

BAC BLANC 2022 - SUJET 1
mercredi 12/01/2022

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

JANVIER 2022

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Durée de l'épreuve : 3 heures 30

L'usage de la calculatrice et du dictionnaire n'est pas autorisé.

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

Ce sujet comporte 13 pages numérotées de 1/13 à 13 /13.

**Le candidat traite au choix 3 exercices
parmi les 5 exercices proposés**

Chaque exercice est noté sur 4 points.

EXERCICE %(4 points)

Principaux thèmes abordés : structure de données (tableaux, dictionnaires) et langages et programmation (spécification).

Objectif de l'exercice :

Les Aventuriers du Rail© est un jeu de société dans lequel les joueurs doivent construire des lignes de chemin de fer entre différentes villes d'un pays.

La carte des liaisons possibles dans la région Occitanie est donnée en **annexe 1 de l'exercice 1**. Dans l'**annexe 2 de l'exercice 1**, les liaisons possédées par le joueur 1 sont en noir, et celles du joueur 2 en blanc. Les liaisons en gris sont encore en jeu.

Codages des structures de données utilisées :

- **Liste des liaisons d'un joueur** : Toutes les liaisons directes (sans ville intermédiaire) construites par un joueur seront enregistrées dans une variable de type "**tableau de tableaux**".

Le joueur 1 possède les lignes directes "Toulouse-Muret", "Toulouse-Montauban", "Gaillac-St Sulpice" et "Muret-Pamiers" (liaisons indiquées en noir dans l'**annexe 2 de l'exercice 2**). Ces liaisons sont mémorisées dans la variable ci-contre.

```
liaisonsJoueur1 = [  
  ["Toulouse", "Muret"],  
  ["Toulouse", "Montauban"],  
  ["Gaillac", "St Sulpice"],  
  ["Muret", "Pamiers"]  
]
```

Remarque : Seules les liaisons directes existent, par exemple ["Toulouse", "Muret"] ou ["Muret", "Toulouse"]. Par contre, le tableau ["Toulouse", "Mazamet"] n'existe pas, puisque la ligne Toulouse-Mazamet passe par Castres.

- **Dictionnaire associé à un joueur** : On code la liste des villes et des trajets possédée par un joueur en utilisant un **dictionnaire de tableaux**. Chaque **clef de ce dictionnaire** est une ville de départ, et chaque **valeur** est un **tableau** contenant les villes d'arrivée possibles en fonction des liaisons possédées par le joueur.

Le **dictionnaire de tableaux** du joueur 1 est donné ci-contre :

```
DictJoueur1 = {  
  "Toulouse": ["Muret", "Montauban"],  
  "Montauban": ["Toulouse"],  
  "Gaillac": ["St Sulpice"],  
  "St Sulpice": ["Gaillac"],  
  "Muret": ["Toulouse", "Pamiers"],  
  "Pamiers": ["Muret"]  
}
```

1. Expliquer pourquoi la liste des liaisons suivante n'est pas valide :

```
tableauliaisons = [ ["Toulouse", "Auch"], ["Luchon", "Muret"],  
  ["Quillan", "Limoux"] ]
```

2. Cette question concerne le joueur n°2 (*Rappel : les liaisons possédées par le joueur n°2 sont représentées par un rectangle blanc dans l'annexe 2 de l'exercice 1*).

a) Donner le tableau `liaisonsJoueur2`, des liaisons possédées par le joueur n°2.

b) Recopier et compléter le dictionnaire suivant, associé au joueur n°2 :

```
DictJoueur2 = {  
    "Toulouse":["Castres", "Castelnaudary"],  
    ...  
}
```

