TD/TP : MANIPULATION DE PILES

Objectifs du TD/TP

- Manipulation la notion (abstraite) de pile informatique indépendamment de son implémentation.

Les implémentations des piles vues en cours se trouvent à votre disposition dans les fichiers `pile_bornee.py` et `pile_non_bornee.py`.

Pour utiliser l'une d'elles, importer le contenu du fichier avec `from ... import *` (**pas de copier-coller**).

L'idée est de n'utiliser **que** les primitives basiques sur les piles (la barrière d'abstraction...) : `empile`, `depile` et `est_vide`. On pourrait aussi utiliser directement des listes python et se limiter à `append()`, `pop()` et `== []`.

Chauffe

1. Écrire une fonction `copie(p)` qui renvoie une copie de la pile p, laissant celle-ci dans le même état que son état initial.

Quelle est la complexité temporelle et spatiale ?

2. Écrire une fonction `miroir(p)` qui renvoie la pile dont les éléments sont les éléments de p, dans l'ordre inverse.

Quelle est la complexité temporelle et spatiale si on autorise la modification de `p` ? Sinon ?

3. Écrire une fonction `taille(p)` calculant la taille d'une pile. Quelle est sa complexité temporelle ? spatiale ?

4. Écrire une fonction permettant de lire (sans l'extraire) le n^e élément d'une pile. On gérera le cas où la pile n'est pas assez grande.

5. Écrire une fonction qui prend une pile non vide en argument et place l'élément situé à son sommet tout au fond de la pile, en conservant l'ordre des autres éléments.

Quelle est sa complexité en temps et en espace ?

Construction de labyrinthes parfaits

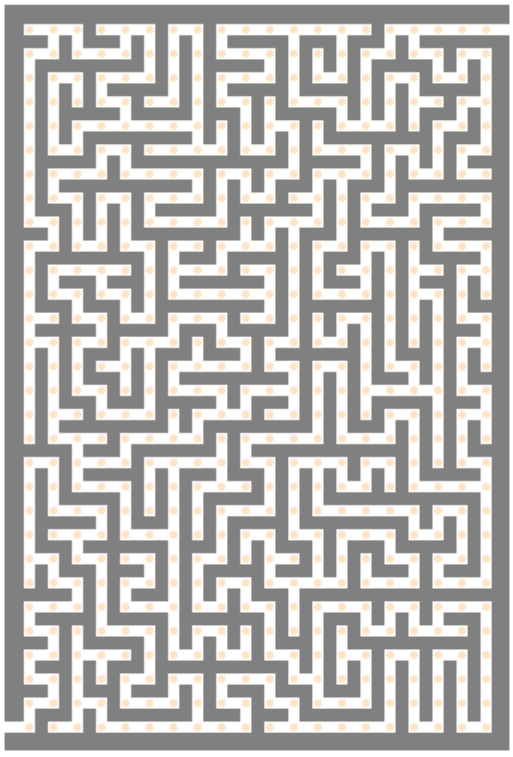
Le but de cette partie est de générer un labyrinthe parfait c'est-à-dire tel qu'il existe un et un seul chemin reliant deux points du labyrinthe.

On a arbitrairement choisi de partir d'en bas à gauche et d'arriver en haut à droite, mais cela fonctionnerait depuis et vers n'importe quel point.

Le labyrinthe représenté ci-contre est de taille 20×30, les points voisins de coordonnées entières sont reliés ou non par un trait blanc.

Pour les informaticiens, il s'agit d'un graphe, et parcourir le labyrinthe revient à faire un parcours (en profondeur, c'est plus intuitif) de ce graphe.

Nous allons donc construire des labyrinthes parfaits en parcourant les points en s'aidant d'une pile.



Utiliser l'entête suivante :

```
from random import randint
import matplotlib.pyplot as plt
```

Nous construirons ici un labyrinthe de format $n \times p$, dont les « cases » auront des coordonnées $(i, j) \in [\![0, n-1]\!] \times [\![0, p-1]\!]$.

1. Écrire une fonction `valide(i, j, n, p)` testant si `(i, j)` est un couple de coordonnées de points valide dans un labyrinthe de taille `(n, p)`.

On va utiliser une liste de listes de booléens `est_visite` tel qu'à chaque instant `est_visite[i][j]` permet de savoir si la case de coordonnées `(i, j)` a déjà été visitée.

2. Écrire une fonction `voisins(case, est_visite)` qui construit et renvoie la liste `liste_voisins` des cases voisines de `case` valides et qui n'ont pas encore été visitées.

3. Écrire une fonction `choisir(L)` qui prend en entrée une liste `L` et renvoie un élément aléatoire de la liste (utiliser `randint`).

4. On souhaite maintenant écrire une fonction `labyrinthe(n, p)` renvoyant un labyrinthe parfait de format (n, p). Le principe sera le suivant :

- On crée et maintient une liste de liste `est_visite[i][j]` comme décrit ci-dessus.
- On crée une pile `pile` contenant les cases (couples d'entiers) au fur et à mesure qu'on les rencontre. Au départ, elle contiendra `(0, 0)`, qui sera la première case visitée.
- On crée une liste (utilisée comme une pile) de `segments` de segments, c'est-à-dire de couples de cases voisines reliées par un chemin.
- Puis, jusqu'à épuisement de la pile, on dépile le sommet appelé `case`.
  - ⋆ S'il n'a plus de voisins disponibles, on ne fait rien.
  - ⋆ Sinon,
    - ◦ On choisit aléatoirement l'un des voisins de `case`, appelé `suivant`.
    - ◦ On le considère comme visité et on ajoute le segment de `case` à `suivant`.
    - ◦ On rempile `case` pour visiter ultérieurement ses autres voisins.
    - ◦ On empile `suivant` pour visiter ses voisins
- On renvoie la liste (pile) des segments.

5. On donne une fonction d'affichage en pièce jointe. Admirer le résultat.

6. Modifier la fonction `labyrinthe` pour qu'elle renvoie aussi la liste de listes `precedent` tel que `precedent[i][j]` soit la (seule) case qui permette d'arriver en `(i, j)` dans la construction du labyrinthe. En déduire une fonction qui détermine le chemin pour aller d'une case à une autre.

7. Modifier l'affichage pour afficher le chemin en rouge.