Représentation des données en machine

RÉSUMÉ

Quand on tape le nombre 42 ou la lettre A au clavier, l'ordinateur ne voit ni chiffres ni lettres : il ne comprend que deux signaux, 0 et 1. Comment fait-il alors pour stocker un nombre, un texte ou une image ? Il les traduit en longues suites de 0 et de 1, selon des conventions précises. Comprendre ces conventions, c'est comprendre *pourquoi* certaines opérations donnent des résultats surprenants (par exemple $0{,}1 + 0{,}2 \neq 0{,}3$) et *comment* les données circulent réellement dans la machine.

① MÉMO

Écriture binaire (base 2)

Un ordinateur ne manipule que des **bits** (0 ou 1). Tout nombre entier positif s'écrit en base 2 à l'aide des puissances de 2 :

$$(13)_{10} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (1101)_2$$

Vers le binaire On effectue des divisions euclidiennes successives par 2 et on lit les restes de bas en haut.
Vers le décimal On additionne les puissances de 2 correspondant aux bits à 1.
En Python `bin(13)` renvoie `'0b1101'` ; `int('1101', 2)` renvoie `13`.

Écriture hexadécimale (base 16)

La base 16 utilise les chiffres `0` à `9` puis les lettres `A` à `F` (valeurs 10 à 15). Chaque chiffre hexadécimal correspond exactement à quatre bits :

$$(2\mathrm{F})_{16} = 2 \times 16^1 + 15 \times 16^0 = (47)_{10} = (0010\ 1111)_2$$

En Python `hex(47)` renvoie `'0x2f'` ; `int('2F', 16)` renvoie `47`.
Usage courant Couleurs (par exemple `#FF8800`), adresses mémoire, codes ASCII.

Codage des entiers

Entiers positifs Sur n bits, on représente les entiers de 0 à $2^n - 1$. Par exemple, sur 8 bits : de 0 à 255.
Entiers relatifs (complément à deux) Sur n bits, on représente les entiers de -2^{n-1} à $2^{n-1} - 1$. Sur 8 bits : de -128 à 127.
Principe du complément à deux Pour coder un nombre négatif, on inverse tous les bits de sa valeur absolue, puis on ajoute 1.

Codage des caractères

ASCII Table de 128 caractères codés sur 7 bits. Les lettres majuscules vont de 65 (`A`) à 90 (`Z`), les minuscules de 97 (`a`) à 122 (`z`), les chiffres de 48 (`0`) à 57 (`9`).
Unicode / UTF-8 Extension qui couvre tous les alphabets et symboles (plus de 150 000 caractères). UTF-8 est un encodage à taille variable : un caractère ASCII tient sur un octet, les autres sur deux à quatre octets.
En Python `ord('A')` renvoie `65` ; `chr(65)` renvoie `'A'`.

Codage des flottants (aperçu)

Les nombres à virgule sont codés selon la norme **IEEE 754**. Un flottant sur 64 bits (type `float` en Python) se décompose en :

- un bit de signe ;
- 11 bits d'exposant ;
- 52 bits de mantisse.

La conséquence principale est que la **précision est finie** : certains nombres décimaux simples comme $0{,}1$ ne sont pas représentables exactement.

Pièges fréquents

- Oublier que le bit de poids fort a la plus grande valeur (à gauche, pas à droite).
- Confondre le nombre de valeurs (2^n) avec la valeur maximale ($2^n - 1$).
- Penser qu'un entier négatif en complément à deux commence par 0 (c'est l'inverse : il commence par 1).
- Confondre `ord()` et `chr()` (l'une donne le code, l'autre le caractère).

Erreurs classiques

Code erroné	Code correct	Explication
`bin(13)` → `1101`	`bin(13)` → `'0b1101'`	`bin()` renvoie une chaîne préfixée par `0b`, pas un entier.
`int('0b1101')` → erreur	`int('0b1101', 2)` ou `int('1101', 2)`	Il faut préciser la base 2 en second paramètre effectif.
`chr('65')` → `TypeError`	`chr(65)` → `'A'`	`chr()` attend un entier, pas une chaîne.
Sur 8 bits, $200 + 100 = 300$	Dépassement (overflow) : $300 > 255$	Sur n bits non signés, la valeur maximale est $2^n - 1$.

