Principes de fonctionnement des machines binaires

2019/2020

Pierluigi Crescenzi
Université de Paris, IRIF

Université de Paris

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

cnrs dépasser les frontières

- Tests et examens
  - CC : résultat des tests en TD / TP (**semaine prochaine** et 10)
  - E0 : partiel (samedi 26 octobre)
  - E1 : examen mi décembre
  - E2 : examen fin juin
- Notes finales
  - Note session 1 : 25% CC + 25% E0 + 50% E1
  - Note session 2 : max(E2, 33% CC + 67% E2)
- Rappel
  - Pas de note $\Rightarrow$ pas de moyenne $\Rightarrow$ pas de semestre
- Site web
  - `moodlesupd.script.univ-paris-diderot.fr`

- Numération et arithmétique
- **Numération et arithmétique en machine**
- **Numérisation et codage (texte, images)**
- Compression, cryptographie, contrôle d'erreur
- Logique et calcul propositionnel
- Circuits numériques

- **Représentation en virgule flottante**
  - Chaque nombre réel peut s'écrire de la façon suivante :
$$n = \pm m \times b^e$$
  - m : mantisse
  - b : base
  - e : exposant
- Exemple
  - $(13,11)_{10} = +1,311 \times 10^1 = +0,1311 \times 10^2 = +131,1 \times 10^{-1}$
  - $(-110,101)_2 = -1,10101 \times 2^2 = -0,110101 \times 2^3 = -11010,1 \times 2^{-2}$

- **Norme IEEE 754** (binary32) : représentation sur $n = 32$ bits
  - Exposant (positif ou négatif) représenté sur $p = 8$ bits
    - *Décalé* (positif): on ajoute à l'exposant la valeur $2^{p-1} - 1 = 127$
  - Mantisse sous la forme signe / valeur absolue
    - 1 bit pour le signe et $k = 23$ bits pour la valeur

- **Norme IEEE 754** (binary32) : représentation sur $n = 32$ bits
  - Exposant (positif ou négatif) représenté sur $p = 8$ bits
    - *Décalé* (positif): on ajoute à l'exposant la valeur $2^{p-1} - 1 = 127$
  - Mantisse sous la forme signe / valeur absolue
    - 1 bit pour le signe et $k = 23$ bits pour la valeur
  - Un nombre flottant normalisé a une valeur v donnée par la formule suivante : $v = s \times 2^e \times m$
    - s = ±1 représente le signe (selon le bit de signe)
    - e est l'exposant avant son décalage de 127
    - $m = 1 + \mathrm{mantisse}$ représente la partie significative (en binaire), d'où $1 \leq m < 2$
      - Mantisse est la partie décimale de la partie significative, comprise entre 0 et 1

- Exemple : $-118,625$

- Exemple : $-118,625$
  - Binaire : $-1110110,101$

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6+127=133=(10000101)_2$

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6+127=133=(10000101)_2$
  - Représentation : $1\ 10000101\ 11011010100000000000000$

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6 + 127 = 133 = (10000101)_2$
  - Représentation : 1 10000101 11011010100000000000000
- Exemple : 0 01111100 01000000000000000000000

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6+127=133=(10000101)_2$
  - Représentation : $1\ 10000101\ 11011010100000000000000$
- Exemple : $0\ 01111100\ 01000000000000000000000$
  - Signe : positif

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6+127=133=(10000101)_2$
  - Représentation : $1\ 10000101\ 11011010100000000000000$
- Exemple : $0\ 01111100\ 01000000000000000000000$
  - Signe : positif
  - Exposant : $e=124-127=-3$

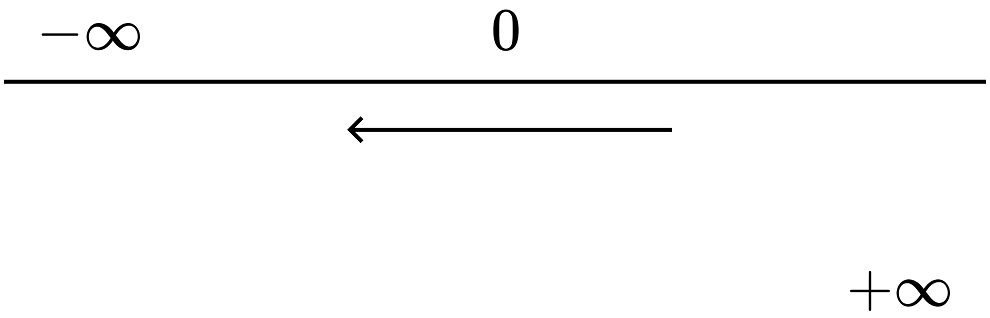
- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6+127=133=(10000101)_2$
  - Représentation : 1 10000101 11011010100000000000000
- Exemple : 0 01111100 01000000000000000000000
  - Signe : positif
  - Exposant : $e=124-127=-3$
  - Mantisse : $(1,01)_2=1,25$

- Exemple : $-118,625$
  - Binaire : $-1110110,101$
  - Bit de signe : 1
  - Mantisse : $1,110110101$
  - Exposant : $6+127=133=(10000101)_2$
  - Représentation : $1\ 10000101\ 11011010100000000000000$
- Exemple : $0\ 01111100\ 01000000000000000000000$
  - Signe : positif
  - Exposant : $e=124-127=-3$
  - Mantisse : $(1,01)_2=1,25$
  - Nombre représenté : $+1,25\times 2^{-3}$ soit $+0,15625$

