REPUBLIQUE TUNISIENNE

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Concours Nationaux d'Entrée
aux Cycles de Formation d'Ingénieurs
Session 2021

الجمهورية التونسية

وزارة التعليم العالي والبحث العلمي

المناظرات الوطنية للدخول
إلى مراحل تكوين المهندسين
دورة 2021

Concours Mathématiques et Physique, Physique et Chimie et Technologie

Epreuve d'Informatique + Correction

Date : **Samedi 26 Juin 2021** **Heure** : **8 H** **Durée** : **2 H** **Nombre de pages** : **10**

Barème : **PROBLEME 1 : 14 points**
PROBLEME 2 : 6 points

DOCUMENTS NON AUTORISES
L'USAGE DES CALCULATRICES EST INTERDIT
Il FAUT RESPECTER IMPERATIVEMENT LES NOTATIONS DE L'ENONCE
VOUS POUVEZ EVENTUELLEMENT UTILISER LES FONCTIONS PYTHON DECRITES A L'ANNEXE (PAGE 10)

PROBLEME 1

Dans ce problème, on s'intéresse à la segmentation d'images en niveaux de gris en régions par la méthode de partage des eaux (issue de la morphologie mathématique).
Il est à noter que le problème peut être traité sans aucun prérequis relatif au domaine du traitement d'images.
Dans ce qui suit nous présentons quelques notions de base puis une description de la méthode utilisée.

Notions de base

- Une image en niveaux de gris est une collection de pixels organisés sous forme matricielle, où chaque pixel est défini par :
  - ses coordonnées `(y,x)` où `y` est l'indice de ligne et `x` est l'indice de colonne ;
  - sa valeur, parmi 256 valeurs possibles entre le noir et le blanc (0 pour le noir et 255 pour le blanc), représentant le niveau de gris du pixel.
- Une image dite normalisée, associée à une image donnée en niveaux de gris, est une image où les niveaux de gris sont des réels dans l'intervalle $[0,1]$.
- Une région de l'image est un ensemble de pixels regroupés selon un critère spécifique.
- Une région uniforme est formée par des pixels ayant exactement le même niveau de gris (même intensité lumineuse).

– Les 8-voisins d'un pixel `p` de coordonnées `(y,x)` sont ses voisins immédiats, comme l'illustre la figure ci-dessous.

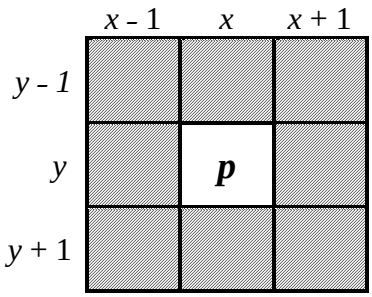

Figure : Les 8-voisins immédiats (cases hachurées) du pixel `p`.

– Deux pixels `p` et `q` n'appartenant pas aux bords de l'image, sont dits voisins si et seulement si `p` est l'un des 8-voisins immédiats de `q` ou réciproquement.
– Une région peut être formée par une ou plusieurs composantes connexes.
– Une composante connexe est formée par des pixels qui sont tous interconnectés selon la relation de 8-voisinage.
– La frontière d'une région `r` est formée par les pixels qui possèdent un voisin extérieur à la région `r`.
– Deux régions `r1` et `r2` sont adjacentes s'il existe un pixel `p` dans la frontière de `r1` et un pixel `q` dans la frontière de `r2` tel que `p` est voisin immédiat de `q`.
– Deux régions `r1` et `r2` sont dites disjointes si elles n'ont aucun pixel en commun.
– La segmentation d'une image consiste à partitionner l'image en régions connexes et disjointes.
– Un masque associé à une image en niveaux de gris, est une image en noir et blanc où chaque pixel a une valeur booléenne, `False` pour le noir et `True` pour le blanc.

Description de la méthode

La méthode de segmentation par la ligne de partage des eaux, abordée par le présent sujet, considère une image en niveaux de gris comme un relief topographique dont on simule l'inondation.
Notons que l'algorithme à implémenter est un algorithme naïf choisi pour les besoins de l'énoncé.
Dans une image en niveaux de gris, les points de fortes intensités (pixels proches du blanc) forment les sommets des collines et les points de basses intensités (pixels proches du noir) forment les minimums du relief.
Le processus d'inondation simule une montée progressive du niveau d'eau à partir des minimums du relief jusqu'à ce que toute l'image soit inondée.
Les points inondés par une seule source d'eau forment le bassin versant associé à cette source. Les différents bassins versants obtenus à la fin du processus correspondent aux régions de la segmentation. Les points, où les eaux provenant de différentes sources d'eau se rencontrent, forment une ligne de partage des eaux. Ces points sont les frontières entre les régions résultantes de la segmentation.
Dans la suite, on cherche à déterminer les différents bassins versants correspondants aux différentes régions de la segmentation à partir des minimums associés ainsi que la ligne de partage des eaux.

