Développement C **L2 CUPGE 2019–2020**

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 `exercice1.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.

Solution

```
#include <stdio.h>

#define TAILLE 100000

void crible ( int * t )
{
}


int
main ( int argc , char * argv[] )
{
    int n ;
    int t[TAILLE] ;
    crible ( t , TAILLE ) ;

    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.

Solution

```
    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.

Solution

```
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 $t[i]$.

Solution

```
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 `exercice2.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 $t[1] = 1$.

Solution

```
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.

Solution

```
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.

Solution

```
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 $n-1$.

Solution

```c
#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.

Solution

```
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 ) ;
```

Solution

```
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.

Solution

```
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.

Solution

```
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 : *Demi-additionneur et additionneur*

On va voir dans cet exercice comment additionner deux entiers ayant un nombre arbitraire de chiffres. Le principe est simple :

1. Chaque entier en entrée est codé par un tableau de chiffres ;
2. On calcule la solution chiffre par chiffre, en n'oubliant pas la retenue.

Pour le 1. on pourrait choisir des chiffres dans n'importe quelle base (au fond, c'est l'ordinateur qui fait les additions), mais pour avoir des exemples faciles à construire, on va utiliser la base 256, donc :

— Chaque chiffre est un *unsigned char* entre 0 et 255 ;
— Un entier est un tableau de *unsigned char*.

Pour les exemples, on va fixer la taille de ces tableaux à 10, mais n'importe quelle valeur est

possible. Pour pouvoir parcourir les tableaux à partir de l'indice 0, on suppose en plus que les chiffres sont écrits dans l'ordre inversé (le chiffre des unités dans la case 0, puis celui des "dizaines", etc.)

(a) Dans la fonction *main*, définissez 3 tableaux de type *unsigned char*, et initialisez les 2 premiers. Vérifiez que le code est correct en compilant et en exécutant le programme (il ne fait rien).

Initialisation :

```
unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
    t3[10] ;
```

Solution

```
#include <stdio.h>

int
main ( int argc , unsigned char * argv[] )
{
  unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
      t3[10] ;
  return 0 ;
}
```

(b) Écrire une fonction qui prend en entrée l'adresse d'un tableau d'*unsigned char* et son nombre de cases, et qui affiche le contenu du tableau. Modifiez la fonction *main* pour afficher les 3 tableaux.

Solution

```c
#include <stdio.h>

void print_tableau_c ( unsigned char * t , int n )
{
  int i ;
  printf ( "[ %3d " , t[0] ) ;
  for ( i = 1 ; i < n ; i ++ )
    printf ( ", %3d " , t[i] ) ;
  printf ( "]\n" ) ;
}

int
main ( int argc , unsigned char * argv[] )
{
```

```
  unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
      t3[10] ;
      print_tableau_c ( t1 , 3 ) ;
      print_tableau_c ( t2 , 3 ) ;
      print_tableau_c ( t3 , 3 ) ;
  return 0 ;
}
```

(c) Écrire une fonction *additionneur* qui prend en entrée :

— l'adresse de deux *unsigned char*, `resultat` et `retenue`, à laquelle elle mettra les valeurs de la somme et de la retenue de la somme suivante ;

— 2 *unsigned char* dont elle doit faire la somme avec la valeur, lors de l'appel de la fonction, contenue à l'adresse `retenue`.

Cette fonction ne renvoie pas de résultat.

Solution

```
void additionneur (
        unsigned char * retenue ,
        unsigned char * resultat ,
        unsigned char c1 ,
        unsigned char c2 )
{
  int tmp ;
  tmp = c1 + c2 + * retenue ;
  *retenue = tmp / 256 ;
  *resultat = tmp % 256 ;
}
```

(d) Écrire une fonction *addition_tableau* qui prend en entrée les adresses de 3 tableaux d'*unsigned char* et leur nombre de cases, et qui écrit dans le premier le résultat de l'addition des 2 autres. Cette fonction renvoie 1 si la retenue finale est 1 (c'est un cas d'erreur), et 0 sinon. Appliquez cette fonction sur les tableaux de la fonction *main*, et affichez le résultat.

Solution

```
int addition_tableau (
unsigned char * res ,
unsigned char * t1 ,
unsigned char * t2 ,
int n )
{
  int i ;
  unsigned char retenue = 0 ;
  for ( i = 0 ; i < n ; i++ )
```

