TP 17 : Retour sur trace, Diviser-pour-régner

On utilisera la ligne de commande suivante pour compiler en C :

```Shell
$ gcc -O0 -Wall -Wextra -Wvla -Werror -fsanitize=address,undefined f.c -o f.exe
```

Au fur et à mesure de vos tests, si vous avez confiance en votre code et vos tests, vous pouvez augmenter le niveau d'optimisation en passant à `-O1`, puis `-O2`, puis `-O3`. Attention, plus le niveau d'optimisation est élevé, plus le temps de compilation est long (et plus les messages d'erreurs sont difficiles à comprendre, par exemple une partie des appels de fonction seront optimisés / supprimés et ne seront plus présent dans la trace de la pile d'appel affichée.)

Exercice 1 – Résolution d'un Sudoku

On va appliquer le principe du backtracking à la résolution de Sudoku. Pour rappel, le principe est le suivant : chaque ligne, colonne et carré de taille 3×3 comporte seulement une occurrence des chiffres de 1 à 9. On vous fournit une grille déjà remplie dans certaines cases et l'on va supposer qu'il existe exactement une solution à cette grille.

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

On représentera une grille comme une matrice d'entiers de taille 9×9, c'est-à-dire `int g[9][9]`. Une case vide est représentée par `g[i][j] = 0`. Dans l'exemple, on a `g[0][0] == 2`.

▶ **Question 1** Implémenter la grille ci-dessus dans une variable globale (définie avant toutes les autres fonctions, y compris la fonction `main`).

▶ **Question 2** Écrire une fonction `void affiche_grille(int grille[9][9])` affichant un sudoku. On pourra par exemple afficher la grille avec le format suivant :

```Shell
|2|5| | |3| |9| |1|
| |1| | | |4| | | |
|4| |7| | | |2| |8|
| | |5|2| | | | | |
| | | | | |9|8|1| | |
| |4| | | |3| | | |
| | | | |3|6| | |7|2|
| |7| | | | | | |3|
|9| |3| | | |6| |4|
```

▶ **Question 3** Implémenter une fonction de signature `void suivant(int grille[9][9], int i, int j, int* x, int* y)` tel que l'appel `suivant(g, i, j, &x, &y)` modifie les valeurs de `x` et `y` pour qu'elles contiennent la prochaine case libre à partir de (i, j) dans l'ordre de gauche à droite puis du bas vers le haut.

▶ **Question 4** Écrire trois tests vérifiant le fonctionnement de `suivant` dans la fonction `main` en utilisant la fonction `assert`.

▶ **Question 5** En déduire une fonction `bool est_remplie(int grille[9][9])` vérifiant si la grille de Sudoku est remplie de nombres. *On ne vérifiera pas que le placement des nombres est correct, juste que chaque case contient un nombre.*

▶ **Question 6** Écrire une fonction `bool valide(int grille[9][9], int i, int j)` renvoyant `true` si le nombre dans la case (i, j) n'invalide pas la grille, `false` sinon.

▶ **Question 7** Écrire trois tests vérifiant le fonctionnement de `valide` dans la fonction `main` en utilisant la fonction `assert`.

Maintenant, on peut appliquer le principe du backtracking pour résoudre un sudoku, dans lequel les variables sont les cases vides et l'ordre dans lequel on fixe les variables est donné par l'application successive de la fonction `suivant` (on l'applique jusqu'à trouver une case vide). Comme pour les n reines en cours, on va modifier la grille en place au cours de l'exploration de l'arbre des choix.

▶ **Question 8** Écrire une fonction récursive `void resout_aux(int grille[9][9], int i, int j)` qui résout la grille en partant de la case (i, j) en la remplissant d'une solution si c'est possible et laisse la grille inchangée sinon.

▶ **Question 9** En déduire une fonction `void resout(int grille[9][9])` qui résout la grille en entrée.

Exercice 2 – Nombre d'inversions dans un tableau

On continue en C. Dans cet exercice, l'objectif sera de compter le nombre d'inversions dans un tableau d'entiers. Une petite question technique d'abord :

▶ **Question 1** Implémenter le tri fusion sur les tableaux en C, avec le prototype suivant : `void tri_fusion(int t[], int deb, int fin)` où l'appel `tri_fusion(t, deb, fin)` trie en place la portion de tableau entre `deb` inclus et `fin` exclus. *Pour la fonction de fusion, on pourra l'écrire avec le prototype* `void fusion(int t[], int deb, int mil, int fin)` *où l'on fusionne les tranches de tableau entre* `deb` *inclus et* `mil` *exclus d'une part, et entre* `mil` *inclus et* `fin` *exclus d'autre part, en supposant ces deux tranches triées.*

Définition 1 – Inversion

Dans un tableau T, une inversion est un couple $(i, j) \in [\![0, n-1]\!]^2$ tel que $i < j$ et $T[i] > T[j]$.

