Ce sujet est adapté des épreuves d'informatique issues des concours (Mines-Ponts, MP, 2016) et (ENS, MP, 2020).

Les deux parties sont indépendantes.

I Modélisation de la propagation d'une épidémie

L'étude de la propagation des épidémis joue un rôle important dans les politiques de santé publique. Les modèles mathématiques ont permis de comprendre pourquoi il a été possible d'éradiquer la variole à la fin des années 1970, et pourquoi il est plus difficile d'éradiquer d'autres maladies comme la poliomyélite ou la rougeole. Ils ont également permis d'expliquer l'apparition d'épidémies de grippe tous les hivers. Aujourd'hui, des modèles de plus en plus complexes et puissants sont développés pour prédire la propagation d'épidémies à l'échelle planétaire telles que le SRAS, le virus H5N1 ou le virus Ebola. Ces prédictions sont utilisées par les organisations internationales pour établir des stratégies de prévention et d'intervention.

Dans tout le problème, on suppose que les bibliothèques `numpy` et `random` ont été importées par

```
import numpy  as np
import random as rd
```

On s'intéresse ici à une méthode de simulation numérique, dite par *automates cellulaires*. Dans ce qui suit, on appelle *grille* de *taille* $n \times n$ une liste de n listes de longueur n, où n est un entier strictement positif.

Pour mieux prendre en compte la dépendance spatiale de la contagion, il est possible de simuler la propagation d'une épidémie à l'aide d'une grille. Chaque case de la grille peut être dans un des quatres états suivants : saine, infectée, rétablie, décédée. On choisit de représenter ces quatre états par les entiers :

0 (Sain), 1 (Infecté), 2 (Rétabli) et 3 (Décédé).

L'état des cases d'une grille évolue au cours du temps et selon des règles simples. On considère un modèle où l'état d'une case à l'instant $t+1$ ne dépend que de son état à l'instant t et de l'état de ses huit cases voisins à l'instant t (noter qu'un case du bord n'a que cinq cases voisins, et trois pour une case d'un coin). Les *règles de transmission* sont les suivantes:

- une case décédée reste décédée;
- une case infectée devient décédée avec une probabilité p_1 ou rétablie avec une probabilité $(1-p_1)$;
- une case rétablie reste rétablie;
- une case saine devient infectée avec une probabilité p_2 si elle a au moins une case voisine infectée et reste saine sinon.

On initialise toutes les cases dans l'état sain, sauf une case choisie au hasard dans l'état infecté.

► **Question 1.** *Écrire en `Python` une fonction `grille(n)` qui retourne une grille de taille $n \times n$. Vous utiliserez pour cela des listes en compréhension.*

► **Question 2.** *Écrire en `Python` une fonction `init(n)` qui construit une grille G de taille $n \times n$ ne contenant que des cases saines puis choisit aléatoirement une des cases et la transforme en case infectée, avant de renvoyer la grille G ainsi modifiée. On pourra utiliser la fonction `randrange(p)` de la biliothèque `random` qui, pour un entier positif `p`, renvoie un entier choisi aléatoirement entre 0 et $p-1$ inclus.*

▶ **Question 3.** *Écrire en Python une fonction `compte(G)` qui a pour argument une grille `G` et renvoie la liste `[n0, n1, n2, n3]` formée des nombres de cases dans chacun des quatre états.*

D'après les règles de transitions, pour savoir si une case saine peut devenir infectée à l'instant suivant, il faut déterminer si elle est exposée à la maladie, c'est-à-dire si elle possède au moins une case infectée dans son voisinage. Pour cela, un étudiant propose en Python la fonction `est_exposee(G,i,j)` suivante.

```
def est_exposee(G, i, j):
    var_x = [-1, 0, 1]
    var_y = [-1, 0, 1]
    liste_cellule_voisines = [G[i+v_x][j+v_y] for v_x in var_x for v_y in var_y]
    if (1 in list_cellule_voisines):
        return True
    else
        return False
```

▶ **Question 4.** *Quelles critiques pouvez vous émettre sur la production de l'élève ? Proposer une correction qui ne dénature pas la solution proposée. Vous mettrez en valeur, en utilisant les annotations de type Python, la signature de la fonction corrigée.*

Après cette question, on considère donné une fonction `est_exposee(G, x, y)` qui se comporte comme attendu, c'est-à-dire renvoie `True` si et seulement si une cellule du voisinage de `G[y][x]` est infectée.

