BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2024

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 15 pages numérotées de 1 / 15 à 15 / 15.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

EXERCICE 1 (6 points)

Cet exercice porte sur les structures de données FILE et PILE, les graphes et les algorithmes de parcours.

Partie A

Une agence de voyages organise différentes excursions dans une région de France et propose la visite de certaines villes. Ces excursions peuvent être visualisées sur le graphe ci-dessous : les sommets désignent les villes, les arêtes représentent les routes pouvant être empruntées pour relier deux villes et les poids des arêtes représentent des distances, exprimées en kilomètre.

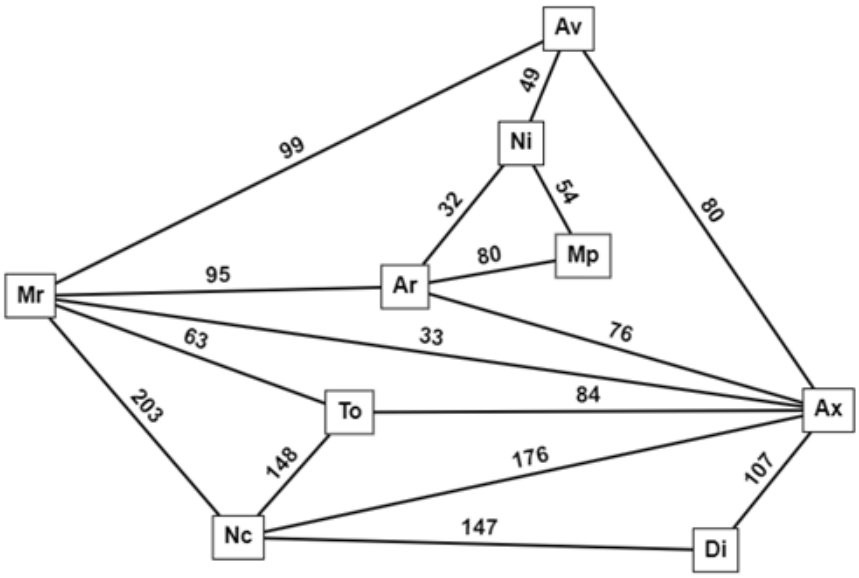

Figure 1. Graphe pondéré

1. Déterminer le plus court chemin allant du sommet Mp au sommet Nc et préciser la longueur, en kilomètres, de ce chemin. Aucune justification n'est attendue.

On souhaite toujours se rendre du sommet Mp au sommet Nc mais en visitant le minimum de villes.

2. Déterminer les deux chemins possibles.

Partie B

L'agence souhaite proposer un itinéraire permettant de visiter toutes les villes. On appelle G le graphe non pondéré ci-dessous.

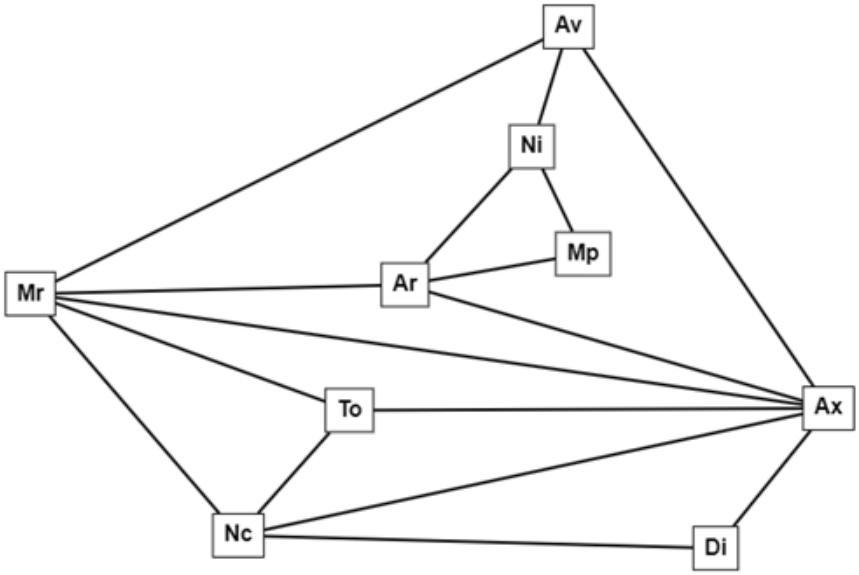

Figure 2. Graphe non pondéré

On choisit d'implémenter un graphe par listes d'adjacence, à l'aide d'un dictionnaire, en langage Python, dont :

- les clés sont les sommets du graphe ;
- la valeur associée à une clé est la liste des voisins de ce sommet clé.