Hypothèses de travail

- Les réponses seront écrites en langage Python.
- Les tableaux à une ou à deux dimensions (vecteurs et matrices) sont représentés par des instances de la classe `numpy.ndarray`.
- Une région de l'image est représentée par un ensemble (instance de la classe `set`) de tuples `p` (`p=(y,x)`) décrivant les coordonnées des pixels appartenant à cette région.
- La bibliothèque `numpy` est importée par : `import numpy as np`.
- Les opérations énumérées ci-dessous sont de complexité asymptotique constante (**`O(1)`**) :
  - L'ajout d'un élément à un dictionnaire ou à un ensemble ;
  - La vérification de l'appartenance d'une clé à un dictionnaire ;
  - L'accès à la valeur associée à une clé `k` dans un dictionnaire ;
  - L'accès à un pixel de coordonnées `(y,x)` dans une image.

L'algorithme de segmentation par la méthode de partage des eaux, à implémenter en **Partie II** fera intervenir les classes utilitaires à construire dans la **Partie I**.

Partie I

Description des classes

- Classe **`PEPS`** :

  Rôle : Cette classe permet de représenter une file d'attente où les opérations d'ajout et de retrait sont basées sur le principe « Premier Entré Premier Sorti ».

  Attributs :

  - **`items`** est un dictionnaire où :
    - chaque clé, notée **`t`**, est un réel indiquant l'instant d'enfilement du pixel ;
    - chaque valeur notée **`p`**, est un tuple formé par les coordonnées `(y,x)` du pixel actuellement enfilé.

  Méthodes :

  - **`__init__(`**…**`)`** : permet d'initialiser l'attribut **`items`** à un dictionnaire vide ;
  - **`__len__(`**…**`)`** : permet de retourner la taille de la file ;
  - **`estvide(`**…**`)`** : retourne `True` si la file est vide, `False` sinon ;
  - **`enfiler(`**…**`)`** : permet d'enfiler les coordonnées relatives au pixel `p` passé en paramètre. L'instant d'enfilement `t` est obtenu en invoquant la fonction **`perf_counter`** du module **`time`**. (Cette fonction ne prend aucun paramètre et retourne un réel) ;
  - **`tete(`**…**`)`** : retourne les coordonnées relatives au pixel le plus ancien dans la file sans le retirer si la file n'est pas vide, sinon déclencher une exception ;
  - **`defiler(`**…**`)`** : permet de retirer les coordonnées relatives au pixel le plus ancien de la file puis le retourner si la file n'est pas vide, sinon déclencher une exception ;
  - **`queue(`**…**`)`** : retourne les coordonnées relatives au pixel le plus récent introduit dans la file sans le retirer si la file n'est pas vide, sinon déclencher une exception.

– Classe **`RegionsUniformes`** :

Rôle : Cette classe permet de représenter les régions uniformes d'une image en niveaux de gris.

Attributs :

- **`nimg`** : une matrice représentant une image en niveaux de gris normalisée ;
- **`reg`** : un dictionnaire où :
  - chaque clé est un niveau de gris de **`nimg`** ;
  - chaque valeur est un ensemble de couples (instance de la classe `set`) contenant les coordonnées relatives aux pixels ayant le niveau de gris dans **`nimg`**.
- **`niv_g`** : un tableau unidimensionnel contenant les clés de **`reg`** triées dans l'ordre croissant.

Méthodes :

- **`__init__(…)`** : permet de créer une nouvelle instance à partir d'une matrice représentant une image normalisée **`nimg`** passée en paramètre ;
- **`__getitem__(…)`** : qui prend en entrée un seul paramètre **`p`** :
  - si **`p`** est un tuple formé par une paire d'entiers, retourne le niveau de gris du pixel de coordonnées **`p`** (**`p=(y,x)`**) ;
  - si **`p`** est un réel dans l'intervalle $[0,1]$, retourne l'ensemble des coordonnées associées aux pixels ayant le niveau de gris **`p`** ;
  - Sinon déclencher une exception avec un message instructif.