② EXEMPLES

Programme 1 · Conversions en Python

```python
# Décimal -> Binaire -> Hexadécimal
n = 42
print(bin(n))      # '0b101010'
print(hex(n))      # '0x2a'

# Binaire -> Décimal
print(int('101010', 2))  # 42

# Hexadécimal -> Décimal
print(int('2a', 16))     # 42
```

Programme 2 · Conversion manuelle décimal vers binaire

```python
def decimale_vers_binaire(n):
    """Renvoie l'écriture binaire de n (entier positif) sous forme de chaîne."""
```

```
    if n == 0:
        return '0'
    bits = ''
    while n > 0:
        bits = str(n % 2) + bits   # le reste donne le bit
        n = n // 2
    return bits

print(decimale_vers_binaire(13))   # '1101'
print(decimale_vers_binaire(255))  # '11111111'
```

Programme 3 · Conversion manuelle binaire vers décimal

```
def binaire_vers_decimale(b):
    """Renvoie la valeur décimale de la chaîne binaire b."""
    resultat = 0
    for bit in b:
        resultat = resultat * 2 + int(bit)
    return resultat

print(binaire_vers_decimale('1101'))  # 13
```

Programme 4 · Table ASCII partielle

```
for code in range(65, 91):
    print(f"{chr(code)} -> {code}", end="   ")
# A -> 65   B -> 66   C -> 67 ...
```

Programme 5 · Complément à deux sur 8 bits

```
def complement_a_deux(n, bits=8):
    """Renvoie la représentation binaire en complément à deux sur 'bits' bits."""
    if n >= 0:
        return bin(n)[2:].zfill(bits)
    else:
        # Inverse les bits de |n| et ajoute 1
        positif = bin(abs(n))[2:].zfill(bits)
        inverse = ''.join('1' if b == '0' else '0' for b in positif)
        return bin(int(inverse, 2) + 1)[2:].zfill(bits)

print(complement_a_deux(5))    # '00000101'
print(complement_a_deux(-5))   # '11111011'
```

③ EXERCICES

Exercice 1 *Conversions de base*

1. Convertir à la main en binaire : $(45)_{10}$, $(100)_{10}$, $(255)_{10}$.
2. Convertir à la main en décimal : $(110110)_2$, $(10000001)_2$.
3. Convertir en hexadécimal : $(11010110)_2$, $(10111110)_2$.
4. Convertir en binaire : $(\texttt{3C})_{16}$, $(\texttt{AB})_{16}$.

Exercice 2 *Capacité et dépassement*

1. Combien de valeurs différentes peut-on coder sur 8 bits ? Sur 16 bits ?
2. Quel est le plus grand entier positif représentable sur 8 bits ? Sur 16 bits ?
3. En complément à deux sur 8 bits, quelles sont les bornes (valeur minimale et maximale) ?
4. Expliquer pourquoi, sur 8 bits non signés, $200 + 100$ ne donne pas 300.

Exercice 3 *Fonctions de conversion en Python*

1. Écrire une fonction `decimale_vers_binaire(n)` qui renvoie la représentation binaire d'un entier positif sous forme de chaîne, **sans utiliser** `bin()`.
2. Écrire une fonction `binaire_vers_decimale(b)` qui prend une chaîne binaire et renvoie l'entier correspondant, **sans utiliser** `int(b, 2)`.
3. Écrire une fonction `decimale_vers_hexa(n)` qui renvoie la représentation hexadécimale d'un entier positif sous forme de chaîne, **sans utiliser** `hex()`.

Exercice 4 *Caractères et ASCII*

1. Écrire une fonction `cesar(texte, decalage)` qui applique le chiffrement de César à une chaîne de lettres majuscules. Chaque lettre est décalée de `decalage` positions dans l'alphabet (avec retour cyclique).
2. Tester avec `cesar("BONJOUR", 3)` qui doit renvoyer `"ERQMRXU"`.
3. Écrire la fonction de déchiffrement.