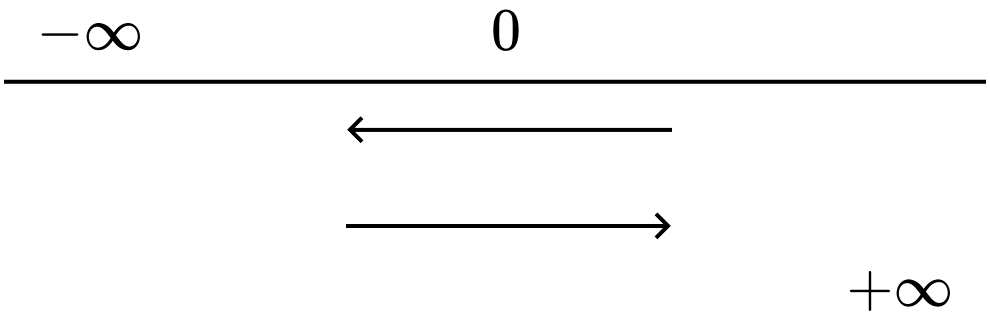
- Exceptions
  - $e = 2^{p-1} - 1$ et $m = 0$: ±∞ (selon le bit de signe)
  - $e = 2^{p-1} - 1$ et $m \neq 0$: NaN (Not-a-Number)

- Exceptions
  - $e = 2^{p-1} - 1$ et $m = 0$: $\pm\infty$ (selon le bit de signe)
  - $e = 2^{p-1} - 1$ et $m \neq 0$: NaN (Not-a-Number)
- 4 modes d'arrondi

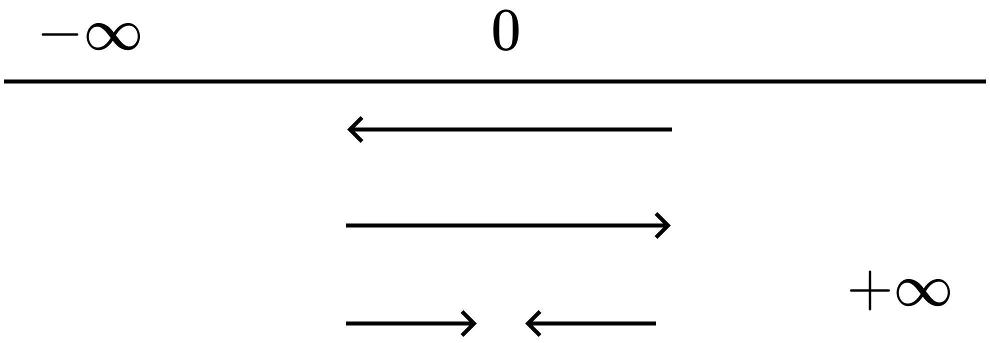
- Exceptions
  - $e = 2^{p-1} - 1$ et $m = 0$: ±∞ (selon le bit de signe)
  - $e = 2^{p-1} - 1$ et $m \neq 0$: NaN (Not-a-Number)
- 4 modes d'arrondi
  - Vers moins l'infini



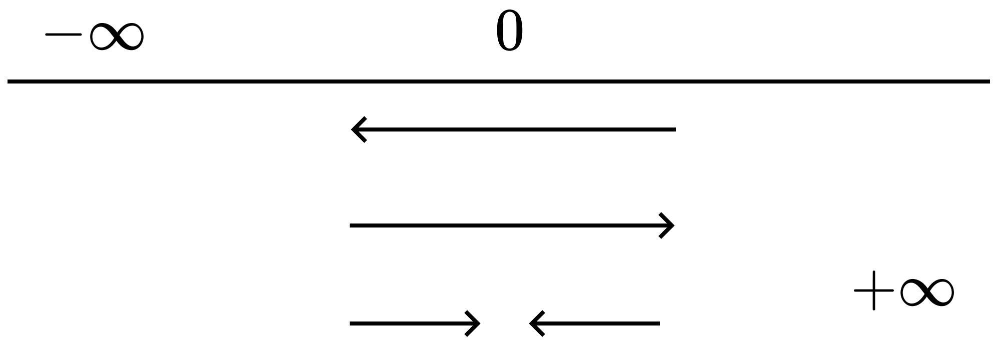
- Exceptions
  - $e = 2^{p-1} - 1$ et $m = 0$: ±∞ (selon le bit de signe)
  - $e = 2^{p-1} - 1$ et $m \neq 0$: NaN (Not-a-Number)
- 4 modes d'arrondi
  - Vers moins l'infini
  - Vers plus l'infini



- Exceptions
  - $e = 2^{p-1} - 1$ et $m = 0$: ±∞ (selon le bit de signe)
  - $e = 2^{p-1} - 1$ et $m \neq 0$: NaN (Not-a-Number)
- 4 modes d'arrondi
  - Vers moins l'infini
  - Vers plus l'infini
  - Vers zéro



- Exceptions
  - $e = 2^{p-1} - 1$ et $m = 0$: ±∞ (selon le bit de signe)
  - $e = 2^{p-1} - 1$ et $m \neq 0$: NaN (Not-a-Number)
- 4 modes d'arrondi



  - Vers moins l'infini
  - Vers plus l'infini
  - Vers zéro
  - Au plus près (par défaut)
    - $1, a_{-1}a_{-2}\ldots a_{-22}a_{-23}\mathrm{GRS}$
      - $\mathrm{G} = 0 : 1, a_{-1}a_{-2}\ldots a_{-22}a_{-23}$
      - $\mathrm{GRS} = 100$ et $a_{-23} = 0 : 1, a_{-1}a_{-2}\ldots a_{-22}a_{-23}$
      - Sinon : $1, a_{-1}a_{-2}\ldots a_{-22}a_{-23} + 0,\underbrace{0\cdots 0}_{22}1$
        - Règles de débordement

