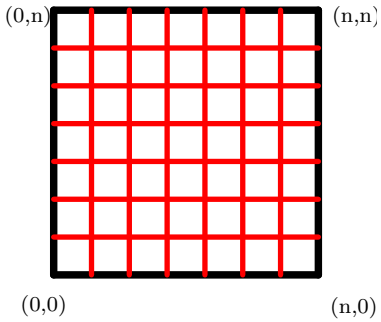
Projet 2 : **Labyrinthe**

L'objectif de ce projet est la génération aléatoire de labyrinthes, et l'étude de chemins pouvant être parcourus dans ceux-ci.

L'algorithme de génération aléatoire de labyrinthes reposera sur une version "randomisée" de l'algorithme de Kruskal pour les graphes.
L'idée essentielle est la suivante :

- on met en place un quadrillage complet (représentant tous les murs possibles)

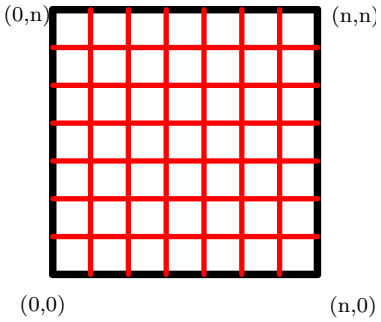


- on passe en revue tous les murs intérieurs et on "fait sauter" ceux qui séparent deux cases ne pouvant être reliées par un chemin dans le labyrinthe.

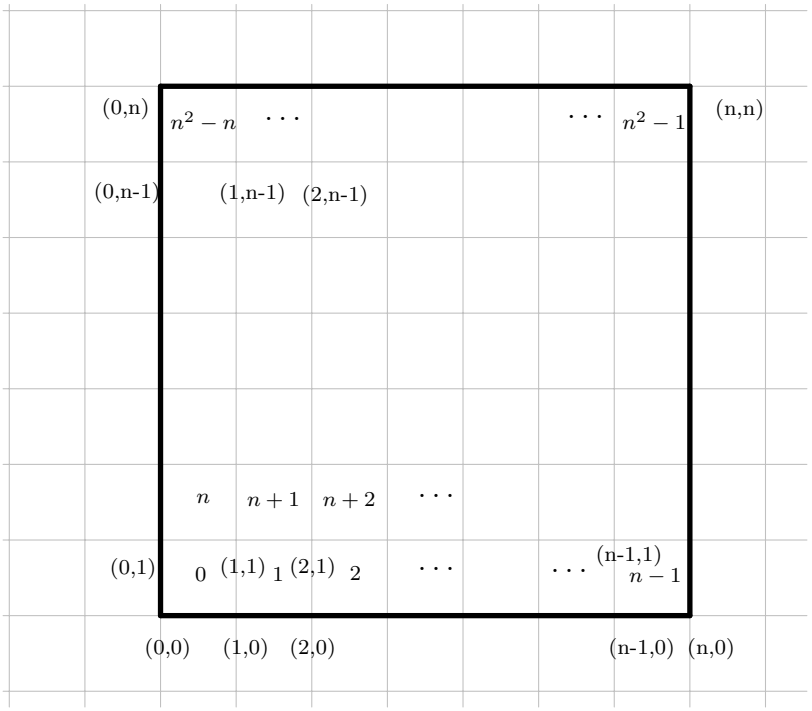
Pour mettre en place le second point, on utilisera un algorithme de type "Union-Find". L'idée sera d'étiqueter chaque case du labyrinthe de sorte que deux cases du labyrinthe (en construction) auront une même étiquette s'il est possible de les relier par un chemin dans celui-ci.
Au départ, chaque case a une étiquette différente (cas du quadrillage complet). On passe en revue tous les murs, et on supprime les murs séparant deux cases n'ayant pas la même étiquette en faisant en sorte que les deux zones accessibles via ce nouveau passage aient désormais la même étiquette.

On pourra ensuite poursuivre l'étude du labyrinthe en proposant par exemple à l'utilisateur de cliquer sur une case du labyrinthe et obtenir un chemin allant de l'entrée du labyrinthe jusqu'à cette case (et donner la longueur de ce chemin). Cette tâche utilisera l'algorithme de parcours en largeur d'un graphe ("Breadth First Search"). On pourra éventuellement poursuivre cette étude en effectuant quelques statistiques de longueurs de chemins séparant l'entrée de la sortie de labyrinthes générés aléatoirement.

Exercice 1 : (saisie des murs)
Dans cet exercice, on se propose de saisir tous les murs intérieurs possibles du labyrinthe (quadrillage complet rouge ci-dessous) et de ne dessiner que les murs extérieurs (ceux en noir ci-dessous). On propose également un codage des cases séparées par chacun des murs. On supposera que le labyrinthe est "carré" avec n cases en largeur et en longueur.



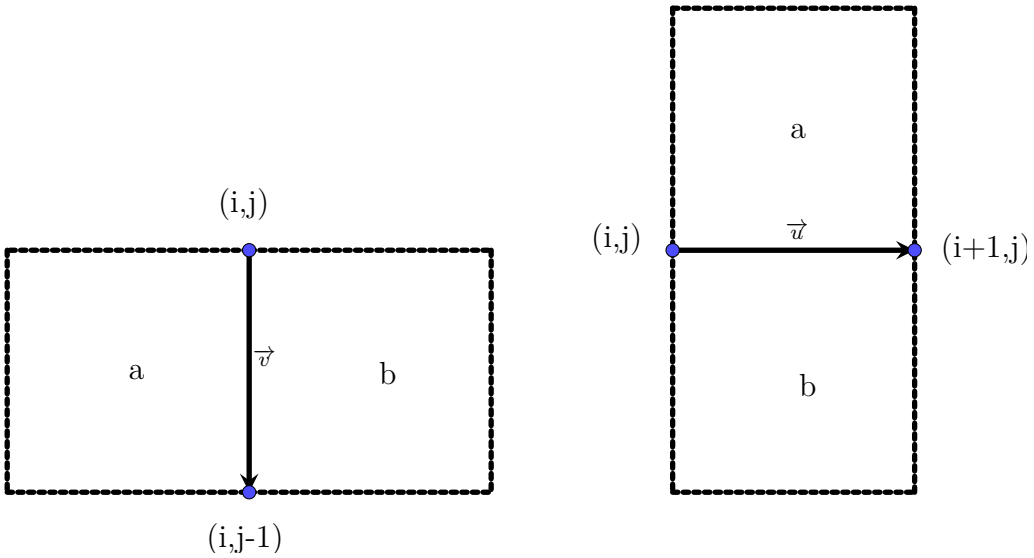
1) a) Écrire une fonction `numCase(i,j,n)` qui affecte un numéro à la case dont les coordonnées du coin inférieur gauche sont (i, j) (où $0 \leqslant i < n$ et $0 \leqslant j < n$). La numérotation des cases devra respecter le schéma ci-dessous.



b) Écrire de même une fonction `CoordCase(num,n)` qui, à partir du numéro d'une case, détermine les coordonnées (i, j) de son coin inférieur gauche.

2) Écrire une fonction `Genere_Aretes(n)` prenant en entrée un entier $n \geqslant 1$ et construisant la liste `Aretes` définie ci-dessous.
Les éléments de `Aretes` seront des listes de trois couples :

- les deux premiers couples sont les coordonnées des extrémités d'un mur horizontal intérieur (soit (i, j), $(i + 1, j)$) ou d'un mur vertical intérieur (soit (i, j), $(i, j - 1)$),
- le troisième couple (a, b) est formé des numéros a et b des deux cases séparées par le mur défini par les deux premiers couples.



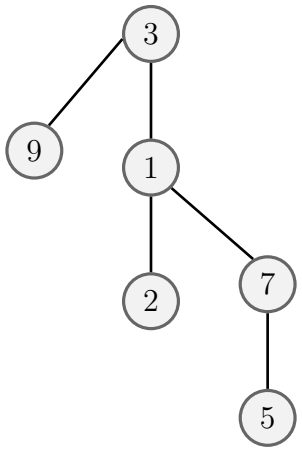
Les murs stockés dans `Aretes` ne devront être **que les murs intérieurs** mais la fonction devra dessiner les autres arêtes ou le "cadre" du labyrinthe.

Exercice 2 : (algorithme Union-Find)

Dans cet exercice, nous allons programmer un algorithme "Union-Find" dont l'objectif est le suivant :
Étant donné un ensemble et des parties de cet ensemble, savoir pour un élément donné à quelle partie il appartient, avec la contrainte supplémentaire que certaines parties peuvent "fusionner" au cours du temps (c'est-à-dire, si A et B sont des parties à l'instant t, elles peuvent ne former qu'une partie $A \cup B$ à l'instant $t+1$).

