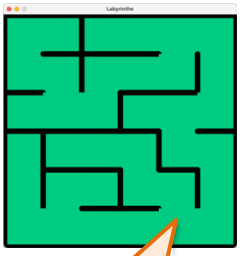
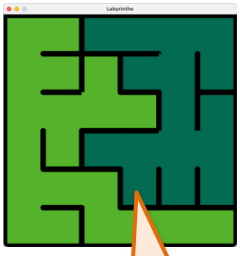
OBJECTIFS

- *Effectuer un mini projet qui regroupe un grand nombre de notion de Python mais SANS programmation Objets…*
- *Coder des éléments graphiques de base : création d'un labyrinthe.*
- *Approfondir l'utilisation des listes.*

INTRODUCTION

En mathématique, un labyrinthe est par définition, une grille multidimensionnelle de cases connexes d'un seul tenant. Il peut être parfait ou non. S'il est parfait, chaque case est reliée à toutes les autres de manière unique.





Labyrinthe PARFAIT

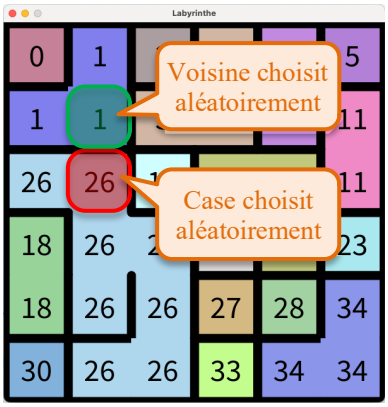
Labyrinthe NON PARFAIT

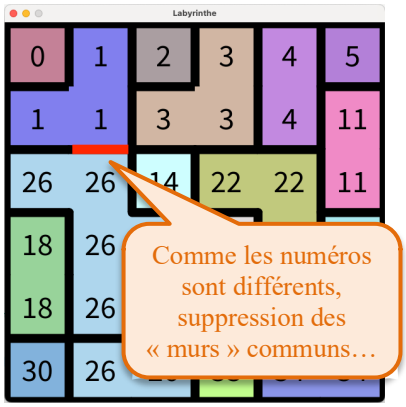
Dans le cadre de ce mini-projet, nous allons créer une grille aléatoire parfaite de dimension 2 où chaque case est un carré. Les images ci-contre illustrent la notion de labyrinthe parfait.

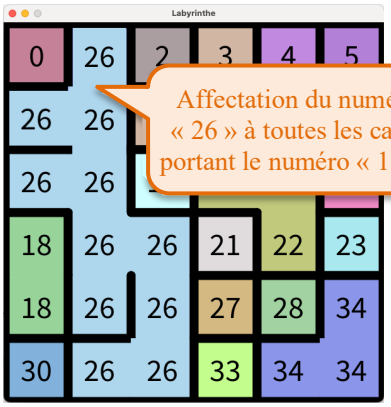
*L'algorithme pour élaborer un labyrinthe aléatoire est assez simple. On part d'une grille où **AUCUNE** case ne communique avec ses cases voisines. De ce fait toutes les cases ont quatre « murs » sur leurs quatre cotés. De plus chaque case dispose d'un **NUMERO UNIQUE**…*

On applique ensuite de manière répétitive les opérations suivantes :

- *On choisit aléatoirement une case dans la grille.*
- *On choisit toujours aléatoirement un voisin à cette case (Nord ou Est ou Sud ou Ouest).*
- *Si les deux cases voisines ne portent pas le même numéro, on les associe. Pour cela, :*
  - *On supprime les « murs » communs à ces deux cases voisines.*
  - *On affecte le numéro de la case principale à toutes les cases de la grille qui portent le numéro de la case voisine.*







ACTIVITE N°1 : CREATION DU LABYRINTHE INITIAL

Pour faire du graphisme en python, il faut mettre en œuvre un GUI (Graphic User interface) qui offre tous les outils nécessaires. Plusieurs modules de GUI existent en Python : « Tkinter » (inclus dans Python), « Kivy », « wxPython », « pygame », « PyQt5 », etc…

*A l'IUT de Colmar, le module qui a été choisi est « **PyQt5** ». Avant tout il est nécessaire d'installer ce module !*

- Lancez « PyCharm ».
- Installer le module « ***PyQt5*** », dans la console de « Pycharm » ( ) avec la commande :

```
pip install PyQt5
```

Les fonctions de base de création du GUI ne sont pas étudiées en détails mais juste données avec quelques annotations relatives à la fonction de base.