- Qu'affichera l'execution de code suivant ?

```
float x = 0.3F;
float y = 0.2F;
float z = 0.1F;
System.out.println((x-y)==(y-z));
```

- Comment le réel 0,3 est codé selon IEEE 754
  - $(0,3)_{10} = 0,0(1001)^{\omega}$
  - Normalisé : $1,(0011)^{\omega} \times 2^{-2}$
  - Signe : 0
  - Exposant : $-2 + 127 = 125 \rightarrow 01111101$
  - Mantisse (au plus près) : $001100110011001100110\underline{10}$
  - Représentation : $0\ 01111101\ 00110011001100110011010$

- Représentation 0,3 : 0 01111101 00110011001100110011010
  - $1,00110011001100110011010 \times 2^{-2}$

- Représentation 0,3 : 0 01111101 00110011001100110011010
  - $1,00110011001100110011010 \times 2^{-2}$
- Représentation 0,2 : 0 01111100 10011001100110011001101
  - $1,10011001100110011001101 \times 2^{-3}$
  - Même exposant : $0,110011001100110011001100110 \times 2^{-2}$

- Représentation 0,3 : 0 01111101 00110011001100110011010
  - $1,00110011001100110011010 \times 2^{-2}$
- Représentation 0,2 : 0 01111100 10011001100110011001101
  - $1,10011001100110011001101 \times 2^{-3}$
  - Même exposant : $0,11001100110011001100110 \times 2^{-2}$
- $0,3 - 0,2$:

```
1,00110011001100110011010 -
0,11001100110011001100110 =
0,01100110011001100110100
```

- Représentation 0,3 : 0 01111101 00110011001100110011010
  - $1,00110011001100110011010 \times 2^{-2}$
- Représentation 0,2 : 0 01111100 10011001100110011001101
  - $1,10011001100110011001101 \times 2^{-3}$
  - Même exposant : $0,110011001100110011001110 \times 2^{-2}$
- $0,3 - 0,2$:

```
1,00110011001100110011010 -
0,11001100110011001100110 =
0,01100110011001100110100
```

- Normalisé : $1,10011001100110011010000 \times 2^{-4}$

- Représentation 0,3 : 0 01111101 00110011001100110011010
  - $1,00110011001100110011010 \times 2^{-2}$
- Représentation 0,2 : 0 01111100 10011001100110011001101
  - $1,10011001100110011001101 \times 2^{-3}$
  - Même exposant : $0,11001100110011001100110 \times 2^{-2}$
- $0,3 - 0,2$:

```
1,00110011001100110011010 -
0,11001100110011001100110 =
0,01100110011001100110100
```

- Normalisé : $1,10011001100110011010000 \times 2^{-4}$
- Représentation : $0\ 01111011\ 10011001100110011010000$
  - Decimal : $\approx 0,10000001$

Numérisation et codage

- Codage : conversion d'une représentation en une autre
  - Code Morse
    - SOS ⇒ . . . _ _ _ . . .
- Le codage est utilisé à différentes fins
  - Représenter quelque chose dans un système numérique
  - Économiser de l'espace (compression de données)
  - Rendre illisibles aux non initiés des données (cryptographie)
  - Résister aux altérations, pertes ou mutations (codes correcteurs)

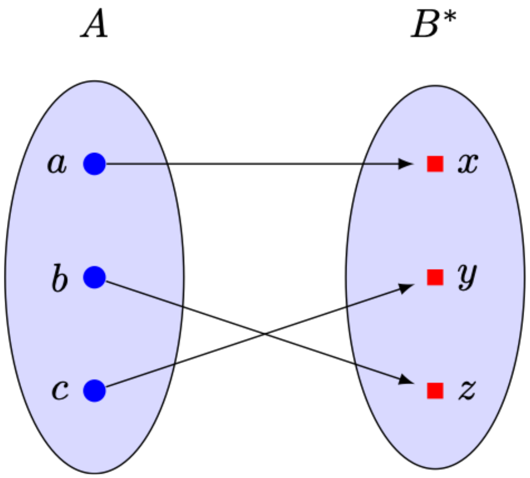
- En informatique
  - Il s’agit de coder de l’information sous la forme d’une suite finie de 0 et 1
    - Comment coder l’alphabet latin ?
    - Comment coder le signal numérisé d’une chanson ?
    - Comment coder l’image prise par un appareil photographique ?
    - Comment crypter le contenu de ma clé USB ?
    - Comment assurer que l’image prise par Curiosity sur la planète Mars parvienne sans altération jusqu’à nous ?

