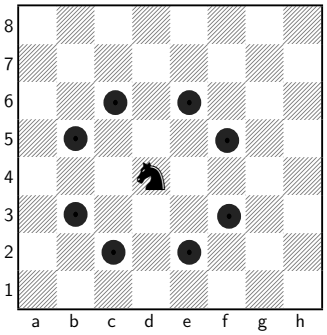
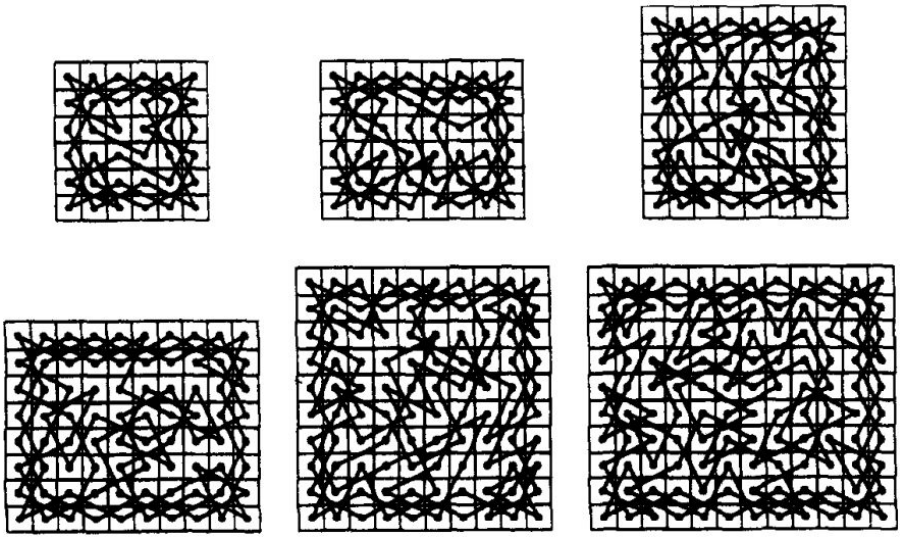
TP 3 - Backtracking (ou retour sur trace)

La ronde du cavalier



On considère un échiquier rectangulaire de dimensions $\ell \times h$ sur lequel on place un cavalier dont on rappelle que le mouvement a la forme d'un « L » imaginaire d'une longueur de deux cases et de largeur une case.
Le but de ce TP est d'écrire une fonction qui détermine une **ronde du cavalier**, c'est-à-dire un parcours de toutes les cases de l'échiquier depuis une position donnée. Cette ronde sera dite **fermée** si la dernière case atteinte permet de revenir à la première case par un mouvement du cavalier.

Si l'on voit l'ensemble des cases de l'échiquier comme les sommets d'un graphe et si l'ensemble des arêtes est formé par les couples de sommets à portée d'un mouvement du cavalier, il s'agit de trouver un **chemin hamiltonien** sur ce graphe.



Quelques exemples de rondes fermées sur des échiquiers
6×6, 8×6, 8×8, 10×8, 10×10 et 12×10

1 Quelques outils

On représente d'abord l'échiquier de taille $\ell \times h$ par une liste de h listes de longueur ℓ représentant les lignes de l'échiquier, de haut en bas. On utilisera l'alias `Case = tuple[int, int]` pour désigner une case par son couple de coordonnées dans cette représentation.

1) Définir une constante `DEPLACEMENTS` donnant la liste des couples (x, y) des coordonnées des vecteurs représentant les déplacements possibles du cavalier.

2) Écrire une fonction de signature `est_dans_l_echiquier(case: Case, l: int, h: int) -> bool` qui teste si une case (i, j) est dans l'échiquier de taille $\ell \times h$.

Pour la suite du TP, on préfère utiliser une unique liste de longueur ℓh pour représenter l'ensemble des cases de l'échiquier. Pour cela, on numérote les cases à partir de 0, de gauche à droite puis de haut en bas. On utilisera l'alias `Position = int` pour désigner un tel numéro de case, que l'on nommera désormais **une position**.

