On considère dans ce TP une version généralisée du jeu de Tic-tac-toe. Le principe du jeu reste le même, mais on cherche à aligner un nombre différents de symboles sur une grille de taille différente. On notera $ttt(k, n)$ un jeu de Tic-tac-toe où le but est d'aligner k symboles (en ligne, colonne ou diagonale) sur une grille $n \times n$.

Exercice 1

[À faire après le TP]

1. Que peut-on dire d'un jeu $ttt(k, n)$ quand $k > n$?

Pour toute la suite du TP, on supposera $k \leqslant n$.

2. Montrer qu'il existe une stratégie gagnante pour le premier joueur facile à mettre en place pour $ttt(2, n)$, pour tout $n \geqslant 2$.
3. Montrer qu'il existe une stratégie gagnante pour le premier joueur facile à mettre en place pour $ttt(3, n)$, pour tout $n \geqslant 4$.

Pour le TP, on utilisera les fichiers `dicts.h` et `dicts.c` (trouvable sur le dépôt). On trouvera également un fichier `TP7.c` à modifier pendant le TP, ainsi qu'un fichier `makefile`. On tapera `make build`, `make run` ou `make all` dans une console (en s'étant placé dans le bon répertoire) pour exécuter l'une des commandes. Elles correspondent à une commande pour compiler, une commande pour exécuter et une commande pour faire les deux à la fois.

Dans la suite du TP, on encode un jeu $ttt(k, n)$ par le type suivant :

```
struct TTT {
    int k;
    int n;
    int* grille;
};

typedef struct TTT ttt;
```

tel que si `jeu` est de type `ttt`, alors `jeu.k` et `jeu.n` représente k et n, et `jeu.grille` correspond à un tableau d'entier unidimensionnel représentant la grille de jeu ligne à ligne. La figure 1 donne un exemple de grille et le tableau associé. On utilise ici des numéros plutôt que les usuels cercles et croix. Les cases vides correspondront à des zéros dans le tableau.

1		1
	2	1
2		

```
int* grille = {1, 0, 1, 0, 2, 1, 2, 0, 0};
```

FIGURE 1 – Une grille de $ttt(3, 3)$ et le tableau associé.

Exercice 2

1. Écrire une fonction `ttt init_jeu(int n, int k)` qui initialise un jeu de $ttt(k, n)$ avec une grille vide (donc ne contenant que des zéros).
2. Écrire une fonction `int* repartition(ttt jeu)` qui renvoie un tableau `tab` de taille 3 correspondant à la répartition des cases dans le jeu : `tab[i]` correspond au nombre d'occurrences de la valeur `i` dans la grille. La fonction renverra une erreur si la grille contient des numéros autres que 0, 1 ou 2.
3. En déduire une fonction `int joueur_courant(ttt jeu)` qui renvoie le numéro du joueur dont c'est le tour (en supposant que c'est toujours le joueur 1 qui commence). La fonction renverra 0 si toutes les cases de la grille sont remplies.
4. Écrire une fonction `void jouer_coup(ttt jeu, int lgn, int cln)` qui joue un coup pour le

joueur courant dans la grille à une ligne et une colonne données. La fonction devra afficher un message d'erreur et ne rien faire si la case correspondante ne contient pas 0. On fera la conversion entre un couple d'indices de ligne et de colonne et l'indice de la case dans le tableau unidimensionnel.

Exercice 3

Pour déterminer l'existence d'un alignement gagnant pour un joueur `joueur` donné, on choisit une case de départ et une direction. On vérifie alors qu'il y a bien k cases consécutives dans cette direction prenant la valeur `joueur`. Il suffit alors de faire la vérification pour toutes les cases de départ et toutes les directions possibles.

On encode une **direction** par un entier `di` correspondant à un différentiel d'indices. Les cases à vérifier depuis la case d'indice `i` seront donc `i`, `i+di`, `i+2*di`, ...

