INFO-H-100 – Informatique – Prof. Th. Massart
1ère année du grade de Bachelier en Sciences de l'Ingénieur
Examen de juin – ***Solutions***

Prénom : Nom : Matricule :

Remarques préliminaires

- On vous demande de répondre à **chaque question sur une feuille séparée**.
- N'oubliez pas d'inscrire votre nom, prénom et numéro de matricule sur chaque feuille.
- Vous disposez de trois heures et vous ne pouvez pas utiliser de notes.
- Vous pouvez utiliser le verso des feuilles pour répondre.
- Si du code vous est demandé,
  - la réponse à la question doit comprendre le code *Python* structuré et conforme aux règles de bonne pratique et conventions,
  - sauf mention contraire, vous ne pouvez utiliser aucune fonction de librairies (pas d'`import`)
  - veillez à découper votre réponse en fonctions de manière pertinente
  - veillez à utiliser des structures de données appropriées.

Question 1 - Théorie (5 points)

Pour chacun des codes suivants, et en supposant que `s` est une chaine de n caractères, que `mat` est une matrice carrée de dimension $n \times n$ et que `dico` est un dictionnaire dont les clés et valeurs sont respectivement des chaînes de caractères de taille maximum 100 et des ensembles de clés existentes.

- expliquez ce qu'il fait et ce qu'il imprime
- donnez l'état du programme lors de chaque `print` grâce à des diagrammes d'état (comme fait au cours).
- donnez la complexité moyenne et maximale ($\mathcal{O}()$) de la **fonction** `foo_`i, $i = 0..2$ ainsi q) en temps d'exécution. Précisez bien les paramètres utilisés pour exprimer la complexité et justifiez bien vos réponses (le résultat seul ne suffit pas pour obtenir des points).

1.
```python
def foo_0(dico):
    res = set({})
    for elem in dico:
        c = dico[elem] - {elem}
        for a in c:
            if elem in dico[a]:
                res.add(elem)
    return res

dico = { "Jean"       : {"Jean", "Catherine"} ,
         "Germaine"   : {"Catherine"},
         "Catherine"  : {"Jean", "Germaine"} ,
         "Luc"        : set({}),
         "Michel"     : {"Luc"}}
print(foo_0(dico))
```

Solution :

- la fonction `foo_0` renvoie la liste des personnes qui ont un contact dans leur liste qui lui aussi connait cette personne (p connait q et vice versa).
  Seront imprimés :
  `{'Jean', 'Catherine', 'Germaine'}`
- diagrammes d'état : voir pythontutor.
- complexités moyennes et maximales :

```
def foo_0(dico):          # O(n^2)   O(n^3)
    res = set({})                 # O(1)     O(1)
    for elem in dico:             # (répétition O(n)) O(n^2)  O(n^3)
        c = dico[elem] - {elem}       # O(1)     O(n)
        for a in c:                   # (répétition O(n)) O(n)      O(n^2)
            if elem in dico[a]:       #   (test O(1) O(n)) O(1)     O(n)
                res.add(elem)         #   O(1)     O(n)
    return res                    # O(1) O(1)
```

Hypothèse : `n` éléments ayant chacun $O(n)$ contacts
Justification : voir cours

2.
```
def foo_1(mat):      # O(n^3)
    n = len(mat)                                    # O(1)
    for i in range(n,1,-1):                         # O(n^3) (répét: O(n))
        for j in range(i-1):                          # O(n^2) (répét: O(n))
            if max(mat[j]) > max(mat[j+1]):             # O(n)   (test en O(n))
                mat[j], mat[j+1] = mat[j+1], mat[j]       # O(1)
        print(mat)                                    # O(n^2)
```

- la fonction `foo_1` effectue un tri par échange (bulle) par ordre croissant des lignes de la matrice sur base de la valeur maximale de chaque ligne.
  Seront imprimé :

```
[[3, 4, 7], [1, 2, 6], [5, 0, 8]]
[[1, 2, 6], [3, 4, 7], [5, 0, 8]]
[[1, 2, 6], [3, 4, 7], [5, 0, 8]]
```

- diagrammes d'état : voir pythontutor.
- complexités moyennes et maximales (elles sont similaires) : $O(n^3)$
  Justification : voir cours (complexité tri bulle + le max d'une liste à n éléments est en $O(n)$

