Développement C **L2 CUPGE 2018–2019**

Travaux pratiques Tableaux

I Exercice : *Crible d'Érathostène Simple*

Pré-requis : Thème Entrées/Sorties, exercice 6.

L'algorithme du crible d'Érathostène est utilisé pour calculer en une fois tous les nombres premiers jusqu'à une certaine limite. On veut calculer un tableau `t` d'entiers tel que `t[i] = 1` si, et seulement si, `i` est un nombre premier. Tout le travail se fait dans le fichier `exercice0.c` à partir du programme de base.

(a) On commence par fixer la taille maximale qu'on accepte pour tester des nombres premiers. Ajouter en haut du fichier, après l'inclusion de `stdio.h` :

```
#define TAILLE 100000
```

(b) On va écrire une fonction `crible` calculant le crible. Pour celà :
— il faut déclarer dans la fonction `main` un tableau d'entiers t de `TAILLE` cases ;
— appeler la fonction `crible` avec l'adresse de ce tableau comme paramètre.
Commencez par écrire une fonction `crible` qui ne fait rien, mais est appelée avec les bons arguments, et vérifiez que le programme compile correctement.

```
#include <stdio.h>

#define TAILLE 100000

void crible ( int * t )
{
}


int
main ( int argc , char * argv[] )
{
    int n ;
    int t[TAILLE] ;
    if ( crible ( t , TAILLE ) != 0 )
    {
        printf ( "Il␣y␣a␣eu␣une␣erreur␣lors␣du␣calcul.\n" ) ;
        return 1 ;
    }
    while ( 1 )
    {
        printf ( "Donnez␣un␣entier␣strictement␣inférieur␣à␣%d :\n" , TAILLE )
        scanf ( "%d" , &n ) ;
        if ( n < 0 )
            break ;
        if ( n >= TAILLE )
```

```
            continue ;
        if ( t[n] == 1 )
            printf ( "%d est un nombre premier.\n" , n ) ;
        else
            printf ( "%d n'est pas un nombre premier.\n" , n ) ;
    }
    return 0 ;
}
```

(c) Lorsqu'on programme en C, il est recommandé de suivre la recommandation suivante :
— si une fonction n'a pas eu d'erreur, elle renvoie l'entier 0 ;
— sinon, elle renvoie un entier (le code d'erreur) qui indique quelle a été l'erreur.
Modifiez le programme pour qu'en cas d'erreur de la fonction `crible`, la fonction `main` s'arrête immédiatement en renvoyant un entier > 0.

```
    if ( crible ( t ) != 0 )
    {
        printf ( "Il y a eu une erreur lors du calcul.\n" ) ;
        return 1 ;
    }
```

(d) Dans la fonction `crible`, commencez par mettre les deux premières cases à 0 (car 0 et 1 ne sont pas premiers) et toutes les autres à 1. Compilez et vérifiez que le programme s'exécute correctement, même s'il ne fait rien pour l'instant.

(e) L'algorithme du crible d'Érathostène contient 2 boucles imbriquées :
— La première parcourt toutes les cases du tableau. Pour chaque case i :
  — Si $t[i] = 0$, passer à la case suivante ;
  — Si $t[i] = 1$, alors i est un nombre premier, et :
    — Pour tous les $j \geq 2$ tels que $i * j \leq$ `TAILLE`, mettre la case $i * j$ à 0 (car $i * j$ est un multiple de i).

(f) Modifiez la boucle externe pour ne pas parcourir toutes les cases jusqu'à `TAILLE` et vous arrêter avant.

```
int crible ( int * t )
{
    int i , j ;
    if ( TAILLE < 2 )
        return 1 ;
    t[0] = t[1] = 0 ;
    for ( i = 2 ; i < TAILLE ; i++ )
        t[i] = 1 ;
    for ( i = 2 ; i * i < TAILLE ; i++ )
        if ( t[i] == 1 )
            for ( j = 2 ; i * j < n ; j++ )
                t[i*j] = 0
    return 0 ;
}
```

(g) Dans la fonction `main`, faire une boucle infinie (dont la condition est toujours vraie) demandant à l'utilisateur un entier et faisant :

