalisé avec des portes **ET**, **OU** et **NON**.

Création d'un circuit

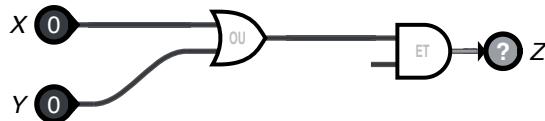
On s'intéresse à présent à la création de ce diagramme réalisant un **OU-X** avec les portes à disposition à partir de sa table de vérité. Plusieurs approches sont possibles, et on constatera que, suivant l'approche, on aurait très bien pu créer un circuit logique différent réalisant la même fonction.

Approche ad hoc

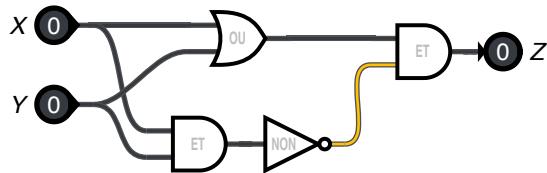
On se dit donc, selon la table de vérité, que la sortie de notre circuit «ou exclusif» doit être 1, donc l'une ou l'autre des entrées X ou Y est à 1, mais pas les deux. On peut ainsi commencer par insérer une porte **OU** dans le diagramme, qui fait une partie du travail. Mais il faut modifier sa sortie, pour ne pas avoir la valeur 1 lorsque les deux entrées sont à 1 : cela contredirait la quatrième ligne de la table de vérité. Comment effectuer cela ? En connectant la sortie de cette porte **OU** à une nouvelle porte **ET** à droite (dont on n'a pas encore déterminé la seconde entrée).

Pourquoi rajouter une porte **ET**? On utilise ici le fait que connecter une porte **ET** à un signal peut *restreindre* les conditions sous lesquelles la nouvelle sortie Z sera 1 (alors qu'au contraire, on aurait pu *étendre* ces conditions si on avait connecté une nouvelle porte **OU**). Comme si, pour être d'accord de finalement livrer 1 sur la sortie, la porte **ET** voulait la «confirmation» d'un autre signal avant de livrer 1...

À ce moment, on a ce diagramme partiel, qui peut être lu comme : «la sortie Z sera 1 lorsque ces deux conditions sont vraies en même temps : (1) le **OU** de X et Y vaut 1, et (2) quelque chose qui reste ici à définir, qui sera connecté à la seconde entrée de la porte **ET**».



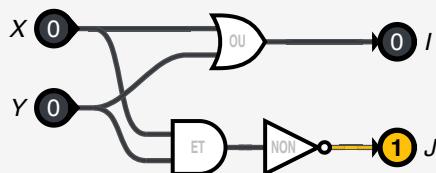
Ce qui reste à définir en complétant avant la porte **ET**, c'est l'exclusion du cas où X et Y valent les deux 1, de manière à ce que la condition (2) puisse être lue comme « X et Y ne sont pas en même temps les deux à 1». Avec une porte **ET** connectée directement aux deux entrées X et Y , on obtient une partie de ceci en créant le signal « X et Y sont les deux à 1». Mais c'est en fait la condition inverse de celle que l'on cherche. Pour l'inverser, on insère un inverseur à la sortie de cette nouvelle porte **ET**, ce qui complète le circuit :



La lecture finale du circuit est donc «la sortie Z sera 1 lorsque ces deux conditions sont vraies en même temps (selon la porte **ET** de droite) : (1) le **OU** de X et Y vaut 1, et (2) X et Y ne sont pas les deux en même temps à 1».

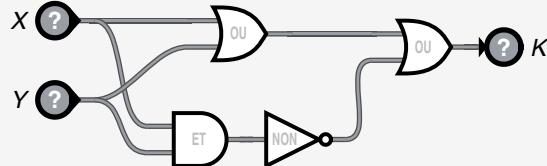
Exercice 2 – Analyse d'un circuit

Ceci est le même circuit que ci-dessus, mais sans la porte **ET** finale. À la place, on a inséré deux sorties intermédiaires, I et J , qui sont les deux signaux qui allaient précédemment à la porte **ET** :



1. Combien de lignes a une table de vérité pour I et J en fonction des deux entrées X et Y ? Écrivez cette table de vérité.
2. Quelle différence y a-t-il entre J et ce qu'on obtient en connectant directement une porte **ET** aux entrées X et Y : quel élément du schéma réalise cette différence ?
3. Dans votre table de vérité, ajoutez une colonne et remplissez-là : elle doit représenter une nouvelle sortie K , qui serait produite si on connectait une porte **OU** en lui donnant I et J comme entrées, comme montré ci-dessous. Le schéma représente ici le circuit dans un état

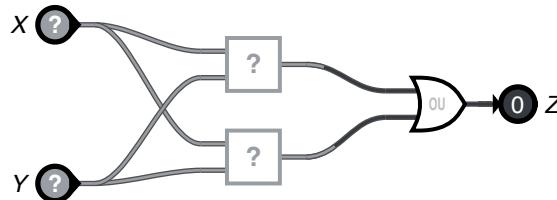
indéterminé, mais les types des portes ont été ajoutés pour vous aider. La sortie K est-elle ici toujours la même que la sortie Z plus haut ? Quelles sont les éventuelles différences ? Finalement, la sortie K a-t-elle un intérêt ?



Approche systématique

Il est parfois difficile d'avoir l'«intuition» nécessaire pour suivre une telle approche ad hoc. Voici donc une autre technique, illustrée avec le même exemple.

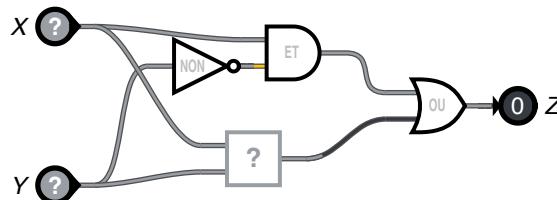
La table de vérité montre qu'il y a deux lignes où la sortie doit valoir 1 : (a) la ligne où $X = 1$ et $Y = 0$, et (b) la ligne où $X = 0$ et $Y = 1$. Si l'on pouvait créer un premier sous-circuit qui livre un 1 lorsque qu'on se trouve dans la circonstance (a) et un autre sous-circuit qui livre un 1 lorsqu'on se trouve dans la circonstance (b), on pourrait ensuite les combiner avec une porte **OU** et ainsi construire notre sortie Z ainsi :



Ici, les deux sous-circuits notés avec «?» et encadrés donc encore à définir — potentiellement avec plus d'une seule porte. Essayons de les créer.

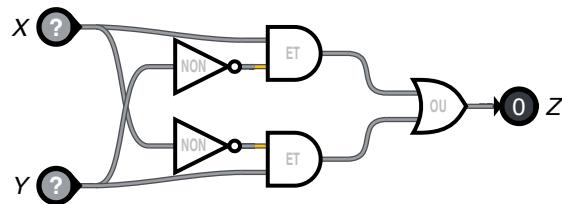
Disons que le sous-circuit du haut correspond à la deuxième ligne de la table de vérité, le cas de figure (a). Pour cette ligne, nous voulons un 1 de sortie lorsque $X = 1$ et $Y = 0$. En lisant littéralement cette dernière phrase, on y détecte un **ET** de deux conditions qui doivent être remplies : $X = 1$ et $Y = 0$. Mais ajouter une porte **ET** directement avec les signaux X et Y ne fera pas l'affaire, parce que cela livrerait un 1 lors que les *deux* entrées X et Y sont à 1. La solution ici, c'est d'*inverser* Y avant l'entrée dans la porte **ET** — ce qui donne bel et bien la condition (a).

On avance ainsi à ceci :



Pour la condition (b), qui correspond à la troisième ligne de la table de vérité, un raisonnement similaire s'applique. À la place d'inverser Y , on inversera cette fois X afin d'obtenir, à la sortie de la nouvelle porte **ET** du bas, un signal qui vaut 1 lorsque $X = 0$ et $Y = 1$.

Voici le circuit final ainsi réalisé :



(Ce schéma ne peut être simulé que dans l'indice de l'exercice suivant.)

Ce que cette approche systématique apprend, c'est qu'un circuit peut toujours être pensé comme un **OU** de toutes les conditions sous lesquelles la sortie doit être à 1. Ces conditions sont elles-mêmes réalisables avec les entrées du circuit avec des portes **ET** et des inverseurs directement connectés aux entrées.

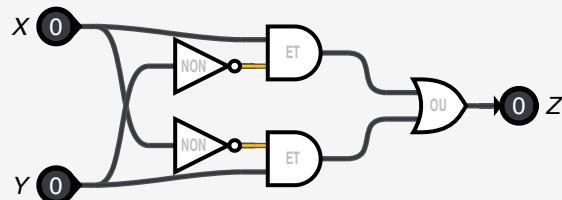
On fait également les constats suivants :

- plusieurs circuits logiques différents peuvent réaliser la même fonction de sortie,
- l'approche systématique décrite ici ne livre pas forcément le circuit le plus compact : on a obtenu un circuit avec cinq portes pour réaliser un **OU-X** alors que l'approche ad hoc a conduit à la construction d'un circuit à quatre portes.

Exercice 3 – Analyse d'un circuit

En annotant le schéma logique avec les quatre cas de figure possibles pour les entrées X et Y , faites l'analyse du circuit **OU-X** ci-dessus construit avec l'approche systématique et montrez que la table de vérité ainsi reconstituée est la même que celle de la porte **OU-X**.

Indice



Exercice 4 – Porte cachée

Quelle est la porte cachée de ce circuit ?

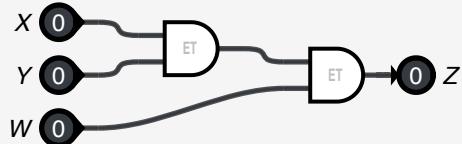


Corrigé

C'est une porte **OU**.

Exercice 5 – Circuit défectueux

Analysez ce circuit. De quel type de portes est-il constitué ? Fonctionne-t-il correctement ? Déterminez ce qui pose problème. Dites ce que fait ce circuit une fois corrigé et écrivez sa table de vérité.



Indice

Voici le circuit corrigé (il a la même apparence que le circuit de la question, mais toutes les portes fonctionnent ici correctement).



Corrigé

Ce circuit est constitué de deux portes **ET**. Mais la porte **ET** de droite semble poser problème, parce qu'elle se comporte comme une porte **OU** ! Le circuit montré dans l'indice se comporte correctement.

Ce circuit, une fois corrigé, implémente en fait un **ET** à trois entrée X , Y et W , où la sortie Z ne vaut 1 que si les trois entrées valent 1. Sa table de vérité, à huit lignes dues aux trois entrées, est ainsi la suivante :

X	Y	W	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Exercice 6 – Conception d'un circuit

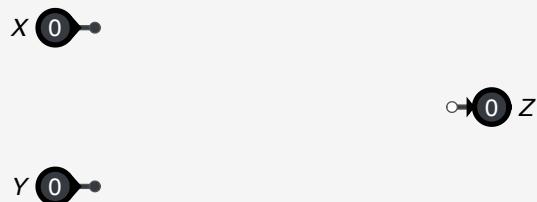
Écrivez la table de vérité de ce circuit, dont une partie est masquée :



Corrigé

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	1

Réalisez ensuite un circuit logique avec les mêmes deux entrées X et Y et la même sortie Z qui implémente cette table de vérité. On peut utiliser des portes **ET** et **OU** et des inverseurs. Glissez les portes depuis la gauche pour en ajouter, et glissez entre les connecteurs rond pour les connecter.



Indice 1

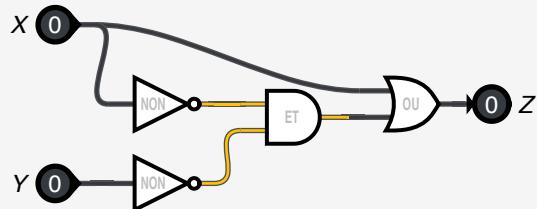
On peut lire cette fonction comme « Z vaut 1 lorsque X et Y sont les deux à 0 (la première ligne de la table de vérité) ou lorsque X est à 1 (les deux dernières lignes)».

Indice 2

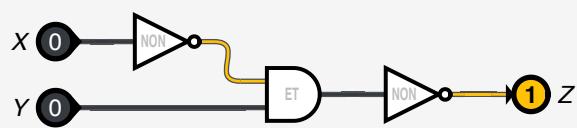
Z est donc le **OU** de X et du **ET** de l'inverse de X et de Y .

Corrigé

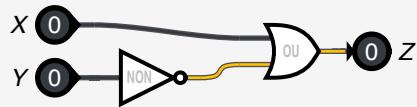
Il y plusieurs solutions possibles. Celle qui correspond aux indices est la suivante :



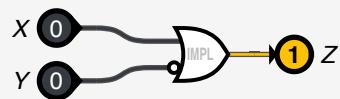
Une autre solution est la suivante, où on se dit qu'on construit d'abord une partie du circuit qui identifie le cas où $X = 0$ et $Y = 1$, et on l'inverse pour correspondre à la table de vérité.



Voici un circuit plus simple, qui fait la même chose mais qui est plus difficile à concevoir d'embrée :



En fait, il existe même une porte spéciale qui réalise exactement la fonction correspondant à la table de vérité, la porte **IMPLIQUE** :



3.2 Additionneur

On a découvert quelques *portes logiques* ainsi que la possibilité de les connecter pour en faire des circuits logiques plus complexes. Ces portes logiques vont maintenant permettre de réaliser l'additionneur annoncé en début de chapitre précédent.

On rappelle qu'on a deux *bits* de sortie à calculer pour la sortie $S = A + B$. S est donc constitué de S_0 , le bit des unités, et de S_1 , le bit représentant la valeur décimale 2. On rappelle ici la *table de vérité* pour S_0 , tirée directement du chapitre précédent :

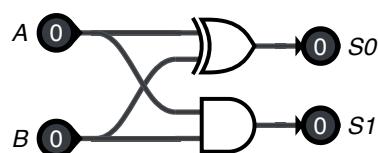
A	B	S_0
0	0	0
1	0	1
0	1	1
1	1	0

En comparant cette table de vérité avec celles des portes logiques, on se rend compte que S_0 n'est autre qu'un **OU-X** (OU exclusif) de A et B .

La table de vérité pour S_1 est :

A	B	S_1
0	0	0
1	0	0
0	1	0
1	1	1

Et on constate que S_1 n'est autre qu'un **ET** logique de A et B . On peut dessiner l'additionneur de deux bits ainsi :



Exercice 1

Vérifiez que ce circuit livre bien les bonnes valeurs de sortie qui correspondent aux tables de vérité ci-dessous. Combien de combinaisons différentes devrez-vous tester ?

Corrigé

Le circuit fonctionne correctement. Il faut tester les quatre combinaisons qui apparaissent dans les tables de vérité.

3.2.1 Additionneur complet

Le circuit précédent est particulièrement intéressant, car il montre qu'il est possible d'utiliser des opérateurs logiques pour réaliser l'opération arithmétique de l'addition. L'additionneur est limité : en fait, on l'appelle un *demi-additionneur*. Il n'est capable d'additionner que deux bits — c'est très limité. En fait, il serait intéressant d'avoir un additionneur de *trois* bits. Pourquoi ? À cause de la manière dont on pose les additions en colonnes.

Lorsqu'on additionne deux nombres à plusieurs chiffres, que ce soit en base 10 ou en base 2, on commence par la colonne de droite, les unités. On connaît le concept de *retenue* : en base 10, si l'addition des unités dépasse 9, on retient 1 dans la colonne des dizaines. En base 2, de façon similaire, si l'addition des unités dépasse... 1, on retient 1 dans la colonne suivante à gauche. C'est ce qu'on a fait avec le demi-additionneur : on peut considérer que la sortie S_0 représente la colonne des unités dans la somme, et la sortie S_1 représente la retenue à prendre en compte dans la colonne suivante.

C'est ici que ça se complique : pour additionner les chiffres de la deuxième colonne, on doit potentiellement additionner *trois* chiffres, et plus seulement deux. On a donc, en entrée, les deux bits A et B qui viennent des nombres à additionner, et aussi potentiellement cette retenue qui vient de la colonne des unités, qu'on appellera C_{in} (pour *carry*, «retenue» en anglais). Ceci est vrai en base 2 comme en base 10. Il faut donc un additionneur plus puissant, à trois entrées, pour prendre en compte cette retenue. Il s'appelle *_additionneur_complet* et livrera deux sorties : le bit de somme, appelé simplement S , et la retenue à reporter pour la colonne suivante, appelée C_{out} .

Exercice 2 – Bases de l'additionneur complet

- Déterminez combien de combinaisons différentes sont possibles pour trois signaux d'entrée A , B et C_{in} qui chacun peuvent valoir soit 1 soit 0.
- Listez toutes ces combinaisons.
- Pour chaque combinaison, déterminez la valeur binaire qui est la somme des trois signaux d'entrée.
- Finalement, avec les informations ainsi obtenues, complétez la table de vérité d'un additionneur complet qui a deux sorties S et C_{out}

Corrigé

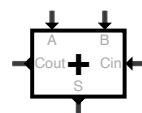
Il y a $2 \cdot 2 \cdot 2 = 2^3 = 8$ combinaisons différentes. Avec la notation $A + B + C =$ valeur en décimal = valeur en binaire, les voici :

- $0 + 0 + 0 = 0_{(10)} = 00_{(2)}$
- $0 + 0 + 1 = 1_{(10)} = 01_{(2)}$
- $0 + 1 + 0 = 1_{(10)} = 01_{(2)}$
- $0 + 1 + 1 = 2_{(10)} = 10_{(2)}$
- $1 + 0 + 0 = 1_{(10)} = 01_{(2)}$
- $1 + 0 + 1 = 2_{(10)} = 10_{(2)}$
- $1 + 1 + 0 = 2_{(10)} = 10_{(2)}$
- $1 + 1 + 1 = 3_{(10)} = 11_{(2)}$

La table de vérité est ainsi :

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

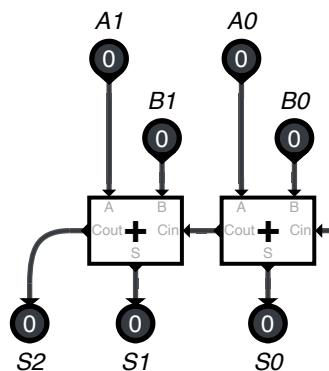
En faisant pour l'instant abstraction des détails d'un additionneur complet, on peut se dire qu'on le dessine simplement ainsi :



3.2.2 Chaînage d'additionneurs

La flexibilité de ce composant fait qu'on peut maintenant facilement l'utiliser pour construire un circuit qui additionne deux nombres A et B à 2 bits chacun (donc de $0 + 0 = 0$ à $3 + 3 = 6$).

Si A est formé de deux bits A_1 et A_0 et que B est formé des deux bits B_1 et B_0 et avec une sortie S sur trois bits S_0 , S_1 et S_2 , on a :



L'additionneur de droite, comme précédemment, additionne les deux bits des unités : A_0 et B_0 . Son entrée C_{in} , qui représente l'éventuel troisième chiffre à additionner issu d'une retenue, n'est pas connectée et est toujours 0, vu qu'il n'y a aucune colonne précédente dans l'addition qui aurait pu en livrer une. Il livre comme première sortie S_0 , le chiffre des unités, et sa seconde sortie C_{out} est la retenue à utiliser pour l'addition des chiffres suivants. C'est pourquoi elle est connectée à l'entrée de la retenue du second additionneur C_{in} , qui va lui ajouter également les deux bits de la colonne suivante, A_1 et B_1 . Les sorties du second additionneur livrent le deuxième bit S_1 de la valeur de sortie, ainsi que la retenue pour la troisième colonne. Comme il n'y a plus de bits d'entrée pour la troisième colonne, cette retenue peut directement être considérée comme le troisième bit de sortie S_2 .

Exercice 3 – Limite de cet additionneur à 2 bits

Avec l'additionneur ci-dessus, est-il possible d'obtenir des 1 sur toutes les sorties, donc d'avoir $S_2 = S_1 = S_0 = 1$?

Indice

Déterminez quel est le nombre décimal qui serait représenté par $S_2 = S_1 = S_0 = 1 : 111_{(2)} = ???_{(10)}$. Ensuite, déterminez les nombres les plus grands représentables sur les deux fois 2 bits d'entrée et tirez-en une conclusion.

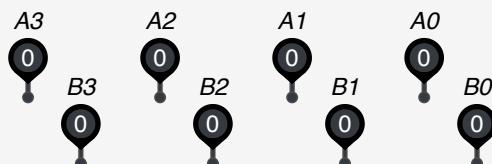
Corrigé

La configuration $S_2 = S_1 = S_0 = 1$ représente le nombre décimal 7. Ce serait le résultat de l'addition. Il faudrait ainsi chercher une configuration des bits d'entrées qui, une fois additionnés, donnent 7. Mais ceci n'est pas possible, car sur chacune des entrées (A_1, A_0) et (B_1, B_0) , la plus grande valeur représentable est $11_{(2)}$, autrement dit $3_{(10)}$ — et c'est impossible d'atteindre 7 en évaluant au maximum $3 + 3$.

Exercice 4 – Additionneur de demi-octets

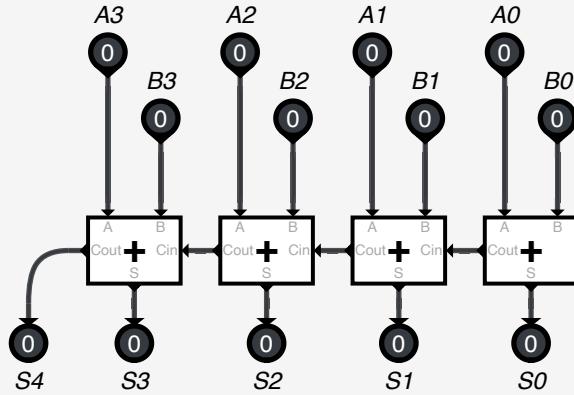
En connectant des additionneurs complets, réalisez un circuit qui additionne deux nombres A et B de quatre bits, numérotés A_0 à A_3 et B_0 à B_3 , respectivement. Combien de bits de sortie doit-il y avoir pour traiter toutes les valeurs possibles ?

Les entrées sont déjà disposées. Glissez autant d'additionneurs et de bits de sortie que nécessaire et connectez les composants du circuit.



Corrigé

On a besoin de cinq bits de sortie. Le schéma, représenté horizontalement et de droite à gauche pour être proche de la représentation selon laquelle les additions se résolvent en colonne, est :



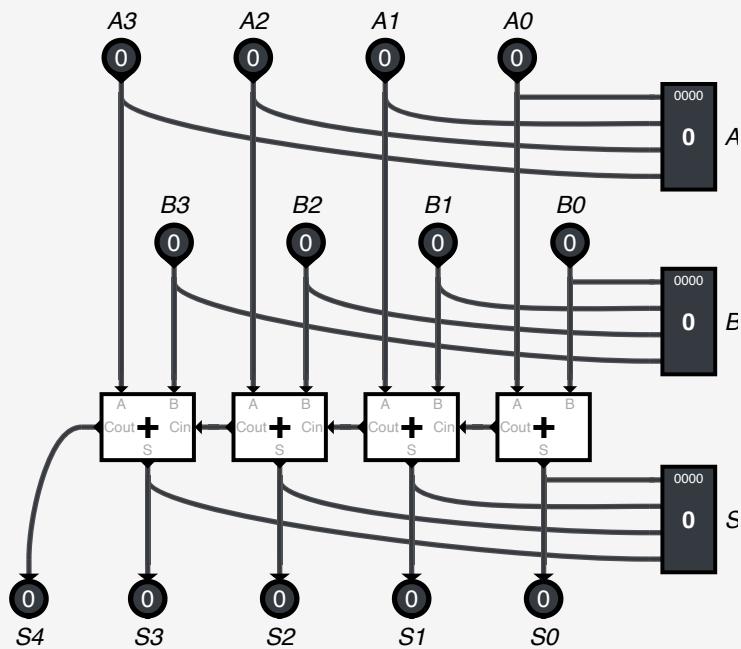
Cet exercice démontre l'opportunité de penser en termes modulaires, ce qui revient souvent en informatique. Ici, on a réalisé qu'un additionneur complet résout un sous-problème bien défini d'une addition générale d'un nombre à n bits, et qu'une fois qu'on a créé un tel additionneur, il suffit d'en connecter plusieurs les uns derrière les autres de manière structurée pour additionner des nombres plus grands.

Exercice 5 – Dépassement de capacité

Le schéma ci-dessous montre le même additionneur de demi-octets de l'exercice précédent, mais, de plus, la valeur en base 10 de ses 4 bits d'entrée pour A et pour B est affichée avec un module d'affichage spécial à droite. La même chose est faite pour représenter la valeur $S = A + B$ (mais seulement sur les quatre premiers bits de S). Actuellement, le circuit effectue le calcul $0 + 0 = 0$.

Réglez les entrées du circuit de manière à lui faire effectuer les additions suivantes, et vérifiez le résultat. Dans quelles circonstances est-il correct et pourquoi est-il de temps en temps incorrect ? Comment, en regard de ceci, interpréter le bit de sortie S_4 , qui est la retenue de l'additionneur de gauche ?

1. $1 + 0$
 2. $3 + 1$
 3. $3 + 3$
 4. $10 + 5$
 5. $14 + 1$
 6. $14 + 2$
 7. $15 + 15$



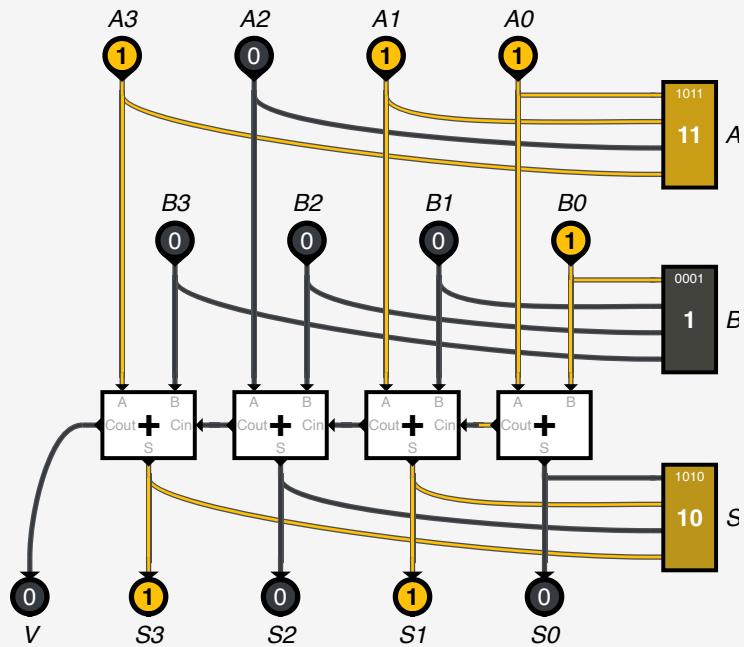
Corrigé

Dès que la somme dépasse 15, elle n'est plus représentable sur les 4 bits qui sont affichés sur la sortie. La plupart des ordinateurs et smartphones actuels représentent les nombres non pas sur 4 bits, mais sur 64. Mais même avec 64 bits, il y a un nombre maximal que l'on peut représenter (en l'occurrence, $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$.) La retenue du dernier additionneur indique si le résultat est valable : il vaut 1 lorsque le résultat de l'addition n'est pas correctement représenté avec les 4 (ou 64) bits de sortie. Dans les processeurs, il porte souvent simplement le nom de *C* (pour *carry*, retenue). On utilisera dorénavant aussi ce nom.

Exercice 6 – Circuit défectueux

L'additionneur de demi-octets ci-dessous a été endommagé et ne fonctionne plus correctement. Par exemple, lorsqu'on lui demande d'effectuer le calcul $11 + 1$, il livre comme réponse 8.

Déterminez quel composant est défectueux dans ce circuit et comment il faudrait le réparer. Vous pouvez changer les entrées pour vérifier ce qui ne fonctionne pas.



Corrigé

La retenue sortant du deuxième additionneur depuis la droite est bloquée à 0 à la place de correctement changer de valeur suivant ses entrées.

Exercice 7 – Design d'un additionneur complet

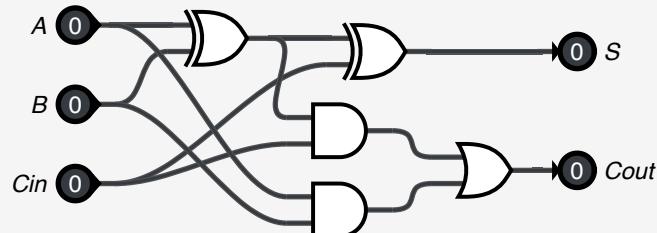
Note : exercice difficile et actuellement peu guidé ici; prochainement complété par davantage d'indications.

En s'aidant de la table de vérité d'un seul additionneur complet, créer un circuit logique qui calcule ses sorties S et C_{out} en fonction des entrées A , B et C_{in} .

Indice

- La sortie S doit être 1 soit lorsque les trois entrées valent 1, soit lorsqu'une seule des trois entrée vaut 1.
- La sortie C_{out} , qui est la retenue, doit être 1 lorsque deux ou trois des trois entrées valent 1.

Corrigé



3.3 ALU

Dans cette section, nous continuons notre exploration de comment les portes logiques, selon leur assemblage, fournissent les fonctionnalités de base des ordinateurs. En particulier, nous nous intéressons à comment faire effectuer plusieurs opérations à un ordinateur via ce qui s'appelle une **unité arithmétique et logique**, ou simplement une ALU.

Dans la section précédente, nous avons vu comment créer, via un assemblage de portes logiques, un circuit qui réalise l'addition de deux nombres de 4 bits. Ce circuit était fixe : avec les deux nombres d'entrées, il réalisait toujours une addition et ne servait ainsi qu'à ça.

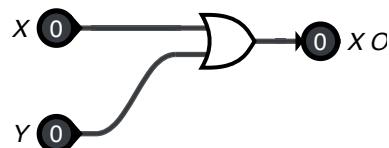
Les ordinateurs ont la propriété d'être programmables : ils savent effectuer plusieurs opérations, et c'est la manière dont ils sont programmés qui va déterminer l'opération qu'ils effectuent. C'est aussi vrai pour des machines plus simples ; une calculatrice de poche, par exemple, pourra effectuer au moins les quatre opérations de base : addition, soustraction, multiplication et division.

Le composant qui nous permettra de sélectionner une opération ou une autre s'appelle «unité arithmétique et logique», communément appelé simplement «ALU» (de l'anglais *arithmetic logic unit*). Avant d'inspecter une ALU, nous avons besoin de comprendre comment on peut sélectionner une opération ou une autre avec des circuits logiques.

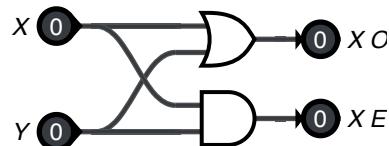
3.3.1 Sélection de l'opération

Comment créer un circuit qui permet de sélectionner une opération à faire, et comment indiquer l'opération à sélectionner ? Essayons d'abord de créer un circuit à deux entrées qui va calculer soit le **OU** soit le **ET** de ces deux entrées.

Nous savons faire un circuit simple qui calcule le **OU** de deux entrées X et Y , avec une seule porte logique :

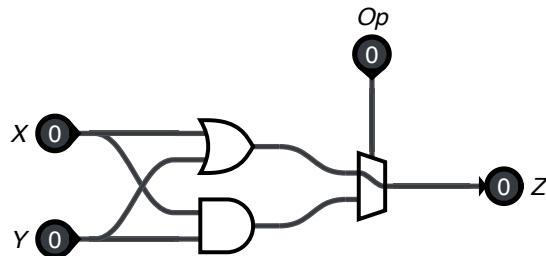


Nous pouvons sans autre y ajouter une porte **ET** pour calculer une autre sortie en parallèle, à partir des mêmes entrées X et Y :

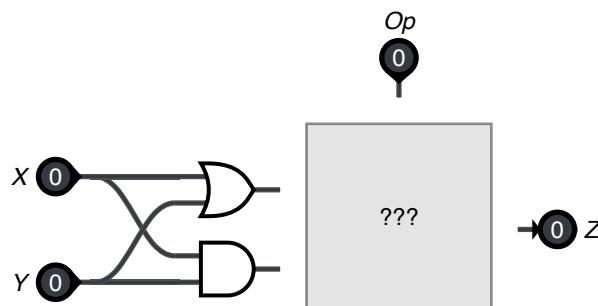


L'idée est maintenant de combiner ces sorties intermédiaires pour n'en avoir plus qu'une, qui sera soit le **OU**, soit le **ET**. Mais comment faire pour indiquer si l'on désire le **OU** ou le **ET** ? Nous pouvons rajouter une entrée pour choisir cette opération. Appelons-la Op , pour «opération». Choisissons une convention : lorsque Op vaut 0, nous souhaitons effectuer l'opération **OU**, et lorsque Op vaut 1, nous souhaitons effectuer l'opération **ET**.

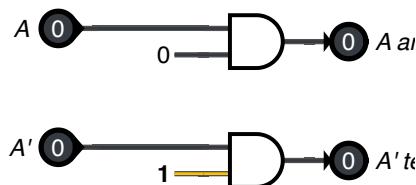
Ce que nous cherchons à créer est conceptuellement ce que fait ce composant tout prêt ci-dessous. Il s'appelle *multiplexeur* et connecte à sa sortie soit le signal du haut, soit le signal du bas, en fonction de la valeur de Op . Essayez de changer Op :



Mais comment construire ce multiplexeur avec les portes logiques que nous connaissons ? Sans le multiplexeur, voici ce que nous devons compléter :



Pour savoir quelles portes utiliser pour recréer le multiplexeur, nous avons besoin de nous rappeler ceci. Lorsqu'un signal, disons A , passe à travers une porte logique **ET** dont la seconde entrée vaut 0 (ici affichée sans qu'on puisse la changer en cliquant dessus), la sortie de cette porte-là sera toujours identique à 0. Cela nous permet ainsi d'*annuler* le signal A — de le mettre à zéro. De manière similaire, si cette seconde entrée vaut 1, le signal A passera tel quel. On peut ainsi voir la porte **ET** comme un annulateur de signal. Vérifiez ceci ici :



Ensuite, lorsqu'un signal, disons B cette fois, passe à travers une porte logique **OU** dont la seconde entrée est annulée et vaut 0, la sortie de cette porte sera toujours identique à B . Vérifiez ceci ici :

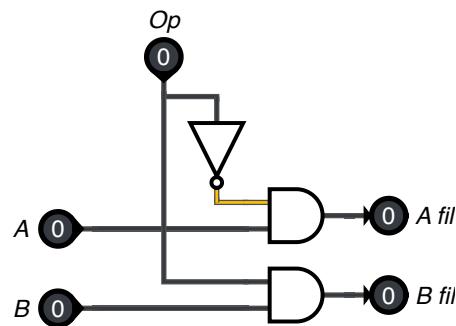


La porte **OU** peut ainsi servir à combiner deux signaux, pour autant que l'un soit annulé.

Avec tout ceci, on peut ainsi construire un multiplexeur (un sélecteur de signal). Supposons qu'on ait les deux signaux A et B : nous pouvons construire un circuit qui combine soit A tel quel et B annulé, soit A annulé et B tel quel. Cela nous aidera pour notre projet initial ! Il faut cependant s'assurer que A soit chaque fois annulé quand B passe tel quel, et inversement. Ceci peut se faire en réutilisant l'idée d'une entrée de contrôle Op ainsi. Nous avons ainsi deux cas :

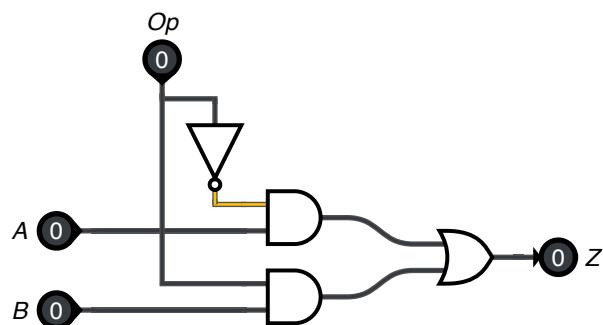
- Lorsque $Op = 0$, on va sélectionner A et annuler B . On va donc faire passer A à travers une porte **ET** à laquelle on donne 1 à l'autre entrée, et faire passer B à travers une porte **ET** à laquelle on donne 0 à la seconde entrée.
- Lorsque $Op = 1$, on va faire exactement l'inverse : sélectionner B et annuler A . On donnera donc un 0 à la porte **ET** qui filtre A , et 1 à la porte **ET** qui filtre B .

En relisant ces lignes, on voit que ce qu'on donne à la seconde entrée de la porte qui filtre B est toujours la même chose que Op , et que ce qu'on donne à la seconde entrée de la porte qui filtre A est toujours l'inverse de Op . On peut donc construire ce circuit avec un inverseur en plus :

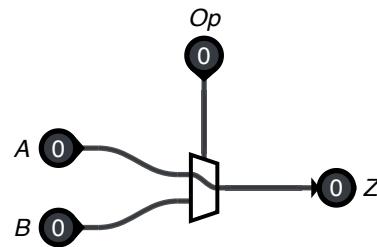


Essayez ce circuit. Quand $Op = 0$, B filtré sera toujours 0 mais A passera, et inversement.

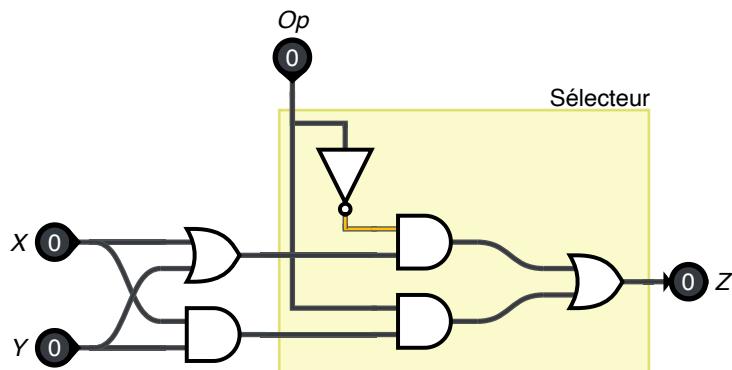
Pour recombiner ces sorties filtrées, il ne nous reste plus qu'à les connecter par une porte **OU** :



Essayez ce circuit pour confirmer qu'il agit comme un sélecteur : lorsque $Op = 0$, on aura sur la sortie $Z = A$, et lorsque $Op = 1$, on aura $Z = B$. Comparez-le avec le multiplexeur tout seul ci-dessous : est-ce que notre circuit fait bien la même chose ?



Ceci nous permet de compléter le circuit lacunaire de début de chapitre pour sélectionner avec le même mécanisme soit le **OU** soit le **ET** de nos deux entrées X et Y . On ajoute les portes de notre sélecteur en connectant à l'entrée A le signal représentant $X \text{ OU } Y$, et à l'entrée B le signal représentant $X \text{ ET } Y$:



Exercice 1 – Test du sélecteur OU/ET

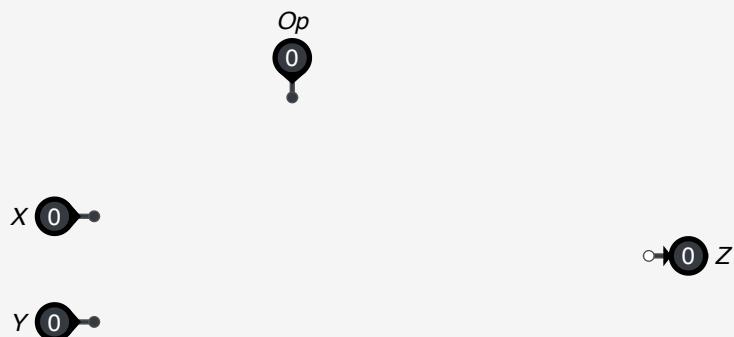
Testez le circuit ci-dessus. Établissez la table de vérité de Z en fonction de X , Y et Op . À l'aide de la table de vérité, montrez que, lorsque $Op = 0$, Z représente bien $X \text{ OU } Y$, et que, lorsque $Op = 1$, Z représente bien $X \text{ ET } Y$.

Nous avons ici construit un circuit qui, grâce à un bit de contrôle Op , sélectionne une opération ou une autre à appliquer à ses deux bits d'entrées X et Y .

Exercice 2 – Construction d'un sélecteur

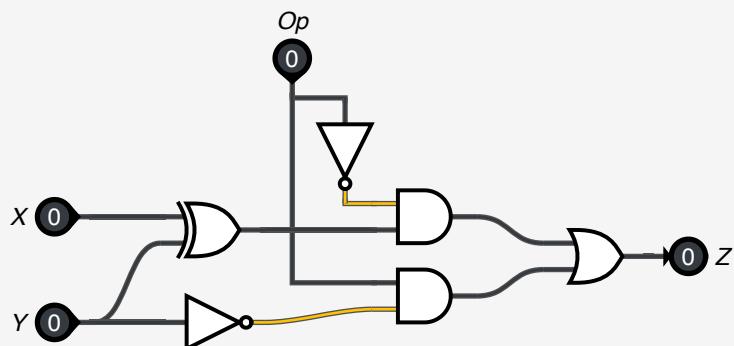
En réutilisant les principes appliqués ci-dessus, construisez un circuit à deux bits d'entrées X et Y et un bit de contrôle Op qui donnera sur sa sortie Z :

- Le **OU** exclusif de X et Y , lorsque $Op = 0$;
- L'inverse du bit Y , lorsque $Op = 1$.



Corrigé

Voici un circuit qui réutilise le sélecteur de signal et qui fournit à ce sélecteur les deux nouvelles entrées décrites, à haut, le **OU** exclusif de X et Y tel que fourni par une porte **OU-X**, et en bas, Y une fois inversé par une porte **NON** :



Exercice 3 – Inverseur conditionnel

En réutilisant les principes appliqués ci-dessus, construisez un circuit à une entrée X avec un bit de contrôle Op qui donnera sur sa sortie Z :

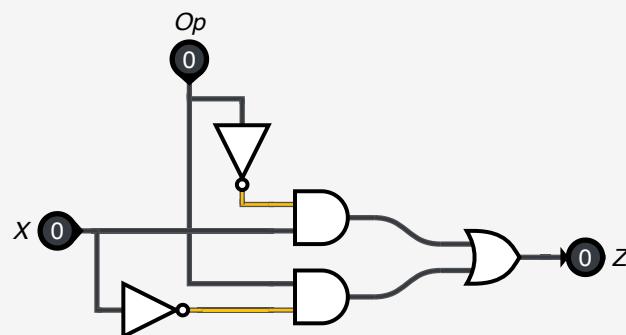
- X tel quel, lorsque $Op = 0$;
- X inversé, lorsque $Op = 1$.

Écrivez la table de vérité de ce circuit. Correspond-elle par hasard à une porte déjà connue ? Serait-ce dès lors possible de simplifier votre circuit ?

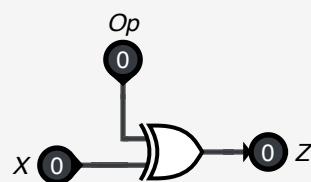


Corrigé

Voici une proposition qui réutilise le sélecteur de signal et qui fournit à ce sélecteur *X* en haut et *X* inversé en bas :



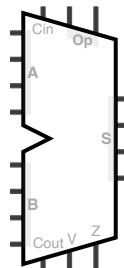
La table de vérité est identique à celle d'une porte **OU-X**. On peut donc simplement remplacer tout le circuit par cette unique porte :



3.3.2 Une ALU à 4 bits

Une unité arithmétique et logique, ou ALU, est un circuit qui ressemble dans ses principes de base à ce que nous venons de faire. L'ALU réalise plusieurs opérations et permet de sélectionner, via un ou plusieurs bits de contrôle, l'opération qui est réalisée. Les opérations proposées sont, comme le nom de l'ALU indique, des opérations arithmétiques (typiquement, l'addition et la soustraction) et des opérations logiques (par exemple, un **ET** et un **OU** logiques).

Nous présentons ici une ALU simple à 4 bits :



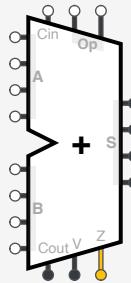
Cette ALU sait effectuer l'addition ou la soustraction de deux nombres entiers représentés sur 4 bits. Elle a ainsi 8 bits d'entrée pour les données et 4 bits de sorties, à gauche et à droite. En plus de l'addition et de la soustraction, elle sait aussi faire les opérations logiques **ET** et **OU** — en tout donc, quatre opérations. Pour sélectionner l'une des quatre opérations, on ne peut plus se contenter d'un seul bit de contrôle, mais nous allons en utiliser deux pour avoir quatre combinaisons possibles. Ce sont les deux entrées supérieures gauches de l'ALU (on ignore ici l'entrée C_{in}).

La convention utilisée pour la sélection de l'opération est la suivante :

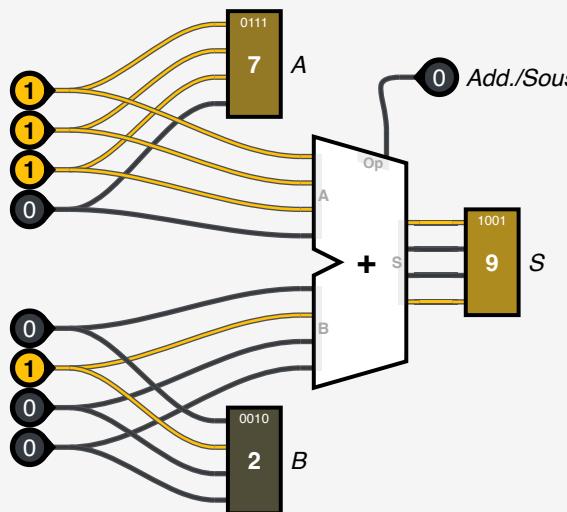
<i>Op</i>	Opération effectuée
00	Addition
01	Soustraction
10	OU
11	ET

Exercice 4 – Test de l'ALU

Connectez cette ALU à 8 entrées et à 4 sorties de manière à lui faire effectuer l'opération $7 + 2 = 9$. Connectez les 4 bits des entrées et de la sortie à des afficheurs de demi-octet pour vérifier leur fonctionnement. Connectez ensuite une entrée pour le bit de contrôle qui permettra d'effectuer la soustraction avec les mêmes données d'entrée, donc $7 - 2 = 5$.



Corrigé



L'ALU a trois sorties en plus, en bas du composant :

- la sortie C_{out} (pour *carry out*) vaut 1 lors d'un dépassement de capacité (si le résultat de l'opération arithmétique représenté sur la sortie n'est pas valable parce qu'il vaudrait davantage de bits pour le représenter ; par exemple, le résultat de $8 + 8 = 16$ n'est pas représentable sur 4 bits, qui suffisent à représenter les valeurs entières jusqu'à 15 seulement) ;
- la sortie V (pour *oVerflow*) vaut 1 lors d'un dépassement de capacité si on considère les entrées et les sorties comme des nombres signés. Nous ne faisons pas cela ici et ignorons cette sortie ;
- finalement, la sortie Z (pour *Zero*) vaut 1 lorsque tous les bits de sortie valent 0.

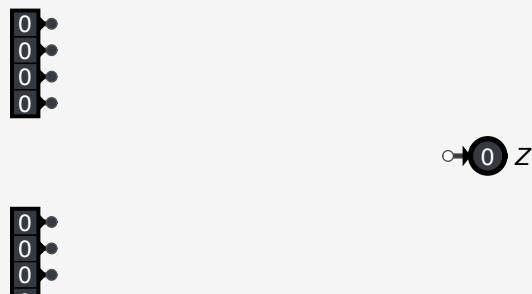
Exercice 5 – Une ALU comme comparateur

En programmation, c'est fréquent de tester, par exemple dans une condition avec un `if`, si deux valeurs sont égales. Par exemple, ce fragment de code affichera «Ces valeurs sont égales !» uniquement si les deux nombres entiers donnés lors de l'exécution du code sont les mêmes :

```
A = int(input("Quel est le premier nombre? "))
B = int(input("Quel est le second nombre? "))
if A == B:
    print("Ces valeurs sont égales!")
```

1
2
3
4

Ce qui nous intéresse spécialement, c'est la comparaison à la ligne 3. Cette comparaison peut être réalisée avec une ALU. Pour cet exercice, créez un circuit avec une ALU qui compare deux nombres de quatre bits et indique sur la sortie Z un 1 si les deux nombres sont égaux et un 0 s'ils sont différents.

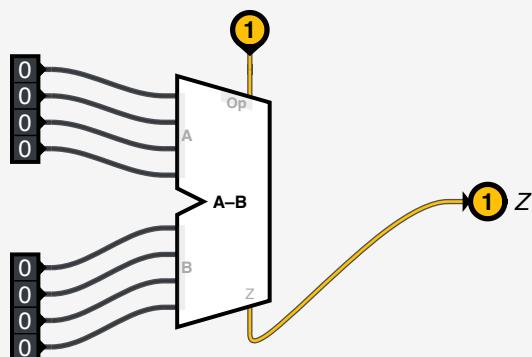


Indice

Deux nombres A et B sont égaux si leur différence est nulle — donc si tous les bits de sortie de la soustraction $A - B$ valent 0.

Corrigé avec ALU — approche arithmétique

On connecte les 8 entrées, on règle l'opération de l'ALU sur soustraction et on utilise la sortie de l'ALU qui indique si tous les bits de sortie sont à zéro. En effet, cela ne se produit que lorsque la différence entre les deux nombres d'entrée est 0 — c'est-à-dire, s'ils sont égaux. On constate qu'on peut ignorer les 4 bits de sorties ici !



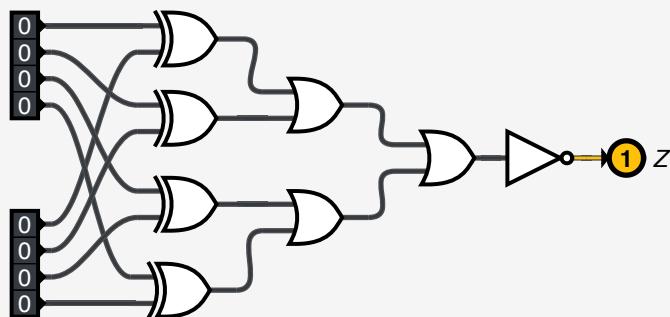
Plus difficile : essayez de réaliser un circuit qui calcule la même valeur de sortie, mais sans utiliser d'ALU.

Indice

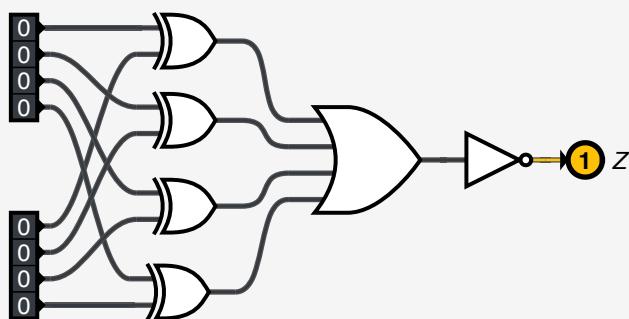
Une porte **OU-X** peut être vue comme un comparateur de deux bits : sa sortie vaudra 1 si et seulement si ses deux entrées sont différentes.

Corrigé sans ALU — approche logique

Cette solution utilise des portes **OU-X** comme comparateurs. On voit ici que 4 portes **OU-X** comparent deux à deux les 8 bits d'entrée. Leurs sorties sont ensuite combinées avec des portes **OU**, afin d'obtenir un signal qui vaudra 1 si au moins une différence est détectée, donc si les deux nombres d'entrées ne sont pas égaux. Il ne reste plus qu'à inverser ce signal pour obtenir la sortie demandée qui, selon la donnée, doit valoir 1 lorsque les nombres sont égaux.



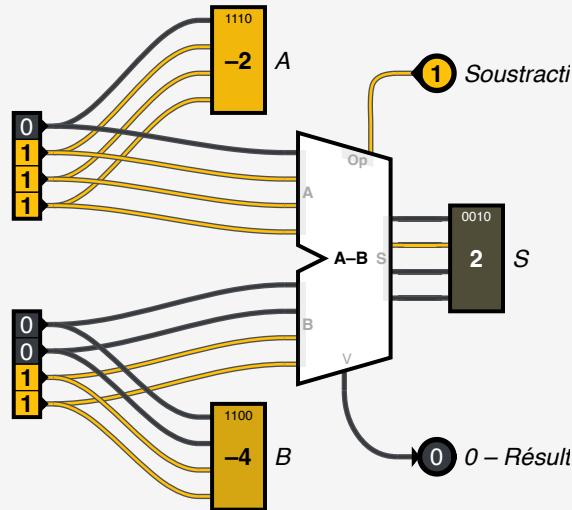
Alternativement, à la place d'utiliser trois portes **OU** dans le schéma ci-dessus, on aurait pu utiliser une grande porte **OU** à quatre entrées :



En résumé, nous avons appris ici ce qu'est une unité arithmétique et logique et avons examiné de plus près comment construire un multiplexeur, un circuit qui est à même de «choisir» parmi plusieurs signaux d'entrées lequel il va propager sur sa sortie. L'ALU est spécialement intéressante, car c'est le premier composant que nous rencontrons qui incarne une des propriétés de base d'un ordinateur, à savoir d'être programmable, en faisant en sorte que l'opération qu'elle effectue dépende d'un signal externe.

Pour aller plus loin

Notre petite ALU peut aussi faire des calculs en utilisant une représentation signée des nombres entiers. Sur 4 bits, une représentation en complément à deux peut représenter les nombres de -8 à $+7$. Il est possible d'utiliser les mêmes afficheurs de demi-octets en mode signé pour effectuer des opérations arithmétiques avec des valeurs négatives, par exemple, ici, $-2 - (-4) = 2$:



Notez que grâce à la représentation en complément à deux, la circuiterie interne de l'ALU peut se permettre de complètement ignorer si ses entrées sont signées ou pas et livrera le bon résultat tant que la convention d'entrée et de sortie reste la même.

Attention, lorsqu'on interprète les entrées et la sortie comme des nombres signés, ce n'est plus la sortie C_{out} de l'ALU qui indique un dépassement de capacité, mais la sortie V , qui est calculée différemment.

Essayez de régler les entrées pour que cette ALU calcule le résultat de $-8 - 4$. Vérifiez qu'un dépassement de capacité est signalé et expliquez pourquoi.

3.4 Mémoire

Les *transistors*, les *portes logiques* et leur représentation en *tables de vérités*, permettent de manipuler des 0 et des 1 au niveau physique... Tant qu'un courant électrique se déplace dans les *circuits*, on est capable de le transformer, de le laisser passer ou de l'arrêter, dans le but d'exprimer des portes «ouvertes» ou des portes «fermées» et donc des nombres binaires. L'ALU, explorée au chapitre précédent, va une étape plus loin et permet de choisir une opération à effectuer en fonction de bits de contrôle supplémentaire, et livre le résultat de l'opération arithmétique ou logique choisie.

Mais comment faire pour *stocker* cette information ? Comment faire pour que l'on se rappelle le résultat d'une addition effectuée par une ALU afin de pouvoir réutiliser cette valeur plus tard ? C'est là que nous avons besoin de *mémoire*.

Dans les ordinateurs, il y a en fait plusieurs types de *mémoires*, qu'on peut classer en deux grandes catégories. La *mémoire volatile*, et la mémoire non volatile. La mémoire volatile s'efface quand la machine est éteinte. C'est le cas de la RAM (*random-access memory*), par exemple. La *mémoire non volatile*, elle, persiste. C'est le cas d'un disque dur ou d'un SSD (*solid-state drive*). Si un smartphone s'éteint inopinément alors qu'on est en train de retoucher une photo sans avoir validé les modifications, ces retouches disparaissent. Elles étaient stockées sur la mémoire volatile. Par contre, au moment où ces retouches sont sauvegardées, elles s'inscrivent dans la mémoire non volatile.

On peut se demander pourquoi on n'utilisera pas que de la mémoire non volatile, vu les «risques» posés par la mémoire volatile. La réponse est que la mémoire non volatile va probablement être entre 100 et 100000 fois moins rapide que la mémoire volatile. On priviliege donc la mémoire volatile comme mémoire de travail rapide d'un ordinateur.

Dans les sections qui suivent, on propose de s'intéresser au cas le plus simple : la construction d'une cellule de mémoire volatile qui sera à même de stocker un bit. Par la suite, nous discuterons de la manière dont ce genre de mémoire est utilisée au cœur des microprocesseurs.

3.4.1 Le verrou SR

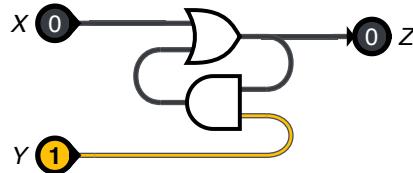
L'idée principale derrière la conception d'un circuit logique qui est capable de stocker un signal est que l'on va utiliser la ou les sorties du circuit en les reconnectant à certaines de ses entrées. Essayons par exemple ce circuit simple avec une seule porte **OU** :



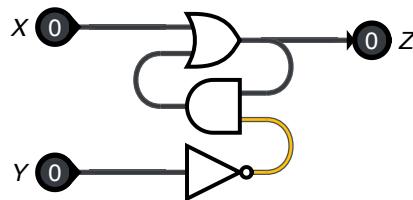
Au début, les deux entrées de la porte valent 0, comme sa sortie. Si l'on essaie de faire passer l'entrée X à 1, on voit que la sortie Z passera à 1 elle aussi, comme il s'agit d'une porte **OU**. Mais comme Z est aussi relié à l'autre entrée de la porte, on a maintenant un circuit dont on ne peut plus modifier la sortie : même si X passe de nouveau à 0, l'autre entrée reste à 1 et suffit donc pour que Z vaille maintenant 1 indéfiniment. On est obligé de remettre le circuit complètement à zéro (l'équivalent de débrancher la prise de courant et de la rebrancher) pour obtenir à nouveau un 0 sur la sortie Z .

Assurément, ce circuit n'est pas très intéressant : il se bloque dans un état sans retour possible. Il faudrait pouvoir faire repasser la valeur de sortie à 0. Pour ce faire, une idée est d'ajouter une porte **ET** avant de faire repasser la sortie de la porte **OU** dans sa propre entrée. Cela nous permet d'annuler le signal de retour si la seconde entrée du **ET** (appelons-la Y) vaut 0.

Essayez ce circuit : quand Y vaut 1, il se comporte comme le circuit précédent, mais se remettra à 0 dès qu' Y passera à 0.



Le circuit est plus facile à utiliser lorsqu'on inverse l'entrée Y , comme dans le circuit suivant :

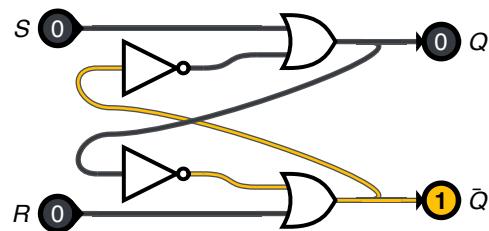


Dans ce circuit-ci, on peut considérer que :

- Tant que Y est inactif (donc vaut 0), le passage de X de 0 à 1 fait passer la sortie Z à 1. Ceci fait donc en sorte que la sortie «verrouille» sur la valeur 1, car elle garde sa valeur même si X repasse à 0.
- Tant que X est inactif, le passage de Y de 0 à 1 fait passer la sortie Z à 0, et la sortie se «verrouille» ainsi à 0 même si Y repasse à 0.

On appelle effectivement ce genre de circuits des *verrous*. Nous avons ici construit un des plus simples. D'habitude, son entrée X est appelée *S*, pour *set* en anglais, qui dénote le stockage d'un 1, et son entrée Y est appelée *R*, pour *reset* en anglais, qui dénote sa remise à zéro. C'est le verrou dit «**SR**», pour *set/reset*. Un tel verrou est donc une minicellule mémoire ! La sortie de ces cellules mémoires est fréquemment appelée Q plutôt que Z , nous allons donc faire pareil dans la suite du texte.

Voici une représentation équivalente du même circuit⁴⁶, donc du même verrou SR :



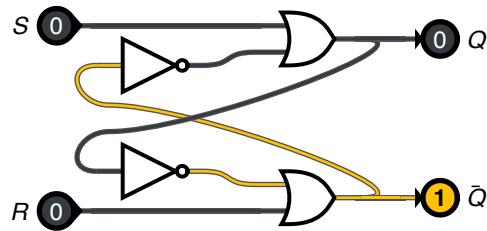
46. On peut montrer l'équivalence de ce circuit et du précédent à l'aide des [lois de De Morgan](#)⁴⁷, qui ne sont volontairement pas abordées dans ce chapitre.