Les sommets sont de type `str`.

3. Donner l'implémentation, en langage Python, du graphe de la figure 2. Le dictionnaire obtenu sera stocké dans une variable nommée `G`. Afin de faciliter la notation manuscrite ainsi que la lisibilité, écrire chaque couple clé/valeur sur une nouvelle ligne.

On considère une file.

4. Indiquer la signification des lettres dans les acronymes LIFO et FIFO.

5. Indiquer l'acronyme utilisé pour désigner la structure de file.

Voici, en langage Python, les opérations pouvant être effectuées sur une telle file :

- `creerFile()` : renvoie une file vide ;
- `estVide(F)` : renvoie `True` si la file `F` est vide et `False` sinon ;
- `enfiler(F, e)` : ajoute l'élément `e` dans la file `F` ;
- `defiler(F)` : renvoie l'élément à la tête de la file `F` en le retirant de la file `F`.

On donne la fonction `parcours` ci-dessous. Cette fonction prend en paramètres un dictionnaire `graphe` représentant un graphe sous la forme de listes d'adjacence, et une chaîne de caractères `sommet` représentant un sommet du graphe.

```
1 def parcours(graphe, sommet):
2     f = creerFile()
3     enfiler(f, sommet)
4     visite = [sommet]
5     while not estVide(f):
6         s = defiler(f)
7         for v in graphe[s]:
8             if not (v in visite):
9                 visite.append(v)
10                enfiler(f, v)
11    return visite
```

6. Donner le résultat renvoyé par l'appel `parcours(G, 'Av')`.
7. Recopier, parmi les deux propositions ci-dessous, celle qui correspond au type de parcours de graphe réalisé par la fonction `parcours` :

- **proposition A :** parcours en largeur ;
- **proposition B :** parcours en profondeur.

Dans la suite de l'exercice, la distance entre deux sommets désignera le nombre d'arêtes séparant ces deux sommets. Ainsi définie, la distance entre les sommets Mp et Nc du graphe de la figure 2 est 3.

8. En modifiant la fonction `parcours`, écrire une fonction `distance` ayant pour paramètres `graphe`, un dictionnaire représentant un graphe sous la forme de listes d'adjacence, et une chaîne de caractères `sommet` représentant un sommet du graphe. Cette fonction renvoie un dictionnaire tel que :
   - les clés sont les sommets du graphe ;
   - la valeur associée à une clé est la distance entre ce sommet clé et le sommet d'origine `sommet`.
9. Donner le résultat renvoyé par l'appel `distance(G, 'Av')`.

On considère une pile.
Voici, en langage Python, les opérations pouvant être effectuées sur une telle pile :

- `creerPile()` : renvoie une pile vide ;
- `estVide(P)` : renvoie `True` si la pile `P` est vide et `False` sinon ;
- `empiler(P, e)` : ajoute l'élément `e` au sommet de la pile `P` ;
- `depiler(P)` : renvoie le sommet de la pile `P` en le retirant de la pile `P`

On donne, ci-dessous, le pseudo-code d'un algorithme de parcours d'un graphe `G` à partir d'un sommet `s` :

```
1  créer une pile p
2  empiler s dans p
3  créer une liste visite contenant s
4  tant que p n'est pas vide
5      x = depiler p
6      si x n'est pas dans la liste visite
7          ajouter x à la liste visite
8          pour chaque voisin v de x
9              empiler v dans p
10         fin pour
11     fin si
12 fin tant que
13 renvoyer visite
```

10. Traduire, dans le corps d'une fonction Python nommée `parcours2`, l'algorithme en pseudo-code donné précédemment. Cette fonction prendra pour paramètres `G`, un dictionnaire représentant un graphe sous la forme de listes d'adjacence, et une chaîne de caractères `s` représentant un sommet du graphe.

11. Donner un résultat possible renvoyé par l'appel `parcours2(G, 'Av')`.

Exercice 2 (6 points)

Cet exercice porte sur les arbres binaires de recherche, la POO et la récursivité.