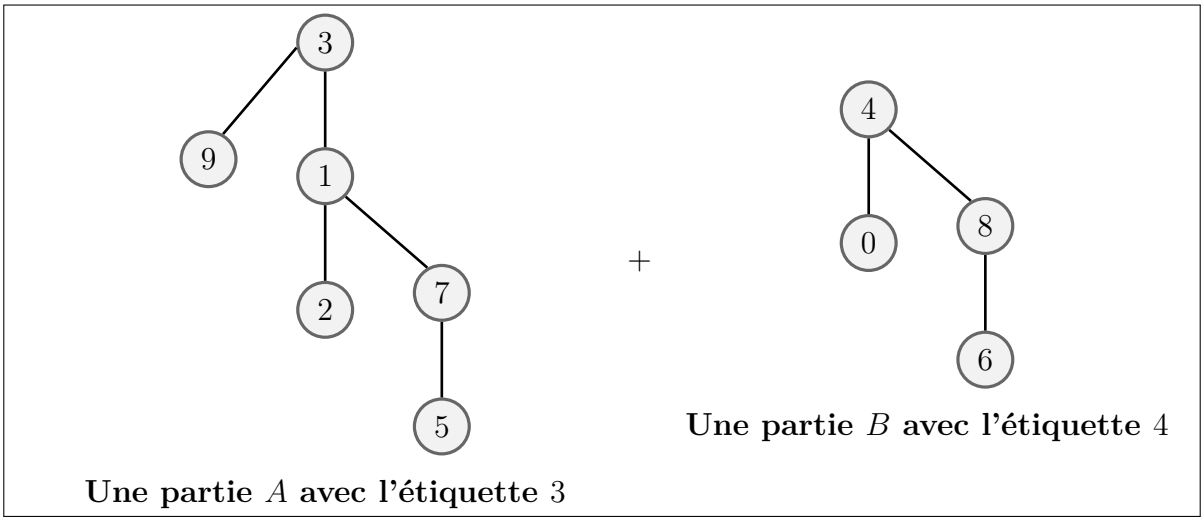
Dans le contexte de notre projet, l'ensemble sera l'ensemble des cases du labyrinthe, les parties seront des parties maximales de cases communicantes dans le labyrinthe (c'est-à-dire telles que deux cases d'une même partie peuvent toujours être reliée par un chemin du labyrinthe). Deux de ces parties seront amenées à se réunir si un mur du labyrinthe qui les sépare est détruit.

Nous représenterons ces parties par des arbres (c'est-à-dire un graphe non orienté connexe acyclique). Concrètement, un arbre aura une forme analogue à celle ci-dessous :

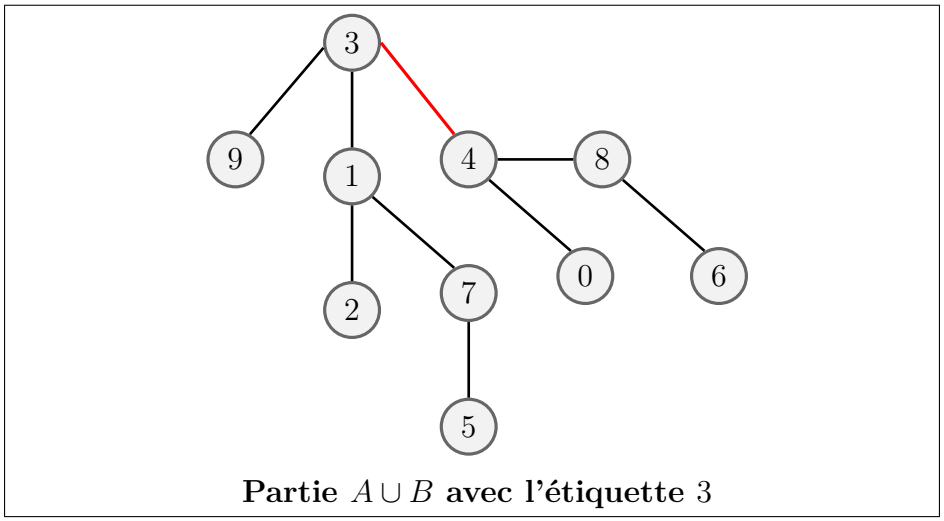


Un arbre représentant l'ensemble $\{3, 1, 9, 2, 7, 5\}$

(un ensemble de noeuds et d'arêtes, avec un noeud racine (ici le noeud 3) et des noeuds accessibles depuis ce noeud racine par une succession d'arêtes. Un arbre est acyclique, c'est-à-dire n'a pas de cycle : on ne peut trouver une succession d'arêtes partant d'un noeud et revenant vers celui-ci).
L'arbre ci-dessus pourra représenter la partie $\{3, 1, 9, 2, 7, 5\}$ et cette partie pourra être "étiquetée" (c'est-à-dire identifiée) par la valeur du noeud racine de l'arbre (ici 3).
L'élément se trouvant à la racine de l'arbre représente donc la partie contenant tous les éléments de l'arbre.
Lorsque l'on unit deux parties, on place simplement la racine de l'une comme père de la racine de l'autre.
On relie deux arbres, pour n'en former plus qu'un et les éléments des deux parties deviennent maintenant éléments de la même partie, représentée par la racine du nouvel arbre.



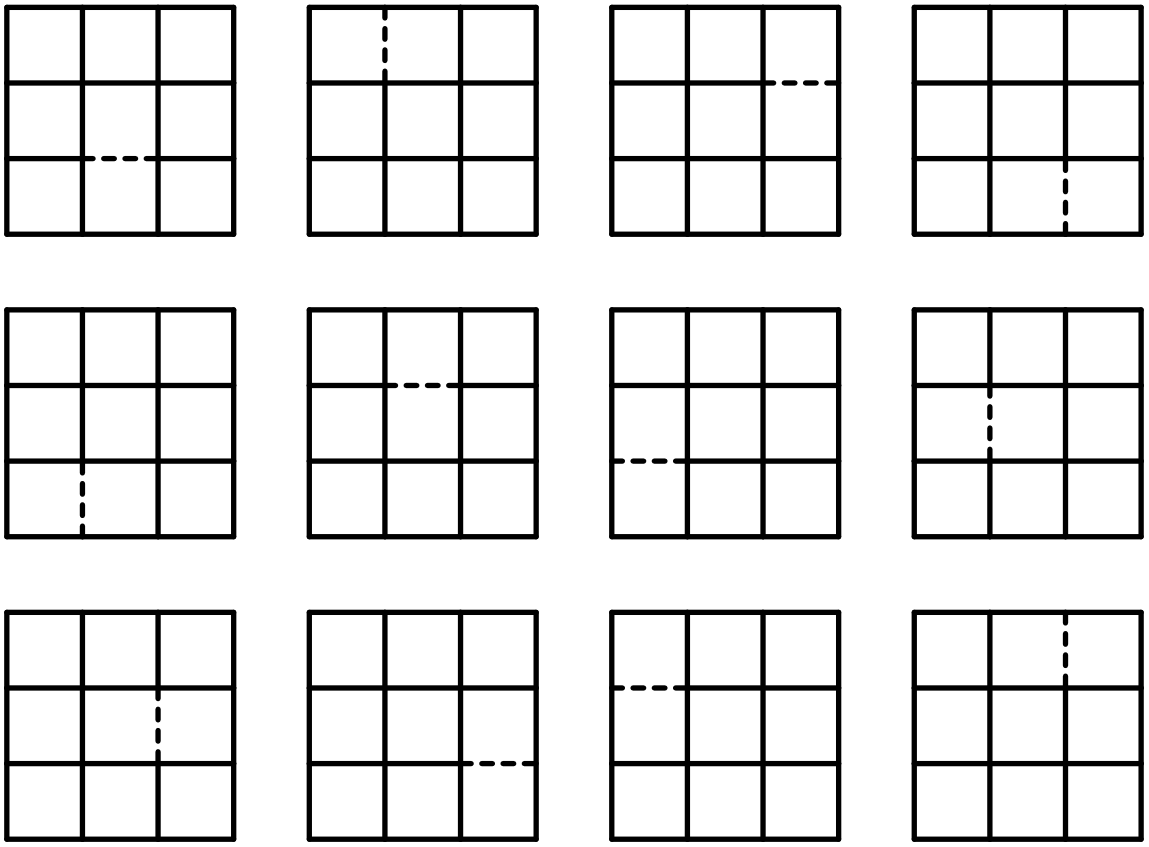
Une partie A avec l'étiquette 3 **Une partie B avec l'étiquette** 4



Partie $A \cup B$ avec l'étiquette 3

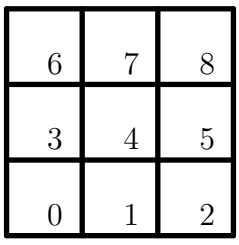
Pour connaître l'étiquette d'une partie à laquelle appartient un élément, il suffit de remonter dans l'arbre dont cet élément fait partie, jusqu'à la racine de cet arbre.
Pour stocker un ensemble d'arbres, on peut utiliser une liste `Peres` dans laquelle on associe à chaque élément (ou noeud) `i` l'indice de son père `Peres[i]` dans l'arbre auquel il appartient (avec `Peres[i]`$=-1$ si le noeud `i` n'a pas de père, c'est-à-dire si `i` est une racine). Nous étudions un exemple en détail ci-dessous.

