**L2-MATH et L2-INFO**
**Autre Paradigme 2021-2022**

# Devoir Maison

**Le jeu du taquin est un puzzle** créé vers 1870. Depuis, il a attiré l'intérêt de nombreux mathématiciens et informaticiens pour sa valeur en tant que problème combinatoire. Le jeu est composé de petits carrés numérotés à partir de 1 qui glissent dans un cadre du format n×m laissant une case vide permettant de modifier la configuration des carrés. Le jeu consiste à remettre dans l'ordre ces carrés à partir d'une configuration initiale quelconque. Le jeu est souvent connu dans les formats 3×3 ou 4×4.

Dans ce DM, on s'intéresse un **uniquement au format 3×3.**

Pour en savoir plus au sujet du taquin :

- le cas (3x3) sur lequel porte le DM : https://rand-asswad.xyz/taquin/
- le cas (4x4) (juste pour info) : https://fr.wikipedia.org/wiki/Taquin

**L'objectif du DM** est, en utilisant l'exemple du taquin, de comparer 3 types de parcours d'un arbre de recherche, étudiés lors du CM 08 :

- Parcours en largeur d'abord
- Parcours en profondeur d'abord
- Parcours en meilleur d'abord

**Le DM comporte 3 parties :**

1. Représentation Haskell d'une grille et fonctions utilitaires
2. Mise en œuvre des 3 parcours : largeur, profondeur et meilleur d'abord,
   - en montrant l'architecture commune des 3 fonctions
   - et en précisant la gestion des états à visiter : file, pile et file à priorité
3. Recherche du chemin allant d'un état initial à l'état final

Dans ce document, vous trouverez l'énoncé de la Partie 1. A venir la partie 2 et la partie 3.

---

## Modalités pratiques

Date limite de remise : **Mercredi 27/04/2022**

Modalités de remise à venir

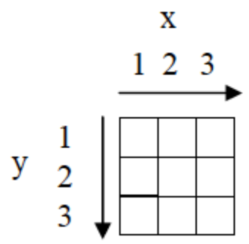
A faire seul ou en binôme (de préférence). Aucun groupe de 3 ou plus.

---

## Partie 1. Représentation d’une grille et fonctions utilitaires

### 1.1 Représentation d’un état

Pour un état (grille) donné, on associe à chaque case un couple de coordonnées (x, y). La case vide sera implicitement associée au carré (virtuel) de numéro 0. Les 8 autres carrés seront numérotés de 1 à 8.




Numérotation des cases

| | | |
|---|---|---|
| 1 | 5 | 3 |
| 8 | | 2 |
| 7 | 4 | 6 |

un état **e**

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

l’état final **ef**

La position d’un carré i dans un état e est un couple (x_i, y_i) tel que le carré i occupe la case de coordonnées (x_i, y_i). Exemples :
Dans l’état e,

- La case vide (carré #0) est en position (2, 2)
- Le carré #1 est en position (1, 1)
- Le carré #2 est en position (3, 2).
- …/...
- Le carré #8 est en position (1, 2)

Un état est représenté par la liste *ordonnée* des positions de ses 9 carrés. L’élément de rang i donne la position du carré numéro i.
Ainsi, l’état e est représenté par la liste suivante :

e = [ (2, 2), (1, 1), (3, 2), (3, 1), (2, 3), (2, 1), (3, 3), (1, 3), (1, 2) ]

et l’état final ef est représenté par :

ef = [(2, 2), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (2, 3), (1 ,3), (1, 2)]

### Codage HASKELL

On défnit les types synonymes :

```
type TilePos = (Int, Int)
type State = [TilePos]
```

***Questions.***

Définir les fonctions suivantes :

a) (whichTileAt pos s) qui retourne le numéro du carré en position pos dans l’état s

b) (posTile i s) qui retourne la position du carré numéro i dans s

c) (posEmpty s) qui retourne la position de la case vide dans l’état s

Exemples :

```
> whichTileAt (2, 2) e    ==> 0
> whichTileAt (1, 3) e    ==> 7
> posTile 6 e    ==> (3, 3)
> posTile 8 e    ==> (1, 2)
> posEmpty e    ==> (2, 2)
```

### 1.2 Visualisation d'un état et d'une succession d'états

On considère la fonction (toString e) ci-dessous qui transforme un état e en une chaîne de caractères.

```
toString :: State -> String
toString s = row 1 s ++ row 2 s ++ row 3 s ++ "\n"

row :: Int -> State -> String
row n s = " " ++ t 1 ++ " " ++ t 2 ++ " " ++ t 3 ++ "\n"
  where t m = show (whichTileAt (m, n) s)

> (toString e)    ==> " 1 2 3\n 8 0 5\n 7 4 6\n\n"
> (toString ef)   ==> " 1 2 3\n 8 0 4\n 7 6 5\n\n"
```

On peut alors visualiser un état comme suit :

```
> putStr (toString e)
1 2 3
8 0 5
7 4 6

> putStr (toString ef)
1 2 3
8 0 4
7 6 5
```

On considère désormais la fonction (toStr es) définie ci-dessous qui transforme une liste d'états es en une chaîne de caractères.

```
toStr :: [State] -> String
toStr []  = ""
toStr (x : xs) = (toString x) ++ (toStr xs)
```

