Durée : 2 heures. L'utilisation de la calculatrice n'est pas autorisée.

Exercice 1

Soit une liste L d'entiers naturels et un entier naturel a. On se propose de déterminer le nombre de fois où apparaît a dans L. Exemple : si $L = [4, 2, 5, 3, 2, 6, 2, 3, 4]$ et si $a = 2$, le nombre en question est 3.

1. Écrire une fonction itérative non récursive qui prend en entrée une liste L d'entiers naturels et un entier naturel a, et qui retourne le nombre de fois où apparaît a dans L.

2. Écrire une fonction récursive qui prend en entrée une liste L d'entiers naturels et un entier naturel a, et qui retourne le nombre de fois où apparaît a dans L.

Exercice 2

On définit une suite par : $\begin{cases} u_0 = 2, \quad u_1 = 1 \\ u_{n+2} = 6\, u_{n+1} + u_n \end{cases}$

On se propose d'écrire une fonction récursive d'entrée un entier naturel n et qui retourne le terme u_n de cette suite.

1. Compléter le script suivant :

```
def suite(n):
    if n==0:
        ...
    if n==1:
        ...
    else:
        ...    6*suite(...) + suite(...)
```

2. Déterminer la complexité de l'algorithme précédent. On comptera les tests et les opérations algébriques.
Remarque : il se sera peut-être utile de se souvenir de la façon dont on étudie une suite arithmético-géométrique.

Exercice 3

On considère la fonction f définie par $\forall x \in \mathbb{R},\ f(x) = x^3 + x - 5$.

1. Montrer que l'équation $f(x) = 0$ admet une unique solution notée c dans l'intervalle $[1, 2]$.

2. On se propose de déterminer une valeur approchée de c.
On définit la fonction suivante qui prend en entrée le réel x et qui retourne $f(x)$.

```
def fonction(x):
    return x**3 + x - 5
```

a. Méthode 1 : dichotomie.

i. Soit epsilon un réel strictement positif. Écrire un algorithme itératif non récursif basé sur la méthode par dichotomie à partir de l'intervalle $[1, 2]$, qui prend en entrée epsilon et qui retourne une valeur approchée de c à epsilon près.

ii. Justifier la terminaison de cet algorithme.

iii. Donner un encadrement du nombre de tours de boucle qu'effectuera cet algorithme en fonction de epsilon.

iv. Proposer un algorithme récursif basé sur la méthode par dichotomie qui prend en entrée deux réels a et b et un réel epsilon strictement positif et qui retourne une valeur approchée de c à epsilon près.

b. Méthode 2 : Newton.

i. Rappeler le principe de la méthode de Newton.

ii. Soit epsilon un réel strictement positif. Écrire un algorithme basé sur la méthode de Newton qui prend en entrée epsilon et qui retourne une valeur approchée de c à epsilon près. On pourra partir de 1 ou 2.

Exercice 4

On s'intéresse dans cet exercice au problème de Cauchy :

$$\begin{cases} \forall t \in I, \quad a\,y''(t) + b\,y'(t) + c\,y(t) + d = 0 \\ y(0) = \alpha \quad \text{et} \quad y'(0) = \beta \end{cases}$$

où $(a, b, c, d) \in \mathbb{R}^4$ et $a \neq 0$.
La méthode d'Euler explicite à 2 pas permet de définir une suite récurrente (y_n) d'ordre 2 qui permet de représenter une approximation de la solution exacte de l'équation différentielle ci-dessus sur un intervalle $[0, M]$, avec $M > 0$.
On donne un pas de temps $h > 0$. On pose $t_n = n\,h$.
Utiliser deux fois la méthode d'Euler explicite sur y' puis sur y'', avec le pas de temps h, pour déterminer y_{n+2} en fonction de y_{n+1}, y_n et h.
Utiliser la méthode d'Euler explicite sur y', avec le pas de temps h, pour déterminer y_1 en fonction de α, β et h.
Proposer une fonction qui a pour variable d'entrée le pas h et qui retourne les deux listes $T = [t_0, t_1, t_2, \ldots]$ et $Y = [y_0, y_1, y_2, \ldots]$ sachant que $t \in [0, M]$.