On notera $\phi(T)$ le nombre d'inversions dans T.

▶ **Question 2** Majorer T, et identifier quels tableaux atteignent ce maximum.

▶ **Question 3** Proposer une fonction "en force brute" de prototype `int nombre_inversions_naif(int t[], int n)` avec `n` la taille de `t` comptant ces inversions et identifier sa complexité en fonction de n.

On va essayer d'améliorer cet algorithme en utilisant une stratégie Diviser-pour-Régner :

— **Diviser :** on sépare le tableau T en deux moitiés T_1 et T_2.

— **Régner :** on calcule récursivement $\phi(T_1)$, $\phi(T_2)$
— **Reconstruire :** on calcule $\psi(T_1, T_2)$ le nombre d'inversions dans T entre éléments de T_1 et éléments de T_2 [a], et on renvoie $\phi(T_1) + \phi(T_2) + \psi(T_1, T_2)$.

La reconstruction utilise le fait qu'une inversion (i, j) est soit entièrement dans T_1, soit entièrement dans T_2, soit "à cheval" entre T_1 et T_2 (c'est-à-dire i est un indice dans T_1 et j dans T_2).
Comme pour le tri fusion sur les tableaux, on va éviter de littéralement découper le tableau en deux et plutôt prendre en entrée de la fonction les deux indices délimitant la portion du tableau dans laquelle on compte le nombre d'inversions. Ainsi, l'étape **diviser** consiste simplement à calculer l'indice au milieu de cette portion de tableau, en $\mathcal{O}(1)$. Le calcul de $\psi(T_1, T_2)$ est plus compliqué :

▶ **Question 4** Justifier que $\psi(T_1, T_2) = \psi(\text{SORT}(T_1), \text{SORT}(T_2))$.

Ainsi, pour calculer $\psi(T_1, T_2)$, on peut d'abord calculer des copies de T_1 et T_2 triées, puis calculer $\psi(\text{SORT}(T_1), \text{SORT}(T_2))$ avec une fonction se restreignant aux tableaux triés. Notons T_1' et T_2' ces copies triées : on remarque que si $T_1'[i] > T_2'[j]$, alors pour tout $k \geq i$ on a $T_1'[k] > T_2'[j]$.

▶ **Question 5** Implémenter une fonction de prototype `int nombre_inversions_croisees(int t1[], int n, int t2[], int m)` tel que pour deux tableaux triés `t1` de taille `n` et `t2` de taille `m`, l'appel `nombre_inversions_croisees(t1, n, t2, m)` renvoie $\psi(T_1, T_2)$.

▶ **Question 6** Déterminer la complexité de la fonction `nombre_inversions_croisees` en fonction de n et m. *On attend une complexité en $\mathcal{O}(n + m)$.*

▶ **Question 7** En déduire une fonction `int nombre_inversions_aux(int t[], int deb, int fin)` calculant le nombre d'inversions de `t` entre les indices `deb` inclus et `fin`. En déduire une fonction `int nombre_inversions(int t[], int taille_t)` calculant le nombre d'inversions du tableau `t`.

Normalement, vous obtenez un algorithme dont la complexité temporelle vérifie la relation de récurrence suivante :

$$C(n) \leq 2C\left(\frac{n}{2}\right) + \mathcal{O}(n \log n)$$

▶ **Question 8** Justifier sommairement que $C(n) = \mathcal{O}\big(n(\log n)^2\big)$.

Dans cette fonction, on trie à chaque étape des sous-tableaux, puis on jette le résultat de ce tri (pour ne garder que le nombre d'inversions obtenu). En fait, on peut décorer l'algorithme du tri fusion pour calculer le nombre d'inversions en même temps que l'on trie.

▶ **Question 9** Adapter la fonction `tri_fusion` pour calculer le nombre d'inversions dans le tableau. La complexité temporelle sera, comme pour le tri fusion, en $\mathcal{O}(n \log n)$.

a. autrement dit $\psi(T_1, T_2) = \phi(T_1 \mathbin{@} T_2)$ avec @ désigne la concaténation.