47. https://fr.wikipedia.org/wiki/Lois_de_De_Morgan

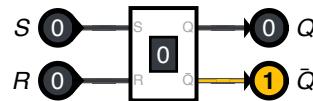
Ce circuit stocke ainsi un bit de donnée — un 0 ou un 1 — qu'on va pouvoir lire via la sortie Q et modifier avec les deux entrées R et S . (La seconde sortie \bar{Q} est ici toujours l'inverse de Q .)

Testez le circuit ci-dessus et observez l'effet de R et S pour vérifier qu'il correspond bien à notre circuit précédent.

On essaie en général d'éviter d'avoir un 1 sur R et sur S en même temps, car cela place le verrou dans un état où \bar{Q} n'est plus l'inverse de Q . Pour cette raison, nous allons plutôt créer le circuit comme dans le schéma suivant. Les connexions sont exactement les mêmes, mais les entrées S et R ne restent pas à 1 lorsqu'on clique dessus, elles retombent à 0 dès que le clic se termine — elles se comportent comme des boutons poussoirs.



Ces verrous sont communs, et pour le reste du chapitre, on simplifiera la notation pour les représenter ainsi, sans changement de fonctionnalité, mais en faisant abstraction des détails internes :



Pour aller plus loin

Voici une vidéo qui illustre ce principe de verrou SR.



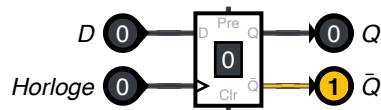
Vidéo 6: <https://www.youtube-nocookie.com/embed/KM0DdEaY5sY?start=298>

3.4.2 La bascule D

Un souci avec le verrou SR est qu'on a rarement un signal d'entrée qui soit facilement exploitable pour être «converti» en cette logique *set/reset*. La plupart du temps, on a simplement un signal donné, disons D , pour «donnée» (ou *data* en anglais), et c'est ce signal-ci qu'on aimerait stocker. Avec ce système, il serait impossible de connecter D à ce verrou ; on ne peut le brancher directement ni à l'entrée S , ni à l'entrée R .

On va utiliser pour cela un circuit similaire, mais qui fonctionne un peu différemment, qui s'appelle une **bascule D**⁴⁸ :

48. Il y a une différence conceptuelle fondamentale entre les verrous et les bascules : les verrous sont des composants dits *asynchrones*, dont l'état peut changer dès qu'une des entrées change, alors que les bascules sont des composants dits *synchrones*, qui ont une entrée appelée Horloge, et dont l'état ne changera qu'au moment où le signal d'horloge effectuera une transition (dans notre cas, passera de 0 à 1). Une discussion plus poussée de ces différences dépasse le cadre de ce cours.



Cette bascule va stocker son entrée D et la propager sur sa sortie Q uniquement lorsque l'entrée spéciale $Horloge$ passe de 0 à 1. Le reste du temps, Q et \bar{Q} garderont leur valeur précédente. Notez que cette bascule a aussi deux entrées Pre et Clr , ici déconnectées, qui servent àforcer l'état interne à valoir 1 ou 0, respectivement, indépendamment du signal D et de l'horloge.

Testez cette bascule. Réglez l'entrée de données D à 1 ou 0 et observez comme la bascule ne réagit pas : sa sortie Q reste telle quelle. Donnez ensuite une impulsion en cliquant sur l'entrée $Horloge$ et voyez comme la valeur de D est maintenant stockée sur la bascule.

Pour aller plus loin

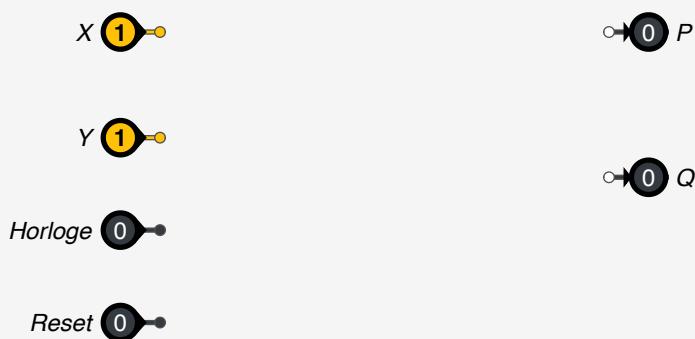
Pour aller plus loin, une vidéo de résumé qui parle aussi des bascules et des registres :



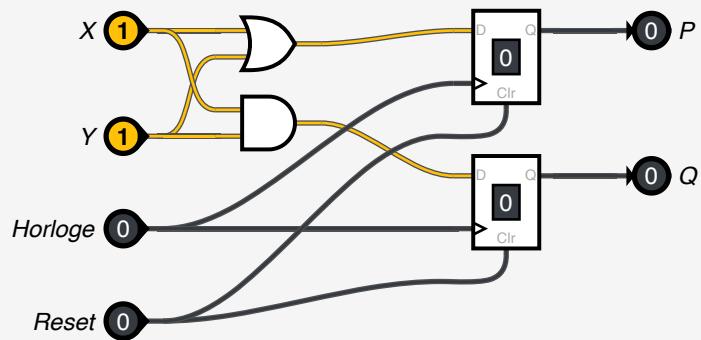
Vidéo 7: <https://www.youtube-nocookie.com/embed/I0-izyq6q5s>

Exercice 1 – Stocker deux bits

Créez un circuit qui calcule, d'une part, le **OU** de deux entrées X et Y , et, d'autre part, le **ET** de ces deux mêmes entrées. À l'aide de bascules D, complétez le circuit de manière à ce qu'il stocke ces deux valeurs calculées lors d'un coup d'horloge et les sorte sur les sorties P et Q , respectivement. Faites finalement en sorte que le signal $Reset$, si activé, réinitialise les bascules à 0. Vérifiez qu'une fois les valeurs stockées par les bascules, des changements sur les entrées X et Y n'aient pas d'effet direct sur P et Q .



Corrigé

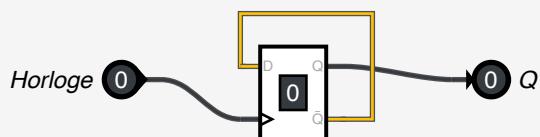
**Exercice 2 – Signal alternatif**

À l'aide d'une bascule, créez un circuit avec une sortie Q qui s'inverse à chaque coup d'horloge.

Horloge 0

0 Q

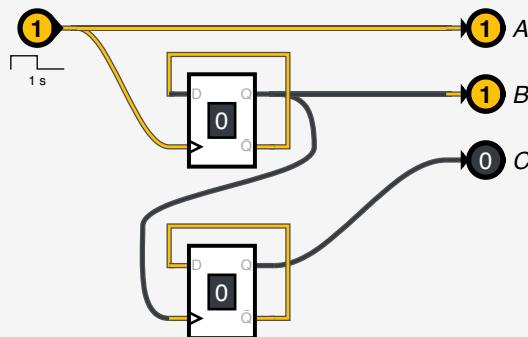
Corrigé



Exercice 3 – Jeu de fréquences

Observez le circuit ci-dessous. L'horloge principale A fonctionne ici toute seule et produit un coup d'horloge par seconde (elle a donc une fréquence d'un hertz — 1 Hz). Que pouvez-vous dire des signaux B et C par rapport au signal A ? Comment expliquer cela avec ce que vous savez des bascules? (Pour simplifier, le délai de propagation est ici presque nul.)

Vous pouvez mettre l'animation en pause et exécuter chaque transition pas à pas pour mieux comprendre ce qui se passe.



Corrigé

Le signal B a une fréquence deux fois plus petite que le signal A , et le signal C , de façon similaire, a une fréquence deux fois plus petite que le signal B . Ainsi, B «bat» à 0.5 Hz et C à 0.25 Hz.

TODO ajouter explication

Si ce petit circuit fonctionne à 1 Hz, les appareils que nous utilisons aujourd’hui ont des horloges qui fonctionnent à plusieurs gigahertz (GHz), c'est-à-dire plusieurs milliards de fois plus vite. On attend ainsi moins d'une nanoseconde entre deux coups d'horloge.

3.4.3 Addition en plusieurs étapes

Dans cet exemple final, nous allons construire un circuit capable d'effectuer l'addition de plusieurs nombres ; par exemple, d'évaluer la somme $1 + 4 + 5 + 3$ pour trouver 13.

Si ce calcul a l'air simple, il s'y cache une subtilité : nous n'avons aucun circuit auquel nous pourrions donner quatre nombres et qui en ferait la somme. Nous ne savons additionner que deux nombres à la fois. Mais nous pouvons additionner progressivement les nombres un à un à une sorte d'«accumulateur» qui stockerait les résultats intermédiaires. Au début, avant d'avoir additionné quoi que ce soit, cet accumulateur représenterait un 0. Ensuite, on y additionnerait, l'un après l'autre, chacun des nombres du calcul ainsi :

$$0 + 1 = 1$$

$$1 + 4 = 5$$

$$5 + 5 = 10$$

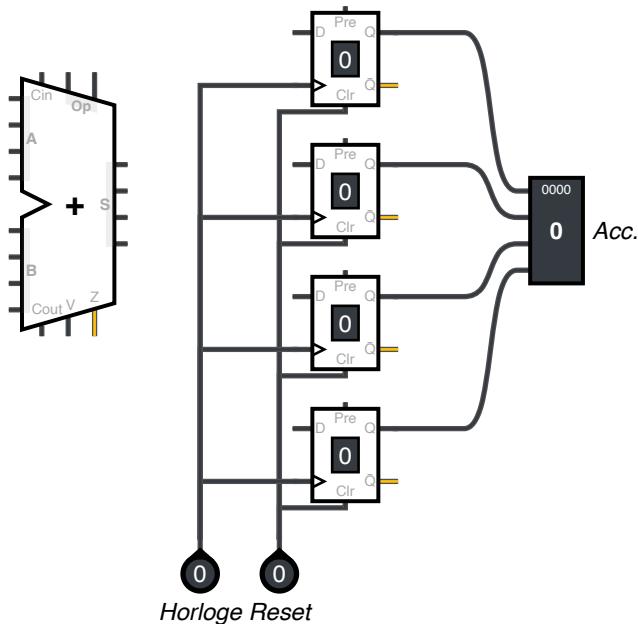
$$10 + 3 = 13$$

Chacune de ces lignes a la forme «accumulateur + nombre à additionner = nouvel accumulateur».

L'avantage de procéder ainsi, en décomposant à l'extrême, est que chaque étape est une addition de précisément deux nombres — et nous savons faire de telles additions avec une ALU.

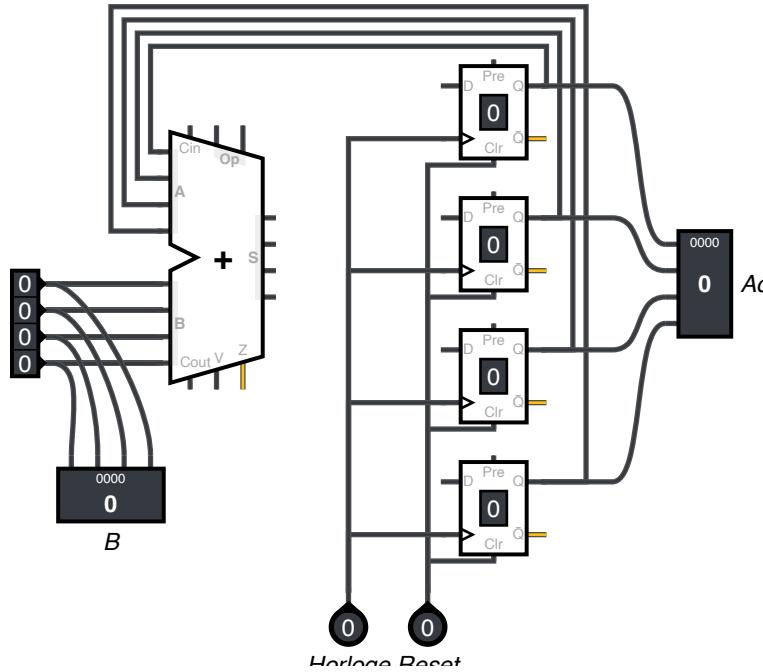
Commençons à créer un circuit capable de faire ceci. Notre ALU opérant sur des nombres de 4 bits, prenons le parti de représenter notre accumulateur via également 4 bits — 4 cellules mémoire, et donc 4 bascules. Pour remettre l'accumulateur à zéro, nous allons connecter un signal unique au *reset* de chacune de ces bascules. Nous allons aussi, comme chaque fois, connecter un signal d'horloge aux bascules, pour leur indiquer leur moment où elles doivent stocker les valeurs qui sont sur leurs entrées respectives. Ajoutons aussi une ALU pour effectuer l'addition et un afficheur décimal pour les 4 bits stockés dans les bascules.

Cela nous donne ce début de circuit, qui pour l'instant n'est pas fonctionnel :



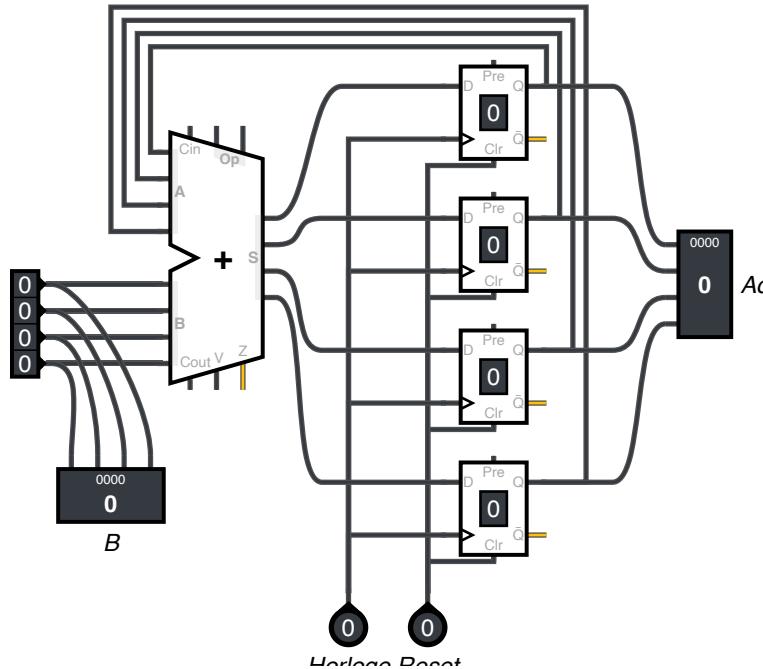
Connectons maintenant les entrées de l'ALU. On se rappelle qu'à chaque étape, l'ALU calculera une addition de la forme «accumulateur + nombre à additionner = nouvel accumulateur». L'entrée *A* de l'ALU est la valeur de l'accumulateur, donc ce qui est stocké par nos bascules. On connecte donc la sortie *Q* de chaque bascule vers le bit d'entrée *A* correspondant de l'ALU.

L'entrée *B* de l'ALU est le nouveau nombre à additionner. Pour cela, nous ajoutons simplement quatre entrées normales, ainsi qu'un afficheur décimal pour nous simplifier la lecture du nombre représenté par ces entrées :



Il reste à connecter la sortie S de l'ALU. Cette sortie nous livre la prochaine valeur à stocker dans l'accumulateur, et nous pouvons ainsi la connecter aux quatre entrées D des bascules.

Voici le circuit final :



Ce circuit fonctionne ainsi : au début du calcul, on réinitialise les bascules à zéro avec le signal *Reset*. Ensuite, on compose le prochain nombre à additionner sur l'entrée **B**. L'ALU va calculer immédiatement la somme $A + B$, mais ce n'est qu'au prochain coup d'horloge que cette somme sera stockée dans les bascules et apparaîtra ainsi à droite. Après avoir donné ce coup d'horloge, donc, on pourra à nouveau composer sur l'entrée **B** le prochain nombre à additionner, et ainsi de suite.

On réalise ici l'importance du coup d'horloge : si les bascules stockaient immédiatement la valeur livrée par l'ALU sans attendre le coup d'horloge, on retrouverait presque sans délai cette valeur sur la sortie des bascules et donc... à l'entrée A de l'ALU, qui recalculerait immédiatement la somme de cette valeur et de l'entrée B , livrerait le résultat sur la sortie vers les bascules, qui feraient à nouveau la propagation immédiate de ceci sur leurs sorties et sur l'entrée A de l'ALU, etc. — le système s'emballerait. Le signal d'horloge veille à ce que l'opération de stockage et de propagation soit coordonnée et se passe au bon moment.

Exercice 4 – Additions avec bascules

Suivez la procédure décrite ci-dessus pour effectuer l'addition $1 + 4 + 5 + 3 = 13$.

3.4.4 Récapitulatif

Au cours des quatre chapitres précédents, nous avons vu comment les portes logiques sont utilisées comme composants de base des ordinateurs. Nous avons d'abord exploré des portes simples comme **OU** et **ET**, puis montré comment ces portes peuvent être combinées en systèmes logiques plus complexes.

Avec des portes, nous avons construit un additionneur de deux bits. Nous avons ensuite été à même, en enchaînant plusieurs additionneurs, de créer un système qui peut additionner non pas simplement deux bits, mais deux nombres entiers codés sur 4 bits chacun.

Nous avons ensuite découvert l'unité arithmétique et logique, capable de réaliser plusieurs opérations différentes avec ses entrées en fonction de bits supplémentaires qui permettent de sélectionner l'opération à effectuer.

Notre dernière étape d'exploration des systèmes logiques nous a menés aux verrous et aux bascules, des composants pensés pour stocker des bits de données et ainsi constituer des cellules de mémoire pour l'ordinateur. Nous avons enfin été capables, avec une ALU et une série de bascules, d'additionner à la chaîne plusieurs nombres, en nous rappelant les résultats des additions intermédiaires.

Il existe bien d'autres éléments qui composent les ordinateurs et nous n'avons pas l'occasion de tous les explorer en détail. Dans la section qui suit, faisons un saut conceptuel et parlons de l'architecture générale des ordinateurs et de la manière dont les grands composants sont interconnectés pour permettre à un ordinateur de remplir les fonctions que nous lui connaissons.

Jeu pour aller plus loin

Dans le jeu en ligne «Nandgame» (<https://nandgame.com>), on construit petit à petit un ordinateur complet juste avec, à la base, des portes **NON-ET**. Elles ont la particularité (avec les portes **NON-OU**, d'ailleurs) de pouvoir simuler toutes les autres portes — y compris un inverseur.

3.5 CPU

Le processeur, en anglais central processing unit (CPU), est un composant qui exécute les instructions machine des programmes informatiques.

On a précédemment détaillé les différents composants et systèmes logiques à partir desquels on peut construire un processeur. On va à présent évoquer l'architecture de von Neumann qui décrit la façon dont le processeur s'insère dans son environnement. Les différents éléments qui constituent le processeur et qui en assurent le bon fonctionnement seront ensuite détaillés.



Gordon Moore

— Naissance 1929 / San Francisco

Bio

Gordon Earle Moore est le cofondateur d'Intel en 1968. Intel est le premier fabricant mondial de microprocesseurs. Gordon Moore est célèbre pour avoir formulé en 1965 une loi empirique portant son nom : **loi de Moore**. Cette loi prédit un doublement de la complexité, et donc du nombre de transistors présents dans les microprocesseurs tous les deux ans. Bien que nous ayons atteint certaines limites physiques au niveau atomique et des effets de bruits parasites liés aux effets quantiques et à la désintégration alpha, la loi est toujours vérifiée aujourd'hui malgré un ralentissement de la progression pour certaines caractéristiques. Ces limites sont aujourd'hui compensées par des puces intégrant de plus en plus de composants de plus en plus complexes.

(micro)-processeur

Le processeur est l'unité de traitement central de l'ordinateur. Il est construit avec des circuits regroupés en systèmes qui produisent des fonctions logiques et arithmétiques en suivant un programme et en utilisant des éléments de mémoire appelés registres. Un microprocesseur est un processeur construit avec un circuit intégré, c'est-à-dire un dispositif qui tient sur quelques cm². Il n'y a donc que la taille qui fasse la différence.

3.5.1 Horloge et accès mémoire

Un processeur est un dispositif synchrone, ce qui signifie que les opérations à l'intérieur du processeur se déroulent de manière synchrone à un temps donné. Pour assurer cette simultanéité, il faut comme pour un orchestre, donner le tempo. Cette fonction de métronome est assurée par une horloge, ou un signal d'horloge. Cette horloge est constituée d'un simple signal carré dont la fréquence atteint aujourd'hui plusieurs gigahertz, c'est-à-dire plusieurs milliards de cycles par seconde.

La notion de *synchronisation*

La notion de synchronisation est centrale. Les systèmes numériques synchrones sont ceux dont les opérations (instructions, calculs, logique, etc.) sont coordonnées par un ou plusieurs signaux d'horloge centralisés, par opposition, aux systèmes asynchrones qui n'ont pas d'horloge globale. Les systèmes asynchrones ne dépendent pas d'heures strictes d'arrivée des signaux ou des messages pour un fonctionnement fiable. La coordination est obtenue via des tests sur l'arrivée des évènements.

Sans entrer dans les détails ici, on notera que dans un système synchrone, il est possible d'assurer une coordination et une cohérence des opérations, ce qui est impossible autrement. Cet aspect devient crucial dans les systèmes distribués qui ne disposent plus de la garantie de synchronisation.

Les circuits asynchrones ont été envisagés comme une alternative possible aux circuits synchrones, plus répandus, particulièrement pour diminuer la consommation d'énergie, augmenter la vitesse, faciliter la conception et augmenter la fiabilité. Il semblerait qu'après avoir été un peu délaissée par le monde de la recherche et du développement depuis les années 1990 et 2000, la thématique des circuits asynchrones suscite à nouveau un regain d'intérêt, en particulier relativement aux impératifs de faible consommation énergétique en relation avec le changement climatique.

L'accès à la mémoire

Rappel

Comme on l'a vu dans l'architecture de von Neumann, la mémoire contient le programme et les données du programme. Un programme peut donc théoriquement se modifier lui-même en se modifiant dans la mémoire, même si, en pratique, toutes les architectures modernes l'interdisent (c'est rarement un effet souhaité !).

L'UCT doit accéder à la mémoire RAM en lecture ou en écriture. Les deux mécanismes sont très similaires, mais avant de regarder plus en détail comment cela fonctionne, il faut d'abord définir comment la mémoire est structurée. La mémoire RAM permet, comme on l'a vu au premier chapitre, d'accéder à tout moment à n'importe quel emplacement.

Pour y accéder, le processeur envoie d'abord l'adresse au module mémoire, puis lit ou écrit la valeur via le bus d'adressage.

Anecdote

Le processeur Intel 80286 (ancêtre des processeurs Pentium), sorti en 1982, présentait un bus de données de 16 bits et un bus d'adresses de 24 bits. De plus l'adressage par segments (relativement compliqué) réduisait l'adressage physique à un adressage sur 20 bits.

Il manque encore un élément : lorsque la mémoire voit une adresse apparaître elle doit pouvoir déterminer s'il s'agit d'une lecture ou d'une écriture. Pour cela deux connexions supplémentaires relient le processeur à la mémoire : une ligne *enable* et une ligne *set*. Lorsque la ligne *enable* est à 1, alors le processeur accède à la mémoire en lecture et sur le bus de données doit apparaître les données qui sont stockées dans la mémoire à l'adresse indiquée sur le bus d'adressage. Lorsque c'est la ligne *set* qui est à 1, alors la mémoire doit enregistrer les données à l'adresse indiquée.

Le contenu de la mémoire

Les instructions : la mémoire contient le programme sous forme de codes qui représentent des instructions à exécuter par le processeur. Ces codes correspondent à un jeu d'instructions propre à chaque modèle de processeur. On parle de langage machine. Pour écrire de tels programmes, on utilise un programme et un langage assembleur, proche de la machine : c'est une représentation exacte du langage machine, mais qui est une interface plus lisible dans la communication machine. C'est le langage de plus bas niveau qui représente le langage machine sous une forme lisible par un humain.

Les données : les données stockées dans la mémoire peuvent être des nombres, des lettres, des chaînes de caractères ainsi que des adresses d'autres emplacements en mémoire. On trouvera plus de détails à ce sujet dans le chapitre *représentation de l'information*.

Question 1

Avec un bus d'adressage de 24 bits, quelle est la taille maximum de la mémoire ?

1. 32ko
2. 16Mo
3. 16Go

Réponse: 2

Question 2

Quelle est la taille maximale de la mémoire pour un processeur 80286, sachant que l'adressage physique est finalement réduit à 20 bits ?

1. 32ko
2. 16Mo
3. 1Mo

Réponse: 3

L'unité de contrôle

L'unité de contrôle reçoit les instructions en provenance de la RAM. Elle s'occupe d'activer les composants qui doivent l'être dans le microprocesseur.

Les registres

Les registres permettent de stocker des valeurs, comme la RAM, mais directement à l'intérieur du processeur. Ils fonctionnent aussi en mode lecture ou écriture. C'est l'unité de contrôle qui détermine si un registre est utilisé en lecture ou en écriture avec deux fils de connexion : *enable* et *set*. En principe ces registres stockent les informations en provenance de la mémoire ou le résultat d'un calcul. Il existe trois registres plus spécifiques :

Le registre d'état

Le registre d'état regroupe les drapeaux (en anglais *flags*). Ils servent à renseigner l'état d'exécution du processeur. Par exemple le drapeau *dépassemement* s'il est mis à 1 signale qu'un dépassement de capacité est survenu, ou encore le drapeau *division par zéro* signale une division par zéro.

Le compteur de programme

Le compteur de programme (registre **PC** pour *Program Counter*) contient l'adresse mémoire de la prochaine instruction devant être exécutée. En principe l'unité de contrôle l'incrémente de un après chaque instruction, mais certaines instructions qui permettent de se *brancher* ailleurs dans le programme modifient différemment ce registre.

Le compteur de pile

Le compteur de pile (registre **SP** pour *Stack Pointer*) contient la position sur une pile. Cette dernière est une zone mémoire à laquelle on ne peut pas accéder aléatoirement, mais uniquement en empilant ou dépilant des éléments.

3.5.2 L'unité arithmétique et logique

L'unité arithmétique et logique (UAL plus communément appelée ALU en abréviation anglaise) effectue tous les calculs arithmétiques et logiques. Quelques-uns de ces composants comme l'additionneur ont été abordés dans le chapitre *De la logique à l'arithmétique*.

Exemple : le 6502

Le 6502, conçu en 1975, est le premier microprocesseur grand public avec un prix de 25\$ (bien en-dessous des concurrents de cette époque). Une de ses premières utilisations pour le grand public fut la console de jeux vidéo Atari 2600. A partir de 1985, Nintendo équipe la NES d'une version modifiée du 6502. Il a équipé également le célèbre Apple II. Il a donné lieu à de nombreuses versions, jusqu'aux processeurs 16 bits actuels de dernière génération.

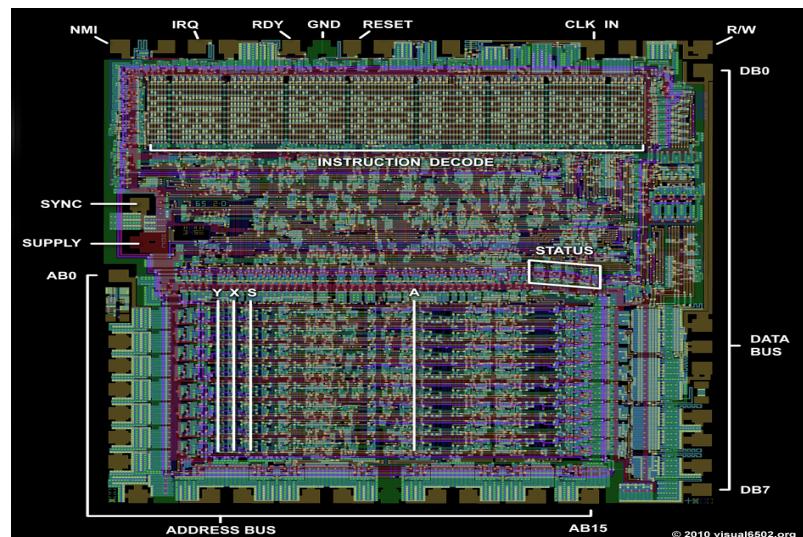


FIG. 3.7 – Ce schéma détaille l'ensemble des transistors du 6502. On voit également quelques-uns des éléments principaux (horloge, registres, etc)

Activité

Simulateur visuel du 6502⁴⁹

Ce simulateur reproduit le fonctionnement complet du 6502 jusque dans l'activité de chaque transistor. On peut clairement visualiser la façon dont la complexité du fonctionnement de ce qu'on appelle communément le *cerveau* de l'ordinateur émerge de la quantité de dispositifs triviaux pris individuellement.

1. Observer le déroulement du programme proposé et tenter d'en déduire le fonctionnement. On pourra s'aider du désassemblateur proposé sur la même page.

Question

Que fait le programme en exemple sur le site visual6502 ? Il parcourt la mémoire et recopie la valeur 40 à des adresses successives. Il effectue une boucle et incrémenté une valeur en mémoire à l'adresse FF. Il additionne deux registres et stocke le résultat dans un autre registre.

Réponse: 2

2. Modifier ou écrire un nouveau programme en allant sur la page *Advanced*.

49. <http://visual6502.org/JSSim/index.html>

Aller plus loin

La partie qui suit présente de manière plus approfondie certaines spécificités des processeurs modernes.

3.5.3 Processeur à noyau unique

C'est le processeur standard : un processeur à noyau unique ou CPU utilise un seul noyau à l'intérieur du processeur.

Avantages :

Un processeur à un seul cœur consomme moins d'énergie que les processeurs à plusieurs coeurs. Ceci est surtout problématique pour les appareils mobiles, où le problème de l'autonomie de la batterie est essentiel. Comme les processeurs à cœur unique consomment moins d'énergie, l'ensemble du système qu'ils font fonctionner chauffe moins. Un processeur à un seul cœur est toujours adapté pour la plupart des applications : vérification du courrier, navigation sur Internet, téléchargement de données, etc. peuvent toujours être traitées par un processeur à noyau unique.

Inconvénients :

C'est un processeur relativement lent. Il n'a pas une grande puissance de calcul pour traiter de grandes opérations complexes, ou plusieurs opérations à la fois. Comme les applications modernes nécessitent une grande puissance de traitement, un processeur monocœur qui les fait fonctionner peut se bloquer, paralysant ainsi l'ensemble du système alors «planté».

3.5.4 Processeur à double cœur

Un processeur à double cœur possède deux coeurs pour exécuter les opérations, intégrés dans un circuit unique pour se comporter comme une seule unité - un seul processeur -, à la différence d'un système multiprocesseur ; toutefois, ces coeurs possèdent leurs propres contrôleurs et caches, ce qui leur permet de travailler plus rapidement que les processeurs à cœur unique.

Avantages :

Un processeur double cœur exécute l'ensemble des tâches beaucoup plus rapidement. Si un processeur à noyau unique est chargé de deux tâches différentes, il ne peut pas les effectuer simultanément. Il passe à toutes les tâches une par une, en série, alors qu'un processeur à double cœur peut effectuer les deux opérations en même temps, en parallèle. Un processeur double cœur «équivaut» à deux ordinateurs en un... mais à un tarif moindre.

Inconvénients :

Peu d'opérations nécessitent réellement la puissance des processeurs double cœur. Une grande partie de la puissance est ainsi gaspillée et vide rapidement la batterie. Un appareil mobile utilisé à des fins informatiques générales, telles que la vérification du courrier électronique, la navigation sur Internet, la saisie de documents et le partage de données, ne nécessite pas réellement la puissance d'un processeur

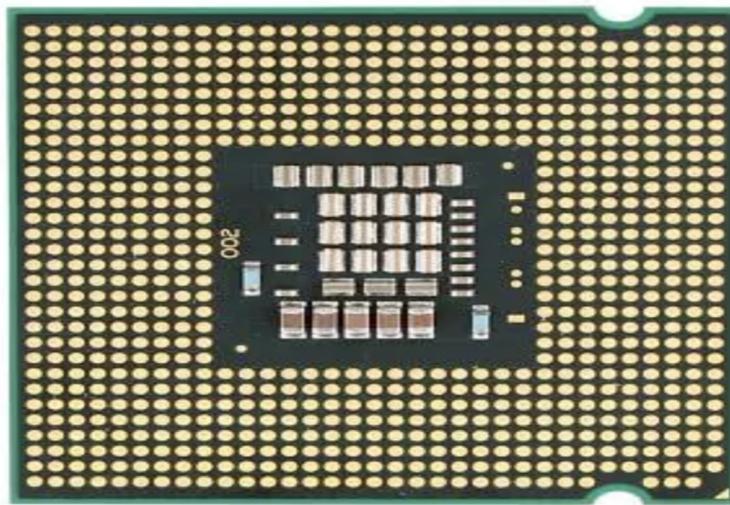


FIG. 3.8 – Microprocesseur bicoeur

double cœur. Pour ces raisons, de nombreux développeurs d'applications mobiles ne programment pas leurs applications pour qu'elles fonctionnent avec des processeurs à multiple cœur, les rendant ainsi incompatibles avec les mobiles qui fonctionnent toujours avec des processeurs à double ou multiple cœur.

3.5.5 Les processeurs quadricœur et autres processeurs à cœurs multiples

En termes simples, un processeur quadricœur possède quatre cœurs et il en va de même pour un processeur hexacœur (six cœurs), octocœur (huit cœurs), etc... Ces cœurs peuvent être soit sur le même circuit intégré, soit sur le même boîtier de puce.



FIG. 3.9 – Microprocesseur quadricœur

Avantages :

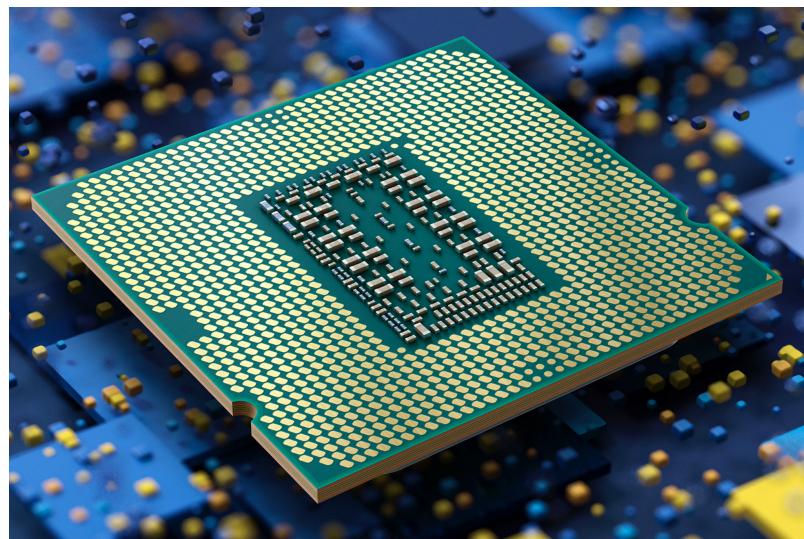


FIG. 3.10 – Microprocesseur octocœur

Le multitâche est le principal avantage des processeurs quadri ou octocœurs. Un plus grand nombre de cœurs offre évidemment une plus grande capacité à effectuer plusieurs tâches en parallèle. Ces processeurs sont utiles pour exécuter des applications qui sont plutôt intensives et nécessitent beaucoup de ressources. Ces applications comprennent les éditeurs vidéo, les antivirus, les programmes graphiques, etc. Les nouveaux processeurs quadricœurs consomment moins d'énergie, dégagent moins de chaleur, et sont donc très efficaces. Ces processeurs sont en fait très en avance sur la technologie actuelle de développement d'applications mobiles, car tous les développeurs ne sont pas capables de programmer des applications fonctionnant sur ces processeurs multiples. De nombreux programmes sont encore écrits pour des processeurs à double ou simple cœur.

Inconvénients :

... encore et toujours la consommation énergétique, vidant très rapidement la batterie.

Le nombre de cœurs de processeur est important dans certaines activités comme le *gaming* : il est de plus en plus courant de trouver des processeurs hexa-cœurs, ou octo-cœurs ; les dernières générations de multiprocesseurs possèdent jusqu'à 12 ou 16 cœurs⁵⁰ !

On doit également mentionner les cœurs logiques, c'est-à-dire les *threads*, comme on les appelle plus communément (tâches en français). La performance d'un monoprocesseur est jugée sur sa capacité à gérer plusieurs «fils» d'instructions. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.

Les technologies d'hyperthreading d'Intel et de multithreading d'AMD permettent à un seul cœur physique de gérer deux tâches simultanément, fonctionnant ainsi comme deux cœurs logiques distincts. Cette technologie est aujourd'hui très performante. La plupart de la gamme Ryzen d'AMD propose le multithreading, y compris les modèles de milieu et de haut de gamme, tandis que l'hyperthreading est pour l'instant réservé aux processeurs i7 et i9 haut de gamme d'Intel.

50. <https://www.futura-sciences.com/tech/comparatifs/meilleur-processeur-comparatif>

3.5.6 Le pipeline

On l'a vu, l'exécution d'une instruction par le microprocesseur implique plusieurs opérations : accès à la mémoire en lecture et en écriture, accès aux registres en lecture et en écriture, opération logique. Pour optimiser la vitesse d'exécution, les processeurs modernes effectuent en série ces opérations. Ainsi, alors que les opérations logiques d'une instruction sont effectuées, l'instruction précédente est déjà chargée en mémoire. La difficulté de ce type d'optimisation réside dans le fait que des branchements conditionnels provoquent l'annulation des instructions déjà chargées. Pour optimiser encore ce genre de procédé, les processeurs font de la prédition dans l'exécution. Ces optimisations sont extrêmement compliquées à gérer.

Anecdote

La vulnérabilité Spectre (ainsi que d'autres vulnérabilités similaires) exploite justement cette fonction de prédition dans l'exécution de branchements conditionnels pour accéder à des emplacements mémoire auxquels le programme ne devrait en principe pas accéder.

Matière à réfléchir. Vite... très vite

Nous avons démontré que finalement nos ordinateurs ont un cerveau très simple dans le fonctionnement de ses éléments de base : des portes logiques qui traitent des **0** ou des **1**. Il est cependant très difficile de se représenter à quel point ces traitements vont vite. Imaginons pour cela que le processeur écrive toutes les opérations qu'il effectue sur un ruban de papier et calculons la vitesse de défilement de ce papier.

Pour cela, nous faisons les hypothèses suivantes :

- Les processeurs actuels ont une cadence d'horloge de 3 GHz, c'est à dire $3 \cdot 10^9 [s^{-1}]$. Pour simplifier, nous allons supposer qu'ils effectuent une opération par cycle¹.
- Nous transcrivons un mot de 64 bits (taille standard pour les processeurs actuels) sur une longueur de 15 cm, ce qui correspond à $15 \cdot 10^{-2} [m]$.

Le calcul devient alors :

$$\begin{aligned} 3 \cdot 10^9 [s^{-1}] &\times 15 \cdot 10^{-2} [m] \\ &45 \cdot 10^7 [m/s] \end{aligned}$$

Que nous convertissons en km :

$$45 \cdot 10^5 [km/s] \text{ ou encore : } 450'000 [km/s]$$

Rappelons que la vitesse de la lumière est :

$$c \cong 300'000 [km/s]$$

Ce qui veut dire que si un microprocesseur, tel que ceux que l'on trouve dans notre ordinateur ou notre smartphone, écrivait sur un ruban de papier tout ce qu'il fait, ce ruban de papier devrait se déplacer à une fois et demi la vitesse de la lumière. Ou encore, ce ruban ferait chaque seconde plus de 11 fois le tour de la terre.

1. En fait les opérations d'un processeur prennent plus d'un cycle pour être réalisées, mais comme les processeurs ont plusieurs coeurs et un pipeline dont nous n'abordons pas ici le fonctionnement, la simplification proposée n'est pas aberrante.

Si les éléments de base sont simples, la complexité et la richesse des expériences numériques comme l'immersion dans un jeu vidéo proviennent de la quantité extraordinaire d'opérations effectuées.

3.6 Architecture générale

Il est commun d'entendre parler du microprocesseur comme du «cœur de l'ordinateur». On se propose de dégager les caractéristiques essentielles de ce qui constitue cette machine «intelligente» appelée ordinateur, tout en explicitant les composants informatiques spécifiques (le matériel ou «hardware»).

Si l'on suit l'évolution de l'ordinateur, depuis les années 50 jusqu'à aujourd'hui, on peut distinguer les éléments caractéristiques illustrés sur la figure suivante.

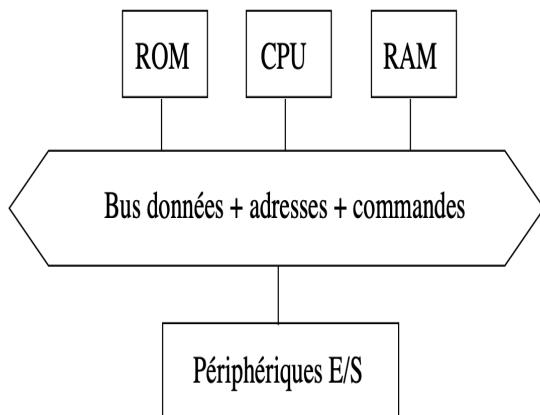


FIG. 3.11 – Schéma simplifié d'un ordinateur

D'un point de vue matériel, on distingue :

- l'alimentation,
- la carte mère,
- le processeur,
- la mémoire vive RAM,
- le disque dur / SSD / eMMC,
- le lecteur-graveur,
- la carte graphique.

On peut également citer les cartes sons, réseau, sorties USB etc. Ce type de matériel n'étant pas indispensable au bon fonctionnement du PC et souvent directement intégré à la carte mère, on ne s'y intéressera pas ici. On distingue ce matériel, partie intégrante de la machine, avec les périphériques externes qui lui sont reliés par des câbles ou des moyens de communication sans fil.

3.6.1 La mémoire

ROM (Read-Only Memory) : ce que l'on nomme ROM constitue une mémoire «fixe», statique de la machine, dont la taille est définie à la conception. On parle de mémoire morte, ou mémoire en lecture seule. Ce qu'elle stocke ne «part pas» lors de la mise hors tension de la machine.

Cette mémoire fixe va intégrer tous les éléments nécessaires en particulier au démarrage de la machine, au lancement du système d'exploitation ; il en est de même en ce qui concerne les facteurs de conversion, tables de constantes, instructions propres de la machine. On distingue différents types de ROM :

- ROM standard,
- EPROM : ce type de mémoire rend programmation et effacement accessibles à l'utilisateur,
- PROM : une programmation unique est possible sur ce type de mémoire,
- EEPROM : c'est une mémoire programmable dont les données peuvent être effacées électriquement (succède à l'UVEPROM dont les données pouvaient être supprimées dans une «chambre à UV»).

Il s'agit d'une mémoire à long terme.

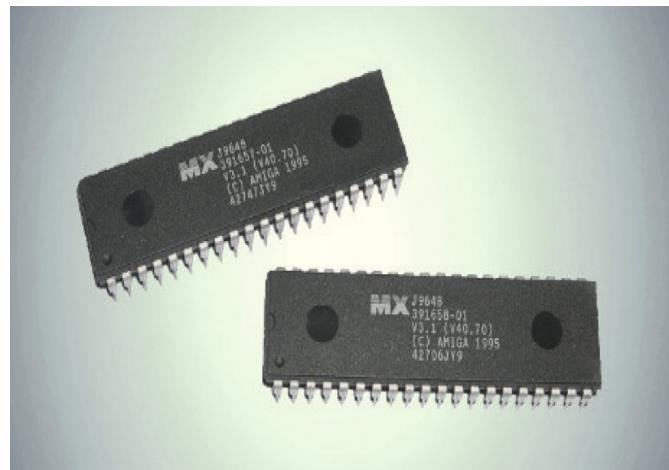


FIG. 3.12 – Barrettes ROM

RAM (Random Access Memory) : cette mémoire est une mémoire volatile, c'est à dire que son contenu va «disparaître» lorsque l'ordinateur est hors tension. On parle aussi de mémoire tampon. L'information étant stockée sous forme électrique dans les transistors, elle disparaît quand l'alimentation est coupée.

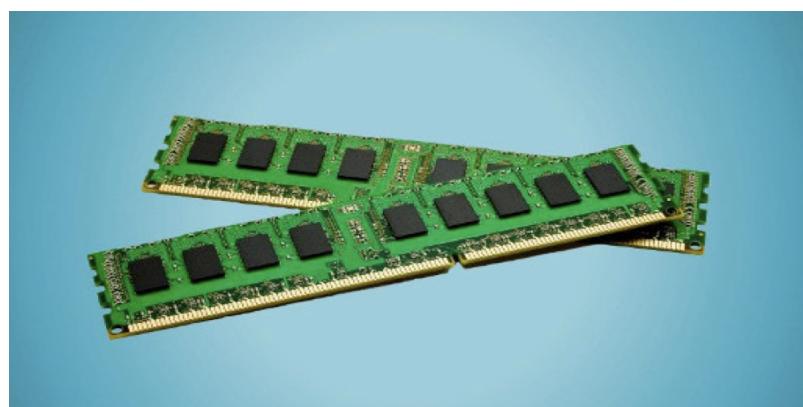


FIG. 3.13 – Barettes RAM

Cette mémoire stocke temporairement et aléatoirement les fichiers sur lequel on travaille : on parle de mémoire vive en français. Elle est accessible en lecture / écriture. Ce type de composants se présente sous la forme de «barrettes» amovibles faciles à remplacer. Les temps d'accès sont très courts (de l'ordre de la nanoseconde), et la capacité de stockage élevée en comparaison avec celle de la ROM. Elle est la plus coûteuse des deux. On distingue deux types de RAM :

- la RAM statique nécessite un flux constant d'énergie pour conserver les données qu'elle contient,
- la RAM dynamique (DRAM, SDRAM) : elle doit être actualisée pour conserver les données qu'elle contient ; elle est plus lente et moins chère que la RAM statique.

Les composants de mémoire RAM existent en général en «barrettes» allant de 1 à 8 Go par unité (les configurations les plus courantes actuellement proposent 4 à 8 Go de RAM), elles sont à choisir en fonction du processeur et de l'utilisation que l'on fait du PC d'une part et des possibilités de la carte mère (capacité totale, nombre d'emplacements disponibles...) d'autre part.

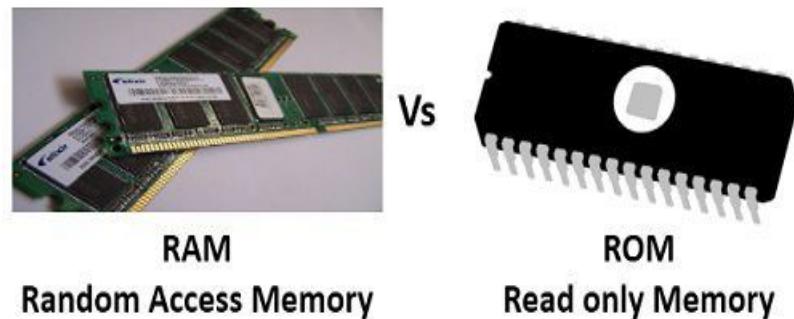


FIG. 3.14 – RAM versus ROM

Une analogie intéressante pour comprendre les particularités et missions respectives de la ROM et de la RAM est celle du commerçant et de la gestion quotidienne de sa caisse. Dans sa journée de travail, le commerçant encaisse, rend de la monnaie, gère donc des flux monétaires fluides et dimensionnés : dans le cas d'une boulangerie par exemple, les flux moyens vont être de l'ordre de quelques francs. En fin de journée, le commerçant compte sa caisse et part déposer la recette du jour à la banque : cela peut représenter alors plusieurs milliers de francs. La caisse représente en quelque sorte la RAM, accessible facilement et rapidement, avec des flux modérés et relativement réguliers ; en revanche sa taille est limitée. Elle est vide le matin, se charge et décharge dans la journée, puis se vide en fin de journée à la clôture. La banque représente la ROM : le stockage de l'argent prend plus de temps, mais l'espace de stockage est beaucoup plus vaste et est sécurisé.

Mémoire externe ou mémoire de masse

Ce terme désigne les supports externes à la machine permettant du stockage supplémentaire et donc venant en complément du stockage initial fixe de la machine (ROM). Par exemple : disque dur interne ou externe, bande magnétique, SSD, disque optique, clé usb, carte SD, ... Cette mémoire permet de stocker une grande quantité d'informations, mais à une vitesse limitée. C'est un peu l'intermédiaire entre la RAM et la ROM.

Sur le disque dur peuvent être enregistrées les données à conserver : les fichiers du système d'exploitation, les logiciels et surtout les données personnelles (photo, vidéo, musique, emails etc...).

Le disque dur se présente sous la forme d'un boîtier rectangulaire, vissé au boîtier du PC, qui intègre toute la mécanique (plateau, bras, tête de lecture...). Plus la vitesse de rotation des plateaux est importante, plus les performances sont élevées : on trouve actuellement des disques durs tournant à 5400, 7200, 10000 ou 15000 RPM (Round Per Minute : tours par minute), les vitesses de 7200 et 10000 RPM étant les plus répandues.

Il est relié à la carte mère grâce à une nappe (câble plat) de type IDE ou grâce aux interfaces SATA (Serial ATA) ou SCSI. Un cavalier à positionner à l'arrière du boîtier permet généralement de le désigner comme disque «Maître», le disque dur principal (Master) ou comme «Esclave», un disque auxiliaire (Slave).

Les disques durs aujourd’hui peuvent contenir des centaines de gigaoctets, voire plusieurs téraoctets de données.



FIG. 3.15 – Disque dur mécanique

Les ordinateurs récents sont de plus en plus équipés de SSD (Solid-State Drive) qui permettent de stocker des données tout comme les disques durs, mais leur conception est purement électronique et non plus mécanique. Ils sont donc plus résistant aux chocs et plus légers - et donc particulièrement adaptés aux ordinateurs portables - et beaucoup plus rapides. Ils ont une taille de plus en plus réduite et des gains de performance importants : temps d'accès réduits, meilleure bande passante que les disques durs traditionnels.

La fiabilité et les capacités des disques durs classiques pérennisent cependant leur utilisation.



FIG. 3.16 – Disque dur SSD

Sur certains ordinateurs portables d'entrée de gamme, le disque dur ou le SSD sont parfois remplacés par un stockage sous forme d'eMMC, solution peu coûteuse comme les cartes SD ou Multimedia Card.

L'avantage, en plus du coût, réside au niveau de l'encombrement et du poids, et convient donc aux ordinateurs portables de petite taille. Compte-tenu des performances, à prix et configuration équivalents, on privilégiera le SSD, puis l'eMMC et enfin le disque dur.

Le lecteur/graveur CD/DVD Un ordinateur peut encore aujourd’hui être équipé d’un graveur, vissé au boîtier, glissé dans un emplacement ouvert sur l’avant du PC, permettant ainsi l’ouverture du tiroir qui recevra le disque optique que l’on appelle plus communément CD (Compact Disc) ou DVD (Digital Versatile Disc). Il est connecté à la carte mère par un câble plat (nappe) IDE ou SATA.



FIG. 3.17 – Lecteur CD

3.6.2 Le CPU (Central Processing Unit)

Il s'agit du processeur de l'ordinateur. C'est le cœur de l'ordinateur, c'est à dire l'espace où va se dérouler l'ensemble des opérations et instructions de la machine. C'est un peu le «cerveau» de la machine. Le CPU va aller chercher les informations dans la ROM en passant par la RAM qui est donc essentielle pour le traitement du processeur. On parle d'Unité Centrale de Traitement en français. Le processeur sert à l'échange de données entre composants informatiques : disques durs – carte graphique – ROM – RAM. Il coordonne, interprète, calcule, exécute.

La puissance du CPU est caractérisée par son nombre de bits, 32 ou 64 bits aujourd'hui, et la fréquence de traitement de l'information qu'il assure caractérise la rapidité avec laquelle il traite les informations. Cette puissance de traitement des cycles CPU, qui est donc la puissance de l'ordinateur, représente la capacité d'un ordinateur à manipuler des données. La puissance de calcul et la rapidité de traitement se trouvent multipliées par le nombres de coeurs éventuellement présents sur la puce. Nombre de bits et fréquence de traitement sont donc deux paramètres essentiels, mais également le nombre de coeurs que le processeur comporte.

Le cœur du processeur est en fait une unité de traitement qui permet de lire des instructions pour effectuer des actions spécifiques. Ainsi, quelle que soit l'action que l'on souhaite effectuer sur la machine, elle est exécutée par le cœur, et s'il y a plusieurs coeurs, qui sont en fait des unités de traitement, on peut effectuer toutes les actions rapidement et en même temps.

Les principaux acteurs du marché sont Intel et AMD.



FIG. 3.18 – Différents types de microprocesseurs simple cœur et multicoeurs

La carte mère

Une carte mère est le composant central de l'ordinateur. Elle est vissée au boîtier du PC, et possède les connecteurs (slots) pour accueillir des dizaines de composants et périphériques en plus des éléments indispensables décrits ici, et gérer les flux logiciels, chaque information envoyée ou reçue par le matériel ou un programme transitant par elle.

Elle intègre également la ROM sur laquelle est enregistrée le BIOS, petit programme gérant la configuration «de base» du matériel et se chargeant de faire le lien avec le système d'exploitation. Ces réglages sont conservés en mémoire même en l'absence de courant grâce au **CMOS**⁵¹, alimenté par la pile de carte mère.

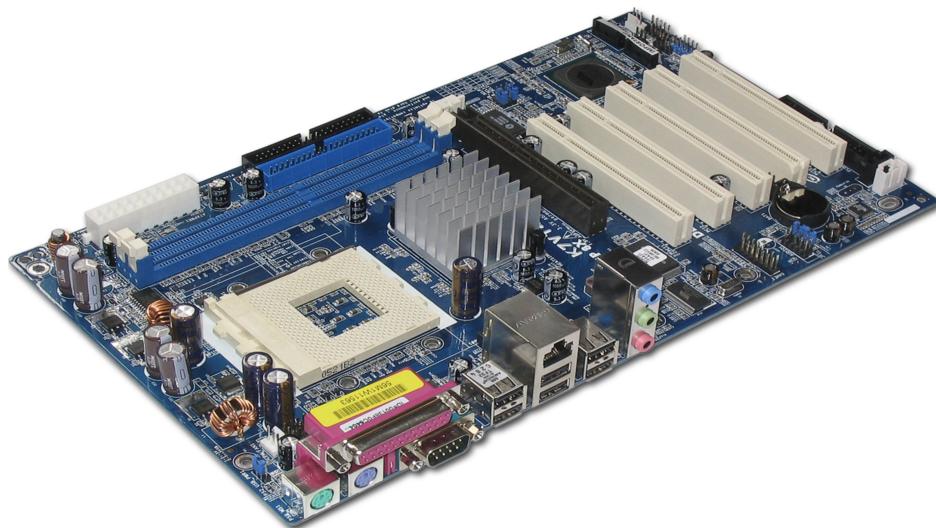


FIG. 3.19 – Carte mère

51. https://fr.wikipedia.org/wiki/Complementary_metal_oxide_semi-conductor

3.6.3 Les entrées-sorties

Un ordinateur traite de l'information au niveau de sa mémoire et de son processeur. Il récupère donc cette information via des ports d'entrée et redistribue une information après traitement via des ports de sortie. L'ensemble de cet environnement d'entrées-sorties constitue ce que l'on nomme les périphériques : clavier, écran, enceintes audio ou casque, imprimante, souris ou pad, disques externes, microphone, réseau ethernet ou wifi, etc. Certains périphériques sont par nature destinés uniquement à l'entrée de données (claviers et souris, microphones), tandis que d'autres s'occupent avant tout de la sortie (imprimantes, écrans non-tactiles) ; d'autres encore permettent à la fois l'entrée et la sortie de données (disques durs, disquettes, CD-ROM inscriptibles, clés usb).



FIG. 3.20 – Unité centrale et périphériques

Interfaçage

Dans une description idéale, le processeur se connecte au bus et envoie sur le bus adresses, données et commandes au périphérique. Ensuite, le processeur va devoir attendre et rester connecté au bus tant que le périphérique n'a pas traité sa demande intégralement en lecture ou en écriture. Mais les périphériques sont lents et le processeur attend le périphérique... Pour résoudre ce problème, on intercale des registres d'interfaçage entre le processeur et les entrées-sorties. Une fois que le processeur a écrit les informations à transmettre dans ces registres, il peut faire autre chose, le registre se chargeant de maintenir et mémoriser les informations à transmettre. Les registres d'interfaçage sont surveillés régulièrement par le processeur pour voir si un périphérique lui a envoyé une information, mais le processeur peut utiliser quelques cycles pour faire son travail en attendant que le périphérique traite intégralement sa commande. Ces registres peuvent contenir des données ou des commandes, des valeurs numériques auxquelles le périphérique répond en effectuant un ensemble d'actions préprogrammées.

Les commandes sont traitées par un contrôleur de périphérique, qui va lire les commandes envoyées par le processeur, les interpréter, et piloter le périphérique de façon à faire ce qui est demandé. Le contrôleur de périphérique génère des signaux de commande qui déclencheront une action effectuée par le périphérique. Certains contrôleurs de périphérique peuvent permettre au processeur de communiquer avec plusieurs périphériques en même temps. C'est notamment le cas pour tout ce qui est des contrôleurs PCI, USB et autres. Certains périphériques, comme les disques IDE intègrent en leur sein ce contrôleur. Certains de ces contrôleurs intègrent un registre d'état, lisible par le processeur, qui contient des informations sur l'état du périphérique. Ils servent à signaler des erreurs de configuration ou des pannes touchant un périphérique.

Le système d'exploitation d'un ordinateur ne connaît pas toujours le fonctionnement d'un périphérique ou de son contrôleur : il faut alors installer un programme qui va permettre la communication avec le périphérique, et qui va gérer transfert des données, adressage du périphérique, etc. Ce petit programme est appelé un *driver* ou pilote de périphérique.

3.6.4 Les bus

Un bus informatique est un dispositif de transmission de données partagé entre plusieurs composants d'un système informatique. Le bus informatique est la réunion des parties matérielles et immatérielles qui permet la transmission de données entre les composants de la machine. On distingue deux types de bus : le FSB (Front Side Bus), ou *bus système*, et le bus d'extension. Le premier permet au processeur de communiquer avec la mémoire vive, le second est une voie de liaison entre le processeur et les cartes d'extension. Des connecteurs d'extension présents sur la carte mère permettent d'y ajouter de nouveaux composants : cartes d'extension tels que carte son, carte d'acquisition vidéo, carte réseau, etc. Il existe différents types de bus d'extension : **ISA**, **EISA**, **PCI**, **PCMCIA**, **VESA**.⁵² On se propose ici de décrire exclusivement les différents types de bus système : bus de données, d'adressage et de commande.

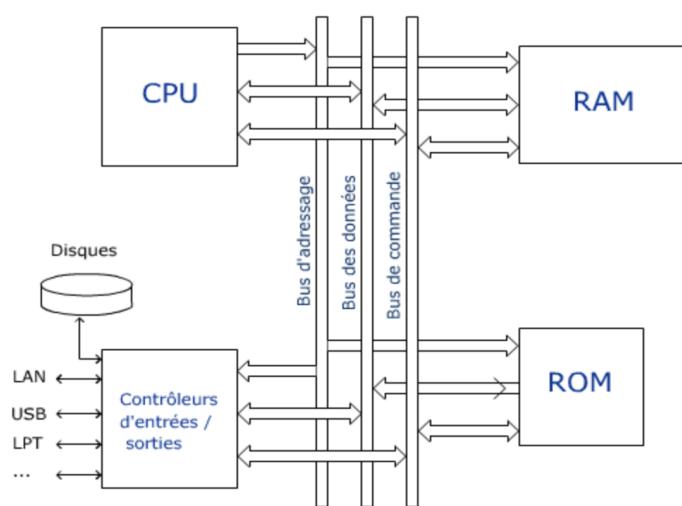


FIG. 3.21 – Schéma général d'un ordinateur

52. <http://www.dicofr.com/cgi-bin/n.pl/dicofr/definition/20010101000612>

Bus de données (Data Bus)

Le bus de données interconnecte le processeur, la mémoire centrale et les contrôleurs de périphériques et véhicule les données entre ces composants. Il est bidirectionnel (contrairement au bus d'adressage) puisque le processeur l'utilise pour lire et pour écrire en mémoire ou dans les entrées-sorties.

