TP3 : fonctions, récursivité, complexité

Algorithmique et projet de programmation — M1 mathématiques

27 septembre 2016

On s'appliquera, dans tous les exercices, à estimer la complexité des algorithmes implémentés.

Le prototype d'une fonction C est la donnée de son type de retour, de son nom et du type de chacun de ses arguments :

```c
fun_t fun_name (arg1_t, arg2_t, ...);
/* or */
fun_t fun_name (arg1_t var1, arg2_t var2, ...);
```

Une bonne pratique de programmation est de déclarer tous les prototypes des fonctions présentes dans un fichier C en tête de celui-ci. Cela permet également d'utiliser les différentes fonctions avant même leurs déclarations. Par exemple :

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_NAME 50

char perfect_square(int);
void hello_name(char*);

int main (int argc, char** argv) {
  char str[MAX_NAME];
  printf("Enter your name: "); scanf("%s",str);
  hello_name(str);

  int n;
  printf("Enter a square: "); scanf("%d",&n);
  if (perfect_square(n)) { printf("Good job!\n") ; }
  else { printf("Not square...\n") ; }

  return EXIT_SUCCESS;
}

char perfect_square(int n) {
  int i;
  for(i = 0; i*i<n; i++);
  return i*i == n ? 1 : 0;
}

void hello_name (char* name) {
  printf("Hello, %s!\n", name);
}
```

1 Récursivité : premier pas

Exercice 1 (Échauffement)**.** Écrire une fonction **récursive** de prototype

```
int gcd (int a, int b);
```

qui calcule le plus grand diviseur commun des deux entiers a et b.

Exercice 2 (Inverse modulaire)**.** Écrire une fonction **récursive** de prototype

```
int mod_inverse (int a , int n);
```

qui retourne l'inverse d'un entier a modulo l'entier n (ou 0 si a n'est pas premier à n). Pour cela, on s'inspirera de l'algorithme d'Euclide étendu qui sert à trouver les coefficients de Bézout du couple $(a\,,n)$, c'est-à-dire des entiers u et v tels que $ua + vn = \gcd(a\,,n)$.

2 Exponentiation

Exercice 3. Écrire une fonction (la plus naïve possible) de prototype

```
int exp_int (int a, int e);
```

qui retourne la valeur de a^e.

Exercice 4. Observer que

$$a^e = \begin{cases} (a^k)^2 & \text{si } e = 2k \\ a(a^k)^2 & \text{si } e = 2k+1 \end{cases}.$$

En déduire une fonction (récursive) de prototype

```
int quick_exp (int a, int e);
```

qui retourne la valeur de a^e.

3 Tests de primalité

Exercice 5. S'inspirer du code écrit pour `quick_exp` pour écrire une fonction de prototype

```
int quick_exp_mod (int a, int e, int n);
```

qui retourne la valeur de $a^e \pmod{n}$. En particulier, la valeur retournée doit être dans l'intervalle entier $[0\,,n-1]$.

Rappelons rapidement le petit théorème de Fermat : si $p \in \mathbb{N}$ est premier, alors pour tout $a \in (\mathbb{Z}/p\mathbb{Z}) \setminus \{0\}$, on $a^{p-1} = 1$. La contraposée de ce théorème nous donne donc une manière d'être sûr qu'un entier n n'est pas premier : il suffit de trouver un nombre $0 < a < n$ tel que $a^{n-1} \neq 1 \pmod{n}$. On appelle cela le test de primalité de Fermat.

Exercice 6. Écrire une fonction

