DS INFO N°1

Exercice 1 (Question de cours - 7 minutes max).

1. Nommer et décrire succinctement les différentes étapes de la compilation d'un code écrit en langage C. Un schéma rapidement fait à la main, clair et annoté avec quelques explications suffit.

2. Quel est le nombre minimal de bits nécessaires pour l'encodage d'un nombre entier naturel q donné ? Démontrez rigoureusement ce résultat.

Corrigé de l'exercice 1.

[Retour à l'énoncé]

1. Voir cours Séquence 2

2. Le nombre minimal N de bits nécessaire pour encoder un entier naturel non nul q est $\lfloor \log_2(q) \rfloor + 1$

N étant le nombre de bits minimal pour encoder q, le bit de poids fort est à 1 car sinon, on pourrait l'enlever et l'on aurait besoin que de $N-1$ bits, ce qui contredirait la minimal de N. Donc q est nécessairement tel que :

$$q \geq (100\ldots000)_2 = 2^{N-1}$$

Par ailleurs, q est au maximum égal à $(11\ldots111)_2$ sur N bits car sinon, cela signifierait qu'il lui faudrait plus de N bits pour pouvoir être encodé. Ainsi :

$$q \leq (11\ldots111)_2 = 2^N - 1 < 2^N$$

On en déduit le premier encadrement :

$$2^{N-1} \leq q < 2^N$$

N est alors le nombre de bits minimal (optimal) pour l'écriture binaire de q.

Comme la fonction logarithme népérien est croissante sur $]0;+\infty[$, les inégalités sont préservées par passage au logarithme :

$$\ln\left(2^{N-1}\right) \leq \ln(q) < \ln\left(2^N\right)$$

$$\Leftrightarrow \quad (N-1)\ln(2) \leq \ln(q) < N\ln(2)$$

$$\Leftrightarrow \quad N - 1 \leq \frac{\ln(q)}{\ln(2)} < N$$

$$\Leftrightarrow \quad \underbrace{N - 1 \leq \log_2(q) < N}_{\text{inégalité 1}}$$

Pour la dernière étape, on a divisé partout par $\ln(2)$ qui est strictement positif (car $2 > 1$), donc le sens des inégalités n'est pas modifié.

Nous cherchons à déterminer l'entier N. Nous allons déjà manipuler les inégalités pour encadrer N.

$$\begin{aligned} & \log_2(q) < N \quad \text{inégalité 1} \\ \Leftrightarrow \quad & \log_2(q) - 1 < N - 1 \end{aligned}$$

En combinant avec $N - 1 \leq \log_2(q)$, on obtient l'encadrement :

$$\log_2(q) - 1 < N - 1 \leq \log_2(q)$$

Ainsi,

$$N - 1 = \lfloor \log_2(q) \rfloor$$

Dont on déduit

$$N = \lfloor \log_2(q) \rfloor + 1$$

Exercice 2 (15 minutes max).

On s'intéresse à la représentation des entiers relatifs sur 8 bits en complément à deux.

1. Donner l'intervalle des entiers relatifs représentables avec cette représentation.
2. Donner, pour chacun des entiers relatifs suivants, leur **représentation binaire et l'écriture hexadécimale** de cette représentation : 0, 1, -1, -7, 53 et -27.
3. Quels sont les entiers relatifs représentés par 11111110 et 10000000 ?

Corrigé de l'exercice 2. *[Retour à l'énoncé]*

1. L'intervalle des entiers relatifs représentables en complément à deux sur 8 bits est :

$$[\![-2^{8-1}; 2^{8-1} - 1]\!] = [\![-128; 127]\!]$$

2. Cette question requiert la maîtrise des techniques suivantes :
   - écriture en base deux d'un nombre entier positif classique, par exemple par la méthode de l'escalier
   - technique complément à deux pour le calcul de la représentation binaire des nombre entiers négatifs
   - technique de conversion binaire vers hexadécimal (regroupement des bits par paquets de 4 et calcul du symbole hexadécimal correspondant à la valeur des 4 bits)

   - 0 est représenté par `0000 0000`, qui s'écrit `00` en hexadécimal.
   - 1 est représenté par `0000 0001`, qui s'écrit `01` en hexadécimal.
   - -1 est représenté par `1111 1111`, qui s'écrit `FF` en hexadécimal.
   - -7 est représenté par `1111 1001`, qui s'écrit `F9` en hexadécimal.
   - 53 est représenté par `0011 0101` (appliquer la méthode de l'escalier par exemple). 53 encodé en complément à 2 s'écrit `35` en hexadécimal (marrant, non ?)
   - Pour -27 on applique la méthode du complément à deux. On écrit d'abord 27 en binaire avec par exemple la méthode de l'escalier. On trouve `0001 1011`. On inverse chaque bit et on ajoute 1. On obtient finalement `1110 0101`, ce qui s'écrit `E5` en hexadécimal.

