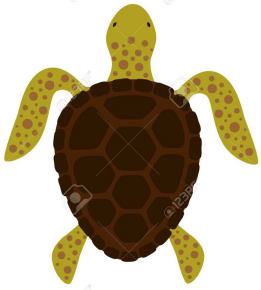
Turtle-rescue

1 Présentation du projet

L'objectif de ce projet est la **réalisation** d'un jeu et de sa **résolution** de manière automatisée.

Le but du jet est d'amener au travers d'un labyrinthe un bébé tortue tout juste sorti de son œuf à sa cachette. La difficulté réside dans le fait que lorsque l'on décide de déplacer le personnage vers le haut, celui-ci continue jusqu'à rencontrer un obstacle (mur ou autre animal). Heureusement pour aider la tortue, 3 compagnons pourront se déplacer selon mes mêmes règles de manière à pouvoir l'aider à trouver son chemin.



0 : Tortue

1 : Coccinelle

2 : Lapin

3 : Escargot

Ce jeu est inspiré du jeu de société Ricochet Robot.

2 Codage du terrain

Le terrain est constitué d'une grille de 16×16 cases pouvant comporter des murs. Chaque case est représentée par un entier codé sur 4 bits indiquant la présence (1) ou l'absence (0) de mur :

GAUCHE	BAS	DROITE	HAUT

Ainsi cette case : ⌜ sera codée par le nombre binaire $\overline{1001}^2$ soit 9.

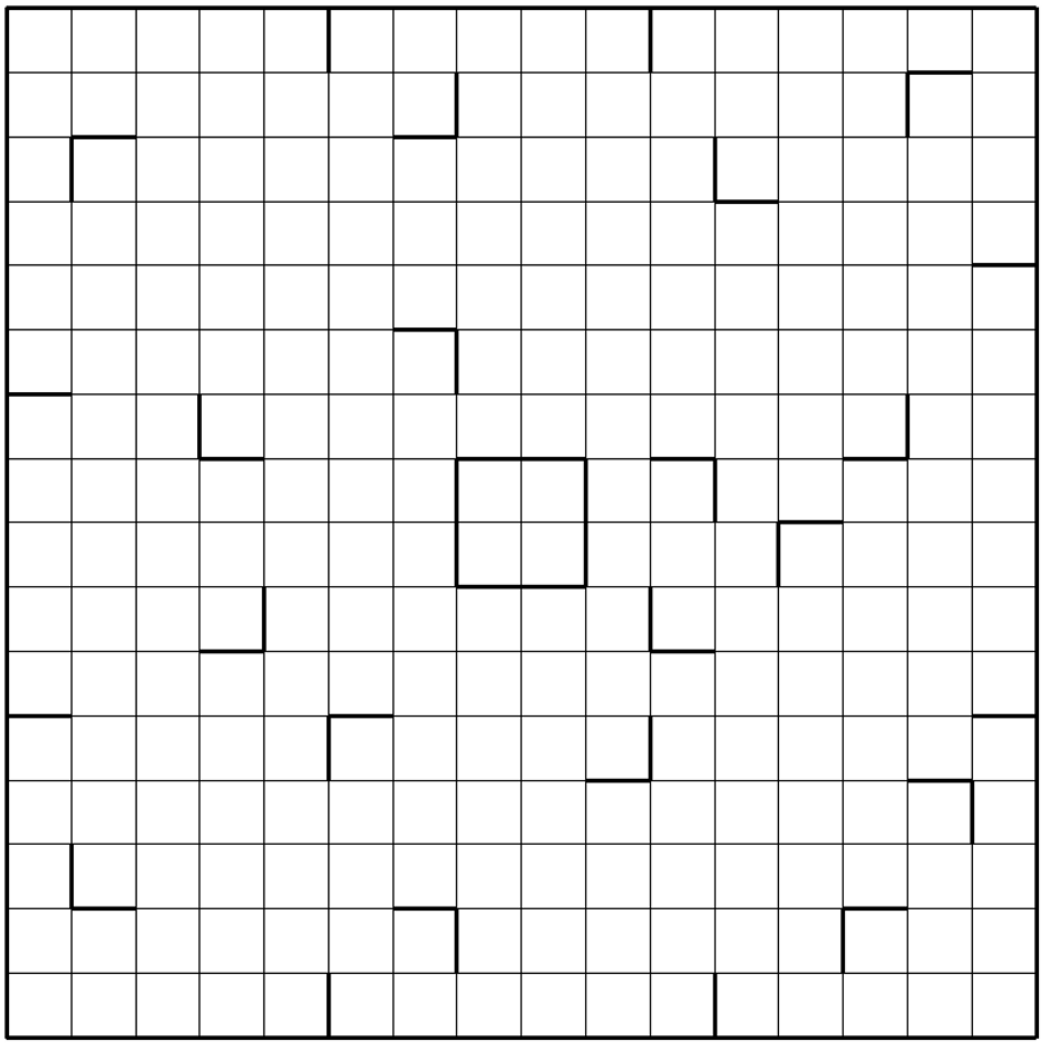
Pour trouver le code d'une case, on peut voir le problème autrement :

- le mur du haut est codé $\overline{0001}^2 = 1$
- celui de droite : ....
- celui du bas : ....
- celui de gauche : ....