Le bus de données est commandé par le CPU, les autres composants y sont connectés à tour de rôle pour répondre aux commandes de lecture ou d'écriture du processeur.

Le débit des données véhiculées par ce bus dépend d'une part des vitesses de transmission ou plus exactement de la capacité des composants à saisir rapidement les signaux des bus et à y répondre aussi vite. La cadence de ces signaux est liée à fréquence de la carte mère.

La largeur du bus est le second critère qui va influencer le débit des transmissions des données. Plus le bus est large et plus important sera le nombre de données qui pourront être véhiculées simultanément. La largeur du bus de donnée peut être comparée au nombre de voies de circulation d'une autoroute. Elle dépend directement de la puissance du processeur.

NB : Les premiers microprocesseurs qui ne pouvaient traiter que 8 bits simultanément avaient un bus de données de 8 bits. Actuellement, les microprocesseurs traitent en général les données par mots de 32 bits mais le bus de donnée est plus large encore (64 bits) ce qui lui permet de véhiculer plus de données en parallèle.

Le bus d'adressage (Address Bus)

Le bus d'adressage (ou bus d'adresse, ou bus mémoire) reçoit du processeur les adresses des cellules mémoire et des entrées/sorties auxquelles il veut accéder. Chacun des conducteurs du bus d'adressage peut prendre deux états, 0 ou 1. L'adresse est donc le nombre binaire qui est véhiculé par ces lignes. La quantité d'adresses qui peuvent ainsi être formées est égale à deux exposant le nombre de bits d'adresse. Ce bus est physiquement constitué de câbles parallèles qui relient le processeur à la mémoire. La taille de ce bus ou sa largeur définissent le nombre de connexions parallèles et dépendent des caractéristiques du processeur et de la RAM. Chaque connexion transporte un bit, un bus de largeur 32 bits transporte 32 bits, ce qui permet de répertorier 2³² adresses mémoire différentes (env. 4 Go). Les deux bus, d'adressage et de données, ne sont pas forcément de largeur identique.

NB : Le processeur 8088 qui équipait des premiers PC n'avait que 20 lignes d'adresse. Il pouvait donc accéder à 2 exposant 20 adresses différentes, soit 1 Mo. C'est pour cette raison que le système d'exploitation DOS qui date de cette époque ne peut pas adresser la totalité de la mémoire des systèmes actuels. Le nombre de lignes du bus d'adresse a ensuite évolué avec les différentes générations de processeurs.

Le bus de commande (Control Bus)

Le bus de commande ou bus de contrôle transporte les ordres et les signaux de synchronisation en provenance du CPU et à destination de l'ensemble des composants matériels via un ensemble de connexions physiques telles que des câbles ou des circuits imprimés. Il s'agit d'un bus bidirectionnel qui transmet également les signaux de réponse des éléments matériels. Le CPU utilise un des signaux pour indiquer le sens des transferts sur le bus de données (lecture ou écriture). Le bus de contrôle est constitué de lignes de contrôle qui envoyent chacune un signal spécifique, tel que lecture, écriture et interruption. La plupart des microprocesseurs incluent des lignes d'horloge système, des lignes d'état et des lignes d'activation d'octets.

3.6.5 Autres composants matériels

L'alimentation

L'alimentation branchée sur le secteur transforme et fournit l'énergie nécessaire à la carte mère, mais l'alimentation est aussi directement reliée à certains composants tel que le lecteur/graveur de DVD ou le disque dur par exemple.

La transformation du courant cause une déperdition d'énergie thermique, un système de ventilation est donc installé dans le coffret de l'alimentation et expulse l'air via l'arrière du boîtier de l'ordinateur.

Une puissance de 400 watts est généralement suffisante pour les ordinateurs en «configuration bureautique» même si certaines alimentations peuvent atteindre les 1000 watts pour des configurations gourmandes en énergie (gaming par exemple).



FIG. 3.22 – Alimentation

La carte graphique

La carte graphique, bien que très importante pour certains usages, est placée en dernière position de cette liste car elle peut-être remplacée par un chipset intégré (jeu de circuit) directement à la carte mère. Toutefois, pour certaines applications et notamment les jeux, gros consommateurs de ressources graphiques, elle est indispensable. En prenant à sa charge la gestion de l'affichage, elle libère le processeur de cette fonction, traite elle-même les informations et utilise sa propre mémoire (voir accélération matérielle).

La carte graphique s'insère dans un connecteur de la carte mère : le port AGP ou le port PCI Express pour les plus récentes. Une fois connectée, les entrées et sorties de la carte sont accessibles par l'arrière du boîtier afin de fournir une image au système de visualisation (écran, TV, projecteur).



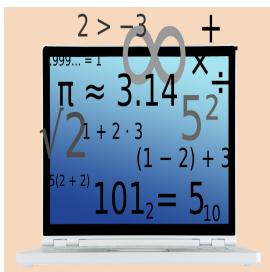
FIG. 3.23 – Carte graphique

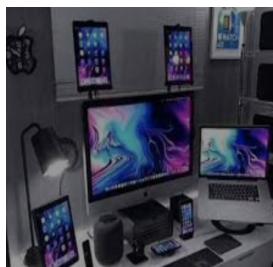
3.7 Conclusion

La large diffusion de l'ordinateur individuel dans les foyers depuis près de 40 ans a bouleversé nos sociétés, nos comportements, nos habitudes. Pour beaucoup, l'ordinateur reste cependant synonyme de complexité, et si la confrontation à la machine aujourd'hui concerne quasiment tout un chacun dans les pays développés, la perception de l'univers numérique, de l'informatique et de la machine «ordinateur» varie considérablement d'une personne à une autre. Interrogations, réactions épidermiques, peurs pour les uns, ou au contraire passion, voire dépendance pour les autres, la discipline ne laisse personne indifférent, et suscite souvent perplexité et angoisse.



Qu'est-ce-qu'un ordinateur ? Comment le définit-on ? Il serait intéressant de faire des sondages sur plusieurs types d'échantillons représentatifs de la population : par type de famille, par tranche d'âge, catégorie socio-professionnelle, géographie, urbains / non-urbains, ... Nul doute concernant l'amplitude et la variabilité des réponses si l'on demande de définir ou qualifier l'ordinateur en quelques mots : boîte, souris, écran, calcul, compliqué, modernité, clavier, jeu, travail... autant de qualificatifs reflétant à la fois la diversité des perceptions relatives à l'ordinateur et l'importance de sa diffusion.





Qu'est-ce-qu'un ordinateur ? On peut fortement douter que l'exploitation des résultats de nos sondages permette de dessiner une réponse homogène... malgré un taux de réponse positive extrêmement important s'agissant d'une prétendue connaissance de la chose !

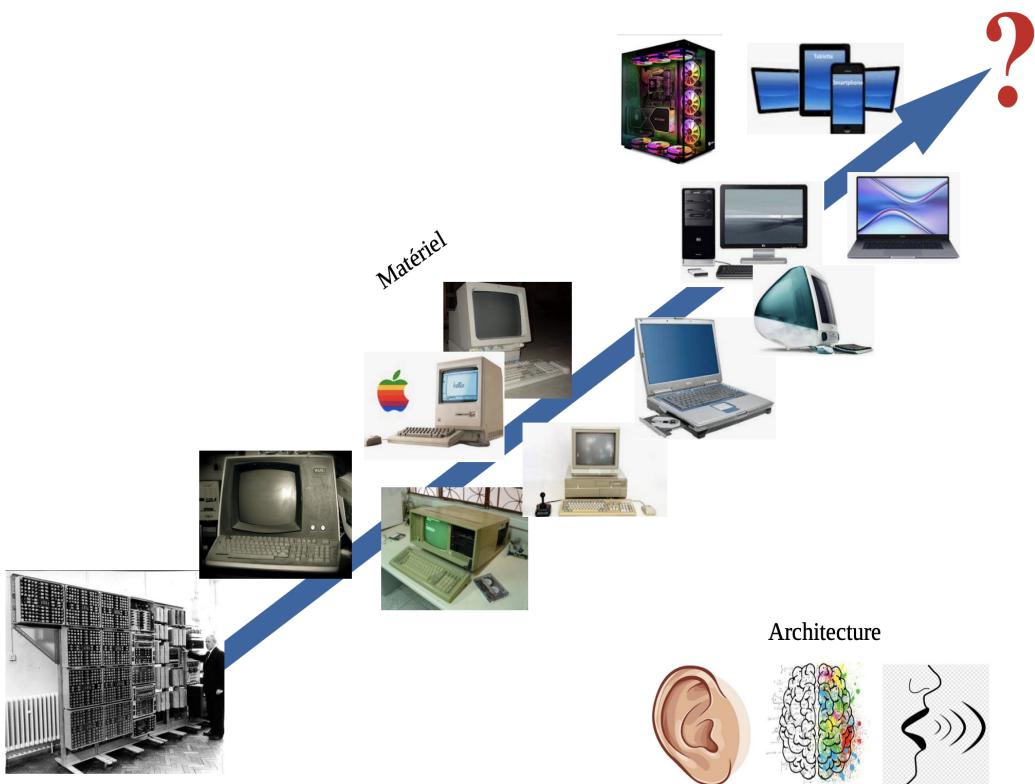
En y regardant d'un peu plus près, il n'est pourtant pas si difficile d'en trouver une définition : c'est l'association d'une **mémoire**, d'un **calculateur** réalisant des opérations / instructions et d'un **véhicule de données**, ou bus de données, transportant les informations entre ces éléments : cet ensemble communique avec l'extérieur via des zones d'**entrées-sorties**.





Entrée _____ calcul - communication - mémoire _____ sortie

Peut-on douter alors de la faiblesse de ce type de réponse dans notre sondage ? Non, à l'évidence. Il est vrai qu'on rentre ici dans un schéma de fonctionnement global, donc avec une vision peut-être un peu technique, mais qui se trouve être étonnamment stable dans le temps pour définir l'ordinateur depuis qu'il existe. Depuis plus de 70 ans, un ordinateur c'est simplement cela : mémoire - unité de calcul - bus - zone entrées sorties.



Comment, devant cette pérennité conceptuelle, expliquer la multitude des perceptions de l'objet par le grand public ? Et si l'on interroge à nouveau la population, il est en revanche fortement à parier que ressorte le fait que l'ordinateur évolue sans cesse, et très rapidement !

On en revient donc, finalement, au problème de la perception : ce que perçoit le plus grand nombre, c'est l'aspect **composants**, le matériel, le «hardware»... qui effectivement a profondément évolué depuis plus de 70 ans : de la carte perforée aux disques SSD en passant par la disquette, des tubes à vides aux microprocesseurs en passant par les transistors, des grosses unités de bureau des années 80 aux smartphones d'aujourd'hui, des écrans monochromes de 9 pouces des premiers Macintosh aux écrans couleurs 24 pouces full HD Wled d'aujourd'hui. Ce que perçoit le plus grand nombre, c'est également le bouleversement rapide et important amené par l'ordinateur, dans la vie sociale, et le travail ; encore un élément qui éloigne l'utilisateur lambda d'une visibilité claire de l'aspect assez immuable de l'ordinateur d'un point de vue conceptuel.

Pour construire un ordinateur il faut donc toujours assembler les mêmes **composants de haut niveau** : de la mémoire RAM et ROM, un CPU avec du câblage, les trois premiers étant complexes du point de vue des composants électroniques de base.

Au sein du CPU, figure une ALU qui réalise les calculs et des registres pour stocker des résultats intermédiaires.

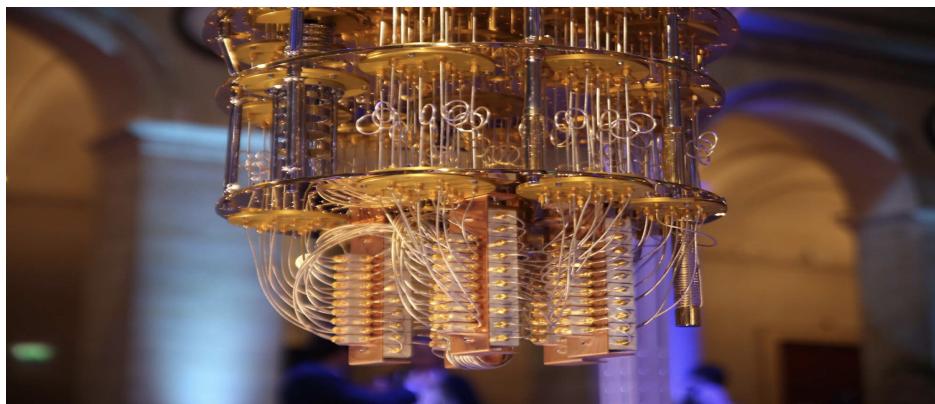
Une ALU se construit avec des portes logiques, constituées de transistors, **composants de bas niveau**.

Chaque nouvelle couche d'abstraction peut être étudiée de manière autonome et donner lieu à une réflexion propre en faisant abstraction des détails de la couche supérieure.

Quid du futur des ordinateurs ? Suivant quels axes va se définir l'amélioration de leurs performances ?

Il semblerait qu'on arrive aujourd'hui à un palier concernant les composants et la loi de Moore. La miniaturisation atteint les limites fixées par l'atome et le comportement aléatoire des électrons.

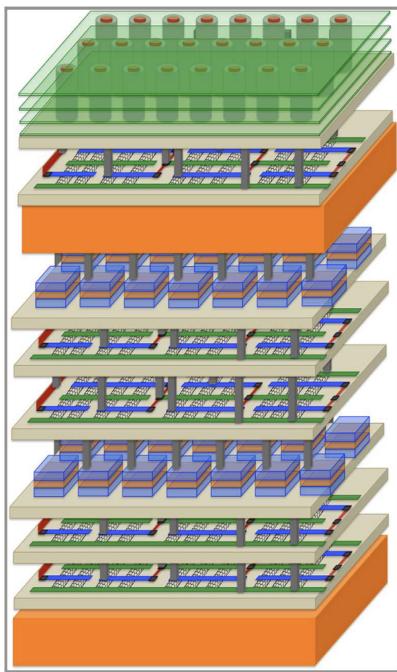
L'avenir est-il réellement vers l'[informatique quantique](#)⁵³, ou l'informatique neuromorphique ? Ou encore la spintronique, qui permet d'effectuer des calculs en renversant la rotation des électrons plutôt que de les déplacer ?



53. <https://www.numerama.com/tech/479292-la-france-se-reve-en-championne-de-l-informatique-quantique-une-nouvelle-frontiere-technologique.html>

Une étape fondamentale semble se franchir aujourd’hui avec une réflexion menée concernant la définition de cette architecture classique quaternaire de Von Neumann. Une nouvelle architecture s’inspire directement du fonctionnement du cerveau, composé de synapses jouant à la fois le rôle de mémoire et de calculateur. Il est alors question de faire jouer à la mémoire un rôle calculatoire et de fusionner les deux fonctions⁵⁴.

Cette approche, 80 ans après Von Neumann, semble apporter des gains exceptionnels en terme de performance, en ce qui concerne la rapidité et la gestion mémoire.



Nouvelle conception d’architecture hybride de type «gratte-ciel» : elle est basée sur des matériaux plus avancés que le silicium ; les unités de mémoire et les transistors à base de nanotubes de carbone sont empilés successivement

54. <https://technologiimedia.net/2018/10/04/des-chercheurs-ont-concu-une-nouvelle-architecture-informatique/>

3.8 TP Portes logiques

Dans ce chapitre, nous allons explorer les portes logiques NON, OU et ET.

- Créez chaque circuit comme demandé
- Quand le circuit est terminé, cliquez sur **Screenshot** (dans le menu à droite)
- La capture d'écran contient le code pour réouvrir le circuit dans l'éditeur Logic⁵⁵
- Déposez vos captures de circuit sur Moodle

3.8.1 Transmission d'un signal

Dans ce premier exemple se trouvent une entrée (in) et une sortie (out). Les deux sont liées par un fil de transmission qui transmet un signal binaire identifié par une couleur :

- 0 (noir)
- 1 (jaune)

Avec le menu contextuel (clic droit sur le fil), vous pouvez changer la couleur du fil ainsi que son délai de propagation.

- Ajoutez un deuxième fil avec un délai de propagation de 100 ms (rapide)
- Ajoutez un troisième fil avec un délai de propagation de 10 ms (instantané)
- Ajoutez un point intermédiaire au milieu du fil et déplacez-le un peu
- Mettez la couleur du fil en rouge



3.8.2 Commutateur/poussoir

Les entrées ont deux modes que vous pouvez changer avec le menu contextuel :

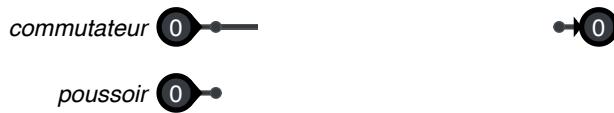
- commutateur : bascule entre l'état 0 et 1
- poussoir : garde la valeur 1 seulement pendant qu'il est appuyé

Changez la deuxième entrée en mode **poussoir** et augmentez le délai de propagation du fil à 1000 ms. Vous verrez alors des paquets d'informations se propager le long du fil.

Vous pouvez ajouter des noms aux entrées sorties.

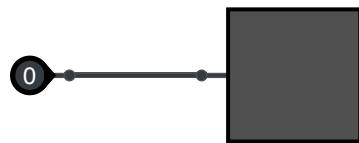
- Ajoutez une entrée *lumière*, configurez en commutateur, et reliez-la à une sortie *lampe*
- Ajoutez une entrée *sonnerie*, configurez en poussoir, et reliez-la à une sortie *alarme*

55. <https://logic.modulo-info.ch/>



3.8.3 Feu de circulation

- Cliquez sur l'entrée pour basculer l'état entre 0 et 1
- Ajoutez deux autres segments carrés pour compléter un feu de circulation.
- Changez l'affichage en grand carré
- Changez les couleurs en jaune et vert
- Ajoutez les noms *rouge, jaune, vert.*

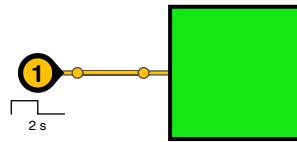


3.8.4 Lampe clignotante

L'entrée horloge (clock) produit un signal qui alterne entre 0 et 1.

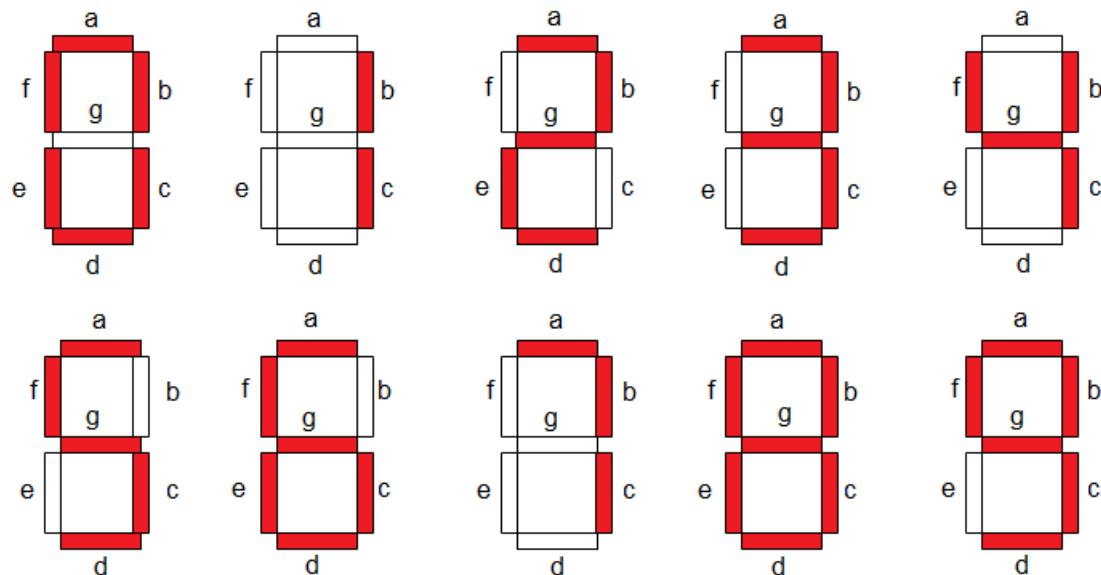
- Ajoutez une deuxième horloge
- Ajoutez une deuxième lampe (grand segment carré)
- Configurez l'horloge pour une période de 1 seconde

Avec le bouton **Pause** vous pouvez arrêter l'horloge.

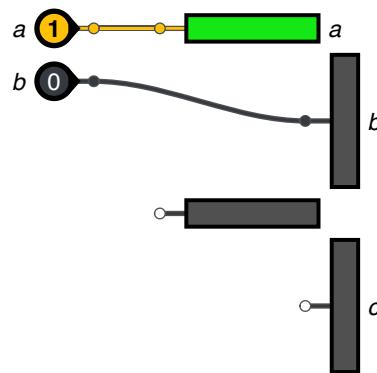


3.8.5 Affichage à 7 segments

Les affichages à 7 segments permettent d'afficher des chiffres à l'aide de 7 diodes lumineuses (LED).



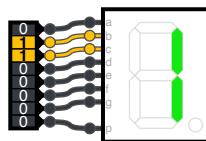
- Ajoutez les entrées et les lampes qui manquent pour compléter cet affichage à 7 segments.
- Tournez la barre avec *Affichage > Barre verticale*
- Ajoutez les étiquettes a-g
- Ajoutez 7 entrées et ajoutez les étiquettes a-g
- Affichez un nombre entre 0 et 9



3.8.6 Affichage à 2 chiffres

Les 7 diodes lumineuses (LED) permettent d'afficher les chiffres de 0 à 9. L'état des lampes (a-g) ainsi que du point décimal § est déterminé par 8 bits.

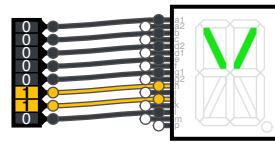
- Ajoutez un deuxième bloc affichage à 7 segments
- Ajoutez une entrée octet
- Connectez les deux automatiquement en alignant les broches
- Configurez les entrées pour afficher le nombre 42



3.8.7 Affichage à 16 segments

Les affichages à 16 segments permettent d'afficher aussi les lettres de l'alphabet, ainsi que des symboles de ponctuation.

- Configurez l'affichage pour afficher la lettre Y
- Ajoutez un deuxième affichage à 16 segments
- Ajoutez deux entrées 8-bits et connectez-les.
- Affichez la première lettre de votre nom



3.8.8 Porte NON

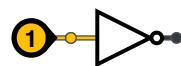
La porte NON inverse un signal. Montrez sa table de vérité.

- Mettez la première entrée à 0 et ajoutez une sortie
- Ajoutez une deuxième porte NON avec l'entrée à 1

Montrez l'effet de multiples portes NON.

- Mettez deux portes NON en série, ajoutez une entrée et une sortie
- Mettez trois portes NON en série, ajoutez une entrée et une sortie

table de vérité

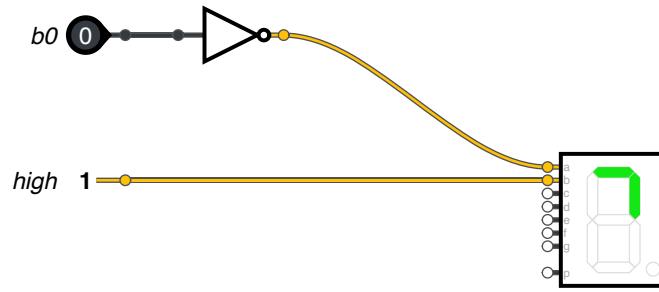


multiples portes NON

3.8.9 Afficher 0 et 1

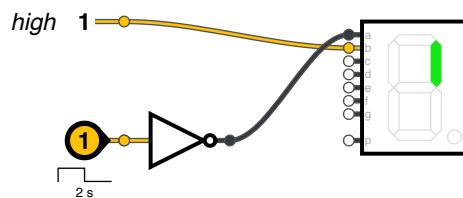
La porte NON inverse un signal.

- Ajoutez les connexions pour afficher 0 ou 1 selon le signal sur l'entrée **b0**.



3.8.10 Alterner 0 et 1

Utilisez une horloge et une porte NON pour afficher les chiffres 0 et 1 en alternance.



3.8.11 Porte OU

Une porte OU donne une sortie 1 si **au moins une** des entrées est à 1.

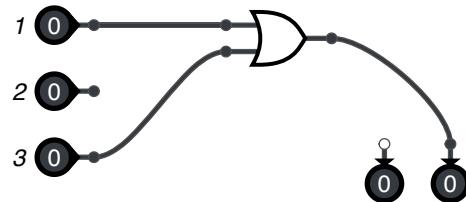
- Montrez la table de vérité pour la porte OU.
- Ajoutez 3 portes OU et mettez les entrées à 01, 10, et 11
- Créez une porte OU avec 3 entrées



3.8.12 Décodeur de clavier

Complétez le circuit pour un décodeur qui a le comportement suivant :

- bouton 1 appuyé produit la sortie binaire 01
- bouton 2 appuyé produit la sortie binaire 10
- bouton 3 appuyé produit la sortie binaire 11

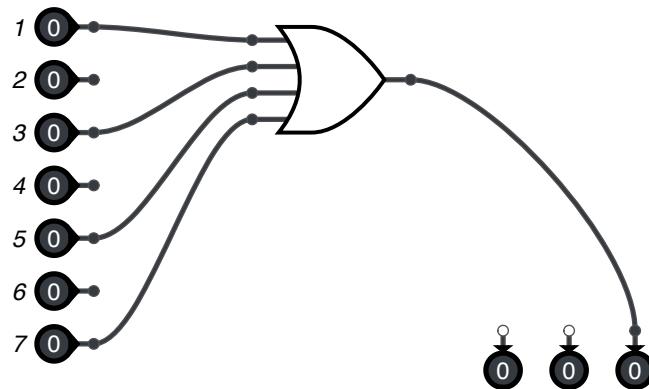


3.8.13 Clavier numérique

Quand on appuie sur une touche d'une calculette électronique, en interne l'action d'appuyer le bouton est transformée en une représentation binaire de la touche appuyée.

Complétez le circuit pour un décodeur qui a le comportement suivant :

- bouton 1 appuyé produit la sortie binaire 001
- bouton 2 appuyé produit la sortie binaire 010
- bouton 3 appuyé produit la sortie binaire 011
- bouton 4 appuyé produit la sortie binaire 100
- bouton 5 appuyé produit la sortie binaire 101
- bouton 6 appuyé produit la sortie binaire 110
- bouton 7 appuyé produit la sortie binaire 111



3.8.14 Porte ET

Une porte ET donne une sortie 1 seulement si **toutes** les entrées sont à 1.

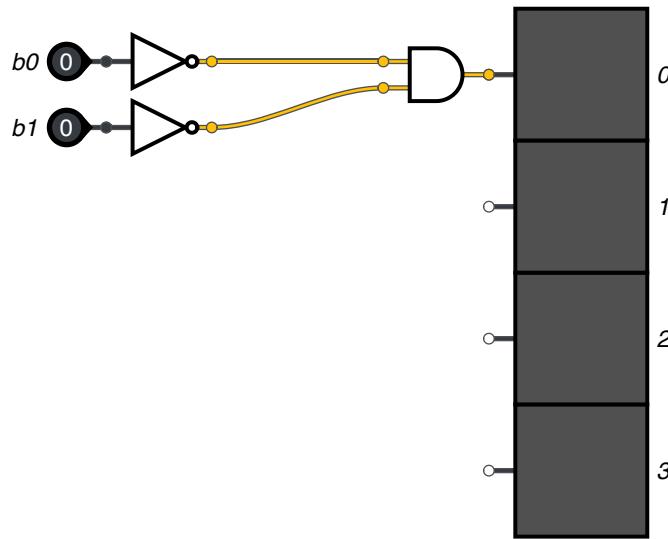
- Montrez la table de vérité pour la porte ET.
- Ajoutez 3 portes ET et mettez les entrées à 01, 10, et 11
- Créez une porte ET avec 3 entrées



3.8.15 Décodeur binaire

Complétez le circuit pour créer un décodeur binaire. Chaque combinaison des deux entrées binaires allume une et une seule des 4 lampes.

- 00 allume seulement lampe 0
- 01 allume seulement lampe 1
- 10 allume seulement lampe 2
- 11 allume seulement lampe 3



3.8.16 Décodeur de dé

Un dé de jeu peut afficher les nombres 1 à 6 à l'aide de 7 lampes. Plusieurs lampes s'allument par paire. Voici la table de vérité.

b2	b1	b0	valeur	a,g	b,f	c,e	d
0	0	0		0	0	0	0
0	0	1	1	0	0	0	1
0	1	0	2	1	0	0	0
0	1	1	3	1	0	0	1
1	0	0	4	1	0	1	0
1	0	1	5	1	0	1	1
1	1	0	6	1	1	1	0
1	1	1		1	1	1	1

Utilisez des portes logiques OU et ET pour créer le circuit de décodage pour afficher les lampes qui correspondent aux nombres 1 à 6.

Le nombre binaire $b_2b_1b_0$ doit allumer les lampes a-g pour afficher ce nombre dans la façon d'un dé à jeu standard.



3.8.17 Décoder 0 à 3

Le tableau ci-dessous montre les segments à allumer pour afficher les nombres 0 à 3 d'un affichage à 7 segments.

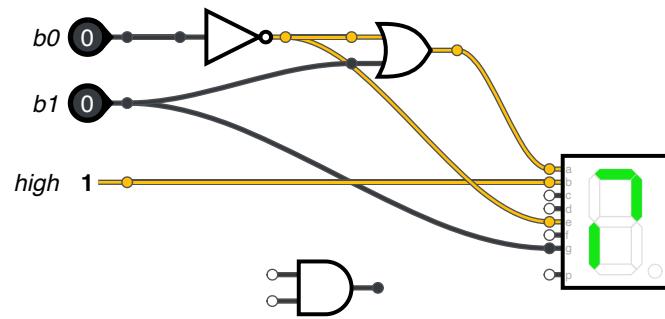
dec	b1	b0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1
3	1	1	1	1	1	1	0	0	1

Ajoutez des portes NON, OR, et AND pour compléter le circuit

Astuce - Essayez de trouver le circuit logique pour chaque colonne. C'est-à-dire il faut trouver le circuit pour allumer le segment. Par exemple pour le segment a vous avez la table de vérité suivante.

b1	b0	a
0	0	1
0	1	0
1	0	1
1	1	1

Certaines colonnes sont identiques. Donc vous pouvez utiliser le même signal.



Pour ce dernier circuit, faites une capture d'écran pour chacune des 4 conditions.

3.9 TP Additionneur

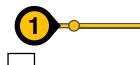
Dans cette section, nous allons explorer d'abord la porte **ou exclusif** (OU-X), qui nous sert à construire un **additionneur simple**. Nous modifions l'additionneur simple pour en créer un **additionneur complet** qui prend en compte la retenue. Cet additionneur 1 bit nous sert à construire des circuits pour créer d'autres opérations telles que :

- addition
- soustraction
- incrémantation
- décrémantation

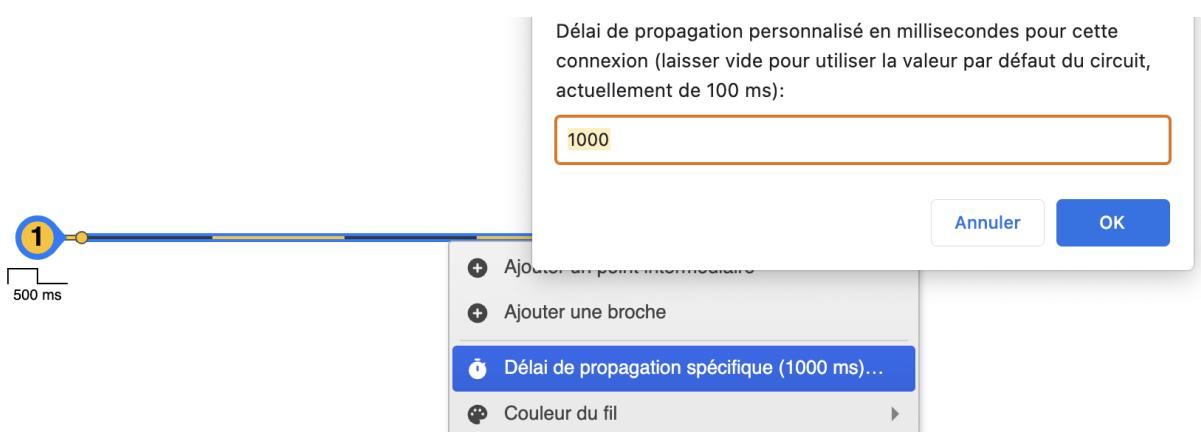
3.9.1 Clock et fréquence

L'entrée horloge (clock) produit un signal qui alterne entre 0 et 1. L'horloge ci-dessous a une période de 500 ms. Le délai de transmission a été mis à 1000 ms. Ceci fait apparaître 2 cycles du signal (en jaune). On peut donc visualiser la fréquence du signal qui est de 2 Hz.

- Ajoutez une deuxième horloge avec une période de 250 ms.
- Visualisez cette fréquence avec un délai de propagation de 1000 ms.
- Quelle est sa fréquence ? Mettez-la comme étiquette.
- Faites la même chose avec une troisième horloge qui a une période de 100 ms.



Rappel : pour choisir le délai de propagation, cliquez avec le bouton droit sur le fil et sélectionnez le menu contextuel *Délai de propagation spécifique...*. Mettez-le à 1000.



3.9.2 Porte OU-X

Une porte OU-X (ou exclusif) avec 2 entrées donne une sortie 1 si **exactement une** des entrées est 1.

- Montrez la table de vérité pour la porte OU-X.
- Ajoutez 3 portes OU-X et mettez les entrées à 01, 10, et 11
- Créez une porte OU-X avec 3 entrées et observez son comportement

Comment se comporte une porte OU-X avec plus que 2 entrées ?



3.9.3 Construire un OU-X

Comment peut-on construire un circuit OU-X avec des portes de base (NON, OU, ET) ? Regardons d'abord la table de vérité.

a	b	OU-X
0	0	0
0	1	1
1	0	1
1	1	0

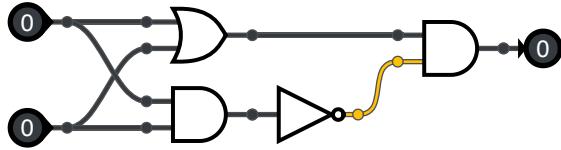
Le circuit ci-dessous représente une porte OU exclusive (OU-X). Mais il y a multiples façons de créer un circuit logique spécifique à partir des éléments de base.

Créez une deuxième façon pour obtenir une porte OU exclusive en partant de l'observation :

(not a and b) or (b and not a)

Utilisez donc :

- 2 portes NON
- 2 portes ET
- 1 porte OU



3.9.4 DéTECTEUR DE PARITÉ

Une porte ou exclusif est un détecteur de parité (pair/impair). La sortie d'une porte ou exclusif est 1 si le nombre des entrées actives est impair.

Ajoutez encore 6 portes OU-X et complétez la table de vérité pour les 8 combinaisons possibles :

- pair : 000, 011, 101, 110
- impair : 001, 010, 100, 111



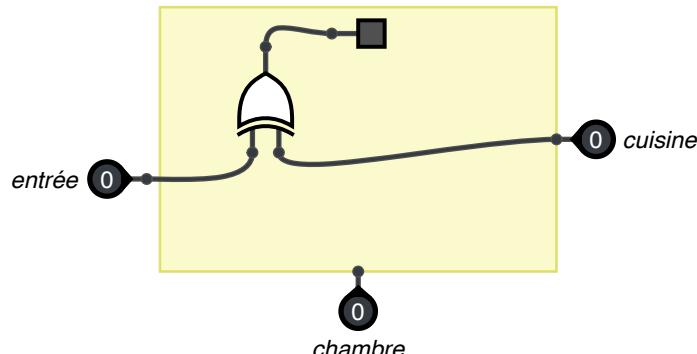
Pour détecter si un nombre d'entrées est pair, il suffit d'ajouter un NON à la sortie du OU-X. On appelle ce circuit un NON-OU-X.

3.9.5 Multiples commutateurs

La porte OU-X permet d'allumer et éteindre une lampe avec des commutateurs multiples.

Dans le schéma ci-dessous, on peut allumer la lumière dans une pièce à partir de la porte d'entrée et de la cuisine.

Ajoutez un circuit pour qu'on puisse également l'allumer depuis la chambre.



3.9.6 Addition binaire

Nous avons maintenant tous les éléments pour construire un additionneur binaire. Rappelons-nous que l'addition binaire est très simple.

A	B	A+B	C	S
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

Le résultat A+B peut être 0, 1 ou 2. Nous avons besoin de deux bits pour représenter le résultat :

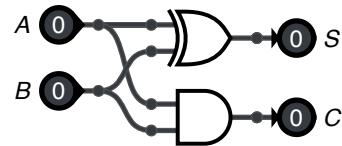
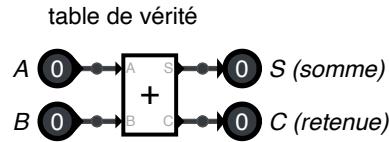
- le bit de somme S
- le bit de retenue C (*carry* en anglais)

En regardant la table de vérité, on constate que :

- la somme S est exprimée par la fonction OU-X
- la retenue C est exprimée par la fonction ET

Vous trouvez le circuit ci-dessous à droite. Vérifiez sa fonction en cliquant sur ses entrées.

Ajoutez encore 3 demi-additionneurs et montrez la table de vérité pour les 4 conditions d'entrée : 00, 01, 10, 11.



3.9.7 Additionneur complet

Dans le cas général de l'addition, nous n'additionnons pas deux bits, mais deux nombres à plusieurs bits. Voici l'addition en colonne de deux nombres 4 bits ($3+11=14$).

$$\begin{array}{r}
 0011 \\
 +1011 \\
 \hline
 1110
 \end{array}$$

Pour être explicite, nous introduisons une ligne supplémentaire qui représente la retenue (C = carry).

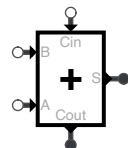
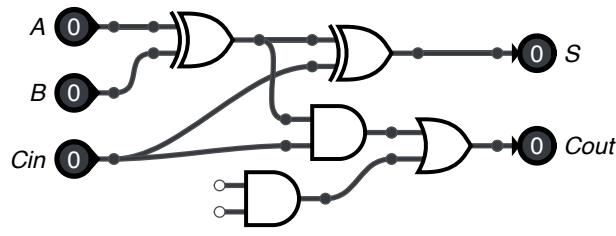
$$\begin{array}{r}
 0110 \text{ (retenue)} \\
 0011 \\
 +1011 \\
 \hline
 1110
 \end{array}$$

L'additionneur de 2 bits de la section précédente n'est plus suffisant. Pour le cas général, nous avons besoin d'un additionneur qui additionne 3 bits. Il faut tenir compte de la retenue (Cin), qu'il faut inclure dans l'addition. Voici donc la table de vérité pour un additionneur complet.

Cin	A	B	Cin+A+B	Cout	S
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Regardez les colonnes et essayez de comprendre avec quelles portes on pourrait le construire. Vous constatez que la colonne S représente la parité. On pourra donc la construire avec des portes OU-X.

- Ajoutez les deux fils qui manquent à l'entrée de la porte ET pour que le circuit produise le signal Cout et se comporte comme un additionneur complet.
- Ajoutez des entrées et sorties au bloc de l'additionneur complet et vérifiez son fonctionnement.

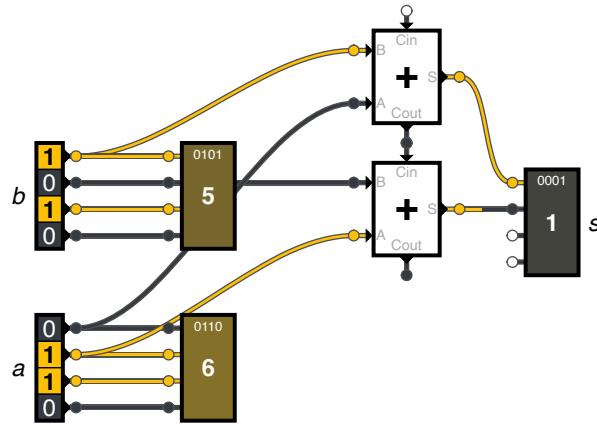


3.9.8 Additionneur 4 bits

Pour additionner deux nombres 4-bits (quartets) nous avons besoin de 4 additionneurs complets. Chaque sortie Cout est liée à la l'entrée Cin de l'additionneur suivant.

Pour additionner **a** et **b** vous devez additionner les bits correspondants : a_0+b_0 , a_1+b_1 , etc.

- Ajoutez les circuits manquants pour additionner deux nombres 4-bits.
- Montrez l'addition de 7+5 dont le résultat devrait être 12.

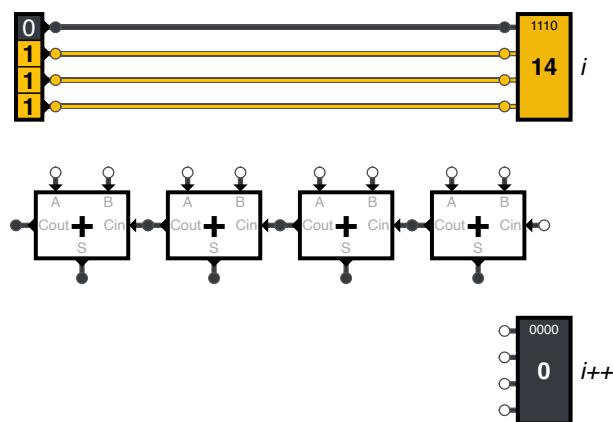


3.9.9 Incrémenter (*i++*)

Additionner 1 à un nombre binaire est une opération très fréquente. Elle est utilisée pour incrémenter le compteur de programme pc (program counter), pour pointer à la prochaine instruction.

Complétez le circuit pour incrémenter la variable *i*. Dans beaucoup de langages de programmation, une variable incrémentée est désignée par *i++*. En Python nous écrivons *i = i + 1*.

D'ailleurs le nom du langage de programmation C++ est une référence à cet opérateur d'incrémantation.

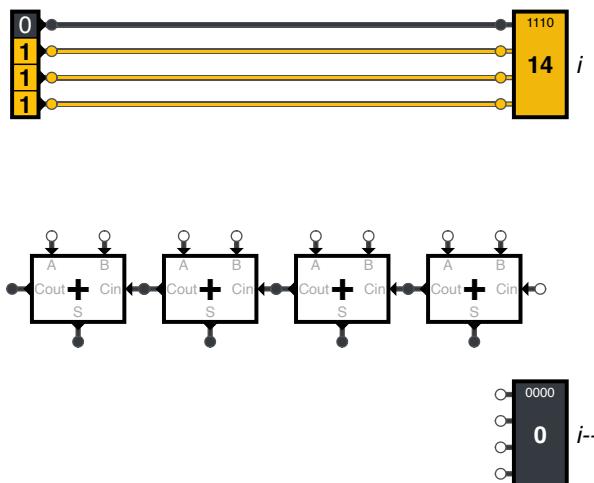


3.9.10 Décrémenter ($i--$)

Soustraire 1 à un nombre binaire est une opération très fréquente. Elle est utilisée pour décrémenter un compteur de boucle i , un pointeur de pile sp (stack pointer), ou un pointeur p vers les adresses de la mémoire.

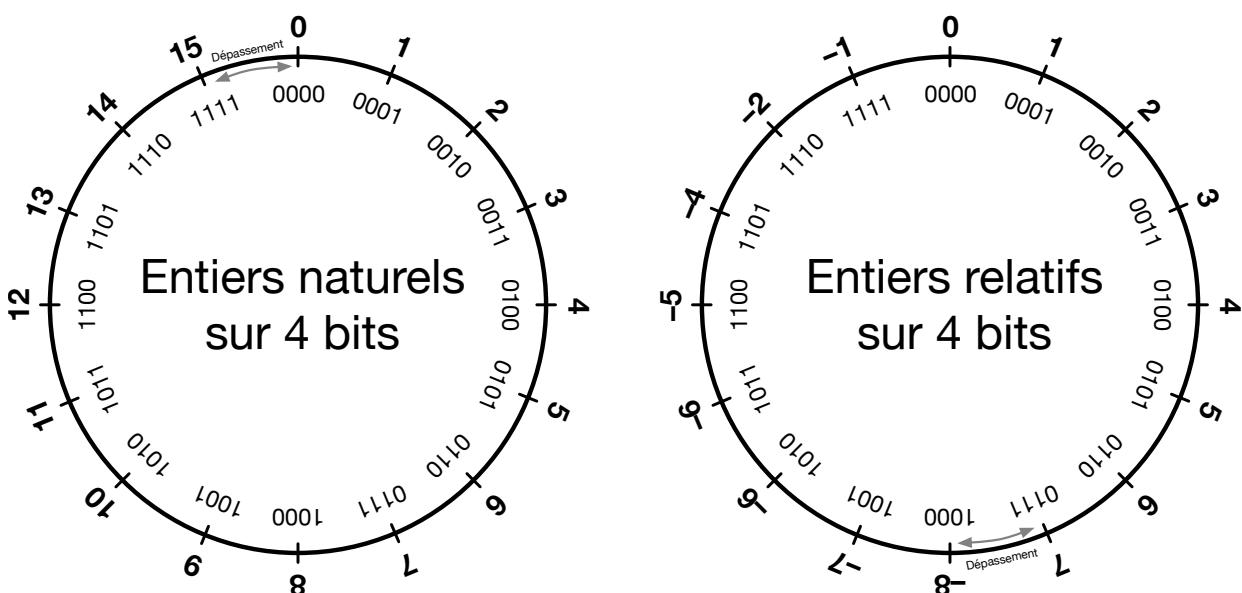
Complétez le circuit pour décrémenter la variable i . Dans beaucoup de langages de programmation, une variable incrémentée est désignée par $i--$. En Python nous écrivons $i = i - 1$.

Astuce : pour décrémenter la valeur i il suffit d'additionner 1111 qui représente la valeur -1 en format signé.



3.9.11 Changer de signe ($-i$)

Les nombres signés sont représentés avec le format *complément à deux*. Pour un nombre 4-bits, ceci nous donne une plage de -8 à +7 pour des entiers relatifs, et une plage de 0 à 15 pour des entiers naturels. Nous constatons que la plage signée n'est pas symétrique : le côté négatif compte un nombre en plus.

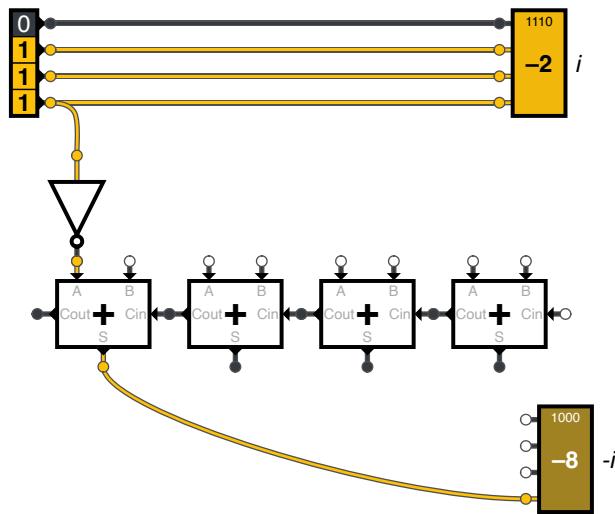


L'opération pour trouver le nombre négatif est : inverser tous les bits (symbolisé par \sim) et additionner 1.Mathématiquement nous pouvons exprimer cette opération comme :

$$-i = \sim i + 1$$

Par exemple, pour obtenir la représentation binaire de -1 nous inversons 0001, ce qui donne 1110 et nous additionnons 1, ce qui donne 1111.

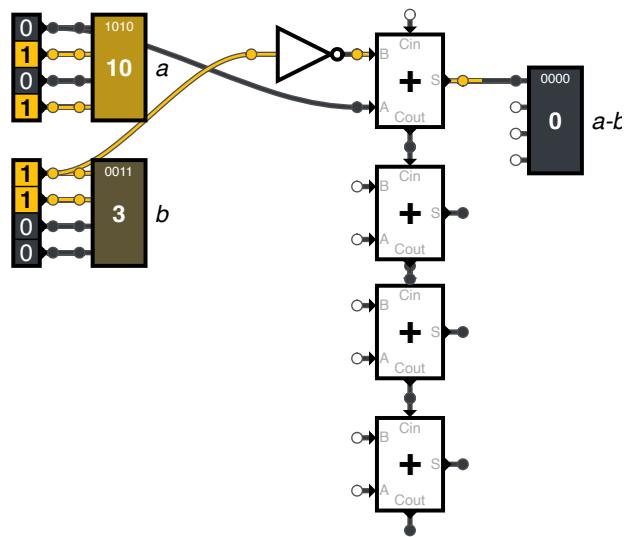
Complétez le circuit pour inverser le signe de la variable i et obtenir son négatif $-i$



3.9.12 Soustraction (a-b)

Pour soustraire deux nombres $a-b$ il suffit d'additionner le nombre négatif du deuxième ($-b$). Ce nombre négatif peut être obtenu en inversant tous les bits et additionner 1.Donc $-b = \sim b + 1$.

Complétez le circuit pour soustraire $a-b$. Le résultat de 10-3 devrait être 7.

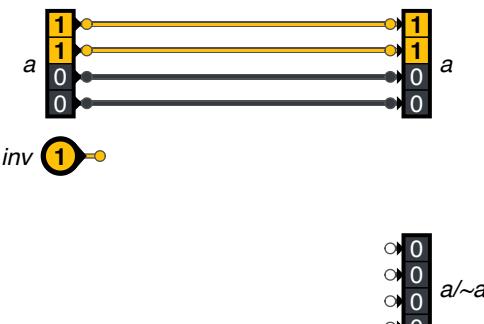


3.9.13 Inversion commutée

L'inverseur commuté permet d'inverser tous les 4 bits d'un nombre avec un signal de contrôle **inv** :

- pour **inv** = 0 la sortie est inchangée (**a**)
- pour **inv** = 1 la sortie est inversée ($\sim a$)

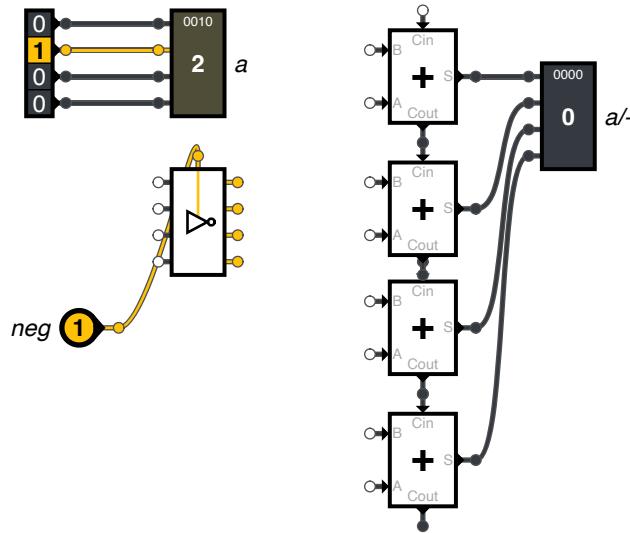
Ajoutez un inverseur commuté pour obtenir $\sim a$ ou **a** selon l'état du sélecteur.



3.9.14 Négation commutée

Complétez le circuit pour pouvoir obtenir $-a$ ou a selon l'état du sélecteur **neg** :

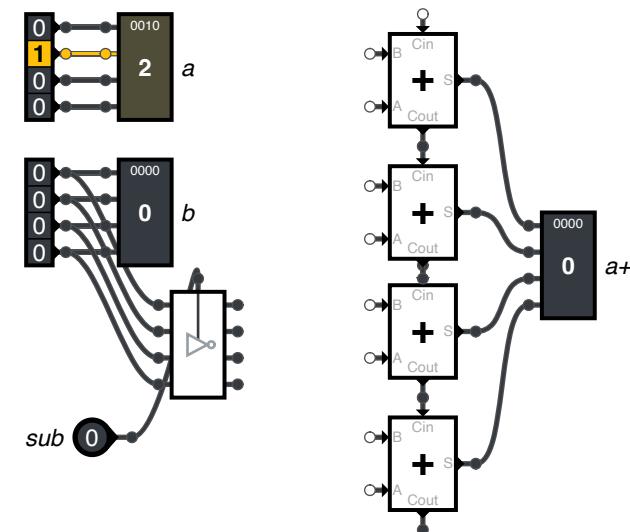
- pour **neg** = 0 la sortie est inchangée (a)
- pour **neg** = 1 la sortie change de signe ($-a$)



3.9.15 Soustraction commutée

Complétez le circuit pour pouvoir obtenir une opération différente selon l'état du sélecteur **sub** :

- pour **sub** = 1 les opérandes sont soustraits ($a-b$)
- pour **sub** = 0 les opérandes sont additionnés ($a+b$)



3.9.16 Les fanions (flags)

Les fanions (flag) sont des signaux qui caractérisent un nombre.

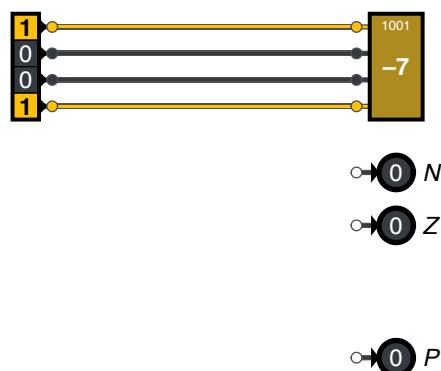
- N pour indiquer que le nombre est négatif
- Z pour indiquer que le nombre est zéro
- P pour indiquer que le nombre de bits à 1 est pair

Par exemple pour le nombre **1001** (-7) on aura N=1, Z=0 et P=1.

Prudence : Faites attention à la différence entre la **parité du nombre** et la **parité des bits**.

- la parité du nombre est exprimée par le bit de poids faible (b0),
- la parité du nombre des bits est obtenue avec une opération OU-X (ou exclusif).

Complétez le circuit pour correctement afficher les fanions N, Z et P.



3.10 TP ALU

L'unité arithmétique et logique (ALU) permet de choisir parmi un certain nombre d'opérations. Nous allons voir comment une ALU peut choisir entre différentes opérations (ET, OU, addition, soustraction) à l'aide d'un **multiplexeur**.

Nous allons découvrir comment des décalages et additions successives peuvent constituer une **multiplication** ou une **division**.

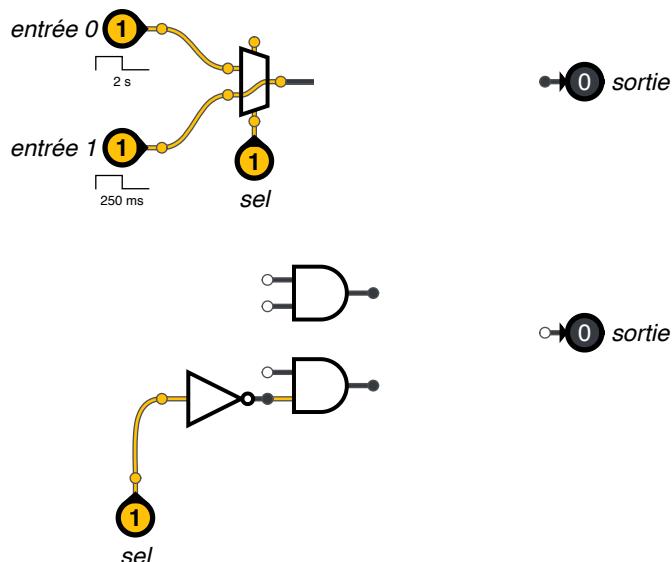
Finalement, un **registre** permet de mémoriser les opérandes du calcul. Un registre particulier appelé **accumulateur** permet de faire des additions successives et *accumuler* une somme courante.

3.10.1 Sélectionneur

L'entrée **sel** du sélectionneur permet de choisir entre deux signaux d'entrée :

- entrée 0 : signal lent (période de 2 s)
- entrée 1 : signal rapide (période de 250 ms)

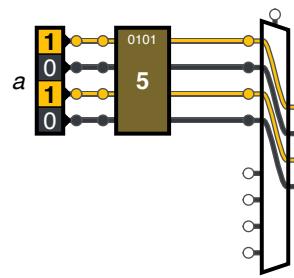
Recréez un tel sélecteur avec des portes NON, ET, OU.



3.10.2 Multiplexeur

Le multiplexeur (MUX) permet de choisir entre deux signaux 4-bits nommés a et b. Ajoutez les éléments qui manquent.

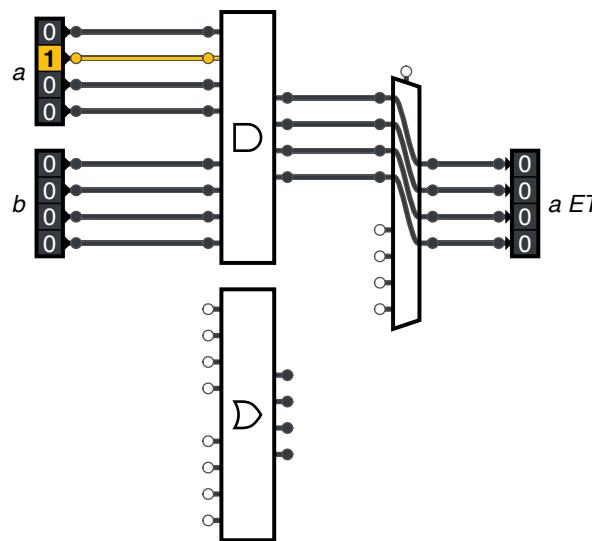
- Ajoutez une deuxième entrée 4-bits avec un affichage.
- Ajoutez un décodeur et affichage à 7 segments.



3.10.3 Sélection d'opérations

Complétez le circuit qui permet de sélectionner entre les deux opérations $a \text{ ET } b$ et $a \text{ OU } b$.

- Connectez a et b aux entrées des 4 portes OU,
- Ajoutez une sortie 4-bits pour afficher le résultat des opérations logiques
- Ajoutez une entrée de sélection pour le multiplexeur.

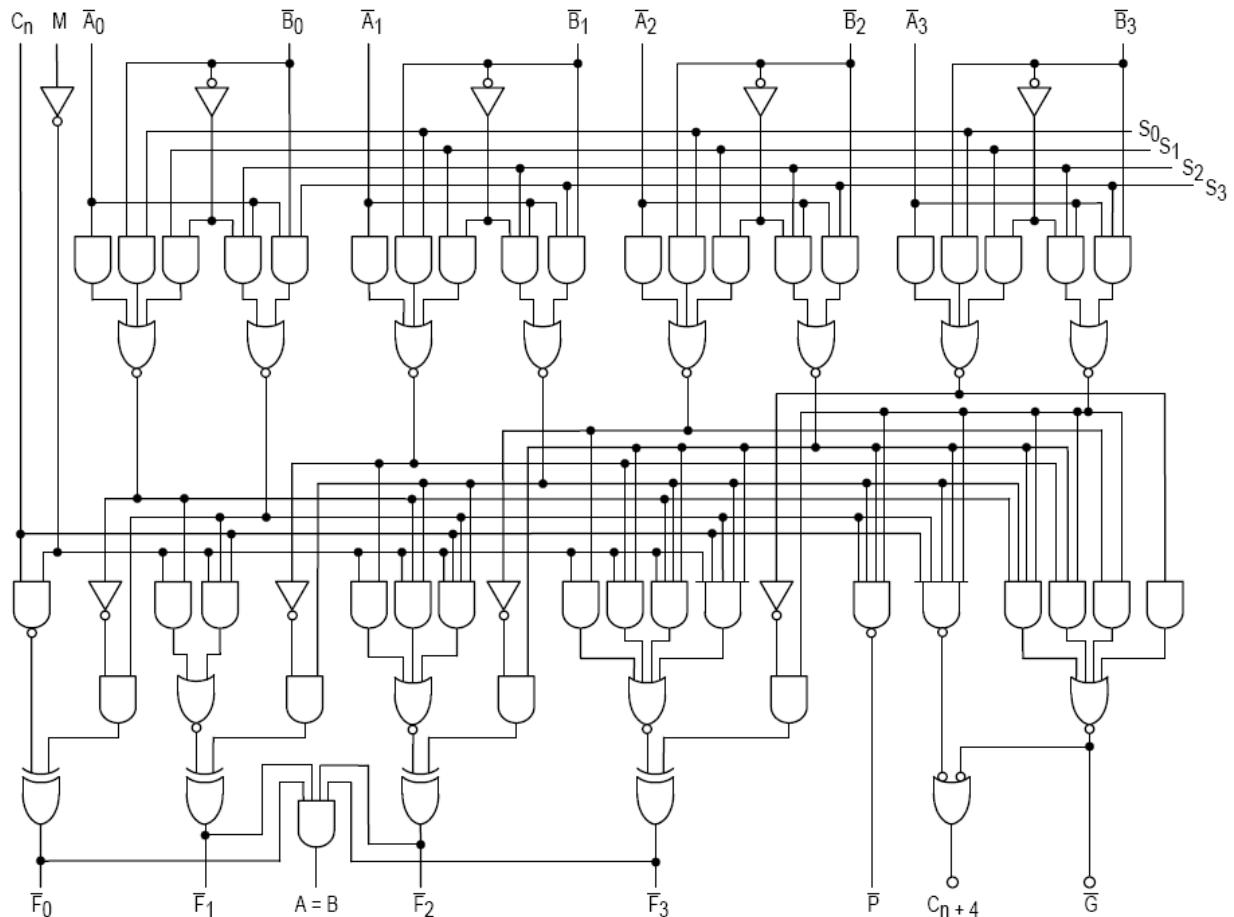


Un clic droit sur la porte quadruple permet de choisir son type (AND, OR, XOR, NAND, NOR, XNOR).

