TP 11. Séquences et dictionnaires.

Pour les PTSI 1 Avant toute chose, créer un répertoire TP11 dans votre répertoire "Informatique", "TPsup" dans votre espace personnel. Sur le site **ENT/Moodle/cours"infoPTSI"**, télécharger `drone.py` que vous copiez dans votre répertoire TP11.

Pour les PTSI 2 Avant toute chose, créer un répertoire TP11 dans votre répertoire "Informatique", "TPsup" dans votre espace personnel. Sur le site `https://ptsilamartin.github.io/info/TP.html`, télécharger `drone.py` que vous copiez dans votre répertoire TP11.

CONSIGNES

- Lors de l'écriture d'une fonction, on utilisera un seul `return`.
- Le signe ♯ est utilisé devant un commentaire. Pour chaque exercice, vous devrez en entête inscrire le titre de l'exercice.
- Les tests effectués seront conservés sous forme de commentaires dans le script de votre programme. Le signe de commentaire ♯ en début de ligne peut être mis en sélectionnant les lignes à commenter et en choisissant « comment out region » dans l'onglet format (pour décommenter, vous pouvez utiliser « uncomment region »).
- On rappelle aussi qu'il est préférable de faire les tests dans le script (au moyen de `print`) plutôt que dans le shell. Il suffit ensuite de mettre en commentaire les `print`.

1 Gérer les stocks de composants pour réaliser des drones

L'entreprise `SuperDrone` réalise le montage et la vente de drones.

Les composants utiles pour réaliser un drone sont :

— le châssis ;
— les 4 moteurs ;
— les 4 hélices ;
— la batterie ;
— le contrôleur de vol ;
— ESC 4 en 1 pour les 4 moteurs (Electronic Speed Controler) ;
— la plaque de distribution de puissance (PDB Power Distribution Board).

Des composants en option sont aussi disponibles (caméra, radiocommande, chargeur, buzzer, leds...) et ne seront pas traités ici.

Dans le fichier `drone.py` donné, vous trouverez 4 dictionnaires dont les clés sont les noms des composants et les valeurs représentent le nombre de composants nécessaire :

```
drone={'moteur':4,'chassis':1,'controleurVol':1,'ESC4en1':1,'batterie':1,'helice':4,
'plaqueDeDistribution':1}
stock={'chassis':0,'moteur':25,'helice':36,'controleurVol':12,'ESC4en1':8,'batterie':20,
'plaqueDeDistribution':7}
limiteMin={'chassis':2,'moteur':8,'helice':8,'controleurVol':2,'ESC4en1':2,'batterie':2,
```

```
'plaqueDeDistribution':2}
limiteMax={'chassis':15,'moteur':60,'helice':60,'controleurVol':15,'ESC4en1':15,'batterie':30,
'plaqueDeDistribution':15}
```

— `drone`, correspond aux composants nécessaires à la réalisation d'un drone. Les clés étant les composants et les valeurs le nombre de composant pour un drone ;

— `stock`, correspond au stock à l'instant considéré ;

— `limitMin`, correspond aux valeurs limites basses du stock pour déclencher une commande ;

— `limitMax`, correspond aux valeurs limites hautes pour reconstituer le stock et pour définir le nombre de composants à commander.

OBJECTIF

Écriture d'un programme qui permette de générer les commandes de composants utiles pour assurer la réalisation des drones attendus par les clients sans avoir de rupture de stock.
On utilisera des objets de type `dict`.

Réaliser un drone

Question 1. Écrire la fonction `realiser1Drone(drone:dict, stock:dict)` qui prend pour argument les dictionnaires `drone` et `stock` et qui renvoie un booléen, `True` si le stock est suffisant pour réaliser un drone, `False` sinon.

Gérer le stock de composants

Question 2. Écrire la fonction `destocker(D:dict, S:dict)` qui prend pour argument les dictionnaires `D` des composants du drone et `S` du stock et qui retire du stock le nombre de composants utiles pour réaliser un drone. Cette fonction ne renvoie rien. Le dictionnaire `S` est modifié par effet de bord.

Lorsque le stock devient insuffisant, une commande est passée et le stock est ré-évalué.

Question 3. Écrire la fonction `stocker(C:dict, S:dict)` qui prend pour argument les dictionnaires `C` correspondant à la commande et `S` du stock et qui ajoute au stock le nombre de composants commandés. Cette fonction ne renvoie rien. Le dictionnaire `S` est modifié par effet de bord, le dictionnaire `C` n'est pas modifié.

Passer une commande

Une commande est passée quand le stock d'un composant est à la limite basse (valeur incluse). Les composants qui n'ont pas atteint la limite basse ne sont pas commandés.

Question 4. Écrire la fonction `commanderComposant(S:dict,limiteMin:dict,limiteMax:dict)` qui prend pour argument les dictionnaires `S` du stock, `limiteMin` et `limiteMax` et qui renvoie un dictionnaire `commande` permettant de reconstituer le stock.

