TP 04 : Entiers et virgule flottante

Introduction: Ce TP est consacré à des exercices sur les entiers et les nombres à virgule flottantes.

1 Un peu de stéganographie

La stéganographie est l'*art* de cacher du texte dans une image. Il existe de nombreuses façons pour le faire, mais celle qui nous intéresse est plutôt simple et en rapport avec le sujet du TP. Vos images peuvent s'encoder de plusieurs façons (d'où les différents format `jpeg`, `png`, `gif`, ...). Mais l'idée reste toujours la même : une image c'est un rectangle composé de pixels. Chaque pixel étant lui-même découpé en trois composantes : **R**ouge, **V**ert, **B**leu (**RGB** en anglais). Ces composantes généralement prennent des valeurs entre 0 et 255, ce qui fait que l'on peut coder un total de 2^{24} couleurs ! Cependant, à l'oeil nu, il est impossible de faire la distinction entre toutes ces couleurs. L'idée est donc d'altérer très légèrement les pixels de l'image pour y stocker notre message. Pour cela, on va considérer que notre message est une suite de 0 et de 1. Puis, on va parcourir nos pixels 1 à 1 et pour chaque composante, le bit de poid faible prendra comme valeur le prochain bit du message à afficher.

Par exemple, supposons qu'on a un pixel dont les composantes RGB sont : $19, 38, 209$, ce qui correspond à cette couleur. Si les trois premiers bits que l'on souhaite encoder sont 011 alors le nouveau pixel sera : $18, 39, 209$, ce qui correspond à cette couleur, vous voyez une différence ?

— À l'adresse suivante `http://www.lsv.fr/~fthire/teaching/2018-2019/archisys/TP/4/image-hidden.png`, vous trouverez une image dans lequel un message a été caché avec la technique présentée ici. Pouvez-vous retrouver le contenu du message ?

— Créer une fonction qui permet de faire l'opération inverse, à savoir étant donné une image et un texte, cache ce texte dans l'image

Pour cet exercice, vous êtes relativement libre d'utiliser le language de programmation de votre choix (je vous invite à utiliser C ou OCaml). Je vous invite aussi à utiliser une librairie pour décoder un fichier au format `.png`.

2 Décoder un entier

Le nombre suivant (en base 10) encode un message. Comment pouvez-vous le décoder ?

```
180560002996863632561850952862120653275409698306190547063566088784214130595792 7
↪ 0545913845431718826564612867924226501704759373213690067 7
↪ 220168228194519515223623063568138322186827982892004398
```

À votre avis, à quoi ça sert de représenter un message sous forme d'un entier en base 10 ?

3 Opérations binaires

Le langage C dispose de mécanismes de manipulation de bits. Par exemple, considérons deux variables x et y de type entier et l'opérateur $\oplus$ (xor). Notons respectivement x_i et y_i le i^{ieme} bit de x et de y. Le résultat de $x \oplus y$ est le mot z tel que $z_i = x_i \oplus y_i$.

Les opérateurs C sont `&` (et), `|` (ou), `^` (xor) et `~` (not).
Attention : ne pas confondre les opérateurs logiques tel que `&&`, `||`, etc, avec les opérateurs de manipulation des mots binaires. En effet, là où `4&2` vaut 0, `4&&2` vaut 1.

Les opérations binaires peuvent être condensées. Ainsi `x=x|2` peut s'écrire `x|=2`, et `x=x^y` peut se noter `x^=y`. Le langage fournit également les opérateurs de décalage à droite `>>` ou à gauche `<<`.

1. Que fait le code suivant :

   ```
   n & (n-1)
   ```

2. Dans le code suivant c et n sont des entiers :

   ```
   for (c = 0; n != 0; n &= (n-1)) c++;
   ```

   Quelle valeur prend c en fonction des valeurs de n ?

