BAC BLANC 2022 - SUJET 2
jeudi 13/01/2022

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 15 pages numérotées de 1/15 à 15/15.

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

Chaque exercice est noté sur 4 points.

Exercice 1 &2 nmg l rq'

Notion abordée : structures de données (dictionnaires)

Une ville souhaite gérer son parc de vélos en location partagée. L'ensemble de la flotte de vélos est stocké dans une table de données représentée en langage Python par un dictionnaire contenant des associations de type `id_velo : dict_velo` où `id_velo` est un nombre entier compris entre 1 et 199 qui correspond à l'identifiant unique du vélo et `dict_velo` est un dictionnaire dont les clés sont : "`type`", "`etat`", "`station`".

Les valeurs associées aux clés "`type`", "`etat`", "`station`" de `dict_velo` sont de type chaînes de caractères ou nombre entier :

- "`type`" : chaîne de caractères qui peut prendre la valeur "`electrique`" ou "`classique`"
- "`état`" : nombre entier qui peut prendre la valeur 1 si le vélo est disponible, 0 si le vélo est en déplacement, -1 si le vélo est en panne
- "`station`" : chaînes de caractères qui identifie la station où est garé le vélo.

Dans le cas où le vélo est en déplacement ou en panne, "`station`" correspond à celle où il a été dernièrement stationné.

Voici un extrait de la table de données :

```
flotte = {
     12 : {"type" : "electrique", "etat" : 1, "station" : "Prefecture"},
     80 : {"type" : "classique", "etat" : 0, "station" : "Saint-Leu"},
     45 : {"type" : "classique", "etat" : 1, "station" : "Baraban"},
     41 : {"type" : "classique", "etat" : -1, "station" : "Citadelle"},
     26 : {"type" : "classique", "etat" : 1, "station" : "Coliseum"},
     28 : {"type" : "electrique", "etat" : 0, "station" : "Coliseum"},
     74 : {"type" : "electrique", "etat" : 1, "station" : "Jacobins"},
     13 : {"type" : "classique", "etat" : 0, "station" : "Citadelle"},
     83 : {"type" : "classique", "etat" : -1, "station" : "Saint-Leu"},
     22 : {"type" : "electrique", "etat" : -1, "station" : "Joffre"}
  }
```

`flotte` étant une variable globale du programme.

Toutes les questions de cet exercice se réfèrent à l'extrait de la table `flotte` fourni ci-dessus.

1.

1.a. Que renvoie l'instruction `flotte[26]` ?

1.b. Que renvoie l'instruction `flotte[80]["etat"]` ?

1.c. Que renvoie l'instruction `flotte[99]["etat"]` ?

2. Voici le script d'une fonction :

```
def proposition(choix):
  for v in flotte:
    if flotte[v]["type"] == choix and flotte[v]["etat"] == 1:
      return flotte[v]["station"]
```

2.a. **Quelles sont** les valeurs possibles de la variable `choix` ?

2.b. **Expliquer** ce que renvoie la fonction lorsque l'on choisit comme paramètre l'une des valeurs possibles de la variable `choix`.

3.

3.a. Écrire un script en langage Python qui affiche les identifiants (`id_velo`) de tous les vélos disponibles à la station "`Citadelle`".

3.b. Écrire un script en langage Python qui permet d'afficher l'identifiant (`id_velo`) et la station de tous les vélos électriques qui ne sont pas en panne.

4. On dispose d'une table de données des positions GPS de toutes les stations, dont un extrait est donné ci-dessous. Cette table est stockée sous forme d'un dictionnaire.

Chaque élément du dictionnaire est du type:

'nom de la station' : (latitude, longitude)

```
stations = {
'Prefecture' : (49.8905, 2.2967) ,
'Saint-Leu' : (49.8982, 2.3017),
'Coliseum' :  (49.8942, 2.2874),
'Jacobins' : (49.8912, 2.3016)
}
```

On **admet** que l'on dispose d'une fonction `distance(p1, p2)` permettant de renvoyer la distance en mètres entre deux positions données par leurs coordonnées GPS (latitude et longitude).

Cette fonction prend en paramètre deux tuples représentant les coordonnées des deux positions GPS et renvoie un nombre entier représentant cette distance en mètres.

Par exemple, `distance((49.8905, 2.2967), (49.8912, 2.3016))` renvoie `9591`

Écrire une fonction qui prend en paramètre les coordonnées GPS de l'utilisateur sous forme d'un tuple et qui renvoie, pour chaque station située à moins de 800 mètres de l'utilisateur :

- le nom de la station ;
- la distance entre l'utilisateur et la station ;
- les identifiants des vélos disponibles dans cette station.

