TERMINALE GENERALE

ENSEIGNEMENT DE SPECIALITÉ

NUMERIQUE ET SCIENCES INFORMATIQUES
Épreuve du 6 février 2025

Durée : **3 heures 30**

Vous n'êtes pas autorisé à quitter la salle de composition
avant une durée de **2 heures** après le début de l'épreuve.

Consignes

• **L'usage de la calculatrice n'est pas autorisé**.

• Vous devez traiter les trois exercices du sujet.

• Vous indiquerez clairement sur vos copies les numéros des exercices et des questions traités.

• Vous devez répondre aux questions uniquement sur vos copies. Les éléments éventuellement écrits sur le sujet ne seront pas corrigés.

• La qualité de la rédaction et la précision des réponses sont des éléments importants dans l'appréciation des résultats.

Exercice 1 (6 points)

Cet exercice porte sur la programmation Python, la programmation orientée objet et la structure de données pile.

Le *mélange faro* consiste à partager un jeu de cartes en deux moitiés et intercaler les cartes de ces deux moitiés.

Pour tout l'exercice on notera `n` le nombre de cartes et on considèrera qu'il est pair.

Pour modéliser le jeu de cartes, on décide d'utiliser une pile qui sera une instance de la classe `Pile` dont on donne ici l'interface.

- Le constructeur Pile ne prend pas de paramètres et renvoie une pile vide.
  `jeu = Pile()` *`# crée une pile vide référencée par jeu`*
- La méthode `empile` prend en paramètre une valeur et l'empile sur la pile.
  `jeu.empile(1)` *`# empile la valeur 1 sur la pile jeu`*
- La méthode `depile` ne prend pas de paramètres et retire le dernier élément empilé d'une pile non vide et renvoie sa valeur.
  `print(jeu.depile())` *`# dépile 1 et affiche cette valeur`*
- La méthode `est_vide` ne prend pas de paramètres et renvoie un booléen indiquant si la pile est vide.
  `print(jeu.est_vide())` *`# affiche True puisque la pile est vide`*

Le jeu de cartes est alors modélisé par une pile appelée `jeu` de sommet 1, puis 2 en dessous, *et caetera* jusqu'au bas de la pile qui contient `n`, comme illustré sur la figure ci-dessous.

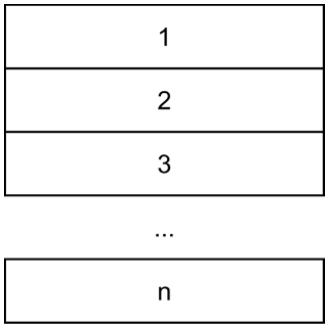

Figure 1. Pile représentant un jeu de cartes

Le mélange faro est réalisé ainsi :

- **Étape 1** : on dépile la moitié de `jeu` et chaque élément dépilé est empilé dans une deuxième pile appelée `moitie1` ;
- **Étape 2** : on dépile le reste de `jeu` et chaque élément dépilé est empilé dans une troisième pile appelée `moitie2` ;
- **Étape 3** : on empile alternativement dans `jeu` et dans cet ordre un élément de `moitie1` puis un élément de `moitie2` jusqu'à vider ces 2 piles.

Dans l'exemple suivant les contenus initiaux de `jeu`, `moitie1` et `moitie2` sont représentés ci-dessous :

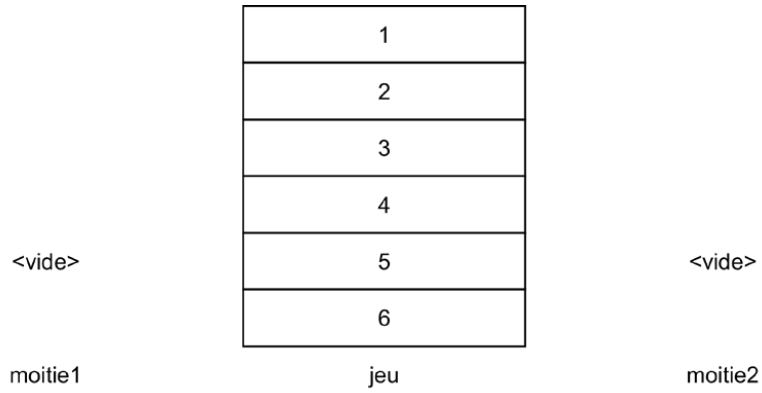

Figure 2. Contenus initiaux des 3 piles

1. Représenter sur votre copie les contenus de ces trois piles à la fin de chaque étape du mélange faro.

Voici le code de la fonction `produire_jeu` qui prend en paramètre un entier `n` supposé pair et qui renvoie une instance de la classe `Pile` qui représente le jeu de cartes.

```
1 def produire_jeu(n):
2     resultat = Pile()
3     for i in range(...):
4         resultat.empile(...)
5     return resultat
```