- Transférez par un copier/coller et dans un nouvel onglet de « PyCharm » le code suivant :

Importation des différents modules nécessaire à cette activité…

```
import sys, random, time

from PyQt5.QtWidgets import QApplication, QGraphicsScene, QGraphicsView
from PyQt5.QtGui import QColor, QPen, QBrush
from PyQt5.QtCore import QTimer, QRect, Qt

# Définition des constantes
DIM_LABY = 3
DIM_FENETRE = 600
DIM_CASE = (DIM_FENETRE) // DIM_LABY
NORD, EST, SUD, OUEST = 1, 2, 3, 4

# Fonctions.
```

Endroit où VOUS allez AJOUTER VOS FONCTIONS…

```
# Fonction principale pour initialiser et lancer le programme.
nb_agglomerations = 0

app = QApplication(sys.argv)

# Vue et scène graphique
view = QGraphicsView()
scene = QGraphicsScene()
view.setScene(scene)
view.setWindowTitle("Labyrinthe")

view.setGeometry(100, 100, DIM_FENETRE + 4, DIM_FENETRE + 4)
scene.setBackgroundBrush(QBrush(QColor(200, 200, 200)))
view.show()

# Initialisation du labyrinthe
initialise_laby()
print(laby)
#dessine_laby(scene, laby)
sys.exit(app.exec_())
```

Création de l'instance dans la quelle va se trouver le GUI.

Création d'une fenêtre graphique (« widget » PyQt).

Création d'une « scène » 2D qui va contenir les éléments qui seront dessinés.

Ajoute la « scène » à la « widget ».

Défintion du titre de la fenêtre.

Définition de la position et de la dimension de la fenêtre graphique.

Définition de la couleur de fond de la fenêtre graphique.

Affichage de la fenêtre graphique.

Note : Le programme ne fonctionne pas à ce stade. Il manque la fonction « `initialise_laby()` ».

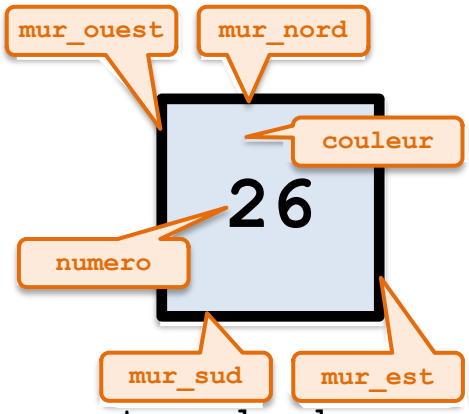
Avant de se lancer dans le codage, il faut réfléchir à la manière de représenter les données qui définissent une case mais aussi l'ensemble des cases que forme le labyrinthe…

On sait qu'au départ chaque case doit avoir un numéro et quatre « murs ». Le numéro va être stocké dans un ***entier****.*

Au fur et à mesure que le labyrinthe se construit, certains murs peuvent être « détruits ». Finalement, chaque case dispose de quatre « murs » qui peuvent exister ou non. On va donc représenter chaque mur par un ***booléen****. S'il est à « `True` », le « mur » existe et lorsqu'il a été détruit.*

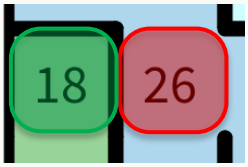
Pour des questions graphiques et pédagogiques, une couleur aléatoire (pastelle) va en plus être associée à chaque case.

L'ensemble de ces données qui caractérisent une case vont être placées dans une liste (à défaut d'objet avec votre niveau actuel en python) :




```
[numero, mur_nord, mur_est, mur_sud, mur_ouest, couleur]
 'int'    'bool'    'bool'    'bool'    'bool'    'QColor'
```

Exemples :




```
[26, False, False, False, True, 0xAED6ED]
[18, True, True, False, True, 0x98D39A]
```

Le labyrinthe va quant à lui, être formé de `DIM_LABY x DIM_LABY` *cases. Pour cela on va utiliser un tableau c'est-à-dire une liste de listes. Autrement dit, le labyrinthe est formé d'une liste de ligne et chaque ligne est formé d'une liste de cases… Cela peut s'illustrer par :*

`laby = [ligne_0, ligne_1, ligne_2, ..... , ligne_dim_laby-1]`

et `ligne_X = [case_0, case_1, case_2, ..... , case_dim_laby-1]`

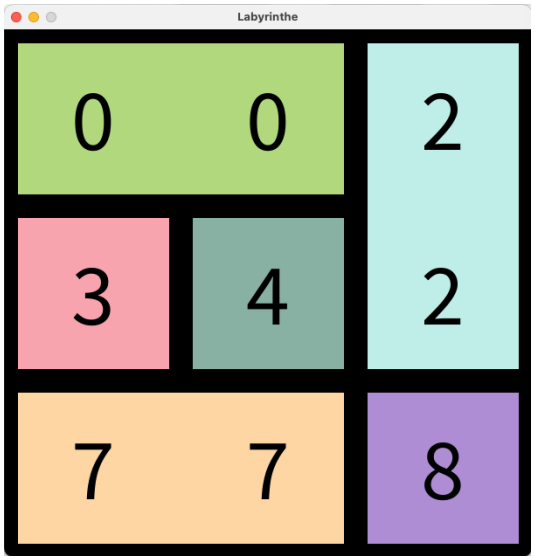
et `case_Y = [numero, mur_nord, mur_est, mur_sud, mur_ouest, couleur]`