On considère un labyrinthe à 9 cases étiquetées par leurs numéros respectifs 0,1,2,...,8.
On fait figurer ci-dessous l'ordre (aléatoire) dans lequel on va considérer dans notre exemple les 12 murs intérieurs à éventuellement détruire (pour relier deux zones ne communiquant pas).



.....

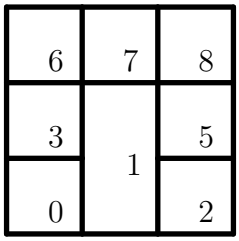
Étape 1 : État de départ du labyrinthe :

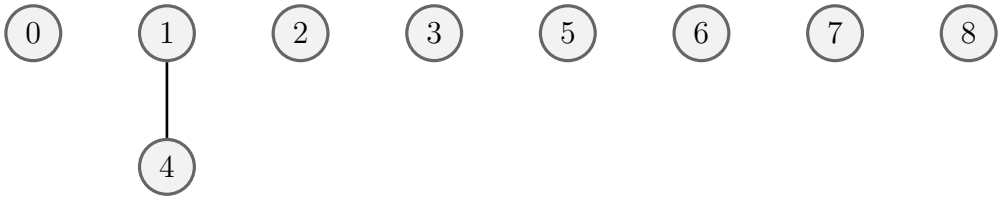


Il y a neuf zones (donc neuf étiquettes).
Chacun des 9 éléments est la racine de son arbre.

```
Peres = [-1,-1,-1,-1,-1,-1,-1,-1,-1]
```

Étape 2 : On considère le mur entre les cases 1 et 4 : on le casse.

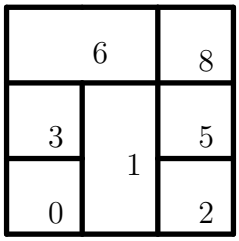




La nouvelle zone {1,4} a pour étiquette 1 et 1 est le père de 4.

`Peres = [-1,-1,-1,-1,1,-1,-1,-1,-1]`

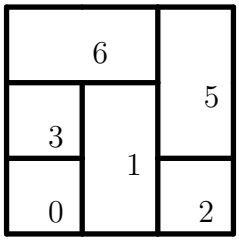
Étape 3 : On considère le mur entre les cases 6 et 7 : on le casse.

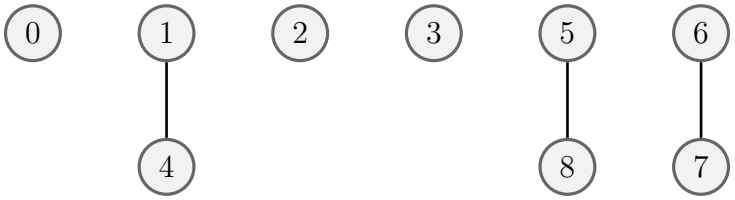


La nouvelle zone {6,7} a pour étiquette 6 qui est aussi le père de 7.

`Peres = [-1,-1,-1,-1,1,-1,-1,6,-1]`

Étape 4 : On considère le mur entre les cases 5 et 8 : on le casse.

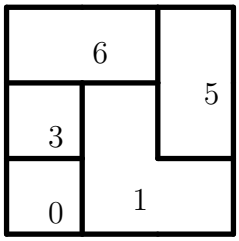


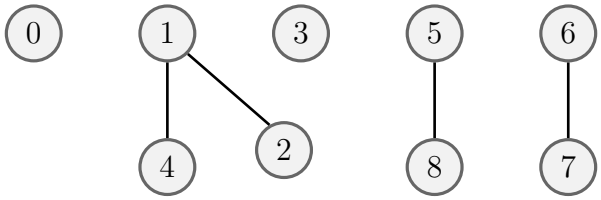


La nouvelle zone {5,8} a pour étiquette 5 qui est aussi le père de 8.

```
Peres = [-1,-1,-1,-1,1,-1,-1,6,5]
```

Étape 5 : On considère le mur entre la zone 1 et la case 2 : on le casse.

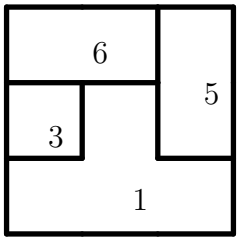


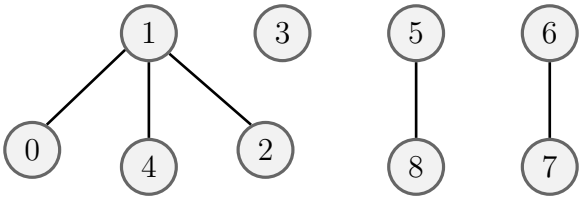


La nouvelle zone {1,2,4} a pour étiquette 1 qui est aussi le père de 4 et 2.

```
Peres = [-1,-1,1,-1,1,-1,-1,6,5]
```

Étape 6 : On considère le mur entre la zone 1 et la case 0 : on le casse.

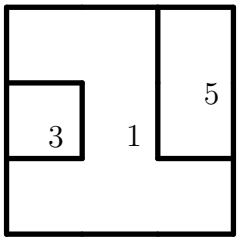


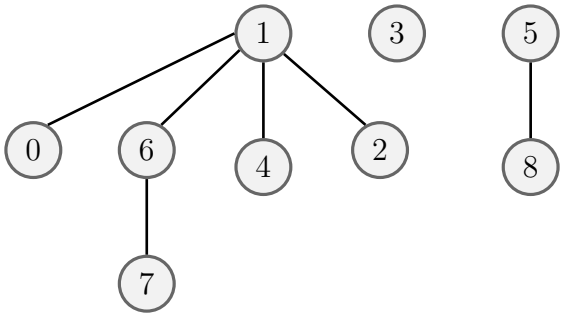


La nouvelle zone {0,1,2,4} a pour étiquette 1 qui est aussi le père de 0,2 et 4.

`Peres = [1,-1,1,-1,1,-1,-1,6,5]`

Étape 7 : On considère le mur entre les zones 6 et 1 : on le casse.

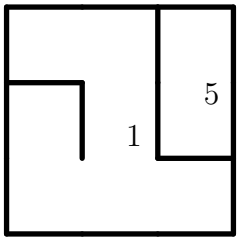


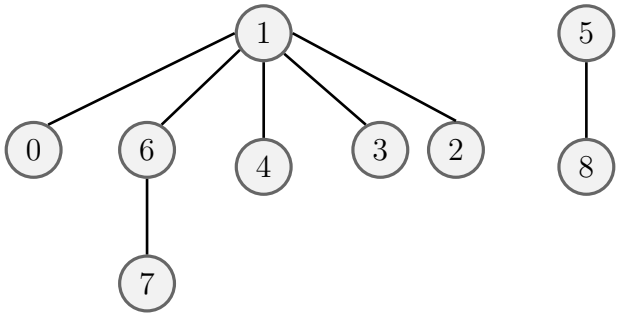


La nouvelle zone {0,1,2,4,6,7} a pour étiquette 1 qui est aussi le père de 0,2, 4 et 6.
6 est toujours le père de 7.

`Peres = [1,-1,1,-1,1,-1,1,6,5]`

Étape 8 : On considère un mur entre la case 3 et la zone 1 : on le casse.

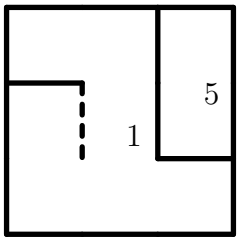




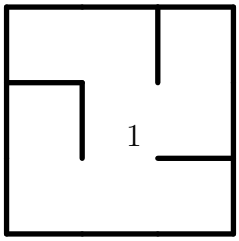
La nouvelle zone {0,1,2,3,4,6,7} a pour étiquette 1 qui est aussi le père de 0,2,3,4 et 6.

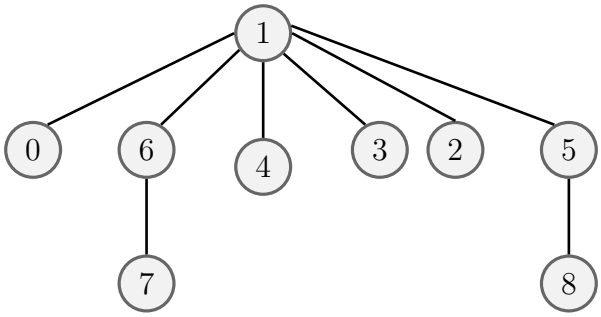
```
Peres = [1,-1,1,1,1,-1,1,6,5]
```

Étape 9 : Le mur ci-dessous (tirets) ne sépare pas deux zones distinctes ne communiquant pas. On ne le casse pas.