Nous disposons d'une classe ABR pour les arbres binaires de recherche dont les clés sont des entiers :

```
1  class ABR():
2      def __init__(self) :
3          # Initialise une instance d'ABR vide.
4
5      def cle(self):
6          # Renvoie la clé de la racine de l'instance d'ABR.
7
8      def sad(self):
9          # Renvoie le sous-arbre droit de l'instance d'ABR.
10
11     def sag(self):
12         # Renvoie le sous-arbre gauche de l'instance d'ABR.
13
14     def est_vide(self):
15         # Renvoie True si l'instance d'ABR est vide et False
sinon.
16
17     def inserer(self, cle_a_inserer):
18         # Insère cle_a_inserer à sa place dans l'instance d'ABR.
```

Considérons ci-dessous trois arbres binaires de recherche :

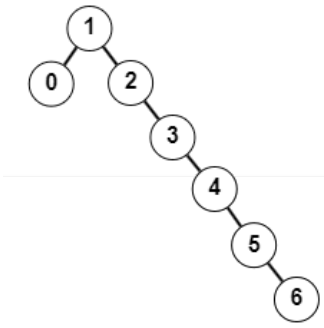

Fig1. Arbre_1

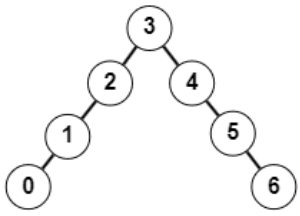

Fig2. Arbre_2

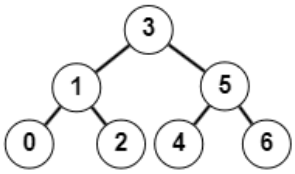

Fig3. Arbre_3

Dans tout l'exercice, nous ferons référence à ces trois arbres binaires de recherche et utiliserons la classe ABR et ses méthodes.

Partie A

1. Un arbre est une structure de données hiérarchique dont chaque élément est un nœud.

   Recopier et compléter le texte ci-dessous en choisissant des expressions parmi `au maximum`, `au minimum`, `exactement`, `feuille`, `racine`, `sous-arbre gauche` et `sous-arbre droit` :

   – Le nœud initial est appelé `...` .
   – Un nœud qui n'a pas de fils est appelé `...`.
   – Un arbre binaire est un arbre dans lequel chaque nœud a `...` deux fils.
   – Un arbre binaire de recherche est un arbre binaire dans lequel tout nœud est associé à une clé qui est :
     • supérieure à chaque clé de tous les nœuds de son `...`
     • inférieure à chaque clé de tous les nœuds de son `...`.

2. Donner dans l'ordre les clés obtenues lors du parcours préfixe de l'arbre no 1.
3. Donner dans l'ordre, les clés obtenues lors du parcours suffixe, également appelé postfixe, de l'arbre no 2.
4. Donner dans l'ordre, les clés obtenues lors du parcours infixe de l'arbre no 3.
5. Recopier et compléter les instructions ci-dessous afin de définir puis de construire, en y insérant les clés dans un ordre correct (il y a plusieurs possibilités, on en demande une) , les trois instances de la classe `ABR` qui correspondent aux trois arbres binaires de recherche représentés plus haut.

```
1  arbre_no1 = ...
2  arbre_no2 = ...
3  arbre_no3 = ...
4  for cle_a_inserer in [..., ..., ..., ..., ..., ..., ...]:
5      arbre_no1....
6  for cle_a_inserer in [..., ..., ..., ..., ..., ..., ...]:
7      arbre_no2....
8  for cle_a_inserer in [..., ..., ..., ..., ..., ..., ...]:
9      arbre_no3....
```

6. Voici le code de la méthode `hauteur` de la classe `ABR` qui renvoie la hauteur d'une instance d'`ABR`:

```
1     def hauteur(self):
2         if self.est_vide() :
3             return -1
4         else :
```

```
5             return 1 + max(self.sag().hauteur(),
6                            self.sad().hauteur())
```

Donner, en utilisant cette méthode, la hauteur des trois instances *arbre_no1*, *arbre_no2* et *arbre_no3* de la classe ABR définies plus haut et qui correspondent aux trois arbres représentés plus haut.

