ENSTA Paris
2023-2024

Résolution de problèmes algorithmiques – IN103

Alexandre Chapoutot

Feuille d'exercices 4

Objectif(s)

- ★ Manipuler des tas et des ensembles disjoints (union-find) ;
- ★ Écrire un algorithme de tri en utilisant une structure de données adaptées ;
- ★ Écrire un générateur de labyrinthes.

PRÉPARATION

Version de la bibliothèque `libin103`

Pour cette séance de TP, nous n'utiliserons pas de nouvelle version de la bibliothèque `libin103`. Nous utiliserons donc la version 1.4 qui doit être installée dans le répertoire `~/Library/libin103-1.4`. Si cela n'est pas le cas se référer à la feuille de TP précédente pour avoir les instructions d'installation.

Matériel pour le TP

Récupérez l'archive associé à cette séance de TP à l'adresse :

`https://perso.ensta-paris.fr/~chapoutot/teaching/in103/practical-work/in103-td4.tar.gz`

EXERCICES

Exercice 1 – Tri par tas

La structure de données tas peut être utilisée pour faire un tri de valeurs.

Fonctions utiles pour cet exercice

- `real_heap_init`
- `real_heap_destroy`
- `real_heap_insert`
- `real_heap_extract`
- `real_heap_size`

Question 1

Mettre en œuvre une fonction nommée, `heapsort` qui permet de trier dans l'ordre croissant un tableau de nombres flottants en utilisant un tas. Le prototype de cette fonction est

```
int heap_sort (double *tab, int size, double *result);
```

— La variable `tab` représente le tableau de données initiales (non triées) ;
— La variable `size` représente la taille de `tab` ;
— La variable `result` représente un nouveau tableau dont les données de `tab` ont été triées.

Un squelette de programme est donné dans le fichier `exo1/heap-sort.c`.

Solution:

```
int heap_sort (double *tab, int size, double *result) {

  int code;
  int i;
  real_heap_t heap;
  real_heap_init (&heap, real_MIN_HEAP);

  for (i = 0; i < size; i++) {
    code = real_heap_insert (&heap, tab[i]);
    if (code != 0) {
      real_heap_destroy (&heap);
      return EXIT_FAILURE;
    }
  }

  i = 0;
  while (real_heap_size (&heap) > 0) {
    double nb;
    code = real_heap_extract (&heap, &nb);
    if (code != 0) {
      real_heap_destroy (&heap);
      return EXIT_FAILURE;
    }
    result[i] = nb;
    i++;
  }

  real_heap_destroy (&heap);

  return 0;
}
```

Explication(s):

✿ A noter que la structure de tas peut être configurée pour être un tas min (la plus petite valeur de l'ensemble à la racine) ou un tas max (la plus grande valeur de l'ensemble à la racine). Dans le cas présent nous voulons un tas min.

Question 2

Comment modifier le prototype de la fonction `heapsort` pour permettre de faire un tri dans l'ordre croissant ou un tri dans l'ordre décroissant. Donnez un possible nouveau prototype.

Solution:

```
int heap_sort (double *tab, int size, double *result, bool inverse_order);
```

Explication(s):

✿ A minima on a besoin d'un drapeau permettant d'indiquer si on veut un tas min ou un tas max, une variable booléenne peut faire l'affaire en considérant que par défaut on fait un tri dans l'ordre croissant (le booléen est fixé à faux).

Solution:
Le code complet est :

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

#include "heap.h"

void print_array (double *tab, int size) {
  int i =0;
  for (i =0; i < size - 1; i++) {
    printf ("%f, ", tab[i]);
  }
  printf ("%f\n", tab[i]);
}

int heap_sort (double *tab, int size, double *result, bool inverse) {

  int code;
  int i;
  real_heap_t heap;
  if (! inverse) {
    real_heap_init (&heap, real_MIN_HEAP);
  }
  else {
    real_heap_init (&heap, real_MAX_HEAP);
  }

  for (i = 0; i < size; i++) {
    code = real_heap_insert (&heap, tab[i]);
    if (code != 0) {
      real_heap_destroy (&heap);
      return EXIT_FAILURE;
    }
  }

  i = 0;
  while (real_heap_size (&heap) > 0) {
    double nb;
    code = real_heap_extract (&heap, &nb);
    if (code != 0) {
      real_heap_destroy (&heap);
      return EXIT_FAILURE;
    }
    result[i] = nb;
    i++;
  }