Exemple pour un labyrinthe 3 x 3 :

```
laby = [ligne_0, ligne_1, ligne_2]
```

avec :

```
ligne_0 = [ [0, True, False, True, True, 0xB3D87E],
            [0, True, True, True, False, 0xB3D87E],
            [2, True, True, False, True, 0xBFEDE8] ]
ligne_1 = [ [3, True, True, True, True, 0xB3D87E],
            [4, True, True, True, True, 0xB3D87E],
            [2, False, True, True, True, 0xBFEDE8] ]
ligne_2 = [ [7, True, False, True, True, 0xB3D87E],
            [7, True, True, True, False, 0xB3D87E],
            [8, True, True, True, True, 0xBFEDE8] ]
```



Rappels sur l'utilisation des tableaux (liste des listes) :

Pour récupérer la liste des données des cases ROUGE et VERTE, il faut écrire :

```
case_R = laby[3][1]
case_V = laby[3][0]
```

Pour supprimer le « mur » commun aux cases ROUGE et VERTE, il faut écrire :

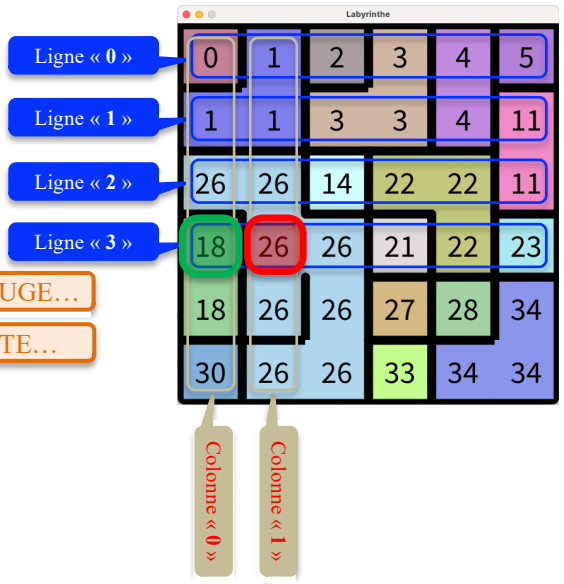
```
laby[3][1][4] = False
laby[3][0][2] = False
```

« Mur » OUEST pour la case ROUGE…

« Mur » EST pour la case VERTE…

Pour affecter le numéro de la case ROUGE au numéro de la case VERTE, on écrit :

```
laby[3][0][0] = laby[3][1][0]
```



Commençons par coder une fonction nommée « `initialise_laby()` » qui va créer la grille du labyrinthe. Cette fonction doit retourner un tableau, autrement dit une liste dont chaque élément est aussi une liste qui correspond à une ligne.

Codez la fonction « `initialise_laby()` » en utilisant la description en langage universel suivante. Cette fonction n'a aucun argument mais retourne un tableau (liste de listes de cases).

```
FONCTION initialise_laby() :
    tab_laby = []

    POUR lig AVEC 0 ≤ lig < dim_laby FAIRE :
        ligne_laby = initialise_ligne(lig)
        AJOUTER ligne_laby A LA LISTE tab_laby
    RETOURNER tab_laby
```

Note : Le programme ne fonctionne toujours pas à ce stade. Il manque la fonction « `initialise_ligne(…)` »

Codez la fonction « `initialise_ligne(…)` » en utilisant la description en langage universel suivante. Cette fonction a pour argument le numéro de la ligne qui va être créée. Elle retourne une liste dont chaque élément est une case du labyrinthe.

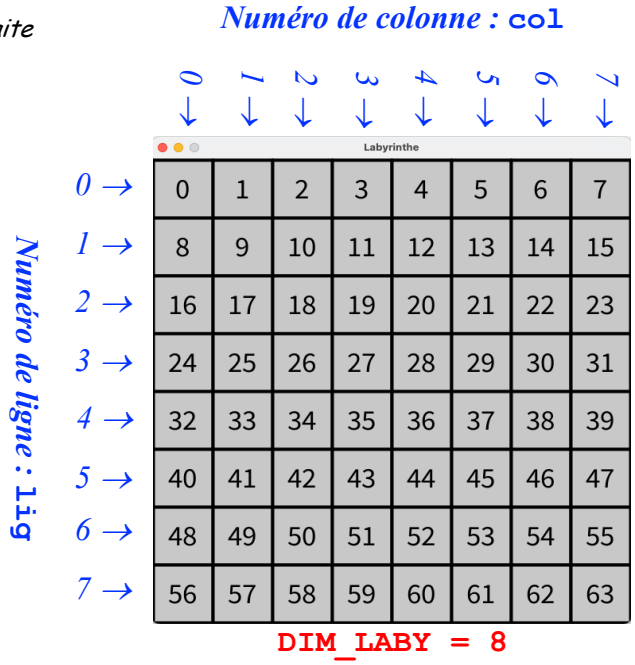
```
FONCTION initialise_ligne(lig) :
    ligne_laby = []

    POUR col AVEC 0 ≤ col < dim_laby FAIRE :
        case_laby = initialise_case(lig, col)
        AJOUTER case_laby A LA LISTE ligne_laby
    RETOURNER ligne_laby
```

Note : Le programme ne fonctionne toujours pas à ce stade. Il manque la fonction « `initialise_case(…)` »

Pour coder la fonction « `initialise_case(…)` », il faut être capable de calculer le numéro de la case en connaissant le numéro de sa ligne et celui de sa colonne…

La figure ci-contre illustre la numérotation que l'on souhaite avoir :



✍ En effectuant des essais sur du papier, trouvez la bonne formule qui permet de calculer le numéro de la case en fonction des variables « **col** » et « **lig** » et de la constante « **DIM_LABY** ».

