

Exercice 1

Partie A

1. Non, car un artiste peut enregistrer plusieurs albums et la clé primaire doit être unique pour chaque entrée

2.

Nightwish
The Rasmus

3.

1986
2001
1986

4.

```
UPDATE CD
SET Annee = 2000
WHERE Titre_album = 'Wishmaster'
```

5.

```
SELECT Titre_album
FROM CD
JOIN Rangement ON CD.id_album = Rangement.id_album
JOIN Artiste ON CD.Nom_artiste = Artiste.Nom_artiste
WHERE Style = 'Metal' AND Numero_etagere = 1
```

6.

Il faut d'abord supprimer les références à cet album dans la table Rangement (afin de ne pas avoir d'attribut qui pointe vers rien), puis, supprimer l'album dans la table CD, puis enfin dans Artiste

```
DELETE FROM CD
WHERE Titre_album = 'Dead Letters'
```

Partie B

7.

Alice et Bob partagent une clé de chiffrement K. Alice utilise K pour chiffrer un message M et obtient M' la version chiffrée de M. M' est transmis à Bob par l'intermédiaire d'un réseau (même si M' est intercepté, les données contenus dans M' seront illisibles), Bob utilise K pour déchiffrer M' et obtient M.

8.

Alice possède une clé publique Kpu et une clé privée Kpr. La clé privée est uniquement connue d'Alice et la clé publique est... publique ! Bob désire envoyer un message M à Alice. Pour chiffrer le message M, Bob va utiliser la clé publique d'Alice Kpu. Il obtient alors le message chiffré M'. Bob envoie M' à Alice. Alice récupère M' et déchiffre M' à l'aide de sa clé privée Kpr.

9.

Le chiffrement asymétrique peut être utilisé pour l'échange de C. Une fois C échangée, il est alors possible d'utiliser un chiffrement symétrique pour la suite des échanges.

Exercice 2

Partie A

1.

```
class Marchandise:
    def __init__(self, p, v):
        assert v > 0
        self.prix = p
        self.volume = v
```

2.

```
m1 = Marchandise(20, 7)
```

3.

```
def ratio(self):
    return self.prix / self.volume
```

4.

```
def prixListe(tab):
    total = 0
    for m in tab:
        total = total + m.prix
    return total
```

Partie B

5.

combinaisons	prix
m1	40
m2	210
m3	160
m4	50
m1 + m2	250
m1 + m3	200
m1 + m4	90
m3 + m4	210

La combinaison qui maximise le prix est donc m1 + m2

6. algorithme glouton

7.

```
def tri(tab: list) -> None:
    n = len(tab)
    for i in range(1, n):
        marchandise = tab[i]
        j = i-1
        while j >= 0 and marchandise.ratio() > tab[j].ratio() :
            tab[j+1] = tab[j]
            j = j - 1
        tab[j+1] = marchandise
```

8.

Il s'agit d'un tri par insertion, le coût est quadratique

9.

```
def charge(tab: list, volume: int) -> list:
    tri(tab)
    chargement = []
    n = len(tab)
    for i in range(n):
        if volume > 0 and tab[i].volume <= volume:
            chargement.append(tab[i])
            volume = volume - tab[i].volume
    return chargement
```

Partie C

10.

```
def chargeOptimale(tab: list, v_restant: int, i: int) -> list:
    if i >= len(tab):
        return []
    else:
        if tab[i].volume > v_restant:
            return chargeOptimale(tab, v_restant, i+1)
        else:
            option1 = chargeOptimale(tab, v_restant, i+1)
            option2 = [tab[i]] + chargeOptimale(tab, v_restant -
            tab[i].volume, i+1)
            if prixListe(option1) > prixListe(option2):
                return option1
            else:
                return option2
```

Exercice 3

Partie A

1.
self.nom de type str
self.denivele de type int
self.longueur de type int
self.couleur de type str
self.ouverte de type booléen
2.

```
def set_couleur(self):  
    if self.denivele >= 100:  
        self.couleur = 'noire'  
    if self.denivele < 100 and self.denivele >= 70:  
        self.couleur = 'rouge'  
    if self.denivele < 70 and self.denivele >= 40:  
        self.couleur = 'bleue'  
    if self.denivele < 40:  
        self.couleur = 'verte'
```
3. Proposition D
4.

```
for piste in lievre_blanc.get_pistes():  
    if piste.get_couleur() == 'verte':  
        piste.ouverte = False
```
5.

```
def pistes_de_couleur(lst, couleur):  
    tab = []  
    for p in lst:  
        if p.get_couleur() == couleur:  
            tab.append(p.get_nom())  
    return tab
```
6.
ligne 2: distance = 0
ligne 6: if piste.get_nom() == nom:
ligne 7: distance = distance + piste.get_longueur()
ligne 8: return distance > 21.1

Partie B

7.
domaine['E']['F']

8.

```
def voisins(G,s):  
    tab = []  
    for k in G[s]:  
        tab.append(k)  
    return tab
```

9.

```
ligne 2: precedent = chemin[0]  
ligne 5: longueur = longueur + G[precedent][chemin[i]]  
ligne 6: precedent = chemin[i]  
ligne 7: return longueur
```

10.

