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`.

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.

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`.

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`.

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 ?

Question 2

Écrivez l'algorithme de génération de labyrinthe.

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);
```

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.

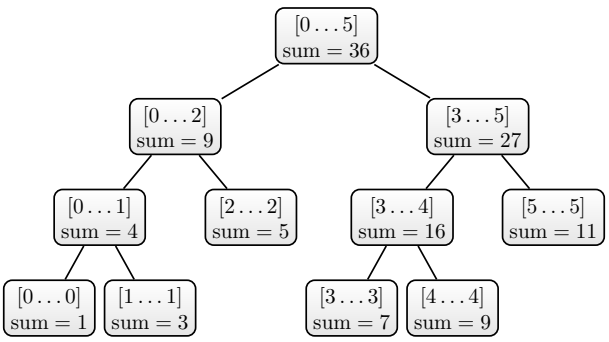

FIGURE 1 – Exemple d'arbre de segments

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.

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);`

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.

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);`

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)`

Question 6
Comment généraliser les arbres de segments pour être utilisés pour d'autres requêtes que celles concernant la somme des valeurs ?

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`