```
    additionneur ( & retenue , res + i , t1[i], t2[i] ) ;
  return retenue ;
}
```

(e) **Attention, ce qui suit peut être perturbant !**
De la même manière que précédement, créez et initialisez 3 tableaux d'entiers dans la fonction main, écrivez une fonction d'affichage des tableaux d'entiers, et *appelez la fonction addition_tableau* sur ces tableaux (attention au nombre de cases, cette fonction a besoin du nombre de cases de type *unsigned char*. Affichez le résultat. Que remarquez-vous ? Est-ce normal ?

Solution

Pour la solution complète, on utilise des demi-additionneurs, qui sont plus proches des circuits utilisés dans les processeurs (notez le *ou logique* pour la retenue finale). De plus, on teste que toutes adresses sont possibles (la seule réellement impossible étant l'adresse `NULL`)

```
#include <stdio.h>


int demi_additionneur (
        unsigned char * retenue ,
        unsigned char * resultat ,
        unsigned char c1 ,
        unsigned char c2 )
{
  int tmp ;
  if ( retenue == NULL )
    return 1 ;
  if ( resultat == NULL )
    return 1 ;
  * resultat = ( c1 + c2 ) % 256 ;
  * retenue = ( c1 + c2 ) / 256 ;
  return 0 ;
}

int additionneur (
        unsigned char * retenue ,
        unsigned char * resultat ,
        unsigned char c1 ,
        unsigned char c2 )
{
  unsigned char retenue1 , retenue2 ;
  if ( retenue == NULL )
    return 1 ;
  if ( resultat == NULL )
    return 1 ;
  if ( demi_additionneur ( & retenue1 , resultat , c1 , c2 ) )
    return 1 ;
  if ( demi_additionneur ( & retenue2 , resultat , * resultat , * retenue )
```

```
    return 1 ;
  * retenue = retenue1 || retenue2 ;
  return 0 ;
}



int addition_c (
unsigned char * res ,
unsigned char * t1 ,
unsigned char * t2 ,
int n )
{
  int i ;
  unsigned char retenue = 0 ;
  for ( i = 0 ; i < n ; i++ )
    if ( additionneur ( & retenue , res + i , t1[i], t2[i] ) )
      return 1 ;
  if ( retenue )
    return 2 ;
  return 0 ;
}

void print_tableau_c ( unsigned char * t , int n )
{
  int i ;
  printf ( "[ %3d " , t[0] ) ;
  for ( i = 1 ; i < n ; i ++ )
    printf ( ", %3d " , t[i] ) ;
  printf ( "]\n" ) ;
}
void print_tableau_i ( unsigned int * t , int n )
{
  int i ;
  printf ( "[ %3d " , t[0] ) ;
  for ( i = 1 ; i < n ; i ++ )
    printf ( ", %3d " , t[i] ) ;
  printf ( "]\n" ) ;
}

int calcule ( char * t3 , char * t1 , char * t2 , int n )
{
  int i ;
  print_tableau_c ( t1 , n ) ;
  print_tableau_c ( t2 , n ) ;
  switch ( addition_c ( t3 , t1 , t2 , n ) )
    {
    case 1:
      printf ( "Erreur lors de l'addition.\n" ) ;
```

```
      return 1 ;
    case 2 :
      printf ( "Il reste une retenue (dépassement).\n" ) ;
      return 2 ;
    default :
      print_tableau_c ( t1 , n ) ;
      print_tableau_c ( t2 , n ) ;
      print_tableau_c ( t3 , n ) ;
    }
}

