TD n° 1 : Types de base

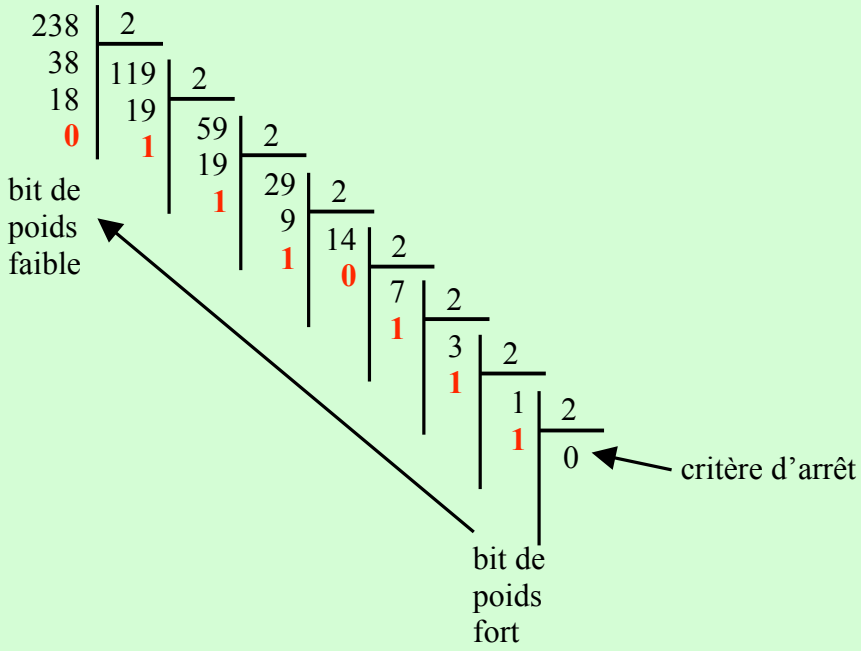
Exercice 1 : Codage des entiers naturels

a) Quel est le plus grand entier naturel que l'on puisse exprimer dans une base B si l'on dispose de L positions ? En conséquence, peut-on représenter le nombre d'électrons dans l'univers (soit un nombre de l'ordre de 10^{78}) dans un entier naturel codé sur 32 bits ?

Dans la base B, on a B chiffres (ou symboles) possibles à chaque position : 0, 1, 2, … B-1.
On a L positions. On peut donc exprimer B^L valeurs différentes, allant de 0 à $B^L - 1$.
Pour un entier codé sur 32 bits, B=2 et L=32, donc on peut exprimer tous les entiers naturels allant de 0 à $2^{32}-1 = 4\,294\,967\,295 \cong 4.10^9$. Donc on ne peut pas aller jusqu'au nombre d'électrons dans l'univers avec ce codage.

b) Convertissez en binaire le nombre 238_{10} en utilisant le principe des divisions successives. Déduisez-en sa représentation hexadécimale en regroupant les bits par paquets de quatre. Peut-on aussi utiliser une méthode de regroupement pour passer de l'écriture binaire de 238 à son écriture décimale ?

Conversion vers la base 2 :



Donc 238_{10} s'écrit en binaire : $1110\ 1110_2$

Valeur décimale des paquets : 14 14
Notation hexadécimale : E E

Remarque : pour éviter la confusion avec une autre base, on peut soit ajouter l'indice 16, soit le préfixe '0x'. On noterait donc ici EE_{16} ou 0xEE (bien insister sur le fait que le x n'a pas le même statut que les lettres de A à F : il ne faut pas multiplier 'x' par une puissance de 16 !).

