Numération et Logique MLJ2E220

R. Lachièze-Rey

Université Paris Descartes
Poly de G. Koepfler et R. Lachièze-Rey

L1 2016-2017

Planning

- Les cours ont lieu les mercredi de 17h30 à 19h
  en BINET
- Les travaux dirigés en fonction de votre groupe...

Vérifiez régulièrement les informations sur

la page Moodle du cours
https://moodle.mi.parisdescartes.fr/course/view.php?id=35

et sur les affichages du secrétariat

Modalités de contrôle

- Trois notes de contrôle continu :
  - *CC*1 : le **13 mars**
  - *CC*2 : en **avril**
  - Quizz de 10 min : En TD, il est important d'être dans votre groupe de TD - on garde les 3 meilleures notes

- **Coefficients**
  - CC1 :Coef 1
  - Quizz : Coef 1 (+1 ou +2 note de participation)
  - CC2 : Coef 2

Plan de la première partie

Plan de la deuxième partie

Objectifs du module

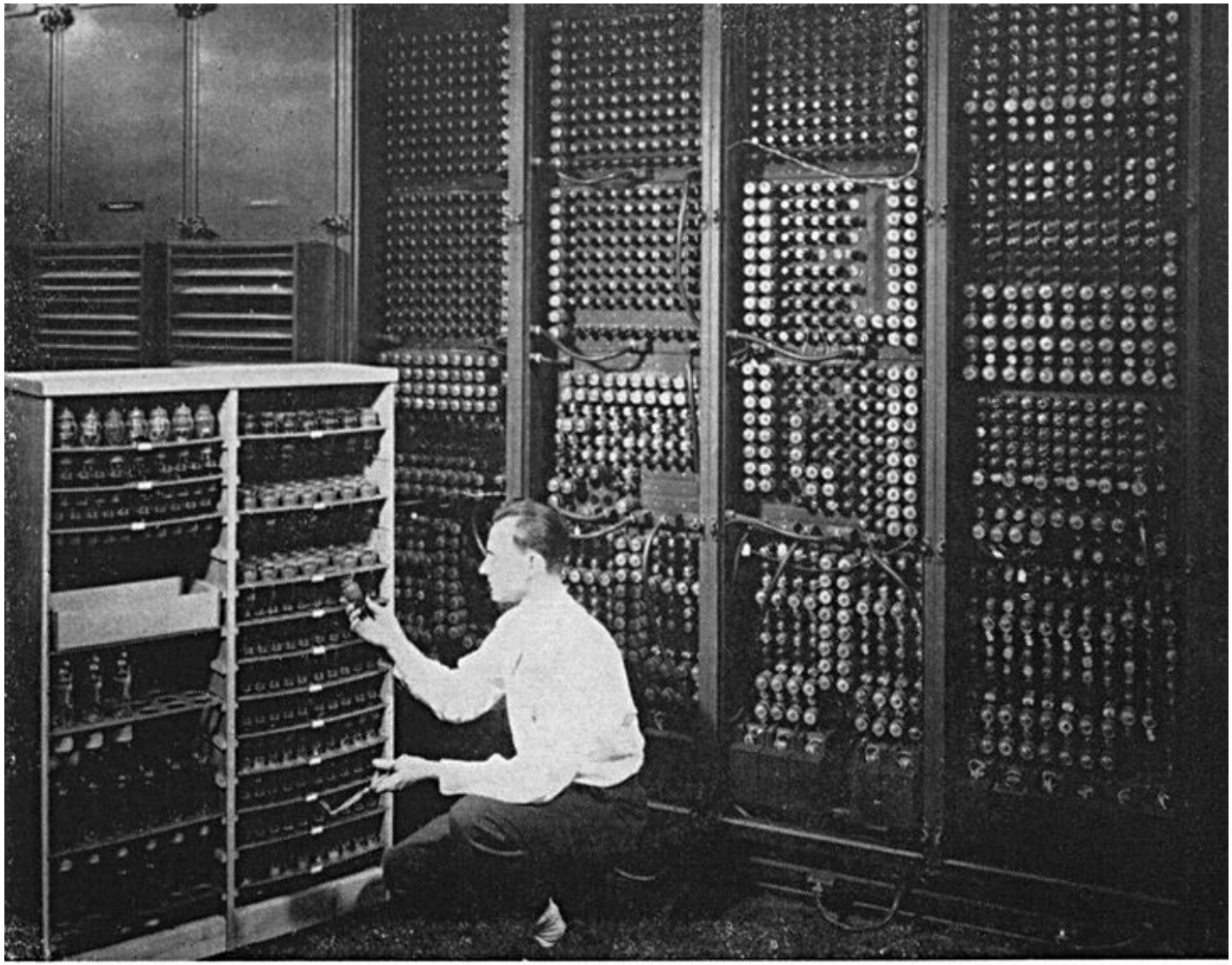
- Comprendre les principes de la numération, les représentations des nombres en machine et les limites de ces représentations
- Comprendre la logique des propositions et ses applications aux déductions et circuits logiques

Instruments de calcul

- Boulier (1100-) ;
- Règle à calcul (1650-1970) ;
- Machine à additionner, à multiplier,
  p. ex. la “Pascaline” de Blaise Pascal (1642) ;
- Machines électroniques,
  Z3 (1938), ENIAC (1946) ;
- Calculatrice (scientifique) de poche électronique (1972) ;
- Ordinateur personnel (1978).

Dans ce cours on n’utilise **pas de calculatrice.**
On s’intéresse aux fondements théoriques !

Voir `http://fr.wikipedia.org/wiki/Instrument_de_calcul`



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

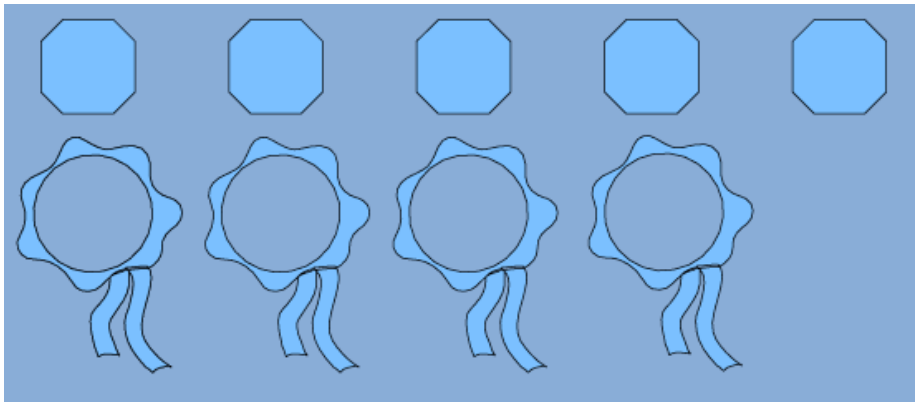
ENIAC, Univ. de Pennsylvanie (1946-55), 30 tonnes sur 167 m^2.
Nombres signés de 10 chiffres : 5.000 additions simples/seconde, 357 multiplications/sec, 38 divisions/sec. (source Wikipedia).

Commençons par “compter”

- La notion de nombre : un, deux, . . . ? beaucoup !
- Sans aide “technologique”, les animaux, homme compris, ne peuvent pas vraiment compter au-delà de 6 ou 7... (cf expérience des oiseaux dans la tour)
- Dans une ancienne tribu brésilienne, on compte jusqu’à 2 !
  Tous les nombres au dessus sont nommés ”boolthа” (beaucoup).

Appariement et grandeur relative de deux collections

- Sans savoir compter, l'homme sait si deux collections ont la même taille, grâce à la technique d'appariement d'objets :

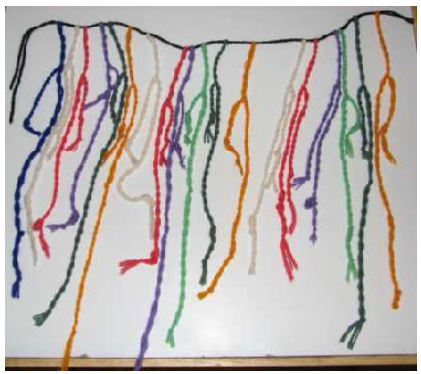


- Par exemple, il peut faire une entaille sur un bâton pour chaque brebis dans un troupeau, ou pour chaque membre de sa tribu. Ca lui permettra de mesurer les variations d'un jour sur l'autre

Repères de dénombrement

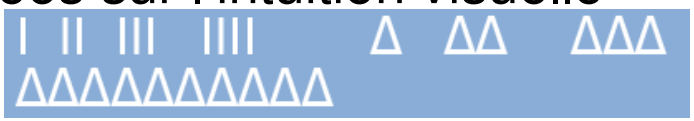
On s'intéresse dorénavant au moyens de comptage ou de dénombrement. Pour cela on peut :

- faire des entailles sur un os, sur une branche
  ou des noeuds sur des cordelettes, ...
- utiliser un repère corporel évident : la main
  ou si besoin, les deux mains, les pieds, la tête, ...



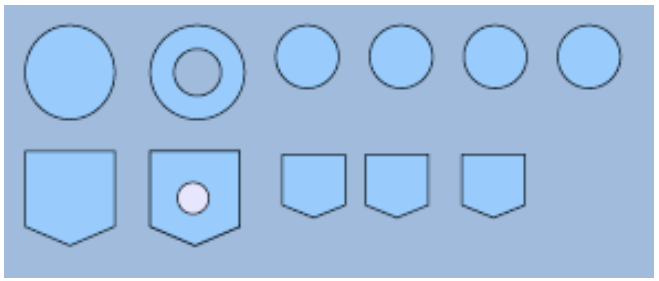
Le nombre et ses symbolisations

Petit à petit, des ”noms” et symboles apparaissent pour les nombres. Ce sont des concepts abstraits, séparés des objets qu’ils dénombrent.

- Symbolisations figuratives : geste de la main, noeuds de cordelette, entailles de l’os et autres objets concrets appariés ;
- Symbolisations verbales : noms intuitifs (main) ou mots détachés de l’intuition (cinq) ;
- Symbolisations écrites :
  - Notations fondées sur l’intuition visuelle
    par exemple 
  - Utilisation de l’initiale du nom du nombre :
    par exemple *C* et *M* chez les romains ;
  - Chiffres : symboles détachés de toute intuition visuelle directe :
    3, 5, 6, . . .

Exemple de représentation des quantités en Mésopotamie

- Fin du IVème millénaire avant notre ère en Mésopotamie on trouve des marques faites en creux dans l'argile avec les significations :

- Sauriez-vous lire ce nombre  ?

$$3600 + 36000 + 10 + 10 + 10 + 10 + 60 + 600 + 1 + 1 + 1 = 40303$$

Remarques

- Les symboles de numération correspondent à des multiplications entre eux par 6 et par 10

$$60 = 6 \times 10, 600 = 10 \times 60, 3600 = 6 \times 600,$$

$$36000 = 3600 \times 10 = 60 \times 60$$

- Un nombre entier est décomposé en multiples de 36000,
  le reste en multiples de 3600
  puis le reste en multiples de 600, ...
- La position des symboles n'a aucune importance.

Notion de base

- Selon les civilisations, le nombre clé de la numération a varié :
  - 60 pour les Sumériens
  - 5 en Amérique du Sud, en Afrique
  - 10 chez les Chinois et les Indiens
  - 12 en Europe dans le système anglais : 12 pouces dans un pied, 12 pence dans un shilling
  - 20 chez les Mayas, au Japon, en Europe, . . . (quatre-vingt)
- Le 10 perdure dans le système décimal actuel ;
  le 60 dans le nombre de minutes/secondes et la mesure d'angles en degrés $6 \times 60 = 360$;

Principe d'additivité

- ⊕ Le nombre de symboles est réduit
  et on répète autant de fois que nécessaire . . .
- ⊕ Il suffit de faire une somme
- ⊖ Fastidieux pour les grands nombres

Exemple :
Ce principe a perduré dans la notation en chiffres romains où l'on a aussi une variante soustractive. Ainsi

$$MMMCCLVIII = 3258\,, \qquad MMMCDXLIV = 3444\,.$$

Principe multiplicatif ou de position

Principe découvert à plusieurs reprises dans des cultures différentes :

- A Babylone (IIème millénaire avant notre ère)
- En Chine (Ier siècle avant notre ère)
  Par exemple 1987 s'écrit 
  et 2026 s'écrit 
  Notez le blanc/vide pour signifier l'absence, i.e. le "0".
- En Inde (vers le Vème siècle après JC)
- Chez les Mayas (IVème au IXème siècle ap. JC)

Exemple de numération de position

- On utilise la base 10
- Il y a dix chiffres de 0 à 9,
  on ajoute 1 à gauche du zéro pour représenter dix objets.
- On obtient ainsi 10, . . . , 99, ensuite 100, . . . , 999, . . .
- Un nombre comme ”7659” est décomposé en

$$7 \times 1000 + 6 \times 100 + 5 \times 10 + 9$$

  et écrit au départ

$$7\ 1000\ 6\ 100\ 5\ 10\ 9$$

- Petit à petit, la notation explicite du rang des unités disparaîtra.

La numération “ moderne ”

- Si l’Inde est considérée comme berceau de la numération moderne, cette culture est arrivée à travers les arabo-musulmans jusque dans l’occident chrétien vers le XIème siècle.
- C’est pourquoi nous parlons de “ chiffres arabes ” d’après un usage établi vers le XVème siècle.
- La civilisation musulmane développera les mathématiques et en particulier le calcul algébrique i.e. la technique de réduction des calculs.

Unités de mesures en informatique

- Un **Bit** (abréviation de BInary digiT)
  C'est la plus petite quantité d'information qui ne peut prendre que deux valeurs 0 ou 1, respectivement "faux" ou "vrai".
  Pour représenter physiquement une information binaire on peut utiliser la polarisation magnétique, le courant électrique, l'intensité lumineuse, ...
  On écrit 1b.

- Un **Octet** (byte en anglais) est composé de 8 bits :

0	1	1	0	0	0	1	0

  On écrit 1o. Les octets sont utiles pour exprimer des quantités de données.

  Note : un octet peut représenter $2^8 = 256$ informations différentes.

Unités de mesures en informatique (suite)

En informatique, les préfixes « kilo », « méga », « giga »,

$$2^{10} = 1024\,,\ 2^{20} = 2^{10} \times 2^{10}\,,\ 2^{30} = 2^{10} \times 2^{10} \times 2^{10}\,.$$

et non pas les puissance de 10 ($10^3 = 1000$, $10^6 = 10^3 10^3$,...) utilisées dans le Système international d'unités (SI).

1 Kilooctet (Ko)	vaut $2^{10} = 1024$ octets
1 Mégaoctet (Mo)	vaut 1024 Ko, soit 2^{20} octets
1 Gigaoctet (Go)	vaut 1024 Mo, soit 2^{30} octets

Attention : ceci est en contradiction avec les recommandations de la Commission électrotechnique internationale (IEC) qui préconise depuis 1999 l'utilisation des **préfixes binaires** !

Préfixes binaires et préfixes décimaux

Préfixes binaires (préfixes CEI)

Nom	Symb.	Facteur
kibi	Ki	$2^{10} = 1\,024$
mébi	Mi	$2^{20} = 2^{10} \times 2^{10}$
gibi	Gi	2^{30}
tébi	Ti	2^{40}
pébi	Pi	2^{50}
exbi	Ei	2^{60}
zébi	Zi	2^{70}
yobi	Yi	2^{80}

Préfixes décimaux (préfixes SI)

Nom	Symb.	Facteur
kilo	k	$10^3 = 1\,000$
méga	M	$10^6 = 1\,000\,000$
giga	G	$10^9 = 1\,000\,000\,000$
téra	T	$10^{12} = 10^3 \times 10^3 \times 10^3 \times 10^3$
péta	P	10^{15}
exa	E	10^{18}
zetta	Z	10^{21}
yotta	Y	10^{24}

- Les préfixes binaires sont préconisés pour les télécommunications et l'électronique. Mais attention, de façon légale on a ainsi $1\text{ko} = 10^3$ octets, $1\text{Mo} = 10^6$ octets et $1\text{Go} = 10^9$ octets.
- Les préfixes décimaux sont utilisés pour les mesures physiques (distance, poids,...).
- L'usage de cette norme n'étant pas encore généralisé, des commerçants de matériel informatique peuvent profiter des confusions : 10% entre tébi et téra !

Voir `http://fr.wikipedia.org/wiki/Pr\unhbox\voidb@x\`

Bases de la numération moderne

- Un entier est représenté par une suite de symboles ou **chiffres**, pris dans un ensemble ou alphabet donné.
- Ces chiffres ont une **position** précise dans la suite de symboles considérée.
- La **base** b indique le nombre de symboles disponibles

$$\{0, 1, 2, \ldots, b-1\}$$

- La suite de chiffres en base b s'interprète dans le système décimal (base 10) grâce à un polynôme arithmétique

$$(s_3\ s_2\ s_1\ s_0)_b = (s_3 * b^3 + s_2 * b^2 + s_1 * b^1 + s_0 * b^0)_{10}$$

où l'on utilise le **poids** b^i pour le symbole à la $i$ème position, s_i.

- On parle de **système pondéré**

Deux principes fondamentaux

1. **Principe de position**
   La signification d'un symbole ou chiffre dépend de son poids, c'est-à -dire de sa place dans la suite des symboles qui est la représentation du nombre.

2. **Principe du zéro**
   Le zéro indique une position où il n'y a pas d'éléments.
   Ainsi 10, signifie 0 unités et 1 dizaine en base 10,
   et de façon générale, en base b, 10 représente le nombre b.

Principales bases utilisées

- **Base décimale** $b = 10$ avec les chiffres : $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
  C'est la base usuelle, utilisée en particulier dans le système métrique des mesures physiques.
- **Base binaire** $b = 2$ avec les chiffres : $\{0, 1\}$.
  Utilisée en informatique, en électricité, base de la logique booléenne ou de l'algèbre de Boole, ...
  C'est aussi la base "minimale" : on ne peut avoir une base qui ne contienne qu'un élément.
- **Base octale** $b = 8$ avec les chiffres : $\{0, 1, 2, 3, 4, 5, 6, 7\}$.
  Utilisée par exemple pour les droits d'accès aux fichiers sous Linux, `chmod 755` au lieu de `rwxr-xr-x`
- **Base hexadécimale** $b = 16$ avec les chiffres :
  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$.
  Très utilisée en micro-informatique (langage assembleur), cette base fournit une représentation compacte des données binaires.

Autres bases utilisées

- **Base 5** avec les chiffres : $\{0, 1, 2, 3, 4\}$.
  Compter jusqu'à 30 avec ses deux mains : les doigts d'une main expriment les unités ceux de l'autre expriment des paquets de cinq.

- **Base 12** établie sur les signes du zodiaque.
  Utile à cause de la divisibilité par 2, 3, 4 et 6.
  Utilisée dans le commerce (oeufs, huîtres, . . .) et pour les heures dans une journée (RV à 2h, cours de 11h à 12h, . . .).

- **Base 20** construite à partir des doigts et des orteils.
  Quelques traces en sont "quatre-vingts", "quatre-vingt-dix", Hôpital des Quinze Vingts,. . .

- **Base 60**, il y a 360^o dans le cercle ; 60 minutes dans une heure, 60 secondes dans une minutes ;
  On a toujours l'expression "soixante-dix",. . .

Système pondéré en base 10

- Base $b = 10$
- Alphabet = $\{0, 1, \ldots, 9\}$
- Représentation polynomiale d'un nombre en base 10 :

$$1942 = 1 \times 10^3 + 9 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

cette écriture utilise 4 positions du code à 10 symboles.

- Chaque position porte le nom de **digit**

Système pondéré en base 2

- Base $b = 2$
- Alphabet = $\{0, 1\}$, i.e. des bits.
- Il y a des digits/bits en des positions particulières :

  MSB, most significant bit ou bit de poids fort.

  LSB, least significant bit ou bit de poids faible,
- Représentation polynomiale d'un nombre en base 2 :

$$(100110)_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

MSB LSB

Donc $(100110)_2 = (38)_{10}$

Système pondéré en base 2 et calculs

- Pour additionner deux nombres en base 2, il suffit de se rappeler que $1 + 1 = (10)_2$ c'est-à -dire $(2)_{10}$.
- Ainsi pour calculer $63 + 28$ en base 2, on peut procéder comme suit :

$(63)_{10}$		111	111
$(28)_{10}$		11	100
retenue		1 1 1	
$(91)_{10}$	1	011	011

  Dès que la somme dépasse $(2)_{10}$ on obtient une retenue !
- Pour multiplier deux nombres en base 2, il suffit de se rappeler que la multiplication par $(10)_2 = (2)_{10}$ entraîne un décalage ("shift" en anglais) vers la gauche, en plaçant 0.

  Exemple : $(1010)_2 * (10)_2 = (10100)_2$

Systèmes pondérés en base b

- Base b
- Alphabet = $\{0, 1, \ldots, b-1\}$
- Code à b symboles
- L'écriture en base b $\quad (s_n\ s_{n-1}\ \ldots\ s_0)_b$

  a comme valeur $\quad s_n b^n + s_{n-1} b^{n-1} + \cdots + s_0 b^0$

  où les s_i sont des "chiffres" de l'alphabet.
- Pour additionner deux nombres en base b, il suffit de se rappeler que dès que la somme dépasse b, on obtient une retenue.
  Exemple : $(34)_5 + (11)_5 = (100)_5 = (25)_{10}$.

Théorème fondamental des bases de la numération

Cas des entiers naturels, éléments de $\mathbb{N}$

Théorème

Soit b un entier naturel avec $b \geq 2$. Alors :

- Pour tout nombre entier $n \in \mathbb{N}^*$, il existe un entier $k \geq 0$ tel que $b^{k-1} \leq n < b^k$;
- Tout nombre entier $n \in \mathbb{N}^*$ se décompose de manière unique sous forme d'une fonction polynomiale en b de degré $k-1$ et avec des coefficients $s_i \in \{0, 1, \ldots, b-1\}$

$$n = \sum_{i=0}^{k-1} s_i b^i \ .$$

Exemples

- Base binaire $b = 2$, symboles $\{0, 1\}$
- Soit par exemple

Nombre $(n)_2 =$	1	0	1	1	0	0	1	1	0	1	0	1
Exposant	11	10	9	8	7	6	5	4	3	2	1	0

- Les bits de poids faible sont à droite.
- Comment écrire ce nombre en base 10 ?

$$2^{11} + 2^9 + 2^8 + 2^5 + 2^4 + 2^2 + 2^0 = (2869)_{10}$$

Exemples (suite)

- Base octale, symboles $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Nombre $(n)_8 =$	5	4	6	5
Exposant	3	2	1	0

Écriture de ce nombre en base 10 :

$$5 * 8^3 + 4 * 8^2 + 6 * 8^1 + 5 * 8^0 = (2869)_{10}$$

- Base hexadécimale,

symboles $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

Le nombre $(B35)_{16}$ s'écrit

$$11 * 16^2 + 3 * 16^1 + 5 * 16^0 = (2869)_{10}$$

Représentation de rationnels

- On peut étendre le principe des systèmes pondérés à des nombres non entiers en utilisant des poids d'exposant négatif, c'est-à -dire de la forme $b^{-i} = \frac{1}{b^i}$, pour $i \geq 1$.
- Base $b = 10$

  Exemple :

  $$156,57 = 1 \times 10^2 + 5 \times 10 + 6 \times 10^0 + 5 \times 10^{-1} + 7 \times 10^{-2}$$

- Ceci permet de représenter les **nombres décimaux**, c'est-à -dire les nombres rationnels ayant un **développement décimal fini**.

  Exemples :

  $$0,333\ ; \quad 45001,004\ ; \quad \frac{1}{5} = 0,2\ ; \quad \frac{3}{2} = 1,5$$

Représentation de rationnels (2)

- Pas tous les rationnels, et a fortiori pas tous les réels, ont une partie décimale limitée :

$1/3$	$=$	$0,3333333333\ldots$ période	$=\sum_{i=1}^{+\infty} 3 \times 10^{-i}$
$1/7$	$=$	$0,142857142857142857\ldots$ période	
$\sqrt{2}$	$=$	$1,41421356237309514 54\ldots$	pas de périodicité

- Un nombre rationnel $x \in \mathbb{Q}_+$ est un nombre décimal si et seulement si l'on peut écrire $x = \dfrac{n}{5^p \times 2^q}$

  où $(n, p, q) \in \mathbb{N}^3$ et n est premier avec 5 et 2.

- Le développement décimal d'un nombre rationnel est soit limité, soit périodique (à partir d'un certain rang).

Représentation de rationnels (3)

- Tout nombre décimal non nul admet aussi un développement décimal infini :

$$1 = 0,999\ldots = \sum_{i=1}^{+\infty} 9 \times \frac{1}{10^i} \; ; \quad 0,2 = 0,19999\ldots$$

- Attention $0,99999 \neq 0,999\ldots = 1$, mais tout nombre réel est la limite d'une suite de nombres décimaux.
  Ceci permet de justifier les calculs par valeurs approchées.

- Afin d'éviter les développements illimités des nombres rationnels, il est souvent préférable de les représenter sous forme de fraction (réduite), on garde le numérateur et le dénominateur en machine.

Représentation de rationnels (4)

On peut utiliser les poids à exposants négatifs dans d'autres bases :

- Base binaire, $b = 2$ et symboles $\{0, 1\}$

$$(101,011)_2 = 1 * 2^2 + 1 * 2^0 + 1 * 2^{-2} + 1 * 2^{-3} = (5,375)_{10}$$

- Base octale et symboles $\{0, 1, 2, 3, 4, 5, 6, 7\}$

$$(65,23)_8 = 6 * 8^1 + 5 * 8^0 + 2 * 8^{-1} + 3 * 8^{-2} = (53,296875)_{10}$$

- Base hexadécimale avec $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

$$(B5, AE)_{16} = 11*16^1+5*16^0+10*16^{-1}+14*16^{-2} = (181,67968)_{10}$$

Représentation de rationnels (5)

- Pour s'avoir si un rationnel positif $x \in \mathbb{Q}_+$ admet un développement limité dans une base $b \geq 2$, il faut d'abord le réduire en une fraction irréductible $x = \frac{a}{c}$. Son développement est infini si et seulement si c a un diviseur premier avec b :

- Exemples :

$$(2/4)_{10} = (1/2)_{10} = (0,1)_2 = (0,8)_{16} = (0,11\underline{1}\ldots)_3$$

$$(1/8)_{10} = (5^4/10^4)_{10} = (0,125)_{10} = (0,001)_2 = (0,1)_8 = (0,2)_{16}$$

$$(1/3)_{10} = (0,33\underline{3}\ldots)_{10} = (0,010\underline{101}\ldots)_2 = (0,1)_3$$

$$(1/6)_{10} = (0,166\underline{6}\ldots)_{10} = (0,0010\underline{101}\ldots)_2 = (0,01111\ldots)_3$$

$$(1/5)_{10} = (0,2)_{10} = (0,00110011\underline{0011}\ldots)_2 = (0,1)_5$$

$$(1/11)_{10} = (0,090\underline{909}\ldots)_{10} = (0.00211002110\underline{0211}\ldots)_3$$
$$= (0.000101110100010111010\underline{001011101}\ldots)_2$$

Conversion entre bases

- Pour passer d'un nombre en base b à un nombre en base 10, on utilise l'écriture polynomiale décrite précédemment.
- Pour passer d'un nombre en base 10 à un nombre en base b, on peut utiliser deux méthodes :
  1. Méthode par soustraction ;
  2. Méthode par multiplication.
- Ces méthodes seront présentées grâce à des exemples, d'abord pour des entiers, ensuite pour des rationnels.
- On présentera aussi une méthode simple pour le passage entre les bases binaire, octale et hexadécimale.

Conversion d'un entier. Méthode par soustraction

Description

Soit $(n)_{10} \in \mathbb{N}^*$ à convertir en base b.
On cherche d'abord $k \in \mathbb{N}$ tel que $b^{k-1} \leq n < b^k$
on aura donc besoin de k positions $(s_{k-1} \cdots s_1 s_0)_b$
On détermine d'abord les digits de **plus fort poids** et ensuite les digits de **poids faible**.

1. s_{k-1} est le nombre de fois que b^{k-1} est dans $n_1 = n$
2. s_{k-2} est le nombre de fois que b^{k-2} est dans $n_2 = n_1 - s_{k-1} b^{k-1}$
3. s_{k-3} est le nombre de fois que b^{k-3} est dans $n_3 = n_2 - s_{k-2} b^{k-2}$

$\vdots$

k-1. s_1 est le nombre de fois que b^1 est dans $n_{k-1} = n_{k-2} - s_2 b^2$

k. $s_0 = n_k = n_{k-1} - s_1 b^1 \in \{0, 1, \ldots, b-1\}$ est le reste

Conversion d'un entier

Soit $n = 173$ à convertir en base $b = 8$.

Comme $8^2 \leq 173 < 8^3$, on a besoin de $k = 3$ positions

1. Dans $n_1 = 173$, combien de fois y a-t-il $8^2 = 64$? 2 fois (MSD) 2
   $n_2 = 173 - (2 * 64) = 45$
2. Dans 45, combien de fois y a-t-il 8 ? 5 fois 5
   $n_3 = 45 - 5 * 8 = 5$
3. On s'arrête car $n_3 = s_0 = 5$ est le reste (LSD) 5

Le résultat est donc $(173)_{10} = (255)_8$

Conversion d'un entier

Soit $n = 173$ à convertir en base $b = 2$.