☞ Codez la fonction « **initialise_case(…)** » en utilisant la description en langage universel suivante. Cette fonction a pour arguments les numéros de ligne et colonne de la case qui va être créée. Elle retourne une liste qui contient les données d'une case.

```
FONCTION initialise_case(lig, col) :
    coul = QColor(ROUGE_ALEATOIRE, VERT_ALEATOIRE, BLEU_ALEATOIRE)
    CALCULER numero
    RETOURNER [numero, True, True, True, True, coul]
```

Indications : Pour les composantes aléatoires de la couleur, vous pouvez utiliser un « `random.randint(…,…)` » de manière à générer un nombre aléatoire compris entre 128 et 255. Cela permet d'avoir des couleurs claires…

Résultat : Vous devez obtenir dans la console le tableau (liste de listes de listes…) similaire à ce qui suit :

```
[ [ [ 0, True, True, True, True, <PyQt...>] , [ 1, True, True, True, True, <PyQt...>] , [ 2, True, True, True, True, <PyQt...>] ],
  [ [ 3, True, True, True, True, <PyQt...>] , [ 4, True, True, True, True, <PyQt...>] , [ 5, True, True, True, True, <PyQt...>] ] ,
  [ [ 6, True, True, True, True, <PyQt...>] , [ 7, True, True, True, True, <PyQt...>] , [ 8, True, True, True, True, <PyQt...>] ] ]
```

ACTIVITE N°2 : AFFICHAGE DU LABYRINTHE

Pour afficher le labyrinthe, il faut dessiner une par une toutes des cases. Commençons déjà par créer les fonctions qui dessinent une case.

☞ Ajoutez la fonction suivante :

```
def dessine_laby (scene, laby):
    scene.clear()
    case = [3, True, True, True, True, QColor(128, 200, 128)]
    dessine_case(scene, case, 200, 200)
```

Note : Le programme ne fonctionne toujours pas à ce stade. Il manque la fonction « `dessine_case(…)` »

*La fonction « **dessine_case(…)** » a pour arguments : la « **scene** » graphique, la liste des données qui caractérisent la case (numéro, états binaires des 4 « murs » et la couleur) et les coordonnées « **x, y** » de la position du coin supérieur gauche de la case. Pour dessiner cette case, il faut suivre les étapes suivantes :*

- *Dessiner un carré (sans bordure) avec la couleur de la case.*
- *Écrire un texte donnant le numéro de la case au centre de la case.*
- *Dessiner de chaque « mur » de la case s'il n'a pas été détruit. Un « mur » est constitué d'une ligne…*

Commencez par « décommenter » la ligne `#dessine_laby(…)` qui se trouve dans le programme principal.

Créez une nouvelle fonction nommée « **`dessine_case(scene, case, x, y)`** » qui dessine la case.

Indications :

Vous avez sur Moodle un complément de cours « `Graphisme en Python (PyQt5).pdf` » qui décrit en détails les fonctions graphiques qu'il faut mettre en œuvre pour dessine une case…

Il est conseillé de prendre le temps d'étudier soigneusement ce complément de cours !!!

Pour le fond de la case, dessinez un carré SANS BORD de côté « `DIM_CASE` » et dont le coin supérieur gauche est aux coordonnées « `x,y` ». Pour mémoire, la couleur est stockée dans le 6ème élément de la liste « `case` »…

```
scene.addRect(x, y, largeur, hauteur, pen, brush)
```

avec : `pen = QPen(Qt.NoPen)`

et : `brush = QBrush(coul_case)`

Pour le numéro qui se trouve dans le 1er élément de la liste « `case` », affichez un texte de couleur noire au centre (approximatif) du carré précédent. Le point de référence d'affichage du texte a pour coordonnées : « `(x + DIM_CASE/3, y + DIM_CASE /4)` ». Pour obtenir un résultat satisfaisant, il est conseillé d'avoir un facteur d'agrandissement de `(DIM_CASE/50)`.