- **Alphabet** de n lettres : $A = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$
- **Mot** : suite finie de lettres $m = m_0 m_1 \ldots m_{l-1}$ où $m_i \in A$
  - l : **longueur** du mot
- A^* : l'ensemble des mots pouvant être écrits avec l’alphabet A
  - Tous les mots de longueur 0 (une mot noté ε)
  - Tous les mots de longueur 1
  - Tous les mots de longueur 2

- **Alphabet** de n lettres : $A = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$
- **Mot** : suite finie de lettres $m = m_0 m_1 \ldots m_{l-1}$ où $m_i \in A$
  - l : **longueur** du mot
- A^* : l'ensemble des mots pouvant être écrits avec l’alphabet A
  - Tous les mots de longueur 0 (une mot noté ε)
  - Tous les mots de longueur 1
  - Tous les mots de longueur 2
- Exemple : $A = \{0, 1\}$
  - 01010010 : mot de longueur 8
  - $A^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}$

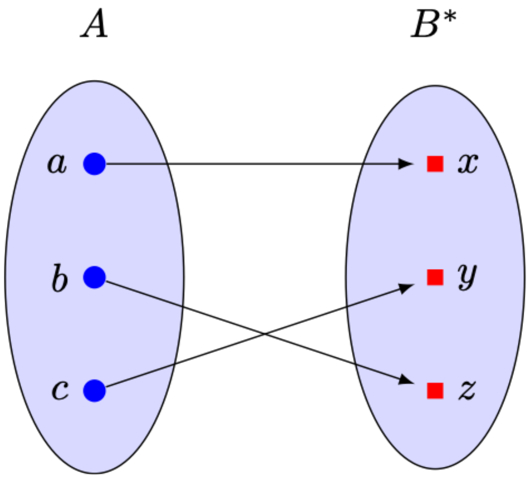
- La représentation de mots écrits dans un alphabet donné A en mots sur un autre alphabet B

- La représentation de mots écrits dans un alphabet donné A en mots sur un autre alphabet B



- On définit le codage des lettres de A en des mots de B^* à l'aide d'une fonction $\tau : A \to B^*$

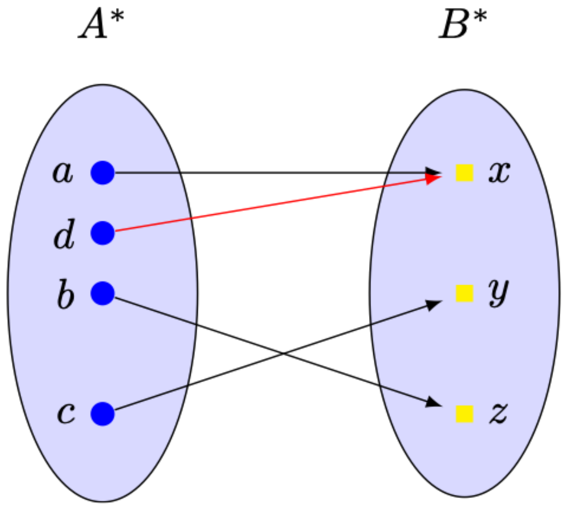
- La représentation de mots écrits dans un alphabet donné A en mots sur un autre alphabet B



- On définit le codage des lettres de A en des mots de B^* à l'aide d'une fonction $\tau : A \to B^*$

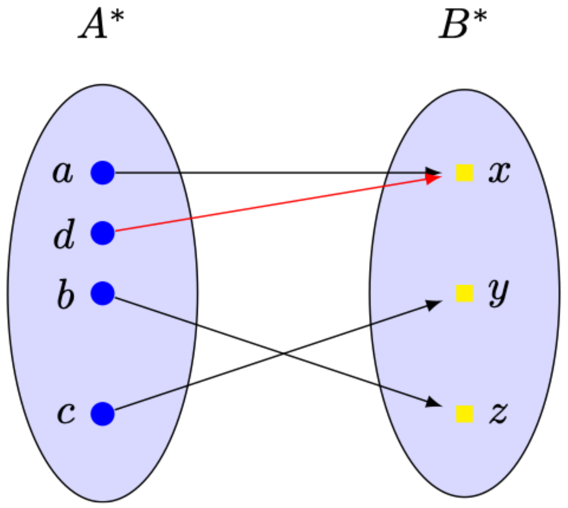
- Le codage par τ d'un mot $m = m_0 m_1 \dots m_{l-1}$ de A^*, consiste à coder chaque lettre et rabouter les codages dans l'ordre
$$\tau(m) = \tau(m_0)\tau(m_1)\dots\tau(m_{l-1})$$
  - On identifie τ sur les mots et τ sur les lettres

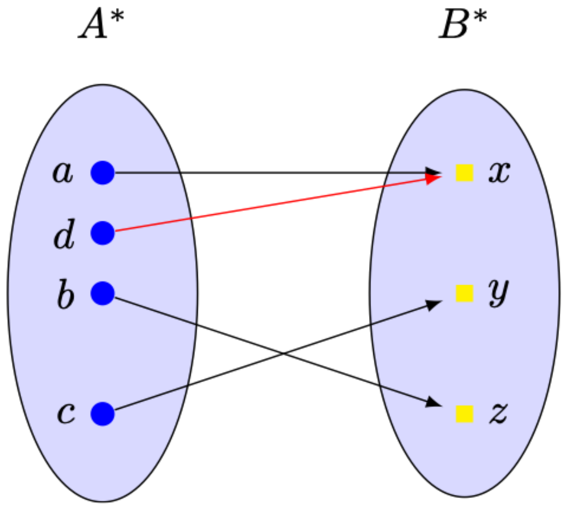
- τ doit être inversible !
  - τ injective (sur les mots)



- τ doit être inversible !
  - τ injective (sur les mots)