On ne peut pas utiliser cette technique de regroupement pour passer de l'écriture binaire d'un nombre à son écriture décimale, car 10 n'est pas une puissance de 2. Remarque : le regroupement par paquets de trois donnerait l'écriture en base 8 (octal).

c) Réalisez l'addition binaire de 238 et de 32 après avoir représenté ces nombres en binaire pur sur 8 bits (unsigned char). Que constatez-vous ? Comment éviter ce problème ici ?

```
     1 1 1 0 1 1 1 0
+    0 0 1 0 0 0 0 0
   1 1 1                    (retenues)
 ------------------
  X 0 0 0 0 1 1 1 0
```

On constate que le résultat ne tient pas sur 8 bits : on a un bit surnuméraire qui est ignoré. Ainsi, 238 + 32 = 14 ! Il s'agit d'un dépassement de capacité (overflow). On peut éviter ce problème en utilisant des variables de type « unsigned int » par exemple.

d) **Question à faire chez soi :** Convertissez 238 en base 16 en utilisant la méthode des divisions successives (divisions par 16). Vérifiez que vous obtenez bien la même représentation qu'en regroupant les bits du codage binaire par paquets de quatre.

Conversion vers la base 16 :

238	16
78	14
14	14
	0 ← critère d'arrêt

Donc $238 = 14*16^0 + 14*16^1$. En hexadécimal, les nombres supérieurs à 9 sont représentés par des lettres : A pour 10, B pour 11, … jusqu'à F pour 15. Donc 238_{10} s'écrit en hexadécimal : EE.

Exercice 2 : Codage des entiers relatifs

a) Quelle est la représentation de -3 et +10 en « binaire - complément à 2 » sur 8 bits (signed char) ?

-3 est négatif donc on passe par les trois étapes suivantes :
- codage de la valeur absolue : $3_{10} = 00000011_2$
- inversion de tous les bits : 11111100
- ajout de 1 : 11111101

Donc -3 est représenté par 11111101_2

+10 est positif donc on se contente de l'exprimer en base 2 : 00001010_2

b) Additionnez à présent les deux nombres (en binaire). Le résultat est-il correct ? Aurait-on obtenu un résultat correct en additionnant les représentations en « signe-valeur absolue » des deux nombres ?

Addition binaire des deux représentations :

```
     1 1 1 1 1 1 0 1    (-3)
+    0 0 0 0 1 0 1 0    (+10)
  1 1 1 1 1             (retenues)
-------------------
  1 0 0 0 0 0 1 1 1
```

En ignorant le bit surnuméraire, on obtient +7, donc le résultat est correct. Donc, pour les entiers relatifs codés en complément à 2, l'apparition d'un bit surnuméraire n'implique pas un dépassement de capacité.

En faisant l'addition binaire des représentations en « signe – valeur absolue », on n'obtient pas la bonne valeur. Le codage en complément à 2 a l'immense intérêt de permettre à la machine de considérer la soustraction de deux entiers comme l'addition du premier à l'opposé du second.

Exercice 3 : Codage des réels

a) Convertir 9 et 5/32 au format IEEE 754 simple précision. Donnez les résultats en écriture binaire et en écriture hexadécimale. Additionnez ces deux nombres.

Remarque pour les encadrants : ici, les deux nombres sont représentables exactement, mais ce n'est pas toujours le cas. La méthode donnée ci-dessous n'est donc pas générale. (Et on ne demandera pas à l'examen le codage d'un nombre non représentable exactement : les différentes méthodes d'arrondi ne sont pas au programme de l'UE. Les étudiants doivent cependant savoir que tous les nombres ne sont pas représentables exactement).

