BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 1

Durée de l'épreuve : **3 heures 30**

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

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 17 pages numérotées de 1/17 à 17/17.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur les algorithmes de tri et d'un parcours d'arbre binaire de recherche et sur la programation orientée objet

Contexte

Un club d'athlétisme organise une compétition inter-scolaire regroupant quatre disciplines :

- le 100 mètres ;
- le saut en longueur ;
- le lancer du poids ;
- le 1500 mètres.

Chaque athlète réalise une performance dans chacune de ces épreuves. Pour établir le classement final, un score global est calculé à partir de ses résultats selon des règles de conversion simples. L'objectif du programme est de modéliser ces athlètes et d'implémenter différentes méthodes de tri pour les classer du premier au dernier.

Le score d'un athlète est la somme des points obtenus dans chaque épreuve. Ces points s'obtiennent selon les règles suivantes :

- points du **100 m** : $(20 - t) * 10$ avec t le temps en secondes ;
- points du **saut en longueur** : $d * 20$ avec d la distance en mètres ;
- points du **lancer de poids** : $d * 10$ avec d la distance en mètres ;
- points du **1500 m** : $(500 - t) * 1$ avec t le temps en secondes.

Partie A : Dictionnaire et tri

1. L'athlète **Alex** a réalisé les performances suivantes :
   - **100 m** : 13.0 s ;
   - **saut en longueur** : 5.2 m ;
   - **lancer de poids** : 9.0 m ;
   - **1500 m** : 310.0 s.

   Calculer le score total de l'athlète.

Pour faire le calcul des points obtenus à une épreuve, on dispose de la fonction `nb_points` ci-dessous :

```
1 def nb_points(epreuve, valeur):
2     points = 0
3     if epreuve == '100m':
4         points = (20 - valeur) * 10
```

```
 5      elif epreuve == 'longueur':
 6          ...
 7      elif epreuve == 'poids':
 8          ...
 9      elif epreuve == '1500m':
10          ...
11      return ...
```

2. Recopier et compléter les lignes 6, 8, 10 et 11 de la fonction `nb_points`.

Les performances d'un athlète sont regroupées dans un dictionnaire dont les clés sont les différentes épreuves. Chaque athlète est représenté par un dictionnaire qui stocke son nom, ses performances et son score. Tous les athlètes sont regroupés dans une liste.

On donne ci-dessous un exemple d'une liste d'athlètes avant le calcul des scores.

```
l_athletes = [
    {'nom': 'Alex',
     'performances': {'100m': 13.0, 'longueur': 5.2, 'poids':
9.0, '1500m': 310.0},
     'score': 0
    },
    {'nom': 'Anna',
     'performances': {'100m': 12.2, 'longueur': 5.8, 'poids':
8.5, '1500m': 230.0},
     'score': 0
    },
    {'nom': 'Rayan',
     'performances': {'100m': 13.5, 'longueur': 5.0, 'poids':
9.5, '1500m': 205.0},
     'score': 0
    }
]
```

Pour faire le calcul du score obtenu par un athlète, on dispose de la fonction `score` ci-dessous :

```
1 def score(athlete):
2     total = 0
3     performances = athlete[...]
4     for epreuve in performances:
5         valeur = performances[...]
6         total += ...
7     athlete['score'] = total
```

3. Recopier et compléter les lignes 3, 5 et 6 de la fonction `score`.

Pour obtenir le classement des athlètes du plus grand score au plus petit score, on réorganise la liste `l` des athlètes avec la fonction `classer(l)` qui prend en paramètre la liste `l` d'athlètes :

```
 1 def classer(l):
 2     n = len(l)
 3     for i in range(n):
 4         max_index = i
 5         for j in range(i + 1, n):
 6             if ...:
 7                 max_index = j
 8         temp = l[i]
 9         l[i] = l[max_index]
10         l[max_index] = temp
```

4. Recopier et compléter la ligne 6 de la fonction `classer`.

5. Parmi les propositions suivantes, choisir celle qui correspond au type de tri effectué par la fonction `classer`.

   - tri par insertion
   - tri par sélection
   - tri à bulles
   - tri fusion

6. Donner le coût de la fonction `classer`, c'est à dire le nombre de comparaisons d'éléments de la liste, en fonction de la taille de la liste.

