Le problème de la représentation des données

- La mémoire d'un ordinateur est composé de $bits$ pouvant prendre uniquement les valeurs 0 et de 1

Le problème de la représentation des données

- La mémoire d'un ordinateur est composé de $bits$ pouvant prendre uniquement les valeurs 0 et de 1
- Le regroupement de 8 bits s'appelle un octet (*byte* en anglais) c'est l'unité minimal de mémoire :

$$1 \text{ octet } = \underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}}_{8 \text{ bits}}$$

Le problème de la représentation des données

- La mémoire d'un ordinateur est composé de $bits$ pouvant prendre uniquement les valeurs 0 et de 1
- Le regroupement de 8 bits s'appelle un octet (*byte* en anglais) c'est l'unité minimal de mémoire :

$$1 \text{ octet } = \underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}}_{8 \text{ bits}}$$

- Toutes les données doivent donc être représenté en utilisant des octets.

Le problème de la représentation des données

- La mémoire d'un ordinateur est composé de $bits$ pouvant prendre uniquement les valeurs 0 et de 1
- Le regroupement de 8 bits s'appelle un octet (*byte* en anglais) c'est l'unité minimal de mémoire :

$$1 \text{ octet } = \underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}}_{8 \text{ bits}}$$

- Toutes les données doivent donc être représenté en utilisant des octets.
- On s'intéresse ici à la représentation des entiers positifs et négatifs, des caractères et des flottants.

De la base 10 à la base 2

- Nous sommes habitués à écrire en utilisant 10 chiffres (0,1,2,3,4,5,6,7,8 et 9), chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.

De la base 10 à la base 2

- Nous sommes habitués à écrire en utilisant 10 chiffres (0,1,2,3,4,5,6,7,8 et 9), chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{1815}^{10}$:

De la base 10 à la base 2

- Nous sommes habitués à écrire en utilisant 10 chiffres (0,1,2,3,4,5,6,7,8 et 9), chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{1815}^{10}$:

  $1 \quad 8 \quad 1 \quad 5$

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.

  Par exemple, pour $\overline{1815}^{10}$:

10^3	10^2	10^1	10^0
1	8	1	5

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{1815}^{10}$:

10^3	10^2	10^1	10^0
1	8	1	5

$= 1 \times 1000 + 8 \times 100 + 1 \times 10 + 5 \times 1 = 1815$

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{1815}^{10}$:

10^3	10^2	10^1	10^0
1	8	1	5

$$= 1 \times 1000 + 8 \times 100 + 1 \times 10 + 5 \times 1 = 1815$$

- De la même façon, on pourrait utiliser simplement 2 chiffres et multiplier chaque chiffre par une puissance de 2 suivant son emplacement dans le nombre.

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{1815}^{10}$:

10^3	10^2	10^1	10^0
1	8	1	5

$$= 1 \times 1000 + 8 \times 100 + 1 \times 10 + 5 \times 1 = 1815$$

- De la même façon, on pourrait utiliser simplement 2 chiffres et multiplier chaque chiffre par une puissance de 2 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{11100010111}^{2}$:

1	1	1	0	0	0	1	0	1	1	1

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour 1815 :

10^3	10^2	10^1	10^0
1	8	1	5

$$= 1 \times 1000 + 8 \times 100 + 1 \times 10 + 5 \times 1 = 1815$$

- De la même façon, on pourrait utiliser simplement 2 chiffres et multiplier chaque chiffre par une puissance de 2 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{11100010111}^2$:

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	0	0	1	0	1	1	1

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour 1815 :

10^3	10^2	10^1	10^0
1	8	1	5

$$= 1 \times 1000 + 8 \times 100 + 1 \times 10 + 5 \times 1 = 1815$$

- De la même façon, on pourrait utiliser simplement 2 chiffres et multiplier chaque chiffre par une puissance de 2 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{11100010111}^2$:

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	0	0	1	0	1	1	1

$=2^{10}+2^9+2^8+2^4+2^2+2^1+2^0$

De la base 10 à la base 2

- Nous sommes habitués à écrire les entiers postifs en utilisant 10 chiffres, chaque chiffre étant multiplié par une puissance de 10 suivant son emplacement dans le nombre.
  Par exemple, pour 1815 :

10^3	10^2	10^1	10^0
1	8	1	5

$= 1 \times 1000 + 8 \times 100 + 1 \times 10 + 5 \times 1 = 1815$

- De la même façon, on pourrait utiliser simplement 2 chiffres et multiplier chaque chiffre par une puissance de 2 suivant son emplacement dans le nombre.
  Par exemple, pour $\overline{11100010111}^2$:

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	0	0	1	0	1	1	1

$=2^{10}+2^9+2^8+2^4+2^2+2^1+2^0$

$=1815$

Ce sont des cas particuliers (avec $b = 10$ et $b = 2$), du théorème suivant :

