TP/Cours
Représentation des nombres

Objectifs :

- ⇨ Connaître la façon de représenter les nombres entiers dans les ordinateurs et les contraintes associées
- ⇨ Connaître le complément à 2 et savoir coder un nombre entier négatif en complément à 2
- ⇨ Découvrir les nombres à virgule flottante permettant de représenter les nombres réels en informatique
- ⇨ Savoir convertir un nombre réel en flottant et inversement

I - Représentation des entiers naturels et des entiers relatifs

Pour représenter des nombres entiers positifs, l'ordinateur utilise logiquement le binaire. Il existe donc un nombre fini de possibilités et l'entier le plus grand que la machine puisse représenter dépend du nombre de bit utilisés pour le stocker.

Q1 :
Quel est le plus grand entier que l'on peut écrire avec 8 bits (taille des mots des premiers ordinateurs personnels) ? 16 bits ? 32 bits (nombre de bit d'une adresse IP) ? 64 bits (tailles des mots des PCs actuels) ?
Quel est le plus grand entier que l'on peut représenter avec Python (faire quelques essais dans la console python) ?

1) Calculs

Les règles de calcul sont alors les mêmes qu'en base 10 (sauf que pour les additions en base 2, il y a plus souvent de retenue !).

Exemple :

Addition

```
Retenues      1 1   1 1
              0011  0010     50
        +     1001  0111   +151
              1100  1001    201
```

multiplication

```
              1010     10
        x      101    X 5
              1010     50
             0000.
            1010..
            110010
```

Q2 :
Poser l'addition binaire $(10110101)_2 + (01110100)_2$ et donner son résultat. De même pour la multiplication $(1001)_2 x (110)_2$

Q3 :
Comment interpréter le calcul de l'image précédente (avec les nombres binaires en vert sur l'écran d'ordinateur) ?

2) Représentation des nombres négatifs

a. Problématique

Se pose à présent la question de la représentation des nombres entiers négatifs.
Une solution serait que le premier bit soit considéré comme un bit de signe et le reste des bits comme la valeur absolue exprimée comme dans le paragraphe précédent.
Cette solution a au moins deux inconvénients. Le premier est l'existence de deux zéros, l'un positif et l'autre négatif. Le second inconvénient est que cette écriture ne prolonge pas les règles d'addition.

Exemple :

	Binaire	Base 10	
	0011 1010	58	NON UTILISE CAR ON OBTIENT DES RESULTATS FAUX
+	1001 0101	$-(001\ 0101)_2 = -21$	
	1100 1111	37	
	$-(100\ 111)_2 = -79$		

b. Complément à deux

La solution choisie est tout de même assez proche et s'appelle complément à deux (voir le lien).

Pour déterminer le complément à 2 d'un nombre binaire, il faut procéder en deux étapes :

- Inverser les bits du nombre (c'est le *complément à 1* ou la *négation*)
- Ajouter 1 au résultat sans tenir compte d'un éventuel dépassement.

Exemple : Le complément à 2 de $(0110\ 0010)_2$ se calcule comme suit :

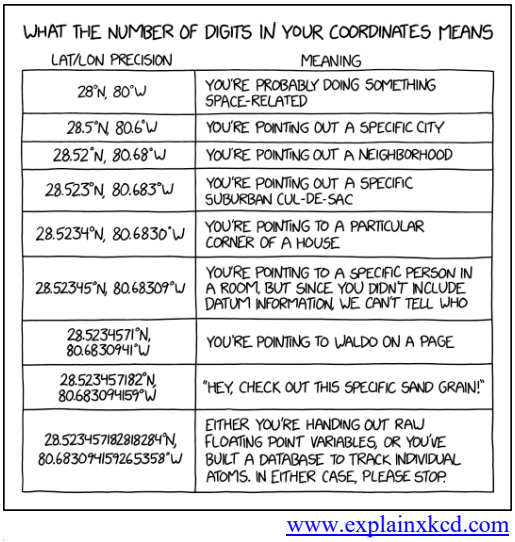
1) Inversion du nombre : $\overline{(0110\ 0010)}_2 = (1001\ 1101)_2$
2) On ajoute 1 : $(1001\ 1101)_2 + 1 = \mathbf{(1001\ 1110)_2}$

