TP 1 : Structure hiérarchique Unir & Trouver

Rappel 1.1 Compilation C :

Compilation d'un fichier `source.c`, qui doit contenir une fonction `main`, en un exécutable `nom_executable`.

```
> gcc -o nom_executable source.c
```

Compilation de plusieurs fichiers `main.c`, qui doit contenir une fonction `main`, et `util1.c` et `util2.c` contenant d'autres fonctions, en un exécutable `nom_executable`.

```
> gcc -c util1.c
> gcc -c util2.c
> gcc -o nom_executable util1.o util2.o source.c
```

Rappel 1.2 Compilation OCaml :

Compilation de plusieurs fichiers `main.ml`, `util1.ml` et `util2.ml` en un exécutable `nom_executable`.

```
> ocamlc -o nom_executable util1.ml util2.ml main.ml
```

I Implémentation par tableau en OCaml

Exercice 1.3 Implémentation en tableau :

Dans cet exercice, on construit un type Unir&Trouver avec une implémentation à base de tableau. Pour chaque élément (les indices du tableau), le tableau associe son représentant.

1. Écrire un type `unirtrouver` pour cette implémentation.
2. Écrire une fonction `init : int -> unirtrouver` qui prend en entrée le nombre d'éléments de la partition et renvoie une partition initiale où chaque élément est dans son propre sous-ensemble.
3. Écrire une fonction `trouver : unirtrouver -> int -> int` qui renvoie le représentant d'un élément donné de la partition.
4. Écrire une fonction `unir : unirtrouver -> int -> int -> unit` qui unit les sous-ensembles des deux éléments donnés en arguments.
5. Écrire les tests pour ces fonctions et justifier leur pertinence.

Exercice 1.4 Coloration connexe :

Dans cet exercice, on souhaite colorier une image initialement en noir et blanc. Une image est représentée par un tableau en 2D dont les cases contiennent un entier représentant une couleur (0 pour le blanc, 1 pour le noir). Le code devra être dans un fichier séparé de l'exercice précédent.

1. Définir un type `image` pour représenter une image 2D.
2. Écrire une fonction `nouvelle_image : int -> image` tel que `nouvelle_image n` renvoie un tableau de taille $n \times n$ tel que chaque case contient la valeur 0 ou 1, aléatoirement.
3. Écrire une fonction `afficher_image_nb : image -> unit` pour afficher dans la console l'image en utilisant '.' pour les cases blanches et '#' pour les cases noires. On obtiendrait par exemple l'affichage suivant.

```
. . # # . # . . # .
# . # . # . # # # #
# # # # . . # # . .
. # . . # # . # . .
. . # # . # . # # #
# # # # # # . # . .
. # . # . . . # # .
. . # . # # # # . .
# # . . . # . # . #
. # . # # . . . . #
```

4. On veut maintenant identifier les composantes connexes, c'est-à-dire les cases de même couleur qui sont connectées par un de leur côté commun. Pour pouvoir utiliser la structure `unirtrouver` définie précédemment il nous faut une fonction `identifiant : int -> int -> int -> int` qui, pour une taille n de l'image, associe à chaque case (i, j) un identifiant unique (la fonction doit être injective). L'écrire et justifier qu'elle est injective.
5. Écrire une fonction `composantes : image -> unirtrouver` qui retourne une partition correspondant aux composantes connexes de l'image.
6. Écrire une fonction `recolore : image -> unirtrouver -> image` qui retourne une image en couleur (la valeur des cases pouvant dépasser 1) correspondant à l'image d'entrée où chaque composante connexe identifiée dans la partition `unirtrouver` a été coloriée avec une couleur qui lui est propre.
7. Écrire une fonction `afficher_image : image -> unit` qui affiche dans la console l'image en couleur, on pourra utiliser les symboles ASCII à partir de la lettre 'a' pour chaque couleur. La coloration de l'image précédente sera alors:

```
a a b b c d e e f g
b a b h i j f f f f
b b b b j j f f k k
l b m m n n o f k k
l l n n p n o f f f
n n n n n n o f q q
r n s n o o o f f q
r r t u f f f f q q
v v u u u f q f q w
x v u y y q q q q w
```

