3. Représentation des nombres entiers et réels en binaire en mémoire

1 Nombres entiers

1.1 Représentation binaire

Tout entier positif n peut s'écrire sous la forme :

$$n = c_q 2^q + c_{q-1} 2^{q-1} + ... + c_k 2^k + ... + c_1 2^1 + c_0 2^0$$

c_q=1 (q est le plus grand entier tel que $n/2^q \neq 0$ (division « entière » ou « euclidienne »)) et les autres c_k valent 0 ou 1. La suite :

$$c_q c_{q-1} ... c_1 c_0$$

constitue la représentation binaire de n.
Pour calculer les c_k on divise successivement n par 2 et à chaque fois on garde le reste, qui vaut 0 ou 1. La suite des restes constitue la liste des c_k par ordre de poids croissant : $c_0 ... c_q$.
Comme en notation décimale on écrit les forts poids à gauche.
Exemple : 29
suite des restes : 1 0 1 1 1 donc dans le bon ordre : 11101

1.2 Ecriture des entiers en binaire en mémoire

Les PC utilisés en TD codent les entiers (du type `int` que nous utiliserons principalement) dans des mots de 4 octets (=32 bits) dont un est réservé pour le signe (par exemple le plus à gauche, figure 1).

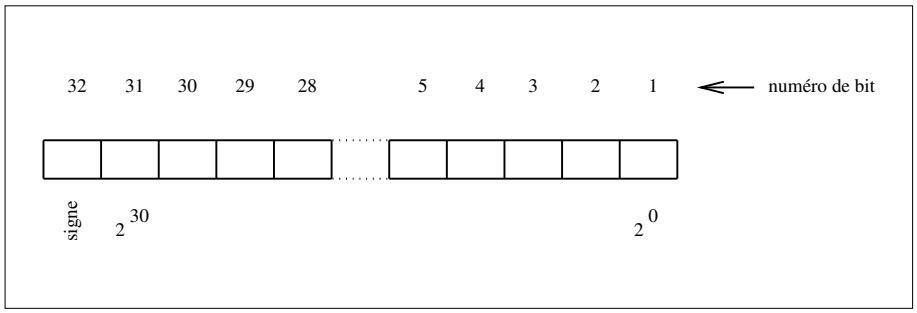

FIGURE 1 –

S'il s'agit d'un entier positif le bit de signe vaut 0 et l'entier lui-même est codé sur les 31 bits restants, on peut donc écrire tous les entiers de 0 à n_{max}=2^{31}-1=2 147 483 647 (ordre de grandeur 2 milliards).
S'il s'agit d'un entier négatif le bit de signe vaut 1. Dans les 31 bits restants on ne code pas la valeur absolue du nombre mais $n + 2^{31} = 2^{31} - |n|$ (cela simplifie l'exécution des opérations).

Exemple

codage de -1
le bit de signe vaut 1
sur les 31 bits on va coder $2^{31} - 1$=1111....1111 (31 fois 1)
donc au total sur les 32 bits des 1 partout

En pratique pour obtenir la représentation binaire de $-n$, n étant un entier positif, on soustrait, en binaire sur 32 bits n de 0 en laissant tomber la dernière retenue.
L'entier négatif de plus grande valeur absolue sera obtenu pour $2^{31} - |n| = 0$ donc n_{min}=-2^{31}= -2 147 483 648 (on gagne la place du 0).
Si lors d'une opération le résultat excède l'intervalle $[n_{min}, n_{max}]$, il n'y a aucun message d'erreur et le résultat est

totalement aberrant (exemple : 2 147 483 647+1=-2 147 483 648).

