#### BACCALAURÉAT GÉNÉRAL

##### ÉPREUVE D’ENSEIGNEMENT DE SPÉCIALITÉ

###### **SESSION 2025**

#### **NUMÉRIQUE ET SCIENCES INFORMATIQUES**

###### **JOUR 2**

##### 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 14 pages numérotées de 1/14 à 14/14.

#####

**Le sujet est composé de trois exercices indépendants.**

**Le candidat traite les trois exercices.**

#####

# EXERCICE 1 (6 points)

*Cet exercice porte sur la programmation orientée objet, l’algorithmique et la programmation en Python.*

Marc souhaite coder le jeu de cartes “6 qui prend” afin d’y jouer contre l’ordinateur ou en ligne avec ses amis. Il va réaliser le code en Python en utilisant la programmation orientée objet, sans se préoccuper de l’interface graphique.

Ce jeu est composé de 104 cartes numérotées de 1 à 104. Chacune possède une ou plusieurs “têtes de bœuf” (notée TdB) qui représentent des points de pénalité. Le but est d’avoir le moins de “têtes de bœuf” possible à la fin de la partie.

## Partie A : La classe Carte

On commence par créer la classe Carte pour modéliser les 104 cartes.

Le constructeur de cette classe prend en paramètre une valeur (de type int) comprise entre 1 et 104 et l’affecte à l’attribut valeur (de type int).

On ajoute l’attribut TdB (de type int) qui contient le nombre de “têtes de bœuf” qui sera calculé à l’aide de la méthode de classe calcul\_TdB (définie à la question suivante) qui n’a pas de paramètre.

1. Écrire le code du constructeur de la classe Carte.

Chaque carte possède une ou plusieurs “têtes de bœuf”.
Ce nombre est déterminé de la façon suivante :

* si la valeur de la carte est divisible par 11 alors la carte reçoit 5 “têtes de bœuf”;
* si la valeur de la carte se termine par 0 alors la carte reçoit 3 “têtes de bœuf”;
* si la valeur de la carte se termine par 5 alors la carte reçoit 2 “têtes de bœuf”;
* si la valeur de la carte ne remplit aucune de ces conditions alors elle ne reçoit qu’1 “tête de bœuf”.

Attention, ces règles se cumulent.

Par exemple, 55 est divisible par 11 et se termine par 5 donc la carte 55 comporte 7 “têtes de bœuf”.

1. Écrire le code de la méthode calcul\_TdB de la classe Carte afin de calculer le nombre de “têtes de bœuf” pour une carte donnée.

On veut ajouter à la classe Carte une méthode est\_superieure\_a qui prend en paramètre une carte autre (de type Carte) et qui renvoie True si la valeur de la carte considérée est strictement supérieure à la valeur de la carte autre.

1. Écrire le code de cette méthode.

La classe Carte possède également la méthode difference qui prend en paramètre une carte autre (de type Carte) et qui renvoie la valeur absolue de la différence entre les valeurs des deux cartes considérées.

## Partie B : La classe Paquet

Maintenant que les cartes sont modélisées, on crée une classe Paquet pour gérer les différents paquets de cartes durant ce jeu.

Le constructeur de la classe Paquet prend en paramètre une liste (de type list) contenant des cartes (de type Carte) que l’on affecte à l’attribut contenu.

On ajoute deux méthodes à cette classe Paquet :

* la méthode afficher permet d’afficher la valeur de toutes les cartes du paquet,
* la méthode ajouter\_carte prend en paramètre une carte (de type Carte) et l’ajoute au contenu du paquet considéré.
1. Compléter le code suivant en recopiant les lignes 6, 7 et 10 sur votre copie :

1 class Paquet:
2 def \_\_init\_\_(self, L):
3 self.contenu = L
4
5 def afficher(self):
6 ...
7 ...
8
9 def ajouter\_carte(self, carte):
10 ...

1. Écrire le code de la méthode nombre\_TdB de la classe Paquet qui renvoie le nombre total de “têtes de bœuf” contenus sur les cartes de ce paquet.

La classe Paquet contient aussi une méthode distribuer qui prend en paramètre un entier naturel nbr correspondant au nombre de joueurs (au maximum 10) à servir et qui renvoie la liste contenant les nbr paquets.

