```python
# DS4-corrige.py

from collections import deque
import matplotlib.pyplot as plt
import random as r
import math as m
import networkx as nx
import numpy as np

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

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}

ncases = 100
nlignes = 10

# PB
# p2 : le rebond :s
# p2 : on recule d'autant de caseS
# p2 : définir l'expression "on dépasse"
# p2 : dans uN premier temps
# p2 : Avec l’exemple de la Figure fig :1

# p8 : manque les infos pour matplotlib



#Q1
def lancerDe() -> int:
    return r.randint(1, 6)

#Q2
def caseFuture(case:int) -> int:
    if case in LSeE[0]:
        return LSeE[1][LSeE[0].index(case)]
    else:
        return case

print("10 ->", caseFuture(10))
print("17 ->", caseFuture(17))
```

```
041| print("9 ->", caseFuture(9))
042|
043| #Q3 complexité de caseFuture ? complexité de in et de
index ? O(n)
044|
045| #Q4
046| def caseFuture2(case:int) -> int:
047|     return dSeE.get(case, case)
048|
049|
050| print("10 ->", caseFuture2(10))
051| print("17 ->", caseFuture2(17))
052| print("9 ->", caseFuture2(9))
053|
054| #Q5 complexité de caseFuture2 ? O(1)
055|
056| #Q6
057| def avanceCase(case:int, de:int, choix:str) -> int:
058|     nextcase = case + de
059|     if nextcase > ncases:
060|         if choix == "r":
061|             nextcase = ncases - (nextcase - ncases)
062|         elif choix == "i":
063|             nextcase = case
064|         else:
065|             nextcase = ncases
066|     return caseFuture2(nextcase)
067|
068| #Q7
069| def partie(choix: str) -> [int]:
070|     liste = [0]
071|     fin = False
072|     while not fin:
073|         liste.append(avanceCase(liste[-1],
lancerDe(), choix))
074|         if liste[-1] == ncases:
075|             fin = True
076|     return liste
077|
078| print(partie("r"))
079| print(partie("i"))
080| print(partie("q"))
081|
082| # 3 Dessiner le plateau
083|
084| #Q8
```

```
085| def position(case: int) -> (int, int):
086|     q = (case - 1) // nlignes
087|     r = (case - 1) % nlignes
088|     if q % 2 == 0:
089|         return (r,q)
090|     else:
091|         return (9 - r,q)
092|
093|
094| for case in range(1, 101): # pour les cases 1 à 100
095|     i, j = position(case)
096|     plt.text(i, j, str(case),
horizontalalignment='center', verticalalignment='center')
097|
098| for caseD, caseA in dSeE.items():
099|     iD, jD = position(caseD)
100|     iA, jA = position(caseA)
101|     if caseA > caseD:
102|         couleur = 'b'
103|     else:
104|         couleur = 'r'
105|     plt.plot(iA, jA, '^', color=couleur)
106|     plt.plot(iD, jD, 'o', color=couleur)
107|     plt.plot([iA, iD], [jA, jD], color=couleur)
108| plt.axis("equal")
109| plt.show()
110|
111| #Q9 caseD et caseA sont respectivement les cases de
départ (clé) et d'arrivée (valeur associée) pour les
serpents et les échelles
112|
113| #Q10 distinction par les couleurs des échelles (en
bleu) et des serpents (en rouge)
114|
115| # 4 Plus court chemin
116|
117| #Q11
118| def casesAccessibles(case: int) -> [int]:
119|     return [avanceCase(case, de, "q") for de in
range(1,7)]
120|
121| print(casesAccessibles(0))
122| print(casesAccessibles(63))
123|
124| #Q12
125| def meilleurChoix(case: int) -> int:
```

```
126|     L = casesAccessibles(case)
127|     maxi = 0
128|     for pos in L:
129|         if pos > maxi:
130|             maxi = pos
131|     return maxi
132|
133| def meilleurChoix2(case:int) -> int:
134|     L = casesAccessibles(case)
135|     L.sort()
136|     return L[-1]
137|
138| print(meilleurChoix(0))
139| print(meilleurChoix(63))
140| print(meilleurChoix2(0))
141| print(meilleurChoix2(63))
142|
143| #Q13
144| def partieGloutonne() -> [int]:
145|     L = [0]
146|     while L[-1] != ncases:
147|         L.append(meilleurChoix(L[-1]))
148|     return L
149|
150| print(partieGloutonne())
151|
152| #Q14 Par exemple une première échelle entre 1 et 11,
puis une autre entre 7 et 100
153| #L'algorithme glouton choisira le gain immédiat en
prenant la première échelle
154| # pour atteindre la case 11 en un coup et au maximum
la case 17 en 2 coups,
155| # alors qu'en deux coups (6 + 1 par exemple) il est
possible de finir la partie.