Gestion automatique

Dans la semaine, l'entreprise `SuperDrone` reçoit les commandes de drones de 4 clients sous la forme d'une liste, `listeCommande=[3,1,5,2]`.

Question 5. Écrire la fonction `satisfaireClient(listeCommande:list,drone:dict,stock:dict, limiteMin:dict,limiteMax:dict)` qui prend pour argument une liste de commande de drones, les dictionnaires `drone`, `stock`, `limiteMin` et `limiteMax` et qui affiche l'état du stock après chaque réalisation d'un drone ainsi que les commandes successives.

2 Snakes and ladders : le jeu

Extrait du travail de T. Kovaltchouk - UPSTI

2.a Présentation du jeu

Le jeu *serpents et échelles* est un jeu de société où on espère monter les échelles en évitant de trébucher sur les serpents. Il provient d'Inde et est utilisé pour illustrer l'influence des vices et des vertus sur une vie.

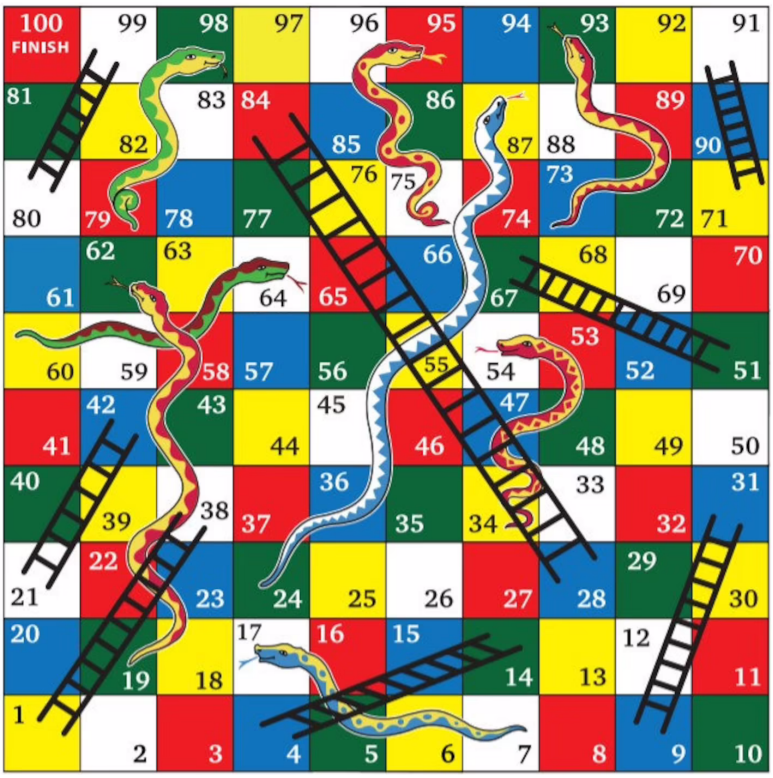

FIGURE 1 – Exemple d'un plateau de serpents et échelles

Le plateau

- Le plateau comporte 100 cases numérotées de 1 à 100 en boustrophédon [1] : le 1 est en bas à gauche et le 100 est en haut à gauche ;
- des serpents et échelles sont présents sur le plateau : les serpents font descendre un joueur de sa tête à sa queue, les échelles font monter un joueur du bas de l'échelle vers le haut.

Déroulement

- Chaque joueur a un pion sur le plateau. Plusieurs pions peuvent être sur une même case. Les joueurs lancent un dé à tour de rôle et ils avancent du nombre de cases marqués sur le dé. S'ils atterrissent sur un bas d'échelle ou une tête de serpent, ils vont directement à l'autre bout ;
- les joueurs commencent sur une case 0 hors du plateau : la première case où mettre leur pion correspond donc au premier lancer de dé ;
- le premier joueur à arriver sur la case 100 a gagné ;
- il existe 3 variantes quand la somme de la case actuelle et du dé dépasse 100 :
  - le rebond : on recule d'autant de cases qu'on dépasse ;

1. à la manière du bœuf traçant des sillons, avec alternance gauche-droite et droite-gauche

— l'immobilisme : on n'avance pas du tout si on dépasse :

— la fin rapide : on va à la case 100 quoi qu'il arrive.

On utilisera les notations suivantes pour les complexités : N_{cases}, le nombre de cases du plateau (100), et N_{SeE} la somme du nombre de serpents et du nombre d'échelle (16 dans notre exemple).

3 Simulation du jeu

Question 6. Écrire une fonction `lancerDe() -> int` qui renvoie un nombre entier compris entre 1 et 6 en utilisant une fonction du module `random`. Vous pourrez vous aider des documentations en annexe.

