

Une **grille de Sudoku généralisé d'ordre n** est une grille de taille $n^2 \times n^2$ subdivisée en n^2 sous-grilles de taille $n \times n$. On cherche alors à remplir toutes les cases de la grille avec les contraintes suivantes : toute valeur de $\llbracket 1, n^2 \rrbracket$ apparaît exactement une fois sur chaque ligne/colonne/sous-grille (le Sudoku usuel est celui d'ordre 3).

Le but de ce TP est d'implémenter un algorithme résolvant des sudokus. Un fichier `fenelon_tp11.zip` est à télécharger sur la page web du cours, et à extraire dans votre dossier de travail.

1 Lecture de fichier et affichage

Pour cette partie, ouvrir le fichier `sudokuio.c`.

On représente en C une grille de sudoku d'ordre n partiellement remplie par une matrice de taille $n^2 \times n^2$ dont les cases vides contiennent la valeur 0 et les autres cases contiennent une valeur dans $\llbracket 1, n \rrbracket$.

1. Écrire une fonction `int** creer_matrice(int n)`; renvoyant une telle matrice de taille $n^2 \times n^2$ allouée sur le tas (pas besoin d'initialiser les valeurs des cases).
2. Écrire une fonction `void liberer_matrice(int** M, int n)`; libérant toute la mémoire allouée sur le tas par une telle matrice.

Le fichier `fenelon_tp11.zip` contient entre autre des fichiers `txt` contenant des exemples de grilles de sudoku partiellement remplies : un tel fichier contient n^2 lignes de n^2 caractères, chaque caractère représentant un entier (qui vaut 0 si la case est vide).

On rappelle les instructions suivantes :

- `fopen(nom_fichier, "r")`; ouvre un fichier en lecture, place un curseur virtuel sur le premier caractère du fichier, et renvoie un pointeur `FILE*` vers ce fichier ;
 - `fclose(fichier)`; ferme le fichier pointé ;
 - `fscanf(fichier, "%1d", adresse)`; lit le caractère actuellement sous le curseur virtuel du fichier pointé, écrit l'entier correspondant à ce caractère à l'adresse indiquée, et avance le curseur sur le prochain caractère.
3. Écrire une fonction `int** lire_sudoku(char* nom_fichier, int n)`; prenant en argument le nom d'un tel fichier et l'ordre n du sudoku en question, et renvoyant une matrice allouée sur le tas représentant cette grille de sudoku d'ordre n .
 4. Écrire une fonction `void afficher_sudoku(int** M, int n)`; qui affiche une grille de sudoku d'ordre n partiellement remplie sur la sortie standard. On sautera une ligne (resp. colonne) toutes les n lignes (resp. colonnes), et on affichera le caractère `"_"` pour une case vide.

```
$ gcc sudokuio.c sudokuio_test.c -o test_io
```

```
$ ./test_io sudoku1.txt
```

```
_72  _4  9__
3_4  _89  1__
819  _6  254

7_1  ___  _95
9__  _2  _7_
___  8_7  _12

4_5  _1  62_
237  ___  5_1
___  _25  7__
```

```
$ ./test_io sudoku2.txt
```

```
64_  5__  9__
__7  _8_  ___
_83  4__  _5_

3__  ___  _74
___  _5  _92
___  32_  8__

__6  _7_  ___
2__  6__  ___
8__  ___  _47
```

2 Résolution d'une grille de sudoku

On s'intéresse maintenant à la résolution d'une grille de Sudoku par un algorithme.

2.1 Remplissage déterministe

Une première technique de base consiste à regarder, pour chaque case, la liste des valeurs possibles (en fonction des valeurs déjà remplies dans la ligne/colonne/sous-grille correspondante). S'il n'y a qu'une seule valeur possible : on peut remplir la case sans se tromper.

5. Écrire une fonction `int* valeurs_possibles(int** M, int n, int i, int j)`; prenant en arguments une matrice `M` représentant une grille de sudoku d'ordre n et les coordonnées (i, j) d'une case de la grille, et renvoyant un tableau `possible` de $n^2 + 1$ entiers tel que :

- `possible[0]` contient le nombre de coups possibles ;
- $\forall v \in \llbracket 1, n^2 \rrbracket$, `possible[v]` contient 1 si la valeur v est possible dans la case (i, j) de `M` (et 0 sinon).

Si la case (i, j) de `M` contient déjà une valeur, on se contentera de mettre `possible[0]` à 0 et de renvoyer le tableau.

6. Écrire une fonction `int valeur_unique(int** M, int n, int i, int j)`; prenant en arguments une matrice `M` représentant une grille de sudoku d'ordre n et les coordonnées (i, j) d'une case de la grille, et renvoyant la seule valeur v qu'il est possible de mettre dans la case (i, j) de `M` s'il n'y a qu'une seule valeur possible (et 0 sinon).

Remarque : on prendra garde aux fuites mémoire.

7. Écrire une fonction `bool passage_deterministe(int** M, int n)`; prenant en arguments une matrice `M` représentant une grille de sudoku d'ordre n et qui parcourt une fois chaque case de la matrice `M` et remplit les cases n'ayant qu'une seule valeur possible. On renverra `true` si au moins une case de `M` a été remplie (et `false` sinon).

Suite à un tel passage, il se peut que d'autres cases deviennent des cases n'ayant qu'une seule valeur possible. Il faut donc continuer à appliquer cette stratégie tant qu'on arrive à remplir des cases.