Représentation de 9 :
Il s'agit d'un nombre positif donc bit de signe = 0
On sait que $9 = 2^3 + 2^0 = 1001_2 = 1001,0_2$
Normalisation (décalage de la virgule derrière le 1 le plus à gauche) = $1,001_2.2^3$
Identification : $(1,f).2^{e-127} = 1,001_2.2^3$ avec f sur 23 bits et e sur 8 bits
f = 00100000000000000000000
e - 127 = 3 soit $e = 130_{10} = 10000010_2$
Donc la représentation IEEE 754 simple précision de 9 est :
0 10000010 00100000000000000000000
Ecriture hexadécimale : (0x)41100000

Représentation de 5/32 :
Il s'agit d'un nombre positif donc bit de signe = 0
$5/32 = (2^2 + 2^0) / 2^5 = 2^{-3} + 2^{-5} = 0,00101_2$
Normalisation (décalage de la virgule derrière le 1 le plus à gauche) = $1,01_2.2^{-3}$
Identification : $(1,f).2^{e-127} = 1,01_2.2^{-3}$

f = 01000000000000000000000
e - 127 = -3 soit e = $124_{10} = 01111100_2$
Donc la représentation IEEE 754 simple précision de 5/32 est :
0 01111100 01000000000000000000000
Ecriture hexadécimale : (0x)3E200000

Addition :
Il faut dénormaliser l'une des deux mantisses pour ramener les deux nombres au plus grand exposant. C'est 5/32 qui a le plus petit exposant, donc c'est lui qui est dénormalisé.
$5/32 = 1{,}01_2.2^{-3} = 0{,}00000101_2.2^3$
Somme des deux mantisses :

```
    1,00100000  (.2^3)
+   0,00000101  (.2^3)
--------------------
    1,00100101  (.2^3)
```

Renormalisation du résultat : pas nécessaire ici, il est déjà normalisé.
Représentation IEEE 754 simple précision de 9+5/32 :
0 10000010 00100101000000000000000
Ecriture hexadécimale : (0x)41128000

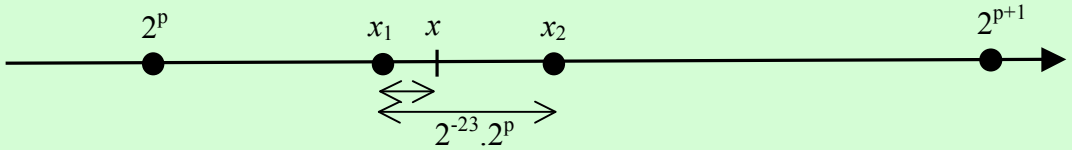
b) Pour le format IEEE simple précision, quel est l'écart entre deux nombres machines successifs ? En conséquence, combien y a-t-il de chiffres décimaux significatifs dans un nombre codé dans ce format ? A votre avis, écrire `float pi=3.14159265358979` a-t-il un sens ?

NB : Vous pouvez corriger cette question directement, sans trop laisser les étudiants chercher : il est peu probable qu'ils trouvent la réponse tout seuls.

Soient x_1 et x_2 deux nombres machine consécutifs de la forme $1,\ldots*2^p$ (ils sont donc compris entre 2^p et 2^{p+1}). Dans le format IEEE 754 simple précision, la mantisse est codée sur 23 bits, donc $|x_2 - x_1| = 2^{-23}.2^p$ (écart *absolu*).

Pour connaître le nombre de chiffres significatifs d'un float, il faut estimer l'écart *relatif* entre x_2 et x_1. Comme x_2 et x_1 sont de l'ordre de 2^p, l'écart relatif $|(x_2 - x_1)/x_1|$ est de l'ordre de 2^{-23}, soit environ 10^{-7}. Un nombre codé au format IEEE 754 simple précision a donc 7 chiffres décimaux significatifs. Ainsi, écrire `float pi=3.14159265358979` n'a pas de sens : on ne peut pas garantir une telle précision dans un float.

Démonstration plus rigoureuse (ne la donner que si on vous la demande) :
Pour connaître le nombre de chiffres significatifs d'un nombre codé dans ce format, il faut estimer la précision du codage, c'est-à-dire l'erreur *relative* commise lorsqu'on représente un nombre quelconque par le nombre machine le plus proche. Soit à présent un nombre x quelconque compris entre x_1 et x_2. Supposons qu'il soit plus proche de x_1, et donc que sa représentation machine soit x_1 (on peut faire le même raisonnement avec x_2).