```
char fermat_test(int n, int k);
```

qui pratique k fois le test de Fermat avec un nombre $0 < a < n$ aléatoire et qui retourne `1` si le nombre n est composé et `0` sinon.

Ainsi, si `fermat_test(n,k)` renvoie `1`, on sait sans aucun doute que n est composé. À l'inverse, si `fermat_test(n,k)` renvoie `0`, on ne peut pas conclure que n est premier : peut-être est-ce un coup de malchance qu'on ait tiré aucun a tel que $a^{n-1} \neq 1 \pmod{n}$. On dira alors que le nombre est *probablement premier au sens de Fermat*.

Une idée pour augmenter s'assurer qu'un nombre probablement premier l'est réellement serait d'augmenter le nombre k de tirages. Malheureusement, même en testant tous les tirages possibles, on ne s'en sort pas : le test de Fermat n'est pas une caractérisation des nombres composés. C'est-à-dire qu'il existe des nombres n **composés** pour lesquels tout $a \in [1, n-1]$ premier à n satisfait à l'équation $a^{n-1} = 1 \pmod{n}$. De tels nombres sont appelés *nombres de Carmichael*. Ainsi, si vous utilisez la fonction `fermat_test` sur un nombre de Carmichael (par exemple 561), vous aurez beau augmenter le nombre k de tests, l'algorithme répondra à coup sûr `0`, vous laissant croire que le nombre est très probablement premier alors que ce n'est pas le cas !

Le test de primalité de Miller-Rabin raffine le test de Fermat. Il se base sur le fait que dans le corps $\mathbb{Z}/p\mathbb{Z}$ (pour p premier), les seules racines carrées de 1 sont 1 et -1. Ainsi, devant la conclusion du théorème de Fermat $a^{p-1} = 1 \pmod{p}$ pour un entier $0 < a < p$, on peut prendre des racines carrées successives jusqu'à ou bien avoir un exposant impair, ou bien tomber sur -1. Plus formellement, si $p \in \mathbb{N}$ est premier, on écrit (de façon unique) $p - 1 = 2^s d$ avec d impair, et alors pour tout entier $a \in [1, p-1]$,

$$a^d = 1 \pmod{p} \quad \text{ou} \quad \exists r \in [0, s-1],\ a^{2^r d} = -1 \pmod{p}$$

Bien sûr, cette propriété de tient plus nécéssairement si p n'est pas premier, et tout nombre a satisfaisant sa négation est appelé un témoin de Miller-Rabin pour p.

Exercice 7. Écrire une fonction de prototype

```
char miller_rabin_test(int n, int k)
```

qui effectue le test de Miller-Rabin k fois avec un nombre $0 < a < n$ tiré au hasard et retourne `1` si n est composé et `0` sinon.

L'avantage du test de Rabin-Miller est qu'il n'y a pas d'équivalent des nombres de Carmichael. Mieux, on a la proposition suivante : pour un nombre impair composé n, $3/4$ au moins des entiers $1 < a < n$ sont des témoins de Miller-Rabin pour n.

Exercice 8. En considérant qu'on tire les entiers de façon uniforme et indépendante dans la fonction `miller_rabin_test`, majorer la probabilité (en fonction de k) que la fonction `miller_rabin_test(n,k)` renvoie `0` pour un nombre impair composé n.

4 Représentation binaire

La décomposition d'un entier $n \in \mathbb{N}$ en base $b \in \mathbb{N} \setminus 0$ is the unique finite sequence $(n_0, \ldots, n_k)$ such that

$$n = \sum_{i=0}^{k} n_i b^i \quad \text{et} \quad \forall i \in \mathbb{N},\ 0 \leq n_i < b.$$

Cette décomposition s'obtient facilement par division euclidienne successive de n par b.

Le but va être de coder des fonctions relatives au changement de bases. Pour cela, il faut pouvoir implémenter les séquences finies en C : on utilise le concept de *tableau*. Un tableau t de longueur ℓ contenant des éléments de type `typ` est plus ou moins une section de ℓ cases mémoires adjacentes chacune de la taille nécessaire pour stocker une variable de type `typ`. La variable t est alors un *pointeur* sur la première case de cette section. Pour accéder aux autres cases, il suffit de "sauter" de `sizeof(typ)` en `sizeof(typ)` à partir de cette première case[1]. Heureusement, le langage C met en place des outils pour faire tout cela assez facilement. Un tableau `t` d'éléments de type `typ` et de longueur ℓ est simplement déclaré par `typ t[`ℓ`]`, où ℓ doit être un nombre effectif (et non pas une variable de type `int` ou autre). On accède à la case numéro i avec la syntaxe `t[i]`. Attention, les cases d'un tableau sont numérotées à partir de 0 en C, ce qui veut dire que i doit être compris entre 0 et $\ell - 1$. Remarquer qu'un tableau est modifiable *en place*, c'est-à-dire qu'une fonction peut modifier ses valeurs directement.

Considérons l'exemple suivant :

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

void table(int,int*);

int main (int argc, char** argv) {

  int tab7[10];
  table(7,tab7);

  printf("Table of 7:\n");
  for(int i = 0; i < 10; i++) {
    printf("7*%d = %d\n", i+1, tab7[i]);
  }

  table(9,tab7);
  printf("\nStill table of 7?\n");
  for(int i = 0; i < 10; i++) {
    printf("7*%d ?= %d\n", i+1, tab7[i]);
  }


  return EXIT_SUCCESS;
}

void table(int n, int* t) {
  for(int i = 0; i < 10; i++) {
    t[i] = n*(i+1);
  }
}
```

1. Tout ceci peut sembler bien abstrait pour le moment, vous deviendrez plus à l'aise avec ces notions d'ici quelques semaines. Pour le moment, laissez-vous guider par les exemples et appliquez les "recettes" données en TPs.

