1 Introduction

Ce TP est consacré au jeu de sudoku. Nous allons nous intéresser à la validité d'une grille complète de sudoku, puis à la résolution d'une grille incomplète.

Le sudoku est un jeu en forme de grille défini en 1979 par l'américain Howard Garns. Populaire au japon, ce jeu s'est propagé à travers le monde par son nom japonais.

Ce jeu se joue en général dans une grille de 9×9 cases, comme cela est représenté en figure 1. Chaque case de la grille peut accueillir un nombre entier entre 1 et 9. Le but du jeu est de réussir à remplir toutes les cases de la grille, de telle sorte que les contraintes suivantes soient respectées :

— Chaque ligne comporte exactement les nombres de 1 à 9,
— Chaque colonne comporte exactement les nombres de 1 à 9,
— Chaque carré de taille 3×3 comporte exactement les nombres de 1 à 9. Ces carrés sont ceux ayant des bordures plus larges sur la figure 1.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

FIGURE 1 – Une grille de sudoku en 9×9.

2 Balbutiements

Créez un fichier **sudoku.c**. La première partie du TP se fera sur ce fichier.

2.1 Modélisation de la grille de sudoku

Dans ce TP, la grille de sudoku sera représentée par un tableau de nombres entiers à deux dimensions, de taille 9×9. Le bout de code suivant vous permet de créer une grille de sudoku non initialisée :

```
int grille[9][9];
```

Les coordonnées de la grille sont notées $(x, y) \in \mathbb{N}^2$. La case de coordonnée $(0, 0)$ est située tout en haut à gauche de la grille. $x \in [0, 9[$ croît lorsqu'on va vers la droite. $y \in [0, 9[$ croît lorsqu'on va vers le bas. Ainsi, pour accéder à la case de coordonnées $(x = 3, y = 5)$ du tableau à deux dimensions **grille**, on se sert de la case **grille[5][3]**.

Créez une grille de sudoku dans votre fonction **main** et initialisez toutes les cases de la grille à 0.

2.2 Affichage de la grille

Afin de pouvoir visualiser facilement une grille de sudoku, je vous propose de créer la fonction dédiée suivante :

```
void display_sudoku(int grid[9][9]);
```

Implémentez la fonction **display_sudoku**, puis testez qu'elle fonctionne correctement en l'appelant depuis votre fonction **main**. Vous êtes libres du format d'affichage, mais l'ordre d'affichage des cases est imposé. Ce sera l'ordre naturel : la case de coordonnées $(x = 0, y = 0)$ doit être affichée en haut à gauche, celle de coordonnées $(x = 0, y = 1)$ doit être affichée juste en dessous...

2.3 Initialisation de la grille

Afin que l'initialisation d'une grille de sudoku ne soit pas trop fastidieuse, je vous propose de créer une fonction **load_grid** dont le rôle est d'initialiser une grille à partir d'une chaîne de caractères (du texte) :

```
void load_grid(int grid[9][9], const char * text);
```

La fonction **load_grid** place les caractères de la chaîne **text** dans la grille **grid**. Les caractères sont d'abord remplis sur la première ligne, de gauche à droite. La seconde ligne est ensuite remplie, également de gauche à droite. Ce processus est répété jusqu'à la dernière ligne. On suppose que **texte** est une chaîne d'au moins $9 \times 9 = 81$ caractères composée uniquement de caractères numériques. Lorsqu'on lit un '4', on met un 4 dans la case correspondante du tableau. Voici une manière d'appeler cette fonction :

```c
void une_fonction()
{
    int grid[9][9];
    // En C, lorsque plusieurs chaînes de caractères
    // constantes sont directement les unes à la suite
    // des autres, elles sont concaténées pour ne
    // former qu'une seule grande chaîne.
    const char * example = "534678912"
                           "672195348"
                           "198342567"
                           "859761423"
                           "426853791"
                           "713924856"
                           "961537284"
                           "287419635"
                           "345286179";

    load_grid(grid, example);
    // À partir d'ici, la contenu de l'exemple est chargé dans grid
}
```

Implémentez la fonction **load_grid** et vérifiez qu'elle marche correctement.

2.4 Exemples de grilles

Afin de tester votre programme, voici deux grilles valides dont vous pouvez vous servir :

```c
const char * valid1 = "534678912"
                      "672195348"
                      "198342567"
                      "859761423"
                      "426853791"
                      "713924856"
                      "961537284"
                      "287419635"
                      "345286179";

const char * valid2 = "639574182"
                      "541829376"
                      "782613954"
                      "198467954"
                      "365982417"
                      "427135869"
                      "956748231"
                      "813296745"
                      "274351698";
```

Vous pouvez modifier ces exemples pour les rendre invalides afin de tester vos fonctions.

