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.

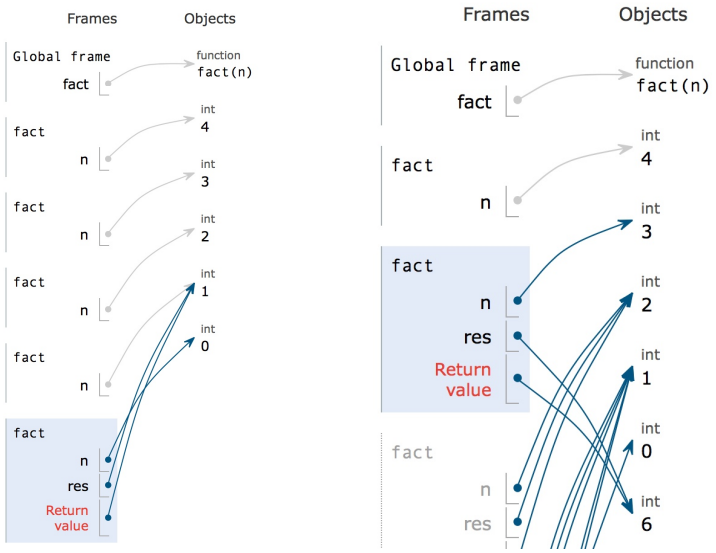
Question 1 - Fonctionnement de Python (2 points)

Expliquer brièvement comment s'effectue la gestion de la mémoire lors de l'exécution d'un programme Python qui appelle une fonction récursive. Pour vous aider dans vos explications, utilisez un exemple de code fonction Python récursive qui calcule la factorielle d'un naturel n reçu en paramètre. Pour illuster vos propos utilisez des diagrammes d'état expliquant comment évolue la mémoire lors de l'exécution. Dans vos diagrammes d'états, veillez à donner complètement les objets existant en mémoire.

Solution. Explications : voir syllabus section 11.5.2.

```
def fact(n):
  """Renvoie la factorielle de n (méthode récursine)."""
  if n == 0:
    res = 1
  else:
    res = n*fact(n-1)
  return res
print(fact(4))
```

(Premier diagramme d'état) état de la mémoire au 5ème appel récursif de `fact` et (second diagramme) au retour pour $n = 3$:



Question 2 - Code Python (5 points)

Pour chacun des codes suivants,

- donnez ce qu'il imprime (avec les appels qui sont donnés) et expliquez de façon générale ce que fait la fonction (si elle résout un problème, lequel ...) ;
- donnez l'état du programme lors de chaque (groupe de) `print` grâce à des diagrammes d'état (comme fait au cours) ;
- sachant que les types des paramètres sont,
  - n : nombre naturel,
  - x : liste d'entiers que vous pouvez supposer de taille n,
  - s et t : strings que vous pouvez supposer de taille n contenant uniquement des caractères ASCII (128 caractères différents possibles),

  donnez la complexité moyenne et maximale ($\mathcal{O}()$) de la **fonction** `foo_`i ($i = 1..5$) 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).

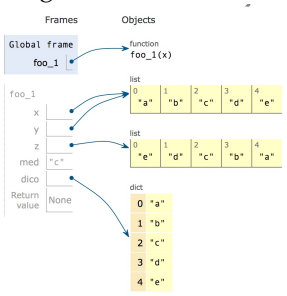
Notez bien : pour expliquer ce que le code fait et donner la complexité, vous pouvez annoter les codes fournis (compléter les docstrings, commenter, ...)

```python
def foo_1(x):                                   # O(1)
    y = x                                       # O(1)   O(1)
    z = x[::-1]                                 # O(n)   O(n)
    med = x[len(x)//2]                          # O(1)   O(1)
    dico = dict(zip(list(range(len(x))),x))     # O(n)   O(n^2)
    print(x)                                    # O(n)   O(n)
    print(y)                                    # O(n)   O(n)
    print(z)                                    # O(n)   O(n)
    print(med)                                  # O(1)   O(1)
    print(dico)                                 # O(n)   O(n)
    print(x is y)                               # O(1)   O(1)
    print(x is z)                               # O(1)   O(1)
foo_1(list("abcde"))
```