Étape 10 : On considère un mur entre les zones 1 et 5 : on le casse.

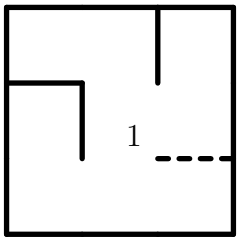




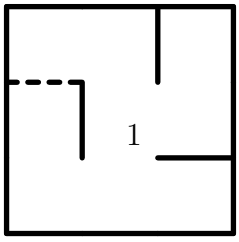
La nouvelle zone {0,1,2,3,4,5,6,7,8} a pour étiquette 1 qui est aussi le père de 0,2,3,4,5 et 6. Et 5 est toujours le père de 8.

```
Peres = [1,-1,1,1,1,1,1,6,5]
```

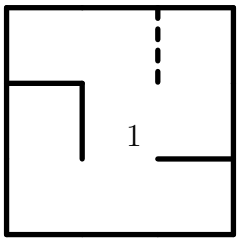
Étape 11 : Le mur ci-dessous (tirets) ne sépare pas deux zones distinctes ne communiquant pas. On ne le casse pas.



Étape 12 : Le mur ci-dessous (tirets) ne sépare pas deux zones distinctes ne communiquant pas. On ne le casse pas.



Étape 13 : Le mur ci-dessous (tirets) ne sépare pas deux zones distinctes ne communiquant pas. On ne le casse pas.



Remarque

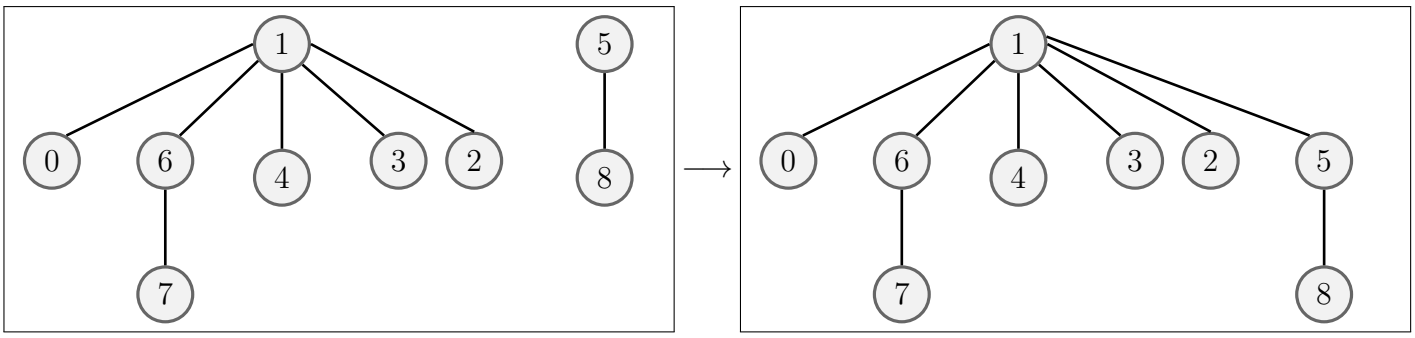
Il faudra éviter des situations où un arbre pourrait ressembler à une longue chaîne comme ci-dessous :

et où déterminer l'étiquette de l'ensemble dont fait partie un élément pourrait finir par demander de nombreuses opérations : passer par tous les noeuds pour atteindre la racine.

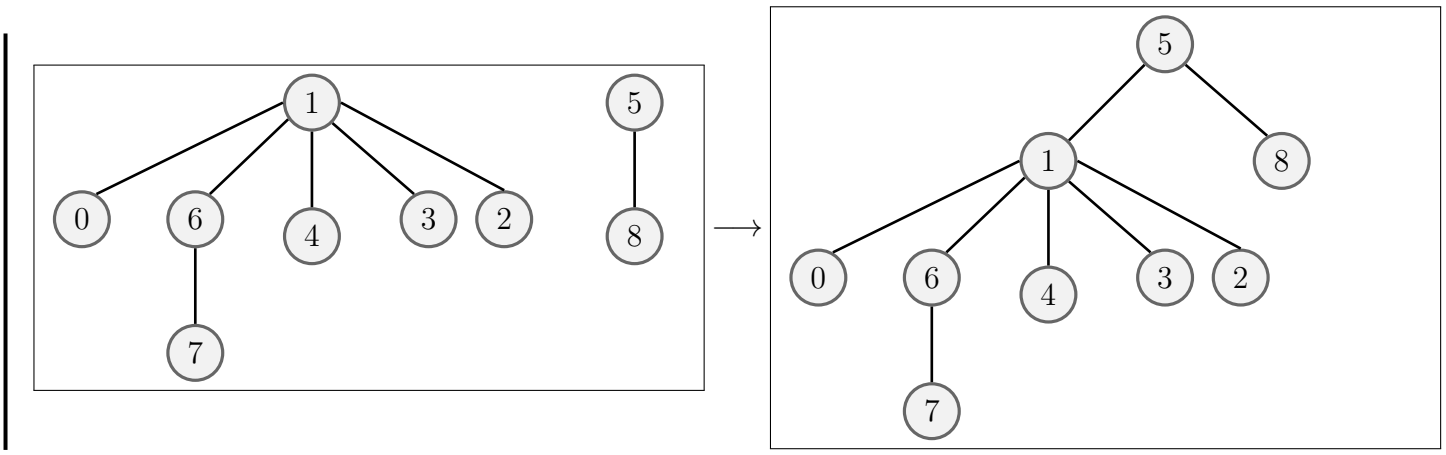
Pour éviter cette situation, il est possible d'équilibrer l'arbre, afin qu'il n'y ait pas d'un côté des branches très courtes, et de l'autre côté des branches très longues, mais des branches qui soient toutes plus ou moins de la même longueur. Cela réduit considérablement la longueur de la branche la plus longue, donc le temps maximal nécessaire pour déterminer l'ensemble dont fait partie un élément.

Un moyen simple d'obtenir un arbre mieux équilibré consiste à bien choisir, au moment de fusionner deux parties, lequel des deux éléments est placé comme père de l'autre. En plaçant comme père l'élément de l'ensemble dont l'arbre a la profondeur la plus grande, on obtient toujours un arbre mieux équilibré qu'en faisant le contraire.

Ainsi, le passage de l'étape 9 à l'étape 10 de l'exemple ci-dessus s'est fait de cette manière (qui permet de conserver un arbre de profondeur 3) :



et non ainsi (ce qui aurait donné un arbre d'une plus grande profondeur (4 ici)) :



1) Écrire une fonction `Etiquette` qui prend en entrée le numéro d'une case (suivant la numérotation indiquée au 1) a) de l'exercice 1) et donne en sortie l'étiquette de l'arbre à laquelle cette case appartient. La fonction devra donc donner en sortie la valeur de la racine de l'arbre correspondant à la partie à laquelle appartient la case. Cette fonction déclarera la liste `Peres` comme variable globale.
2) Écrire une fonction `UnitParties` prenant en entrée les étiquettes de deux arbres et effectuant la "fusion" de ces deux arbres.
Cette fonction déclarera `Peres` et `Profondeurs` comme variables globales où `Profondeurs[i]` est la profondeur de l'arbre dont l'étiquette est `i` (on conviendra que `Profondeurs[i]` vaudra 1 si `i` ne correspond à aucune étiquette d'arbres).
L'opération de fusion se fera simplement en déclarant la racine de l'un des deux arbres comme le père de l'autre de sorte que la profondeur du nouvel arbre soit la plus petite possible. On fera bien attention de mettre à jour la liste `Peres` (bien sûr) mais aussi la liste `Profondeurs`.
3) Écrire une fonction `LabyAleatoire` prenant en entier $n \geqslant 1$ en entrée et construisant en sortie un labyrinthe aléatoire "carré" avec n^2 cases. Cette fonction devra :

- Dessiner le "cadre" du labyrinthe (uniquement si vous avez oublié de le faire à la question 2) de l'exercice 1 dans la fonction `Genere_Aretes`).
- Déclarer globales les variables `Peres` et `Profondeurs` et les initialiser correctement.
- Créer la liste `Aretes` des arêtes intérieures du labyrinthe (*cf* fonction de la question 2) de l'exercice 1).
- Tirer aléatoirement, une à une et sans remise, chaque arête intérieure et obtenir les étiquettes des deux zones qu'elle séparent.
- Unir les deux zones si les deux étiquettes sont distinctes (cas où l'on "casse" le mur),
- Sinon, si les deux zones ont la même étiquette, dessiner le mur correspondant à l'arête et garder en mémoire cette arête dans une liste `Aretes_Bis` (initialement vide).
- Donner en sortie la liste `Aretes_Bis` ci-dessus des arêtes intérieures du labyrinthe.