	1	
...	CASE	...
	...	

Il suffit alors d'ajouter les "valeurs" des murs pour déterminer le code d'une case.

► Voici le plateau du jeu. Indiquer dans chaque case le code correspondant.



Le terrain sera codé par une liste de 256 valeurs contenant les codes de chaque case.

► Les cases seront donc indicées par des entiers de 0 (case en haut à gauche) à 255 (case en bas à droite). Combien de bits sont nécessaires pour coder l'indice d'une case ? On dit alors que l'information est codée sur un octet. **1 octet = ... bits**.

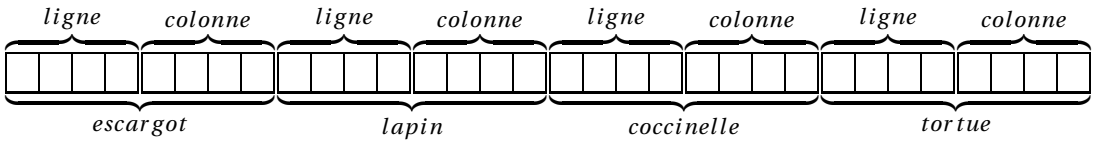
► Ecrire une fonction `lignecolonne` recevant l'indice d'une case et retournant un couple indiquant la ligne et la colonne de cette case. Par exemple `lignecolonne(37)` retourne `(2,5)`

3 Notion de `situation`

► Ecrire une fonction `dec2bin` recevant deux entiers `n` et `l` et retournant l'écriture de l'entier `n` codé sur `l` bits. Par exemple `dec2bin(9,6)` retourne `[0,0,1,0,0,1]`

► Ecrire la fonction `bin2dec` réalisant l'opération inverse. `bin2dec([0,1,0,1])` renvoie `5`.

Dans ce projet, on remarque qu'une situation de jeu est entièrement déterminée par les positions des 4 personnages. Chaque position étant codée sur 1 octet, on peut donc coder la situation d'une partie sur 4 octets (on dit que l'on code sur un **mot double** ou parfois **mot long**).



► Sur un mot double, on peut donc un entier naturel entre ... et ...

► Compléter le tableau correspondant à la situation 599 872 009 qui pourra vous aider à vérifier vos fonctions.

animal	case	ligne	colonne
escargot			
lapin			
coccinelle			
tortue			

► Ecrire une fonction `case` telle que `case(s,j)` retourne l'indice de la case du joueur numéro `j` pour une situation `s`.

► Ecrire une fonction `gagne` telle que `gagne(s,but)` retourne un booléen indiquant pour la situation `s` et l'indice de l'emplacement `but` de la cachette, si la partie est gagnée.

Pour la suite, on décide coder de la même manière que les murs les déplacements :

1 pour le haut, 2 à droite, 4 en bas et 8 à gauche.