On a :
$|x - x_1| \leq |x_2 - x_1|$
$|x - x_1| \leq 2^{-23}.2^p$

On voit que l'erreur *absolue* commise en représentant un nombre réel x par le nombre machine le plus proche dépend de l'exposant (p), donc de la valeur de x. Il nous faut l'erreur *relative*.

On a $2^p \leq x$. Donc $2^{-23}.2^p \leq 2^{-23}.x$. Ainsi :
$|x - x_1| \leq 2^{-23}.2^p \leq 2^{-23}.x$

En ne gardant que les termes extrêmes :
$|x - x_1| \leq 2^{-23}.x$
$|(x - x_1)/x| \leq 2^{-23}$

L'erreur relative commise en représentant un nombre réel x par le nombre machine le plus proche est donc (au pire) de l'ordre de $2^{-23} \cong 10^{-7}$. Un nombre codé au format IEEE 754 simple précision a donc 7 chiffres décimaux significatifs.

Exercice 4 : Opérateurs et conversions implicites en C

Etant données les définitions ci-dessous,

```
char monChar1 = 'a', monChar2 = 97;
int monInt = 1;
long monLong = 1L;
float monFloat = 1.0f;
double monDouble = 1.0;
```

donner le type et la valeur de chacune des expressions suivantes :

a) `monChar1 + monChar2`
int, 194 (promotion numérique des char en int, et le code ASCII de 'a' est 97)

b) `monChar1 + 1`
int, 98

c) `monChar1 + 1.0`
double, 98.0

d) `monChar1 + 1.0f`
float, 98.0f

e) `monFloat + monFloat`
float, 2.0f

f) `monChar1 + monInt + monLong`

long, 99L

```
g)  monChar1 + monInt + monLong + monFloat + monDouble
```

double, 101.0

```
h)  monChar1 = monChar1 + monInt + monLong + monFloat + monDouble
```

char, 'e' ou 101

La somme est de type double mais est convertie en char lors de l'affectation. La valeur de l'expression est la valeur de retour de l'opérateur =, c'est-à-dire la valeur mise dans la variable. Le fait que l'affectation renvoie quelque chose (en plus de faire l'affectation elle-même) est une particularité du C. C'est pour cela qu'écrire par mégarde if(i=2) au lieu de if(i==2) ne provoque pas d'erreur de compilation : i=2 a bien une valeur (2 si i est un int), donc la ligne est syntaxiquement correcte. Elle équivaut à écrire if(2), ce qui est toujours vrai (en C, tout ce qui n'est pas 0 est vrai).

```
i)  monChar1 == monChar2
```

1, autrement dit vrai. En profiter pour rappeler la différence entre = et == en C (et rappeler aussi que dans la norme algorithmique, la comparaison se fait avec un = et l'affectation avec une flèche).

```
j)  monChar1 = monChar2 + monInt + monLong + monFloat + monDouble, monFloat = 2
```

float, 2.0f

La valeur d'une expression contenant des virgules est celle de la dernière opération. L'expression indiquée vaut donc la même chose que l'expression monFloat = 2, soit la valeur de retour de l'affectation.

Exercice 5 : Algorithmique sur des entiers relatifs

On dispose, dans le langage algorithmique, d'un type `booleen` pouvant prendre les valeurs `{vrai, faux}` et sur lequel on peut effectuer des opérations de comparaison (opérateur =), d'affectation (opérateur ←) et de négation (fonction `non(b: booleen): booleen`). On définit alors le type « Entier64 » comme suit :

```
type Entier64 = tableau[1..64] de booleens
```

Un Entier64 permet de coder un entier compris entre -2^{63} et $2^{63} - 1$, de façon identique au codage dans un *signed char*, un *short* ou un *int*, mais sur plus de bits.

a) Ecrire la fonction `addition(e1: Entier 64, e2: Entier64): Entier64` qui retourne la somme de 2 Entier64. On utilisera uniquement les opérations de comparaison, d'affectation et de négation de booléens. On pourra recourir à une variable interne pour la gestion de la retenue. Remarque : on bénéficie toujours du type `entier` « normal », qui permet de gérer des entiers qui restent relativement petits, comme l'indice d'un tableau.