Solution :
Imprime :

```
['a', 'b', 'c', 'd', 'e']
['a', 'b', 'c', 'd', 'e']
['e', 'd', 'c', 'b', 'a']
c
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
True
False
```

Ce code ne semble pas résoudre de problème spécifique.
Diagramme d'état :



Complexité de la fonction `foo_1` : moyenne = $O(n)$; max = $O(n^2)$ (voir code). En effet, toutes les instructions qui demandent de parcourir tous les éléments d'une liste sont en $O(n)$. La création du dictionnaire est en moyenne en $O(n)$; au maximul en $O(n^2)$ si on suppose que chaque insertion prend un temps en $O(k)$ où k est le nombre d'éléments insérer jusqu'à présent (à la fin $k = n$).

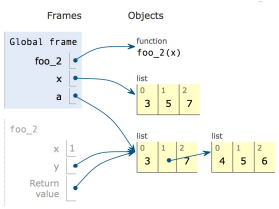
```python
def foo_2(x):           # O(1)
    y = x[:]            # O(n)
    x = 1               # O(1)
    y[x] = [4,5,6]      # O(1)
    print(y)            # O(n)
    return y            # O(1)

x = [3, 5, 7]
a = foo_2(x)
print(x)
```

```
print(a)
print(y)
```

Solution :
Imprime :

```
[3, [4, 5, 6], 7]
[3, 5, 7]
[3, [4, 5, 6], 7]
Traceback ...
    print(y)
NameError: name 'y' is not defined
```

Diagramme d'état :



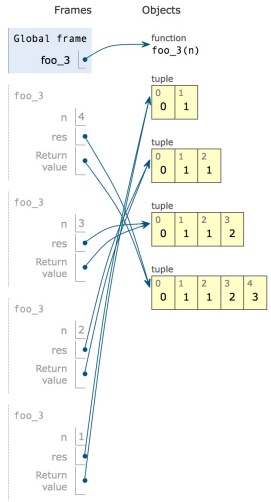
Ce code ne semble pas résoudre de problème spécifique.
Complexité de la fonction `foo_2` : max = moyenne : $O(n)$ (voir code).

```
def foo_3(n):      # O(n^2)
    """


    """
    if n == 1:                          #  O(n) + complexité de l'exéc. récursive
        res = (0,1)                         # O(1)
    else:                               # O(n) + complexité de l'exéc. récursive
        res = foo_3(n-1)                    # complexité de l'exécution récursive
        res = res + (res[-2] + res[-1], )   # O(n)
    print(res)                              # O(n)
    return res                              # O(1)

print(foo_3(4))
```

Solution :
Imprime :

```
(0, 1)
(0, 1, 1)
(0, 1, 1, 2)
(0, 1, 1, 2, 3)
(0, 1, 1, 2, 3)
```

Diagrammes d'état :



Ce code construit un tuple avec les nombres de Fibonacci 0 à n.
Complexité de `foo_3` max = moyenne = $O(n^2)$ (voir code). En effet, au total pour n on aura une complexité en $O(n + (n-1) + (n-2) + ... + 1) = O(n^2)$. Le `print(res)` a une complexité qui dépend de la taille du tuple `res` (donc de `n`).

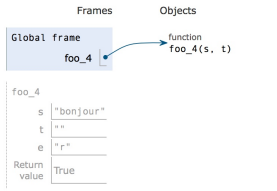
```
def foo_4(s,t):                         # O(n^2)
    for e in s:                           # O(n^2)
        t = t[:t.find(e)]+t[t.find(e)+1:]       # O(n)
```

```
    print()                                  # O(1)
    return len(t) == 0                       # O(1)

print(foo_4('bonjour','nuroobj'))
```

Solution :
Imprime :

```
True
```

Diagrammes d'état :