3. L'entier relatif représenté par 11111110 est -2, on le voit immédiatement si on se rappelle que 11111111 représente -1 et qu'on parcourt les nombres négatifs en "reculant" sur la droite, cf le cours. L'entier relatif représenté par 10000000 est le tout dernier entier négatif représentable qui est -128 d'après la question 1. Pareil, cela est évident si l'on a en tête la correspondance entre les entiers relatifs et la manière donc la représentation en complément à 2 les projette sur les entiers naturels.

Exercice 3 (20-25 minutes).

On souhaite encoder des nombres flottants selon la philosophie de la norme IEEE754 avec $E = 7$ bits pour l'exposant **biaisé** et $M = 8$ bits pour la mantisse. On rappelle que les nombres normalisés sont ceux pour lesquels l'exposant biaisé n'est ni minimal ni maximal. On rappelle que le décalage à appliquer pour l'exposant biaisé est $2^{E-1} - 1$.

1. Avec cette convention, quel est le nombre flottant représenté par 1100000000000000 ?

2. Donner l'encodage binaire de $x = 0.103$

3. Quel est le nombre réel associé à l'encodage de x ? Commenter brièvement.

4. Donner l'encodage binaire de $y = -32.5$. Quelle différence avec l'encodage de x ?

5. Combien de nombres flottants normalisés peut-on représenter avec ce système ?

6. Quel est le plus petit nombre flottant normalisé positif représentable avec ce système ?

7. Quel est le plus grand nombre flottant normalisé représentable avec ce système ?

Corrigé de l'exercice 3.

[Retour à l'énoncé]

1. L'encodage doit d'abord être découpé en 3 parties :

$$\underbrace{1}_{\text{1 bit de signe}} \quad \underbrace{1000000}_{\text{7 bits d'exposant biaisé}} \quad \underbrace{00000000}_{\text{8 bits de mantisse}}$$

Signe : le bit de signe est à 1 : c'est donc un nombre négatif

Exposant : l'exposant biaisé a pour valeur $2^6 = 64$. On retire le décalage de $2^{7-1}-1 = 64 - 1 = 63$ pour obtenir le vrai exposant qui est donc 1

Mantisse : la mantisse est à 0 donc il n'y a pas de nombre après la virgule

On en déduit que le nombre flottant encodé est :

$$-1.0 \times 2^1 = -2$$

NB : On peut donc encoder un nombre entier avec un encodage flottant. Cela est utile lorsque l'on est amené à faire des calculs avec des nombres de types différents, par exemple quand on veut diviser un nombre flottant par un entier.
En OCaml, il est rigoureusement interdit de faire des opérations entre nombres de types différents.
En C, le compilateur autorise des opérations de **transtypage** acceptables, c'est-à-dire n'entraînant pas de dépassement de capacité. C'est le cas lorsque l'on veut transformer un entier en un flottant : dans ce cas, l'entier sera ré-encodé sous la forme d'un flottant avant que les deux nombres flottants soient envoyés dans les registres du microprocesseur.

2. **Signe :** le premier bit est à 0 car x est un nombre positif

Exposant : la plus grande puissance de 2 qui « rentre » dans 0.103 est$2^{-4} = \frac{1}{16}$.

Ainsi, récrivons x pour faire apparaître cette puissance :

$$x = 0.103 \times 16 \times 2^{-4} = 1.648 \times 2^{-4}$$

On a donc déjà l'exposant, -4, qu'il faut penser à décaler de 63 : l'exposant à encoder est donc $-4 + 63 = 59$, qui s'encode en binaire comme 0111011.

