1ère Spé. NSI Gr. 2 **Contrôle #4, 05/04/2024**
Lycée Fustel de Coulanges **Durée: 1 Heure**

Ce contrôle comporte 5 questions; le nombre maximal possible de points est de 22. Les réponses sont à porter sur une copie comportant votre nom. Il n'est pas nécessaire de répondre aux questions dans l'ordre — commencez par celles où vous vous sentez le plus à l'aise *(mais ne tentez les questions bonus qu'après avoir fini le reste!!)*.
Les calculatrices ne sont pas autorisées.

---

1. *Questions de cours*
   (a) (1 ½ points) QCM – les réponses sont à porter sur votre copie; il n'est pas nécessaire de justifier vos réponses; il y a une bonne réponse par question
      i. Si j'ai déterminé qu'un algorithme était de complexité quadratique (également noté $\mathcal{O}(n^2)$), et si je double la taille des données en entrée, alors la durée de traitement...
         ◯ ... restera la même.
         ◯ ... doublera.
         ◯ ... triplera.
         √ **... quadruplera.**

```python
def compte_elts(liste):
    compteur = 0
    for element in liste:
        compteur += 1
    return compteur
```

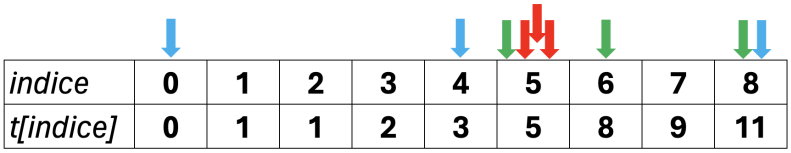
      ii. La fonction ci-dessus a une complexité en (n étant la taille de `liste`)...
         ◯ ... $\mathcal{O}(n^2)$.
         √ **... $\mathcal{O}(n)$.**
         ◯ ... $\mathcal{O}(log_2(n))$.
         ◯ ... $\mathcal{O}(1)$
      iii. Un tri "stable"...
         √ **... maintient les positions relatives des éléments équivalents.**
         ◯ ... place les éléments en ordre décroissant.
         ◯ ... a une complexité supérieure à un tri instable.
         ◯ ... a une complexité inférieure à un tri instable.
   (b) (2 points) Expliquez en deux ou trois phrases le principe d'une approche gloutonne à la résolution d'un problème.

Solution: Les éléments de réponse attendus étaient (tous n'étaient pas obligatoirement à intégrer à votre réponse):

- Une approche gloutonne vise à résoudre des problèmes d'optimisation – c'est-à-dire qu'ils ne recherchent pas *la* solution à un problème, mais la solution *la meilleure possible* (selon un axe d'optimisation défini) à ce problème.
- Une approche gloutonne est une méthode de résolution qui fait, à chaque étape, le choix le plus avantageux à court terme ou le plus "gourmand" sans prendre en compte les conséquences futures. Ce choix est fait dans l'espoir qu'il mènera à une solution globale optimale du problème.
- On parle d'*optima locaux* – ce qui décrit la notion de "meilleur choix" local, pour chaque étape prise individuellement.
- Les algorithmes gloutons ne fournissent pas nécessairement la meilleure solution possible – mais ils peuvent le faire dans certains cas, comme celui du rendu de monnaie caractérisé par le fait que l'ensemble des solutions possibles (l'ensemble des dénominations possibles en euros) constitue un *système canonique*.

(c) (2 points) Supposons que la fonction `recherche_dicho(table, elt)` implémente la recherche dichotomique telle que nous l'avons étudiée en cours. Quelles sont les valeurs qui vont être examinées lors de l'appel `recherche_dicho([0, 1, 1, 2, 3, 5, 8, 9, 11], 7`? *Indice: la première va être table[4] qui vaut 3.*[1]