Bien entendu, cette fonction fera appel aux fonctions `Genere_Aretes`, `Etiquette` et `UnitParties` créées précédemment.
On pourra se souvenir que, si `L` est une liste, l'instruction `a=L.pop(i)` permet de stocker dans `a` l'élément d'indice `i` de la liste `L` tout en le retirant de cette liste.
Cette fonction `LabyAleatoire` a donc un double objectif : afficher le labyrinthe et garder en mémoire les arêtes intérieures de celui-ci (dans `Aretes_Bis`).

Exercice 3 :

Dans cet exercice, on se contentera de tester du code pour mieux comprendre ce qui sera fait ultérieurement. On gardera pour les exercices suivants les modifications réalisées dans la question 1) et on enrichira ce qui aura été constaté à la question 2.

1) Après les déclarations des variables globales `Peres` et `Profondeurs` de la fonction `LabyAleatoire` précédente, rajouter la ligne :

```
fig = plt.figure()
```

(cette ligne a pour effet de stocker la figure qui sera tracée dans la variable `fig`)
puis remplacer la dernière ligne de la fonction

```
return Aretes_Bis
```

par

```
return (Aretes_Bis, fig)
```

(on renvoie donc en sortie le couple formée de la liste `Aretes_Bis` et de la figure tracée `fig`)

2) Écrire à la suite de la fonction `LabyAleatoire` la fonction suivante :

```
def Coord_Point(n):
    fig = LabyAleatoire(n)[1]
    def clic(event):
        print('points cliqués:', event.xdata, event.ydata)
    fig.canvas.mpl_connect('button_press_event', clic)
```

La tester pour quelques valeurs de n (pas trop grandes). On cliquera avec la souris sur quelques points du labyrinthe et on regardera ce qu'il se passe sur la console. Expliquer alors précisément le rôle de cette fonction.

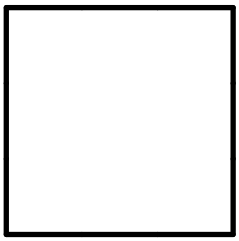
Exercice 4 :

Nous allons nous servir de ce que l'on a observé dans l'exercice précédent pour obtenir le tracé et la longueur d'un chemin reliant deux cases du labyrinthe pointées par l'utilisateur avec la souris. Mais pour pouvoir effectuer cette tâche (trouver un tel chemin et sa longueur), nous allons avoir besoin de représenter le labyrinthe sous forme d'un graphe non orienté, c'est-à-dire un ensemble de noeuds et d'arêtes reliant certains de ces noeuds.

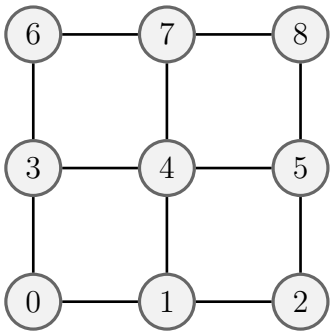
Les cases du labyrinthe seront les noeuds du graphe. Deux noeuds du graphe seront reliés par une arête si et seulement si ils correspondent à deux cases voisines non séparées par un mur.

Voici ci-dessous deux exemples :

Exemple 1 : Cas d'un labyrinthe de 9 cases, sans mur :

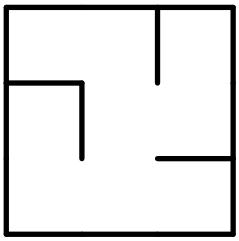


Un labyrinthe à 9 cases, sans mur

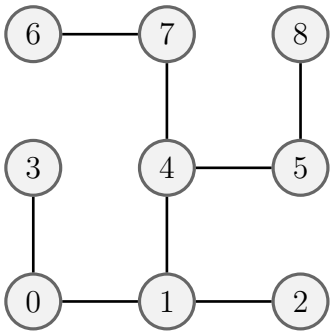


Sa représentation sous forme de graphe

Exemple 2 : Cas d'un exemple de labyrinthe à 9 cases



Un labyrinthe



Sa représentation sous forme de graphe

Un graphe non orienté ayant m noeuds numérotés de 0 à $m-1$ sera représenté par une structure de donnée adaptée qu'on appelle *liste d'adjacence*. Pour un tel graphe, la liste d'adjacence représentant G sera la liste `L` telle que :
pour tout $k \in [\![0, m-1]\!]$, `L[k]` est la liste des noeuds adjacents au noeud `k`, c'est-à-dire la liste des noeuds relié au noeud `k` par une arête. Par exemple, la liste d'adjacence du graphe du premier exemple (labyrinthe sans mur) serait :

$$L = [[1,3],[0,2,4],[1,5],[0,4,6],[1,3,5,7],[2,4,8],[3,7],[4,6,8],[5,7]]$$

et la liste d'adjacence du graphe du deuxième exemple serait :

$$L = [[1,3],[0,2,4],[1],[0],[1,5,7],[4,8],[7],[4,6],[5]]$$

Pour représenter notre labyrinthe sous forme d'une liste d'adjacence, on commencera par faire comme s'il n'y avait pas de mur puis on parcourera la liste `Aretes_Bis` générée par la fonction `LabyAleatoire` de l'exercice 2 pour voir les arêtes du graphes qui disparaissent à cause de la présence d'un mur.
1) Écrire une fonction Python `Liste_init` prenant en entrée un entier naturel $n \geqslant 1$ et donnant en sortie la liste d'adjacence d'un labyrinthe **sans mur** ayant n^2 cases numérotées de 0 à n^2-1 (selon la numérotation définie dans l'exercice 1).
2) Écrire une fonction Python `Liste_Laby` prenant en entrée un entier $n \geqslant 1$, une liste `Murs` (de la même forme que la liste `Aretes_Bis` de la fonction `LabyAleatoire`) stockant les divers murs d'un labyrinthe à n^2 cases sous forme de listes de trois couples (comme précisé dans la question 2 de l'exercice 1). Cette

fonction devra donner en sortie la liste d'adjacence du graphe représentant le labyrinthe défini par la liste `Murs`.

Indication : La fonction fera appel à la fonction `Liste_init` de la question 1) pour initialiser la liste d'adjacence. Puis, en effectuant un seul parcours de la liste `Murs`, elle supprimera quelques coefficients des listes générées par `Liste_init` en fonction de la présence de murs.

Exercice 5 :

Dans cet exercice, on se propose de faire apparaître sur le labyrinthe un chemin reliant deux cases du labyrinthe choisies par l'utilisateur. Tout d'abord, il faut associer à un clic de souris le numéro de la case qui a été pointée.

1) Écrire une fonction Python prenant en entrée la figure d'un labyrinthe carré, sa largeur n (nombre de case d'une ligne) et déterminant le numéro `num` de la dernière case qui a été cliquée avec la souris par l'utilisateur. On complétera la code suivant (et en particulier la sous-fonction `clic`) :

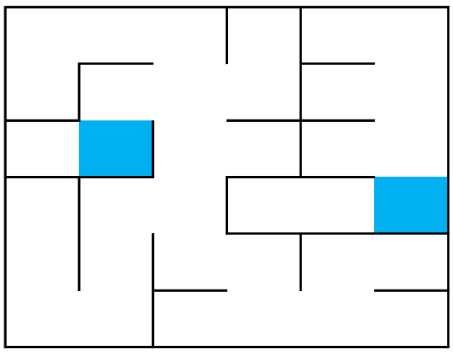
```
def Case(fig,n):
    global num
    num = 0
    def clic(event):
        .
        .
        .
    fig.canvas.mpl_connect('button_press_event', clic)
```

On pourra insérer l'instruction `print(num)` dans la fonction `clic` pour tester la validité des résultats obtenus.

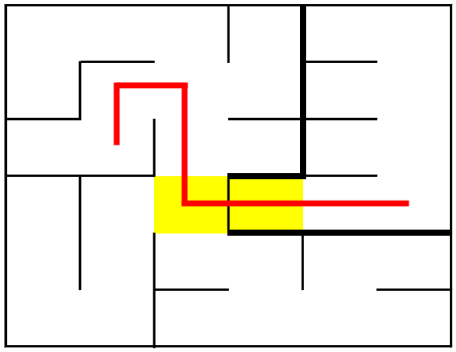
2) Avant de donner le principe de l'algorithme qui va permettre de tracer un chemin reliant deux cases choisies par l'utilisateur, il est déjà important de justifier qu'un tel chemin existe **toujours**.

