Objectif

- Révision des collections (List, Set, Map).
- Parcours & recherche dans des collections.
- Analyse de performance, notion de complexité (*O*(n), *O*(1), …).
- Révision algorithme de backtracking.

Énoncé du problème

Un labyrinthe a été modélisé dans une Map<Point, List<Point>>, un Point étant une case du labyrinthe *(coordonnées <x,y>)*. Pour chaque Point *(clé de la map)*, on a comme valeurs la liste des Points reliés à celui-ci *(c'est-à-dire la liste des passages possibles)*. S'il n'y a pas de passage entre 2 cases juxtaposées, c'est qu'il y a un mur entre celles-ci. On a également à disposition une List<Point> contenant la liste des cases contenant un fromage *(à trouver)*. Vous êtes chargé de développer 5 méthodes permettant de parcourir rapidement ce labyrinthe à la recherche des fromages.

Travail à effectuer : méthodes à coder (dans la classe PyRat)

- `boolean` **`fromageIci`**`(Point pos)`

  Regarde dans la liste des fromages s'il y a un fromage à la position pos.
  Indication : il faut effectuer une boucle de parcours dans la List<Point> fromages à la recherche de pos. Cette liste a du être soit passée en paramètre, soit stockée dans un attribut de la classe. La méthode contains effectue cette boucle de recherche.

- `boolean` **`fromageIci`**`_EnOrdreConstant(Point pos)`

  Idem précédent, mais avec une recherche en ordre constant *(accès direct sans boucle de recherche)*.
  Indication : c'est-à-dire qu'il ne faut pas effectuer de boucle (sinon, ce n'est plus en ordre constant !). La méthode contains effectue une boucle !!! Il faut donc avoir à disposition une structure de données permettant d'avoir un accès direct: par exemple une matrice boolean[][] tabFromages, ou un HashSet<Point>. Cette nouvelle structure doit avoir été remplie préalablement dans preprocessing.

- `boolean` **`passagePossible`**`(Point de, Point a)`

  Indique si le joueur peut passer de la position (du Point) « **de** » au point « **a** » *(s'il n'y a pas de mur)*.
  Indication : il faut regarder dans la Map<Point, List<Point>> laby si le point «de» existe, et s'il a dans sa liste de points reliés le Point «a».

- `boolean` **`passagePossible`**`_EnOrdreConstant(Point de, Point a)`

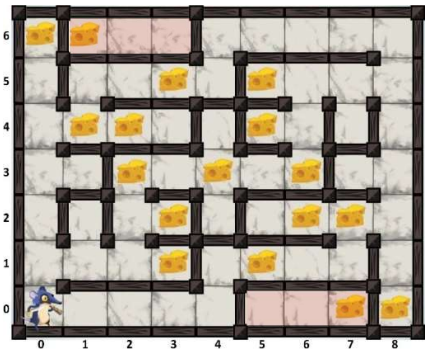
  Idem précédent, mais sans recherche du point « de » puis parcours des points reliés. Il vous faut donc avoir une seule structure de données *(idéalement un Set)* permettant de trouver immédiatement *(en **un seul** accès / une seule condition, et non pas chercher «de» puis «a» !)* s'il y a un passage entre « de » et « a ».
  Indication : il faut avoir à disposition une structure de données permettant d'avoir un accès direct à l'aide de la clé [de-a]. Un HashSet permet d'accéder sans boucle à une valeur d'après sa clé; la clé étant formée de plusieurs champs (2: de & a), il faut créer une structure de données contenant ces 2 champs, et l'utiliser comme clé d'accès à un HashSet correctement initialisé dans preprocessing.

- `List<Point>` **`pointsInatteignables`**`(Point pos)`

  Retourne la liste des points qui ne peuvent pas être atteints depuis la position « **pos** ».
  Indication : parcours complet de tout le labyrinthe (algorithme standard de backtracking), en conservant la liste des points visités. Ensuite, il faut retourner la liste de tous les Points qui n'ont pas été visités.

Remarques

- Vous pouvez modifier la méthode « **`turn`** » pour rajouter les tests que souhaitez effectuer.
- Vous pouvez rajouter tout ce que vous voulez dans la méthode « **`preprocessing`** » afin d'améliorer les performances de vos différentes méthodes.
- Vous pouvez rajouter des attributs dans la classe PyRat
  (ou éventuellement rajouter des paramètres dans les méthodes à développer).



Labyrinthe de test : afin de valider vos résultats, voici le labyrinthe utilisé :