3.10.4 ALU

L'unité arithmétique et logique, ALU (arithmetic and logic unit), est la partie de l'ordinateur qui effectue les différents calculs arithmétiques et logiques.

Ci-dessous vous pouvez voir les circuits logiques d'une ALU 4-bits très utilisée dans les années 60 et 70, le modèle 74181.



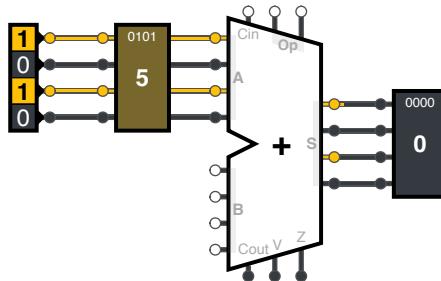
L'ALU dont nous disposons peut effectuer 4 opérations :

- addition (00)
- soustraction (01)
- OU logique (10)
- ET logique (11)

Avec l'ALU ci-dessous, ajoutez

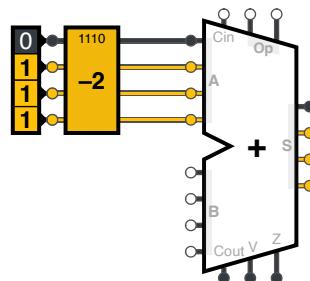
- la deuxième entrée (B) avec un bloc d'affichage 4 bits
- un bloc d'affichage 4 bits pour la sortie (S)
- les 3 entrées (Cin , Op0, Op1)
- les 3 sorties ($Cout$, V, Z)

Ensuite, testez les 4 opérations. Montrez une soustraction.



3.10.5 Addition signée

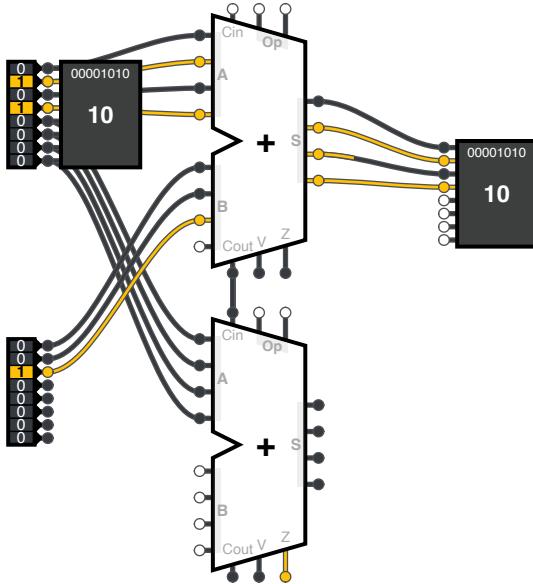
Interprétez les nombres binaires comme des nombres signés. Vous pouvez configurer l'afficheur 4 bits avec son menu contextuel. Complétez l'additionneur 4-bit et montrez que l'addition de -2 et -3 donne bien -5.



3.10.6 Addition 8 bits

Pour additionner un nombre à 8-bits, il faut combiner deux ALU 4-bits. Dans ce cas il faut connecter Cout de la première ALU avec Cin de la deuxième.

- Complétez le circuit pour afficher l'addition de deux nombres binaires 8-bits.
- Ajoutez une entrée pour soustraire deux nombres.
- Montrez une soustraction correcte, dont le résultat est plus grand que 30

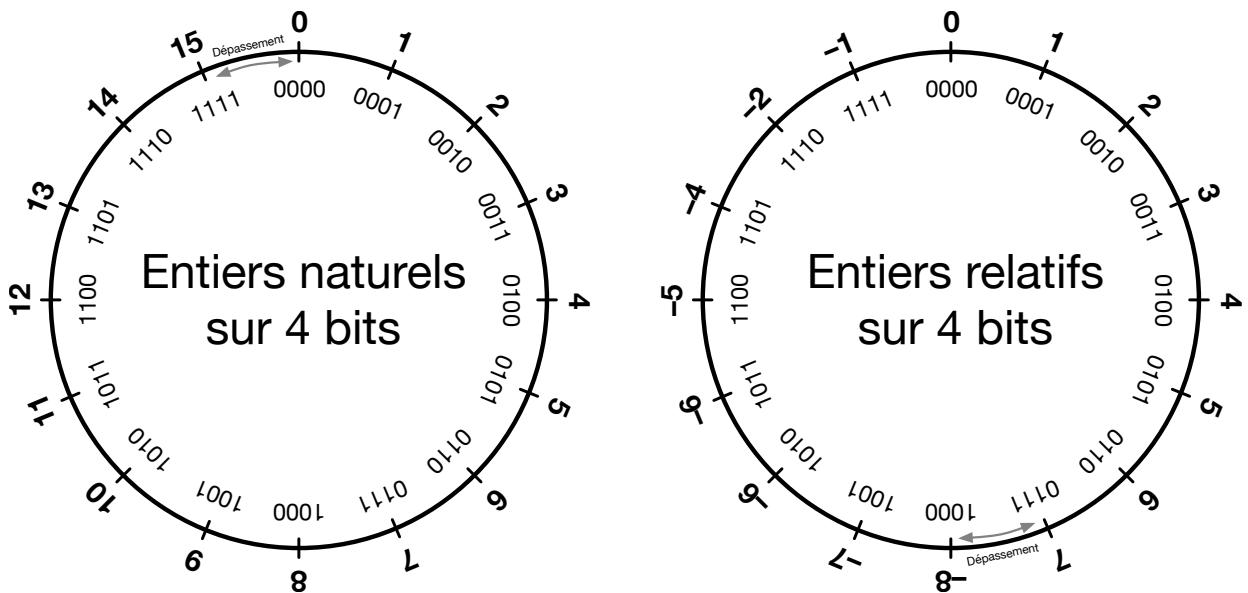


3.10.7 Carry et Overflow (V)

L'ALU possède deux sorties pour indiquer un dépassement de plage de résultat. Le résultat affiché est alors décalé de 16. Ce cas est signalé par l'ALU à l'aide de deux signaux de sortie spéciaux.

- C (carry) signale un dépassement pour des nombres non signés,
 - V (overflow) signale un dépassement pour des nombres signés.

L'addition de deux nombres naturels (0 à 15) peut produire un résultat de 0 à 30. L'addition de deux nombres relatifs (-8 à 7) peut produire un résultat de -16 à 14. Dans certains cas on aura donc besoin de 5 bits pour représenter correctement le résultat.



Pour les nombres non signés (première ALU)

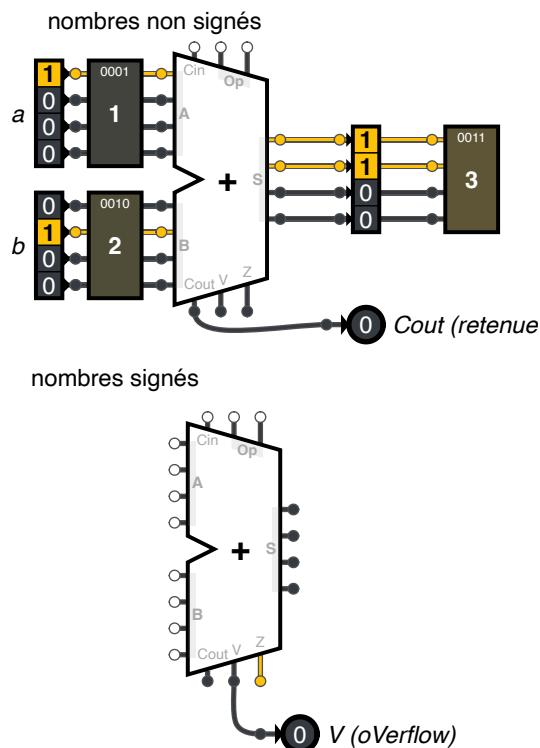
- Choisissez des valeurs a et b qui produisent un dépassement ($C = 1$), et donc un affichage incorrect sur une sortie de 4 bits.

- Ajoutez en plus un affichage 8 bits et connectez-y les 4 bits de S et C comme 5e bit, pour afficher le résultat correct.

Pour les nombres signés (deuxième ALU)

- Ajoutez 2 entrées 4 bits et 1 sortie 4 bits.
- Ajoutez 3 affichages 4 bits configurés (via menu contextuel) en nombres signés.
- Choisissez des valeurs a et b négatives qui produisent un dépassement ($V = 1$).
- Ajoutez un affichage 8 bits configuré (via le menu contextuel) en nombres signés, et connectez-y les 4 bits de S et C comme 5e bit.

Attention : Dans ce cas vous devez connecter C également avec les 3 autres bits (b5-b7) pour faire une propagation du bit de signe et traiter correctement le cas des nombres négatifs.



3.10.8 Multiplier 1x1 bit

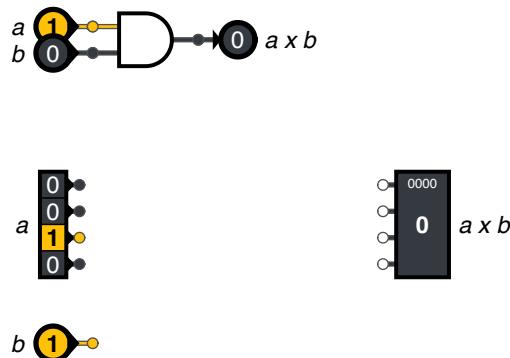
Les règles de la multiplication 1-bit sont très simples. Voici la table de vérité.

a	b	$a \times b$
0	0	0
0	1	0
1	0	0
1	1	1

On voit tout de suite que ceci correspond à la porte ET. Dans l'exemple si dessous vous voyez une porte ET pour multiplier a et b , les deux ayant juste 1 bit.

- Vérifiez le bon fonctionnement du multiplicateur 1-bit
- Ensuite, utilisez 4 portes ET pour créer un multiplicateur a (4-bits) fois b (1-bit).

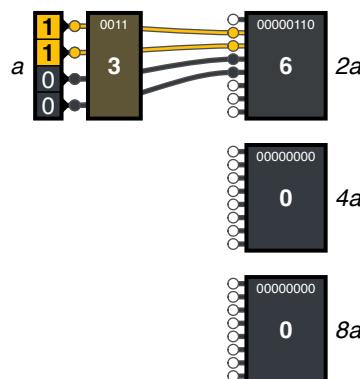
- Basculez b entre 0 et 1 pour vérifier si votre circuit fonctionne correctement.



3.10.9 Multiplier par 2, 4, 8

La multiplication par une puissance de 2 est facile. Il suffit de décaler les bits. Le circuit ci-dessous calcule $2a$ en décalant d'un bit en direction du poids fort.

- Complétez le circuit pour calculer et afficher $4a$ et $8a$.
- Vérifiez avec $a=5$. Votre affichage devrait montrer 10, 20 et 40.



3.10.10 Multiplier 1x4 bit

Le circuit ci-dessous utilise un multiplexeur 8x4 pour faire la multiplication d'un nombre **a** (4 bits) avec un nombre **b** (1 bit), au lieu des 4 portes ET utilisées précédemment.

Nous pouvons représenter un nombre binaire **b** par une séquence de 4 chiffres binaires : $b_3b_2b_1b_0$. Chaque bit b_i contrôle la multiplication de son poids (2^i) avec a .

- $b_0 \cdot 2^0 a$
- $b_1 \cdot 2^1 a$

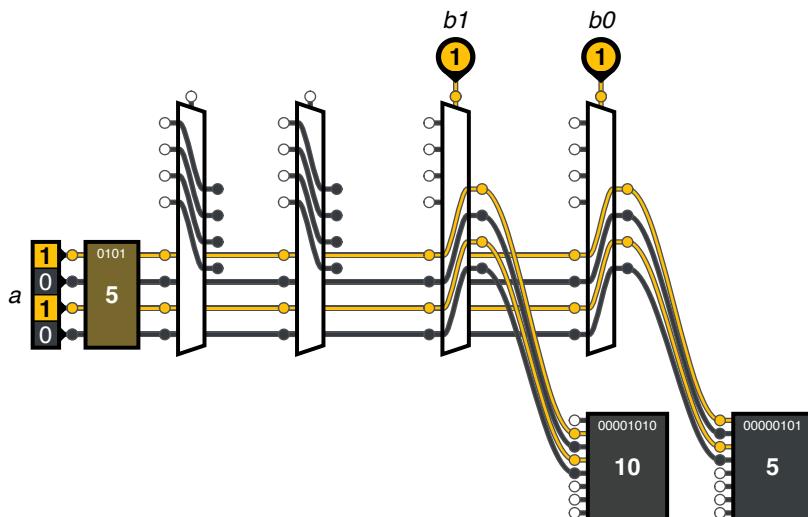
- $b_2 \cdot 2^2 a$
- $b_3 \cdot 2^3 a$

Pour compléter l'opération de multiplication 4x4 bits, la dernière étape sera d'additionner les 4 nombres.

Complétez le circuit avec :

- deux entrées que vous appelez **b2** et **b3**
- un affichage 8 bits qui affiche 4a sous contrôle de b2
- un affichage 8 bits qui affiche 8a sous contrôle de b3

Essayez de multiplier deux nombres, par exemple a=5 et b=5 (0101). Vous trouvez le résultat de la multiplication en additionnant les 4 nombres affichés.



3.10.11 Multiplier 4x4 bits

La multiplication 4 x 4 bits nécessite :

- 4 multiplexeurs pour la multiplication 4 x 1 bit
- 3 additionneurs pour additionner les 4 opérandes décalés

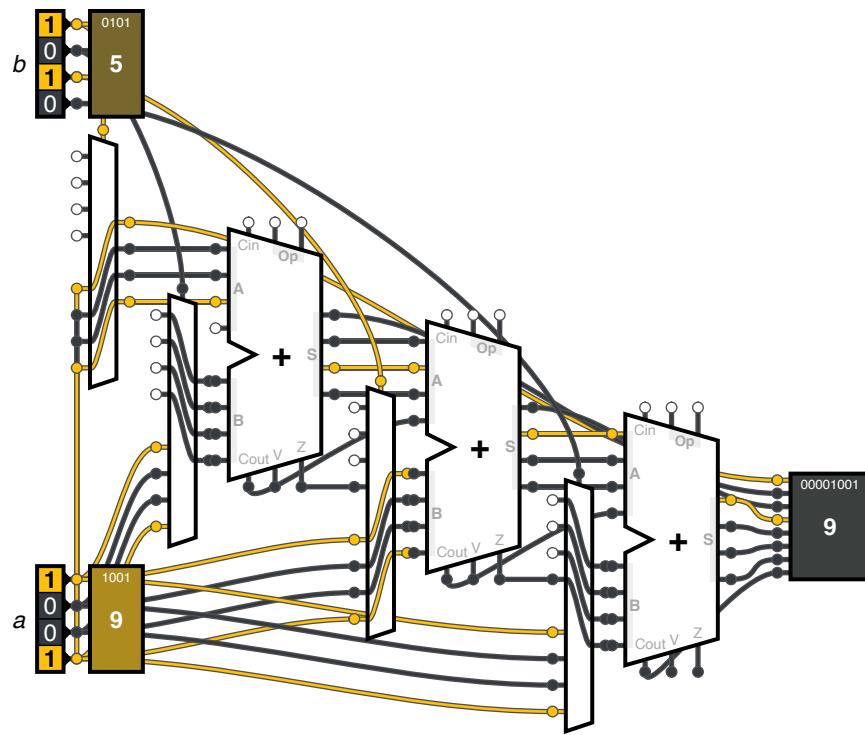
Pour multiplier $0101 \times 1001 = 00101101$ ($5 \times 9 = 45$) nous écrivons en colonnes ceci :

$$\begin{array}{r}
 1 & 1001 \\
 0 & 0000 \\
 1 & 1001 \\
 0 & +0000 \\
 \hline
 00101101
 \end{array}$$

Cet algorithme peut être exprimé mathématiquement comme

$$\text{produit} = \sum_{i=0}^4 (b_i \cdot a) \cdot 2^i$$

Modifiez a et b dans le circuit multiplicateur 4 x 4 bits ci-dessus vérifiez que vous obtenez bien le produit de a et b. Faites une capture d'écran avec la plus grande valeur possible.



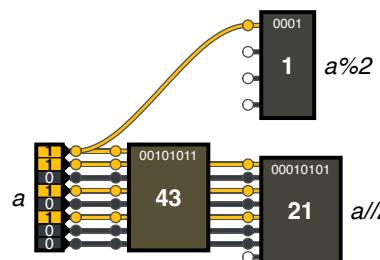
3.10.12 Diviser par 2, 4 et 8

La division par une puissance de 2 est très simple. Il suffit de décaler le nombre binaire. Pour diviser par 2 nous décalons d'une unité, et nous obtenons :

- La division entière ($a//2$)
- Le reste de la division, l'opération modulo ($a\%2$)

Ajoutez deux affichages 8 bits pour la division par 4 et 8 Ajoutez deux affichages 4 bits pour le modulo 4 et 8 Ajoutez les étiquettes ($a\%4$, $a//4$, etc.)

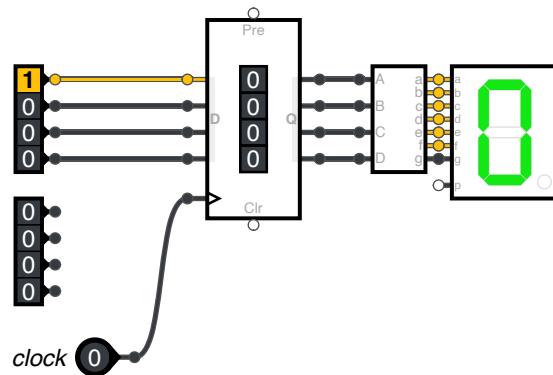
Par exemple pour $43//8$ vous devriez obtenir 5, et pour $43\%8$ vous devriez obtenir 3.



3.10.13 Registre

Le registre que nous allons voir plus en détail dans le prochain chapitre permet de mémoriser une donnée. Avec un coup d'horloge (clock), les 4-bits de données sont mémorisés.

Ajoutez un deuxième registre, décodeur et affichage à 7 segments, pour permettre d'afficher un nombre décimal de 00 à 99 ou un nombre hexadécimal de 00 à FF.



3.10.14 Accumulateur

Un accumulateur est un registre spécial qui *accumule* une somme. La sortie de l'accumulateur est reliée avec l'entrée A de l'ALU. À chaque coup d'horloge du registre, le calcul $acc + b$ est effectué et affiché.

Par exemple dans le circuit ci-dessous, l'accumulateur contient 3. Au prochain coup d'horloge, l'entrée b qui est 2 y sera additionnée. Ceci permet de calculer une somme courante.

Voici un exemple typique, calculer la somme 1+3+7. En Python ceci correspondrait à :

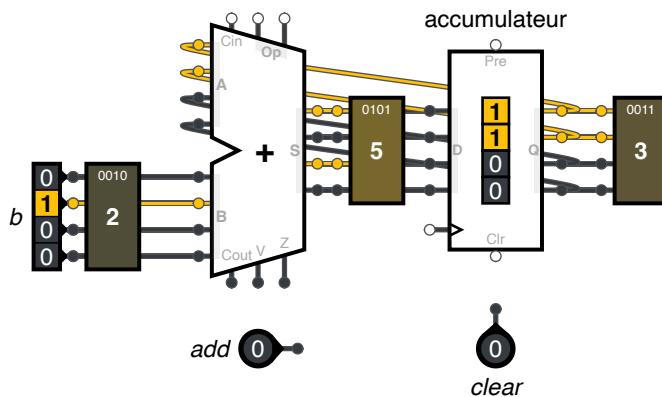
```
acc = 0      # clear
acc += 1    # add
acc += 3    # add
acc += 7    # add
print(acc)
```

Avec le circuit ci-dessous ceci correspond à 4 étapes :

1. **clear**
2. **b=1 et add**
3. **b=3 et add**
4. **b=7 et add**

Connectez les entrées **clear** et **add** au bon endroit et calculez $1+3+7$.

Attention : tenez le bouton suffisamment longtemps pour laisser propager les signaux jusqu'au bout.



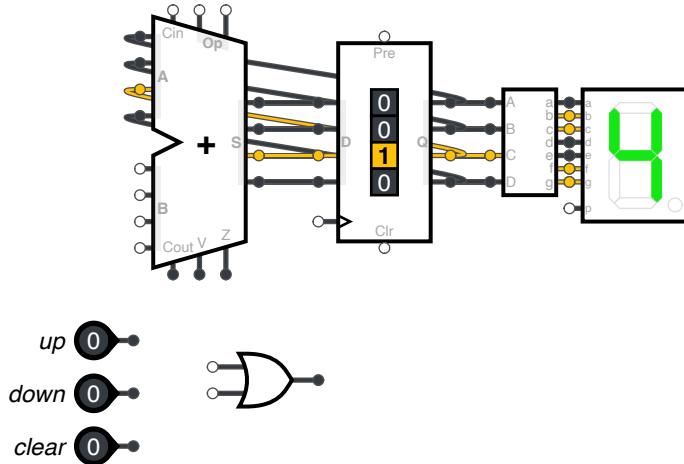
3.10.15 Incrémente/décrémente

Certains appareils électroniques ont très peu de touches et on doit utiliser juste deux boutons. C'est le cas pour régler la température ou le volume.

Compléter le circuit pour les boutons

- **up** pour incrémenter la valeur (clock)
 - **down** pour décrémenter la valeur (clock + soustraction)
 - **clear** pour mettre la valeur à zéro

Attention au délai de transmission par défaut de 100 ms. Il faut soit appuyer plus longtemps sur les boutons, ou diminuer ce délai.



3.10.16 Comparer

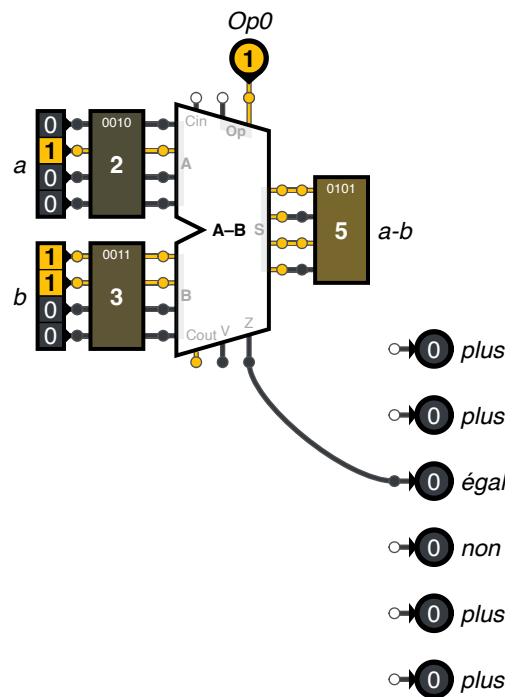
En Python nous disposons de 6 comparateurs pour comparer deux nombres a et b :

- > plus grand
- \geq plus grand ou égal
- == égal
- \neq non égal
- \leq plus petit ou égal
- < plus petit

Nous pouvons créer ces 6 comparaisons en utilisant une ALU qui soustrait deux nombres a et b et quelques portes logiques. Voici quelques astuces :

- quand $a-b$ est zéro alors $Z=1$, donc **a égal b**
- quand $a-b$ est négatif, alors $C=1$, donc **a plus petit que b**
- quand vous combinez les deux $Z=1$ ou $C=1$, vous trouvez **a plus petit ou égal à b**

Utilisez les portes ET, OU et NON pour décoder les 6 types de comparaisons.



3.11 TP Mémoire

Les circuits que nous avons vus jusqu'à maintenant s'appellent **circuits combinatoires**. Leur sortie est le seul résultat de leurs entrées. Une même entrée produit toujours la même sortie. Le circuit n'a pas de mémoire.

La famille de circuits que nous allons découvrir s'appelle **circuit séquentiel**. Ces circuits permettent de mémoriser un état.

3.11.1 Cellule de mémoire

Une cellule de mémoire élémentaire peut être créée avec une porte OU, ou la sortie est connectée avec une de ses entrées.

Circuit 1

- Connectez la sortie de la porte OU via la porte OUI avec une de ses entrées
- Montrez que vous avez une cellule de mémoire qui peut mémoriser une seule fois (fusible)

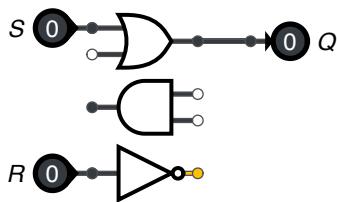
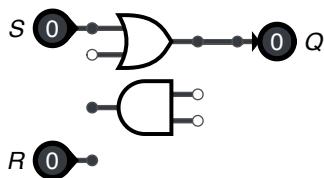
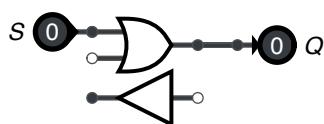
Circuit 2

- Connectez la sortie de la porte OU via la porte ET avec une de ses entrées
- Montrez que vous avez une cellule de mémoire qui peut mémoriser S (set) et qui peut être remis avec R (reset)

Circuit 3

- Connectez la sortie de la porte OU via la porte ET avec une de ses entrées
- Contrôlez la porte ET via une porte NON
- Mettez l'entrée R en pousoir

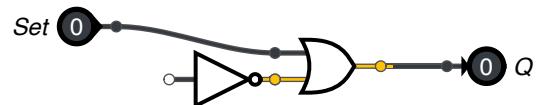
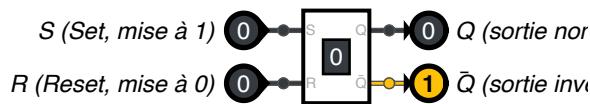
Attention : Pour éviter des effets d'oscillation, vous devez appuyer sur le bouton pressoir plus longtemps que 200 ms ou diminuer fortement le temps de propagation dans la boucle.



3.11.2 Verrou SR

Un verrou SR (set-reset) permet de “verrouiller” (mémoriser) une information.

- Jouez avec le verrou SR pour comprendre son fonctionnement
- Recréez le verrou SR avec les composants NOT et OR



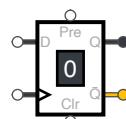
$\circ \rightarrow 0 \quad Q_-$

Reset $\bullet 0 \bullet$

3.11.3 Bascule D

La bascule D (data) recopie la donnée sur l’entrée **D** vers sa sortie **Q** à chaque front montant de l’horloge **clock**.

- ajoutez les 6 entrées/sorties à la bascule D,
- lisez et comprenez les étiquettes,
- observez et comprenez son comportement.

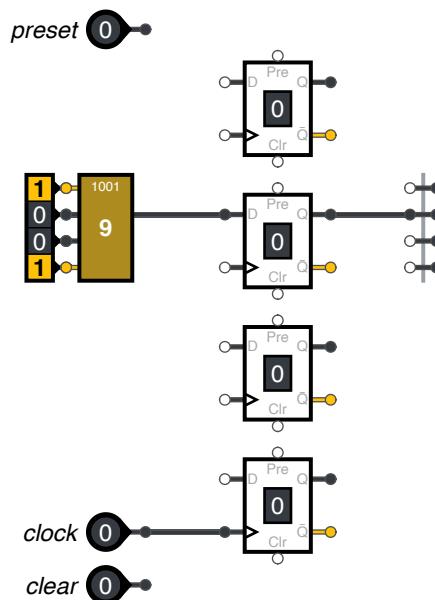


3.11.4 Registre 4 bits

Le **registre** est un circuit qui peut mémoriser multiples bits sur un coup d'horloge. À partir de 4 bascules D, nous pouvons construire un registre pour mémoriser 4 bits.

Ajoutez les connexions qui manquent :

- Connectez chacun des 4 bits d'entrée vers une entrée **D** de la bascule.
- Connectez chacun des 4 bits de sortie **Q** vers son bit de sortie correspondante.
- Ajoutez une sortie 4 bits et un affichage 4 bits
- Connectez les 4 entrées **preset**, **clock** et **clear**



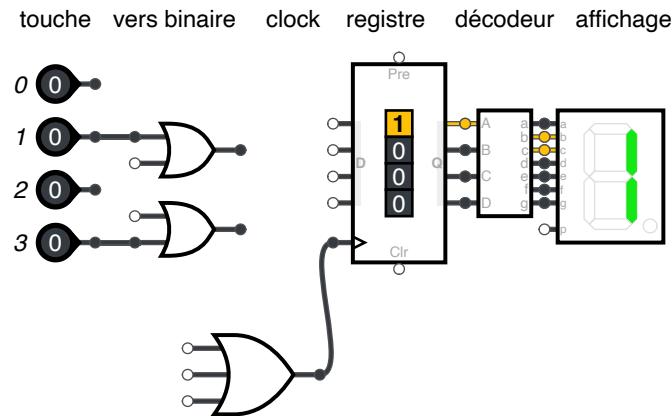
3.11.5 Décodeur de touche

Quand on appuie sur une touche d'une calculatrice électronique, telle que la TI-30, la valeur de la touche est transformée en binaire 4 bits, enregistré dans un **registre 4 bits**, et affiché avec un affichage à 7 segments.

Les étapes sont :

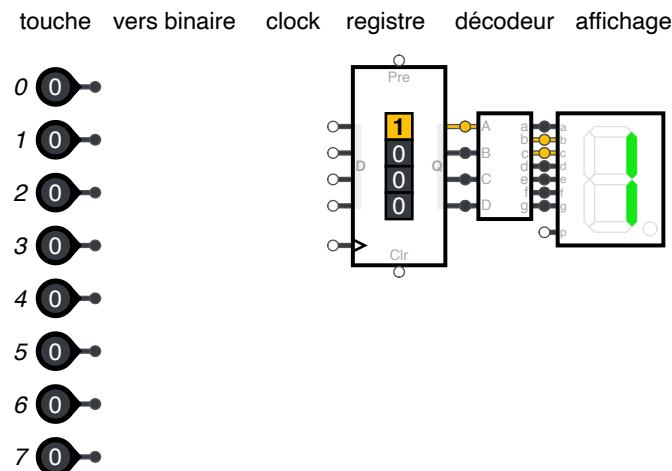
- une **touche 0 - 9** est appuyé
- une conversion en **binnaire** est faite
- une entrée **clock** est produite
- la valeur est mémorisée dans un **registre**
- la valeur est décodée vers les signaux a-g
- la valeur numérique est montrée sur l'**affichage** à 7 segments

Complétez le circuit pour traiter les touches 0 à 3



3.11.6 Décodeur pour 8 touches

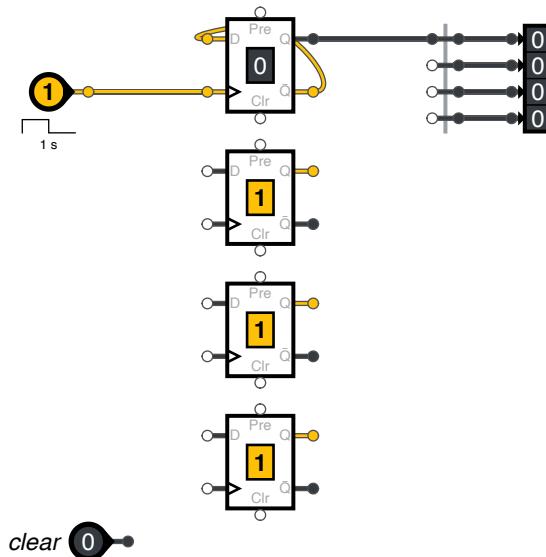
Complétez le circuit pour traiter les touches 0 à 7



3.11.7 Compteur 4 bits

Une bascule D avec une rétroaction de la sortie inversée \bar{Q} vers son entrée D divise la fréquence de l'horloge par 2. Nous avons effectivement un compteur 1 bit.

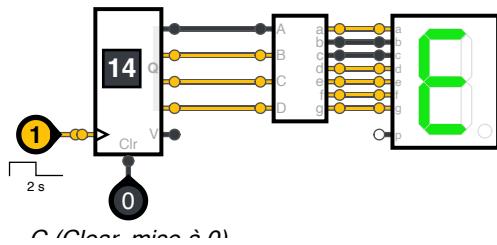
- Complétez le circuit pour construire un compteur 4 bits.
- Le circuit final doit compter de **0000** vers **1111**.
- Ajoutez un affichage 4 bits
- Connectez aussi les 4 signaux **clear** pour la remise du compteur



3.11.8 Compteur 8 bits

Le compteur 4 bits utilise un signal d'horloge et incrémenté à chaque coup d'horloge. Un décodeur à 7 segments transforme les 4 signaux qui représentent un nombre binaire de 0 à 16 vers les sorties correspondant pour activer les bonnes lampes de l'affichage à 7 segments.

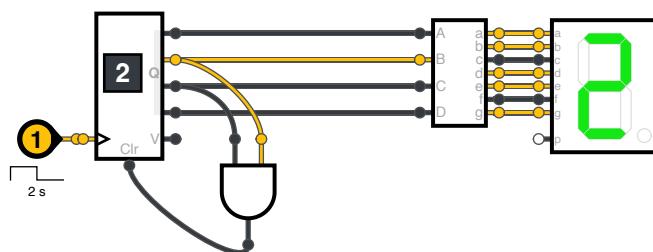
- Utilisez le signal de sortie V (overflow) pour faire fonctionner un deuxième compteur
- Ceci donnera un compteur 8 bit, permettant de compter de 00 à FF (255)
- Diminuez la période de l'horloge à 250 ms



3.11.9 Compteur avec remise

Pour créer une montre, un minuteur ou une alarme, nous devons compter à 60, 12 ou 24. L'entrée Reset peut être utilisée pour remettre le compteur. Une porte ET détecte le nombre 6 et remet le compteur

- Ajoutez un deuxième compteur
- Configurez-le pour qu'il compte de 0 à 9
- Ajoutez le décodeur et un affichage à 7 segments
- Utilisez les deux compteurs pour faire un compteur qui affiche les nombres 00 à 59
- Diminuez la période à 1 seconde



3.11.10 Rouler un dé

Un dé électronique utilise 7 LEDs pour afficher les points. Une horloge rapide, liée par une porte ET vers l'entrée d'un compteur compte rapidement de 0 à 5. À 6 le compteur est remis à 0 à l'aide d'une autre porte ET.

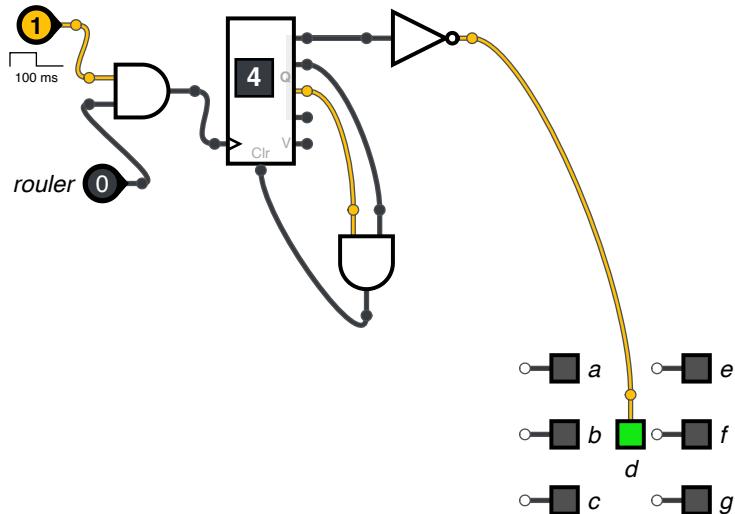
Voici la table de vérité pour les 7 LEDS.

dice	bin	a,g	b,f	c,e	d
1	000	0	0	0	1
2	001	1	0	0	0
3	010	1	0	0	1
4	011	1	0	1	0
5	100	1	0	1	1
6	101	1	1	1	0

Ajoutez les portes logiques appropriées pour implémenter ce dé électronique. Par exemple, le signal pour lampe a,g correspond à

b0 or b1 or b2

Trouvez les autres circuits et construisez ce dé électronique.

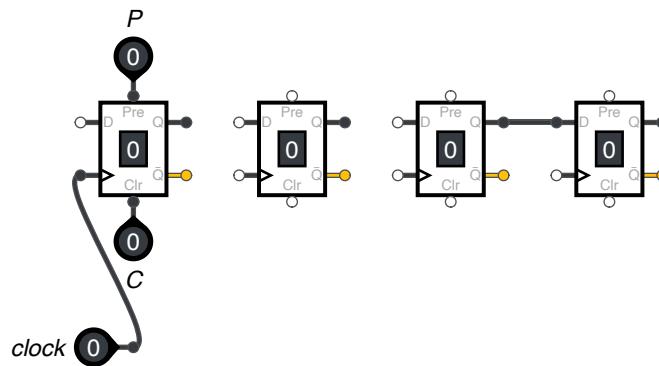


3.11.11 Registre à décalage

Un registre de décalage propage une information d'un registre vers l'autre.

- Placez tous les entrées P (preset) et C (clear)
- Connectez la sortie Q avec l'entrée D suivante
- Connectez tous les entrées clock
- Testez le circuit à décalage

Si ça fonctionne, connectez la sortie du registre à décalage avec son entrée et connectez une entrée horloge à 1 second à l'entrée clock. Vous avez un **registre à décalage circulaire**.

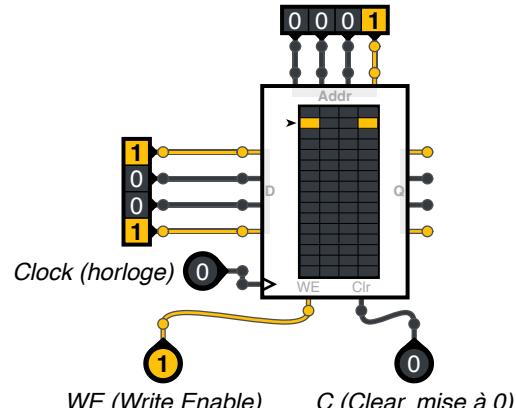


3.11.12 RAM (mémoire vive)

La RAM (Random Access Memory) de 16×4 bits permet de stocker 16 mots de 4 bits. Les quatre bits de l'adresse **Addr** déterminent l'endroit où seront écrites les données **D**.

- **Addr** détermine l'endroit des données
- **D** signifie les données à écrire
- **Q** représente les 4 bits des données lus
- **WE** (Write Enable) permet d'écrire si 1
- **clock** transmet les données sur D en mémoire
- **Clr** (clear) remet toute la mémoire à zéro

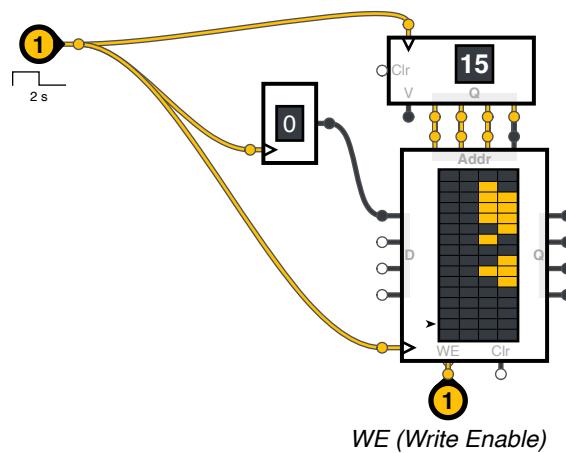
À l'adresse **0001** (1) se trouvent déjà les données **1001**. Ceci ressemble à des yeux. Ajoutez d'autres valeurs pour en faire un smiley.



3.11.13 RAM avec bits aléatoires

Le circuit suivant utilise un compteur pour créer les 16 adresses et écrire un bit aléatoire dans la RAM.

Complétez le circuit pour écrire 16 x 4 bits aléatoires dans la mémoire.

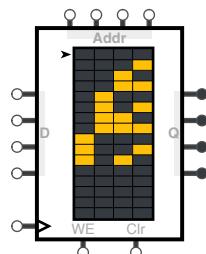


3.11.14 RAM avec binaire

Utilisez un compteur 4-bit pour remplir automatiquement la RAM avec les valeurs de 0 à 15. Voici le contenu à écrire dans la RAM.

```
0000  
0001  
0010  
...  
1111
```

Utilisez la sortie overflow (V) du compteur pour faire une mise à zéro de la RAM, à chaque fois quand elle est pleine. Dans l'image ci-dessous, la RAM est déjà remplie jusqu'à **1010** (10).

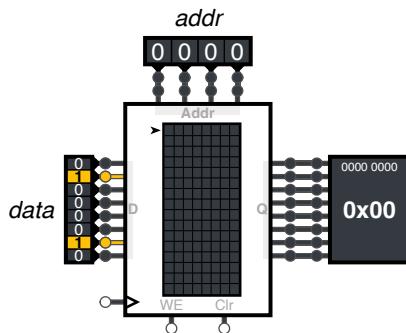


3.11.15 RAM avec image

La mémoire peut contenir des images. Chaque bit représente un pixel. Un des premiers jeux vidéo, **Space Invaders**, utilise des images monocouleurs pour présenter des envahisseurs extraterrestres.



Remplissez la mémoire avec l'image d'un *space invader* 8x8 bits. L'image sera alors visible dans la partie visualisation du bloc RAM 16x8 bits.



3.11.16 RAM avec ASCII

La mémoire peut contenir du code ASCII. Voici les codes ASCII des 6 lettres du mot ON AIR, exprimées en binaire et en hexadécimal.

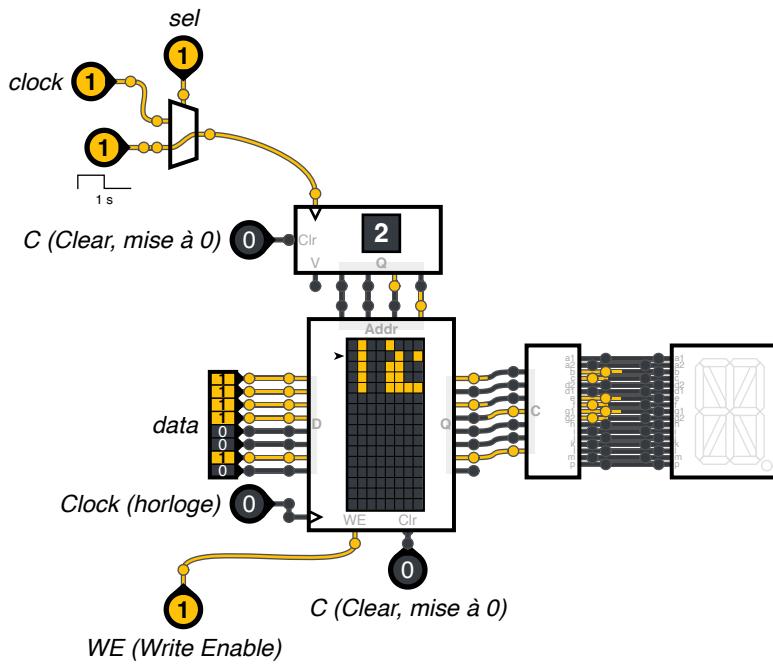
```

O = 0b01001111 0x4f
N = 0b01001110 0x4e
    = 0b00100000 0x20
A = 0b01000001 0x41
I = 0b01001001 0x49
R = 0b01010010 0x52

```

Le circuit ci-dessous contient le mot HELLO en mémoire RAM. Un compteur avec une horloge 1 Hz affiche le contenu de la mémoire en boucle vers un affichage à 16 segments.

Remplacez le contenu de la mémoire pour afficher le mot ON AIR.



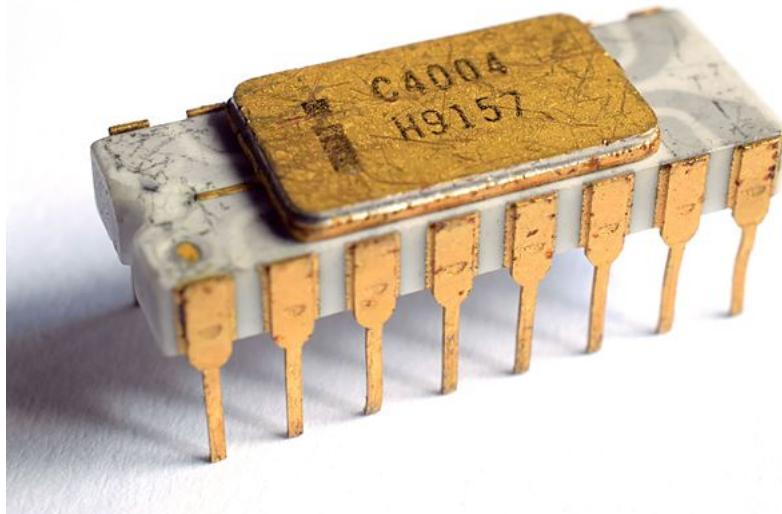
3.12 TP CPU

Le processeur, aussi appelé CPU (Central processing Unit) ou unité de traitement central, lit des instructions dans la mémoire programme et les exécute.

Tout ce que le processeur fait peut être décrit en 3 lignes :

- chercher une instruction dans la mémoire de programme (Fetch)
- exécuter cette instruction (Execute)
- incrémenter le pointeur vers la prochaine instruction (Increment)

Dans cette section nous allons étudier comment encoder des instructions en code binaire, et comment ensuite exécuter ce code dans le CPU. Nous allons nous inspirer du premier microprocesseur, la puce Intel 4004, sortie en 1971.

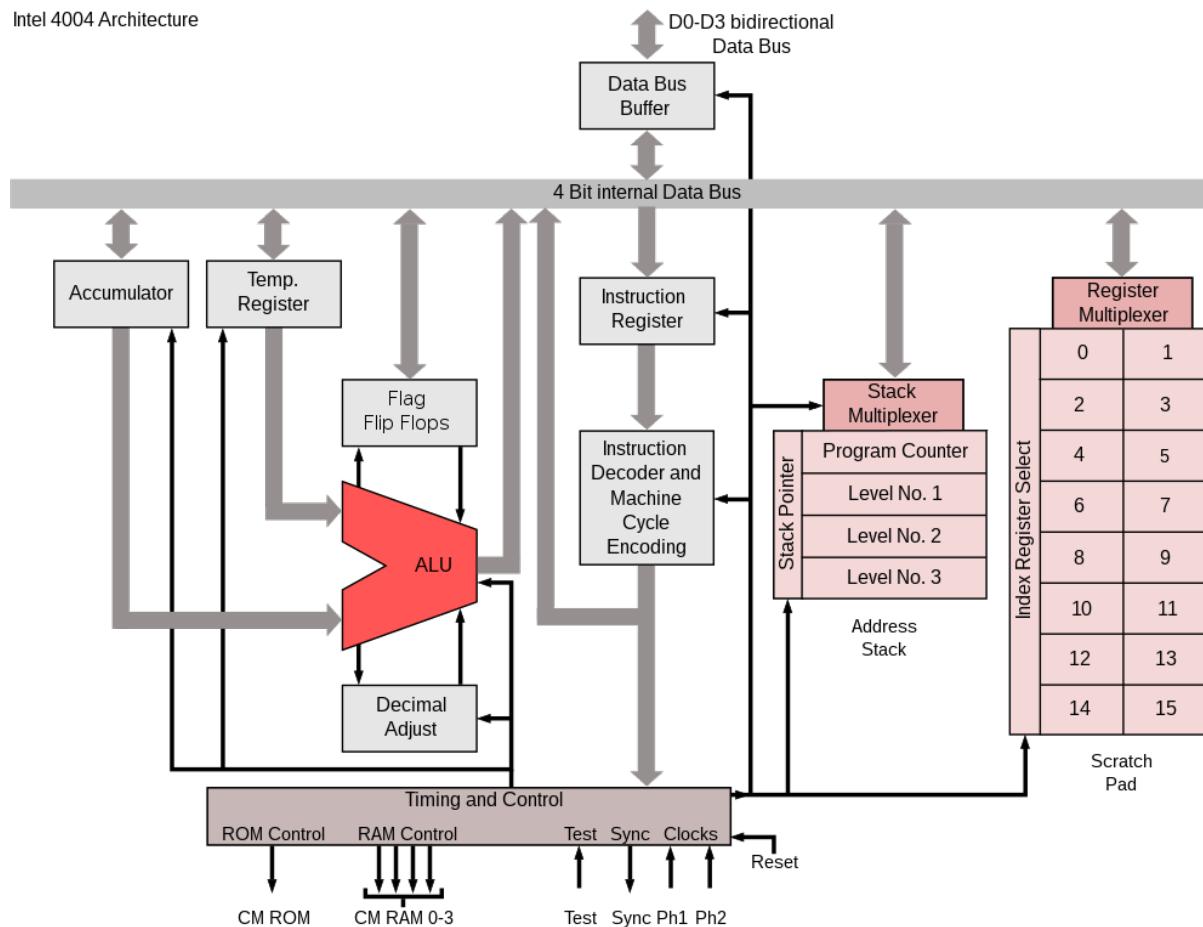


3.12.1 Intel 4004

Le premier CPU sur un seul circuit intégré fut le 4004 commercialisé par Intel en 1971. Il contenait les éléments suivants :

- une ALU (unité arithmétique et logique)
- 16 registres de travail
- un accumulateur (accumulator)
- un bus de données (data bus)
- un bus d'adresses (adress bus)
- des fanions (flags)
- un registre d'instruction (instruction register)
- un compteur de programme (program counter)
- un pointeur de pile (stack pointer)
- une pile (stack)
- une unité de contrôle (control unit)

Voici le schéma de l'architecture du 4004.



3.12.2 Langage assembleur

Nous allons commencer tout de suite avec un exemple de programme pour le 4004, en langage assembleur. Voici un bout de programme qui additionne deux nombres 4 bits.

```

ADD2
; add two 4bit numbers on the Intel 4004
;
    FIM P0, $A2 ; initialize: R0=2 R1=A
    LDR R0        ; load R0 into accumulator
    ADD R1        ; add R1 into accumulator
    XCH R1        ; and store in R1

```

Le point-virgule (;) sert comme symbole de commentaire. C'est l'équivalent du # en Python. Un programme en assembleur est typiquement structuré en 4 colonnes :

1. Une étiquette pour désigner une adresse de programme (ADD2)
2. Une mnémonique de l'opération (FIM, LDR, ADD, XCH)
3. Des données (P0, \$A2, R0, R1)
4. Des commentaires en fin de ligne

3.12.3 Le langage machine

Le langage machine, ou code machine, est la suite de bits qui est interprétée par le processeur d'un ordinateur exécutant un programme informatique. C'est le langage natif d'un processeur, c'est-à-dire le seul qu'il puisse traiter. Il est composé d'instructions et de données à traiter codées en binaire.

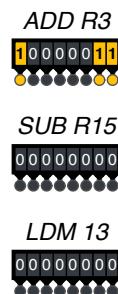
- 1000RRRR additionne (ADD) le registre RRRR à l'accumulateur
- 1001RRRR soustrait (SUB) le registre RRRR de l'accumulateur
- 1101DDDD charge (LDM = load) le nombre DDDD vers l'accumulateur

Le code machine est composée de :

- la partie instruction ou **opcode**, tel que 1000=ADD, 1001=SUB
- la partie donnée ou **data**, qui indique un registre RRRR où un nombre DDDD

Par exemple l'instruction en assembleur ADD R3 se traduit en code machine comme 10000011.

Trouvez les deux autres codes machine.



3.12.4 Mémoire de programme

Le CPU 4004 traite des données de 4 bits. Les données que ce processeur peut traiter avec une seule instruction sont limitées à une plage de 0 à 15 (de 0000 à 1111). On dit que c'est un processeur 4 bits ou une **architecture 4 bits**.

Chaque instruction par contre est encodée sur 8 bits. Ce processeur pourrait donc avoir au maximum 256 instructions différentes. En réalité il a 46 instructions.

Le vrai CPU 4004 peut adresser un espace mémoire de programme avec une adresse 12 bits. Ceci lui permet d'adresser un maximum de 2^{12} instructions différentes dans sa mémoire programme. Ici nous simplifions beaucoup et utilisons des adresses de 4 bits. Notre mémoire programme a une taille de 16 x 8 bits. Notre programme peut avoir un maximum de 16 instructions.

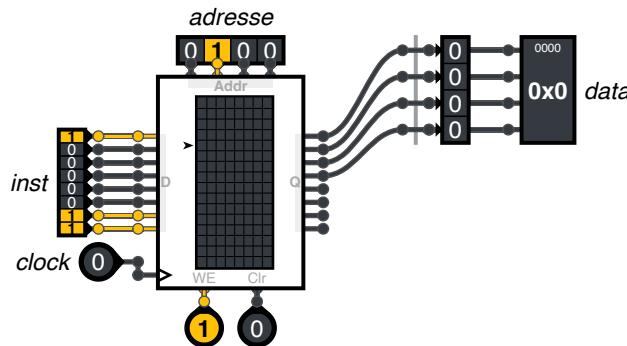
Beaucoup d'instructions sont composées d'une partie

- instruction (4 bits), appelée **opcode**
- données (4 bits), appelé **data**

Dans le circuit ci-dessous, ajoutez :

- une broche 4 bits
- une sortie 4 bits
- un affichage 4 bits (en hexadécimal)
- une étiquette **opcode**

- Mettez les instructions ADD R3, SUB R15 et LDM 13 dans les 3 premiers octets de la mémoire programme.



3.12.5 Le jeu d'instructions

On appelle **jeu d'instruction** (instruction set) la totalité des instructions qu'un processeur peut exécuter. Ces instructions sont représentées par une abréviation à 3 lettres (mnémonique). Les premières 14 instructions sont composées d'une partie :

- instruction (4 bits), appelée **opcode**, de 0000 à 1101
- données (4 bits), appelé **data**

La partie 'data' peu représenter 3 types de données :

- AAAA une adresse dans la mémoire programme
- RRRR un registre dans la banque des registres
- DDDD une donnée immédiate (un nombre de 0 à 15)

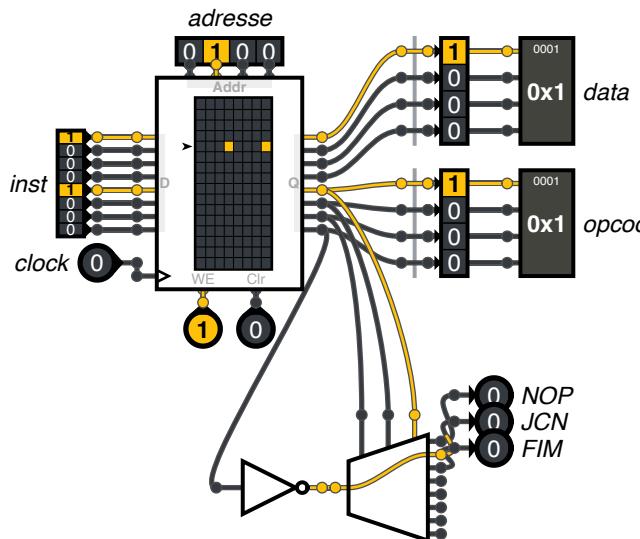
NOP 00000000	No Operation
JCN 0001AAAA	Jump Conditional
FIM 0010RRR0	DDDDDDDD Fetch Immediate
FIN 0011RRR0	Fetch Indirect
JUN 0100AAAA	Jump Unconditional
JMS 0101AAAA	Jump to Subroutine
INC 0110RRRR	Increment
ISZ 0111RRRR	AAAAAAAA Increment and Skip
ADD 1000RRRR	Add register
SUB 1001RRRR	Subtract register
LDR 1010RRRR	Load register
XCH 1011RRRR	Exchange register
BBL 1100DDDD	Branch Back and Load
LDM 1101DDDD	Load Immediate data

3.12.6 Décoder une instruction

Pour décoder les 14 instructions, nous pouvons utiliser un démultiplexeur. Pour compléter le circuit de décodage d'instruction :

- Ajoutez un deuxième démultiplexeur
- Ajoutez les 11 sorties manquantes
- Ajoutez les étiquettes (FIN à LDM)
- Remplissez la mémoire programme avec les 14 instructions (NOP à LDM)
- Vérifiez que chaque instruction est décodée correctement

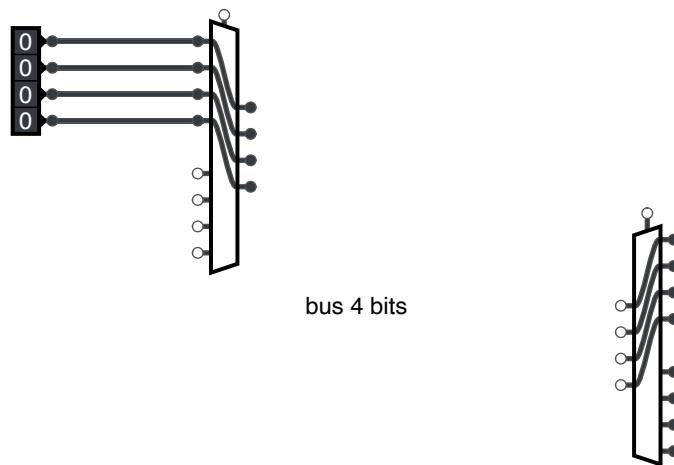
Par exemple à l'adresse **0100** (4) se trouve l'instruction **00010001** (JCN 1) et le décodeur active correctement la sortie JCN.



3.12.7 Bus 4 bits

Une ALU doit acheminer différents signaux sur une même ligne de transfert des données. On appelle un tel chemin un bus de données. Pour y connecter plusieurs sources, nous devons utiliser un multiplexeur.

- Ajoutez une deuxième entrée 4 bits
- Liez le multiplexeur et le démultiplexeur à travers une broche 4 bits
- Ajoutez les 2 entrées de sélection
- Ajoutez 4 affichages 4 bits pour montrer tous les signaux



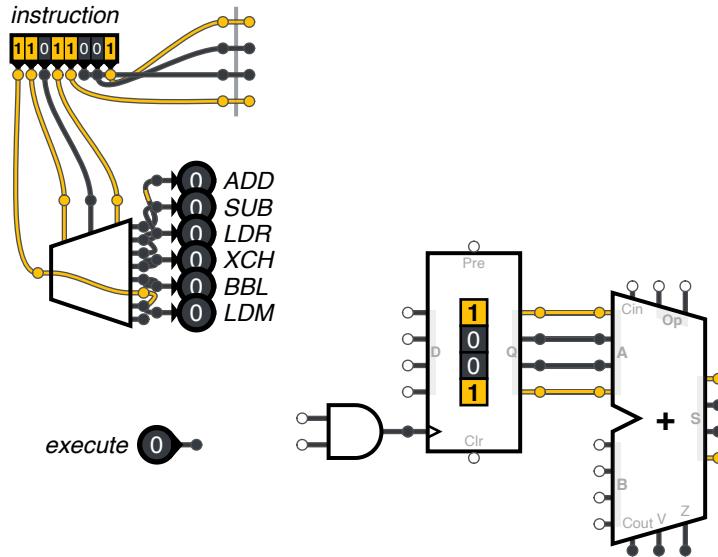
3.12.8 Charger imméd. (LDM)

La commande LDM (Load immediate) va charger une valeur directe (0-15), contenue dans le code de l'instruction, dans l'accumulateur.

1101DDDD

La partie 1101 est l'opcode (LDM) et la partie DDDD représente les 4 bits des données à charger dans l'accumulateur.