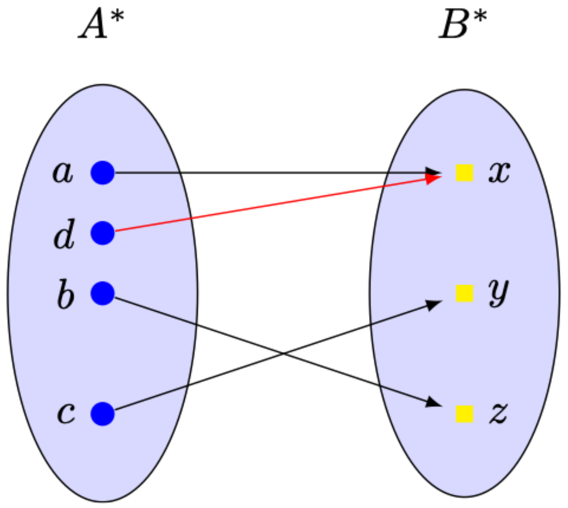
- $A = \{a, b, c\}$ et $B = \{0, 1\}$





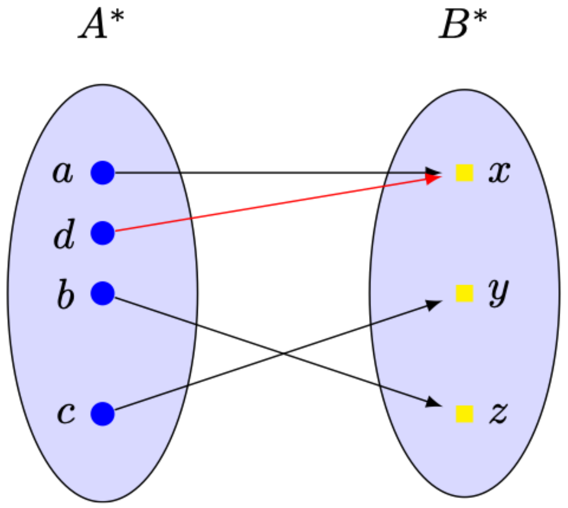
- τ doit être inversible !
  - τ injective (sur les mots)

- $A = \{a, b, c\}$ et $B = \{0, 1\}$
  - $\tau(a) = 00$, $\tau(b) = 11$, $\tau(c) = 111110$
    - Est une fonction de codage acceptable



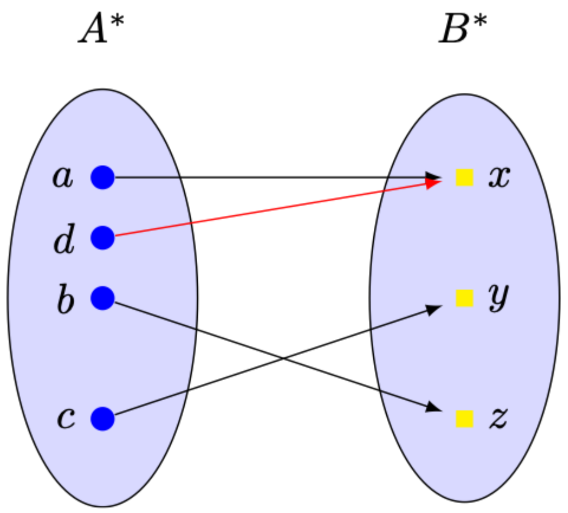
- τ doit être inversible !
  - τ injective (sur les mots)

- $A = \{a, b, c\}$ et $B = \{0, 1\}$
  - $\tau(a) = 00$, $\tau(b) = 11$, $\tau(c) = 111110$
    - Est une fonction de codage acceptable
      - 111100111110 c'est ...



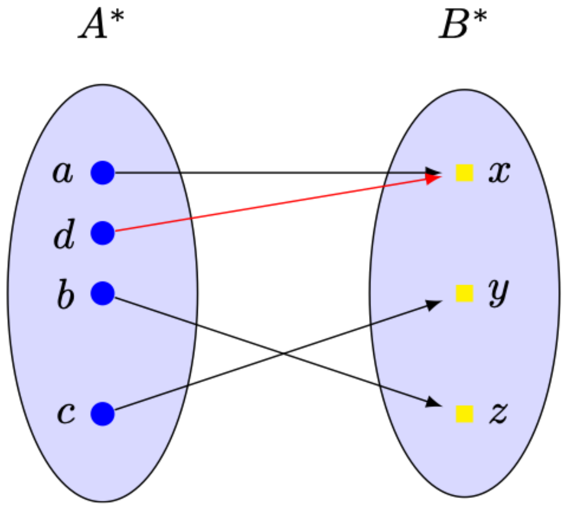
- τ doit être inversible !
  - τ injective (sur les mots)

- $A = \{a, b, c\}$ et $B = \{0, 1\}$
  - $\tau(a) = 00$, $\tau(b) = 11$, $\tau(c) = 111110$
    - Est une fonction de codage acceptable
      - 111100111110 c'est ...
        - $bbac$



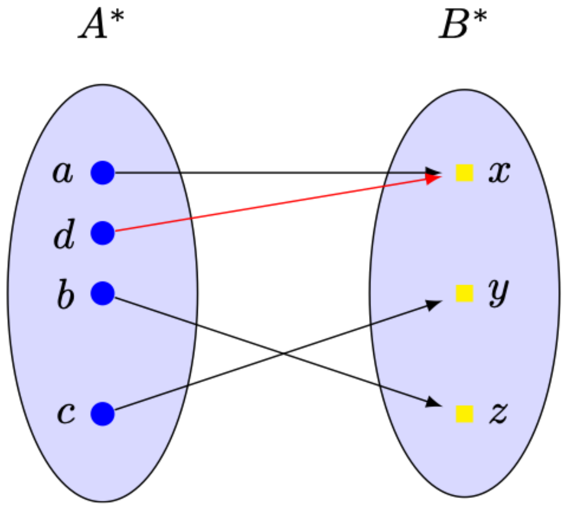
- τ doit être inversible !
  - τ injective (sur les mots)

