EXERCICE 1 (6 points)

Cet exercice porte sur l'exécution d'un programme Python et sur la décidabilité.

Dans cet exercice, on dira qu'un appel `f(x)`, où `f` est une fonction Python prenant un argument et `x` est une valeur, **termine** lorsque l'évaluation de `f(x)` renvoie toujours une valeur au bout d'un nombre fini d'étapes. A l'opposé, un tel appel ne termine pas s'il est possible qu'il effectue des instructions à l'infini.

Partie A : boucle `while`

Commençons par un premier exemple, avec une fonction prenant un entier en argument et utilisant une boucle `while`.

```
1 def f1(n):
2     i = n
3     while i != 10:
4         i = i + 1
5     return i
```

1. Sur votre copie, donner les valeurs successives de la variable `i` lors de l'exécution de `f1(7)`, et indiquer si `f1(7)` termine.

2. Indiquer si l'appel `f1(-2)` se termine. Si oui, indiquer la valeur renvoyée.

3. Sur votre copie, donner les 5 premières valeurs prises par la variable `i` lors de l'exécution de `f1(12)`, et indiquer si l'appel `f1(12)` va terminer ou non.

4. Préciser pour quels entiers `n` l'appel `f1(n)` se termine.

Partie B : fonction récursive

Prenons maintenant un deuxième exemple, avec une fonction récursive (prenant elle aussi un entier en argument).

```
1 def f2(n):
2     if n == 0:
3         return 0
4     else:
5         return n + f2(n-2)
```

5. L'appel `f2(4)` termine-t-il ? Si oui, indiquer la valeur renvoyée par `f2(4)` ; sinon, justifier brièvement.

6. L'appel `f2(5)` termine-t-il ? Si oui, indiquer la valeur renvoyée par `f2(5)` ; sinon, justifier brièvement.

7. Déterminer l'ensemble des entiers naturels `n` pour lesquels l'appel `f2(n)` termine.

8. Écrire une fonction Python `infini`, récursive, telle que l'appel `infini(n)` ne termine pour aucun entier `n`.

Partie C : le problème de l'arrêt

On se demande maintenant s'il est possible d'écrire une fonction `arret` qui prend en arguments une chaîne de caractères `code_f` contenant le code d'une fonction `f` et un argument `x` de `f`, et tel que `arret(code_f, x)` renvoie `True` si l'appel `f(x)` va terminer et `False` sinon.

Dans la suite de cet exercice, on suppose disposer d'une telle fonction `arret` et on implémente la fonction suivante, utilisant cette fonction `arret`, ainsi que la fonction `infini` de la question précédente dont l'appel `infini(n)` ne termine jamais quelle que soit la valeur de `n`.

```
1 def paradoxe(x):
2     if arret(x, x):
3         infini(42)
4     else:
5         return 0
```

De même, on suppose disposer d'une variable `code_paradoxe` contenant le code de la fonction `paradoxe` sous la forme d'une chaîne de caractères, et on s'intéresse à l'appel `paradoxe(code_paradoxe)`.

Cet appel de fonction commence par effectuer le test `arret(code_paradoxe, code_paradoxe)` dans le `if` de la ligne 2.

9. Dans le cas où `arret(code_paradoxe, code_paradoxe)` renvoie `True`, préciser la prochaine instruction à être exécutée. Dans ce cas, expliquer si l'appel `paradoxe(code_paradoxe)` termine.

10. Dans le cas où `arret(code_paradoxe, code_paradoxe)` renvoie `False`, préciser la prochaine instruction à être exécutée. Dans ce cas, expliquer si l'appel `paradoxe(code_paradoxe)` termine.

11. En déduire qu'une telle fonction `arret` ne peut exister.

EXERCICE 2 (6 points)

Cet exercice porte sur la récursivité.

Au rugby, une équipe peut marquer, pour simplifier :

- soit 3 points (pénalité) ;
- soit 5 points (essai non transformé) ;
- soit 7 points (essai transformé).

Partie A

On souhaite savoir s'il est possible d'obtenir un score donné avec uniquement des pénalités, c'est-à-dire avec une succession de "coups" à 3 points.

1. Écrire une fonction `possible_avec_penalites_seules` qui prend en paramètre un score et qui renvoie `True` si le score passé en paramètre peut être marqué uniquement avec des pénalités.

Exemple:

```
>>> possible_avec_penalites_seules(15)
True
>>> possible_avec_penalites_seules(10)
False
```

2. Recopier et compléter le tableau suivant qui précise, pour les scores de 0 à 10, les évolutions du score menant à un total donné et le nombre de façons différentes d'obtenir ce total. Par exemple, pour obtenir un score de 8, en partant de 0, il y a 2 possibilités :

- Soit l'équipe marque un essai non transformé, atteignant 5 points, puis une pénalité, atteignant 8 points ;
- Soit l'équipe marque une pénalité, atteignant 3 points, puis un essai non transformé, atteignant 8 points.

score	liste des solutions	nombre de solutions
0	[0]	1
1	[]	0
2		
3		
4		
5		
6		
7		
8	[0, 5, 8], [0, 3, 8]	2
9		1
10	[0,3,10]	3

En notant $f(n)$ le nombre de possibilités d'obtenir le score n, on admet que pour $n > 6$ on a la relation suivante :

$$f(n) = f(n-3) + f(n-5) + f(n-7)$$

3. Vérifier cette relation pour $n = 10$ à l'aide du tableau établi à la question 2.

On veut écrire une fonction récursive `nb_solutions`, qui prend en paramètre un entier positif quelconque correspondant à un score, et qui renvoie le nombre de façons d'obtenir ce score donné.