7. Recopier et compléter le code de la méthode `est_presente` ci-dessous qui renvoie *True* si la clé *cle_a_rechercher* est présente dans l'instance d'ABR et *False* sinon :

```
1     def est_present(self, cle_a_rechercher):
2         if self.est_vide() :
3             return ...
4         elif cle_a_rechercher == self.cle() :
5             return ...
6         elif cle_a_rechercher < self.cle() :
7             return ...
8         else :
9             return ...
```

8. Expliquer quelle instruction, parmi les trois ci-dessous, nécessitera le moins d'appels récursifs avant de renvoyer son résultat :

   - `arbre_no1.est_presente(7)`.
   - `arbre_no2.est_presente(7)`.
   - `arbre_no3.est_presente(7)`.

Partie B

9. On rappelle que la fonction `abs(x)` renvoie la valeur absolue de *x*. Par exemple :

```
>>> abs(3)
3
>>> abs(-2)
2
```

On donne la méthode `est_partiellement_equilibre(self)` de la classe ABR. Cette méthode renvoie `True` si l'instance de la classe ABR est l'implémentation d'un arbre partiellement équilibré et `False` sinon :

```
1.    def est_partiellement_equilibre(self) :
2.        if self.est_vide() :
3.            return True
4.        return abs(self.sag().hauteur() -
self.sad().hauteur() ) <= 1)
```

Expliquer ce qu'on appelle ici un arbre *partiellement équilibré*.

Un arbre binaire est *équilibré* s'il est partiellement équilibré et si ses deux sous-arbres, droit et gauche, sont eux-mêmes équilibrés. Un arbre vide est considéré comme équilibré.

10. Justifier que, parmi les trois arbres définis plus haut, deux sont partiellement équilibrés.

11. Justifier que, parmi les trois arbres définis plus haut, un seul est équilibré.

12. Définir et coder la méthode récursive `est_equilibre` de la classe `ABR` qui renvoie `True` si l'instance de la classe `ABR` est l'implémentation d'un arbre équilibré et `False` sinon.

EXERCICE 3 (8 points) ou (6 points en enlevant la partie A)

Cet exercice porte sur les protocoles réseau, les bases de données relationnelles et les requêtes SQL, l'algorithmique et la programmation en Python.

Cet exercice est composé de 3 parties indépendantes.

Partie A : Le réseau informatique d'un hôpital

Dans un hôpital, le service informatique a créé le réseau ci-dessous :