▶ **Question 5.** *Écrire une fonction `suivant(G, p1, p2)` qui fait évoluer toutes les cases de la grille `G` à l'aide des règles de transition et renvoie une nouvelle grille correspondant à l'instant suivant. Vous expliciterez une fonctionn `bernouilli(p)` qui simule une variable aléatoire de Bernouilli de paramètre p: `bernouilli(p)` vaut 1 avec probabilité `p` et 0 avec probabilité `(1-p)`. On rappelle que la fonction `random` de la bibliothèque `random` renvoit un flottant dans l'intervalle* $[0, 1)$.

▶ **Question 6.** *Soucieux de la complexité spatiale du programme, un élève vous demande si l'on pourrait effectuer la modification de la grille "en place". Que lui répondez-vous ?*

Avec les règles de transition du modèle utilisé, l'état de la grille évolue entre les instants t et $t+1$ tant qu'il existe au moins une case infectée.

▶ **Question 7.** *Écrire en Python une fonction `simulation(n, p1, p2)` qui réalise une simulation complète (une simulation s'arrête lorsque la grille n'évolue plus) avec une grille de taille $n \times n$ pour les probabilités p_1 et p_2, et renvoie la liste `[x0, x1, x2, x_2]` formée des proportions de cases dans chacun des quatre états à la fin de la simulation.*

▶ **Question 8.** *Que vaut `x1` à la fin de la simulation ? Quelle relation vérifient `x0`, `x1`, `x2` et `x3` ? Comment obtenir la valeur `x_atteinte` de la propotion des cases qui ont été atteintes par la maladie pendant une simulation à partir du résultat de `simulation(n, p1, p2)` ?*

On fixe `p1` à 0.5 et on calcule la moyenne des résultats de plusieurs simulations pour différentes valeurs de `p2`. On obtient la courbe représentée en Figure 1.

On appelle *seuil critique de la pandémie* la valeur de `p2` à partir de laquelle plus de la moitié de la population a été atteinte par la maladie à la fin de la simulation.

▶ **Question 9.** *Vous soumettez le problème d'estimer le seuil critique de la pandémie à vos élèves. Certains d'entre eux suggèrent, se rappelant d'une leçon de l'année passée, d'utiliser une recherche dichotomique. Après avoir rappelé l'algorithme de recherche dichotomique (appliqué à la situation) et sa complexité, vous discuterez brièvement de la pertinence de cette proposition – comme vous le feriez devant vos élèves.*

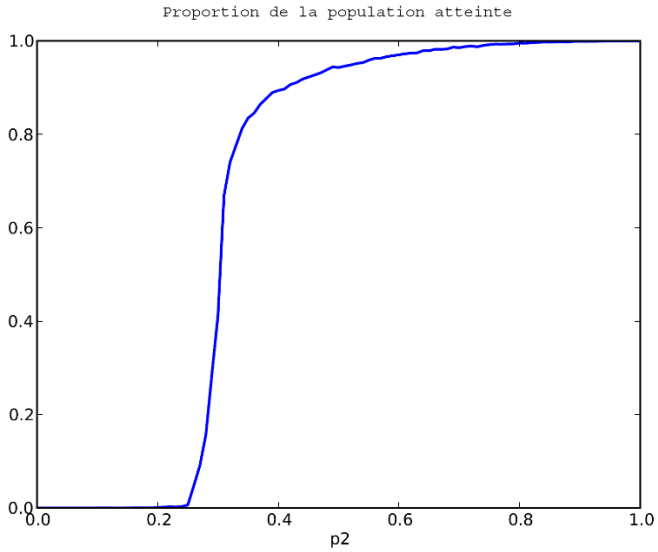


Figure 1: Représentation de la proportion de la population qui a été atteinte pendant la simulation en fonction de la probabilité p_2.