Exercice 9. Écrire une fonction de prototype

```
void dec2bin(unsigned int n, char* bin);
```

qui écrit dans le tableau `bin` la décomposition binaire (i.e. en base $b = 2$) de l'entier n. Tous les tableaux de votre programme auront une taille arbitraire fixée à l'avance de 32 cases (justifiez !).

Exercice 10. Écrire la fonction inverse

```
unsigned int bin2dec(char* bin);
```

qui renvoie l'entier dont `bin` est la décomposition binaire.

Exercice 11. Pour rendre les tests des deux fonctions précédentes plus faciles à lire, on peut coder une fonction

```
void pp_bin (char* bin);
```

qui affiche à l'écran l'écriture usuelle en binaire du nombre dont la décomposition est `bin`. Par exemple, l'éxécution de :

```
char bin5[32];
dec2bin(5, bin5);
pp_bin(bin5);
```

doit afficher à l'écran `101`.

5 À rendre pour le 18 octobre 2016

Exercice 12 (Représentation binaires des relatifs)**.** La définition de la décomposition binaire mise en place ci-dessus n'est valable que pour les positifs. Pour coder les entiers négatifs, il suffit de donner leur signe et la décomposition binaire de leur valeur absolue. Il faut donc trouver un artifice qui permette d'incorporer le signe à la décomposition binaire : c'est ce que fait la méthode qui suit et qu'on appelle *le complément à 2*.

Avec des tableaux de 0 et 1 de n cases, on peut représenter les 2^n entiers relatifs contenu dans $[-2^{n-1}, 2^{n-1} - 1]$ où l'on décide de représenter les entiers $k \in [0, 2^{n-1} - 1]$ par la représentation binaire de k, et les entiers $k \in [-2^{n-1}, -1]$ par la représentation binaire de $2^n + k$. La réprésentation d'un nobre k ainsi obtenu est appelée la représentation en complément à 2 de k.

Pour l'exercice, on fixe le nombre de bits $n = 32$.

(1) Écrire une fonction `char is_pos(char* bin)` qui renvoie 1 si le nombre représenté par `bin` en complément à 2 est positif ou nul et 0 s'il est strictement négatif.

(2) Écrire une fonction `void opposite(char* bin, char* opp)` qui écrit dans `opp` la représentation en complément à 2 de $-k$, où k est le nombre dont `bin` est la représentation en complément à 2.

Exercice 13. Le langage `C` permet en fait de manipuler les entiers dans leur représentation binaire **directement** sans passer par une conversion explicite sous forme de tableau. On dispose de certaines opérations *bits à bits* :

`a & b`	bitwise AND	`5 & 3` donne `1`
`a | b`	bitwise OR	`5 | 3` donne `7`
`a ^ b`	bitwise XOR	`5 ^ 3` donne `6`
`~a`	bitwise NOT	`~22` donne `9`
`a >> n`	right shift by n	`22 >> 2` donne `5`
`a << n`	left shift by n	`22 << 1` donne `44`

Pour comprendre ce qu'il se passe, il suffit d'écrire chaque entier sous forme binaire et d'effectuer les opérations bit par bit : par exemple l'écriture binaire de 5 est `101` et celle de 3 est `11`, ainsi l'entier `5 & 3` a pour écriture binaire `1`. Pensez-y comme suit :

```
  101 (binary for 5)     101 (binary for 5)     101 (binary for 5)
& 011 (binary for 3)   | 011 (binary for 3)   ^ 011 (binary for 3)
  ---                    ---                    ---
  001 (binary for 1)     111 (binary for 7)     110 (binary for 6)

~10110 (binary for 22)     10110 >> 2             10110 << 1
 -----                     ----------             ----------
 01001 (binary for 9)      101 (binary for 5)     101100 (binary for 44)
```

(1) Persuadez-vous (sur papier) que les opérations `%2`, `/2` et `*2` peuvent s'écrire avec les opérations bits à bits.

(2) En déduire un algorithme calculant le pgcd de deux entiers positifs en exploitant l'observation suivante :

$$\gcd(a,b) = \begin{cases} 2\gcd(a/2, b/2) & \text{si } a \text{ et } b \text{ pairs} \\ \gcd(a/2, b) & \text{si } a \text{ pair et } b \text{ impair} \\ \gcd(a-b, b) & \text{si } a \text{ et } b \text{ impairs et } a \geq b \end{cases}$$