Mantisse : il nous reste à calculer la mantisse, on n'hésite pas à utiliser la calculatrice :

$$\begin{aligned} 1.648 &= 1 + 0.648 \\ &= 1 + 2^{-1} + 0.148 \\ &= 1 + 2^{-1} + 2^{-3} + 0.023 \\ &= 1 + 2^{-1} + 2^{-3} + 2^{-6} + 7.375 \times 10^{-3} \\ &= 1 + 2^{-1} + 2^{-3} + 2^{-6} + 2^{-8} + 3.46875 \times 10^{-3} \end{aligned}$$

et on s'arrête là car de toute façon, nous n'avons que 8 bits pour stocker la mantisse, donc on ne pourra pas encoder des puissances de 2 plus petites.

$$1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + 1 \times 2^{-6} + 0 \times 2^{-7} + 1 \times 2^{-8}$$

La mantisse est donc encodé par 10100101

En conclusion, l'encodage binaire de $x = 0.103$ avec cette représentation est :

0 0111011 10100101

3. Le nombre réel encodé par

0 0111011 10100101

a la valeur $2^{-4} \times (1 + 2^{-1} + 2^{-3} + 2^{-6} + 2^{-8})$ qui n'est pas égale à 0.103. La différence entre les deux est $2^{-4} \times 3.46875 \times 10^{-3} \approx 2.17 \times 10^{-4}$. Ainsi, l'encodage d'un nombre réel sous la forme d'un flottant conduit souvent à une perte de précision sur ce flottant. Plus la mantisse a un nombre de bits important, plus la précision sera forte. Toutefois, il faut également un nombre suffisant de bits pour encoder l'exposant si l'on souhaite représenter un large éventail de nombres (des très grands et des très petits).

4. On décomposer y comme nous l'avons fait pour x, en cherchant d'abord la plus grande puissance de 2 qui rentre dans y, puis en décomposant la mantisse :

$$y = -32.5 = -2^5 \times 1.015625 = -2^5 \times (1 + 0.015625) = -2^5 \times (1 + 2^{-6})$$

Signe : le premier bit est à 1 car y est un nombre négatif

Exposant : l'exposant réel est 5, il faut le décaler de 63, ce qui donne un exposant biaisé de 68, encodé en binaire comme 1000100.

Mantisse : Elle vaut

$$1 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + 1 \times 2^{-6} + 0 \times 2^{-7} + 0 \times 2^{-8}$$

La mantisse est donc encodé par 00000100

En conclusion, l'encodage binaire de $y = -32.5$ avec cette représentation est :

1 1000100 00000100

La différence avec l'encodage de x tient au fait que cette fois-ci, l'encodage est **exact** : le nombre flottant correspond exactement à la valeur du nombre réel encodé. En fait, y est ce que l'on appelle un nombre **dyadique**, parfaitement décomposable en puissance de 2.

5. Pour calculer le nombre de flottants représentables avec ce système, il faut faire du dénombrement. Chaque bit peut prendre uniquement deux valeurs, donc le nombre de possibilités est a priori :

$$2 \times 2^7 \times 2^8$$

Mais attention, les nombres normalisés sont ceux pour lesquels l'exposant n'est ni maximal (1111111) ni minimal (0000000). Il faut donc retirer ces deux cas. Finalement, le nombre total de nombres **normalisés** représentables est :

$$2 \times (2^7 \underbrace{-2}_{\text{cas particuliers}}) \times 2^8 = 2^{16} - 2^{10} = 2^{10} \times (2^6 - 1) = 31 \times 2^1 1 = 64 \times 2048 = 64512$$

Ce n'est pas beaucoup...

6. Le plus petit nombre normalisé positif représentable avec ce système est celui dont l'exposant biaisé vaut 0000001 et dont la mantisse vaut 00000000. Cela donne donc, en pensant à enlever le biais de l'exposant l'exposant :

$$2^{1-63} \times 1.0 = 2^{-62} \approx 2.17 \times 10^{-19}$$

7. Le plus grand nombre normalisé représentable avec ce système est celui dont l'exposant biaisé vaut 1111110 et dont la mantisse vaut 11111111. Cela donne donc, en pensant à enlever le biais de l'exposant l'exposant :

$$2^{126-63} \times \left(\sum_{k=0}^{7} 2^{-k} \right) = 2^{63} \times \frac{1 - 2^{-8}}{1 - 2^{-1}} = 2^{64} \times (1 - 2^{-8}) \approx 1.84 \times 10^{19}$$

Exercice 4 (8 minutes).

Soit $n \in \mathbb{N}$. On note $x_n = 1.0 \times 10^n$.

1. Pour de grandes valeurs de n, que va-t-il se passer informatiquement lors de l'évaluation de $(x_n + 1) - x_n$? Commentez brièvement.