  real_heap_destroy (&heap);

  return 0;
}

int main (){

  int code;
  int size = 7;
  double tab[] = { 0.1, -2.0, 12.3, 3.14159, -1.34, 202.9, -2.67 };
  double* result;
  result = malloc (size * sizeof(double));
  if (result == NULL) {
    return EXIT_FAILURE;
  }

  print_array (tab, size);

  code = heap_sort (tab, size, result, true);
  if (code == 0) {
    print_array (result, size);
  }

  free(result);
```

Question 3

En vous appuyant sur la fonction `heapsort`, mettez en œuvre une fonction, nommée `k_max`, qui permet d'extraire les k plus grandes valeurs d'un tableau d'entiers. Le prototype de la fonction est :

```
int k_max (double *tab, int size, double *result, unsigned int k);
```

— La variable `tab` représente le tableau de données initiales ;
— La variable `size` représente la longueur de `tab` ;
— La variable `result` représente le tableau des k plus grandes valeurs ;
— La variable `k` représente le nombre k des plus grandes valeurs souhaitées.

Un squelette de programme est donné dans le fichier `exo1/k_max.c`.

Solution:

```
int k_max (double *tab, int size, double *result, unsigned int k) {
  double *sortedTab;
  int code;

  if (k == 0) {
    return 0;
  }
  else if (k > size) {
    k = size;
  }

  sortedTab = malloc (size * sizeof(double));
  if (sortedTab == NULL) {
    return EXIT_FAILURE;
  }

  code = heap_sort (tab, size, sortedTab);
  if (code != 0) {
    return EXIT_FAILURE;
  }


  for (int i = 0; i < k; i++) {
    result[i] = sortedTab[i];
  }

  return 0;
}
```

Explication(s):

- Il faut prendre quelques cas pathologiques en compte comme $k = 0$ ou $k >$ taille du tableau.
- Autrement, il suffit d'appeler la fonction tri par tas (avec la configuration d'un tas max) et d'extraire les k premières valeurs.

Question 4 – « Tassification » – Optionnel

Dans le cours a été donné l'algorithme pour « tasser » un tableau sans recourir à l'utilisation d'un tableau supplémentaire. Mettez en œuvre la fonction dont le prototype est

```
void build_max_heap (double* tab, int size);
```

Cette fonction mettre le tableau de `double` sous la forme d'un tas max. Un squelette de programme est donné dans le fichier `exo1/heapify.c`.

Solution:

```
void max_heapify(double* tab, int size, int i) {
  int left = 2 * i + 1;
  int right = 2 * i + 2;
  int largest = i;

  if ( (left < size) && (tab[left] > tab[largest]) ) {
    largest = left;
  }

  if ( (right < size) && (tab[right] > tab[largest]) ) {
    largest = right;
  }

  if (largest != i) {
    double temp = tab[i];
    tab[i] = tab[largest];
    tab[largest] = temp;
    max_heapify (tab, size, largest);
  }
}