Cette liste est initialisée avec nbr instances de classe Paquet dont le contenu est vide.

Chaque joueur reçoit 10 cartes. La première carte du paquet est donnée au premier joueur, la deuxième au deuxième joueur et ainsi de suite.

Ainsi chacun des nbr paquets contiendra 10 cartes prises du paquet sur lequel on appelle la méthode (donc enlevées de ce paquet initial).

1. Écrire le code de la méthode distribuer de la classe Paquet.

La classe Paquet possède également les méthodes suivantes :

* la méthode prendre\_carte n’a pas de paramètre et renvoie la carte choisie par l’utilisateur dont il saisit la valeur lors de l’appel de cette méthode (on demande une valeur tant que celle-ci n’est pas valable, ainsi la carte est obligatoirement une carte du paquet).

Cette carte est supprimée du paquet ;

* la méthode trier n’a pas de paramètre et permet de trier les cartes du paquet dans l’ordre croissant de leur valeur.

## Partie C : La classe Joueur et modélisation du plateau de jeu -> classe Plateau

Il reste à gérer les joueurs : leur nom, leur paquet de cartes, etc.
C’est le rôle de la classe Joueur.

Le constructeur de la classe Joueur prend en paramètres :

* le nom du joueur (de type str) que l’on affecte à l’attribut nom ;
* un paquet de cartes (de type Paquet) que l’on affecte à l’attribut main.

On ajoute également les attributs cartes\_ramassees qui est initialisé avec un paquet de cartes vide et penalite initialisé à 0.

1 class Joueur():
2 def \_\_init\_\_(self, nom, main):
3 self.nom = nom
4 self.main = main
5 self.cartes\_ramassees = Paquet([])
6 self.penalite = 0

1. Écrire une commande Python permettant d’instancier le joueur nommé Joueur 1 ayant pour paquet de cartes L[0] et qui affecte l’objet créé à la variable J1.

La classe Joueur possède également la méthode ramasser\_paquet qui prend en paramètre un paquet (de type Paquet), l’ajoute à l’attribut cartes\_ramassees, et met à jour l’attribut penalite en lui ajoutant le nombre total de “têtes de bœuf” contenus dans le paquet ramassé.

Maintenant que les trois classes sont prêtes, on initialise le jeu pour deux joueurs dans le programme principal.

1. Compléter le script ci-dessous en recopiant les lignes 3, 7 et 9 sur votre copie :

1 from random import \*
2 # créer les 104 cartes du jeu initial grâce à une liste par compréhension
3 jeu = [... for i in range (..., ...)]
4 # mélanger cette liste de cartes
5 shuffle(jeu)
6 # instancier le paquet de cartes avec cette liste de cartes
7 jeu\_initial = ...
8 # distribuer 10 cartes aux deux joueurs que l'on intancie en les nommant `J1` et `Ordi`.
9 distri = ...
10 Ordi = Joueur("Ordi", distri[0])
11 J1 = Joueur("J1", distri[1])

# EXERCICE 2 (6 points)

*Cet exercice porte sur la programmation Python, la programmation orientée objet, les bases de données relationnelles et les requêtes SQL.*

*Cet exercice est composé de deux parties indépendantes.*

## Partie A

Dans cette partie, nous allons utiliser une base de données sur les champignons. Le tableau ci-dessous nous donne un extrait du résultat obtenu, après avoir effectué la requête suivante :

SELECT \* FROM champignon;



Les attributs nom, lamelle et couleur sont des chaines de caractères. La taille du chapeau et la longueur du pied des champignons sont des nombres exprimés en centimètres.

1. Mathilde vient de trouver un champignon avec des lamelles et de couleur orange. Écrire la requête qu’elle doit écrire pour avoir la liste des noms des champignons possibles pour sa trouvaille.
2. Par ailleurs, Romain a trouvé un champignon sans pied et un chapeau de 15 cm. Écrire la requête qu’il doit écrire pour avoir la liste des noms des champignons possibles pour son spécimen.