On peut ainsi continuer à faire les opérations comme précédemment, mais le nombre le plus grand que l'on peut représenter est maintenant deux fois plus petit :

Nombre de bits	Nombre le plus petit	Nombre le plus grand
8	- 128	+ 127
16	- 32768	+ 32 767
32	- 2 147 483 648	+ 2 147 483 647
64	- 9 223 374 036 584 775 808	+ 9 223 374 036 584 775 807

Q4 :
1) Déterminer, en binaire, le complément à 2 du nombre $(21)_{10}$.
2) Poser, **en binaire**, la soustraction $(58)_{10} - (21)_{10}$ (en fait l'addition 58 + (-21)). Convertir le résultat en base 10 pour vérifier qu'on obtient bien le résultat attendu.

II – La représentation des nombres réels



www.explainxkcd.com

Après les nombres entiers viennent naturellement les nombres réels ou du moins leur approximation dans les machines. L'ordinateur n'ayant qu'une mémoire finie, il ne peut stocker tous les chiffres d'un nombre comme $\sqrt{2}$ ou un 1/3. Il va donc falloir faire une approximation.

1) Nombres binaires à virgule

Jusqu'ici on a vu uniquement les nombres binaires entiers. Généralisons maintenant aux nombres à virgules en étudiant un exemple.

Quelle est la valeur décimale du nombre binaire $(1011,011)_2$?

$$\begin{aligned}(1011,011)_2 &= 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2} + 1\times 2^{-3}\\ &= 1\times 8 + 0\times 4 + 1\times 2 + 1\times 1 + 0\times 0{,}5 + 1\times 0{,}25 + 1\times 0{,}125\\ &= 8 + 2 + 1 + 0{,}25 + 0{,}125\\ &= (11{,}375)_{10}\end{aligned}$$

On remarque que les bits situés à droite de la virgule correspondent tout simplement à des puissances de 2 négatives.

Application n°1 :
Ecrire le nombre (110,1101)2 en base 10.

2) La représentation normalisée

De la même manière qu'on représente un résultat numérique en notation scientifique en physique, on peut également représenter un nombre binaire sous la forme d'un nombre multiplié par une puissance de 2.

Avec les nombres binaires, on peut utiliser le même principe :

N = 101101,0111001 (soit 45,4453125 en décimal)
= $1{,}011010111001 \times 2^5$ en décalant la virgule de 5 crans vers la gauche

Rappel : La notation scientifique

$N = a \times 10^{exp}$ avec $1 \leq a < 10$

a s'écrit donc sous la forme x,yztw……
où x est un chiffre entre 1 et 9

Exemples : $5127 = 5{,}127 \times 10^3$
$0{,}000963 = 9{,}63 \times 10^{-4}$

On peut remarquer qu'en binaire le chiffre juste avant la virgule sera toujours 1 (ça ne peut pas être 0 sinon on pourrait encore décaler la virgule et il n'y a pas d'autres chiffres que 0 en dehors de 1). Pour économiser un bit, on choisit donc **de ne pas stocker en mémoire le 1 qui se trouve avant la virgule**. On obtient alors la **représentation normalisée** :

$N = 1,\underbrace{011010111001}_{\text{mantisse}} \times 2^{\overbrace{5}^{\text{exposant}}}$ en décalant la virgule de 5 crans vers la gauche

Ici la représentation normalisée de N est donc mantisse = 011010111001 et exposant = 101 (5 en base 10)

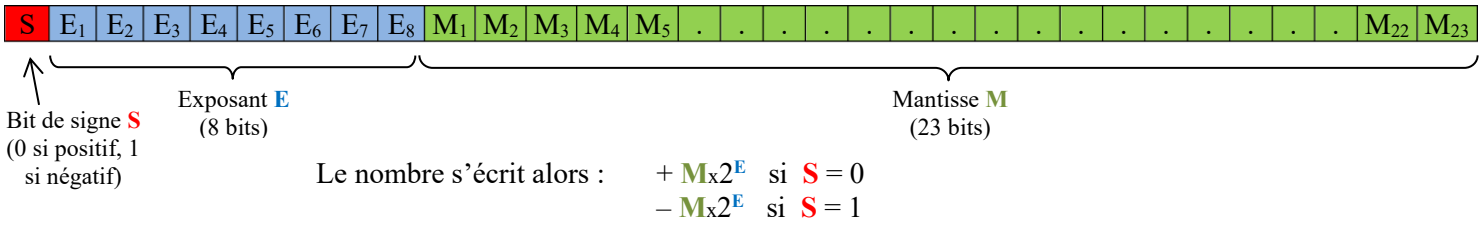
Application n°2 :
Donner la représentation normalisée du nombre $(110,1101)_2$.