void build_max_heap (double* tab, int size) {
  /* Mieux partir du noeud floor(size / 2), le dernier noeud interne
     avant les feuilles */
  for (int i = size-1; i >= 0; i--) {
    max_heapify (tab, size, i);
  }
}
```

Explication(s):

- ✿ Dans le cours est donnée l'algorithme d'une fonction auxiliaire qui permet de « tasser » un tableau à partir d'une position. Il faut donc définir cette fonction auxiliaire `max_heapify`.
- ✿ La fonction `build_max_heap` appellera cette fonction auxiliaire `max_heapify` qui sera appeler sur tous les éléments du tableau à partir de la fin.
- ✿ On peut améliorer la complexité de la fonction `build_max_heap` en ne partant pas des feuilles mais du dernier nœud interne qui est à la positon size/2.

Exercice 2 – Générateur de labyrinthes

L'objectif de cet exercice est de construire un programme qui permet de générer un labyrinthe à partir d'une grille carrée. Intuitivement l'algorithme de génération du labyrinthe fonctionne de la manière suivante

— On tire au hasard un mur (c_1, c_2) dans la liste L de tous les murs (et ce mur ne sera plus considéré par la suite).
— Si les cases concernées par ce mur c_1 et c_2 ne sont pas dans une même classe d'équivalence alors on ignore le mur (on ne met pas le mur dans M) et on met c_1 et c_2 dans la même classe d'équivalence.
— Si elles sont dans la même classe d'équivalence, on met le mur (c_1, c_2) dans la liste M des murs formant le labyrinthe.
— On recommence ces étapes tant que toutes les cases ne sont pas dans la même classe d'équivalence.
— Les murs du labyrinthe sont ceux qui ont été mis dans liste M et ceux qui n'ont pas été utilisé dans L.

Fonctions utiles pour cet exercice

- `integer_uf_init`
- `integer_uf_destroy`
- `integer_uf_add_element`
- `integer_uf_components`
- `integer_uf_are_connected`
- `integer_uf_union`

Un squelette de programme est donné dans le fichier `exo2/maze-generator.c`

Question 1
Analysez le contenu du squelette de programme, quels sont les éléments qui sont déjà donnés ?

Solution:
Explication(s):

- Il y une structure décrivant un mur qui est définie. Un mur est une transition entre deux cases.
- La numérotation des cases est séquentielle dans la grille.
- Il y a une boucle qui génère la liste de tous les murs possibles dans une grille par un parcours de toutes les cases de la grille et on observe les cases voisines (positionné au sud et à l'est).
- Il y a une variable `maze` qui doit regrouper les murs sélectionnés pour former le labyrinthe.

Question 2
Écrivez l'algorithme de génération de labyrinthe.

Solution:

```
/* Debut de l'algorithme de generation du labyrinthe */
integer_uf_t dset;
integer_uf_init (&dset, totalCell);

for (int i = 0; i < totalCell; i++) {
  integer_uf_add_element (&dset, i);
}

/* Boucle principale de l'algorithme */
int len = nbEdge-1;
int r;
int cptWall = 0;
while (integer_uf_components(&dset) > 1) {
  r = rand() % len;

  edge_t e = edges[r];

  edges[r] = edges[len];
  edges[len] = e;
  len--;

  if (!integer_uf_are_connected (&dset, e.cell1, e.cell2)) {
    integer_uf_union (&dset, e.cell1, e.cell2);
  }
  else {
    maze[cptWall++] = e;
  }
}
```

Explication(s):

- On définit une structure de données union-find
- Il faut penser à ajouter toutes les cases dans cette structure de données avec la fonction `integer_uf_add_element`.
- Petite astuce pour tirer au sort les murs : on considère un tableau et à chaque tirage aléatoire on échange cet élément avec le dernier élément du tableau. On décrémente la taille du tableau. On a ainsi un ensemble de murs qui décroit.
- Autrement l'algorithme est assez direct

POUR S'ENTRAÎNER À LA MAISON

Exercice 1 – Palindrome

Un palindrome est une chaîne de caractères qui se lit de la même manière de gauche à droite et de droite à gauche.

Fonctions utiles pour cet exercice

- `character_stack_init`
- `character_stack_destroy`
- `character_stack_size`
- `character_stack_push`
- `character_stack_pop`
- `character_stack_peek`
- `character_queue_init`
- `character_queue_destroy`
- `character_queue_size`
- `character_queue_enqueue`
- `character_queue_dequeue`
- `character_queue_peek`
- `character_list_init`
- `character_list_head`
- `character_list_tail`
- `character_list_next`
- `character_list_data`
- `character_list_destroy`

Question 1

En utilisant un nombre fixe de structures de données de type pile ou file et un nombre fixe de variables de type entier et de type caractère, écrire un algorithme qui détermine si une chaîne de caractères est un palindrome. Nous supposerons que la chaînes de caractères ne contient pas de caractères espaces ni de caractères accentués.

Le prototype de la fonction à programmer sera

```
bool is_palindrome (char* str);
```

Solution:

```
bool is_palindrome (char* str) {

  int len = strlen (str);
  character_stack_t stack;
  character_stack_init (&stack);

  int i;
  for (i = 0; i < len / 2; i++) {
    character_stack_push (&stack, str[i]);
  }
  if (len % 2 == 1) {
    /* Ne pas considerer le caractere du milieu */
    i++;
  }
  for (; i < len; i++) {
    char c;
    character_stack_pop (&stack, &c);
    if (c != str[i]) {
      character_stack_destroy (&stack);
      return false;
    }
  }
  character_stack_destroy (&stack);
  return true;
}
```

Explication(s):

- ✿ Un point de vigilance est sur la taille de la chaîne de caractères si celle-ci est paire ou impaire. Dans ce dernier cas il faut faire un traitement particulier.
- ✿ Une seule pile est utile pour cet algorithme qui est relativement simple. On empile la première moitié de la chaîne puis on dépile en comparant les éléments de la dernière moitié de la chaîne.

APPROFONDISSEMENT

Exercice 1 – Arbres de segments

L'utilisation de tableaux pour représenter des arbres est commune dans la définition de structures de données arborescentes. Un arbre de segments est conçu spécialement pour gérer de manière efficace des problèmes impliquant des requêtes sur des tableaux de la forme :

— récupérer une information sur un intervalle d'indices $[\ell, r]$;
— modifier la valeur stockée à un indice i du tableau.

L'information associée à un intervalle d'indices $[\ell, r]$ peut concerner la somme des valeurs, la valeur maximale ou minimale, etc. Par exemple, le tableau suivant

```
int array[] = { 1, 3, 5, 7, 9, 11 };
```

est représenté par l'arbre de segments donné dans la figure 1 dans laquelle a été représentée l'ensemble des indices au niveau des nœuds ainsi que la somme des valeurs associées à ces indices.

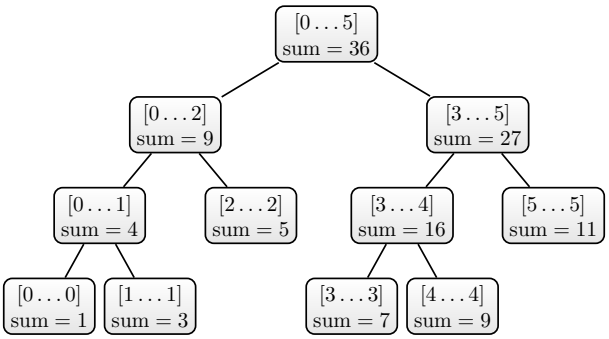

FIGURE 1 – Exemple d'arbre de segments

L'utilisation d'une structure arborescente permet d'avoir des complexité logarithmique pour accéder à la somme des élements. La solution naïve d'itérer sur les indices du tableau pour calculer l'information cherchée (somme, max, etc.) correspond à une complexité linéaire en nombre d'éléments.

Dans la suite de cet exercice, nous allons tout d'abord nous concentrer sur un arbre de segments pour des requête liées à la somme des valeurs du tableau pour des intervalle d'indices. Nous généraliserons ensuite pour différents types d'opération dans un second temps.

Les éléments pour cet exercice sont donnés dans le répertoire `exo4`.

Question 1 – Dichotomie sur les tableaux

La construction des arbres de segments reposent fortement sur un parcours dichotomique des tableaux. Avant de débuter la mise en œuvre des arbres de segments, codons une fonction qui affiche les éléments d'un tableau en appliquant le principe de dichotomie. La fonction aura le prototype suivant :

```
void print_array (int* array, int size);
```

et elle utilisera une fonction auxiliaire pour effectuer le parcours du tableau, cette fonction auxiliaire aura le prototype suivant :

```
void print_array_aux (int* array, int size, int debut, int fin);
```

Il est usuel d'avoir une fonction auxiliaire pour gérer la mise en œuvre d'un algorithme récursif. Cela permet d'avoir une interface de programmation simple pour l'utilisateur.

Solution:

```
void print_array_aux (int* array, int size, int start, int end, char* str) {
  if (start < 0 || end > size) {
    return;
  }

  printf ("%s : start = %d and end = %d\n", str, start, end);
  if (start == end) {
    printf ("array[%d] = %d\n", start, array[start]);
    return;
  }

  int mid = (start + end) / 2;
  print_array_aux (array, size, start, mid, "left");
  print_array_aux (array, size, mid + 1, end, "right");
}

void print_array (int* array, int size) {
  print_array_aux (array, size, 0, size-1, "whole");
}
```

Explication(s):

- ✿ On peut constater que le parcours dichotomique préseve l'ordre des indices. On peut faire un parallèle avec un arbre binaire de recherche avec un parcours infixe.
- ✿ La récursion démarre de l'intervalle $[0, \text{size} - 1]$
- ✿ Dans la correction un argument de type chaine de caractères a été ajouté pour essayer de tracer les parties du tableau qui sont parcourus lors des appels récursifs, on voit apparaître un parcours en profondeur d'abord.

Nous allons suivre la convention de nommage de la bibliothèque `libin103` pour mettre en œuvre l'interface de programmation des arbres de segments. Nous ne considérerons que des valeurs entières dans cet exerice.

La structure de données utilisées pour les arbres de segments est la suivantes