int
main ( int argc , unsigned char * argv[] )
{
  unsigned int
    i1[3] = { 12345 , 12345 , 0 } ,
    i2[3] = { 66666 , 44444 , 0 } ,
      i3[3] ,
i ;
  unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
      t3[10] ;
      calcule ( t3 , t1 , t2 , 10 ) ;
      calcule ( ( unsigned char * ) i3 , ( unsigned char * ) i1 , ( unsigned
      print_tableau_i ( i1 , 3 ) ;
      print_tableau_i ( i2 , 3 ) ;
      print_tableau_i ( i3 , 3 ) ;
  return 0 ;
}
```

Le résultat attendu est que l'addition est correcte pour les tableaux d'entiers. Quand on y pense ; c'est très perturbant, car normalement, dans chaque case d'entier, c'est comme-ci on faisait l'addition de la gauche vers la droite (on avait inversé l'ordre des chiffres dans le codage en tableau).

La raison est que les processeurs Intel ont évolué à partir d'ordinateurs (les processeurs 8086) qui ne faisaient d'additions que *char* par *char*. Par conservatisme, même maintenant, les entiers continuent d'être stockés dans l'ordre inverse, appelé en anglais *Big Endian* : le dernier char dans la représentation est celui qui contient les chiffres les plus importants. À l'inverse, les tablettes et téléphones portables, qui utilisent des processeurs de type ARM, stockent les entiers dans l'ordre naturel (*Little Endian*, les chiffres les moins importants sont à la fin de la représentation).

V Exercice : *Tri lent*

Trier un tableau signifie ordonner ses éléments du plus petit au plus grand. On va voir dans cet exercice et les suivants plusieurs algorithmes de tri, en commençant par un algorithme simple : On trie un tableau de n entiers en :

— Trouvant l'indice $0 \leq i_{\max} < n$ tel que la valeur $t[i_{\max}]$ soit maximale dans le tableau ;

— On échange cette valeur avec celle contenue dans la case d'indice $n-1$;
— On trie les $n-1$ valeurs restantes.

(a) Écrivez un programme initial contenant :
— une fonction qui affiche un tableau d'entiers (*cf.* exercice précédent) ;
— une fonction *tri* qui ne fait rien pour l'instant, mais qui triera un tableau d'entiers de n cases
— Dans la fonction *main* :
  — un tableau d'entiers initialisé (et avec des valeurs dans le désordre) ;
  — un appel à la fonction d'affichage ;
  — un appel à la fonction de tri ;
  — un second appel à la fonction d'affichage.

Solution

```
#include <stdio.h>

void affichage ( int * t , int n )
{
  int i ;
  printf ( "[ %d " , t[0] ) ;
  for ( i = 1 ; i < n ; i++ )
    printf ( ", %d " , t[i] ) ;
  printf ( "]\n'' ) ;
}

void tri ( int * t , int n )
{
}

int main ( int argc , char * argv [] )
{
  int t[10] = {5,8,1,4,9,2,2,3,6,7} ;
  affiche ( t , 10 ) ;
  tri ( t , 10 ) ;
  affiche ( t , 10 ) ;
  return 0 ;
}
  
```

(b) Écrivez une fonction *echange* qui échange les valeurs de deux cases de type `int` dont l'adresse est donnée en paramètre.

Solution

```
void echange ( int * a , int * b )
{
  int c ;
```

```
  c = *a ;
  *a = *b ;
  *b = c ;
}
```

(c) Écrivez une fonction qui calcule l'indice d'un élément maximal d'un tableau de n entiers.

Solution

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

(d) Complétez la fonction *tri* pour qu'elle trie effectivement un tableau de n entiers.

Solution

```
void tri ( int * t , int n )
{
  int i , i_max ;
  for ( i = n - 1 ; i > 0 ; i-- )
  {
    index_max ( & i_max , t , i ) ;
    echange ( t + i , t + i_max ) ;
  }
}
```

(e) Écrivez une deuxième version qui calcule le minimum du tableau, et l'échange avec le premier élément.

Solution

```
void indice_min ( int * res , int * t , int n )
{
  int i ;
  *res = 0 ;
  for ( i = 1 ; i < n ; i++ )
    if ( t[*res] > t[i] )
      *res = i ;
}
```

```c
void tri2 ( int * t , int n )
{
  int i_min ;
  for (  ; n > 0 ; t++ , n-- )
  {
    index_min ( & i_min , t , n ) ;
    echange ( t , t + i_min ) ;
  }
}
```

VI Exercice : *Tri rapide*

Le tri rapide est un tri en plusieurs étapes. Sur un tableau t de n entiers :

1. une première fonction `choix_pivot` choisit un indice pivot p et l'échange avec l'élément d'indice 0 ;
2. une fonction, qu'on appellera `drapeau_hollandais` parcourt le tableau pour que, par échanges successifs, tous les éléments plus petits que le pivot soient placés avant dans le tableau, et tous les éléments plus grand soient placés après ;
3. La procédure de tri est ensuite appelée récursivement sur les zones bleu (éléments strictement plus petits que le pivot) et rouge (éléments strictement plus grand que le pivot.

(a) Comme dans l'exercice précédent, initialisez et affichez un tableau, et écrivez une fonction de tri qui ne fait rien.

Solution

```c
#include <stdio.h>
// on inclut stdlib pour utiliser la fonction rand
#include <stdlib.h>
#define TAILLE 100000
#define TAILLE_AFFICHAGE 10

/*
  On commence par une fonction de tri qui ne fait rien.
  Les fonctions des questions suivantes sont à écrire au dessus
  de cette fonction, et on changera bien sûr la définition de la
  fonction tri pour qu'elle fasse quelque chose.
*/
void tri ( int * t , int n )
{
  return ;
}