Pour finir, vous souhaitez faire étudier (de façon simpliste) l'effet d'une campagne de vaccination à vos élèves. L'idée est d'utiliser une fonction d'initialisation de grille `init_vac(n, q)` qui initialise une grille où une fraction q de la population est déjà *rétablie*, et une cellule est infectée.

▶ **Question 10.** *Proposer, comme vous le feriez pour vos élèves, un énoncé demandant d'écrire en* ***Python*** *une fonction* ***init_vac(n, q)****. Celui-ci prendra la forme d'un code* ***faux*** *pour* ***init_vac(n, q)****, d'une remarque/question permettant aux élèves de réaliser le problème afin de le corriger. Vous l'accompagnerez d'une brève correction.*

▶ **Question 11.** *Vous avez appris qu'un de vos étudiants venait de perdre un de ses parents à la suite d'une contagion à un virus épidémique. Proposer une adaptation (thématique) du sujet à la situation. Quelles extensions d'activité pourriez-vous alors envisager ?*

II Constructions et explorations de labyrinthes

Nous nous intéressons ici à l'étude de labyrinthes. Une première sous-partie s'attache à la génération de labyrinthes *parfaits*. Une seconde étudie des algorithmes pour explorer des labyrinthes, possiblement en présence de monstres.

On considère des graphes non orientés, sans boucles (sans arête d'un sommet vers lui-même). On rappelle qu'un sous-graphe de G est un graphe H tel que $\mathsf{sommets(H)} \subset \mathsf{sommets(G)}$ et $\mathsf{aretes(H)} \subset \mathsf{aretes(G)}$. Pour un graphe G et un sous-ensemble X de sommets de G, on définit le sous-graphe de G *induit* par X comme le graphe dont les sommets sont X et dans lequel deux sommets sont reliés s'ils sont reliés dans G. Pour deux entiers n et m, la *grille* de taille $n \times m$ est le graphe dont les sommets sont $\{0, 1, \cdots, n \times m - 1\}$ et dont les arêtes sont les paires de sommets qui sont

- soit de la forme $(v, v+1)$ pour $0 \leq v < nm$ tel que $v \mod m \neq m-1$,
- soit de la forme $(v, (v+m))$ pour $0 \leq v < (n-1)m$.

▶ **Question 12.** *Représenter la grile de taille* 3×5.

Pour un entier $n > 0$, la fonction `randrange(n)` du module `random` renvoie un entier tiré uniformément dans l'ensemble $\{0, 1, \cdots, n-1\}$.

On représentera un graphe de n sommets en `Python` par sa liste d'adjacence, c'est-à-dire par une liste `graphe` de taille n telle que `graphe[k]` contiennent la liste des voisins du sommets `k`. Le nombre de sommets du graphe est donné par `len(graphe)`.

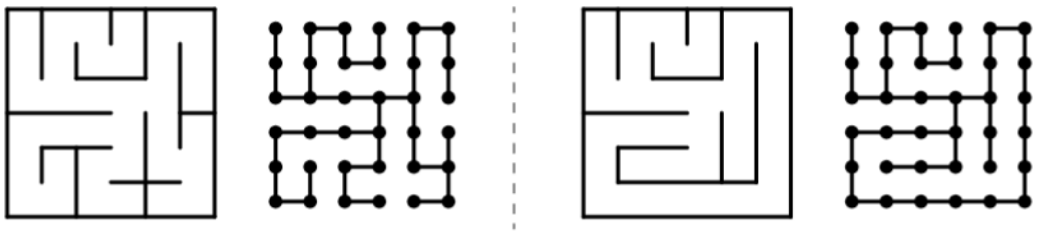
On suppose données plusieurs fonctions auxiliaires:

- `ajoute_arete(G: list, bout1: int, bout2: int) -> NoneType` qui ajoute au graphe `G` une arête entre les sommets `bout1` et `bout2` si elle n'est pas déjà présente. On supposera que `bout1` et `bout2` sont des élements de $\{0, \cdots, n-1\}$.
- `graphe_vide(n: int) -> list` qui renvoie un nouveau graphe à n sommets et sans aucune arête.
- `aretes(G: list) -> list` qui renvoie la liste de toutes les arêtes d'un graphe donné, sous la forme de paires d'entiers. Le graphe état non orienté, la paire (v, w) et la paire (w, v) apparaissent toutes deux dans ce tableau.

