Projet ***À remettre au plus tard le 30 octobre***

Consignes

Vous devez déposer sur Moodle le ou les fichiers contenant votre projet. Il devra contenir une brève présentation de votre travail : ce qui est fait (ou non), ce qui fonctionne correctement (ou non), des choix que vous avez effectués le cas échéant, et tout commentaire pertinent.

Chaque fonction doit être commentée (son rôle, ses arguments, son résultat et toute explication qui peut aider la lecture et la compréhension de votre travail). Les noms des fonctions et autres identificateurs doivent également permettre de faciliter la lecture.

Le projet doit être écrit en utilisant la portion de Ocaml étudiée en cours. Vous pouvez, sans les redéfinir, utiliser les fonctions prédéfinies sur les listes `mem`, `map`, `fold_left` et `fold_right`, `find`, `find_all`, `exists` et `for_all`.

Il sera apprécié que vous utilisiez des fonctions d'ordre supérieur dès que cela est pertinent.

La qualité et l'efficacité (éviter de parcourir plusieurs fois les mêmes structures si ce n'est pas nécessaire) de votre programme seront pris en compte dans la notation.

Le projet est individuel. Toute ressemblance suspecte entre projets sera sanctionnée.

Présentation du projet

On considère un puzzle carré de taille nxn dont une seule pièce peut être déplacée et uniquement en échangeant sa place avec une pièce adjacente. Chaque pièce est numérotée de 0 à n^2-1 et la pièce 0 correspond à la pièce mobile.

0	1	2
3	4	5
6	7	8

Figure 1: Exemple de puzzle

La pièce 0 peut donc se déplacer dans 4 directions au maximum : Droite, Bas, Gauche, Haut. La direction Droite signifiant que la pièce 0 échange sa place avec la pièce directement à sa droite.

1	0	2
3	4	5
6	7	8

Figure 2: Exemple après mouvement Droite

Si la pièce 0 se situe dans un coin, ses mouvements seront contraints. Dans le cas de la Figue 1 la pièce 0 ne peut se déplacer qu'à Droite ou en Bas.

L'objectif du puzzle est de retrouver la configuration finale (Figure 1) à partir d'une configuration aléatoire et en utilisant uniquement les mouvements de la pièce 0.

Un puzzle sera représenté par un couple (grille, taille). On appellera grille une liste de pièces. (On peut traduire la configuration finale par une liste triée de 0 à n^2-1). La représentation en Caml de la figure 1 est la suivante :

```
# let puzzle1 = ([0;1;2;3;4;5;6;7;8], 3) ;;
val puzzle : int list * int = ([0; 1; 2; 3; 4; 5; 6; 7; 8], 3)
```

Dans ce projet vous devrez représenter, mélanger et résoudre un puzzle.

Exercice 1 : Fonctions préliminaires

1. Afin d'interagir avec une grille, écrivez les fonctions suivantes :
   a. - `position grille x` qui renvoie la position de la pièce de valeur x dans la grille
   b. - `valeur grille p` qui renvoie la valeur présente à la position p de la grille
   c. - `echange grille v1 v2` qui renvoie une grille dont les valeurs v1 et v2 ont été échangées (on suppose que dans une grille chaque valeur n'apparaît qu'une seule fois)

```
# position [1;4;2;3;0;5;6;7;8] 0;;
- : int = 4
# valeur [1;4;2;3;0;5;6;7;8] 4;;
- : int = 0
# echange [0;1;2;3;4;5;6;7;8] 0 4;;
- : int list = [4; 1; 2; 3; 0; 5; 6; 7; 8]
```

2. Afin de pouvoir déplacer la pièce 0, écrivez la fonction `(deplacer puzzle direction)` qui renvoie une grille dont la pièce zéro s'est déplacée dans la direction indiquée. Si le déplacement n'est pas possible on renvoie la grille inchangée. Vous devrez définir un type énuméré `direction` pouvant prendre les valeurs `Droite`, `Bas`, `Gauche`, et `Haut`

Attention une grille est représentée par une liste mais il faut prendre en compte que la pièce se déplace en réalité dans un tableau à deux dimensions. On pourra utiliser les fonctions précédentes pour réaliser cette fonction.

```
# deplacer ([1;2;3;4;0;5;6;7;8],3) Droite;;
- : int list = [1; 2; 3; 4; 5; 0; 6; 7; 8]
# deplacer ([1;2;3;4;0;5;6;7;8],3) Bas;;
- : int list = [1; 2; 3; 4; 7; 5; 6; 0; 8]
# deplacer ([1;2;3;4;0;5;6;7;8],3) Gauche;;
- : int list = [1; 2; 3; 0; 4; 5; 6; 7; 8]
# deplacer ([1;2;3;4;0;5;6;7;8],3) Haut;;
- : int list = [1; 0; 3; 4; 2; 5; 6; 7; 8]
# deplacer ([0;1;2;3;4;5;6;7;8],3) Gauche;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8]
```