> Remarques :
>
> 1) Dans notre pseudo-langage, un tableau est un type (composé d'un certain nombre de variables contigues de même type) et une fonction peut retourner un tableau. Malheureusement, on verra dans les cours ultérieurs que la mise en oeuvre de ce concept en C est un peu différente : le retour d'un tableau dans une fonction est un peu délicat !

2) Avant la norme C99, pas de type bool en C (il faut utiliser des entiers). En C99, on peut déclarer des variables de type bool, mais elles restent codées en interne comme des petits entiers. Donc, en réalité, il ne serait pas optimal en termes de taille mémoire d'implanter le type Entier64 comme un tableau de bool.

C'est donc un bon moment pour rappeler l'intérêt de distinguer pseudo-langage et langage de programmation…

```
Fonction addition(e1 : Entier64, e2 : Entier64) : Entier64
Précondition : e1+e2 doit appartenir à l'intervalle -2^63 .. 2^63−1
Résultat : Renvoie la somme de e1 et e2
Paramètres en mode donnée : e1, e2
Variables locales :
      retenue : booleen
      res : Entier64
      i : entier
Début
      retenue ← faux

      {Dans le langage algo, l'indice du tableau va de 1 à 64}
      Pour i allant de 64 à 1 par pas de -1 Faire
            Si e1[i] = e2[i] Alors
                  res[i] ← retenue
                  retenue ← e1[i]
            Sinon
                  res[i] ← non(retenue)
            Fin si
      Fin pour

      Retourne res
Fin addition
```

b) Ecrire une procédure `incremente(e: Entier64)` qui ajoute 1 un Entier64, en utilisant la fonction d'addition. On supposera que l'on a déjà écrit la procédure d'affectation entre Entier64 :

```
Procédure affecte(e: Entier64, val: Entier64)
Précondition : val doit appartenir à l'intervalle -2^63 .. 2^63−1
Postcondition : e+ = val
Paramètre en donnée-résultat : e
Paramètre en donnée : val
```

```
Procedure incremente(e : Entier64)
Précondition : e+1 doit appartenir à l'intervalle -2^63 .. 2^63−1
Postcondition : e+ = e− + 1
Paramètre en mode donnée-résultat : e
Variables locales
      un : Entier64
      i : entier
Début
      Pour i allant de 1 à 63 par pas de 1 Faire
            un[i] ← faux
      Fin pour
      un[64] ←vrai

      affecte(e, addition(e, un))
Fin incremente
```

Remarque aux encadrants : Ce n'est pas la façon la plus efficace de procéder. En effet, l'appel à addition revient à parcourir l'ensemble des bits de la variable à incrémenter. Or un tel parcours exhaustif n'est pas forcément nécessaire. Si le bit de poids faible est à 0, il suffit de le mettre à 1. Sinon, on remonte vers les bits de poids fort en mettant au passage les bits parcourus à 0, et on transforme le premier zéro rencontré en 1. On ne parcourt que la plus longue séquence de 1, et pas tous les bits de e. Attention : dans la boucle, il faut tenir compte du fait qu'on peut ne rencontrer aucun 0 (si l'Entier64 est rempli de 1, c'est-à-dire s'il code l'entier -1). Vous pouvez demander à ceux qui s'ennuieraient d'essayer d'écrire cette version :

```
Procédure incrémente(e : Entier64)
Précondition : e+1 doit appartenir à l'intervalle -2^63 .. 2^63–1
Postcondition : e+ = e– + 1
Paramètre en mode donnée-résultat : e
Variables locales
      i : entier
Début
      i ← 64
      Tant que (i>0 et e[i]=vrai) Faire
            e[i] ← faux
            i  ← i - 1
      Fin tant que
      {en sortie de boucle, on a e[i]=faux ou i=0}
      Si i ≠ 0 Alors
            e[i] ← 1
      Fin si
Fin incremente
```