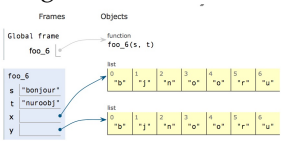
L'autre diagramme d'état est moins intéressant (voir pythontutor pour l'obtenir).
Ce code teste si deux strings sont anagrammes entre eux.
Complexité de `foo_4` max = moyenne = $O(n^2)$ (voir code). En effet, au total pour n on aura une complexité en $O(n)$ pour un find ou (au maximum) la concaténation des deux parties de t.

```
def foo_5(s,t):                      # O(n log n)
    x = list(s)                          # O(n)
    y = [e for e in t]                   # O(n)
    x.sort()                             # O(n log n)
    y.sort()                             # O(n log n)
    print()                              # O(1)
    return x == y                        # O(n)

print(foo_5('bonjour','nuroobj'))
```

Solution :
Imprime :

```
True
```

Diagrammes d'état :



L'autre diagramme d'état est moins intéressant (voir pythontutor pour l'obtenir).
Ce code teste si deux strings sont anagrammes entre eux.
Complexité de `foo_5` max = moyenne = $O(nlogn)$ (voir code). En effet, au total pour n on aura une complexité en $O(n)$ pour la construction des listes et $O(nlogn)$ pour les tris.

Question 3 - Manipulation de fichiers et de données (5 points)

Considérons le fichier d'une société représentant (de facon extrêmement simplifiée) les achats de clients. Les informations sont mémorisées dans un fichier texte. Chaque ligne du fichier porte sur un achat effectué par le client correspondant. Le nom des clients est toujours formé d'un seul mot et est séparé du montant de l'achat par un espace. Le fichier est trié sur le nom des clients. Par exemple le fichier suivant indique que le Client2 a effectué trois achats différents pour des montants respectifs de 300, 120 et 440 €.

Client1 140
Client1 200
Client2 300
Client2 120
Client2 440
Client3 500
Client4 210
Client4 100
Client5 520
Client6 200

Ecrire la fonction `pack(fNameIn, fNameOut)` qui écrit dans le fichier `fNameOut` les achats regroupés de façon à ce que les clients n'apparaissent plus qu'une fois (dans le même ordre) avec le montant cumulé (la somme) de leurs achats.

Client1 340
Client2 860
Client3 500
Client4 310
Client5 520
Client6 200

Vous avez la garantie, sans devoir le vérifier, que le fichier `fNameIn` respecte le format présenté et qu'il contient au moins un achat.

Solution.

La solution suivante est simple et efficace. On profite de ce que les clients sont regroupés par noms pour traiter une ligne à la fois séquentiellement sans mémoriser le contenu complet du fichier dans une structure intermédiaire.

```
def pack(fNameIn, fNameOut):
    fout = open(fNameOut, "w")
    prec = ""
    for line in open(fNameIn):
        actuel, nvtMontant = line.split()
        if actuel == prec:                # Est-ce le même que le précédent ?
            montant += int(nvtMontant)    # Si oui, montant cumulé
        else:
            if prec != "":                # Sinon, on écrit le résultat sauf la 1ère fois
                fout.write(prec + ' ' + str(montant) + '\n')
            prec, montant = actuel, int(nvtMontant)  # On recommence avec le suivant
    fout.write(prec + ' ' + str(montant) + '\n')     # Pas oublier le dernier...
    fout.close()
```