3. À partir du tableau de tableaux contenant les liaisons d'un joueur, on souhaite construire le dictionnaire correspondant au joueur. Une première proposition a abouti à la fonction `construireDict` ci-dessous.

```
1  def construireDict(listeLiaisons):  
2      """  
3      listeLiaisons est un tableau de tableaux représentant la  
4      liste des liaisons d'un joueur comme décrit dans le problème  
5      """  
6      Dict={}  
7      for liaison in listeLiaisons :  
8          villeA = liaison[0]  
9          villeB = liaison[1]  
10         if not villeA in Dict.keys() :  
11             Dict[villeA]=[villeB]  
12         else :  
13             destinationsA = Dict[villeA]  
14             if not villeB in destinationsA :  
15                 destinationsA.append(villeB)  
16     return Dict
```

a) Écrire sur votre copie un assert dans la fonction `construireDict` qui permet de vérifier que la `listeLiaisons` n'est pas vide.

b) Sur votre copie, donner le résultat de cette fonction ayant comme argument la variable `liaisonsJoueur1` donnée dans l'énoncé et expliquer en quoi cette fonction ne répond que partiellement à la demande.

c) La fonction `construireDict`, définie ci-dessus, est donc partiellement inexacte.

Compléter la fonction `construireDict` pour qu'elle génère bien l'ensemble du dictionnaire de tableaux correspondant à la liste de liaisons données en argument. À l'aide des numéros de lignes, on précisera où est inséré ce code.

Exercice 2 (4 points)

Cet exercice traite des notions de piles et de programmation orientée objet.

On crée une classe `Pile` qui modélise la structure d'une pile d'entiers.

Le constructeur de la classe initialise une pile vide.

La définition de cette classe sans l'implémentation de ses méthodes est donnée ci-dessous.

```
class Pile:
    def __init__(self):
        """Initialise la pile comme une pile vide."""

    def est_vide(self):
        """Renvoie True si la liste est vide, False sinon."""

    def empiler(self, e):
        """Ajoute l'élément e sur le sommet de la pile,
        ne renvoie rien."""

    def depiler(self):
        """Retire l'élément au sommet de la pile et le renvoie."""

    def nb_elements(self):
        """Renvoie le nombre d'éléments de la pile. """

    def afficher(self):
        """Affiche de gauche à droite les éléments de la pile, du fond
        de la pile vers son sommet. Le sommet est alors l'élément
        affiché le plus à droite. Les éléments sont séparés par une
        virgule. Si la pile est vide la méthode affiche « pile
        vide »."""
```

Seules les méthodes de la classe ci-dessus doivent être utilisées pour manipuler les objets `Pile`.

1.

- a. Écrire une suite d'instructions permettant de créer une instance de la classe `Pile` affectée à une variable `pile1` contenant les éléments 7, 5 et 2 insérés dans cet ordre.

Ainsi, à l'issue de ces instructions, l'instruction `pile1.afficher()` produit l'affichage : 7, 5, 2.

- b. Donner l'affichage produit après l'exécution des instructions suivantes.

```
element1 = pile1.depiler()
pile1.empiler(5)
pile1.empiler(element1)
pile1.afficher()
```

2. On donne la fonction mystere suivante :

```
def mystere(pile, element):
    pile2 = Pile()
    nb_elements = pile.nb_elements()
    for i in range(nb_elements):
        elem = pile.depiler()
        pile2.empiler(elem)
        if elem == element:
            return pile2
    return pile2
```

a. Dans chacun des quatre cas suivants, quel est l'affichage obtenu dans la console ?

- Cas n°1

```
>>>pile.afficher()
7, 5, 2, 3
>>>mystere(pile, 2).afficher()
```
- Cas n°2

```
>>>pile.afficher()
7, 5, 2, 3
>>>mystere(pile, 9).afficher()
```
- Cas n°3

```
>>>pile.afficher()
7, 5, 2, 3
>>>mystere(pile, 3).afficher()
```
- Cas n°4

```
>>>pile.est_vide()
True
>>>mystere(pile, 3).afficher()
```

b. Expliquer ce que permet d'obtenir la fonction mystere.

3. Écrire une fonction etendre(pile1, pile2) qui prend en arguments deux objets Pile appelés pile1 et pile2 et qui modifie pile1 en lui ajoutant les éléments de pile2 rangés dans l'ordre inverse. Cette fonction ne renvoie rien.