Exemple :
On calcule les factorielles successives à partir de 1. A partir d'un certain rang on va dépasser l'entier maximum représentable. Pour savoir à quel rang cela se produit, à chaque fois qu'on a calculé $n!$ on divise le résultat par n, c'est à dire qu'on calcule $n!/n$. Si on retrouve $(n-1)!$ c'est que la valeur de $n!$ calculée était correcte, sinon c'est qu'elle était fausse. On obtient :

1 !=1	1 !/1=1
2 !=2	2 !/2=1
3 !=6	3 !/3=2
4 !=24	4 !/4=6
5 !=120	5 !/5=24
6 !=720	6 !/6=120
7 !=5040	7 !/7=720
8 !=40320	8 !/8=5040
9 !=362880	9 !/9=40320
10 !=3628800	10 !/10=362880
11 !=39916800	11 !/11=3628800
12 !=479001600	12 !/12=39916800
13 !=1932053504	13 !/13=148619500
14 !=1278945280	14 !/14=91353234
15 !=2004310016	15 !/15=133620667

On constate que le calcul devient faux à partir de 13 inclus.
A part cette limitation en grandeur, les opérations arithmétiques entre entiers sont exactes (en n'oubliant pas que la division est la division euclidienne : partie entière du résultat de la division exacte, exemple, 2/3=0 et 5/4=1).
Sur les PC ayant un processeur 64 bits il est possible d'utiliser des entiers de type `long int` écrits sur 8 octets (64 bits) donc compris entre -2^{63}=-9 223 372 036 854 775 808 et 2^{63}-1=9 223 372 036 854 775 807 (ordre de grandeur 10 milliards de milliards).

2 Nombres réels

2.1 Représentation binaire

Tout réel positif r peut s'écrire sous la forme :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + c_1 2^1 + c_0 2^0 + c_{-1}2^{-1} + c_{-2}2^{-2} + ... + c_k 2^k + ...$$

avec $c_q = 1$ et les c_k valant 0 ou 1 (q peut être négatif). La suite :

$$c_q c_{q-1} ... c_1 c_0 . c_{-1} ... c_k ...$$

constitue la représentation binaire de r, en général infinie du côté des k négatifs (par convention on met un point entre c_0 et c_{-1}).
Les c_k ne sont jamais tous égaux à 1 à partir d'un certain rang k_0 en direction des k négatifs. En effet on aurait alors :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + c_{k_0+1}2^{k_0+1} + c_{k_0}2^{k_0} + c_{k_0-1}2^{k_0-1} + ... \quad \text{avec } c_{k_0+1} = 0 \text{ et } c_k = 1 \text{ pour } k \leq k_0$$

soit :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + 0 \times 2^{k_0+1} + 2^{k_0} + 2^{k_0-1} + ...$$

Mais, comme on a :

$$2^{k_0} + 2^{k_0-1} + 2^{k_0-2} + ... = 2^{k_0+1}$$

r s'écrirait :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + 2^{k_0+1} + 0 \times 2^{k_0} + 0 \times 2^{k_0-1} + ... = c_q 2^q + c_{q-1}2^{q-1} + ... + 2^{k_0+1}$$

c'est à dire que c_{k_0+1} passerait de 0 à 1 et tous les c_k pour $k \leq k_0$ passeraient de 1 à 0 et finalement le représentation binaire de r serait :

$$c_q c_{q-1} ... 100000 ...$$

Pour calculer c_k on divise d'abord r par 2^k, puis on prend la partie entière, et on prend le reste de la division par 2 de cette partie entière.
En effet :

$$r/2^k = ... + c_{k+1}2 + c_k 2^0 + c_{k-1}2^{-1} + ...$$

$c_{k-1}2^{-1} + ..$ est strictement inférieur à 1 puisque les c_k ne sont jamais tous égaux à 1 à partir d'un certain rang.
Pour $k \geq 0$ il suffit de calculer la représentation binaire de l'entier partie entière de r car on voit que la partie décimale n'intervient pas.

Exemple : 3.25

k				
...	...	terminé	...	...
2	3.25/4	0	0/2	0
1	3.25/2	1	1/2	1
0	3.25/1	3	3/2	1
-1	3.25× 2	6	6/2	0
-2	3.25× 4	13	13/2	1
-3	3.25× 8	26	26/2	0
...	...	terminé	...	...

terminé car on a que des nombres pairs
donc 3.25 s'écrit en binaire 11.01000... que des 0 exactement.