3.
```
def foo_2(s):                                  # O(n^2) O(n^2)
    res = len(s) == 1 or len(s) == 0             # O(1)   O(1)
    if not res:                                  # O(n) O(n) + appel récursif
        res = s[0] == s[-1] and foo_2(s[1:-1])     # O(n) O(n) + appel récursif
    print(s)                                     # O(n)
    return res

s = "hannah"
print(foo_2(s))
```

Solution :

- la fonction `foo_2` renvoie vrai si le mot introduit est un palindrome.
  Seront imprimés :

```
nn
anna
hannah
True
```

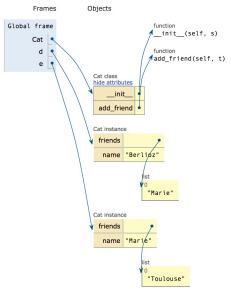
- diagrammes d'état : voir pythontutor.
- complexités moyennes et maximales en $O(n^2)$: $n/2$ appels et chaque instance en $O(n)$ (plus précisément $1 + 3 + 5... + n$).
  Notons que sans les print, la complexité serait en $O(n)$.

4.
```
class Cat:

    def __init__(self, s):
        self.name = s
        self.friends = []

    def add_friend(self, t):
        self.friends.append(t)

d = Cat('Berlioz')
e = Cat('Marie')
d.add_friend('Marie')
e.add_friend('Toulouse')
print(d.friends)
print(e.friends)
```

Solution :
Imprime :

```
['Marie']
['Toulouse']
```

Diagrammes d'état :



Ce code crée une nouvelle classe d'objet `Cat` ainsi que 2 objets de cette classe. Il appelle pour chacun, une méthode qui lui ajoute un ami dans son attribut `friends`.

Question 2 - Théorie (2 points)

En vous aidant des exemples de la question 1, expliquez **en détails** les mécanismes suivants, et montrez où ils sont utilisés dans les codes de la question 1 :

1. Le hashing
2. La récursivité
3. le pack / unpack

1. Le hashing
   (a) * Principe de la table de hashing (0.4)
   (b) * Collision / rehasing (0.4)
   (c) + Avantages : accès moyen en O(1) (+ 0.2)
2. La récursivité
   (a) * Principe : fonction que s'appelle elle même directement ou indirectement via des appels à d'autres fonctions (0.3)
   (b) * la gestion de la récursivité par python utilise le stack runtime : à chaque appel, une nouvelle frame est créée, en fin d'appel, on retourne à la frame précédante qui a retenu le contexte local (0.4)
3. le pack / unpack
   (a) * Pack : le fait de créer un tuple de valeurs (0.3)
   (b) * Unpack : le fait de remettre chaque composante d'un tuple dans une variable (0.2)
   (c) + utilisé lors d'assignation et dans les passages de paramètres lors d'appel de fonctions (+ 0.2)

Question 3 - Tri (5 points)

Nous recevons toujours plus d'emails aujourd'hui. Il est donc important de mettre en place des techniques afin de trier au maximum les emails en fonction de leur importance.

Nous pouvons considérer que la boîte de réception est une liste (nommée `mails`) contenant des emails. Chaque email est représenté par une sous-liste contenant 2 éléments :

- Le nom de l'envoyeur
- L'ancienneté de l'email (en nombre d'heures)

Exemple :
```
mails = [["Jean Dejaegere", 13], ["Chloé Colomer", 2], ["Yves Lonchamps", 16]]
```

Afin de savoir si le mail est une publicité ou non, les noms des envoyeurs de publicité ont été répertoriés dans une liste (`pub`).
Exemple :
```
pub = ["Jonathan Dupont", "Gilles Silovy", "Jean Dejaegere", "Jean-Philippe Rosenfeld"]
```

Afin de savoir combien de mails ont été envoyés par chaque envoyeur, un dictionnaire dico a été créé ayant pour clef le nom de l'envoyeur et comme valeur le nombre de mails envoyés par l'envoyeur.
Exemple :
`dico={"Josianne Dubois" : 12, "Chloé Colomer" : 31, "Archibald Hebborn" : 18}`

Nous vous demandons d'écrire la fonction `tri_mails(mails, pub, dico)` qui trie les emails selon les règles suivantes (par ordre décroissant d'importance) :

- Les mails non-publicitaires doivent se retrouver au début de la liste
- Deux mails appartenant à la même catégorie (publicités ou non-publicité) seront d'abord départagés selon leur ancienneté
- Si les deux premières conditions ne sont pas suffisantes pour départager deux emails, celui dont l'envoyeur a envoyé le plus d'emails doit être placé en premier

Nous vous demandons d'effectuer ce tri soit par insertion soit par sélection.

