TP D'OPTION INFORMATIQUE 1
Application des arbres

1 Implémentation des files de priorité

1. Écrire une fonction `remonter` prenant en argument un tableau t et un entier i, supposant que t est un i-sur-tas, et permutant ses éléments pour en faire un tas.
2. Écrire une fonction `entasser` , prenant en argument un tableau t , un entier i et un entier k, et permutant les éléments du i-sous-tas formé des k premiers éléments de t de façon à en faire un tas.
3. On considère le type suivant pour implémenter les files de priorité :

```
type ('a,'b) file_prio =
    {mutable taille :int;
    tas : ('a*'b) option array};;
```

Implémenter les opérations suivantes :
- **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 ;
- **enfiler f c v** : modifie la file f en lui ajoutant la valeur v associée à la priorité c ;
- **défiler f** : si f est non-vide, modifie la file f en supprimant une valeur de priorité maximale, et renvoie cette valeur.

Déterminer leur complexité.

2 Résolution de Sudoku par backtracking

L'objectif de cette partie est de résoudre un problème de Sudoku : on considère une grille 9×9 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 3×3 partitionnant la grille contiennent une et une seule fois chaque numéro.

1. Écrire une fonction `afficher_grille` prenant en argument un tableau de tableau 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, en travaillant en place dans la grille (on n'oubliera pas de remettre un 0 dans une case après avoir essayé de lui attribuer toutes les valeurs sans atteindre de solution).

3 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.
L'implémentation de ces files de priorité va utiliser un tas binaire, comme dans la partie 1, mais également une table de hachage `position` 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 `Hashtbl.find 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 ('a,'b) file_prio =
    {mutable taille :int;
    tas : ('a * 'b) option array;
    position : ('b,int) Hashtbl.t }
;;
```

1. Donner une représentation possible 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.

   Déterminer leur complexité.

4 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.