Décomposition en base b

Tout entier $n \in \mathbb{N}$ peut s'écrire sous la forme :

$$n = \sum_{k=0}^{p} a_k b^k$$

avec $p \geq 0$ et $a_k \in [\![0; b-1]\!]$. De plus, cette écriture est unique si $a_p \neq 0$ et s'appelle *décomposition en base b de n* et on la note $n = \overline{a_p \dots a_1 a_0}^b$

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^2$

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^2$
- $\overline{1101001011}^2$

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^2$
- $\overline{1101001011}^2$
- $\overline{421}^5$

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^2$
- $\overline{1101001011}^2$
- $\overline{421}^5$
- $\overline{3EA}^{16}$

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^{2}$
- $\overline{1101001011}^{2}$
- $\overline{421}^{5}$
- $\overline{3EA}^{16}$
  On travaille ici en base 16, donc avec 16 chiffres, les lettres majuscules de A à F représentent les "chiffres" 10 à 15.

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^{2} = \overline{139}^{10}$
- $\overline{1101001011}^{2}$
- $\overline{421}^{5}$
- $\overline{3EA}^{16}$

  On travaille ici en base 16, donc avec 16 chiffres, les lettres majuscules de A à F représentent les "chiffres" 10 à 15.

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^{2} = \overline{139}^{10}$
- $\overline{1101001011}^{2} = \overline{843}^{10}$
- $\overline{421}^{5}$
- $\overline{3EA}^{16}$

  On travaille ici en base 16, donc avec 16 chiffres, les lettres majuscules de A à F représentent les "chiffres" 10 à 15.

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^{2} = \overline{139}^{10}$
- $\overline{1101001011}^{2} = \overline{843}^{10}$
- $\overline{421}^{5} = \overline{111}^{10}$
- $\overline{3EA}^{16}$

  On travaille ici en base 16, donc avec 16 chiffres, les lettres majuscules de A à F représentent les "chiffres" 10 à 15.

Exemples

Ecrire en base 10 les nombres ci-dessous

- $\overline{10001011}^2 = \overline{139}^{10}$
- $\overline{1101001011}^2 = \overline{843}^{10}$
- $\overline{421}^5 = \overline{111}^{10}$
- $\overline{3EA}^{16} = \overline{1002}^{10}$

  On travaille ici en base 16, donc avec 16 chiffres, les lettres majuscules de A à F représentent les "chiffres" 10 à 15.

Limitations mémoire et dépassement de capacité

- Le nombre de bits représentant un entier est limité, le plus grand nombre représentable sur n bits est :
  $\overline{1\dots1}^2 = 2^{n-1} + \dots + 1 = 2^n - 1$

Limitations mémoire et dépassement de capacité

- Le nombre de bits représentant un entier est limité, le plus grand nombre représentable sur n bits est :
  $\overline{1\dots1}^2 = 2^{n-1} + \dots + 1 = 2^n - 1$
- En C, les valeurs maximales représentables suivant le type d'entier positif utilisé sont donc :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*

Limitations mémoire et dépassement de capacité

- Le nombre de bits représentant un entier est limité, le plus grand nombre représentable sur n bits est :
  $\overline{1\ldots1}^2 = 2^{n-1} + \cdots + 1 = 2^n - 1$
- En C, les valeurs maximales représentables suivant le type d'entier positif utilisé sont donc :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $2^8 - 1 = 255$

En cas de dépassement de capacité (*overflow* ou *underflow*), le résultat obtenu est calculé modulo la plus grande valeur maximale plus 1.

Limitations mémoire et dépassement de capacité

- Le nombre de bits représentant un entier est limité, le plus grand nombre représentable sur n bits est :
  $\overline{1\dots1}^2 = 2^{n-1} + \cdots + 1 = 2^n - 1$
- En C, les valeurs maximales représentables suivant le type d'entier positif utilisé sont donc :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $2^8 - 1 = 255$
  - `uint32_t` : $2^{32} - 1 = 4\,294\,967\,295$ ($\geq$ 4 milliards)

En cas de dépassement de capacité (*overflow* ou *underflow*), le résultat obtenu est calculé modulo la plus grande valeur maximale plus 1.
Par exemple, Les dépassement de capacité sur un `uint8_t` sont calculés modulo 256.

Limitations mémoire et dépassement de capacité

- Le nombre de bits représentant un entier est limité, le plus grand nombre représentable sur n bits est :
  $\overline{1\ldots1}^2 = 2^{n-1} + \cdots + 1 = 2^n - 1$
- En C, les valeurs maximales représentables suivant le type d'entier positif utilisé sont donc :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $2^8 - 1 = 255$
  - `uint32_t` : $2^{32} - 1 = 4\,294\,967\,295$ ($\geq$ 4 milliards)
  - `uint64_t` : $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ ($\geq$ 18 milliards de milliards)

  En cas de dépassement de capacité (*overflow* ou *underflow*), le résultat obtenu est calculé modulo la plus grande valeur maximale plus 1.
  Par exemple, Les dépassement de capacité sur un `uint8_t` sont calculés modulo 256.
- En OCaml, il n'y a pas nativement de type entier non signé.