```
typedef struct integer_stree_t {
  int size; /**< nombre d'elements dans le tableau initial  */
  int max_size; /**< nombre d'elements dans l'arbre */
  int *tree; /**< tableau representant l'arbre de segment */
} integer_stree_t;
```

Elle est définie dans le fichier `exo4/integer_stree.h` ainsi que le prototype des fonctions composant l'API.

Question 2

Donnez le code des fonctions :

- `void integer_stree_init (integer_stree_t* stree);`
- `void integer_stree_destroy (integer_stree_t* stree);`

Solution:

Pour la fonction d'initialisation, on a :

```
void integer_stree_init (integer_stree_t* stree) {
  stree->size = 0;
  stree->max_size = 0;
  stree->tree = NULL;
}
```

et pour la fonction de destruction, on a :

```
void integer_stree_destroy (integer_stree_t* stree) {
  stree->size = 0;
  stree->max_size = 0;
  free(stree->tree);
}
```

Question 3

Mettez en œuvre la fonction principale de la construction des arbres de segments qui à partir d'un tableau d'entiers

et sa taille construit l'arbre. Pour cela, l'algorithme effectue un parcours dichotomique du tableau et à chaque appel récursif l'indice du tableau associé à l'arbre est calculé à l'aide des formules suivantes :

— Pour un nœud i, l'indice du fils gauche est à la position $2 \times i + 1$
— Pour un nœud i, l'indice du fils droit est à la position $2 \times i + 2$

Nous utiliserons une fonction auxiliaire pour la mise en œuvre de la récursion. Les prototypes des fonctions sont :

— `int integer_stree_build (integer_stree_t* stree, int* datas, int n);`
— `static int integer_stree_build_aux (integer_stree_t* stree, int* datas, int ss, int se, int current);`

Les arguments `ss` et `se` indiquent les indices de début et de fin de l'intervalle de dichotomie tandis que l'argement `current` indique le nœud courant de l'arbre dans lequel ajouter un élément.

Solution:
Pour la fonction principale, on a :

```
int integer_stree_build (integer_stree_t* stree, int* datas, int n) {
  int x = (int) (ceil(log2(n))); //Height of segment tree
  int max_size = 2 * (int) pow(2, x) - 1; //Maximum size of segment tree
  stree->tree = malloc (sizeof(int) * max_size);
  if (stree->tree == NULL) {
    return -1;
  }

  integer_stree_build_aux(stree, datas, 0, n - 1, 0);

  stree->size = n;
  stree->max_size = max_size;

  return 0;
}
```

et pour la fonction auxiliaire, on a :

```
static int integer_stree_build_aux (integer_stree_t* stree, int* datas, int ss, int se, int curre
  if (ss == se) {
    stree->tree[current] = datas[ss];
    return datas[ss];
  }

  int mid = get_middle(ss, se);
  stree->tree[current] =
    integer_stree_build_aux(stree, datas, ss, mid, current * 2 + 1)
    + integer_stree_build_aux(stree, datas, mid + 1, se, current * 2 + 2);
  return stree->tree[current];
}
```

Explication(s):

- ✿ Le mot cle `static` permet d'indiquer que cette fonction est privée et donc pas utilisable hors du paquet.
- ✿ La première étape dans la fonction principale consiste à calculer le nombre d'éléments devant composer l'arbre. En fonction du nombre d'éléments du tableau en argument on estime la hauteur d'un arbre binaire devant contenir ces éléments avant d'en déduire la langueur du tableau représentant l'arbre.
- ✿ Dans la fonction auxiliaire, on découpe le tableau en argument suivant le principe de la dichotomie, on coupe au milieu l'intervalle puis on traite la partie gauche puis la partie droite.

Question 4

Donnez le code de la fonction qui permet d'interroger l'arbre de segments pour un intervalle d'indice donné. Comme pour la fonction de construction, on utilisera une fonction auxiliaire pour gérer la mise en œuvre récursive. Les prototypes des fonctions sont

— `int integer_stree_query (integer_stree_t* stree, int qs, int qe, int* result);`
— `static int integer_stree_query_aux (integer_stree_t* stree, int ss, int se, int qs, int qe, int current);`

Solution:
Pour la fonction principale, on a :

