

**Exercice 1 : 6 points - Tris, ABR et POO****Partie A : Dictionnaire et tri**

1. Le détail du score est :

- 100 mètres :  $(20 - 13) \times 10 = 70$  ;
- saut en longueur :  $5,2 \times 20 = 104$  ;
- lancer de poids :  $9 \times 10 = 90$  ;
- 1500 mètres :  $500 - 310 = 190$ .

Le score d'Alex est donc de  $70 + 104 + 90 + 190 = 454$ .

```
2. def nb_points(epreuve, valeur):
    points = 0
    if epreuve == '100m':
        points = (20 - valeur) * 10
    elif epreuve == 'longueur':
        points = valeur * 20
    elif epreuve == 'poids':
        points = valeur * 10
    elif epreuve == '1500m':
        points = 500 - valeur
    return points
```

```
3. def score(athlete):
    total = 0
    performances = athlete['performances']
    for epreuve in performances:
        valeur = performances[epreuve]
        total += nb_points(epreuve, valeur)
    athlete['score'] = total
```

```
4. def classer(l):
    n = len(l)
    for i in range(n):
        max_index = i
        for j in range(i + 1, n):
            if l[j]['score'] > l[max_index]['score']:
                max_index = j
        temp = l[i]
        l[i] = l[max_index]
        l[max_index] = temp
```

5. Il s'agit du tri par sélection dans l'ordre décroissant.

6. La fonction classer met en œuvre le tri par sélection. Lors du premier tour de boucle, il faut effectuer  $n - 1$  comparaisons, lors du deuxième  $n - 2, \dots$  jusqu'au dernier tour de boucle où l'on fait 1 comparaisons.

Donc le nombre de comparaisons est  $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2}$ . Le coût est donc quadratique  $\mathcal{O}(n^2)$ .

**Partie B : Arbre binaire de recherche**

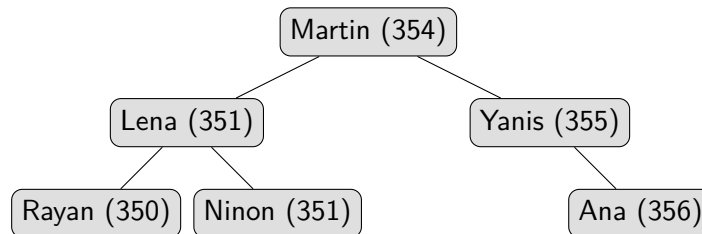
7. On saisit alex = Athlete('Alex', 13.0, 5.2, 9.0, 310.0).

```

8. def calculer_score(self):
    points = (20 - self.m100) * 10
    points += self.longueur * 20
    points += self.poids * 10
    points += 500 - self.m1500
    return points # on renvoie la valeur qui est affectée
                  # à self.score dans le constructeur

```

9. On obtient l'arbre ci-dessous :



```

10. def inserer(self, athlete):
    if athlete.score < self.valeur.score:
        if self.gauche == None: # ou is None
            self.gauche = Noeud(athlete)
        else:
            self.gauche.inserer(athlete)
    else:
        if self.droite == None: # ou is None
            self.droite = Noeud(athlete)
        else:
            self.droite.inserer(athelte)

```

11. Il s'agit d'un parcours en profondeur dans lequel on visite les sous-arbres droits, puis l'on traite le nœud en cours et enfin on visite le sous-arbre gauche. Il s'agit donc d'un parcours infixe *inversé*.

Dans cet arbre, les grands scores sont placés à droite. On obtient donc un classement des scores dans l'ordre décroissant.

12. Si l'on utilise les dictionnaires et le tri par sélection :

- *Avantage* : légèreté, pas de données superflues ;
- *Inconvénient* : tri de coût quadratique.

Si l'on utilise les arbres binaires de recherche :

- *Avantage* : tri efficace, de coût linéarithmique  $\mathcal{O}(n \log_2(n))$  en moyenne ;
- *Inconvénient* : empreinte mémoire plus lourde.

## Exercice 2 : 6 points - Sécurisation des communications et programmation

1. On a  $(6, 2) = 'C'$  ;  $(3, 6) = 'O'$  ;  $(3, 5) = 'D'$  ;  $(2, 2) = 'E'$ . Le message est donc 'CODE'.

2. La table de correspondance est la suivante :

```
# 1  2  3  4  5  6
['S', 'E', 'C', 'U', 'R', 'I'] # 1
['T', 'Y', '1', '0', '2', '4'] # 2
['A', 'B', 'D', 'F', 'G', 'H'] # 3
['J', 'K', 'L', 'M', 'N', 'O'] # 4
['P', 'Q', 'V', 'W', 'X', 'Z'] # 5
['3', '5', '6', '7', '8', '9'] # 6
```

On a 'B' = (3, 2); 'A' = (3, 1); 'C' = (1, 3).

3. La donnée du tableau permet de chiffrer et de déchiffrer le message. Il s'agit donc d'une méthode de chiffrement symétrique.