SOLUTIONS DES EXERCICES

Corrigé de l'exercice 1.

1. Divisions successives par 2 :
   - $(45)_{10} = 32 + 8 + 4 + 1 = 2^5 + 2^3 + 2^2 + 2^0 = (101101)_2$;
   - $(100)_{10} = 64 + 32 + 4 = 2^6 + 2^5 + 2^2 = (1100100)_2$;
   - $(255)_{10} = 2^8 - 1 = (11111111)_2$.
2. Somme des puissances de 2 :
   - $(110110)_2 = 2^5 + 2^4 + 2^2 + 2^1 = 32 + 16 + 4 + 2 = (54)_{10}$;
   - $(10000001)_2 = 2^7 + 2^0 = 128 + 1 = (129)_{10}$.
3. On regroupe les bits par paquets de quatre (de droite à gauche) :
   - $(1101\ 0110)_2 = (\texttt{D6})_{16}$ car $1101 = 13 = \texttt{D}$ et $0110 = 6$;
   - $(1011\ 1110)_2 = (\texttt{BE})_{16}$ car $1011 = 11 = \texttt{B}$ et $1110 = 14 = \texttt{E}$.
4. On développe chaque chiffre hexadécimal en quatre bits :
   - $(\texttt{3C})_{16} = (0011\ 1100)_2$ car $\texttt{3} = 0011$ et $\texttt{C} = 1100$;
   - $(\texttt{AB})_{16} = (1010\ 1011)_2$ car $\texttt{A} = 1010$ et $\texttt{B} = 1011$.

Corrigé de l'exercice 2.

1. Sur n bits, on code 2^n valeurs. Sur 8 bits : $2^8 = 256$ valeurs. Sur 16 bits : $2^{16} = 65\,536$ valeurs.
2. Le plus grand entier positif sur n bits est $2^n - 1$. Sur 8 bits : 255. Sur 16 bits : $65\,535$.
3. En complément à deux sur 8 bits : de $-2^7 = -128$ à $2^7 - 1 = 127$.
4. $200 + 100 = 300$, or $300 > 255 = 2^8 - 1$. La valeur dépasse la capacité : on obtient $300 - 256 = 44$ (le résultat « revient à zéro » après 255, c'est le phénomène de dépassement).

Corrigé de l'exercice 3.

```python
def decimale_vers_binaire(n):
    if n == 0:
        return '0'
    bits = ''
    while n > 0:
        bits = str(n % 2) + bits
        n = n // 2
    return bits

def binaire_vers_decimale(b):
    resultat = 0
    for bit in b:
        resultat = resultat * 2 + int(bit)
    return resultat

def decimale_vers_hexa(n):
    if n == 0:
        return '0'
    chiffres = '0123456789ABCDEF'
    h = ''
    while n > 0:
        h = chiffres[n % 16] + h
        n = n // 16
    return h
```

Vérification :

- `decimale_vers_binaire(42)` renvoie `'101010'` ;

— `binaire_vers_decimale('101010')` renvoie `42` ;
— `decimale_vers_hexa(255)` renvoie `'FF'`.

Corrigé de l'exercice 4.

```python
def cesar(texte, decalage):
    resultat = ''
    for c in texte:
        if 'A' <= c <= 'Z':
            code = ord(c) - ord('A')          # position 0..25
            nouveau = (code + decalage) % 26  # décalage cyclique
            resultat += chr(nouveau + ord('A'))
        else:
            resultat += c                     # caractère inchangé
    return resultat

# Chiffrement
print(cesar("BONJOUR", 3))   # "ERQMRXU"

# Déchiffrement : on décale dans l'autre sens
def dechiffrer_cesar(texte, decalage):
    return cesar(texte, -decalage)

print(dechiffrer_cesar("ERQMRXU", 3))  # "BONJOUR"
```

Principe : `ord('B') - ord('A') = 1`. En ajoutant le décalage 3, on obtient 4, soit `chr(4 + 65) = 'E'`. Le modulo 26 assure le retour cyclique (`Z` + 1 → `A`).

Vérification : B→E, O→R, N→Q, J→M, O→R, U→X, R→U. ✓