```
int integer_stree_query (integer_stree_t* stree, int qs, int qe, int* result) {
  // Check for erroneous input values
  if (qs < 0 || qe > (stree->size - 1) || qs > qe) {
    *result = 0;
    return -1;
  }

  *result = integer_stree_query_aux (stree, 0, integer_stree_size(stree) - 1, qs, qe, 0);
  return 0;
}
```

Pour la fonction auxiliaire, on a :

```
static int integer_stree_query_aux (integer_stree_t* stree, int ss, int se, int qs, int qe, int c
  // If segment of this node is a part of given range, then return the
  // sum of the segment
  if (qs <= ss && qe >= se) {
    return stree->tree[current];
  }

  // If segment of this node is outside the given range
  if (se < qs || ss > qe) {
    return 0;
  }

  // If a part of this segment overlaps with the given range
  int middle = get_middle(ss, se);
  return integer_stree_query_aux (stree, ss, middle, qs, qe, 2 * current + 1) +
    integer_stree_query_aux (stree, middle + 1, se, qs, qe, 2 * current + 2);
}
```

Question 5

Pour terminer la programmation de l'API des arbres de segments, il faut coder la fonction de mise à jour d'une valeur du tableau qui utilisera les mêmes principes que les autres fonctions en s'appuyant sur une fonction auxiliaire. Les prototypes des fonctions sont :

- `int integer_stree_update (integer_stree_t* stree, int index, int value);`
- `static void integer_stree_update_aux (integer_stree_t* stree, int ss, int se, int current, int index, int value)`

Solution:
Pour la fonction principale, on a :

```
int integer_stree_update (integer_stree_t* stree, int index, int value) {
  if (index < 0 || index > (integer_stree_size(stree) - 1)) {
    return -1;
  }

  integer_stree_update_aux (stree, 0, integer_stree_size(stree) - 1, 0, index, value);
  return 0;
}
```

Pour la fonction auxiliaire, on a :

```
static void integer_stree_update_aux (integer_stree_t* stree, int ss, int se, int current, int in
  // Base Case: If the input index lies outside the range of this segment
  if (index < ss || index > se) {
    return;
  }

  if (ss == se && ss == index) {
    stree->tree[current] = value;
    return;
  }

  // If the input index is in range of this node, then update the value
  // of the node and its children
  if (se != ss) {
    int middle = get_middle(ss, se);
    integer_stree_update_aux(stree, ss, middle, 2 * current + 1, index, value);
    integer_stree_update_aux(stree, middle + 1, se, 2 * current + 2, index, value);
    stree->tree[current] = stree->tree[current*2+1] + stree->tree[current*2+2];
    return;
  }
}
```

Question 6

Comment généraliser les arbres de segments pour être utilisés pour d'autres requêtes que celles concernant la somme des valeurs ?

Solution:
Cf voir les fichiers `exo4/integer_streep.h` et `exo4/integer_streep.c`.
Explication(s):

- ✿ Il faut pouvoir passer en paramètre de la fonction d'initialisation deux informations :
  - — la fonction combinateur de deux valeurs de l'arbre (addition, multiplication, etc.). Le type de cette fonction est `int (*) (int, int)`. Un pointeur sur une fonction qui prend deux paramètres entiers et renvoie une valeur entière.
  - — un élément neutre associé à cette fonction combinateur, essentiellement dans la fonction de requête.
  - — Pas beaucoup de changements à prévoir, aux endroits où on a fait une addition on applique la fonction combinateur.

Exercice 2 – Application - Indicatrice d'Euler

L'objectif de cet exercice est de fournir une réponse au problème donné sur le site CODEFORCES, plus précisément l'exercice

`https://codeforces.com/contest/594/problem/D`

qui concerne le calcule de la fonction indicatrice d'Euler à partir d'un nombre généré par le produit d'éléments d'un tableau. Plusieurs requêtes sont enchainées.

Dans les contraintes de résolution, il ne faut pas que l'exécution d'un jeu de tests ne dépasse une durée de 3s et une consommation mémoire de de 256Mo. La lecture des données d'entrée est réalisée sur l'entrée strandard (`stdin`) et le

résultat est affiché sur la sortie standard (`stdout`). Quelques fonctions de lecture utiles pour cet exercice sont données dans le fichier `exo5/read.c`.

Par exemple, une exécution du programme suivrait le schéma :