  Cette méthode doit avoir une complexité asymptotique en **`O(1)`**.

Travail demandé

Pour la classe **`PEPS`** :

1. Écrire la méthode **`__init__`**.

```
def __init__(self):
  self.items = {}    #self.items=dict()
```

---

2. Écrire la méthode **`__len__`**.

```
def __len__(self):
  return len(self.items)
```

---

3. Écrire la méthode **`estvide`**.

```
def estvide(self):
  return len(self) == 0    # return len(self.items)==0
```

---

4. Écrire la méthode **`enfiler`**.

```
def enfiler(self, item):
  from time import perf_counter
  self.items[perf_counter()] = item
```

5. Écrire la méthode **`tete`**.

```
def tete(self):
    assert not self.estvide()
    return self.items[min(self.items)]
  #version 2
  def tete(self):
    if len(self) == 0:
      raise Exception(" Erreur File Vide ! ")
    return self.items[min(self.items)]
  #version 3
  def tete(self):
    try:
      if len(self)!=0:
        return self.items[min(self.items.keys())]
    except: print("File Vide")
```

6. Écrire la méthode **`defiler`**.

```
def defiler(self):
    assert not self.estvide()
    return self.items.pop(max(self.items))
#version 2
def defiler(self):
    assert len(self) != 0, "Erreur File Vide!"
    return self.items.pop(min(self.items))
```

7. Écrire la méthode **`queue`**.

```
def queue(self):
    assert not self.estvide()
    return self.items[max(self.items)]
#version 2
def queue(self):
    if len(self) == 0:
      raise Exception(" Erreur File Vide ! ")
    return self.items[max(self.items) ]
```

Pour la classe **`RegionsUniformes`** :

8. Écrire la méthode **`__init__`**.

```
def __init__(self, nimg):
  self.nimg = nimg.copy()
  self.reg = {}
  h, w = self.nimg.shape
  for y in range(h):
    for x in range(w):
      ton = self.nimg[y,x]
      if ton in self.reg:
        self.reg[ton].add((y,x))
      else:
        self.reg[ton] = {(y,x)}
  self.niv_g = np.array(sorted(self.reg))
```

---

9. Exprimer en fonction de **`h`** et **`w`**, respectivement le nombre de lignes et de colonnes de l'image **`nimg`**, la complexité asymptotique des instructions permettant de créer le dictionnaire **`reg`**.

`O(h x w )`

---

10. Écrire la méthode **`__getitem__`**. Cette méthode doit avoir une <u>complexité asymptotique en **`O(1)`**</u>.

```
def __getitem__(self, p):
    if isinstance(p, tuple): # if type(p)==  tuple
      return self.nimg[p]
    elif isinstance(p, float) and 0 <= p <= 1:
      return self.reg[p]
    else:
      raise Exception(" p must be an float or a tuple !!!")
```

Partie II

On s'apprête dans cette partie à normaliser puis segmenter selon la méthode de partage des eaux, une image en niveaux de gris représentée par une matrice.

<u>Travail demandé</u>

1. Écrire une fonction, nommée **`normaliser`**, qui prend en entrée une matrice **`img`**, calcule pixel par pixel la matrice **`nimg`** représentant l'image en niveaux de gris normalisée de même taille.

Pour un pixel de coordonnées **`(y,x)`**, la formule de normalisation est :

$$nimg_{y,x} = \frac{img_{y,x} - v_{min}}{v_{max} - v_{min}}$$

où :

- v_{min} est le niveau de gris le plus faible dans **img**.
- v_{max} est le niveau de gris le plus élevé dans **img**.
- v_{min} n'est jamais égale à v_{max}.
- $0 \leq y < h$ avec h le nombre de lignes de **img**.
- $0 \leq x < w$ avec w le nombre de colonnes de **img**.

Cette fonction retourne la matrice **nimg**.

```
def normaliser(img):
  vmin = img.min()
  vmax = img.max()
  nimg = np.empty(img.shape, dtype = np.float)
  h, w = img.shape
  for y in range(h):
    for x in range(w):
      nimg[y,x] = (img[y,x]- vim)/(vmax - vmin)
  return nimg
#version2
def normaliser(img):
  imin = img.min()
  imax = img.max()
  return (img - imin)/(imax - imin)
```