Il est aussi demandé de faire cette affichage si et seulement si le numéro est supérieur ou égale à « 0 »…

```
texte_num = scene.addText(numero)
text_num.setPos(position_x,position_y)
text_num.setScale(facteur_agrandissement)
```

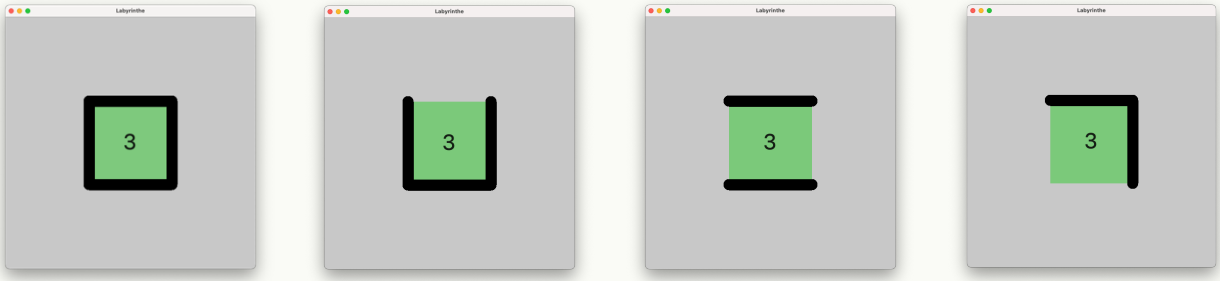
Pour les « murs », il est nécessaire de tester chacun des quatre booléens de la liste « `case` » (2ème, 3ème, 4ème et 5ème élément de la liste), et de dessiner le cas échéant, une ligne noire de longueur « `DIM_CASE` » sur les côtés NORD, EST, SUD ou OUEST du carré affiché précédemment. Pour avoir la bonne épaisseur de ligne, utilisez la valeur issue du calcul suivant : `1+(79//DIM_LABY)` »…

```
pen = QPen(Qt.black, epaisseur, Qt.SolidLine, Qt.RoundCap)
scene.addLine(x1, y1, x2, y2, pen)
```

On veut des bouts arrondis…

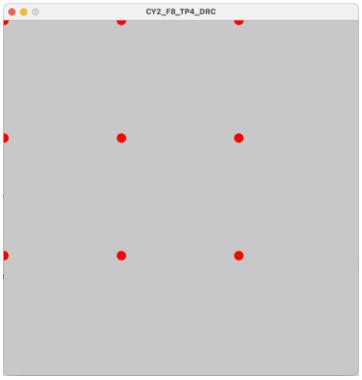
***Résultat :** Vous devez obtenir une case verte avec un 3 au centre de la fenêtre.*

En modifiant les valeurs des booléens dans la liste « `case = [3, …]` » qui se trouve dans la fonction « `dessine_laby(…)` », vous obtenez des variantes sur l'affichage des « murs »…



*On dispose à présent d'une fonction qui permet d'afficher une case. Modifions la fonction « **dessine_laby()** » pour qu'elle affiche toutes les cases du labyrinthe. Cela va se faire par le biais de DEUX boucles imbriquées : « **for lig in** … » et « **for col in** … »…*

*A partir des valeurs des variables « **lig, col** », il faut calculer les coordonnées « **x, y** » du coin supérieur gauche où doit être dessiné la case… La figure donnée ci-contre illustre par des points rouges les différentes coordonnées « **x, y** » produites par ces deux boucles imbriquées…*



- Complétez la fonction « **dessine_laby()** » de manière à avoir l'affichage complet de la grille de cases…

***Résultat** : Vous devez obtenir une grille de 3 x 3 cases comme celle-ci (sauf pour les couleurs) :*

*Si vous changer la valeur de « **DIM_LABY** » tout en haut de votre code, vous obtiendrez plus de cases… La valeur de « **DIM_LABY** » doit être entière et comprise entre 2 et 30 (inclus)…*

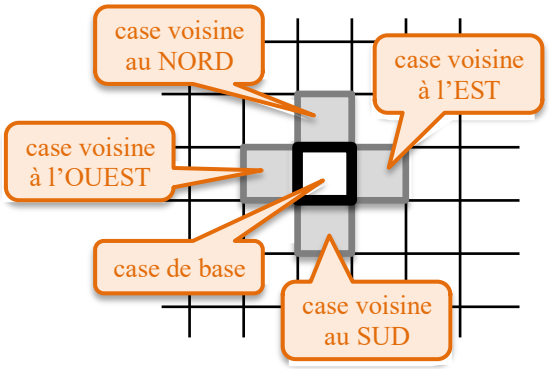
*Remettez après vos différents essais, la valeur « **DIM_LABY = 3** » pour la suite de l'activité…*

ACTIVITE N°3 : SELECTION ALEATOIRE D'UNE CASE ET D'UNE CASE VOISINE

*Pour respecter l'algorithme de création d'un labyrinthe décrit au début du mini-projet, il faut sélectionner aléatoirement une case dans le labyrinthe. Cette case sera appelée « **case de base** ».*

*Il faut aussi choisir toujours aléatoirement une case appelée « **case voisine** » qui se trouve soit au Nord, soit à l'Est, soit au Sud ou encore à l'Ouest de la « **case de base** ». Le schéma ci-contre illustre cet aspect…*

*Avant d'écrire le code de recherche de la « **case de base** » et de la « **case voisine** », Commençons par modifier un peu le programme principal :*



- Apportez les modifications suivantes :

```
    laby = initialise_laby()
    print(laby)
    dessine_laby(scene, laby)
    while True:
        selection = select_cases()
        #agglomere_cases(selection, laby)
        #nb_agglomerations += 1
        #dessine_laby(scene, laby)
        break
```