Comme $2^7 \leq 173 < 2^8$, on a besoin de 8 bits

1	Dans $n_1 = 173$, y a-t-il $2^7 = 128$?	oui	(MSB)1
2	$n_2 = 173 - 128 = 45$; dans 45, y a-t-il $2^6 = 64$?	non	0
3	il reste donc $n_3 = 45$; dans 45, y a-t-il 32 ?	oui	1
4	$n_4 = 45 - 32 = 13$; dans 13, y a-t-il 16 ?	non	0
5	il reste $n_5 = 13$; dans 13, y a-t-il 8 ?	oui	1
6	$n_6 = 13 - 8 = 5$; dans 5, y a-t-il 4 ?	oui	1
7	$n_7 = 5 - 4 = 1$; dans 1, y a-t-il 2 ?	non	0
8	il reste $n_8 = s_0 = 1$, la conversion est finie		(LSB)1

Le résultat est donc $(173)_{10} = (10101101)_2$

Conversion d'un entier. Méthode par division

Description

Soit $(n)_{10} \in \mathbb{N}^*$ à convertir en base b : $(n)_{10} = (s_{k-1} \dots s_1 s_0)_b$
On utilise la **division euclidienne**, encore appelée **division entière**.

1. on effectue la division entière de n par b :
   $n = d_1 \times b + r_1$, on garde $s_0 = r_1$
2. on effectue la division entière de d_1 par b :
   $d_1 = d_2 \times b + r_2$, on garde $s_1 = r_2$

   $\vdots$

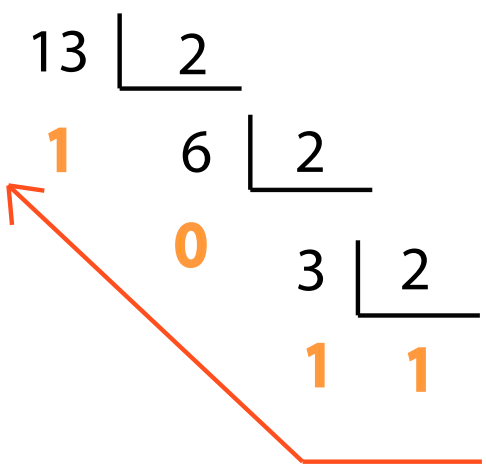
k-1. on effectue la division entière de d_{k-2} par b :
   $d_{k-2} = d_{k-1} \times b + r_{k-1}$, on garde $s_{k-2} = r_{k-1}$

k. quand $d_{k-1} \in \{0, 1, \dots, b-1\}$, $s_{k-1} = d_{k-1}$ est le reste

On détermine d'abord les digits de **faible poids** et ensuite les digits de **poids fort**.

Conversion d'un entier

Soit $n = 13$ à convertir en base $b = 2$



$$13 = 6 \times 2 + 1$$

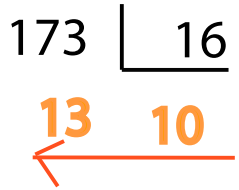
$$6 = 3 \times 2 + 0$$

$$3 = 1 \times 2 + 1$$

Le résultat est donc $(13)_{10} = (1101)_2$

Conversion d'un entier

- Soit 173 à convertir en base $b = 16$



$173 = 10 \times 16 + 13$ avec $10 = A_{16}$ et $13 = D_{16}$

Le résultat est $(173)_{10} = (AD)_{16}$

Les multiples de la base b

On considère la forme polynomiale d'un entier écrit en base b

$$\begin{aligned} n_b &= (s_k s_{k-1} \dots s_1 s_0)_b \\ &= s_k b^k + s_{k-1} b^{k-1} + s_{k-2} b^{k-2} .... + s_1 b^1 + s_0 b^0 \end{aligned}$$

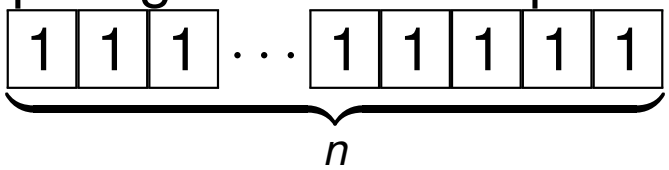
On constate que

1. un multiple de b se termine par 0, $s_0 = 0$;
   il s'écrit $n = b\,(s_k b^{k-1} + s_{k-1} b^{k-2} + s_{k-2} b^{k-3} + \cdots + s_1)$
2. un multiple de b^2 se termine en 00, $s_0 = s_1 = 0$;
3. un multiple de b^3 se termine en 000 ;

   et ainsi de suite.

Représentation binaire d'entiers naturels

- Avec n bits on peut représenter 2^n valeurs, c'est-à -dire dire tous les entiers de 0 à $2^n - 1$;
- Le plus grand entier représentable sur n bits s'écrit :



et vaut $2^n - 1$;

- L'entier $2^n - 1$ est toujours un nombre impair ;
- Ecriture en binaire des nombres 255, 257, 260, 510, 1024, 1019.
  - D'abord chercher les puissances de 2 les plus proches :
    $255 = 2^8 - 1$, $257 = 2^8 + 1$, $260 = 2^8 + 4$,
    $510 = (2^9 - 1) - 1$, $1024 = 2^{10}$ et $1019 = (2^{10} - 1) - 4$;
  - En déduire l'écriture en base 2 :

$$\begin{aligned} (255)_{10} &= (11111111)_2\,, & (257)_{10} &= (100000001)_2\,,\\ (260)_{10} &= (100000100)_2\,, & (510)_{10} &= (111111110)_2\,,\\ (1024)_{10} &= (10000000000)_2\,, & (1019)_{10} &= (1111111011)_2\,. \end{aligned}$$

Conversion facile : Binaire-Octal

- En informatique les bases binaire, octale et hexadécimale sont fréquemment utilisées.
- Toutes ces bases étant des puissances de deux, 2^1, 2^3 et 2^4, il y a des conversions particulièrement simples.
- Pour écrire les 8 symboles de la base octale on a besoin de trois bits

$$(0)_8 = (000)_2,\ (1)_8 = (001)_2, \ldots, (6)_8 = (110)_2\,; (7)_8 = (111)_2\ .$$

1. Pour passer de l'octal en binaire :
   on remplace chaque chiffre octal par les trois bits correspondants.
2. Pour passer du binaire en octal :
   on parcourt le nombre binaire de la droite vers la gauche en regroupant les chiffres binaires par paquets de 3 (en complétant éventuellement par des zéros).
   Il suffit ensuite de remplacer chaque paquet de 3 par le chiffre octal.

Conversion facile : Binaire-Hexadécimal

- Pour écrire les 16 symboles de la base hexadécimale on a besoin de quatre bits

$(0)_{16} = (0000)_2$; $(1)_{16} = (0001)_2$; $(2)_{16} = (0010)_2$; $(3)_{16} = (0011)_2$;
$(4)_{16} = (0100)_2$; $(5)_{16} = (0101)_2$; $(6)_{16} = (0110)_2$; $(7)_{16} = (0111)_2$;
$(8)_{16} = (1000)_2$; $(9)_{16} = (1001)_2$; $(A)_{16} = (1010)_2$; $(B)_{16} = (1011)_2$;
$(C)_{16} = (1100)_2$; $(D)_{16} = (1101)_2$; $(E)_{16} = (1110)_2$; $(F)_{16} = (1111)_2$.

1. Pour passer de l'hexadécimal en binaire :
   on remplace chaque chiffre hexadécimal par les quatre bits correspondants.
2. Pour passer du binaire en hexadécimal :
   on parcourt le nombre binaire de la droite vers la gauche en regroupant les chiffres binaires par paquets de 4 (en complétant éventuellement par des zéros).
   Il suffit ensuite de remplacer chaque paquet de 4 par le chiffre hexadécimal.

Conversion facile, exemples

1. Convertir $(A01)_{16}$ en binaire :
   - on sait que $A_{16} = (1010)_2$; $0_{16} = (0000)_2$ et $1_{16} = (0001)_2$;
   - donc $(A01)_{16} = (\underbrace{1010}_{A}\underbrace{0000}_{0}\underbrace{0001}_{1})_2$.

2. Convertir $(10110)_2$ en base 16 :
   - le regroupement par paquets de quatre donne 0001 0110 ;
   - on associe à chaque paquet le chiffre hexadécimal :

$$\underbrace{0001}_{1}\underbrace{0110}_{6}$$

   - d'où $(10110)_2 = (16)_{16} = (22)_{10}$

Conversion facile, exemples (2)

Pour transformer des nombres avec une partie fractionnaire on procède de la même façon,
mais les regroupements se font **de part et d'autre** de la virgule !

Exemple : conversion de $(1001101011,11001)_2$ en base 16

- on regroupe en paquets de 4 bits de part et d'autre de la virgule

$$00\,10\ 0110\ 1011\ ,\ 1100\ 10\,00$$

- et on associe les chiffres hexadécimaux

$$\underbrace{0010}_{2}\,\underbrace{0110}_{6}\,\underbrace{1011}_{B},\underbrace{1100}_{C}\,\underbrace{1000}_{8}$$

- d'où $(1001101011,11001)_2 = (26B,C8)_{16}$
  $= 2*16^2+6*16^1+11*16^0+12*16^{-1}+8*16^{-2} = (619,78125)_{10}$

Conversion facile, exemples (3)

Convertir $(B5, AE)_{16}$ en base 2 :

- on sait que
  $(B)_{16} = (1011)_2$, $(5)_{16} = (0101)_2$; $(A)_{16} = (1010)_2$
  et $(E)_{16} = (1110)_2$
- d'où

$$(\underbrace{B}_{1011}\underbrace{5}_{0101}, \underbrace{A}_{1010}\underbrace{E}_{1110})_2$$

- et $(B5, AE)_{16} = (10110101, 10101110)_2$

Conversion facile (4)

Conversion de $(1001101011,11001)_2$ en octal.

- on regroupe en paquets de 3 bits de part et d'autre de la virgule

$$001\ 001\ 101\ 011\ ,110\ 010$$

- et on associe les chiffres octaux

$$\underbrace{001}_{1}\underbrace{001}_{1}\underbrace{101}_{5}\underbrace{011}_{3}\ ,\ \underbrace{110}_{6}\underbrace{010}_{2}$$

- d'où

$(1001101011,11001)_2 = (1153,62)_8$
$= 1 * 8^3 + 1 * 8^2 + 5 * 8^1 + 3 * 8^0 + 6 * 8^{-1} + 2 * 8^{-2} = (619,78125)_{10}$

Conversions facile : autre bases

- Cette facilité de conversion a lieu car $8 = 2^3, 16 = 2^4$. Le même effet se produit pour chaque passage entre deux bases où l'une est une puissance de l'autre :

$$(822)_9 = (\underbrace{8}_{2\cdot 3^1+2}\ \underbrace{2}_{2}\ \underbrace{2}_{2})_9 = (220202)_3$$

$$(2110002)_4 = (\underbrace{2}_{2}\ \underbrace{11}_{B}\ \underbrace{00}_{0}\ \underbrace{02}_{2})_4 = (2B02)_{16}$$

- Le même principe s'applique aux nombres à virgules

exemple : adresse MAC

- L'adresse MAC d'une machine, est un identifiant unique au monde qui identifie cette machine (ordinateur, smartphone, imprimante, ...) parmi toutes les autres.
- Elle est constituée de 6 octets en héxadécimal, par exemple :`5E:FF:56:A2:AF:15`.
- Le nombre de machines que l'on peut coder avec ce système est donc de $(2^8)^6 \approx 200\,000$ milliards, on devrait pouvoir s'en sortir...
- Chaque groupe de 2 chiffres "hexadécimaux" se code en 8 chiffres en base 2 :
  - $(5E)_{16} \rightarrow (0101\ 1110)_2$
  - $(FF)_{16} \rightarrow (1111\ 1111)_2$
  - ....

  On a donc bien 6 octets :(01011110)(11111111)...

Conversion de nombres avec partie fractionnaire

- Pour passer d'un nombre en base b, avec partie fractionnaire, à un nombre en base 10,
  on utilise l'écriture polynomiale décrite précédemment.

- Pour passer d'un nombre en base 10, avec partie décimale, à un nombre en base b :

  1. On transforme la partie entière, par la méthode de soustraction ou de division, par rapport à b.
  2. On transforme la partie décimale, par la méthode de soustraction ou de division mais par rapport à b^{-1}
     Note : on verra que cette méthode revient en fait à une **multiplication** par b !

Conversion de la partie fractionnaire (1)

- On s'intéresse à la partie fractionnaire (à droite de la virgule), c'est à dire aux réels dans l'intervalle $(x)_{10} \in]0,1[$ et l'on veut

$$(x)_{10} = (0, s_{-1}s_{-2}\cdots s_{-k}\cdots)_b \quad \text{où } s_{-k} \in \{0,1,\ldots,b-1\}, k \geq 1$$

- Méthode par **soustraction**
  - on détermine d'abord les digits de **plus fort poids** et ensuite les digits de **poids faible**.
  - c'est-à -dire, dans l'ordre, les coefficients de $b^{-1}, b^{-2}, \ldots, b^{-k}, \ldots$
  - pour $k \geq 1$,
    - on détermine combien de fois b^{-k} se trouve dans
      $$x - s_{-1}b^{-1} - \ldots - s_{-(k-1)}b^{-(k-1)}$$
    - on recommence avec $b^{-(k+1)}$ et
      $$x - s_{-1}b^{-1} - \ldots - s_{-(k-1)}b^{-(k-1)} - s_{-(k)}b^{-k}$$
  - **ce procédé ne s'arrête pas nécessairement.**

Conversion de la partie fractionnaire (2)

- Soit $(x)_{10} \in]0,1[$, on veut

$$(x)_{10} = (0, s_{-1}s_{-2}\cdots s_{-k}\cdots)_b \quad \text{où } s_{-k} \in \{0,1,\ldots,b-1\}, k \geq 1$$

- Méthode par **multiplication**
  - on a $x = d \times b^{-1} + r$ où $d \in \{0,1,\ldots,b-1\}$ est le nombre de fois que b^{-1} est dans x et $r \in [0,x[$ est le reste
  - en pratique, au lieu de diviser par b^{-1}, on **multiplie par** b :
    - on calcule $x \times b = d + b \times r$
    - on garde $d \in \{0,1,\ldots,b-1\}$ qui est à gauche de la virgule, le reste $\tilde{x} = b \times r \in [0,1[$ est à droite de la virgule
    - si $\tilde{x} \neq 0$, on recommence en multipliant $\tilde{x}$ par b
  - ce procédé ne s'arrête pas nécessairement
  - on détermine d'abord les digits de **plus fort poids** et ensuite les digits de **poids faible** !

Conversion de la partie décimale

Convertir $(0,28125)_{10}$ en base 2 par soustraction

On détermine successivement les bits coefficients de $2^{-1}, 2^{-2}, 2^{-3}, \ldots$

	Bit
$2^{-1} = 0,50000$ est 0 fois dans $0,28125$	0
$2^{-2} = 0,25000$ est 1 fois dans $0,28125$	1
et $0,28125 - 0,25000 = 0,03125$	
$2^{-3} = 0,12500$ est 0 fois dans $0,03125$	0
$2^{-4} = 0,06250$ est 0 fois dans $0,03125$	0
$2^{-5} = 0,03125$ est 1 fois dans $0,03125$	1
et $0,03125 - 0,03125 = 0$	

Le reste étant nul, on s'arrête et

$$(0,28125)_{10} = (0,01001)_2$$

Conversion de la partie décimale

Convertir $(0,28125)_{10}$ en base 2 par multiplication

0,28125*2 = 0,5625	le coefficient de 2^{-1} est	0
0,56250*2 = 1,125	le coefficient de 2^{-2} est	1
0,12500*2 = 0,25	le coefficient de 2^{-3} est	0
0,25000*2 = 0,5	le coefficient de 2^{-4} est	0
0,50000*2 = 1,0	le coefficient de 2^{-5} est	1

Le reste étant nul, on s'arrête et

$$(0,28125)_{10} = (0,01001)_2$$

Conversion de la partie décimale

Convertir $(0,408)_{10}$ en base 2 par multiplication

- 0,408 *2 = 0,816 — le coefficient de 2^{-1} est 0
- 0,816 *2 = 1,632 — le coefficient de 2^{-2} est 1
- 0,632 *2 = 1,264 — le coefficient de 2^{-3} est 1
- 0,264 *2 = 0,528 — le coefficient de 2^{-4} est 0
- 0,528 *2 = 1,056 — le coefficient de 2^{-5} est 1
- 0,056 *2 = 0,112 — le coefficient de 2^{-6} est 0
- 0,112 *2 = 0,224 — le coefficient de 2^{-7} est 0
- 0,224 *2 = 0,448 — le coefficient de 2^{-8} est 0
- 0,448 *2 = 0,896 — le coefficient de 2^{-9} est 0
- 0,896 *2 = 1,692 — le coefficient de 2^{-10} est 1

Le processus ne s'arrête pas !
La période de longueur 100 apparaît à partir de s_{-47}

Conversion de la partie décimale en base 8

Convertir $(0,28125)_{10}$ en base 8 par multiplication

0,28125*8 = 2,25000	le coefficient de 8^{-1} est	2
0,25000*8 = 2,00000	le coefficient de 8^{-2} est	2

Le reste étant nul, on s'arrête et $(0,28125)_{10} = (0,22)_8$

On peut vérifier en passant par la base 2 :

On avait trouvé $(0,28125)_{10} = (0,01001)_2$

Il suffit de décomposer par paquets de 3 et écrire les symboles :

$$0\ ,\ \underbrace{010}_{2}\ \underbrace{010}_{2}$$

Codage binaire de l'information

- L'architecture actuelle des ordinateurs nécessite une représentation en binaire de toute information :
  $\{0, 1\}$, $\{$faux, vrai$\}$, $\{$éteint, allumé$\}$, $\{$noir, blanc$\}$,...
- Dans un manuel chinois, le Yi Jing (premier millénaire av. JC), on trouve un système binaire lié au $\{$Yin, Yang$\}$ ou $\{$actif, passif$\}$
- Leibniz (1646-1716) connaît ces travaux et publie en 1703 un Compte Rendu de l'Académie des Sciences au sujet de la représentation des nombres en binaire.
- Dans le cadre de ses travaux en logique, Boole (1815-1864) crée une algèbre n'acceptant que deux valeurs numériques : 0 et 1.
  C'est la naissance de l'**algèbre de Boole** ou **calcul booléen**.

Comment coder les nombres entiers en machine ?

- Il faut pouvoir représenter les entiers relatifs,
  i.e les entier naturels munis d'un signe.
- Les opérations arithmétiques $+$, $-$, $\times$ et $/$ doivent être faciles à effectuer.
- Quelle que soit l'architecture du matériel,
  la taille des **mots mémoire** est toujours limitée : 16, 32, 64,. . . bits.

  Il faut

  1. représenter l'information de la façon la plus compacte possible.
  2. avoir des mots mémoire de taille suffisamment grande
     afin de ne pas produire des dépassements de capacité (en anglais overflow).

ARIANE 5

Le vol 501 de la fusée Ariane 5 (4 juin 1996) s'est soldé par un échec

37 secondes plus tard. . .

ARIANE 5 (suite)

- La commission d'enquête a notamment relevé qu'une conversion d'un nombre en virgule flottante (sur 64 bits) vers un entier signé (sur 16 bits) a causé un dépassement de capacité.
- Entraînant toute une série d'actions, cette erreur a abouti à la destruction de la fusée.
- Le nombre en question provenait des mesures d'accélération horizontale de la fusée Ariane 5.
- Le code en question provenait de la fusée Ariane 4, dont l'accélération maximale pouvait être codée sur 16 bits
  or les accélérations sont 5 fois plus fortes pour Ariane 5 !
- Avec une perte de matériel d'une valeur totale de 370 M$, c'est l'un des bugs informatiques les plus coûteux de l'histoire.

Voir `http://fr.wikipedia.org/wiki/Vol_501_d'Ariane_5`

Codage des entiers en binaire

On va présenter quatre codages, avec leur avantages et défauts :

1. Codage binaire naturel signé
2. Décimal codé binaire
3. Codage “complément à deux”
4. Codage “complément à un” (brièvement)

Codage binaire naturel signé

- On fixe la longueur des mots
- Le bit de poids fort est réservé au signe

  Exemple : sur 4 bits,
  $+6$ est codé par 0110 et -6 est codé par 1110

- Avantages et inconvénients :
  - (+) Codage/décodage très facile.
  - (+) Représentation des entiers négatifs.
  - (−) Il y a deux représentations de 0.
  - (−) Les opérations arithmétiques ne sont pas faciles.

Décimal codé binaire

Dans cet ancien système, aussi appelé **binary coded decimal** (BCD), chaque chiffre décimal est codé en binaire sur 4 bits :

$$0_{10} = 0000\ ; 1_{10} = 0001\ ; \ldots\ 8_{10} = 1000\ ; 9_{10} = 1001\ .$$

Exemple : L'entier positif 19032 est codé par

0001 1001 0000 0011 0010

Avantages et inconvénients :

- (+) Codage/décodage facile.
- (+) Pas de limitation des grandeurs représentées.
- (-) On gâche de l'espace : 6 combinaisons sont non utilisées.
- (-) Opérations arithmétiques compliquées.

Remarque : Des représentations de type BCD sont utilisés dans des systèmes de bases de données (SGBD) car elles permettent de stocker des nombres plus grands et de manière plus précise.

• Utilisé aussi dans des systèmes basiques avec affichage digital, un sous-circuit pour chaque chiffre.

Complément à 2 : CA2

1. On fixe le nombre d'octets que l'on va utiliser pour stocker des nombres et faire des opérations : 1,2,3,4,... octets
2. On en déduit la quantité d'informations que l'on va pouvoir coder dans ce système :
   - 8 bits (1 octet) : On peut coder $2^8 = 256$ nombres
   - 16 bits (2 octets) : On peut coder $2^{16} = 65536$ nombres
   - ...
3. Le système doit pouvoir coder les nombres positifs et négatifs :
   - 1 octet : 256 nombres $\rightarrow$ de -128 à +127
   - 2 octets : 512 nombres $\rightarrow$ de -32768 à +32767
   - ...

- On code les entiers positifs par les séquences de bits commençant par 0 pour que leur code soit exactement leur écriture binaire
- On code les entiers négatifs par les séquences commençant par 1
- Sur 8 bits

0	0	0	0	0	0	0	0	$\rightarrow 0$
0	0	0	0	0	0	0	1	$\rightarrow 1$
				...				
0	1	1	1	1	1	1	1	$\rightarrow 127$
1	0	0	0	0	0	0	0	$\rightarrow -128$
				...				
1	1	1	1	1	1	1	0	$\rightarrow -2$
1	1	1	1	1	1	1	1	$\rightarrow -1$

- Le 1er bit est naturellement appelé “bit de signe”

Complément à deux

- On travaille sur k bits et on représente $\{-2^{k-1}, \ldots, 2^{k-1}-1\}$.
  8 bits :$\{-128, \ldots, 127\}$.
- Le complément à deux d'un entier négaif $-n$ est obtenu par le calcul de la quantité $2^k - n$.
  8 bits
  $-1 : (2^8 - 1)_{10} = (127)_{10} = (11111111)_2$
  $-2 : (2^8 - 2)_{10} = (126)_{10} = (11111110)_2$
  $-127 : (2^8 - 127)_{10} = (2^8 - 2^7) = (128)_{10} = (10000000)_2$
- Le complément à deux du complément à deux d'un entier positif $n \leqslant 2^{k-1}$ est l'entier lui même.
- Le nom complet de cette opération est "complément à 2^k" qui est en général tronqué en "complément à 2".

Complément à deux sur k bits ($CA2_k$)

Si l'entier n est codé sur k bits en complément à deux :

$$n \quad : \quad \boxed{s_{k-1}}\boxed{s_{k-2}}\cdots\boxed{s_3}\boxed{s_2}\boxed{s_1}\boxed{s_0}$$

alors $(n)_{10} = -s_{k-1}2^{k-1} + \sum_{i=0}^{k-2} s_i\, 2^i$.

En complément à deux, le bit de signe s_{k-1} a comme poids -2^{k-1} .

Pour $k = 8$ bits, on a les entiers signés :

$$\begin{array}{rcl}
-128 & = & \boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0} \\
 & \vdots & \\
-1 & = & \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1} \\
0 & = & \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0} \\
1 & = & \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1} \\
 & \vdots & \\
127 & = & \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}
\end{array}$$

Complément à deux

Pour un codage sur 4 bits on a :

0	=	0	0	0	0	-8	=	1	0	0	0
1	=	0	0	0	1	-7	=	1	0	0	1
2	=	0	0	1	0	-6	=	1	0	1	0
3	=	0	0	1	1	-5	=	1	0	1	1
4	=	0	1	0	0	-4	=	1	1	0	0
5	=	0	1	0	1	-3	=	1	1	0	1
6	=	0	1	1	0	-2	=	1	1	1	0
7	=	0	1	1	1	-1	=	1	1	1	1

On a 2^4 valeurs distinctes de $-8 = -2^{4-1}$ à $+7 = 2^{4-1} - 1$.
Vérifier que :

- Si $n \in \mathbb{N}^*$, l'entier négatif $(-n)_{10}$ est codé par $(2^4 - n)_2$;
- si le code *CA*2, $m = (s_3 s_2 s_1 s_0)$ commence par $s_3 = 1$, alors m représente la valeur $(m)_{10} - 2^4$.

Algorithme de conversion (à savoir faire dans les deux sens)

- Soit un nombre $m \in \mathbb{Z}$
- Si $m \notin \{-2^{k-1}, ..., 2^{k-1} - 1\}$, on ne peut pas l'écrire en $CA2_k$.
  Sinon, on distingue 2 cas :
  - Si $m \in \{0, ..., 2^{k-1} - 1\}$ est positif, on écrit simplement son écriture binaire, et on rajoute des 0 devant pour former k bits. Exemple : $m = 27$ en $CA2_8$:$27 = 16 + 8 + 2 + 1 = (11011)_2$, donc le codage est [00011011]
  - Si $m = -n$ avec $n \in \{1, ..., 2^{k-1}\}$, il faut "calculer" $2^k - n$ en binaire, mais on a une méthode simple :
    exemple : $m = -27$ en $CA2_8$
    - On écrit n en binaire sur 8 bits
      $27 = (00011011)_2$
    - On inverse bit à bit
      [00011011] devient [11100100]
    - On ajoute 1 en binaire
      [11100100] devient [11100101]
      Remarque :
      $2^8 - 27 = 256 - 27 = 229 = 128 + 64 + 32 + 4 + 1 = (11100101)_2$

Décodage

- Pour le décodage d'un code qui commence par 0, on fait simplement la conversion binaire $\rightarrow$ décimal Exemple : [00011011] est le code de $27 = (11011)_2$
- Si le code commence par 1, il s'agit du codage d'un nombre du type $-n$. Pour trouver n, il y a deux méthodes :
  1.
     - On extrait les $(k-1)$ bits derrière le 1.
       Exemple : [11100101] devient (1100101)
     - On retranche 1
       (1100101) devient (1100100)
     - On inverse bit à bit
       (1100100) devient (0011011)
     - On convertit du binaire au décimal
       (0011011) devient 27
  2.
     - On extrait les $(k-1)$ bits derrière le 1.
     - On inverse bit à bit
       (1100101) devient (0011010)
     - On AJOUTE 1
       (110010) devient (110011)
     - On convertit du binaire au décimal

Complément à deux

Avantages et inconvénients :

- (+) Codage/décodage facile.
- (+) Représentation unique de zéro.
- (+) Opérations arithmétiques faciles (cf. addition).
- (-) Taille mémoire fixée.

Qu'est-ce qu'un bon système d'addition ?

- Un bon codage de nombres doit être facilement compatible avec les opérations mathématiques de base, **addition** et **soustraction**
- Comme la taille est fixée, on ne peut pas faire en sorte que toute addition de nombre reste dans la portée
  $110 + 80 = 190$, ou $-110 - 80 = -90$, on peut toujours trouver deux nombres en $CA2_8$ dont l'addition va sortir de l'intervalle autorisé $\{-127, ..., 128\}$
- Voilà ce qu'on demande au système :
  - Si l'on additionne deux nombres de signes opposés qui sont dans l'intervalle du $CA2_k$, alors on a facilement le résultat (on ne peut pas sortir de l'intervalle par une opération de ce type)
    $110 + (-80) = 30$ et $-110 + 80 = -30$, tous ces nombres sont bien codables en $CA2_8$
  - Si l'on additionne deux nombres de même signe, le système doit
    - Soit nous donner facilement le résultat si il est dans le bon intervalle
      $110 + 24 = 154, -110 - 24 = -154$
    - Soit nous laisser détecter facilement si le résultat sort de l'intervalle
      $110 + 80 = 190$ "OVERFLOW"

Complément à deux : addition, signes opposés

- Le résultat est toujours représentable.
- Exemple sur un octet, $k = 8$:

$+63$	:		0011	1111
-63	:		1100	0001
report		1	1 1 1 1	1 1 1
		1	0000	0000
0	:		0000	0000

- On ne tient pas compte de la retenue.