3. On va étudier une méthode qui compte efficacement le nombre de 1 dans un mot de longueur 2^k (pour un $k \geq 0$), c'est à dire dans $\mathcal{O}(k)$ opérations, en supposant que 2^k est la taille d'un registre.
   Soit $l \leq k$ et n un mot de longueur 2^k. On appelle l-*bloc* les blocs de 2^l bits consécutifs dans n tel que ces blocs ne se chevauchent pas. (Par exemple, il y a huit 2-blocs de longueur 4 dans un mot de 32 bits.) Le l-*compte* de n est le mot de longueur 2^k tel que chacun de ses l-blocs contient le nombre de 1 du l-bloc correspondant dans n. Trivialement, tout mot égale son propre 0-compte. On cherche à produire le k-compte de n.
   Dans ce qui suit, on va supposer que $k = 5$, et du coup on travaille avec les registres de 32 bits. La méthode est pourtant facile à généraliser.

   (a) Trouvez une opération qui produit le 1-compte de n (en temps constant).

   (b) Généraliser et itérer cette opération pour calculer le 5-compte de n.

4. On travaille avec des registres de 64 bits. Soit $n = (stuvwxyz)_2$ un octet, avec s le bit le plus significatif et z le moins significatif. Que donne l'expression C suivante ?

   ```
   (n * 0x0202020202 & 0x010884422010) % 1023
   ```

4 Trouver le bit de poids le plus faible

Dans cette partie nous étudierons une méthode pour efficacement trouver la position du bit de 1 du poids le plus faible dans un mot binaire. Soit $x \neq 0$ un entier, du coup sa représentation contient au moins un bit de 1. Par exemple, si la représentation binaire de x est 10011000, alors le bit recherché est le 1 suivi par les trois zéros finaux. Dans ce cas, on note $\ell(x) = 3$. L'objectif est de trouver $\ell(x)$ pour un $x \neq 0$ donné. On présente cette méthode pour les mots de $2^3 = 8$ bits, mais elle peut être généralisée à 2^n bits pour n'importe quelle valeur de $n > 0$.

La première étape consiste à étudier les séquences dites de *De Bruijn*. Soit $\mathbb{B} := \{0, 1\}$ et $\mathbb{B}^n$ l'ensemble des mots binaires de longueur n. Une séquence de De Bruijn d'ordre n est un mot binaire qui contient tout élément de $\mathbb{B}^n$.

1. Donnez une borne inférieure triviale pour la longueur d'une séquence de De Bruijn d'ordre n.

Le graphe G_n (*graphe de De Bruijn d'ordre* n) permet de construire des séquences d'ordre n. Les sommets de G_n sont les éléments de $\mathbb{B}^n$. Les sommets $b_1b_2...b_n$ et $c_1c_2...c_n$ sont liés par des arrêtes orientées si et seulement si $b_2 = c_1, b_3 = c_2, ...b_n = c_{n-1}$. Autrement dit, un sommet est lié au suivant en supprimant son premier bit et en ajoutant un des bits possibles à la fin. La Figure 1 représente G_2.

2. Dessinez le graphe G_3.

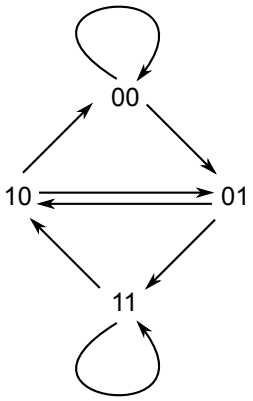

FIGURE 1 – Le graphe G_2.

Une séquence de De Bruijn d'ordre n correspond à un cycle hamiltonien dans G_n. Un cycle hamiltonien est un cycle passant uniquement une fois par tous les sommets du graphe avant de revenir au point de départ. On va considérer des cycles commençant et se terminant par le sommet 000. Par exemple, le seul cycle hamiltonien de la figure 1 page 3 est 00 $\rightarrow$ 01 $\rightarrow$ 11 $\rightarrow$ 10 $\rightarrow$ 00. Pour obtenir une séquence De Bruijn, il convient de commencer par noter le sommet initial (00) et les bits rajoutés à la fin de chaque sommet, ce qui donne, dans ce cas, 00110. On peut prouver qu'il existe un cycle hamiltonien dans chaque graphe de De Bruijn (voir Annexe).