3 Validité d'une grille complète de sudoku

Dans cette partie, nous nous intéressons au problème suivant : est-ce qu'une grille complète (dont toutes les cases ont des valeurs) est valide (respecte toutes les contraintes) ? On va créer une fonction booléenne dont le rôle est de répondre à cette question :

```
bool is_grid_valid(const int grid[9][9]);
```

Toutes les contraintes qui doivent être respectées dans le sudoku sont les mêmes. Elles consistent toutes à vérifier qu'un ensemble de 9 nombres comporte les nombres entiers de 1 à 9. Ainsi, je pense que réaliser les fonctions suivantes permet de simplifier le problème :

- Une fonction dont le rôle est de vérifier qu'un tableau de 9 nombres entiers représente les nombres entiers entre 1 et 9,
- Une fonction dont le rôle est d'extraire une ligne donnée d'une grille de sudoku dans un tableau de 9 nombres entiers,
- Une fonction dont le rôle est d'extraire une colonne donnée d'une grille de sudoku dans un tableau de 9 nombres entiers,
- Une fonction dont le rôle est d'extraire les valeurs d'un carré de taille 3×3 d'une grille de sudoku, en commençant à partir de coordonnées (x, y), et de stocker ces valeurs dans un tableau de 9 nombres entiers.

Implémentez la fonction **is_grid_valid**. Vous êtes pour cela libre des fonctions à implémenter : vous pouvez faire celles suggérées, ne faire aucune sous-fonction ou découper le problème différemment. Modifiez ensuite votre **main** afin qu'il affiche si la grille est valide ou non en appelant la fonction que vous venez d'implémenter. Si la grille est invalide, vous devez afficher pourquoi (si un nombre de la grille n'est pas entre 1 et 9, si la ligne l est invalide, si la colonne c est invalide, ou si le carré commençant aux coordonnées (x, y) est invalide).

4 Résolution d'une grille de sudoku incomplète

Le problème de résoudre une grille de sudoku est un problème difficile, puisqu'il fait partie des problèmes dits NP-complets. On peut cependant tenter de le résoudre par une méthode de type *brute-force*, qui essaye toutes les possibilités jusqu'à ce que toute la grille soit remplie (en respectant toutes les contraintes). Ce type de méthode s'essouffle pour des grandes tailles de grille, mais le cas auquel nous nous intéressons ici est assez simple.

Copiez votre fichier **sudoku.c** vers un fichier **sudoku2.c**. La suite du TP est à faire dans le fichier **sudoku2.c**.

4.1 Modélisation

Afin d'indiquer que la valeur d'une case est inconnue, nous allons nous servir de la valeur 0. Modifiez votre fonction **load_grid** si nécessaire afin qu'elle transforme les '0' lus en 0. Dans la suite du TP, nous allons appeler **trou** une case dont la valeur est inconnue. Voici la grille de la figure 1 sous forme de chaîne de caractères :

```
const char * tocomplete = "530070000"
                          "600195000"
                          "098000060"
                          "800060003"
                          "400803001"
                          "700020006"
                          "060000280"
                          "000419005"
                          "000080079";
```

Une fois que votre programme marchera sur cet exemple, vous pourrez essayer de résoudre une grille vide ou presque vide comme la suivante :

```
const char * empty =    "420000000"
                        "000000000"
                        "000000000"
                        "000000000"
                        "000000000"
                        "000000000"
                        "000000000"
                        "000000000"
                        "000000042"
```

Pour résoudre ce problème, nous avons besoin de savoir où sont les trous initiaux de la grille. Un trou peut être identifié par ses coordonnées dans la grille. Le nombre de trous étant borné par la taille de la grille ($9 \times 9 = 81$ dans ce cas), je vous propose de les stocker dans un tableau à deux dimensions comme celui-ci :

```
int holes[9*9][2];
```

La première dimension de ce tableau permet de stocker chaque trou (les numéros de trous pouvant être dans $[0, 9*9[$). Toutes les cases de ce tableau ne seront pas forcément utilisées puisque le nombre de trous dépend de la grille initiale choisie. Dans le tableau **holes**, dans le trou numéro **hole**, on va utiliser la première case de la seconde dimension pour stocker la coordonnée x et la seconde pour stocker la coordonnée y. Voici un exemple de code correspondant à la phrase précédente :

```
int hole = 0;
printf("Le trou d'indice %d est situé en (x=%d,y=%d)\n",
       hole, holes[hole][0], holes[hole][1]);
```