Partie B : Arbre binaire de recherche

Une nouvelle approche est adoptée, fondée sur la programmation orientée objet et la mise en œuvre d'un arbre binaire de recherche, afin de modéliser les athlètes et de produire leur classement automatique en fonction de leur score.

```
 1 class Athlete:
 2     def __init__(self, nom, m100, longueur, poids, m1500):
 3         self.nom = nom
 4         self.m100 = m100
 5         self.longueur = longueur
 6         self.poids = poids
 7         self.m1500 = m1500
 8         self.score = self.calculer_score()
 9
10     def calculer_score(self):
11         ...
```

7. Créer l'objet `alex` qui modélise l'athlète **Alex** avec ses performances données à la question 3

8. Écrire le code de la méthode `calculer_score()`.

Une fois créés, les athlètes sont insérés dans un arbre binaire de recherche. Chaque athlète y constitue un nœud, positionné selon son score de manière à assurer la propriété suivante : pour chaque athlète, tous les athlètes de son sous-arbre gauche

ont un score strictement plus petit et tous les athlètes de son sous-arbre droit un score plus grand ou égal.

9. Dessiner l'arbre obtenu après l'insertion, dans l'ordre, des six athlètes suivants.

Ordre	Nom	Score
1	Martin	354
2	Lena	351
3	Rayan	350
4	Yanis	355
5	Ninon	351
6	Ana	356

La classe `Noeud` qui assure l'organisation structurée des athlètes dans un arbre binaire de recherche.

```
class Noeud:
    def __init__(self, athlete):
        self.valeur = athlete
        self.gauche = None
        self.droite = None

    def inserer(self, athlete):
        if athlete.score < self.valeur.score:
            if self.gauche == ...:
                self.gauche = Noeud(athlete)
            else:
                ...
        else:
            if self.droite == ...:
                self.droite = Noeud(athlete)
            else:
                ...
```

10. Recopier et compléter le code de la méthode `inserer`.

On ajoute la méthode ci-dessous à la classe `Noeud`, qui permet de classer les athlètes de manière décroissante.

```
    def classer(self, classement=None):
        if classement is None:
            classement = []
        if self.droite is not None:
            self.droite.classer(classement)
        classement.append(self.valeur.nom)
```

```
7           if self.gauche is not None:
8               self.gauche.classer(classement)
9           return classement
```

11. Expliquer quel type de parcours est mis en œuvre dans la méthode `classer`, et préciser dans quel ordre sont explorés les nœuds et comment cela permet de classer les athlètes en fonction de leur score.

12. Donner un avantage et un inconvénient des deux approches utilisées pour classer les athlètes :

    - Utiliser des dictionnaires et un tri
    - Utiliser un arbre binaire de recherche approche objet avec insertion dans un arbre binaire

Exercice 2 (6 points)

Cet exercice porte sur la sécurisation des communications et la programmation.

Le chiffrement de Polybe est un algorithme de chiffrement par substitution qui utilise un tableau dans lequel sont réparties les 26 lettres de l'alphabet et les 10 chiffres.

Exemple de tableau :

	1	**2**	**3**	**4**	**5**	**6**
1	Q	7	A	X	2	J
2	9	E	H	0	R	M
3	L	Z	4	W	D	O
4	6	V	N	B	8	K
5	P	Y	1	S	T	F
6	G	C	3	I	U	5

Chaque caractère est alors associé à un couple d'entiers construit à partir de sa position dans le tableau. Dans l'exemple ci-dessus, la lettre `N` située ligne 4, colonne 3 est associée au couple d'entier (4,3).

Avec la répartition donnée dans le tableau ci-dessus, le message `NSI` sera chiffré sous la forme (4,3) (5,4) (6,4).

1. Déchiffrer le message (6,2) (3,6) (3,5) (2,2) en utilisant la grille ci-dessus.

Une façon simple de construire un tableau consiste à choisir un mot qu'on appellera *clé*, à écrire cette clé dans le tableau puis à compléter avec les autres lettres dans l'ordre alphabétique, et enfin avec les chiffres de 0 à 9 par ordre croissant. Dans tout l'exercice, la clé utilisée sera toujours composée de lettres différentes, sans aucune répétition.

