Informatique — MPSI/PCSI, 2023–2024

# Corrigé du devoir surveillé n° 1

## Exercice 1 – Arrêts aux stations essences

**Question 1.1.** Il suffit de partir d'un indice `i` égal à `-1` (correspondant au cas où on n'a pas encore atteint la première station) et le l'incrémenter tant qu'il n'est pas égal au dernier indice de la liste et que l'on peut atteindre la station suivante, c'est-à-dire que `L[i+1]` ne dépasse pas la capacité du réservoir `c`. À la sortie de la boucle, on indique le résultat du test indiquant si on a pu atteindre la dernière station ou non.

```
def possible(L,c):
    i=-1
    while i<len(L)-1 and L[i+1]<=c:
        i+=1
    return(i==len(L)-1)
```

**Question 1.2.** Comme précédemment, tant qu'on n'a pas atteint le dernier indice de la liste et que l'on peut atteindre la station suivante, on incrémente l'indice `i`. Il faut toutefois penser à décrémenter la capacité `c` de la quantité `L[i]` pour tenir compte du carburant consommé pour atteindre cette nouvelle station.

```
def prochaine_station(L,c,i=-1):
    while i<len(L)-1 and L[i+1]<c:
        i+=1
        c-=L[i]
    return(i)
```

**Question 1.3.** On crée un indice `i` indiquant la station atteinte, initialisé à `-1` (la voiture n'étant par encore partie), et une liste d'arrêts `A` contenant initialement l'indice `-1`. Tant que le dernier arrêt n'est pas atteint, on recherche quelle est la prochaine station atteignable avec la fonction précédente, et on l'ajoute à la liste des arrêts. L'indice de cette station est la nouvelle valeur de `i`. À la sortie de boucle, on retourne la liste `A`.

```
def arrets(L,c):
    i=-1
    A=[-1]
    while i<len(L)-1:
        i=Prochaine_station(L,c,i)
        A.append(i)
    return(A)
```

**Question 1.4.** Dans le cas où l'on n'a pas atteint la dernière station, on peut utiliser la méthode du candidat.

On incrémente l'indice `i` de 1 et on suppose que l'on fait le plein à la station atteinte : l'indice `imin` indiquant la station la moins chère est alors égal à `i`.

Tant que le réservoir n'est pas vide, on tente alors d'aller une station plus loin comme à la question 1.2. Si l'on atteint une station où le prix de l'essence est moins cher que le minimum obtenu jusqu'ici, on modifie `imin` en conséquence.

On retourne en sortie de boucle `imin`, indice de la station où le prix est moins cher parmi celles qu'on a atteintes.

```
def prochaine_station_eco(L,c,i=-1):
    if i<len(L)-1:
        i+=1
        imin=i
        c-=L[i][0]
        while i<len(L)-1 and L[i+1][0]<c:
            i+=1
            c-=L[i][0]
            if L[i][1]<L[imin][1]:
                imin=i
    return(imin)
```



**Question 1.5.** Il suffit de modifier la fonction de la question 1.3 en remplaçant la fonction `prochaine_station` par `prochaine_station_eco`.

```
def arrets_eco(L,c):
    i=-1
    A=[-1]
    while i<len(L)-1:
        i=Prochaine_station_eco(L,c,i)
        A.append(i)
    return(A)
```

**Question 1.6.** On peut initialiser une somme `s` à 0 puis parcourir la liste des arrêts, et, pour chaque indice `k` compris entre 0 et `len(A)-1`, faire au moyen d'une seconde boucle la somme des distances entre la station d'indice `A[k]+1` et `A[k+1]+1`. Une fois arrivé à cette dernière station, on fait le plein : on multiplie alors la somme des distances obtenue par la prix du carburant à cette station `L[A[k+1]][1]` et on ajoute à la somme à payer.

```
def cout_pleins(L,A):
    s=0
    for k in range(len(A)-1):
        d=0
        for l in range(A[k]+1,A[k+1]+1):
            d+=L[l][0]
        s+=d*L[A[k+1]][1]
    return(s)
```

Ce même calcul peut s'effectuer au moyen de la fonction `sum` et en générant les séquences de termes à sommer en compréhension.

```
def cout_pleins(L,A):
    return(sum(sum(L[l][0] for l in range(A[k-1]+1,A[k]+1))*L[A[k]][1] for k in range(len(A))))
```

## Exercice 2 – *Quelques éléments de cryptographie*

**Question 2.1.** Une liste par compréhension, utilisant les fonctions `ord` et `chr` permet de répondre à la question. Attention de ne pas oublier le « décalage » : le point de code Unicode du caractère `'A'` étant égal à `ord('A')=65`. L'implémentation suivante convient :

```
def alph_clair():
    return([chr(65+k) for k in range(26)])
```

**Question 2.2.** De façon analogue, il est possible d'obtenir l'alphabet chiffré par application de la formule. Attention au « décalage » et à la gestion des parenthèses. L'implémentation suivante convient :

```
def alph_crypt(a,b):
    return([chr(65+(a*k+b)%26) for k in range(26)])
```

