Informatique - DM 1 - Corrigé

Remarques d'ordre général :

Ne pas oublier de laisser une marge à gauche et un espace suffisant pour les appréciations et conseils du correcteur. Écrire lisiblement est toujours un plus très apprécié des correcteurs. Si cela ne vous est pas naturel, des efforts constants tout au long de l'année vous permettront des progrès.

Une première méthode pour concevoir un code consiste à réaliser l'action recherchée « à la façon d'une machine » sur des exemples que vous jugez représentatifs du cas général. Envisagez plusieurs manières de faire, la première qui vient à l'esprit n'est pas forcément la plus simple ou la plus efficace.

On peut alors écrire du code.

Ne pas confondre la chaine de caractères `'True'` avec la constante booléenne `True`.

Ne pas utiliser `print` sauf si c'est clairement et absolument obligatoire.

Ne pas oublier de tester les fonctions écrites en particulier sur les cas « limites ». Que penser d'une fonction qui renvoie `True` lorsque vous lui demandez si 25 est premier ?

Exercice 1

1. Une première version avec une boucle `while` où l'on teste les diviseurs k de n sans dépasser $\sqrt{n}$. À l'issue de la boucle, si k^2 dépasse n c'est que l'on a jamais rencontré de k tel que `n % k == 0` et on retourne donc `True`, sinon, c'est que l'on a arrêté la boucle à cause de la condition `(n % k) != 0` et on retourne `False`. Ainsi, on retourne plus simplement la condition `k**2 > n` qui s'évalue en `True` ou `False` :

```
def est_premier(n):
    k = 2
    while k**2 <= n and (n % k) != 0:
        k += 1
    return k**2 > n
```

Une deuxième version avec une boucle `for` et un `return` à l'intérieur.

```
from math import sqrt

def est_premier2(n):
    for k in range(2, int(sqrt(n))+1):
        if n % k == 0:
            return False
    return True
```

Commentaires de correction :

Il était inutile de tester séparément le cas des entiers pairs par un test `if` puisque si n est pair, on sort tout de suite de la boucle à moins de prendre ensuite un pas de 2 dans la boucle. D'une manière générale, un code qui commence par examiner des cas particuliers est souvent un code mal écrit ou mal conçu (pourquoi ne pas tester aussi si n est un multiple de 3, de 5, de 7 etc.).

Pour une fonction qui opère un test, on vérifiera que la réponse est donnée dés qu'elle est acquise. Ainsi, compter tous les diviseurs pour finir par tester si le nombre de diviseurs est égal à 2 est mauvais.

2. Le dernier chiffre (au sens de celui le plus à droite) s'obtient en calculant le reste de la division euclidienne de n par 10. Puis on recommence avec le quotient. Il faut donc utiliser `insert` pour ajouter chaque chiffre en début de liste. On continue tant que l'on est pas arrivé à 0 :

```
def liste_chiffres(n):
    L = []
    while n > 0:
        L.insert(0, n % 10)
        n = n // 10
    return L
```

Commentaires de correction :

Il faut savoir utiliser correctement le quotient et le reste de la division euclidienne que l'on obtient à l'aide des opérateurs : `//` et `%`. Tester une divisibilité par `n/k == int(n/k)` est du bricolage. À éviter absolument !

3. On réserve le dernier élément de la liste dans une variable temporaire, on l'efface puis on l'insère en première position.

```
def decale_liste(L):

    temp = L[-1]
    del L[-1]
    L.insert(0, temp)
```

Commentaires de correction :
Attention, il ne fallait pas ici renvoyer une nouvelle liste mais modifier la liste passée en paramètre par effet de bord.
On rappelle que :
Lorsqu'une question est rédigée sous la forme « Écire une fonction `nom_fonction` prenant en paramètre machin et qui renvoie truc », on ne s'attend pas à ce qu'elle produise un effet de bord. Si c'est le cas, ce sera considéré comme une erreur : « effet de bord non désiré ». Cela donne souvent lieu à des erreurs difficiles à détecter. Le code fait bien ce qu'on lui demande mais il fait aussi autre chose dont on n'a pas conscience et qui génère des erreurs.

Par contre, si une question est rédigée (comme ici) sous la forme « Écrire une fonction `nom_fonction` prenant en paramètre machin et qui opère - une certaine action - sur le paramètre machin » alors c'est au contraire un effet de bord qui est recherché.

4. Il y a plusieurs stratégies possibles pour faire ce calcul. On peut partir de la gauche ou de la droite (i.e. du chiffre des unités).

