D.S. 2 informatique commune MPSI 1

Question 0. a) On calcule : $65 = 64 + 1 = 2^6 + 1$ donc son écriture binaire est `1000001` (écriture sur 7 bits).

Ensuite `97=64+32+1=65+32` ce qui s'écrit en binaire, sur 7 bits, `1100001`.

Comme $97 - 65 = 32 = 2^5$, pour passer de l'écriture binaire d'une majuscule à une miniscule, il suffit de modifier un seul bit (le bit numero 5 en appelant bit numéro 0 le bit des unités).

b) Le seul piège avec la méthode `lower` est qu'elle s'applique au type `str` non mutable. Donc comme le dit d'ailleurs l'énoncé, elle retourne une nouvelle chaîne de caractères, mais elle ne modifie pas la chaîne à laquelle on l'applique, d'où la nécessité de stocker sa valeur de retour dans le programme ci-dessous. Ne pas oublier aussi la condition `if numero>=97 and numero<=122:` qui fait qu'on ne codera que les caractères correspondant à des lettres et pas les caractères spéciaux.

```
def ConvertitTexteEnTab(texte):
    tab=[]
    texteminus=texte.lower()
    for  lettre in texteminus:
        numero=ord(lettre)
        if numero>=97 and numero<=122:
            tab.append(numero-97)
    return tab
```

La fonction suivante est plus facile : attention seulement à ne pas utiliser de `append` pour les chaînes de caractères. Le `append` ne s'applique qu'aux listes.

```
def ConvertitTabEnTexte(tab):
    texte=''
    for valeur in tab:
        texte=texte+chr(valeur+97)
    return texte
```

Question 1. On donne deux versions. La première ne fait que des tests d'égalités d'entrées de listes, ce qui permet de bien évaluer sa complexité, comme demandé ensuite par l'énoncé à la question 2. La seconde s'autorise le test d'égalités entre listes, ce test faisant lui-même la boucle qu'on a programmé à la version 1.

```
def enTeteDeSuffixe(mot,tab,k):
    suffixe = tab[k:]
    for i in range(len(mot)):
        if mot[i]!=suffixe[i]:
            return False
    return True

def enTeteDeSuffixe2(mot,tab,k):
    suffixe = tab[k:]
    m = len(mot)
    return mot == suffixe[:m]
```

Question 2. La seule chose qui demande du soin est de savoir où arrêter le `range` de la boucle `for`. La dernière position possible pour le début de `mot` est `len(tab)-len(mot)`.

```
def rechercherMot(mot,tab):
    for i in range(len(tab)-len(mot)+1):
        if enTeteDeSuffixe(mot, tab, i):
            return True
    return False
```

Remarque : avec la fonction `enTeteDeSuffixe2`, on n'aura pas de `out of range` même si on écrit `for i in range(len(tab))` dans `rechercherMot`. C'est un avantage du *slicing*.

Complexité : Au maximum, si la boucle va à la fin, `TeteDeSuffixe(mot,tab,k)` fera `m=len(mot)` tours de boucles, donc `m` tests d'égalités. Complexité en $O(len(mot))$.

En notant $n = len(tab)$ et $m = len(mot)$, l'appel de `rechercherMot(mot,tab)` va, au maximum effectuer $n-m+1$ tours de boucles et à chaque tour de boucles l'appel de `enTeteDeSuffixe(mot,tab,i)` fait au plus m tests d'égalités élémentaires.

Au total, on fait donc au plus $m(n-m+1) \leq mn$ tours de boucles. La complexité est un $O(mn)$.

Question 3.

```
def compterOccurences(mot, tab):
    n = 0
    for i in range(len(tab)-len(mot)+1):
        if enTeteDeSuffixe(mot, tab, i):
            n=n+1
    return n
```

Question 4.

```
def frequenceLettre(tab):
    l = [0]*26
    for lettre in tab:
        l[lettre]+=1
    return l
```

La fonction ci-dessous est en $O(len(tab))$. Une autre solution souvent utilisée est la suivante :

```
def fl2(tab):
    L=[0]*26
    for i in range(26):
        for j in range(len(tab)):
            if tab[j]==i:
                L[i]=L[i]+1
    return L
```

Comparaison d'efficacité : dans la fonction `fl2` on fait $26 * len(tab)$ test d'égalités en plus. Il se trouve qu'en pratique cela change radicalement l'efficacité pratique comme le montre le test suivant :

```
# essai
from random import randint
tab=[]
for i in range(10000):
    tab.append(randint(0,25))
from time import clock
a=clock()
frequenceLettre(tab)
b=clock()
fl2(tab)
c=clock()
print("temps frequence lettre : ", b-a)
print("temps fl2",c-b)
```

Avec le résultat suivant :

```
temps frequence lettre :  0.0007429999999999382
temps fl2 0.014185000000000336
```

Le temps est multiplié d'un facteur 50, même si les deux algo. sont en $O(len(tab))$.

La fonction suivante donne le même résultat mais à chaque tour de boucle, on a un appel à `compterOccurence` en $O(len(tab))$ Le problème avec cette fonction est qu'elle appelle `enTeteDeSuffixe` dans laquelle on a crée un tableau `suffixe` à chaque appel. Sur un grand tableau, cela occasionne beaucoup d'affectations supplémentaires. Le résultat est spectaculaire.

```
def fl3(tab):
    l=[0]*26
    for i in range(26):
        l[i]=compterOccurences([i],tab)
    return l
```

```
temps fl3 2.522245
```

Le temps est 100 fois plus long qu'avec `fl2` et donc 5000 fois plus long qu'avec la première fonction. Si on modifie la fonction `enTeteDeSuffixe` en enlevant la création de la liste `suffixe` en remplaçant `suffixe[i]` par `tab[k+i]` on diminue le temps d'un facteur 10.