```
./req.x < tests/euler-function-1.in | diff - tests/euler-function-1.out
```

Ce schéma permet d'envoyer les données du problème sur l'entrée standard du programme `req.x` et de récuper le résultat sur la sortie standard pour faire la comparaison avec le résultat attendu. Si rien de ne s'affiche à l'issu de cette commande c'est le résultat calculé par `req.x` est conforme au résultat attendu. Dans le répertoire `exo5/tests` il y a deux jeux de données (entrée/sortie) pour vous permettre de tester votre programme.

Pour rappel, la fonction d'Euler [1] donne le nombre d'entiers compris entre 1 et n (inclus) qui est premier avec n.

1. https://fr.wikipedia.org/wiki/Indicatrice_d\%27Euler

Solution:
Première partie.La solution de cet exercice nécessite l'utilisation de plusieurs structures de données dont les arbres de segments. Dans la solution proposée, l'utilisation d'ensembles est adpotée.
Principe de l'algorithme de la solution :
Explication(s):

- ✿ L'idée est d'utiliser un arbre de segments pour calculer les produits des valeurs d'un tableau désignées par un intervalle d'indices. Cet arbre de segment utilise la fonction multiplication comme combinateur de valeurs. Attention notre implémentation des arbres de segments commencent la numérotation à 0 alors que dans l'exercice les tableaux ont le premier indice à 1.

```
int combine_prod (int x, int y) {
  // neutral_elmt = 1
  return x * y;
}

int main (void) {
  int size = read_int ();
  //printf ("%d %d\n", size, INT_MAX);
  int* array = malloc (size * sizeof(int));
  if (array == NULL) {
    return EXIT_FAILURE;
  }
  read_array (array, size);

  integer_streep_t streep;
  integer_streep_init (&streep, combine_prod, 1);

  integer_streep_build (&streep, array, size);

  int nb_req = read_int ();

  while (nb_req-- > 0) {
    int left = 0;
    int right = 0;
    fscanf (stdin, "%d␣%d", &left, &right);

    int result = 0;
    int code = integer_streep_query (&streep, left-1, right-1, &result);
    if (code == 0) {
      euler_function_2 (result);
    }
  }

  integer_streep_destroy (&streep);
  free (array);
  return EXIT_SUCCESS;
```

Explication(s):

- ✿ Une fois la valeur calculée v, une décomposition en facteur premiers est réalisée et chaque facteur est ajouté dans un ensemble pour supprimer les occurrences multiples.

```
void factorize (int n, integer_set_t* factors) {
  int x = 2;
  while (n > 1) {
    if (n % x == 0) {
      integer_set_insert (factors, x);
      n = n / x;
    }
    else {
      x++;
    }
  }
}
```

Solution:
Seconde partie
Explication(s):

- ✿ Pour toutes les valeurs entières m entre 1 et n vérifiez si tous les facteurs de v ne divisent pas m

```
void euler_function_2 (int number) {
  integer_set_t factors;
  integer_set_init (&factors);
  factorize (number, &factors);

  int sum = 1;
  /* Pas necessaire de tester 1 et n car ils sont diviseurs */
  for (int i = 2; i < number; i++) {
    integer_list_elmt_t* elem = integer_list_head (&factors);
    bool flag = false;
    for (; elem != NULL; elem = integer_list_next (elem)) {
      int n = integer_list_data (elem);

      if (i % n == 0) {
        flag = true;
        break;
      }
    }

    if (flag == false) {
      sum++;
    }
  }
  integer_set_destroy (&factors);
  printf ("%d\n", sum % ((int)(1e9 + 7)));
```

- ✿ A noter qu'une autre solution passe par le caclul du PGCD entre m et v mais il semble que cette solution ne soit pas assez efficace (au moins sur la machine de l'enseignant) pour respecter la contrainte de temps d'exécution. Néanmoins, dans la correction de cet exercice une version avec le PGCD a été mise en œuvre pour comparer les deux approches.