3. Trouvez deux séquences de De Bruijn d'ordre 3 en utilisant le graphe G_3 dessiné précédemment.

Choisissez une séquence de De Bruijn d'ordre 3 que l'on va appeller s par la suite. Si s est composée de bits $b_0 b_1 b_2 \cdots$ et s' une séquence de longueur 3, on appelle *index* de s' la valeur i telle que $b_i b_{i+1} b_{i+2} = s'$. Par exemple, dans 0001110100 l'index de 000 est de 0 et l'index de 001 est de 1.

4. Complétez le tableau des indices ci-dessous pour votre séquence s.

chaîne	index dans s
000	0
001	
010	
011	
100	
101	
110	
111	

5. Considérons $0 \leq j < 8$. En utilisant la séquence s choisie précédemment, on considère l'expression suivante :
   $e(j) := ((s \ll j) \gg 7)\ \&\ 7$
   Ici, $\ll$ et $\gg$ désignent respectivement un décalage à gauche et un décalage à droite, et & est l'opérateur «et». Quelle est la relation entre $e(j)$ et j ? Est-ce qu'on peut récupérer j étant donné $e(j)$?
6. Quelle valeur donne $x\ \&\ (-x)$ où $-x$ est représenté par complément à deux ?
7. Proposez une implémentation de $l(x)$.

5 Virgule flottante

Rappel : en C, le type `float` représente des valeurs réelles selon le standard IEEE 754 dans la variante de 32 bit, avec 1 bit pour le signe, 8 pour l'exposant et 23 pour la mantisse. On considère le type suivant qui représente ces composants par trois entiers :

```
typedef struct { int signe; int exposant; int mantisse; } fc;
```

1. Écrivez une fonction C qui décompose un `float` dans ses trois composants. (autrement dit, le paramètre d'une telle fonction est un `float`, et elle renvoit un `fc`). Par exemple, la représentation en IEEE 754 de 2.5 est :

   ```
   0 . 1000 0000 . 010 0000 0000 0000 0000 0000
   ```

   Dans ce cas, la structure renvoyée contiendrait `sign` $= 0$, `exposant` $= 0x80 = 128$ et `mantisse` $= 0x200000 = 2097152$.

   Rappel : Pour ce faire, il convient de se servir de la conversion des types en C (*typecast*), p.ex., `(int)f`, où `f` est un `float`, interprète le contenu binaire de `f` comme un entier. Assurez-vous d'abord que `int` a la même taille sur votre machine !
2. Créez une fonction qui fait l'inverse, c'est à dire qui renvoie le `float` correspondant à une structure `fc` donnée.
3. Réalisez l'addition réelle sur la base de l'addition des entiers, en passant par les structures `fc`. Pour simplifier, on fera les restrictions suivantes : (i) les deux opérandes sont positifs ; (ii) on ne traite pas les cas spéciaux NaN/Inf etc.

   L'addition dans le type `fc` se fait en trois étapes :

   (a) Uniformiser les deux valeurs, c'est à dire si les deux exposants sont différents, on ajuste la mantisse d'une des deux selon la différence.

   (b) Faire la somme des deux mantisses, en tenant compte du bit "caché" représentant la 1.

   (c) Normaliser la mantisse pour quelle soit dans $[1, 2)$, tout en ajustant l'exposant du résultat.

Annexe : Existence des cycles hamiltoniens dans les graphes de De Bruijn

Pour prouver que tous les graphes de De Bruijn possèdent un cycle hamiltonien, on prouve les propriétés suivantes :

1. Pour tout $n \geq 1$, le graphe G_n admet un cycle eulérien.
2. Pour un graphe G, soit $A(G)$ son graphe adjoint. (Les sommets de $A(G)$ sont les arêtes des G, et (e, e') est une arête d'$A(G)$ si e et e' sont adjacentes.) On prouve que $G_{n+1} = A(G_n)$ pour tout $n \geq 1$.
3. Pour conclure, il suffit de considèrer que tout cycle eulérien dans un graphe G correspond à un cycle hamiltonien de $A(G)$.