8. Tester l'ensemble avec

```
let _ =
  let img = nouvelle_image 10 in
  afficher_image_nb img;
  let taches = composantes img in
  let img_coloree = recolore img taches in
  afficher_image img_coloree
```

II Implémentation par arbre en C

Dans cette partie, on utilise une implémentation par arbre en C. Vérifier tout d'abord la bonne compilation avec le Makefile.

```
> cd src
> make
gcc -c unionfind.c
gcc -o unionfind_maze unionfind.o unionfind_maze.c
> ./unionfind_maze 10
Construction d'un labyrinthe de taille 10 x 10 ...
>
```

Le fichier `unionfind.c` contient l'ensemble des fonctions permettant de manipuler une structure Unir&Trouver (son en-tête associée contient les déclarations de ces fonctions).

Exercice 1.5 Fonctionnement par pointeurs :

1. Quelle est la structure de données obtenue en appelant `init_partition(3);` ? Représenter les structures `element_t`, avec les pointeurs.
2. À quoi sert la fonction `free_partition` ?
3. Réécrire la fonction `find` sans récursivité.
4. Quelles assertions sont nécessaires à ces fonctions pour garantir un bon fonctionnement ?

Exercice 1.6 Labyrinthe $\star$:

Le but de cet exercice est de se servir de la structure Unir&Trouver pour construire un labyrinthe *parfait*, c'est-à-dire un labyrinthe où il existe exactement un chemin pour lier deux cases. On représente un labyrinthe par un tableau en 2D tel que chaque case précise si elle a un mur à l'Ouest et un au Sud. La case du tableau contient un entier et on utilise la représentation binaire pour préciser quels murs sont présents:

Binaire	Décimal	Signification	Constante
0 0	0	Aucun mur	`AUCUN_MUR`
0 1	1	Mur à l'Ouest	`MUR_OUEST`
1 0	2	Mur au Sud	`MUR_SUD`
1 1	3	Murs sur les deux côtés	`DEUX_MURS`

Remarque 1.7 Opérations binaires (H.P.) :

Pour utiliser cette représentation binaire, on dispose en C d'opérateurs binaires, qui appliquent une opération bit à bit sur les opérandes.

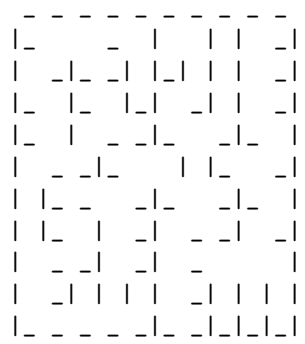
Le OU bit à bit, `a | b`, applique le OU logique sur chaque bit de `a` et `b`. Ex: `0110 | 1100 = 1110`.

Le ET bit à bit, `a & b`, applique le ET logique sur chaque bit de `a` et `b`. Ex: `0110 & 1100 = 0100`.

Le XOR bit à bit, `a ^ b`, applique le OU exclusif logique sur chaque bit de `a` et `b`. Ex: `0110 ^ 1100 = 1010`.

Le tableau 2D peut être représenté par un tableau à une dimension: pour accéder à la case (i, j) d'un tableau 2D de taille $n \times n$, on accède à la case $i \times n + j$ dans le tableau à une dimension. *Il est recommandé d'écrire un fonction intermédiaire pour ce calcul, afin d'éviter des erreurs de copier-coller.*

1. Écrire une fonction `int *init_labyrinthe(unsigned int taille);` qui renvoie un pointeur vers un tableau de taille `taille x taille`. Chaque case du tableau correspond à une case du labyrinthe, initialement avec un mur à l'ouest et un au sud.
2. Écrire une fonction `void afficher_labyrinthe(int *labyrinthe, unsigned int taille)` pour afficher le labyrinthe dans la console. On souhaite un rendu tel que ci-dessous:



pour un tableau ici de taille 10×10 et où la première case, en haut à gauche, a un mur Sud mais pas de mur Ouest. (Utiliser l'opérateur `&` pour vérifier si une case a un certain type de mur.)