4.2 Une fonction pour vous aider à débugger

Afin de debugger facilement les fonctions que vous allez implémenter par la suite, je vous fournis une fonction qui a pour rôle d'afficher, pour chaque trou, les valeurs pouvant y être placées. Voici la fonction en question :

```
void display_holes(const int grid[9][9], const int holes[9*9][2],
↪  int nb_holes)
{
    printf("There are %d holes\n", nb_holes);
    for (int i = 0; i < nb_holes; ++i)
    {
        printf("Hole %d : (%d, %d), ", i, holes[i][0],
        ↪  holes[i][1]);
        int possible_values[9];
        int nb_pvalues = extract_possible_values(grid,
        ↪  holes[i][0], holes[i][1], possible_values);

        printf("available = ");
        display_int_array(possible_values, nb_pvalues, true);
    }
}
```

La fonction **display_holes** suppose que vous avez créé une fonction **display_int_array** :

```
void display_int_array(const int * array, int size, bool breakline);
```

La fonction **display_int_array** affiche le tableau d'entiers **array** de taille **size**. Cet affichage est suivi d'un saut de ligne si et seulement si **breakline** est vrai.

4.3 À la recherche des trous perdus

```
int find_holes(const int grid[9][9], int result[9*9][2]);
```

Implémentez la fonction **find_holes**, qui calcule le nombre de trous **nb_holes** dans la grille de sudoku **grid** et renvoie cette valeur. Les coordonnées des trous doivent être placés dans les **nb_holes** premières cases de **result** par cette fonction.

4.4 Recherche des valeurs pouvant être mises dans un trou

```
int extract_possible_values(const int grid[9][9], int x, int y,
↪  int result[9]);
```

Implémentez la fonction **extract_possible_values**, qui calcule le nombre de valeurs possibles **nb_values** pouvant être placées dans le trou de coordonnée (**x**, **y**) de la grille **grid**. Les valeurs possibles doivent êtres placées dans les **nb_values** premières cases du tableau **result**. Les valeurs possibles sont celles qui ne créeraient aucune violation des contraintes de la grille si elles étaient placées dans la case de coordonnées (x, y) de la grille. Il est assez facile de faire des erreurs en implémentant cette fonction, je vous invite donc à vous servir de la fonction **display_holes** fournie et de vérifier manuellement la validité de votre fonction.

4.5 *Brute-forcing*

L'algorithme que je vous propose de mettre en oeuvre est le suivant : on va essayer, de manière récursive, de combler tous les trous avec des valeurs valides. Dès qu'on place une valeur, on essaye de remplir le trou suivant. La récursivité n'étant pas au programme, je vous fournis deux fonctions qui vont faire ce calcul. Ces fonctions supposent que vous avez implémenté correctement les fonctions **find_holes** et **extract_possible_values**.

La fonction **bf_sudoku_rec** est une fonction récursive et n'est pas faite pour être appelée directement. Vous pouvez appeler la fonction **bf_sudoku**, qui vous renvoie vrai si et seulement si la grille a une solution. Si la grille a une solution, la grille donnée en paramètre sera remplie par la fonction. Si la grille n'a pas de solution, la grille reste inchangée.

```
bool bf_sudoku_rec(int grid[9][9],
                   const int holes[9*9][2],
                   int hole)
{
    if (hole >= 0)
    {
        int possible_values[9];
        int nb_pvalues = extract_possible_values(grid,
        ↪  holes[hole][0], holes[hole][1], possible_values);

        for (int i = 0; i < nb_pvalues; ++i)
        {
            grid[holes[hole][1]][holes[hole][0]] =
            ↪  possible_values[i];
            if (bf_sudoku_rec(grid, holes, hole-1))
                return true;
        }
        grid[holes[hole][1]][holes[hole][0]] = 0;

        return false;
    }
    else
        return true;
}

bool bf_sudoku(int grid[9][9])
{
    int holes[9*9][2];
    int nb_holes = find_holes(grid, holes);
    display_holes(grid, holes, nb_holes);

    return bf_sudoku_rec(grid, holes, nb_holes - 1);
}
```

Résolvez la grille de sudoku grâce à un appel de **bf_sudoku**. Si cela ne fonctionne pas, il doit y avoir un problème dans vos fonctions précédentes.