Exemple : Avec la clé `2048ALGORITHMES`, l'ordre d'insertion des caractères dans le tableau est `2048ALGORITHMESBCDFJKNPQUVWXYZ135679`, la grille obtenue est donc :

	1	**2**	**3**	**4**	**5**	**6**
1	2	0	4	8	A	L
2	G	O	R	I	T	H
3	M	E	S	B	C	D
4	F	J	K	N	P	Q
5	U	V	W	X	Y	Z
6	1	3	5	6	7	9

2. Chiffrer le message `BAC` avec la clé `SECURITY1024`.

3. Expliquer pourquoi le chiffrement de Polybe peut être qualifié de symétrique.

Le code de la fonction `generer_ordre` est fourni ci-dessous :

```
1 def generer_ordre(cle):
2     ordre_insertion = cle
3     alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
4     for lettre in alphabet:
5         if lettre not in ordre_insertion:
6             ordre_insertion = ordre_insertion + lettre
7     return ordre_insertion
```

4. Quel sera le résultat de l'appel `generer_ordre('AXU7')` ?

5. Coder la fonction `grille_vide(n)` qui renvoie un tableau de n lignes constituées chacune de n chaînes de caractères vides. Par exemple, l'appel `grille_vide(3)` renvoie `[['','',''],['','',''],['','','']]`.

Le code de la fonction `generer_grille` est fourni ci-dessous :

```
1 def generer_grille(cle):
2     ordre_insertion = generer_ordre(cle)
3     grille = grille_vide(6)
4     indice = 0
5     for i in range(...):
6         for j in range(...):
7             grille[i][j] = ...
8             indice = indice + 1
9     return grille
```

6. Recopier et compléter les lignes 5, 6 et 7 de la fonction `generer_grille` qui prend en paramètre la clé et renvoie un tableau représentant la grille.

7. Recopier et compléter la ligne 5 de la fonction `dechiffrer` qui prend en paramètres une clé et le message chiffré sous la forme d'un tableau de tuples puis renvoie le message déchiffré sous la forme d'une chaîne de caractères. Par exemple, comme `NSI` se chiffre en (4,4) (3,3) (2,4) avec la clé `2048ALGORITHMES`, l'appel `dechiffrer('2048ALGORITHMES',[(4,4),(3,3),(2,4)])` renvoie `'NSI'`.

```
1 def dechiffrer(cle, message):
2     resultat = ''
3     grille = generer_grille(cle)
4     for t in message:
5         resultat = resultat + grille[...][...]
6     return resultat
```

On souhaite maintenant écrire une fonction `chiffrer`. Pour éviter d'avoir à parcourir la grille pour chaque lettre du message à chiffrer, on va construire un dictionnaire dont les clés sont les lettres de l'alphabet et les 10 chiffres et dont les valeurs sont les positions associées dans la grille sous forme de tuples.

Par exemple, le début du dictionnaire associé à la grille qui correspond à la clé `2048ALGORITHMES` est : `{'2': (1, 1), '0': (1, 2), '4': (1, 3), '8': (1, 4), 'A': (1, 5) ... }`

8. Écrire la fonction `generer_dico` prenant en paramètre la clé et renvoyant le dictionnaire associé.

9. Écrire la fonction `chiffrer` qui prend en paramètre la clé et le message puis renvoie le message chiffré sous forme de liste de tuples.

Alice souhaite envoyer des informations secrètes à Bob chaque jour. Pour cela, ils décident d'utiliser le chiffrement de Polybe et pour plus de sécurité, ils changeront de clé quotidiennement. Alice propose à Bob d'utiliser des méthodes de chiffrements asymétriques pour s'échanger la clé.

10. Expliquer la différence entre un algorithme de chiffrement symétrique et un algorithme de chiffrement asymétrique.

Exercice 3 (8 points)

Cet exercice porte sur la programmation orientée objet, la récursivité et les bases de données relationnelles.

Le champ de mines du démineur est représenté par une grille.

Chaque case de cette grille peut cacher une mine, ou être vide.

Le déroulement du jeu :

- Si on choisit sur une case libre qui a au moins une mine dans une case voisine, un chiffre apparaît. Ce chiffre indique combien de mines se trouvent dans les cases voisines. Une case a au maximum 8 cases voisines.
- Si on choisit une case vide dont toutes les cases voisines sont vides, la case s'affiche vide. Ensuite, le jeu dévoile automatiquement toutes les cases vides voisines, et ainsi de suite, jusqu'à ce qu'il rencontre des cases avec des chiffres.
- La fin de la partie : Si on choisit sur une mine, la partie est perdue. Si on découvre toutes les cases libres, la partie est gagnée.