On peut alors visualiser une liste d'états comme suit :

```
> toStr [e, ef]   ==> " 1 2 3\n 8 0 5\n 7 4 6\n\n 1 2 3\n 8 0 4\n 7 6 5\n\n"
> putStr (toStr [e, ef])
1 2 3
8 0 5
7 4 6

1 2 3
8 0 4
7 6 5

>
```

***Question.*** En utilisant les fonctions concat et map, proposer une 2nde définition de la fonction (toStr es)

### 1.3 Distances entre cases

***Questions.***

a) Définir la fonction (distHamming pos1 pos2) qui calcule la distance de Hamming entre 2 cases définies par leurs coordonnées respectives.

b) Définir la fonction (distManhattan pos1 pos2) qui calcule la distance de Manhattan entre 2 cases définies par leurs coordonnées respectives.

Exemples :

```
> distHamming (1, 1) (1, 3)   ==> 1
> distHamming (1, 1) (1, 1)   ==> 0

> distManhattan (1, 2) (2, 3)   ==> 2
> distManhattan (3, 3) (1, 2)   ==> 3
```

### 1.4 Fonctions heuristiques h1 et h2

Pour estimer le coût pour aller d'un état courant s à l'état final ef, on considère, deux fonction heuristiques :

- la fonction h1, somme des distances de MANHATTAN entre la position de chaque carré[1] dans s et sa place finale dans ef.
- la fonction h2, somme des distances de HAMMING entre la position de chaque carré[1] dans s et sa place finale dans ef.

```
> h1 e    ==> 4                    > h2 e   ==> 3
> h1 ef    ==> 0                   > h2 ef    ==> 0

s = [(2,3), (1,2), (1,1), (3,1), (3,2), (3,3), (2,2), (1,3), (2,1)]

> putStr (toString s)
  2 8 3
  1 6 4
  7 0 5

> h1 s    ==> 5                    > h2 s   ==> 4
```

***Question.*** Définir les fonctions h1 et h2

### 1.5 Fonction (sucesseurs s)

La fonction (successeurs s) associe à un état s la liste de ses successeurs.

***Question***. Compléter la définition ci-dessous :

```
successeurs :: State -> [State]

successeurs s = [swap i s | i <- [1 .. (length s) - 1],  valide i s]
        where
                valide i s =
                swap i s =
```

```
> putStr (toString e)
 1 2 3
 8 0 5
 7 4 6

> putStr (toStr (successeurs e))
 1 0 3
 8 2 5
 7 4 6

 1 2 3
 8 4 5
 7 0 6

 1 2 3
 8 5 0
 7 4 6

 1 2 3
 0 8 5
 7 4 6
```

[1] *NB. Dans cette définition,* ***on ne fera pas intervenir la case vide,*** *mais seulement les carrés de 1 à 8.*

***Indications.*** *Rappel de l'état e :*

*1 2 3*
*8 0 5*
*7 4 6*

*- la fonction (valide i s) détermine si le carré i peut être échangé avec le carré 0 dans l'état s. Ainsi, pour l'état e, on peut échanger le carré #0 avec les carrés 2, 4, 5 et 8.*

```
-- exemple
   > [i | i <- [1 .. (length e) - 1], valide i e]   ==> [2, 4, 5, 8]
```

*- la fonction (swap i s) échange le carré i avec le carré 0 dans l'état s.*

```
-- exemples
> swap 2 e   ==> [(2,1), (1,1), (2,2), (3,1), (2,3), (3,2), (3,3), (1,3), (1,2)]
> swap 4 e   ==> [(2,3), (1,1), (2,1), (3,1), (2,2), (3,2), (3,3), (1,3), (1,2)]
> swap 5 e   ==> [(3,2), (1,1), (2,1), (3,1), (2,3), (2,2), (3,3), (1,3), (1,2)]
> swap 8 e   ==> [(1,2), (1,1), (2,1), (3,1), (2,3), (3,2), (3,3), (1,3), (2,2)]
```

**NB.**

Pour la suite du DM, vous êtes obligatoirement dans l'une des 2 situations suivantes :

**A.** Vous avez réussi à définir la fonction (successeur s) tel que demandé au début de cette section, alors utilisez votre propre définition (Situation A). Pour tester votre définition, vous pouvez utiliser la fonction (test s)

```
-- Situation (A) utilisez votre propre définition de la fonction (successeurs s)
-- vous pouvez tester votre définition à l'aide du prédicat (test s)

-- successeurs s = ../...

test :: State -> Bool
test s = (successeurs s) == (successeursBis s)
```

**B.** Vous n'avez pas réussi à définir la fonction (successeur s) tel que demandé au début de cette section ; dans ce cas, prenez comme définition la fonction (successeursBis s) ci-dessous (Situation B). Cela vous permettra de continuer le DM et de pouvoir tester les 2 parties à venir.

```
-- Situation (B) utilisez la définition ci-dessous qui vous permettra de continuer le DM

successeurs = successeursBis

successeursBis :: State -> [State]
successeursBis (e:ts) = inter [e] ts
   where inter (e:ts) [] = []
         inter (e:ts1) (t:ts2)
           | distManhattan e t == 1 = (t:ts1 ++ e:ts2) : (inter (e:ts1++[t]) ts2)
           | otherwise = inter (e:ts1++[t]) ts2
```