/*
  L'utilisation de TAILLE_AFFICHAGE est optionnelle,
  et est juste présente pour permettre d'afficher partiellement
  de grands tableaux.
```

```
*/
void affiche ( int * t , int n )
{
  int i ;
  if ( n == 0 )
  {
    printf ( "[ ]\n" ) ;
    return ;
  }
  printf ( "[ %d " , t[0] ) ;
  for ( i = 1 ; ( i < n ) && ( i < TAILLE_AFFICHAGE ) ; i++ )
    printf ( ", %d " , t[i] ) ;
  if ( n > TAILLE_AFFICHAGE )
   printf ( ", ... " ) ;
  printf ( "]\n" ) ;
}

/*
  la fonction suivante est optionnelle, on peut
  choisir une petite taille et rentrer les valeurs dans
  le code au moment de la définition du tableau.
*/
void initialise ( int * t , int n )
{
  int i ;
  for ( i = 0 ; i < n ; i++ )
    t[i] = rand ( ) ;
}

int
main ( int argc , char * argv[] )
{
  int t[TAILLE] ;
  initialise ( t , TAILLE ) ;
  affiche ( t , TAILLE ) ;
  tri ( t , TAILLE ) ;
  affiche ( t , TAILLE ) ;
  return 0 ;
}
```

(b) On va aussi écrire une `choix_pivot` qui, elle aussi, ne fait rien. Les variantes de l'algorithme de tri rapide sont basées sur des fonctions de choix plus élaborées, mais on codera uniquement la version de base.

Solution

```
void choix_pivot ( int * t , int n )
{
  return ;
```

```
}
```

(c) **Procédure de drapeau hollandais.** Dans cette procédure, un tableau est divisé en 4 zones :

- Une zone bleu, qui commence toujours à l'indice 0, mais peut être vide, et qui contient les éléments strictement plus petits que le pivot ;
- Une zone blanche, qui commence juste après la zone bleu, et qui contient les éléments égaux au pivot. Au début, cette zone blanche contient juste le premier élément du tableau (le pivot) ;
- Une zone rouge, qui part de la fin du tableau, et contient tous les éléments strictement plus grands que le pivot. Au départ, cette zone est vide ;
- une zone grise, qui contient au départ tous les éléments du tableau sauf le premier, et qui correspond aux éléments qui n'ont pas encore été comparés au pivot. Elle est entre la zone blanche et la zone grise ;

Ces zones sont délimitées par 3 indices, `premier_blanc`, `dernier_blanc`, et `dernier_gris`. Ces indices évoluent quand on compare le dernier élément blanc avec l'élément suivant (*cf.* schéma ou S2). À chaque itération la zone grise doit diminuer d'un élément, et le parcourt du tableau est terminé quand la zone grise est vide. La procédure doit alors indiquer le premier et le dernier indice de la zone blanche.

Solution

```
void
echange ( int * a , int * b )
{
  int tmp ;
  tmp = *a ;
  *a = *b ;
  *b = tmp ;
}
void partition ( int * t , int n , int * premier , int * dernier )
{
  int dernier_gris = n -1 ;
  /* le pivot est a l'indice 0 du tableau */
  * premier = * dernier = 0 ;
  while ( *dernier < dernier_gris )
  {
    if ( t[ *dernier + 1] == t[ *dernier ] )
    {  // blanc
       *dernier = ( *dernier ) + 1 ;
       continue ;
     }
     if ( t[ *dernier + 1 ] < t[ *dernier ] )
     { // bleu
       echange ( t + *dernier + 1, t + *premier ) ;
       *premier = *premier + 1 ;
       *dernier = *dernier + 1 ;
       continue ;
     }
```

```
      // rouge
      echange ( t + *dernier + 1 , t + dernier_gris-- ) ;
  }
}
```

(d) La procédure de tri rapide récupère ensuite les indices de début et de fin de la zone blanche (les éléments entre ces deux indices sont correctement placés), et est appelée récursivement sur les zones bleu et rouge si elles ont strictement plus qu'un élément.

Solution

```
void tri ( int * t , int n )
{
  int fin_premier , debut_second ;
  if ( n <= 1 )
    return ;
  choix_pivot ( ) ;
  partition ( t , n , & fin_premier , & debut_second ) ;
  tri ( t , fin_premier ) ;
  tri ( t + debut_second + 1 , n - debut_second - 1 ) ;
}
```