On donne ci-dessous les résultats attendus pour certaines instructions.

```
>>>pile1.afficher()
7, 5, 2, 3
>>>pile2.afficher()
1, 3, 4
>>>etendre(pile1, pile2)
>>>pile1.afficher()
7, 5, 2, 3, 4, 3, 1
>>>pile2.est_vide()
True
```

4. Écrire une fonction `supprime_toutes_occurences(pile, element)` qui prend en arguments un objet `Pile` appelé `pile` et un élément `element` et supprime tous les éléments `element` de `pile`.

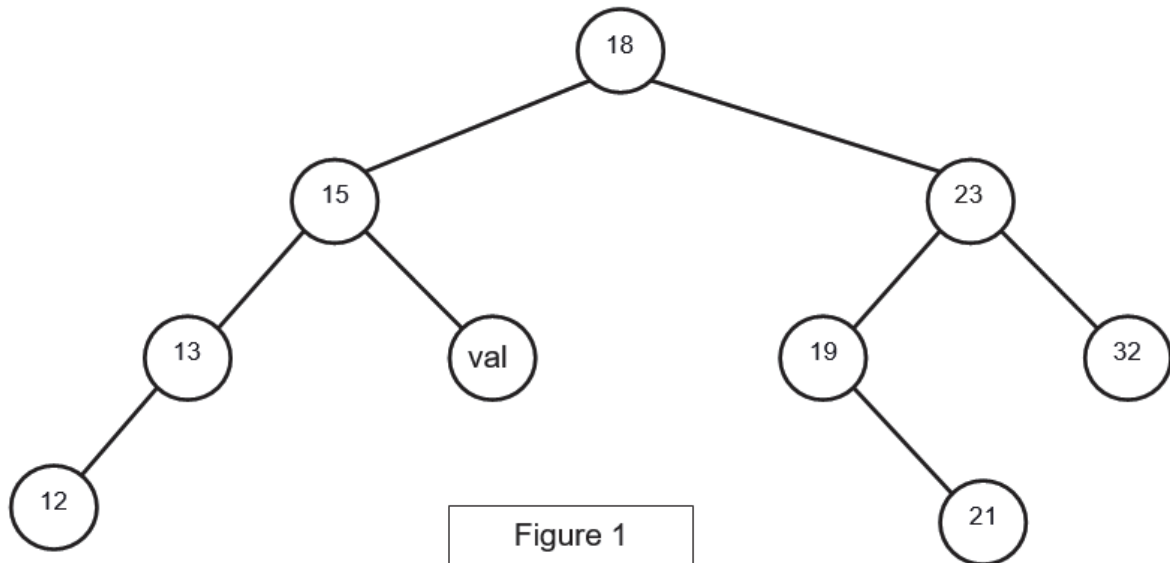
On donne ci-dessous les résultats attendus pour certaines instructions.

```
>>>pile.afficher()
7, 5, 2, 3, 5
>>>supprime_toutes_occurences (pile, 5)
>>>pile.afficher()
7, 2, 3
```

EXERCICE ' (4 points)

Cet exercice porte sur les arbres binaires de recherche.

Dans cet exercice, les arbres binaires de recherche ne peuvent pas comporter plusieurs fois la même clé. De plus, un arbre binaire de recherche limité à un nœud a une hauteur de 1. On considère l'arbre binaire de recherche représenté ci-dessous (figure 1), où **val** représente un entier :



1.
 - a. Donner le nombre de feuilles de cet arbre et préciser leur valeur (étiquette).
 - b. Donner le sous arbre-gauche du nœud 23.
 - c. Donner la hauteur et la taille de l'arbre.
 - d. Donner les valeurs entières possibles de val pour cet arbre binaire de recherche.

On suppose, pour la suite de cet exercice, que **val** est égal à 16.