Solution: Comme suggéré par l'énoncé, utilisons la technique du tableau de valeurs que nous avons déjà utilisée en cours:



indice	0	1	2	3	4	5	6	7	8
t[indice]	0	1	1	2	3	5	8	9	11

Debut	Fin	Milieu	`table[Milieu]`
0	8	4	3
5	8	6	8
5	5	5	5

Le tableau ci-dessus montre bien que trois valeurs sont examinées successivement par l'algorithme avant d'arriver à la conclusion de l'absence de l'élément 7 dans la table fournie en entrée. Ces trois valeurs sont (c'est évidemment toujours `t[milieu]` qui est évalué — voir le cours sur l'algorithme de recherche dichotomique si cela ne vous parait pas clair):

1. `t[4] = 3`
2. `t[6] = 8`
3. `t[5] = 5`

2. (6 points) *Détermination de complexités*

```python
def f1(n):
    x = 0
    for i in range(n):
        x = x + 1
    return x
```

```python
def f2(n):
    x = 0
    for i in range(1000000000):
        x = x + 1
    return x
```

```python
def f3(n):
    x = 0
    for i in range(n):
        for j in range(5):
            x = x + 1
    return x
```

```python
def f4(n):
    x = 0
    i = 0
    while i * i < n:
        x = x + 1
        i = i + 1
    return x
```

```python
def f5(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x = x + 1
    return x
```

```python
def f6(n):
    x = 0
    for i in range(n):
        x = x + 1
    for j in range(n):
        x = x + 1
    return x
```

Déterminer la complexité de chacune des fonctions `f1` à `f6` en fonction de n (vous pouvez utiliser la notation $\mathcal{O}()$ ou la terminologie "constante / linéaire / quadratique / autre"). Il est demandé pour chaque fonction une brève phrase de justification de la réponse.

Solution: Les complexités respectives de ces fonctions sont:

f1 i parcourt toutes les valeurs de 0 à $(n-1)$, donc la complexité est **linéaire** en $\mathcal{O}(n)$; on se convainc aisément en effet que si n double, alors le nombre d'itérations de la boucle doublera aussi.

f2 n n'apparait même pas dans la fonction – la complexité de celle-ci n'en dépend donc absolument pas, on est donc sur une complexité **constante**, en $\mathcal{O}(1)$; on se convainc aisément en effet que si n double, alors le nombre d'itérations de la boucle ne changera pas — il restera à 1.000.000.000.

[1] N'hésitez pas à utiliser sur votre copie un tableau donnant les valeurs successives des indices "`debut`", "`fin`", et "`milieu`".

f3 Il y avait ici un (tout petit) piège: on a bien deux boucles imbriquées, mais le nombre d'itérations de la boucle intérieure est fixe (à 5), donc ne dépend pas de n; la complexité dépend donc exclusivement de la boucle extérieure pour laquelle i parcourt toutes les valeurs de 0 à $(n-1)$, donc la complexité est **linéaire** en $\mathcal{O}(n)$; on se convainc aisément en effet que si n double, alors le nombre d'itérations de la boucle doublera aussi.

f4 La fonction la plus difficile des 6: $i \times i$ évolue de 0 à $(n-1)$, donc i lui même évolue, de 1 en 1, de 0 à approximativement $\sqrt{n}$. Le nombre d'itérations de la boucles dépend donc de la racine carrée de n – et sa complexité est donc en $\mathcal{O}(\sqrt{n})$. On se convainc d'ailleurs aisément que si n double, alors le nombre d'itérations de la boucle passera d'environ $\sqrt{n}$ à $\sqrt{2n}$ – et aura donc été multiplié par $\sqrt{2}$.

f5 Là nous nous trouvons dans une configuration de boucles imbriquées "classique" où la boucle intérieure accomplit n itérations pour chaque itération de la boucle extérieure qui en accomplit elle-même n également – on est donc dans une complexité **quadratique** en $\mathcal{O}(n^2)$; on se convainc aisément en effet que si n double, alors le nombre d'itérations de la boucle intérieure doublera, celui de la boucle extérieure également et donc au total le nombre d'opérations aura quadruplé (aura donc été multiplié par 2^2).

f6 Pas de piège en tant que tel ici mais quelque chose à retenir: la complexité est une question d'augmentation en fonction de la taille de l'entrée – donc une fonction qui réalise n opérations et une qui en réalise $2 \times n$ comme, à l'évidence, celle-ci, ont la même complexité puisqu'un doublement de n reviendra à un doublement du nombre d'opérations: complexité **linéaire** en $\mathcal{O}(n)$.