Autre exemple : .1

On part de $k = 0$ et on voit tout de suite qu'il n'y aura rien du côté des $k > 0$.

k	$.1/2^k$	$E(.1/2^k)$	reste $\left[E(.1/2^k)\right]/2$
0	.1/1=.1	0	0
-1	.1× 2=.2	0	0
-2	.2× 2=.4	0	0
-3	.4× 2=.8	0	0
-4	.8× 2=1.6	1	1
-5	1.6× 2=3.2	3	1

On voit alors que $3.2 = 3 + .2$: le 3 donnera toujours des multiples de 2 donc ne contribuera pas au reste, seul le .2 compte. Or le .2 a déjà été rencontré pour $k = -1$ donc à partir de $k = -6$ la séquence 0011 se répète à l'infini. La représentation binaire de .1 est donc 0.00011 0011 0011 ...

Exercice :

Montrer que la représentation binaire de .3 est également infinie et vaut 0.0 1001 1001 1001 ...

2.2 Ecriture des réels en binaire en mémoire

On utilisera les réels de type `double` qui, sur les PC utilisés en TD, sont écrits sur 8 octets (64 bits). Il existe aussi des réels écrits sur 4 octets (type `float` sur les PC utilisés en TD). Pour simplifier on va décrire le cas de 4 octets, mais tout sera directement transposable au cas de 8 octets.
On met 2^q en facteur dans l'expression du réel positif r quelconque vue ci-dessus :

$$r = \left(c_q 2^0 + c_{q-1}2^{-1} + ... + c_k 2^{k-q} + ...\right) \times 2^q$$

avec $k = q, q-1, ... -\infty$.
Puisque $c_q = 1$ et en écrivant en binaire le facteur entre parenthèses :

$$r = 1.c_{q-1}c_{q-2}...c_k... \times 2^q$$

Au total on a simplement décalé le point q fois vers la gauche dans l'expression binaire initiale de r et compensé en multipliant par 2^q.
La suite des c_k s'appelle la « mantisse binaire » .

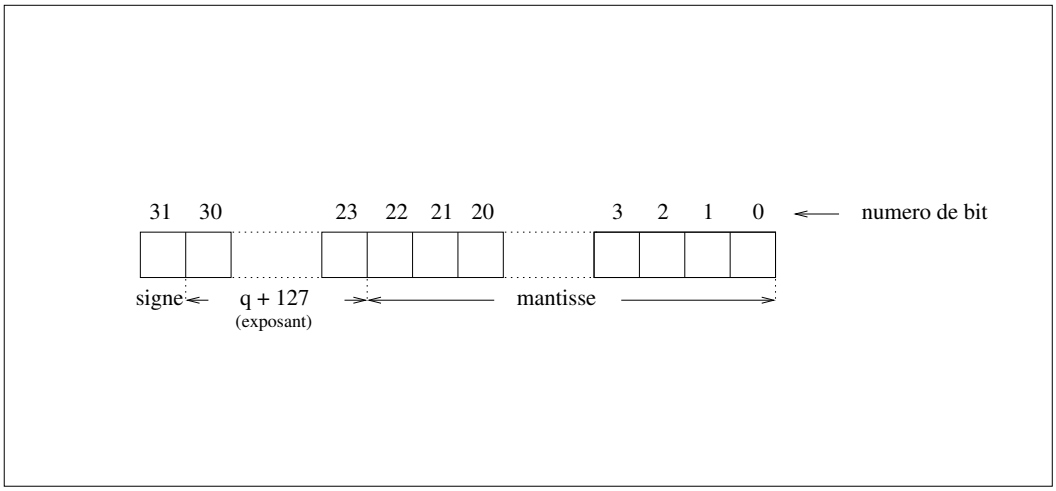

FIGURE 2 –

Le bit de gauche est pour le signe : 0 pour + et 1 pour −.
Les réels positifs et négatifs sont codés exactement de la même manière en dehors du bit de signe (contrairement au cas des entiers).
Dans les 8 suivants on met (en binaire) $q + 127$ (l'exposant est décalé de 127 pour n'avoir à stocker que des nombres positifs).
On pourrait donc à priori avoir $q+127$ variant de 0 à 255. Mais la valeur 0 est réservée pour signaler les nombres trop petits et la valeur 255 les nombres trop grands ou non définis. Ces deux valeurs 0 et 255 ont donc une signification spéciale. On a donc $1 \leq q + 127 \leq 254$, les valeurs extrêmes permises de q sont donc -126 et 127.
Dans les 23 restants on met les c_k.