---

2. Écrire une fonction, nommée **voisins**, qui prend en entrée un tuple **p** contenant les coordonnées d'un pixel et retourne un ensemble formé par les coordonnées des 8 pixels voisins immédiats du pixel de coordonnées **p**. Aucun traitement particulier n'est à prévoir pour les pixels associés aux bords de l'image.

```
def voisins(p):
  y, x = p
  res = set()
  for yp in range(y-1, y+2):
    for xp in range(x-1, x+2):
      res.add((yp, xp))
  res.remove((y,x))
  return res
#version2
def voisins(p):
  y, x = p
  res = set()
  for i in range(-1,2):
    for j in range(-1,2):
      res.add((y+i,x+j))
  return res - {p}
```

3. Écrire une fonction, nommée **`sont_voisins`**, qui prend deux tuples **`p`** et **`q`** représentant les coordonnées de deux pixels, retourne `True` si **`p`** et **`q`** sont voisins et `False` sinon.

```
def sont_voisins(p, q):
  return p in voisins(q)
```

---

4. Écrire une fonction, nommée **`masque`**, qui prend en entrée :

- un ensemble de tuples, noté **`r`**, représentant une région ;
- un tuple de 2 entiers, noté **`taille`**, contenant respectivement la hauteur et la largeur du masque à retourner.

Cette fonction retourne un masque où seuls les pixels de la région **`r`** sont blancs.

```
def masque(r, taille):
  img = np.zeros(taille, np.bool)
  for y, x in r:
    img[y,x] = True
  return img

#version 2
def masque(r, taille):
  res = np.zeros(shape = taille, dtype = np.bool)
  y, x = zip(*r)
  res[y,x] = True
  return res
#version 3
def masque(r, taille):
  h, w = taille
  return np.array([[(y,x) in r for x in range(w)] for y in range(h)])
```

---

5. Écrire une fonction, nommée **`frontiere`**, qui prend en entrée un ensemble de tuples représentant une région **`r`**, retourne un ensemble contenant les coordonnées des pixels appartenant à la frontière de **`r`**.

```
def frontiere(r):
  s = set()
  for p in r:
    for q in voisins(p):
      if q not in r:
        s.add(p)
        break
  return s
```

---

6. Écrire une fonction, nommée **`adjacentes`**, qui prend en entrée deux régions **`r1`** et **`r2`**, retourne `True` si **`r1`** et **`r2`** sont adjacentes, `False` sinon.

```
def adjacentes(r1, r2):
```

```
  for p in frontiere(r1):
    for q in frontiere(r2):
      if sont_voisins(p,q):
        return True
  return False
#version2
def adjacentes(r1,r2):
    f1=frontiere(r1)
    f2=frontiere(r2)
    for p in f1:
        if q in f2:
            if sont_voisins(p,q):
                return True
    return False
```

---

7. Écrire une fonction, nommée **`decouper`**, qui prend en entrée une région **`r`** et qui permet de la partitionner, comme décrit dans la suite, en sous-ensembles de pixels où chaque sous-ensemble est une composante connexe.

   Une composante connexe **`cc`** d'une région **`r`** est obtenue en faisant propager dans **`r`**, la relation de 8-voisinage à partir d'un pixel **`p`** de **`r`**.
   On commence par déterminer les pixels **`pv`** voisins immédiats de **`p`** qui sont dans **`r`**. Puis, on détermine pour chaque **`pv`** ses voisins immédiats qui sont dans **`r`** et ainsi de suite jusqu'à ce qu'il n'y ait plus de voisins à explorer. L'ensemble des pixels obtenus forme une région connexe.
   Pour traiter les différents niveaux de propagation, on utilise une file **`f`** instance de la classe **`PEPS`**.
   L'algorithme $\mathcal{A}$ suivant permet d'extraire une composante connexe à partir d'un pixel courant de **`r`** comme suit :

   a) Créer un ensemble vide noté **`cc`** relatif à la composante connexe en cours de propagation ;
   b) Enfiler dans **`f`** le pixel courant puis le retirer de **`r`** afin de ne plus le traiter ;

   Tant que la file **`f`** n'est pas vide, appliquer les étapes **c)** et **d)** :

   c) Défiler un pixel de la file qui sera le pixel courant, puis l'insérer dans l'ensemble **`cc`** ;
   d) Appliquer l'étape **b)** à tous les pixels voisins immédiats du pixel courant qui sont dans **`r`**.