2. On rappelle qu'un parcours infixe depuis un nœud consiste, dans l'ordre, à faire un parcours infixe sur le sous arbre-gauche, afficher le nœud puis faire un parcours infixe sur le sous-arbre droit.
Dans le cas d'un parcours suffixe, on fait un parcours suffixe sur le sous-arbre gauche puis un parcours suffixe sur le sous-arbre droit, avant d'afficher le nœud.
 - a. Donner les valeurs d'affichage des nœuds dans le cas du parcours infixe de l'arbre.
 - b. Donner les valeurs d'affichage des nœuds dans le cas du parcours suffixe de l'arbre.

3. On considère la classe `Noeud` définie de la façon suivante en Python :

```
class Noeud():
    def __init__(self, v):
        self.ag = None
        self.ad = None
        self.v = v

    def insere(self, v):
        n = self
        est_insere = False
        while not est_insere :
            if v == n.v:
                est_insere = True
            elif v < n.v:
                if n.ag != None:
                    n = n.ag
                else:
                    n.ag = Noeud(v)
                    est_insere = True
            else:
                if n.ad != None:
                    n = n.ad
                else:
                    n.ad = Noeud(v)
                    est_insere = True

    def insere_tout(self, vals):
        for v in vals:
            self.insere(v)
```

} Bloc 1

} Bloc 2

} Bloc 3

a. Représenter l'arbre construit suite à l'exécution de l'instruction suivante :

```
racine = Noeud(18)
racine.insere_tout([12, 13, 15, 16, 19, 21, 32, 23])
```

- b. Ecrire les deux instructions permettant de construire l'arbre de la figure 1. On rappelle que le nombre **val** est égal à 16.
- c. On considère l'arbre tel qu'il est présenté sur la figure 1. Déterminer l'ordre d'exécution des blocs (repérés de 1 à 3) suite à l'application de la méthode `insere(19)` au nœud racine de cet arbre.

4. Ecrire une méthode `recherche(self, v)` qui prend en argument un entier `v` et renvoie la valeur `True` si cet entier est une étiquette de l'arbre, `False` sinon.

Exercice 4 (4 points)

Cet exercice porte sur les bases de données.

Dans cet exercice, on pourra utiliser les mots clés suivants du langage SQL :

SELECT, FROM, WHERE, JOIN, ON, INSERT INTO, VALUES, MIN, MAX, OR, AND.

Les fonctions d'agrégation `MIN(propriete)` et `MAX(propriete)` renvoient, respectivement, la plus petite et la plus grande valeur de l'attribut `propriete` pour les enregistrements sélectionnés.

Des acteurs ayant joué dans différentes pièces de théâtre sont recensés dans une base de données `Theatre` dont le schéma relationnel est donné ci-dessous :

- `Piece` (`idPiece`, `titre`, `langue`)
- `Acteur` (`idActeur`, `nom`, `prenom`, `anneeNaiss`)
- `Role` (`#idPiece`, `#idActeur`, `nomRole`)

Dans ce schéma, les clés primaires sont soulignées et les clés étrangères sont précédées du symbole #.

L'attribut `idPiece` de la relation `Role` est une clé étrangère faisant référence à l'attribut `idPiece` de la relation `Piece`.

L'attribut `idActeur` de la relation `Role` est une clé étrangère faisant référence à l'attribut `idActeur` de la relation `Acteur`.

Tous les attributs dont le nom est préfixé par `id` sont des nombres entiers ainsi que l'attribut `anneeNaiss`. Les autres attributs sont des chaînes de caractères.

1. Expliquer pourquoi il n'est pas possible d'insérer une entrée dans la relation `Role` si les relations `Piece` et `Acteur` sont vides.
2. Dans la pièce « Le Tartuffe », l'acteur Micha Lescot a joué le rôle de Tartuffe. L'identifiant de Micha Lescot est 389761 et celui de cette pièce est 46721. Écrire une requête SQL permettant d'ajouter ce rôle dans la table (ou relation) `Role`.
3. Expliquer ce que fait la requête SQL suivante.

```
UPDATE Piece
SET langue = "Anglais"
WHERE langue = "Américain" OR langue = "Britannique";
```

4. Pour chacun des quatre items suivants, écrire une requête SQL permettant d'extraire les informations demandées.
 - a. Le nom et prénom des artistes nés après 1990.
 - b. L'année de naissance du plus jeune artiste.
 - c. Le nom des rôles joués par l'acteur Vincent Macaigne.
 - d. Le titre des pièces écrites en Russe dans lesquelles l'actrice Jeanne Balibar a joué.