  On effectue $+63 + (256 - 63) - 2^8$.

Complément à deux : addition, signes opposés

- Exemples sur un octet, $k = 8$:

-63	:		1100	0001		$+63$	:		0011	1111
$+28$	:		0001	1100		-28	:		1110	0100
report		0	0			report		1	1 1 1 1	1
								1	0010	0011
-35	:		1101	1101		$+35$	:		0010	0011

- S'il n' y a pas de retenue, on lit le résultat directement.

  S'il y a une retenue, on la néglige, i.e. on retranche 2^8.

Complément à deux : addition, même signe

- L'addition de deux nombres de même signe peut donner lieu à un **dépassement de capacité** !
- Cas de deux entiers de signe positif.
  On a un dépassement de capacité quand la retenue est distincte du dernier bit de report (i.e. celui sur le bit de signe).

  Exemples : sur 8 bits, nombres de -128 à $+127$

$+35$	:		0010	0011
$+65$	:		0100	0001
report		0	0 0 0 0	0 1 1
$+100$	:		0110	0100

$+103$	:		0110	0111
$+65$	:		0100	0001
report		$0\neq$	1 0 0 0	1 1 1
$+168$	$\neq$		1010	1000

à droite dépassement de capacité, on obtient un nombre qui représente -88 en complément à deux.

Complément à deux : addition, même signe

- Cas de deux entiers de signe négatif.

  On a toujours une retenue, que l'on oublie.
  En effet, on calcule $(2^k - n_1) + (2^k - n_2) - 2^k$.

  On a un dépassement de capacité quand la retenue est distincte du dernier bit de report (i.e. celui sur le bit de signe).

  Exemples : sur 8 bits, nombres de -128 à $+127$

-35	:		1101	1101
-65	:		1011	1111
report		1	1 0 0 0	0 1 1
		1	1001	1100
-100	:		1001	1100

-103	:		1001	1001
-65	:		1011	1111
report		$1\neq$	0 1 1 1	1 1 1
		1	0101	1000
-168	$\neq$		0101	1000

Dépassement de capacité.

Résumé pour l'addition en "complément à deux"

- 2 nombres de signes opposés
  - ⋆ Le résultat est représentable avec le nombre de bits fixés, *pas de dépassement de capacité* ;
  - ⋆ s'il y a une retenue, on l'oublie !
  - ⋆ On lit directement le résultat codé en *CA2*
- 2 nombres de même signe
  - ⋆ Il y a *dépassement de capacité* si la retenue est distincte du dernier bit de report (i.e. celui sur le bit de signe) ;
  - ⋆ s'il y a une retenue on l'oublie ! (une fois qu'on a vérifié qu'il n'y avait pas overflow)
  - ⋆ On lit directement le résultat codé en *CA2*

Conclusion :

1. L'addition en codage complément à deux est simplement l'addition binaire. On ne garde jamais la retenue.
2. On détecte les dépassements de capacité grâce à un seul test pour tous les cas de figure.
3. Pour les autres opérations arithmétiques sur les entiers signés, le codage complément à deux présente des avantages similaires. Ce codage est donc souvent choisi en pratique.

Complément à 1

Il existe un autre système que l'on n'étudiera pas en détail : Le "Complément à 1" ($CA1_k$). Il est très similaire mais a les différences suivantes :

- Entiers positifs : commencent par 0 $\Rightarrow$ code = écriture binaire
- Entiers négatifs : commencent par 1
- Sur 8 bits

0	0	0	0	0	0	0	0	$\rightarrow 0$
0	0	0	0	0	0	0	1	$\rightarrow 1$
								...
0	1	1	1	1	1	1	1	$\rightarrow 127$
1	0	0	0	0	0	0	0	$\rightarrow -127$
								...
1	1	1	1	1	1	1	0	$\rightarrow -1$
1	1	1	1	1	1	1	1	$\rightarrow$ " $-0'' = 0$

Avantages

- Plus facile de coder/décoder les nombres négatifs : On inverse simplement bit à bit
  $-27 :\to 27 = (11011)_2 \to [11100100]$ est le codage de -27

Inconvénients

- Opérations arithmétiques plus compliquées
- Détection de l'overflow plus compliquée
- 2 manières de coder le "0", et donc 2 tests à faire pour voir si un nombre est nul
- On code un nombre de moins : $-127 \to 127$ au lieu de $-128 \to 127$

Application du codage complément à deux dans le langage C

Dans le fichier `/usr/include/bits/wordsize.h`
se trouve la taille des mots mémoire : `#define __WORDSIZE 32`

Dans le fichier `/usr/include/limits.h` sont précisés :

Type	Taille	Magnitude
signed char	8 bits	- 128 à + 127
unsigned char	8 bits	0 à 255
signed short int	16 bits	- 32 768 à + 32 767
unsigned short int	16 bits	0 à 65 535
signed (long) int	32 bits	- 2 147 483 648 à + 2 147 483 647
unsigned (long) int	32 bits	0 à 4 294 967 295

Note : `gcc 4.5.1` et `ISO C99 Standard: 7.10/5.2.4.2.1`

Exemple de code C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
  unsigned char    uc1, uc2, uc3;    signed char   sc1, sc2, sc3;
  unsigned short   ui1, ui2, ui3;    signed short  si1, si2, si3;

  printf("\n Taille de  char     : %d octets \n\n",sizeof(char));

  uc1 = 200 ; uc2 = 60 ; uc3 = uc1 + uc2 ;
  printf("(unsigned char) uc1 = %d, uc2 = %d, uc1+uc2 = %d \n",uc1, uc2, uc3) ;

  sc1 =  103 ; sc2 =  65 ; sc3 = sc1+sc2 ;
  printf("(signed char) sc1 = %d, sc2 = %d, sc1+sc2 = %d \n",sc1, sc2, sc3) ;

  sc1 = -103 ; sc2 = -65 ; sc3 = sc1+sc2 ;
  printf("(signed char) sc1 = %d, sc2 = %d, sc1+sc2 =  %d \n",sc1, sc2, sc3) ;

  printf("\n Taille de  short    : %d octets\n\n",sizeof(short));

  ui1 = 6000 ; ui2 = 60000 ; ui3 = ui1+ui2 ;
  printf("(unsigned short) ui1 = %d,  ui2 = %d, ui1+ui2 = %d \n",ui1, ui2, ui3) ;

  si1 = -10000 ; si2 = -30000 ; si3 = si1+si2 ;
  printf("(signed short) si1 = %d, si2 = %d, si1+si2 = %d \n",si1, si2, si3) ;
}
```

Exemple de code C : affichage des résultats

```
Taille de  char      : 1 octet(s)

(unsigned char ) uc1 =  200, uc2 = 60, uc1+uc2 = 4

(signed char) sc1 =  103, sc2 =  65, sc1+sc2 = -88
(signed char) sc1 = -103, sc2 = -65, sc1+sc2 =  88

Taille de  short     : 2 octet(s)

(unsigned short) ui1 =   6000, ui2 =  60000,
                                    ui1+ui2 =   464
(signed short)   si1 = -10000, si2 = -30000 ,
                                    si1+si2 = 25536
```

Ces faux résultats sont dus au dépassement de capacité des opérations effectuées en code complément à deux.

Conclusion

- La représentation des nombres entiers est limitée par la taille du mot mémoire qui leur est affectée.
- Le code le plus souvent utilisé est le code complément à deux. Il n'a qu'une seule représentation du zéro, les opérations arithmétiques et la détection de dépassement de capacité sont faciles à effectuer.
- Dans tous les cas et en fonction des architectures d'ordinateurs il y aura toujours des opérations dont le résultat n'est pas représentable.
  Sans précautions, elles engendrent des résultats aberrants ou empêchent la poursuite des calculs.

- Souvent il n'est pas commode d'utiliser une représentation en virgule fixe :
  - La masse de la terre est de 5 973 600 000 000 000 000 000 000 kg ;
  - La masse du soleil est de 19 891 $\underbrace{000\cdots000}_{26 \text{ zéros}}$ kg ;
  - La masse d'un électron est de 0,$\underbrace{00\cdots00}_{27 \text{ zéros}}$91093822 grammes ;
  - La masse d'un proton est de 0,$\underbrace{00\cdots00}_{23 \text{ zéros}}$16726 grammes.
- On utilise plutôt la notation scientifique de la forme $a \times 10^e$, $e \in \mathbb{Z}$. Pour la notation scientifique **normalisée** on a $1 \leq |a| < 10$, tandis que pour la **notation ingénieur** $1 \leq |a| < 10^3$ et l'exposant e est un multiple de 3.
  Exemples : Pour les masses de la terre et du soleil, on écrit $5{,}9736 \times 10^{24}$ kg et $1{,}9891 \times 10^{30}$ kg.
  Pour l'électron et le proton on a $9{,}1093822 \times 10^{-31}$ kg et $1{,}6726 \times 10^{-27}$ kg.

Réels en machine

Comment représenter des nombres réels en machine ?

- Le choix de la taille du mot mémoire influence la précision de la représentation des nombres.
  - Pour représenter exactement un rationnel $r = \frac{n}{d}$ il faut garder le numérateur $n \in \mathbb{Z}$ et le dénominateur $d \in \mathbb{N}^*$,
    sauf si dans la base choisie, r admet un développement fini ;
  - Un nombre irrationnel $x \in \mathbb{R} \setminus \mathbb{Q}$ ne peut jamais être représenté exactement.
- Sur un ordinateur, on utilise les nombres à virgule flottante de la forme $x = s \times m \times b^e$
  où b est la base ; $s \in \{-1, +1\}$ est le signe ;
  la mantisse m, ou significande, précise les chiffres significatifs ;
  l'exposant e donne l'ordre de grandeur.
  Exemple : en base 10

$$-37{,}5 = -37500 \times 10^{-3} = -0{,}000375 \times 10^{5} = -0{,}375 \times 10^{2}\,.$$

Réels en virgule flottante, base 10

Représentation en virgule flottante normalisée $x = s \times m \times 10^e$

Exemple : $x = -0{,}375 \times 10^{+2}$

- → le signe du nombre $s = (-1)^{s_m}$, avec $s_m \in \{0, 1\}$;
- → la mantisse m est un réel dans $]0, 1[$:
  tous les chiffres significatifs sont à droite de la virgule ;
- → le digit de poids fort de la mantisse est différent de zéro,
  le zéro est donc non représentable ;
- → l'exposant e est un entier relatif ;
- → la virgule et la base sont représentées de façon implicite ;
- → cette représentation du nombre est unique.

s_m	e	d_{-1}	d_{-2}	$\cdots$	d_{-p}

$$= (-1)^{s_m}\, 0, d_{-1} d_{-2} \ldots d_{-p} \times 10^e \quad \text{avec } d_{-1} \neq 0$$

Par **convention**, la représentation de 0 ne contient que des zéros.

Exemple en base dix (1)

- On considère une représentation avec
  1. une mantisse de 3 chiffres décimaux ;
  2. un exposant de 2 chiffres décimaux ;
  3. deux bits de signe.
- Exemple : $37{,}5 = 0{,}375 \times 10^2$ est représenté par

+	+	0	2	3	7	5

- Les nombres strictement positifs représentables vont de

$+0{,}100 \times 10^{-99}$:

+	-	9	9	1	0	0

à $+0{,}999 \times 10^{+99}$:

+	+	9	9	9	9	9

- Les nombres strictement négatifs représentables vont de

$-0{,}999 \times 10^{+99}$:

-	+	9	9	9	9	9

à $-0{,}100 \times 10^{-99}$:

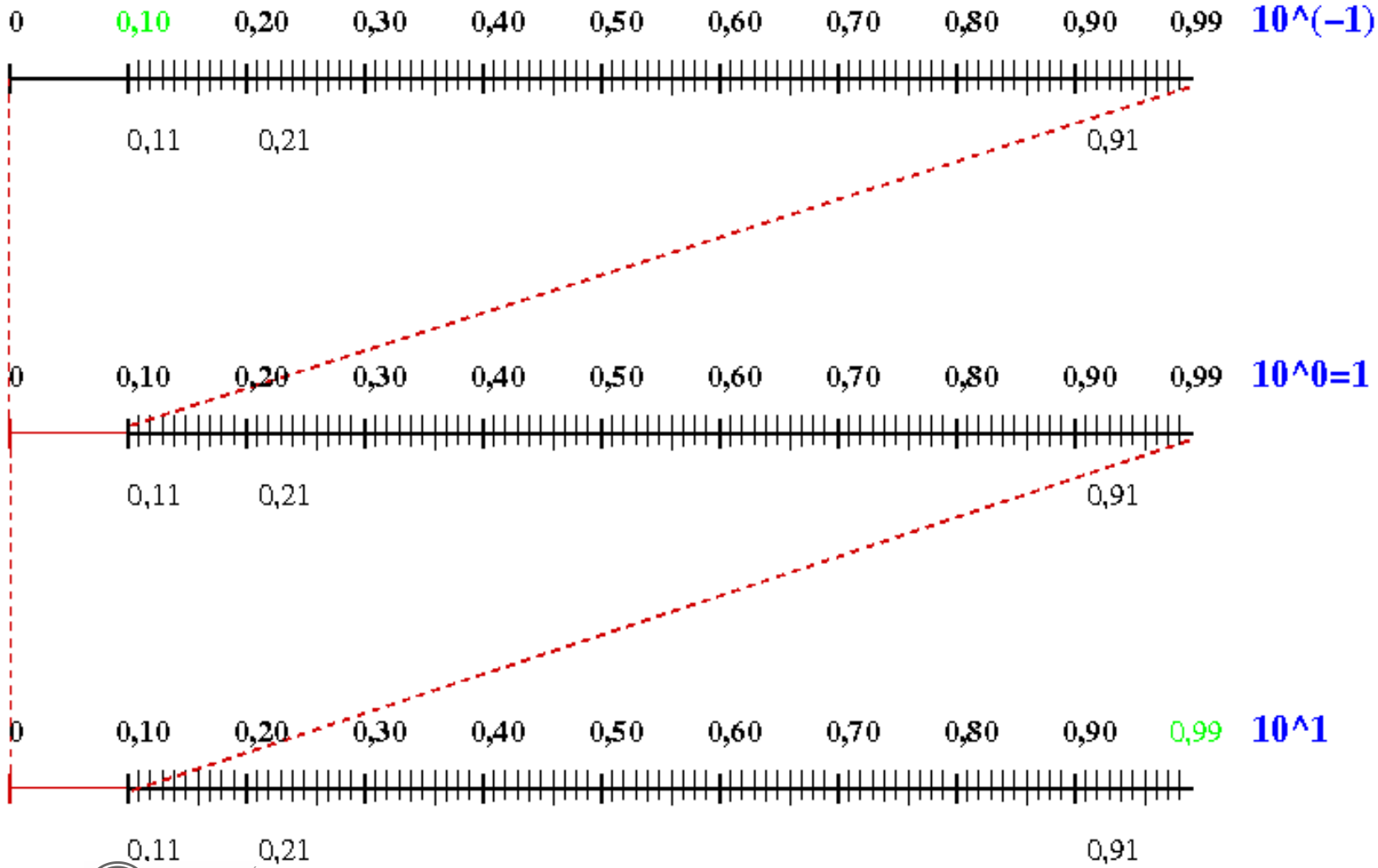
-	-	9	9	1	0	0

- Tous les réels de l'intervalle $[-0{,}999 \times 10^{+99} ; 0{,}999 \times 10^{+99}]$ ne sont pas représentables (que $36 \cdot 10^4 + 1$).

Exemple en base dix (2)

Représentation avec mantisse de 2 chiffres décimaux ($p = 2$) et l'exposant $e \in \{-1, 0, 1\}$. Nombres strict. positifs de

0,01 : | + | - | 1 | 1 | 0 | à 9,90 : | + | + | 1 | 9 | 9 |



Problèmes de la représentation en virgule flottante

- On ne peut pas représenter des réels plus grands que 9,9 .
  Une opération ayant comme résultat un tel nombre engendre un dépassement de capacité ou **overflow**.
- On ne peut représenter des réels $x \in \mathbb{R}$ pour $0 < x < 0{,}010$.
  Une opération ayant comme résultat un tel nombre engendre un soupassement de capacité ou **underflow**.
- De l'intervalle $[0\,;\,9,9]$ on ne représente que 271 nombres réels.
  Ces valeurs représentées ne sont pas distribuées de façon uniforme.

  Une opération ayant comme résultat un nombre x non représentable engendre une **erreur d'arrondi** :
  on doit approximer le "vrai" résultat x par un réel $\tilde{x}$ représentable dans le système virgule flottante choisi.

Problèmes de la représentation en virgule flottante

Exemples : base 10, $p = 2$ et $e \in \{-1, 0, 1\}$.

- Les nombres 3 et 7 sont représentables :
  $3 = 0{,}30\ 10^1$; $7 = 0{,}70\ 10^1$.
  Mais le résultat de $3 + 7 = 10 = 0{,}10\ 10^2$ n'est plus représentable, d'où **overflow**.

- Les nombres $0{,}010 = 0{,}10\ 10^{-1}$ et $0{,}011 = 0{,}11\ 10^{-1}$ sont représentables, mais leur différence $0{,}011 - 0{,}010 = 0{,}001 = 0{,}10\ 10^{-2}$ ne l'est pas.

  De même $0{,}010/2 = 0{,}005 < 0{,}010$ n'est pas représentable.
  On a donc affaire à un **underflow**.

- Si on relâche la condition que le digit de plus fort poids soit non nul, on peut représenter ces nombres :
  $0{,}001 = 0{,}01\ 10^{-1}$ et $0{,}005 = 0{,}05\ 10^{-1}$
  Ces nombres "sous-normaux" (**subnormal**) sont utilisés pour représenter des quantités très petites mais non nulles.

Problèmes de la représentation en virgule flottante

Exemples **erreurs d'arrondi** : base 10, $p = 2$ et $e \in \{-1, 0, 1\}$.

- Le résultat de l'opération $1{,}0 - 0{,}011 = 0{,}989$ n'est pas représentable, et sera arrondi vers $0{,}99 = 0{,}99\ 10^0$
- De même, $5 + 0{,}09 = 0{,}509\ 10^1$ sera arrondi vers $0{,}51\ 10^1$.
- Soit $a = -9$, $b = 9$ et $c = 0{,}011$ et $d = a + b + c$:

$$d = (a + b) + c = 0{,}011 \neq a + (b + c) = 0$$

En effet, $b + c = 9{,}011 = 0{,}9011\ 10^1$ est non représentable et est arrondi vers $0{,}90\ 10^1 = b$.

L'addition des nombres en virgule flottante n'est pas associative !

- Prévoir le résultat de

$$\left(\left(\frac{1}{3}\right) * 3\right) - 1\ .$$

Bilan : représentation en virgule flottante

Soit la représentation en virgule flottante en base b :

$$(-1)^s \, 0, d_{-1} d_{-2} \ldots d_{-(p-1)} d_{-p} \cdot b^e$$

où $e_m \leq e \leq e_M$, $d_i \in \{0, 1, \ldots, b-1\}$ et $d_{-1} \neq 0$.

1. Les nombres à virgule flottante sont un sous-ensemble fini de $\mathbb{R}$ qui n'est pas stable pour les opérations arithmétiques.
2. Même si p et $e_M - e_m$ sont grands :
   $\rightarrow$ Les nombres en virgule flottante ne sont pas répartis de façon uniforme.
   $\rightarrow$ Il y aura toujours des overflows, underflows et erreurs d'arrondi.
3. Le nombre $\varepsilon = b^{1-p}$ est tel que entre 1 et $1 + \varepsilon$ aucun réel n'est représentable.
   Ce nombre (**précision machine**) sert à majorer les erreurs d'arrondi et d'approximation sur un système donné.

Exemple de code C

```
#include <stdio.h>
 int main( int argc, char **argv )
 {
    float machEps = 1.0f;

    do {
       machEps /= 2.0f;
    }
    while ((float)(1.0 + (machEps/2.0)) != 1.0);

    printf( "\n Epsilon  machine = %G\n", machEps );
    return 0;
 }
```

On trouve comme résultat `1.19209E-07`$= 1{,}192\ 10^{-7} \sim 2^{-23}$.

On verra que cette valeur “expérimentale” de la précision machine correspond bien au format `float` du langage `C`.

Réels en virgule flottante en base 2

- Pour le codage de nombres en virgule flottante en binaire, on peut apporter quelques améliorations, présentées dans la suite.
- Le standard **IEEE 754-2008** fixe, entre autres, comment représenter des nombres flottants en simple (4 octets) et double (8 octets) précision.

  En langage `C` ceci correspond aux formats `float` et `double`.

  IEEE = Institute of Electrical and Electronics Engineers
  (association professionnelle internationale, de droit américain)
- Ce standard ne fixe pas que les formats, mais aussi :
  les modes d'arrondi, les calculs avec les nombres flottants, des valeurs particulières, la détection de problèmes (overflow, ...).
- On va s'intéresser à deux points concernant le format :
  les exposants biaisés et le bit implicite.

Réels en virgule flottante en base 2

On veut représenter les nombres en virgule flottante sur une machine suivant le format

signe mantisse 1 bit	exposant 4 bits	mantisse normalisée 8 bits

- Par exemple $(0{,}015)_8 = (0{,}000001101)_2 = (0{,}1101)_2 * (2^{-5})_{10}$;
- Mantisse positive, donc bit de signe égal à 0 ;
- Mantisse normalisée $(0,1101)_2$ avec exposant $(-5)_{10}$
- Codage en complément à deux sur 4 bits :
  $(5)_{10} = (0101)_2$, d'où le code CA2 : 1011
- Représentation du nombre :

0	1011	11010000

- Un comparateur logique, opérant bit à bit et de gauche à droite, ne peut pas facilement comparer des nombres (car il n'est pas pratique de comparer les nombres en CA2)

0	1011	11010000
0	0011	11010000

Exposant biaisé

- Afin de simplifier les comparaison sur les mots mémoire, les exposants signés sont codés grâce à un décalage.
- En code *CA*2, sur k bits, on représente les entiers signés :

$$-2^{k-1} \leq n \leq 2^{k-1} - 1$$

- En ajoutant 2^{k-1}, on translate l'intervalle sur

$$0 \leq n + 2^{k-1} \leq 2^k - 1$$

- Le nombre 0 est codé par 2^{k-1} : $\boxed{1}\underbrace{\boxed{0}\cdots\boxed{0}}_{k-1 \text{ fois}}$

-2^{k-1} est codé par $\underbrace{\boxed{0}\boxed{0}\cdots\boxed{0}}_{k \text{ fois}}$ et $2^{k-1} - 1$ par $\underbrace{\boxed{1}\boxed{1}\cdots\boxed{1}}_{k \text{ fois}}$

- La valeur 2^{k-1} s'appelle le biais ou le décalage
  En anglais, on parle de **bias**, **offset** ou **excess**.
- En pratique, il suffit de changer le bit de signe

Exposant biaisé

- Si $0 \leq c(n) \leq 2^k - 1$, alors il code l'entier signé $n = c(n) - 2^{k-1}$.
  Si $-2^{k-1} \leq n \leq 2^{k-1} - 1$, alors son code est $c(n) = n + 2^{k-1}$.
- Ce codage est souvent utilisé dans des circuits qui ne peuvent pas traiter des nombres positifs et négatifs.
  Par exemple en traitement du signal (DSP).
- Pour un codage sur $k = 4$ bits et un biais de $2^{4-1} = 8$:

		CA2	biaisé			*CA2*	biaisé
0	=	0 0 0 0	1 0 0 0	-8	=	1 0 0 0	0 0 0 0
1	=	0 0 0 1	1 0 0 1	-7	=	1 0 0 1	0 0 0 1
2	=	0 0 1 0	1 0 1 0	-6	=	1 0 1 0	0 0 1 0
3	=	0 0 1 1	1 0 1 1	-5	=	1 0 1 1	0 0 1 1
4	=	0 1 0 0	1 1 0 0	-4	=	1 1 0 0	0 1 0 0
5	=	0 1 0 1	1 1 0 1	-3	=	1 1 0 1	0 1 0 1
6	=	0 1 1 0	1 1 1 0	-2	=	1 1 1 0	0 1 1 0
7	=	0 1 1 1	1 1 1 1	-1	=	1 1 1 1	0 1 1 1

Exposant biaisé : exemple de codage

On veut représenter les nombres en virgule flottante sur une machine suivant le format

signe mantisse 1 bit	exposant 4 bits	mantisse normalisée 8 bits

- Encore l'exemple
  $(0{,}015)_8 = (0{,}000001101)_2 = (0{,}1101)_2 * (2^{-5})_{10}$;
- Mantisse positive, donc bit de signe égal à 0 ;
- Mantisse normalisée $(0{,}1101)_2$ avec exposant $(-5)_{10}$
- Codage biaisé de l'exposant sur 4 bits :
  le biais est $2^{4-1} = 8$, l'exposant biaisé est $-5 + 8 = 3_{10}$,
  écrit en binaire sur 4 bits, l'exposant est codé par $(0011)_2$.
- Représentation du nombre :

0	0011	11010000

Intêret de l'exposant biaisé

- Un comparateur logique, opérant bit à bit et de gauche à droite, peut facilement comparer les nombres :

0	0011	11010000
0	1011	11010000

- Il suffit d'aller de gauche à droite, le premier nombre qui à un 1 quand l'autre a un 0 est le plus grand (à part pour le signe).

0	0010	11010000	
0	0011	11010000	est plus grand

0	0011	11010100	est plus grand
0	0011	11010000	

- Faire des additions en CA2 n'a plus de sens avec des codes biaisés, c'est uniquement pour le codage et la comparaison

Exposant biaisé : codage

- Sur k bits, pour coder un entier signé, on ajoute le biais 2^{k-1} et l'on représente le résultat en binaire naturel
- Inversement, étant donné un code sur k bits, on transforme le code binaire naturel non signé en décimal et l'on retranche le biais 2^{k-1}.
- Sur 3 bits, le biais est $2^{3-1} = +4$

-4	s'écrit 000	car	-4+4=0	et le code 000	représente 0-4=-4
-3	s'écrit 001	car	-3+4=1	et le code 001	représente 1-4=-3
-2	s'écrit 010	car	-2+4=2	et le code 010	représente 2-4=-2
-1	s'écrit 011	car	-1+4=3	et le code 011	représente 3-4=-1
0	s'écrit 100	car	0+4=4	et le code 100	représente 4-4= 0
1	s'écrit 101	car	1+4=5	et le code 101	représente 5-4= 1
2	s'écrit 110	car	2+4=6	et le code 110	représente 6-4= 2
3	s'écrit 111	car	3+4=7	et le code 111	représente 7-4= 3

Procédé du bit caché

- En base 2, la représentation normalisée de la mantisse commence toujours par 1.
- Le bit le plus significatif ne représente aucune information : on peut donc supposer sa présence de façon **implicite**, c'est-à -dire sans le coder !
- Sur un mot de p bits, on gagne ainsi un bit pour avoir une mantisse de $p+1$ **bits significatifs**.
  On **double** les mantisses représentables.
- Attention : on peut maintenant avoir une mantisse ne contenant que des zéros !
- Attention : en base $b > 2$, la mantisse normalisée commence par $d_{-1} \in \{1, 2, \ldots, b-1\}$.

  Le procédé du bit caché ne s'applique pas !

Procédé du bit caché

Exemple :

- Mantisse sur $p = 3$ bits
- Avec la contrainte du bit de poids fort non nul, nous n'avons que 4 mantisses distinctes :

  $100 = (0,5)_{10}, 101 = (0,625)_{10}, 110 = (0,75)_{10}, 111 = (0,875)_{10}$.
- Si l'on suppose qu'il y a un bit caché qui vaut 1, la mantisse contient 3 bits, mais l'on code une mantisse normalisée de 4 bits.
- Il y a maintenant 8 mantisses possibles :

1	000	soit $(0{,}5000)_{10}$;	1	100	soit $(0{,}7500)_{10}$;
1	001	soit $(0{,}5625)_{10}$;	1	101	soit $(0{,}8125)_{10}$;
1	010	soit $(0{,}6250)_{10}$;	1	110	soit $(0{,}8750)_{10}$;
1	011	soit $(0{,}6875)_{10}$;	1	111	soit $(0{,}9375)_{10}$.

- La précision augmente sans frais !

Bit caché et exposant biaisé. Exemple

Représentation comprenant :
un bit de signe, un exposant biaisé de 3 bits et une mantisse de 3 bits, avec utilisation du bit caché. Pour trouver le successeur d'un nombre, il suffit de prendre tout le code et d'ajouter 1 en binaire.

Code	Valeur binaire (notation sc.)	Valeur binaire (notation à virgule)	Valeur décimale
0 110 000	$0{,}1000.10^{10}$	10,00	2,00
0 110 001	$0{,}1001.10^{10}$	10,01	2,25
0 110 010	$0{,}1010.10^{10}$	10,10	2,50
0 110 011	$0{,}1011.10^{10}$	10,11	2,75
0 110 100	$0{,}1100.10^{10}$	11,00	3,00
0 110 101	$0{,}1101.10^{10}$	11,01	3,25
0 110 110	$0{,}1110.10^{10}$	11,10	3,50
0 110 111	$0{,}1111.10^{10}$	11,11	3,75
0 111 000	$0{,}1000.10^{11}$	100,0	4,00
0 111 001	$0{,}1001.10^{11}$	100,1	4,50
0 1	$0{,}1010.10^{11}$	101,0	5,00

Bilan : bit caché et exposant biaisé

Avantages du codage utilisant un bit caché et un exposant biaisé :

1. on peut facilement comparer deux nombres ;
2. on obtient le successeur d'un nombre en ajoutant simplement 1 au code ;
3. on augmente la précision de la mantisse sans coût matériel.

Format flottant IEEE

1. Le standard IEEE utilise le bit caché pour la mantisse : il est de poids 2^0 et la mantisse est donc $1, d_{-1} \cdots d_{-p}$;
2. Le décalage/biais de l'exposant sur k bits est de $2^{k-1} - 1$ et on ne représente que des exposants signés de $-2^{k-1} + 2$ à $2^{k-1} - 1$. Pour $k = 8$, biais égal à 127 et les exposants vont de -126 à +127.
3. Les exposants biaisés 0 et $2^k - 1$ sont utilisés pour représenter des nombres sous-normaux, l'infini `Inf` et un nombre non défini `NaN`
4. Résumé format IEEE 754 $\boxed{s}\boxed{e}\boxed{d_{-1}}\boxed{d_{-2}}\cdots\boxed{d_{-p}}$

Nom	Taille	Signe s	Exposant e	Biais	Mantisse	Précision $p+1$	Chiffres significatifs
(float)	32 bits	1 bit	8 bits	127	23 bits	24	7
(double)	64 bits	1 bit	11 bits	1023	52 bits	54	16

Exemple de code C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
  float somme= 0.0f, inc= 0.1f;  int i;

  printf("\n Taille de  float : %d octet(s) \n",sizeof(float));

  for ( i=1; i<=10; ++i ) somme+= inc ;
  if( somme == 1 )
    printf("\n    10*0.1 = 1  \n");
  else
    printf("\n    10*0.1 <> 1 \n");

  printf("\n somme = %f\n",somme);      /* Format par défaut */
  printf("\n somme = %.15g,  inc = %.15g \n\n",somme,inc);
                                       /* Format étendu     */
}
```

Exemple de code C : affichage des résultats

```
Taille de  float      : 4 octet(s)

   10*0.1 <> 1

 somme = 1.000000

 somme = 1.00000011920929,  inc = 0.100000001490116
```

- Ce n'est donc pas une bonne idée de faire des test d'égalité entre des nombres en virgule flottante.
- Explication : on a $(0{,}1)_{10} = (0.0001100110\underline{0011}\ldots)_2$
  Pour coder $(0{,}1)_{10}$ en virgule flottante, avec une mantisse en binaire, on est obligé de faire une approximation.
- Il vaut mieux faire un test d'inégalité :

  `if( fabs(somme-1)< epsilon )`

  où `epsilon` est la précision, par exemple 10^{-6} .