2. Pour de grandes valeurs de n, comparer les évaluations par la machine des expressions $1 + (x_n - x_n)$ et $(1 + x_n) - x_n$. Quelle propriété usuelle de l'addition a-t-on perdu lors du calcul sur machine ? (répondre avec concision et clarté)

3. On note $y_n = 1.0 \times 10^{-n}$. On considère les expressions : $\dfrac{(y_n + x_n) - x_n}{y_n}$ et $\dfrac{y_n + (x_n - x_n)}{y_n}$. Que se passera-t-il lors de l'évaluation de ces expressions par une machine pour de grandes valeurs de n ? (répondre avec concision et clarté)

Corrigé de l'exercice 4.

[Retour à l'énoncé]

1. L'évaluation de $(x_n + 1)$ va être effectuée en priorité. Malheureusement, comme x_n et 1 ont des ordres de grandeur très différents, le 1 va être absorbé par x_n et, pour n suffisamment grand, l'évaluation de $(x_n + 1)$ sera égale à x_n. Plus précisément, x_n s'écrit, en notation scientifique binaire :

$$x_n = 2^m \times \frac{10^n}{2^m}$$

où $m = \lfloor \log_2(10^m)$ est l'exposant tel 2^m est la plus grande puissance de 2 rentrant dans 10^n.

$$x_n + 1 = 2^m \times \frac{10^n}{2^m} + 1 = 2^m \times \left(\underbrace{\frac{10^n}{2^m}}_{\text{toujours compris entre 0 et 1}} + \quad 2^{-m} \right)$$

Plus n grandit, plus m grandit aussi au point que m devienne plus grand que le nombre de bits de la mantisse. Cela entraîne que 2^{-m} passe en dessous de la précision machine est abandonné. Le résultat final de cette évaluation est donc, au niveau machine :

$$2^m \times \frac{10^n}{2^m}$$

c'est-à-dire $10^n = x_n$! Ainsi, l'évaluation de $(x_n + 1) - x_n$ donnera 0 pour de grandes valeurs de n, alors qu'elle devrait donner 1. Cela est lié

2. L'expression $1 + (x_n - x_n)$ sera correctement évaluée à 1. $(1 + x_n) - x_n$, qui devrait être évaluée à 1, sera évaluée à 0 pour les mêmes raisons que précédemment. C'est la propriété d'associativité de l'addition entre entiers qui a été perdue. Cette propriété dit que, $\forall a, b, c \in \mathbb{Z}$, on a toujours :

$$(a + b) + c = a + (b + c)$$

3. $\dfrac{y_n + (x_n - x_n)}{y_n}$ sera correctement évalué à 1 car la soustraction $(x_n - x_n)$ a lieu entre deux valeurs ayant le même ordre de grandeur (et carrément identiques !). Par contre, l'expression $\dfrac{(y_n + x_n) - x_n}{y_n}$ qui d'un point de vue mathématique devrait donner le même résultat 1, sera évaluée à 0 par une machine pour de grandes valeurs de n. En effet, y_n sera absorbé par x_n pour de grandes valeurs de n pour les mêmes raisons que précédemment.

Exercice 5 (Interpréteur OCaml - 5 minutes).
Prévoir les réponses de l'interpréteur OCaml.

Corrigé de l'exercice 5. *[Retour à l'énoncé]*

```
        OCaml version 4.05.0

# let a = 1024;;
val a : int = 1024
# let b = 256.;;
val b : float = 256.
# let c = 128 + 256;;
val c : int = 384
# let d = 256 + 512.;;
Characters 14-18:
  let d = 256 + 512.;;
                ^^^^
Error: This expression has type float but an expression was expected of type
         int
# let e = 256. + 512.;;
Characters 8-12:
  let e = 256. + 512.;;
          ^^^^
Error: This expression has type float but an expression was expected of type
         int
# let b = 4 in 2 *. b;;
Characters 13-14:
  let b = 4 in 2 *. b;;
               ^
Error: This expression has type int but an expression was expected of type
         float
# let a = 16 and b = 4 in a + b;;
- : int = 20
# let a = 16 in let b = 4 * a;;
Characters 27-29:
  let a = 16 in let b = 4 * a;;
                            ^^
Error: Syntax error
# let b = let a = 16 in 4 * a;;
val b : int = 64
# let a = 2 in let b = 4 * a in a + b;;
- : int = 10
# let x =
let x = 1 in
let x = x + 1 in
x + 2;;
val x : int = 4
#
```