Une station où aucun vélo n'est disponible ne doit pas être affichée.

Exercice 2 (4 points)

Thème abordé : structures de données : les piles

On cherche à obtenir un mélange d'une liste comportant un nombre **pair** d'éléments. Dans cet exercice, on notera N le nombre d'éléments de la liste à mélanger.

La méthode de mélange utilisée dans cette partie est inspirée d'un mélange de jeux de cartes :

- On sépare la liste en deux piles :
  - ⇨ à gauche, la première pile contient les N/2 premiers éléments de la liste ;
  - ⇨ à droite, la deuxième pile contient les N/2 derniers éléments de la liste.
- On crée une liste vide.
- On prend alors le sommet de la pile de gauche et on le met en début de liste.
- On prend ensuite le sommet de la pile de droite que l'on ajoute à la liste et ainsi de suite jusqu'à ce que les piles soient vides.

Par exemple, si on applique cette méthode de mélange à la liste `['V','D','R','3','7','10']`, on obtient pour le partage de la liste en 2 piles :

Pile gauche
`'R'`
`'D'`
`'V'`

Pile droite
`'10'`
`'7'`
`'3'`

La nouvelle liste à la fin du mélange sera donc `['R','10','D','7','V','3']`.

1. Que devient la liste `['7','8','9','10','V','D','R','A']` si on lui applique cette méthode de mélange ?

On considère que l'on dispose de la structure de données de type pile, munie des seules instructions suivantes :

`p = Pile()` : crée une pile vide nommée `p`

`p.est_vide()` : renvoie Vrai si la liste est vide, Faux sinon

`p.empiler(e)` : ajoute l'élément `e` dans la pile

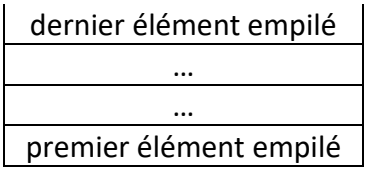
`e = p.depiler()` : retire le dernier élément ajouté dans la pile et le retourne (et l'affecte à la variable `e`)

`p2 = p.copier()` : renvoie une copie de la pile `p` sans modifier la pile `p` et l'affecte à une nouvelle pile `p2`

2. Recopier et compléter le code de la fonction suivante qui transforme une liste en pile.

```
def liste_vers_pile(L):
    '''prend en paramètre une liste et renvoie une
pile'''
    N = len(L)
    p_temp = Pile()
    for i in range(N):
      …….
    return …….
```

3. On considère la fonction suivante qui partage une liste en deux piles. Lors de sa mise au point et pour aider au débuggage, des appels à la fonction `affichage_pile` ont été insérés. La fonction `affichage_pile(p)` affiche la pile `p` à l'écran verticalement sous la forme suivante :




```
def partage(L):
    N = len(L)
    p_gauche = Pile()
    p_droite = Pile()
    for i in range(N/2):
        p_gauche.empile(L[i])
    for i in range(N/2,N):
        p_droite.empile(L[i])
    affichage_pile(p_gauche)
    affichage_pile(p_droite)
    return p_gauche, p_droite
```

Quels affichages obtient-on à l'écran lors de l'exécution de l'instruction : `partage([1,2,3,4,5,6])` ?

4.

4.a Dans un cas général et en vous appuyant sur une séquence de schémas, **expliquer** en quelques lignes comment fusionner deux piles `p_gauche` et `p_droite` pour former une liste `L` en alternant un à un les éléments de la pile `p_gauche` et de la pile `p_droite`.

4.b. **Écrire** une fonction `fusion(p1,p2)` qui renvoie une liste construite à partir des deux piles `p1` et `p2`.

5. Compléter la dernière ligne du code de la fonction `affichage_pile` pour qu'elle fonctionne de manière récursive.

```
def affichage_pile(p):
  p_temp = p.copier()
  if p_temp.est_vide():
    print('____')
  else:
    elt = p_temp.depiler()
    print('| ', elt, ' |'))
    ...                          # ligne à compléter
```

EXERCICE 3 (4 points)

Cet exercice porte sur les arbres binaires de recherche et la programmation orientée objet.

On rappelle qu'un arbre binaire est composé de nœuds, chacun des nœuds possédant éventuellement un sous-arbre gauche et éventuellement un sous-arbre droit. Un nœud sans sous-arbre est appelé feuille. La taille d'un arbre est le nombre de nœuds qu'il contient ; sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles. Ainsi la hauteur d'un arbre réduit à un nœud, c'est-à-dire la racine, est 1.