3) Écrire une fonction de signature `numero(case: Case, l: int) -> Position` qui prend les coordonnées

d'une case de l'échiquier ainsi que la largeur de l'échiquier et qui renvoie le numéro de la case, et une fonction de signature `coord(numero: Position, l: int) -> Case` qui fait le contraire.

4) Écrire une fonction de signature :

```
a_portee_depuis(origine: Position, l: int, h: int) -> list[Position]
```

qui prend une position `origine` et les dimensions de l'échiquier et renvoie la liste des positions à portée d'un mouvement du cavalier depuis la position `origine`.

2 Recherche récursive

Pour déterminer s'il existe une **ronde du cavalier** depuis une certaine `origine` sur un échiquier de taille $\ell \times h$, on va utiliser une fonction auxiliaire qui sera récursive.

Cette fonction auxiliaire aura pour paramètres un entier représentant la `position` actuelle du cavalier, ainsi qu'un dictionnaire représentant le chemin déjà parcouru depuis l'origine, dictionnaire `chemin` qui à chaque position atteinte fait correspondre le rang de celle-ci dans la ronde. Elle teste s'il est possible de terminer la ronde du cavalier en complétant ce chemin en utilisant l'algorithme suivant.

Si le chemin a déjà parcouru toutes les positions de l'échiquier, on renvoie la réponse `True`. Sinon, on crée un nouveau dictionnaire contenant le chemin déjà effectué et on lui ajoute l'une des positions à portée qui n'est pas encore parcourue (associée à son rang de passage). On fait alors un appel récursif avec comme paramètres cette nouvelle position et ce nouveau chemin. Si ces paramètres aboutissent à une solution, c'est-à-dire à une ronde du cavalier, on renvoie la réponse `True`. Sinon, on poursuit les tests avec la position à portée suivante. Si aucune position à portée n'aboutit à une solution, on renvoie `False`.

5) Quel est l'intérêt d'utiliser un dictionnaire, plutôt qu'une liste, pour stocker le chemin parcouru ?

6) Comment peut-on tester facilement si l'on a parcouru toutes les cases de l'échiquier ?

7) Avec un tel algorithme, quelle est la hauteur maximale de la pile de récursion ? Quelle est la taille totale de la mémoire temporaire nécessaire si on considère qu'un dictionnaire occupe une place proportionnelle au nombre de ses clés ?

Dans la suite de ce TP, on utilisera l'alias de type `Chemin = dict[Position]` qui servira à représenter un chemin effectué par le cavalier, comme expliqué ci-dessus.

On rappelle qu'on peut créer un nouveau dictionnaire qui est la réunion de deux dictionnaires par la syntaxe `nouveau_dico = dico1 | dico2` (valable en Python ≥ 3.9) ou bien par la syntaxe `nouveau_dico = {**dico1, **dico2}` (valable pour toute version de Python, mais moins claire).

8) Écrire une fonction de signature :

```
existe_ronde(origine: Position, l: int, h: int) -> bool
```

qui renvoie `True` s'il existe une ronde de cavalier sur un échiquier $\ell \times h$ depuis cette `origine` et `False` sinon.

Cette fonction contiendra donc la définition d'une fonction auxiliaire :

```
existe_rec(position: Position, chemin: Chemin) -> bool
```

qui prend pour paramètres une `position` et le `chemin` parcouru depuis l'origine jusqu'à la position actuelle et renvoie `True` si l'on peut compléter ce chemin en une ronde du cavalier et `False` sinon.

On pourra également créer auparavant une liste `voisins: list[list[Position]]` telle que pour chaque `position`, `voisins[position]` est la liste des positions qui sont à portée.

9) Améliorer les fonctions précédentes pour écrire une fonction de signature :

```
cherche_ronde(origine: Position, l: int, h: int) -> Chemin
```

qui renvoie une ronde de cavalier s'il en existe une et renvoie un dictionnaire vide sinon.

10) Tester les fonctions précédentes avec un échiquier 5×5 en prenant l'origine en 0 puis en 1. Vous devriez obtenir une réponse instantanément. Tester ensuite sur un échiquier 8×8 en choisissant 23 pour origine... et constatez que vous n'obtenez pas de réponse, même au bout de plusieurs minutes.