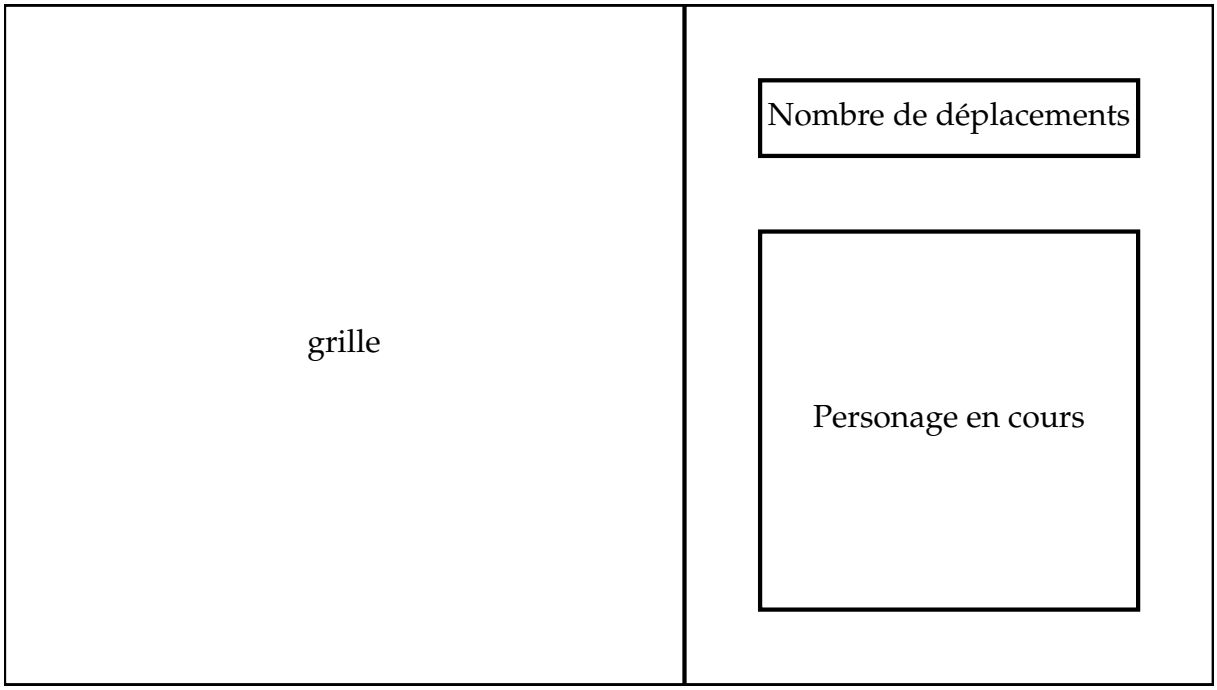
► Ecrire une fonction `possible` telle que `possible(s,j,d)` retourne un booléen indiquant si `j` à partir d'une situation `s` peut aller dans la direction `d`.

► Ecrire une fonction `deplacements` telle que `deplacements(s,j)` retourne la liste des directions possibles pour le joueur `j` à partir d'une situation `s`.

► Ecrire une fonction `nouvelle` telle que `nouvelle(s,j,d)` retourne la nouvelle situation sous forme d'un entier si le joueur `j`, à partir d'une situation `s`, se déplace dans la direction `d`.

4 L'interface graphique

Pour la programmation de l'interface graphique, on va utiliser le module `Tkinter` disponible dans toutes les distributions Python. Reportez-vous au petit mémo pour découvrir les fonctions disponibles. On propose la configuration d'écran suivante de 1600×900 pixels. Le dessin du terrain étant déjà disponible sous forme d'une image, composée de 256 cases de forme carrées de 54 pixels de côté chacun.



► Écrire un programme plantant la position initiale du décor. Pour afficher le nombre de déplacements, nous aurons besoin de la méthode `create_text` sur le canvas :

```
T = fond.create_text(250,120, text = 'Coucou', font = ('Times',16), fill='#FFFF00')
```

Cette ligne écrit en jaune (couleur #FFFF00) sur le canvas `fond` *le texte « coucou » centré autour du point de coordonnées* $(250, 120)$ *avec la police Times de taille 16. L'objet texte est placé dans une variable* `T` *permettant de le modifier par la suite.*

.

► Ajouter la possibilité de pouvoir changer de personnage avec les touches 0, 1, 2 et 3. Pour cela, on aura recours à la méthode `itemconfig` sur le canvas qui permet de modifier un paramètre d'un item du canvas :

```
fond.itemconfig(T, text="hello")

img1 = PhotoImage(file='image1.gif') # on place les images dans des variables GLOBALES.
img2 = PhotoImage(file='image2.gif')
Im = fond.create_image(100, 100, image=img1, anchor='nw')
fond.itemconfig(Im, image = img2) # Modification de l'image.
```

La première ligne permet de modifier le contenu du texte `T` *définit dans l'exemple précédent, la dernière permet de changer l'image représentée par l'objet* `Im`*.*

.

Attention : les images doivent être des variables GLOBALES !

► Écrire une fonction `place` telle que `place(s)` place les personnages à l'endroit indiqué par la situation `s`.

► Terminer la gestion des touches directionnelles pour obtenir un jeu abouti.