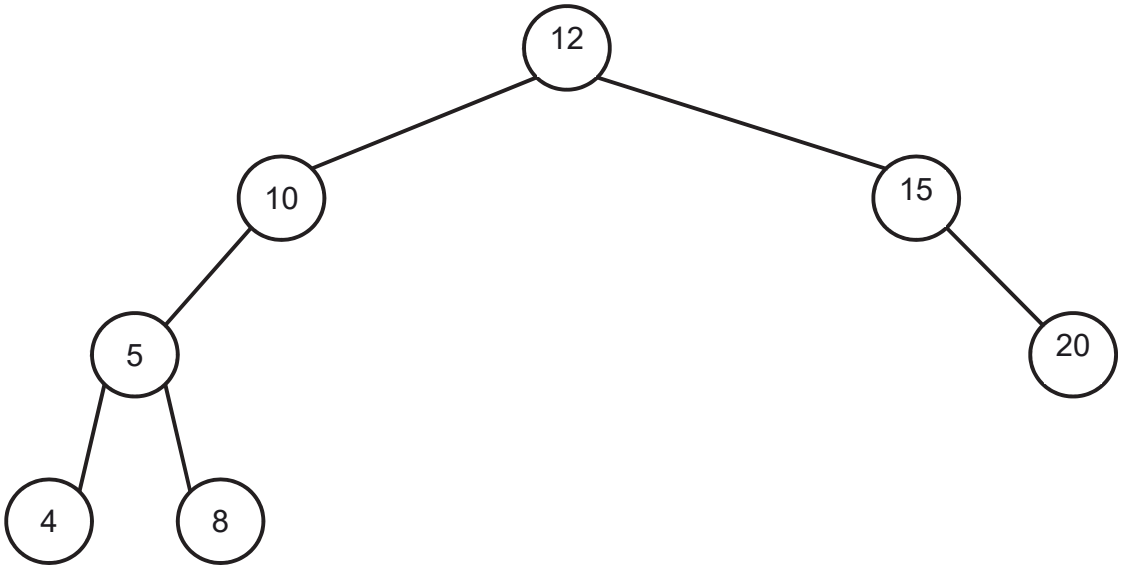
Dans un arbre binaire de recherche, chaque nœud contient une clé, ici un nombre entier, qui est :

- strictement supérieure à toutes les clés des nœuds du sous-arbre gauche ;
- strictement inférieure à toutes les clés des nœuds du sous-arbre droit.

Ainsi les clés de cet arbre sont toutes distinctes.

Un arbre binaire de recherche est dit « bien construit » s'il n'existe pas d'arbre de hauteur inférieure qui pourrait contenir tous ses nœuds.

On considère l'arbre binaire de recherche ci-dessous.



1. **a**. Quelle est la taille de l'arbre ci-dessus ?
   b. Quelle est la hauteur de l'arbre ci-dessus ?

2. Cet arbre binaire de recherche n'est pas « bien construit ». Proposer un arbre binaire de recherche contenant les mêmes clés et dont la hauteur est plus petite que celle de l'arbre initial.

3. Les classes `Noeud` et `Arbre` ci-dessous permettent de mettre en œuvre en Python la structure d'arbre binaire de recherche. La méthode `insere` permet d'insérer récursivement une nouvelle clé.

```
class Noeud :

    def __init__(self, cle):
        self.cle = cle
        self.gauche = None
        self.droit = None

    def insere(self, cle):
        if cle < self.cle :
            if self.gauche == None :
                self.gauche = Noeud(cle)
            else :
                self.gauche.insere(cle)
        elif cle > self.cle :
            if self.droit == None :
                self.droit = Noeud(cle)
            else :
                self.droit.insere(cle)

class Arbre :

    def __init__(self, cle):
        self.racine = Noeud(cle)

    def insere(self, cle):
        self.racine.insere(cle)
```

Donner la représentation de l'arbre codé par les instructions ci-dessous.

```
a = Arbre(10)
a.insere(20)
a.insere(15)
a.insere(12)
a.insere(8)
a.insere(4)
a.insere(5)
```

4. Pour calculer la hauteur d'un arbre non vide, on a écrit la méthode ci-dessous dans la classe `Noeud`.

```
def hauteur(self):
    if self.gauche == None and self.droit == None:
        return 1
    if self.gauche == None:
        return 1+self.droit.hauteur()
    elif self.droit == None:
        return 1+self.gauche.hauteur()
    else:
        hg = self.gauche.hauteur()
        hd = self.droit.hauteur()
        if hg > hd:
            return hg+1
        else:
            return hd+1
```

Écrire la méthode `hauteur` de la classe `Arbre` qui renvoie la hauteur de l'arbre.