11) (facultatif) Écrire une fonction de signature :

```
affichage_texte(ronde: Chemin, l: int, h: int) -> None
```

qui prend pour paramètres une ronde et les dimensions de l'échiquier et qui affiche ligne par ligne le rang auquel chaque position est atteinte. On tâchera de faire en sorte que chaque case occupe la même taille en largeur dans l'affichage, afin de respecter l'alignement.

3 L'heuristique de Warnsdorff

L'inconvénient de l'algorithme de retour sur trace utilisé dans la partie précédente est qu'il parcourt parfois un très grand nombre de positions avant d'arriver dans une impasse. Par ailleurs, il dépend fortement de l'ordre dans lequel on parcourt les cases à portée pour les tester.

Pour remédier à ce problème, Warnsdorff a eu l'idée toute simple de trier les positions à portée et non encore atteintes (positions que l'on dira *accessibles*) dans l'ordre croissant du nombre de positions accessibles depuis elles. Ainsi, on commence par tester les chemins qui ont le moins d'embranchements possible. C'est ce que l'on appelle *l'heuristique de Warnsdorff*.

12) Écrire une fonction de signature :

```
tri(positions: list[Position], chemin: Chemin, voisins: list[list[Position]]) ->
 list[Position]
```

qui renvoie une liste triée de positions dans l'ordre croissant du nombre de cases accessibles depuis elles, à partir de la liste `positions`, du `chemin` déjà parcouru et de la liste d'adjacence `voisins`. On cherchera une fonction de complexité linéaire en la longueur de la liste `positions` pour effectuer cette opération.

13) Modifier la fonction `cherche_ronde` pour écrire une fonction de signature :

```
cherche_ronde_Warnsdorff(origine: Position, l: int, h: int) -> Chemin
```

qui renvoie une ronde de cavalier en utilisant l'heuristique de Warnsdorff, s'il en existe une, et renvoie un chemin vide sinon. La tester sur un échiquier 8×8 depuis la case 23 et constater que cette fois-ci, on obtient une réponse presque immédiate.

14) Modifier encore la fonction précédente pour écrire une fonction de signature :

```
cherche_ronde_fermee(l: int, h: int) -> Chemin
```

qui cherche une ronde *fermée* sur l'échiquier $\ell \times h$.

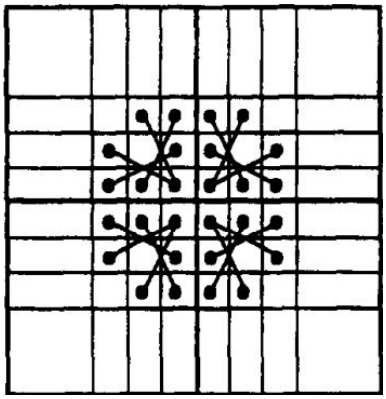
4 Toujours plus grand

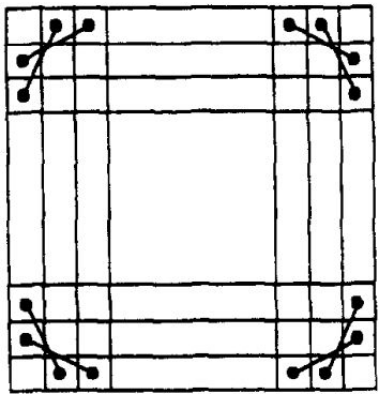
On peut prouver mathématiquement que si ℓ et h sont supérieurs ou égaux à 5, il existe toujours une ronde du cavalier (mais pas forcément depuis n'importe quelle case). Si de plus ℓ ou h est pair, il existe toujours une ronde fermée du cavalier.

Il a également été prouvé que, contrairement au problème général de recherche d'un circuit hamiltonien dans un graphe, qui est NP-complet, la recherche d'une ronde cyclique du cavalier peut se faire en $O(\ell \times h)$.

15) Prouver que l'algorithme que vous avez écrit à la question précédente s'exécute dans le pire des cas en $O(7^{\ell \times h})$.