- $A = \{a, b, c\}$ et $B = \{0, 1\}$
  - $\tau(a) = 00$, $\tau(b) = 11$, $\tau(c) = 111110$
    - Est une fonction de codage acceptable
      - 111100111110 c'est ...
        - $bbac$
  - $\tau(a) = 0$, $\tau(b) = 01$, $\tau(c) = 10$
    - N'est pas une fonction acceptable



- τ doit être inversible !
  - τ injective (sur les mots)

- $A = \{a, b, c\}$ et $B = \{0, 1\}$
  - $\tau(a) = 00$, $\tau(b) = 11$, $\tau(c) = 111110$
    - Est une fonction de codage acceptable
      - 111100111110 c'est ...
        - $bbac$
  - $\tau(a) = 0$, $\tau(b) = 01$, $\tau(c) = 10$
    - N'est pas une fonction acceptable
      - 010 c'est ...



- τ doit être inversible !
  - τ injective (sur les mots)

- $A = \{a, b, c\}$ et $B = \{0, 1\}$
  - $\tau(a) = 00$, $\tau(b) = 11$, $\tau(c) = 111110$
    - Est une fonction de codage acceptable
      - 111100111110 c'est ...
        - $bbac$
  - $\tau(a) = 0$, $\tau(b) = 01$, $\tau(c) = 10$
    - N'est pas une fonction acceptable
      - 010 c'est ...
        - ac ou ba ?

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$
  - $A = \{a, b, c\}$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0,1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0,1\}$
  - $A = \{a,b,c\}$
    - Il faut $k = 2$: $\tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0,1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0,1\}$
  - $A = \{a,b,c\}$
    - Il faut $k = 2$: $\tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a,b,\ldots,z\}$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$
  - $A = \{a, b, c\}$
    - Il faut $k = 2$: $\tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a, b, \ldots, z\}$
    - Il faut $k = 5$: $\tau(a) = 00000$, $\tau(b) = 00001$, …, $\tau(z) = 11001$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$
  - $A = \{a, b, c\}$
    - Il faut $k = 2 : \tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a, b, \ldots, z\}$
    - Il faut $k = 5 : \tau(a) = 00000$, $\tau(b) = 00001$, …, $\tau(z) = 11001$
  - $A = \{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$
  - $A = \{a, b, c\}$
    - Il faut $k = 2 : \tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a, b, \ldots, z\}$
    - Il faut $k = 5 : \tau(a) = 00000$, $\tau(b) = 00001$, …, $\tau(z) = 11001$
  - $A = \{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$
    - Il faut $k = 6$

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$
  - $A = \{a, b, c\}$
    - Il faut $k = 2 : \tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a, b, \ldots, z\}$
    - Il faut $k = 5 : \tau(a) = 00000$, $\tau(b) = 00001$, …, $\tau(z) = 11001$
  - $A = \{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$
    - Il faut $k = 6$
- Le décodage est facile à partir du découpage du mot image en blocs de k lettres

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0,1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0,1\}$
  - $A = \{a,b,c\}$
    - Il faut $k = 2 : \tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a,b,\ldots,z\}$
    - Il faut $k = 5 : \tau(a) = 00000$, $\tau(b) = 00001$, ..., $\tau(z) = 11001$
  - $A = \{A,B,\ldots,Z,a,b,\ldots,z,0,1,\ldots,9\}$
    - Il faut $k = 6$
- Le décodage est facile à partir du découpage du mot image en blocs de k lettres
  - $A = \{a,b,c\}$ et $B = \{0,1\}$ et $\tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$

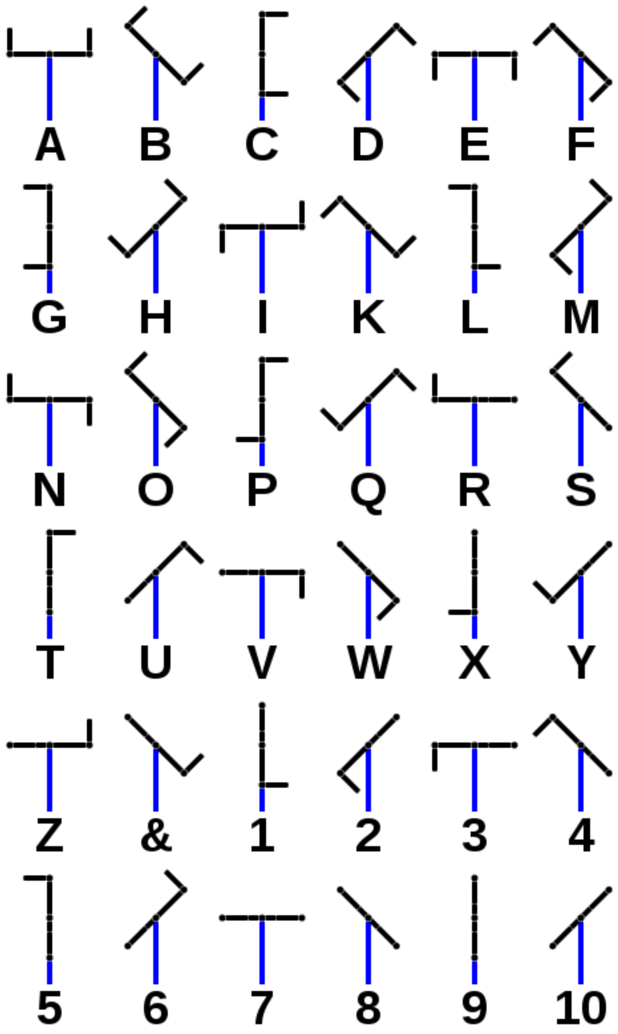
- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0, 1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0, 1\}$
  - $A = \{a, b, c\}$
    - Il faut $k = 2 : \tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a, b, \ldots, z\}$
    - Il faut $k = 5 : \tau(a) = 00000$, $\tau(b) = 00001$, …, $\tau(z) = 11001$
  - $A = \{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$
    - Il faut $k = 6$
- Le décodage est facile à partir du découpage du mot image en blocs de k lettres
  - $A = \{a, b, c\}$ et $B = \{0, 1\}$ et $\tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $000000010010 = 00\ 00\ 00\ 01\ 00\ 10$ c'est …