Questions à faire chez soi :

c) Supposons que l'on ait déjà écrit les sous-programmes suivants, en plus des précédents :

```
Procédure initialiseZero(e: Entier64)
Précondition : aucune
Postcondition : e+ = 0
Paramètre en donnée-résultat : e

Procédure decremente(e: Entier64)
Précondition : l'entier e-1 doit appartenir à l'intervalle -2^63 .. 2^63–1
Postcondition : e+ = e- - 1
Paramètre en donnée-résultat : e

Fonction estSuperieur(e1: Entier64, e2: Entier64) : booleen
Précondition : e1 et e2 appartiennent à l'intervalle -2^63 .. 2^63–1
Résultat : retourne vrai si e1 > e2, faux sinon
Paramètres en mode donnée : e1, e2
```

Proposez alors une fonction de calcul du produit de 2 Entier64 qui ne fasse intervenir que des opérations d'addition et de décrémentation d'Entier64, en plus des opérations de comparaison et d'affectation d'Entier64. On utilisera 4 variables internes de type Entier64 : `zeroE64`, `res`, `nb_ajouts_restants` et `qte_ajoutee`. On écrira l'algorithme pour le cas où e1 et e2 sont positifs, puis on expliquera comment procéder dans le cas plus général (sans écrire l'algorithme).

L'algorithme donné en ci-dessous ne traite que le cas où e1 et e2 sont positifs. Dans le cas le plus général, on se ramène à calculer le produit p de valabs(e1) et valabs(e2) puis à renvoyer p ou -p suivant le signe de e1 et de e2.

```
Fonction multiplication(e1:Entier64, e2:Entier64) : Entier64
Précondition : e1*e2 doit appartenir à l'intervalle -2^63 .. 2^63–1
Résultat : Renvoie le produit de e1 et e2
Paramètres en mode donnée : e1, e2
Variables locales :
      res, nb_ajouts_restants, qte_ajoutee : Entier64
      zeroE64 : Entier64
Début
      initialiseZero(res)
      initialiseZero(zeroE64)
      affecte(nb_ajouts_restants, e1)
      affecte(qte_ajoutee, e2)
      Tant que (estSuperieur(nb_ajouts_restants, zeroE64) Faire
            affecte(res, addition(res,qte_ajoutee))
            decremente(nb_ajouts_restants)
      Fin tantque
Fin multiplication
```

d) Améliorez l'algorithme précédent en supposant que l'on dispose aussi des sous-programmes suivants :

```
Procédure multiplieParDeux(e: Entier64)
Précondition : e et 2*e doivent appartenir à l'intervalle -2^63 .. 2^63–1
Postcondition : e+ = 2*e-
Paramètre en donnée-résultat : e

Procédure diviseParDeux(e: Entier64)
Précondition : e et e/2 doivent appartenir à l'intervalle -2^63 .. 2^63–1
Postcondition : e+ = e-/2 (attention, c'est une division entière)
Paramètre en donnée-résultat : e

Fonction estImpair(e: Entier64) : booleen
Précondition : e appartient à l'intervalle -2^63 .. 2^63–1
Résultat : retourne vrai si e est impair faux sinon
Paramètres en mode donnée : e
```

Indice : posez la multiplication en binaire.