Solution.

```
def tri_mails(mails,pub,dico):
    n = len(mails)
    for i in range(n - 1):
        i_min = i
        for j in range(i + 1, n):
            if vient_avant(mails[j], mails[i_min], pub, dico):
                i_min = j
        mails[i], mails[i_min] = mails[i_min], mails[i]

def vient_avant(m1, m2, pub, dico):
    if (m1[0] in pub) != (m2[0] in pub):
        res = m2[0] in pub
    elif m1[1] != m2[1]:
        res = m1[1] > m2[1]
    else:
        res = dico(m1[0]) > dico(m2[0])
    return res
```

Question 4 - Fichiers (4 points)

Votre système d'exploitation possède un fichier "/etc/passwd" contenant la liste de tous les utilisateurs. Voici à quoi ressemble chaque ligne du fichier :

user :hashed_password\n

Où "user" est le nom d'utilisateur et "hashed_password" le hash du mot de passe de l'utilisateur.

Le hash d'un mot de passe X est l'entier qui est renvoyé par la fonction :

res = my_hash(X)

Par exemple, si l'utilisateur "Gilles" possède le mot de passe "vive_INFO", et que my_hash("vive_INFO") renvoie 123456, alors le fichier "/etc/passwd" contiendra la ligne suivante :

Gilles :123456\n

Vous savez que les autres utilisateurs de votre ordinateur ne sont pas tous très prudents et que certains auront probablement choisi comme mot de passe un mot du dictionnaire. À l'aide du fichier "words.txt" contenant tous les mots du dictionnaire, et du fichier "/etc/passwd", retrouvez le nom d'utilisateur et le mot de passe de ces utilisateurs et écrivez ces résultats dans le fichier "incautious_users.txt".

Pour cette question, nous supposerons que chaque mot du dictionnaire a un hash différent.

Solution.

```
# On suppose qu'il n'y a pas deux mots de même hash
def hashed_dico_words(word_file):
    """
    Renvoie un dictionnaire (hash -> mot du dico ayant cet hash)
    """
    words = {}
    for word in open(word_file):
        word = word.strip()
        words[my_hash(word)] = word
    return words

def find_incautious(word_file, passwd_file, output_file):
```

```
    words = hashed_dico_words(word_file)
    users = [l.strip().split(":") for l in open(passwd_file).readlines()]
    out = open(output_file, "w")

    for user, hsh in users:    # O(N)
        hsh = int(hsh)
        if hsh in words:       # O(1)
            out.write(user + ":" + words[hsh] + "\n")
    out.close()
```

Question 5 - Récursivité (4 points)

Sur l'écran d'accueil de votre smartphone, vous pouvez placer des dossiers (de raccourcis) d'applications. Imaginons que ces dossiers puissent contenir des sous-dossiers.

Une application sera représentée par un tuple `(nom, identifiant)`, l'identifiant étant un entier unique définissant le raccourci vers l'application. Un dossier d'applications sera représenté par un tuple

`(nom, liste d'applications ou de sous-dossiers).`

Ecrivez une fonction `find_apps(name, dossier_apps)` qui reçoit un mot et un dossier d'applications et qui renvoie une liste de toutes les applications (liste de tuples) dont le nom contient le mot donné.

Par exemple :

```
dossier_apps = ("Mes Applications", [
    ("Facebook", 123),
    ("WhatApp", 37),
    ("Google Apps", [
        ("Google", 1),
        ("Google Maps", 28),
        ("Youtube", 44),
        ("Google Drive", 17)
        ]
    ),
    ("Utile", [
        ("Horloge", 56),
        ("Calculatrice", 52)
        ]
    ),
    ("Peu utile", [
        ("Météo", 107),
        ("Téléchargements", 8),
        ("Encore Google", [
            ("Google Play Films", 174),
            ("Google Play Musique", 66)
            ]
        )
        ]
    )
    ]
)

##On recherche toutes les applis de Google
>>> find_apps("Google", dossier_apps)
[('Google', 1), ('Google Maps', 28), ('Google Drive', 17), ('Google Play Films', 174), ('Google Play Musique', 66)]
```

Solution.

```
def find_apps(name, dossier_apps):
    res = []
    for c in dossier_apps[1]:
        if type(c[1]) == list:
            res.extend(find_apps(name, c))
        elif name in c[0]:
            res.append(c)
    return res
```