   Si l'on part de la gauche voici une première version naïve :

```
def nombre_liste2(L):
    n = len(L)
    s = 0
    for k in range(n):
        s += L[k]*10**(n-1-k)
    return s
```

Ce n'est pas très efficace car on calcule beaucoup de puissances de 10.
Si l'on veut partir de la droite, on peut aussi calculer les puissances de 10 petit à petit ce qui est beaucoup mieux.

```
def nombre_liste4(L):
    n = len(L)
    s = 0
    p = 1
    for k in range(n-1, -1, -1):
        s += L[k]*p
        p *= 10
    return s
```

Enfin la bonne version au sens où l'on minimise les calculs puisque l'on ne calcule pas les puissances de 10. Il s'agit de la **méthode de Hörner** basée sur l'idée suivante :

$$1789 = 1.10^3 + 7.10^2 + 8.10 + 9 = (1.100 + 7.10 + 8).10 + 9 = ((1.10 + 7)10 + 8).10 + 9$$

```
def nombre_liste(L):
    s = 0
    for c in L:
        s = 10*s+c
    return s
```

Toutes ces versions diffèrent par le nombre de multiplications effectuées : pour la version 1 il y en a de l'ordre de n^2. Pour la version 2 c'est $2n$. Enfin avec la méthode de Hörner c'est n.

5. Il suffit d'enchaîner correctement les trois fonctions précédentes :

```
def decale_nombre(n):

    L = liste_chiffres(n)
    decale_liste(L)
    return nombre_liste(L)
```

Commentaire de correction : Noter l'action par effet de bord de la fonction `decale_liste`

6. Une première solution consiste à commencer par déterminer le nombre de chiffres qui composent `n` puis à effectuer autant de test de primalité pour les nombres successivement obtenus par décalage. Si tous les tests ont été effectués, on renvoie `True` sinon on sort de la boucle en renvoyant `False`.

```
def est_solution(n):

    k = len(liste_chiffres(n))
    for i in range(k):
        if not(est_premier(n)):
            return False
        n = decale_nombre(n)
    return True
```

Une seconde solution teste si on est revenu au nombre de départ tant que les nombres obtenus par `decale_nombre` sont premiers :

```
def est_solution2(n):

    m = n
    while(est_premier(m)):
        m = decale_nombre(m)
        if m == n:
            return True
    return False
```

7. Il suffit de tester tous les nombres comportant N chiffres.

```
def cherche_solution(N):
    for n in range(10**(N-1), 10**N):
        if est_solution(n):
            return n
```

8. Si la fonction `est_premier` est suffisamment efficace, la réponse, pour $N = 6$, est obtenue avec la fonction précédente en quelques secondes au plus : 193939.
   Commentaire de correction :
   Si vous devez attendre très longtemps, il y a certainement une erreur de conception dans l'une ou l'autre fonction.

Exercice 2

Une première version naïve dont la complexité est en $O(n^3)$.

```
def plus_grande_sous_somme_naif(L):

    n = len(L)
    record = L[0]
    for i in range(0, n):
        for j in range(i + 1, n + 1):
            somme = 0
            for k in range(i, j):
                somme = somme + L[k]
            if somme > record:
                record = somme
    return record
```

On peut améliorer ce code pour obtenir une version quadratique

```
def plus_grande_sous_somme_quad(L):

    n = len(L)
    record = L[0]

    for i in range(0, n):
        somme = 0
        for j in range(i + 1, n + 1):
            somme += L[j - 1]
```

```
            if somme > record:
                record = somme
    return record
```

Enfin, l'algorithme de Kadane donne une complexité linéaire.

On remarque qu'il est facile de déterminer, lors du parcours de la liste, le maximum d'une somme qui se termine au rang courant (c'est le rôle de la variable `max_local`) : si le maximum M de toutes les sommes qui se terminent au rang $i-1$ est négatif, le maximum des sommes qui se terminent au rang i est égal au seul élément en position i sinon sa valeur sera augmentée du terme en position i.

Il suffit alors de prendre le plus grand de ces maximas pour obtenir le maximum recherché. Remarque : l'usage de la fonction `max` est toléré ici sur un nombre fini d'arguments (mais pas sur des listes).

```
def plus_grande_sous_somme(L):
    ''' Recherche de la plus grande sous-somme non vide dans une
    liste par l'algorithme de Kadane  '''

    max_global = L[0]
    max_local = L[0]

    for i in range(1, len(L)):
        max_local = max(0, max_local) + L[i]
        max_global = max(max_global, max_local)
    return max_global
```

Une autre version du même algorithme, sans recours à `max` (vu dans une copie) :

```
def plus_grande_sous_somme2(L):
    ''' Solution d'un eleve  '''
    Smax = L[0]
    S = 0
    for x in L:
        if S < 0:
            S = x
        else:
            S += x
        if S > Smax:
            Smax = S
    return Smax
```

Commentaire de correction : Il apparait clairement que plusieurs solutions proposées n'ont pas été assez testées.