4. Déterminer tous les cas de base, pour chaque entier n de 0 à 6, de cette fonction récursive.

5. Écrire la fonction récursive `nb_solutions`, qui prend en paramètre un entier positif quelconque correspondant à un score, et qui renvoie le nombre de possibilités d'obtenir ce score donné.

6. Lors de l'appel de `nb_solutions(score)`, on se rend compte que le nombre d'appels récursif augmente très rapidement lorsque `score` augmente. Nommer une méthode algorithmique optimisant le nombre d'appels récursifs.

On veut écrire une fonction récursive `solutions_possibles`, qui prend en paramètre un entier positif quelconque correspondant à un score, et qui renvoie la liste composée de toutes les listes représentant les possibilités d'obtenir ce score.

Par exemple :

```
>>> solutions_possibles(8)
[[0, 5, 8], [0, 3, 8]]
```

7. Déterminer quelles lignes du tableau permettent de construire rapidement la liste renvoyée par l'appel `solutions_possibles(11)`.

8. Recopier et compléter la fonction `solutions_possibles` suivante, qui prend en paramètre un score et qui renvoie la liste des possibilités d'obtenir ce score :

```
def solutions_possibles(score):
    if score < 0:
        resultat = []
    elif score == 0:
        resultat = ...
    else:
        resultat = []
        for coup in [..., 5, ...]:
            liste = solutions_possibles(score - coup)
            for solution in liste:
                solution.append(...)
            resultat.append(...)
    return resultat
```

EXERCICE 3 (8 points)

Cet exercice porte sur la programmation Python (dictionnaire), les bases de données relationnelles et les requêtes SQL.

Le Tour de France est une course cycliste qui se déroule chaque année. Chaque jour, les coureurs s'affrontent pour remporter l'étape du jour, ce qui détermine un classement d'étape. Le coureur avec le temps cumulé le plus bas sur l'ensemble des étapes mène le classement général. Chaque participant est repéré par un dossard et appartient à une équipe. En 2023, 22 équipes de 8 coureurs, soit 176 cyclistes ont pris le départ du tour.

Partie A

Dans cette partie, nous allons utiliser trois dictionnaires.

Le premier, appelé `participants`, a pour clés les noms complets des coureurs et pour valeurs les numéros de dossard correspondants.

Le deuxième, appelé `temps_etapes`, utilise les numéros de dossard comme clés et contient une liste des temps d'arrivée de chaque étape en seconde.

Le troisième, appelé `classement_general`, utilise également les numéros de dossard comme clés et indique le classement général mis à jour à la fin de chaque nouvelle étape.

Par exemple, à la fin de la quatrième étape, voilà les trois premiers éléments de ces dictionnaires :

```
participants = {"VINGEGAARD Jonas": 1, "BENOOT Tiesj": 2, "KELDERMAN
Wilco": 3, ...}

temps_etapes = {1: [15781, 17199, 16995, 15928], 2: [15960, 17199,
16995, 15928], ...}

classement_general = {1: 6, 2: 30, 3: 13, ...}
```

1. En utilisant ces dictionnaires, écrire une instruction permettant d'obtenir :

- le numéro de dossard de `PHILIPSEN Jasper` ;
- le classement général de `PHILIPSEN Jasper` ;
- le temps, en seconde, mis par le cycliste `PINOT Thibaut` pour courir la quatrième étape.

2. Écrire une fonction `calcul_temps_total` qui a pour paramètre le numéro d'un dossard `d` et qui renvoie le temps total en seconde mis par ce coureur depuis le départ du tour de France.

3. Le dictionnaire `temps_etapes` étant remis à jour après la fin d'une étape, recopier et compléter les lignes 8, 9 et 14 du programme suivant afin que le dictionnaire `classement_general` soit aussi mis à jour.

```
1  classement = []
2
3  for numero_dossard in temps_etapes:
4      element = (numero_dossard,
                  calcul_temps_total(numero_dossard))
5      classement.append(element)
6      pos = len(classement) - 2
7
8      while pos >= 0 and element[...] < classement[pos][...]:
9          classement[pos + 1] = ...
10         pos = pos - 1
11         classement[pos + 1] = element
12
13 for i in range(len(classement)):
14     classement_general[...] = i + 1
```

On suppose qu'on dispose d'un tableau `tableau_temps` composé de tuples contenant le numéro du dossard, le nom, et le temps total en seconde de chaque coureur, trié par ordre croissant de temps. On donne ci-dessous un aperçu du début du tableau :

```
tableau_temps = [(1, "VINGEGAARD Jonas", 65903),
                 (3, "KELDERMAN Wilco", 65987),
                 (2, "BENOOT Tiesj", 66082),
                 ...]
```

On souhaite créer une variable Python `tableau_final` de type liste de listes :

```
[[1, "VINGEGAARD Jonas", 65903],
 [3, "KELDERMAN Wilco", 84],
 [2, "BENOOT Tiesj", 179]]
...
```

Pour le premier, la troisième valeur de la première liste est le temps total mis par le vainqueur. Pour les autres coureurs, la troisième valeur des autres listes est l'écart de temps mis avec le premier.

