INFO-H-100 – Informatique – Prof. Th. Massart
1ère année du grade de Bachelier en Sciences de l'Ingénieur
Examen de janvier – ***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 le mode de fonctionnement d'un compilateur et d'un interpréteur. Python de base est-il interprété ou compilé ? Expliquez quels en sont les avantages et les inconvénients pour ce langage.

Solution. → Voir syllabus et cours. : Il fallait expliquer les 6 phases de la compilation, le cycle de l'interpréteur, dire que de base Python est un langage interprété, ce qui induit une série d'avantages et inconvénients (à détailler).

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, ...)

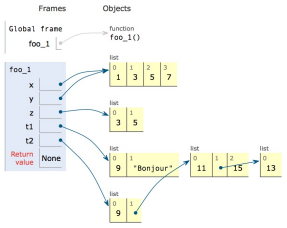
```python
def foo_1():                        # O(1)
    x = y = [1,3,5,7]               # O(1)
    z = x[1:3]                      # O(1)
    t1 = [9, [11, [13], 15]]        # O(1)
    t2 = t1[:]                      # O(1)
    t1[1] = "Bonjour"               # O(1)
    print(x)                        # O(1)
    print(y)                        # O(1)
    print(t1)                       # O(1)
    print(t2)                       # O(1)
    print(x is y)                   # O(1)
```

```
    print(x is z)                # O(1)
    print(t1 is t2)              # O(1)
foo_1()
```

Solution :
Imprime :

```
[1, 3, 5, 7]
[1, 3, 5, 7]
[9, 'Bonjour']
[9, [11, [13], 15]]
True
False
False
```

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



Complexité de la fonction `foo_1` : max = moyenne = $O(1)$ (voir code). En effet, cette fonction n'a pas de paramètre et effectue à chaque appel exacetement les même instructions. Le temps mis poir chaque instruction est donc bien constant.

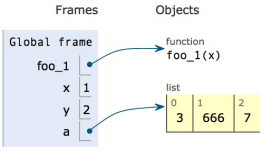
```
def foo_2(x):                # O(1)
    y = x                    # O(1)
    x = 1                    # O(1)
    y[x] = 666               # O(1)
    return y                 # O(1)

x = 1
y = 2
a = foo_2([3,5,7])
print(x)
print(y)
print(a)
```

Solution :
Imprime :

```
1
2
[3, 666, 7]
```

Diagramme d'état :



Ce code ne semble pas résoudre de problème spécifique.
Complexité de la fonction `foo_2` : max = moyenne : $O(1)$ (voir code). Justification supplémentaire : idem `foo_1`.

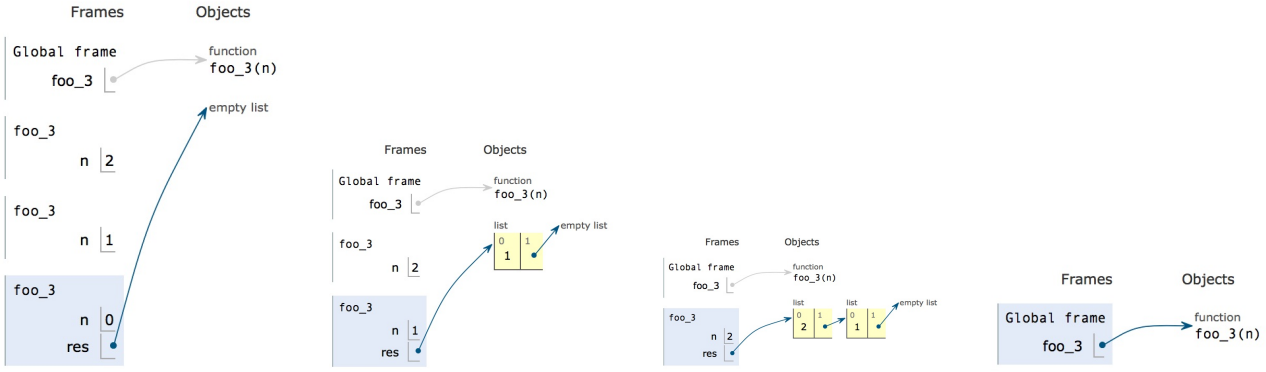
```
def foo_3(n):                # O(n) + complexité de de l'exécution de  foo_3(n-1) (donne O(n^2) voir explications)
    if n == 0:               #   O(1) (cond O(1)) + complexité de de l'exécution de  foo_3(n-1)
        res = []             #      O(1)
    else:                    #   O(1) + complexité de de l'exécution de  foo_3(n-1)
        res = [n, foo_3(n-1)]#      O(1) + la complexité de l'exécution de  foo_3(n-1)
    print(res)               #   O(n)
    return res               #   O(1)

print(foo_3(2))
```

Solution :
Imprime :

```
[]
[1, []]
[2, [1, []]]
[2, [1, []]]
```

Diagrammes d'état :



Ce code construit une imbrication de listes : cas de base $\ell_0 = []$, cas $\ell_i = [i, l_{i-1}]$
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 (ici c'est plutôt le nombre d'imbrication) de la liste `res` (donc de `n`).