   La liste, de tous les ensembles **`cc`** associées aux composantes connexes de **`r`**, est obtenue en appliquant l'algorithme $\mathcal{A}$ jusqu'à ce que **`r`** se réduise à un ensemble vide. Cette fonction retourne la liste de toutes les composantes connexes de **`r`**.

```
def decouper(r):
  f = PEPS()
  lst = []
```

```
while len(r) != 0:
  cc = set()
  p = r.pop()
  f.enfiler(p)
  while not f.estvide():
    cc.add(f.defiler())
    for q in voisins(p):
      if q in r:
        r.remove(q)
        f.enfiler(q)
  lst.append(cc)
return lst
```

---

8. Ecrire une fonction, nommée **`segmenter`**, qui prend en entrée une image normalisée **`nimg`** et qui retourne un dictionnaire **`res`** contenant aussi bien les régions finales résultantes de la segmentation de **`nimg`** selon le principe de la ligne de partage des eaux ainsi que les pixels formant cette ligne, considérée ici comme une région.

Le dictionnaire **`res`** a le format suivant :

- Chaque clé est un entier identifiant la région. Chaque nouvelle clé est attribuée à partir du nombre d'éléments déjà dans **`res`**. La clé particulière de valeur -1 est attribuée à la ligne de partage des eaux.
- Chaque valeur est un ensemble contenant les coordonnées des pixels de la région.

NB : Le dictionnaire **`res`** subit exclusivement l'une des opérations suivantes :

- Insérer une nouvelle région.
- Fusionner (par union ensembliste) une région **`c`** avec une autre région **`r`** déjà dans **`res`**. Le résultat de la fusion aura la même clé que **`r`**.

L'algorithme de la segmentation est ci-dessous détaillé :

a) Instancier à partir de **`nimg`** l'objet **`ru`** instance de la classe **`RegionsUniformes`** ;

b) Créer un dictionnaire **`res`**, initialement vide, qui contiendra progressivement les régions en cours de formation y compris les pixels formant la ligne de partage des eaux ;

c) Créer un ensemble vide **`lpe`** qui contiendra les pixels de la ligne de partage des eaux en cours de formation ;

d) Pour chaque niveau de gris **`g`** du vecteur **`niv_g`** de l'objet **`ru`** :

- Déterminer dans la liste **`lcc`** les ensembles associés aux composantes connexes de la région uniforme ayant ce niveau de gris **`g`** ;
- Pour chaque composante connexe **`c`** de **`lcc`** :
  - **i.** Déterminer les identifiants des régions de **`res`** adjacentes à **`c`** dans une liste notée **`adj`** ;
  - **ii.** Si **`c`** n'est adjacente à aucune région de **`res`** alors l'insérer comme nouvelle région dans **`res`** ;
  - **iii.** Si **`c`** est adjacente à une seule région **`r`** de **`res`** alors la fusionner avec la région **`r`** dans **`res`** ;

iv. Si **`c`** est adjacente à plus qu'une seule région alors fusionner **`c`** avec **`lpe`** dans le dictionnaire **`res`**.

```
def segmenter(nimg):
  ru = RegionsUniformes(nimg)
  res = {}
  lpe = set()
  res[-1] = lpe
  for g in ru.niv_g:
    lcc = decouper(ru[g])
    for c in lcc:
      #accepter la création de adj par programmation
      adj = [idr for idr in res if adjacentes(res[idr], c)]
      if len(adj) == 0:
        res[len(res)] = c
      elif len(adj) == 1:
        res[adj[-1]] |= c   # res[adj[0]] |= c
      else:
        res[-1] |= c
  return res
```

PROBLEME 2

Une entreprise agricole implémente une base de données relationnelle pour gérer les interventions auprès d'agriculteurs propriétaires de parcelles de terrains (portion de terrain pour usage agricole) ainsi que les paies mensuelles des intervenants.
Au besoin, un agriculteur fait appel à cette entreprise pour accomplir une ou plusieurs interventions à travers un ou plusieurs intervenants (ouvriers, techniciens, ingénieurs, etc.) spécialisés payés, selon un salaire journalier, à la fin de chaque mois.

Contraintes à considérer

- Un agriculteur possède une ou plusieurs parcelles de terrain.
- Une parcelle de terrain appartient à un et un seul propriétaire.
- L'entreprise paie ses intervenants mensuellement, en fonction du nombre de jours de leurs interventions.
- Une intervention est un travail effectué par un intervenant pendant un certain nombre de jours consécutifs sur une parcelle de terrain.
- Une intervention ne dépasse pas 28 jours.
- Une intervention peut être accomplie au cours d'un mois ou bien s'étaler sur deux mois consécutifs.
- Un intervenant est considéré comme un employé de l'entreprise.
- Un intervenant ne peut pas démarrer une nouvelle intervention avant de terminer l'intervention en cours.

