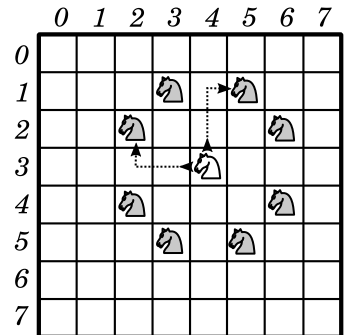
Devoir d'informatique numéro 0

À traiter pendant les vacances d'été



Les échiquiers traditionnels comportent 8 lignes et 8 colonnes. Ici, on considère des échiquiers comportant n lignes et n colonnes, c'est à dire n^2 cases repérées par deux entiers i et j dans $[0, n-1]$, sans considération de leur couleur. Sous Python, une case est définie par le tuple `(i, j)`. On envisage le déplacement d'un cavalier, pièce qui obéit à la règle suivante : en un mouvement, il se déplace de deux cases selon l'horizontale ou la verticale, puis d'une case dans la direction orthogonale. Sur la figure ci-contre par exemple, le cavalier blanc situé sur la case $(3, 4)$ peut atteindre en un coup chacune des 8 positions où un cavalier gris a été dessiné. Le nombre de positions accessibles est plus réduit si le cavalier se trouve près des bords de l'échiquier puisqu'il n'a pas le droit d'en sortir. Les 8 mouvements possibles sont codés par une liste de tuples `Dep` correspondant à des vecteurs déplacements et traitée comme une variable globale dans tout le problème.

`Dep = [(-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1)].`

On se propose ici de résoudre le problème suivant : à partir d'une position initiale (i_0, j_0) donnée, le cavalier peut-il atteindre toute autre case et, si oui, par quelle série minimale de déplacements ?

1. Coder la fonction `liste_cases_accessibles(i, j, n)` prenant en argument la taille n de l'échiquier et les entiers i et j désignant une position du cavalier, et renvoyant la liste des positions qu'il peut occuper après un déplacement. Dans la suite, ce typage comme liste et non comme une pile autorise le parcours par valeur. Voici pour vous aider deux exemples d'appel de cette fonction. Le premier correspond à la figure précédente.

```
In [233]: liste_cases_accessibles(3, 4, 8)
Out[233]: [(1, 5), (2, 6), (4, 6), (5, 5), (5, 3), (4, 2), (2, 2), (1, 3)]
In [235]: liste_cases_accessibles(0, 4, 5)
Out[235]: [(2, 3), (1, 2)]
```

2. Pour répondre au problème posé, on introduit un tableau `E` de numpy contenant des flottants [1], indexé par deux entier i et j. Par définition, `E[i,j]` est le nombre minimal de déplacements menant en (i, j) à partir (i_0, j_0). Bien entendu, on pose `E[i0, j0] = 0`. Pour calculer les autres valeurs de E, on va parcourir de proche en proche les positions du cavalier en les « cochant » et en revenant en arrière lorsqu'on sera coincé. L'algorithme repose donc sur une structure de pile.

On initialise d'abord tous les éléments de `E` à $+\infty$. Puis on introduit une pile vide sur laquelle on place le point de départ $(i_0, j0)$ et on inscrit `E[i0, j0] = 0`. Ensuite, on dépile et on examine une par une toutes les cases accessibles en un déplacement supplémentaire à partir de celle dépilée. On affecte la valeur 1 aux éléments de `E` correspondants et on les empile. Puis on dépile à nouveau et on reprend le même travail. À chaque étape, on découvre de nouvelles cases atteignables en un déplacement supplémentaire et on affecte à `E` ce nombre de déplacements, à condition qu'il soit inférieur à la valeur déjà inscrite pour ce même élément. Il se peut en effet qu'une même case soit atteinte par deux chemins distincts et il faut dans ce cas garder le plus court. Ces étapes d'empilage-dépilage se poursuivent aussi longtemps que la pile n'est pas vide.