	0	1	2	3	4	5	6
0	1	-1	2	1	2	1	1
1	1	2	3	-1	2	-1	1
2	1	2	-1	2	2	1	1
3	2	-1	2	2	1	1	0
4	-1	2	1	1	-1	2	1
5	1	1	0	1	1	2	-1

Figure 1. Une grille 6x7 du démineur avec les informations du jeu.

La figure 1 représente une grille du jeu du démineur de 6 lignes : hauteur et de 7 colonnes : largeur, où l'on a dévoilé la position des 8 mines de la grille représentées par l'entier relatif -1 ainsi que les informations utiles au joueur qui lui seront dévoilées au fur et à mesure du jeu.

- La case de coordonnées (1, 2) contient le chiffre 3 car elle a trois de ses cases voisines qui contiennent une mine : les cases (0, 1), (2, 2) et (1, 3).
- La case (3, 6) contient le chiffre 0 car aucune case qui lui est voisine ne contient de mine.

Partie A : La classe `Demineur`

On propose de programmer le jeu du démineur à l'aide du langage de programmation Python. On utilise alors le paradigme de programmation orienté objet. On donne un extrait de la classe `Demineur` suivante.

```
 1 class Demineur :
 2     def __init__(self, hauteur, largeur, pourcentage_mines)
:
 3         '''
 4         Pour un pourcentage de 15,6 on donnera au paramètre
 5         pourcentage_mines la valeur de 0.156.
 6         '''
 7         assert ..., 'Le pourcentage de mines doit être
compris entre 10% et 30%.'
 8         ...hauteur = hauteur
 9         ...largeur = largeur
10         ...pourcentage_mines = pourcentage_mines
```

1. Recopier et compléter la ligne 7 de l'assertion pour que le pourcentage de mines respecte la précondition recommandée.

2. Recopier et compléter les lignes 8, 9 et 10 et compléter les pointillés avec le code manquant.

On donne les pourcentages suivants, conçus pour offrir un bon équilibre entre challenge et jouabilité.

- Débutant : Grille de 8x8 avec ce qui représente 10 mines, environ 15,6% de mines.
- Intermédiaire : Grille de 16x16 cases avec 40 mines, ce qui représente environ 15,6% de mines.
- Expert : Grille de 30x16 cases avec 99 mines, ce qui représente environ 20,6% de mines.

3. Créer une instance `demineur_intermediaire`, de la classe `Demineur` en respectant les recommandations pour le niveau de jeu intermédiaire pour le jeu du démineur.

Partie B : Création de la grille du démineur

La grille du jeu du démineur peut être modélisée par un objet Python de type `list`. Par exemple une grille vide d'un démineur de hauteur 3 et de largeur 5 peut-être représentée par l'objet Python suivant :

```
grille_vide = [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0,
0]]
```

4. Recopier et compléter la méthode `grille_demineur_vide` de la classe `Demineur` qui permet d'initialiser une grille vide du jeu.

```
        def grille_demineur_vide(self):
            return [[0 for _ in range(...)] for _ in
    range(...)]
```

On ajoute dans le constructeur de la classe `Demineur` l'attribut `grille_demineur` qui va permettre d'initialiser une grille vide.

```
        self.grille_demineur = self.grille_demineur_vide()
```

Les mines doivent être placées de façon aléatoire dans la grille et seront représentées par l'entier relatif -1.

On donne alors une méthode `placer_mines`, incomplète, de la classe `Demineur`.

```
 1     def placer_mines(self):
 2         '''
 3         Cette fonction doit placer de façon aléatoire les
mines dans la
 4         grille.
 5         Une mine sera représentée par le nombre -1.
 6         La méthode permet de mettre à jour l'attribut
grille_demineur
 7         '''
 8         compteur_mines = 0 # Nombre de mines placées dans la
grille.
 9         # nombre_bombes contient le nombre de mines à placer
dans la grille.
10         nombre_bombes = self.largeur*self.hauteur \
11                         *self.pourcentage_mines
12         while  compteur_mines < nombre_bombes:
13             ligne = randint(0, self.hauteur - 1)
14             colonne = randint(0, ...)
15             if self.grille_demineur[...][...] == 0:
16                 self.grille_demineur[...][...] = -1
17                 compteur_mines  = ...
```

La documentation Python donne les informations suivantes :

```
>>> from random import randint
>>> help(randint)
Help on method randint in module random:

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

5. Recopier et compléter les lignes 14, 15, 16 et 17 de la méthode `placer_mines` de la classe `demineur`.

On dispose des deux méthodes suivantes.