2.2.1 Plus grand et plus petit nombre représentable

Cas général

Il découle de ce qui précède que la plus grande valeur absolue représentable est :
1.11...11 (23 fois 1 après le point)$\times\ 2^{127} = (2^{24}\text{-}1)\times\ 2^{104} \simeq 3.40282347\ 10^{38}$
et la plus petite :
1.00...00 (23 fois 0 après le point)$\times\ 2^{-126} = 2^{-126} \simeq 1.17549435\ 10^{-38}$.

Cas particulier des nombres sub-normaux

Tout ce paragraphe peut être sauté en première lecture.
En réalité du côté des petites valeurs il y a une astuce supplémentaire qui permet d'écrire des nombres jusqu'à une valeur absolue de $2^{-149} \simeq 1.4012984\ 10^{-45}$, mais avec de moins en moins de chiffres significatifs (nombres sub-normaux).
En effet lorsque l'exposant est nul mais la mantisse non nulle, on convient que le mot de 32 bits représente le nombre :

$$0.c_{q-1}c_{q-2}...c_k... \times 2^{-126} \quad \text{et non plus} \quad 1.c_{q-1}c_{q-2}...c_k... \times 2^q$$

La plus grande valeur absolue sub-normale est donc :
$0.11\ldots11$ (23 fois 1 après le point)$\times\ 2^{-126} = (2^{23}\text{-}1) \times\ 2^{-149} \simeq 1.17549421\ 10^{-38}$
et la plus petite :
$0.00\ldots01$ (22 fois 0 et 1 fois 1 après le point)$\times\ 2^{-126} = 2^{-149} \simeq 1.40129846\ 10^{-45}$

Récapitulation

Le tableau suivant résume les différents cas pour les nombres positifs, (le cas des nombres négatifs étant strictement identique au bit de signe près) :

	s i g n e	<q+127 >	< mantisse >	valeur decimale
Plus grand normal	0	11111110	11111111111111111111111	3.40282347 10+38
Plus petit normal	0	00000001	00000000000000000000000	1.17549435 10-38
Plus grand sub-normal	0	00000000	11111111111111111111111	1.17549421 10-38
Plus petit sub-normal	0	00000000	00000000000000000000001	1.40129846 10-45
Zero positif	0	00000000	00000000000000000000000	0.
Trop grand ou non def.	0	11111111	... ...	

2.2.2 Nombres représentables

L'ordinateur ne peut donc représenter de façon exacte qu'un nombre fini de nombres réels. Si on met de côté les nombres sub-normaux, il représente toutes les puissances de 2 de 2^{-126} à 2^{127} et, dans chaque intervalle $[2^p,2^{p+1}[$ [1], 2^{23} nombres puisque pour une valeur de l'exposant la mantisse peut prendre 2^{23} valeurs. Ces nombres sont régulièrement espacés de $(2^{p+1}-2^p)/2^{23} = 2^p/2^{23}$ puisque la mantisse augmente par pas de 2^{-23}. Dans l'intervalle immédiatement supérieur ils sont deux fois plus espacés (figure 3).
Dans l'intervalle $[2^{127},2^{128}[$ les nombres sont espacés de 2^{104}=202824096036516704239472512860 16.