- Liez les bits b0-b3 avec l'accumulateur
- Utilisez la porte ET pour charger cette valeur dans l'accumulateur seulement si le signal **execute** est activé ET l'instruction LDM est décodée
- Placez un affichage à la sorte de l'accumulateur et à la sortie de l'ALU
- Mettez une valeur dans les bits b0-b3 et exécutez l'instruction avec **execute**



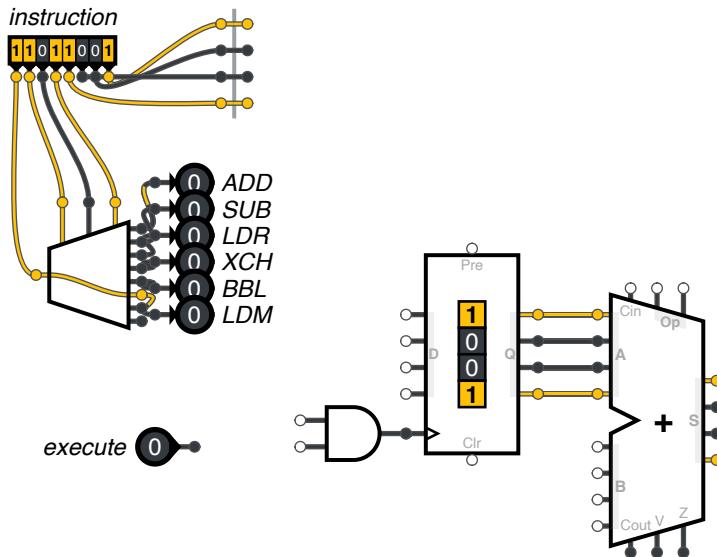
3.12.9 Charger depuis reg (LDR)

L'instruction LDR (load from register) charge l'accumulateur avec le contenu d'un des 16 registres.

1010RRRR

La partie **1010** est l'opcode (LD) et la partie **RRRR** représente un des 16 registres

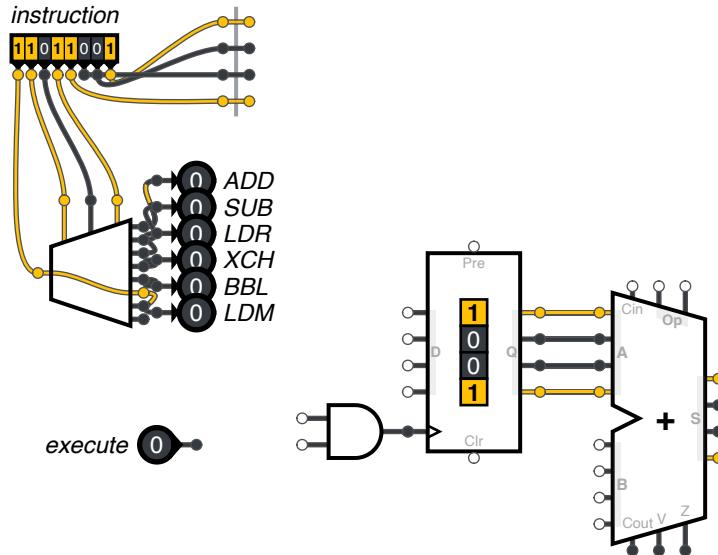
- Ajoutez la RAM avec les 16 registres
- Créez le circuit de décodage pour charger registre RRRR dans l'accumulateur
- Placez un affichage à la sortie de l'accumulateur et à la sortie de l'ALU
- Mettez une valeur dans R5
- Mettez l'instruction LDR R5 et exécutez l'instruction avec **execute**



3.12.10 Choix entre LDR/LDM

Avec les deux opcode différents, le circuit de décodage du CPU choisit un registre ou une donnée immédiate comme valeur à charger dans l'accumulateur.

- Ajoutez la RAM avec les 16 registres
- Ajoutez un multiplexeur 8x4
- Créez le circuit de décodage pour choisir entre un registre RRRR ou une donnée immédiate DDDD
- Placez un affichage à la sortie de l'accumulateur et à la sortie de l'ALU
- Mettez une valeur dans R5
- Mettez l'instruction LDR R5 et exécutez l'instruction avec **execute**
- Mettez l'instruction LDM 13 et exécutez l'instruction avec **execute**



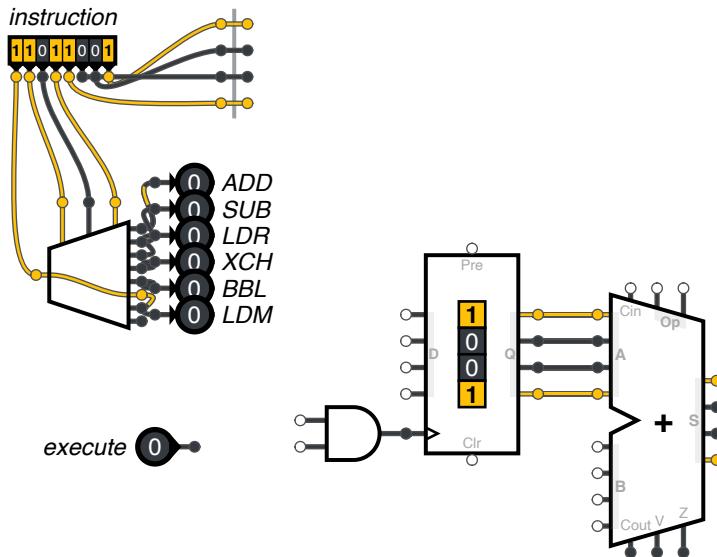
3.12.11 Choix entre ADD/SUB

L'addition et la soustraction se distinguent dans l'opcode d'un seul bit.

- 1000RRRR ADD additionner registre RRRR à l'accumulateur
- 1001RRRR SUB soustraire registre RRRR de l'accumulateur

Créez le circuit pour décoder et exécuter ces deux instructions.

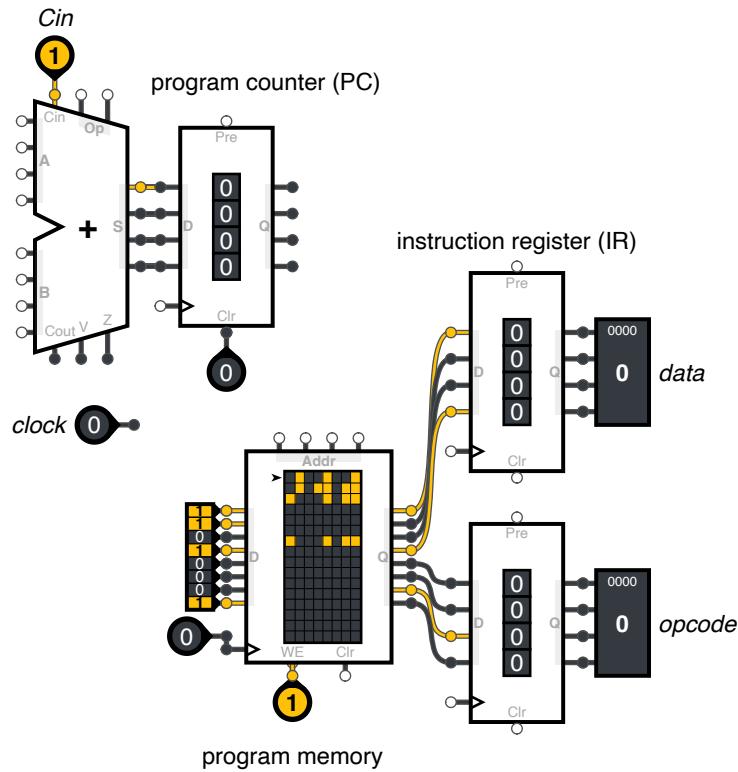
- Ajoutez la RAM avec les 16 registres
- Créez le circuit de décodage pour choisir entre ADD et SUB
- Placez un affichage à la sortie de l'accumulateur et nommez le **acc** (y mettre 9)
- Placez un affichage à la sortie de la RAM et nommez le **reg**
- Placez un affichage à la sortie de l'ALU et nommez le **result**
- Mettez la valeur 3 dans R5
- Mettez l'instruction ADD R5 et vous devriez avoir **result = 12** ($9 + 3$)
- Mettez l'instruction SUB R5 et vous devriez avoir **result = 6** ($9 - 3$)



3.12.12 Program counter (PC)

Le pointeur de programme, PC (program counter), pointe toujours à la prochaine instruction dans la mémoire de programme. Le contenu à l'adresse pointé par le PC est celui qui est chargé dans le registre d'instruction (IR) et exécuté au prochain pas.

- Liez la sortie du PC avec l'entrée A du l'ALU pour incrémenter de 1 à chaque pas
- Placez un affichage à la sortie du registre et nommez le **PC**
- Liez le PC avec l'entrée adresse de la mémoire de programme
- Liez l'entrée **clock** avec l'horloge de l'ALU et l'horloge des deux registres IR
- Faites avancer le PC et chargez des instructions successives dans le registre IR



3.12.13 Le saut (jump)

Le saut est une instruction qui permet de changer l'avancement linéaire du compteur de programme. L'instruction de saut a la forme :

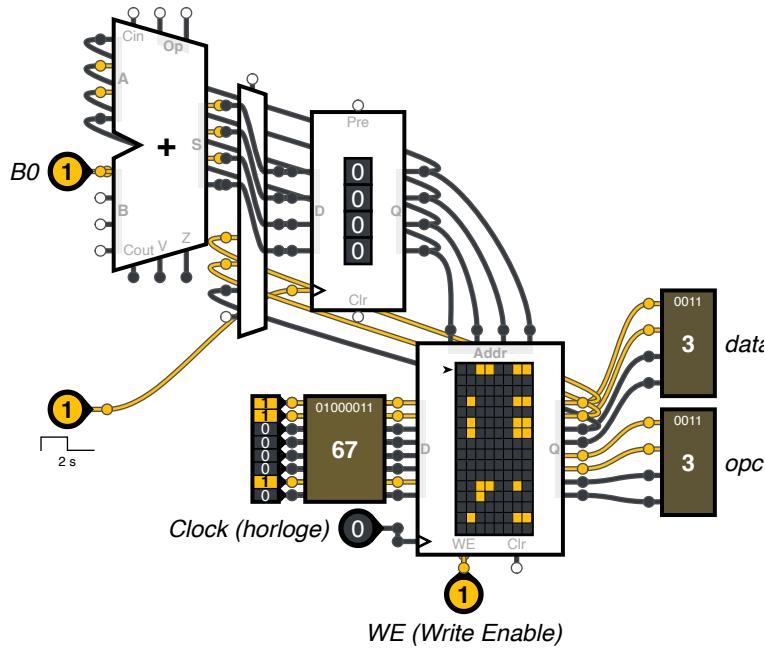
Jump Unconditional JUN 0100AAAA

Pour l'exécuter, le CPU doit d'abord détecter l'opcode 0100. Ceci peut être fait avec une porte ET à 4 entrées et des inverseurs.

Ensuite la valeur actuelle du compteur de programme doit être remplacée par la nouvelle adresse de destination du saut AAAA. Pour ceci nous utilisons un multiplexeur 8 vers 4.

Utilisez des portes NON et ET pour décoder l'opcode 0100 et l'utiliser pour sélectionner entre incrémentation normale et destination de saut.

A l'adresse 14 se trouve l'instruction 01000011 (JUN 3). Si le décodeur fonctionne correctement le programme va faire une boucle entre les addresses 3 et 14.



3.12.14 La pile (stack)

La pile est un espace de sauvegarde temporaire. Elle est utilisée pour sauvegarder les adresses de retour lors d'un saut vers une sous-routine.

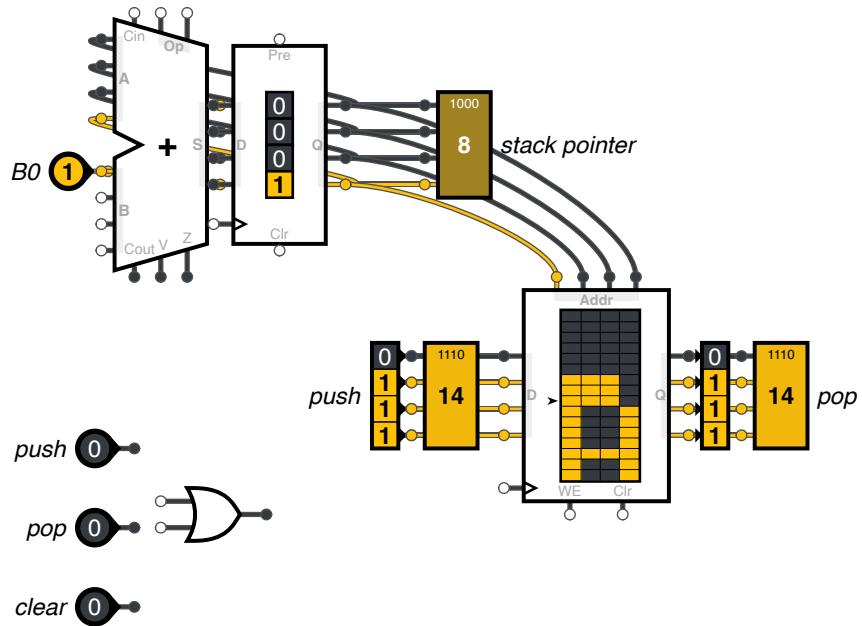
Ici nous créons une pile de 16 mots à 4 bits. D'habitude on commence la pile en bas de l'espace mémoire (adresse 15) et on *empile* les valeurs.

Le pointeur de pile (stack pointer) est un registre 4 bit, qui utilise une ALU pour être incrémenté ou décrémenté. Si l'entrée Op = 0 il incrémente. Si l'entrée Op = 1 il décrémente.

Ajoutez les circuits de contrôle.

- Le signal **clear** efface la pile et met le pointeur de pile à 1111 (tout en bas)
- Le signal **push** choisit la décrémentation (sp-) et envoie un coup d'horloge vers le registre du pointeur et la pile
- Le signal **pop** choisit l'incrémentation (sp++) et envoie un coup d'horloge vers le registre du pointeur et la pile

Mettez dans la pile 3141592 (les 7 premiers chiffres du nombre pi) avec l'instruction **push**. Ensuite, lisez ces chiffres dans l'ordre inverse avec l'instruction **pop**



3.12.15 Projet final

Ouvrez l'[éditeur logique](#)⁵⁶ dans une page entière du navigateur et choisissez un des projets suivants :

1. Complétez l'architecture du CPU 4004 pour en faire un processeur fonctionnel. Voici le [jeu d'instructions](#)⁵⁷ complet.
2. Créez une calculatrice avec les touches 0 à 10, les opérations + et -, et un affichage à 7 segments avec 4 chiffres ou plus. Le point décimal est optionnel.
3. Créez une horloge avec un affichage à 7 segments du style HH:MM:SS, avec des boutons **up/down** pour mettre le temps.
4. Créez un minuteur avec un affichage à 7 segments du style MM:SS qui décompte, avec des boutons **up/down** pour mettre le temps, et des boutons **start/stop/clear**.
5. Créez un chronomètre avec un affichage à 7 segments du style MM:SS . S qui affiche des dixièmes de seconde. Ajoutez des boutons **start/stop/clear**.

56. <https://logic.modulo-info.ch/>

57. <http://e4004.szyc.org/iset.html>

3.12.16 Nand Game

Dans le jeu [Nand Game](#)⁵⁸ vous allez construire un ordinateur à partir de composants de base.

Le jeu se compose d'une série de niveaux. Dans chaque niveau, vous êtes chargé de construire un composant qui se comporte selon une spécification. Ce composant peut ensuite être utilisé comme bloc de construction dans le niveau suivant.

Le jeu ne nécessite aucune connaissance préalable de l'architecture informatique ou des logiciels, et ne nécessite aucune compétence en mathématiques au-delà de l'addition et de la soustraction. (Cela demande un peu de patience - certaines tâches peuvent prendre un certain temps à résoudre !)

Votre première tâche est de créer un composant nand (Non-Et).

Bonne chance !

L'ENIAC, l'un des tout premiers ordinateurs opérationnels, conçu en 1945, à la fin de la Seconde Guerre mondiale, pour calculer des trajectoires de missiles, était constitué de 17468 tubes électroniques de la taille d'une main, qui cassaient en moyenne une fois par semaine. Il s'étendait sur 170 mètres carrés et pesait plus de 25 tonnes. Il était capable d'exécuter environ 5000 opérations par seconde.

Pour comparaison, les microprocesseurs des smartphones modernes exécutent de l'ordre de plusieurs centaines de *milliards* d'opérations par seconde.

En 1991, 1 Go (un milliard d'octets) de mémoire non volatile coûtait environ 45000 dollars. Aujourd'hui, un smartphone moderne dispose de l'ordre de 256 Go d'espace de stockage, ce qui aurait coûté à l'époque 11.5 millions de dollars¹.

À quoi servaient précisément 17468 tubes électroniques de l'ENIAC dans le rôle que l'ordinateur avait ? Et par quoi ont-ils été remplacés dans nos machines modernes ? Et comment se fait-il qu'un objet qui tient dans la poche puisse contenir 256 fois plus d'espace disque qu'un ordinateur des années 1990 ?

Dans ce chapitre, vous découvrirez comment sont construits les ordinateurs, comment sont organisés leurs différents composants pour leur permettre d'effectuer des milliards de calculs à la seconde alors qu'ils ne comprennent que la distinction entre 0 et 1, allumé ou éteint.

58. <https://nandgame.com/>

1. <https://www.aei.org/technology-and-innovation/the-a12-chip-estimating-innovation-with-iphone-prices/>

3.13 Objectifs

- Découvrir les **éléments de base** qui composent l'ordinateur.
- Comprendre les notions de **système logique** et de **microprocesseur**.
- Appréhender l'importance de **l'architecture des ordinateurs** pour optimiser les performances et effectuer des tâches informatiques spécifiques.

3.14 Personnages-clés

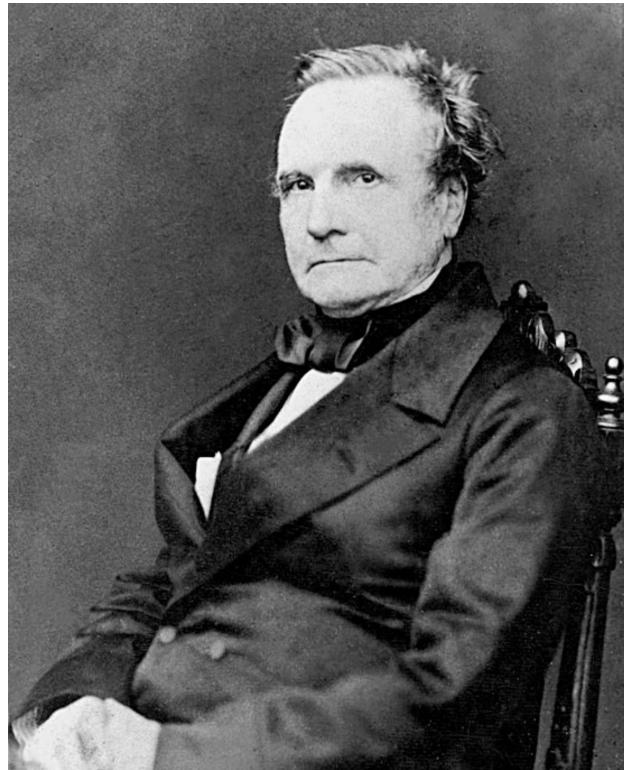


Anita Borg

1949-2003

Anita Borg⁵⁹ est une informaticienne américaine. Elle a notamment travaillé pour Digital Equipment Corporation où elle a développé une méthode permettant de concevoir des systèmes mémoriels à haute vitesse.

59. https://fr.wikipedia.org/wiki/Anita_Borg



Charles Babbage

1791-1871

Charles Babbage⁶⁰ fut le premier inventeur à énoncer le principe d'un ordinateur. C'est en 1834, pendant le développement d'une machine à calculer destinée au calcul et à l'impression de tables mathématiques, qu'il eut l'idée d'y incorporer des cartes du métier Jacquard, dont la lecture séquentielle donnerait des instructions et des données à sa machine.

60. https://fr.wikipedia.org/wiki/Charles_Babbage

Algorithmique II

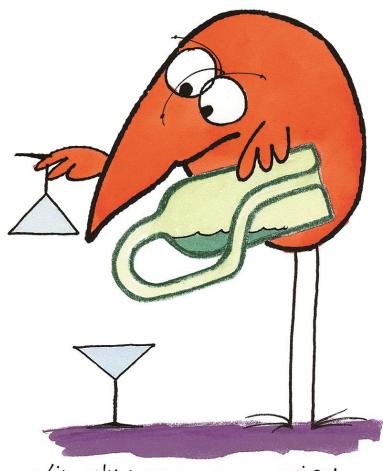
4.0 Introduction

4.0.1 Quoi ?

Pour résoudre un problème, il faut commencer par le décomposer en sous-problèmes. Pour chaque sous-problème à résoudre, on décrit les opérations à réaliser sous la forme d'un *algorithme*. Il existe une multitude d'*algorithmes* pour résoudre un problème, mais ils ne se valent pas tous.

L'**algorithmique** étudie les propriétés de ces *algorithmes*. Cette analyse est nécessaire pour nous aider à décider quel *algorithme* utiliser. On se propose à présent de passer en revue quelques propriétés importantes des *algorithmes*.

La devise Shadok du mois.



*S'IL N'Y A PAS DE SOLUTION
C'EST QU'IL N'Y A PAS DE PROBLÈME.*

4.0.2 Pourquoi ?

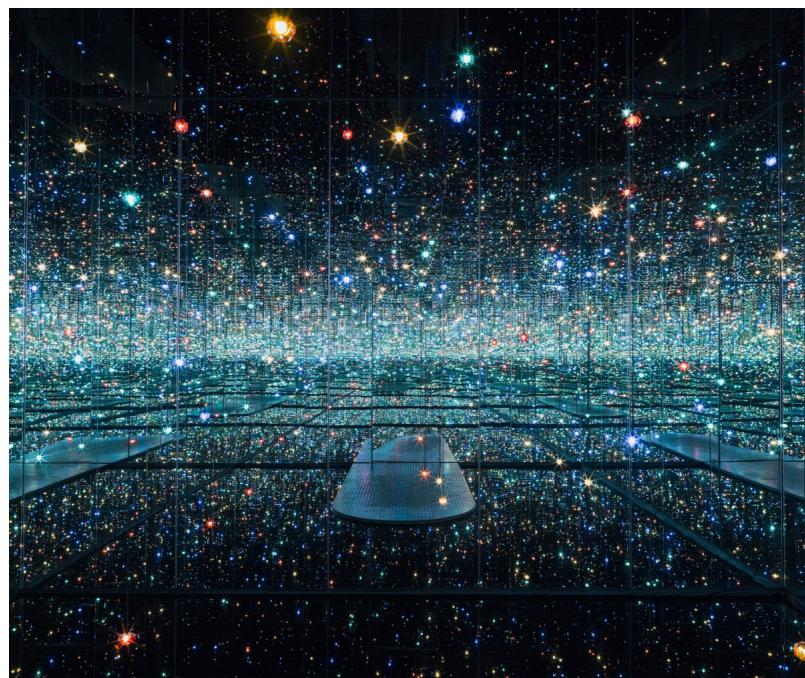
Si tous les chemins mènent à Rome, on ne peut en emprunter qu'un. Lorsqu'on est face à plusieurs chemins pour arriver au même résultat, il est important de choisir le chemin le plus optimal.

Vous avez déjà rencontré plusieurs algorithmes pour arriver jusqu'ici. Encore plus fort, vous avez rencontré plusieurs algorithmes pour résoudre un même problème, ce qui nous met face à un dilemme : quelle algorithme choisir ? Et y a-t-il une solution à tout problème ?

4.0.3 Comment ?

Dans un premier temps nous allons nous intéresser à la notion de complexité : comment déterminer la vitesse d'un algorithme ? Si plusieurs *bonnes* solutions existent, alors il faut choisir la plus rapide. Mais sera-t-elle toujours la solution la plus rapide ?

Dans un deuxième temps, si vous le souhaitez, vous pouvez ouvrir la porte merveilleuse de la récursivité, à la manière des *Infinity Mirror Room* de Yayoi Kusama.



4.0.4 Objectifs

A la fin de ce chapitre, vous saurez ce qui fait qu'un algorithme est un bon algorithme et quels critères prendre en considération pour choisir le meilleur algorithme. Vous verrez également qu'il existe des problèmes relativement simples que l'on n'arrive toujours pas à résoudre.

- Pouvoir déterminer quelle est la meilleure solution pour un problème donné, en fonction de critères objectifs.
- Comprendre pourquoi certains problèmes simples n'ont pas de solution exacte.
- [Optionnel] Créer des fonctions récursives, qui s'appellent elles-mêmes.

4.1 Terminaison et complexité

Matière à réfléchir – Compte à rebours

Voici une version naïve du compte à rebours des secondes pour le passage du Nouvel An :

```
# Compte à rebours
1
def compte_a_rebours(nb_secondes) :
2
    while True :
3
        print(nb_secondes)
4
        nb_secondes = nb_secondes - 1
5
6
compte_a_rebours(10)
7
```

Qu'arrive-t-il lorsqu'on exécute ce *programme* ?

Corriger le programme pour qu'il s'arrête à 0.

Qu'arrive-t-il lorsque l'on exécute la nouvelle version du programme avec la valeur -10 en entrée ou `compte_a_rebours(-10)` ?

4.1.1 Principe de terminaison

La **terminaison** est une propriété essentielle des *algorithmes*, qui garantit que les calculs de l'algorithme finiront par s'arrêter. Lorsque l'on conçoit un algorithme, il est important de faire en sorte que les calculs s'arrêtent à un moment donné. Cette garantie doit tenir pour toutes les entrées possibles. Voici un exemple d'algorithme qui compte et qui ne se termine pas :

```
# Algorithme qui compte infini
Variable i : numérique
i ← nombre donné par l'utilisateur
Tant que i > 0
    i ← i + 1
    Afficher i
Fin Tant que
```

Si on exécute cet *algorithme*, le *programme* ne s'arrête jamais : *i* est *incrémenté* de 1 indéfiniment. En pratique, si on retranscrit cet algorithme en programme et que l'on exécute le programme, le programme finira par s'arrêter lorsque les nombres représentés seront trop grands pour être représentés.

Exercice 1 – L'infini en programme

Retranscrire l'algorithme infini en programme. Après combien de boucles le programme s'arrête ?

Solution 1 – L'infini en programme

La solution de l'exercice est donnée directement dans le texte qui suit.

Pour faire en sorte que le programme finisse par s'arrêter, nous pouvons le modifier ainsi :

```
# Algorithme qui compte toujours infini
Variable i : numérique
i ← nombre donné par l'utilisateur
Tant que i != 10000
    i ← i + 1
    Afficher i
Fin Tant que
```

Exercice 2 – L'infini ne finit plus de finir

L'algorithme ci-dessus est appelé «Algorithme qui compte toujours infini». Pourquoi est-il toujours infini ? Dans quel cas cet algorithme ne s'arrête jamais ?

Solution 2 – L'infini ne finit plus de finir

La solution de l'exercice est donnée directement dans le texte qui suit.

Dans la version ci-dessus, si l'utilisateur entre une valeur plus grande que 10000, ou encore une valeur à virgule, l'algorithme ne s'arrête pas. Il peut être implicite pour la personne qui programme qu'un décompte se fait toujours avec des nombres entiers, mais il doit prendre des précautions face aux utilisateurs. Voici une version de l'algorithme de décompte qui s'arrête dans tous les cas :

```
# Algorithme qui compte et qui s'arrête
Variable i : numérique
i ← nombre donné par l'utilisateur
Tant que i < 10000
    i ← i + 1
    Afficher i
Fin Tant que
```

En programmant, nous devons nous assurer que nos programmes se terminent dans tous les cas, autrement il ne seront pas utilisables. Il ne suffit pas de compter sur la bienveillance des utilisateurs.

Le saviez-vous ? – Conjecture de Syracuse

De nos jours, on ne sait toujours pas si ce programme termine pour chaque entrée n . Ce problème est connu sous le nom la *conjecture de Collatz* ou *la conjecture de Syracuse* :

```

def Collatz(n) :
    while n > 1 :
        if n % 2 == 0 :
            n = n / 2
        else :
            n = 3 * n + 1
        print(n, '\n')

Collatz(4)

```

4.1.2 Principe de complexité

Matière à réfléchir – Record de vitesse

On souhaite comparer deux algorithmes qui permettent de résoudre le même problème, afin d'utiliser l'algorithme qui permet de résoudre le problème plus rapidement. Mais comment pourrait-on calculer la vitesse d'un algorithme ?

Il est important lorsqu'on utilise un *algorithme* de nous préoccuper de son *efficacité*. Mais comment calculer l'efficacité d'un algorithme, comment calculer sa vitesse ?

Est-ce qu'on peut utiliser la taille de l'algorithme pour prédire le temps qu'il va prendre à s'exécuter ? En d'autres termes, est-ce qu'un algorithme de 10 lignes est toujours plus lent qu'un algorithme de 5 lignes ? Nous avons vu que l'algorithme infini du chapitre précédent est très court (seulement 5 lignes), mais en théorie il ne s'arrête jamais. Une *boucle* rallonge le code de seulement 2 lignes, mais rallonge le temps d'exécution de manière importante.

On pourrait croire qu'il suffit de programmer un algorithme et de chronométrier le temps que ce programme prend à s'exécuter. Cette métrique est problématique, car elle ne permet pas de comparer différents algorithmes entre eux lorsqu'ils sont exécutés sur différentes machines. Un algorithme lent *implémenté* sur une machine dernière génération pourrait prendre moins de temps à s'exécuter qu'un algorithme rapide implémenté sur une machine datant d'une dizaine d'années.

Pour mesurer le temps d'exécution (ou la vitesse) d'un algorithme, il existe un critère plus objectif : le **nombre d'instructions élémentaires**. De manière formelle et rigoureuse, on ne parle pas d'efficacité, mais plutôt de la **complexité d'un algorithme**, qui est en fait contraire à son efficacité. L'analyse de la complexité d'un algorithme étudie la quantité de ressources, par exemple de temps, nécessaires à son exécution.

Le saviez-vous ? – Compliqué

Est-ce que *complexe* veut dire la même chose que *compliqué*? Une chose compliquée est difficile à saisir ou à faire, alors qu'une chose complexe est composée d'éléments avec de nombreuses interactions imbriquées.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais que l'on doit garantir la terminaison d'un algorithme.
2. Je sais que la complexité d'un algorithme peut nous donner une indication de sa vitesse.
3. Je sais que la complexité est une fonction du nombre d'instructions élémentaires.

4.2 Algorithmes de recherche

Les ordinateurs passent la majorité de leur temps à chercher : ils cherchent des fichiers, ils cherchent des sites Web, ils cherchent des informations dans un site Web, ils cherchent votre compte lorsque vous vous loguez sur un site Web, ils cherchent des *posts* à vous montrer et des personnes à qui vous connecter. Nous allons commencer par estimer la complexité de deux algorithmes de recherche importants.

La complexité ne reflète pas la difficulté à implémenter un algorithme, comme on pourrait le croire, mais à quel point l'algorithme se complexifie au fur et à mesure que le nombre des entrées augmente. La complexité mesure le temps d'exécution d'un algorithme (ou sa rapidité) en termes du nombre d'instructions élémentaires exécutées en fonction de la taille des données.

4.2.1 Recherche linéaire

La manière la plus simple pour rechercher un élément dans un tableau (une liste en Python) consiste à parcourir le tableau et à comparer l'élément recherché à tous les éléments du tableau :

```
# Algorithme de recherche&nbsp;linéaire

Tableau Eléments          # données stockées dans un tableau (une liste en Python)
n ← longueur(Eléments)    # la variable n contient le nombre d'éléments
élément_recherché ← entrée # l'élément recherché est un paramètre de l'algorithme
i ← 1                      # index pour parcourir la liste

Répéter Pour i = 1 à n
    si Eléments[i] == élément_recherché
        Retourner « Oui »
    Fin Répéter

Retourner « Non »
```

Quelle est la complexité de cet *algorithme de recherche linéaire*? Pour répondre à cette question, il faut estimer le nombre d'*instructions* élémentaires nécessaires pour rechercher un élément dans un tableau. Cenombre dépend naturellement de la taille du tableau n : plus le nombre d'éléments dans le tableau est grand, plus on a besoin d'instructions pour retrouver un élément.

Supposons que le tableau contienne 10 éléments. Pour trouver l'élément recherché, il faut au moins deux *instructions* par élément du tableau : une instruction pour accéder à l'élément du tableau (ou `Elements[i]`) et une autre instruction pour le comparer à `élément_recherché`. Dans le cas du tableau à 10 éléments, cet algorithme prendrait 20 instructions élémentaires, 2 (*instructions*) multiplié par le nombre d'éléments dans le tableau. Mais si le tableau contient 100 éléments, le nombre d'*instructions* élémentaires monte à 200. De manière générale, si le nombre d'éléments dans le tableau est n , cela prend $2n$ *instructions* pour le parcourir et pour comparer ses éléments.

Cette estimation n'est pas exacte. Nous n'avons pas pris en compte les instructions élémentaires qui permettent d'incrémenter `i` et de vérifier si `i == longueur(Nombres)`. Lorsqu'on prend en compte ces 2 instructions supplémentaires liées à l'utilisation de `i`, le nombre d'*instructions* passe de 200 à 400 (ce qui correspond à $4n$). Il faut également y ajouter les 4 *instructions* d'initialisation avant la *boucle*, plus l'*instruction* de retour à la fin de l'algorithme, ce qui fait monter le nombre d'*instructions* de 400 à 405

ou $(4n + 5)$. Attention, le nombre exact peut être différent, car il dépend de l'implémentation sur la machine. Mais, ce qui nous intéresse ici n'est pas tant le nombre exact d'instructions, si c'est 205 ou 410 ou 815. Ce qui nous intéresse ici est plutôt l'**ordre de grandeur** de l'algorithme ou comment le nombre d'instructions élémentaires grandit avec la taille du tableau n .

Cet algorithme est de complexité **linéaire**. Une complexité linéaire implique que le nombre d'instructions élémentaires croît linéairement en fonction du nombre d'éléments des données : $cn + a$, où c et a sont des *constantes*. Dans ce cas précis, c vaut 4 et a vaut 5. Si le tableau contient 10 éléments, il faut environ 45 instructions ; pour 100 éléments il faut environ 405 instructions ; pour 1000 éléments il faut environ 4005 instructions et ainsi de suite. Le nombre d'instructions grandit de manière linéaire en fonction de la taille des données n , et cette relation est représentée par une ligne lorsqu'on la dessine (voir la figure ci-dessous).

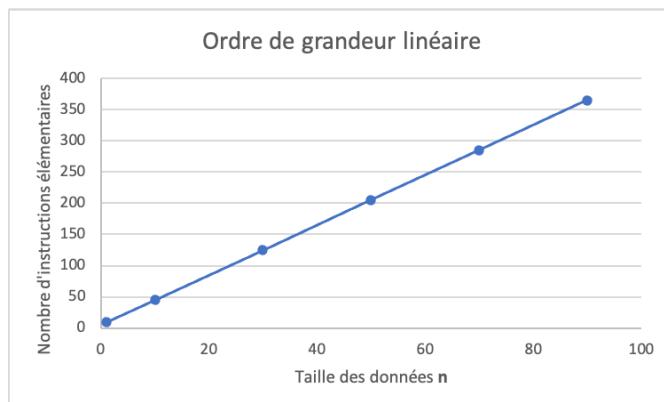


FIG. 4.1 – Complexité linéaire. La complexité de l'algorithme de recherche linéaire, comme son nom l'indique, est linéaire. La relation entre la taille du tableau n et le nombre d'instructions nécessaires pour retrouver un élément dans ce tableau représente une ligne.

Exercice 1 – Compter jusqu'à n

Ecrire un algorithme qui affiche tous les nombres de 1 à n .

Combien d'instructions élémentaires sont nécessaires lorsque n vaut 100 ?

Quelle est la complexité de cet algorithme ?

Solution 1 – Compter jusqu'à n

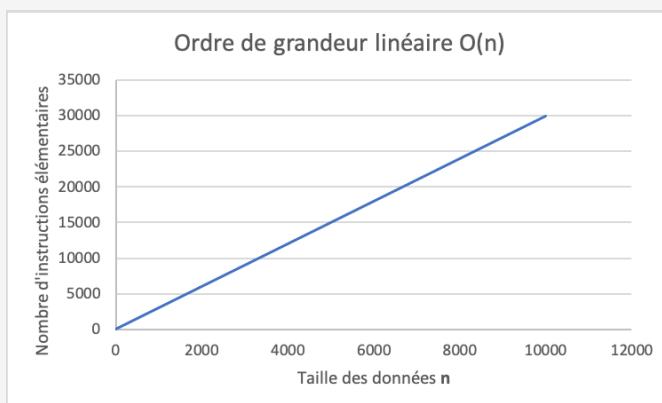
```
Variable n : numérique
Variable i : numérique
```

```
Répéter Pour i = 1 à n
    Afficher(i)
Fin Pour
```

L'initialisation des variables n et i compte pour 2 instructions élémentaires. Chaque passage de la boucle correspond à trois instructions élémentaires : 1 instruction qui affiche i , 1 instruction qui incrémente i de 1 et finalement une instruction qui compare i à n (pour savoir si la boucle s'arrête ou si elle continue). Le total d'instructions élémentaires pour le cas où n vaut 100 est $3 * 100 + 2$ ou 302 instructions élémentaires.

Il faut se rendre compte que cette estimation du nombre d'instructions élémentaires est approximatif, et non pas exact. Par exemple, l'instruction élémentaire **Afficher(i)** englobe certainement plusieurs instructions à l'exécution et prend de plus en plus de temps à mesure que i grandit (affiche de plus en plus de caractères).

La complexité (ou l'ordre de grandeur) de cet algorithme est linéaire, comme illustré dans ce graphique :



Exercice 2 – Compter par pas de 2

Ecrire un algorithme qui affiche tous les nombres *pairs* de 1 à n .

Combien d'instructions élémentaires sont nécessaires lorsque n vaut 100 ?

Quelle est la complexité de cet algorithme ?

Solution 2 – Compter par pas de 2

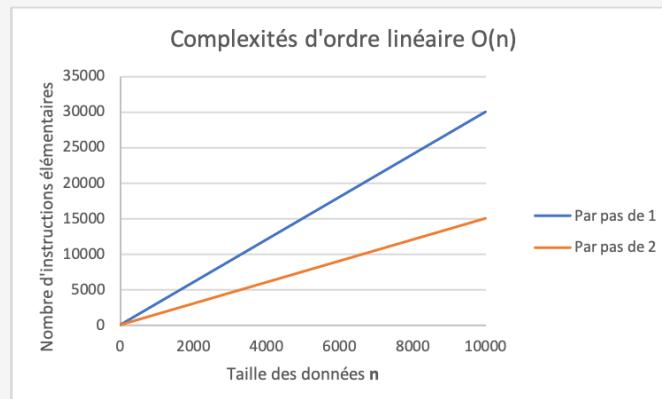
```
Variable n : numérique
Variable i : numérique
```

```
Répéter Pour i = 2 à n, par pas de 2
    Afficher(i)
Fin Pour
```

La seule ligne qui change par rapport à la solution de l'exercice précédent est l'incrément de la boucle par pas de 2.

L'initialisation des variables i et n compte pour 2 instructions élémentaires. Chaque passage de la boucle correspond à trois instructions élémentaires : 1 instruction qui affiche i , 1 instruction qui incrémente i de 2 et finalement 1 instruction qui compare i à n (pour savoir si la boucle s'arrête ou si elle continue). Pour le cas où n vaut 100, la boucle sera parcourue 50 fois (par pas de 2). Le total d'instructions élémentaires est donc $3 * 50 + 2$ ou 152 instructions élémentaires.

La complexité (ou l'ordre de grandeur) de cet algorithme est également linéaire, comme illustré dans le graphique. Il faut noter que l'ordre de grandeur est le même que pour l'exercice précédent, seule la vitesse de croissance change.



La différence de croissance se cache dans la constante c de l'ordre de grandeur $cn + a$. La valeur de c dans l'exercice précédent est supérieure à la valeur de c dans cet exercice. Dans un premier temps, on peut ignorer la valeur de cette constante c . Cependant elle peut devenir importante lorsque l'on doit comparer des algorithmes du même ordre entre eux.

Exercice 3 – Recherche linéaire

Programmer l'algorithme de recherche linéaire en Python. Rechercher une valeur entre 1 et 1000000 dans un tableau qui contient les valeurs allant de 1 à 1000000.

Pour quelle valeur l'algorithme est le plus rapide ? Le plus lent ? Indice : utiliser le module `time` pour chronométrer le temps d'exécution.

Solution 3 – Recherche linéaire

Voici un programme possible pour l'algorithme de recherche linéaire. Si on chronomètre le temps d'exécution, la valeur 1 est trouvée très rapidement en comparaison à la valeur 1000000. C'est beaucoup plus lent de rechercher une valeur qui se trouve à la fin du tableau, qu'au début du tableau.

```
# algorithme de recherche linéaire
def search_lin(search_list, search_element, verbose=0) :

    # boucle pour parcourir la liste
    for element in search_list :
```

1
2
3
4
5

```
if verbose :  
    print("L'élément comparé est : " + str(element) + "\n")  
  
# l'élément de la liste correspond à l'élément recherché  
if element == search_element :  
    return True  
  
# aucun élément ne correspond  
return False  
  
import time  
  
last = 1000000  
ma_liste = list(range(1,last+1))  
  
# mettre verbose à 1 pour avoir une vue de ce qui se passe  
# attention, cela fausse les temps de calcul (plus temps d'affichage)  
verbose = 0  
  
# chronomètre le temps de recherche de l'élément 1000000  
time_start = time.time()  
search_lin(ma_liste, last, verbose)  
time_1000000 = time.time() - time_start  
print("Recherche linéaire 1000000 : " + str(round(time_1000000,7)) + "  
secondes.\n")  
  
# chronomètre le temps de recherche de l'élément 1  
time_start = time.time()  
search_lin(ma_liste, 1, verbose)  
time_1 = time.time() - time_start  
print("Recherche linéaire 1 : " + str(round(time_1,7)) + " secondes.")
```

Exercice 4 – Recherche linéaire (dans le désordre)

On a vu dans l'exercice précédent que cela prend moins de temps pour trouver un élément au début de la liste qu'un élément de la fin de la liste. Qu'est-ce qui arrive si les éléments de la liste ne sont pas dans l'ordre ? Essayer en utilisant la fonction `shuffle()` du module `random`.

Solution 4 – Recherche linéaire (dans le désordre)

Si on mélange l'ordre des éléments, la valeur 1 pourrait se retrouver en fin de tableau et pourrait prendre très longtemps à retrouver. Au contraire, la valeur 1000000 pourrait se retrouver en début de tableau et pourrait prendre très peu de temps à retrouver. La situation est donc beaucoup moins prévisible. Le code ci-dessous est à utiliser avec la fonction `search_lin()` de la solution précédente.

```

1 import time
2 import random
3
4 last = 1000000
5 ma_liste = list(range(1,last+1))
6 # mélange les éléments du tableau au hasard
7 random.shuffle(ma_liste);
8
9 # mettre verbose à 1 pour avoir une vue de ce qui se passe
10 # attention, cela fausse les temps de calcul (plus temps d'affichage)
11 verbose = 0
12
13 # chronomètre le temps de recherche de l'élément 1000000
14 time_start = time.time()
15 search_lin(ma_liste, last, verbose)
16 time_1000000 = time.time() - time_start
17 print("Recherche linéaire 1000000 : " + str(round(time_1000000,7)) + " secondes.\n")
18
19 # chronomètre le temps de recherche de l'élément 1
20 time_start = time.time()
21 search_lin(ma_liste, 1, verbose)
22 time_1 = time.time() - time_start
23 print("Recherche linéaire 1 : " + str(round(time_1,7)) + " secondes.")

```

Pour aller plus loin

Pour décrire mathématiquement les ordres de complexité, on utilise la notation « Grand O ». Pour dire qu'un ordre de complexité est linéaire, on écrit par exemple qu'il est $O(n)$.

4.2.2 Recherche binaire

Matière à réfléchir – Le dictionnaire

Comment faites-vous pour rechercher un mot dans un dictionnaire ?

Utilisez-vous l'algorithme de recherche linéaire, parcourez-vous tous les éléments un à un ?

Quelle propriété du dictionnaire nous permet d'utiliser un autre algorithme de recherche que l'algorithme de recherche linéaire ?

Si on doit rechercher un élément dans un tableau qui est **déjà trié**, l'*algorithme* de recherche linéaire n'est pas optimal. Dans le cas de la recherche d'un mot dans un dictionnaire, la recherche linéaire implique que l'on va parcourir tous les mots du dictionnaire dans l'ordre, jusqu'à trouver le mot recherché. Mais nous ne recherchons pas les mots dans un dictionnaire de la sorte. Nous exploitons le fait que les mots du dictionnaire sont triés dans un ordre alphabétique. On commence par ouvrir le dictionnaire sur une page au hasard et on regarde si le mot recherché se trouve avant ou après cette page. On ouvre ensuite une autre page au hasard dans la partie retenue du dictionnaire. On appelle cette méthode la **recherche binaire** (ou la recherche dichotomique), car à chaque itération elle *divise l'espace de recherche en deux*. En effet, à chaque nouvelle page ouverte, nous éliminons environ la moitié du dictionnaire qui nous restait à parcourir. Voici une description de l'algorithme de recherche binaire :

```
# Algorithme de recherche&nbsp;binaire

Tableau Eléments      # les données du tableau (liste en Python) sont triées
n <- longueur(Eléments)    # la variable n contient le nombre d'éléments
élément_recherché <- entrée # l'élément recherché est un paramètre de l'algorithme
trouvé <- Faux           # indique si l'élément a été retrouvé
index_début <- 0          # au début on cherche dans tout le tableau
index_fin <- n            # au début on cherche dans tout le tableau

# tant que l'élément n'est pas trouvé et que le sous-tableau retenu n'est pas vide
Tant que trouvé != Vrai et n > 0 :

    # on identifie le milieu du sous-tableau retenu
    index_milieu = (index_fin - index_début)/2 + index_début

    # si l'élément recherché correspond à l'élément du milieu du sous-tableau retenu
    if Eléments[index_milieu] == élément_recherché :
        trouvé = Vrai
    else :

        # si l'élément du milieu est plus grand que l'élément recherché
        # on retient la première moitié comme sous-tableau
        if Eléments[index_milieu] > élément_recherché :
            index_fin = index_milieu

        # si l'élément du milieu est plus petit que l'élément recherché
```

(suite sur la page suivante)

(suite de la page précédente)

```

# on retient la deuxième moitié comme sous-tableau
else :
    index_début = index_milieu+1

# calcule le nombre d'éléments du sous-tableau retenu
n = index_fin - index_début

Fin Tant que

Retourner trouvé

```

Prenons le temps d'étudier cet *algorithme*. Que fait-il ? La *boucle Tant que* permet de parcourir le tableau `Eléments` et d'y rechercher `élément_recherché` tant qu'il n'est pas trouvé (tant que `trouvé != Vrai`). A la première itération (au premier passage dans la *boucle*, on vérifie si l'élément au milieu du tableau `Eléments` n'est justement pas l'élément recherché. Les éléments de la liste étant triés, si l'élément au milieu du tableau est plus grand que l'élément recherché, cela indique que `élément_recherché` se trouve dans la première partie du tableau. Si l'élément au milieu du tableau est plus petit que l'élément recherché, cela indique que l'élément recherché se trouve au contraire dans la deuxième moitié du tableau. Pour la suite de la recherche, on retient soit la première, soit la deuxième moitié du tableau, selon si l'élément recherché est plus grand ou plus petit que l'élément du milieu. A chaque prochaine itération (à chaque passage dans la *boucle*), on vérifie si l'élément au milieu du nouveau sous-tableau retenu n'est pas l'élément recherché.

Dans la **recherche linéaire**, chaque passage de la *boucle* permettait de comparer un élément à l'élément recherché et l'espace de recherche diminuait seulement de 1 (l'espace de recherche correspond aux nombre d'emplacements encore possibles). Dans la **recherche binaire**, chaque passage de la *boucle* divise l'espace de recherche par deux et nous n'avons besoin de parcourir plus qu'une moitié (de la moitié, de la moitié, de la moitié, etc.) du tableau.

Le nombre d'étapes nécessaires pour rechercher un élément dans un tableau de taille 16 de façon dichotomique correspond donc au nombre de fois que le tableau peut être divisé en 2 et correspond à 4 (comme on peut le voir sur la figure ci-dessous), parce que :

$$16/2/2/2/2 = 1 \text{ donc}$$

$$2 * 2 * 2 * 2 = 16 \text{ ou}$$

$$2^4 = 16$$

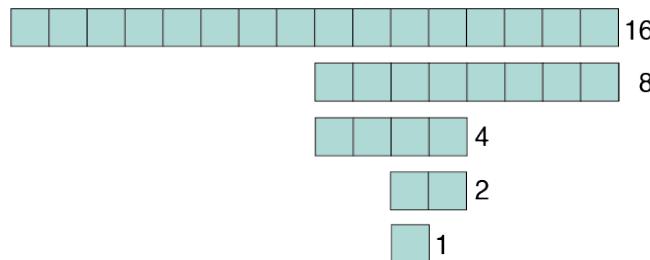


FIG. 4.2 – Exemple de parcours d'un tableau trié. Seulement 4 étapes sont suffisantes pour parcourir un tableau trié de 16 éléments à la recherche d'un élément qui se trouve à la onzième position. A chaque étape, l'espace de recherche est divisé par 2. Si le tableau n'était pas trié, 11 étapes seraient nécessaires (on doit vérifier chaque élément dans l'ordre).

Si on généralise, le nombre d'étapes x nécessaires pour parcourir un tableau de taille n est :

$$2^x = n \quad \text{par conséquent}$$

$$x = \log_2(n) \log(n) \quad \text{la simplification peut être faite car l'ordre de grandeur est le même}$$

La complexité de l'algorithme de recherche binaire est donc **logarithmique**, lorsque n grandit nous avons besoin de $\log(n)$ opérations. La figure ci-dessous permet de comparer les ordres de grandeur logarithmique et linéaire. On remarque qu'un algorithme de complexité logarithmique est beaucoup plus rapide qu'un algorithme de complexité linéaire, car il a besoin de beaucoup moins d'instructions élémentaires pour trouver une solution.

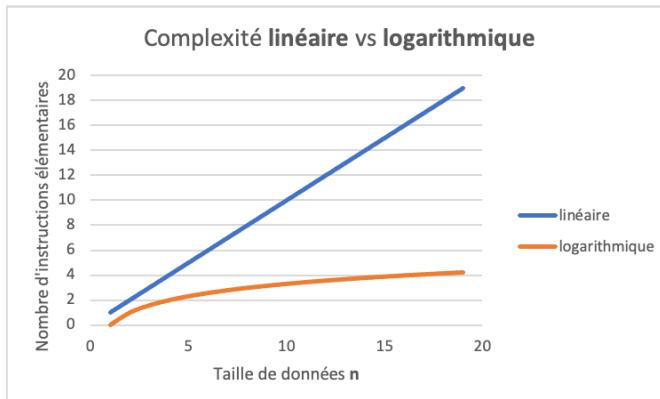


FIG. 4.3 – Comparaison de complexités logarithmique et linéaire. Un algorithme de complexité logarithmique est plus rapide qu'un algorithme de complexité linéaire.

L'algorithme de la recherche binaire se base sur une stratégie algorithmique de résolution de problèmes très efficace, que l'on appelle la stratégie « **diviser pour régner** ». Cette stratégie qui consiste à :

Diviser : décomposer le problème initial en sous-problèmes ;

Régner : résoudre les sous-problèmes ;

Combiner : calculer une solution au problème initial à partir des solutions des sous-problèmes.

Les sous-problèmes étant plus petits, ils sont plus faciles et donc plus rapides à résoudre. Les algorithmes de ce type en plus d'être efficaces, se prêtent à la résolution en parallèle (par exemple sur des multiprocesseurs).

Exercice 5 – Recherche binaire

Programmer l'algorithme de recherche binaire en Python. Rechercher une valeur entre 0 et 100 dans un tableau qui contient les valeurs allant de 1 à 99.

Pour quelle valeur l'algorithme est le plus rapide ? Pour quelle valeur l'algorithme est le plus lent ?
Indice : mettre un mode verbose pour afficher ce que fait l'algorithme.

Solution 5 – Recherche binaire

Voici un programme possible pour l'algorithme de recherche binaire. L'algorithme est le plus rapide si on recherche la valeur qui se trouve au milieu du tableau (la valeur 50) et il est le plus lent lorsque l'on recherche la première ou dernière valeur du tableau (la valeur 1 ou 99).

```

# algorithme de recherche&nbsp;binaire
def search_bin(search_list, search_element, verbose = 0) :
    # détermine les limites de la liste considérée
    start = 0
    end = len(search_list)
    # tant que la liste
    while end-start :
        # accède l'élément du milieu, division entière, il faut un index
        middle = (end-start) // 2 + start
        if verbose :
            print("Elément comparé : " + str(search_list[middle]) + "\n")
        # compare l'élément au milieu de la liste
        if search_element == search_list[middle] :
            if verbose :
                print("Elément retrouvé !\n")
            return True
        # l'élément est plus petit que l'élément du milieu
        elif search_element < search_list[middle] :
            # search_list devient dans la première moitié de search_list
            end = middle
            if verbose :
                print("1ère moitié de la liste : " + str(search_list[start:end])
                      + "\n")
        # l'élément est plus grand que l'élément du milieu
        else :
            # search_list devient la deuxième moitié de search_list
            start = middle + 1
            if verbose :
                print("2ème moitié de la liste : " + str(search_list[start:end])
                      + "\n")
    # aucun élément ne correspond
    if verbose :
        print("L'élément " + str(search_list[middle]) + " n'a pas été retrouvé
              ... \n")
    return False

ma_liste = list(range(1,100))
mon_element = 25
# recherche de l'élément mon_element
search_bin(ma_liste, mon_element, 1)

```

Exercice 6 – Recherche binaire (dans le désordre)

Est-ce qu'on peut utiliser l'algorithme de recherche binaire si le tableau n'est pas trié ? Essayer avec la fonction `shuffle()` du module `random`.

Solution 6 – Recherche binaire (dans le désordre)

Si le tableau n'est pas trié, l'algorithme n'est pas garanti de trouver l'élément recherché, car il peut facilement passer à côté. Le code ci-dessous est à utiliser avec la fonction `search_bin()` donnée dans la solution précédente.

```

1 import random
2
3 last = 99
4 ma_liste = list(range(1, last+1))
5
6 mon_element = random.randint(1, last)
7 print("L'élément recherché est : " + str(mon_element) + "\n")
8
9 # recherche de l'élément mon_element
10 search_bin(ma_liste, mon_element, 1)
11
12 random.shuffle(ma_liste)
13 print("Tableau mélangé... : " + str(ma_liste) + "\n")
14 search_bin(ma_liste, mon_element, 1)

```

Exercice 7 – Recherche linéaire versus binaire

Reprendre les programmes de recherche linéaire et recherche binaire en Python et les utiliser pour rechercher un élément dans un tableau à 100 éléments : quel algorithme est le plus rapide ?

Augmenter la taille du tableau à 1000, 10000, 100000, 1000000 et 10000000. Rechercher un élément avec vos deux programmes. Quel algorithme est plus rapide ? Est-ce significatif ?

Est-ce que **un million** vous semble être un grand nombre pour une taille de données ?

Solution 7 – Recherche linéaire versus binaire

Comme prévu par les estimations de complexité, avec sa complexité logarithmique, c'est l'algorithme de la recherche binaire qui est plus rapide. Le gain de temps devient de plus en plus important au fur et à mesure que le nombre d'éléments dans le tableau grandit. Pour cent éléments, la recherche binaire est environ **2 fois** plus rapide que la recherche linéaire, alors que pour un million d'éléments, elle est plus de **1000 fois** plus rapide.

On peut remarquer que le temps pris par la recherche binaire change peu avec la taille du tableau, ce qui n'est pas le cas de la recherche linéaire. Il faut environ **10 secondes** pour trier un million d'éléments avec l'algorithme linéaire, alors que moins de **10 millisecondes** suffisent à l'algorithme binaire. Les systèmes actuels traitent des données bien plus volumineuses qu'un million, pensez à toutes les vidéos sur le Web ou tous les utilisateurs d'un réseau social. Tout serait très lent, trop lent si on n'avait pas pensé à diviser pour régner.

```

import time
import random

# stocke les résultats
resultat_lin, resultat_bin = [], []
# longueurs des listes
nb = [100, 1000, 10000, 100000, 1000000, 10000000]

# pour toute les longueurs des listes
for last in nb :

    # créer la liste
    ma_liste = list(range(1,last+1))
    # rechercher un élément au hasard
    mon_element = random.randint(1,last)
    print("L'élément recherché est : " + str(mon_element))
    # recherche&nbsp;linéaire
    time_1 = time.time()
    search_lin(ma_liste, mon_element, 0)
    time algo lin = round(time.time() - time_1, 7)
    resultat_lin.append(time algo lin)
    # recherche&nbsp;binaire
    time_1 = time.time()
    search_bin(ma_liste, mon_element, 0)
    # des fois, c'est tellement rapide que le temps pris vaut 0
    time algo bin = round(max(time.time() - time_1, 0.000001), 7)
    resultat_bin.append(time algo bin)

print("Linéaire (secondes) : " + str(resultat_lin))
print("Binaire (secondes) : " + str(resultat_bin))

```

Le saviez-vous ? – Espace-temps et énergie

Nous allons surtout étudier la complexité des algorithmes en rapport avec le temps. Mais la complexité d'un algorithme peut également être calculée en rapport avec toutes les ressources qu'il utilise, par exemple des ressources d'**espace en mémoire** ou de **consommation d'énergie**.

4.2.3 Exercices

Exercice 8 – Recherche binaire aléatoire

Modifier votre programme de recherche binaire : au lieu de diviser l'espace de recherche exactement au milieu, le diviser au hasard. Cette recherche avec une composante aléatoire s'apparente plus à la recherche que l'on fait lorsque l'on cherche un mot dans le dictionnaire.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais que la complexité temporelle donne une indication sur la vitesse d'un algorithme, en mesurant le nombre d'instructions élémentaires.
2. Je sais qu'un algorithme de complexité linéaire est plus lent qu'un algorithme de complexité logarithmique.
3. Je peux utiliser la stratégie « diviser pour régner » pour rechercher rapidement avec l'algorithme de recherche binaire.

4.3 Algorithmes heuristiques

Matière à réfléchir – Tour du monde

Vous avez décidé de faire le tour du monde. Choisissez cinq pays que vous souhaitez visiter et placez-les sur une carte. Essayez de trouver le meilleur itinéraire pour visiter ces cinq pays.

Quels critères avez-vous pris en compte pour décider du meilleur itinéraire ? Avez-vous essayé de trouver la plus petite distance à parcourir ?

Vous avez décidé de visiter dix pays. Est-ce qu'il est aussi facile de trouver un itinéraire optimal ?

Imaginez que vous souhaitez visiter tous les pays du monde (un peu moins de 200). Combien y a-t-il d'itinéraires possibles ? Comment s'appelle ce nombre ?

Si le calcul d'un itinéraire prenait 1 milliseconde, combien de temps faudrait-il pour trouver la meilleure solution en énumérant toutes les solutions possibles ? Pour comparaison, le nombre d'atomes dans l'univers est d'ordre 10^{80} .

4.3.1 Complexité exponentielle

Il existe des problèmes difficiles à résoudre. Nous allons nous pencher sur un problème qui s'appelle le **problème du sac à dos**. Prenons un sac à dos et une multitude d'objets qui ont chacun un poids. Notre objectif est de choisir les objets à mettre dans le sac à dos pour le remplir au maximum, mais sans dépasser sa capacité. Donc la question que l'on se pose est la suivante : quels objets devrions-nous emporter, sans dépasser le poids maximal que le sac à dos peut contenir ?

Exercice 1 – Le problème du sac à dos

Comment procéderiez-vous pour résoudre ce problème du sac à dos ? Prenez le temps d'imaginer un *algorithme* qui puisse résoudre ce problème ?

Appliquer cet algorithme pour 4 objets de poids 1, 3, 5 et 7 kg et un sac de capacité de 10 kg.

Est-ce que votre algorithme donne toujours la meilleure solution ?

Solution 1 – Le problème du sac à dos

La solution est donnée dans le texte qui suit.