```
    def voisines(self, coordonnees_case):
        '''
        La méthode voisines renvoie les coordonnées des cases
voisines de la case de la grille
        dont les coordonnées sont données par le tuple
coordonnees_case passé en paramètre.
        Cette méthode renvoie une liste de tuple.

        >>> demineur_1 = Demineur(3, 5,0.20)
        >>> demineur_1.voisines((0, 0))
        [(1, 1), (1, 0), (0, 1)]
        >>> demineur_1.voisines((1, 2))
        [(0, 1), (0, 3), (0, 2), (2, 1), (2, 3), (2, 2), (1,
1), (1, 3)]
        '''

    def nombre_voisines_avec_mines(self, coordonnees_case):
        '''
        La méthode nombre_voisine_avec_mines renvoie le nombre
de cases voisines contenant une mine
        à la case dont les coordonnées sont passés en
paramètre(coordonnees_case) à la fonction.
        '''
```

6. Écrire le code Python de la méthode `nombre_voisines_avec_mines`. On ne demande pas d'écrire le code de la méthode `voisines`.

7. Écrire une méthode `generer_demineur` qui permet de mettre à jour l'attribut `grille_demineur` pour que chaque cellule ne contenant pas de mine, contienne le nombre de mines qui lui sont voisines.

   Par exemple :