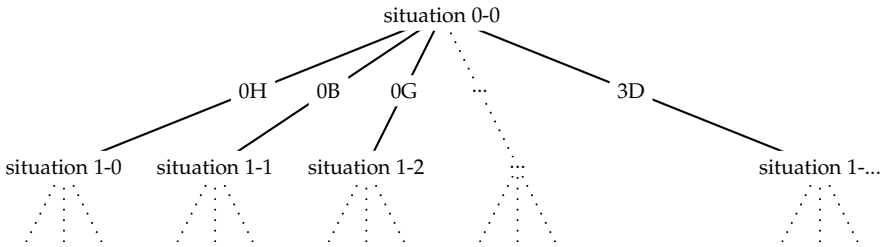
5 Solveur de jeu

A présent que toutes les fonctions ont été mises en place, intéressons-nous à la résolution automatique du jeu... Pour cela nous allons avoir besoin d'un certain nombre de variables, voici une proposition :

- `nbC` : un entier indiquant le nombre de coups joués.
- `ATraiter` : une liste contenant l'ensemble des situations que l'on peut atteindre en `nbC` coups.
- `Deja` : une liste contenant les situations déjà visitées.
- `Depl` : une liste indiquant le chemin à prendre : plus précisément, `Depl[i]` indique (par une chaine de caractères par exemple "0H1B0D") comment se rendre à `Atraiter[i]` à partir de la situation initiale.

▷ Quel est l'intérêt de conserver en mémoire les situations déjà visitées ?

Nous pouvons visualiser les déplacements possibles sous forme d'un arbre.



▷ Combien au maximum, une situation donnée peut-elle avoir de fils ? Combien de situations possibles cela fait-il à explorer au bout de 4 étapes ?

▷ Regarder sur un cours d'informatique ce que l'on appelle un **parcours en profondeur** d'un arbre et un **parcours en largeur**. Lequel vous parait le plus pertinent pour gérer notre problème ?

► Programmer ce parcours.

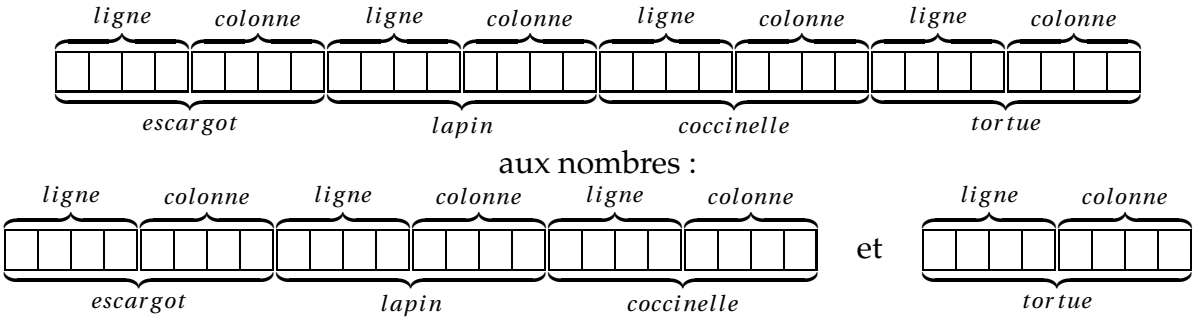
```
TANT QUE Atrairer n'est pas vide et que l'on a pas gagné
    Récupérer la situation s la plus ancienne de Atraiter
    Récupérer le chemin ch correspondant dans Depl
    Supprimer s et ch de leurs listes respectives.
    POUR tous les déplacements d possibles à partir de s
        Déterminer la nouvelle situation s' lorsque l'on effectue le déplacement d
        SI s' n'a pas déjà été visitée ALORS
            L'ajouter à Atraiter
            Ajouter le déplacement réalisé dans Depl
        FINSI
    FIN POUR
FIN TANT QUE
Répondre au problème.
```

6 Optimisation

Une première optimisation peut consister à **limiter les changements de type**. Par exemple : passer d'un nombre en liste binaire, puis inversement.

On va chercher à optimiser les calculs de la fonction position.

◇ Idée 1 : Quelle opération mathématique permet de passer du nombre :



Utiliser cette remarque pour optimiser votre fonction `position`

◇ Idée 2 : on va utiliser des **shifts** et des **masques binaires** :

- Un **shift** permet de décaler les bits d'un nombre vers la gauche ou vers la droite. En Python on effectue un shift via la commande << (shift gauche) ou >> (shift droit) : `14 » 2` retourne `3`, car $14 = \overline{1110}^2$, on obtient donc $3 = \overline{11}^2$.
- Les **masques binaires** permettent de faire des opérations bits à bits très rapidement, on dispose des opérations ET (**`and`**), OU (**`or`**) et XOU (ou exclusif) dont les tables sont données ci-dessous.

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	1

Utiliser cette nouvelle technique pour faire de nouveaux tests de rapidité.

► Utiliser la même idée pour optimiser `case`

TO BE CONTINUED