L'algorithme le plus simple pour résoudre ce problème est un **algorithme de force brute** (ou un algorithme exhaustif), qui consiste à énumérer toutes les combinaisons d'objets que pourrait contenir le sac à dos, l'une après l'autre, et de calculer le poids total pour chaque combinaison. Après avoir calculé toutes les combinaisons, il suffit de sélectionner la combinaison dont le poids se rapproche le plus de la

capacité du sac à dos, sans la dépasser. Vous trouverez ci-dessous la solution pour l'exemple de l'exercice précédent (« oui » signifie que l'on met l'objet dans le sac à dos et « non » signifie que l'on ne met pas l'objet dans le sac à dos).

+ Combinaison	1kg	3kg	5kg	7kg	Poids total +
1	non	non	non	non	0kg
2	oui	non	non	non	1kg
3	non	oui	non	non	3kg
4	oui	oui	non	non	4kg
5	non	non	oui	non	5kg
6	oui	non	oui	non	6kg
7	non	oui	oui	non	8kg
8	oui	oui	oui	non	9kg
9	non	non	non	oui	7kg
10	oui	non	non	oui	8kg
11	non	oui	non	oui	10kg
12	oui	oui	non	oui	11kg
13	non	non	oui	oui	12kg
14	oui	non	oui	oui	13kg
15	non	oui	oui	oui	15kg
16	oui	oui	oui	oui	16kg

La meilleure solution se trouve à la 11ème ligne, la capacité du sac à dos (10 kg) est atteinte lorsqu'on y met le deuxième et le quatrième objet.

Exercice 2 – Le problème du sac à dos avec 10 objets

Combien de combinaisons possibles existent pour le problème du sac à dos avec 10 objets ?

Solution 2 – Le problème du sac à dos avec 10 objets

La solution est donnée dans le texte qui suit.

Mais, combien y a-t-il de combinaisons possibles si on a 10 objets ? Pour chaque objet, on a deux choix possibles : le mettre dans le sac à dos ou ne pas le mettre dans le sac à dos (*to take or not to take, that is the question*). Comme ces deux possibilités existent pour chacun des 10 objets, le nombre de combinaisons possibles vaut :

$$2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 2^{10}$$

Pour n objets, le nombre de solutions possibles est 2^n . Si on a 2 objets, il y a donc 4 combinaisons différentes d'objets dans le sac à dos (aucun objet, le premier objet, le deuxième objet et les deux objets ensemble). Pour 3 objets, le nombre de combinaisons est 8. Pour 5 objets, nous avons 32 possibilités à explorer. Mais déjà pour 10 objets, ce nombre dépasse les 1000 combinaisons possibles. Pour 100 objets, ce nombre devient prohibitif et vaut 10^{30} . Si on doit résoudre ce problème avec 270 objets sous la main, le nombre de

combinaisons possibles dépasse le nombre d'atomes dans l'univers, c'est-à-dire 10^{80} . Si le calcul du poids d'une combinaison prenait une microseconde, il nous faudrait pour résoudre ce problème bien plus que le temps de l'existence de l'univers, plus de 14 milliards d'années. Ces nombres sont réellement vertigineux. Cela va de soi, nous n'avons pas tout ce temps à disposition...

L'ordre de complexité de type 2^n est un ordre de **complexité exponentielle**. Cela vaut aussi pour d'autres constantes que 2, par exemple 10^n ou 1.1^n . Lorsqu'un algorithme est d'ordre de complexité exponentielle, cela veut dire que le temps nécessaire pour résoudre le problème croît exponentiellement en fonction de la taille des données n (voir figure ci-dessous). Les problèmes de complexité exponentielle ne peuvent être résolus dans un temps raisonnable, pour des données à partir d'une certaine taille.

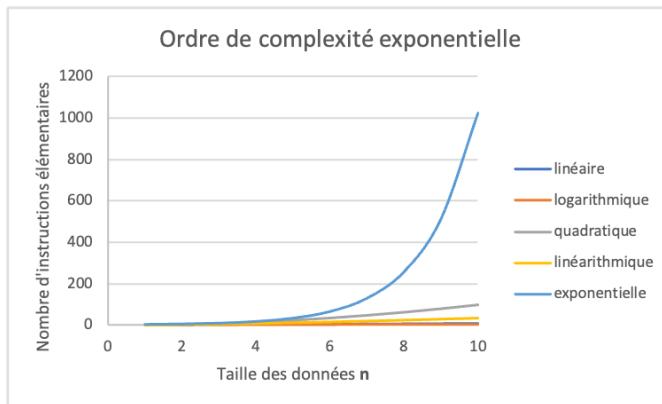


FIG. 4.4 – Complexité exponentielle. Comparaison de l'ordre de complexité exponentielle avec les ordres de complexité vus jusqu'ici. Dans un ordre de complexité exponentielle, le nombre d'instructions élémentaires grandit très rapidement avec la taille des données, et l'algorithme est très lent.

Lorsqu'il est trop difficile de trouver une solution exacte à un problème, nous ne devons pas nous avouer vaincus. Dans ce cas, nous pouvons tout de même rechercher une solution inexacte, mais qui se rapproche autant que possible de la solution optimale. Les algorithmes qui aboutissent à des solutions non optimales ou inexactes, sont appelés des **heuristiques**.

Un algorithme heuristique pour le problème du sac à dos pourrait être l'algorithme suivant : prendre les objets du plus petit au plus grand poids jusqu'à remplir le sac à dos, ce qui nous permettrait de mettre le plus d'objets possible. En suivant cet algorithme heuristique, dans l'exemple du premier exercice, on prendrait les trois premiers objets et on aurait un sac à dos rempli à 9 kg au lieu des 10 kg de capacité maximale du sac à dos. Cette solution est suffisamment proche de la meilleure solution, mais elle n'est pas la meilleure solution.

Une solution heuristique est donc une solution intuitive, qui se base sur une **stratégie d'essais et d'erreurs**, qui en quelque sorte repose sur la chance. Un algorithme heuristique est plus rapide que l'algorithme de force brute qui énumérerait toutes les solutions possibles afin de trouver la meilleure solution, mais on paie le prix de cette efficacité par de la précision. Un algorithme heuristique aboutit à une solution moins précise et moins complète, à une solution sans garantie. Quand un problème est trop complexe, il ne peut être résolu que par des algorithmes heuristiques, aboutissant dans certains cas à des mauvaises solutions.

Le saviez-vous ? – Que veut dire heuristique ?

Le mot **heuristique** nous vient du grec ancien, plus précisément du terme *heuriskēin*, qui veut dire trouver, inventer, découvrir.

Ce même terme a donné un autre mot bien connu *eurêka*.

L'algorithme heuristique qu'on vient de voir est en fait un **algorithme glouton**, un algorithme qui choisit une solution *localement optimale* (qui choisit la meilleure solution en apparence à un moment donné) sans se préoccuper de toutes les solutions possibles. On espère ainsi que toutes ces décisions localement optimales mènent vers une très bonne solution. C'est un peu comme si on cherchait à atteindre le plus haut sommet d'une montagne, entourés de brouillard, et qu'on prenait une décision sur le chemin à emprunter uniquement en fonction de ce que l'on peut voir juste autour de nous. On pourrait prendre le chemin le plus pentu en espérant qu'il nous mène à un sommet très haut, mais une fois arrivés en haut d'un sommet, on ne peut savoir si notre sommet est bien le plus haut. On l'espère...

Il n'y a pas que des *heuristiques* gloutonnes. Un autre exemple de solution heuristique, très utilisée dans les jeux vidéos, est le calcul de distance entre deux objets. Ce calcul est très important par exemple lorsque l'on souhaite détecter si deux objets sont en collision. Pythagore nous dit que cette distance vaut la racine carrée de la somme de a et b au carré. Mais ce calcul est difficile, et même si on peut le calculer de manière exacte, il prend beaucoup de temps à calculer s'il y a beaucoup d'objets affichés à l'écran. On préfère ainsi estimer cette distance par un calcul bien plus simple $a + b$, que l'on sait faux, mais qui est suffisamment proche lorsque les objets sont alignés (voir la figure ci-dessous).

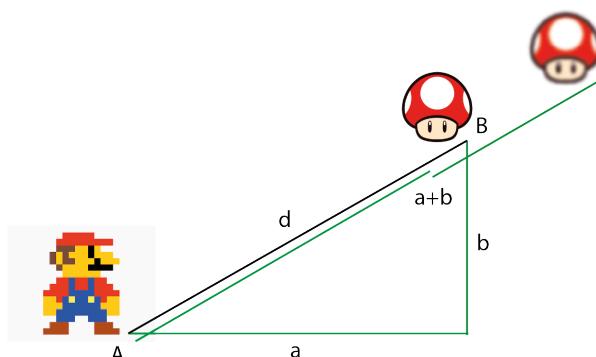


FIG. 4.5 – Exemple d'heuristique. Dans les jeux vidéos, on préfère estimer la distance d entre deux objets A et B par la somme des longueurs des côtés de l'angle droit $a + b$, plutôt que de calculer la racine carrée de la somme des carrés des longueurs des côtés de l'angle droit $d^2 = a^2 + b^2$ (théorème de Pythagore). Même si ce calcul est inexact, il est beaucoup plus rapide à calculer quand il y a beaucoup d'objets à afficher à l'écran, et il est suffisamment précis lorsque les deux objets sont alignés.

Il existe encore d'autres types d'algorithmes *heuristiques*, plus lents, mais qui permettent de s'approcher davantage de la solution optimale. Ils utilisent par exemple des stratégies de résolution statistiques, génétiques ou neuronales. L'apprentissage automatique à qui l'on doit les succès récents de l'intelligence artificielle repose sur des algorithmes heuristiques. La majorité des problèmes que l'on tente de résoudre aujourd'hui sont difficiles et leurs algorithmes de résolution ne trouvent pas la meilleure solution.

Pour aller plus loin

Voici un problème à un million de dollars, un parmi les sept problèmes mathématiques du prix du millénaire qui rapporteront de l'argent à la personne qui les résoudra.

On appelle la classe des problèmes qui sont faciles à résoudre la classe des problèmes P . Ces algorithmes peuvent être résolus en un temps polynomial en fonction de la taille des données n , ou $O(n^a)$.

Une autre classe de problèmes sont les problèmes difficiles à résoudre qui sont d'ordre de complexité exponentielle. Lorsqu'on arrive à vérifier rapidement (en temps polynomial) si une solution proposée permet de résoudre le problème, il s'agit d'une classe de problèmes appellée NP ou « non déterministe polynomial ».

On souhaite savoir si les problèmes NP peuvent être résolus en un temps P ou non, ou en d'autres termes : est-ce que $P = NP$?

S'il s'avérait que c'est bien le cas (ce qui est tout de même peu probable), beaucoup de problèmes difficiles à résoudre deviendraient d'un seul coup plus faciles à résoudre. Un des ces problèmes est le **problème de repliement des protéines** en biologie qui cherche de nouveaux médicaments. Cela pourrait également signifier la fin de la cryptographie telle qu'elle existe actuellement.

4.3.2 Exercices

Exercice 3 – L'univers dans un sac à dos

L'âge estimé de l'univers est de 14 milliards d'années. Si le calcul d'une combinaison d'objets dans le problème du sac à dos prenait une microseconde, pour quel nombre d'objets serait-il possible de trouver une solution exacte sans dépasser l'âge de l'univers ?

Exercice 4 – Parcours du parcours du parcours de listes

Quelle est la complexité d'un algorithme qui pour chacun des éléments d'une liste de n éléments, doit parcourir tous les éléments d'une autre liste de n éléments, puis pour chacune des combinaisons de deux éléments doit encore parcourir une troisième liste de n éléments ?

Si vous avez besoin de travailler sur un exemple plus concret, quelle est complexité de l'algorithme qui calcule tous les menus possibles d'un restaurant à partir d'une liste de n entrées, une liste de n plats et une liste de n desserts ?

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais reconnaître un algorithme de force brute.
2. Je sais reconnaître un algorithme heuristique.
3. Je sais reconnaître un algorithme glouton.
4. Je comprends pourquoi un algorithme de complexité exponentielle est lent.

4.4 Algorithmes de tri [niveau avancé]

Nous venons de voir que pour rechercher de manière efficace, les données doivent être triées. Mais quelle est la complexité de l'algorithme du Tri par sélection vu dans le chapitre Trie, cherche et trouve ? C'est sa complexité qui nous donnera une indication sur sa rapidité.

4.4.1 Tri par sélection

Pour rappel, l'*algorithme du Tri par sélection* parcourt le tableau à la recherche des plus petits éléments. Afin de trouver le plus petit élément du tableau, il faut commencer par parcourir tous les éléments du tableau. Cette opération prend cn instructions : c instructions pour l'accès et la comparaison des éléments du tableau, multiplié par le nombre d'éléments n . Il faut ensuite trouver le plus petit élément des éléments restants $n - 1$, et ainsi de suite. Concrètement, on va parcourir jusqu'à n éléments, n fois (pour chacun des éléments). La complexité du Tri par sélection est donc proportionnelle à $n * n$ (n^2), on parle de complexité **quadratique**.

Si on compare les complexités vues jusqu'ici pour un tableau de 1000 éléments on obtient :

Complexité	Instructions élémentaires pour 1000 éléments
Linéaire	1000
Logarithmique	10
Quadratique	1000000

Avec une complexité quadratique, le Tri par sélection est un algorithme relativement lent.

Exercice 1 – Complexité du Tri par insertion

Quelle est la complexité de l'algorithme de **Tri par insertion** ? En d'autres termes, si le tableau contient n éléments, combien faut-il d'instructions pour trier ce tableau ? Pour rappel, le Tri par insertion parcourt le tableau dans l'ordre et pour chaque nouvel élément, l'insère à l'emplacement correct des éléments déjà parcourus.

Solution 1 – Complexité du Tri par insertion

Dans le pire cas, lorsque les éléments sont dans l'ordre inverse, on doit comparer chacun des n éléments avec 1 à n éléments. La complexité de l'algorithme du Tri par insertion est donc $n * n = n^2$ ou **quadratique**.

Pour aller plus loin – Calcul de complexité

Si vous souhaitez connaître les détails du calcul de complexité, lisez ce qui suit.

Pour calculer la somme totale d'instructions nécessaires, il faut additionner les termes qui permettent de retrouver le plus petit élément. La première fois que l'on recherche le plus petit élément il faut parcourir n éléments. La deuxième fois, il reste à parcourir $n - 1$ éléments. La troisième fois, il faut parcourir les $n - 2$ éléments restants. Et ainsi de suite, jusqu'à ce qu'il ne reste plus qu'un élément.

Par exemple, si le tableau contient les éléments $[5, 2, 3, 6, 1, 4]$, pour trouver le plus petit élément 1 à la première itération on doit parcourir tout le tableau, ou 6 éléments. A la deuxième itération, on met l'élément 1 de côté et on parcourt le tableau $[5, 2, 3, 6, 4]$, ce qui fait 5 éléments. On met le plus petit élément 2 de côté et dans la troisième itération on parcourt le tableau $[5, 3, 4, 6]$, ce qui fait 4 éléments. On met 3 de côté et on parcourt encore le tableau $[5, 4, 6]$, ce qui fait 3 éléments. Finalement on se retrouve avec les tableaux $[5, 6]$ à 2 éléments et $[5]$ à 1 élément. La somme totale d'éléments parcourus est $6 + 5 + 4 + 3 + 2 + 1 = 21$.

Si on généralise on obtient :

$$n + (n - 1) + (n - 2) + \dots + (n/2 + 1) + n/2 + \dots + 3 + 2 + 1$$

En réarrangeant les termes deux par deux de l'extérieur vers l'intérieur on obtient plusieurs fois le même terme :

$$(n + 1) + ((n - 1) + 2) + ((n - 2) + 3) + \dots + ((n/2 + 1) + n/2)$$

$$(n + 1) + (n + 1) + (n + 1) + \dots + (n + 1) =$$

$$(n + 1) * n/2 =$$

$$n * n/2 + n/2 =$$

$$n^2/2 + n/2$$

Si on reprend l'exemple ci-dessus, on aurait :

$$6 + 5 + 4 + 3 + 2 + 1 =$$

$$(6 + 1) + (5 + 2)(4 + 3) =$$

$$7 * (6/2) = 7 * 3 = 21 \quad \text{ou}$$

$$6^2/2 + 6/2 = 36/2 + 3 = 18 + 3 = 21$$

Le terme dominant dans la somme $n^2/2 + n/2$ est $n^2/2$, plus n grandit plus $n/2$ est insignifiant par rapport à $n^2/2$. Par exemple, pour $n = 1000$, $n^2/2$ vaut 500000, alors que $n/2$ vaut seulement 500.

Cette somme nous donne le nombre d'éléments parcourus. Mais pour chacun de ces éléments, plusieurs instructions sont exécutées, comme l'accès aux éléments et leur comparaison. Ces instructions et le terme qui divise par 2 peuvent être absorbés dans une *constante* c qui multiplie le terme quadratique n^2 . En ajoutant une constante a pour prendre en compte le nombre d'instructions qui ne dépendent pas de la taille des données (comme les initialisations au début de l'algorithme), on obtient l'ordre de grandeur $cn^2 + a$. L'ordre de grandeur est donc **quadratique**.

Exercice 2 – Complexité du Tri à bulles

Quelle est la complexité de l'algorithme de **Tri à bulles**? En d'autres termes, si le tableau contient n éléments, combien faut-il d'instructions pour trier ce tableau ? Pour rappel, le Tri à bulles compare les éléments deux par deux en les réarrangeant dans le bon ordre, afin que l'élément le plus grand remonte vers la fin du tableau tel une bulle d'air dans de l'eau. Cette opération est répétée n fois, pour chaque élément du tableau.

Solution 2 – Complexité du Tri à bulles

Dans le cas du Tri à bulles, pour chacun des n éléments on parcourt jusqu'à n éléments de la liste, ce qui nous donne une complexité $n * n = n^2$ ou une complexité quadratique.

4.4.2 Tri rapide

Les trois *algorithmes* de tri vus dans le chapitre précédent – le Tri par sélection, le Tri par insertion et le Tri à bulles – sont des algorithmes de complexité quadratique. Si on devait utiliser ces tris dans les systèmes numériques en place, on passerait beaucoup de notre temps à attendre. Il existe d'autres algorithmes de tri qui sont bien plus rapides. Nous allons voir un *algorithme* de tri tellement rapide, qu'on lui a donné le nom **Tri rapide**.

On commence par prendre un élément du tableau que l'on définit comme **élément pivot**. Cet élément pivot est en général soit le premier ou le dernier élément du tableau, soit l'élément du milieu du tableau ou encore un élément pris au hasard. On compare ensuite tous les autres éléments du tableau à cet élément pivot. Tous les éléments qui sont plus petits que le pivot seront mis à sa gauche et tous les éléments qui sont plus grands que le pivot seront mis à sa droite, tout en conservant leur ordre (voir la deuxième ligne de la figure ci-dessous).

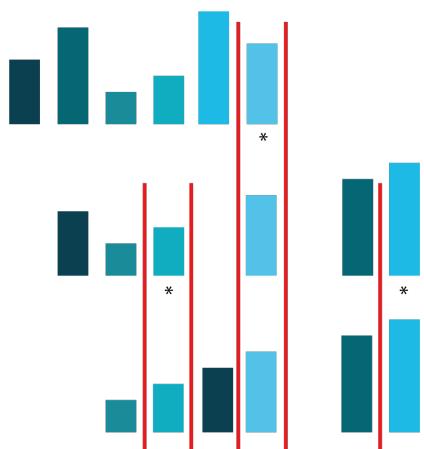


FIG. 4.6 – Tri rapide. Illustration du tri rapide sur les mêmes données que celles utilisées pour illustrer les algorithmes de tri du chapitre précédent. On choisit comme élément pivot le dernier élément des tableaux à trier. Tous les éléments qui sont plus petits que le pivot se retrouvent à sa gauche, tous les éléments plus grands que le pivot se retrouvent à sa droite. L'ordre est conservé.

Après la répartition des éléments autour de l'élément pivot en fonction de leur taille, on se retrouve avec deux tableaux non triés, un sous-tableau à chaque côté de l'élément pivot. On répète les mêmes opérations que pour le tableau initial. Pour chaque sous-tableau, celui de gauche et celui de droite du pivot, on détermine un nouvel élément pivot (le dernier élément du sous-tableau). Pour chaque nouvel élément pivot, on met à gauche les éléments du sous-tableau qui sont plus petits que le pivot et on met à droite les éléments du sous-tableau qui sont plus grands que le pivot. **On répète** ces mêmes opérations jusqu'à ce qu'il ne reste plus que des tableaux à 1 élément (plus que des pivots). A ce stade, le tableau est trié.

Intéressons-nous maintenant à la complexité de cet algorithme. À chaque étape (à chaque ligne de la figure ci-dessus), on compare tout au plus n éléments avec les éléments pivots, ce qui nous fait un multiple de n instructions élémentaires. Mais combien d'étapes faut-il pour que l'algorithme se termine ?

Dans le meilleur cas, à chaque étape de l'algorithme, l'espace de recherche est divisé par 2. Nous avons vu dans le chapitre Recherche binaire que lorsqu'on divise l'espace de recherche par deux, on obtient une complexité logarithmique. Le nombre d'étapes nécessaires est donc un multiple de $\log(n)$.

Pour obtenir le nombre total d'instructions élémentaires on multiplie le nombre d'instructions par étape avec le nombre d'étapes. La complexité que l'on obtient est une fonction de $n \log(n)$, il s'agit d'une complexité **linéarithmique**. Un algorithme avec une complexité linéarithmique est plus lent qu'un algorithme de complexité linéaire (la recherche linéaire) ou de complexité logarithmique (la recherche binaire). Par contre, il est bien plus rapide qu'un algorithme de complexité quadratique (le tri par sélection). La figure ci-dessous permet de comparer la vitesse de croissance des complexités étudiées jusqu'ici. Le tri rapide est donc vraiment l'algorithme de tri le plus rapide vu jusqu'ici et la complexité nous permet de comprendre pourquoi c'est le cas.

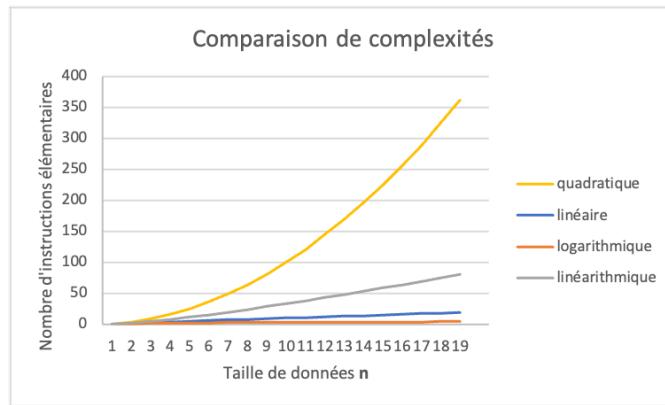


FIG. 4.7 – Comparaison de complexités. La vitesse de croissance en fonction de la taille du tableau n est montrée pour toutes les complexités vues jusqu'ici. Plus le nombre d'instructions élémentaires est grand en fonction de la taille des données, plus l'algorithme est lent.

La première question que l'on se pose lorsqu'on analyse un algorithme est son ordre de complexité temporelle. Si l'algorithme est trop lent, il ne sera pas utilisable dans la vie réelle. Pour le problème du Tri, la stratégie « **diviser pour régner** » vient à nouveau à la rescousse.

Exercice 3 – Le pire du Tri rapide

Que se passe-t-il si on essaie de trier un tableau déjà trié avec l'algorithme du **Tri rapide**, en prenant toujours comme pivot le dernier élément ? Essayer par exemple avec le tableau $[1, 2, 3, 4, 5, 6, 7]$.

Combien d'étapes sont nécessaires pour que l'algorithme se termine ? Quelle est la complexité dans ce cas ? Est-ce qu'un autre choix de pivot aurait été plus judicieux ?

Solution 3 – Le pire du Tri rapide

Si on simule l'algorithme de Tri rapide pour le tableau $[1, 2, 3, 4, 5, 6, 7]$ avec comme pivot le dernier élément on se retrouve avec les sous-tableaux suivants (le pivot est affiché en gras, les éléments déjà triés sont affichés en italique) :

```
[1, 2, 3, 4, 5, 6, **7**]
[1, 2, 3, 4, 5, **6**][*7*] []
[1, 2, 3, 4, **5**][*6*] [] [*7*] []
[1, 2, 3, **4**][*5*] [] [*6*] [] [*7*] []
[1, 2, **3**][*4*] [] [*5*] [] [*6*] [] [*7*] []
[1, **2**][*3*] [] [*4*] [] [*5*] [] [*6*] [] [*7*] []
[*1*][*2*] [] [*3*] [] [*4*] [] [*5*] [] [*6*] [] [*7*] []
[] [*1*] [] [*2*], [*3*][*4*] [] [*5*] [] [*6*] [] [*7*] []
[1] [2] [3] [4] [5] [6] [7]
```

Lorsque les éléments du tableau sont déjà triés, l'espace de recherche n'est plus divisé par deux. On se retrouve avec des sous-tableaux déséquilibrés, vides d'un côté et pleins de l'autre. Le nombre d'étapes n'est donc plus $\log(n)$, mais vaut n (7 étapes de traitement). Lorsqu'on multiple le nombre d'étapes (lignes) au nombre d'éléments à comparer par ligne, on est plutôt dans une complexité $n * n$ (ou n^2), donc quadratique. Dans ce scénario, le tri rapide n'est donc plus si rapide. Le choix du pivot est alors crucial et dépend du contenu du tableau.

Si on prend comme pivot l'élément du milieu du tableau, on se retrouve avec des sous-tableaux équilibrés, qui contiennent un nombre similaire d'éléments. Dans ce cas l'algorithme a besoin de moins d'étapes pour trouver la solution, de l'ordre de $\log(n)$, ici 3 lignes et équivalent à $\log_2(7)$, de traitement au lieu de 7 auparavant :

```
[1,2,3,**4**,5,6,7]
[1,**2**,3][*4*],[5,**6**,7]
[**1**][*2*][**3**][*4*],[**5**][*6*][**7**]
[][*1*][][*2*][][*3*][][*4*][][*5*][][*6*][][*7*]
[1][2][3][4][5][6][7]
```

Pour aller plus loin

Même si deux algorithmes de tri ont la même complexité temporelle, c'est-à-dire qu'ils prennent un temps comparable pour trier des données, ils ne prennent pas la même place en mémoire. Pour un algorithme qui prend peu de place en mémoire (par exemple le tri par insertion), on dit qu'il a une plus petite « **complexité spatiale** ».

Si on trie un tableau qui est en fait déjà trié avec le tri par insertion, la complexité dans ce cas est linéaire. Au contraire, si on trie ce même tableau avec le tri rapide, la complexité dans ce cas est quadratique. On voit donc que selon le tableau que l'on trie, le tri rapide peut être bien plus lent que le tri par insertion.

Une analyse complète d'un algorithme consiste à calculer la complexité non seulement dans le **cas moyen**, mais aussi dans le **meilleur cas** et dans le **pire cas**.

Pour aller plus loin

Une analyse complète va également calculer les constantes qui influencent l'ordre de complexité. Ces constantes ne sont pas importantes lors d'une première analyse d'un algorithme. En effet, les constantes n'ont que peu d'effet pour une grande taille des données n , c'est uniquement le terme qui grandit le plus rapidement en fonction de n qui compte, et qui figure dans un premier temps dans l'ordre de complexité. Par contre, lorsque l'on souhaite comparer deux algorithmes de même complexité, il faut estimer les constantes et choisir l'algorithme avec une constante plus petite.

La notation « Grand O », que l'on utilise pour écrire mathématiquement la complexité, désigne en fait la complexité dans le pire cas. Les différentes complexités vues jusqu'ici seraient notées : $O(n)$, $O(\log(n))$, $O(n^2)$ et $O(n\log(n))$. Arrivez-vous à trouver les adjectifs correspondants ?

Exercice 4 – Le meilleur et le pire du Tri par insertion

Que se passe-t-il si on essaie de trier un tableau déjà trié avec l'algorithme du **Tri par insertion**? Essayer par exemple avec le tableau $[1, 2, 3, 4, 5, 6, 7]$.

Combien d'étapes sont nécessaires pour que l'algorithme se termine? Quelle est la complexité dans ce cas?

Que se passe-t-il si on essaie de trier un tableau déjà trié, mais dans l'ordre inverse de celui qui est souhaité, avec l'algorithme du Tri par insertion? Essayer par exemple avec le tableau $[4, 3, 2, 1]$.

Solution 4 – Le meilleur et le pire du Tri par insertion

Si on simule l'algorithme de Tri par insertion pour le tableau $[1, 2, 3, 4, 5, 6, 7]$ on se retrouve avec la configuration suivante (l'élément inséré est affiché en gras, l'élément auquel on le compare en italique) :

```

[**1**, 2, 3, 4, 5, 6, 7]
[*1*] [**2**, 3, 4, 5, 6, 7]
[1, *2*] [**3**, 4, 5, 6, 7]
[1, 2, *3*] [**4**, 5, 6, 7]
[1, 2, 3, *4*] [**5**, 6, 7]
[1, 2, 3, 4, *5*] [**6**, 7]
[1, 2, 3, 4, 5, *6*] [**7**]
[1, 2, 3, 4, 5, 6, 7]

```

On voit qu'il y a besoin de 7 étapes, ou n étapes, car autant que d'éléments dans le tableau. Dans chaque étape on n'a besoin de comparer qu'une fois, avec l'élément précédent. La complexité dans ce cas est $n * 1 = n$ ou linéaire. Pour des données presque triées, le Tri par insertion est encore plus rapide que le Tri rapide.

A premier abord, trier le tableau $[5, 4, 3, 2, 1]$ avec le Tri par insertion ne présente pas de difficultés. Regardons ce qui se passe :

```

[**5**, 4, 3, 2, 1]
[*5*] [**4**, 3, 2, 1]
[4, *5*] [**3**, 2, 1]
[*4*, **3**, 5] [2, 1]
[3, 4, *5*] [**2**, 1]
[3, *4*, **2**, 5] [1]
[*3*, **2**, 4, 5] [1]

```

[2, 3, 4, *5*] [**1**]

[2, 3, *4*, **1**, 5]

[2, *3*, **1**, 4, 5]

[*2*, **1**, 3, 4, 5]

[1, 2, 3, 4, 5]

Cette fois-ci on se retrouve dans la pire configuration pour le Tri par insertion, où chaque élément doit être comparé à chaque autre élément. Ici nous avons besoin de 11 étapes de traitement pour trier 5 éléments, alors qu'avant 7 étapes suffisaient pour trier 7 éléments. Lorsqu'on doit trier un grand nombre d'éléments, cette différence est significative et peut rendre un algorithme nonutilisable.

4.4.3 Exercices

Exercice 5 – Une question à un million

Si une instruction prend 10^{-6} secondes, combien de temps faut-il pour trier un tableau d'un million d'éléments avec le tri à sélection comparé au tri rapide (sans tenir compte de la constante) ?

Exercice 6 – Une question de pivot

Trier le tableau suivant avec l'algorithme de tri rapide : [3, 6, 8, 7, 1, 9, 4, 2, 5] à la main, en prenant le dernier élément comme pivot. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Est-ce que le choix du pivot est important ?

Exercice 7 – Une question de sélection

Trier le tableau suivant avec l'algorithme de tri par sélection : [3, 6, 8, 7, 1, 9, 4, 2, 5] à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 8 – Une question d'insertion

Trier le tableau suivant avec l'algorithme de tri par insertion : [3, 6, 8, 7, 1, 9, 4, 2, 5] à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 9 – Une question de bulles

Trier le tableau suivant avec l'algorithme de tri à bulles : [3, 6, 8, 7, 1, 9, 4, 2, 5] à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 10 – Une question de chronomètre

Créer une liste qui contient les valeurs de 1 à n dans un ordre aléatoire, où n prend la valeur 100. Implémenter au moins deux des trois algorithmes de tri vu au cours.

A l'aide du module `time` et de sa fonction `time()`, chronométrer le temps pour le tri d'une liste de 100, 500, 1000, 10000, 20000, 30000, 40000 puis 50000 nombres. Noter les temps obtenus et les afficher sous forme de courbe dans un tableur.

Ce graphique permet de visualiser le temps d'exécution du tri en fonction de la taille de la liste. Que peut-on constater ? Sur la base de ces mesures, peut-on estimer le temps que prendrait le tri de 100000 éléments ? Lancer votre programme avec 100000 éléments et comparer le temps obtenu avec votre estimation.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais que grâce à la stratégie algorithmique « diviser pour régner », je ne passe pas mon temps à attendre que l'ordinateur me donne une réponse.
2. Je sais comment trier de manière rapide.
3. Je sais que la complexité temporelle peut changer selon la configuration des données, en plus du cas moyen, il est également utile d'estimer le pire et le meilleur cas.

4.5 Récursivité [niveau avancé]

Ce chapitre est prévu en tant que chapitre optionnel. Il présente un autre algorithme de tri célèbre, le **Tri par fusion**. Cet algorithme utilise la **récursivité**, une stratégie qui consiste en ce qu'un algorithme s'invoque lui-même. La récursivité, c'est un peu comme si on essayait de définir le terme « définition » en disant c'est une phrase qui nous donne la définition de quelque chose. C'est certes circonvolu que de vouloir utiliser dans une définition *la chose-même* que l'on est en train de définir, mais si on respecte quelques conditions, « ça fonctionne » !

4.5.1 Tri par fusion

Un autre *algorithme* de tri célèbre, inventé par John von Neumann en 1945, est le **Tri par fusion**. L'algorithme se base sur l'idée qu'il est difficile de trier un tableau avec beaucoup d'éléments, mais qu'il est très facile de trier un tableau avec juste deux éléments. Il suffit ensuite de fusionner les plus petits tableaux déjà triés.

L'algorithme commence par une phase de **division** : on divise le tableau en deux, puis on divise à nouveau les tableaux ainsi obtenus en deux, et ceci jusqu'à arriver à des tableaux avec un seul élément (voir la figure ci-dessous). Comme pour la recherche binaire, le nombre d'étapes nécessaires pour arriver à des tableaux de 1 élément, en divisant toujours par deux, est $\log(n)$.

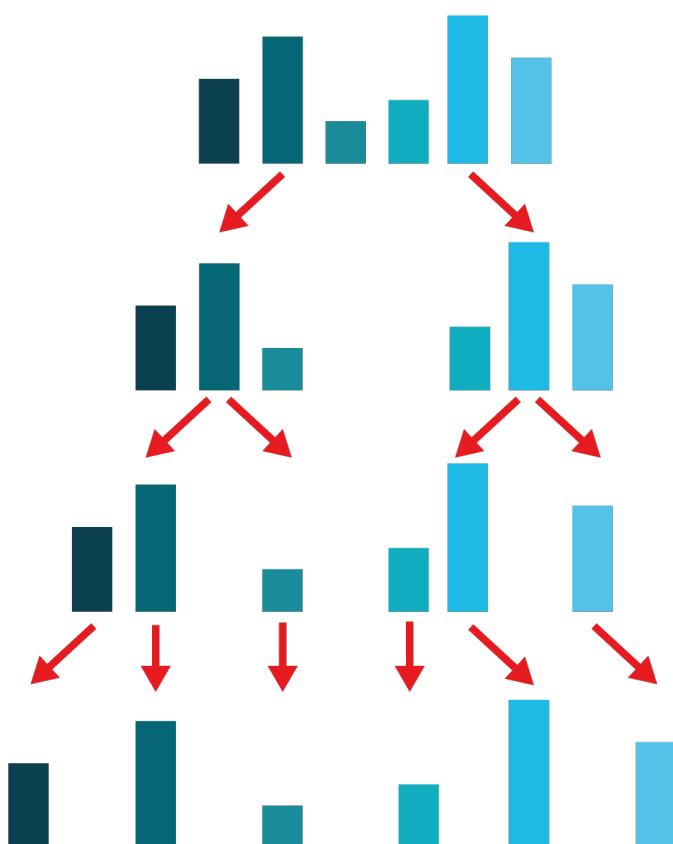


FIG. 4.8 – Phase de division. Illustration de la première phase du Tri par fusion : on commence par diviser le tableau en deux, puis à chaque étape on divise à nouveau les tableaux ainsi obtenus par deux, jusqu'à ce qu'il n'y ait plus que des tableaux à 1 élément.

La deuxième phase de *fusion* commence par fusionner des paires de tableaux à un élément, dans un *ordre trié*. Il suffit d’assembler les deux éléments du plus petit au plus grand, comme on peut le voir sur la 2ème ligne de la figure ci-dessous. Dans les prochaines étapes, on continue à fusionner les tableaux par paires de deux, tout en respectant l’ordre de tri (lignes 3 et 4 de la figure). On continue de la sorte jusqu’à ce qu’il n’y ait plus de tableaux à fusionner.

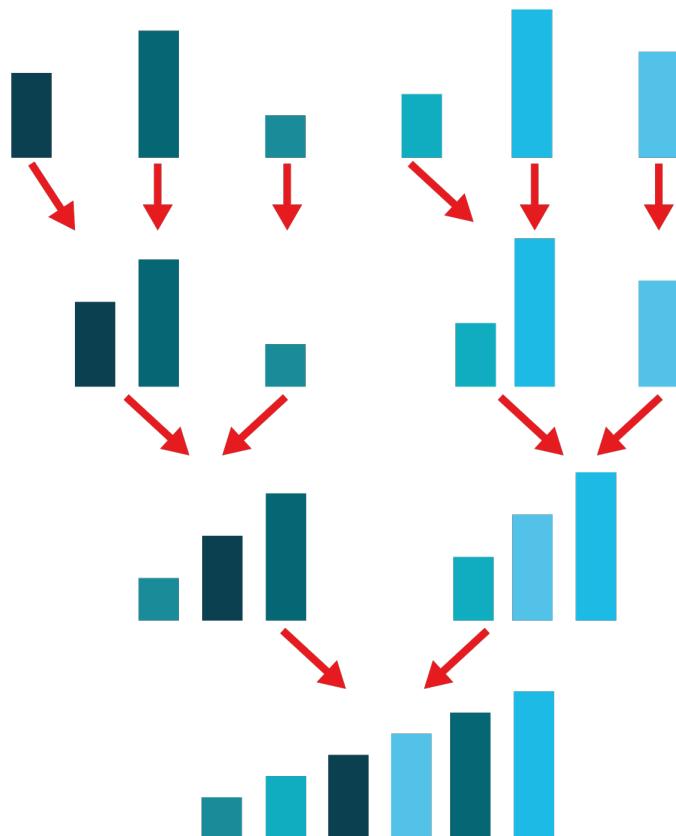


FIG. 4.9 – Phase de fusion. Illustration de la deuxième phase du Tri par fusion : on commence par fusionner les tableaux à un élément, en faisant attention à respecter l’ordre de tri (ligne 2); puis par fusionner à nouveau les tableaux obtenus à l’étape précédente, toujours en respectant l’ordre de tri (lignes 3 et 4). On continue de la sorte jusqu’à ce qu’il n’y ait plus qu’un tableau unique (ligne 4).

La fusion de tableaux **déjà triés**, par rapport à des tableaux non-triés, est très facile. Il suffit de comparer les premiers éléments des deux tableaux à fusionner et de prendre le plus petit des deux. Concrètement, on enlève le plus petit élément des deux tableaux pour le mettre dans le nouveau tableau fusionné. On compare ensuite les premiers éléments de ceux qui restent dans les tableaux à fusionner et on prend à nouveau le plus petit des deux pour le mettre à la suite dans le tableau fusionné.

Chaque étape de la phase de fusion consiste à comparer deux éléments n fois, autant de fois qu'il y a d'éléments à fusionner. Le temps de calcul grandit donc linéairement en fonction de la taille du tableau n (plus il y a d'éléments dans le tableau, plus la fusion prend du temps). En tout il y a besoin de $\log(n)$ étapes (fusion deux par deux), dont chacune prend un temps qui dépend de n , ce qui nous donne un ordre de complexité linéarithmique.

4.5.2 Focus sur la récursivité

Nous allons maintenant programmer l'*algorithme* du Tri par fusion. Pour rappel, la première phase de l'*algorithme* divise *continuellement* le tableau par deux, comme illustré dans la première figure ci-dessus. Voici le code qui permet de diviser un tableau en deux une seule fois :

```
# Tri par fusion
def tri_par_fusion(elements):

    ### Phase DIVISION

    # détermine l'indice au milieu du tableau (division entière)
    milieu = len(elements)//2

    # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
    elements_gauche = elements[:milieu]

    # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
    elements_droite = elements[milieu:]
```

La division utilisée pour déterminer le milieu du tableau est une division entière // au lieu de /. En effet, on souhaite obtenir un résultat entier et non un nombre à virgule, car les indices pour accéder aux éléments du tableau doivent être des entiers. Par exemple, si le tableau contient 5 éléments, cela n'aurait pas de sens de prendre les premiers 2.5 éléments, et 5//2 nous retournerait 2.

Ce qui suit est très intéressant. Dans l'étape d'après, on souhaite faire exactement la même chose pour les nouveaux tableaux `elements_gauche` (équivalent à `elements[:milieu]`) et `elements_droite` (équivalent à `elements[milieu:]`), c'est-à-dire que l'on souhaite à nouveau les diviser en deux, comme sur la deuxième ligne dans la première figure ci-dessus. On va donc appeler la fonction `tri_par_fusion` sur les deux moitiés de tableaux :

```
# prend tous les éléments depuis le début, jusqu'à (et sans) milieu
elements_gauche = tri_par_fusion(elements[:milieu])

# prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
elements_droite = tri_par_fusion(elements[milieu:])
```

Regardez bien ce qui se passe. Nous avons fait appel à la même *fonction* `tri_par_fusion` que l'on est en train de définir ! Pour l'instant cette fonction ne fait que diviser le tableau `elements` en deux, elle va donc diviser le tableau reçu en entrée en deux. Au début le tableau en entrée sera le tableau entier, mais ensuite il s'agira des deux moitiés du tableau, puis des moitiés de la moitié et ainsi de suite. La fonction `tri_par_fusion` appelle la fonction `tri_par_fusion` (elle s'appelle donc elle-même), qui va à nouveau s'appeler et ainsi de suite...

Si on laisse le programme tel quel, on est face à un problème. La fonction `tri_par_fusion` continue de s'appeler elle-même et ce processus ne s'arrête jamais. En réalité, il faut arrêter de diviser lorsque les tableaux obtenus ont au moins un élément ou lorsqu'ils sont vides, car dans ces cas on ne peut plus les diviser en deux. On rajoute donc cette **condition d'arrêt** de la récursion :

```

# condition d'arrêt la récursion
if len(elements) <= 1:
    return(elements)

# détermine l'indice au milieu du tableau (division entière)
milieu = len(elements)//2

# prend tous les éléments depuis le début, jusqu'à (et sans) milieu
elements_gauche = tri_par_fusion(elements[:milieu])

# prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
elements_droite = tri_par_fusion(elements[milieu:])

```

Voici le programme appliqué sur l'exemple de la figure. Essayez de comprendre dans quel ordre sont appelées les fonctions `tri_par_fusion` et avec quel paramètre en entrée. Pour une meilleure visibilité, nous affichons l'état des *variables* avec `print`.

```

# Tri&nbsp;par&nbsp;fusion : phase de division
def division(elements, ligne, side=0):
    # nous dit où on en est
    print("Appel de la fonction division avec ", str(elements), "ligne", ligne,
          "depuis", side)

    # correspond à la ligne sur la figure division du tri&nbsp;par&nbsp;fusion
    ligne = ligne + 1

    # condition d'arrêt la récursion
    if len(elements) <= 1:
        return(elements)

    # détermine l'indice au milieu du tableau (division entière)
    milieu = len(elements)//2

    # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
    elements_gauche = division(elements[:milieu], ligne, 'gauche')
    if elements_gauche :
        print('Eléments à gauche : ', elements_gauche, 'de :', elements, "
              ligne", ligne)

    # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
    elements_droite = division(elements[milieu:], ligne, 'droite')
    if elements_droite :
        print('Eléments à droite : ', elements_droite, 'de :', elements, "
              ligne", ligne)

division([3,5,1,2,6,4], 0)

```

Une *fonction* qui s'appelle elle-même est appelée ***fonction récursive***. Il s'agit d'une *mise en abîme*, d'une *définition circulaire*. Lorsqu'on entre dans la fonction, des opérations sont exécutées et on fait à nouveau **appel à la même fonction**, mais cette fois-ci avec **d'autres éléments en entrée**, afin de refaire les mêmes opérations, comme le montre cette figure :

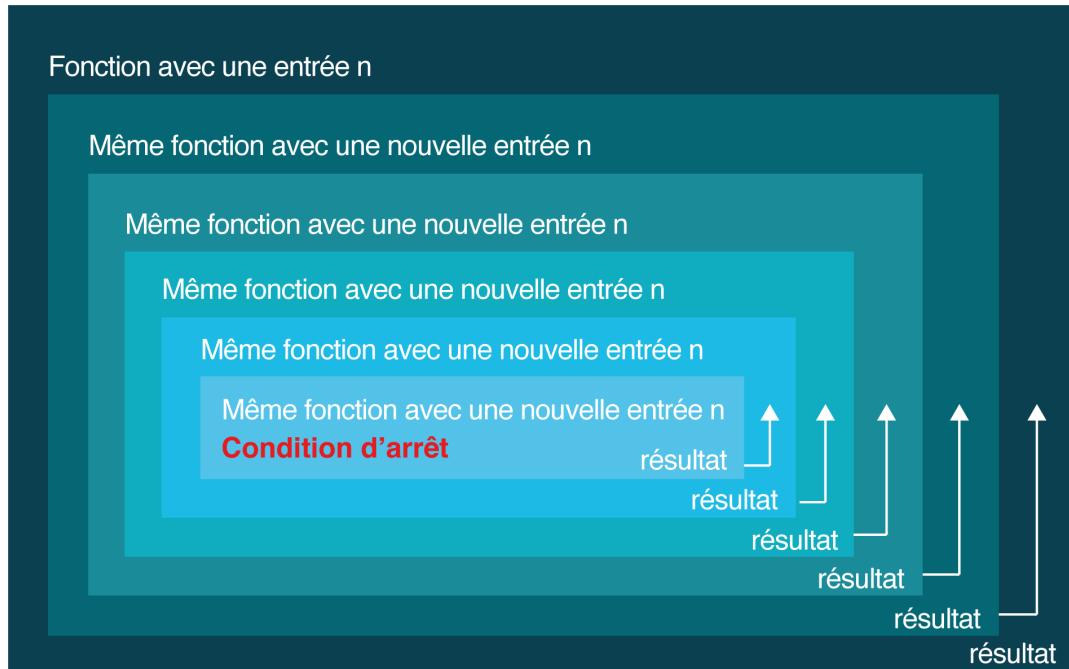


FIG. 4.10 – Schéma d'une fonction récursive. La fonction s'appelle elle-même, toujours avec un autre paramètre en entrée, jusqu'à ce que la condition d'arrêt soit remplie. A ce moment-là, un résultat est calculé et retourné à la fonction du dessus (celle qui a appelé la fonction). Ainsi tous les résultats sont retournés au fur et à mesure et permettent de calculer la fonction souhaitée.

Les deux ingrédients indispensables à toute *fonction récursive* sont donc :

1. un **appel à la fonction elle-même** à l'intérieur de la définition de la fonction.
2. une **condition d'arrêt**, qui permet de terminer les appels imbriqués.

Exercice 1 – Position de la condition d'arrêt

Sans la condition d'arrêt, un programme récursif ne se termine pas, et s'appelle soi-même indéfiniment. Il est important que cette condition d'arrêt précède l'appel récursif à la fonction. Pourquoi est-ce le cas ?

Solution 1 – Position de la condition d'arrêt

Cliquer ici pour voir la réponse

Si la condition d'arrêt est après l'appel à la fonction, la fonction est appelée avant d'avoir pu vérifié si la condition d'arrêt est remplie. Dans ce cas, la condition d'arrêt n'est jamais testée.

Maintenant que nous avons programmé la première phase de division du Tri par fusion il nous faut programmer la deuxième phase de fusion (voir la deuxième figure du Tri par fusion). Nous allons définir cette phase de fusion de manière récursive :

```
# Phase de fusion du Tri par fusion
def fusion(elements_gauche, elements_droite):

    # trouve le plus petit premier élément des deux listes
    if elements_gauche[0] < elements_droite[0]:

        # appelle fusion récursivement avec le reste des listes
        elements_reste = fusion(elements_gauche[1:], elements_droite)

        # crée une liste fusionnée avec le résultat
        elements_fusion = [elements_gauche[0]] + elements_reste

    else:

        # appelle fusion récursivement avec le reste des listes
        elements_reste = fusion(elements_gauche, elements_droite[1:])

        # crée une liste fusionnée avec le résultat
        elements_fusion = [elements_droite[0]] + elements_reste

    return(elements_fusion)
```

Quelle est la différence entre le code dans la partie `if` de la condition et dans la partie `else` de la condition ? Lorsqu'on fusionne deux tableaux qui sont **déjà triés**, le plus petit élément se trouve parmi les premiers éléments des deux tableaux à fusionner. On commence alors par prendre le plus petit des premiers éléments des deux tableaux à fusionner, que l'on met au début de notre tableau fusionné. On refait ensuite la même opération avec le reste des éléments : on sélectionne le plus petit élément des tableaux de départ et on le met à la suite de notre tableau fusionné. On recommence de la sorte tant qu'il n'y ait plus d'éléments dans les tableaux.

Dans la partie `if` de la fonction `fusion`, c'est le tableau de gauche qui contient le plus petit élément. On prend cet élément pour le mettre au début d'un nouveau tableau fusionné et on appelle la fonction `fusion` sur les éléments restants. Dans la partie `else` on fait la même chose, sauf que l'on commence notre tableau fusionné par le premier élément du tableau de droite.

Mais n'y a-t-il pas quelque chose qui manque à cette fonction ? En effet, il manque la condition d'arrêt. Il faut arrêter la fusion lorsqu'un des deux tableaux à fusionner est vide. Dans ce cas la solution de fusionner un tableau vide avec un autre tableau est triviale : c'est l'autre tableau non vide. Mettons ceci sous forme de code :

```
# Phase de fusion du Tri par fusion
def fusion(elements_gauche, elements_droite):

    # conditions d'arrêt de la récursivité
    if elements_gauche == []:
```

(suite sur la page suivante)

(suite de la page précédente)

```

        return(elements_droite)
if elements_droite == []:
    return(elements_gauche)

# trouve le plus petit premier élément des deux listes
if elements_gauche[0] < elements_droite[0]:

    # appelle fusion récursivement avec le reste des listes
    elements_reste = fusion(elements_gauche[1:], elements_droite)

    # crée une liste fusionnée avec le résultat
    elements_fusion = [elements_gauche[0]] + elements_reste

else:

    # appelle fusion récursivement avec le reste des listes
    elements_reste = fusion(elements_gauche, elements_droite[1:])

    # crée une liste fusionnée avec le résultat
    elements_fusion = [elements_droite[0]] + elements_reste

return(elements_fusion)

```

Pour que le programme soit complet, il faut faire appel cette fonction `fusion` dans la fonction `tri_fusion` ci-dessus :

```

# Phase de division du Tri par fusion
def tri_par_fusion(elements):

    ### Phase DIVISION

    # condition d'arrêt la récursion
    if len(elements) <= 1:
        return(elements)

    # détermine l'indice au milieu du tableau (division entière)
    milieu = len(elements)//2

    # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
    elements_gauche = tri_par_fusion(elements[:milieu])

    # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
    elements_droite = tri_par_fusion(elements[milieu:])

    # fusionne les éléments un par un, puis deux par deux, etc..
    resultat = fusion(elements_gauche, elements_droite)

    # retourner le résultat
    return(resultat)

```

Ces deux *fonctions fusion et division* ensemble implémentent l'*algorithme* du Tri par fusion de manière récursive. La récursivité est un concept difficile à appréhender. Le mieux est d'essayer de coder différents algorithmes récursifs et d'afficher ce qui se passe au fur et à mesure. Voici le programme du tri par fusion :

```
# TRI PAR FUSION                                         1
# Phase de fusion                                         2
def fusion(elements_gauche, elements_droite):           3
    # conditions d'arrêt de la récursivité               4
    if elements_gauche == []:                            5
        print("\n4. Tableau gauche vide :", elements_droite) 6
        return(elements_droite)                           7
    if elements_droite == []:                            8
        print("\n5. Tableau droite vide :", elements_droite) 9
        return(elements_gauche)                           10
    # trouve le plus petit premier élément des deux listes 11
    if elements_gauche[0] < elements_droite[0]:          12
        # appelle fusion récursivement avec le reste des listes 13
        elements_reste = fusion(elements_gauche[1:], elements_droite) 14
        # crée une liste fusionnée avec le résultat            15
        elements_fusion = [elements_gauche[0]] + elements_reste 16
        # affiche ce qui se passe                            17
        print("\n6. Retour fusion :", [elements_gauche[0]], '+', 18
              elements_reste)                                19
    else:
        # appelle fusion récursivement avec le reste des listes 20
        elements_reste = fusion(elements_gauche, elements_droite[1:]) 21
        # crée une liste fusionnée avec le résultat            22
        elements_fusion = [elements_droite[0]] + elements_reste 23
        # affiche ce qui se passe                            24
        print("\n7. Retour fusion :", [elements_droite[0]], '+', 25
              elements_reste)                                26
    # retourner le résultat                               27
    return(elements_fusion)                            28
```

La fonction `division` s'appelle aussi elle-même, mais appelle également la fonction `fusion` précédemment définie :

```
# TRI PAR FUSION                                         1
# Phase de division                                       2
def division(elements, ligne, side=0):                   3
    # appelle division récursivement avec le reste des listes 4
    elements_reste = division(elements[1:], ligne, side) 5
```

```

# nous dit où on en est
print("\n1. Appel de la fonction division avec ", str(elements), "ligne",
      ligne, "depuis", side) 6

# correspond à la ligne sur la figure division du tri&nbsp;par&nbsp;fusion
ligne = ligne + 1 7

# condition d'arrêt la récursion
if len(elements) <= 1: 8
    return(elements)

# détermine l'indice au milieu du tableau (division entière)
milieu = len(elements)//2 9

# prend tous les éléments depuis le début, jusqu'à (et sans) milieu
elements_gauche = division(elements[:milieu], ligne, 'gauche') 10
if elements_gauche :
    print('\n2. Eléments à gauche : ', elements_gauche, 'de :', elements
          , "ligne", ligne) 11

# prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
elements_droite = division(elements[milieu:], ligne, 'droite') 12
if elements_droite :
    print('\n3. Eléments à droite : ', elements_droite, 'de :', elements
          , "ligne", ligne) 13

# fusionne les éléments un par un, puis deux par deux, etc..
resultat = fusion(elements_gauche, elements_droite) 14

# retourner le résultat
return(resultat) 15

resultat = division([3,5,1,2,6,4], 0) 16

print("\nVoici le tableau trié : ", resultat) 17

```

4.5.3 Exercices

Exercice 2 – Fractale

Une fractale est un objet géométrique, dont la définition récursive est naturelle. Essayez le code suivant pour différentes valeurs de n (augmenter à chaque fois de 1).

Essayez de comprendre comment le flocon se construit de manière **récursive**. Vous pouvez aussi varier la longueur du segment dessiné et la vitesse d'affichage en décommentant la ligne correspondante.

```

import turtle 1

def courbeKoch(n, segment) :
    if n == 0 : 2
        pass 3
    else : 4
        courbeKoch(n-1, segment) 5
        segment.forward(10) 6
        courbeKoch(n-1, segment) 7
        segment.left(60) 8
        courbeKoch(n-1, segment) 9
        segment.forward(10) 10
        courbeKoch(n-1, segment) 11

```

```

        turtle.forward(segment)      5
else :                         6
    courbeKoch(n-1, segment/3)  7
    turtle.left(60)            8
    courbeKoch(n-1, segment/3)  9
    turtle.left(-120)          10
    courbeKoch(n-1, segment/3) 11
    turtle.left(60)            12
    courbeKoch(n-1, segment/3) 13
                                14

def flocon(n, segment) :        15
    for i in range(3) :         16
        courbeKoch(n, segment) 17
        turtle.left(-120)       18

                                19

turtle.hideturtle() # cache la tortue 20
# turtle.speed(0) # ACCELERE LA TORTUE 21
turtle.forward(-10) # positionne la tortue en haut à gauche 22
turtle.left(150)     23
turtle.forward(150)   24
window = turtle.Screen() 25
window.bgcolor("black") # tableau noir 26
turtle.color("white") # dessine avec une trace blanche 27
turtle.setheading(0) # orientation initiale de la tête : droite 28

                                29

# AUGMENTER ICI 30
n = 1                         31
# DIMINUER ICI 32
segment = 300                  33
flocon(n, segment) # dessine le flocon 34
turtle.exitonclick() # garde la fenêtre ouverte 35

```

Exercice 3 – Une question de fusion

Trier le tableau suivant avec l'algorithme de tri par fusion : [3, 6, 8, 7, 1, 9, 4, 2, 5] à la main.
Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 4 – Dans l'autre sens

En Python, proposer une fonction qui inverse l'ordre des lettres dans un mot. Vous pouvez parcourir les lettres du mot directement ou à travers un indice.

Proposer une autre fonction qui inverse l'ordre des lettres dans un mot de manière récursive.

Exercice 5 – Factorielle

La fonction factorielle $n!$ en mathématiques est le produit de tous les nombres entiers jusqu'à n . C'est une des fonctions les plus simples à calculer de manière récursive. Elle peut être définie comme ceci :

$$n! = (n - 1)! * n$$

Programmer cette fonction de manière récursive en Python. Proposer également une implémentation itérative de la factorielle où les éléments de 1 à n sont traités l'un après l'autre.

4.6 Conclusion

Important

L'analyse de complexité des algorithmes nous permet de sélectionner les meilleurs algorithmes pour un problème donné et nous permet de comprendre pourquoi certains problèmes ne peuvent pas être (à ce stade) résolus dans un temps raisonnable.

L'algorithme a permis de mettre en place des stratégies intelligentes de résolution de problèmes comme les principes de « diviser pour régner » ou encore la récursivité. Ces stratégies ont rendu possibles les avancées technologiques fulgurantes du dernier demi-siècle.

Pour des problèmes difficiles, s'il est impossible de trouver la solution exacte, on peut souvent trouver une solution approchée. L'étude formelle de l'algorithme nous permet d'estimer la qualité de notre solution approchée.

À retenir

Dans la pratique, il est important de garantir qu'un algorithme va se **terminer**.

L'algorithme de tri rapide (et du tri par fusion) est plus efficace que les algorithmes de tri vus dans les chapitres précédents. Ceci est possible grâce à la stratégie algorithmique « **diviser pour régner** », qui divise un grand problème difficile à résoudre en plein de petits sous-problèmes plus faciles à résoudre. La solution au grand problème s'obtient en combinant les solutions des petits problèmes.

L'**ordre de complexité** d'un algorithme nous dit si l'algorithme est lent ou rapide. Un algorithme avec un ordre de complexité logarithmique est plus rapide qu'un algorithme avec complexité linéaire, qui à son tour est plus rapide qu'un algorithme de complexité quadratique, ou pire, exponentielle.

Un algorithme **récursif** est un algorithme qui fait appel à lui-même. Une condition d'arrêt est nécessaire pour que l'algorithme se termine.

Un algorithme avec une **complexité exponentielle** implique que le temps nécessaire pour résoudre problème est trop long en pratique. Dans ce cas, on ne va pas pouvoir trouver une solution exacte, mais seulement une solution approchée en utilisant des méthodes heuristiques.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais appliquer l'algorithme de recherche binaire.
2. Je comprends comment fonctionne la stratégie algorithmique « diviser pour régner ».
3. Je sais appliquer l'algorithme de tri rapide.
4. Je sais calculer la complexité temporelle d'un algorithme.
5. [En option] Je comprends comment fonctionne la récursivité.
6. Je sais pourquoi un algorithme de complexité exponentielle est lent.

7. Je comprends ce qu'est une solution heuristique.

Pour aller plus loin – Quelques liens web

Visualisation de problèmes

<https://imgur.com/gallery/voutF>

<https://interstices.info/le-probleme-du-sac-a-dos/>

<https://visualgo.net/en>

<https://graphonline.ru/fr>

<https://clementmihalescu.github.io/Pathfinding-Visualizer/>

Problèmes difficiles

<https://www.franceculture.fr/emissions/le-journal-des-sciences/le-journal-des-sciences-du-mardi-01-decembre-2020>

https://www.bfmtv.com/sciences/ou-est-charlie-l-algorithme-pour-le-detecter-du-premier-coup_AN-201502100004.html

<https://www.lebigdata.fr/algorithme-definition-tout-savoir>

P = NP ?