8. Écrire une fonction `void remplir_deterministe(int** M, int n)` ; qui remplit la grille `M` de manière déterministe tant que c'est possible.

```
$ gcc sudokuio.c sudoku.c -o test_sudoku
```

```
$ ./test_sudoku sudoku1.txt
```

Matrice initiale :

```
_72  __4  9__
3_4  _89  1__
819  __6  254

7_1  ___  _95
9__  __2  _7_
___  8_7  _12

4_5  __1  62_
237  ___  5_1
___  _25  7__
```

Après le remplissage déterministe :

```
672  __4  9__
354  289  167
819  __6  254

7_1  __3  _95
9__  __2  _7_
5__  8_7  _12

4_5  __1  62_
237  __8  541
1__  _25  7__
```

```
$ ./test_sudoku sudoku2.txt
```

Matrice initiale :

```
64_  5__  9__
__7  _8_  ___
_83  4__  _5_

3__  ___  _74
___  _5_  _92
___  32_  8__

__6  _7_  ___
2__  6__  ___
8__  ___  _47
```

Matrice après le remplissage déterministe :

```
64_  5__  9__
__7  _8_  ___
_83  4__  _5_

3__  ___  _74
___  _5_  _92
___  32_  8__

__6  _7_  ___
2__  6__  ___
8__  ___  _47
```

Cette stratégie peut permettre de résoudre entièrement certaines grilles de sudoku très simples, mais cela ne suffit pas pour toutes les grilles. On pourrait essayer de mettre en place d'autres techniques effectuant des raisonnements analysant plusieurs cases d'un coup et arrivant à en déduire qu'une case contient nécessairement telle valeur, mais cela ne suffirait pas non plus sur certaines grilles de Sudoku. On s'intéresse alors à une autre approche.

2.2 Algorithme de backtracking

Un Sudoku est dit **diabolique** s'il possède bien une unique solution, mais qu'on ne pourra pas la trouver en effectuant uniquement des raisonnements sur les valeurs déjà présentes pour trouver des cases n'ayant qu'une seule valeur possible. On admet qu'il existe de telles grilles.

Ainsi, pour résoudre une grille de sudoku diabolique, on peut commencer à essayer de remplir la grille de manière déterministe, mais il y arrivera forcément un moment où on est bloqué : dans ce cas, il faut choisir une case non remplie de la grille, y mettre l'une des valeurs possibles, et essayer de continuer à remplir le sudoku. Si l'on arrive à finir de remplir la grille, c'est qu'on avait fait le bon choix. Sinon (on est bloqué), il faut effacer tout ce qu'on a fait depuis ce choix, et essayer un autre choix.

Cette stratégie algorithmique s'appelle le **retour sur trace** (ou "**backtracking**" en anglais).

Il est en général plus facile de coder une telle stratégie avec une fonction récursive : on effectue l'un des choix possibles sur la case courante, puis on fait un appel récursif qui essaye de remplir la case suivante et qui renvoie un booléen indiquant si ce choix a abouti à une solution. Si l'appel récursif renvoie **true**, on n'essaye pas d'autre choix (c'était le bon !) et on renvoie **true** pour propager l'information aux autres appels récursifs déjà empilés, sinon, on essaye un autre choix (et on refait alors un appel récursif). Si aucun des choix possibles n'a abouti, on renvoie **false** pour propager l'information aux autres appels récursifs déjà empilés.

9. Écrire une fonction **bool backtracking(int** M, int n, int i, int j)**; prenant en arguments une matrice **M** représentant une grille de sudoku d'ordre n et les coordonnées (i, j) d'une case de la grille, et qui effectue la stratégie par retour du trace décrite précédemment.
10. Quelle est la complexité temporelle de votre fonction dans le pire des cas (en fonction de l'ordre n et du nombre v de cases vides) ?
11. Est-il rentable de commencer par faire un remplissage déterministe ?
12. Écrire une fonction **bool resolution_sudoku(int** M, int n)**; remplissant le plus possible une grille **M** de sudoku d'ordre n et renvoyant **true** si elle possède une solution (et **false** sinon).
13. Modifier les commentaires dans la fonction **main** pour tester vos fonctions.

```
$ gcc sudokuio.c sudoku.c -o test_backtracking
```

```
$ ./test_backtracking sudoku1.txt
```

Matrice initiale :

```
_72  __4  9__
3_4  _89  1__
819  __6  254
```

```
7_1  ___  _95
9__  __2  _7_
___  8_7  _12
```

```
4_5  __1  62_
237  ___  5_1
___  _25  7__
```

Après le backtracking :

```
672  154  938
354  289  167
819  376  254
```

```
721  643  895
948  512  376
563  897  412
```

```
485  731  629
237  968  541
196  425  783
```

Sudoku résolu !

Vérification de la grille finale :
La grille est valide :)

```
$ ./test_backtracking sudoku2.txt
```

Matrice initiale :

```
64_  5__  9__
__7  _8_  ___
_83  4__  _5_
```

```
3__  ___  _74
___  __5  _92
___  32_  8__
```

```
__6  _7_  ___
2__  6__  ___
8__  ___  _47
```

Après le backtracking :

```
642  517  983
517  983  426
983  462  751
```

```
325  896  174
168  745  392
794  321  865
```

```
456  178  239
279  634  518
831  259  647
```

Sudoku résolu !

Vérification de la grille finale :
La grille est valide :)

3 Génération de grilles de sudoku

Si vous avez fini ce sujet, vous pouvez essayer de programmer les choses suivantes :

- tester si une grille possède une unique solution ;
- rendre une grille la plus difficile possible (on enlève la valeur d'une case tant que ça ne change pas le fait qu'il n'y a qu'une solution) ;
- générer des grilles de Sudoku aléatoirement.