5. Écrire les méthodes `taille` des classes `Noeud` et `Arbre` permettant de calculer la taille d'un arbre.

6. On souhaite écrire une méthode `bien_construit` de la classe `Arbre` qui renvoie la valeur `True` si l'arbre est « bien construit » et `False` sinon.

On rappelle que la taille maximale d'un arbre binaire de recherche de hauteur h est $2^h - 1$.

a. Quelle est la taille minimale, notée t_min, d'un arbre binaire de recherche « bien construit » de hauteur h ?

b. Écrire la méthode `bien_construit` demandée.

Exercice 4 (4 points)

Gestion d'un club de handball

Thèmes abordés : bases de données

Un club de handball souhaite regrouper efficacement toutes ses informations. Il utilise pour cela des bases de données relationnelles afin d'avoir accès aux informations classiques sur les licenciés du club ainsi que sur les matchs du championnat. Le langage SQL a été retenu.

On suppose dans l'exercice que tous les joueurs d'une équipe jouent à chaque match de l'équipe.

La structure de la base de données est composée des deux tables (ou relations) suivantes:

Table `licencies`	
Attributs	**Types**
`id_licencie`	INT
`prenom`	VARCHAR
`nom`	VARCHAR
`annee_naissance`	INT
`equipe`	VARCHAR

Table `matchs`	
Attributs	**Types**
`id_matchs`	INT
`equipe`	VARCHAR
`adversaire`	VARCHAR
`lieu`	VARCHAR
`date`	DATE

Ci-dessous un exemple de ce que l'on peut trouver dans la base de données :
Exemple **non exhaustif** d'entrées de la table `licencies`

`id_licencie`	`prenom`	`nom`	`annee_naissance`	`equipe`
63	Jean-Pierre	Masclef	1965	Vétérans
102	Eva	Cujon	1992	Femmes 1
125	Emile	Alinio	2000	Hommes 2
247	Ulysse	Trentain	2008	-12 ans

Exemple **non exhaustif** d'entrées de la table `matchs`

`id_match`	`equipe`	`adversaire`	`lieu`	`date`
746	-16 ans	PHC	Domicile	2021-06-19
780	Vétérans	PHC	Exterieur	2021-06-26
936	Hommes 3	LSC	Exterieur	2021-06-20
1032	-19 ans	LOH	Exterieur	2021-05-22
1485	Femmes 2	CHM	Domicile	2021-05-02
1512	Vétérans	ATC	Domicile	2021-04-12

1.

1.a. L'attribut `nom` de la table `licencies` pourrait-il servir de clé primaire ? **Justifier**.

1.b. **Citer** un autre attribut de cette table qui pourrait servir de clé primaire.

2.

2.a. **Expliquer** ce que renvoie la requête SQL suivante :

```
SELECT prenom,nom FROM licencies WHERE equipe ="-12ans"
```

2.b. **Que renvoie** la requête précédente si `prenom,nom` est remplacé par une étoile (*****) ?

2.c. **Ecrire** la requête qui permet l'affichage des dates de tous les matchs joués à domicile de l'équipe *Vétérans*.

3. **Ecrire** la requête qui permet d'inscrire dans la table `licencies`, *Jean Lavenu* né en *2001* de l'équipe *Hommes 2* et qui aura comme numéro de licence *287* dans ce club.

4. On souhaite mettre à jour les données de la table `licencies` du joueur *Joseph Cuviller*, déjà inscrit. Il était en équipe *Hommes 2* et il est maintenant en équipe *Vétérans*. Afin de modifier la table dans ce sens, **proposer** la requête adéquate.

5. Pour obtenir le nom de tous les licenciés qui jouent contre le LSC le 19 juin 2021, **recopier et compléter** la requête suivante :

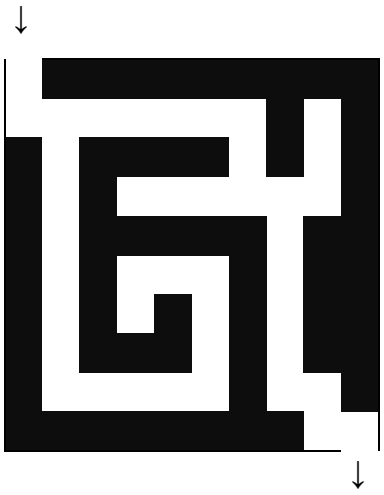
```
SELECT nom FROM licencies
JOIN Matchs ON licencies.equipe = matchs.equipe
WHERE …… ;
```

Exercice 5 (4 points)

Thèmes abordés : programmation Python, tuples et listes

L'objectif de cet exercice est de mettre en place une modélisation d'un jeu de labyrinthe en langage Python.

