

## TP D'OPTION INFORMATIQUE 1

### Application des arbres

## 1 Résolution de Sudoku par backtracking

L'objectif de cette partie est de résoudre un problème de Sudoku : on considère une grille  $9 \times 9$  représentée par un tableau de tableaux, chaque case contenant un nombre de 0 à 9. Les 0 correspondent aux cases libres : l'objectif est de les remplacer par des nombres de 1 à 9 de sorte que chaque ligne, chaque colonne et chacun des 9 carrés  $3 \times 3$  partitionnant la grille contiennent une et une seule fois chaque numéro.

1. Écrire une fonction `afficher_grille` prenant en argument un tableau de tableaux d'entiers et l'affichant de façon lisible.
2. Écrire trois fonctions `test_ligne`, `test_colonne`, `test_carre`, prenant en argument une grille partiellement remplie, un indice de ligne `i` et un indice de colonne `j`, et testant si la case  $(i, j)$  respecte les contraintes pour les lignes, les colonnes ou les carrés.
3. Écrire une fonction `resoudre_sudoku` prenant en argument une grille partiellement remplie et affichant une grille complétée respectant les contraintes du sudoku. On supposera que la grille initiale admet bien une solution. On procédera par backtracking.

## 2 Files de priorité avec modification

L'objectif de cette partie est d'implémenter une structure de files de priorité ajoutant une opération permettant de modifier la priorité associée à une valeur. On supposera que les valeurs présentes dans la file sont toujours distinctes, et comprises entre 0 et `c-1` (la capacité de la file).

L'implémentation de ces files de priorité va utiliser un tas binaire, similairement au cas vu en cours, mais également un tableau `position` de taille `c` qui stocke la position dans le tas binaire de toutes les valeurs qui y sont présentes : si `v` est une valeur présente dans le tas, alors `position.(v)` est l'indice `i` auquel on trouve le couple  $(p, v)$  dans le tas, avec `p` la priorité associée à `v`. Le type de ces files de priorité sera donc :

```
type file_prio = {
mutable taille : int ;
tas:(int*int) array;
position : int array};;
```

Par défaut, la position d'une valeur non présente dans le tas sera égale à `c`.

1. Donner la représentation dans ce type d'une file de priorité de capacité 5, contenant la valeur 3 avec priorité 10, la valeur 2 avec priorité 1 et la valeur 4 avec priorité 0.
2. Implémenter les opérations de cette structure de données :
  - `Créer_file_prio (c)` : renvoie une file de priorité vide de capacité `c`
  - `est_vide f` : renvoie un booléen indiquant si la file est vide
  - `ajouter_valeur f p v` : modifie la file `f` en lui ajoutant la valeur `v` associée à la priorité `c`, ou affiche un message d'erreur si la file est pleine.
  - `defiler f` : modifie la file `f` en supprimant une valeur de priorité maximale, et renvoie cette valeur, ou affiche un message d'erreur si la file est vide.
  - `modifier_prio f p v` : modifie la priorité de la valeur `v` à `p`, ou affiche un message d'erreur si la valeur `v` n'est pas dans la file. On prendra garde d'obtenir une complexité logarithmique pour cette fonction.

### 3 Tri par tas

Le tri par tas d'un tableau  $t$  suit les étapes suivantes :

- On fixe une variable `taille`, valant initialement la longueur de  $t$ , et on va interpréter les `taille` premières cases de  $t$  comme la représentation d'un arbre binaire complet à gauche.
- On parcourt  $t$  de droite à gauche et, pour chaque élément (qui est alors la racine d'un sous-tas), on l'entasse. À l'issue de cette boucle,  $t$  représente donc un tas.
- on met l'élément d'indice 0 à sa place en le permutant avec l'élément d'indice `taille - 1`. On décrémente `taille` pour sortir cet élément du tas (et donc ne plus le déplacer). On entasse l'élément arrivé sur l'indice 0 pour retrouver un tas.
- On itère l'étape précédente pour chaque élément du tableau.

Implémenter le tri par tas. Préciser sa complexité.