Conclusion

- La représentation des réels en machine nécessite de choisir la taille mémoire :
  souvent 4 octets ou 8 octets, des fois 16 octets.
- Les nombres réels représentables en machine sont en nombre fini, ils constituent un ensemble discret.
- Les calculs en flottant peuvent provoquer des **underflow**, **overflow** et **erreurs d'arrondi**.
  Certains standards permettent de gérer des **exceptions** :
  `Inf`, `NaN`, nombres sous-normaux,...
- On mesure la puissance d'une unité de calcul en virgule flottante en **FLOPS** (en anglais, FLoating point Operations Per Second).
  En juin 2013, l'ordinateur Tianhe-2 de la NUDT, Chine, a effectué $33,86$ petaFLOPS $= 33,86 \times 10^{15}$ FLOPS contre 10^{10} FLOPS pour un processeur "normal" à $2.5 - GHz$.

Propagation d'erreurs - météorologie

- Nate Silver p.... ? Tel Prix Nobel a essayé un modèle pour prévoir la météo sur une période de quelques heures
- Après avoir fait deux fois la même simulation, il s'aperçoit qu'il a des résultats complètements différents
- Il réalise que la seconde fois, l'un des tehcnicients a tronqué les données après la troisièmle décimale.
  Exemple : La pression a pu passer de $1,23668923$ à $1,237$, et il en a résulté une grande différence de résultat
- L'erreur s'est propagée de manière "exponentielle"
- "Théorie du chaos" ?

Propagation d'erreurs

- La manipulation de nombres dans un langage de programmation entraîne souvent des erreurs liées à l'arrondi
- Lorsque l'on effectue des opérations entre nombres arrondis, on obtient un résultat dont l'erreur est en général plus importante.
- Exemple : Addition.
  - Soit x un nombre dont on connaît une valeur approchée x_0, à un nombre Δ près
  - Δ est appelée "incertitude" - elle peut venir du système de stockage, ou d'une mesure externe imprécise, ou autre...
  - On note $x = x_0 \pm \Delta$
  - Exemple $\pi = 3,14 \pm \Delta$ avec $\Delta = 0,01$ ($\pi = 3.14159265359...$)
  - Alors on ne connaît $\pi + \pi$ qu'à 2Δ près : $\pi + \pi = 6,28 \pm 0,02$

Propagation d'erreurs (II)

- Lors de l'addition/soustraction, on additionne les incertitudes (mais on ne les soustrait pas !)
- $\boxed{\text{Si } x = x_0 \pm \Delta \text{ et } y = y_0 \pm \Delta', \; x - y = x_0 - y_0 \pm (\Delta + \Delta')}$
- Lors de la multiplication, la formule est la suivante :

  $\boxed{xy = x_0 y_0 \pm (|x_0|\Delta' + |y_0|\Delta)}$
- Cette formule ressemble à la formule de dérivation du produit de fonctions : $(fg)' = f'g + fg'$.
- **Exemple** : Si $x = 2 \pm 0,01$, $y = -3 \pm 0,2$, alors $xy = -6 \pm \Delta$ avec $\Delta = 2 \cdot 0,2 + 3 \cdot 0,01 = 0,43$
- **Preuve :**

$$|xy - x_0 y_0| = \left|(x_0 + \Delta)(y_0 + \Delta') - xy\right|$$

$$= \left|\underline{x_0 y_0} + x_0\Delta' + y_0\Delta + \Delta\Delta' - \underline{x_0 y_0}\right| \leqslant |x_0|\Delta' + |y_0|\Delta + \Delta\Delta'$$

et on considère que $\Delta\Delta'$ est "d'ordre 2", donc négligeable (ici : $\Delta\Delta' = 0,2 \cdot 0,01 = 0,002$).

Propagation d'erreurs (III)

- **Exemple :** On a $x = 3/4 \pm 0,08$, $y = 10 \pm 0,1$, $z = -3 \pm 0,01$. Calculer l'incertitude sur $x(y+z)$.
-

$$y + z = 7 \pm (0,01 + 0,1) = 7 \pm 0,11$$

donc

$$\begin{aligned} x(y+z) &= (3/4 \pm 0,08)(7 \pm 0,11) \\ &= 21/4 \pm (7 \cdot 0,08 + 0,11 \cdot 3/4) = 5,25 \pm 0,6425. \end{aligned}$$

En effet, si on a le pire cas de figure
$x = 0,75 + 0,08$, $y = 10,1$ et $z = -3 + 0,01$,

$$x(y+z) = 5,8266.$$

Addition sur grandes sommes

- Soit $n = 1$ million, $x_1, \ldots, x_n$ des données pour lesquelles on a une incertitude de Δ
- Soit

$$S = \sum_{i=1}^{n} x_i$$

- Alors l'incertitude sur S est de $n\Delta$.
- Soit

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

- Alors l'incertitude sur $\bar{x}$ est $n \cdot \frac{1}{n}\Delta = \Delta$
- Autrement dit, l'incertitude ne croît pas lors du passage à la moyenne.

Calculer sans retenue

- Rappel : pour représenter un nombre en base b, on utilise l'alphabet $\{0, 1, \ldots, b-1\}$.
  On faisant la somme de deux chiffres, on est en général obligé d'utiliser le principe de position pour représenter le résultat.
  Exemples : en base $b = 10$, on a $7 + 8 = 15$,
  en base $b = 2$, $1 + 1 = 10$.
  En utilisant le 0 et le principe de position, on peut noter n'importe quel entier grâce aux chiffres $\{0, 1, \ldots, b-1\}$.
- Mais dans certaines applications, il faut tenir compte de la périodicité de ce qui est représenté. On compte
  1. les heures de la journée de 1h à 24h ;
  2. les jours de la semaine de lundi (1) à dimanche (7) ;
  3. les angles sont mesurés de 0° à 360° ;
  4. le signal d'un phare passe de façon périodique toutes les x sec. ;
  5. une roue mécanique à m dents a fait un tour entier quand les m dents se retrouvent dans leur position initiale ;

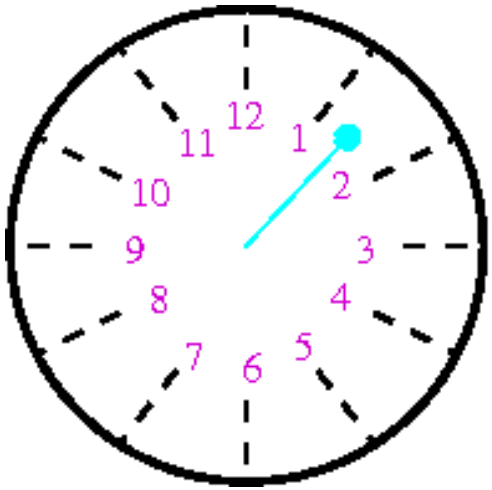
  :

Calcul modulaire

- Pour formaliser ces problèmes, on utilise la théorie de nombres, plus précisément l'arithmétique modulaire, encore appelée le calcul modulaire.
- En Europe ce sont Euclide (approx. de -325 à -265) et Diophante d'Alexandrie (approx. de 210 à 284) et en Chine Sun Zi (vers 300) et Qin Jiushao (approx. de 1202 à 1261), qui étudient des problèmes de ce type.
- Applications aujourd'hui :
  Calcul de clés de contrôle, codes correcteurs, cryptographie, ...
- Définitions : soient a et b deux entiers relatifs, on dit que
  1. a divise b s'il existe $k \in \mathbb{Z}$ tel que $b = ka$. On note $a|b$.
  2. a et b sont premiers entre eux si leur plus grand commun diviseur est 1. On note $\mathrm{pgcd}(a, b) = 1$;

Calcul modulaire : exemple

Horloge avec une aiguille et 12 graduations de 1 à 12 :



- Ici 12 est identique à 0h et 24h,
  tandis que 18h est identifié à 6h,...
- De façon générale, pour $r \in \{1, \ldots, 12\}$,
  r représente les entiers de la forme $n = r + 12k$ où $k \in \mathbb{Z}$.
- En remplaçant 12 par 0, donc pour $r \in \{0, 1, \ldots, 11\}$,
  on peut dire que r représente les entiers n tels que le reste de la division euclidienne de n par 12 est r. C'est l'ensemble

$$\{\ldots, r-48, r-36, r-24, r-12, r, r+12, r+24, r+36, r+48, \ldots\}$$

Calcul modulaire : propriétés

Soit $p \in \mathbb{N} \setminus \{0, 1\}$, alors

1. a et b sont congrus modulo p si et seulement si $a = b + pk$ pour un certain entier k, si et seulement si $a - b$ est un multiple de p, c'est-à -dire si $p|(a - b)$.
   On note $a \equiv b \mod p$.
2. On a $a \equiv b \mod p$ si et seulement si
   a et b ont le même reste dans la division euclidienne par p.
   Soit r ce reste, alors $a = kp + r$ pour $k \in \mathbb{Z}$ et $r \equiv a \mod p$.
3. Si $a \equiv b \mod p$ et si $b \equiv c \mod p$, alors $a \equiv c \mod p$.
4. Si $a_1 \equiv b_1 \mod p$ et $a_2 \equiv b_2 \mod p$, alors
   $a_1 + a_2 \equiv b_1 + b_2 \mod p$, $a_1 a_2 \equiv b_1 b_2 \mod p$,
   et pour tout entier $n \in \mathbb{Z}$, $na \equiv nb \mod p$.
5. Tout entier $m \in \mathbb{Z}$ a un représentant dans $F_p = \{0, 1, \ldots, p - 1\}$.
   Les calculs modulo p sur les entiers peuvent ainsi se ramener aux calculs modulo p dans F_p.

Exemples

$$12 + 7 \equiv 3 + 7 \equiv 10 \equiv 1 \mod 9$$

$$12 * 7 \equiv 3 * 7 \equiv 21 \equiv 3 \mod 9 \text{ car } 21 = 2 * 9 + 3$$

$$2000 \equiv 20 * 100 \equiv 20 * 20 * 5 \equiv 2 * 2 * 5 \equiv 20 \equiv 2 \mod 6$$

Application : Il est 15h et ma montre sonne toutes les heures. Dans 17240 minutes, combien de temps devrais-je attendre pour que ma montre sonne ? On fait les calculs modulo 60 :

$$\begin{aligned} 17240 &\equiv 1724 * 10 \equiv 862 * 2 * 10 \equiv 2 * 2 * 431 * 10 \\ &\equiv 2 * 2 * (240 + 191) * 10 \equiv 2 * 2 * 191 * 10 \\ &\equiv 2 * 2 * (180 + 11) * 10 \equiv 2 * 2 * 11 * 10 \equiv 4 * 110 \\ &\equiv 4 * 50 \equiv 200 \equiv 20 \mod 60 \end{aligned}$$

Il faudra donc attendre $60 - 20 = 40$ minutes.

Application 2 : Jean est né un mardi. Jacques est né 4 ans et 212 jours plus tard. Quel jour est né Jacques ?

$$3 * 365 + 366 + 212 \equiv ? \mod 7$$

On a $350 = 7 * 50$. Donc $365 \equiv 350 + 15 \equiv 15$ donc $366 \equiv 16$. Donc, comme $210 = 3 * 70$,

$$3 * 365 + 366 + 212 \equiv 3 * 15 + 16 + 210 + 2 \equiv 3 * 1 + 2 + 2 \equiv 0$$

donc Jacques est né un mardi aussi.

Calcul modulaire

- Exemple : calculer $x \equiv 23^6 \mod 7$ et $y \equiv 233^6 \mod 7$, or $23^6 = 148035889 = 7 * 21147984 + 1$ et $233^6 = 16000726539569$! mais modulo 7 on a : $23^6 \equiv 2^6 = 2^3 \cdot 2^3 \equiv 1 \cdot 1$
- On représente les opérations dans F_p grâce à des tableaux :
  - Pour $p = 2$, $F_2 = \{0, 1\}$ et

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

  - Pour $p = 3$, $F_3 = \{0, 1, 2\}$ et

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

*	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

  - Pour $p = 4$, $F_4 = \{0, 1, 2, 3\}$ et

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Inverse et opposé

- On appelle **opposé** d'un nombre x dans F_p le nombre y tel que $x + y \equiv 0$. On note aussi $y \equiv -x \mod p$.
  Exemple : $-5 = 3 \mod 8$
- On appelle **inverse** d'un nombre x modulo p un nombre y tel que $x \cdot y \equiv 1 \mod p$
  Exemple : $3^{-1} = 2 \mod 5$ car $2 \cdot 3 = 6 \equiv 1 \mod 5$
- **Un nombre n'a pas toujours d'inverse !**
  Exemple : 2 n'a pas d'inverse modulo 4, car $2x$ est toujours pair, et ne peut donc pas être égal à 1 modulo 4.
- Si p est un nombre premier, tous les éléments de F_p ont un inverse, à part 0.

Petit théorème de Fermat

Theorem

Soit p un nombre premier. Tout nombre x vérifie $x^{p-1} = 1$ dans F_p, c'est-à-dire

$$x^{p-1} \equiv 1 \text{ modulo } p.$$

- Dans F_3 : $2^2 = 4 \equiv 1$
- Dans F_5 : $2^4 = 16 \equiv 1$
- Dans F_7 : $2^6 = 64 = 63 + 1 \equiv 1$
- Dans F_7 : $6^6 \equiv (-1)^6 \equiv 1$
- Dans F_7 : $4^6 \equiv 2^6 \cdot 2^6 \equiv 1 \cdot 1 \equiv 1$

Petit théorème de Fermat : exemples

- Dans F_{17} :

$$\begin{aligned} 2^{16} &= 65\,536 = 51\,000 + 14536 = (3 \cdot 17\,000) + 14536 \\ &\equiv 17\,000 - 2464 \equiv -2464 \equiv -3400 + 1000 - 64 \equiv 936 \\ &\equiv 680 + 256 = 2 * 340 + 170 + 86 \equiv 51 + 35 \equiv 34 + 1 \equiv 1 \end{aligned}$$

- Dans F_{17} : $9^{16} \equiv (-8)^{16} \equiv (-1)^{16} \cdot (2^{16})^3 \equiv 1 \cdot 1 \cdot 1 \equiv 1$
- La preuve du petit théorème de Fermat (1640) repose sur des principes d'arithmétique, et prend environ une demi-page
- A ne pas confondre avec le "grand théorème de Fermat", dont la preuve fait une centaine de pages, démontré en 2001 par Andrew Wiles ce qui lui a valu la médaille Fields (récompense suprême en mathématiques).

Preuve que $x^p = x$ (hors-programme)

- Comme p est premier, $x^{p-1} \equiv 1 \Leftrightarrow x^p \equiv x$.
- Vrai pour $x = 1$: $x^p = 1^p = 1 = x$
- Vrai pour $x = 2$?
  $2^p = (1+1)^p = 1 + C_p^1 2^{p-1} + C_p^2 2^{p-2} + \ldots + C_p^{p-1} 2 + 1$
- Or $C_p^k = \frac{p!}{k!(p-k)!} = \frac{p \cdot (p-1) \cdots (p-k+1) \cdot \cancel{(p-k) \ldots 3 \ldots 2}}{\underbrace{k}_{<p} \cdot \underbrace{(k-1)}_{<p} \cdots 2 \cdot 1 \cdot \cancel{p-k \ldots 2}}$
- Pour $1 < k < p$, C_p^k est un multiple de p
- Donc $2^p \equiv 1 + p \cdot (\ldots.) + 1 \equiv 2!$
- Idem : $3^p \equiv (2+1)^p = 2^p + C_p^1 2^{p-1} + \cdots + C_p^{p-1} 2 + 1 \equiv 2^p + p \cdot (\ldots) + 1 \equiv 2 + 1 \equiv 3$
- Par récurrence : pour tout k : $(k+1)^p \equiv k^p + 1 \equiv k + 1$

Calcul modulaire : clé de sécurité

- Pour éviter des erreurs lors de la saisie d'un numéro, comme un chiffre erroné ou la permutation de chiffres, on adjoint souvent une **clé de sécurité**.

- Exemple : numéro INSEE ou " numéro de sécurité sociale"
  Il est composé de 13 chiffres, $s = s_{12}s_{11}\dots s_0$,
  et permet de coder le sexe, l'année, le mois, le département de naissance,etc. d'un individu
  et d'une clé de sécurité $c(s)$ à deux chiffres calculé comme suit

$$c(s) = 97 - (s \mod 97) \in \{1, \dots, 97\}.$$

  Le calcul modulo le nombre premier 97 permet de détecter des erreurs de saisie dans s (cf. TD).

- D'autres exemples sont les codes barres EAN 13, le code ISBN,...

Codes asymétriques basés sur la décomposition de grand nombres : principe simplifié

- On choisit un nombre premier p très grand, et **secret**, beaucoup plus grand que le nombre de mots qu'on veut coder.
  Exemple : $p = 927662347$
- On choisit un nombre e premier avec $p-1$ dans $\{1, \ldots, p-1\}$ qui est **public** . C'est la **clé publique**
  Exemple : $e = 101$
- **Théorème** : Il existe un nombre d tel que $d \cdot e = k \cdot (p-1) + 1$, i.e. $de \equiv 1 \mod p-1$ (c'est l'inverse de d dans F_{p-1}). C'est la **clé privée**. On peut la trouver en essayant toutes les possibilités.
  Exemple : $d = 587825645$ **car**
  $101 \times 587825645 = 64 \times 927662346 + 1$

Codes asymétriques basés sur la décomposition de grand nombres : principe simplifié : codage et décodage

- Admettons que l'on veuille coder un message, qui est un nombre m dans $\{1, \ldots, p-1\}$
- On transmet le message codé $c(m) = m^e \mod p$. Bien que e soit public, quelqu'un qui intercepterait le message ne pourrait pas en déduire m car il faut connaître p pour ça. **Exemple :** $m = 100\,000, c(m) = 100\,000^{101} \equiv 54446622 \mod F_p$ (se calcule en 101 étapes=rapide pour un ordinateur)
- Pour décoder le message, il faut calculer $c(m)^d$. En effet, d'après le petit théorème de Fermat,

$$c(m)^d = (m^e)^d = m^{ed} = m^{(p-1)k+1} = (\underbrace{m^{p-1}}_{\equiv 1})^k m \equiv m.$$

Exemple : $54446622^{587825645} \equiv 100\,000 \mod F_p$

Problème du code précédent

- Pour coder un message, il faut connaître e et p. Il est donc facile d'en déduire d, et de ainsi décoder tous les messages
- Ca ne convient pas pour que n'importe qui puisse coder un message, mais seul le détenteur de la clé privée puisse le décoder.

Calcul modulaire : Algorithme RSA

- Proposé par Rivest, Shamir et Adleman en 1977.
- Algorithme de cryptographie asymétrique,
  très utilisé dans le commerce électronique (protocole SSL).
- C'est un algorithme à clé publique CPub pour chiffrer
  et à clé privée CPriv pour déchiffrer un message :
  - Alice engendre les clés CPub et CPriv ;
  - Alice envoie CPub à Bob ;
  - Bob utilise CPub pour coder son message **M** en **C(M)**,
    il envoie **C(M)** à Alice par un canal non sécurisé ;
  - N'importe quel nombre **M** peut être envoyé (pas seulement un grand nombre premier)
  - Seule Alice peut décoder **C(M)** grâce à CPriv, et retrouver **M**.

Calcul modulaire : Algorithme RSA

- Calcul des clés :
  1. Choisir deux nombres premiers distincts (et grands) p et q ;
  2. Calculer $n = pq$ et $\varphi = (p-1)(q-1)$;
  3. Choisir e, avec $1 < e < \varphi$ et $\text{pgcd}(e, \varphi) = 1$ (on admet) ;
  4. Déterminer d vérifiant $ed \equiv 1 \mod \varphi$ ($\Leftrightarrow \exists k \in \mathbb{Z}, ed = \varphi \cdot k + 1$)

  Alors CPub = (n, e) et CPriv = d
- Chiffrement du message :
  Si M est un entier à chiffrer, $0 \leq M < n$, alors $C(M) \equiv M^e \mod n$.
- Déchiffrement du message :
  Si $C(M)$ est un entier chiffré, alors

$$C(M)^d \equiv M$$

  Grâce ente autres au Petit théorème de Fermat.
- La force de RSA est qu'il n'existe actuellement pas d'algorithme rapide pour factoriser un entier n grand en ses facteurs.
  Un record datant de 2009 a permis de factoriser un nombre de 768 bit

Exemple simple

- On prend $p = 29, q = 37$ qui sont secrets.
- On a donc $n = pq = 1073$, ce nombre est public (mais il est difficile de retrouver p et q, c'est la toute l'astuce).
- On a $\varphi = (p-1)(q-1) = (29-1)(37-1) = 1008$, qui reste secret.
- On choisit un nombre e premier avec $\varphi = 1008$. Par exemple, $e = 71$. Il est public
- On calcule d tel que $d * e \equiv 1[\varphi]$. On a $d = 1079$ (Ca peut se faire facilement par un ordinateur **si on connait** φ).

Comment envoyer un message codé ? Par exemple 'HELLO', dont le code ASCII est $m = 7269767679$ (en réalité, 072069076076079). Comme on est dans un système basé sur n, on ne peut transmettre que des nombres $< n$. On découpe donc en plusieurs mots (et le récepteur recollera les mots décodés) : $726, 976, 767, 900$.

Exemple simple (II)

On les code en les élevant a la puissance 71 :

$$726^{71} \equiv 436 \mod 1073$$
$$976^{71} \equiv 822 \mod 1073$$
$$767^{71} \equiv 825 \mod 1073$$
$$900^{71} \equiv 552 \mod 1073$$

La clé publique est $(n, e) = (1073, 71)$.
Le récepteur reçoit le message $436, 822, 825, 552$. Que faire ?
Si il ne connait pas d :

- Il sait que d est le nombre tel que $d * e \equiv 1[\varphi]$.
- Comment calculer φ ? $\varphi = (p-1)(q-1)$, mais il faut retrouver p et q. A partir de $n = 1073$, ca n'est pas évident si n est très grand !

Exemple simple (III)

Si il connait d**,** il n'a qu'a élever les messages a la puissance d (modulo n). Ici $d = 1079$ car

$$71 * 1079 \equiv 1 \mod 1073 \ (\Leftrightarrow 71 * 1079 = 1 + \text{quotient} \times 1073) :$$

:

$$\begin{aligned} 436^{1079} &\equiv (726^{71})^{1079} \equiv 726^{71*1079} \\ &\equiv (726)^1 \times (726)^{\text{quotient}\times(p-1)(q-1)} \\ &\equiv (726)^1 \times \left(\underbrace{(726)^{\times(p-1)(q-1)}}_{\substack{\equiv 1 \mod pq \\ \text{(petit théorème de Fermat)}}} \right)^{\text{quotient}} \equiv 726 \end{aligned}$$

$$822^{1079} \equiv 976^{1+\text{quotient}*1008} \equiv 976 \mod 1073$$

$$825^{1079} \equiv 767 \mod 1073$$

$$552^{1079} \equiv 900 \mod 1073$$

Calcul modulaire : Factorisation RSA-768

Le nombre *RSA-768* a 232 décimales en base 10 et 768 bits en base 2. Il a été factorisé en 2009, après 31 mois de calculs avec plusieurs centaines de machines, en deux nombres premiers de 116 décimales.

RSA-768

$$
\begin{aligned}
= \quad & 123018668453011775513049495838496272077285356959533479219732245215172640050726365751874520219978646938995647494277406384592519255732630345373154826850791702612214291346167042921431160222124047927473779408066535141959745985690214341 3 \\
= \quad & 33478071698956898786044169848212690817704794983713768568912431388982883793878002287614711652531743087737814467999489 \\
& \times \\
& 36746043666799590428244633799627952632279158164343087642676032283815739666511279233373417143968102700927987363089 17
\end{aligned}
$$

Calcul modulaire : Factorisation RSA-768

Publication : Factorization of a 768-bit RSA modulus
in Cryptology ePrint Archive : Report 2010/006
par Kleinjung, Aoki, Franke, Lenstra, Thomé, Bos, Gaudry, Kruppa, Montgomery, Osvik, te Riele, Timofeev, Zimmermann.
Extrait :

> *Our computation required more than* 10^{20} *operations. With the equivalent of almost 2000 years of computing on a single core 2.2GHz AMD Opteron, on the order of* 2^{67} *instructions were carried out. The overall effort is sufficiently low that even for short-term protection of data of little value, 768-bit RSA moduli can no longer be recommended. This conclusion is the opposite of the one arrived at on [39], which is based on a hypothetical factoring effort of six months on 100 000 workstations, i.e., about two orders of magnitude more than we spent.*

Voir aussi `http://fr.wikipedia.org/wiki/RSA-768`

Attaques contre l'algorithme RSA

- Il est très dur de factoriser le nombre $n = pq$ et d'en déduire p et q, et la quantités de grands nombres premiers p et q est bien trop grande pour tous les tester. Par contre, les chercheurs en cryptographie ne connaissent pas tant de méthodes que ça pour générer p et q, donc une de ces méthodes devrait aboutir au bon résultat...
- Attaque par chronométrage : En mesurant le temps de calcul que met une machine à déchiffrer un message, on peut se faire une idée de la clé de déchiffrement
- On intercepte le message codé $C = M^e$, et si e est petit on peut retrouver M directement (Daniel Bleichenbacher, 1998, Suisse)

Autre application : Génération d'un nombre aléatoire

- Il est impossible pour un ordinateur de simplement "créér" un nombre aléatoire
- Remarque : C'est également impossible pour un humain, les nombres ne sont pas vraiment aléatoires. Si l'on demande à quelqu'un de donner une suite de 100 "0" et "1", elle n'aura pas les mêmes propriétés que l'aléatoire "pur" : pas assez de sequences de 5 "1" d'affilée, etc...

Tirage d'un nombre aléatoire entre 0 et 1

- On part d'un grand nombre premier. Au hasard...$p = 927662347$
- On procède de la manière suivante : On part d'une graine ($\texttt{seed} = 100\,000$) dans $\{1, \ldots, p-1\}$
- On élève cette graine à une grande puissance décidée à l'avance. $e = 101$
- On obtient un nombre "aléatoire" $x = 100\,000^{101} = 54446622$
- Le nombre aléatoire dans $[0, 1]$ est $x/p \in [0, 1]$
- Il ne peut pas vraiment prendre toutes les valeurs, mais de toutes façons le système de stockage n'accepte pas toutes les valeurs possibles non plus.
- Pour le prochain tirage, on redéfinit la graine comme le dernier nombre tiré : $\texttt{seed} = x$
- On ne tombera jamais sur "1" car le seul nombre $x \in F_p$ tel que $x^p \equiv 1 \mod p$ est $x = 1$.