3. Pour construire le labyrinthe parfait on utilise une structure Unir&Trouver où chaque élément est une case du tableau. Deux cases appartiennent au même sous-ensemble de la partition s'il existe un chemin entre les deux. On ne supprime donc un mur entre deux cases que si elles appartiennent à des classes différentes. Pour construire le labyrinthe, on prend donc chaque paire de cases adjacentes dans un ordre aléatoire et on décide de supprimer ou non le mur selon la classe des deux cases.

   Définir une structure `mur` contenant deux champs correspondants aux deux cases adjacentes à ce mur.

4. Pour un tableau de taille n, combien de murs doit-on considérer ?

5. Pour parcourir les murs dans un ordre aléatoire, on crée la liste des murs que l'on va ensuite mélanger. Écrire une fonction `struct mur *liste_murs(unsigned int taille)` qui renvoie l'ensemble des murs de cases adjacentes existant dans un tableau de taille `taille x taille`.

6. Écrire une fonction `void permuter_murs(struct mur* liste_murs, int premier, int second)` qui permute dans la liste donnée en entrée les murs d'indice `premier` et `second`.

7. Le mélange appliqué sera le *mélange de Knuth*:

   `Knuth` (tableau, n)
   begin
   | **for** *i = 0* **to** *n - 1* **do**
   | | $j \leftarrow$ Entier aléatoire entre 0 et i (inclus);
   | | Échanger les cases d'indices i et j dans le tableau;
   | **end**
   end

   Écrire une fonction `void melanger(struct mur *liste_murs, unsigned int taille)` qui applique le mélange de Knuth sur la liste de murs. Pour cela on appellera `rand()` qui renvoie un entier aléatoire entre 0 et `RAND_MAX` (une constante).

8. Écrire une fonction `void construire(int *labyrinthe, unsigned int taille)` qui reçoit en entrée une grille `labyrinthe` avec tous ses murs et qui la modifie pour en faire un labyrinthe parfait:

   - Création de la liste des murs
   - Mélange de Knuth sur cette liste
   - Création d'une partition pour les murs
   - Parcours de la liste mélangée: pour chaque mur, si les classes des deux cases sont différentes alors le mur est supprimé de `labyrinthe` et les deux cases fusionnent leurs classes.

Pour aller plus loin

- Les opérations sur le labyrinthe ont besoin de connaître sa taille qui devient une composante essentielle de l'objet: créer une structure `labyrinthe` qui stockera le tableau en plus de sa taille, adapter les fonctions précédentes.

Annexes

OCaml

Pour manipuler des tableaux:

- `Array.make n val` renvoie un tableau de taille n dont toutes les cases contiennent `val`.
- `Array.init n f` renvoie un tableau de taille n dont toutes les cases ont été remplies en appliquant `f : int -> 'a` sur l'indice de la case.

  Par exemple, `Array.init 4 (fun x -> 2 * x) = [0; 2; 4; 6]`
- `Array.make_matrix m n val` renvoie un tableau 2D de taille $m \times n$ dont toutes les cases valent `val`.
- `Array.length tableau` renvoie la taille (un entier) d'un tableau.
- `Array.map f tableau` applique une fonction `f : 'a -> 'b` sur chaque élément de `tableau : 'a array` et renvoie le tableau ainsi construit.
- `Array.iter f tableau` applique une fonction `f : 'a -> unit` sur chaque élément de `tableau : 'a array`.

Pour manipuler des tables d'association (dictionnaires):

- `Hashtbl.create n` Renvoie une table de hachage de taille n.
- `Hashtbl.add table cle valeur` Ajoute l'association clé $\rightarrow$ valeur à la table de hachage.
- `Hashtbl.mem table cle` renvoie `true` ssi la cle a une valeur associée dans la table.
- `Hashtbl.find table cle` renvoie la valeur associée à la clé dans la table.

Conversion entier vers caractère et inversement;

- `char_of_int : int -> char` Renvoie le caractère associé au code décimal ASCII.
- `int_of_char : char -> int` opération inverse.

Pour générer des nombres aléatoires:

- `Random.self_init ()` initialise la graine d'aléatoire en utilisant l'heure (à la milliseconde près).
- `Random.init graine` initialise la graine d'aléatoire avec la valeur donnée.
- `Random.int n` renvoie un entier aléatoire dans $[0, n-1]$.