- Code de **longueur fixe**
  - Toutes les images par τ sont de même longueur k
    - Si $B = \{0,1\}$, il suffit d'avoir $k \geq \log_2(|A|)$ pour coder A sur B^*
- Exemples avec $B = \{0,1\}$
  - $A = \{a,b,c\}$
    - Il faut $k = 2 : \tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $A = \{a,b,\ldots,z\}$
    - Il faut $k = 5 : \tau(a) = 00000$, $\tau(b) = 00001$, …, $\tau(z) = 11001$
  - $A = \{A,B,\ldots,Z,a,b,\ldots,z,0,1,\ldots,9\}$
    - Il faut $k = 6$
- Le décodage est facile à partir du découpage du mot image en blocs de k lettres
  - $A = \{a,b,c\}$ et $B = \{0,1\}$ et $\tau(a) = 00$, $\tau(b) = 01$, $\tau(c) = 10$
  - $000000010010 = 00\ 00\ 00\ 01\ 00\ 10$ c'est …
    - $aaabac$

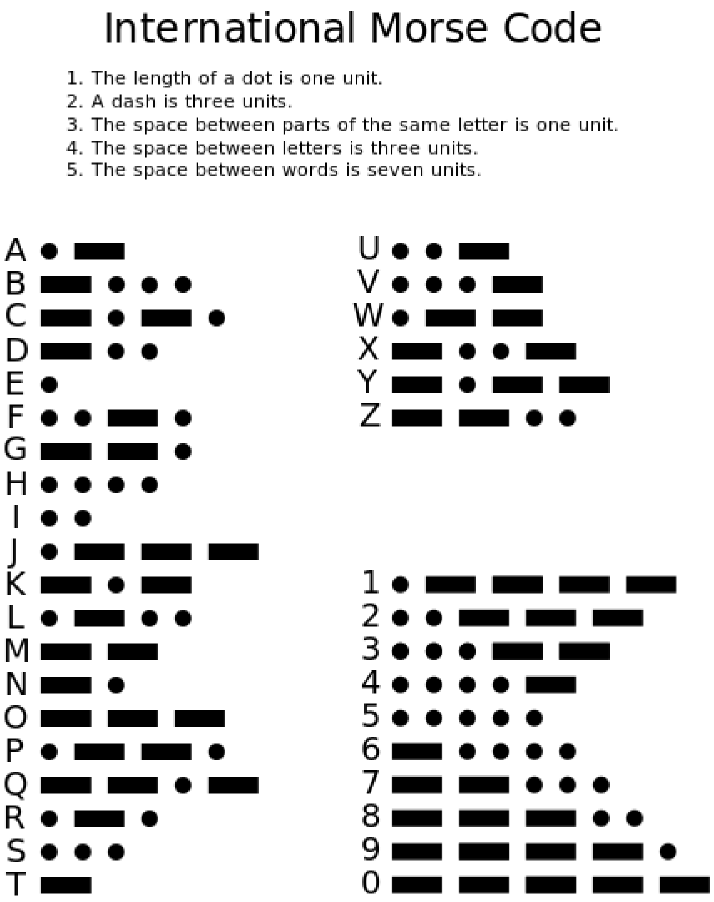
- Codes à **longueur variable**
  - Ces codes sont plus difficiles à construire
  - Les codes **préfixes** sont des codes de longueur variable pas trop difficiles
    - Un code préfixe est un code pour lequel aucune image d'une lettre n'est le préfixe de l'image d'une autre lettre
  - Exemple
    - $A = \{a, b, c, d\}$ et $B = \{0, 1\}$
    - $\tau(a) = 0$, $\tau(b) = 10$, $\tau(c) = 110$, $\tau(d) = 1110$
  - Ils sont très utiles si la fréquence d'apparition des symboles de A n'est pas uniforme
    - Code de Huffman (on verra bientôt)

- Les premiers codes

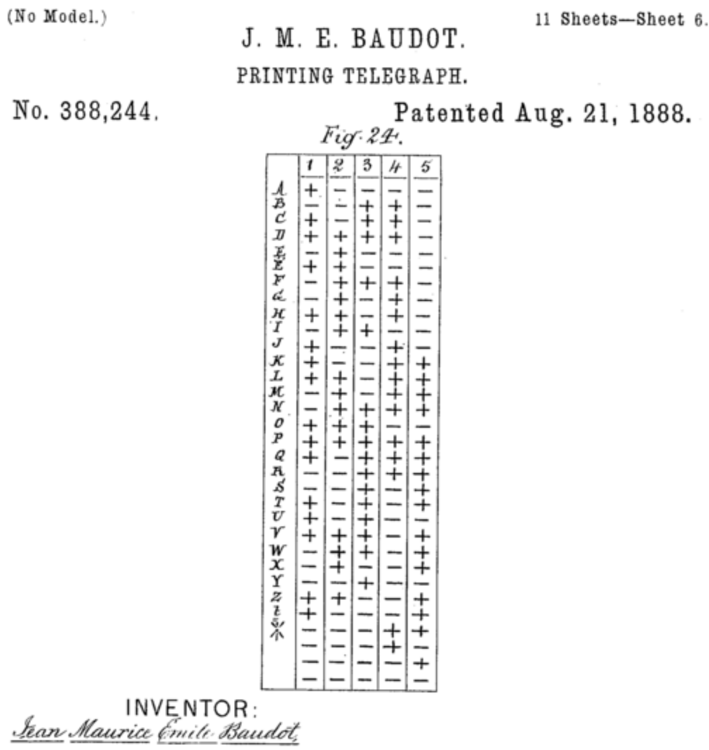
- Les premiers codes
  - Le télégraphe de Claude Chappe (1794)
    - À l'aide des éléments mobiles