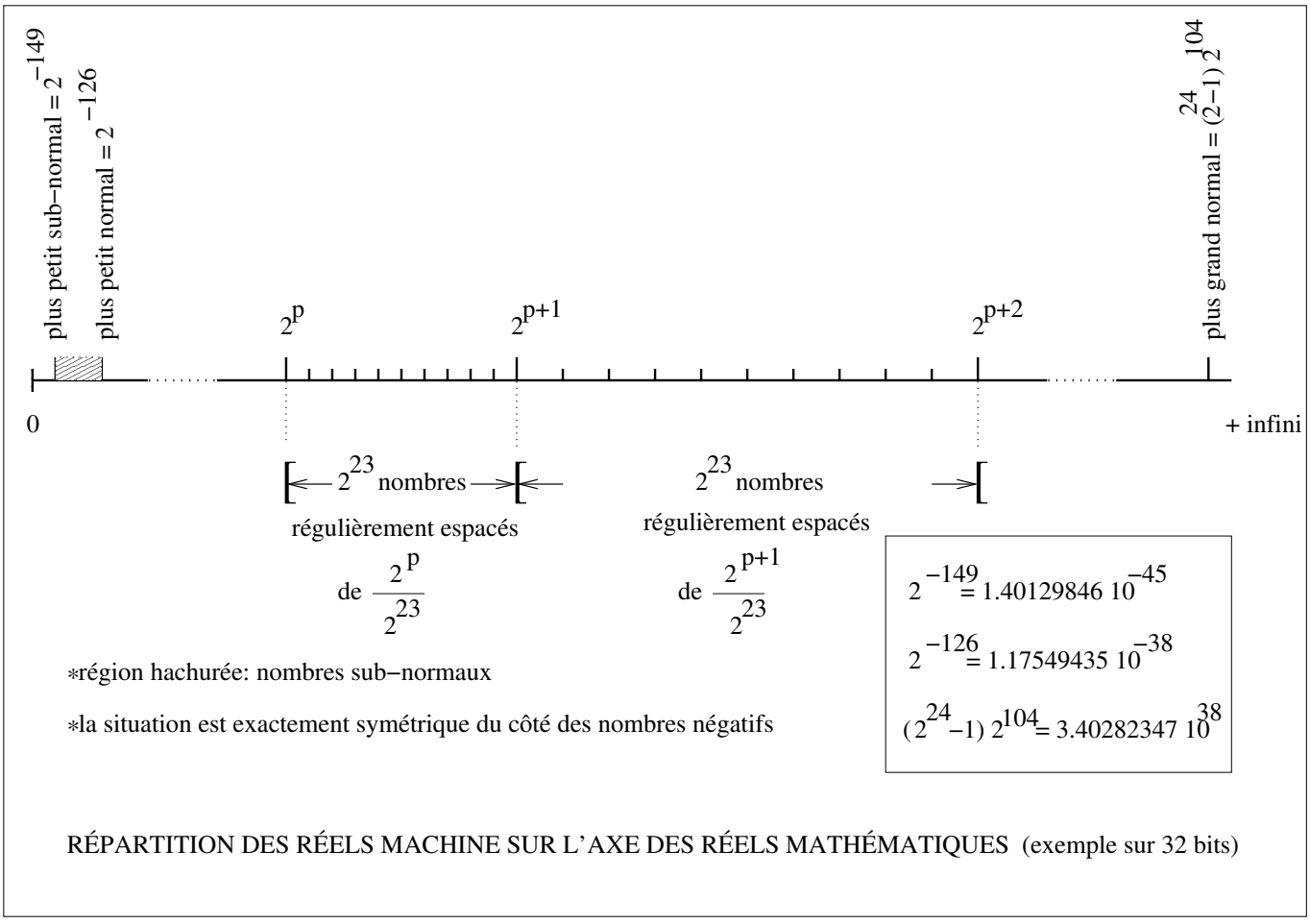

FIGURE 3 –

1. p pouvant donc varier de -126 à 127

Entre 2^{23} et 2^{24} les nombres sont espacés de 1 puis entre 2^{24} et 2^{25} espacés de 2. Ce qui fait que si on ajoute 1 à 2^{24} on obtient un nombre non représenté, puisque seuls le sont 2^{24} et $2^{24}+2$, et $2^{24}+1$ est tronqué à 2^{24}.
Ceci est illustré par le programme suivant [2] :

```
#include<iomanip>
#include<iostream>
using namespace std;
int main() {
  float x = 0., x1 = 1.;
  while(x<x1) {
    x = x1;
    x1 = x+1.;
  }
  cout << setprecision(16);
  cout << "On ne peut ajouter 1 en float que jusqu'à =" << x << endl;
  return 0;
}
```

dont le résultat est : *On ne peut ajouter 1 en float que jusqu'à =16777216*
C'est un peu comme si on ne pouvait soulever son pied que de trente centimètres et que l'on doive monter un escalier dont les marches deviennent de plus en plus hautes : il arriverait un moment où l'on resterait définitivement sur la même marche. On constate ainsi qu'avec un réel sur 4 octets on va moins loin pour énumérer les entiers de 1 en 1 qu'en utilisant un entier sur 4 octets (16777216 au lieu de 2147483647). C'est parce que de la place est perdue par l'exposant. Les réels sur 4 octets permettent de représenter des nombres bien plus grands que 2147483647 mais qui sont bien plus espacés que les entiers.