La version alternative suivante a comme avantage de séparer la lecture du fichier, le traitement des clients et l'écriture dans le fichier résultat en 3 étapes. Cela est obtenu au prix d'une liste intermédiaire qui mémorise le contenu complet du fichier (probablement coûteux si le fichier est trop grand).

```
def lecture(fNameIn):
    ls = []
    for line in open(fNameIn):
        client, montant = line.split()
        ls.append([client, int(montant)]) # le montant est un int
    return ls

## Les clients sont regroupés dans la même liste en considérant chaque client
## sauf le premier (indice origine) et si c'est le même que celui déjà considéré
##  (à l'indice destination), on y accumule les montants.
## Quand on trouve un autre client, destination avance.
## A la fin, on raccourcit la liste au niveau du dernier client (dernière valeure de destination)
def list_pack(ls):
    orig = 1
    dest = 0
    while orig < len(ls):
        if ls[dest][0] == ls[orig][0]:
            ls[dest][1] += ls[orig][1]
        else:
            dest += 1
            ls[dest] = ls[orig]
        orig += 1
    del ls[dest + 1: ]
    return ls

def ecriture(ls, fNameOut):
    f = open(fNameOut, "w")
    for x in ls:
        f.write(x[0] + ' ' + str(x[1]) + '\n')
    f.close()

def pack(fNameIn, fNameOut):
    ls = lecture(fNameIn)
    list_pack(ls)
    ecriture(ls, fNameOut)
```

Les deux solutions précédentes sont de complexité O(n) et sont, en ce sens, optimales.

Une troisième solution utilise un dictionnaire et ne profite pas de l'information fournie dans l'énoncé qui dit que le fichier est trié sur le nom des clients. Elle offre le bénéfice d'être (peut-être ?) plus simple. Malheureusement elle cumule le défaut de la deuxième solution, qui nécessite une mémorisation complète du fichier, avec celui d'être de complexité en meilleur cas O(n * log n) (à cause du tri) voire plus grave O(n ** 2) à cause du

dictionnaire. Donc, pour un gros fichier de clients, cette solution consommerait inutilement beaucoup de place mémoire et serait potentiellement plus lente !

```
def lecture(fNameIn):
    dico = {}           # Un dico {clients -> montant achats}
    for line in open(fNameIn):
        nom, montant = line.split()
        dico[nom] = dico.get(nom, 0) + int(montant)
    return dico

def ecriture(ls, fNameOut):
    f = open(fNameOut, 'w')
    for line in ls:
        f.write(line[0] + ' ' + str(line[1]) + '\n')
    f.close()

def pack(fNameIn, fNameOut):
    dico = lecture(fNameIn)
    ecriture(sorted(dico.items()), fNameOut)
```

Question 4 - Tri (4 points)

Les secteurs alimentaires du Reblochon, du pain et des filets de maquereaux se disputent la place de l'aliment le plus nutritif. De nombreuses études ont été conduites pour classer les 3 secteurs alimentaires et mènent à des résultats tout à fait différents qui sont rangés et numérotés dans la liste `sondages`.

Par exemple :

```
sondages = [['1','LeReblochon', 'LePain', 'LeMaquereau'],
            ['2','LeMaquereau', 'LeReblochon','LePain'],
            ['3','LeReblochon', 'LeMaquereau', 'LePain'],
            ['4','LePain', 'LeMaquereau', 'LeReblochon'],
            ['5','LeReblochon', 'LeMaquereau', 'LePain'],
            ['6','LePain', 'LeReblochon', 'LeMaquereau'],
            ['7','LeMaquereau', 'LeReblochon','LePain'],
            ['8','LeMaquereau', 'LePain', 'LeReblochon']]
```

(Le nombre de sondages inclus dans `sondages` peut être très élevé).

Le secteur des filets de maquereaux décide d'analyser tous ces sondages et vous demande d'écrire la fonction `triSondage(sondages, partenaire, concurrent)`. Cette fonction doit classer les sondages dans l'ordre qui met le plus proche de la première place, le `partenaire` (par exemple `'LeMaquereau'`). En cas d'ex-aequo, l'avantage est donné au sondage qui met `concurrent` (par exemple `'LePain'`) à la plus mauvaise place possible.