Coder la fonction `nombre_coups_min(i0, j0, n)` qui prend en argument les coordonnées du point de départ et la taille de l'échiquier et renvoie le tableau `E` complété. Pour les cases non accessibles, `E[i, j]` restera à $+\infty$. Voici la structure du code pour vous aider.

```
def nombre_coups_min(i0, j0, n) :
    E = float('inf') * np.ones([n, n])
    Pile = ....
    E[i0, j0] = .....

    while  .... :
        i, j = ....
        n_coups =  .....
```

5

1. Des entiers suffiraient, mais on souhaite utilise le flottant infini `float('inf')`.

```
        L =  .....
        for (i1, j1) in L :                                    10
            if  ..... :
                .....
                .....
    return E
```

3. ⌨ Exécuter la fonction du machine en choisissant (i_0, j_0) et n. Par exemple, l'appel `nombre_coups_min(0, 0, 4)` renvoie `array([[0, 3, 2, 5], [3, 4, 1, 2], [2, 1, 4, 3], [5, 2, 3, 2]])`. Vérifiez que vous obtenez bien cela.

Au lieu d'utiliser une pile pour explorer l'échiquier avec retour en arrière, on peut utiliser un codage récursif dont l'idée est la suivante.

— À partir d'un point de départ, on détermine les cases accessibles en un mouvement supplémentaire ;

— pour chacune de ces cases on regarde si cette manière de l'atteindre est plus avantageuse. Si oui, on note le nouveau nombre de coups nécessaires *et on reprend l'exploration de la même manière depuis cette nouvelle case.*

C'est la phrase en italique qui conduit à la récursivité. La structure de la fonction récursive est la suivante.

```
def nombre_coups_min_rec(i0, j0, n, E) :
    L = .....
    n_coups = .....
    for (i1, j1) in L :
        if  ......                                             5
            E[i1, j1] = n_coups
            .....
    return None
```

Remarquer que cette fonction récursive ne contient pas explicitement de cas d'arrêt. Les appels cesseront naturellement lorsque tous l'échiquier aura été parcouru. Avant l'appel à la fonction, il faudra initialiser le tableau `E`. La fonction ne renvoie rien, mais `E` est modifié au fil des appels. Après l'exécution, il contiendra les valeurs voulues. Il est donc pratique d'encapsuler la fonction récursive dans une autre sous la forme suivante.

```
def applique(i0, j0, n) :
    E = float('inf') * np.ones([n, n])
    E[i0, j0] = 0
    nombre_coups_min_rec(i0, j0, n, E)
    return E                                                   5
```

4. Coder la fonction `nombre_coups_min_rec`.

5. ⌨ Utiliser la fonction `applique` pour retrouver les résultats obtenus plus haut.

On souhaite reconstituer tout l'itinéraire minimal du cavalier du point de départ (i_0, j_0) jusqu'à une case qu'il peut atteindre. Pour cela, on introduit un tableau d'antécédents `Ant` défini par le fait que `Ant[i][j]` est la case où se trouve le cavalier juste avant d'arriver en (i, j). En reprenant l'exemple de la question précédente, on a `Ant[1][3] = (2, 1)` car la case $(1, 3)$ est atteinte à partir de la case $(2, 1)$. Par convention, la case (i_0, j_0) est son propre antécédent et les cases inatteignables ont pour antécédent $(-1, -1)$. Sous Python, `Ant` est typé comme une liste de n listes de n tuples et on l'initialise commodément par compréhension.

```
Ant = [[(-1, -1) for j in range(n)] for i in range(n)]
```

6. Coder la fonction `itineraire_min(i0, j0, n)` qui renvoie le tableau `E` des nombre de coups et la liste `Ant`. Elle s'inspire directement de la fonction `nombre_coups_min` et ne nécessite que l'ajout de quelques instructions. Au choix, vous utiliserez la méthode mettant en œuvre une pile ou bien la version récursive. Si vous choisissez la récursivité, il vous faudra coder la fonction `itineraire_min_rec(i0, j0, n, E, Ant)` qui ne renvoie rien mais

modifie les tableaux `E` et `Ant`, puis l'encapsuler dans une fonction auxiliaire `applique_itineraire(i0, j0, n)` chargée des initialisations et de renvoyer les résultats.