En fait, le labyrinthe construit dans l'exercice 2 a la propriété que deux cases peuvent toujours être reliées par un chemin.

Montrons-le par l'absurde. Supposons qu'il existe deux cases ne pouvant pas être reliées par un chemin, par exemple comme pour les deux cases bleues du labyrinthe ci-dessous :



Pour relier ces deux cases, un chemin devrait donc forcément traverser un ou plusieurs murs séparant des cases ne communiquant pas dans le labyrinthe. Donnons-nous un tel chemin. Il passe donc par deux cases contiguës, séparées par un mur, ne communiquant pas dans le labyrinthe (en jaune ci-dessous).

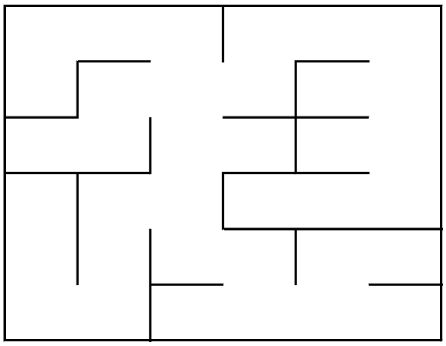


Lors du traitement de ce mur par l'algorithme de la fonction `LabyAleatoire` de l'exercice 2, il aurait dû être détruit (car tous les murs présents dans le labyrinthe final étaient déjà présents à ce moment là (en particulier ceux en gras sur le dessin), et donc ces deux cases jaunes ne communiquaient pas), ce qui est contradictoire.

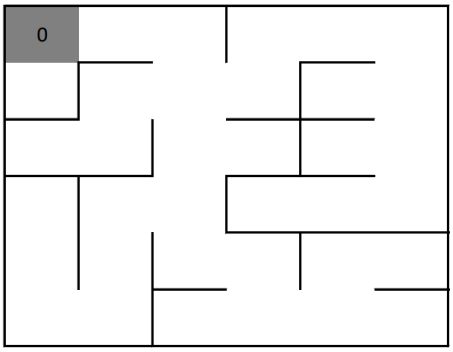
Passons à présent à l'algorithme permettant de relier deux cases arbitraires du labyrinthe par un plus court chemin.
Le principe sera le suivant : marquer successivement chaque case du labyrinthe de sa distance à la case de départ, en partant de la case de départ. Pour faire cela on effectuera ce qu'on appelle un parcours en largeur du graphe correspondant au labyrinthe (en anglais : "breadth first search") : on marque la première case (celle de départ), puis ses voisines (celles à distance 1), puis les voisines des voisines (celles à distance 2), etc.
La structure de donnée importante à utiliser est ce qu'on appelle une file, c'est-à-dire une liste dans laquelle on fera plusieurs fois les opérations suivantes :

- ajouter un élément en fin de liste,
- traiter puis supprimer l'élément en début de liste.

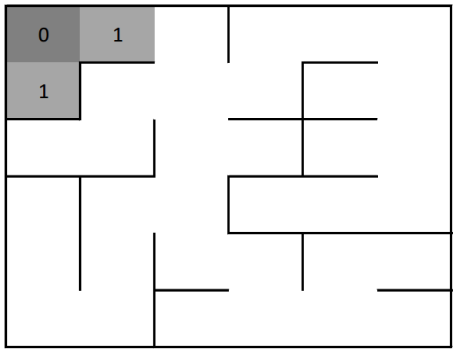
Cette file contiendra successivement le numéro de la case correspondant au départ (à distance 0 de la case de départ), les numéros des cases à distance 1 du départ, les numéros des cases à distance 2, etc.
On a illustré le fonctionnement du parcours en largeur sur un exemple (voir ci-dessous) où la case de départ est l'entrée du labyrinthe (case du coin supérieur gauche) et la case d'arrivée est la case du coin inférieur droit.
On a noté $a_i, b_i, c_i, \ldots$ les numéros des cases à distance i de l'entrée, selon leur ordre d'entrée dans la file. Sur le labyrinthe, on a marqué les cases traitées de leur distance à l'entrée du labyrinthe. Enfin, les cases en gris foncée sont les cases en cours de traitement (celles dont on cherche les cases voisines à insérer à l'étape suivante dans la file).



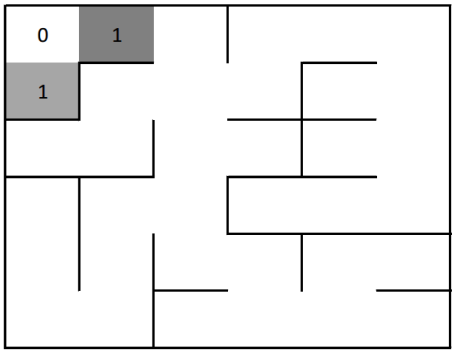
File = []



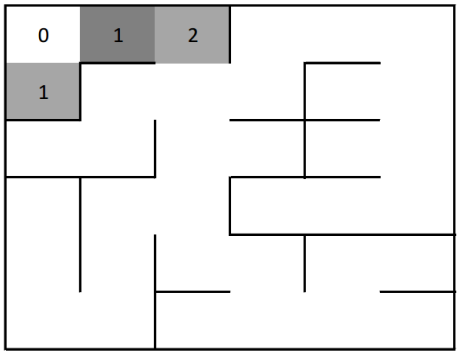
Ajout de a_0
File = $[a_0]$



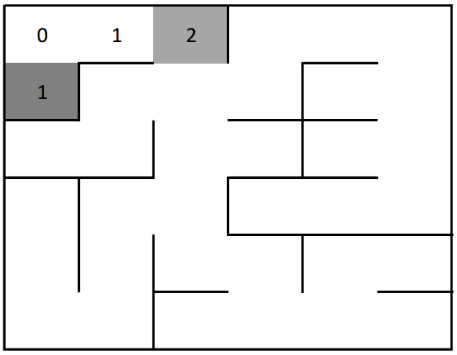
Ajout des voisins de a_0
File = $[a_0, a_1, b_1]$



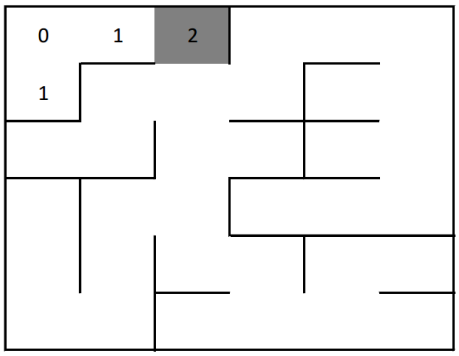
Suppression de a_0
File = $[a_1, b_1]$



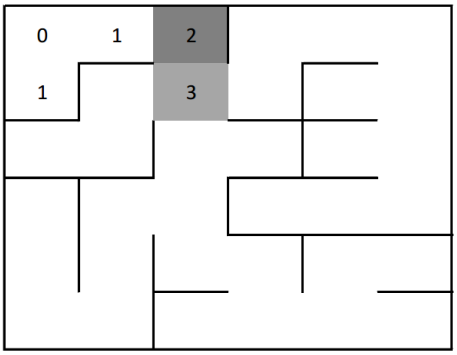
Ajout du voisin de a_1
File = $[a_1, b_1, a_2]$



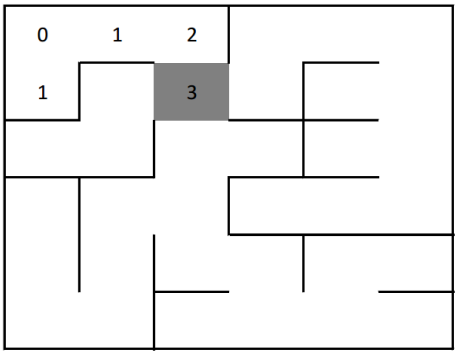
Suppression de a_1
File = $[b_1, a_2]$



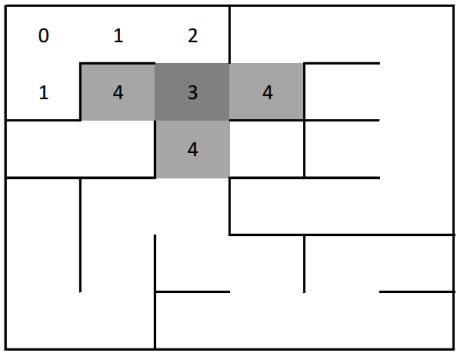
Suppression de b_1
File = $[a_2]$



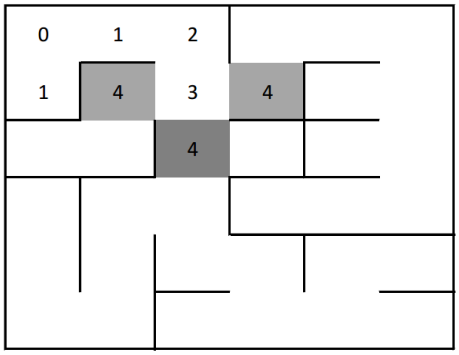
Ajout du voisin de a_2
File = $[a_2, a_3]$