<https://www.youtube.com/watch?v=AgtOCNCejQ8>

CHAPITRE 5

Réseaux

Aujourd’hui quand nous parlons de **réseaux**, il n’est pas toujours clair s’il s’agit de réseaux sociaux, du web, de la 4G, du wifi ou plus généralement d’Internet, le réseau des réseaux. Dans ce chapitre, nous allons parler d’Internet, qui est une infrastructure permettant à des machines de communiquer entre elles sans être directement connectées entre elles.

5.0 Introduction

Internet est une infrastructure essentielle qui a complètement changé notre vie, que ce soit dans les relations sociales, l’éducation, la recherche, le commerce, la santé, etc. Dans ce chapitre, nous allons voir, dans les grands principes, comment Internet fonctionne et en quoi son fonctionnement est différent d’autres réseaux de communication qui l’ont précédé, tels que les réseaux postaux ou téléphoniques. En effet, une innovation majeure d’internet est qu’il s’agit d’un réseau *décentralisé*, et ceci explique dans une large mesure son succès et ce qu’il est devenu, même si certains craignent une recentralisation d’Internet autour des géants du numérique tels que Google et Amazon.

5.0.1 Origine d’Internet

Les réseaux de communication existaient bien avant Internet, par exemple :

- le réseau de télégraphie optique de Chappe (1794)
- le réseau de télégraphie électrique de Morse (1843)
- le réseau téléphonique de Bell (1877)
- le réseau de télégraphie par ondes radio de Marconi (1896)

Ces anciens réseaux avaient besoin d’opérateurs ou opératrices pour la transmission des messages, ou, pour les plus récents, ils étaient centralisés. Cela signifie qu’il y a un point central du réseau par lequel passent toutes les communications. Après la 2e guerre mondiale, qui avait vu le nivellement de villes entières par des bombardements aériens (comme à Dresde) et la bombe atomique, l’armée américaine a décidé de financer

le développement d'un réseau de communication décentralisé (ou distribué), qui serait moins vulnérable à une attaque. L'idée était qu'un réseau de communication centralisé pouvait facilement être mis hors service par un adversaire en détruisant le point central, alors qu'un réseau de communication sans point central serait beaucoup plus difficile à attaquer.

Document historique

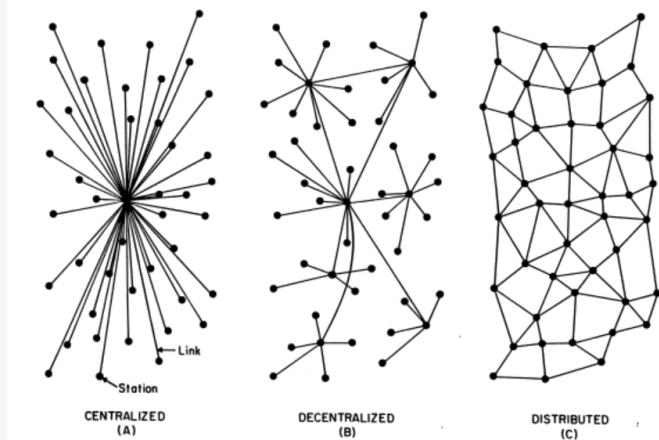


FIG. 5.1 – Image tirée de l'article proposant de réaliser un réseau décentralisé. Baran, *On Distributed Communications : I. Introduction to distributed communications networks*, RAND CORP CALIF, 1964, disponible [ici](#). Cette image illustre la différence entre un réseau centralisé (à gauche), dans lequel toutes les communications passent par un point central, et un réseau distribué (droite), dans lequel tous les noeuds ont plus ou moins la même importance. Le réseau du milieu représente un intermédiaire décentralisé, entre le réseau complètement centralisé de gauche et le réseau distribué de droite

Les universitaires américains ont été associés à la conception de ce réseau et l'ont utilisé pour partager des informations et des ressources entre universités. Ainsi est né Internet, par une association entre universitaires attachés surtout à la libre circulation de l'information et des militaires aux visées plutôt sécuritaires. Dès les années 70, le mouvement hippie, séduit par les possibilités d'auto-organisation et la philosophie non hiérarchique d'Internet a investi cette infrastructure et a développé une “cyberculture” qui marquera durablement l'histoire d'Internet, de l'émergence des réseaux sociaux aux cryptomonnaies. En 1983, les militaires ont déconnecté leur partie du réseau du reste d'Internet pour des raisons de sécurité.

5.0.2 Structure d'internet

Internet est souvent décrit comme un *réseau de réseaux*. En effet, Internet est construit sur une structure de *réseaux locaux* interconnectés les uns aux autres. Par exemple, les ordinateurs d'une école, d'une entreprise ou d'un appartement peuvent être reliés entre eux par le wifi, ou des câbles Ethernet et constituer un réseau local. Le réseau local est ensuite connecté, par le biais d'un *routeur*, au reste d'Internet. Ainsi Internet est constitué d'une myriade de sous-réseaux connectés (et potentiellement enchâssés) les uns aux autres. Ces réseaux sont connectés par les *dorsales d'Internet*, des câbles de fibre optique capables de transférer des données à haut débit, qui traversent les continents et les océans.

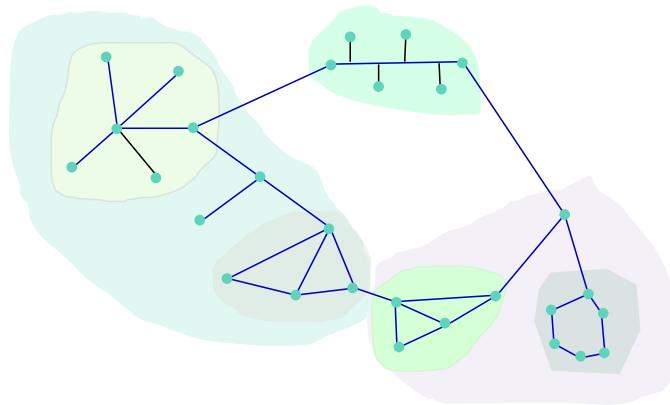


FIG. 5.2 – Un réseau de sous-réseaux. Les points représentent les machines, alors que les traits indiquent les connexions entre les machines. La couleur de fond indique les sous-réseaux.

Micro-activité – Les câbles sous-marins d’Internet

Aller sur le site <https://www.fiberatlantic.com/submarinecablemap/> et regarder la carte des câbles sous-marins d’Internet. Trouver le câble qui relie l’Afrique du Sud à l’Inde. Comment s’appelle-t-il, depuis quand existe-t-il et quelle est sa longueur ? À qui appartient-il et quand est-il prévu de le mettre hors service ?

5.0.3 Fonctionnement d’Internet

Cette section présente une vue d’ensemble des éléments centraux du fonctionnement d’Internet et qui seront repris dans la suite du chapitre.

Adressage

Tout réseau de communication a besoin d’un système d’adresses afin de pouvoir distinguer et joindre les différents destinataires. Dans un réseau décentralisé qu’est Internet, le système d’adressage doit permettre à chaque machine connectée au réseau d’être identifiable et joignable, sans causer de quiproquo. Cela passe par une gestion hiérarchique des adresses.

Routage

Dans un réseau centralisé comme le téléphone traditionnel, un opérateur central (le standard dans le cas du téléphone) relie tous les appareils branchés au réseau. Lorsque deux personnes veulent entrer en communication, l’opérateur central les met en relation, c’est-à-dire relie leurs appareils. C’était d’abord fait à la main, puis automatiquement (avec le téléphone à cadran rotatif). Dans un réseau décentralisé, la mise en lien doit se faire de manière décentralisée. C’est ce qu’on appelle le *routage*.

Commutation par paquets

Dans un réseau centralisé comme le téléphone, lorsque deux personnes sont en communication, elles “occupent la ligne” : ces deux personnes ne sont pas joignables par d’autres personnes (c’est ce qu’indique le signal “occupé”), mais cela n’implique pas d’autres personnes qui veulent communiquer entre elles (sauf si le standard est surchargé). C’est ce qu’on appelle la *commutation de circuits*, car un circuit électrique est créé entre les deux appareils qui communiquent.

Imaginons maintenant qu’une relation soit établie entre deux appareils d’un réseau décentralisé. Si l’on avait une commutation de circuits, toutes les connexions utilisées pour acheminer l’information entre deux appareils seraient inutilisables pour les autres personnes utilisatrices du réseau, comme illustré dans la figure ci-dessous.

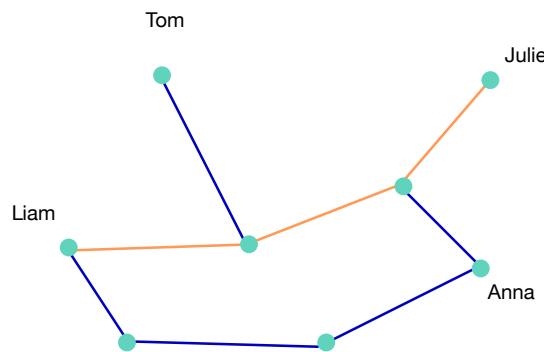


FIG. 5.3 – Si on réservait un circuit entre les ordinateurs de Liam et Julie (en orange) lorsque ceux-ci sont en communication par internet, Tom ne pourrait pas entrer en même temps en communication avec Anna, car tous les segments oranges seraient occupés.

Cela pourrait conduire rapidement à une saturation du réseau. Pour éviter ce problème, il a été décidé de “découper” la communication en petits morceaux (appelés les paquets) et d’envoyer chaque morceau individuellement. C’est ce qu’on appelle la *commutation par paquets*. C’est un peu comme si au téléphone on raccrochait et on se rappelait entre chaque mot. Cela ne monopolise pas la ligne et permet de mener plusieurs conversations en même temps sur la même ligne. En plus, cela ne sollicite la ligne que lorsqu’une information est transmise et pas, par exemple, pendant les moments de silence, ou lorsqu’on est en train de rédiger sa réponse sur une messagerie.

Les protocoles

Lorsque deux personnes entrent en communication, elles se mettent d’accord (souvent implicitement) sur la manière dont elles communiquent, par exemple la langue qu’elles vont utiliser, la manière dont on signifie le début et la fin d’une conversation (les salutations), etc. Pour les machines, ces règles doivent être précisées beaucoup plus en détails, car les machines n’ont pas la faculté d’interprétation et d’adaptation des humains. Dans un réseau décentralisé, il faut spécifier quels sont exactement les signaux que doit envoyer une machine qui veut se connecter au réseau et établir une communication avec une autre machine. C’est ce qu’on appelle le *protocole de communication*, par exemple comment la machine indique qu’elle veut établir ou fermer une communication, comment elle indique avec qui elle veut établir une communication, etc. Il y a pleins de protocoles différents, par exemple le protocole qui régit comment une machine veut se connecter par wifi à un routeur, le protocole qui régit comment demander le contenu d’une page web à un serveur (HTTP), le protocole qui régit comment envoyer un email sur le réseau (SMTP), ou se connecter à distance sur une

autre machine (SSH). Ces protocoles peuvent être ouverts (ou publics) est cela permet à chacun ou chacune de les utiliser, ou ils peuvent être fermés (ou privés) ce qui limite leur utilisation à l'entreprise ou l'entité qui les a inventés.

5.0.4 Organisation du chapitre

Dans le reste de ce chapitre, nous allons aborder plus concrètement les notions décrites ci-dessus et en approfondir certains aspects. Pour illustrer notre propos, considérons la situation suivante.

Ai-je compris ? – l'application aux champignons

Imaginons qu'Alice est partie à la cueillette aux champignons dans la forêt. Elle pense avoir trouvé un beau bolet, mais pour plus de sécurité, consulte avec son téléphone portable un site web spécialisé dans les champignons de notre région, www.champignons.ch. Que se passe-t-il réellement entre derrière l'écran de son téléphone ? C'est ce que nous allons découvrir dans ce chapitre.

5.1 Adressage

L'adresse est une notion importante en communication, qui permet à une personne ou une machine de s'adresser à une autre personne ou machine spécifique. Le rôle d'un système d'adressage, tel que celui de la poste, est de permettre d'identifier et de joindre une destination sans ambiguïté, et c'est pour ceci que chaque adresse doit être unique.

5.1.1 Les noms de domaine

Le nom *champignons.ch* est ce qu'on appelle un *nom de domaine*. Les noms de domaines sont gérés par l'ICANN, une organisation non gouvernementale à but non lucratif basée aux États-Unis dont la fonction principale est la gestion de l'adressage sur Internet. Les noms de domaines sont gérés de manière hiérarchique, selon le *nom de domaine de premier niveau*, c'est à dire la “terminaison” de l'adresse (.ch, .org, .fr, etc.) Ainsi la gestion des adresses en .ch est confiée à Switch, une fondation suisse dont c'est le rôle principal. La personne qui a créé le site *champignons.ch* a donc réservé ce nom de domaine auprès de Switch (en passant par un intermédiaire) et peut le conserver moyennant un paiement d'environ CHF 15.- par an.

Le saviez-vous ?

Au début, les noms de domaine de premier niveau étaient limités à quelques possibilités, telles que “.com” pour les organisations commerciales, “.edu” pour les universités (américaines), “.gov” pour le gouvernement (américain), “.mil” pour l'armée (américaine), “.org” pour les organisations (à but non lucratif) et, dès les années 80, différents pays ont décidé d'enregistrer des noms de domaine de premier niveau pour leur pays, par exemple “.ch” pour la Suisse, “.fr” pour la France. Puis il a été décidé d'ouvrir d'autres noms de domaine et de les mettre aux enchères. Une entreprise a alors décidé de vendre des domaines “.sucks” qu'elle a vendus très cher à certaines grandes entreprises (par exemple apple.sucks) qui avaient peur que ce site ne devienne une plateforme pour les critiquer.

Si les noms de domaines sont pratiques pour désigner des adresses sur Internet, les machines, elles, utilisent des nombres pour référencer les machines connectées à Internet, c'est ce qu'on appelle les *adresses IP*. Ainsi, la personne qui a enregistré le site *champignons.ch* a également reçu une (ou plusieurs) adresse IP de la part de Switch ou d'un intermédiaire.

Micro-activité

Déterminer à l'aide du site web <https://www.nic.ch/whois/> qui a enregistré le nom de domaine champignons.ch.

5.1.2 Les adresses IP

Version 4 (IPv4)

Afin de pouvoir identifier chacune des machines connectées à Internet, il a été décidé de leur attribuer à chacune un nombre, un peu à la manière dont les numéros de téléphone sont attribués à chaque téléphone du réseau téléphonique. Dans sa version la plus courante, ce nombre est codé sur 32 bits, ce qui donne à peu près 4 milliards de possibilités (2^{32}). On pensait alors (c'était en 1982) que 4 milliards d'adresses seraient amplement suffisantes pour pouvoir accommoder toutes les machines pendant encore beaucoup d'années, et qu'Internet ne dépasserait pas les 4 milliards de machines connectées. A cette époque, il n'y avait que quelques centaines d'ordinateurs connectés à Internet. Afin de rendre ces adresses plus lisibles pour les humains, on décompose d'habitude une adresse IP de 32 bits en quatre groupes de 8 bits séparés par un point. Chaque groupe de 8 bits peut alors être représenté comme un nombre décimal entre 0 et 255 ($2^8 - 1$).

Exercice 1

Lesquelles des adresses suivantes sont des adresses IP valides :

1. 240.264.23.2
2. 123.8.12.2.34
3. 123.23.2
4. 205.233.12.23

Version 6 (IPv6)

Avec le développement d'Internet, il est vite devenu clair que le nombre de machines connectées à Internet allait dépasser le nombre d'adresses IP différentes, et c'est pourquoi un nouveau type d'adressage a été développé dès les années 90, IPv6 (Internet Protocol, version 6). Il a été décidé de coder les adresses IP sur 128 bits. Plutôt que de les représenter avec 16 nombres entre 0 et 255, il a été décidé de coder en 8 nombres hexadécimaux entre 0000 et FFFF. Chaque chiffre de 0 à F représente ainsi 4 bits, et chaque nombre de 4 chiffres hexadécimaux représente donc $4 \cdot 4 = 16$ bits. En prenant 8, on arrive bien à $8 \cdot 16 = 128$ bits.

Par exemple 4E3F.DEA7.409B.412C.2516.4A2B.2CFE.1282 pourrait constituer une adresse IPv6 valide. Elle est en effet constituée de 8 nombres à quatre chiffres hexadécimaux.

Actuellement, les deux types d'adresses IPv6 et IPv4 coexistent sur Internet, la version IPv4 étant encore largement plus répandue. Une adresse IP peut donc soit être sur 32 bits soit sur 128 bits.

Exercice 2

Parmi les adresses suivantes, indiquer lesquelles sont au format IPv4, lesquelles sont IPv6 et lesquelles ne sont pas valides. Justifier sa réponse.

1. 128.23.54.45
2. 31.43.132.45.51.654.4355.4325
3. 1923.2123.1323.4324.4241.2434.7657.5757

4. ADEFE.ACDEA.AABCD.DDEBC.FFEDA.AEABC.ACade.EFDF
5. 1230.121D.12AEAB.1231D.4324B.2765.5435D.4378
6. D2G3.4234.534FG.2141.12GE.12AD.85C2.GE32
7. 123A.3213.564E.6746.2DD2.A897
8. 124.234.432.21

Gouvernance

Comme les noms de domaine, les adresses IP sont gérées hiérarchiquement. Ainsi, les adresses IPv4 de la forme 46.x.x.x (c'est-à-dire celles qui commencent par 46 = 00101110) sont assignées au Centre de Coordination Européen qui les répartit entre différents *Registres Internet locaux* tels que Switch qui va pouvoir louer une partie de ces adresses IP à des organisations, des entreprises (par exemple des fournisseurs d'accès Internet) ou des particuliers qui en feraient la demande.

Certains blocs d'adresses IP sont réservés à des usages particuliers. Par exemple les adresses 10.x.x.x ou 192.168.x.x sont réservées aux réseaux privés, c'est-à-dire des machines qui ne communiquent pas directement avec le reste d'Internet. Ainsi, ces adresses peuvent être utilisées au sein du réseau interne des entreprises, ou pour faire communiquer différents appareils connectés (imprimante, télévision, ordinateurs, ou smartphones) au sein d'une maison. Dans l'exemple ci-dessous, un fournisseur d'accès à Internet (tel que Swisscom par exemple) a reçu toutes les adresses de type 213.221.x.x. Il en garde une partie pour son propre usage, par exemple pour son site web et les machines qui opèrent le réseau. Une autre partie des adresses sera louée à des entreprises ou des particuliers qui sont ses clients. Ceux-ci bénéficieront donc d'une adresse IP leur permettant d'être joignables par le reste d'Internet.

Micro-activité

- Déterminer à l'aide de cette page Wikipedia⁶² à quel continent sont allouées les adresses IP suivantes :
 - 212.x.x.x
 - 154.x.x.x
 - 20.x.x.x
- Déterminer à l'aide de ce site⁶³ l'entité suisse qui possède le plus d'adresses IP

62. https://en.wikipedia.org/wiki/List_of_assigned/_8_IPv4_address_blocks

63. <https://www.nirsoft.net/countryip/ch.html>

Exercice 3

- Combien y a-t-il d'adresses IP de type 192.168.x.x ?
- Combien y aurait-il eu d'adresses IP possibles s'il avait été décidé de l'encoder sur 24 bits ?
- Donnez la représentation binaire de l'adresse IP 10.0.45.12

Réseau privé

Les particuliers et entreprises ont généralement un réseau privé, un *intranet*, qui utilise les adresses 10.x.x.x. L'appareil qui permet de connecter ce réseau privé au reste d'Internet est un *routeur*, par exemple la boîte wifi qui est fournie par le fournisseur d'accès. Ce routeur a à la fois une adresse locale (dans notre exemple 10.0.1.1) pour être joignable depuis le réseau privé et une adresse globale (213.221.190.41 dans l'exemple ci-dessous) pour être atteignable depuis le reste d'Internet. Le routeur joue un peu le rôle du secrétariat de l'école en s'occupant de transmettre le courrier entre l'intérieur et l'extérieur de l'école. De manière similaire, le secrétariat a d'habitude deux boîtes aux lettres, une pour les documents déposés par des personnes qui sont à l'intérieur de l'école (élèves, personnel enseignant) et une destinée au facteur qui amène le courrier en provenance de l'extérieur du gymnase.

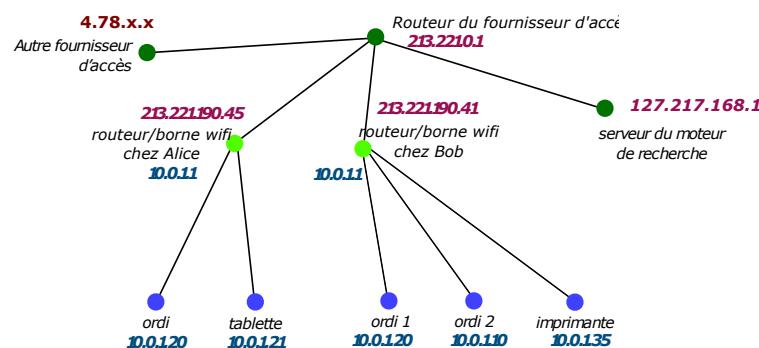


FIG. 5.4 – Exemple de distribution des adresses IP, avec un fournisseur d'accès ayant obtenu les adresses 213.221.x.x, rt qui fournit un accès Internet à Alice et Bob. Les routeurs, en vert clair, ont deux adresses IP.

Micro-activité

- A l'aide d'un navigateur web, aller sur le site <https://www.whatismyip.com> et déterminer sa propre adresse IP.
- Dans un terminal taper la commande suivante qui détermine votre adresse IP :
 - sur Mac Os ou Linux : `ipconfig getifaddr en0`
 - sur Windows : `ipconfig`
- Obtient-on la même réponse ? Pourquoi ?

Adressage statique et dynamique

Une adresse IP peut être allouée de manière *statique* ou *dynamique*. Dans le cas de l'adressage statique, on configure la machine en lui indiquant son adresse IP, est c'est elle qui annonce au réseau quelle est son adresse IP, afin que les messages puissent lui parvenir. La machine conserve ainsi toujours la même adresse IP, de la même façon qu'un téléphone conserve toujours le même numéro (sauf si on le reconfigure en modifiant par exemple la carte SIM). Dans le cas de l'adressage dynamique, la machine demande une adresse IP au moment où elle se connecte à Internet. Cette demande se fait auprès d'un serveur qui va lui allouer une adresse IP disponible parmi celles qu'il a à disposition. C'est un peu comme si chaque fois qu'on allumait son téléphone, on recevait un autre numéro pour être joignable. Si c'est nous qui initions les appels, cela ne pose pas vraiment de problème, mais si on veut être joignable, cela devient problématique, car les autres ne sauront pas comment nous trouver. Mais cela a d'une part l'avantage d'éviter qu'une machine non connectée monopolise une adresse IP sans l'utiliser et d'autre part, cela donne un (petit) degré d'anonymat et de sécurité en plus, car il sera plus difficile de cibler précisément notre machine et intercepter nos messages sur Internet.

Ainsi les serveurs (les sites web, par exemple), qui doivent être joignables en tout temps ont généralement une adresse IP statique, alors que les machines des utilisateurs et utilisatrices ont souvent une adresse IP dynamique. Lorsqu'on fait un abonnement Internet, le fournisseur d'accès propose d'habitude une adresse IP dynamique (cela lui permet d'économiser les adresses IP en sa possession), mais il est également possible, en payant un peu plus, d'obtenir une adresse IP statique.

Micro-activité

En regardant les paramètres réseaux, déterminer si sa machine a une adresse IP statique (manuel) ou dynamique (DHCP).

Exercice 4

1. Vous souhaitez entrer en communication avec votre ami-e, mais vous avez les deux des adresses IP dynamiques. Quel moyen pourriez-vous imaginer pour que vous puissiez vous joindre.
2. En tant que propriétaire d'un site web, vous avez accès aux adresses IP des machines qui visitent votre site. Pouvez-vous dès lors identifier une même personne qui revient plusieurs fois sur votre site ?
3. Depuis votre adresse IP dynamique, vous êtes entré en communication avec un site web illégal. La police peut-elle vous retrouver à partir de votre adresse IP ? Si oui comment, si non pourquoi ?

Solution 4

1. Vous pouvez vous connecter tous deux à un serveur central qui a une adresse IP fixe et qui s'occupera de relayer vos messages à vos adresses dynamiques. C'est ce que fait un serveur mail ou de messagerie telle que Signal ou WhatsApp.
2. Si elle a une adresse IP dynamique, alors elle aura probablement des adresses IP différentes lors de ses visites en des jours différents. On ne pourra donc pas l'identifier en regardant uniquement son adresse IP. Par contre, en enregistrant d'autres paramètres que son navigateur voudra bien nous transmettre, tels que son système d'exploitation, la langue, l'appareil, etc., on peut reconstituer son empreinte numérique et l'identifier ainsi. C'est ce qu'on appelle en anglais le *fingerprinting*, que l'on peut bloquer avec certains navigateurs⁶⁴.
3. Oui, votre fournisseur d'accès à Internet doit garder une trace de quelle adresse IP a été allouée à qui et à quel moment. La police peut dès lors exiger ces informations en cas de soupçons.

64. <https://www.mozilla.org/fr/firefox/features/block-fingerprinting/>

5.1.3 Système de noms de domaine

Pour récapituler ce qui a été vu précédemment, les humains utilisent les noms de domaines pour les machines, alors que les machines, elles, utilisent les adresses IP. Afin que ces deux modes de recensement des machines soient cohérents entre eux, il est nécessaire de disposer d'un annuaire qui fera correspondre les noms de domaines aux adresses IP. Ceci est analogue aux annuaires téléphoniques ou aux contacts du smartphone qui permettent de faire correspondre le nom des personnes que l'on veut atteindre (qui serait équivalent au nom de domaine) au numéro de téléphone (qui est analogue à l'adresse IP). Cet annuaire est ce qu'on appelle le *système de noms de domaine* (Domain Name System ou DNS selon l'appellation anglaise). Au début d'Internet, il s'agissait simplement d'un fichier texte librement accessible qui listait le nom de domaines et les adresses IP correspondantes. Ce fichier était maintenu à la main. Maintenant, il s'agit de machines, les serveurs DNS dans le réseau auprès desquelles il est possible d'obtenir l'adresse IP correspondante à un nom de domaine.

Ces machines sont aussi organisées hiérarchiquement de telle sorte que chaque serveur DNS ne stocke que les noms de domaines correspondant à une sous-partie du réseau.

Le saviez-vous ? – Le hacking de DNS

Une méthode de hacking consiste à mettre en ligne un serveur DNS malveillant qui va diriger le traffic vers des faux sites web se faisant passer pour des vrais. Par exemple, un hacker pourrait mettre en ligne un DNS malveillant indiquant une fausse adresse IP pour le site google.com, et à cette adresse, mettre un serveur web ayant la même page d'accueil que Google. Lorsque une quelqu'un essaiera de se connecter à son compte google sur le faux site, ce site enregistrera simplement le login et mot de passe et renverra sur le vrai site web. Le hacker aura ainsi le login et mot de passe du compte google de la personne, pouvant ainsi avoir accès à ses emails et documents. La difficulté pour le hacker est de "convaincre" que son serveur DNS est fiable.

Le saviez-vous ? – La censure par le DNS

Une des méthodes à disposition d'un état qui souhaite empêcher ses habitants d'accéder à certains sites consiste à interdire aux serveurs de DNS du pays de répondre correctement aux requêtes concernant certains noms de domaine, voire de renvoyer des fausses adresses IP lorsque les requêtes DNS sont interceptées. En Chine, par exemple, Facebook.com est interdit, et les DNS chinois vont refuser de retourner l'adresse IP du site de Facebook. Cette censure peut parfois être contournée en recourant à des serveurs DNS situés à l'extérieur du pays.

Ai-je compris ? – L'exemple d'Alice

L'organisation qui a développé l'application aux champignons, a obtenu le nom de domaine *champignons.ch* et une adresse IP statique. Pour qu'Alice puisse aller sur ce site, son téléphone va envoyer une requête à un serveur DNS avec le nom de domaine "champignons.ch". Cette requête transitera par différents serveurs DNS organisés hiérarchiquement jusqu'à ce qu'un serveur DNS puisse y répondre, et la réponse sera retransmise jusqu'au téléphone d'Alice. Le téléphone d'Alice a lui aussi reçu une adresse IP dynamique de la part de son opérateur téléphonique afin que le serveur web puisse lui envoyer la page web du site.

5.2 Paquets et protocoles

Une fois que l'on connaît l'adresse d'un destinataire, il est possible d'établir un contact et de lui transmettre de l'information. Sur Internet, ceci se fait en découplant cette information en petits paquets que l'on étiquette de façon bien précise. La manière dont ceci se fait est définie par les *protocoles* d'Internet.

5.2.1 Les paquets

Dès leur origine, les systèmes de communication se sont développés selon deux modes distincts selon les supports utilisés. Soit on maintient un “canal de communication” ouvert par exemple avec le téléphone ou la communication radio (le talkie-walkie). Dans ce cas, le récepteur et l'émetteur entrent en communication et l'information est envoyée de manière continue de l'émetteur au récepteur. Le récepteur ne peut pas être en communication avec plusieurs émetteurs à la fois. Dans le second cas de figure, par exemple le courrier postal ou le télégramme, les informations sont envoyées “en bloc”, typiquement par messages acheminés en une fois. Dans ce cas, le récepteur peut recevoir des messages de différentes personnes de manière presque simultanée, et le fait d'envoyer un message à quelqu'un ne va pas empêcher quelqu'un d'autre d'entrer en communication et nous envoyer des messages.

Afin d'éviter de bloquer les lignes de communication, Internet s'est développé selon ce second mode, et c'est pourquoi il était justifié d'évoquer ci-dessus des *messages* qui étaient envoyés et circulaient dans le réseau. En effet, toute information envoyée par Internet est découpée en petits *paquets* qui sont envoyés indépendamment les uns des autres. Ainsi, lorsque le serveur hébergeant le site www.champignons.ch va envoyer une image de champignon à Alice, cette image sera découpée en petits paquets qui seront chacun envoyés séparément à Alice. Cela a l'avantage que si, pour une raison ou une autre, une partie de l'image se perd en route, il n'y a pas besoin de renvoyer toute l'image, mais uniquement les parties qui se sont perdues. Cela permet aussi à une machine de maintenir plusieurs canaux de communications ouverts simultanément. C'est ce qu'on appelle la *commutation par paquets* parce que ce sont les paquets qui sont adressés individuellement à leur destinataire. À l'inverse, dans le cas du téléphone traditionnel, lorsqu'on appelle quelqu'un, un circuit électrique est établi entre les deux téléphones pour leur permettre de communiquer (à l'exclusion des autres téléphones), c'est ce qu'on appelle la *commutation de circuits*.

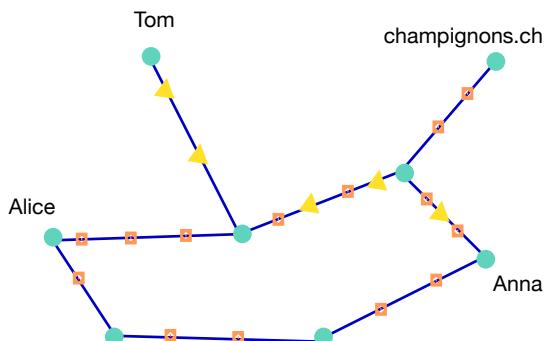


FIG. 5.5 – Les données envoyées de Alice à *champignons.ch* sont découpées en petits paquets (représentés par des carrés orange). Cela permet de partager les lignes avec d'autres utilisateurs et utilisatrices tels que Anna et Tom qui communiquent également en s'envoyant des paquets (représentés par des triangles jaunes). On peut noter que ces paquets ne prennent pas tous forcément le même chemin pour arriver à destination.

Les protocoles IP (Internet Protocol) et TCP (Transmission Control Protocol) décrivent le format ainsi que la gestion possible de ces paquets.

5.2.2 Le protocole IP

L'envoi d'un paquet par la poste suit certaines règles, telles que la position et le format de l'adresse de destination, la position et le format de l'adresse d'expédition, la position du timbre et son montant en fonction du poids et de la destination. Sans ces règles, l'acheminement du paquet ne peut pas être assuré. De manière analogue l'envoi d'un paquet sur Internet doit suivre certaines règles pour être acheminé. C'est le protocole IP qui définit ces règles.

Selon ce protocole un paquet est constitué d'une suite de 0 et de 1 que l'on peut séparer en deux parties.

1. L'entête qui donne des informations sur le paquet (son émetteur, sa destination, sa taille, etc.)
 2. Les données (appelées aussi la *charge utile*) qui forment le contenu du paquet, c'est-à-dire les informations que l'on veut transmettre.



L'entête joue le rôle de l'étiquette sur un paquet envoyé par la poste. On y indique l'adresse de destination, l'adresse de l'expéditeur (appelée aussi l'adresse source), mais aussi d'autres informations telles que la version d'IP utilisée (4 ou 6), la longueur totale du paquet, ainsi que sa "durée de vie". Sa durée de vie indique au bout de combien de temps le paquet peut être abandonné pour éviter d'avoir des paquets qui circulent indéfiniment sans trouver leur destinataire. Dans la version IPv4, l'entête fait au minimum 20 octets, remplis comme dans l'image ci-dessous.

Ainsi, les 4 premiers bits indiquent la version d'IP utilisée (donc 0100 si c'est la version 4), les quatre suivants donnent la longueur de l'entête en lignes de 32 bits, et la longueur totale (en octets) du paquet est données par les troisième et quatrième octets de l'entête IP. L'adresse IP de la source occupe les octets 13 à 16, et celle de la destination les octets 17 à 20.

Exercice 1

Un paquet avec l'entête IP suivante (en hexadécimal) circule sur Internet :

45 00 00 14 00 01 00 00 0A 00 BF 88 C1 C8 DC EA 91 E8 C0 C5

Déterminer de quelle version de protocole IP il s'agit, la longueur du paquet ainsi que les adresses IP (en binaire) de l'émetteur et du récepteur.

Solution 1

Chaque chiffre hexadécimal représente 4 bits, et donc chaque nombre à deux chiffres représente un octet (8 bits). Selon la spécification de l'entête, d'IP est donnée par les 4 premiers bits, donc le premier chiffre de l'entête qui est 4. C'est donc en entête en IPv4. La longueur du paquet est donnée en hexadécimal par les octets 3 et 4, donc 00 14 c'est à dire 20 octets (en décimal). Ce paquet ne contient donc pas de données. L'adresse de l'émetteur (l'adresse source) est donnée à la quatrième ligne de 32 bits (ou 4 octets), c'est donc C1 C8 DC EA en hexadécimal, c'est à dire 1100 0001 1100 1000 1101 1100 1110 1010 en binaire. La ligne suivante donne l'adresse de destination qui est 91 E8 C0 C5 en hexadécimal ou 1001 0001 1110 1100 0000 1100 0101 en binaire.

5.2.3 Le protocole TCP

Contrairement à une lettre dans laquelle on peut écrire tant qu'on veut, un paquet IP a une taille maximale fixe de 65535 octets, et donc on sera parfois obligé de découper une information (par exemple une image ou une vidéo) en plusieurs paquets IP afin de l'envoyer. Le récepteur doit ensuite reconstruire l'information à partir des paquets reçus et confirmer qu'il a bien tout reçu et que rien n'a été perdu en route (ce qui arrive parfois, comme avec la poste). Le protocole TCP (Transmission Control Protocol) permet aux machines réceptrice et émettrice de s'assurer que l'information a bien été transmise et reconstituée.

Pour cela, l'information est découpée en morceaux de taille inférieure à la taille maximale des paquets IP, et chaque morceau est numéroté (avec des nombres consécutifs) et envoyé dans un paquet IP. La machine réceptrice sait ainsi comment reconstituer l'information et peut vérifier qu'il ne lui manque pas de morceaux.

Exercice 2

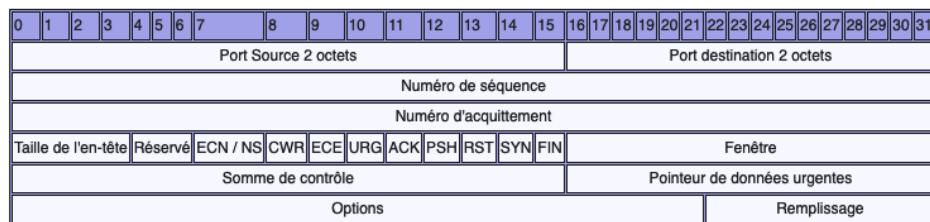
On désire envoyer par email une photo de 2 Mo. De combien de paquets au minimum aura-t-on besoin pour envoyer cette photo ?

Entête

De manière similaire au protocole IP, le protocole TCP est constitué d'un *entête* placé au début des données du paquet IP et qui contient des informations sur les numéros de morceaux envoyés et reçus. En effet, la machine réceptrice va envoyer une quittance (*acknowledgement* en anglais) pour chaque paquet reçu, de manière à ce que la machine émettrice puisse renvoyer un paquet qui n'aurait pas été acheminé à destination. Un paquet envoyé par les protocoles TCP et IP, contient donc l'entête IP, suivi de l'entête TCP, suivi des données, tel que représenté ci-dessous.

en-tête IP en-tête TCP données

L'entête TCP est constitué d'au moins 20 octets contenant les informations suivantes :



Comme le montre la figure ci-dessus, les quatre premiers octets contiennent les ports source et de destination. Un port est un peu comme une boîte aux lettres à l'intérieur d'un ordinateur. Les ports sont numérotés sur 16 bits, donc de 0 à $2^{16} - 1$. Un ordinateur qui est en connexion simultanée avec plusieurs ordinateurs pourra par exemple assigner un port différent à chaque connexion, ce qui lui permettra de ne pas mélanger les messages reçus de ses divers interlocuteurs. Dans ce contexte et contrairement à un port USB, un port n'a pas de réalité matérielle, il est réalisé de manière logicielle par le système d'exploitation.

Les quatre octets suivants contiennent le numéro de séquence qui va permettre au programme qui reçoit les paquets de les remettre dans l'ordre selon ce numéro. Le numéro d'acquittement sont utilisés par le destinataire pour indiquer quels sont les paquets qui ont été reçus, permettant ainsi à la machine émettrice de savoir quels paquets se sont perdus en chemin et doivent être envoyés à nouveau. Les quatre octets suivants contiennent divers éléments permettant aux deux machines en communication de se synchroniser, notamment divers fanions indiquant si on veut initier, ou terminer la connexion, ainsi que "Fenêtre" par lesquels le récepteur indique à l'émetteur combien de place il lui reste dans la pile des paquets à trier et traiter. Ceci permet à l'émetteur d'adapter le rythme auquel il envoie les paquets pour ne pas déborder le récepteur. Enfin la "Somme de contrôle" est un code correcteur d'erreur qui permet de vérifier si l'entête n'a pas été altéré en chemin.

Exercice 3

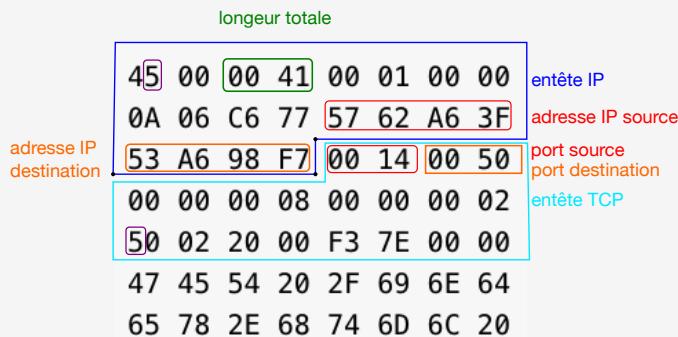
Un paquet a été intercepté sur Internet avec le contenu initial suivant indiqué en hexadécimal :

```
45 00 00 41 00 01 00 00
0A 06 C6 77 57 62 A6 3F
53 A6 98 F7 00 14 00 50
00 00 00 08 00 00 00 02
50 02 20 00 F3 7E 00 00
47 45 54 20 2F 69 6E 64
65 78 2E 68 74 6D 6C 20
```

- Déterminer quelle partie de cet entête correspond à l'entête IP, et laquelle correspond à l'entête TCP.
- Indiquer l'adresse IP et le port de l'émetteur de ce paquet.
- Indiquer l'adresse IP et le port du destinataire de ce paquet.
- Quelle est la longueur du paquet ?
- Combien d'octets du paquet ne sont pas représentés ci-dessus ?

Solution 3

- Le premier chiffre de l'entête étant un 4, c'est le format IPv4. La taille de l'entête IP est donc donnée par les bits 4 à 7, et donc le deuxième chiffre hexadécimal de l'entête qui est un 5 (en mauve). L'entête IP correspond donc aux 5 premiers mots de 32 bits, c'est-à-dire aux 20 premiers, octets donc aux 20 premiers nombres à 2 chiffre hexadécimaux (en bleu dans l'image ci-dessous). L'entête TCP suite directement et sa taille est donnée par le début du 13e octet, qui est un 5 (en mauve) dans notre exemple. L'entête TCP fait donc également $5 \cdot 4 = 20$ octets, en cyan. L'image ci-dessous indique comment interpréter cet entête.

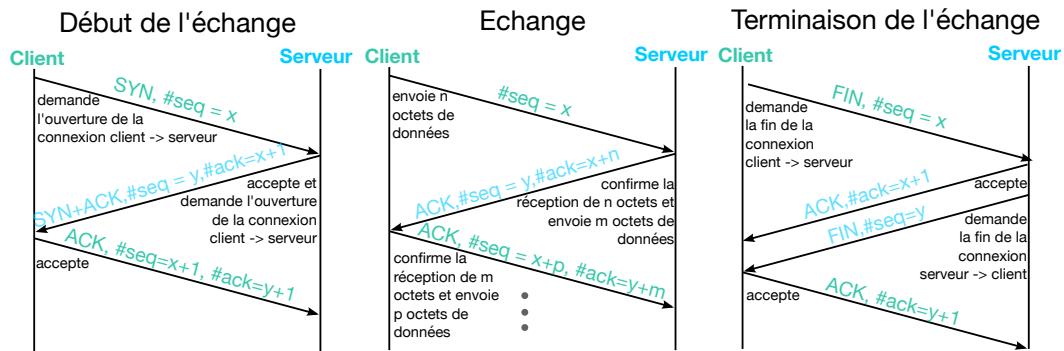


- L'adresse IP de la source en hexadécimal est 57 62 A6 3F ou 87.78.166.63 en notation usuelle. Le no de port est 0014 en hexadécimal, donc $1 \cdot 16 + 4 = 20$ en décimal.
- L'adresse IP du destinataire en hexadécimal est 53 A6 98 F7 ou 83.166.152.247 en notation usuelle. Le no de port est 0050 en hexadécimal, donc $5 \cdot 16 + 0 = 80$ en décimal.
- La longueur totale est de $41_{16} = 4 \cdot 16 + 1 = 65$ octets, y compris l'entête.
- Seuls 56 octets sont représentés ci-dessus, il manque donc $65 - 56 = 9$ octets.

Déroulement

Le protocole TCP applique une structure *client-serveur* à communication entre deux machines. Cela signifie qu'une des machines, appelées le serveur, va se mettre en mode d'écoute, et attendre que d'autres machines la contactent. C'est la machine cliente qui va prendre l'initiative d'initier la communication en envoyant un message TCP (juste l'entête, sans les données) à la machine serveur. Le protocole TCP spécifie les messages qui doivent être envoyés de part et d'autre pour initier la connexion, comment ensuite envoyer et quittancer les données échangées, et comment mettre fin et terminer la connexion une fois que tout a été envoyé et quittancé.

La figure ci-dessous indique comment les fanions SYN, FIN de l'entête TCP sont utilisés pour indiquer que l'on veut respectivement initier et terminer une connexion, et comment le fanion ACK est utilisé pour confirmer la bonne réception de la demande ou des données, avec les numéros des séquences (#seq) et et d'acquittement (#ack).



Exercice 4

Un serveur reçoit un paquet TCP avec le contenu suivant dans l'entête. Que cela signifie-t-il, et comment le serveur est-il censé réagir si tout se passe bien ?

1. FIN = 1, no de séquence = 257
2. SYN = 1, no de séquence = 745
3. ACK = 1, no de séquence = 343, no d'acquittement = 746,

Solution 4

1. Le client souhaite mettre fin à la connexion. Le serveur répond avec ACK=1, no d'acquittement=258
2. Le client souhaite établir une connection, le serveur répond avec SYN=1, ACK=1, no d'acquittement=746 et un no de séquence.
3. Le client a reçu les paquets jusqu'à l'octet 746 (non compris) et envoie un message numéroté 343. Le serveur envoie un paquet avec l'ACK=1, no d'acquittement 343+m. S'il y a p octet à envoyer le serveur l'inclut et un no de séquence 746+p.

Exercice 5

- Quelle est no de séquence maximal que l'entête TCP peut contenir ?
- Que peut-on faire si le nombre de paquets envoyés est tel que ce nombre est dépassé ?

5.2.4 Le protocole UDP

TCP n'est pas l'unique moyen de transmettre des messages par internet. Par exemple, si l'important est que les données soient transmises rapidement, même si certaines sont perdues en route, on peut utiliser le protocole UDP. Avec ce protocole, l'émetteur envoie des paquets au destinataire sans vérifier que ce dernier les reçoit, ou même qu'il est présent à l'adresse de destination. L'entête UDP ne contient que quatre champs de 2 octets chacun, déjà présent dans l'entête TCP.

2 octets	2 octets	2 octets	2 octets
port source	port destination	longueur totale	somme de contrôle

Il n'y a donc pas de numérotations des séquences, ni de système d'acquittement, ce qui fait que si un paquet est perdu ou s'arrive en retard par rapport aux autres paquets, on l'expéditeur et le destinataire n'ont aucun moyen de le savoir. Par contre, cela permet d'aller plus vite, donc ce protocole est surtout utilisé dans les applications en temps réel.

Exercice 6

Indiquer pour les applications suivantes, si le protocole TCP ou UDP était plus adapté, et indiquer pourquoi.

1. La lecture d'une page web
2. Une application de téléphonie par Internet
3. Le streaming d'une vidéo
4. Une application bancaire en ligne
5. Un jeu vidéo en ligne

Ai-je compris ? – L'exemple d'Alice

Pour entrer en communication avec le serveur web, le téléphone d'Alice va utiliser le protocole TCP. Le téléphone va donc créer un entête TCP dans lequel il indiquera (par l'utilisation du fanion SYN) qu'il souhaite établir une connection avec le site web. Devant cet entête il mettra également un entête IP dans lequel il indiquera (entre autres) les adresses IP du téléphone d'Alice (comme source) et du site web (comme destination). Ces deux entêtes formeront un paquet IP qui sera envoyé à travers le réseau jusqu'au serveur web qui répondra par un autre paquet, selon le protocole TCP. Une fois la connection établie, le navigateur web d'Alice pourra demander au serveur le contenu de la page web. Celle-ci sera découpée en petit morceaux qui seront numérotés et envoyés séparément au téléphone d'Alice qui enverra des acquittement pour les paquets reçus. Le serveur pourra ainsi renvoyer les paquets pour lesquel il n'a pas reçu d'acquittement.

5.3 Routage

Le **routage** est le mécanisme par lequel des chemins sont sélectionnés dans un réseau pour acheminer les données d'un expéditeur jusqu'à une destination.

Pour comprendre le routage, il faut distinguer deux types de machines qui font fonctionner Internet :

- le **routeur**, qui sert d'intermédiaire dans la transmission d'un message,
- l'**hôte** qui émettent ou reçoivent un message.

5.3.1 Les routeurs

Les *routeurs* sont des ordinateurs spécialisés dont le rôle est de relayer et d'orienter correctement les informations qui circulent sur Internet. Si Internet est représenté par un graphe dont les arêtes représentent les canaux de communication, alors les routeurs sont situés aux noeuds du graphe et décident dans quelle direction faire suivre une information afin qu'elle atteigne son destinataire. Les routeurs sont donc comme des facteurs disposés aux intersections du réseau Internet qui vont lire la destination des messages qui leur arrivent et les rediriger vers la prochaine intersection de manière à les rapprocher de leur destination. Les hôtes sont généralement aux l'extrémités du graphe.

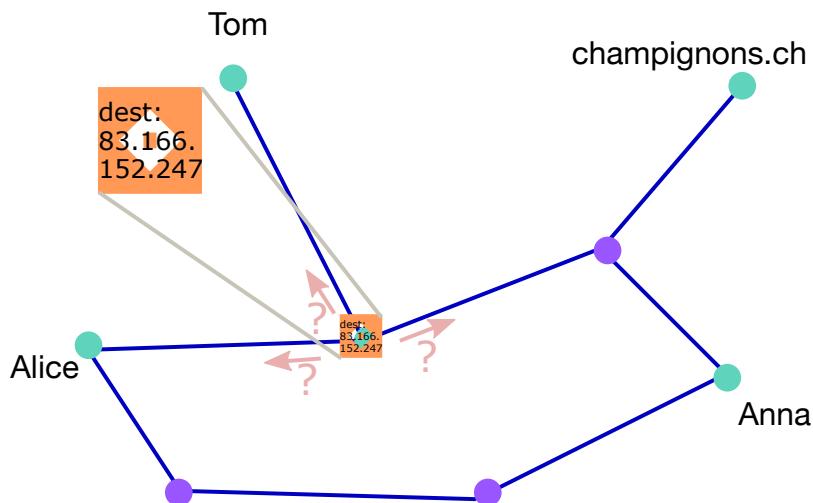


FIG. 5.6 – Un routeur regarde le destinataire de chaque paquet qu'il reçoit et le redirige dans la bonne direction vers le prochain routeur ou le destinataire. Dans notre exemple, le paquet de données (représenté par le carré orange) qu'Alice veut envoyer au serveur champignons.ch transite par différents routeurs (représenté en violet), qui décident par où faire transiter le message en fonction de l'adresse de destination du paquet.

Pour ceci, les routeurs s'aident de *tables de routage* qui leur indique la direction à suivre pour chaque destination.

5.3.2 Les tables de routage

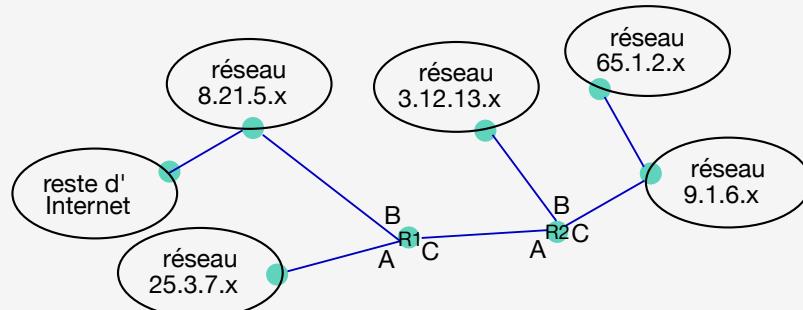
Une table de routage est un tableau qui indique dans quelle direction orienter un message en fonction de son destinataire. Conceptuellement, on peut imaginer une table de routage comme un tableau à deux colonnes, la première colonne contenant l'adresse IP de destination (ou un sous réseau à laquelle elle appartient), la seconde colonne indiquent *l'interface* à laquelle il faut envoyer les messages destinés à cette adresse. L'interface représente la “direction” dans laquelle envoyer le message (par exemple un port Ethernet, un câble en fibre optique, un émetteur wifi). Ainsi lorsqu'un nouveau message atteindra le routeur, celui-ci regardera dans sa table de routage la ligne contenant le sous-réseau le plus spécifique incluant l'adresse IP de destination et le fera suivre dans l'interface correspondante (qui est elle-même connectée soit à un autre routeur soit au destinataire).

Destinataire	Interface
127.1.1.1	A
34.234.15.x	B
87.45.x.x	C
87.33.x.x	C
X.X.X.X	D

La dernière ligne représente la *passerelle par défaut* qui indique où envoyer les messages dont l'adresse ne correspond à aucune autre ligne de la table.

Exercice 1

Remplir les tables de routage simplifiées des routeurs R1 et R2 du réseau suivant dans lequel les interfaces sont représentées par les lettres A,B, et C.



Les tables de routage contiennent souvent des informations, c'est-à-dire des colonnes, supplémentaires. Elle peuvent par exemple contenir une colonne “Distance” qui indique le nombre de d'étapes avant d'arriver à destination. Les voisins directs sont ainsi une distance de 1, alors que les voisins des voisins ont une distance de 2, etc. D'autres informations peuvent figurer comme le coût de transmission d'un paquet, ou le traffic maximal que cette route peut supporter.

Exercice 2

Ajouter une colonne “Distance” à la table de routage de l'exercice précédent.

Pour aller plus loin – Masques de réseau

Pour qu'une machine sache si une autre machine est dans le même sous-réseau qu'elle, son sous-réseau est spécifié par un *masque* de réseau composé d'une suite de 32 bits (en IPv4) dont les n premiers sont à 1 et les $32 - n$ suivants sont à 0. Par exemple, une machine peut avoir une adresse IP 128.178.23.132 avec un masque de 11111111.11111111.11111111.00000000. Cela signifie que toutes les machines qui ont la même adresse IP là où le masque vaut 1 sont dans le même sous-réseau. Dans notre exemple, cela correspond à toutes les adresses IP 128.178.23.x. Pour gagner de la place, les masques sont aussi exprimés en 4 nombres décimaux, dans notre exemple 255.255.255.0, ou alors, pour faire encore plus court, on peut simplement spécifier le nombre de 1 du masque, ce qui donne, toujours pour le même exemple, 128.178.23.132/24.

Ainsi toutes les adresses IP qui n'a pas les mêmes n premiers bits, fait partie d'un différent sous-réseau. Pour lui envoyer des paquets, il faudra passer par la *passerelle par défaut* (*default gateway* en anglais) qui est le routeur qui s'occupe de communiquer avec l'extérieur du sous-réseau.

Dans des petits réseaux locaux, cette table de routage peut être construite manuellement, mais généralement c'est le routeur qui construit sa propre table de routage en interaction avec les routeurs voisins.

5.3.3 Le routage dynamique

Dans la pratique, le réseau de connections qui constituent Internet change et évolue constamment : de nouvelles machines se connectent au réseau, changent d'adresse IP, des routeurs tombent en panne, certaines connexions s'ajoutent ou disparaissent, par exemple en cas de dommages aux câbles. Cela ne serait pas gérable pour des humains de constamment mettre à jour les tables de routage pour les adapter à la configuration du réseau. C'est pourquoi un système automatisé de mise à jour des tables de routage est utilisé. C'est ce qu'on appelle le *routage dynamique*. Cela permet non seulement de gérer les changements de configuration du réseau, mais également les phénomènes de congestion du trafic.

Le protocole RIP

Le protocole RIP (Routing Information Protocol) est une des manières les plus anciennes et les plus simples de faire du routage dynamique. Toutes les 30 secondes, chaque routeur envoie à tous ses voisins le contenu de sa table de routage. Lorsqu'un routeur reçoit une ligne de la table de routage de son voisin dont la destination n'est pas incluse dans sa propre table, il l'ajoute à sa table en indiquant comme interface, celle le connectant avec ce voisin. De plus lorsqu'un routeur que son voisin dispose d'un chemin plus court pour atteindre une destination, reçoit une ligne de la table de routage de son voisin dont la destination est incluse dans sa table, il modifie sa table de routage pour faire passer par ce voisin les messages pour cette destination.

Exercice 3

La table de routage d'un routeur 1 contient les lignes suivantes :

Destinataire	Interface	Distance
114.2.1.1	A	1
12.251.x.x	B	2
12.25.x.x	C	1
87.33.x.x	C	8
...

Ce routeur reçoit de son voisin, le routeur 2 sur l'interface B une table contenant les lignes suivantes (les interfaces ne sont pas indiquées) :

Destinataire	Interface	Distance
12.251.x.x	-	1
12.252.x.x	-	3
87.33.x.x	-	5
...

Comment le routeur 1 peut-il compléter sa table de routage avec les informations reçue par son voisin ?

Solution 3

Il peut ajouter la ligne suivante :

Destinataire	Interface	Distance
12.252.x.x	B	4

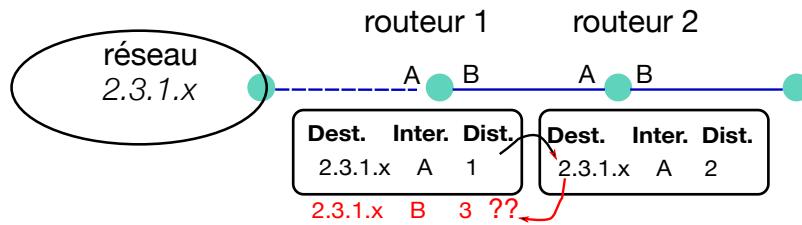
En effet, c'est une nouvelle destination que le routeur 1 peut maintenant joindre en transmettant les paquets au routeur 2 (sur l'interface B) qui saura les transmettre dans la bonne direction.

Le routeur 1 peut aussi modifier la ligne pour la destination 87.33.x.x en mettant

Destinataire	Interface	Distance
87.33.x.x	B	6

Cette destination déjà existante pour le routeur 1, mais elle peut être transmise plus rapidement en passant par le routeur 2 sur l'interface B (en $5+1=6$ étapes) que par l'interface C (en 8 étapes)

Toutefois, si l'on applique cette méthode telle quelle, cela peut créer des situations où des erreurs dans les tables de routage se propagent à travers le réseau. Considérons par exemple le bout de réseau suivant :



Lorsque la connexion en traitillés fonctionne, le routeur 1 remplit sa table de routage et la transmet au routeur 2. Les tables de routage de ces routeurs contiennent donc les lignes indiquées en noir dans la figure. Si la connexion en traitillé se rompt, le routeur 2, va effacer de sa table de routage la ligne concernant la destination 2.3.1.x, mais il serait tenté d'ajouter la ligne en rouge lorsqu'il recevra la table du routeur 2, ce qui serait erroné.

C'est pour éviter ces problèmes que la protocole RIP applique un certains nombre de principes, appliqués également par d'autres protocoles de routage.

1. Ne pas transmettre à une interface une information déjà reçue par cette interface. Ainsi, dans l'exemple ci-dessus, selon ce principe, le routeur 2 ne pourra pas transmettre sa ligne au routeur 1, car cette information lui vient du routeur 1. C'est le principe de l'*horizon séparé* (*split horizon*).
2. Si une route est bouchée, transmettre cette information aux voisins. Dans le protocole RIP on indique ceci par une distance égale à 16. Toute destination à distance supérieure à 15 est considérée comme inaccessible. Dans l'exemple ci-dessus, le routeur 1 remplacera simplement la distance par 16 (au lieu de 1) et transmettra cette information au routeur 2 qui mettra à jour sa table de routage. C'est le principe de l'*empoisonnement de route* (*route poisoning*).

Ai-je compris ? – L'exemple d'Alice

Les paquets IP échangés entre le téléphone d'Alice et le serveur sont acheminé de routeur en routeur. Chaque routeur consulte sa table de routage pour savoir dans quelle direction transférer le paquet reçu. Ces tables de routage se constituent automatiquement en échangeant des informations avec les routeurs voisins.

5.4 World Wide Web

Le **World Wide Web** (WWW), littéralement la « toile (d'araignée) mondiale », est un système qui permet de consulter avec un navigateur, à travers l'Internet, des pages accessibles sur des sites.

5.4.1 Historique

Pendant ses premières décennies (jusque dans les années 90), seuls les universitaires, les militaires, certaines entreprises et une communauté d'enthousiastes (largement issue du mouvement hippie) utilisaient Internet. Les utilisations principales étaient la discussion écrite (le *chat* dans un terminal), la connexion sur un ordinateur à distance, l'email et le transfert de fichiers entre ordinateurs.

Pour aller chercher un fichier se trouvant sur un autre ordinateur, il fallait savoir exactement sur quelle machine celui-ci se trouvait et où il se situait dans cette machine. Il fallait donc établir des listes de ressources et de leur location dont la maintenance et l'utilisation étaient fastidieuses.