Exemple

```c
#include <stdio.h>
#include <stdint.h>

int main()
{
    uint8_t n1 = 240;
    uint32_t n2 = 0;
    n1 = n1 + 20;
    n2 = n2 - 1;
    printf("valeur de n1 = %u\n",n1);
    printf("valeur de n2 = %u\n",n2);
}
```

Quel est l'affichage produit par le programme ci-dessus ? Expliquer.

Correction

```c
#include <stdio.h>
#include <stdint.h>

int main()
{
    uint8_t n1 = 240; // 8 bits donc valeur maximale 255
    uint32_t n2 = 0; // valeur minimale 0 (non signé)
    n1 = n1 + 20; // overflow : 260
    n2 = n2 - 1; // underflow : -1
    printf("valeur de n1 = %u\n",n1); // 4 (car 260 = 4 modulo 256)
    printf("valeur de n2 = %u\n",n2); // 4294967295 (car -1 =
↪   4294967295 modulo 4294967296)
}
```

Complément à deux

- La stratégie qui consiste à prendre *un bit de signe* et à représenter la valeur absolue de l'entier sur les autres présente deux difficultés : 0 est représenté deux fois et surtout l'addition binaire bit à bit ne fonctionne pas.

Complément à deux

- La stratégie qui consiste à prendre *un bit de signe* et à représenter la valeur absolue de l'entier sur les autres présente deux difficultés : 0 est représenté deux fois et surtout l'addition binaire bit à bit ne fonctionne pas.
- La méthode utilisée est celle du complément à 2, sur n bits, on compte négativement le bit de poids 2^{n-1} et positivement les autres.

Complément à deux

- La stratégie qui consiste à prendre *un bit de signe* et à représenter la valeur absolue de l'entier sur les autres présente deux difficultés : 0 est représenté deux fois et surtout l'addition binaire bit à bit ne fonctionne pas.
- La méthode utilisée est celle du complément à 2, sur n bits, on compte négativement le bit de poids 2^{n-1} et positivement les autres.
  Par exemple, sur 8 bits :
  $\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0} = -2^7 + 2^4 + 2^3 + 2^1 = -101$
- De façon générale, sur n bits, la valeur en complément à deux de la suite bits $(b_{p-1} \dots b_0)$ est :

$$-b^{p-1} + \sum_{k=0}^{n-2} b_k \, 2^k$$

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1}-1$

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1}-1$
- En C, les valeurs extrêmes représentables sont :

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1} - 1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1}-1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $[\![-2^7; 2^7-1]\!] = [\![-128; 127]\!]$

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1} - 1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $[\![-2^7; 2^7 - 1]\!] = [\![-128; 127]\!]$
  - `uint32_t` : $[\![-2^{31}; 2^{31} - 1]\!]$

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1} - 1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $[\![-2^7; 2^7 - 1]\!] = [\![-128; 127]\!]$
  - `uint32_t` : $[\![-2^{31}; 2^{31} - 1]\!]$
  - `uint64_t` : $[\![-2^{63}; 2^{63} - 1]\!]$

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1} - 1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $[\![-2^7; 2^7 - 1]\!] = [\![-128; 127]\!]$
  - `uint32_t` : $[\![-2^{31}; 2^{31} - 1]\!]$
  - `uint64_t` : $[\![-2^{63}; 2^{63} - 1]\!]$

⚠ Un dépassement de capacité est un comportement indéfini.

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1} - 1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $[\![-2^7; 2^7 - 1]\!] = [\![-128; 127]\!]$
  - `uint32_t` : $[\![-2^{31}; 2^{31} - 1]\!]$
  - `uint64_t` : $[\![-2^{63}; 2^{63} - 1]\!]$

  ⚠ Un dépassement de capacité est un comportement indéfini.
- En OCaml, les entiers sont codés sur 64 bits mais un bit est réservé par le langage, l'intervalle représentable est donc $[\![-2^{62}; 2^{62} - 1]\!]$.

Conséquences de la représentation en complément à 2

- Les difficultés de la stratégie du *un bit de signe* sont levées.
- Le plus petit petit représentable sur n bits est alors -2^{n-1} et le plus grand $2^{n-1} - 1$
- En C, les valeurs extrêmes représentables sont :
  - `uint` : *au min 16 bits, usuellement 32 bits, dépendant du compilateur*
  - `uint8_t` : $[\![-2^7; 2^7 - 1]\!] = [\![-128; 127]\!]$
  - `uint32_t` : $[\![-2^{31}; 2^{31} - 1]\!]$
  - `uint64_t` : $[\![-2^{63}; 2^{63} - 1]\!]$

  ⚠ Un dépassement de capacité est un comportement indéfini.
- En OCaml, les entiers sont codés sur 64 bits mais un bit est réservé par le langage, l'intervalle représentable est donc $[\![-2^{62}; 2^{62} - 1]\!]$.
  Les dépassements de capacité sont calculés modulo 2^{63} puis ramené dans l'intervalle précédent.