Exercice 5

But de l'exercice

Le jeu d'échec se joue sur un échiquier, c'est à dire sur un plateau de 8×8 cases. Ces cases sont référencées de $a1$ à $h8$ (voir figures).
Une pièce, appelée le cavalier, se déplace suivant un "L" imaginaire d'une longueur de deux cases et d'une largeur d'une case.
Exemple (figure 1) : un cavalier situé sur la case $d4$ atteint, en un seul déplacement, une des huits cases $b5$, $c6$, $e6$,$f5$, $f3$, $e2$, $c2$ ou $b3$.
Dans toute la suite de l'exercice, on appellera **case permise** toute case que le cavalier peut atteindre en un déplacement à partir de sa position.

Le but de cet exercice est d'écrire un programme faisant parcourir l'ensemble de l'échiquier à un cavalier **en ne passant sur chaque case qu'une et une seule fois**

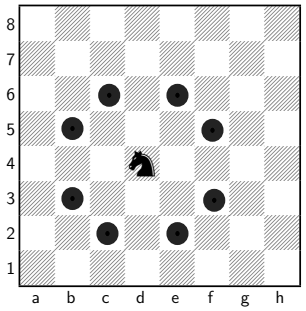

FIGURE 1
déplacements permis

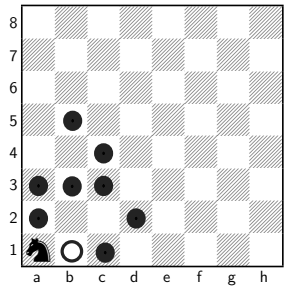

FIGURE 2
un exemple

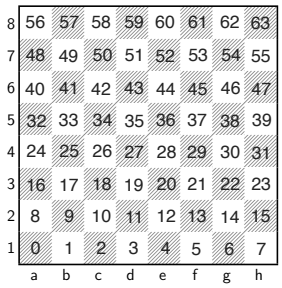

FIGURE 3
numérotation

Motivation et méthode retenue

Une première idée est de faire parcourir toutes les cases possibles à un cavalier en listant à chaque déplacement les cases parcourues. Lorsque celui-ci ne peut plus avancer, on consulte le nombre de cases parcourues.

- Si ce nombre est égal à $64 = 8 \times 8$, alors le problème est résolu.
- Sinon, il faut revenir en arrière et tester d'autres chemins.

1. **Exemple** : on considère le parcours suivant d'un cavalier démarrant en $a1$ (figure 2)

$$a1, b3, c1, a2, c3, b5, a3, c4, d2$$

Avec ce début de parcours, au déplacement suivant :

a. Le cavalier va en $b1$. Peut-il accomplir sa mission ?

b. Le cavalier ne va pas en $b1$. Peut-il accomplir sa mission ?

Il convient donc dans la résolution du problème proposé d'éviter de se retrouver dans la situation repérée en cette première question.

Dans tout ce qui suit, nous nommerons **coordonnées** d'une case la liste d'entiers $[i, j]$ où i représente le numéro de ligne et j le numéro de colonne (tous deux compris entre 0 et 7). Par exemple, la case $b3$ a pour coordonnées $[2, 1]$.
D'autre part, les cases sont numérotées de 0 à 63 en partant du coin gauche comme indiqué en figure 3.
Nous appellerons *indice* d'une case, l'entier $n \in [\![0, 63]\!]$ ainsi déterminé. Ainsi la case $b3$ a pour indice 17.

2. Ecrire une fonction `Indice` qui prend en argument la liste des coordonnées d'une case et qui renvoie son indice. Ainsi `Indice` appliquée à la liste `L=[2,1]` doit retourner 17.

3. Ecrire une fonction `Coord` qui à l'indice n d'une case associe la liste $[i, j]$ de ses coordonnées. Ainsi `Coord` appliquée à 17 doit retourner $[2, 1]$.

4. On considère la fonction Python `CasA` suivante :

```
def CasA(n):
    Deplacements=[[1,-2],[2,-1],[2,1],[1,2],[-1,2],[-2,1],[-2,-1],[-1,-2]]
    L=[]
    i,j=Coord(n)
    for d in Deplacements:
        u=i+d[0]
        v=j+d[1]
        if u>=0 and u<8 and v>=0 and v<8:
            L.append(Indice([u,v]))
    return(L)
```

a. Que renvoient `CasA(0)` et `CasA(39)`.

b. Expliquer en une phrase ce que fait cette fonction.