3. (2 points) *Fonction de recherche de 0*

```
1  def compter_zeros(t):
2  ''' Fonction qui compte le nombre de zéros après chaque élément de t'''
3      n = len(t)
4      compte = [0] * n # Rappel: renvoie un tableau de longueur n de 0: [0, 0, ..., 0]
5      for i in range(n):
6          for j in range(i+1, n):
7              if t[j] == 0:
8                  compte[i] += 1
9      return compte
```

Exécutons cette fonction sur un tableau spécifique: `compter_zeros([1, 0, 2, 0])`. Sur votre copie, complétez le tableau suivant avec les valeurs successives des variables – ajoutez autant de lignes qu'il y aura de passages dans la comparaison de la ligne 7 "`if t[j] == 0:`"[2]; puis concluez en indiquant ce que renverra la fonction.

Ligne Code	`compte`	`i`	`j`	`t[i]`	`t[j]`
7	`[0,0,0,0]`	0	1	1	0
7	`[1,0,0,0]`	0	2	1	2
7	...	...	...	...	...

Solution:

Ligne Code	`compte`	`i`	`j`	`t[i]`	`t[j]`
7	`[0,0,0,0]`	0	1	1	0
7	`[1,0,0,0]`	0	2	1	2
7	`[1,0,0,0]`	0	3	1	0
7	`[2,0,0,0]`	1	2	0	2
7	`[2,0,0,0]`	1	3	0	0
7	`[2,1,0,0]`	2	3	2	0

Lors du dernier passage dans le test "`if t[j] == 0:`" la condition sera satisfaite (puisque `t[3] = 0`), donc la fonction renverra comme tableau final `[2, 1, 1, 0]`.

4. *Fichier CSV* — On considère le code suivant:

[2] Indice: il y en aura 6 en tout, les deux déjà présents dans l'énoncé inclus.

```
import csv
fichier = open('Specialites.csv', 'r', encoding = 'utf-8')
table = list(csv.DictReader(fichier))
```

Et soit le fichier `Specialites.csv` contenant les données suivantes:

```
Eleve,Classe,Age,Spe1,Spe2,Spe3
Loubna,1G5,16,Maths,NSI,Physique
Olivier,1G2,17,NSI,Maths,SES
Lenny,1G2,17,LLC-Anglais,SES,NSI
Anju,1G3,16,LLC-Anglais,NSI,SES
Sophie,1G5,15,Maths,NSI,SES
```

(a) (½ point) Qu'est-ce qui va s'afficher à l'exécution du code suivant?

```
for i in range(len(table)):
    print(table[i]['Spe2'])
```

Solution: Pas de difficulté particulière ici – juste un affichage des valeurs successives de la colonne Spe2, donc:

```
NSI
Maths
SES
NSI
NSI
```

(b) (1 point) Complétez la fonction suivante pour qu'elle fasse ce qui est spécifié.

```
def CompteEleves(spe, table):
    ''' Fonction qui renvoie le nombre d'élèves ayant choisi la spécialité passée en argument'''
    n = len(table)
    compte = 0
    for i in range(n):
            # A COMPLETER: le "if" qui va tester si l'élève d'indice i a la spécialité "spe"
            ...
                compte += 1
    return compte
```

Solution:

```
def CompteEleves(spe, table):
    ''' Fonction qui renvoie le nombre d'élèves ayant choisi la spécialité passée en argument'''
    n = len(table)
    compte = 0
    for i in range(n):
        # On vérifie parmi les trois spécialités
        if spe in (table[i]["Spe1"], table[i]["Spe2"], table[i]["Spe3"]):
            compte += 1
    return compte
```

Note: on pouvait également procéder avec 3 égalités avec des "`or`" — ce qui revenait au même:

```
if spe == table[i]["Spe1"] or spe == table[i]["Spe2"] or spe == table[i]["Spe3"]:
```

(c) (2 points) Rédigez une fonction AgeMoy(classe) qui renvoie l'âge moyen des élèves présents dans la classe passée en argument. Par exemple `AgeMoy('1G5')` renverra la valeur 15.5 (la moyenne de 15 et 16)[3].