- Les premiers codes
  - Le télégraphe de Claude Chappe (1794)
    - À l'aide des éléments mobiles
  - Le code Morse (attribué à Samuel Morse, 1832)
    - Indépendant du support de transmission
    - Il code dans un alphabet binaire (code de longueur variable)



- Les premiers codes
  - Le télégraphe de Claude Chappe (1794)
    - À l’aide des éléments mobiles
  - Le code Morse (attribué à Samuel Morse, 1832)
    - Indépendant du support de transmission
    - Il code dans un alphabet binaire (code de longueur variable)
  - Le code Baudot (Émile Baudot, 1874)
    - Trois fois plus rapide que le code Morse
    - Il a constitué la première normalisation d’un alphabet numérique international
    - Le baud est une unité de mesure en transmission (nombre de symboles / secondes)



- Les demandes de normalisation de codage d'alphabets
  - Baudot (1874, 5 bits)
  - BCD (1954, 6 Bits)
  - 7-bits ASCII (1972, norme ISO/CEI 646)
  - 8-bits ISO/CEI-8859-1 dit Latin-1 (1986) ou ISO/CEI-8859-15 dit Latin-9 (1998)
    - Des extensions au code ASCII
  - 16-bits UNICODE et 32-bits UNICODE (1987)
    - Définit les jeux de caractères, leur numérotation et nommage, etc
    - Le codage peut-être de longueur variable (UTF-8 (8, 16, 24, 32 bits), UTF-16 (16, 32 bits)) ou de longueur fixe UTF-32 (32 bits)
    - Java utilise le jeu de caractères Unicode encodé via UCS-2 (16 bits)

- Le code ASCII originel (American Standard Code for Information Interchange)
  - Codage des caractères alphabétiques latins non accentués, majuscules et minuscules, chiffres, signes et symboles annexes, caractères spéciaux dits de contrôle (94 caractères)
  - Sur 7 bits, ou sur 8 bits avec le bit de poids fort égal à 0
    - On utilisait parfois ce huitième bit pour réaliser une somme de contrôle, permettant de valider la transmission

USASCII code chart

b_4	b_3	b_2	b_1	$b_7 b_6 b_5$ → / Row ↓ Column →	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

- ASCII étendu (ISO/CEI 8859-1)
  - 191 caractères
  - Où est œ ? pas encore €

ISO/CEI 8859-1																
	x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
0x	*positions inutilisées*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*positions inutilisées*															
9x																
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

- ASCII étendu (ISO/CEI 8859-15)
  - 191 caractères
  - Ne permet pas de supporter plusieurs langues en même temps
  - Passage à un codage plus long
    - Le type `char` en Java est sur 16 bits (encodage UCS-2)

ISO/CEI 8859-15

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*non utilisé*															
1x																
2x		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*non utilisé*															
9x																
Ax		¡	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

- Java et les caractères
  - Le type `char` est en fait un type entier 16 bits nonsignés dont la représentation utilise l'encodage UNICODE UCS-2
  - Un littéral de caractères est
    - Directement le caractère entouré de simples apostrophes comme 'A' ou 'z' ou encore 'ä'
    - Le code UNICODE du caractère entouré d'apostrophes et préfixé par \u comme '\u011F'
  - Dans certains cas il est utile de préfixer le caractère souhaité à l'aide du caractère \
    - Comme pour le littéral de caractère représentant l'apostrophe ou le caractère '\' lui-même
    - Ou pour représenter des caractères spéciaux comme le passage à la ligne ou la tabulation
  - Java autorise l'utilisation de certains caractères UNICODE UCS-2 y compris pour les identificateurs
    - Il ne faut pas trop en abuser
    - L'insertion dans un fichier de caractères UCS-2 est dépendante du logiciel et du système hôte

- Exemple

```
System.out.println('A');
System.out.println('ä');
System.out.println('\u00E4');
System.out.println('\\');
System.out.println("C1\tC2\nC3\tC4");
```

- Les pages webs d'Internet utilisent le langage de description HTML pour structurer le contenu (basiquement : du texte)
  - Le texte est nécessairement encodé, c'est pourquoi HTML permet de préciser quel encodage a été utilisé
- HTML permet de spécifier l'encodage via
  - Pour HTML4

    ```
    <meta http-equiv="Content-Type"
    content="text/html;charset="ISO-8859-1"/>
    ```
  - Pour HTML5

    ```
    <meta charset="UTF-8"/>
    ```
  - Pour XHTML5

    ```
    <?xml version="1.0" encoding="UTF-8"?>
    ```
  - Si l'on indique rien c'est UTF-8 pour HTML5 et ISO-8859-1 pour HTML < 5 par défaut