## I – Représentation binaire et conversion entre types

Dans la suite on considère que les entiers signés sont représentés en « complément à 2 » et que sur un octet, l'intervalle représenté est [-128 +127]

1. On considère des entiers sur 8 bits :
   Quelle valeur est représentée en « non signé » par `11011011` ?

*128+64+16+8+2+1 =* ***219***

   Même question si l'octet représente une valeur signée ?

*Deux façons de procéder (la configuration représente un nombre négatif)*

*A – On calcule* ***son opposé*** *en complément à 2*

*Complément à 1 : on inverse les `0/1` et on obtient `00100100`*

*Auquel on ajoute `1` : `00100101` soit 32 +4 + 1 =* ***37***

*Le nombre initial correspond donc à* ***-37***

*B – On sait que par principe du complément à 2, la valeur absolue du nombre négatif est obtenue (sur 8 bits) en déduisant de 256 la valeur du nombre vu comme un non signé.*
*Donc en signé, notre nombre est : - (256 – 219) =* ***-37****.*

2. Donnez la représentation sur 32bits (« simple précision ») de `2.0`. Exprimez le résultat sous forme d'une séquence de chiffres en hexadécimal (base 16).

*`2.0` s'écrit sous forme normalisée `1.0...0*2`$^{+1}$.*
*On rappelle qu'il y a 1 bit de signe, 8 bits d'exposant et 23 bits de mantisse.*
*La mantisse est donc formée de `1` suivi de `0` après la virgule (en nombre convenable). Sauf qu'on ne stocke pas le premier `1` dans un flottant normalisé puisqu'on sait retrouver cette valeur, ce qui permet de garder un chiffre significatif de plus. Donc dans la partie « mantisse », on aura 23 fois la valeur `0`.*
*Pour l'exposant normalisé on doit ajouter 127, on obtient donc 128 soit en binaire `10000000`*
*Le nombre est positif donc le bit de signe est `0`.*
*Soit au final `01000000000000000000000000000000` ou en découpant en groupes de 4 bits*
*`0100 0000 0000 0000 0000 0000 0000 0000`*
*on traduit chaque groupe par le symbole hexadécimal correspondant :*
***`40000000`*** *ou en notation C/C++:* ***`0x40000000`*** *le préfixe `0x` indique la base 16.*

3. Soit la séquence suivante notée en hexadécimal (base 16) : `c377c127`
   Donnez son équivalent en binaire, ce qui doit donner une séquence de 32 bits.

*`1100 0011 0111 0111 1100 0001 0010 0111`*

   Si cette séquence binaire représente un nombre flottant en 32 bits (« simple précision »), quelle est la valeur en base 10 de ce flottant ?

*Le bit le plus à gauche est le bit de signe : `1` donc* ***nombre négatif***
*Les 8 bits qui suivent : `10000110` (= 134) correspondent à la valeur de 127 + notre exposant, qui vaut donc 134-127 =* ***7****.*
*Les chiffres de la mantisse après la virgule correspond à la la séquence*
*`11101111100000100100111`*
*à laquelle il faut ajouter le 1 devant la virgule qui n'est jamais stocké. Donc,*
*`1,11101111100000100100111* 2`$^{7}$
*= `11110111,1100000100100111`*
*La partie entière* ***`11110111`*** *donne en décimal :****247***
*La partie fractionnaire `0,1100000100100111` donne : ½ + ¼ + 1/256 + 1/2048 + ... =*
***0,7545013427734375*** *(valeur calculée grâce à `bc` sous Unix). Au final* ***-247,7545013427734375*** *,*
*Le flottant dont on était parti en imprimant la représentation en hexa était -247,7545*

4. Soit le fragment de programme suivant (on suppose qu'un short occupe deux octets) :

```
short s = 128; char c;
     c = s;
     s = c;
```

Quelle est la valeur finale de `s` ?

*Représentation binaire sur 16 bits de 128 : `0000000 10000000`*
*Représentation binaire de c après l'affectation (avec débordement) : `10000000` qui représente donc maintenant -128*
*La conversion vers une plage d'entiers plus grande conserve le signe de la valeur, négative donc, et la valeur absolue : `1000000 10000000`*

*La valeur imprimée sera donc **-128**.*

5. On exécute le programme ci-dessous :

```
int main() {
  char a, b, c, k;
  a = b = SCHAR_MIN; // -128
  c = (a+b);
  c = c / 2; // c vaut l'équivalent de (a+b)/2;
  k = b-a;
  k = k / 2;
  k = a + k; // k vaut a + (b-a)/2;
  cout << "a: " << (int) a << ", b: " << (int) b << ", c: " << (int) c
       << ", k: " << (int) k << endl;

  a = b = SCHAR_MAX; // 127
  c = (a+b);
  c = c / 2;
  k = b-a;
  k = k / 2;
  k = a + k; // au final k vaut a + (b-a)/2
  cout << "a: " << (int) a << ", b: " << (int) b << ", c: " << (int) c
       << ", k: " << (int) k << endl;
  a = SCHAR_MIN +1; // pour avoir [-127 +127]
  b = SCHAR_MAX;
  c = (a+b);
  c = c / 2;
  k = b-a;
  k = k / 2;
  k = a + k; // au final k vaut a + (b-a)/2
  cout << "a: " << (int) a << ", b: " << (int) b << ", c: " << (int) c
       << ", k: " << (int) k << endl;
```

Impressions obtenues :

```
a: -128, b: -128, c: 0,  k: -128
```

*Le calcul de `(a+b)` donne un débordement sur 8 bits et laisse `0` dans `c`.*
*`b-a` vaut `0` donc la valeur finale de `k` est celle de `a`: `-128`*

```
a: 127,  b: 127,  c: -1, k: 127
```

*Le calcul de (a+b) donne un débordement sur 8 bits qui laisse dans `c` la représentation de `-2`.*
*Après division par 2, `c` vaut `-1`. Comme avant le calcul initial de `k` donne `0`.*

```
a: -127, b: 127,  c: 0,  k: -128
```

*Cette fois-ci le calcul de (a+b) donne `0` donc c vaudra `0`. Par contre `b-a` laisse dans `k` la représentation binaire de `-2 (11111110)` et la division par 2 donne `-1 (11111111)`. La somme de `k` et de `a (10000001)` donne -127 + (- 1 ) = `-128`*