Pour obtenir un meilleur algorithme de recherche, on utilise la méthode *diviser pour régner*. Lorsque ℓ et h sont suffisamment grands (supérieurs ou égaux à 12) et au moins l'un des deux est pair, on découpe l'échiquier en 4 morceaux rectangulaires les plus égaux possibles, de sorte que longueurs et hauteurs soient supérieurs ou égales à 6 et que dans chaque rectangle au moins un côté soit pair.





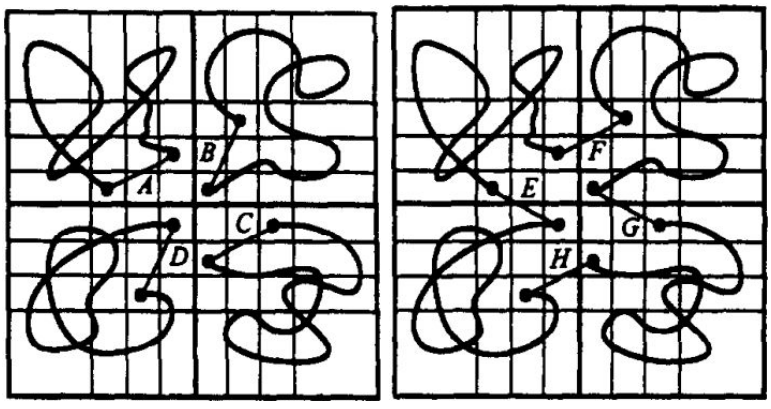
On sait alors déterminer (récursivement) une ronde fermée sur chaque rectangle, et il ne reste plus qu'à réarranger le tout pour n'en faire plus qu'une seule sur le grand rectangle de départ.
Pour que la méthode fonctionne, il est néanmoins nécessaire d'imposer des contraintes sur les arêtes présentes dans les angles, comme indiqué sur la figure ci-contre.

16) Écrire une fonction de signature :

```
cherche_ronde_fermee_contrainte(l: int, h: int, contrainte: dict[Position]) ->
 Chemin
```

qui, en plus des dimensions de l'échiquier, prend aussi comme paramètre un dictionnaire `contrainte` qui à certaines positions associe une position suivante imposée, afin de s'assurer qu'une arête donnée soit parcourue, et qui renvoie une ronde fermée respectant ces contraintes. On remarquera que la fonction auxiliaire aura besoin d'un paramètre supplémentaire `precedente` indiquant la position précédente, afin de ne pas parcourir une arête imposée indéfiniment dans les deux sens.

Pour réarranger les 4 rondes fermées obtenues récursivement, on recherche dans le bon coin de chacune des rondes la bonne arête (notées A, B, C, D sur la figure ci-dessous), on les supprime, et on fusionne les rondes en ajoutant les arêtes notées E, F, G, H au sein du grand rectangle.



Pour raccorder les morceaux de chemin des 4 rondes, il sera plus aisé d'utiliser des listes de positions plutôt que des dictionnaires.

17) Écrire une fonction de signature :

```
to_list(ronde: Chemin) -> list[Position]
```

qui prend une ronde sous la forme d'un dictionnaire de type `Chemin` et qui renvoie la liste des positions dans l'ordre du parcours.

Écrire également une fonction de signature :

```
to_dict(l_parcours: list[Position]) -> Chemin
```

qui fait l'inverse de la fonction précédente.

18) Écrire enfin une fonction de signature :

```
cherche_grande_ronde_fermee(l: int, h: int) -> Chemin
```

qui exécute l'algorithme *Diviser pour régner* décrit plus haut. On pourra lui adjoindre une mémoïsation, afin d'éviter de rechercher plusieurs fois la même ronde, ainsi qu'une méthode pour trouver une ronde sur échiquier $h \times \ell$ connaissant une ronde sur un échiquier $\ell \times h$, pour gagner du temps dans la recherche.

Pour simplifier cette dernière question, on pourra se limiter au cas où $\exists p \in \mathbf{N}, \ell = h = 6 \times 2^p$ et faire des tests sur un échiquier 12×12 puis 24×24.