On décide de représenter un labyrinthe par un tableau carré de taille `n`, dans lequel les cases seront des 0 si l'on peut s'y déplacer et des 1 s'il s'agit d'un mur. Voici un exemple de représentation d'un labyrinthe :



```
laby=[[0,1,1,1,1,1,1,1,1,1],
      [0,0,0,0,0,0,0,1,0,1],
      [1,0,1,1,1,1,0,1,0,1],
      [1,0,1,0,0,0,0,0,0,1],
      [1,0,1,1,1,1,1,0,1,1],
      [1,0,1,0,0,0,1,0,1,1],
      [1,0,1,0,1,0,1,0,1,1],
      [1,0,1,1,1,0,1,0,1,1],
      [1,0,0,0,0,0,1,0,0,1],
      [1,1,1,1,1,1,1,1,0,0]]
```

L'entrée du labyrinthe se situe à la première case du tableau (celle en haut à gauche) et la sortie du labyrinthe se trouve à la dernière case (celle en bas à droite).

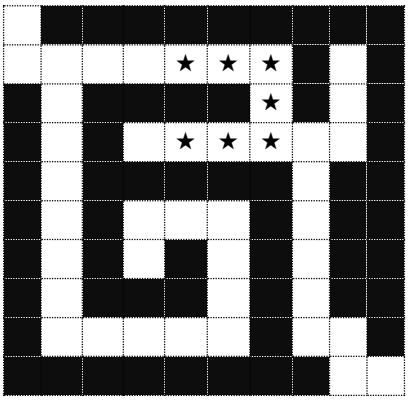
1. **Proposer**, en langage Python, une fonction `mur`, prenant en paramètre un tableau représentant un labyrinthe et deux entiers `i` et `j` compris entre 0 et `n-1` et qui renvoie un booléen indiquant la présence ou non d'un mur. Par exemple :

```
>>mur(laby, 2, 3)
True
>>mur(laby, 1, 8)
False
```

Un parcours dans le labyrinthe va être représenté par une liste de **cases.** Il s'agit de couples `(i,j)` où `i` et `j` correspondent respectivement aux numéros de ligne et de colonne des cases successivement visitées au long du parcours. Ainsi, la liste suivante

`[(1,4),(1,5),(1,6),(2,6),(3,6),(3,5),(3,4)]`

correspond au parcours repéré par des étoiles ⋆ ci-contre :



La liste `[(0,0),(1,0),(1,1),(5,1),(6,1)]` ne peut correspondre au parcours d'un labyrinthe car toutes les cases parcourues successivement ne sont pas adjacentes.

2. On considère la fonction `voisine` ci-dessous, écrite en langage Python, qui prend en paramètres deux cases données sous forme de couple.

```
def voisine(case1, case2) :
  l1, c1 = case1
  l2, c2 = case2
  # on vous rappelle que **2 signifie puissance 2
  d = (l1-l2)**2 + (c1-c2)**2
  return (d == 1)
```

2.a. Après avoir remarqué que les quantités `l1-l2` et `c1-c2` sont des entiers, **expliquer** pourquoi la fonction `voisine` indique si deux cases données sous forme de tuples `(l,c)` sont adjacentes.

2.b. **En déduire** une fonction `adjacentes` qui reçoit une liste de cases et renvoie un booléen indiquant si la liste des cases forme une chaîne de cases adjacentes.

Un parcours sera qualifié de ***compatible avec le labyrinthe*** lorsqu'il s'agit d'une succession de cases adjacentes accessibles (non murées). On donne la fonction `teste(cases, laby)` qui indique si le chemin `cases` est un chemin possible compatible avec le labyrinthe `laby` :

```
def teste(cases, laby) :
    if not adjacentes(cases) :
        return False
    possible = True
    i = 0
    while i < len(cases) and possible:
        if mur(laby, cases[i][0], cases[i][1]) :
            possible = False
        i = i + 1
    return possible
```

3. **Justifier** que la boucle de la fonction précédente se termine.
4. **En déduire** une fonction `echappe(cases, laby)` qui indique par un booléen si le chemin `cases` permet d'aller de l'entrée à la sortie du labyrinthe `laby`.