— Si l'entier est strictement négatif, on sort de la boucle avec `break` (c'est le seul moyen de sortir de cette boucle infinie) ;
— Si l'entier est plus grand ou égal à la taille, on ne peut pas faire de calcul, donc on passe à l'itération suivante avec `continue` ;
— Sinon, il faut afficher si l'entier i qui a été lu est premier en regardant la valeur de $t[i]$.

```
int
main ( int argc , char * argv[] )
{
    int n ;
    int t[TAILLE] ;
    if ( crible ( t , TAILLE ) != 0 )
    {
        printf ( "Il␣y␣a␣eu␣une␣erreur␣lors␣du␣calcul.\n" ) ;
        return 1 ;
    }
    while ( 1 )
    {
        printf ( "Donnez␣un␣entier␣strictement␣inférieur␣à␣%d :\n" , TAILLE ) ;
        scanf ( "%d" , &n ) ;
        if ( n < 0 )
            break ;
        if ( n >= TAILLE )
            continue ;
        if ( t[n] == 1 )
            printf ( "%d␣est␣un␣nombre␣premier.\n" , n ) ;
        else
            printf ( "%d␣n'est␣pas␣un␣nombre␣premier.\n" , n ) ;
    }
    return 0 ;
}
```

II Exercice : *Crible d'Érathostène utile*

Commencez par copier dans `exercice0.c` le programme de l'exercice précédent.

(a) Modifiez la fonction `crible` pour que la case $t[i]$ contienne :
— i si i est un nombre premier ;
— j qui est le plus grand diviseur premier de i sinon.
Pour $i = 1$, on prend $t[1] = 1$.

```
int crible ( int * t )
{
    int i , j ;
    t[0] = 0 ;
    t[1] = 1 ;
    for ( i = 2 ; i < TAILLE ; i++ )
        t[i] = i ;
    for ( i = 2 ; i * i < TAILLE ; i++ )
        if ( t[i] == i )
```

```
            for ( j = 2 ; i * j < TAILLE ; j++ )
                t[i*j] = i ;
    return 0 ;
}
```

(b) Écrivez une fonction qui prend en entrée le crible et un entier, et affiche la décomposition en facteurs premiers de cet entier.

```
int
decomposition ( int * t , int i )
{
    int j ;
    if ( ( i < 1 ) || ( i >= TAILLE ) )
    {
        printf ( "Pas de décomposition possible pour %d.\n" , i ) ;
        return 1 ;
    }
    for ( j = i ; t[j] != j ; j = j / t[j] )
        printf ( "%d " , t[j] ) ;
    printf ( "%d\n" , t[j] ) ;
    return 0 ;
}
```

(c) Modifiez la fonction `main` pour qu'elle affiche la décomposition en facteurs premiers des entiers que donne l'utilisateur.

```
int
main ( int argc , char * argv[] )
{
    int n ;
    int t[TAILLE] ;
    if ( crible ( t ) != 0 )
    {
        printf ( "Il y a eu une erreur lors du calcul.\n" ) ;
        return 1 ;
    }
    while ( 1 )
    {
        printf ( "Donnez un entier strictement inférieur à %d :\n" , TAILLE ) ;
        scanf ( "%d" , & n ) ;
        if ( n < 0 )
            break ;
        if ( n >= TAILLE )
            continue ;
        printf ( "La décomposition de %d en facteurs premiers est :\n\t" , n ) ;
        decomposition ( t , n ) ;
    }
    return 0 ;
}
```

III Exercice : *Recherche de majorité*

Le but de cet exercice est de chercher dans un tableau d'entiers aléatoires si une des valeurs apparaît dans plus de la moitié des cases.

Obtenir un entier aléatoire. Pour obtenir un entier suffisament aléatoire, on va utiliser un *générateur de nombres pseudo-aléatoires*. Ce générateur doit être *initialisé* par un entier différent à chaque lancement du programme, sinon ce seront toujours les mêmes nombres qui seront choisis. Pour celà, on initialise le GNPA avec l'heure actuelle :

```
srand ( time ( NULL ) ) ;
```

Il est nécessaire d'inclure les bibliothèques `time.h` et `stdlib.h`. Ensuite, chaque fois qu'on a besoin d'un entier aléatoire, on appelle la fonction `rand`.

On fixe une taille de tableau à 10 éléments.

(a) Écrire un programme qui demande à l'utilisateur un entier positif n, et initialise un tableau `t` avec des entiers entre 0 et $n-1$.