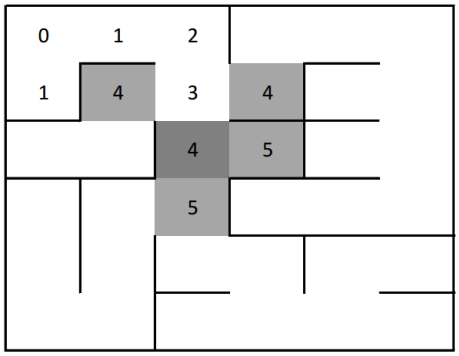
Suppression de a_2
File = $[a_3]$



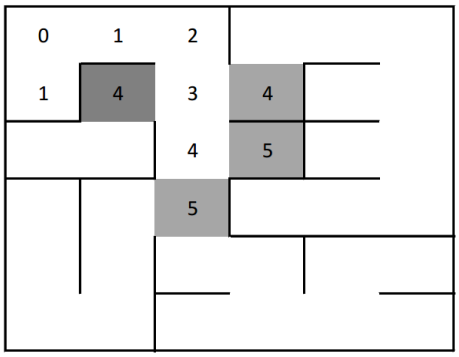
Ajout des voisins de a_3
File = $[a_3, a_4, b_4, c_4]$



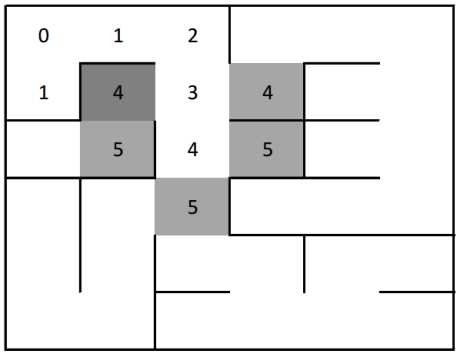
Suppression de a_3
File = $[a_4, b_4, c_4]$



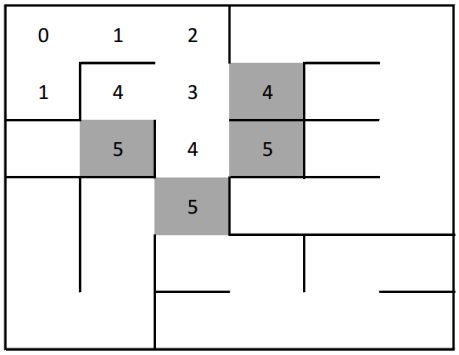
Ajout des voisins de a_4
File = $[a_4, b_4, c_4, a_5, b_5]$



Suppression de a_4
File = $[b_4, c_4, a_5, b_5]$



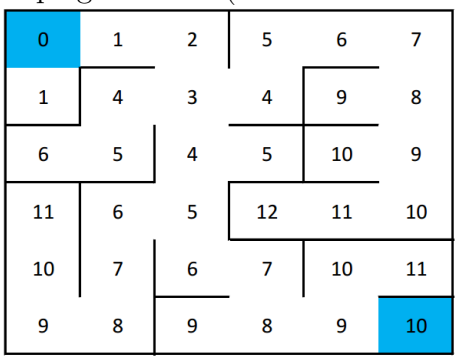
Ajout du voisin de b_4
File = $[b_4, c_4, a_5, b_5, c_5]$



Suppression de b_4
File = $[c_4, a_5, b_5, c_5]$

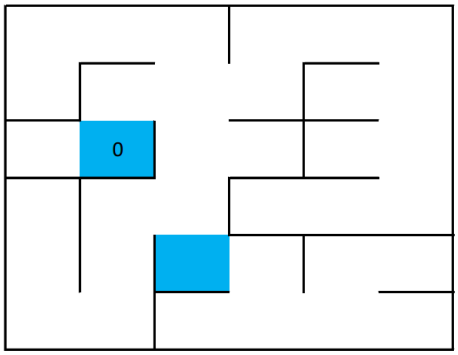
etc.

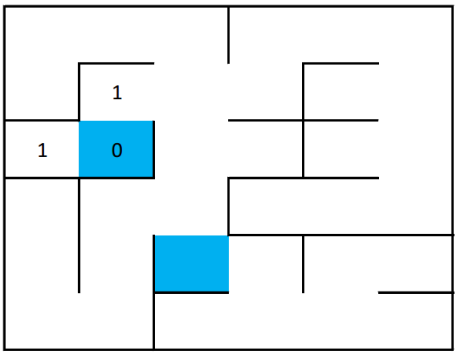
On aboutit finalement au marquage suivant (avec la case de départ et d'arrivée en bleu) :

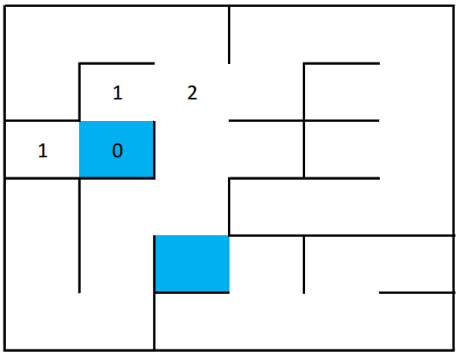


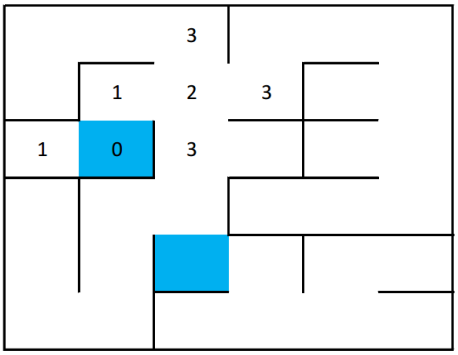
File = [] (la file est vide à la fin de l'algorithme dans ce cas)

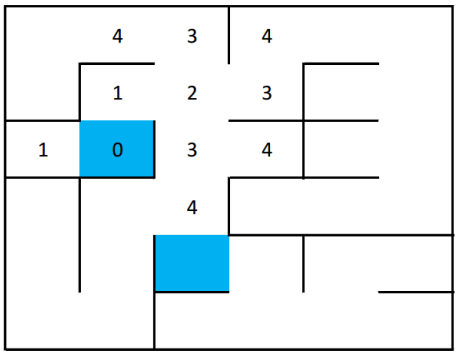
Pour parcourir les voisins d'une case donnée, on utilisera la liste d'adjacence `L` représentant le labyrinthe : par exemple, les voisins d'une case i seront tout simplement les éléments de la sous-liste `L[i]`.
On est sûr, à la fin du processus, que la case de départ et d'arrivée seront marquées de leur distance à la case de départ (car elles sont toutes accessibles à partir de la case de départ comme cela a été expliqué plus haut).
Ci-dessous, un autre exemple avec deux autres cases pour le départ et l'arrivée :

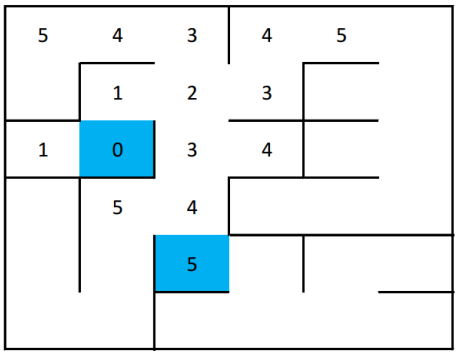




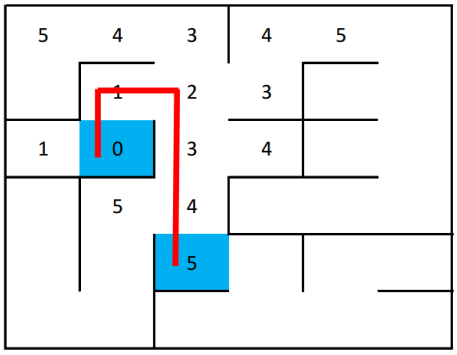








Pour tracer le chemin le plus court reliant les deux cases de départ et d'arrivée, il suffira de partir de la case d'arrivée puis de "remonter" jusqu'à l'*unique* case marquée 0 (c'est-à-dire la case de départ) en suivant les distances décroissantes (5, 4, 3, 2, 1 et 0 dans le deuxième exemple) :



a) Écrire une fonction Python qui :

- prend en entrée deux numéros n_1 et n_2 de noeuds d'un graphe G non pondéré, non orienté, ainsi que la représentation de G sous forme de liste d'adjacence.