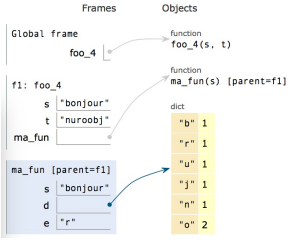
```
def foo_4(s,t):    # complexité  moyenne:   O(n)  maximale: O(n)      (Hypothese : n = taille de s et t)
    def ma_fun(s):                          #   O(n)          O(n)
        d = {}                              #      O(1)         O(1)
        for e in s:                         #      O(n)         O(n)           (nb d'iteration: n)
            d.setdefault(e,0)               #        O(1)         O(1)
            d[e] += 1                       #        O(1)         O(1)
        print(d)                            #      O(n)         O(1)
        return d                            #      O(1)         O(1)
    return ma_fun(s) == ma_fun(t)           #   O(n)          O(n)

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

Solution :
Imprime :

```
{'b': 1, 'r': 1, 'u': 1, 'j': 1, 'n': 1, 'o': 2}
{'r': 1, 'b': 1, 'u': 1, 'j': 1, 'n': 1, 'o': 2}
True
```

Premier diagramme d'état :



Les autres diagrammes d'état sont moins intéressants (voir pythontutor pour les obtenir).
Ce code teste si les deux strings sont égaux à l'ordre des caractères prêt (bijection entre les deux séquences).
Complexité : (Hypothèse, s et t de même taille n) moyenne et maximale $O(n)$: la taille du dictionnaire `d` est au maximum de 128 élements : de ce fait toute manipulation (comme celle de l'instruction `setdefault`) est en $O(1)$ (temps borné par une constante).

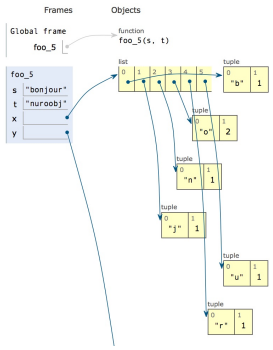
```
def foo_5(s,t):                                                    # complexite moyenne  O(n^2) maximale O(n^2)
    x = [(s[i], s.count(s[i])) for i in range(len(s)) if s[i] not in s[:i]]          # O(n^2)       O(n^2)
    y = [(t[i], t.count(t[i])) for i in range(len(t)-1,-1,-1) if t[i] not in t[i+1:]] # O(n^2)       O(n^2)
    x.sort()                                                                         # O(n log n)   O(n^2)
    y.sort()                                                                         # O(n log n)   O(n^2)
    print()                                                                          # O(1)         O(1)
    return x == y                                                                    #  O(n)        O(n)

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

Solution :
Imprime :

```
True
```

Diagrammes d'état (y est semblable à x) :



L'autre diagramme d'état est moins intéressant (voir pythontutor pour l'obtenir).
Comme le code précédent, ce code teste si les deux strings sont égaux à l'ordre des caractères prêt (bijection entre les deux séquences).
Complexités moyenne et maximale : les compréhensions de liste donne la complexité : n comptages chacun de complexité O(n).

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

Écrivez une fonction `remplace_lettres(lettres, texte, nouveau)` qui reçoit 3 noms de fichiers en arguments. Cette fonction doit écrire dans un nouveau fichier (*nouveau*) le contenu d'un fichier (*texte*), où chacune des lettres a été remplacée par "x" fois une autre lettre en respectant la casse. Les instructions de remplacement sont indiquées dans le fichier *lettres*.

Par exemple, le fichier `lettres.txt` :

```
A E 3
E O 2
I U 3
P R 4
U I 1
Y A 3
```

indique qu'il faut remplacer tous les "a" ou "A" du fichier *texte* par "eee" ou "EEE" dans le fichier *nouveau*.

L'appel

```
>>> remplace_lettres("lettres.txt", "texte.txt", "nouveau_texte.txt")
```

avec le fichier `texte.txt` :

```
Bien sûr, dit le renard. Tu n'es encore pour moi qu'un petit garçon tout semblable
à cent mille petits garçons. Et je n'ai pas besoin de toi. Et tu n'as pas besoin de
moi non plus.
```

produirait le fichier `nouveau_texte.txt` avec le contenu suivant :

```
Buuuoon sûr, duuut loo rooneeerd. Ti n'oos ooncoroo rrrroir mouuu qi'in rrrrootuuut
geeerçon toit soombleeebloo à coont muuulloo rrrrootuuuts geeerçons. OOt joo n'eeeuuu
rrrreees boosouuun doo touuu. OOt ti n'eees rrrreees boosouuun doo mouuu non rrrrlis.
```

Remarque : Vous avez la garantie que le fichier *lettres* respectera le format indiqué dans l'exemple, à savoir qu'une lettre à remplacer n'apparaît qu'une seule fois dans le fichier (pas d'indication contradictoire) et qu'elle est suivie de la lettre de remplacement et d'un nombre le tout séparés par des espaces.