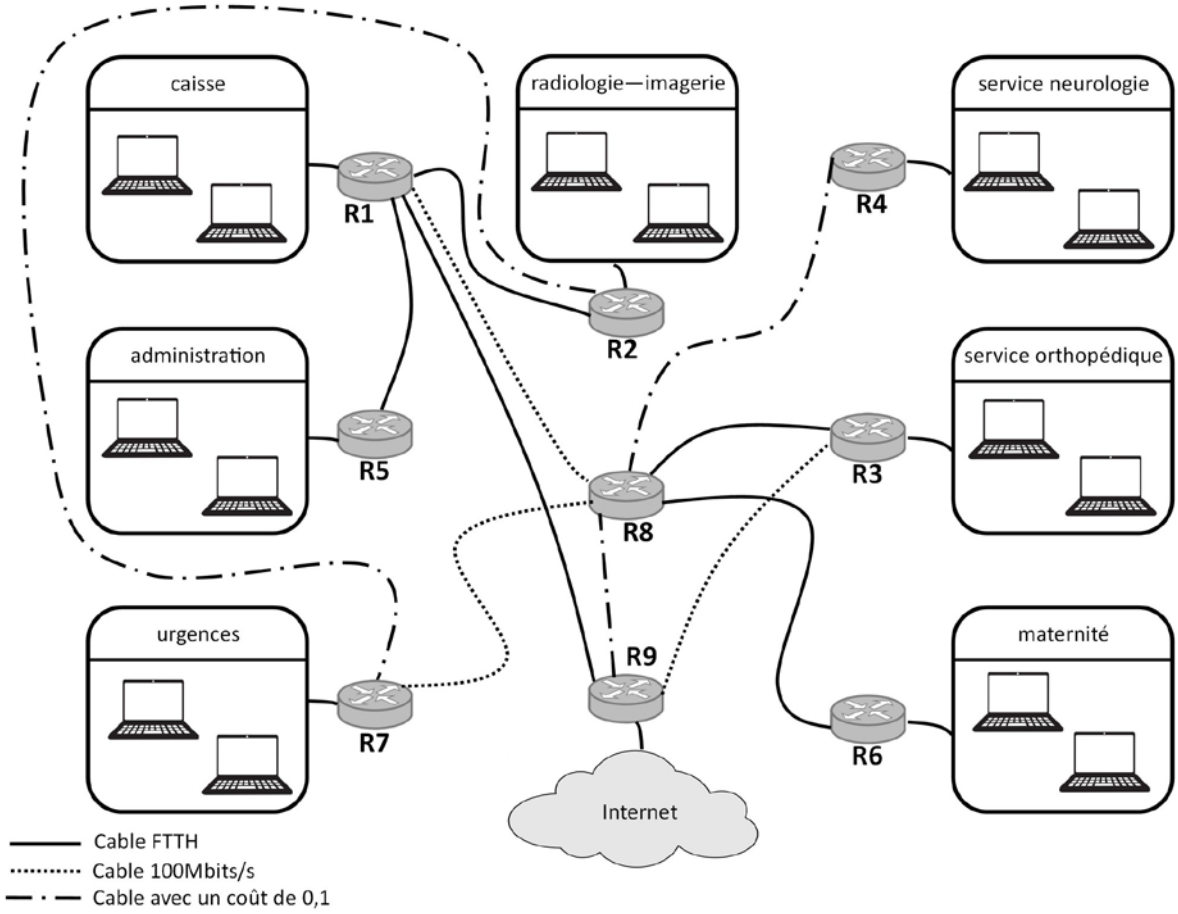

Figure 1. réseau informatique de l'hôpital

Dans un premier temps, on s'intéresse au protocole RIP, qui minimise le nombre de sauts entre routeurs.

1. Donner les chemins possibles d'un paquet de données partant du service de neurologie à destination du service d'imagerie en utilisant le protocole RIP.

2. Recopier et compléter la table des coûts du nœud R2 selon le protocole RIP.

Nœud R2	
Destination	Coût
R1	1
…	…

On s'intéresse maintenant au protocole de routage OSPF. Ce dernier cherche à minimiser la somme des coûts des liaisons entre les routeurs empruntés par un paquet.

Le coût c d'une liaison est donné par :

$$c = \frac{10^8}{d}$$

où d est la bande passante en bit/s de la liaison.

La bande passante des liaisons FTTH (fibre optique : Fiber To The Home) est de 10 Gbit/s et celle des liaisons FastEthernet de 100 Mbit/s.

3. Calculer le coût d'une liaison de communication par la technologie FTTH.

Sur la figure 1, les liaisons sont représentées par un style de trait différents en fonction de leur débit. La légende y est indiquée en bas à gauche.

4. Avec le protocole OSPF, donner le chemin pris par un paquet partant du service de neurologie à destination du service d'imagerie.

Partie B : le dossier médical d'un patient

Lorsqu'un patient se présente dans un hôpital, on lui demande sa carte de sécurité sociale (carte vitale) ainsi qu'une pièce d'identité. En effet, les secrétaires des différents services doivent mettre à jour les données du dossier médical du patient. Ainsi le dossier permet aux médecins de l'hôpital d'avoir accès aux fichiers contenant les divers examens effectués par le patient (les clichés des IRM ou radios, les résultats de ses prises de sang, …).

Pour gérer et partager toutes ces données, le service informatique de l'hôpital a créé une base de données dont le modèle relationnel est donné par le schéma présenté sur la Figure 2. Sur ce schéma, un attribut souligné indique qu'il s'agit d'une clé primaire et un dièse # avant un attribut indique qu'il s'agit d'une clé étrangère.