***Note** : Le programme ne fonctionne pas à ce stade. Il manque la fonction « **select_cases()** »…*

*La fonction « `select_cases()` » va commencer par choisir la « **case de base** ». Cela se fait très simplement en prenant deux valeurs aléatoires parmi les numéros de ligne et les numéros de colonne.*

> ***Note** : La fonction « `random.randrange(DIM_LABY)` » fournit un nombre aléatoire « d » tel que :*
>
> $$0 < d \leq DIM\_LABY$$
>
> *Cette fonction est plus pratique pour effectuer cela que la fonction « `random.randint(0, DIM_LABY-1)` » !!!*

*Cette fonction « `select_cases()` » va ensuite, chercher aléatoirement une « **case voisine** » à la « **case de base** » avec la fonction : « `random.choice([NORD, EST, SUD, OUEST])` ».*

> ***Note** : Cette fonction « `random.choice([…])` » de la bibliothèque « `random` » fournit une valeur aléatoire parmi les éléments de la liste fournie en argument…*

Cette fonction retourne finalement une liste qui contient TROIS informations :

- *Le numéro de la ligne de la « **case de base** ».*
- *Le numéro de la colonne de la « **case de base** ».*
- *Le mur de la « **case de base** » qui touche la « **case voisine** ».*

Le code de cette fonction est donné ci-dessous.

Ajoutez à votre script le code suivant :

```
def select_cases():
    while True :
        l = random.randrange(DIM_LABY)
        c = random.randrange(DIM_LABY)
        mur_voisin = random.choice([NORD, EST, SUD, OUEST])

        print('Ligne case_base   : ', l)
        print('Colonne case_base : ', c)
        print('Numéro case_base  : ', laby[l][c][0])
        nom_murs = ["", "NORD", "EST", "SUD", "OUEST"]
        print('Mur voisin        : ', nom_murs[mur_voisin])
        return [l, c, mur_voisin]
```

Lignes à supprimer à la fin de l'activité 3…

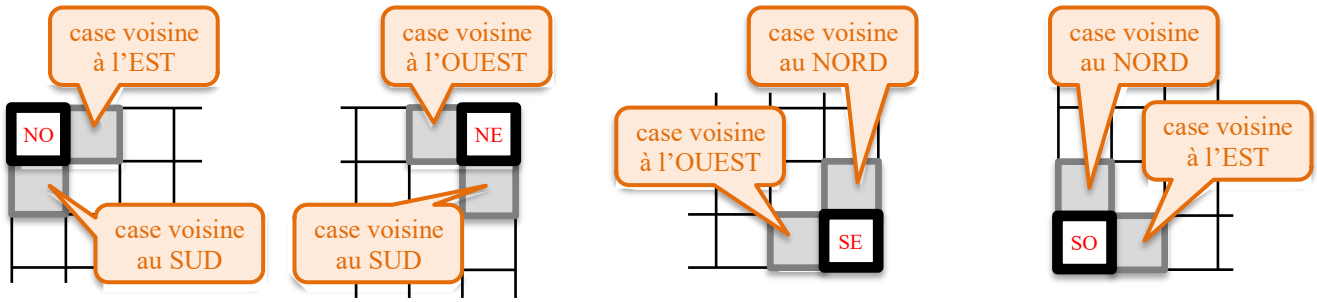
Si vous lancez plusieurs fois votre script, vous pouvez tomber sur des cas de figures qui pose un problème… Observez ce cas :

```
Ligne case_base   :  2
Colonne case_base :  2
Numéro case_base  :  8
Mur voisin        :  EST
```

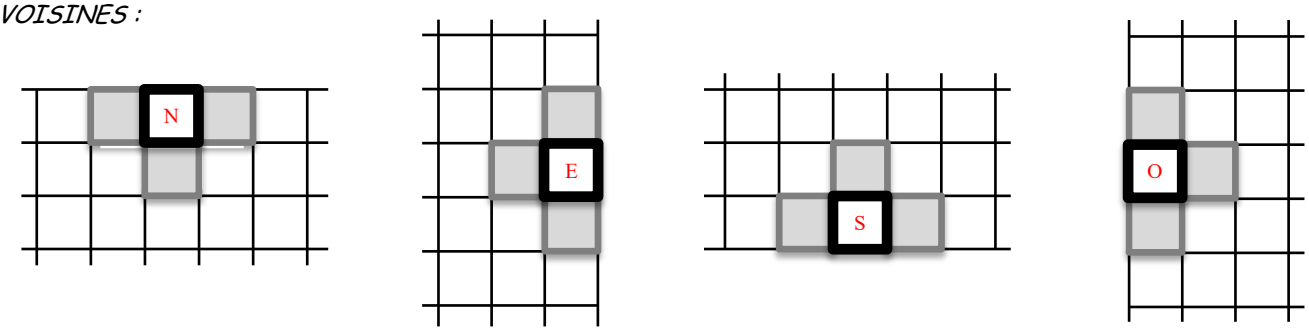
La « **case de base** » est celle de la ligne 2 et colonne 2 avec le numéro 8. Le mur voisin à le mur « **EST** ». Cela signifie que la « **case voisine** » se trouve à droite de la « **case de base** » en l'occurrence en dehors de la grille !!!