156|
157| #5 étude du graphe correspondant
158|
159| #Q15
160| def eliminationDoublon(L:[int]) -> [int]:
161|     return list(set(L))
162|
163| L=[0,15,23,15,69,7,69]
164| print(eliminationDoublon(L))
165|
166| #Q16 Je dirais plutôt O(n) dans mon cas
167|
```

```
168| #Q17
169| def eliminationDoublon(L:[int]) -> [int]:
170|     return [cle for cle in {L[i]: None for i in
range(0,len(L))}]
171|
172| L=[0,15,23,15,69,7,69]
173| print(eliminationDoublon(L))
174|
175| # Remarque : l'utilisation de set ne garantit pas la
conservation de l'ordre des éléments dans la liste.
176| # il semblerait qu'en passant par le dictionnaire on
conserve bien l'ordre... pas sûr que ce soit fiable. Qu'en
penses-tu ?
177|
178| #Q18
179| G = {}
180| for i in range(0, ncases):
181|     if i not in dSeE:
182|         G[i] =
eliminationDoublon(casesAccessibles(i))
183| G[ncases] = []
184| print(G)
185|
186| #Q19 Le plateau étudié ne nécessite pas d'éliminer
les doublons. Toutefois on
187| # pourrait imaginer un plateau avec une échelle entre
1 et 3 et une autre entre 2 et 3
188| # Ainsi en démarrant à la case 0 et ayant lancé un 1
ou un 2 le pion se retrouvera à
189| # la case 3. 3 serait vu comme un successeur multiple
de 0. Le graphe serait multi-arêtes.
190| # Cela complexifirait (inutilement pour notre étude)
le parcours du graphe (HP).
191|
192| #Q20 #G = Ncases - NseE + 1
193|
194| def chemin(G, depart, arrivee):
195|     predecesseur = {}
196|     file = deque([depart])
197|     predecesseur[depart] = None
198|     while file:
199|         sommet = file.popleft()
200|         for successeurDeSommet in G[sommet]:
201|             if successeurDeSommet not in
predecesseur.keys():
202|                 file.append(successeurDeSommet)
```

```
203|                 predecesseur[successeurDeSommet] =
sommet
204|     return arrivee in predecesseur
205|
206| print(chemin(G, 0, 100))
207|
208| #Q21 Parcours en largeur. Utilisation d'une file
explicite ?
209|
210| #Q22 Compliqué !!! car il faut déjà comprendre ce
qu'il y a dans predecesseur
211| def chemin2(G, depart, arrivee):
212|     predecesseur = {}
213|     file = deque([depart])
214|     predecesseur[depart] = None
215|     while file:
216|         sommet = file.popleft()
217|         for successeurDeSommet in G[sommet]:
218|             if successeurDeSommet not in
predecesseur.keys():
219|                 file.append(successeurDeSommet)
220|                 predecesseur[successeurDeSommet] =
sommet
221|
222|     if arrivee in predecesseur:
223|         L=[]
224|         pos = arrivee
225|         while pos != None:
226|             L.append(pos)
227|             pos = predecesseur[pos]
228|         L.reverse()
229|         return L
230|     else:
231|         return []
232|
233| print(chemin2(G, 0, 100))
234|
235| #Q23 Je ne comprends pas la question. La fonction
précédente avec le même
236| # début et la même fin renvoie toujours le même
résultat.
237| # Impossible de faire intervenir le hasard ou une
recherche exaustive pour
238| # trouver le plus court chemin à moins de
reprogrammer une autre fonction
239| # chemin. Ou tester des routage en impossant un point
```

```
de passage par exemple
240| # aller de 0 à 100, c'est aller de 0 à n puis de n à
100.
241| # Mais comment choisir le bon nombre d'étapes et où
les positionner ? Cela
242| # peut vite devenir gourmand comme recherche.
243|
244| def partieOptimale() -> [int]:
245|     # Je donne ma langue au chat
246|     return None
247|
248| # 6 Statistiques
249|
250| #Q24 partie contient la liste des cases utilisées
pour aller de 0 à 100.
251| # Le pion est en position 0 au départ. Le premier
coup joué permet de
252| # quitter cette position pour atteindre une nouvelle
position. Le nb de coups
253| # joués est inférieur de 1 au nombre de cases
utilisées dans la partie
254| # (pb habituel des arbres et des intervalles).
255|
256| #Q25 Attention ce n'est pas très clair ce que
contient L. Il faut le deviner/
257| # l'interprêter.
258| def moyenne(L:[int]) -> float:
259|     somme = 0
260|     for l in L:
261|         somme += l
262|     return somme/len(L)
263|
264| #Q26
265| def variance(L:[int]) -> float:
266|     var = 0
267|     for l in L:
268|         var += l**2
269|     return var / len(L) - (moyenne(L))**2
270|
271| #Q27
272| def ecartType(L:[int]) -> float:
273|     return m.sqrt(variance(L))
274|
275| #Q28 Réponse 2
276| def mediane(L):
277|     n = len(L)
```

```
278|     for i in range(n):
279|         cle = L[i]
280|         j = i
281|         while 0 < j and cle < L[j-1]:
282|             L[j] = L[j-1]
283|             j = j-1
284|         L[j] = cle
285|     return L[n//2]
286|
287| long=[4,8,3,1,9,5]
288| print(long)
289| print(mediane(long))
290|
291| #Q29 Tri par insertion. Meilleur O(n), pire O(n²)
```