Il est temps de passer à la représentation graphique de l'itinéraire du cavalier depuis la case initiale (i_0, j_0) jusqu'à une case finale (i_1, j_1). Pour cela, on introduit un tableau de numpy `A` de taille $n \times n$ qui va coder l'image : l'élément `A[i, j]` vaut 1 si le cavalier passe par (i, j) et 0 sinon. On l'initialise par

```
A = np.zeros([n, n], dtype = int)
```

À l'issue des calculs, l'instruction `plt.imshow(A)` permettra ensuite d'obtenir un affichage du type représenté sur la figure 1. On numérote aussi les cases parcourues avec le numéro du déplacement correspondant. Pour cela, on utilise la fonction `text` du module `matplotlib`. L'instruction suivante permet par exemple d'écrire l'entier X dans la case (i, j). Attention à bien metre j en abscisse et i en ordonnée.

```
plt.text(j, i, X, va = "center", ha = "center") # Attention : j, i
```

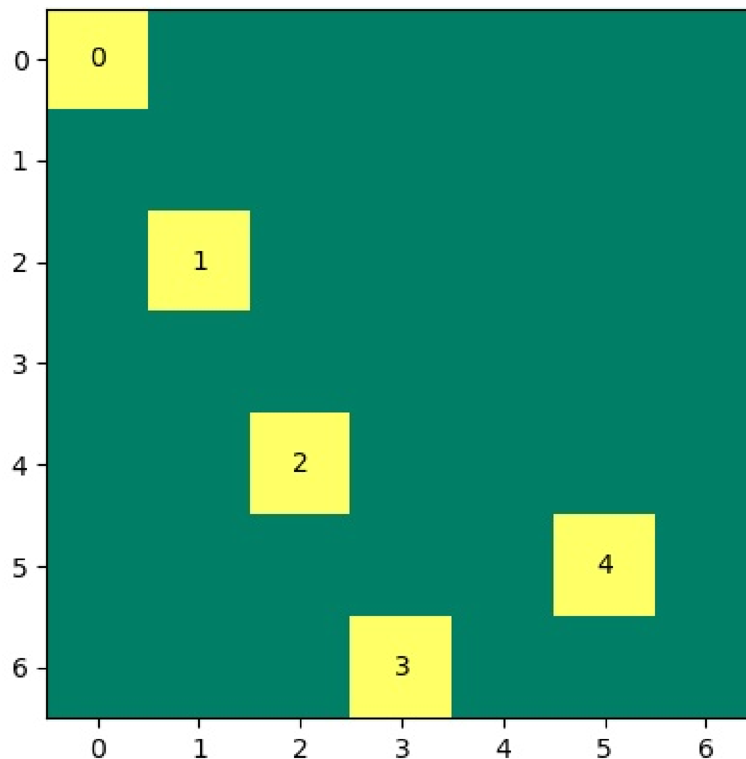


FIGURE 1 – Exemple d'itinéraire du cavalier pour $n = 7$, de la case $(0, 0)$ vers la case $(5, 5)$

7. Écrire la fonction `dessine_chemin(i0, j0, i1, j1, E, Ant)` qui dessine le chemin suivi par le cavalier. Elle utilise les tableaux `E` et `Ant` obtenus par la fonction `itineraire_min`. La méthode consiste à parcourir le chemin à l'envers à partir de $(i1, j1)$, en choisissant à chaque fois l'antécédent de la case courante.

8. 🖮 Utiliser les fonctions précédente pour représenter le plus court chemin du cavalier de la case $(0, 1)$ (position initiale du cavalier blanc) vers la cas $(7, 4)$ (position initiale du roi noir).

9. Exporter dans un fichier au format png (ou jpg) l'image montrant le résultat. Nommer ce fichier sous la forme `Prenom_Nom.png` (ou .jpg) puis le déposer dans le dossier partagé indiqué ci-dessous.

https://nuage04.apps.education.fr/index.php/s/4mwtbotCxHDkBEo