Les champignons sont classés selon différents ordres selon la table ordre suivante où l’on trouve également la classe à laquelle appartient chaque ordre. Voici le contenu de la table ordre.

| table ordre |
| --- |
| id | nom | classe |
| 1 | agaricales | agaricomycètes |
| 2 | trémellales | phragmabasidiomycètes |
| 3 | bolétales | agaricomycètes |
| 4 | amanitales | agaricomycètes |
| 5 | cantharellales | agaricomycètes |
| 6 | polyporales | basidiomycètes |
| 7 | clavariales | homobasidiomycètes |
| 8 | tricholomatales | agaricomycètes |

1. Donner la clé étrangère de la table champignon faisant référence à la clé primaire id de la table ordre.
2. Écrire la requête permettant d’obtenir la liste des noms des champignons appartenant à la classe des agaricomycètes.

Le champignon nommé amanite solitaire est un champignon blanc qui appartient à l’ordre des amanitales. Il possède des lamelles. Son chapeau mesure entre 6 et 20 cm et son pied entre 4 et 10 cm.

1. Écrire une requête permettant d’ajouter ce champignon avec l’id 56 dans la table champignon.

On décide d’ajouter les différentes toxicités des champignons en ajoutant un attribut id\_toxicite dans la relation champignon et une table toxicite. Voici le contenu de la table toxicite :

| id\_tox | type | effets |
| --- | --- | --- |
| 1 | très toxique | Entrainant la mort |
| 2 | toxique | Entrainant des problèmes digestifs ou nerveux |
| 3 | à rejeter | Champignons suspects ou ayant un mauvais gout |
| 4 | comestible | Champignons pouvant être consommés |

1. Écrire le schéma relationnel de la base de données contenant ces trois tables en soulignant les clés primaires et en faisant précéder les clés étrangères du caractère #.

Une erreur s’est glissée dans cette table : le champignon de nom amanite citrine est très toxique car son ingestion peut entrainer la mort.

1. Écrire la requête permettant de corriger cette erreur dans la table champignon.
2. Écrire une requête permettant d’obtenir les noms des champignons de l’ordre des amanitales de toxicité très toxique.

## Partie B

Quentin fait quelques recherches afin de trouver les recettes possibles avec des champignons selon les saisons. Il trouve notamment les informations suivantes pour le lactaire délicieux :

Lactaire délicieux :

* localisation : sous les pins
* saison : été
* recette : lactaires grillés à l’huile d’olive
* cuisson : 12 minutes à feu moyen

Pour chaque champignon, il souhaite stocker les informations dans un objet de la classe Champignon définie ci-dessous :

1 class Champignon():
2 def \_\_init\_\_(self, nom, lieu, saison, recette, cuisson):
3 self.nom = nom
4 self.localisation = lieu
5 self.saison = saison
6 self.recette = recette
7 self.cuisson = cuisson

Donc l’instruction permettant d’instancier l’objet champignon1 du champignon lactaire délicieux est la suivante :

champignon1 = Champignon('Lactaire délicieux', 'Sous les pins', 'été', 'Lactaires grillés à l\'huile d\'olive', '12 minutes à feu moyen')

Quentin décide de stocker l’ensemble des instances de classe Champignon dans une variable liste\_champi de type list.

1. Recopier et compléter le programme ci-dessous permettant d’afficher le(s) nom(s) des champignons que l’on trouve lors de la saison été.

for e in liste\_champi:
 if ...... == 'été':
 print(...)

La plaque de cuisson de Quentin est défectueuse. Il ne peut cuire des aliments qu’à feu moyen. Il écrit donc rapidement le programme suivant afin de savoir s’il peut cuire ses champignons à feu moyen.

1 for c in liste\_champi:
2 if c.nom == 'Lactaire délicieux':
3 return c.cuisson == 'feu moyen'

Ce programme renvoie False, alors que le lactaire délicieux se cuisine bien à feu moyen.

1. Expliquer pourquoi il n’obtient pas le résultat attendu alors que Quentin n’a fait aucune erreur dans sa liste de champignons.

Malgré ce problème, Quentin ne souhaite pas modifier son travail et téléphone à son ami Iris qui lui fournit une fonction recherche\_textuelle qui prend en paramètre deux chaines de caractères texte et mot et qui retourne True si mot est un mot appartenant à texte et False sinon.