Deuxième partie II

La logique

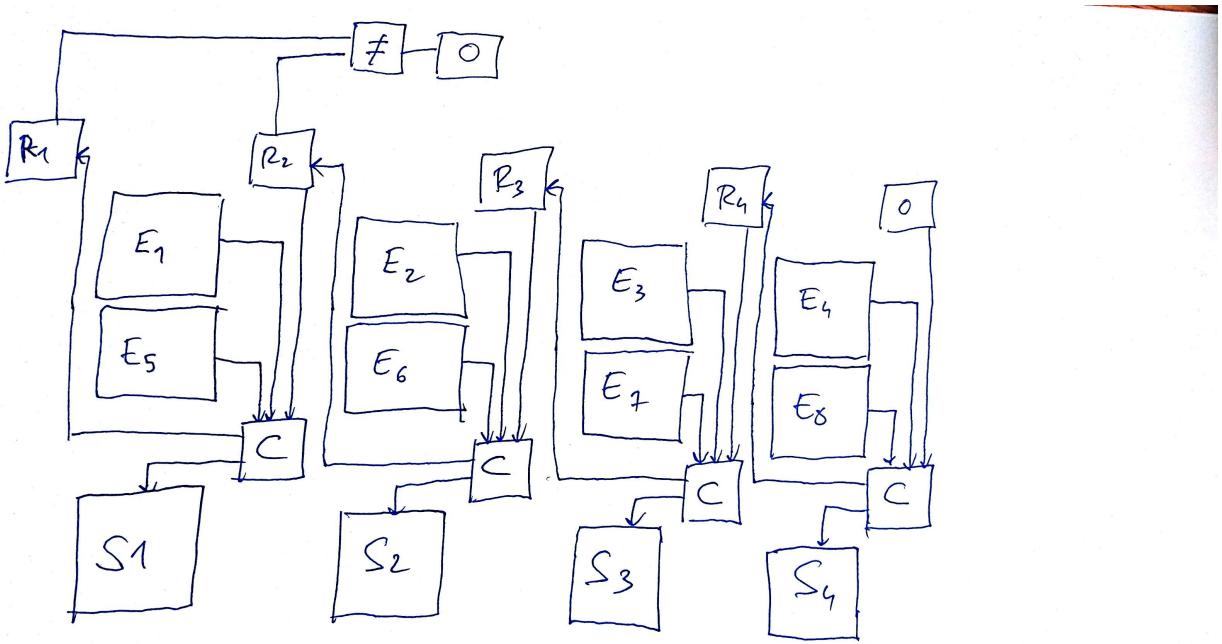
Introduction

- La logique est utile dans beaucoup de domaines :
  - Conception de circuits.
  - Vérification de preuves.
  - Preuves de programmes.
  - Programmation logique.
  - Simulation de raisonnements en intelligence artificielle.
  - Linguistique
  - ...
- Nous utiliserons le calcul des propositions :
  bien que limité, c'est la première étape dans la définition de la logique et du raisonnement.
- Le calcul des prédicats qui englobe le calcul des propositions et qui permet une formalisation achevée du raisonnement mathématique, sera très brièvement abordé.

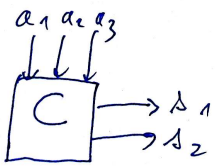
Exemple : Addition en $CA2_4$

Ecrivons un circuit logique qui donne le résultat et teste l'overflow pour une addition en CA2 sur 4 bits :

- Entrées : $E_1, E_2, E_3, E_4; E_5, E_6, E_7, E_8$ à valeurs dans $\{0, 1\}$
- Sorties : S_1, S_2, S_3, S_4, O (overflow), à valeurs dans $\{0, 1\}$



avec



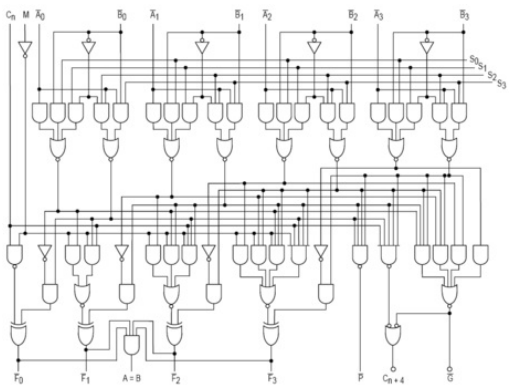
$S_1 = \begin{cases} 1 \text{ si 2 valeurs parmi } a_1, a_2, a_3 \text{ sont égales à 1} \\ 0 \text{ sinon} \end{cases}$

$S_2 = \begin{cases} 0 \text{ si exactement 2 valeurs sont égales à 1} \\ 1 \text{ sinon} \end{cases}$

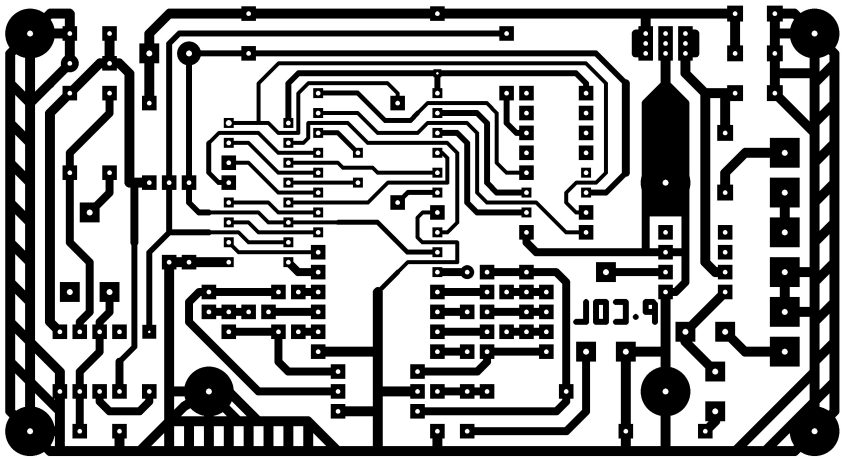
Du circuit logique au circuit imprmé

Pour fabriquer un dispositif physique qui teste l'overflow, il faut construire des composants informatiques capables de réaliser les opérations logiques :

- ET : Prend en entrée deux circuits électriques, et donne en sortie un courant électrique si les deux circuits d'entré sont alimentés, et pas de courant sinon.
- OU : Prend en entrée deux circuits électriques, et donne en sortie un courant électrique si l'un des deux circuits d'entrée est alimenté, et pas de courant sinon.
- NON, OU EXCLUSIF, etc...



$\Rightarrow$



Utilisation de la logique en maths

Comment écrire : “La fonction f n’a pas de limite en x” de manière simple avec des quantificateurs ? Il faut écrire le contraire de “f a une limite en x”, c’est-a-dire de :

$$\exists l \in \mathbb{R},\ \forall \varepsilon > 0,\ \exists \alpha > 0;\ \forall y \in [x-\alpha, x+\alpha], |f(y) - l| < \varepsilon.$$

Ça nous donne, en niant tous les quantificateurs :

$$\forall l \in \mathbb{R}; \exists \varepsilon > 0; \forall \alpha > 0; \exists y \in [x-\alpha, x+\alpha], |f(y) - l| \geqslant \varepsilon.$$

Introduction

- Une proposition est une affirmation du type ”il pleut” ou ”2+2=3” ; ou “π est un nombre rationnel.” On peut lui affecter une valeur de vérité : vrai ou faux.
- Un prédicat est une proposition dont la véracité dépend de variables :”f a une limite en x”,“x est rationnel”
- Une proposition ne contient ni des variables, ni des quantificateurs.
- En calcul des prédicats on utilise les quantificateurs : “Tout étudiant.e habite à Paris”, “il existe un élément de l’ensemble A qui est rationnel”.
- Toutes les phrases ne rentrent pas dans le système vrai/faux. Ca n’est pas le cas des phrases auto-référentes comme “cette phrase est fausse”, ou des phrases optatives comme “que la force soit avec vous !”. Par contre, “je souhaite que la force soit avec vous” est une proposition valide.

1. Le calcul des propositions traite du raisonnement sur les propositions. Il définit les règles de déduction qui relient les propositions, tout ceci **indépendamment de leur contenu**. De la même manière qu'on peut faire des opérations sur des fonctions mathématiques $f(x), g(x), h(x)$, indépendamment de leurs valeurs :

$$f * (g + h) = fg + fh.$$

2. On ne traite que des valeurs booléennes $\{$vrai, faux$\}$ ou $\{1,0\}$.
3. Dans le système de calcul des "prédicats", il y aura une notion de contexte qui sera importante. Par exemple, la valeur de "Je m'appelle Jean" dépend de la personne qui énonce cette phrase. La valeur de "Je suis en Europe", dépend de l'endroit où est énoncée cette phrase. Plus précisément, un prédicat est une proposition contenant des variables, comme "X est en Europe", où X est une variable d'entrée du système.

Syntaxe et sémantique

Dans les théories de la logique mathématique, en particulier en calcul des propositions, on considère deux aspects :

- La syntaxe, où l'on définit le langage du calcul des propositions par les règles d'écriture des formules.
- La sémantique qui détermine les règles d'interprétation des formules. On attribue des valeurs de vérité (vrai/faux) aux propositions élémentaires et on explique comment les connecteurs se comportent vis-à -vis de ces valeurs de vérité. On exprime souvent ce comportement par une table de vérité.

Exemple de syntaxe et sémantique

On considère les phrases dans une langue :

- La syntaxe fixe les règles d'écriture des phrases.

  PHRASE = (SUJET — VERBE — COMPLEMENT)
- La sémantique permet l'interprétation des phrases.
- Exemples :
  1. "le chat boit son lait"
  2. "le fermier conduit un troupeau"
  3. "le lait boit son chat"
  4. "un troupeau conduit le fermier"

  Une phrase dont la syntaxe est correcte, n'a pas nécessairement un sens
- De plus, toutes les phrases qui sont sémantiquement correctes ne rentrent pas dans le système vrai/faux. Ca n'est pas le cas des phrases auto-référentes comme "cette phrase est fausse".

Les constituants du langage

La syntaxe du calcul des propositions utilise

- Les **variables propositionnelles** ou **propositions atomiques**.
  Notées $p_1, p_2, \ldots$ ou p, q, r,...
- Les **opérateurs** ou **connecteurs**.
  Ils permettent la construction de propositions plus complexes.

$\neg$	non	négation
$\wedge$	et	conjonction
$\vee$	ou	disjonction
$\rightarrow$ ou $\Rightarrow$	implique	implication
$\leftrightarrow$ ou $\Leftrightarrow$	équivaut	équivalence

- La ponctuation “(“ et ”)”, les parenthèses permettent de lever les ambiguïtés.

Ces éléments constituent l’alphabet du calcul propositionnel (remarque : il est infini, car on peut construire une infinité de propositions !).

Les formules propositionnelles

Grâce à cet alphabet, on peut construire l'ensemble des mots qui est l'ensemble des suites finies d'éléments de l'alphabet.

L'ensemble des **formules** ou **expressions bien formées** du calcul des propositions est le plus petit ensemble de mots tel que

1. les variables propositionnelles, ou atomes, sont des formules ;
2. si A est une formule, alors $\neg A$ est une formule ;
3. si A et B sont des formules, alors $(A * B)$ est une formule, où $*$ est l'un des connecteurs binaires, $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

La proposition "$A \rightarrow B$" est une variable qui vaut 1 s'il est vrai que B est vrai lorsque A est vraie, mais elle ne veut pas dire que A implique B, ou que A et B sont toutes les deux vraies. De même, $\neg A$ ne veut pas dire que A est forcément faux, c'est une variable qui peut prendre les valeurs 0 ou 1.

Exemple

- Considérons la formule $(((\neg a \vee b) \wedge c) \rightarrow (\neg\neg a \wedge \neg b))$

  On omet en général les parenthèses extrêmes des formules, d'où

  $$((\neg a \vee b) \wedge c) \rightarrow (\neg\neg a \wedge \neg b)$$

- Vérification de la cohérence des parenthèses :
  On attribue un poids +1 à la parenthèse ouvrante, un poids -1 à la parenthèse fermante et 0 aux autres symboles.
  La somme des poids d'une formule est alors nulle.
  Attention : ceci ne suffit pas à garantir que la formule est syntaxiquement correcte.

- Les parenthèses sont importantes pour lever les ambiguïtés lorsque l'on utilise des connecteurs binaires.

  Exemple : comment interpréter $p \rightarrow q \rightarrow r$?
  Soit $(p \rightarrow q) \rightarrow r$, soit $p \rightarrow (q \rightarrow r)$.

Distribution de vérité

- Une distribution de vérité δ est une application de l'ensemble des variables propositionnelles dans l'ensemble des valeurs de vérité $\{0, 1\}$.

  Exemple : $\delta : \{a, b\} \longrightarrow \{0, 1\}$,
  or comme $\delta(a) = 0$ ou 1, et de même pour $\delta(b)$, il y a $2^2 = 4$ distributions de vérité possible :

a	0	0	1	1
b	0	1	0	1

- Une distribution δ étant fixée, on définit $\delta(F)$, ou val(F, δ), pour toute formule F, à partir des tables de vérité.

  Une distribution de vérité δ se prolonge ainsi en une application de l'ensemble des formules dans $\{0, 1\}$.

Tables de vérité des connecteurs

- On peut écrire $\delta(\neg a)$:

a	$\neg a$
1	0
0	1

- Les tables de vérité des connecteurs binaires sont

a	b	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \leftrightarrow b$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

- On peut ainsi donner le comportement associé à chaque formule :
  - $\neg a$ prend la valeur 1 si et seulement si a prend la valeur 0.
  - $a \rightarrow b$ prend la valeur 0 si et seulement si a prend la valeur 1 et b prend la valeur 0.

  ⋮

Sémantique : valeurs d'une formule

- Une distribution δ donnée est un modèle de F si $\delta(F) = 1$.

  Exemple : la distribution $\delta(a) = 1$ et $\delta(b) = 0$ est un modèle pour la formule $F = a \vee b$.

- Une formule F est une tautologie si pour toute distribution δ, on a $\delta(F) = 1$.

  On dit aussi que F est **valide**. On note $\models F$.

  Exemple : La formule $a \vee \neg a$ est une tautologie, donc $\models (a \vee \neg a)$

- Une formule F est une antilogie si pour toute distribution δ, on a $\delta(F) = 0$.

  On dit aussi que F est une **contradiction** ou **insatisfaisable**.

  Exemple : La formule $a \wedge \neg a$ est une antilogie.

Sémantique : valeurs d'une formule

- Si la formule F prend au moins une fois la valeur 1, on dit que F est **satisfaisable**.
- Deux formules F et G sont équivalentes
  si et seulement si pour toute distribution δ on a $\delta(F) = \delta(G)$.

  On note "$F \underline{eq}\ G$" la propriété
  "les formule F et G sont équivalentes".
- Exemple : $F = (a \vee b)$ et $G = \neg(\neg a \wedge \neg b)$. On a bien $F \underline{eq}\ G$.

- Attention : ne pas confondre "$\leftrightarrow$" et $\underline{eq}$.

  Le premier est un symbole du langage formel, ou un opérateur, au même titre que $\vee, \wedge, \neg$,

  le second un symbole du métalangage.

  Ainsi $(F \leftrightarrow G)$ est une formule,

  tandis que "$F \underline{eq}\ G$" énonce une propriété des formules F et G.

  Et "$F \underline{eq}\ G$" est un énoncé équivalent à $\models F \leftrightarrow G$.

- Dans l'analogie avec les fonctions mathématiques, $a \leftrightarrow b$ s'apparente avec l'égalité $a = b$, alors que l'équivalence $\underline{eq}$ s'apparente à l'égalité entre fonctions. Par exemple, les deux fonctions mathématiques sur $\{0, 1\}$

$$f(x, y) = \mathbf{1}_{\{x=y\}},$$
$$g(x, y) = \mathbf{1}_{\{x=1, y=1\}} + \mathbf{1}_{\{x=0, y=0\}}$$

  sont toujours égales, c'est-à-dire $f = g$. En logique, on écrirait ca

$$(x \leftrightarrow y) \underline{eq} ((x \wedge y) \vee (\neg x \wedge \neg y))$$

Exemple 1

Évaluons $A = ((\neg a \vee b) \wedge c) \rightarrow (\neg\neg a \wedge \neg b)$.

On pose $F = ((\neg a \vee b) \wedge c)$ et $G = (\neg\neg a \wedge \neg b)$.

a	b	c	$\neg a \vee b$	F	G	$F \rightarrow G$
0	0	0	1	0	0	1
0	0	1	1	1	0	0
0	1	0	1	0	0	1
0	1	1	1	1	0	0
1	0	0	0	0	1	1
1	0	1	0	0	1	1
1	1	0	1	0	0	1
1	1	1	1	1	0	0

La formule A est satisfaisable et la distribution $(0, 0, 0)$ est un modèle de A.

Exemple 2

Évaluons $B = ((p \vee r) \wedge (q \vee \neg r)) \rightarrow (p \vee q)$.

On pose $F = (p \vee r) \wedge (q \vee \neg r)$.

p	q	r	p ∨ r	q ∨¬ r	F	p∨q	B
0	0	0	0	1	0	0	1
0	0	1	1	0	0	0	1
0	1	0	0	1	0	1	1
0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1
1	0	1	1	0	0	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

La formule B est une tautologie : $\models ((p \vee r) \wedge (q \vee \neg r)) \rightarrow (p \vee q)$

Quelques tautologies

Pour toutes formules A, B :

$\models (A \to A)$ identité

$\models (A \vee \neg A)$ tiers exclus

$\models (A \to (A \vee B))$

$\models (A \wedge \neg A) \to B$

$\models ((A \wedge B) \to A)$

$\models [A \to (B \to A)]$

Pour toutes formules A, B et C :

$\models (A \to B) \to [(B \to C) \to (A \to C)]$ exemple de syllogisme

$\models \{[A \to (B \to C)] \to [(A \to B) \to (A \to C)]\}$

$\models \{(\neg B \to \neg A) \to [(\neg B \to A) \to B)]\}$

Illustration du syllogisme
$\models (A \to B) \to [(B \to C) \to (A \to C)]$

- Prenons $A =$"il pleut", $B =$"il fait gris", $C =$"le sol glisse".
- Si il pleut, qu'il fait gris et que le sol ne glisse pas, on a $A = 1, B = 1, C = 0$. Donc $A \to B$ est vrai, $B \to C$ est faux (car il fait gris mais le sol ne glisse pas), et $A \to C$ est faux. Comme la formule $x \to y$ n'est fausse que si $y = 1$ et $x = 0$, alors la formule $(B \to C) \to (A \to C)$ est vraie (car ses sous-formules sont fausses). On a donc bien

$$\models \underbrace{\underbrace{(A \to B)}_{\text{vrai}} \to \underbrace{[(B \to C) \to (A \to C)]}_{\text{vrai}}}_{\text{vrai}}$$

- Si il pleut et le sol glisse mais qu'il ne fait pas gris, ca donne :
- $(A \to B)$ est faux, donc la grosse implication est vraie (indépendamment du membre de droite)

Formules équivalentes

Lois de DE MORGAN

Pour toutes formules A et B :

- On lit "non(A et B) équivaut à (non A) ou (non B)"

$$\neg(A \wedge B) \ \underline{eq} \ (\neg A) \vee (\neg B)$$

- On lit "non(A ou B) équivaut à (non A) et (non B)"

$$\neg(A \vee B) \ \underline{eq} \ (\neg A) \wedge (\neg B)$$

Formules équivalentes

Pour toutes formules propositionnelles p, q et r :

- $(\neg\neg p)$ $\underline{eq}$ p idempotence de $\neg$
- $((p \wedge q) \vee r))$ $\underline{eq}$ $((p \vee r) \wedge (q \vee r))$ distributivité de $\vee$ par rapport à $\wedge$
- $((p \vee q) \wedge r)$ $\underline{eq}$ $((p \wedge r) \vee (q \wedge r))$ distributivité du $\wedge$ par rapport à $\vee$
- $(p \rightarrow q)$ $\underline{eq}$ $(\neg q \rightarrow \neg p)$ contraposition
- $(p \rightarrow q)$ $\underline{eq}$ $(\neg p \vee q)$
- $\neg(p \rightarrow q)$ $\underline{eq}$ $(p \wedge \neg q)$
- $(p \leftrightarrow q)$ $\underline{eq}$ $((p \rightarrow q) \wedge (q \rightarrow p))$
- $(p \leftrightarrow q)$ $\underline{eq}$ $(p \wedge q) \vee (\neg p \wedge \neg q)$

Systèmes complets de connecteurs

Un système de connecteurs est dit complet si toute formule valide (syntaxe correcte) du calcul des propositions peut se construire à partir des connecteurs du système.

Exemple 1 : le système $\{\neg, \wedge, \vee\}$ est un système complet de connecteurs. En effet, $(a \rightarrow b)$ $\underline{eq}$ $\neg(a \wedge \neg b)$, et $(a \leftrightarrow b)$ $\underline{eq}$ $(a \wedge b) \vee (\neg a \wedge \neg b)$.

Exemple 2 : grâce aux Lois de De Morgan, les systèmes $\{\neg, \wedge\}$ et $\{\neg, \vee\}$ sont complets.

Exemple 3 : comme $(p \vee q)$ $\underline{eq}$ $(\neg p \rightarrow q)$ le système $\{\neg, \rightarrow\}$ est complet aussi.

Application : Lors de la construction de circuits logiques ceci permet d'utiliser un nombre limité de "circuits de base".

Sous formules

- Les règles syntaxiques d'écriture des formules du calcul propositionnel permettent de construire des formules.
- On a souvent besoin de simplifier une formule ou de l'écrire sous une forme normalisée (circuits logiques, démonstration automatique).
- On introduit la notion de sous formule :
  - Si A est une formule, alors $\neg A$ est une formule.
    On dit que "$\neg$" est le connecteur principal de $\neg A$
    et que A est la sous-formule (immédiate) de $\neg A$.
  - Si A et B sont des formules, alors $(A * B)$ est une formule,
    où $*$ est l'un des connecteurs binaires, $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.
    On dit que "$*$" est le connecteur principal de $(A * B)$
    et que A et B sont les deux sous-formules (immédiates) de $(A * B)$.
- Par induction :
  Les sous-formules des sous-formules d'une formule A sont des sous-formules de A.

Sous formules

- Si B est une sous-formule de A, et si B' $\underline{eq}$ B,
  alors la formule A', obtenue en remplaçant B par B' dans A,
  est équivalente à A.

- $\underline{\text{Exemple}}$: Soit $A = ((\neg a \wedge b) \vee c) \rightarrow (\neg\neg a \vee \neg b)$.
  On sait que $(\neg\neg a)$ $\underline{eq}$ a.
  Donc A $\underline{eq}$ $[((\neg a \wedge b) \vee c) \rightarrow (a \vee \neg b)]$.

- Ceci peut permettre de simplifier les formules,
  $\underline{\text{i.e.}}$ de les remplacer par des formules équivalentes plus courtes.

Formes normales

- On appelle littéral ou atome une formule qui est soit une variable propositionnelle, soit la négation d'un telle variable :
  si p et q sont des variables propositionnelles, alors p et $\neg q$ sont des atomes.
- La conjonction de formules $p_1, p_2, \ldots, p_n$ est la formule $p_1 \wedge p_2 \wedge \cdots \wedge p_n$.
- La disjonction de formules $p_1, p_2, \ldots, p_n$ est la formule $p_1 \vee p_2 \vee \cdots \vee p_n$.

On peut choisir comme normalisation de ne pas utiliser les "$\rightarrow, \leftrightarrow$", de mettre les "et" a "l'intérieur", les "ou" à l'extérieur, et les "non", devant les atomes. Ca nous donne :

- Une formule est sous forme normale disjonctive (FND) si elle est écrite sous forme de disjonctions de conjonctions de littéraux.

  Exemples : p, $(p \vee (\neg q))$, $(p \wedge q) \vee (\neg q)$ sont des FND.
  Mais pas $\neg(p \vee q)$, ni $p \vee (q \wedge (r \vee s))$.

Si au contraire, on choisit comme normalisation de mettre les "ou" a "l'intérieur" et les "et" à "l'extérieur", on a :

- Une formule est sous forme normale conjonctive (FNC) si elle est écrite sous forme de conjonctions de disjonctions de littéraux.

  Exemples : q, $(p \wedge (\neg q))$, $(p \vee q) \wedge (\neg q)$ sont des FND.
  Mais pas $\neg(p \wedge (\neg q))$, ni $p \wedge (q \vee (r \wedge s))$.
- Remarque : Les formules a (ou $\neg a$), et $a \vee b$, $a \wedge b$ sont à la fois sous FND et sous FNC

Formes normales

- Toute formule du calcul propositionnel est équivalente à une formule sous forme normale disjonctive et aussi à une formule sous forme normale conjonctive.
- Pour convertir une formule en forme normale, on utilise les lois de De Morgan, la distributivité des opérations $\vee$ et $\wedge$, l'une par rapport à l'autre, et l'idempotence de $\neg$.
- L'écriture sous forme normale peut agrandir la formule de manière exponentielle.

  Exemple : $(p_1 \vee q_1) \wedge (p_2 \vee q_2)$ est sous FNC et il y a 2 termes.

  Sa FND comporte 2^2 termes :

$$(p_1 \wedge p_2) \vee (q_1 \wedge p_2) \vee (p_1 \wedge q_2) \vee (q_1 \wedge q_2)$$

- Grâce aux tables de vérité, on peut facilement trouver les FND et FNC.

Algèbre de Boole

- Le nom d'**algèbre de Boole** ou **calcul booléen** est en l'honneur du mathématicien britannique George Boole (1815-1864), considéré comme créateur de la logique moderne.
- Le calcul booléen appliqué au calcul des propositions permet une approche algébrique pour traiter les formules logiques.

- On introduit les opérateurs arithmétiques binaires “+” pour $\vee$ et “.” pour $\wedge$. L'opérateur $\bar{a}$ représente l'opération unaire $\neg a$ du calcul des propositions.

  Exemple : $(a \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$ s'écrit $(a + \bar{c}).(\bar{a} + \bar{b} + \bar{c})$

- La structure d'algèbre de Boole s'applique dans d'autres cadres. Un exemple est l'algèbre de Boole des parties d'un ensemble E, $\mathcal{P}(E)$. L'opération“+” est alors la réunion $\cup$, l'opération “.” est l'intersection $\cap$ et $\bar{A}$ désigne le complément de A dans E. La phrase précédente s'écrirait

$$(A \cup \overline{C}) \cap (\overline{A} \cup \overline{B} \cup \overline{C}),$$

  et les opérations ensemblistes obéissent aux memes lois que les opérations logiques.

Propriétés de base

L'ensemble $\{0,1\}$ est muni des opérations "+", "." et "$\neg$" vérifiants :

- associativité : $(a+b)+c = a+(b+c)$ et $(a.b).c = a.(b.c)$
- commutativité : $a+b = b+a$ et $a.b = b.a$
- éléments neutre : $0+a = a$ et $1.a = a$
- idempotence : $a+a = a$ et $a.a = a$
- involution : $\bar{\bar{a}} = a$
- complémentarité : $a.\bar{a} = 0$ et $a+\bar{a} = 1$
- éléments absorbants : $a+1 = 1$ et $a.0 = 0$
- distributivité de . par rapport à + : $a.(b+c) = a.b+a.c$
- distributivité de + par rapport à . : $a+(b.c) = (a+b).(a+c)$

Attention : utiliser ces notations et règles dans le bon contexte !

En particulier ne pas confondre avec les lois sur le corps F_2, vues en calcul modulaire. Par exemple, $1+1 \equiv 1$ est faux dans F_2.

Propriétés de base

- Les lois de De Morgan s'écrivent :

$$\overline{a+b} = \bar{a}.\bar{b} \qquad \text{et} \qquad \overline{a.b} = \bar{a} + \bar{b}\,.$$

- Les simplifications suivantes peuvent être utiles

$$a + \bar{a}.b = a + b \qquad \text{et} \qquad (a+b).(a+c) = a + b.c$$

- Le calcul booléen est utilisée en électronique pour simplifier des circuits logiques ou en programmation pour simplifier des tests logiques.