Les serpents et les échelles sont représentés par un dictionnaire `dSeE` tel que, pour une case de départ numérotée `i`, `dSeE[i]` donne le numéro de la case d'arrivée.
Avec l'exemple de la figure 1, on a :

```
dSeE = {  1: 38,  4: 14,  9: 31, 17:  7, 21: 42, 28: 84, 51: 67, 54: 34,
         62: 19, 64: 60, 71: 91, 80: 99, 87: 24, 93: 73, 95: 75, 98: 79}
```

Question 7. Écrire la fonction `caseFuture(case: int) -> int` qui prend en argument le numéro de la case et qui renvoie le numéro de la case où va se trouver le joueur en atterrissant sur la case numérotée `case`. Par exemple, `caseFuture(5)` renvoie 5 (c'est un numéro de case stable), `caseFuture(1)` renvoie 38 (c'est un numéro de case avec échelle) et `caseFuture(17)` renvoie 7 (c'est un numéro de case avec une tête de serpent).

Question 8. Quelle est la complexité de cette fonction ?

Question 9. Écrire une fonction `avanceCase(case: int, de: int, choix: str) -> int` qui renvoie la case d'arrivée lorsqu'on part de la case `case` et qu'on a comme résultat au lancer du dé la valeur `de`. La variable `choix` est une chaine de caractère correspondant à la stratégie de fin différente : `"r"` pour le rebond, `"i"` pour l'immobilisme et `"q"` pour une fin rapide.

Question 10. Écrire une fonction `partie(choix: str) -> [int]` qui lance une partie à un joueur et renvoie la liste successive des cases visitées sur le plateau. Elle commencera donc forcément par `0` et finira forcément par `100`. Le choix du mode de fin est en argument, de façon similaire à la question précédente.

3.a Plus court chemin

On souhaite, dans cette partie, utiliser un algorithme glouton pour trouver la partie la plus courte.

Question 11. Écrire une fonction `casesAccessibles(case: int) -> [int]` qui renvoie la liste des 6 cases accessibles pour la case donnée en entrée. Vous utiliserez la fonction `avanceCase` de la question 9. La liste renvoyée `cases` doit avoir le codage suivant : `case[i]` doit correspondre à la case d'arrivée avec le résultat de dé `i+1` (donc la liste retournée doit toujours avoir une longueur de 6). On prendra l'option de fin rapide.

Question 12. Écrire une fonction `meilleurChoix(case: int) -> int` qui renvoie la meilleure case accessible depuis `case`. Il est interdit d'utiliser la fonction `max` dans cette question.

L'algorithme glouton consistera à choisir la valeur du dé permettant de maximiser son déplacement à chaque coup.

Question 13. Écrire une fonction `partieGloutonne() -> [int]` qui renvoie la liste des cases par lesquelles passe le pion dans l'algorithme glouton.

Cette dernière fonction nous renvoie `[0, 38, 44, 50, 67, 91, 97, 100]`.

Question 14. Construire un exemple de plateau, contenant par exemple 2 échelles et pas de serpent, pour lequel notre algorithme ne trouve pas le chemin le plus court en nombre de coups : vous préciserez le résultat de l'algorithme glouton et un exemple d'une partie strictement plus rapide.

Annexe

Utilisation du module `random`

On vous donne les docstrings correspondant à deux fonctions du module `random` :

```
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

Complexité des opérations sur les listes et dictionnaires

Principales opérations sur les listes `n`, longueur de la liste `L`, `k`, un indice valide en négatif (`1` à `n`).

Opération	Moyen
Longueur (`len(L)`)	$O(1)$
Accès en lecture d'un élément	$O(1)$
Accès en écriture d'un élément	$O(1)$
Copie (`L.copy()` ou `L[:]`)	$O(n)$
Ajout (`L.append(elt)` ou `L+=[elt]`)	$O(1)$
Extension (`L1.extend(L2)` ou `L1+=L2`)	$O(n_2)$
Concaténation (`L1 + L2`)	$O(n_1 + n_2)$
Test de présence (`elt in L`)	$O(n)$
Désempiler dernier (`L.pop()`)	$O(1)$
Désempiler autre (`L.pop(-k)`)	$O(k)$
Maximum ou minimum (`max(L)` et `min(L)`)	$O(n)$
Tri (`L.sort()` ou `sorted(L)`)	$O(n\log(n))$

Principales opérations sur les dictionnaires `n`, longueur du dictionnaire `d`, `k`, une clé du dictionnaire.

Opération	Moyen
Longueur (`len(d)`)	$O(1)$
Accès en lecture d'un élément (`x = d[k]`)	$O(1)$
Accès en écriture d'un élément (`d[k] = x`)	$O(1)$
Copie (`d.copy()`)	$O(n)$
Ajout (`d[k] = x` la première fois)	$O(1)$
Test de présence (`k in d`)	$O(1)$
Retrait d'un élément (`del d[k]` ou `d.pop(k)`)	$O(1)$