```
>>> demineur_nsi = demineur(6, 6, 0.278)
>>> demineur_nsi.generer_demineur()
>>> demineur_nsi.grille_demineur
[[2, -1, 2, 1, 2, 1], [2, -1, 4, -1, 2, -1], [2, 3, -1, 2,
2, 1], [2, -1, 2, 2, 1, 1], [-1, 3, 2, 2, -1, 2], [1, 2, -
1, 2, 2, -1]]
```

	0	1	2	3	4	5
0	2	-1	2	1	2	1
1	2	-1	4	-1	2	-1
2	2	3	-1	2	2	1
3	2	-1	2	2	1	1
4	-1	3	2	2	-1	2
5	1	2	-1	2	2	-1

Figure 2. Représentation de la grille 6x6 du démineur de l'exemple précédent.

Partie C : L'interface utilisateur du jeu du démineur

Les parties précédentes ont permis la création d'une grille du démineur. L'utilisateur se verra afficher une grille où les informations de la grille ne seront pas apparentes. Le joueur fait le choix d'une case et en fonction de ce choix des cases seront découvertes.

- Si la case contient une mine toutes les cases sont alors dévoilées.
- Dans le cas contraire une ou plusieurs cases seront dévoilées.

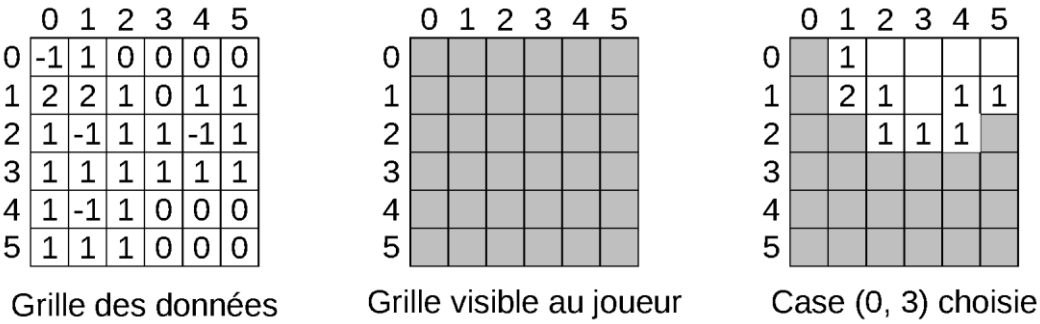

Figure 3. Gérer le visuel du joueur.

Sur la figure 3 on peut lire:

- sur la grille de gauche, les informations du démineur.
- sur la grille au centre, le visuel présenté au joueur, aucune information n'est visible au départ du jeu.
- sur la grille de droite, les informations révélées quand le joueur a choisi la case de coordonnées (0, 3).

On propose de modéliser la visibilité des cellules pour l'utilisateur de la façon suivante.

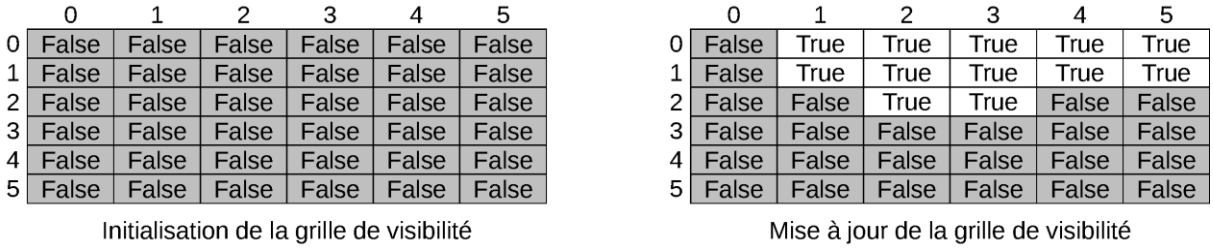

Figure 4. Évolution de la grille de visibilité.

Une grille est utilisée pour connaître l'état de visibilité de l'information des cases.

- `False` : L'information de la case n'est pas visible par l'utilisateur.
- `True` : L'information de la case est visible à l'utilisateur.

Si on considère la figure 4, le constructeur de la classe `Demineur` initialisera l'attribut `grille_visibilite` avec `[[False for_ in range(6)] for _ in range(6)]`.

8. Écrire une méthode `visibilite` de la classe `Demineur` qui permettrait de mettre à jour l'attribut `grille_visibilite` en fonction du choix de l'utilisateur d'une case du démineur. On pourra utiliser la récursivité.

Partie D : Jouer en ligne au démineur

Dans cette partie, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND`, `OR`) et `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE`
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

On donne la possibilité aux joueurs d'enregistrer les scores en ligne pour le jeu du démineur.

On dispose d'une base de données relationnelles avec les tables suivantes.

- `Joueur` : enregistrement du pseudo et du mot de passe de connexion du joueur.

Joueur		
id_joueur	pseudo	mot_de_passe
1	Grimdal	EgxGB6a3bRinllon
2	Raptor	J17NtfMS3Dudjjln
3	PetiteFée	UuukBSvj01VrGoGD
4	Kirna	7NcDFPNl0Xy1MEPb

- `Meilleur_score` : enregistrement du meilleur score d'un joueur pour chaque niveau joué.

Meilleur_score			
joueur	niveau	score	temps
1	facile	1200	72
2	expert	1196	366
3	intermédiaire	997	230
2	intermédiaire	997	200
4	expert	1097	400

- `Demineur` : enregistrement des caractéristiques de chaque niveau du démineur.

Demineur		
niveau	dimension	pourcentage mines
facile	8x8	15.6
intermédiaire	16x16	15.6
expert	30x16	20.6

9. Donner les clés étrangères de la table `Meilleur_score` en précisant pour chacune d'elle à quelle clé primaire elle fait référence.

Une requête envoyée à la base de données a renvoyé la réponse suivante.

niveau	score
expert	1196
intermédiaire	997

10. Donner une requête qui permet d'obtenir les informations du tableau précédent.

Kirna change son mot de passe par cGhhxDE4 en place de 7NcDFPNl0Xy1MEPb.

11. Écrire la requête qui permet de mettre à jour la table `Joueur`.

On donne la requête suivant.

```
SELECT pseudo FROM Joueur
JOIN Meilleur_score
ON Joueur.id_joueur = Meilleur_score.joueur
WHERE niveau = 'expert' AND score > 1000 AND temps < 400;
```

12. Donner le résultat de cette requête.

Une erreur a été faite à la création de la table `Demineur`. Le niveau facile doit être désigné par le mot débutant.

13. Écrire les requêtes qui permettent de corriger la base de données.