En considérant que les deux variables `partenaire` et `concurrent` sont données comme suit :

```
triSondage(sondages, 'LeMaquereau', 'LePain')
```

La liste `sondages` sera alors modifiée de la manière suivante :

```
sondages = [['2', 'LeMaquereau', 'LeReblochon', 'LePain'],
            ['7', 'LeMaquereau', 'LeReblochon', 'LePain'],
            ['8', 'LeMaquereau', 'LePain', 'LeReblochon'],
            ['3', 'LeReblochon', 'LeMaquereau', 'LePain'],
            ['5', 'LeReblochon', 'LeMaquereau', 'LePain'],
            ['4', 'LePain', 'LeMaquereau', 'LeReblochon'],
            ['1', 'LeReblochon', 'LePain', 'LeMaquereau'],
            ['6', 'LePain', 'LeReblochon', 'LeMaquereau']]
```

Solution.

```
def conditionTri(el1, el2, partenaire, concurrent):
    return el1.index(partenaire) < el2.index(partenaire) or
            (el1.index(partenaire) == el2.index(partenaire) and el1.index(concurrent) > el2.index(concurrent))


def triSondage(s, partenaire, concurrent):
    for k in range(1, len(s)):
        temp = s[k]
        j = k
        while j > 0 and conditionTri(temp, s[j - 1], partenaire, concurrent):
            s[j] = s[j - 1]
            j -= 1
        s[j] = temp

triSondage(sondages, 'LeMaquereau', 'LePain')
```

Question 5 - Récursivité (4 points)

Pour le maintien de l'arbre généalogique d'une famille, chaque personne est représentée par un tuple formé de son prénom et de la liste (éventuellement vide) de ses enfants. Contrairement à des pratiques courantes, les familles qui nous intéressent sont caractérisées par le fait de ne jamais donner deux fois le même prénom à deux personnes différentes. Le prénom suffit donc à identifier une personne.
Voici un exemple d'arbre généalogique :

```
arbre = ("Lucy", [
    ("Bill", [
        ("Mary", []),
        ("Fred", [("Paul", []), ("John", [])]),
        ("Jean", [])
        ]),
    ("Rick", []),
    ("Jack", [
        ("Jane", []),
        ("Bob", [("May", [])])
        ])
    ])
```

La figure 1 illustre cet arbre généalogique.

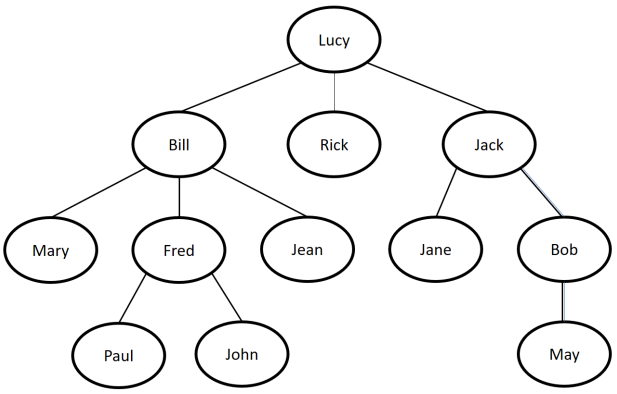


FIGURE 1 – Arbre généalogique

Ecrivez une fonction qui va chercher dans un arbre généalogique, tous les enfants d'un parent donné. Cette fonction a 2 pramètres :

- `arbre` : la liste qui représente l'arbre généalogique
- `parent` : le prénom du parent dont il faut trouver les enfants (un string)

La fonction renverra la liste (éventuellement vide) des prénoms des enfants de `parent`. Si le `parent` n'apparaît pas dans l'arbre, la fonction renvoie `None`.

Par exemple, le résultat du bout de code suivant :

```
for parent in ["Lucy", "Fred", "Rick", "Al"]:
    print("Les enfants de", parent, "sont", enfants(arbre, parent))
```

Est :

```
Les enfants de Lucy sont ['Bill', 'Rick', 'Jack']
Les enfants de Fred sont ['Paul', 'John']
Les enfants de Rick sont []
Les enfants de Al sont None
```

Solution.

```
def enfants(arbre, parent):
    result = None
    if arbre[0] == parent:
        result = [x[0] for x in arbre[1]]
    k = 0
    while k < len(arbre[1]) and result == None:
        result = enfants(arbre[1][k], parent)
        k += 1
    return result
```