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

#define TAILLE 10

void initialise ( int * t , int modulo )
{
    int i ;
    for ( i = 0 ; i < TAILLE ; i++ )
        t[i] = rand ( ) % modulo ;
}
void lit_entier_positif ( int * n )
{
    int lu , c ;
    do
    {
        printf ( "Donnez un entier positif :\n" ) ;
        lu = scanf ( "%d" , n ) ;
        do
        {
            c = getchar ( ) ;
        } while ( c != '\n') ;
    } while ( ( lu != 1 ) || ( *n <= 0 ) ) ;

}
int
main ( int argc , char * argv[] )
{
    int n ;
    int t[TAILLE] ;
    srand ( time ( NULL ) ) ;
    printf ( "Choisissez le nombre de valeurs différentes possibles :\n" ) ;
```

```
    lit_entier_positif ( & n ) ;
    initialise ( t , n ) ;
    return 0 ;
}
```

La recherche d'un candidat majoritaire se fait en 2 phases :

1. d'abord, on parcourt une fois toutes les cases du tableau pour trouver un candidat possible. Pour celà :
   — si la majorité actuelle est de 0, l'élément courant devient candidat avec une majorité de 1 ;
   — si la majorité actuelle est strictement plus grande que 0 :
      — si l'élément courant est égal au candidat courant, on augmente la majorité de 1 ;
      — sinon, on diminue la majorité de 1

   Cet algorithme garantit que **s'il existe un élément majoritaire**, alors le candidat est cet élément ;
2. Ensuite, on parcourt le tableau une deuxième fois, et on compte le nombre de fois où le candidat apparaît.
   — Si ce nombre est strictement supérieur à `TAILLE / 2` alors on a trouvé l'élément majoritaire ;
   — Sinon, aucune valeur n'est majoritaire.

(b) Écrire la fonction `trouve_candidat` qui trouve le candidat qui peut être majoritaire :

```
void trouve_candidat ( int * t , int * candidat ) ;
```

avec `t` l'adresse de la première case d'un tableau de `TAILLE` entiers, et `candidat` l'adresse de la variable dans laquelle il faut mettre la valeur trouvée.

```
void trouve_candidat ( int * t , int * candidat )
{
    int i ;
    int majorite ;
    for ( i = 0 , majorite = 0 ; i < TAILLE ; i++ )
        if ( majorite == 0 )
        {
            majorite = 1 ;
            *candidat = t[i] ;
        }
        else
        {
            if ( t[i] == *candidat )
                majorite += 1 ;
            else
                majorite -= 1 ;
        }
}
```

(c) Écrire la fonction `verifie_candidat` qui vérifie que le candidat trouvé est majoritaire.

```
int verifie_candidat ( int * t , int candidat ) ;

int verifie_candidat ( int * t , int candidat )
{
```

```
    int i ;
    int occurrences ;
    for ( i = 0 , occurrences = 0 ; i < TAILLE ; i++ )
        occurrences += ( t[i] == candidat ) ;
    return ( occurrences > ( TAILLE / 2 ) ) ;
}
```

(d) Afin de vérifier le résultat, écrire une fonction qui affiche le contenu du tableau d'entiers.

```
int
affiche_tableau ( int * t )
{
    int i ;
    for ( i = 0 ; i < TAILLE ; i++ )
        printf ( "%d " , t[i] ) ;
    printf ( "\n" ) ;
    return 0 ;
}
```

(e) Complétez la fonction `main` et vérifiez que votre programme marche. Il est conseillé de prendre de choisir $n = 2$ ou $n = 3$ pour avoir de bonnes chances d'avoir un élément majoritaire.

```
int
main ( int argc , char * argv[] )
{
    int n ;
    int t[TAILLE] ;
    int candidat ;
    srand ( time ( NULL ) ) ;
    printf ( "Choisissez le nombre de valeurs différentes possibles :\n" ) ;
    lit_entier_positif ( & n ) ;
    initialise ( t , n ) ;
    affiche_tableau( t ) ;
    trouve_candidat ( t , & candidat ) ;
    if ( verifie_candidat ( t , candidat ) )
    {
        printf ( "L'élément %d est majoritaire.\n" , candidat ) ;
    }
    else
    {
        printf ( "Il n'y a pas d'éléments majoritaires dans le tableau.\n") ;
    }
    return 0 ;
}
```

IV Exercice : *Tri lent*

V Exercice : *Tri rapide*