TP2 – Les fonctions

Projet de programmation M1

29 Septembre 2015

Exercice 1. [Échauffement]

1. Quelle est la taille de l'ensemble $\{0,1\}^n$? Combien peut-on représenter de nombres différents avec n chiffres en base 2 ? Quel est le plus petit ? Quel est le plus grand ?
2. La notation binaire n'est pas pratique pour représenter les nombres négatifs. On pourrait réserver un bit pour le signe, mais cela compliquerait la fonction addition. Les machines utilisent la représentation en complément à deux : sur n bits, on représente les entiers de 0 à $2^{n-1}-1$ par leur écriture binaire. Et les entiers k de -2^{n-1} à -1 par l'écriture binaire de 2^n+k.
   (a) Quelle est la représentation en complément à deux de 5 et de -6 sur 4 bits ?
   (b) Étant donné la représentation d'un nombre en complément à deux, comment connaître son signe ?
   (c) On additionne deux nombres représentés en complément à deux comme on additionnerait deux nombres binaires. Expliquez alors pourquoi sur la machine, `MAX_INT + 1 = MIN_INT` (où rappelons-le, `MAX_INT` = $2^{31}-1$, `MIN_INT` = -2^{31}).
   (d) Étant donné la représentation de p, comment calculer la représentation de $-p$? Est-ce que si on additionne les représentations comme deux nombres binaires, on obtient bien la représentation de la somme ?

Exercice 2. [Copie] Exécutez le programme suivant :

```c
#include <stdio.h>

void f(int y) {
  int x;
  y = 0;
  x = 0;
}

int main() {
  int x = 1, y=1, i=0;
  for(i=0;i<2;i++) {
    int x;
    x=5;
  }
  f(x);
  printf("%d\n", x);
  return 0;
}
```

1. Quelle est la valeur de x à la fin du programme ? Pourquoi ? Si vous trouvez cela choquant, appelez `f` sur `3*x`. Est-ce que ça a un sens ?
2. Remplacez `x` par `y` dans la fonction `main`. Qu'affiche ce programme ? Si vous trouvez cela choquant, dites-vous que c'est la même notion de variables libres/liées qu'en mathématiques.

Exercice 3. En C, on dispose d'une fonction `rand()` qui retourne un entier aléatoire compris entre 0 et une constante RAND_MAX (qui est en général égale à $2^{31}-1$, c'est-à-dire la plus grande valeur que peut contenir un `integer`) tiré selon une distribution qui se veut proche de la distribution uniforme. Il faut initialiser le générateur pseudo-aléatoire au début du programme avec une valeur initiale ; on utilise pour cela la fonction `srand` et on choisit en général comme valeur l'horloge de l'ordinateur. Toutes ces fonctions se trouvent dans la librairie `stdlib.h` qu'il faut penser à inclure dans vos programmes. Voici un exemple :

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   /* Intialise le generateur */
   srand((unsigned) time(NULL));
   printf("%d\n", rand());
   return(0);
}
```

Dans toute la suite, on fait l'hypothèse que rand() initialisé ainsi renvoie bien une valeur tirée uniformément entre 0 et RAND_MAX.

1. Écrire une fonction `int uniforme(int a)` qui renvoie un entier aléatoire tiré uniformément sur $[0, a[$ si $a > 0$ et $]a, 0]$ sinon.
2. Écrire une fonction `int simule(int a, int n)` qui simule `n` tirages uniformes sur $[0, a[$ avec votre fonction `uniforme` et renvoie le nombre de 0 obtenu ainsi.

Exercice 4.

1. Écrire une fonction `puissance(int a, int b)` qui renvoie a^b. Combien de multiplications doit faire votre programme ?
2. En remarquant que $a^{2b} = (a^2)^b$ et $a^{2b+1} = a \cdot a^{2b}$, implémentez une fonction `puissance_rapide(int a, int b)` qui renvoie a^b en beaucoup moins d'étapes. Si vous avez du mal, essayer d'avoir deux variables auxiliaires e et c initialisées à $e = b$ et $c = a$ et d'avoir une boucle `while(e>1)` telle qu'à chaque itération de la boucle, vous ayez toujours l'invariant $c^e = a^b$. Votre boucle doit distinguer le cas e pair du cas e impair.

Exercice 5. Le petit théorème de Fermat stipule que si p est premier alors $a^{p-1} \equiv 1 \mod p$ pour tout $0 < a < p$. Cela donne lieu à un test de primalité. Soit un entier n. On veut savoir s'il est premier. On teste alors pour un certain nombre de a choisi uniformément entre 1 et n si $a^{n-1} \equiv 1 \mod n$. Si on trouve un témoin a tel que $a^{n-1} \not\equiv 1 \mod n$, alors on en conclut que n n'est pas premier. Sinon, on conclut que n est probablement premier.

1. Créez une fonction `puissance_rapide_mod(int a,int b,int n)`, qui adapte `puissance_rapide` pour calculer $a^b \mod n$. Écrivez une fonction `fermat(int n, int k)` qui teste k témoins aléatoires. Si tous les témoins passent le test de Fermat, votre fonction renverra 1. Sinon, elle renverra un témoin qui prouve que n n'est pas premier.
2. Ce test souffre d'un problème majeur : certains nombres non-premiers vérifient la condition de Fermat. Ce sont les nombres de Carmichael. Le plus petit étant 561. Vérifiez avec un programme que 561 vérifie $a^{560} \equiv 1 \mod 561$ pour tout $a < 561$.

Le test de Rabin-Miller permet de s'affranchir de cette limite du test de Fermat et est souvent utilisé en pratique comme un test de primalité fiable. Le test de Rabin-Miller peut s'écrire ainsi : si p est premier alors pour tout $a < p$,

$$a^d \equiv 1 \mod p \text{ ou bien } \exists r \in \{0, \dots, s-1\}, a^{2^r d} \equiv -1 \mod p$$

où d et s sont tels que $p - 1 = 2^s d$ avec d impair. Si a ne vérifie pas la condition ci-dessus, alors on sait que p n'est pas premier et on dit que a est un témoin de Miller.

3. Écrivez une fonction `residu(int n)` qui étant donné n, renvoie d impair tel qu'il existe s tel que $2^s d = n$. (Il suffit pour cela de diviser n par 2 jusqu'à ce qu'il soit impair).
4. Écrivez une fonction `test_temoin(int n, int a)` qui teste si a est un témoin de Miller.
5. Écrivez une fonction `rabinmiller(int n, int k)` qui teste k témoins tirés uniformément. Si aucun n'est un témoin de Miller, votre fonction doit renvoyer 1. Sinon, elle renvoie un témoin de Miller.

Pour le 5 octore

Finissez d'implémenter le test de Rabin-Miller. Envoyez-moi votre code par mail avec comme sujet "[TPAlgo] TP2".