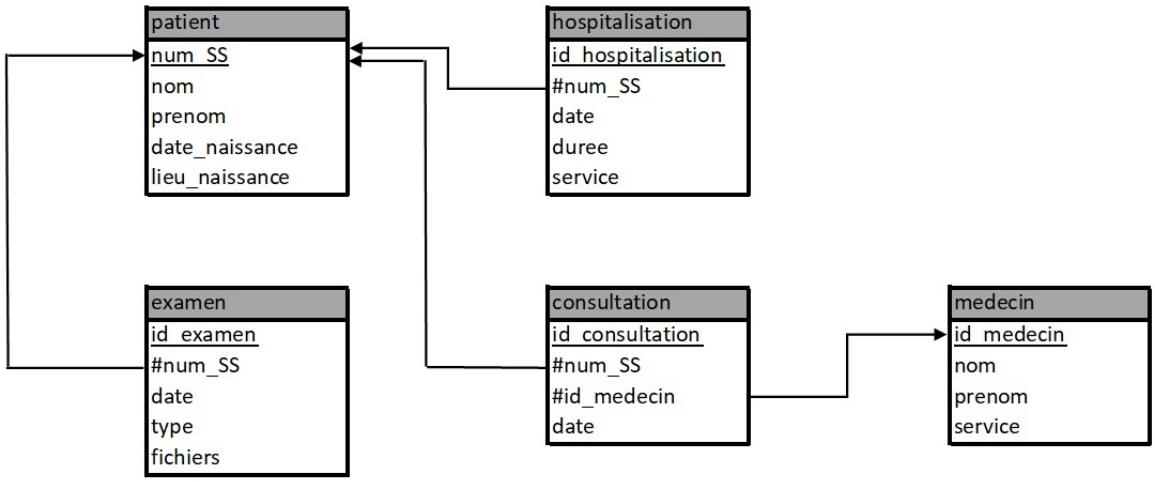

Figure 2. base de données de l'hôpital

Toutes les clés primaires de cette base de données sont de type `INT` (sauf `num_SS` qui est de type `TEXT`), les dates sont de type `DATE` (`JJ/MM/AAAA`) et tous les autres attributs sont de type `TEXT`.

L'énoncé de cet exercice utilise tout ou une partie des mots clefs du langage SQL suivants : `SELECT, DISTINCT, FROM, WHERE, JOIN … ON, UPDATE … SET,DELETE, INSERT INTO … VALUES, LIKE`.

La commande `LIKE 'a%'` permet de rechercher toutes les chaînes de caractères qui commencent par un "a" et `LIKE '%a'` celles qui se terminent par un "a".

Patient				
num_SS	Nom	prenom	date_naissance	lieu_naissance
2 95 07 75 156 189 55	Baujean	Emma	03/07/1995	Paris
1 01 12 69 267 326 21	Tardus	Kylian	16/12/2001	Lyon
2 12 10 33 014 673 82	Delpuis	Sarah	23/10/2012	Bordeaux
1 90 03 37 549 312 43	Montpart	Vincent	30/03/1990	Tours

5. Décrire simplement le résultat obtenu avec la requête SQL ci-dessous :

```
SELECT nom,prenom FROM patient WHERE num_SS LIKE '1%';
```

6. Ecrire une requête SQL permettant d'afficher le numéro de Sécurité Sociale des patients hospitalisés dans le service intitulé `orthopédique` durant l'année 2023.

7. Ecrire une requête SQL permettant d'afficher le type et la date de chaque examen de la patiente Mme Baujean Emma.

8. Ecrire une requête SQL permettant d'afficher le nom et le prénom de tous les patients du médecin M. ARNOS Pierre.

Partie C : la sécurité des mots de passe d'un médecin

Pour utiliser les ordinateurs de cet hôpital, tout le personnel doit saisir son identifiant puis son mot de passe. Pour davantage de sécurité, ce dernier doit être fort et changé régulièrement.

On appelle "mot de passe fort" une chaîne de caractères composée :

- d'au minimum 12 caractères ;
- d'au moins 2 majuscules ;
- d'au moins 2 chiffres ;
- d'au moins 2 symboles parmi **#@!?%<>=€$+-*/&**.

Par exemple, un médecin utilise le mot de passe fort : `@20!HôPiTaL&24#`.