Le schéma relationnel de la base de données décrit ci-dessous a été élaboré pour les besoins de l'énoncé.

- **Agriculteur** (idAg, nomAg, prenomAg, villeAg)

La table **Agriculteur** enregistre les informations concernant un agriculteur :

- idAg : identifiant de l'agriculteur (numéro de CIN) de type chaîne de caractères, clé primaire ;
- nomAg : nom de l'agriculteur de type chaîne de caractères ;
- prenomAg : prénom de l'agriculteur de type chaîne de caractères ;
- villeAg : ville de l'agriculteur de type chaîne de caractères.

- **Parcelle** (idPar, designation, villeP, superficie, #idAg)

  La table **Parcelle** enregistre les informations des parcelles de terrain des agriculteurs :

  - idPar : identifiant de la parcelle de terrain de type chaîne de caractères, clé primaire ;
  - designation : nom de la parcelle de terrain de type chaîne de caractères ;
  - villeP : ville où se situe la parcelle de terrain de type chaîne de caractères ;
  - superficie : superficie de la parcelle de terrain exprimée en hectare de type réel ;
  - idAg : identifiant de l'agriculteur propriétaire de la parcelle de terrain de type chaîne de caractères, clé étrangère qui fait référence à la table **Agriculteur**.

- **Intervenant** (idIn, nomIn, prenomIn, villeIn, salaireJ)

  La table **Intervenant** enregistre les informations relatives aux intervenants de l'entreprise agricole :

  - idIn : identifiant d'un intervenant (numéro de CIN) de type chaîne de caractères, clé primaire ;
  - nomIn : nom d'un intervenant de type chaîne de caractères ;
  - prenomIn : prénom d'un intervenant de type chaîne de caractères ;
  - villeIn : ville d'un intervenant de type chaîne de caractères ;
  - salaireJ : salaire journalier d'un intervenant de type réel.

- **Intervention** (#idIn, #idPar, dateDebut, nbJours)

  La table **Intervention** enregistre les interventions des intervenants :

  - idIn : identifiant d'un intervenant de type chaîne de caractères, clé étrangère qui fait référence à la table **Intervenant** ;
  - idPar : identifiant d'une parcelle de terrain de type chaîne de caractères, clé étrangère qui fait référence à la table **Parcelle** ;
  - dateDebut : date du début d'une intervention de type date au format "AAAA-MM-JJ" ;
  - nbJours : durée effective de l'intervention en nombre de jours consécutifs de type entier.

Partie 1

Exprimer en algèbre relationnelle les requêtes permettant de :

1. Donner les noms, prénoms et villes des grands agriculteurs. Un grand agriculteur est celui qui possède au moins une parcelle de terrain d'une superficie de plus de 10 hectares.

$$\prod_{\text{nomAg, prenomAg, villeAg}} \left(\sigma_{\text{superficie} \geq 10} \left(Agriculteur \underset{\text{idAg}}{\bowtie} Parcelle \right) \right)$$

2. Donner les identifiants des intervenants qui possèdent des parcelles de terrains.

$$\prod_{\text{idAg}} (\text{Parcelle}) \cap \prod_{\text{idIn}} (\text{Intervenant})$$

ou

$$\prod_{\text{idAg}} \left(\text{Parcelle} \underset{\text{Parcelle.idAg} = \text{Intervenant.idIn}}{\bowtie} \text{Intervenant} \right)$$

Partie 2

Exprimer en SQL les requêtes suivantes permettant de :

1. Créer la table **Parcelle** en respectant les contraintes spécifiées ci-dessus sachant que la table **Agriculteur** est déjà créée et remplie.

```
CREATE TABLE  Parcelle (
    idPar TEXT ,
    designation TEXT ,
    villeP TEXT ,
    superficie REAL,
    idAg TEXT ,
    PRIMARY KEY (idPar),
    FOREIGN KEY(idAg) REFERENCES Agriculteur (idAg)    );
/* ou bien */
CREATE TABLE Parcelle (
    idPar TEXT PRIMARY KEY,
    designation TEXT,
    villeP TEXT,
    superficie REAL,
    idAg TEXT REFERENCES Agriculteur );
```

Dans la suite on suppose que les tables de la base de données sont toutes créées et remplies en respectant les contraintes déjà énumérées.