[3] N'hésitez pas à commencer par en rédiger l'algorithme en pseudo-code: des points seront attribués à cela même si le code final est faux ou absent.

Solution:

```python
def AgeMoy(classe):
    ''' Fonction qui renvoie l'age moyen des élèves présents dans la classe passée en argument'''
    n = len(table)
    CumulAge = 0 # âge cumulé de tous les élèves
    compte = 0 # nombre d'élèves
    for i in range(n):
        # On vérifie si l'élève est dans la classe spécifiée
        if table[i]["Classe"] == classe:
            CumulAge += int(table[i]["Age"]) # DictReader ne renvoie que des alphanumériques
            compte += 1
    moy = CumulAge / compte
    return moy
```

5. *Ré-écriture du tri par sélection*

(a) (2 points) Ecrivez le pseudo-code d'une fonction `Prochain_Min(liste, indice_courant)` qui prend en entrée une liste et un indice et qui renvoie l'indice de la valeur minimale présente dans `liste` entre l'indice `indice_courant` (inclus) et la fin de la liste. Par exemple: `Prochain_Min([10,13,11,12], 0)` renverra 0 (correspondant à la valeur 10) et `Prochain_Min([10,13,11,12], 1)` renverra 2 (correspondant à la valeur 11).

Solution:

Entrée: Liste, liste d'éléments triables; indice_courant, un entier correspondant à l'indice de départ
Sortie: Resultat, entier correspondant à l'indice du minimum de la liste à partir d'indice_courant

fonction PROCHAINMIN(Liste, indice_courant)
    $n \leftarrow$ len(Liste)
    Resultat $\leftarrow$ *indice_courant* ▷ Initialisation du résultat
    CurMin $\leftarrow$ *Liste*[*indice_courant*] ▷ Initialisation du minimum
    pour i allant de *indice_courant* à n **faire**
        si *Liste*[i] $<$ *CurMin* **alors** ▷ On a trouvé un nouveau minimum
            CurMin $\leftarrow$ *Liste*[i]
            Resultat $\leftarrow i$
        fin si
    fin pour
    retourner Resultat
fin fonction

(b) (1 point) Traduisez le pseudo-code que vous venez de rédiger en fonction codée en Python.

Solution:

```python
def Prochain_Min(lst, indice):
    ''' Fonction qui renvoie le minimum de la sous-liste de lst débutant à indice et
    terminant à la fin de la liste'''
    n = len(lst)
    res = indice
    curmin = lst[indice]
    for i in range(indice, n):
        if lst[i] < curmin:
            curmin = lst[i]
            res = i
    return res
```

(c) (2 points) Complétez le code suivant (parties A et B) pour qu'il réalise le tri par sélection d'une table passée en argument tel que nous l'avons vu en cours[4]:

[4]Un petit rappel, pour gagner du temps dans la permutation: le code "`a , b = b , a`", en Python, met la valeur de a dans b et celle

```
def TriSelect(table):
    ''' Fonction qui applique le tri par sélection à table et renvoie la table triée'''
    n = len(table)
    for i in range(n):
        # A COMPLETER - A: appel à votre fonction Prochain_Min
        ...
        # A COMPLETER - B: permutation des valeurs pour placer le ième plus petit élément à l'indice i
        ...
    return table
```

Solution:

```
def TriSelect(table):
    ''' Fonction qui applique le tri par sélection à table et renvoie la table triée'''
    n = len(table)
    for i in range(n):
        indice_a_permuter = Prochain_Min(table, i)
        table[i], table[indice_a_permuter] = table[indice_a_permuter], table[i]
    return table
```

(Question bonus 1): En conservant le modèle que l'on a utilisé dans la question précédente pour le tri par sélection (une fonction principale et une sous-fonction qui cherche l'indice du minimum), écrivez une implémentation de ce même tri mais en ordre décroissant.