Exercice) (4 points)

Cet exercice porte sur l'algorithmique et la programmation en Python. Il aborde les notions de tableaux de tableaux et d'algorithmes de parcours de tableaux.

Partie A : Représentation d'un labyrinthe

On modélise un labyrinthe par un tableau à deux dimensions à n lignes et m colonnes avec n et m des entiers strictement positifs.

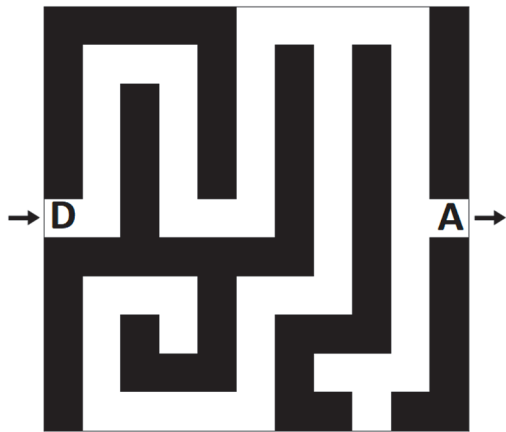
Les lignes sont numérotées de 0 à $n - 1$ et les colonnes de 0 à $m - 1$.

La case en haut à gauche est repérée par $(0,0)$ et la case en bas à droite par $(n-1, m-1)$.

Dans ce tableau :

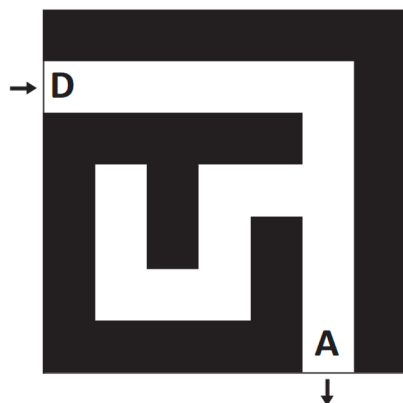
- 0 représente une case vide, hors case de départ et arrivée,
- 1 représente un mur,
- 2 représente le départ du labyrinthe,
- 3 représente l'arrivée du labyrinthe.

Ainsi, en Python, le labyrinthe ci-dessous est représentée par le tableau de tableaux `lab1`.



```
lab1 = [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [2, 0, 1, 0, 0, 0, 1, 0, 1, 0, 3],
        [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1]]
```

1. Le labyrinthe ci-dessous est censé être représenté par le tableau de tableaux `lab2`. Cependant, dans ce tableau, un mur se trouve à la place du départ du labyrinthe. Donner une instruction permettant de placer le départ au bon endroit dans `lab2`.



```
lab2 = [[1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 3, 1]]
```

2. Écrire une fonction `est_valide(i, j, n, m)` qui renvoie `True` si le couple (i, j) correspond à des coordonnées valides pour un labyrinthe de taille (n, m) , et `False` sinon. On donne ci-dessous des exemples d'appels.

```
>>> est_valide(5, 2, 10, 10)
True
>>> est_valide(-3, 4, 10, 10)
False
```

3. On suppose que le départ d'un labyrinthe est toujours indiqué, mais on ne fait aucune supposition sur son emplacement. Compléter la fonction `depart(lab)` ci-dessous de sorte qu'elle renvoie, sous la forme d'un tuple, les coordonnées du départ d'un labyrinthe (représenté par le paramètre `lab`). Par exemple, l'appel `depart(lab1)` doit renvoyer le tuple $(5, 0)$.

```
def depart(lab) :
    n = len(lab)
    m = len(lab[0])
    ...
```

4. Écrire une fonction `nb_cases_vides(lab)` qui renvoie le nombre de cases vides d'un labyrinthe (comprenant donc l'arrivée et le départ). Par exemple, l'appel `nb_cases_vides(lab2)` doit renvoyer la valeur 19.