---

2. Supprimer les intervenants n'ayant effectué aucune intervention depuis 2015.

```
DELETE FROM Intervenant
WHERE idIn NOT IN (
        SELECT idIn
        FROM Intervention
        WHERE dateDebut > "2015-01-01"
        );
```

```
/* ou bien */
DELETE FROM Intervenant
WHERE idIn NOT IN (
    SELECT idIn
    FROM Intervention
    WHERE dateDebut > "2015%"
    );
```

---

3. Déterminer les noms des intervenants ayant le salaire journalier le plus élevé.

```
SELECT nomIn
FROM  Intervenant
WHERE salaireJ = (
    SELECT MAX(salaireJ)
    FROM Intervenant
    );
```

---

4. Déterminer le coût des interventions faites sur la parcelle de désignation "Parcelle2" depuis le début de l'offre des services de l'entreprise agricole pour cette parcelle.

```
SELECT SUM(nbJours*salaireJ)
FROM Parcelle P, Intervenant I, Intervention It
Where (P.idPar =It.idPar)  AND (I.idIn=It.idIn) AND
(designation like "Parcelle2");
```

---

5. Déterminer les noms et les prénoms des agriculteurs possédant plus que trois parcelles.

```
SELECT nomAg, prenomAg
FROM Parcelle p, Agriculteur A
WHERE  P.idAg = A.idAg
GROUP BY  A.idAg
HAVING COUNT(*)>=3;
```

---

6. Déterminer le nombre des interventions réalisées par mois durant l'année 2020.

Indication

On donne la fonction `strftime` pouvant être invoquée dans une requête SQL et ayant la syntaxe suivante :

`Strftime(format,date)` qui pour une date identifiée par `date` et un format `format` (comme illustré dans le tableau ci-dessous) retourne une chaîne de caractères comme le montre l'exemple.

Format	Signification
%d	Jour du mois sur 2 caractères de "01" à "31"
%m	Mois de l'année sur 2 caractères de "01" à "12"
%y	L'année sur 4 caractères.

Exemple :

```
strftime("%d","2021-06-26") retourne "26"
strftime("%m","2021-06-26") retourne "06"
strftime("%Y","2021-06-26") retourne "2021"
```

```
SELECT strftime ("%m", dateDebut) AS mois, COUNT(*)
FROM Intervention
WHERE strftime ("%Y", dateDebut)="2020"
GROUP BY  mois
```

Partie 3

Dans la suite les fonctions demandées doivent être écrites en Python. On désigne par **Cur** le curseur d'exécution de requêtes et **dico** un dictionnaire tel que chaque clé est un numéro de mois et chaque valeur est le nombre de jours pour ce mois. On ne gère pas les années bissextiles.

```
dico={1:31,2:28,3:31,4:30,5:31,6:30,7:31,8:31,9:30,10:31,11:30,12:31}
```

1. Écrire une fonction **salaire_jour** qui prend en paramètre **Cur** et l'identifiant **Id** d'un intervenant, retourne son salaire journalier.

```
def salaire_jour(Cur, Id):
    Cur.execute("select salaireJ from intervenant where idIn like
 '{}'".format(Id))
    return Cur.fetchone()[0]
```

2. Écrire une fonction **t_precedent** qui prend en paramètre un mois **M** et une année **A**, retourne un tuple contenant le mois précédant **M** ainsi que l'année correspondante.

```
def t_precedent (M, A):
    if M in range(2,13):
        M=M-1
    else:
        M=12
        A=A-1
    return((M,A))
```

3. Écrire une fonction **salaire_mois** qui prend en paramètre **Cur**, un mois **M**, une année **A** et l'identifiant **Id** d'un intervenant, retourne son salaire mensuel s'il a effectué des interventions et 0 sinon.

```
def salaire_mois (Cur, M, A , Id):
    t=t_precedent(M,A)
    k1=''
    if len(str(M))==1:
        k1='0'
    k2=''
    if len(str(t[0]))==1:
        k2='0'
    Cur.execute("""
    SELECT strftime('%d',dateDebut),strftime('%m',dateDebut),
                        strftime('%Y',dateDebut),nbJours
    FROM  Intervenant I, Intervention It
    WHERE I.idIn=It.idIn
    AND  dateDebut>='{}-{}{}-01'
    AND  dateDebut<='{}-{}{}-{}'
    AND I.idIn={}""".format(t[1],k2,t[0],A,k1,M,dico[M],idIn))
    L=Cur.fetchall()
    nj=0
    for i in L:
        if int(i[1])==M:
            if int(i[0])+i[3]<=dico[M]:
                nj+=i[3]
            else:
                nj+=dico[M]-int(i[0])+1
        else:
            if int(i[0])+i[3]>dico[t[0]]:
                nj+=int(i[0])+i[3]-dico[t[0]]-1
    tf=salaire_jour(Cur,Id)
    return nj*tf
```