Les deux sous-parties qui suivent sont indépendantes.

II.1 Construction de labyrinthes

On se donne un graphe G connexe. On appelle *labyrinthe* sur G un sous-graphe connexe de G qui a le même ensemble de sommets que G. Par exemple, on peut partir d'un graphe G dont les sommets sont les cases d'une grille rectangulaire et dont les arêtes correspondent aux cases adjacentes (nord, sud, est, ouest). Les arêtes du labyrinthe correspondent alors aux ouvertures permettant de passer d'une case à une autre. Les arêtes de g qui ne sont pas dans le labyrinthe correspondent aux murs. Voici deux exemples :



On dit qu'un labyrinthe est *parfait* lorsqu'il existe un unique chemin entre toute paire de sommets. Dans l'exemple ci-dessus, seul le labyrinthe de gauche est parfait. Dans cette partie, on cherche à construire des labyrithes parfaits.

II.1.1 Mélange de Knuth

L'algorithme du *mélange de Knuth* permet de permuter aléatoirement les éléments d'un tableau. Pour un tableau `tab` de taille `n`, cet algorithme effectue `n` échanges : pour $O \leq$ `i` $<$ `n`, la `i`-ème étape échange l'élément `tab[i]` avec l'élement `tab[j]`, pour un entier `j` pris au hasard entre 0 et `i` inclus. On admettra que ce mélange effectue une permutation aléatoire *uniforme* du tableau, c'est-à-dire que toutes les permutations sont équi-probables.

▶ **Question 13.** *Écrire en `Python` une fonction `melange_Knuth(tab: list) -> NoneType` qui reçoit en argument un tableau et le permute aléatoire avec le mélange de Knuth.*

II.1.2 Parcours en profondeur aléatoire

Pour construire un labyrinthe aléatoire sur g, on peut utiliser l'algorithme suivant. On effectue *parcours en profondeur* (noté DFS pour depth-first-search par la suite) du graphe G. Lorsqu'un somme v est traité, on permute aléatoirement la liste de ses voisins et on la parcourt. Pour chaque voisin w de v qui n'a pas encore été visité, on ajoute l'arête (v, w) au labyrinthe et on traite w.

▶ **Question 14.** *Écrire une fonction* `labyrinthe1(G: list) -> list` *qui prend en argument un graphe G connexe et renvoie un labyrithe sur G construit avec un DFS aléatoire. Celle-ci s'appuiera sur une sous-fonction récursive.*

▶ **Question 15.** *En identifiant un invariant du parcours en profondeur réalisé par* `labyrinthe1`*, montrer que la labyrinthe construit par la fonction précédente est parfait.*

II.1.3 Labyrinthe issus de classes d'équivalence

On s'intéresse maintenant à une structure de données qui sera utilisée par la suite pour construire un labyrinthe parfait, l'*union-find*. Cette structure de données représente une relation d'équivalence sur l'ensemble $\{0, \cdots, n-1\}$. Pour cela, on choisit un représentant arbitraire dans chaque classe d'équivalence et on se donne un tableau `lien` de taille `n` qui va permettre de retrouver ce représentant. Pour un représentant `r`, on a `lien[r]==r`. Pour tout autre élément `i` de la classe de `r`, on aboutit à `r` en suivant les valeurs données par le tableau `lien` : `lien[i]==r`, ou `lien[lien[i]]==r`, ou `lien[lien[lien[i]]]==r`, etc. Par exemple, le tableau

```
lien = [2, 1, 5, 3, 3, 5, 5]
```

représente la relation d'équivalence dont les classes sont $\{0, 2, 5, 6\}$, $\{0\}$ et $\{3, 4\}$, pour lesquelles on a choisi les représentants 5, 1 et 3 respectivement.