C'est en voulant résoudre ce problème que Tim Berners-Lee, un scientifique anglais du Conseil Européen de la Recherche Nucléaire (CERN) à Genève, a développé les technologies du Web entre 1989 et 1991. Celles-ci se sont rapidement développées après que le CERN les ait gratuitement mises à disposition du public. Des centres de recherche, universités, entreprises (d'informatique et de média) et d'autres organisations ont créé leur site web afin de pouvoir facilement diffuser des informations par ce canal. Ceci offrait un usage supplémentaire de l'ordinateur dont les foyers se sont équipés massivement, diffusant ainsi l'accès à Internet au sein de la population américaine et européenne.

5.4.2 Les technologies du Web

Le web repose sur trois technologies mises ensemble et qui permettent de naviguer dans une “toile” de documents. La première, l'URL, spécifie un format permettant de spécifier la localisation d'un document. La seconde, le protocole HTTP, permet de demander et de recevoir un document identifié par son URL. La troisième, le langage HTML, permet de décrire le contenu d'un document (une page web) pouvant contenir des liens vers d'autres documents spécifiés par leur URL.

Ces trois technologies sont rassemblées dans un *navigateur web*, un programme qui permet de

1. spécifier une page web à visiter en indiquant son URL, typiquement dans une barre de navigation
2. demander la page web au serveur correspondant et la réceptionner en utilisant le protocole HTTP
3. afficher le contenu de la page web (décrise au format HTML), y compris les liens cliquables permettant d'afficher d'autres pages web.

Document historique

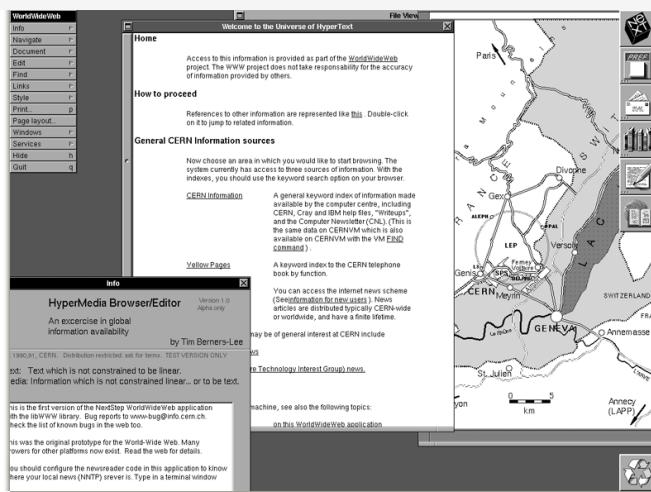


FIG. 5.7 – Un des premiers navigateurs web développé par Tim Berners-Lee. Les images s'affichaient sur des fenêtres séparées.

URL

L'URL (*Uniform Resource Locator*) est une manière de spécifier la localisation d'un document disponible sur Internet. Un exemple d'URL peut être par exemple “<https://www.champignons.ch/fichiers/fr/contact.html>”.

Une URL comporte trop parties, qui sont les suivantes dans notre exemple [<https://www.champignons.ch/fichiers/fr/contact.html>]. Autrement dit, une URL se compose généralement de la manière suivante :

protocol :hôte/chemin

1. Le *protocole*, dans notre exemple **https**, indique le protocole utilisé pour avoir accès à la ressource. Pour le web, ce protocole est toujours **http** ou **https**, sa version sécurisée. Mais l'URL étant aussi utilisée en dehors du web, il y a d'autres protocoles possibles, par exemple **ftp** pour faire du transfert de fichier.
2. L'*hôte* spécifie la machine (ou le serveur) où aller chercher le fichier. Cela peut être un nom de domaine, mais également une adresse IP
3. Le *chemin* indique quel fichier on souhaite obtenir de la part du serveur. On part de la racine “/” (connue du serveur) et on descend dans l'arborescence selon les répertoires indiqués. Par exemple **/fichiers/fr/contact.html** est le fichier **contact.html** qui se trouve dans le répertoire **fr** qui se trouve lui-même dans le répertoire **fichiers**.

Dans le protocole HTTP, si le chemin est un répertoire (et pas un fichier), le fichier par défaut **index.html** présent dans ce répertoire est envoyé par le serveur.

HTTP

HTTP (acronyme de HyperText Transfer Protocol) est le protocole qui régit la manière dont un client web (par exemple le navigateur web de Alice) et un serveur web (par exemple le site www.champignons.ch) vont interagir l'un avec l'autre.

Par exemple si le client demande au serveur de lui envoyer la page web `accueil.html`, il lui enverra requête GET suivante :

```
GET accueil.html HTTP/1.1
```

ce qui signifie “envoie-moi la page `accueil.html` avec la version 1.1 du protocole HTTP”. S'il trouve la page en question, le serveur pourra alors envoyer la réponse suivante :

`HTTP/1.1 200 OK` suivie de diverses informations ainsi que la page web demandée. Le code `200 OK` indique que la requête peut être honorée.

Si la page `accueil.html` n'existe pas, alors le serveur pourra l'indiquer au client en envoyant la réponse suivante :

```
HTTP/1.1 404 Not Found
```

Le navigateur web pourra alors afficher l’“erreur 404” au pour l'utilisateur.



FIG. 5.8 – Le serveur retourne un message d’erreur s’affiche lorsqu’on demande une page qui n’existe pas.

Il y a d’autres sortes de requêtes que le client peut envoyer au serveur, par exemple POST pour envoyer une information du client au serveur, utilisé par exemple lorsqu’on remplit un formulaire en ligne.

Si un utilisateur utilise le protocole HTTP pour surfer sur le web, une tierce personne qui a accès au trafic sur Internet peut savoir quelle page web a été visitée par cet utilisateur et ce que le serveur lui a envoyé comme information (par exemple des messages privés). C'est pourquoi on utilise généralement plutôt le protocole HTTPS qui encrypte les requêtes et réponses HTTP. Ainsi un observateur peut toujours savoir avec quel site on communique lorsqu'on surfe sur le web en regardant l'entête IP, mais ne pourra pas connaître les détails des pages demandées et transmises.

HTML

HTML (HyperText Markup Language) est un langage de description des pages web. Il permet de spécifier le contenu et l'apparence d'une page web afin que le navigateur web puisse l'afficher. Supposons par exemple que le site www.champignons.ch envoie à Alice une page web contenant le nom d'un champignon ainsi qu'une photo de celui-ci. Une manière de décrire cette page avec le langage HTML serait la suivante :

```
<html>
<body>
  <h1 style="color:red"> L'amanite tue-mouche </h1>
  <p> L'amanite tue-mouche est très belle mais très dangereuse ! </p>
  
</body>
</html>
```

Les éléments de cette page sont indiqués par des *balises* indiquées par des crochets pointus (`<>`) et peuvent être imbriqués les uns dans les autres. Ainsi la page (entre `<body>` et `</body>`) contient un titre (entre `<h1>` et `</h1>`) de couleur rouge, un paragraphe de texte (entre `<p>` et `</p>`) ainsi qu'une image (``) disponible dans le fichier `photo.jpg` et de hauteur 250 pixels. Cette page pourra ainsi être affichée de la manière suivante dans le navigateur web.

L'amanite tue-mouche

L'amanite tue-mouche est très belle mais très dangereuse !



La plupart des navigateurs web permettent de visualiser le *code HTML* des pages visitées. Un aspect important de la création de sites web consiste à écrire du code HTML qui sera mis sur le serveur pour être transmis au visiteur du site web. Cela peut se faire en écrivant directement du code HTML dans un fichier texte, ou à l'aide d'un outil de création de sites web qui se charge d'écrire le code HTML selon les indications données par la personne concevant le site.

5.4.3 Les évolutions du Web

Javascript

Dans la version originale d'HTML, les moyens d'interagir avec une page web étaient très limités, par exemple cliquer sur les liens que la page proposait. Les personnes utilisant et développant le web ont vite voulu enrichir l'interactivité. C'est pourquoi, en 1995, les développeurs de Netscape, le navigateur web populaire de l'époque, ont ajouté la possibilité d'intégrer des programmes dans les pages web. Ils ont pour ceci inventé un langage de programmation, javascript, qui puisse être interprété et exécuté par le

navigateur web. Cela permettait d'avoir une page web avec du contenu dynamique qui réagisse aux actions des personnes utilisatrices, par exemple pour changer la langue du texte lorsque on appuie sur un petit drapeau. Cela permet aussi de programmer des animations sur une page web.

Le Web dynamique

Au début, les pages web étaient des fichiers HTML stockés sur les serveurs. C'est ce qu'on appelle le web *statique*. Si les sites web statiques existent toujours, par exemple modulo-info.ch, beaucoup de sites web sont dynamiques, c'est-à-dire que le fichier HTML est généré par le site au moment où la requête est faite. Cela permet de servir une page différente selon l'utilisateur ou selon les arguments de la requête qui sont des indications supplémentaires ajoutée à la requête après l'URL après le signe ?.

Micro-activité

Effectuer une recherche sur un navigateur web et consulter la barre de navigation. Quels sont les arguments de votre requête et pouvez-vous en comprendre la signification ?

Exercice 1

Parmi les sites web suivants, lesquels ont besoin d'être dynamiques et lesquels peuvent se contenter de fournir un contenu statique ?

1. Un site d'achats en ligne
2. Un site indiquant les horaires d'ouverture et de fermeture d'un magasin.
3. Un site de consultation du catalogue d'une bibliothèque
4. Un site de présentation d'une entreprise
5. Un site avec les documents d'un cours universitaire ou scolaire
6. Un site d'e-banking

Pour les sites qui peuvent être statiques, quelles possibilités supplémentaires pourraient être offertes par un site dynamique ?

Le Web 2.0

Le web 2.0 fait référence à la tendance, initiée au début des années 2000, de proposer des pages web permettant aux internautes de contribuer du contenu, et pas uniquement de lire des fichiers comme c'était le cas jusqu'alors. Les blogs, forums, wikis, et les réseaux sociaux font partie de ce développement qui voit exploser l'aspect participatif du web. Ce n'est en effet plus nécessaire de connaître la syntaxe HTML et d'avoir son propre serveur pour mettre du contenu à disposition des internautes de la planète. Si les développements techniques qui ont permis le web 2.0, étaient peu importants, ils ont eu un effet considérable sur la manière dont la population s'est approprié le web pour en faire un espace d'expression et de partage duquel ont émergé des réseaux sociaux d'aujourd'hui.

Ai-je compris ? – L'exemple d'Alice

Pour demander la page web du site *champigons.ch* le navigateur web d'Alice va utiliser le protocole HTTP (ou HTTPS) qui spécifie des mots de code indiquant qu'on veut accéder à une page donnée ou envoyer des informations au serveur. Le serveur va répondre avec le même protocole pour envoyer la page ou un message d'erreur si ce n'est pas possible. La page, elle-même, est envoyée au format HTML, c'est à dire sous forme de texte et d'images. Le navigateur va lire ce texte qui va lui indiquer ce qu'il faut afficher pour Alice.

5.5 Interopérabilité

Si Internet a connu un développement aussi remarquable, c'est aussi grâce à certains choix techniques et de gouvernance qui ont permis de le rendre accessible relativement facilement. Des personnes, organisations et entreprises pouvaient ainsi participer à sa construction et son développement.

Certains de ces choix sont décrits ci-dessous.

5.5.1 Un modèle en couches

La communication sur Internet se fait selon une pile de protocoles qui s'ajoutent les uns aux autres tout en étant indépendant les uns par rapport aux autres selon un *modèle en couches*. Ainsi, le protocole HTTP est responsable de l'échange de pages web, mais toute la partie s'assurant du bon transfert et de la bonne réception des paquets est gérée par le protocole TCP. Mais ce protocole repose sur le protocole IP pour l'envoi des paquets individuels qui lui-même repose sur différents protocoles selon que les paquets circulent par le wifi, un câble sous-marin, la 4G ou de la fibre optique. Ainsi les niveaux supérieurs peuvent s'abstraire des niveaux inférieurs, et vice-versa. Si un nouveau support physique de communication est inventé (par exemple la téléportation quantique), il suffit de développer un protocole de communication propre à ce support et on pourra utiliser le protocole IP pour la transmission de paquets, ce qui permettra à ce nouveau support de s'intégrer sans difficulté à Internet.

On a ainsi défini le modèle de la Fig. 5.9 en 4 "couches" de protocoles : la première couche "liaison réseau" définit comment les données sont transmises entre deux appareils directement connectés ou du même réseau local. Le protocole en question dépend donc du type de connexion entre les deux appareils (wifi, câble électrique, fibre optique, etc.). La deuxième couche, "Internet", définit comment les données sont transmises entre deux machines du réseau, c'est le protocole IP vu précédemment. La troisième couche "transport" définit comment les données sont segmentées (c'est-à-dire découpée et morceaux) et envoyées par l'émetteur et reconstituées et quittancées par le récepteur, c'est le protocole TCP également vu précédemment. La quatrième couche est la couche applicative qui définit comment deux applications (ou programmes) communiquent entre elles, par exemple le protocole HTTP qui détermine la communication entre un navigateur web et un serveur web. D'autres exemples figurant dans cette couche pourraient inclure la manière dont l'application TikTok d'un smartphone communique avec le serveur de TikTok.

Ainsi, lorsque le navigateur web d'Alice demande une page au serveur web, ces deux applications (le navigateur et le serveur) sont en communication en utilisant le protocole HTML. Pour transmettre la requête HTML d'Alice, une connexion entre Alice et le serveur web sera établie en utilisant le protocole TCP. Au besoin, ce protocole découpera la requête ou la page web en petits morceaux et ajoutera les entêtes TCP à chaque morceau, qui sera envoyé individuellement en utilisant le protocole IP (en ajoutant y donc l'entête IP). Selon le type de connexion, (4G, wifi, cable), les paquets IP seront transmis selon différents protocoles à des routeurs qui les achemineront jusqu'au destinataire qui rassemblera les paquets selon le protocole TCP et fournira la requête HTML d'Alice au serveur web ou la page web demandés au navigateur d'Alice.

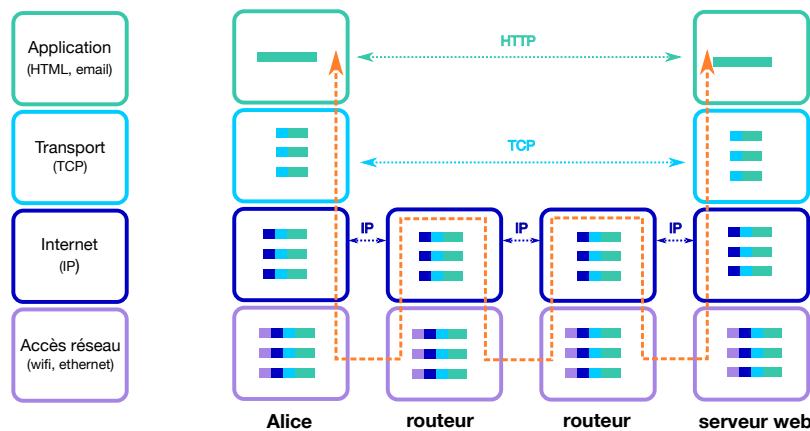


FIG. 5.9 – Gauche : le modèle en 4 couches de la communication par Internet. Droite : Exemple de l’application de ce modèle à la communication entre le navigateur web d’Alice et un serveur web. Lorsque les applications veulent s’échanger de l’information, cet échange se fait selon un protocole de la couche “Application” par exemple HTTP. La requête ou la page web ainsi formées (représentées par le rectangle vert) sont ensuite traitées par le protocole TCP de la couche “Transport” qui va découper ces données en petits paquets, y ajouter une entête TCP numérotée (en bleu ciel) et envoyés à leur destinataire par le protocole IP de la couche “Internet”. Avec ce protocole, une seconde entête (représentées en bleu foncé) est ajoutée à chaque paquet qui est envoyé sur le réseau par un protocole propre au type de réseau utilisé (en y ajoutant encore une entête mauve). L’information ne passera d’habitude pas directement d’Alice au serveur web, mais par des routeurs qui achemineront les paquets IP jusqu’à leur destination, en utilisant les différents protocoles de la couche d’accès réseau selon les besoins.

Ce modèle en 4 couches a été ensuite développé en un modèle en 7 couches appelé OSI (pour *Open System Interconnection*). Dans ce modèle, la couche d'accès réseau est séparée en deux couches, la couche physique qui décrit comment le signal est codé dans un médium donné (fibre optique, onde électromagnétique, etc.) et la couche de liaison qui indique comment un groupe de bits (appelé une trame) est envoyé au sein un réseau local (par exemple le wifi, ou un réseau ethernet). Ce modèle ajoute deux couches à la couche “Application”, la couche de présentation qui spécifie comment les données sont encodées (par exemple avec la table ASCII) et potentiellement encryptées, et la couche “Session” qui gère les questions d'authentification et d'autorisation, par exemple lorsque vous vous connectez à un service payant sur le web.

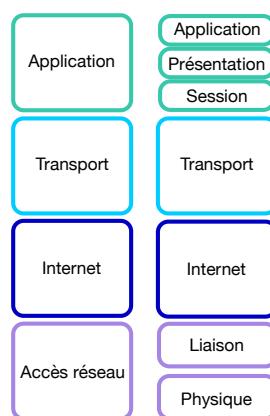


FIG. 5.10 – Le modèle (TCP-IP) en quatre couches et, à sa droite, le modèle OSI en sept couches. La couche “Application” a été séparée en trois couches, et la couche “Accès réseau” a été séparée en deux couches.

Le modèle OSI indique la manière dont les choses devraient se passer pour garantir à la fois la sécurité et la modularité des communications, mais il n'est dans la pratique pas toujours entièrement respecté, en particulier pour les couches supérieures.

5.5.2 Des protocoles ouverts

Les protocoles décrits ci-dessus ont été établis sur proposition de différentes personnes travaillant principalement dans les universités ou les entreprises de télécommunication et adoptés par consensus après beaucoup de discussion. L'idée principale étant qu'Internet n'appartient à personne et qu'il s'agit d'une oeuvre collective à laquelle toute personne dotée des compétences nécessaires peut contribuer. Ces protocoles sont *ouverts* dans le sens que chacun peut y avoir accès et les implémenter à sa manière. Par exemple, quelqu'un qui souhaiterait développer un routeur peut avoir accès à toutes les informations nécessaires pour le faire.

La collaboration autour de la définition des protocoles d'Internet est structurée autour l'Internet Society, une association à but non lucratif dont le but est le développement d'Internet. L'Internet Engineering Task Force et un groupe de personnes qui discutent des aspects techniques d'Internet. Ce groupe est (théoriquement) ouvert à toute personne qui souhaite s'y impliquer. Les discussions s'articulent autour de "Requests For Comments" (RFC) qui sont des documents publics qui proposent des idées qui sont discutées, et, pour certaines d'entre elles, adoptées par consensus. Le premier RFC, RFC1 a été formulé en 1969 pour proposer un protocole de communication sur ARPANET, le projet de recherche militaire américain qui a donné naissance à Internet. Toutes les technologies d'Internet décrites ci-dessus ont été proposées par le biais de RFC, par exemple IPv4, RIP, HTML, etc.

Micro-activité

Chercher et lire le RFC 8962, en particulier l'abstract et les parties 7 et 8, ainsi que la date. De quoi s'agit-il ?

5.5.3 La neutralité d'Internet

Un des principes fondateurs d'internet est sa *neutralité*. Cela signifie que les paquets IP sont acheminés vers leur destination sans discrimination de source, de destination ou de contenu. Contrairement à la poste suisse, où certains courriers (par exemple le courrier A) sont prioritaires par rapport à d'autres, les paquets IP sont tous logés à la même enseigne sur Internet. Cela permet d'éviter que certains services (par exemple un site web) puissent payer plus cher pour que ses paquets arrivent plus rapidement chez leurs destinataires et offrir ainsi un service plus rapide au détriment d'autres services. Certains acteurs, tels que les fournisseurs d'accès à Internet se sont opposés à la neutralité du net, car cela leur aurait permis de mettre leurs clients en concurrence sur les débits fournis et ainsi augmenter leurs tarifs et donc leurs bénéfices. Ou alors, il leur serait possible de privilégier l'acheminement des paquets liés à leurs propres services (par exemple Swisscom, pourrait privilégier l'acheminement des paquets liés à son service de télévision au détriment d'autres chaînes.)

Le respect de la neutralité d'Internet est différent de pays en pays, certains, comme la Suisse l'ayant inscrite dans la loi.

Micro-activité

Lire l'article 12e⁶⁵ de la loi fédérale sur les télécommunications qui concerne la neutralité d'Internet. Quel alinéa garantit la neutralité du réseau ? Cette garantie est-elle absolue ?

65. https://www.fedlex.admin.ch/eli/cc/1997/2187_2187_2187/fr#art_12_e

5.5.4 L'universalité d'Internet en question

Avec la montée en puissance des entreprises privées, une partie de l'ouverture qui caractérisait les débuts d'Internet est remise en question. Ainsi, les entreprises qui ont développé les réseaux sociaux l'ont fait en utilisant des protocoles fermés (ou privés). Par exemple, le protocole par lequel l'application Whatsapp communique avec le serveur est gardé secret. Ceci permet d'empêcher que d'autres personnes ne développent des applications compatibles avec Whatsapp et y fassent ainsi concurrence. Pour illustrer la différence avec un protocole ouvert (ou public), on peut comparer Whatsapp (protocole fermé) avec l'email qui repose sur un protocole ouvert (SMTP). Le fait que le protocole de l'email soit public permet à toute personne qui en a les capacités d'offrir un service d'email. (Certaines personnes installent ainsi leur propre serveur email hébergé sur leur ordinateur.) Tous ces services d'email différents, qui peuvent être commerciaux, privés, ou artisanaux, sont compatibles les uns avec les autres, car ils suivent le même protocole SMTP. Ceci est très différent de Whatsapp qui ne peut être utilisé qu'avec l'application Whatsapp et donc tous les messages Whatsapp sont centralisés chez une seule entreprise. On observe certaines tentatives de créer des réseaux sociaux sur des protocoles ouverts, par exemple Mastodon pour le microblogging, PeerTube pour la vidéo, ou diaspora, mais leur succès reste limité, notamment car elles n'ont pas les ressources financières qui leur permettraient de rivaliser avec leurs concurrentes à visée lucrative.

Une autre tendance qui remet en question la décentralité d'Internet est le développement du cloud. Avec les services de cloud, les documents, les données et les sites web se concentrent dans les serveurs des entreprises offrant ces services. Ainsi, si une panne affecte un de ces services offerts par Google ou Microsoft, ou si la sécurité d'un tel service est compromise, les répercussions seront globales.

Enfin, si Internet donnait à ses débuts une impression d'universalité, on s'est rendu compte que l'utilisation des caractères ASCII, la syntaxe de HTML et de l'URL étaient peu propices aux alphabets non latins, et qui ne s'écrivent pas de gauche à droite. Tout choix "technique" est ancré dans un contexte social et culturel duquel il est difficile de faire abstraction. Des initiatives pour rendre Internet plus universel ont été prises, comme le fait de pouvoir entrer des URL en caractères chinois. Il n'en reste pas moins que les structures qui gèrent Internet restent des entités de droit américain, et que l'occident garde une place prépondérante dans le façonnage d'Internet. Certains états plus autoritaires, tels que la Chine ou la Russie, souhaitent avoir un contrôle plus strict de ce que leur population font sur Internet et tentent de filtrer certains contenus. À terme, il n'est pas exclu que se développent plusieurs réseaux en parallèle avec des politiques d'ouverture très différentes.

5.6 Conclusion

En quelques décennies, Internet est passé d'un moyen de communication et d'échange utilisés par des communautés relativement restreintes issues du monde académique, des mouvements hippies, et autres enthousiastes à une infrastructure essentielle de notre organisation sociale et économique. Cette évolution, rendue possible par le caractère ouvert et non hiérarchique d'Internet, a fait émerger de nouveaux acteurs économiques, politiques, ou criminels qui tirent profit de la dépendance de nos sociétés à cette infrastructure. Ainsi, conçu initialement comme un outil de résilience, Internet et peut-être en train de devenir également un point de vulnérabilité de nos modes de vie. Avec la multiplication des cyberattaques, les tentatives de manipulation de masse à travers les réseaux sociaux, les principes d'ouverture et de bonne foi qui ont guidé le développement initial d'Internet sont mis à mal. La notion de cybersécurité, prend une importance grandissante, que ce soit contre le vol de données, le vol d'identité, les logiciels de rançons, ou les attaques de serveurs.

Dans l'Union Européenne, le Règlement général sur la protection des données (RGPD), entré en vigueur en 2018, définit ainsi toute une série de principes que les entreprises et organismes doivent respecter afin de garantir la protection des données personnelles. Ces principes incluent par exemple le droit de savoir qui détient quelles données sur nous, le droit d'obtenir la correction des données erronées ou leur effacement, l'obligation pour les sites web d'obtenir le consentement des usagers concernant l'utilisation de leurs données des cookies, ainsi que l'obligation d'informer les autorités et les personnes touchées en cas de fuites de données, par exemple lors d'une cyberattaque.

Micro-activité – Le Règlement général sur la protection des données (RGPD)

Aller sur le site <https://www.cnil.fr/fr/reglement-europeen-protection-donnees/> et regarder le sommaire.

1. Combien d'articles ce règlement contient-il ?
2. Lire l'article 5 et résumer les grands principes de ce règlement.

Ai-je compris ?

1. Je sais comment est structuré Internet.
2. Je sais ce qu'est une adresse IP et un nom de domaine et à quoi cela sert.
3. Je sais ce qu'est un protocole et à quoi cela peut servir.
4. Je sais à quoi sert un entête IP et le protocole TCP.
5. Je sais à quoi sert un routeur.
6. Je sais à quoi sert une table de routage et quelle information elle contient.
7. Je sais comment un routeur peut dynamiquement une table de routage.
8. Je comprends la différence entre Internet et le Web.
9. Je connais les trois technologies du Web et comment un navigateur web les utilise.
10. Je comprends le principe des couches de protocoles.
11. Je comprends la différence entre un protocole ouvert (public) et fermé (privé) et quels en sont les enjeux pour Internet

CHAPITRE 6

Histoire de l'informatique

6.0 Histoire de l'informatique

Attention

Ce document est en cours de rédaction.

Si l'histoire de l'informatique prend ses racines loin dans notre histoire, cette science, encore très jeune, trouve sa place au milieu du XXe siècle. L'éclosion de l'informatique se situe en 1940, à la croisée d'opportunités technologiques, d'enjeux stratégiques et de contributions d'autres branches. C'est ainsi au cœur des conflits de la Deuxième guerre mondiale que l'importance de calculateurs programmables se fait sentir.

Le terme “informatik” apparaît en 1957 pour le traitement automatique de l'information dans un article de Karl Steinbuch. Il est officialisé en Français en 1962 par Philippe Dreyfus. En Anglais, pour des questions de droits des marques, le terme restera *computer science*.



6.0.1 Préinformatique

Les premiers dispositifs de calcul utilisent des petits cailloux (*calculus* en latin). Cet artifice était également utilisé pour compter les bêtes d'un troupeau par les bergers.

Les machines mécaniques



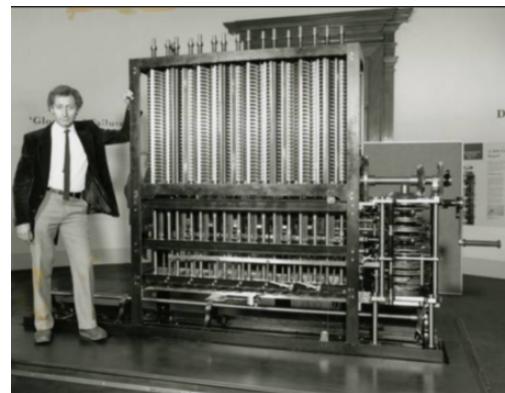
La machine d'Anticythère, plus de 100 ans av. J.-C., propose une représentation mécanique des mouvements astronomiques. En 1642, Blaise Pascal conçoit la Pascaline, une machine à calculer mécanique qui se base cette fois sur une représentation arithmétique traduite dans des rouages mécaniques. Au tout début du XIXe siècle, la machine de Joseph Jacquard dispose d'un mécanisme programmable basé sur des aiguilles et des cartes perforées. Enfin, cette histoire des machines mécaniques se termine en 1948 avec la calculatrice Curta, probablement le dispositif le plus abouti avant l'avènement des calculettes électroniques, fabriquée au Lichtenstein et vendue à plus de 140'000 exemplaires.



Mise au point au début du XXe siècle, la machine *enigma*, utilisée par les Nazis pendant la Deuxième guerre mondiale, est une machine électromécanique qui lance la transition entre les dispositifs mécaniques et les systèmes électroniques. Elle joue également le rôle de catalyseur pour une partie des travaux d'Alan Turing.

Du simple calcul à la séquence

Le mot algorithme découle de la latinisaton du nom de Muhammad Ibn Mūsā al-Khuwārizmī au [VIII]{.smallcaps} et IXe siècles. Les boites à musique utilisent par la suite un cylindre hérisse de picots, tout comme les automates, à l'instar du canard de Vaucanson¹ (1734). Ces cylindres deviennent par la suite des bandes ou cartes perforées comme dans les métiers de Jacquard ou les orgues de barbarie. Au début du IXe siècle, Georges Boole formalise la logique moderne et ce qu'on appelle aujourd'hui en son honneur l'algèbre de Boole. Cet algèbre est repris plus tard dans la construction des systèmes logiques à la base du fonctionnement des ordinateurs.



En 1837, Charles Babbage propose une machine analytique à l'architecture très innovante. On y trouve une mémoire, une unité arithmétique et logique, ainsi qu'un lecteur de cartes perforées (emprunté aux machines de Jacquard). Lady Ada Lovelace, considérée comme la première informaticienne de l'histoire, collaborera avec Babbage en créant les notions algorithmiques de boucles et de branchements conditionnels, qui déboucheront sur la conception d'une deuxième version de sa machine. Avec les notions de variables et de sous-programme, on trouve esquissés dans cette machine quasiment tous les éléments de l'architecture des ordinateurs modernes. On leur doit notamment la notion de machine générale, au-delà du dispositif dédié à une tâche ou un calcul particulier.

Éclosion

La Deuxième guerre mondiale catalyse des développements sans précédent dans les sciences. On voit ainsi apparaître le spatial, le nucléaire et l'informatique. Ces trois domaines répondent à des enjeux stratégiques et reçoivent un soutien considérable. C'est le cas d'Alan Turing dont le projet est soutenu par Winston Churchill lui-même pour décrypter les messages des nazis.

L'informatique prend son essor à la croisée de trois enjeux majeurs : les développements de l'électronique, la cryptologie et les besoins en calculs paramétriques et automatiques notamment liés à la balistique.

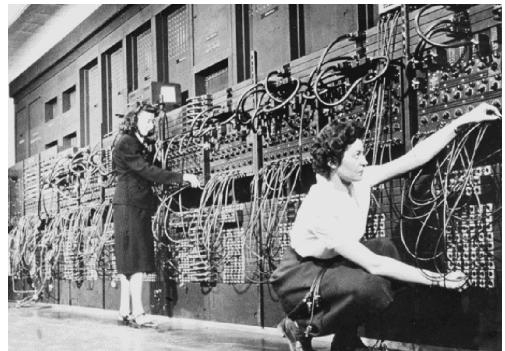
1. Souvent décrit comme la première tentative de robotisation, mais son comportement ne suit qu'un automatisme pré-établi

Développement de l'électronique



Le radar constitue un dispositif plus que stratégique pendant la Deuxième guerre mondiale pour prévenir les raids aériens de l'Allemagne nazie. La RAF (Royal Air Force) défend jalousement cette technologie et l'entoure du secret le plus absolu. Pour justifier les interventions au bon endroit et au bon moment de ses forces aériennes, la propagande diffusera largement l'idée fausse que les aviateurs de la RAF voyaient mieux la nuit parce qu'il mangeaient beaucoup de carottes. D'ailleurs, ce boniment a encore cours aujourd'hui pour convaincre les enfants. Les radio-communications représentent aussi un outil stratégique largement développé en particulier dans la résistance française. Des entreprises suisses comme Kudelski y trouvent leurs racines. L'usage du *tube électronique* ou *tube à vide* ou encore *lampe* développée au début du XXe siècle constitue un appui essentiel à ces développements, en particulier pour le radar. On retrouve également cette technologie utilisée à large échelle dans les premiers ordinateurs qui apparaissent au sortir de la guerre.

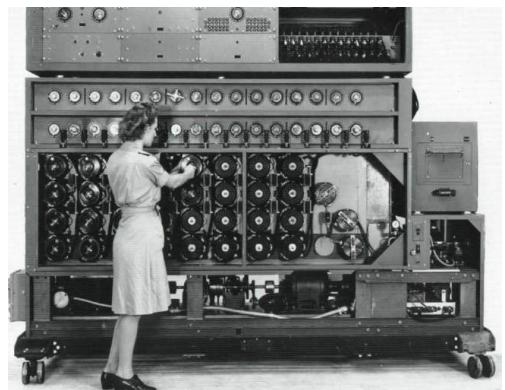
Balistique et projet Manhattan



L'équation bien connue $ax^2 + bx + c = 0$ est à la base des premiers calculs balistiques. Mais la prise en compte de paramètres atmosphériques (humidité, vent, frottement, etc.) rend rapidement les calculs bien plus compliqués. L'armée prépare les *tirs* en remplissant des tables qui sont longues et fastidieuses à établir. L'utilisation d'une machine à faire automatiquement des calculs fait rapidement son chemin dans ce domaine et aboutit à la mise au point d'un des premiers calculateurs automatiques : l'ENIAC.

John Von Neumann, un des pères de la bombe atomique, participe au projet Manhattan² et travaille au *laboratoire national de Los Alamos*. Ses travaux dans la mise au point de la bombe A nécessitent de longs et coûteux calculs. Ils déboucheront sur des algorithmes comme la méthode de Monte-Carlo visant à optimiser les calculs et la mise au point d'ordinateurs permettant de raccourcir ces derniers. On lui attribue la proposition d'une architecture d'ordinateur qui porte son nom et a encore cours aujourd'hui dans les ordinateurs modernes.

Cryptologie



Les *U-Boot*³, sous-marins de l'Allemagne nazie, font des ravages en mer du nord et dans l'atlantique nord. Ainsi, durant le premier semestre 1942, ce sont 2.5 millions de tonnes de ravitaillement, essentiellement à destination du Royaume-Uni, qui sont envoyés par le fond. L'essentiel de la stratégie des U-Boot reposait sur l'envoi de communications cryptées au moyen d'une version sophistiquée de la machine *Enigma* réputée inviolable. À Bletchley Park, principal site de décryptage du Royaume Uni, Alan Turing met au point une

2. Projet top secret de l'armée américaine qui aboutit à la première bombe atomique utilisée dans un conflit armé à Hiroshima : *little boy* larguée par le bombardier B-29 *Enola Gay* sur la tristement célèbre ville.

3. Abréviation d'*Unterseeboot* qui signifie sous-marin en allemand. Les forces sous-marines allemandes étaient commandées par l'amiral Dönitz en appliquant sa stratégie de la meute.

machine électro-mécanique capable de déchiffrer les messages cryptés en une vingtaine de minutes, alors que cela prenait plusieurs jours dans le meilleur des cas à des humains. Ces calculateurs automatiques exploraient bien plus rapidement les possibilités et les contraintes posées par la machine *Enigma*.

6.0.2 Histoire moderne de l'informatique

À la fin de la Deuxième guerre mondiale, les universités, qui ont reçu l'appui de l'armée, se retrouvent donc avec des ordinateurs, à l'époque de gigantesques machines, relativement lentes (quelques dizaines d'opérations par seconde).

Premiers ordinateurs issus de la Seconde guerre mondiale

Allemagne

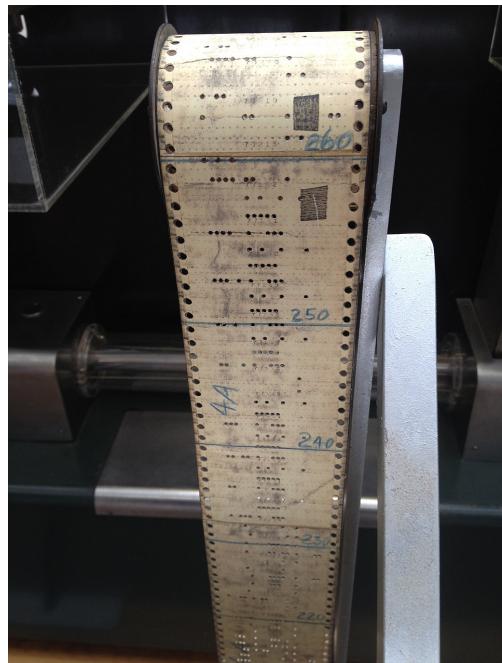
- **Z3, Berlin** : est une machine entièrement électromécanique constituée de relais, dont la construction s'est achevée en 1941. Elle a été entièrement détruite par des bombardements en 1943. Elle est elle aussi considérée comme le premier ordinateur programmable par certains historiens.



Etat-Unis

- **ENIAC, Université de Pennsylvanie** : opérationnel à la fin de 1945, dévoilé au public en février 1946. Il pèse 30 tonnes et occupe une surface de $167 m^2$. Sa fonction est principalement de calculer des tables de tirs. Ces dernières étaient calculées par des femmes engagées par l'armée, et ce sont également ces six femmes appelées “*The Computers*” qui seront les premières programmeuses de cette machine⁴. L'ENIAC est le premier calculateur entièrement programmable.
- Les ordinateurs **Mark 1 et II** construits par IBM à l'*Université de Harvard*, livré en 1944. L'équipe de programmeurs est rapidement rejoints par une jeune mathématicienne recrutée par l'US Navy : Grace Hopper.

4. “The computers”, film documentaire de Jon Palfreman, Kathy Kleiman et Kate McMahon sorti le 12 février 2016, disponible en VOD sur Viméo.



Royaume Uni

- **Colossus**, opérationnel en décembre 1943 à Bletchley Park.
- **Manchester Mark I**, à l'*Université de Manchester*, opérationnel en avril 1949 sur lequel Alan Turing a lui-même effectué des travaux de programmation.

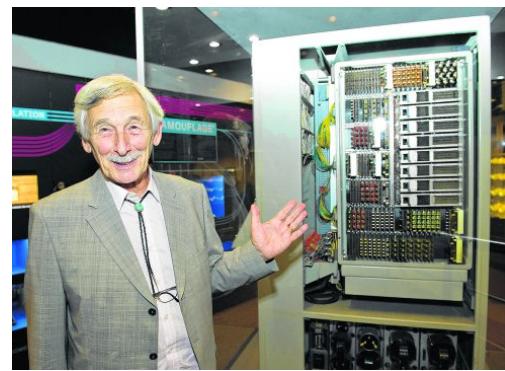
Russie

L'histoire des développements informatiques en Russie à cette époque n'est pas très claire. Les États-Unis interdisent jusque dans les années nonantes l'exportation de tout matériel de calcul sensible, en particulier au niveau des puces électroniques. Les Russes développent leurs propres architectures comme par exemple l'ordinateur **Setun** à l'*Université d'État de Moscou* en 1958, le tout premier ordinateur à travailler en ternaire et non en binaire.

Deuxième vague de calculateurs militaires

Suisse

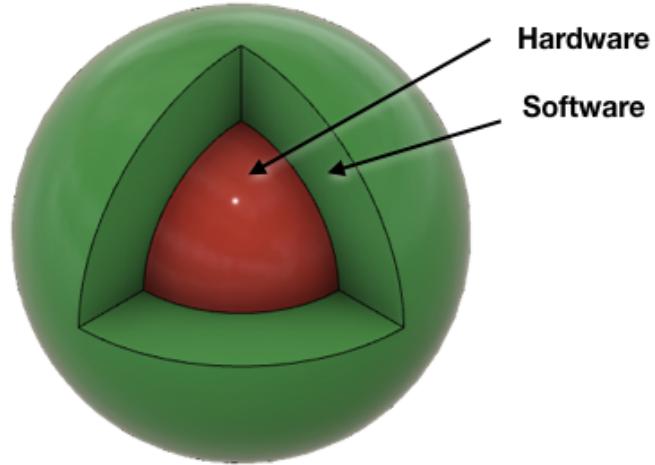
- **Cora 1**, EPFL : pour répondre aux besoins de l'armée, la société Contraves AG développe une machine classée *secret défense* pour réaliser des calculs balistiques et fabriquée à soixante exemplaires. Son concepteur, Peter Toth, commence la conception de cette machine dès la fin des années cinquante.



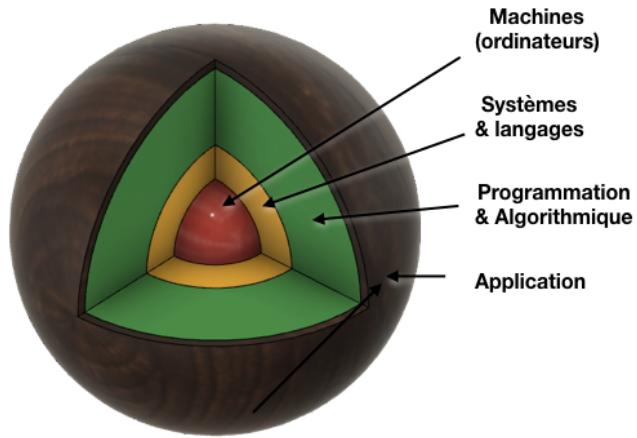
6.0.3 Académisation et apparition de la science informatique

Matériel et logiciels

On réalise rapidement que l'informatique repose sur deux domaines bien distincts : le matériel et les logiciels (en anglais *Hardware* et *Software*, termes qui apparaissent pour la première fois en 1953).



Les premiers ordinateurs sont difficiles à programmer et à utiliser. On se consacre alors essentiellement à des travaux de calcul. Dans le monde académique, on réalise cependant rapidement tout le potentiel de ces machines programmables. Mais un traitement plus conceptuel de l'information demande des outils et notions plus abstraits. La planète informatique incorpore alors plusieurs domaines avec un enrichissement de la partie logicielle qui se construit sur du matériel de plus en plus complexe et performant.



On réalise donc rapidement que le logiciel représente un élément essentiel, et en particulier : le système d'exploitation. Pour concevoir les logiciels, il devient également important de disposer de langages de plus haut niveau que le langage compris par la machine elle-même (le *langage machine*).

Les langages de programmation

Nous donnons ici une synthèse de l'histoire des langages de programmation. Il en existe aujourd'hui un nombre incalculable et on ne trouve pas de recensement exact de toutes les bizarries qui ont été proposées durant les dernières décennies. Ainsi en 1993, Urban Müller invente le *Brainfuck*, dont l'ambition est de proposer un langage simple et pouvant s'exécuter sur une machine de Turing. Le Brainfuck est effectivement Turing-complet, mais les programmes sont difficiles à lire et peu efficaces.

Avant l'élosion

Le premier langage de programmation, avec le tout premier programme informatique jamais écrit par l'humanité, est attribué à Lady Ada Lovelace vers 1840. On trouve ensuite Alan Turing avec sa machine éponyme en 1936, puis Konrad Zuse en 1945.

Premiers langages informatiques

Le premier véritable langage informatique qui s'exécute sur une machine électromécanique revient à nouveau à une femme avec le langage *A-0 System* mis au point par Grace Hopper en 1951. On trouve ensuite les langages suivants qui ont, pour certains, presque disparu, mais qui existent encore pour certaines applications :

- 1954, **FORTRAN**.
- 1958, **ALGOL 58**.
- 1959, **Lisp**, premier langage dit *fonctionnel*.

Invention du transistor

L'invention du transistor en 1957 permet de remplacer les tubes peu fiables, coûteux, encombrants et consommant beaucoup d'énergie. Dès la fin des années cinquante et jusqu'au début des années soixante, les ordinateurs plus fiables et plus complexes utilisent ainsi cette nouvelle technologie.

Nouvelle génération de langages de programmation

Les langages de programmation suivants font leur apparition :

- 1960, **Cobol**.
- 1962, **Simula I**, considéré comme étant à la base des langages orientés objet avec la notion de *classe*.
- 1964, **Basic**, langage impératif facile d'accès et qui démocratise la programmation.
- 1968, **Logo**, le langage à la tortue, mis au point par Seymour Papert pour enseigner la programmation aux enfants.

Apparition des circuits intégrés

Les circuits intégrés sont inventés par Jack Kilby en 1958. C'est le début de l'intégration de quantités phénoménales de transistors dans de toutes petites puces électroniques proposant des fonctions de calcul de plus en plus rapides et complexes. Cette intégration s'accompagne d'une croissance dans le cadre de ce qu'on appelle la troisième vague d'ordinateurs. On peut désormais envisager d'embarquer des unités de calculs programmables dans des dispositifs comme les fusées du programme Apollo de la NASA ou les missiles balistiques intercontinentaux dès le milieu des années soixante. Les ordinateurs commercialisés par IBM puis DEC et enfin Hewlett Packard ne sont plus des machines spécialisées pour un domaine mais couvrent l'ensemble des domaines commerciaux et scientifiques. Cette génération d'ordinateur fonctionne avec une fréquence d'horloge voisine de 1MHz et effectue une centaine de milliers d'opérations par seconde.

Loi de Moore

Le chimiste Gordon Earle Moore fonde la société Intel en 1968. Il énonce déjà en 1963 la loi de Moore qui prédit que le nombre de transistors dans les circuits intégrés doublera tous les 18 mois. Cette loi s'avère relativement valable jusqu'à nos jours où cette croissance commence à flétrir. En 2006, les processeurs d'Intel sont cadencés à 3.6GHz et dépassent 80 million de transistors.

Du mini au micro

Au début des années septante, apparaît la dénomination *miniordinateur* pour marquer une diffusion plus large de ces machines désormais moins encombrantes dans tous les domaines. En 1971, le microprocesseur est inventé par Marcian Hoff (société Intel) et Federico Faggin. Le premier microprocesseur Intel est présenté la même année. On parle de quatrième génération d'ordinateur avec l'arrivée du microprocesseur et son intégration dans son développement qui connaît un nouvel essor.

Langage de programmation

Dès les années septante, les langages suivant apparaissent :

- 1971, **Pascal**, inventé par Niklaus Wirth, professeur à l'école Polytechnique de Zurich.
- 1972, **Smalltalk**, premier langage pur objet et qui marque avec ce nouveau paradigme le début d'une nouvelle ère dans la programmation.
- 1972, **C**, longtemps le langage de programmation le plus utilisé, à la base du système UNIX et encore aujourd'hui au coeur de la plupart des développements autour du système Linux.
- 1974, **SQL**, un langage de requêtes utilisé encore utilisé aujourd'hui pour l'extraction d'information dans la plupart des bases de données (relationnelles).

6.0.4 Apparition de la micro-informatique : Small is beautifull

En 1973 apparaissent les premiers *micro-ordinateurs* qui sont destinés à un usage personnel (simultanément en France et en Allemagne). Le coût des mini-ordinateurs est prohibitif pour certaines sociétés et certains champs d'applications, la micro-informatique permet une diffusion encore plus large de la technologie informatique. Citons parmi ces pionniers l'Apple I de Steve Jobs et Steve Wozniak (figure 0.1{reference-type="ref" reference="AppleI"}), le Commodore PET (figure 0.2{reference-type="ref" reference="PET"}).



Langages de programmation

Avec l'extension de l'informatique à de nombreux domaines, une nouvelle discipline voit le jour dans les années huitantes : le génie logiciel ou la science de la construction logicielle. De nouveaux langages apparaissent pour appuyer ce domaine et renforcer une production de programmes sous pression alors que les besoins en fiabilité deviennent de plus en plus cruciaux.

- 1983, **Ada**, projet du DoD⁵ pour répondre au besoin de mettre en place des projets informatiques plus fiables.
- 1983, **C++**, une évolution du C dans un langage objet.
- 1985, **Postscript**, un langage permettant de composer des documents et qui puisse être interprété par une imprimante.
- 1991, **Python**, langage d'abord confidentiel qui a connu par la suite un essor considérable avec son adoption dans de nombreux développements de Google.
- 1995, **Java**, langage inventé par Sun Microsystems pour répondre au besoin d'indépendance du hardware alors qu'Internet se généralise.
- 1995, **PHP**, langage conçu pour rendre les pages Web interactives, avant leur envoi par le serveur.
- 1995, **JavaScript**, langage conçu pour rendre les pages interactives du côté du client et développé par Netscape⁶.

Démocratisation

Dès 1982, le commodore 64 (figure 0.3{reference-type="ref" reference="commodore64"}), vendu à 20 millions d'exemplaires (il n'existe pas de chiffre exact) et encore aujourd'hui l'ordinateur le plus vendu au monde. Le Raspberry Pi s'approche de ces chiffres, mais en cumulant les ventes de plusieurs modèles différents.



Il est précédé du VIC-20 de la même marque et de nombreuses autres propositions d'ordinateurs extrêmement bon marché et souvent vendus comme des consoles de jeu intelligentes. Dans les plus connus on trouve ainsi le ZX80 (figure :numref :zx80) ou le micro de la BBC.

5. Department of Defense de l'armée américaine, ainsi nommé en référence à la première programmeuse

6. Aujourd'hui disparue, la société Netscape a été fondée par les premiers acteurs du World Wide Web et créateurs du premier navigateur, ancêtre de Firefox



Les ordinateurs ne sont pas encore connectés, mais les utilisateurs, souvent très jeunes, fondent des clubs, se réunissent et échangent des programmes de démonstration. C'est dans ces communautés que naissent souvent les premiers *hackers* qui se lancent des défis pour concevoir sur ces machines à petit budget des animations souvent impressionnantes.

La micro en Suisse

Smaky⁷ est une famille de micro-ordinateurs développés en Suisse dès 1974 au LAMI-EPFL, dirigé par le Professeur Jean-Daniel Nicoud. Commercialisé dès 1978 par la société EPSITEC SA, il intègre rapidement des fonctionnalités réseau et surtout présente dès 1980, avec le Smaky 8, un système d'exploitation multitâche préemptif⁸ qui en fait un ordinateur puissant qui dispose déjà d'une interface avec des fenêtres. Le développement du Smaky a complètement cessé en 1995 et la société EPSITEC SA s'oriente vers le développement de logiciels comme CRESUS, un logiciel de comptabilité adapté au marché suisse.



7. [smaky.ch⁶⁶](http://www.smaky.ch)

66. <http://www.smaky.ch>

8. le multitâche préemptif désigne la capacité d'un système d'exploitation à exécuter ou arrêter une tâche planifiée en cours (Source : Wikipedia). Ce n'est qu'à partir des années 2000 que Windows supporte pleinement ce mécanisme

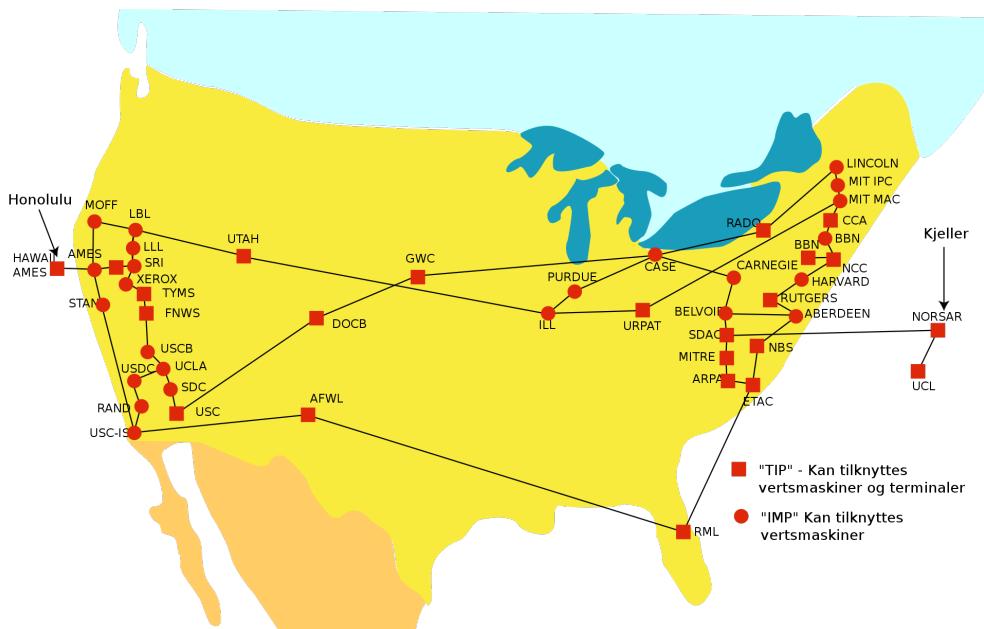
Parallèlement aux Smakys, le Professeur Jean-Daniel Nicoud, développe des machines bon marché à l'intention des clubs d'électronique, d'abord le crocus, puis le Dauphin en 1977 pour 270 CHF. Ce dernier, abondamment utilisé dans les cours du LAMI, est encore aujourd'hui une référence pour apprendre à programmer en langage machine et si ça production a cessé, on trouve un simulateur (disponible à l'adresse : <https://www.epsitec.ch/dauphin/> avec toute la documentation nécessaire) qui permet d'en explorer le fonctionnement.



6.0.5 Internet

Internet, le réseau des réseaux, est d'abord un projet de l'armée américaine pour mettre en place un réseau décentralisé capable de résister à une attaque nucléaire. En effet, les réseaux de télécommunications, initialement, sont conçus sur une architecture en étoile avec un central (téléphonique) dont la destruction entraîne la mise hors service de l'ensemble du réseau.

Le réseau ARPANET⁹ est opérationnel pour la première fois en septembre 1969. Il connecte d'abord quelques universités américaines, comme on peut le voir sur la carte ci-dessous :



Le terme Internet est utilisé pour la première fois en 1972 par Robert E. Kahn. Ce dernier est avec Vint Cerf l'inventeur du protocole TCP/IP qui est la base du réseau Internet. Le terme Internet remplace officiellement le terme Arpanet en 1983.

9. Advanced Research Projects Agency Network

Comme pour les premiers ordinateurs, on retrouve dans le développement d'Internet une impulsion de départ dans les enjeux militaires qui ont suivi la Deuxième guerre mondiale, puis une diffusion dans le monde académique associé aux projets de recherche de l'armée américaine.

USENET et modem

Les connexions au réseau Internet se font souvent avec des MODEM qui permettent des transmissions de quelques centaines de caractères par seconde. Les USENET sont des forums d'échanges où les premières communautés de *Hackers* partagent des programmes, des démos et des astuces.

6.0.6 Le logiciel libre

Alors que le monde industriel s'organise au début des années huitante pour protéger la propriété intellectuelle liée à la conception de programmes, Richard Stallman, chercheur au laboratoire d'intelligence artificielle du MIT¹⁰ s'irrite de cette opacité. L'anecdote rapporte qu'il aurait été particulièrement agacé lorsqu'un de ses anciens collègues aurait refusé de lui confier le code source d'un pilote d'imprimante *buggé*. Il aurait alors ré-écrit complètement le pilote en question et se serait ensuite interrogé sur la meilleure façon de protéger la diffusion libre du programme en question.



Il invente alors la notion de *copyleft*, par opposition au *copyright* : cette licence garantit la libre distribution et l'impossibilité d'aliéner ce droit. En 1984 il crée le projet GNU¹¹ et fonde la *Free Software Foundation*.

Son idée de logiciel libre va durablement modifier le paysage du logiciel informatique et concerne aujourd'hui des millions de produits.

10. Massachusetts Institute of Technology

11. Gnu is Not Unix est un projet de réaliser un système équivalent au système UNIX, mais dont le code source serait librement distribué. Ce projet sera intégré par Linus Torvald qui crée le noyau *Linux*

6.0.7 Le World Wide Web

Time Berners Lee, collaborateur du CERN, constate en 1989 que ces collègues commandent régulièrement des copies d'articles scientifiques ce qui occupe plusieurs employés et utilise beaucoup de papier. Il décide alors de mettre au point un système qui permettrait de partager plus efficacement sur le réseau les résultats des chercheurs et invente pour cela trois protocoles :

- **HTTP** pour Hyper Text Transfer Protocol, un protocole de transfert de documents hypertextes.
- **HTML** pour Hyper Text Markup Language, un langage de description qui permet la mise en forme de documents.
- **URL** pour Uniform Resource Locator, qui définit un adressage universel pour identifier et localiser les documents sur le réseau.

En 1992, il donne à cet ensemble le nom de World Wide Web, la toile mondiale, qui permet de disposer sur Internet des documents liés entre eux par des liens hypertextes. En 1993 il obtient du CERN l'autorisation d'ouvrir la toile au domaine public. En 1992 également, deux étudiants du NCSA (National Center for Supercomputing Applications), Eric Bina et Marc Andreessen, développent le premier navigateur Internet : NCSA Mosaic qui sera disponible pour le public dès 1993. Une bonne partie de l'équipe de développement de Mosaic fonde l'entreprise Netscape¹² qui sera pionnière dans le développement du Web.

Le Web 2.0

Alors que Microsoft et Netscape se plongent dans la guerre des navigateurs, avec Netscape qui représente 90% des navigateurs en 1996, les innovations se succèdent à rythme effréné. Netscape propose trois innovations majeures :

- Les cookies qui permettent de stocker des informations persistant au fil des interactions avec un utilisateur.
- Les CSS (Cascading Style Sheet) qui permettent une mise en forme beaucoup plus riche et cohérente des documents Web.
- Le Javascript qui permet l'exécution de script au sein des documents et permet ainsi des interactions encore plus élaborées avec l'utilisateur.

En 2006 le nombre d'utilisateurs de Netscape tombe à 1%, mais toutes ces innovations permettent le développement du Web 2.0, terme qui apparaît pour la première fois en 2004 et accompagne la naissance d'un nouveau WWW avec en particulier l'apparition des réseaux sociaux.

6.0.8 Ubiquité

Le réseau devient sans fil au milieu des années nonante. D'abord une solution marginale, la bande passante disponible évolue de manière continue et permet d'imaginer de plus en plus d'applications sans installation filaire compliquée et coûteuse à mettre en place.

La miniaturisation de puces électroniques intégrant de plus en plus de fonctionnalités et en particulier la capacité de communiquer par ondes radio permet d'envisager des dispositifs intelligents disséminés un peu partout, on parle d'*ubiquité* du réseau.

12. l'ancêtre de Firefox et d'une bonne partie des développements encore d'actualité

Les smartphones

L'iPhone a été présenté pour la première fois en juin 2007 par Steve Jobs. Ce n'est pas le premier smartphone commercialisé, puisque Blackberry et Treo proposaient déjà des téléphones portables avec des caractéristiques d'ordinateurs dès 1999 pour le premier et 2002 pour le second.

l'IoT

Alors que le marché de la téléphonie mobile explose dès la fin des années nonante, il trouve rapidement sa place dans les applications industrielles avec la mise en place de réseaux de capteurs et de commandes distantes. Avec le développement des smartphones et l'accès au réseau Internet, ce développement connaît un second souffle sous l'appellation *Internet of Things (IoT)*, Internet des Objets. La voiture, la cafetière et même la machine à laver le linge voient leurs possibilités décuplées par l'adjonction de puces électroniques connectées.

6.0.9 Futur

Dans une science qui évolue vite, en perpétuel changement, il est difficile de prédire le futur sans prendre le risque de la boule de crystal.

La suprématie quantique

En octobre 2019, Google a annoncé avoir mis au point un ordinateur quantique capable de faire mieux que les meilleurs super-calculateurs traditionnels. Alors qu'un ordinateur classique manipule des informations élémentaires faites de 0 et de 1 (des bits), un ordinateur quantique manipule des q-bits. Ces derniers stockent de l'information comme une superposition de 0 et de 1 avec une proportion variable. La quantité d'informations ainsi stockée et traitée s'accroît de manière exponentielle. Deux algorithmes sont réputés dans le domaine de l'informatique quantique : l'algorithme de Grover et l'algorithme de Shor. Le premier permet une recherche beaucoup plus efficace dans un index et le deuxième permet une décomposition beaucoup plus efficace en nombres premiers et donne la possibilité de *casser* les codes utilisés couramment aujourd'hui très rapidement. Cela signifierait que la sécurité liée au cryptage de bon nombre de communications et de transactions serait compromise¹³.

Cette section présente une “brève” histoire de l'informatique.

Pour celles et ceux que cela intéresse, le [musée bolo](#)⁶⁷ possède une collection fascinante de machines d'époque, ordinateurs et consoles de jeux vidéos du passé.

13. Problème que certaines sociétés ont déjà anticipé, comme ID Quantique : https://en.wikipedia.org/wiki/ID_Quantique

67. <https://www.museebolo.ch/>