Là encore, l'algorithme donné ne fonctionne que si e1 et e2 sont positifs. Même principe que précédemment pour le généraliser à des nombres négatifs.
Exemple avec e1 = 23 et e2 = 17 :

```
          10001   (e2)
x         10111   (e1)
---------------
          10001   (1* e2)
+        10001.   (1* 2*e2)
+       10001..   (1* 2*2*e2)
+      00000...   (0* 2*2*2*e2)
+     10001....   (1* 2*2*2*2*e2)
```

La quantité à (éventuellement) ajouter est multipliée par deux à chaque étape en la décalant d'un cran vers la gauche. Pour savoir si on va effectivement l'ajouter, on regarde si le bit de e1 vaut 1 ou non. La première fois, i.e. pour le bit de poids faible, il suffit de tester si e1 est impair. Pour passer au bit suivant, on décale e1 vers la droite (=division entière par 2) et on teste si le dernier bit du nombre obtenu vaut 1 (= on teste si le nombre est impair).

```
Fonction multiplication(e1 : Entier64, e2 : Entier64) : Entier64
Précondition : e1*e2 doit appartenir à l'intervalle -2^63 .. 2^63–1
Résultat : Renvoie le produit de e1 et e2
Paramètres en mode donnée : e1, e2
Variables locales :
      res, nb_ajouts_restants, qte_ajoutee : Entier64
      zeroE64 : Entier64
Début
      initialiseZero(res)
      initialiseZero(zeroE64)
      affecte(nb_ajouts_restants, e1)
      affecte(qte_ajoutee, e2)

      Tant que estSuperieur(nb_ajouts_restants, zeroE64) Faire
            Si estImpair(nb_ajouts_restants) alors
                  affecte(res, addition(res,qte_ajoutee))
            Fin si
            multiplieParDeux(qte_ajoutee)
            diviseParDeux(nb_ajouts_restants)
      Fin tantque
Fin multiplication
```

e) Comparez les deux méthodes de multiplication en estimant le nombre d'opérations à effectuer :
   a) nombre de tours de boucle
   b) pour chaque tour, les nombres maximum de comparaisons, d'additions, d'affectations et de décrémentations, de tests de parité, de divisions par 2 et de multiplications par 2.

Quelle méthode vous semble la plus efficace ? Pourquoi ?

Première méthode : e_1 tours de boucle avec à chaque fois 1 comparaison, 1 addition, 1 affectation et 1 décrémentation.

Deuxième méthode : le nombre de passages dans la boucle correspond au nombre de bits nécessaires pour écrire e_1 en base 2. S'il faut n bits pour écrire e_1, cela signifie qu'il est compris entre 2^{n-1} et 2^n.

$2^{n-1} \leq e_1 < 2^n$

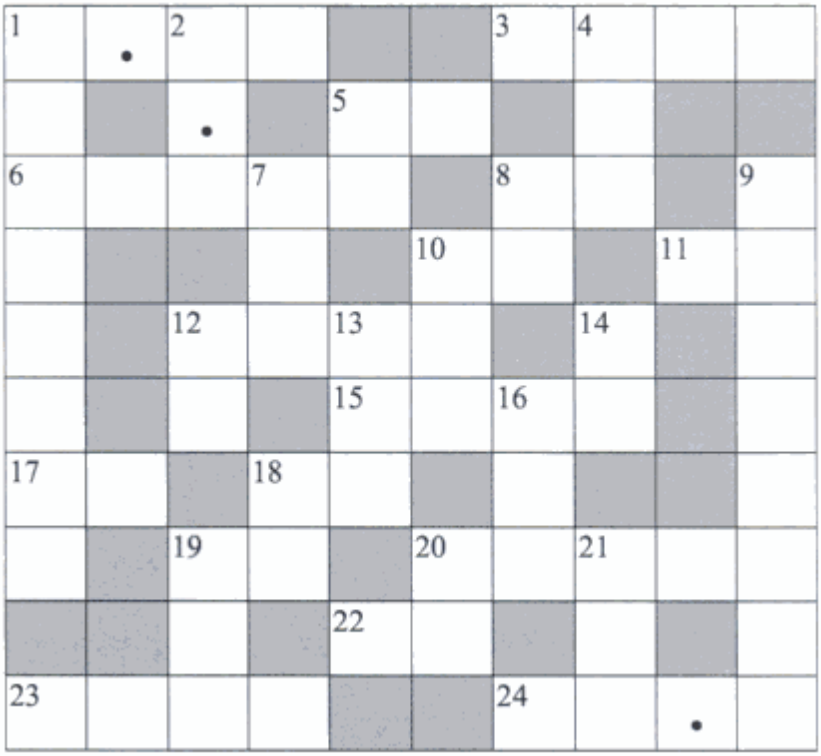
$(n-1).\ln(2) \leq \ln(e_1) < n.\ln(2)$ (ln fonction croissante)