4. On obtient : 'AXU7BCDEFGHIJKLMNOPQRSTUVWXYZ012345689'.

```
5. def grille_vide(n):
    return [['' for j in range(n)] for i in range(n)]
```

```
6. def generer_grille(cle):
    ordre_insertion = generer_ordre(cle)
    grille = grille_vide(6)
    indice = 0
    for i in range(6):
        for j in range(6):
            grille[i][j] = ordre_insertion[indice]
            indice = indice + 1
    return grille
```

```
7. def dechiffrer(cle, message):
    resultat = ''
    grille = generer_grille(cle)
    for t in message:
        ligne = t[0] - 1
        colonne = t[1] - 1
        resultat = resultat + grille[ligne][colonne]
    return resultat
```

8. Si l'on parcourt (une fois) la grille afin d'établir le dictionnaire, on peut faire :

```
def generer_dico(cle):
    grille = generer_grille(cle)
    dico = {}
    for i in range(6):
        for j in range(6):
            caractere = grille[i][j]
            dico[caractere] = (i + 1, j + 1)
    return dico
```

Si on s'interdit de parcourir la grille, on parcourt l'ordre d'insertion :

```
def generer_dico(cle):
    ordre_insertion = generer_ordre(cle)
```

```

dico = {}
i = 1
j = 1
for caractere in ordre_insertion:
    dico[caractere] = (i, j)
    j += 1
    if j == 7:
        i += 1
        j = 1
return dico

```

9. `def chiffrer(cle, message):`  
`dico = generer_dico(cle)`  
`resultat = []`  
`for caractere in message:`  
 `resultat.append(dico[caractere])`  
`return resultat`  
*# ou, avec une liste en compréhension*  
*# return [dico[caractere] for caractere in message]*

10. Dans un chiffrement symétrique la même clé est utilisée pour chiffrer et déchiffrer.

Dans un chiffrement asymétrique, ces opérations nécessitent deux clés distinctes.

### Exercice 3 : 8 points - POO, récursivité et SQL

#### Partie A : La classe Demineur

1. On fait `assert 0.1 <= pourcentage_mines <= 0.3, 'Le pourcentage de mines ...'`.

2. `class Demineur :`  
 `def __init__(self, hauteur, largeur, pourcentage_mines):`  
 `assert 0.1 <= pourcentage_mines <= 0.3, 'Le pourcentage de mines ...'`  
 `self.hauteur = hauteur`  
 `self.largeur = largeur`  
 `self.pourcentage_mines = pourcentage_mines`

3. On saisit `demineur_intermediaire = Demineur(16, 16, 0.156)`.

#### Partie B : Création de la grille du démineur

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

5. `while compteur_mines < nombre_bombes:`  
 `ligne = randint(0, self.hauteur - 1)`  
 `colonne = randint(0, self.largeur - 1)`  
 `if self.grille_demineur[ligne][colonne] == 0:`  
 `self.grille_demineur[ligne][colonne] = -1`  
 `compteur_mines = compteur_mines + 1`

```

6. def nombre_voisines_avec_mines(self, coordonnees_case):
    total = 0
    for (ligne, colonne) in self.voisines(coordonnees_case):
        if self.grille_demineur[ligne][colonne] == -1:
            total += 1
    return total
    # ou, avec la fonction sum :
    # return sum(self.grille[i][j] == -1 for (i, j) in
    ↪ self.voisines(coordonnees_case))

```

7. On considère que la méthode `placer_mine` a déjà été appelée.

```

def generer_demineur(self):
    for ligne in range(self.hauteur):
        for colonne in range(self.largeur):
            if self.grille_demineur[ligne][colonne] != -1:
                self.grille_demineur[ligne][colonne] =
                ↪ self.nombre_voisines_avec_mines((ligne, colonne))

```

### Partie C : L'interface utilisateur du jeu du démineur

```

8. def visible(self, coords):
    ligne, colonne = coords
    if self.grille_demineur[ligne][colonne] == -1:
        self.grille_visibilite = [
            [True for _ in range(self.largeur)] for _ in range(self.hauteur)
        ]
    else:
        self.grille_visibilite[ligne][colonne] = True
        if self.grille_demineur[ligne][colonne] == 0:
            for i, j in self.voisines(coords):
                if not self.grille_visibilite[i][j]:
                    self.visible((i, j))

```

### Partie D : Jouer en ligne au démineur

9. Les clés étrangères de la table `Meilleur_score` sont :

- joueur qui fait référence à `Joueur.id_joueur` ;
- niveau qui fait référence à `Demineur.niveau`.

10. Comme suggéré par l'énoncé, il existe plusieurs réponses. On propose :

```

SELECT niveau, score
FROM Meilleur_score
WHERE joueur = 2;

```

```

11. UPDATE Joueur
SET mot_de_passe = cGhxDE4
WHERE pseudo = 'Kirna';

```

12. Cette requête permet d'obtenir les pseudos des joueurs ayant réalisé un score strictement supérieur à 1000 sur des grilles `'expert'` en strictement moins de 400 secondes. On obtient donc :

pseudo

Raptor

13. `UPDATE` Demineur  
`SET` niveau = 'débutant'  
`WHERE` niveau = 'facile';

**Attention**, cette action peut potentiellement poser problème car l'attribut `Demineur.niveau` est utilisé comme clé étrangère dans la table `Meilleur_score`. On fait l'hypothèse (hors-programme) que la base de données est paramétrée pour répercuter les mises à jour des clés étrangères vers les autres tables (avec `ON UPDATE CASCADE` par exemple).

Si on ignore cette possibilité de mise à jour en cascade, il faut alors :

- supprimer les entrées de `Meilleur_resultat` faisant référence à des grilles 'facile' ;
- mettre à jour la table `Demineur` avec la nouvelle valeur (en utilisant la requête proposée plus haut) ;
- ré-insérer les valeurs supprimées en leur donnant la nouvelle valeur `niveau = 'débutant'`.