*Il en résulte que le choix de la « **case voisine** » dépend de la « **case de base** » et nécessite un peu plus d'attention car les bords de la grille sont des cas particuliers… Étudions ces cas particuliers :*

- *Pour les QUATRE cases des coins du labyrinthe, il y a UNIQUEMENT DEUX CASES VOISINES :*



– *Pour toutes les cases qui sont sur les bords du labyrinthe (en dehors des coins) , il y a UNIQUEMENT TROIS CASES VOISINES :*



– *Les cases qui ne sont pas sur les bords (et implicitement pas sur les coins) ont QUATRE CASES VOISINES. Cette situation n'est pas un cas particulier !*

Complétez la fonction « **`select_cases()`** » par le code qui traite les cas particuliers évoqués ci-dessus. Il est conseillé de commencer par traiter les 4 coins puis les 4 bord et finalement les cases centrales…

***Indications** : Vous pouvez créer une nouvelle fonction (« `select_voisin(l,c)` » par exemple) qui admet comme argument les valeurs « l » et « c » et qui retourne un mur voisin qui existe forcément…*

***Exemple** :* `return random.choice( [EST, SUD] )` *pour la coin en haut à gauche…*

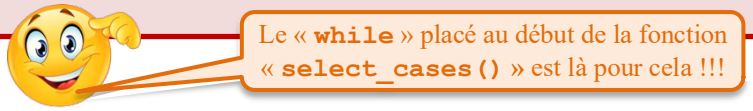
Après avoir testé votre code (plus de cas problématiques) supprimez les lignes comme indiquer dans le code de la fonction « **`select_cases()`** » donné page précédente…

ACTIVITE N°4 : VERIFICATION DES NUMEROS DES DEUX CASES VOISINES

*Nous disposons pour l'instant d'un code qui est capable de trouver aléatoirement la « **case de base** » et une « **case voisine** » aléatoire valide. En revanche, nous n'avons pas vérifié que ces deux cases voisines portent des NUMEROS DIFFERENTS…*

Il est donc nécessaire de rajouter une condition dans la recherche des deux cases aléatoires voisines. Pour mémoire, il faut appliquer la règle suivante :

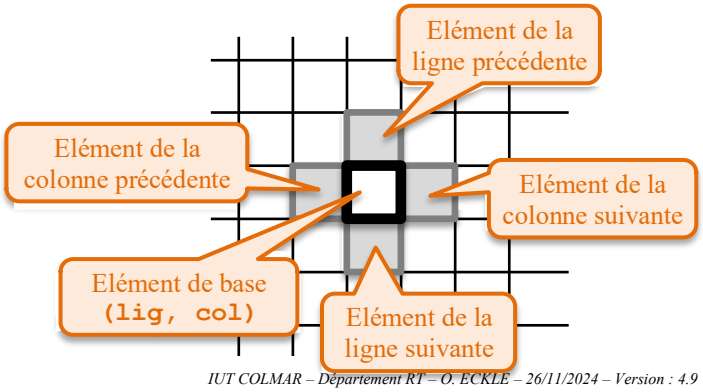
*Lorsque l'on recherche une valeur aléatoire qui doit vérifier une condition particulière, les « bons » codeurs utilisent une boucle folle « **`while True :`** » dans laquelle on fait le tirage aléatoire et l'on vérifie la condition. Si la condition est vérifiée, on génère un « **`break`** » ou tout simplement un « **`return`** » si c'est une fonction qui réalise le test de condition. Sinon, on reste dans la boucle folle…*



*Avant de passer au codage, il se pose un problème : Comment récupérer les données (notamment le numéro) de la « **case voisine** » en connaissant la « **case de base** » et la position de la voisine (Nord, Sur, Est ou Ouest)…*

*Considérons un tableau (liste de liste) dans lequel nous avons un élément de base référencé pas ses deux index notés « **`lig`** » et « **`col`** ». Quels sont les index des quatre cases voisines ?*

✍ Sur une feuille de papier, trouvez les valeurs des index des 4 éléments voisins en fonction des index « **`lig`** » et « **`col`** » de l'élément de base.



Nous sommes à présent en mesure de compléter le fonction « `select_cases()` » pour inclure la vérification des NUMEROS DIFFERENTS…

- Ajoutez le code qui permet de retourner (« `return [l, c, mur_voisin]` ») si et seulement si la « **case de base** » et la « **case voisine** » ont des numéros différents.

> ***Note** : Dans le cas contraire, rien n'est retourné et la boucle « `while` » va chercher aléatoirement un autre couple de « case de base » et « case voisine » …*

ACTIVITE N°5 : ASSOCIATION DES DEUX CASES VOISINES

*Pour respecter l'algorithme de création du labyrinthe que a été présentée au début de ce mini-projet, Il est nécessaire d'« associer » les deux cases voisines trouvées avec le code qui précède. Cette association consiste à supprimer les « murs » communs et d'affecter au numéro et à la couleur de la « **case voisine** » celui et celle de la « **case de base** ».*

Nous allons créer une nouvelle fonction nommée « `agglomere_cases(selection, laby)` ». Son 1er argument est la liste produite par la fonction « `select_cases(…)` ».