**Question 2.3.** Pour définir la fonction `indexcar`, on commence par noter que l'on recherche l'indice `i` d'un élément dont on sait qu'il est dans la liste `L` donnée en argument ; donc il n'est pas nécessaire de préciser de condition `i<len(L)` lors du parcourt que l'on arrête dès l'élément trouvé. Le code suivant convient.

```
def indexcar(L,x):
    i=0
    while L[i]!=x:
        i+=1
    return(i)
```

**Question 2.4.** Pour implémenter la fonction `texte_crypt`, il est nécessaire de parcourir toutes les lettres de la chaîne de caractères, de trouver le caractère chiffré correspondant si ce n'est pas une espace, puis de le concaténer à une nouvelle chaîne de caractères (opération `+`). Le caractère chiffré correspondant à un caractère « clair » `e`, d'indice « clair » `i=ord(e)-65`, correspond au $i$-ième caractère de l'alphabet chiffré, soit `crypt[i]`. Le code suivant convient.

```
def texte_crypt(T):
    S=""
    for e in T:
        if e!=" ":
            S+= crypt[ord(e)-65]
        else:
            S+=" "
    return(S)
```

**Question 2.5.** Pour implémenter la fonction `texte_decrypt`, il est nécessaire de parcourir toutes les lettres de la chaîne de caractères, de trouver le caractère clair correspondant si ce n'est pas une espace, puis de le concaténer à une nouvelle chaîne de caractères. Pour trouver l'indice du caractère clair ou déchiffré correspondant à un caractère chiffré `e`, il est nécessaire de faire appel à la fonction `indexcar`, telle que l'indice « clair » soit `i=indexcar(crypt, e)`. On trouve alors le caractère en clair `clair[i]`. Le code suivant convient.

```
def texte_decrypt(T):
    S=""
    for e in T:
        if e!=" ":
            S+=clair[indexcar(crypt, e)]
        else:
            S+=" "
    return(S)
```

**Question 2.6.** Pour définir la fonction `texte_crypt_dir`, il suffit de reprendre la structure de la fonction `texte_crypt` et de ne modifier que la détermination de la lettre chiffrée. Sachant que le caractère chiffré correspondant à un caractère « clair » `e`, d'indice « clair » `x=ord(e)-65`, correspond au $y$-ième caractère, avec $y = (a \times x + b)\ [26] \in [\![0, 25]\!]$ c'est-à-dire au caractère de code `65+y`, avec `y=(a*x+b)%26`. Le code suivant convient.

```
def texte_crypt_dir(T,a,b):
    S=""
    for e in T:
        if e!=" ":
            S+= chr(65+(a*(ord(e)-65)+b)%26)
        else:
            S+=" "
    return(S)
```

**Question 2.7.** L'inverse mdulaire `u` de l'entier `a` n'existe que si `a` et `26` sont premiers entre eux, c'est-à-dire que si $\text{PGCD}(a, 26) = 1$. Dans ce cas, il est tel que $au \equiv 1\ [26]$. En supposant donc qu'il existe (et dans ce cas il est unique, cf. théorème de Bézout), pour le trouver il suffit de parcourir tous les entiers tant que le reste de la division de $au$ par 26 est différent de 1. On en déduit l'implémentation suivante :

```
def cle_decrypt(a):
    assert PGCD(a,26)==1
    u=1
    while (u*a)%26!=1:
        u+=1
    return(u)
```

où on a rajouté une pré-condition sur le fait que $a$ doit être premier avec 26 en faisant appel à une fonction `PGCD` (qui s'appelle `gcd` dans le module `math`).

**Question 2.8.** Pour définir la fonction `texte_decrypt_dir`, il suffit de reprendre la structure de la fonction `texte_decrypt` et de ne modifier que la détermination de la lettre déchiffrée. Après avoir trouvé l'inverse modulaire `u=cle_decrypt(a)`, on trouve l'indice du caractère « clair » correspondant au caractère chiffré `e`, d'indice `y=ord(e)-65`, avec l'expression donnée dans le sujet, soit $x = u \times (y - b)\ [26] \in [\![0, 25]\!]$ c'est-à-dire au caractère de code `65+x`, avec `x=(i*(y-b))%26`.



Le code suivant convient.

```
def texte_decrypt_dir(T,a,b):
    u=cle_decrypt(a)
    S=""
    for e in T:
        if e!=" ":
            S+=chr(65+(u*(ord(e)-65-b))%26)
        else:
            S+=" "
    return(S)
```

**Question 2.9.** Pour déterminer la fréquence d'une lettre dans une chaîne de caractères, il suffit de mettre en place 26 compteurs, c'est-à-dire autant que de lettres et d'incrémenter chaque compteur dès que la lettre correspondante apparaît. En mettant les 26 compteurs, initialisés à 0, dans une liste `F`, un simple parcourt des lettres de la chaîne de caractères suffit. Pour cela, il convient d'associer à chaque lettre `e`, son compteur `k=ord(e)-65` et de l'incrémenter avec `F[k]+=1`. On forme la liste des 26 couples (lettre, nombre d'occurrences) par un simple parcourt par des 26 indices. Le code suivant convient.

```
def an_freq(T):
    F = [0 for k in range(26)]
    for e in T:
        F[ord(e)-65]+=1
    return([(chr(65+k),F[k]) for k in range(26)])
```