Par exemple : 84 = 65987 - 65903 et 179 = 66082 - 65903.

4. Recopier et compléter le programme pour qu'il en soit ainsi :

```
5  tableau_final = []
6  difference_temps = 0
7  premier = True
8  for ligne in tableau_temps:
9      coureur = [ligne[0]]
10     coureur.append(ligne[1])
11     if premier:
12         temps_premier = ligne[2]
13         coureur.append(temps_premier)
14         premier = False
15     else:
16         difference_temps = ligne[2] - ...
17         coureur.append(...)
18     tableau_final.append(...)
```

Partie B

Dans cet exercice, on pourra utiliser les mots clés suivants du langage SQL :

`SELECT`, `FROM`, `WHERE`, `JOIN`, `ON`, `INSERT INTO`, `VALUES`, `MIN`, `MAX`, `OR`, `AND` et `ORDER BY`.

Si `propriete` est un des attributs d'une relation, les fonctions d'agrégation `MIN(propriete)`, `MAX(propriete)`, `SUM(propriete)` renvoient, respectivement, la plus petite, la plus grande valeur et la somme des valeurs des attributs sélectionnés.

On considère la base de données du tour de France 2023 dont le schéma relationnel est donné ci-dessous :

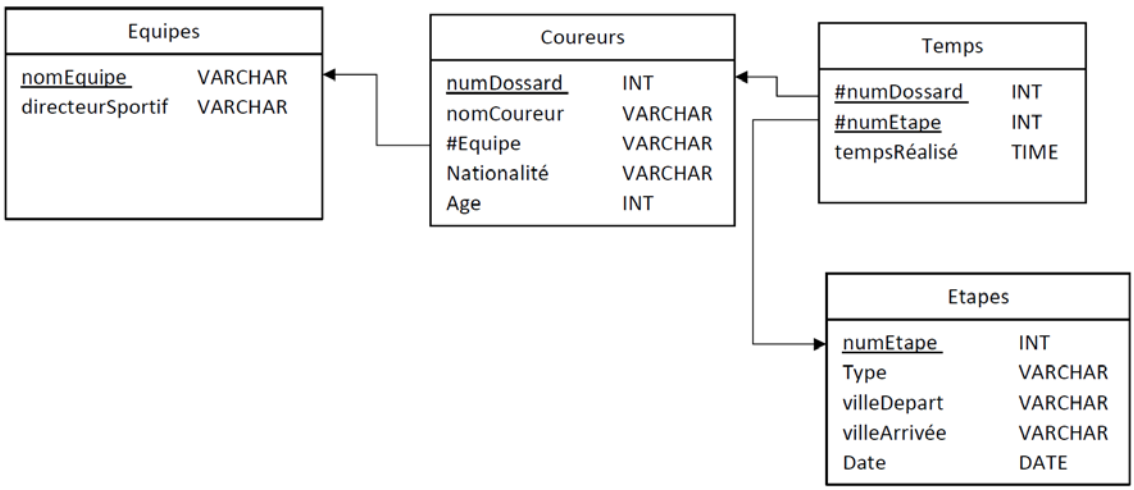

Figure 1. Schéma Relationnel

Dans ce schéma, les clés primaires sont soulignées et les clés étrangères sont précédées du symbole #.

5. Expliquer pourquoi, dans la relation `Temps`, il est nécessaire de prendre le couple (`numDossard`, `NumEtape`) comme clé primaire.

6. Expliquer ce que renvoie la requête SQL suivante :

```
SELECT nomCoureur
FROM Coureurs
WHERE Equipe = 'Cofidis';
```

7. Écrire une requête SQL permettant d'obtenir les dates de toutes les étapes de type `'contre-la-montre'` du tour de France 2023.

8. Écrire une requête SQL permettant d'obtenir le nom du directeur sportif du coureur BARDET Romain.

9. À la fin de la cinquième étape, on veut actualiser la table `Temps` avec les données du jour. Expliquer pourquoi la suite des deux requêtes SQL ci-dessous provoque une erreur.

```
INSERT INTO Temps VALUES (1, 5, 14267);
INSERT INTO Etapes VALUES(5, 'Montagne', 'Pau', 'Laruns',
05/07/2023);
```

10. Expliquer quelle modification est à effectuer pour apporter une solution au problème constaté à la question précédente.

11. Écrire une requête SQL donnant le temps total en course mis par BARDET Romain depuis le départ du tour de France 2023.