- Suivant le langage de programmation, le contexte,. . . ,
  les opérations sont notées de différentes façons :
  - le "." est aussi noté "$\wedge$", "&", "&&" ou "AND" ;
  - le "+" est aussi noté "$\vee$", "—", "——" ou "OR" ;
  - le "$\neg$" est aussi noté "$\neg$", " !", "NOT" ;

Fonction booléenne

- Les formules du calcul des propositions deviennent des fonctions booléennes c'est-à -dire des applications de $\{0,1\}^n \longrightarrow \{0,1\}$ où n est le nombre de variables.
  On parle aussi de fonctions logiques.
- Pour une fonction booléenne de n variables $f(x_1, \ldots, x_n)$
  - on appelle minterme un produit m qui contient chaque variable x_i $(1 \leq i \leq n)$, ou sa négation, une seule fois et tel que $m = 1$ entraîne que $f(x_1, \ldots, x_n)$ est vraie. Il se peut que l'unique minterme de f soit f elle-meme (exemple : $f(x) = x_1$).
  - on appelle maxterme une somme M qui contient chaque variable x_i $(1 \leq i \leq n)$, ou sa négation, une seule fois et telle que $M = 0$ entraîne que $f(x_1, \ldots, x_n)$ est fausse.
- Pour une fonction logique f on peut dire que
  - la **FND** de f est la disjonction des mintermes de f ;
  - la **FNC** de f est la conjonction des maxtermes de f.

Fonction booléenne. Exemple

On considère une fonction $f : \{0,1\}^3 \longrightarrow \{0,1\}$ définie par

a	b	c	$f(a,b,c)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

On obtient

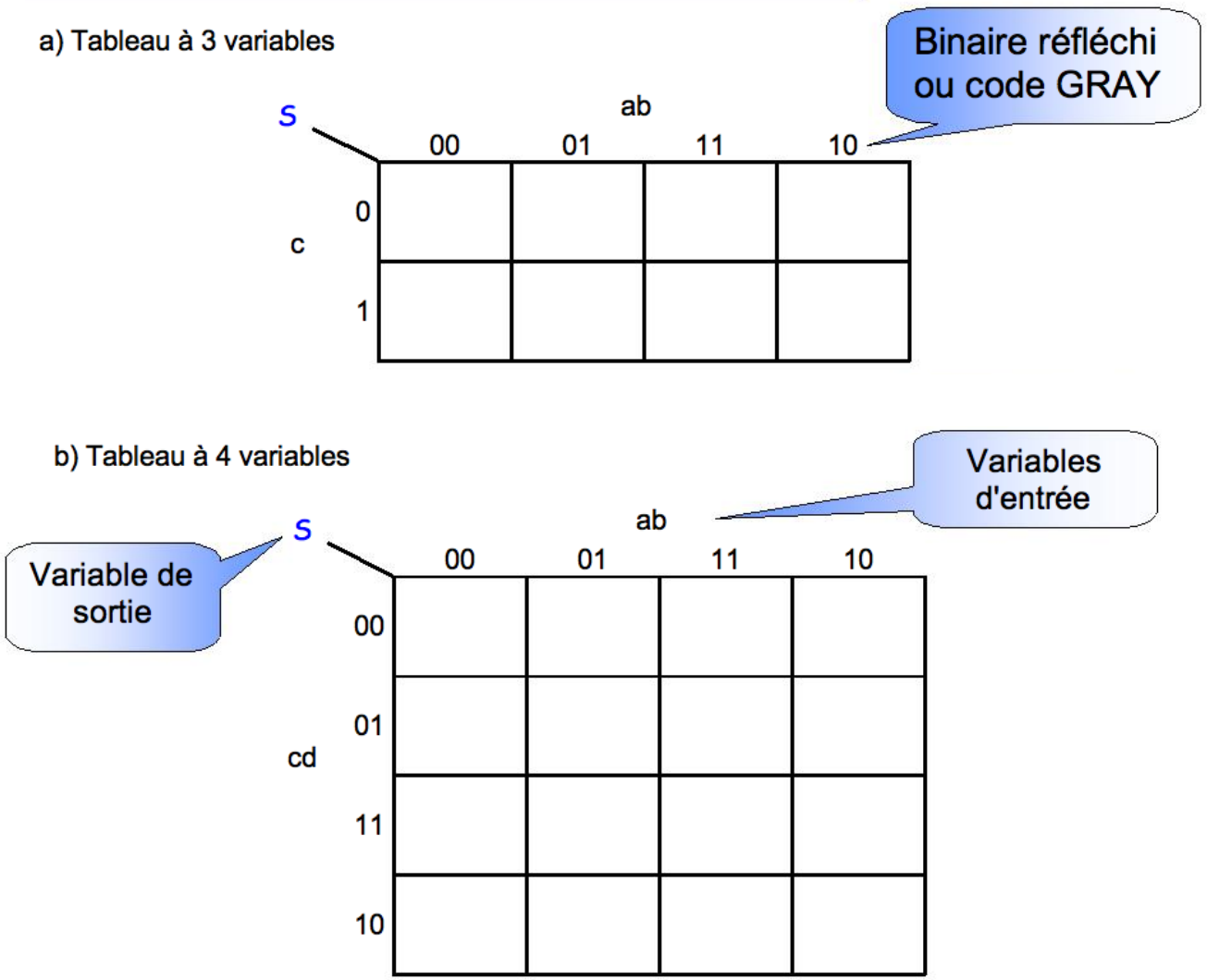
$$\begin{aligned}
\text{FND de } f &= (\bar{a}.\bar{b}.\bar{c}) + (\bar{a}.b.\bar{c}) + (a.\bar{b}.\bar{c}) + (a.\bar{b}.c) + (a.b.\bar{c}) \\
&= \underline{\text{“somme des mintermes”}} \\
\text{FNC de } f &= (a+b+\bar{c}).(a+\bar{b}+\bar{c}).(\bar{a}+\bar{b}+\bar{c}) \\
&= \underline{\text{“produit des maxtermes”}}
\end{aligned}$$

Explication de la terminologie : f est plus “grande” que chacun de ses mintermes, et plus “petite” que chacun de ses maxtermes.

Table de Karnaugh

- La simplification d'une expression logique par le **tableau de Karnaugh** est une méthode développé en 1953 par Maurice Karnaugh, ingénieur en télécommunications au laboratoires Bell.
- La méthode de Karnaugh consiste à présenter les états d'une fonction logique, non pas sous la forme d'une table de vérité, mais en utilisant un **tableau à double entrée**.
- Chaque case du tableau correspond à une combinaison des variables d'entrées, donc à une ligne de la table de vérité.
- Le tableau de Karnaugh aura autant de cases que la table de vérité possède de lignes.
- Les lignes et les colonnes du tableau sont numérotées selon le code binaire réfléchi (code de Gray) :
  à chaque passage d'une case à l'autre, une seule variable change d'état.

Tableaux de Karnaugh. Exemple



On remplit le tableau grâce à la fonction booléenne $S = f(a, b, c, d)$.

Table de Karnaugh. Somme

Pour obtenir une somme on procède comme suit

- **Regrouper** les cases adjacentes de “1” par paquets de taille des puissance de 2.

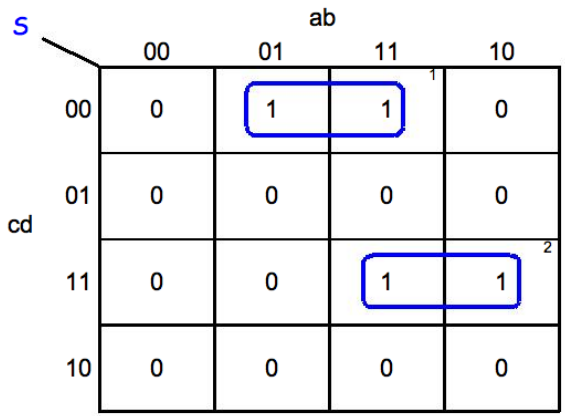
  Pour minimiser le nombre de paquets, prendre les rectangles le plus grand possible : $2^n, \ldots, 16, 8, 4, 2, 1$.
- Une même case peut faire partie de plusieurs regroupements.
- Les regroupements peuvent se faire au delà des bords : les côtés/coins ont des codes Gray voisins.
- Toute case contenant “1” doit faire partie d’au moins un regroupement, mais aucun “0” ne doit y être.
- Pour chaque rectangle, on **élimine** les variables qui changent d’état, l’on ne conserve que celles qui restent fixes.
  On **multiplie** les variables fixes par “.”
  afin d’obtenir des **mintermes** de S.
- Les produits obtenus sont ensuite **sommés** avec “+”
  et l’on obtient une **FND** de S.

Table de Karnaugh

Exemple :

Simplifier la fonction $S = \bar{a}.b.\bar{c}.\bar{d} + a.b.c.d + a.\bar{b}.c.d + a.b.\bar{c}.\bar{d}$

La table de Karnaugh associée est



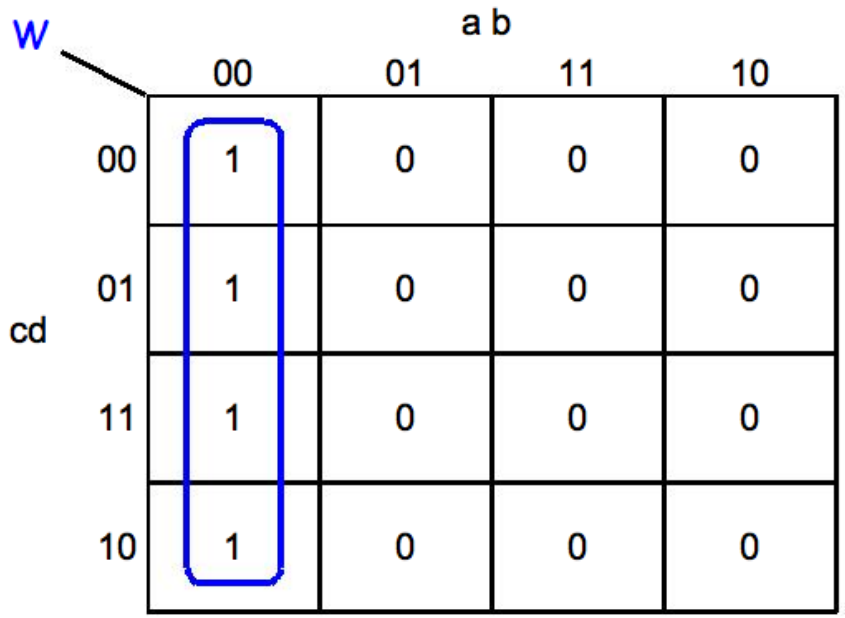
- 1^{er} regroupement : a change d'état et est éliminé, b vaut 1, c et d valent 0, d'où le minterme $b\bar{c}\bar{d}$
- 2^{eme} regroupement : b change d'état et est éliminé, a, c et d valent 1, d'où le minterme acd
- On fait la somme des mintermes et $S = (acd) + (b\bar{c}\bar{d})$

Table de Karnaugh

Exemple : Soit

$$W = \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}d + \bar{a}\bar{b}cd + \bar{a}\bar{b}c\bar{d}$$

On dresse la table de Karnaugh :



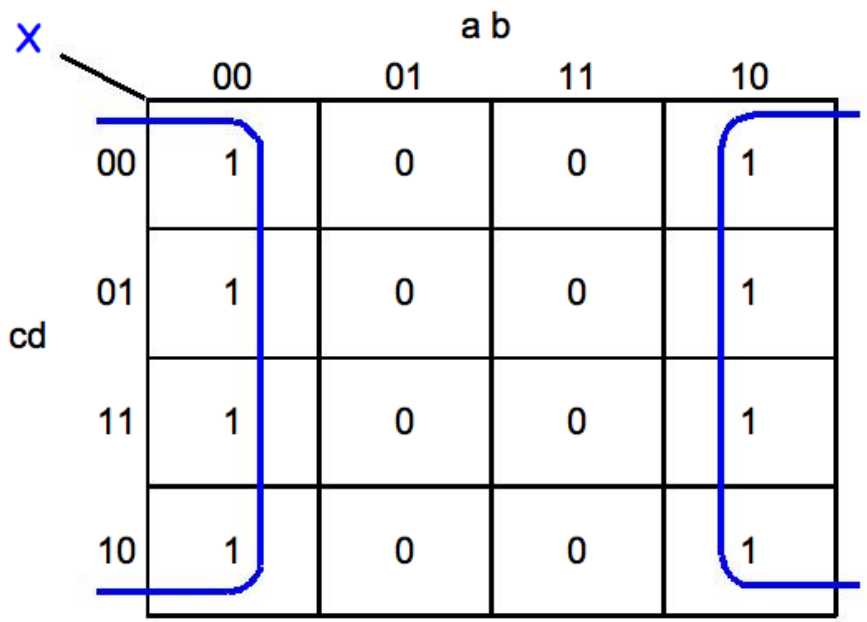
$W = \bar{a}\,\bar{b}$

Table de Karnaugh

Exemple : Soit

$$X = \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}d + \bar{a}\bar{b}cd + \bar{a}\bar{b}c\bar{d} + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}d + a\bar{b}cd + a\bar{b}c\bar{d}$$

On dresse la table de Karnaugh :



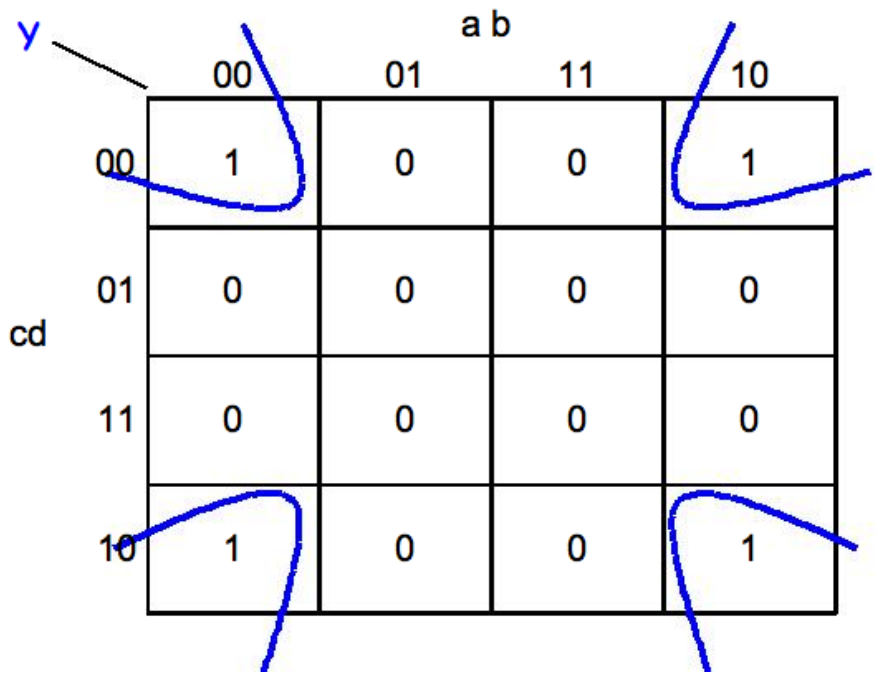
$X = \bar{b}$

Table de Karnaugh

Exemple : Soit

$$Y = \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}c\bar{d} + a\bar{b}\bar{c}\bar{d} + a\bar{b}c\bar{d}$$

On dresse la table de Karnaugh :



$Y = \bar{b}\,\bar{d}$

Logique appliquée à la criminologie :)

Résolution de problèmes par le calcul des propositions
en utilisant la simplification par table de Karnaugh.

Lors d'une enquête de l'inspecteur Maigret, les personnes A, B, C et D sont suspectées. Il est établi que :

1. Si A et B sont coupables, il en est de même de C.
2. Si A est coupable, l'un au moins de B et C est aussi coupable.
3. Si C est coupable, D l'est aussi.
4. Si A est innocent, D est coupable.

Peut-on établir la culpabilité de l'un ou plusieurs des suspects ?

On définit les propositions :

$a =$ '« A est coupable '», $\quad b =$ '« B est coupable '» ,
$c =$ '« C est coupable '» et $d =$ '« D est coupable '».

Application (suite)

On traduit les quatre affirmations en langage des propositions et l'on écrit leur conjonction :

$$M = [(a \wedge b) \rightarrow c] \wedge [a \rightarrow (b \vee c)] \wedge [c \rightarrow d] \wedge [\neg a \rightarrow d]$$

On dresse la table de Karnaugh de M :

cd \ ab	00	01	11	10
00	0	0	0	0
01	1	1	0	0
11	1	1	1	1
10	0	0	0	0

D'où $M = \bar{a}.d + c.d$,

en langage des propositions on a $M = (\neg a \wedge d) \vee (c \wedge d)$ qui se factorise en $M = (\neg a \vee c) \wedge d$.

On peut donc conclure que D est certainement coupable !

Table de Karnaugh. Produit

- Obtenir la FNC d'une expression (i.e. l'obtenir comme un produit) via son tableau de Karnaugh revient à obtenir la FND de sa négation (i.e. obtenir sa négation sous forme de somme)
- Pour obtenir la FND de $\overline{S}$, il faut déterminer les cases qui rendent $\overline{S}$ vraies, c'est donc les cases qui contiennent 0
- On fait ensuite des regroupements de 0 (comme on fait avec les 1 pour obtenir la FND)
- On obtient ainsi une somme pour $\overline{S}$, qui se transforme en produit quand on applique la négation pour obtenir S.

Table de Karnaugh. Produit

Pour obtenir un produit on procède comme suit

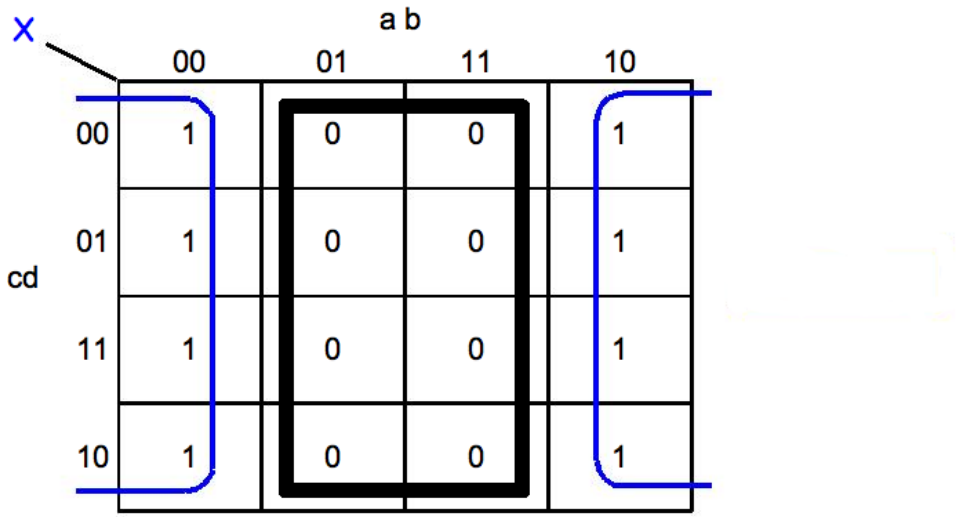
- **Regrouper** les cases adjacentes de “0” par paquets de taille des puissance de 2.

  Pour minimiser le nombre de paquets, prendre les rectangles le plus grand possible : 2^n, . . . , 16, 8, 4, 2,1.
- Une même case peut faire partie de plusieurs regroupements.
- Les regroupements peuvent se faire au delà les bords : les côtés/coins ont des codes Gray voisins.
- Toute case contenant “0” doit faire partie d’au moins un regroupement, mais aucun “1” ne doit y être.
- Pour chaque rectangle, on **élimine** les variables qui changent d’état, l’on ne conserve que celles qui restent fixes.
  On **somme** les variables fixes avec “+”
  afin d’obtenir des **maxtermes** de S.
- Les sommes obtenus sont ensuite **multipliées** avec “.”
  et l’on obtient une **FNC** de S.

Table de Karnaugh

Exemple : On reprend la table de Karnaugh de

$$X = \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}d + \bar{a}\bar{b}cd + \bar{a}\bar{b}c\bar{d} + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}d + a\bar{b}cd + a\bar{b}c\bar{d}$$



$$\overline{X} = b, X = \bar{b}$$

1. Soit on regroupe les “0” et on fait le **produit des maxtermes** de variables qui ne changent pas et qui rendent faux X.
2. Soit on regroupe les “1”et on fait la somme des mintermes de variables qui ne changent pas et qui rendent vrai X.

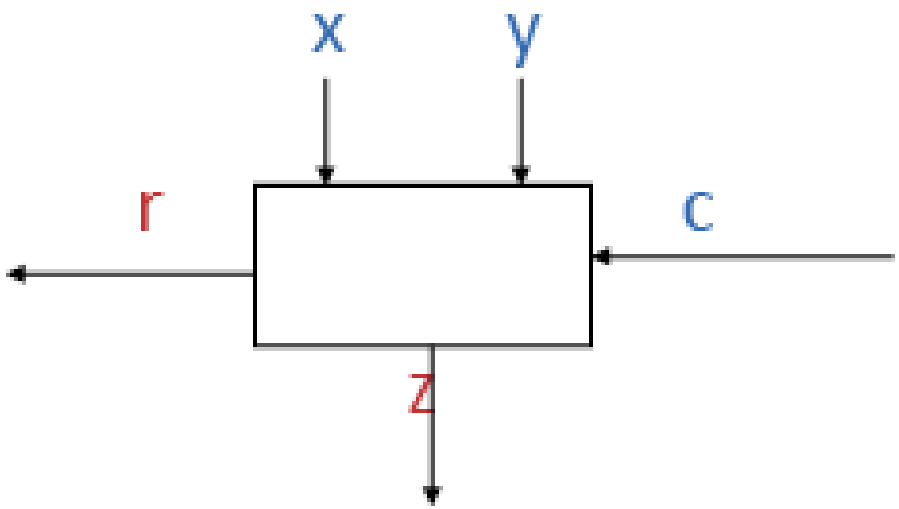
Circuits logiques et booléens

- Les circuits logiques, composants de base des ordinateurs, sont conçus à partir de circuits élémentaires correspondants aux opérations booléennes “.”, “+” et “ $^-$ ”.
- Un circuit logique peut être vu comme une boîte noire ayant
  $n \geq 1$ ports d'entrée $e_1, e_2, \ldots, e_n$
  $m \geq 1$ ports de sortie $s_1, s_2, \ldots, s_m$
- Il traite des informations codées sur n bits et donne des informations codées sur m bits.
- Le codage de l'information, en entrée ou sortie, est représenté par l'absence (0) ou la présence (1) d'une tension électrique.
- On appelle ces circuits *logiques* car un bit d'information 0 est assimilé à la valeur de vérité *faux* et 1 à *vrai*
- On représente ainsi une application de $\{0,1\}^n$ dans $\{0,1\}^m$

Exemple : additionneur 1 bit

Soit un circuit électronique destiné à l'addition bit à bit.

Le schéma est le suivant

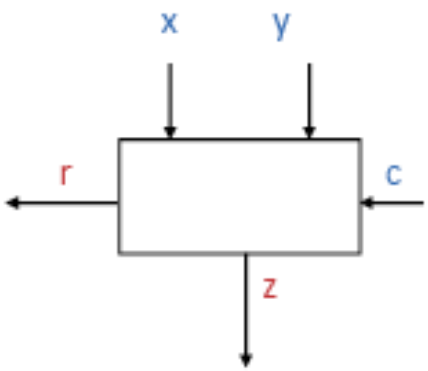


- Les entrées sont les bits X et Y et le report C .
- Les sorties sont le bit de sortie Z et la retenue r.
- Le résultat de $(X+Y+C)_2$ est donné par Z et r.

Table de vérité de l'additionneur bit à bit

On obtient ainsi la table de vérité

entrées			sorties	
x	y	c	z	r
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



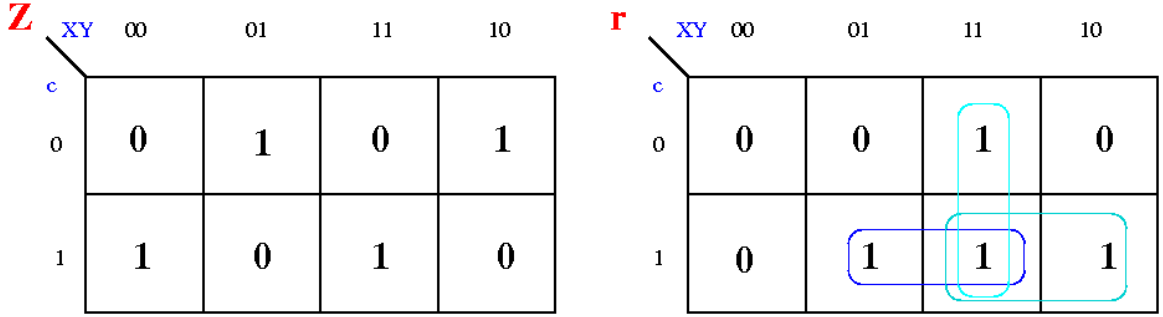
D'où :

$$Z = (\bar{x}.\bar{y}.c) + (.\bar{x}.y.\bar{c}) + (x.\bar{y}.\bar{c}) + (x.y.c)$$

$$r = (\bar{x}.y.c) + (x.\bar{y}.c) + (x.y.\bar{c}) + (x.y.c)$$

Additionneur : table de Karnaugh

- On peut dresser les tables de Karnaugh pour Z et r :



- On déduit une forme normale disjonctive simplifiée pour r

$$r = (x.y) + (x.c) + (y.c)$$

Il suffit de trois . et deux $+$ pour implémenter la retenue.

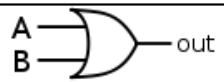
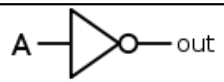
- La sortie Z s'écrit plus facilement avec l'opérateur $\oplus$, "ou exclusif"

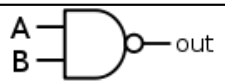
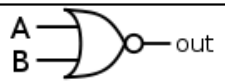
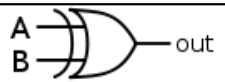
$$Z = (x \oplus y) \oplus c\,, \qquad \text{avec}$$

$\oplus$	0	1
0	0	1
1	1	0

Représentation des portes logiques élémentaires

- Un circuit est représenté grâce aux opérateurs de bases.
- La représentation des portes logiques de base est définie par des normes ANSI/IEEE.

Nom	Symbole	Opération
ET		$A.B$
OU		$A + B$
NON		$\overline{A}$

Nom	Symbole	Opération
NON-ET		$\overline{A.B}$
NON-OU		$\overline{A + B}$
OU exclusif		$A \oplus B$

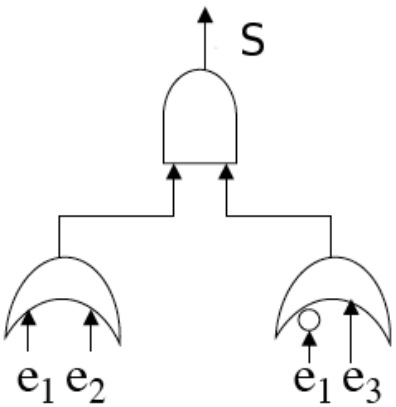
Voir `http://fr.wikipedia.org/wiki/Fonction_logique`

- La porte NON est souvent appelé **inverseur**.
  Utilisée en entrée ou en sortie, elle est représentée comme une “bulle”.

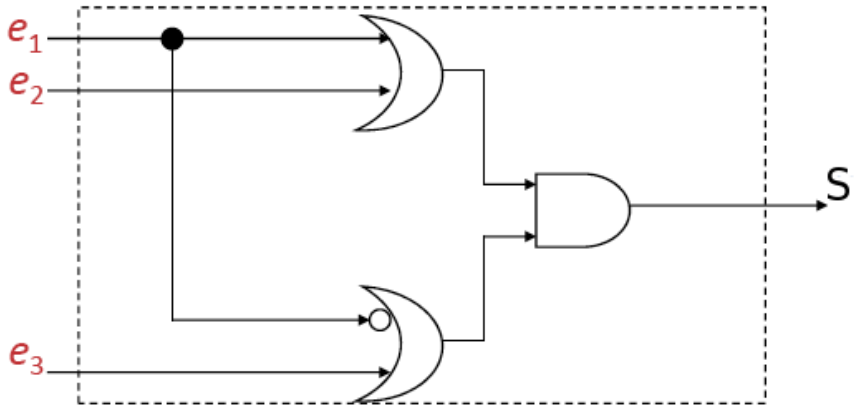
Des formules aux circuits

Exemple : $S = (e_1 + e_2).(\bar{e_1} + e_3)$

- Le circuit doit avoir autant d'entrées que de variables booléennes.
- On introduit des bifurcations par des • pour répéter des variables.