>>> recherche\_textuelle('dans une forêt', 'forêt')
True
>>> recherche\_textuelle('près de la mer', 'mare')
False

1. Corriger le programme de la question précédente en utilisant la fonction recherche\_textuelle.

# Exercice 3 (8 points)

*Dans cet exercice il est question de tableaux, de dictionnaires, de recherche de chemins dans un graphe, de piles, de files et de POO.*

On considère une liste de mots de quatre lettres. Par exemple *gars*, *grue*, *mars*, *mors*, *ours*, *purs*, *durs* …

Deux mots sont voisins s’ils ne diffèrent que d’une seule lettre sans se soucier de l’ordre des lettres dans les mots. Par exemple :

* *mars* et *mors* sont voisins (on change le *a* en *o*);
* *mors* et *ours* sont voisins (on change le *m* en *u*);
* *grue* et *ours* ne sont pas voisins (il faudrait changer *g* en *o* **et** *e* en *s*).

L’exercice consiste à partir d’un mot de départ (par exemple *mars*) pour atteindre un mot de destination (par exemple *ours*) en passant de voisins en voisins et en empruntant le moins de voisins possible.

Les mots utilisés, du mot de départ au mot de destination, forment alors le chemin emprunté. Par exemple *mars*, *mors*, *ours* est le chemin qui relie le mot *mars* au mot *ours*.

On considère qu’il est toujours possible de trouver un tel chemin.

## Partie A

Pour résoudre notre problème nous aurons besoin, entre autres, d’une structure de pile, d’une structure de file et d’un graphe.

1. Décrire le fonctionnement d’une Pile.
2. Décrire le fonctionnement d’une File.

On va utiliser un graphe dont les sommets sont les mots et dont les arêtes relient deux sommets si les mots sont voisins.

1. Expliquer pourquoi un graphe non orienté est adapté à la situation.
2. Dessiner le graphe si la liste de mots est : *gars, mars, mors, ours* et *purs*.

## Partie B

La distance entre deux mots est le nombre minimum de lettres qu’il faut modifier pour passer de l’un à l’autre. Par exemple, la distance entre *mars* et *mors* est 1 (on change le *a* en *o*) tandis que la distance entre *grue* et *ours* est 2 (on change le *g* en *o* et le *e* en *s*).

Deux mots sont voisins si et seulement si la distance entre eux vaut 1.

Dans toute cette partie, on dispose d’une variable globale *TAB\_MOTS*, un tableau (type *list* en Python) dont les éléments sont des chaines de quatre caractères qui correspondent à des mots.

1. Recopier et compléter la fonction chaine\_vers\_tab(mot) ci-dessous. Cette fonction prend en paramètre une chaine de quatre caractères et renvoie un tableau (type *list* en Python) dont les éléments sont les caractères de cette chaine.
* Ainsi l’appel de la fonction chaine\_vers\_tab('ours') renvoie ['o', 'u', 'r', 's'].
* 1. def chaine\_vers\_tab(mot):
2. tab\_lettres = ...
3. for ... in ... :
4. tab\_lettres....
5. return tab\_lettres
1. Expliquer pourquoi la fonction ci-dessous renvoie effectivement la distance entre les deux mots *mot1* et *mot2*, deux chaines de quatre caractères.
* 1. def distance(mot1, mot2):
2. tab = chaine\_vers\_tab(mot1)
3. for lettre in mot2:
4. if lettre in tab:
5. tab.remove(lettre)
6. return len(tab)
1. Recopier et compléter la fonction renvoie\_voisins(mot) qui renvoie un tableau dont les éléments sont les mots de *TAB\_MOTS* qui sont voisins de *mot*, une chaine de quatre caractères, passé en paramètre.
* 1. def renvoie\_voisins(mot):
2. tab\_voisins = ...
3. for voisin\_possible in ... :
4. if ... :
5. tab\_voisins....
6. return ...

## Partie C

Il nous faut maintenant chercher le chemin le plus court entre deux mots.

On dispose pour cela d’une classe File et d’une classe Pile.

Voici les méthodes de la classe File dont nous aurons besoin :

* est\_vide(self) qui renvoie *True* si l’instance de File est vide et *False* sinon ;
* defiler(self) qui, si l’instance de File n’est pas vide, lui enlève la tête et la renvoie ;
* enfiler(self, element) qui enfile *element* dans l’instance de File.

Voici les méthodes de la classe Pile dont nous aurons besoin :

* est\_vide(self) qui renvoie *True* si l’instance de Pile est vide et *False* sinon;
* depiler(self) qui, si l’instance de Pile n’est pas vide, lui enlève le sommet et le renvoie.
* empiler(self, element) qui empile *element* dans l’instance de Pile.