2. Recopier et compléter sur votre copie le code de la fonction `produire_jeu`.

Ci-après figure le code de la fonction `scinder_jeu` qui prend en paramètres une instance `p` de taille paire de la classe `Pile` qui est le jeu que l'on veut partager en 2 moitiés, un entier `n` qui est la taille de la pile et qui renvoie deux piles qui sont les deux moitiés du jeu.

```
1 def scinder_jeu(p, n):
2     m1 = Pile()
3     m2 = Pile
4     for i in range(n):
5         m1.empile(p.depile())
6     for i in range(n):
7         m2.empile(p.depile())
8     return m1, m2
```

3. Ce code comporte des erreurs. Indiquer les numéros de lignes à rectifier ainsi que les rectifications à apporter.

4. Écrire une fonction `recombiner` qui prend en paramètres deux instances `m1` et `m2` de la classe `Pile` qui sont respectivement la première et la deuxième moitié d'un jeu de cartes et qui renvoie une instance de la classe `Pile` qui est le jeu obtenu en y empilant alternativement et dans cet ordre les éléments de `m1` et de `m2`.

5. En utilisant les fonctions `scinder_jeu` et `recombiner`, écrire une fonction `faro` qui prend en paramètres `p` une instance de la classe `Pile` qui est le jeu que l'on veut mélanger, un entier `n` qui est la taille de la pile et qui renvoie une instance de la classe `Pile` qui contient le jeu obtenu en appliquant le mélange faro.

Une propriété mathématique assure qu'étant donné un jeu de n cartes (n pair), en répétant suffisamment de fois le mélange faro, on finira par remettre le jeu dans l'ordre initial. On aimerait trouver, pour un entier n donné, ce nombre minimal de répétitions nécessaires. Pour cela, on considère une fonction `identiques` qui prend en arguments deux instances de la classe `Pile` et qui renvoie un booléen indiquant si ces deux piles ont les mêmes éléments, en même nombre et dans le même ordre.

La fonction `identiques` ne modifie pas les piles données en entrée.

Pour s'assurer que la fonction `identiques` fonctionne correctement, on a commencé à produire un jeu de tests :

```
1 p1 = Pile()
2 p1.empile(1)
3 p2 = Pile()
4 assert not identiques(p1, p2)
```

6. Compléter ce jeu de tests pour s'assurer que l'on couvre les deux cas suivants :
* les piles sont différentes, mais de même taille ;
* les piles sont identiques.

7. Écrire une fonction `ordre_faro` qui prend en paramètres un entier `n` pair et qui renvoie le plus petit nombre de répétitions du mélange faro pour qu'un jeu de n cartes soit remis dans son ordre initial.

Exercice 2 (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 trois instructions permettant d'obtenir :

* le numéro de dossard de `PHILIPSEN Jasper` dans la variable `num_dos` ;

* le classement général de `PHILIPSEN Jasper` dans la variable `clas_gen` ;

* le temps, en seconde, mis par `PHILIPSEN Jasper` pour courir la quatrième étape dans la variable `temps_etape4`.

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 les lignes 16, 17 et 18 du programme suivant 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 cette partie, 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 :