## Exercice 3 – Codage de brins d'ADN

**Question 3.1.** Il suffit de vérifier que le caractère est dans l'ensemble {A,C,G,T}. Le code suivant convient :

```
def base(b):
    return(b=="A" or b=="C" or b=="G" or b=="T")
```

**Question 3.2.** On parcourt successivement toutes les bases du brin tant qu'elles correspondent à des bases azotées d'ADN. Le code suivant convient :

```
def ADN(B):
    n=len(B)
    k=0
    while k<n and base(B[k]):
        k+=1
    return(k==n)
```

**Question 3.3.** Pour fabriquer un brin d'ARNm à partir d'un brin d'ADN, il suffit de remplacer les bases T par des bases U. Le code suivant convient.

```
def ARNm(B):
    A=""
    for c in B:
        if c!="T":
            A+=c
        else:
            A+="U"
    return(A)
```

**Question 3.4.** Pour comparer deux codons, il faut les comparer lettre à lettre, dans la limite des trois présentes. Dès que l'on trouve deux lettres différentes, on arrête le parcours. Le code suivant convient.

```
def codon(A,B):
    i=0
    while i<3 and A[i]==B[i]:
        i+=1
    return(i==3)
```

**Question 3.5.** Pour vérifier si un codon `C` est dans une liste de codons `L`, on aurait pu écrire `C in L`, mais la clause `in` a volontairement été interdite pour reproduire cette fonction. L'idée de l'algorithme est de faire appel à la fonction `codon` sur chaque élément de la liste, tant qu'on a pas trouvé le codon correspondant à `C`. Le code suivant convient.

```
def codons(C,L):
    i=0
    while i<len(L) and not(codon(C,L[i])):
        i+=1
    return(i<len(L))
```

Ce code est évidemment strictement équivalent à :

```
def codons(C,L):
    i,j=0,0
    while i<len(L) and j!=3:
        while j<3 and C[j]==L[i][j]:
            j+=1
        i+=1
    return(i<len(L))
```

qui est nettement moins lisible et ne profite pas de la modularisation offerte avec la définition de la fonction `codon`.

**Question 3.6.** Commençons par préciser qu'une séquences de nucléotides ne peut être un gène que si :

— sa longueur est multiple de 3 ;

— les trois premières bases sont ATG ;

— les trois dernières forment le codon stop (TAA, TGA ou TAG).

Ce que l'on traduit par conjonction des trois propositions sous la forme :

```
def gene(B):
    return(len(B)%3==0 and codon(B[:3], "ATG") and codons(B[-3:], ["TAA", "TAG", "TGA"]))
```

On notera simplement que la coupe des 3 premiers éléments de `B` se note `B[:3]` et celle des 3 derniers `B[-3:]`.

**Question 3.7.** Pour trouver les gènes d'un brin, il est nécessaire de le parcourir par nucléotides jusqu'à trouver le codon ATG. On note `i` l'indice de début. Partant de ce point, il s'agit de « sauter » de codon en codon (par pas de 3 lettres donc) jusqu'à trouver un codon stop. On note `j` l'indice de la première lettre du codon stop. Le gène correspond alors à la coupe `B[i:j+3]`. On l'ajoute à une liste de gènes `G`. On répète cette stratégie tant qu'il reste au moins 6 bases pour définir les codons start et stop. Le code suivant implémente cette idée.

```
def genes(B):
    G=[]
    i=0
    while i<len(B)-6:
        while i<len(B)-6 and not(codon(B[i:i+3], "ATG")):
            i+=1
        j=i+3
        while j<len(B)-3 and not(codons(B[j:j+3], ["TAA", "TAG", "TGA"])):
            j+=3
        G.append(B[i:j+3])
        i=j+4
    return(G)
```

On notera juste le saut de 4 nucléotides à la fin pour tester la base suivante au gène.

**Question 3.8.** Pour définir la séquence des acides aminés d'une protéine à partir d'un gène `G` de longueur $n =$ `len(G)`, il faut commencer par lui retirer ses codons *Met* (start), c'est-à-dire commencer le parcours de `G` à l'indice 3, et stop, en terminant le parcours de `G` à l'indice $n-4$. Après quoi, il faut parcourir le reste du brin `G[3:-3]` de codons en codons, c'est-à-dire par pas de 3 nucléotides de façon à déterminer successivement tous les acides aminés par simple appel de la fonction `AA` sur la coupe de 3 lettres `G[k:k+3]`, $k \in [\![3, n-6]\!]$. C'est ce que réalise cette simple définition par compréhension.

```
def proteine(G):
    return([AA(G[k:k+3]) for k in range(3,len(G)-3,3)])
```



**Question 3.9.** Pour définir la fonction `proteines`, on peut faire appel à la fonction `genes` pour isoler les différents gènes, puis pour chacun déterminer la protéine transcodée par simple appel de la fonction `proteine`. Le code suivant convient.

```
def proteines(B):
    return([proteine(G) for G in genes(B)])
```

*— Fin du corrigé —*