2.2.3 Troncature

Puisque les nombres représentables exactement sont en nombre fini un nombre quelconque est en général tronqué (à la valeur immédiatement inférieure, en valeur absolue), sauf s'il n'a que des zéros après le $23^{ième}$ chiffre de sa mantisse. Un réel ne peut être en général écrit qu'avec un nombre de chiffres significatifs exacts limité et il est important de connaître les limitations que cela entraîne.
On constate que des nombres qui s'écrivent très simplement en décimal comme .3 par exemple, n'ont pas une représentation exacte dans la machine : .3 est approché par .3000000119... . Donc dès l'introduction de ces nombres dans la machine il y a une certaine erreur, avant même tout calcul. Ensuite, après une opération le résultat sera lui-même tronqué ce qui entraîne une nouvelle erreur. Selon le calcul effectué les erreurs ainsi introduites peuvent se compenser plus ou moins, auquel cas l'erreur n'augmente que lentement au fil des opérations ou au contraire s'amplifier très rapidement et le résultat devient complètement faux.

Exercice :

Calculer les termes successifs de la suite :

$$u_0 = e - 1 \quad u_n = n u_{n-1} - 1 \quad e \text{ étant la base de la fonction exponentielle } 2.718..$$

Comparer les résultats obtenus avec des réels de 4 ou 8 octets.
Déterminer mathématiquement la limite de cette suite.

Remarque

Des nombres qui ont une représentation finie en binaire ont une représentation finie en décimale. Il n'y a donc pas de nouvelle erreur introduite quand on retraduit le résultat en décimal, à condition de prendre suffisamment de décimales.

Les nombres réels de type `double` sont écrits dans des mots de 8 octets (64 bits).

Le tableau suivant résume les limites et la précision des réels de type `float` et de type `double` avec les PC utilisés en TD :

2. Exemple emprunté à Francis Hecht

Type	Nb de bits signe	Nb de bits exposant	Nb de bits chiffres significatifs	Plus petite valeur absolue représentable	Plus grande valeur absolue représentable	Nb minimum de chiffres significatifs exacts en décimale
`float` (4 octets)	1	8	24	$\simeq 1.175\ 10^{-38}$ ($\simeq 1.401\ 10^{-45}$)	$\simeq 3.403\ 10^{38}$	6
`double` (8 octets)	1	11	53	$\simeq 2.225\ 10^{-308}$ ($\simeq 4.941\ 10^{-324}$)	$\simeq 1.798\ 10^{308}$	15

Dans le tableau ci-dessus les nombres sub-normaux sont placés entre parenthèses.

Rappel : $1.175\ 10^{-38}$ s'écrit en C : 1.175 e-38

Pour les nombres sub-normaux il n'y a pas de diagnostic, mais le nombre de chiffres significatifs exacts est moindre. Pour les valeurs absolues inférieures à la plus petite valeur sub-normale ou supérieures à la plus grande valeur normale il y a un diagnostic de dépassement.
Contrairement au cas des entiers il y a donc un diagnostic de dépassement pour les réels. Ces diagnostics seront vus en TD.

Remarques

Pour les nombres normaux le nombre de bits des chiffres significatifs est le nombre de bits de la mantisse plus un puisqu'ils s'écrivent 1.mantisse binaire $\times 2^q$.

Le mode de représentation des nombres que l'on vient de voir fait partie d'une norme (IEEE) qui se retrouve dans de nombreux langages de programmation.

Par contre le fait qu'en C un entier de type `int` ou un réel de type `float`, par exemple, soient écrits sur 4 octets ne fait pas partie de la norme du C. Cela peut varier d'un ordinateur à un autre et les valeurs données ici concernent les PC utilisés en TD. Mais le principe reste le même. On verra comment connaître précisément le nombre d'octets utilisés pour chaque type par un ordinateur donné.

3 Conclusion importante

Ce qui précède montre que si on interprète un mot de 4 octets par exemple, comme un entier ou comme un réel les deux résultats obtenus seront totalement différents.

Exercice

Se donner une suite quelconque de 4 octets et l'interpréter selon les règles vues précédemment soit comme un entier, soit comme un réel.