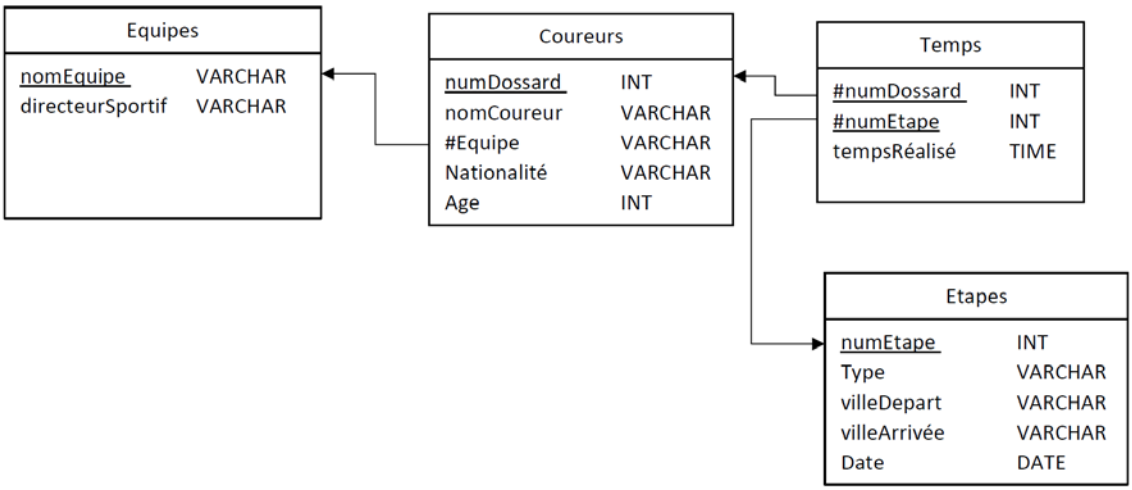

Figure 1. Schéma Relationnel

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.

Exercice 3 (6 points)

Thèmes abordés : structures de données, programmation.

Le « jeu de la vie » se déroule sur une grille à deux dimensions dont les cases, qu'on appelle des « cellules », par analogie avec les cellules vivantes, peuvent prendre deux états distincts : « vivante » (= 1) ou « morte » (= 0).

Une cellule possède au plus huit voisins, qui sont les cellules adjacentes horizontalement, verticalement et diagonalement.

À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

- Règle 1 : une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît) ; sinon, elle reste à l'état « morte »
- Règle 2 : une cellule vivante possédant deux ou trois voisines vivantes reste vivante, sinon elle meurt.

Voici un exemple d'évolution du jeu de la vie appliquée à la cellule centrale :

1	1	0
0	0	0
0	0	1

devient par la règle 1

	1	

1	0	0
0	1	1
1	0	0

reste par la règle 2

	1	

0	0	0
1	1	0
0	0	0

devient par la règle 2

	0	

1	1	0
0	1	1
1	1	0

devient par la règle 2

	0	

Pour initialiser le jeu, on crée en langage Python une grille de dimension 8x8, modélisée par une liste de listes.

1. Initialisation du tableau :

a. Parmi les deux scripts proposés, indiquer celui qui vous semble le plus adapté pour initialiser un tableau de 0. Justifier votre choix.

Choix 1	Choix 2

`1 | ligne = [0,0,0,0,0,0,0,0]`
`2 | jeu = []`
`3 | for i in range(8) :`
`4 |  jeu.append(ligne)`
	`1 | jeu = []`
	`2 | for i in range(8) :`
	`3 |  ligne = [0,0,0,0,0,0,0,0]`
	`4 |  jeu.append(ligne)`

b. Donner l'instruction permettant de modifier la grille `jeu` afin d'obtenir

```
>>> jeu
[[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0]]
```

2. a. Écrire en langage Python une fonction `remplissage(n, jeu)` qui prend en paramètres un entier `n` et une grille `jeu`, et qui ajoute aléatoirement exactement `n` cellules vivantes dans le tableau `jeu`.
Remarque : Pour cela, on pourra importer et utiliser la fonction `randint` du module `random`.

b. Quelles conditions doit vérifier le paramètre `n` dans la fonction `remplissage` ?

On propose la fonction en langage Python `nombre_de_vivants(i, j, jeu)` qui prend en paramètres deux entiers `i` et `j` ainsi qu'une grille `jeu` et qui renvoie le nombre de voisins **vivants** de la cellule `tab[i][j]` :

```
1 | def nombre_de_vivants(i, j, jeu) :
2 |     nb = 0
3 |     voisins = [(i-1,j-1), (i-1,j), (i-1,j+1), (i,j+1),
4 |                (i+1,j+1), (i+1,j), (i+1,j-1), (i,j-1)]
5 |     for e in voisins :
6 |         if 0 <= ... < 8 and 0 <= ... < 8 :
7 |             nb = nb + jeu[...][...]
8 |     return nb
```

3. Recopier et compléter les lignes 6 et 7 pour que la fonction réponde à la demande.

4. En utilisant la fonction `nombre_de_vivants(i, j, jeu)` précédente, écrire en langage Python une fonction `transfo_cellule(i, j, jeu)` qui prend en paramètres deux entiers `i` et `j` ainsi qu'une grille `jeu` et renvoie ce que sera le nouvel état de la cellule `jeu[i][j]` à la prochaine étape (0 ou 1) sans modifier la grille `jeu`.