Partie B : Recherche d'une solution dans un labyrinthe

On suppose dans cette partie que les labyrinthes possèdent un unique chemin allant du départ à l'arrivée sans repasser par la même case. Dans la suite, c'est ce chemin que l'on appellera solution du labyrinthe.

Pour déterminer la solution d'un labyrinthe, on parcourt les cases vides de proche en proche. Lors d'un tel parcours, afin d'éviter de tourner en rond, on choisit de marquer les cases visitées. Pour cela, on remplace la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4.

1. On dit que deux cases d'un labyrinthe sont voisines si elles ont un côté commun. On considère une fonction `voisines(i, j, lab)` qui prend en arguments deux entiers i et j représentant les coordonnées d'une case et un tableau `lab` qui représente un labyrinthe. Cette fonction renvoie la liste des coordonnées des cases voisines de la case de coordonnées (i, j) qui sont valides, non visitées et qui ne sont pas des murs. L'ordre des éléments de cette liste n'importe pas.

Ainsi, l'appel `voisines(1, 1, [[1, 1, 1], [4, 0, 0], [1, 0, 1]])` renvoie la liste $[(2, 1), (1, 2)]$.

Que renvoie l'appel `voisines(1, 2, [[1, 1, 4], [0, 0, 0], [1, 1, 0]])` ?

2. On souhaite stocker la solution dans une liste `chemin`. Cette liste contiendra les coordonnées des cases de la solution, dans l'ordre. Pour cela, on procède de la façon suivante.

- Initialement :
 - déterminer les coordonnées du départ : c'est la première case à visiter ;
 - ajouter les coordonnées de la case départ à la liste `chemin`.
- Tant que l'arrivée n'a pas été atteinte :
 - on marque la case visitée avec la valeur 4 ;
 - si la case visitée possède une case voisine libre, la première case de la liste renvoyée par la fonction `voisines` devient la prochaine case à visiter et on ajoute à la liste `chemin` ;
 - sinon, il s'agit d'une impasse. On supprime alors la dernière case dans la liste `chemin`. La prochaine case à visiter est celle qui est désormais en dernière position de la liste `chemin`.

a. Le tableau de tableaux `lab3` ci-dessous représente un labyrinthe.

```
lab3 = [[1, 1, 1, 1, 1, 1],
        [2, 0, 0, 0, 0, 3],
        [1, 0, 1, 0, 1, 1],
        [1, 1, 1, 0, 0, 1]]
```

La suite d'instructions ci-dessous simule le début des modifications subies par la liste `chemin` lorsque l'on applique la méthode présentée.

```
# entrée: (1, 0), sortie (1, 5)
chemin = [(1, 0)]
chemin.append((1, 1))
chemin.append((2, 1))
chemin.pop()
chemin.append((1, 2))
chemin.append((1, 3))
chemin.append((2, 3))
```

Compléter cette suite d'instructions jusqu'à ce que la liste `chemin` représente la solution. *Rappel : la méthode `pop` supprime le dernier élément d'une liste et renvoie cet élément.*

b. Recopier et compléter la fonction `solution(lab)` donnée ci-dessous de sorte qu'elle renvoie le chemin solution du labyrinthe représenté par le paramètre `lab`. On pourra pour cela utiliser la fonction `voisines`.

```
def solution(lab):
    chemin = [depart(lab)]
    case = chemin[0]
    i = case[0]
    j = case[1]
    ...
```

Par exemple, l'appel `solution(lab2)` doit renvoyer `[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)]`.

graphique 1 : Extrait des liaisons possibles en Occitanie