Cette nouvelle fonction ne retourne rien ! Elle est appelée dans le programme principal à une condition :

- Décommentez les 3 lignes correspondante du programme principale…
- Codez la fonction « `agglomere_cases(selection, laby)` »…

> ***Indications** :*
>
> *On peut commencer par placer dans des variables* `l, c` *et* `mur_voisin` *le contenu de* `selection`.
>
> *Puis affecter à* `numero_base` *le numéro de la case de base avec* `laby[l][c][0]`.
>
> *De même pour la couleur dans la variable* `coul_base`.
>
> *Il faut ensuite chercher les valeurs des index* `l_voisin` *et* `c_voisin` *de la case voisine à partir de* `l, c` *et* `mur_voisin`.
>
> *Puis affecter à* `numero_voisin` *le numéro de la case voisine.*
>
> *La suppression du bon mur dans la case de base se fait avec :* `laby[l][c][mur_voisin]=False`.
>
> *Pour le mur de la case voisine, il faut commencer par trouver de quel mur il s'agit…*
>
> *En considérant le « dictionnaire » :* `mur_oppose = {NORD: SUD, SUD: NORD, EST: OUEST, OUEST: EST}`
> *cela devient facile… le mur à supprimer dans la case voisine est :* `mur_oppose[mur_voisin]`.
>
> *Pour finir, il faut balayer toute la grille avec 2 boucles imbriquées et pour chaque case ayant pour numéro :* `numero_voisin`, *lui affecter* `numero_base` *et* `coul_base`…

- Testez votre code…

> ***Résultat** : Vous devez obtenir une grille avec DEUX cases associées un peu comme cela :*
>
> 
>

ACTIVITE N°6 : GENERATION DU LABYRINTHE COMPLET

Pour obtenir le labyrinthe complet, il suffit de répéter l'ensemble des opérations jusqu'à ce qu'il n'y ait plus qu'un seul numéro dans la grille. Pour cela on va comptabiliser le nombre d'agglomérations…

*Au départ, avant de faire des agglomérations, la grille comporte « **DIM_LABY x DIM_LABY** » numéros différents car c'est le nombre de cases…*

Après UNE agglomération, UN numéro a disparu. Après DEUX agglomérations, DEUX numéros ont disparu. Après TROIS agglomérations, TROIS numéros ont disparu. Et ainsi de suite.

*Le labyrinthe est terminé lorsqu'il ne reste plus qu'un seul numéro. Autrement dit lorsque l'on fait « **DIM_LABY x DIM_LABY - 1** » agglomérations.*

- Modifiez la condition de la boucle « **while** » qui se trouve dans le programme principal pour obtenir le labyrinthe complet. Bien évidemment, il faut aussi supprimer le « **break** » …
- Testez votre code.

***Résultat** : Le résultat final du labyrinthe apparait immédiatement. On ne voit pas les différentes étapes du calcul…*

Il serait intéressant de ralentir la création du labyrinthe complet pour voir les différentes étapes d'agglomérations. Pour cela, on va mettre en œuvre une pause avec le timer.

- Ajoutez tout juste après le « **while … :** » du programme principal, le code suivant :

```
while nb_agglomerations …
    end_time = time.time() + 10 / DIM_LABY ** 2
    while time.time() < end_time:
        QApplication.processEvents()ef select_cases()
    selection = select_cases()
```

Comme l'ensemble du code présent dans ce script travail avec des dimensions du labyrinthe et de la fenêtre paramétrables, il est possible de modifier le nombre de cases ou/et la dimension de la fenêtre sans avoir à toucher à l'ensemble du code…

- Modifiez la valeur de « **DIM_LABY** » tout en haut du script… Il faut évidemment être raisonnable dans ces changements.

***Remarque** : Avec un nombre de cases assez important, on obtient des effets graphiques intéressants…*

ACTIVITE N°7 : PRESENTATION FINALE

Une fois que le labyrinthe a été complètement généré, on aimerait avoir un fond gris clair et faire disparaitre les numéros.

- Proposez une modification du code pour obtenir ce résultat.

***Indications** : Vous pouvez créer une fonction de nettoyage de la grille qui est appelée uniquement lorsque le labyrinthe est terminé.*

Si le code a été fait comme il a été demandé (notamment dans l'affichage du labyrinthe), il suffit de mettre à « -1 » les numéros de toutes les cases…

Il serait pratique de créer une case d'entrée et une case de sortie du labyrinthe. Il est proposé de placer aléatoirement la case d'entrée sur le côté GAUCHE du labyrinthe. La case de sortie est placée aussi aléatoirement mais sur le côté DROIT.

La case d'entrée pourrait être coloriée en ROUGE tandis que celle de sortie en VERT. Bien évidemment, deux murs doivent être supprimés pour bien matérialiser l'entrée et la sortie.

- Proposez une seconde modification du code pour obtenir ce résultat.