DM3 : Programmation en C

À rendre le 8 Janvier

En tant que premier DM en C, ce DM devra être rendu écrit à la main sur copie.

Exercice 1. Programmation en C

Pour chacune des spécifications suivantes, proposer une implémentation en C.

1. Une fonction de prototype `bool est_bissextile(int annee)` qui détermine si une année est bissextile, c'est-à-dire si l'année vérifie l'une des propriétés suivantes
   — Elle est multiple de 4 mais pas de 100 ;
   — Elle est multiple de 400.
2. Une fonction de prototype `int clamp(int a, int b, int x)` de sorte à ce que `clamp(a, b, x)` calcule la fonction suivante :

$$f_{a,b}(x) = \begin{cases} b \text{ si } x > b, \\ a \text{ si } a > x, \\ x \text{ sinon.} \end{cases}$$

Correction :

1.
```c
bool est_bissextile(int annee){
    return (annee%400 == 0)||((annee%4==0)&&(annee%100!=0));
}
```

2.
```c
int clamp(int a, int b, int x){
    if (x>b){
        return b;
    }
    if (a>x){
        return a;
    }
    return x;
}
```

Exercice 2. Un programme en C

1. Proposer une fonction de prototype `bool est_parfait(int n)` qui renvoie si un entier n est parfait, c'est-à-dire si la somme des diviseurs de n qui sont différents de n est égale à n.
2. En déduire un programme en C qui demande un nombre en entrée, et qui affiche tous les nombres parfaits inférieurs à ce nombre.

On prendra soin de proposer un programme complet.

Correction :

1.
```c
bool est_parfait(int n){
int somme_diviseurs = 0;
for (int i = 1; i<n ; i++){
    if (n%i==0){
        somme_diviseurs += i;
    }
}
return somme_diviseurs == n;
}
```

2.
```c
#include <stdio.h>
#include <stdbool.h>

bool est_parfait(int n){
    int somme_diviseurs = 0;
    for (int i = 1; i<n ; i++){
        if (n%i==0){
            somme_diviseurs += i;
        }
    }
    return somme_diviseurs == n;
}

int main(){
    int max;
    scanf("%d",&max);
    for (int i = 0; i<max+1; i++){
        if (est_parfait(i)){
            printf("%d\n", i);
        }
    }
    return 0;
}
```

Exercice 3. Implémentation de l'addition à l'aide d'opérateurs logiques

Dans cet exercice, il n'est possible d'utiliser que les opérateurs logiques ($>>$, $<<$, &, |, ^...), mais pas les opérateurs arithmétiques ($+$, $-$, $*$ et $/$)

On cherche à implémenter l'ajout de deux nombres.

L'idée principale est la suivante : à chaque étape, on ajoute les deux nombres à ajouter sans prendre en compte les retenues, puis on calcule un nouveau nombre qui est composé des retenues que nous aurions du ajouter.

On répète ce calcul jusqu'à obtenir un nombre de retenue égal à 0.

Par exemple, pour calculer $19 + 21$, on cherche à ajouter les nombres suivants :

0	1	0	0	1	1

0	1	0	1	0	1

Sans compter les retenues, notre résultat serait :

0	0	0	1	1	0

Or, les retenues apparaissent à deux endroits dans le calcul, il faut donc encore ajouter :

1	0	0	0	1	0

Sans prendre en compte les retenues, on obtiendrait :

1	0	0	1	0	0

Mais il reste une retenue que nous devons ajouter :

0	0	0	1	0	0

Après, avoir ajouté cette retenue, on obtient donc

1	0	0	0	0	0

Auquel il faut rajouter une dernière retenue :

0	0	1	0	0	0

On obtient ainsi :

1	0	1	0	0	0

Ici, on s'arrête, car on n'a plus de retenue.
Ce qui correspond bien à 40, le résultat attendu.
Cela correspond donc à l'algorithme suivant :

Algorithme 1 : Addition logique

Entrée : a et b deux entiers
Sortie : La somme des entiers a et b
Tant Que $b \neq 0$ **Faire**
  $r \leftarrow$ retenues de l'addition de a et b
  $a \leftarrow$ addition sans retenue de a et b
  $b \leftarrow r$
Renvoyer a

Dans la suite, on raisonne sur le type unsigned et on suppose qu'il est représenté sur 32 bits.

1. Proposer une fonction de prototype unsigned ajout_sans_retenue(unsigned a, unsigned b) de sorte à ce que le résultat de l'ajout de

a_{31}	$\cdots$	a_0

et de

b_{31}	$\cdots$	b_0

soit

c_{31}	$\cdots$	c_0

où c_i est égal $a_i + b_i$ modulo 2.

2. Proposer une fonction de prototype `unsigned retenue(unsigned a, unsigned b)` de sorte à ce que le calcul de la retenue de

a_{31}	$\cdots$	a_0

et de

b_{31}	$\cdots$	b_0

soit

c_{31}	$\cdots$	c_0

où, pour $i > 0$, c_i est égal à 1 si et seulement si a_{i-1} et b_{i-1} sont égaux à 1 tous les deux.

3. En déduire une fonction `unsigned ajouter_logique(unsigned a, unsigned b)` qui ajoute a et b sans utiliser d'opérations arithémtiques.
4. Montrer que cet algorithme termine.

   On pourra s'intéresser au nombre de bits à droite du nombre qui ne peuvent pas être des retenues.
5. Quel est le nombre maximum d'étapes ? Dans quel cas cela arrive ?
6. Montrer que cet algorithme est correcte.
7. Est-ce que cet algorithme fonctionne pour l'ajout de nombres signés ?
8. Proposer une fonction `unsigned retirer_logique(unsigned a, unsigned b)` qui calcule $a - b$ de manière similaire.

Correction :

1.
```c
unsigned ajouter_sans_retenue(unsigned a, unsigned b){
    return a^b;
}
```

2.
```c
unsigned retenue(unsigned a, unsigned b){
    return (a&b)<<1;
}
```

3.
```c
unsigned ajouter_logique(unsigned a, unsigned b){
    unsigned r = 0;
    while(b!=0){
        r = retenue(a, b);
        a = ajouter_sans_retenue(a, b);
        b = r;
    }
    return a;
}
```

4. À l'étape k, au moins les k bits de poids faibles de la retenue sont égaux à 0. En effet, on décale d'une case vers la gauche à chaque étape, et on ne peut avoir de retenue que dans un endroit où il y avait une retenue juste à droite avant (ou à la première étape).

   Donc cet algorithme termine.

5. Grâce à la remarque de la question précédente, on a au plus 32 étapes, qui a lieu quand on ajoute $2^{32} - 1$ avec 1 par exemple.

6. On remarque qu'à chaque tours de boucle la somme $a + b$ ne change pas, et qu'elle donne le bon résultat à la fin quand $b = 0$, et que donc a a pour valeur la somme des entrées.

7. Cet algorithme fonctionne pour l'addition des nombres signés car l'addition arithmétique est compatible avec les nombres en complément à deux.

```c
unsigned retenue_moins(unsigned a, unsigned b){
    return ((~a)&&b)<<1;
}

unsigned retirer_logique(unsigned a, unsigned b){
    unsigned r = 0;
8.  while(b!=0){
        r = retenue_moins(a, b);
        a = ajouter_sans_retenue(a, b);
        b = r;
    }
    return a;
}
```