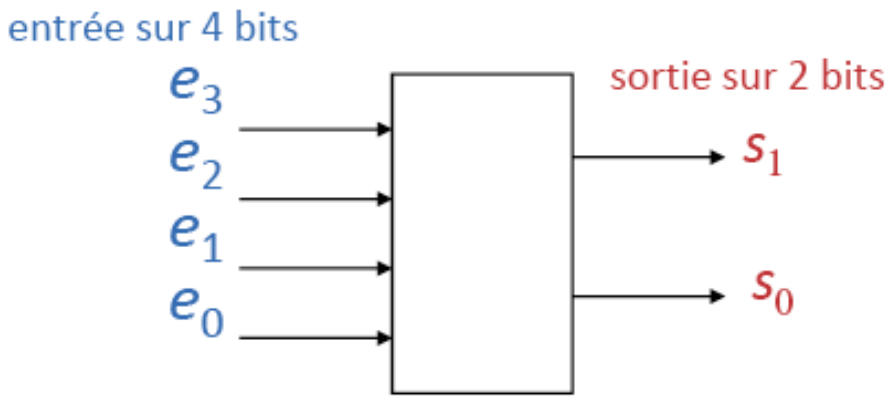
devient



Exemple : reste de la division par 3

Spécifier un circuit logique tel que

- l'entrée est un entier *n* compris entre 0 et 15
- la sortie est le reste de la division entière de *n* par 3
- alors
  1. l'entrée peut être codée en binaire naturel sur 4 bits ;
  2. la sortie peut être codée sur 2 bits ;
  3. Le schéma est



Exemple : reste de la division par 3

La table de vérité est

entier	e_3	e_2	e_1	e_0	s_1	s_0	Valeur du reste
0	0	0	0	0	0	0	**0**
1	0	0	0	1	0	1	**1**
2	0	0	1	0	1	0	**2**
3	0	0	1	1	0	0	**0**
4	0	1	0	0	0	1	**1**
5	0	1	0	1	1	0	**2**
6	0	1	1	0	0	0	**0**
7	0	1	1	1	0	1	**1**
8	1	0	0	0	1	0	**2**
9	1	0	0	1	0	0	**0**
10	1	0	1	0	0	1	**1**
11	1	0	1	1	1	0	**2**
12	1	1	0	0	0	0	**0**
13	1	1	0	1	0	1	**1**
14	1	1	1	0	1	0	**2**
15	1	1	1	1	0	0	**0**

Exemple : reste de la division par 3

L'on obtient les formes normales :

$$s_0 = (\bar{e}_3.\bar{e}_2.\bar{e}_1.e_0) + (\bar{e}_3.e_2.\bar{e}_1.\bar{e}_0) + (\bar{e}_3.e_2.e_1.e_0) + (e_3.\bar{e}_2.e_1.\bar{e}_0) + (e_3.e_2.\bar{e}_1.e_0)$$

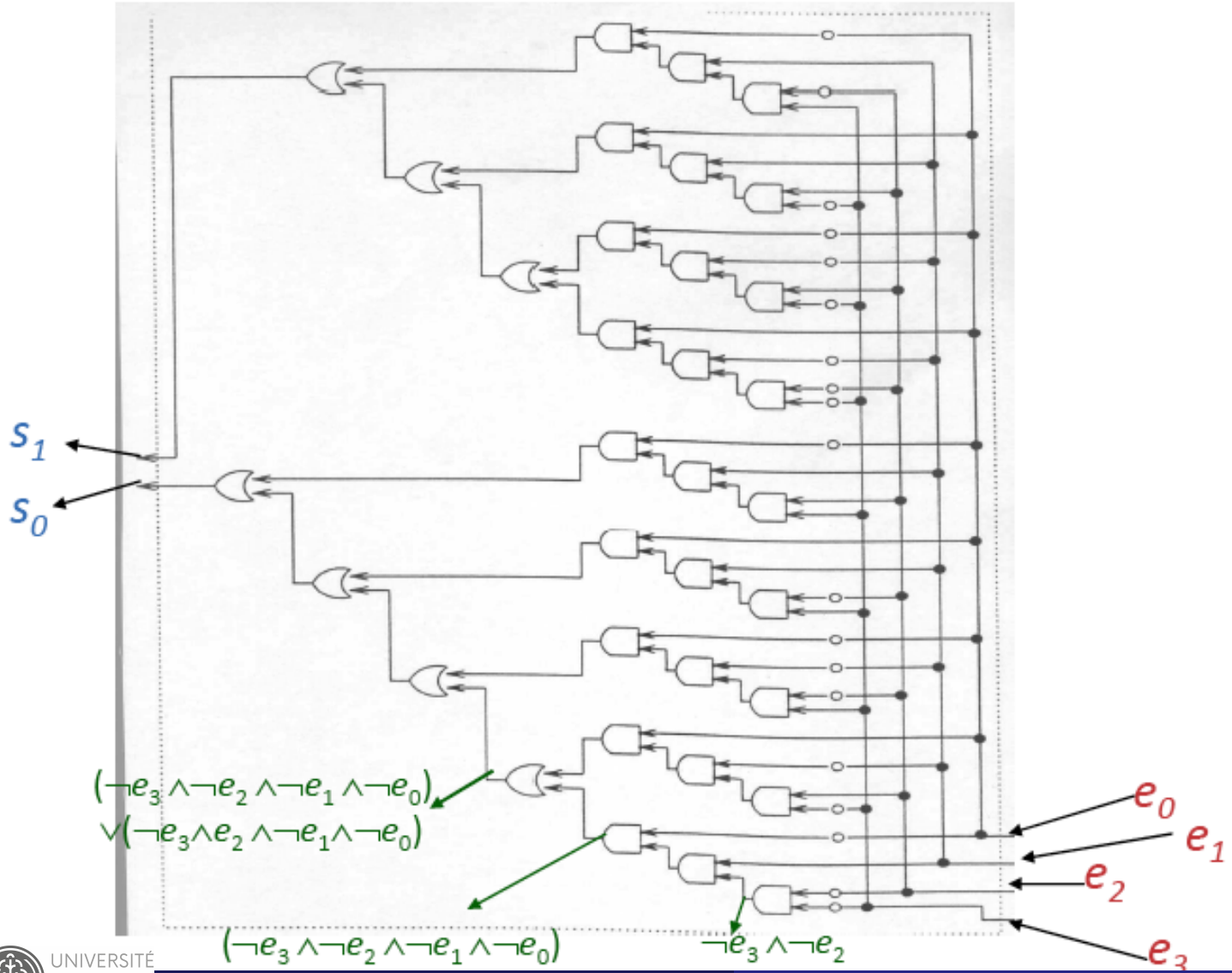
et

$$s_1 = (\bar{e}_3.\bar{e}_2.e_1.\bar{e_0}) + (\bar{e}_3.e_2.\bar{e}_1.e_0) + (e_3.\bar{e}_2.\bar{e}_1.\bar{e}_0) + (e_3.\bar{e}_2.e_1.e_0) + (e_3.e_2.e_1.\bar{e}_0)$$

d'où l'on peut tirer le schéma du circuit...

Exemple : reste de la division par 3

Le circuit. . .



Exemple : circuit additionneur 16 bits

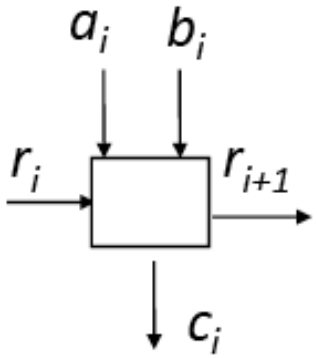
Un circuit additionneur 16 bits a 32 entrées et 17 sorties :

- 16 entrées reçoivent un premier entier naturel codé sur 16 bits ;
- 16 autres entrées reçoivent un deuxième entier naturel codé sur 16 bits ;
- 16 sorties constituent la représentation en base 2 de la somme deux entiers en entrée ;
- Le 17ème bit est un éventuel dépassement de capacité.

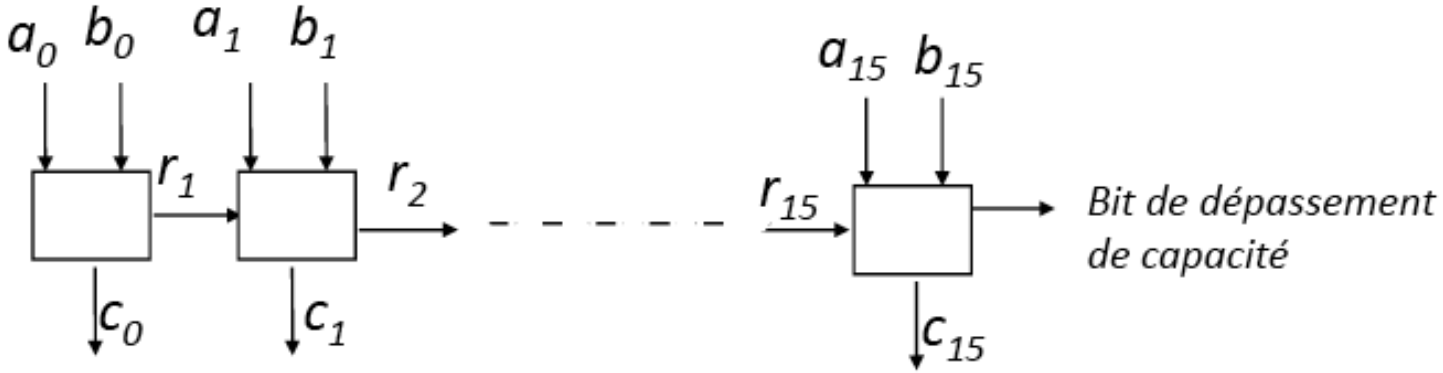
- Une spécification par table de vérité n'est pas envisageable.
- De même, écrire les 17 formules spécifiant les 17 sorties sous forme normale est laborieux !
- On construit/développe un circuit additionneur 1 bit.
  On composera autant de '≪circuit 1 bit '≫ que nécessaire.
- Le circuit sera un circuit séquentiel encore appelé bascule :
  une partie du circuit doit attendre le résultat d'une autre partie du circuit, d'où besoin d'une horloge.

Exemple : circuit additionneur 16 bits

- L'additionneur 1 bit avec retenue

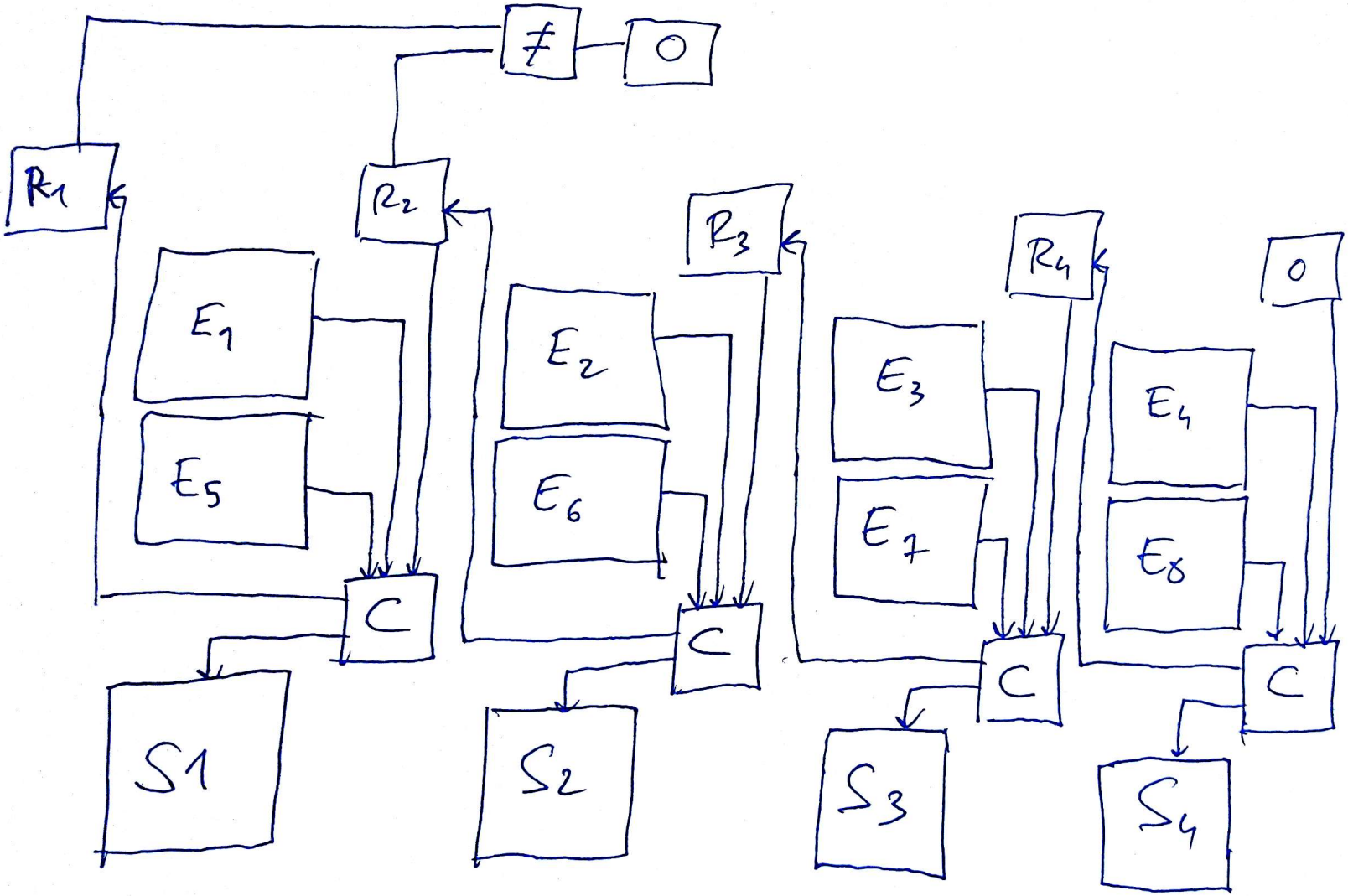


- L'additionneur 16 bits consiste en la succession de 16 additionneurs 1 bit



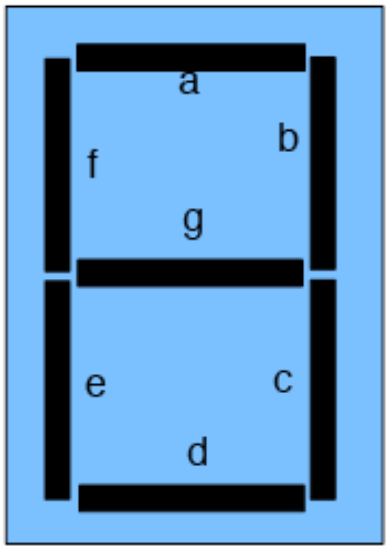
- Le $i$ème circuit 1 bit doit attendre le report du $(i-1)$ème circuit 1 bit, $1 \leq i \leq 16$, avant de commencer.

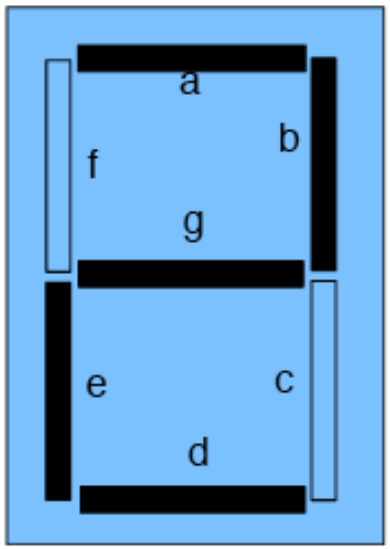
Représentation globale dans le cas simplifié d'addition en $CA2_4$



Affichage à cristaux liquides

- Un afficheur numérique est composé de sept barres : a, b, c, d, e, f et g qui peuvent être allumées ou éteintes ;
- On veut afficher les chiffres de 0 à 9 : l'entrée du circuit est le code binaire du chiffre à afficher, la sortie est le signal qui allume ou éteint chaque barre.
- Exemple pour afficher 2 : il faut que a=1, b=1 , c=0 , d=1 , e=1 , f=0 et g=1 .





Affichage à cristaux liquides

Circuit de la barre e : soient p, q, r et s les quatre bits d'entrée pour coder les 10 chiffres.
La table de vérité pour la barre e s'écrit alors

p	q	r	s	e
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0

Affichage à cristaux liquides

On écrit la formule E

$$E = (\bar{p}.\bar{q}.\bar{r}.\bar{s}) + (\bar{p}.\bar{q}.r.\bar{s}) + (\bar{p}.q.r.\bar{s}) + (p.\bar{q}.\bar{r}.\bar{s})$$

On simplifie

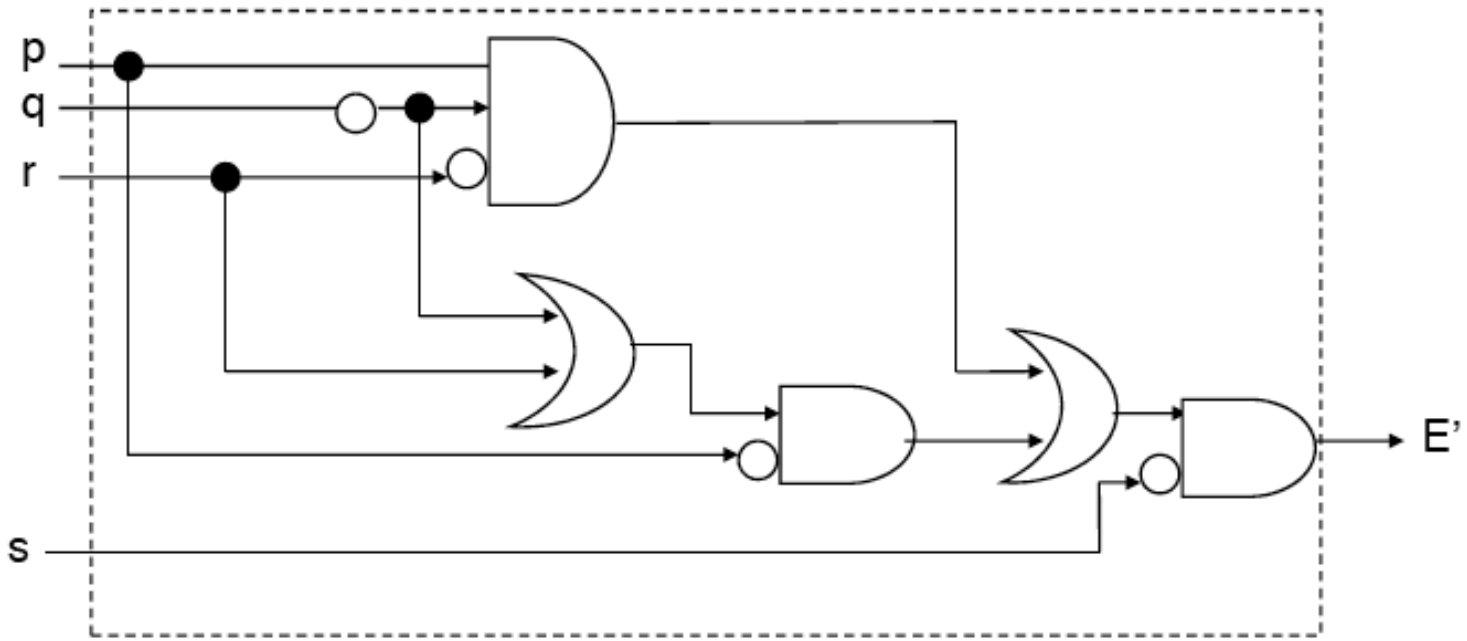
$$E = \bar{s}.([\bar{p}.(\bar{q} + r)] + (p.\bar{q}.\bar{r}))$$

L'affichage à cristaux liquides

a partir de

$$E = \bar{s}.([\bar{p}.(\bar{q} + r)] + (p.\bar{q}.\bar{r}))$$

on obtient le circuit de la barre e :



Notation polonaise

- Pour écrire les expressions algébriques impliquant des opérateurs binaires, on utilise en général la notion infixée :
  l'opérateur est écrit **entre** les opérandes.

  Exemples : "$2+2$" ; "$2*(4-5)$" ; "$p \wedge (q \vee r)$" ; "$(a.b)+c$" ...

- En mathématiques, on utilise le plus souvent une notation préfixée pour les opérateurs unaires :
  $\sin\theta$ "sinus theta" ; $\log x$ "logarithme de x" .
  Une exception est $n!$ qui se lit "factorielle n".

- Le mathématicien polonais Jan Lukasiewicz a proposé en 1924 une notation préfixée pour les opérateurs binaires :
  l'opérateur est écrit **avant** les opérandes.

  Exemples : "$2+2$" s'écrit "+ 2 2" et "$2*(4-5)$" s'écrit "* 2 - 4 5"

- En honneur de Lukasiewicz on parle de notation polonaise.

Notation polonaise

- Cette notation ne nécessite pas de parenthèses
  et est sans ambiguïté si les opérateurs sont binaires :

  Exemples : “* - 3 2 4” signifie “*(- 3 2) 4”, c’est-à-dire “(3-2)*4” ;
  “/ * 4 2 2” signifie “(4*2)/2” et “* / 4 2 2” signifie “(4/2) * 2”.

- Des notations préfixe sont utilisées dans des langages de programmations tels que `Lisp`, `Tcl`, `Apl`.

- Dans la notation polonaise inverse ou notation postfixée
  l’opérateur est écrit **après** les opérandes.

  Exemples : “2 2 +” ou “3 4 * 5 1 - *”

  Dans les années 1970/80, les calculatrices HP (Hewlett-Packard)
  ont utilisé ce principe :
  il y a une touche [ENTER] qui permet de remplir une **pile**,
  une touche [CHS], mais pas les touches [=], [(] ou [)].
  On utilise moins de touches qu’avec la notation infixe.

Notation polonaise

- Afin d'éviter des ambiguïtés un opérateur est soit unaire, soit binaire,...
- Lukasiewicz a introduit les notations suivantes pour le calcul des propositions

Not. standard	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
Not. polonaise	Np	Kpq	Apq	Cpq	Epq

- Exemple : Soit la formule $F = ((\neg a \vee b) \wedge c) \rightarrow (\neg\neg a \wedge \neg b)$

  Elle s'écrit $CKANabcKNNaNb$
  (lecture facile $C(K(A(Na)b)c)(K(N(Na)(Nb))))$

- Cette notation
  1. n'utilise aucune parenthèse ;
  2. définit la formule F sans ambiguïté.

Notation polonaise préfixée

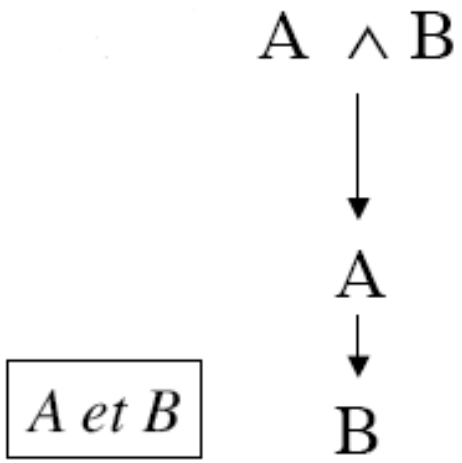
- Pas n'importe quelle suite de symboles ne représente une formule.
- Mais la notation polonaise préfixée permet un test de bonne formation très simple :

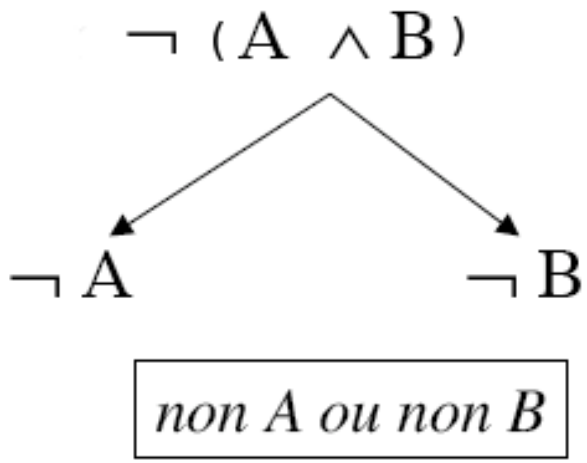
  On affecte
  1. le poids -1 aux variables
  2. le poids +1 aux connecteurs binaires K, A, C et E
  3. le poids 0 au connecteur unaire N
- Alors
  1. La somme des poids d'une formule est -1.
  2. Toute somme partielle à partir de la gauche est positive.
- Exemples :
  ACpqr est bien formée et signifie "$(p \to q) \vee r$"
  ACpEqr est mal formée : la somme vaut 0.

Arbres de Beth

- Ce sont des arbres qui peuvent remplacer les tables de vérité ou tables de Karnaugh, lorsqu'on recherche des distributions de vérité qui rendent une ou plusieurs formules vraies.
- On parle aussi de **méthode des arbres de vérité**, **méthode des tableaux**. L'appellation "arbres de Beth" est en honneur du logicien néerlandais Evert Willem Beth (1908-1964).
- De ces arbres on peut également déduire des FND ou FNC souvent simplifiées par rapport à la méthode des tables de vérité.
- Méthode :
  1. on commence l'arbre en plaçant toutes les formules à vérifier sur une branche ;
  2. de façon inductive, on décompose chaque formule grâce aux arbres associés aux opérations de base du calcul des propositions ;
  3. lorsque seul les atomes ou leur négation restent, on a terminé.

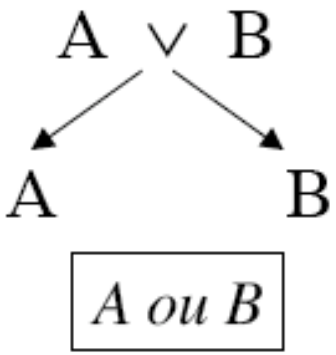
Arbres de Beth pour la conjonction

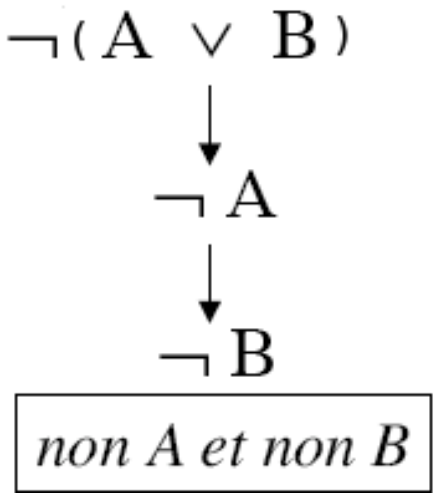




1. Si toutes les deux formules A et B sont des conséquences de la formule initiale, on ajoute les deux sur la branche (cas du $\wedge$) ;
2. la négation $\neg$ est simplifiée en utilisant les lois de De Morgan ;
3. si l'une au moins est une conséquence, on crée une branche pour A et une pour B (cas de $\vee$).

Arbres de Beth pour la disjonction

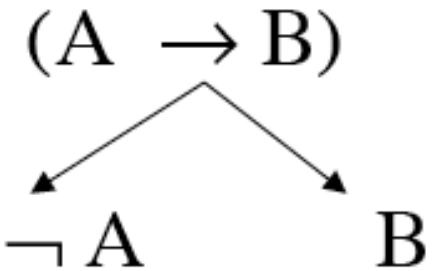




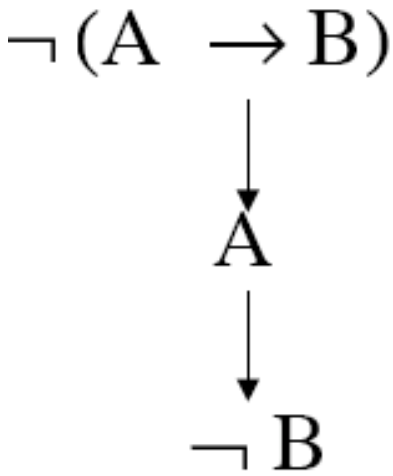
1. Si l'une au moins des formules A ou B est une conséquence, on crée une branche pour A et une pour B (cas de $\vee$) ;
2. la négation $\neg$ est simplifiée en utilisant les lois de De Morgan ;
3. si toutes les deux formules sont des conséquences de la formule initiale, on ajoute les deux sur la branche (cas du $\wedge$).

Arbres de Beth pour l'implication

Rappel : $A \to B \underline{\text{eq}} \neg A \vee B$.