5. Rappel. Dans une fonction les variables locales sont des variables qui apparaissent à l'appel d'une fonction et qui disparaissent à la fin de l'exécution de la fonction (par exemple un indice de boucle).
Si on souhaite utiliser et éventuellement modifier des variables par l'intermédiaire d'une fonction, il faut les définir comme globales, par l'instruction `global`. On peut alors les récupérer modifiées ou pas après l'exécution de la fonction.
Soit la fonction `Init` ci dessous qui ne prend aucun argument et qui utilise deux variables globales :

```
def Init():
    global ListeCoups,ListeCA
    ListeCoups=[]
    ListeCA=[CasA(n) for n in range(64)]
```

Après exécution de cette fonction `Init()`, la commande `ListeCA[0]` renvoie-t-elle $[5]$, $[10, 17]$, $[10, 17, 0]$, $[17, 0, 10]$, $[\,]$ ou une autre valeur ?

6. Au cours de la recherche, lorsqu'on déplace le cavalier vers la case d'indice n, cet indice n doit être retiré de la liste des *cases permises* à partir de la position n.
Exemple : après exécution de la fonction `Init()`, la liste des cases permises depuis $b1$ est $[a3, c3, d2]$ et `ListeCA[1]=[16,18,11]`.
La liste des cases permises depuis $a3$ est $[b5, c4, c2, b1]$ et `ListeCA[16]=[33,26,10,1]`.
Puis, on choisit de commencer le parcours en posant le cavalier en $b1$. Cette case doit donc être retirée de la liste des cases permises de $a3$, $c3$ et $d2$. En particulier pour $a3$, la liste `ListeCA[16]` devient `[33,26,10]`.
Cette méthode nous permet de détecter les blocages : le cavalier arrive sur la case d'indice n, n est alors retiré de toutes les listes `ListeCA[k]` pour toute case k permise pour n. Si dès lors l'une de ces listes devient vide, nous dirons que nous somme alors dans une *situation critique*, cela signifiera que la case d'indice k ne peut plus être atteinte que depuis la case d'indice n. Par conséquent,

- si le cavalier se déplace sur une autre case que celle d'indice k, alors cette dernière ne pourra plus jamais être atteinte ;
- si le cavalier se déplace sur la case d'indice k, il est bloqué pour le coup suivant. Soit la mission est accomplie, soit le cavalier n'a pas parcouru toutes les cases.

Le programme va réaliser la recherche en maintenant à jour la variable globale `ListeCoups` afin qu'elle contienne en permanence la liste des positions successives occupées par le cavalier au cours de ses tentatives de déplacement. Nous avons alors besoin d'écrire trois fonctions.

a. Ecrire une fonction `OccupePosition` qui

- prend comme argument un entier n (indice d'une case), l'ajoute à la fin de la variable globale `ListeCoups`,
- puis enlève n de toutes les listes `ListeCA[k]` pour toutes les cases k permises depuis la case d'indice n,
- renvoie enfin la valeur `True` si nous sommes dans une situation critique et `False` sinon.

On pourra utiliser la méthode `remove` *qui permet de retirer d'une liste le premier élément égal à l'argument fourni. Si l'argument ne fait pas partie de la liste, une erreur sera retournée*

```
L=[1,2,3,4,2,5]
L.remove(2) # cette commande modifie L en [1,3,4,2,5]
L.remove(6) # cette commande provoque une erreur
```

b. Ecrire une fonction `LiberePosition` qui ne prend pas d'argument et qui

- récupère le dernier élément n de la variable globale `ListeCoups` (i.e. n est l'indice de la dernière case jouée à l'aide de la fonction `OccupePosition(n)`),
- puis l'enlève de `ListeCoups`,
- et enfin, qui ajoute n à toutes les listes `ListeCA[k]` pour toutes les cases d'indice k permises depuis la case d'indice n.

On pourra utiliser la méthode `pop` *qui renvoie le dernier élément d'une liste et le supprime de cette même liste.*

```
L=[1,2,3,4,2,5,2]
n=L.pop()  # après cette commande, on a  n=2 et L=[1,2,3,4,2,5]
```

c. Ecrire une fonction `TestePosition` d'argument un entier n (indice d'une case) qui :

- occupe la position d'indice n,
- vérifie si la situation est critique.
  Si c'est le cas, la fonction vérifiera si les 63 cases sont occupées et, dans ce cas renverra `True` pour indiquer que la recherche est terminée. Si les 63 cases ne sont pas occupées, la fonction libérera la case d'indice n et renverra `False`.
  Dans le cas contraire, la fonction vérifiera avec `TestePosition` toutes les cases d'indice k jouables après celle d'indice n (on prendra garde à affecter une variable locale avec la liste `ListeCA[n]` puisque celle-ci risque d'être modifiée lors des appels suivants). La fonction retournera `True` dès que l'un des appels à `TestePosition` retourne `True` ou libérera la case d'indice n et retournera `False` sinon.