Exercice 2 : Mélange d'un puzzle

3. Écrire une fonction `(melange puzzle iterations)` qui effectue un nombre de mouvements aléatoires correspondant au nombre d'itérations. Le nombre de mouvements correspond au nombre de mouvements vraiment réalisés (un mouvement qui n'est pas possible n'est pas comptabilisé). Les mouvements seront tirés au hasard[1]. La fonction renvoie une grille résultant des mouvements réalisés.

```
# melange ([0;1;2;3;4;5;6;7;8],3) 100;;
- : int list = [5; 1; 4; 6; 0; 7; 8; 2; 3]
```

4. Écrire une fonction (testerChemin puzzle chemin) qui renvoie la grille obtenue à partir d'un puzzle en effectuant le chemin proposé. Le chemin est une liste de directions dont le premier élément est la dernière direction empruntée et le dernier élément est la première direction empruntée.

```
# testerChemin ([0;1;2;3;4;5;6;7;8],3) [Gauche;Bas;Droite];;
- : int list = [1; 4; 2; 0; 3 ;5 ; 6; 7; 8]
```

1. On pourra pour cela utiliser la fonction `Random.int : int → int`.
`Random.int b` retourne un entier entre `0` et `(b-1)`.

Exercice 3 : Résolution

Pour trouver un chemin permettant d'obtenir le puzzle final à partir d'une grille aléatoire, vous testerez 3 parcours différents. Dans un premier temps vous utiliserez un parcours naïf de toutes les grilles. Le parcours naïf se déroule ainsi :

- On dispose du puzzle final (que l'on cherche à obtenir) et d'une liste des prochaines grilles à parcourir. La liste des prochaines grilles est composée de grilles ainsi que le chemin permettant d'y accéder à partir de la grille initiale. Lors du premier appel de la fonction la liste des prochaines grilles contient la grille initiale.
- Tant que la grille en tête de la liste des prochaine grilles n'est pas la grille du puzzle final
  - On calcule les grilles pouvant être atteintes avec les différents déplacements à partir de la grille en tête de la liste des prochaines grilles
  - On ajoute à la fin de la liste des prochaines grilles, les grilles obtenues avec les différents déplacements et le chemin permettant d'y accéder à partir de la grille initiale
- On retourne le chemin permettant d'obtenir la grille finale

Vous devrez écrire une fonction (`resoudre grille_init grille_finale taille parcours`) qui renvoie le chemin obtenu avec le parcours proposé (l'argument parcours prendra une valeur entre 1 et 3 dont la valeur 1 correspond au parcours naïf).

5. Écrire une fonction (`parcours1 grille_finale taille prochaines_grilles`) qui renvoie le chemin obtenu à l'issue du parcours naïf. En déduire la définition de la fonction resoudre pour parcours1.

```
# resoudre [3;1;2;4;7;5;0;6;8] [0;1;2;3;4;5;6;7;8] 3 1 ;;
- : direction list = [Haut; Gauche; Haut; Droite]
# resoudre [6;3;1;4;0;2;7;8;5] [0;1;2;3;4;5;6;7;8] 3 1 ;;
- : direction list = [Haut;Haut;Gauche;Gauche;Bas;Bas;Droite;Droite;Haut;Gauche]
# resoudre [1;4;3;6;2;8;7;5;0] [0;1;2;3;4;5;6;7;8] 3 1 ;;
- : direction list = [Haut;Gauche;Bas;Droite;Haut;Gauche;Gauche;Bas;Droite;
Droite;Haut;Haut;Gauche;Bas;Gauche;Haut]
```

Ces exemples sont présents uniquement à titre indicatif. Les résultats obtenus peuvent être légèrement différents. Pensez à tester vos fonctions avec plusieurs grands mélanges (plus de 1000 mouvements par exemple) et de vérifier avec testerChemin que les chemins obtenus mènent bien à une solution.

Le parcours naïf peut être long pour certains cas (comme le troisième exemple) car on parcourt plusieurs fois les mêmes grilles, pour améliorer cette fonction on peut mémoriser les grilles déjà parcourues.

6. Écrire une fonction (`parcours2 grille_finale taille prochaines_grilles grilles_parcourues`) qui renvoie le chemin obtenu avec le parcours naïf en mémorisant les grilles déjà parcourues. Modifiez la définition de la fonction resoudre pour intégrer parcours 2.

```
# resoudre [3;1;2;4;7;5;0;6;8] [0;1;2;3;4;5;6;7;8] 3 2 ;;
- : direction list = [Haut; Gauche; Haut; Droite]
# resoudre [6;3;1;4;0;2;7;8;5] [0;1;2;3;4;5;6;7;8] 3 2 ;;
- : direction list = [Haut;Haut;Gauche;Gauche;Bas;Bas;Droite;Droite;Haut;Gauche]
# resoudre [1;4;3;6;2;8;7;5;0] [0;1;2;3;4;5;6;7;8] 3 2 ;;
- : direction list = [Haut;Gauche;Bas;Droite;Haut;Gauche;Gauche;Bas;Droite;
Droite;Haut;Haut;Gauche;Bas;Gauche;Haut]
# resoudre [10;12;4;5;9;14;8;3;0;11;6;15;13;7;2;1]
[0;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15] 4 2;;
- : Pas de résultat après 1h de recherche.
```

Le parcours 2 doit permettre de résoudre toutes les grilles de taille 3 avec un temps plus court que le parcours 1. Testez votre fonction avec des grilles aléatoires. Les grilles de taille 4 restent généralement très longues à résoudre avec cette méthode.