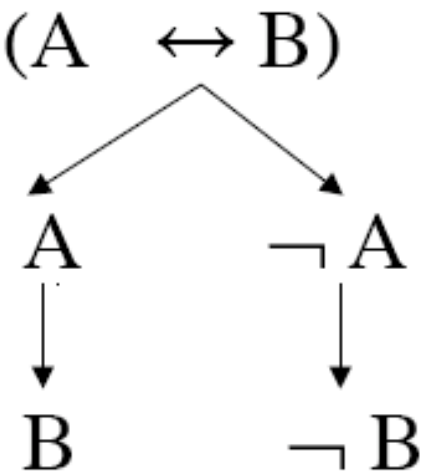


non A ou B

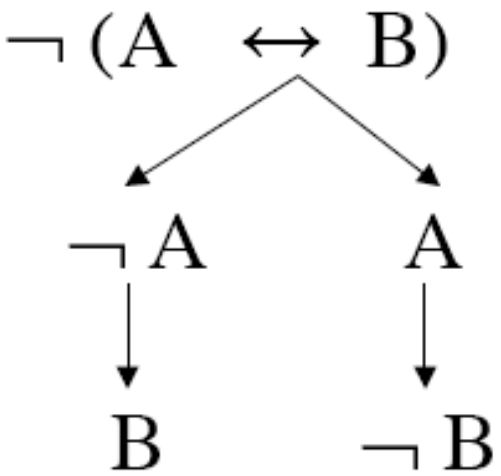


A et non B

Arbres de Beth pour l'équivalence



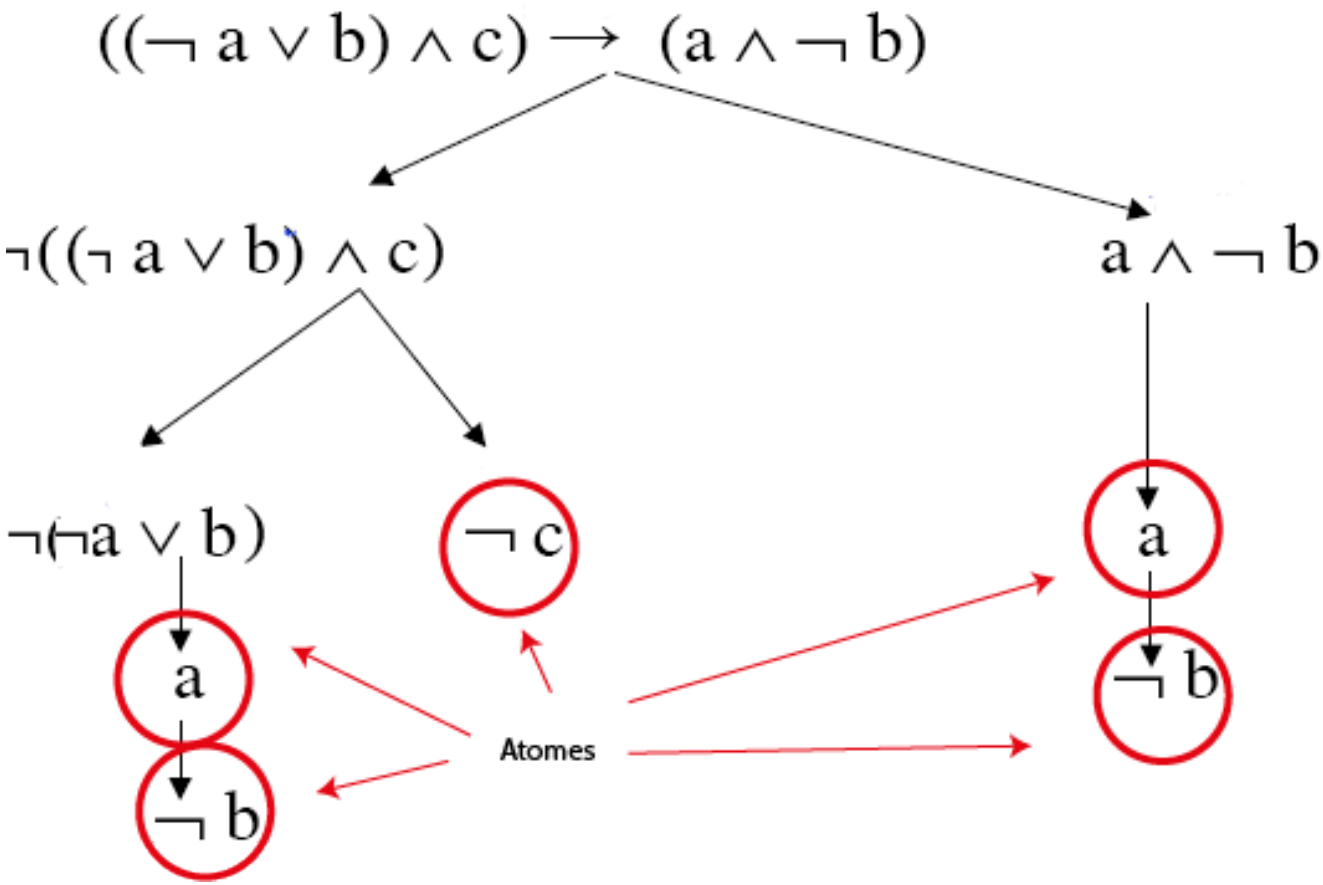
(A et B) ou (non A et non B)



(non A et B) ou (A et non B)

Arbres de Beth

Exemple : $F = ((\neg a \vee b) \wedge c) \rightarrow (a \wedge \neg b)$



Arbres de Beth

Exemple : F (suite)

- On peut déduire de cet arbre les distributions de vérité qui rendent la formule F vraie :

  Si l'on rend vrais tous les atomes d'une branche, on rend vraies toutes les formules de cette branche donc la formule d'origine.

- Il suffit que $(a \wedge \neg b)$ ou $\neg c$ soient vrais pour que F le soit.

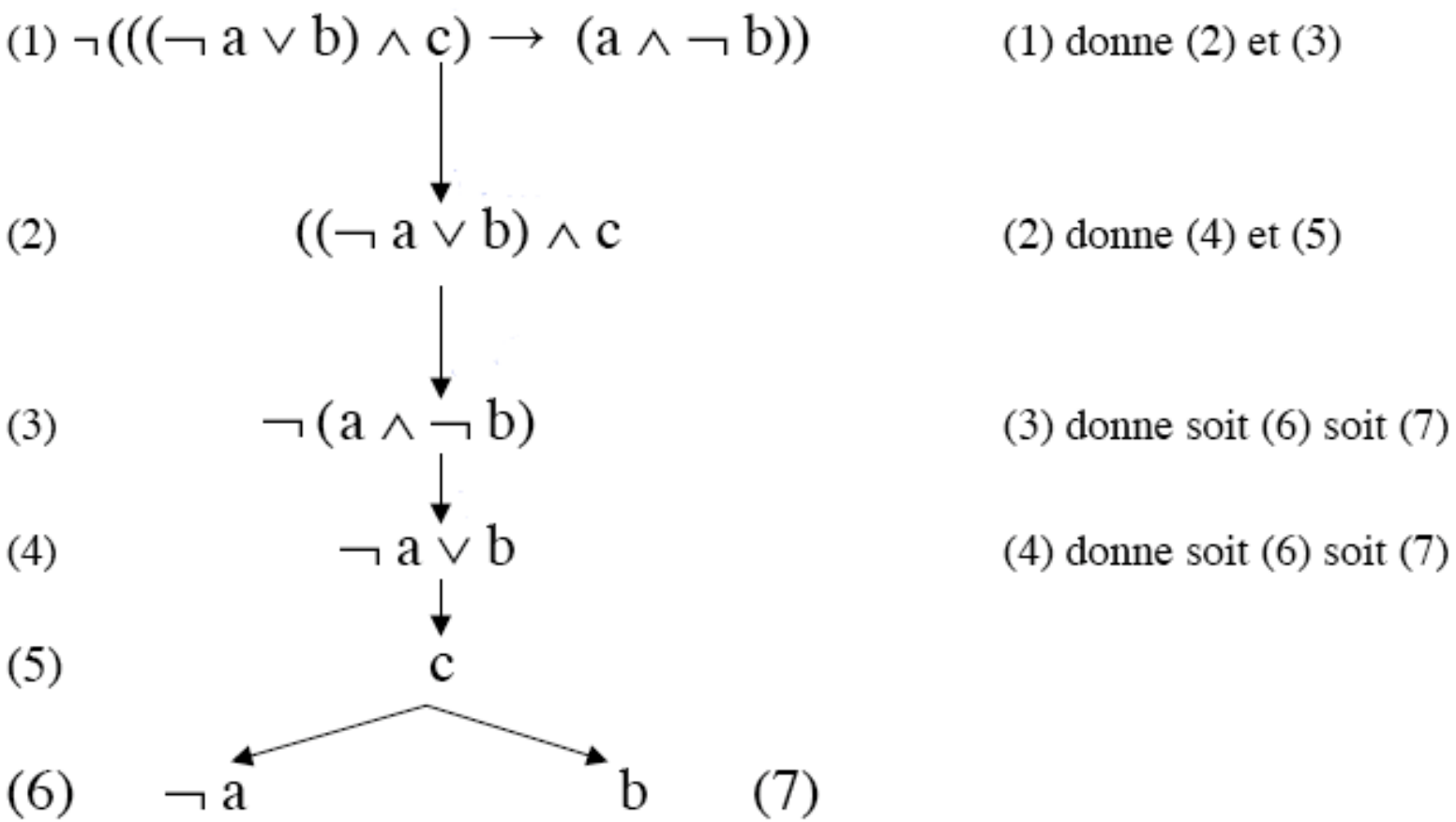
- Ceci permet de trouver une FND de la formule F :

$$F = (a \wedge \neg b) \vee \neg c$$

- Dans l'arbre de F il y a 3 branches mais 2 sont identiques.

Arbres de Beth

Exemple : $\neg F = \neg(((\neg a \vee b) \wedge c) \rightarrow (a \wedge \neg b))$



Arbres de Beth

Exemple : $\neg F$ (suite)

- On a deux branches :
  celle de gauche $c \wedge \neg a$ et celle de droite $c \wedge b$.
- D'où la FND de $\neg F$:

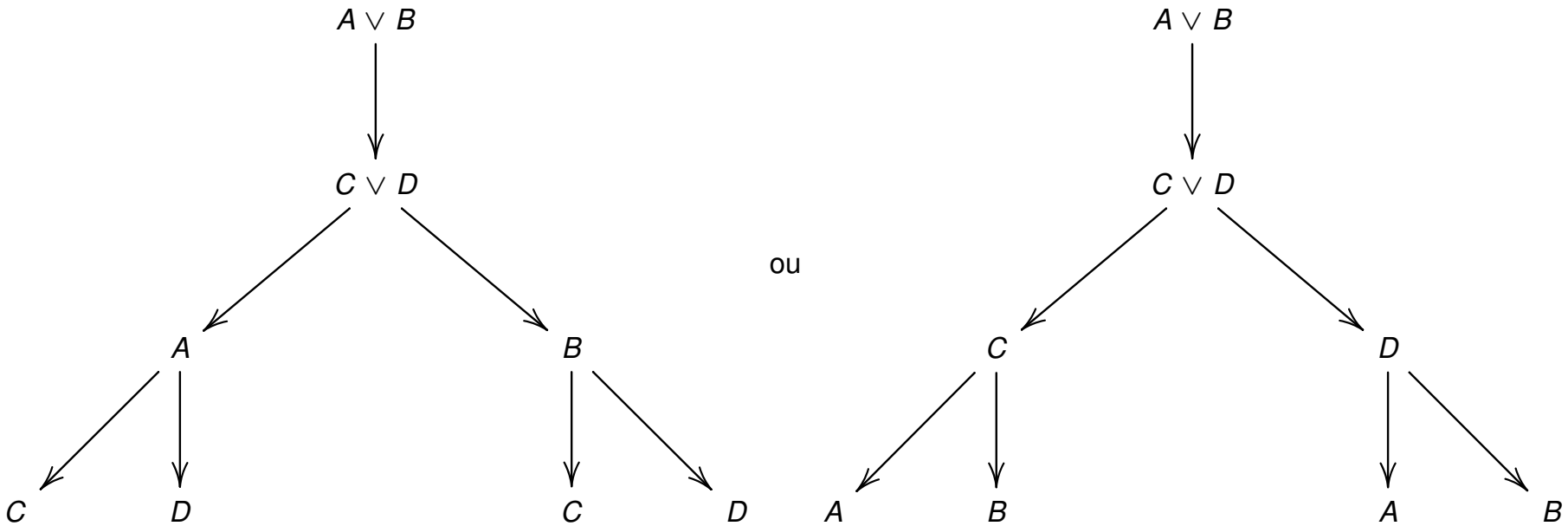
$$\neg F = (c \wedge \neg a) \vee (c \wedge b)$$

- On obtient une FNC de la formule F à partir de la FND de $\neg F$:

$$F = \neg((c \wedge \neg a) \vee (c \wedge b)) = (\neg c \vee a) \wedge (\neg c \vee \neg b)$$

Développer plusieurs “OU”

- Quand il y a plusieurs disjonctions les unes en-dessous des autres, on peut les développer dans l’ordre que l’on veut



- On obtient 4 branches qui nous donnent $(A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (C \wedge D)$

Branches fermées, arbres fermés

- Pour satisfaire une formule qui est à l'origine de l'arbre, il suffit de satisfaire tous les atomes **d'une** des branches de l'arbre.

- Si une branche comporte une variable propositionnelle **et** sa négation, on dit que l'on a une branche fermée.

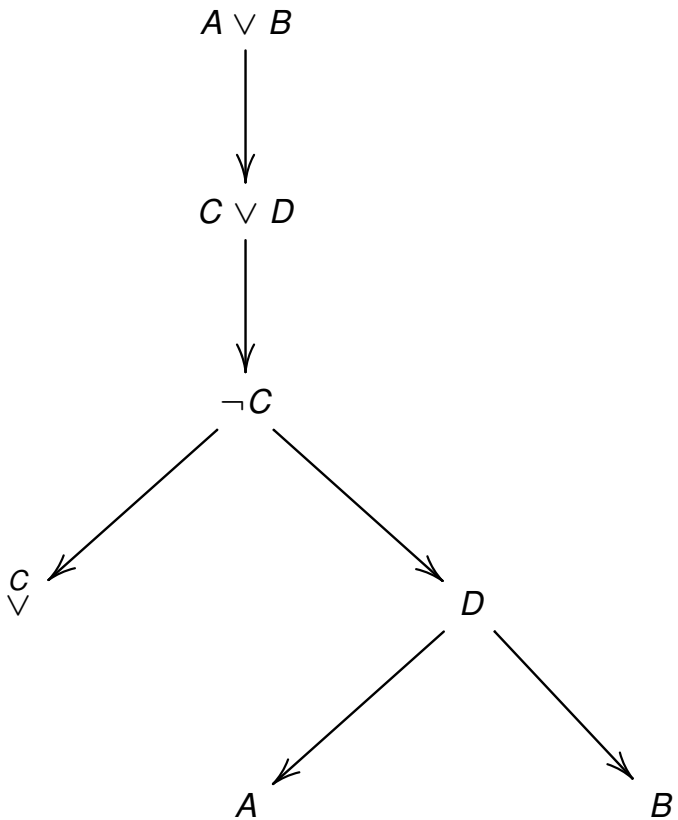
- Si **toutes** les branches d'un arbre sont fermées, on dit que l'on a un arbre fermé.
  Dans ce cas la formule d'origine n'est **pas satisfaisable**, ou **antilogique**.

- Ainsi pour montrer qu'une formule F est **tautologique**, il suffit de former un arbre dont l'origine est la formule $\neg F$:

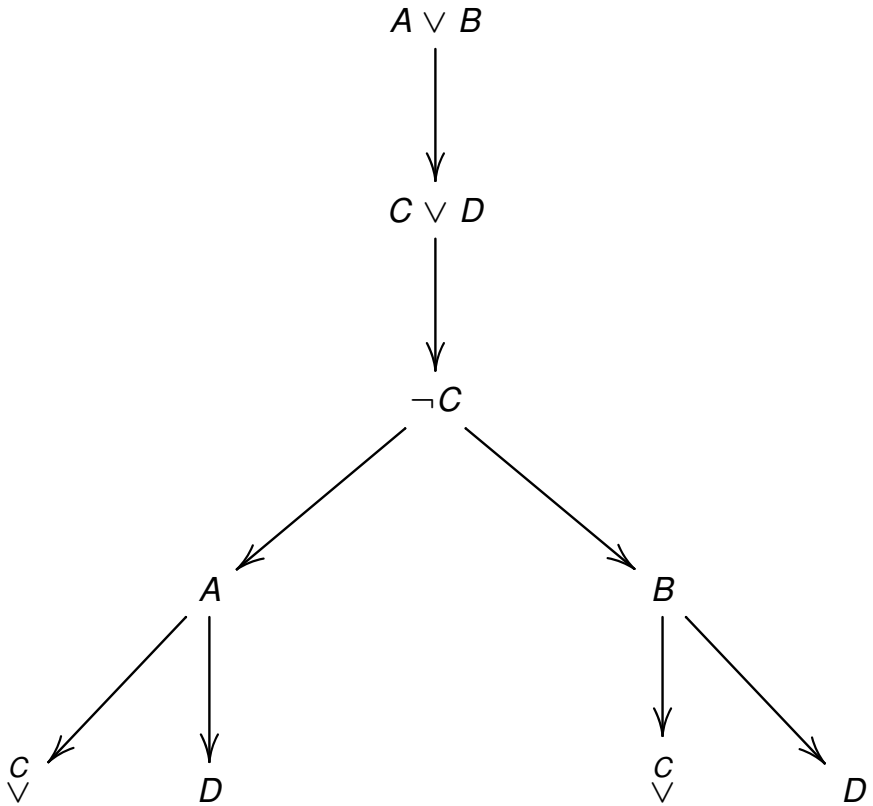
  si l'arbre de $\neg F$ est fermé, F est tautologique puisque $\neg F$ n'est pas satisfaisable.

Développer plusieurs “OU”

- Comme on arrête de développer une branche qui se ferme, on peut en tirer parti pour gagner du temps en choisissant quelles disjonctions développer en premier. Ex : Développement de $(A \vee B) \wedge (C \vee D) \wedge (\neg C)$



plutôt que



- On obtient $\neg C \wedge D \wedge (A \vee B)$ et $\neg C \wedge ((A \vee D) \vee (B \vee D))$, qui sont bien équivalentes

Notion de conséquence logique

Soit Σ un ensemble de formules et C une formule.

- L'ensemble Σ est dit satisfaisable si il existe une distribution de vérité qui satisfait toutes les formules de Σ à la fois :

$$\exists \delta : \forall F \in \Sigma, \delta(F) = 1$$

- La formule C est dite conséquence de Σ si toute distribution satisfaisant toutes les formules de Σ, satisfait aussi C :

$$\forall \delta : (\forall F \in \Sigma, \delta(F) = 1) \Rightarrow \delta(C) = 1$$

  On note $\Sigma \models C$

- L'ensemble Σ est appelé "ensemble d'hypothèses" et C est la "conclusion".

Arbres de Beth et ensemble de formules consistant

- On considère un ensemble de formules $\{H_1, H_2, \ldots, H_n\}$.
- On construit un arbre dont ces formules sont les formules initiales : elles forment le '≪ tronc '≫ de l'arbre (ca revient à écrire l'arbre de $H_1 \wedge H_2 \wedge \cdots \wedge H_n$).
- On développe l'arbre en appliquant les règles à chacune des formules.
  - Si il existe au moins une branche ouverte, alors il existe au moins une distribution qui satisfait toutes les formules $H_1, H_2, \ldots, H_n$.
    L'ensemble de formules est dit consistant.
  - Si toutes les branches de l'arbre sont fermées, les formules $H_1, H_2, \ldots, H_n$ ne peuvent pas être simultanément satisfaites.
    L'ensemble de formules est dit inconsistant ou insatisfaisable.

Arbres de Beth et légitimité d'une déduction

- Pour savoir si, à partir de l'ensemble de formules Σ, on peut déduire la formule C, il faut répondre à la question :

  L'ensemble $\Sigma \cup \{\neg C\}$ est-il consistant ou non consistant ?

- C'est équivalent à $\Sigma \wedge \neg C$ est-il satisfaisable ?

- S'il est **inconsistant**, toutes les branches sont fermées, il n'existe donc aucune distribution de vérité qui satisfasse à la fois les formules de Σ et $\neg C$.

  Donc toute distribution qui satisfait Σ, satisfait aussi C et la déduction est légitime. D'une certaine manière, on peut dire que "Σ implique C"

- S'il est **consistant**, l'arbre possède une branche ouverte, il existe donc une distribution qui satisfait à la fois les hypothèses, les formules de Σ, et la négation de la conclusion $\neg C$.

  Donc la déduction n'est pas légitime.

Arbres de Beth et légitimité d'une déduction

Exemples :

- $(p \wedge q) \models p$ car $(p \wedge q) \wedge \neg p$ n'est pas satisfaisable (c'est une antilogie)
- $p \models (p \vee q)$ car $p \wedge \neg(p \vee q)$ n'est pas satisfaisable (c'est une antilogie)
- $\Sigma = \{p, q \to (p \to r)\}$ et $C = q \to r$, alors $\Sigma \models C$
- Que peut-on dire des déductions suivantes ?
  1. $\{p \to (q \to r),\ \neg s \vee p,\ q\} \models (s \to r)$
  2. $\{(p \wedge q) \to r,\ r \to s,\ q \wedge \neg s\} \models p$

Récréation (revisitée)

Reprenons l'exemple suivant dans le contexte de la déduction :

Lors d'une enquête de l'inspecteur Maigret, les personnes A, B, C et D sont suspectées. Il est établi que :

1. Si A et B sont coupables, il en est de même de C.
2. Si A est coupable, l'un au moins de B et C est aussi coupable.
3. Si C est coupable, D l'est aussi.
4. Si A est innocent, D est coupable.

On note $x =$ "X est coupable" pour $X \in \{A, B, C, D\}$.

Pour

$$\Sigma = \{(A \wedge B) \to C,\ A \to (B \vee C),\ c \to D,\ \neg A \to D\}$$

on a alors que Σ est consistant et que $\Sigma \models d$. En effet,

la conjonction des formules de Σ a comme FND $(\neg A \wedge D) \vee (C \wedge D)$.

Arbres de Beth : remarques importantes

- On peut traiter les formules du tronc dans un ordre quelconque :

  mais afin de simplifier l'arbre, on commence par développer les conjonctions.

- Il peut être judicieux de numéroter les formules et leurs enfants.

- On marque une formule qui a été traitée afin de ne pas la traiter deux fois.

- Lorsque l'on traite une formule, on écrit les sous-formules auxquelles elle donne naissance à l'extrémité de toutes les branches ouvertes qui passent par la formule considérée.

- Dès qu'une branche est fermée, on n'a pas besoin de reproduire dans sa suite d'éventuelles sous-formules.

Propriétés de la déduction

- Si Σ est un ensemble de formules **inconsistant**,
  alors pour toute formule C , la déduction $\Sigma \models C$ est légitime.

  D'un ensemble de formules inconsistant, on peut déduire n'importe quoi.

- Exemple :

  L'ensemble des proverbes est inconsistant. Il contient par exemple '≪ tel père, tel fils '≫ et '≪ à père avare, enfant prodigue '≫ qui sont des énoncés négation l'un de l'autre.

  On peut donc déduire de l'ensemble des proverbes tout énoncé quel qu'il soit.

Propriétés de la déduction

- Si C est une **tautologie**, C est déductible de tout ensemble de formules Σ.
- Soient Σ et Σ' sont deux ensembles de formules,

  $$\text{si } \Sigma \subset \Sigma' \text{ et } \Sigma \models C \text{ alors } \Sigma' \models C \ .$$

  Si une déduction est légitime, on peut toujours ajouter des hypothèses, la déduction reste légitime.
- Une tautologie étant déductible de tout ensemble de formules, elle est en particulier déductible de l'ensemble vide $\emptyset$.

  Il est donc cohérent d'utiliser le même symbole $\models$ pour noter
  - $\Sigma \models C$ pour dire '« C est déductible de Σ »
  - $\models F$ pour dire '« F est tautologique »,
    ce qui peut être vu comme abréviation de $\varnothing \models F$

Théorème de la déduction

- Si $\Sigma = \{H_1, H_2, \ldots, H_n\}$ est un ensemble fini d'hypothèses, il est équivalent d'écrire

$$\{H_1, H_2, \ldots, H_n\} \models C \qquad \text{ou} \qquad H_1 \wedge H_2 \wedge \cdots \wedge H_n \models C$$

- Théorème de la déduction
  Si Σ est un ensemble de formules et H une formule donnée, alors

$$\Sigma \cup \{H\} \models C \text{ si et seulement si } \Sigma \models H \rightarrow C$$

  Ce théorème est évident à partir de la définition sémantique de la déduction, mais ne l'est plus si l'on donne une définition de la déduction purement syntaxique.

- Pour $\Sigma = \varnothing$ on obtient $H \models C$ si et seulement si $\models (H \rightarrow C)$ ce qui exprime la relation qui existe entre le connecteur $\rightarrow$ et la déduction

Quelques exemples de déduction

- Pour tester la déduction $\{p \to (q \to r), \neg s \vee p, q\} \models (s \to r)$
  on peut tester la déduction équivalente :
  $\{p \to (q \to r), \neg s \vee p, q, s\} \models r$
  ce qui augmente l'ensemble d'hypothèses mais simplifie la conclusion.
- Tester par la méthode des arbres la déduction

$$\{p, q \to (p \to r)\} \models (q \to r)$$

- Vérifier que les formules suivantes sont tautologiques

$$[A \to (B \to C)] \to [(A \to B) \to (A \to C)]$$

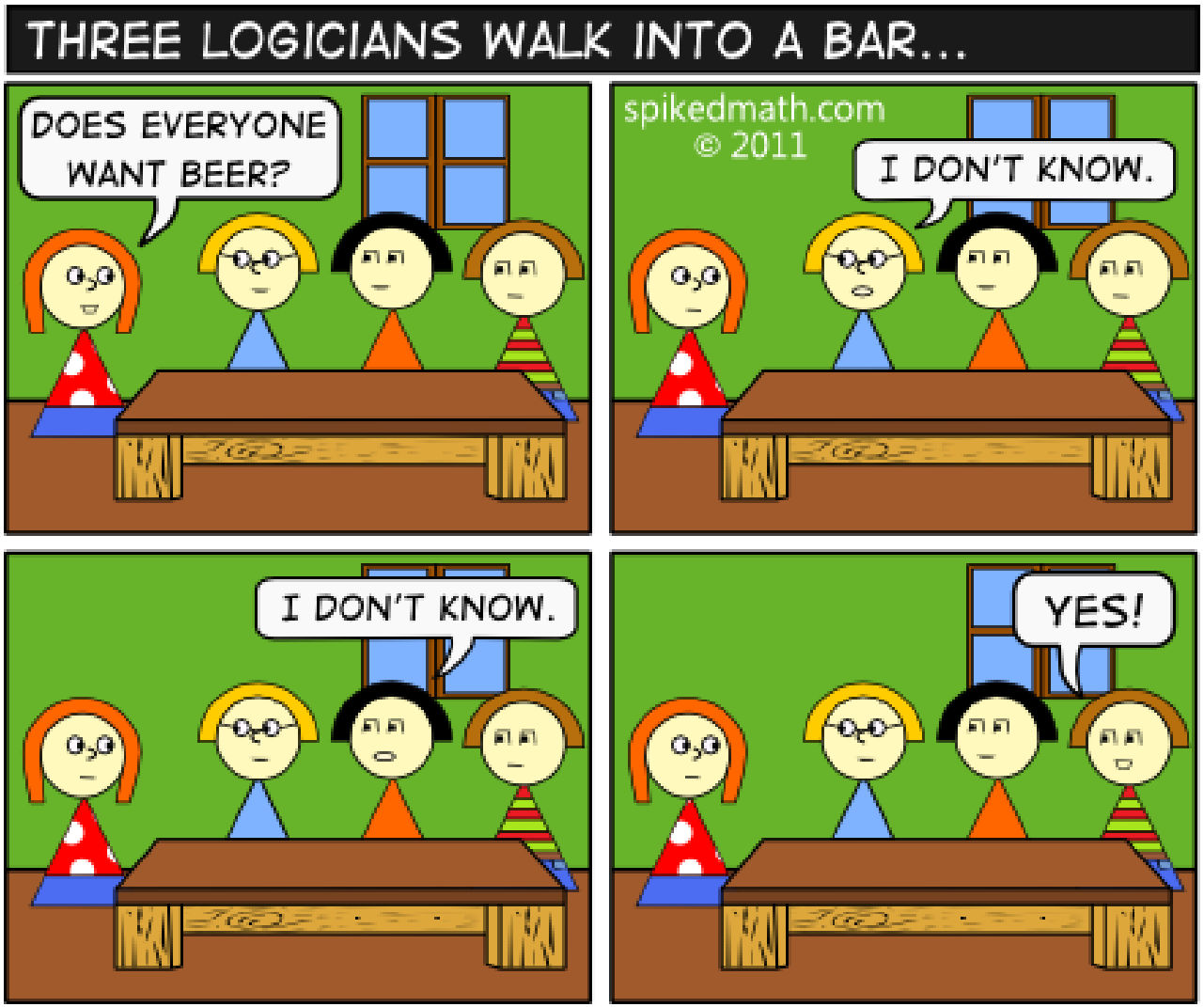
$$(\neg s \to \neg p) \to [(p \to (r \to q)) \to (p \to (p \to s))]$$

- Vérifier que la formule

$$(\neg B \to \neg A) \to [(\neg B \to A) \to B)]$$

  est tautologique.
  De quel schéma de raisonnement s'agit-il ?

Enigme...


THREE LOGICIANS WALK INTO A BAR...
DOES EVERYONE WANT BEER?
spikedmath.com
© 2011
I DON'T KNOW.
I DON'T KNOW.
YES!

Solution

On suppose que C veut de la bière et on suit son raisonnement :

- A : Le 1er prend de la bière
- B : Le second en prend
- T : Tout le monde en prend
- a : Le premier connaît la réponse à la question
- b : Le second connaît la réponse à la question
- On sait $A \wedge B \rightarrow T$ est vraie
- On sait aussi $\neg A \rightarrow (a \wedge \neg T), \neg B \rightarrow (b \wedge \neg T)$

Le cercle des menteurs

100 personnes sont réunies en cercle et chaque personne dit “Mon voisin de gauche ment”.

- On choisit arbitrairement un individu No 1, et on numérote les autres en allant vers la gauche (dans le sens des aiguilles d’une montre)
- On pose $M_i =$“Le i-ème ment toujours”
- On sait que $\neg M_1 \to M_2, \neg M_2 \to M_3, \ldots, \neg M_{100} \to M_1$
- On suppose que 1 dit la vérité. Qui sont alors les menteurs ?
- On suppose que 2 dit la vérité. Qui sont alors les menteurs ?
- Que se passe-t-il si on a le même problème avec 101 personnes ?