- marque tous les noeuds de G, partant de n_1, de leur distance au noeud n_1 jusqu'au marquage du noeud n_2 (*cf* deuxième exemple ci-dessus), en suivant l'algorithme BFS ("Breadth First Search") décrit ci-dessus.

  Pour effectuer ce marquage, on utilisera une liste `Marques` à n éléments (n étant le nombre de noeuds de G). `Marques[k]` sera alors la distance séparant le noeud n_1 du noeud `k` si le noeud n_2 n'a pas été encore atteint.

  Au début, on aura donc `Marques[`n_1`]`$= 0$ et `Marques[k]`$= +\infty$ pour tous les autres noeuds, puis, pour tout noeud j voisin de n_1, `Marques[`j`]`$= 1$, etc.

  On utilisera également une file comme il a été expliqué. Cette file sera une liste initialisée à $[n_1]$ dans laquelle on va insérer tous les noeuds en cours (ou en attente) de traitement et retirer ceux qui ont déjà été traité (*cf* exemple 1 illustré).

  Le pseudo-code à traduire sera alors essentiellement le suivant :

  > Tant que le noeud n_2 n'a pas été marqué
  > prendre et supprimer le noeud `u` au début de la file
  > `distance = Marques[u] + 1`
  > mettre les voisins non marqués de ce noeud en fin de file et les marquer de `distance`

- La fonction donnera la liste `Marques` obtenue en sortie.

b) Écrire une fonction Python qui affiche la liste des noeuds d'un chemin le plus court de n_2 à n_1. Plus précisément, cette fonction devra :

- prendre en entrée :

  - deux numéros n_1 et n_2 de noeuds d'un graphe G non pondéré, non orienté,

  - la représentation de G sous forme de liste d'adjacence,

  - la liste `Marques` décrite dans la question précédente (et donnée en sortie par la fonction précédente avec n_1 et n_2 en entrée).

- donner en sortie la liste $[n_2, x_0, x_1, x_2, \ldots, x_p, n_1]$ des noeuds de G telle que

  x_0 est un noeud voisin de n_2,

  x_{j+1} est un noeud voisin de x_j (pour $0 \leqslant j \leqslant p-1$)

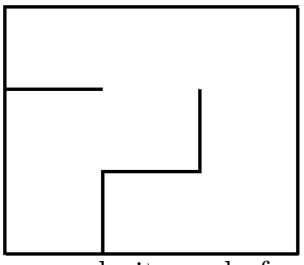
  n_1 est un noeud voisin de x_p

  et, si d est la distance de n_2 à n_1, alors $d-1$ est la distance de x_0 à n_1, $d-2$ est la distance de x_1 à n_1, $\ldots$, 1 est la distance de x_p à n_1.

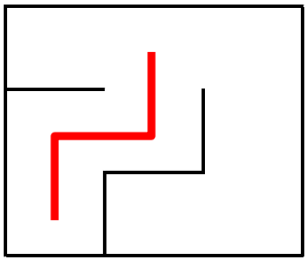
  (ainsi $d = 2+p$ ou $d = 1$ si les noeuds "intermédiaires" de n_1 à n_2 n'existent pas, ce qui est le cas si n_2 et n_1 sont voisins, ou $d = 0$ si $n_2 = n_1$).

  Dans le cas où $n_1 = n_2$, la liste en sortie sera réduite à $[n_1]$.

3) a) Écrire une fonction Python qui, à partir de n (la largeur et longueur du labyrinthe), et d'une liste L de numéros de cases successivement voisines de ce labyrinthe, donne la liste des abscisses et des ordonnées des points du chemin reliant ces cases (formé de lignes brisées horizontales ou verticales passant par les centres des cases).
Par exemple, si $n = 3$ et $L = [0, 3, 4, 7]$ avec le labyrinthe ci-dessous,



le chemin correspondant serait



et on voudrait que la fonction donne en sortie :
$X =$`[0.5,0.5,1.5,1.5]` et $Y =$`[0.5,1.5,1.5,2.5]`.

Cette fonction pourra utiliser la fonction `CoordCase` de l'exercice 1, question 1) b).
b) Écrire la fonction ci-dessous dans votre éditeur, à la suite des précédentes fonctions :

```
def Trace(fig,n):
    global num, x, y, compteur, X, Y, lignes
    num = 0
    compteur = 0
    X, Y = [], []
    lignes, = plt.plot([],[])
    def clic(event):
        global num, coord, compteur, liste, X, Y, lignes
        num = numCase(int(event.xdata),int(event.ydata),n)
        x, y = CoordCase(num,n)
        compteur += 1
        if compteur%3 > 0:
            X.append(x)
            Y.append(y)
            if compteur%3 == 2:
                lignes.set_xdata(X)
                lignes.set_ydata(Y)
                plt.draw()
        else:
            X, Y = [], []
            lignes.set_xdata(X)
            lignes.set_ydata(Y)
            plt.draw()
    fig.canvas.mpl_connect('button_press_event', clic)
```

puis faire appel à la fonction `LabyAleatoire` (question 3 de l'exercice 2) et à la fonction `Trace` précédente en tapant dans la console

```
a, fg = LabyAleatoire(15)
Trace(fg,15)
```

Cliquer alors plusieurs fois sur des cases du labyrinthe et constater ce qui se passe.
Quelques explications :
`lignes, = plt.plot([],[])` permet de générer un objet de type 'matplotlib.lines.Line2D' initialisée comme étant vide.
Si `X` est une liste d'abscisses et `Y` une liste d'ordonnées, les instructions :
`lignes.set_xdata(X)`
`lignes.set_ydata(Y)`
ont alors l'effet de mettre à jour `lignes` pour qu'il soit équivalent à `plt.plot(X,Y)` (ou plus précisément son contenu).
L'instruction `plt.draw()` a ici pour effet de réaliser effectivement le tracé demandé.
Essayer de comprendre le code de la fonction `Trace`. En particulier, quel est le rôle de la variable compteur ?
c) Une fois la fonction `Trace` bien comprise, essayer de la modifier pour écrire une fonction `Trace_Chemin` prenant en entrée `fig` et la liste d'adjacence G représentant le graphe du labyrinthe. Si cette fonction est appelée à la suite de `LabyAleatoire`, l'utilisateur doit pouvoir voir s'afficher un plus court chemin d'une case du labyrinthe à une autre, cases qu'il aura choisies avec la souris. Il lui sera aussi indiqué à chaque fois la longueur d'un tel chemin et il pourra répéter l'expérience autant de fois qu'il le veut.
Cette fonction `Trace_Chemin` fera appel aux fonctions des questions 2) a), 2) b) et 3) a) de cet exercice 5 et elle pourra avoir pour squelette :

```
def Trace_Chemin(fig,n):
    global compteur, ... ...
    .
    .
    .
    def clic(event):
        global compteur, ... ...
        .
        .
        .
        if compteur%3 > 0:
            .
            .
            if compteur%3 == 2:
                .
                .
                .
        else:
            .
            .
            .
    fig.canvas.mpl_connect('button_press_event', clic)
```

Exercice 6 :

Les fonctions de la question 2 de l'exercice 3 et de la question 1 de l'exercice 5 sont à présent inutiles et peuvent être effacées.

Écrire à présent la fonction Python principale du projet (celle articulant toutes les précédentes) :
cette fonction ne prendra pas d'argument, demandera à l'utilisateur la largeur n du labyrinthe, affichera le labyrinthe puis permettra à l'utilisateur de cliquer plusieurs fois sur deux cases pour avoir le plus court chemin entre ces deux cases, ainsi que sa longueur.
Cette fonction fera donc appel à la fonction `LabyAleatoire` de l'exercice 2 (question 3), à la fonction `Liste_Laby` de l'exercice 4 (question 2) et à la fonction `Trace_Chemin` du 3) c) de l'exercice 5.

Extensions possibles :

- Faire apparaître une entrée et une sortie au labyrinthe (en haut à gauche et en bas à droite par exemple).
- Généraliser la fonction `LabyAleatoire` pour permettre la gestion des labyrinthes rectangulaires (et pas seulement les labyrinthes carrés). Il faudrait surtout alors reprendre les fonctions de l'exercice 1.
- Sur plusieurs simulations, faire la moyenne des longueurs des plus courts chemins de l'entrée à la sortie du labyrinthe en fonction de la largeur n du labyrinthe. Tracer alors la valeur de ces moyennes en fonction de n (sous forme d'une ligne brisée ou d'un diagramme en barres). Il est possible d'automatiser ces simulations sous forme d'une fonction (plutôt que de les faire à la main). Commenter les résultats obtenus.
- Pour l'exercice 2, question 3, personnaliser la façon dont sont tirées sans remise les murs intérieurs du labyrinthe (on peut choisir les indices suivant la loi uniforme, ou alors suivant une loi binomiale par exemple, ou même une loi de probabilité personnalisée...). Voir alors l'effet sur le labyrinthe...