▶ **Question 16.** *Écrire une fonction* `ce_representant(lien: list, e: int) -> int` *qui prend en arguments une relation d'équivalence sur $\{0, \cdots, n-1\}$ et un entier $0 \leq i < n$, et renvoie le représentant de la classe i. Quelle est sa complexité temporelle dans le pire cas ?*

▶ **Question 17.** *Écrire une fonction* `ce_union(lien: lien, ea: int, eb: int) -> NoneType` *qui prendre en argument une relation d'équivalence et qui fusionne les classes d'équivalence des élements* `ea` *et* `eb`*. Si* `ea` *et* `eb` *sont déjà dont la même classe, cette fonction ne fait rien. Sinon, elle choisit arbitrairement un nouveau représentant*[1]*.*

On utilise maintenant la structure de classes d'équivalences pour constuire un labyrinthe parfait H à partir d'un graphe G connexe à n sommets. L'algorithme procède ainsi :

1. on construit une relation d'équivalence sur $\{0, \cdots, n-1\}$ où chaque classe est un singleton (l'ensemble des classes est $\{\{0\}, \cdots, \{n-1\}\}$);
2. on construit le tableau de toutes les arêtes du graphe G, puis on le mélange avec `melange_knuth`.
3. on parcourt ce tableau mélangé et, pour chaque arête (v, w), si v et w ne sont pas dans la même classe d'équivalence, on ajoute l'arête (v, w) au labyrinthe H et on fusionne les classes de v et w avec `ce_union`.

▶ **Question 18.** *Écrire une fonction* `labyrinthe2(G: list) -> list` *qui prend en argument un graphe G connexe et renvoie un labyrinthe sur G selon l'algorithme décrit ci-dessus.*

▶ **Question 19.** *Montrer que l'étape 3 de l'algorithme maintient l'invariant suivant : pour toute classe d'équivalence X, le sous-graphe de H induit par X est un labyrinthe parfait du sous-graphe de G induit par X. En déduire que le labyrinthe construit par la fonction* `labyrinthe2` *est parfait.*

[1] Ce n'est pas ce qui est fait en pratique: des stratégies d'union plus fines permettent d'améliorer considérablement la complexité de `ce_representant` en limitant la longueur des *chaines de liens*.

II.2 Exploration de labyrinthes

On se pose maintenant la question de résoudre un labyrinthe, c'est-à-dire de chercher un chemin d'un sommet source `src` donné à un sommet destination `dst` donné. On suppose toujours que labyrinthe est connexe, mais on ne considère plus uniquement des labyrinthes parfaits: il peut exister plusieurs chemin de la source à la destination.

▶ **Question 20.** *En vous inspirant de l'algorithme de parcours en largeur, donner un algorithme renvoyant la longueur d'un plus court chemin entre* `src` *et* `dst` *dans le labyrinthe* `G`*. Vous préciserez bien les entrées et sorties de l'algorithme, ainsi que les structures de données utilisées.*

▶ **Question 21.** *Montrer que cet algorithme renvoie bien la longueur d'un plus court chemin de la source* `src` *à la destination* `dst`*, comme vous le feriez devant vos élèves.*

On considère maintenant qu'il y a des monstres dans le labyrinthe (les sommets `src` et `dst` pouvant tout à fait en accueillir). Notre but est maintenant de chercher des chemins du sommet source `src` au sommet destination `dst` qui passent par un minimum de monstres.

▶ **Question 22.** *Adapter l'algorithme que vous avez proposé plus haut pour qu'il retourne un chemin dans* `G` *entre* `src` *et* `dst` *passant par un minimum de montres.*

▶ **Question 23.** *Adapter l'algorithme pour qu'il propose le plus court chemin entre* `src` *et* `dst` *dans* `G` *parmi ceux passant par un minimum de monstres. Comment s'appelle l'ordre que vous avez introduit sur les distances considérées dans ce dernier algorithme ?*

••• FIN •••