La fonction est récursive, car elle s'appelle elle-même

11.

```
def parcours_dep_arr(G, depart, arrivee):  
    liste = parcours(G, depart)  
    res = []  
    for l in liste :  
        n = len(l)  
        if l[n-1] == arrivee and l not in res:  
            res.append(l)  
    return res
```

12.

```
def plus_court(G, depart, arrivee):  
    liste_chemins = parcours_dep_arr(G, depart, arrivee)  
    chemin_plus_court = liste_chemins[0]  
    minimum = longueur_chemin(G, chemin_plus_court)  
    for chemin in liste_chemins:  
        longueur = longueur_chemin(G, chemin)  
        if longueur < minimum :  
            minimum = longueur  
            chemin_plus_court = chemin  
    return chemin_plus_court
```

13.

Le chemin le plus court n'est pas forcément le plus rapide car il faut aussi tenir compte du dénivelé. Un chemin court avec un fort dénivelé positif sera moins intéressant qu'un chemin plus court avec un dénivelé positif proche de zéro.

Proposition de correction

Exercice 1

Partie A

Q1

- attribut : itinéraire
- méthode : remplir_grille()

Q2

- a = 4
- b = 7

Q3

```
def remplir_grille(self):  
    i, j = 0, 0 # Position initiale  
    self.grille[j][i] = 'S' # Case de départ marquée d'un S  
    for direction in self.itineraire:  
        if direction == 'D':  
            i = i + 1 # Déplacement vers la droite  
        elif direction == 'B':  
            j = j + 1 # Déplacement vers le bas  
        self.grille[j][i] = '*' # Marquer le chemin avec '*'  
    self.grille[j][i] = 'E' # Case d'arrivée marquée d'un E
```

Q4

```
def get_dimensions(self) -> tuple:  
    """  
    @return longueur et largeur de l'itinéraire  
    """  
    return self.longueur, self.largeur
```

Q5

```
def remplir_grille(self):  
    i, j = 0, 0 # Position initiale  
    self.grille[j][i] = 'S' # Case de départ marquée d'un S  
    for direction in self.itineraire:  
        if direction == 'D':  
            i = i + 1 # Déplacement vers la droite  
        elif direction == 'B':  
            j = j + 1 # Déplacement vers le bas  
        self.grille[j][i] = '*' # Marquer le chemin avec '*'  
    self.grille[j][i] = 'E' # Case d'arrivée marquée d'un E
```

Partie B

Q6

```
def itineraire_aleatoire(m : int, n : int) -> str:
    """
    @param m -- longueur de l'itinéraire
    @param n -- largeur de l'itinéraire
    @return l'itinéraire
    """
    itineraire = ""
    i, j = 0, 0
    while i != m and j != n:
        deplacement = choice(['D', 'B'])
        itineraire += deplacement
        if deplacement == 'D':
            j += 1
        else:
            i += 1
    if i == m:
        itineraire = itineraire + 'D'*(n-j)
    if j == n:
        itineraire = itineraire + 'B'*(m-i)
    return itineraire
```

Partie C

Q7

Pour une grille de 1 dimension, il n'y a qu'un seul chemin possible : aller toujours à droite ('D') ou toujours vers le bas ('B').

- exemple, pour 1x3 : "DDD".
- exemple, pour 3x1 : "BBB".

Q8

Chaque chemin vers la case (m, n) provient de (m-1, n) : un mouvement vers le bas 'B', ou de (m, n-1) : un mouvement vers la droite 'D'.

Donc, le nombre de chemins vers (m, n) est la somme du nombre de chemins vers (m-1, n) + du nombre de chemins vers (m, n-1).

Q9

```
def nombre_chemins(m : int, n : int) -> str:
    """
    @param m -- longueur de l'itinéraire
    @param n -- largeur de l'itinéraire
    @return nombre de chemins possibles
    """
    if m == 1 or n == 1:
        return 1
    return nombre_chemins(m-1, n) + nombre_chemins(m, n-1)
```