Annexe : suite $u_0 = e - 1 \quad u_n = nu_{n-1} - 1$

Programme :

```
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  float uf;
  double ud;
  int i, imax;
  imax = 100;
  cout << "n        float          double" << endl;
  for(uf = exp(1.)-1., ud = exp(1.)-1., i = 1; i <= imax; i++) {
    cout << i << "    " << uf << "    " << ud << endl;
    uf = i*uf-1.;
    ud = i*ud-1.;
  }
  return 0;
}
```

Résultat :

```
n        float          double
1    1.71828    1.71828
2    0.718282    0.718282
3    0.436564    0.436564
4    0.309691    0.309691
5    0.238765    0.238764
6    0.193824    0.193819
7    0.162943    0.162916
8    0.1406    0.140415
9    0.124802    0.123323
10    0.123215    0.109911
11    0.232147    0.0991122
12    1.55362    0.090234
13    17.6434    0.0828081
14    228.365    0.0765057
15    3196.1    0.0710802
16    47940.6    0.066203
17    767048    0.0592478
18    1.30398e+07    0.00721318
19    2.34717e+08    -0.870163
20    4.45962e+09    -17.5331
21    8.91923e+10    -351.662
22    1.87304e+12    -7385.9
23    4.12069e+13    -162491
24    9.47758e+14    -3.73729e+06
25    2.27462e+16    -8.96949e+07
26    5.68655e+17    -2.24237e+09
27    1.4785e+19    -5.83017e+10
28    3.99196e+20    -1.57415e+12
29    1.11775e+22    -4.40761e+13
30    3.24147e+23    -1.27821e+15
31    9.72441e+24    -3.83462e+16
32    3.01457e+26    -1.18873e+18
33    9.64661e+27    -3.80394e+19
34    3.18338e+29    -1.2553e+21
```

```
35    1.08235e+31    -4.26802e+22
36    3.78822e+32    -1.49381e+24
37    1.36376e+34    -5.37771e+25
38    5.04591e+35    -1.98975e+27
39    1.91745e+37    -7.56106e+28
40    inf    -2.94881e+30
41    inf    -1.17953e+32
42    inf    -4.83605e+33
43    inf    -2.03114e+35
44    inf    -8.73391e+36
45    inf    -3.84292e+38
46    inf    -1.72931e+40
47    inf    -7.95485e+41
48    inf    -3.73878e+43
49    inf    -1.79461e+45
50    inf    -8.79361e+46
51    inf    -4.3968e+48
52    inf    -2.24237e+50
53    inf    -1.16603e+52
54    inf    -6.17997e+53
55    inf    -3.33718e+55
56    inf    -1.83545e+57
57    inf    -1.02785e+59
58    inf    -5.85876e+60
59    inf    -3.39808e+62
60    inf    -2.00487e+64
61    inf    -1.20292e+66
62    inf    -7.33782e+67
63    inf    -4.54945e+69
64    inf    -2.86615e+71
65    inf    -1.83434e+73
66    inf    -1.19232e+75
67    inf    -7.86931e+76
68    inf    -5.27243e+78
69    inf    -3.58526e+80
70    inf    -2.47383e+82
71    inf    -1.73168e+84
72    inf    -1.22949e+86
73    inf    -8.85234e+87
74    inf    -6.46221e+89
75    inf    -4.78203e+91
76    inf    -3.58653e+93
77    inf    -2.72576e+95
78    inf    -2.09883e+97
79    inf    -1.63709e+99
80    inf    -1.2933e+101
81    inf    -1.03464e+103
82    inf    -8.3806e+104
83    inf    -6.87209e+106
84    inf    -5.70383e+108
85    inf    -4.79122e+110
86    inf    -4.07254e+112
87    inf    -3.50238e+114
88    inf    -3.04707e+116
89    inf    -2.68142e+118
90    inf    -2.38647e+120
91    inf    -2.14782e+122
```

```
92    inf    -1.95452e+124
93    inf    -1.79816e+126
94    inf    -1.67228e+128
95    inf    -1.57195e+130
96    inf    -1.49335e+132
97    inf    -1.43362e+134
98    inf    -1.39061e+136
99    inf    -1.3628e+138
100    inf    -1.34917e+140
```