1. À quelle valeur `di`, en fonction de n, correspondent les directions Est, Sud-est, Sud et Sud-ouest ?
2. Pourquoi n'est-il pas nécessaire d'envisager les 4 autres directions pour vérifier si un joueur est gagnant ?
3. Écrire une fonction `bool alignement(ttt jeu, int i, int di, int joueur)` qui prend en argument un jeu, un indice de départ, une direction et un joueur et indique s'il existe un alignement gagnant pour le joueur depuis la case de départ, dans la direction donnée. On prendra garde, en gardant en mémoire la ligne et la colonne de la case courante, à ne pas sortir de la grille.
4. En déduire une fonction `bool gagnant(ttt jeu, int joueur)` qui indique si un joueur est gagnant ou non.

Exercice 4

Pour faire le calcul des attracteurs, il est nécessaire de garder des résultats pré-calculés en mémoire. Pour ce faire, on encode une grille d'un jeu $ttt(k, n)$ par un entier à n^2 chiffres en base 3.

1. Écrire une fonction `int encodage(ttt jeu)` qui calcule un tel entier. On supposera qu'il n'y a pas de dépassement d'entiers (on travaillera avec des petites grilles).

On se donne une structure de dictionnaire `dict` implémentée par une table de hachage. On dispose des primitives suivantes :

- `void dict_free(dict D)` libère la mémoire occupée par un dictionnaire ;
- `dict create(void)` crée un dictionnaire vide ;
- `int size(dict D)` renvoie la taille d'un dictionnaire ;
- `bool member(dict D, int k)` teste l'appartenance d'une clé à un dictionnaire ;
- `int get(dict D, int k)` renvoie la valeur associée à une clé dans un dictionnaire ;
- `void add(dict* D, int k, int v)` ajoute une association (clé, valeur) à un dictionnaire ;
- `void del(dict* D, int k)` supprime une association d'un dictionnaire.

Attention, dans les fonctions précédentes et pour la question suivante, on utilise un pointeur de dictionnaire en argument, car le dictionnaire est modifié par la fonction.

2. Écrire une fonction `int attracteur(ttt jeu, dict* D)` qui prend en argument un jeu et un dictionnaire et renvoie le numéro de l'attracteur auquel appartient la grille du jeu. La fonction devra mémoriser le résultat dans le dictionnaire s'il n'y est pas déjà avant de renvoyer la valeur. On considèrera qu'une position nulle est dans l'attracteur 0.
3. En déduire une fonction `int strategie_optimale(ttt jeu, dict D)` qui détermine le coup optimal à jouer étant donné un jeu et un dictionnaire contenant des numéros d'attracteurs.

Exercice 5

On cherche à créer une interface pour jouer au jeu contre l'ordinateur.

1. Écrire une fonction `void afficher(ttt jeu)` qui affiche le jeu en console. Voici par exemple une manière d'afficher un jeu $ttt(3,4)$. On indiquera les numéros de lignes et de colonnes.

```
  0 1 2 3
 +-+-+-+-+
0|O|O|X|O|
 +-+-+-+-+
1| |X| | |
 +-+-+-+-+
2| | | | |
 +-+-+-+-+
3| | |X| |
 +-+-+-+-+
```

2. Écrire une fonction `void jouer_partie(int k, int n)` qui crée une partie de $ttt(k,n)$ et permet de jouer contre l'ordinateur. La fonction devra :
   - demander si le joueur humain souhaite commencer ou non ;
   - demander à chaque coup du joueur humain la ligne et la colonne où il souhaite jouer ;
   - afficher la grille après chaque coup joué (par l'humain ou l'ordinateur) ;
   - arrêter la partie dès qu'un joueur a gagné ou quand la grille est pleine, et afficher le joueur gagnant.

   On rappelle que la saisie de valeur pendant l'exécution de la fonction peut se faire de la manière suivante :

```c
char c;
printf("Voulez-vous commencer ? (o/n) ");
scanf("%c", &c);

int lgn;
printf("Saisir la ligne : ");
scanf("%d", &lgn);
```