Solution.

```
def load_lettres(lettres):
    """
    Retourne un dictionnaire dont les clés sont (en minuscules et en majuscules)
    les lettres à remplacer du fichier lettres et dont les clés sont la suite
    du nombre correspondant de lettres de remplacement.
    Par exemple le fichier de l'énoncé donnera le dictionnaire
    {'E': 'OO', 'a': 'eee', 'I': 'UUU', ... 'A': 'EEE', ...}
    """
```

```
    dr = {}
    for l in open(lettres):
        (lettre, remplacant, nb) = l.split()
        dr[lettre.upper()] = remplacant.upper() * int(nb)
        dr[lettre.lower()] = remplacant.lower() * int(nb)
    return dr


def remplace_lettres(lettres, texte, nouveau):
    dr = load_lettres(lettres)
    text_in = list(open(texte).read())
    text_out = []
    for c in text_in:
        text_out.append(dr.get(c, c))

    fout = open(nouveau, "w")
    fout.write("".join(text_out))
    fout.close()

>>> remplace_lettres("lettres.txt", "texte.txt", "nouveau.txt")
```

Question 4 - Tri (4 points)

Écrivez une fonction `sort_words(data)` qui reçoit comme paramètre `data` une liste de mots (chaînes de caractères) et qui la trie principalement en ordre décroissant selon la taille des mots et puis, en cas d'ex aequo (c'est-à-dire si plusieurs mots ont la même taille), selon l'ordre lexicographique. Vous ne pouvez pas utiliser les fonctions intégrées de tri offertes par Python. Vous pouvez utiliser le tri par sélection ou le tri par insertion. Le fonction doit être codée de la manière la plus optimale possible. La fonction ne renvoie rien mais change juste la liste `data` correspondante.

Par exemple :

```
>>> ls = ['JEAN','JOHN','ZARGRDZ','BARBARA','LEA','JEAN-PHILIPPE']
>>> sort_words(ls)
>>> print(ls)
['JEAN-PHILIPPE', 'BARBARA', 'ZARGRDZ', 'JEAN', 'JOHN', 'LEA']
```

Solution.

```
def premier_mot_avant(mot1, mot2):
    """
    mot1 vient avant mot2 s'il est plus long ou, en cas d'égalité
    s'il vient avant dans l'ordre lexicographique
    """
    return len(mot1) > len(mot2) or (len(mot1) == len(mot2) and mot1 < mot2)

def min_pos_from(ls, start):
    """La position du min de ls en commencant à l'indice start.
    Pre: start dans ls
    """
    res = start
    for j in range(start, len(ls)):
        if premier_mot_avant(ls[j], ls[res]):
            res = j
    return res

def swap(ls, i1, i2):
    ls[i1], ls[i2] = ls[i2], ls[i1]

# Tri par sélection
def sort_words(ls):
    for i in range(len(ls) - 1):
        pos = min_pos_from(ls, i)
        swap(ls, i, pos)
```

Question 5 - Récursivité (4 points)

Sur nos ordinateurs, les fichiers sont organisés en dossiers.

Un dossier est défini par son nom et caractérisé par son contenu. Un dossier peut contenir des fichiers et d'autres (sous-)dossiers. Un fichier est, lui, défini par son nom et caractérisé par une certaine taille (en bytes). Nous décidons de représenter le contenu d'un dossier par un dictionnaire dont les clés seraient les noms de chaque élément qu'il contient et dont les valeurs associées seraient soit un entier (pour les fichiers), soit des dictionnaires pour les sous-dossiers.

La taille d'un dossier est définie comme la somme des tailles de tous les fichiers et de tous les (sous-)dossiers qu'il contient.

Écrivez une fonction récursive `size` qui reçoit un dictionnaire représentant le contenu d'un dossier et renvoie la taille du dossier. Par exemple :

```
>>> dossier = {
  "fichier1.txt": 5,
  "dossier1": {
    "fichier2.txt": 7,
    "dossier3": {}
  },
  "fichier3.txt": 12,
  "dossier2": {
    "fichier4.txt": 2
  }
}
>>> print(size(dossier))
26
```

Solution.

```
def size(dossier):
    total = 0
    for (name, val) in dossier.items():
        if type(val) == int:
            total += val
        else:
            total += size(val)
    return total
```