3) Virgule flottante

La représentation des nombres en **virgule flottante** permet de définir une implémentation pratique pour la représentation normalisée en prenant en compte également les nombres négatifs.
Ces nombres sont régis par un standard, l'IEEE-754 , qui définit la répartition et l'usage des bits d'un nombre.

Les formats usuels sont la *simple précision* sur 32 bits et la *double précision* sur 64 bits[1].

Voyons comment se compose un nombre en virgule flottante en simple précision[2] :



Le nombre s'écrit alors : $+ M \times 2^E$ si $S = 0$
$- M \times 2^E$ si $S = 1$

Le codage de l'exposant est particulier. En effet, celui-ci peut être positif (nombres supérieurs à 1) ou négatif (nombres inférieurs à 1). On aurait pu penser que le plus simple était alors de coder l'exposant en complément à deux comme on l'a vu précédemment, mais la comparaison des nombres flottant aurait alors été plus complexe.
La solution retenue est de coder l'exposant comme un entier positif en utilisant un décalage d'exposant.
La valeur du décalage dépend du nombre n de bits utilisés pour stocker l'exposant et vaut $2^{n-1} - 1$.
Pour la simple précision, avec 8 bits d'exposant, le décalage est donc de $2^{(8-1)}-1 = 2^7 - 1 = 128 - 1 = 127$

Ainsi le nombre binaire stocké dans E_1 à E_8 est la valeur de l'exposant + 127

Pour la mantisse, c'est plus simple car elle contient simplement les bits de la mantisse de la représentation normalisée.

Reprenons notre exemple précédent : N = 101101,0111001
On a vu qu'en représentation normalisée, on avait mantisse = 011010111001 et exposant = 101 (5 en décimal)

- ⇨ Le nombre est positif, donc S = 0
- ⇨ L'exposant est 5, on stocke donc la valeur 5 + 127 = 132, soit 1000 0100 en binaire
- ⇨ La mantisse vaut 011010111001

Le nombre en virgule flottante est donc : 0100 0010 0011 0101 1100 1000 0000 0000

Autre exemple un peu plus difficile :
Prenons $N = (-7,3)_{10}$

1. Le nombre est négatif, donc le bit de signe vaut 1 et on travaille maintenant sur la valeur absolue de N : +7,3
2. Il nous faut ensuite le convertir en binaire :
   - ⇨ Partie entière : $(7)_{10} = (111)_2$
   - ⇨ Partie fractionnaire : 0,3

0,3 x 2 = 0,6 → **0**
0,6 x 2 = 1,2 → **1**
0,2 x 2 = 0,4 → **0**
0,4 x 2 = 0,8 → **0**
0,8 x 2 = 1,6 → **1**
(se répète à l'infini)
0,6 x 2 = 1,2 → **1**
0,2 x 2 = 0,4 → **0**
0,4 x 2 = 0,8 → **0**
0,8 x 2 = 1,6 → **1**
0,6 x 2 = 1,2 → **1**

> Méthode de conversion décimal -> binaire pour nombres à virgule
>
> Il faut décomposer le nombre en sa partie avant la virgule que l'on traitera comme vu dans le chapitre précédent (divisions successives par 2) et la partie après la virgule (partie fractionnaire) où on doit utiliser une autre technique.
>
> Pour la partie après la virgule 0,… on multiplie ce nombre par 2. La partie entière du résultat nous donne le premier bit de la conversion. Ensuite on reprend la partie fractionnaire que l'on multiplie par deux, et ainsi de suite jusqu'à ce que le résultat de la multiplication donne 1,0 qui est le critère d'arrêt.
>
> Plus d'info et des exercices d'entrainement sur :
> https://www.youtube.com/watch?v=Tmg8BgiGPes

Le nombre est donc 0,**0100110011001**…$[1001]_\infty$

[1] Le type `float` de python 3.x repose sur une représentation des nombres en virgule flottante avec **double précision**.
[2] En double précision, il y a toujours 1 bit de signe, mais un exposant sur 11 bits et une mantisse sur 52 bits.