$\ln(e_1))/\ln(2) < n \leq \ln(e_1)/\ln(2) + 1$ (ln(2) positif)

Par exemple pour e_1=23, on obtient $4.52 < n \leq 5.52$, soit n=5 bits donc 5 tours de boucle.
On a donc au pire $\ln(e_1)/\ln(2) + 1$ tours de boucle. A chaque passage, on fait 1 comparaison, 1 test de parité, 1 multiplication par 2, une division par 2, et éventuellement une addition et une affectation.

La multiplication par 2 et la division entière par 2 ne sont pas coûteuses puisqu'il s'agit de simples décalages de bits. Le test de parité n'est pas coûteux non plus. L'opération la plus coûteuses est en fait l'addition. La seconde méthode est donc la plus rapide : on fait toujours moins de tours de boucle qu'avec la première méthode, et en plus, on ne fait pas forcément l'addition à chaque tour (cela dépend de la valeur de e_1).

Bonus pour vous entraîner !



Chaque case peut contenir un chiffre entre 0 et 9, une lettre entre A et F, un point, ou encore le signe - .

Vous supposerez que les nombres binaires sur 8 bits sont sous la forme « complément à 2 » (donc signed), sauf si autre chose est précisé.

Précision sur le codage BCD (non vu en cours mais simple à comprendre) : En BCD, les nombres sont représentés en chiffres décimaux et chacun de ces chiffres est codé sur quatre bits. Exemple : Pour coder le nombre 127, il suffit de coder chacun des chiffres 1, 2 et 7 ce qui donne 0001, 0010, 0111.

Horizontalement

1) 1.75 en binaire : 1.11
3) D_{16} en binaire : 1101
5) 00101100 en décimal : 44
6) 22 en binaire : 10110
8) 01001000 en décimal : 72
10) 00010010 en décimal : 18
11) 01110001_{BCD} en décimal : 71
12) -6 en binaire sur 4 bits : 1010
15) pseudo-mantisse de 31 lorsqu'il est stocké dans un float (sans les 0 finaux) : 1111
17) pseudo-mantisse de 0,875 lorsqu'il est stocké dans un float (sans les 0 finaux) : 11
18) 00001011 en décimal : 11
19) 11110111 en décimal : -9
20) -3 en binaire sur 5 bits : 11101
22) 01010000_{BCD} en décimal : 50
23) 10001111 en décimal : -113
24) 2.5 en binaire : 10.1

Verticalement

1) -94 en binaire sur 8 bits : 10100010
2) 1.5 en binaire : 1.1
4) $7A_{16}$ en décimal : 122
5) 00101000 en décimal : 40
7) 416 en hexadécimal : 1A0
8) valeur en « Vertic. 4 » - valeur en « Horiz. 5 », en décimal : 78
9) -1 en binaire sur 8 bits : 11111111
10) 01100001 – 01011100 en binaire : 101
12) 11100110 + 00011101 en binaire : 11
13) 01101111 en décimal : 111
14) 00101001 en décimal : 41
16) l'exposant de $1/2^{122}$ quand ce nombre est stocké dans un float (écrire l'exposant en binaire, en tenant compte du décalage) : 101
18) 00010011 en décimal : 19
19) 10111001 en décimal : -71
20) 00111110 + 11000100 : 10
21) 384 en hexadécimal : 180