---

4. Écrire une fonction **`salaire_an`** qui prend en paramètre **`Cur`**, une année **`A`** et l'identifiant **`Id`** d'un intervenant, retourne son salaire annuel.

```
def salaire_an(Cur, A , Id):
    L=[]
    for M in range(1,13):
        L.append(salaire_mois(Cur, M, A , Id))
    return(sum(L))
#version2
def salaire_an(Cur, A , Id):
    return sum(salaire_mois(Cur, M, A , Id) for M in range(1,13))
```

---

5. Ecrire le script python permettant de :

- importer le module **`sqlite3`** ;
- se connecter à la base de données **`'BDAgricole.db'`**;

- créer le curseur **`cur`** d'exécution ;
- afficher les salaires annuels de tous les intervenants de l'entreprise agricole pour l'année 2020.

```
import sqlite3
BD=sqlite3.connect('BDAgricole.db')
Cur=BD.cursor()
dico={1:31,2:28,3:31,4:30,5:31,6:30,7:31,8:31,9:30,10:31,11:30,12
:31}
Cur.execute("select idIn from Intervenant")
L=Cur.fetchall()
for id in L:
   print("id={} salaire annuel={} ".format(id[0], salaire_an(Cur,
    2020 , id[0])))
```

ANNEXE – Quelques Fonctions/Méthodes Python

Sans aucune obligation, les fonctions suivantes pourraient vous être utiles.

Module numpy

- M.**shape** ou **shape**(M) retourne un tuple formé par le nombre de lignes et le nombre de colonnes d'une matrice M.
- **zeros**(…) retourne un tableau initialisé par des 0.
- **ones**(…) retourne un tableau initialisé par des 1.
- M.**copy**() retourne une copie de M.
- M.**min**() ou **min**(M) et M.**max**() ou **max**(M) retournent respectivement la valeur minimale et la valeur maximale d'une matrice M.
- np.**all**(M) retourne `True` si tous les éléments de M valent `True` et `False` sinon.
- np.**any**(M) retourne `True` si M contient au moins un élément qui vaut `True` et `False` sinon.

Opérations sur les itérables (str, tuple, list, dict, etc.)

- **len**(it) retourne le nombre d'éléments de l'itérable it.
- **range**(d,f,p) retourne la séquence des valeurs entières successives comprises entre d et f, f exclu, par pas=p.
- **min**(it) retourne la valeur minimale de l'itérable it.
- **max**(it) retourne la valeur maximale de l'itérable it.
- **sum**(it) retourne la somme des éléments de l'itérable it.
- x **in** it vérifie si x appartient à it.
- **sorted**(it) retourne une liste contenant les éléments de it dans l'ordre croissant.
- lst**.sort**() trie la liste lst dans l'ordre croissant.
- lst**.count**(val) retourne le nombre d'occurrences de val dans la liste lst.
- lst.**append**(val) ajoute val à la fin de la liste lst.
- lst.**remove**(val) supprime la première occurrence val de la liste lst.
- lst.**index**(val) retourne l'indice de la première occurrence val de la liste lst.
- source.**split**(motif) retourne une liste formée par des chaînes de caractères résultantes du découpage de la chaîne source autour de la chaîne motif.
- motif.**join**(itérable de chaînes) retourne une chaîne de caractères résultante de la concaténation des éléments de l'itérable intercalés par le motif.
- motif.**format**(paramètres) retourne une chaîne de caractères obtenue en substituant dans l'ordre chaque caractère {} dans motif par un objet dans paramètres.
- d.**values**() retourne un itérable formé par les valeurs du dictionnaire d.
- d.**items**() retourne un itérable de couples (k,v) ou k est une clé du dictionnaire d et v est la valeur associée.
- s.**add**(obj) ajoute obj à un ensemble s (instance de la classe `set`).
- s.**remove**(obj) supprime obj de l'ensemble s (instance de la classe `set`).
- s1.**union**(s2) retourne l'ensemble union des deux ensembles s1 et s2.
- s1.**intersection**(s2) retourne l'ensemble intersection des deux ensembles s1 et s2.
- s1.**difference**(s2) retourne l'ensemble différence des deux ensembles s1 et s2.