Solution: Rien de bien compliqué ici - on va juste chercher le maximum au lieu de chercher le minimum à chaque fois. On va donc remplacer la fonction Prochain_Min par:

```
def Prochain_Max(lst, indice):
    ''' Fonction qui renvoie le maximum de la sous-liste de lst débutant à indice et
    terminant à la fin de la liste'''
    n = len(lst)
    res = indice
    curmax = lst[indice]
    for i in range(indice, n):
        if lst[i] > curmin:
            curmax = lst[i]
            res = i
    return res
```

Et il suffira alors de remplacer dans la fonction `TriSelect` l'appel à:

`Prochain_Min(table, i)`

par:

`Prochain_Max(table, i)`.

(Question bonus 2): En utilisant ce que vous avez fait à la question 6 *et* ce que vous avez fait à la question bonus 1, écrivez une implémentation du tri par sélection qui classe tous les nombres pairs de la liste par ordre croissant à gauche de la liste en sortie, et tous les nombres impairs par ordre décroissant à droite. Par exemple, si on appelle cette fonction `TriSelTordu(table)`, l'appel `TriSelTordu([1, 9, 8, 10, 6, 5, 11, 23, 2])` renverra [2, 6, 8, 10, 23, 11, 9, 5, 1].

---

de b dans a.

Solution: La solution la plus simple à ce problème consiste à découper tout d'abord la liste en deux, puis à trier chacune d'entre elles séparément, puis enfin à "recoller" les deux listes ensemble.

Dans le code qui suit on part du principe que sont déjà définies:

`TriSelectCrois` La fonction qui a été rédigée en réponse à la question 5 (*et pas 6 comme le disait l'énoncé...*) s'appuyant sur la fonction `Prochain_Min`: elle renvoie une table triée par ordre croissant au moyen du tri par sélection.

`TriSelectDeCrois` La fonction qui a été rédigée en réponse à la question bonus 1 s'appuyant sur la fonction `Prochain_Max`: elle renvoie une table triée par ordre décroissant au moyen du tri par sélection.

```
1  def TriSelTordu(lst):
2      ''' Fonction qui répond à l'énoncé...'''
3      n = len(lst)
4      LstPair = []
5      LstImpair = []
6      for i in range(n):
7          if lst[i] % 2 == 0:
8              LstPair.append(lst[i])
9          else:
10             LstImPair.append(lst[i])
11     LstPair = TriSelectCrois(LstPair)
12     LstImPair = TriSelectCrois(LstImPair)
13     return LstPair + LstImpair
```

(Question bonus 3): Une technique pour repérer le plagiat dans un texte consiste à repérer les enchainements de mots (plutôt que les mots individuels). Écrivez une fonction `Plagiat(txt)` qui prend en entrée une chaîne de caractères et renvoie les 2 enchainements de mots les plus fréquents qu'elle contient. Par exemple si l'on donne à la variable txt la valeur "Le vent souffle fort sur la plaine. Les arbres dans la plaine se courbent sous le vent. Le vent, le vent, toujours le vent.", `Plagiat(txt)` renverrait les enchainements "le vent" (présent cinq fois) et "la plaine" (deux fois)[5].

Solution: En simplifiant on pouvait proposer une fonction de ce type:

[5]Rappel: la commande "`lst = txt.split()`" crée une liste `lst` dont les éléments sont les mots de `txt`.

```python

def Plagiat(txt):
    ''' Fonction qui renvoie les deux enchainements de mots les plus fréquents de txt'''
    # On découpe le texte en mots
    lst = txt.split()
    res = {}

    # On fait l'inventaire des enchainements
    for i in range(1, len(lst)):
        if (lst[i-1], lst[i]) not in res:
            res[(lst[i-1], lst[i])] = 1
    else:
        res[(lst[i-1], lst[i])] += 1

    # On identifie les deux plus fréquents
    curmax1 = ((),0)
    curmax2 = ((),0)
    for enchainement in res:
        if res[enchainement] > curmax1[1]:
            curmax2 = curmax1
            curmax1 = (enchainement,res[enchainement])
        elif res[enchainement] > curmax2[1]:
            curmax2 = (enchainement,res[enchainement])
    return curmax1, curmax2
```

Notez que cette fonction est vraiment simpliste — elle fait notamment abstraction des majuscules, de la ponctuation... Il faudrait à l'évidence nettement l'améliorer pour pouvoir s'en servir "réellement".