On voit que ce nombre possède une infinité de décimales (un peu comme 1/3 en base 10) et peut s'écrire :

$N = 111,0100110011001100110011001\ldots[1001]_{\infty}$

3. Ecrivons-le en notation normalisée : $N = 1,1101001100110011001100110011001\ldots[1001]_{\infty} \times 2^{2}$
4. Extrayons la mantisse et l'exposant : Mantisse = $1101001100110011001100110011001\ldots[1001]_{\infty}$
   Exposant : $(2)_{10}$
5. Décalons l'exposant : $2 + 127 = (129)_{10} = (1000\ 0001)_{2}$
6. Assemblons le signe, l'exposant décalé et la mantisse (tronquée à 23 bits) :

$N = [1100\ 0000\ 1110\ 1001\ 1001\ 1001\ 1001\ 1001]_{\text{virgule flottante simple précision}}$

Application n°3 :
Donner la représentation en virgule flottante simple précision du nombre $N = (165,8125)_{10}$.

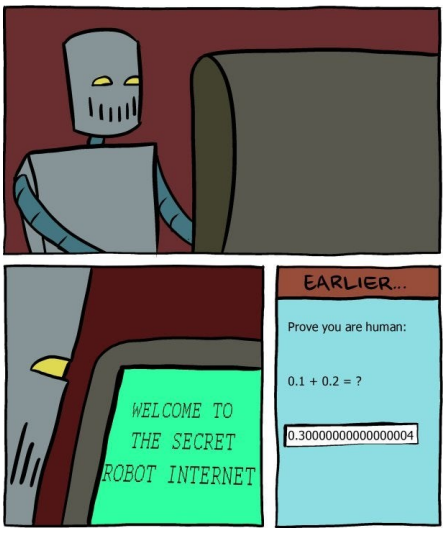
4) Valeurs particulières

La norme IEEE 754 réserve les exposants 000...000 (uniquement des 0) et 111...111 (uniquement des 1) pour coder des valeurs particulières :

Exposant	Mantisse	Valeur représentée
000...000	000...000	**0** (zéro[3])
000...000	000...001 à 111...111	nombre **dénormalisé** valeur = ± **0**,mantisse * $2^{-126 \text{ ou } -1022}$
111...111	000...000	± infini
111...111	000...001 à 111...111	**NaN** (Not a Number - *pas un nombre*) exemple : 0 / 0

Si vous n'êtes pas très au point sur les nombres flottants et que vous souhaitez une autre explication et des exercices d'application, vous pouvez suivre le lien suivant :
https://iut-info.univ-reims.fr/users/nourrit/codages/page12.html

5) Précision des calculs en virgule flottante



Application n°4 :
1) Utiliser la console python pour déterminer le résultat des expressions suivantes :

```
1.0 + 1.0 == 2.0    =>
4.5 + 5.5 == 10     =>
0.4 + 0.4 == 0.8    =>
0.1 + 0.2 == 0.3    =>
200 + 1e-14[4] -200 == 1e-14  =>
```

2) En décomposant les calculs, essayez d'expliquer les deux derniers résultats.

[3] Il existe donc deux zéros dans la norme IEEE 754 : +0 et -0 en fonction du bit de signe !
[4] Le « e » en informatique correspond à « $\times 10^{\cdots}$ » ainsi `1e-14` est la notation pour le nombre 1×10^{-14}.

3) Quelle sera la dernière valeur affichée par le programme ci-contre ? Essayez de deviner puis vérifiez en l'exécutant.

```
x = 0.0
while x != 3.5:
    print(x)
    x = x + 0.1
print("Terminé !")
```

CONCLUSION :

Q5

1) Ecrire le nombre 2020 en virgule flottante simple précision.

2) Combien vaut le nombre suivant stocké en virgule flottante simple précision :

`1 0111 1100 1101 0100 0000 0000 0000 000` ?