On dispose des méthodes :

- `isalpha` qui permet de tester si un caractère est une lettre ;
- `isupper` qui permet de tester si un caractère est une majuscule ;
- `isdigit` qui permet de tester si un caractère est un chiffre.

```
>>> 'A'.isalpha()
True
>>> 'a'.isupper()
False
>>> '5'.isdigit()
True
```

De plus, on affecte tous les symboles dans une variable globale `liste_symboles` de type `str`:

```
liste_symboles = '#@!?%<>=€$+-*/&'
```

9. On considère une fonction `mdp_fort` qui prend en paramètre une chaîne de caractères `mdp` et qui renvoie `True` si `mdp` est un mot de passe fort et `False` sinon.
   Compléter le script de cette fonction `mdp_fort` en recopiant les lignes 2, 10, 12 et 14 sur votre copie :

```
1  def mdp_fort(mdp):
2      if ... :
3          return False
4      majuscules = 0
5      chiffres = 0
6      symboles = 0
7      for caractere in mdp :
8          if caractere.isupper():
9              majuscules+=1
10         if ... :
11             chiffres+=1
12         if ... :
```

```
13              symboles+=1
14      if ... :
15          return False
16      return True
```

Pour aider le personnel, le service informatique a mis en place une fonction `creation_mdp` permettant de générer aléatoirement un mot de passe.
Elle prend en paramètres quatre entiers de type `int` :

- la longueur `n` du mot de passe ;
- le nombre `nbr_m` de majuscules qu'il doit contenir au minimum ;
- le nombre `nbr_c` de chiffres qu'il doit contenir au minimum ;
- le nombre `nbr_s` de symboles qu'il doit contenir au minimum.

Cette fonction renvoie une chaîne de caractères respectant les conditions précédentes.

10. Compléter le script de cette fonction `creation_mdp` en recopiant les lignes 9 et de 13 à 19 sur votre copie.

```
1   def creation_mdp(n, nbr_m, nbr_c, nbr_s):
2       mdp = ''
3       caracteres='abcdefghijklmnopqrstuvwxyz' + \
4                  'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789' + \
5                  '#@!?%<>=€$+-*/&'
6       majuscules = 0
7       chiffres = 0
8       symboles = 0
9       while ... :
10          # la variable 'c' contient un caractère
11          # choisi aléatoirement dans la variable 'caracteres'
12          c = choice(caracteres)
13          if ... :
14              ...
15          if ... :
16              ...
17          if ... :
18              ...
19          mdp = ...
20      return mdp
```

Si un personnel n'utilise pas le générateur de mot de passe, il a tendance à y insérer des mots de la langue française.
Par exemple, le mot de passe `@20!HôPiTaL&24#` est fort mais on y trouve le mot "hôpital".
Si un mot de passe ne contient pas de mots de 4 lettres ou plus de la langue française ou étrangère, on dit que c'est un "mot de passe extra fort".

On dispose de la variable `dicoFR` (de type `list`) qui contient tous les mots de la langue française.

On dispose également de la fonction `transforme` qui prend en paramètre une chaîne de caractères et qui renvoie une nouvelle chaîne de caractère en transformant toutes les lettres majuscules en minuscules. Par exemple :

```
>>> transforme('@20!HôPiTaL&24#')
@20!hôpital&24#
```

Pour simplifier, on suppose que les mots présents dans un mot de passe sont entiers (pas d'abréviation) et sont séparés par des chiffres ou des symboles.

On considère une fonction `recherche_mot` qui prend en paramètre un mot de passe `mdp` (de type `str`) et qui renvoie des mots présents (de type `str`) dans `mdp`. Exemple :

```
>>> recherche_mot('@20!NeUrO&24#')
['neuro']
>>> recherche_mot('Chef!14NeuroA@85!')
['Chef','Neuro']
```

11. Compléter le script de cette fonction `recherche_mot` en recopiant les lignes 5, 6, 15, 16, 18 sur votre copie.

```
1  def recherche_mot(mdp):
2      mot = transforme(mdp)
3      trouve = []
4      i = 0
5      while ... :
6          if ... :  # si le caractère est un chiffre
7              i = i+1
8          elif mot[i] in liste_symboles:
9              i = i+1
10         else:
11            # si le caractère est une lettre, on prend les
12            # lettres qui la suivent jusqu'au moment où
13            # on trouve un chiffre ou un symbole
14             chaine = ''
15             while ... :
16                 chaine = ...
17                 i = i+1
18             ...
19     return trouve
```

12. Ecrire une fonction `mdp_extra_fort` qui prend en paramètre une chaîne de caractères `mdp` et qui renvoie `True` si `mdp` est un mot de passe extra fort et `False` sinon.