N.B. En outre, pour les grandes listes Python, il n'est pas vrai que l'accès à toutes les entrées se fait avec le même temps.

Question 5.

Il est demandé que la fonction *affiche* les mots de deux lettres et leur fréquence. On utilise donc la commande `print` et les résultats sont affichés sous la forme `mot : fréquence`.

Une première version vue sur le copies est la suivante :

```
def AFB_naif(tab):
    for i in range(len(tab)-1):
        mot=tab[i:i+2]
        print(mot,":",compterOccurences(mot,tab))
```

Cette fonction présente le défaut de faire des affichages multiples... elle affichera chaque mot autour de fois qu'il apparaît.

```
def AFB(tab):
    LettresPresentes=[]
    freq=frequenceLettre(tab)
    for i in range(26):
        if freq[i]!=0:
            LettresPresentes.append(i)
           #  on a frabriqué un tableau, sans répétition, des lettres présentes dans tab
    for n in LettresPresentes:  # parcours des valeurs !
        for k in LettresPresentes:
            occ=compterOccurences([n,k],tab)
            if acc!=0:
                print([n,k],"repeté",occ,"fois")
```

Question 6.

```
def minimum(tab):
    temoin=tab[0]
    imin=0
    for i in range(len(tab)):
        if tab[i]<temoin:
            imin=i
            temoin=tab[i]
    return (temoin,imin)
```

Question 7

```
def TriSelect(tab):
    tabtrie=[]
    while tab!=[]:
        imin=minimum(tab)[1]
```

```
        tabtrie.append(tab.pop(imin))
    return tabtrie
```

Question 8

La fonction `minimum` effectue exactement n comparaisons `tab[i]<temoin` : une comparaison par tour de boucles. La fonction `TriSelect` effectue n appel à la fonction `minimum` : un par tour de boucles. Donc $n \times n = n^2$ comparaisons.

Question 9

```
def comparerSuffixe(tab,k1,k2):
    suff1 = tab[k1:]
    suff2 = tab[k2:]

    if suff1 == suff2:# ne se produit que si k1==k2
        return 0 # car sinon les suffixes sont de longueurs
        # différentes !
    for i in range(min(len(suff1), len(suff2))):
        if suff1[i] < suff2[i]:
            return -1
        elif suff1[i] > suff2[i]:
            return 1
    if len(suff1)<len(suff2):
        return -1
    else :
        return 1
```

Question 10 On doit modifier la fonction `minimum` pour qu'elle calcule le minimum pour l'ordre lexicographique, autrement dit, on remplace l'ordre usuel `<` qui apparaît dans la fonction minimum par `comparerSuffixes`. Il faut pendre bien garde d'utiliser les valeurs de `T`.

Les valeurs `T[i]` représenteront des numéros de suffixes de `tab`.

La fonction `miniSuffixe` renvoie l'indice dans T de la plus petite valeur `T[i]`. C'est important de comprendre cette différence indice/valeur dès qu'on va faire des `pop` dans `T`.

```
def miniSuffixe(tab,T):
"renvoie l'indice dans T du plus petit suffixe "
    imin=0
    for i in range(len(T)):
        if comparerSuffixe(tab,T[i],T[imin])==-1:
            imin=i
    return imin

def TriSuffixe(tab,T):
    tabtrie=[]
    while T!=[]:
        imin=miniSuffixe(tab,T)

        tabtrie.append(T.pop(imin))
    return tabtrie
```

Question 11.

```
def comparerMotSuffixe(mot, tab, k):
    suff = tab[k:]
    m=len(mot)
    if mot==suff[:m] :
```

```
        return 0
    for i in range(min(len(mot), len(suff))):
        if suff[i] < mot[i]:
            return 1
        elif suff[i] > mot[i]:
            return -1
    # comme mot!=suff[:m] si on arrive à la fin
    # de cette boucle c'est que suff est plus
    # petit que mot donc
    return 1
```

Quelques commentaires : à la ligne 4, même si `m` est plus grand que la longueur de `suff` les opérations de slicing en Python ne provoquent pas de out of range.

Ensuite à la fin de la boucle `for`, si celle-ci s'exécute jusqu'au bout c'est que les tranches de `mot` et de `suff` jusqu'au min. de `len(mot)` et de `len(suff)` sont identiques. Or on a déjà passé le cas où mot est au début de `suff.` Donc c'est que `suff.` est de longueur plus petite (strictement) que mot (cas pas intéressant pour notre utilisation mais qu'importe).

Question 12

a) C'est une « question de cours »

```
def RechercheD(a,T):
    " recherche a dans le tableau T déjà trié"
    g=0
    d=len(T)
    while d-g>1 :
        m=(g+d)//2
        if T[m]==a:
            return True
        if T[m]<a:
            g= m
        else:
            d= m
    # A ce stade on a g-d=1. on teste encore si T[g]==a
    if T[g]==a :  return True


    return False
```

b) On doit ici faire la recherche dichotomique de `mot` dans le tableau ordonné des suffixes de `tab` qui est `tabS`. On utilise donc `comparerMotSuffixe` mais l'indice qu'on donne pour le suffixe dans `tab` est obtenue comme une valeur dans le tableau `tabS`.

```
def rechercherMot2(mot,tab,tabS):
    g,d=0,len(tabS)
    while d-g>1:
        m=(g+d)//2
        if comparerMotSuffixe(mot,tab,tabS[m])==-1:
            d=m
        if comparerMotSuffixe(mot,tab,tabS[m])==1:
            g=m
        else:
            return True
    if tab[tabS[g]:]==mot:
        return True
    return False
```