Pour finir on souhaite améliorer la recherche du chemin en choisissant plus intelligemment les prochaines grilles à parcourir et réduire le nombre de grilles parcourues. Pour cela nous ajoutons une notion de distance entre deux grilles qui calcule le nombre de déplacements nécessaires pour replacer chaque pièce à sa place.

7. Écrire une fonction (`distance grille_initiale grille_finale taille`) qui calcule à partir d'une grille et de sa taille sa distance par rapport à la grille finale. La distance entre deux grilles se calcule en comptant le nombre de déplacements nécessaires pour que chaque valeur se retrouve à sa position finale.

Exemple : Soit la grille suivante :

4	3	2
1	0	5
6	7	8

La distance est de 8 par rapport à la grille finale car il faut 2 déplacements pour que 4 retrouve sa position finale, 2 déplacements pour la pièce 3, 0 déplacement pour la pièce 2, 2 déplacements pour la pièce 1 et 2 déplacements pour la pièce 0. Les pièces 5 à 8 sont déjà placées donc il faut 0 déplacement pour qu'elles retrouvent leur position finale.

```
# distance [4; 3; 2; 1; 0; 5; 6; 7; 8] [0; 1; 2; 3; 4; 5; 6; 7; 8] 3;;
- : int = 8
```

8. Écrire une fonction (`parcours3 grille_finale taille prochaines_grilles grilles_parcourues`) qui renvoie un chemin permettant de retrouver la grille_finale en utilisant la fonction de distance précédente. prochaines_grilles est une liste triée par ordre croissant sur la distance composée de triplets (grille,chemin,distance). Modifiez la définition de la fonction resoudre pour intégrer parcours3.

```
# resoudre [3;1;2;4;7;5;0;6;8] [0;1;2;3;4;5;6;7;8] 3 3 ;;
- : direction list = [Haut; Gauche; Haut; Droite]
# resoudre [6;3;1;4;0;2;7;8;5] [0;1;2;3;4;5;6;7;8] 3 3 ;;
- : direction list = [Haut;Haut;Gauche;Gauche;Bas;Bas;Droite;Droite;Haut;Gauche]
# resoudre [1;4;3;6;2;8;7;5;0] [0;1;2;3;4;5;6;7;8] 3 3 ;;
- : direction list = [Haut;Haut;Gauche;Gauche;Bas;Droite;Haut;Droite;Bas;Bas;
Gauche;Haut;Gauche;Bas;Droite;Droite;Haut;Gauche;Gauche;Bas;Droite;Haut;
Droite;Bas;Gauche;Gauche;Haut;Droite;Droite;Bas;Gauche;Haut;Gauche;Bas;
Droite;Haut;Droite;Haut;Gauche;Gauche]
# resoudre [10;12;4;5;9;14;8;3;0;11;6;15;13;7;2;1]
[0;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15] 4 3;;
- : direction list = [Haut;Haut;Gauche;Gauche;Bas;Droite;Haut;Droite;Bas;Bas;
Gauche;Haut;Gauche;Bas;Droite;Droite;Haut;Gauche;Gauche;Bas;Droite;Haut;
Droite;Bas;Gauche;Gauche;Gauche;Haut;Droite;Haut;Gauche;Bas;Bas;Droite;
Haut;Haut;Gauche;Bas;Droite;Droite;Bas;Droite;Haut;Gauche;Gauche;Bas;
Droite;Haut;Droite;Bas;Gauche;Haut;Gauche;Haut;Droite;Haut;Gauche;Bas;Bas;
Droite;Haut;Gauche;Haut;Droite;Bas;Gauche;Haut;Droite;Bas;Bas;Droite;Haut;
Gauche;Haut;Droite;Bas;Bas;Gauche;Haut;Droite;Haut;Gauche;Gauche;Bas;
Droite;Bas;Gauche;Haut;Gauche;Bas;Droite;Droite;Haut;Gauche;Haut;Droite;
Droite;Bas;Gauche;Bas;Droite;Haut;Haut;Gauche;Gauche;Bas;Droite;Haut;
Droite;Bas;Bas;Gauche;Haut;Droite;Haut;Gauche;Bas;Gauche;Haut;Droite;Bas;
Droite;Haut;Gauche;Bas;Bas;Bas;Droite;Haut;Haut;Gauche;Gauche;Gauche;Bas;
Droite;Droite;Haut;Gauche;Bas;Bas;Droite;Droite;Haut;Haut;Haut;Gauche;
Gauche;Bas;Bas;Gauche;Haut;Haut;Droite;Bas;Bas;Gauche;Bas;Droite;Droite;
Droite;Haut;Haut]
```

Le parcours 3 doit permettre de résoudre toutes les grilles de taille 4 en un temps raisonnable mais avec un chemin pas toujours optimal.