La fonction dic\_parent(mot\_depart, mot\_final) ci-dessous renvoie le chemin entre mot\_depart, qui est le mot de départ, et mot\_final , qui est celui qu’on cherche à atteindre, sous la forme d’un dictionnaire {sommet parcouru : sommet précédent} :

* 1. def dic\_parent(mot\_depart, mot\_final):
2. file\_voisins = File()
3. parent = {mot\_depart : None}
4. mot = mot\_depart
5. file\_voisins.enfiler(mot)
6. while not file\_voisins.est\_vide() and not mot == mot\_final:
7. mot = file\_voisins.defiler()
8. for voisin in renvoie\_voisins(mot):
9. if not voisin in parent:
10 parent[voisin] = mot
11. file\_voisins.enfiler(voisin)
12. return parent

On donne les résultats ci-dessous qui correspondent au graphe de la partie A :

* >>> renvoie\_voisins('mars')
['gars', 'mors']
>>> renvoie\_voisins('gars')
['mars']
>>> renvoie\_voisins('mors')
['mars', 'ours']
>>> renvoie\_voisins('ours')
['mors', 'purs']

Voici le début de l’exécution pas à pas de la fonction dic\_parent en prenant *‘mars’* pour mot de départ et *‘ours’* pour mot final :

* + Avant le début de la boucle :
		- *parent* = {‘mars’ : *None*}
		- *file\_voisins* contient uniquement *‘mars’*
	+ Premier tour de boucle :
		- on défile *‘mars’*
		- les voisins de *‘mars’* sont *‘gars’* et *‘mors’* . Ils ne sont pas encore présents dans le dictionnaire *parent* donc ils ont tous les deux pour parent *‘mars’* et on les enfile dans cet ordre dans *file\_voisins*. Ainsi on obtient :
		- *parent* = {*’mars’* : *None*, *‘gars’*: *’mars’*, *‘mors’*: *‘mars’*}
		- file\_voisins contient, de la tête à la queue, *‘gars’* puis *‘mors’*.
	+ Deuxième tour de boucle :
		- * on défile *‘gars’*
			* le seul voisin de \_‘gars\_’ est ‘*mars*’. ‘*mars*’ est déjà dans *parent*. Ainsi on obtient :
			* *parent* = {*’mars’* : *None*, *‘gars’*: *’mars’*, *‘mors’*: *‘mars’*}
			* *file\_voisins* contient uniquement *‘mors’*
1. Poursuivre le déroulement de la fonction pas à pas sur le modèle ci-dessus en détaillant chaque tour de boucle jusqu’à l’issue de la fonction.

Dans cette question on souhaite construire une instance de Pile dans laquelle on va empiler chaque mot du chemin en remontant les mots, depuis le mot final jusqu’au mot de départ, grâce au dictionnaire renvoyé par la fonction dic\_parent.

1. Recopier et compléter la fonction renvoie\_pile(parent, mot\_final).

Cette fonction prend en paramètres :

* + - parent, un dictionnaire obtenu grâce à la fonction dic\_parent ;
		- mot\_final, le mot final.

Elle renvoie une pile dont le premier élément empilé est le mot final et dont le sommet est le mot de départ.

* + 1. def renvoie\_pile(parent, mot\_final):
	2. ma\_pile = Pile()
	3. mot = mot\_final
	4. while mot != ... :
	5. ma\_pile....
	6. mot = ...
	7. return ma\_pile
1. Recopier et compléter la fonction construit\_chemin :
	* + son paramètre est une pile de mots obtenue grâce à la fonction renvoie\_pile ;
		+ elle renvoie un tableau dont les éléments sont les mots du chemin recherché dans le bon ordre.
	* 1. def construit\_chemin(ma\_pile):
	2. tab = ...
	3. while ...:
	4. mot = ...
	5. tab....
	6. return tab
2. Coder, en utilisant les fonctions précédentes, la fonction chercher\_chemin de paramètres *mot\_depart*, le mot de départ, et *mot\_final*, le mot à atteindre, et qui renvoie un tableau dont les éléments sont les mots qui constituent le chemin du mot de départ jusqu’au mot final.