DEVOIR SURVEILLÉ N° 1 **CORRECTION**
Informatique - MPSI 2

Exercice n° 1

1. **a.** Cela correspond à $\boxed{3^3 = 27.}$

b. Cela correspond au quotient de la division euclidienne de 27 par 5, donc il s'agit de $\boxed{5.}$

c. Le premier booléen est associé à $6 \neq 2 \times 3$, qui est faux, le second à $2^3 = 8$ qui est vrai. Avec le connecteur ou, la valeur est donc $\boxed{\textbf{True}.}$

d. Il s'agit du reste de la division euclidienne de 27 par 4 : il s'agit donc de $\boxed{3.}$

2. **a.** $\boxed{\text{'r'}}$

b. $\boxed{\text{On aura un message d'erreur, car la chaîne est trop courte.}}$

c. $\boxed{\text{'for'}}$

d. $\boxed{\text{'tique'}}$

3. **a.** Il faut concaténer les deux listes : $\boxed{[1,2,3,4,5,6,7,8].}$

b. Il faut concaténer la liste L par elle-même 3 fois : $\boxed{[1,2,3,1,2,3,1,2,3].}$

Exercice n° 2

Voir les TP pour avoir la correction de cet exercice.

Exercice n° 3

1. La boucle doit être réalisée 3 fois : La valeur de x est modifiée pour valoir successivement 2, 5 et 26, d'où le résultat $\boxed{26.}$

2. Même principe, mais les valeurs deviennent 10, 101 et enfin 33. Le résultat est donc $\boxed{33.}$

3. $25 < 100$, donc l'appel renvoie $\boxed{0.}$

4. La fonction a été définie de façon récursive : on a tout d'abord $1 + g(202.3, 10)$, $g(202.3, 10)$ renvoie $1 + g(20.23, 10)$ et enfin, $g(20.23, 10) = 1 + g(2.023, 10)$.
Comme $g(2.023, 10)$ renvoie 0, on obtient le résultat : $\boxed{3.}$

Exercice nº 4

1.
```
def syracuse(u0,n) :
    u=u0
    #On réalise n fois la relation de récurrence
    for i in range (n) :
        if u%2==0 :
            u=u/2 #Si le terme est pair, on divise par 2
        else :
            u=3*u+1 #Si le terme est impair, on multiplie par 3 et on
             ajoute 1
    return u #A la fin de la boucle, on a le terme de rang n.
```

2.
```
def tempsVol(u0) :
    u=u0
    n=0 #On initialise le rang.
    #Tant que le terme est différent de 1, on calcule
    while u!=1 :
        if u%2==0 :
            u=u/2 #Si le terme est pair, on divise par 2
            n+=1 #On met à jour le rang
        else :
            u=3*u+1 #Si le terme est impair, on multiplie par 3 et on
             ajoute 1
            n+=1 #On met à jour le rang
    return n #A la fin de la boucle, n est le rang associé à 1.
```

3.
```
def tpsVolMax(n) :
    max=0 #On initialise le rang maximal à 0.
    p=0 #On initialise le terme initial à 0.
    #On parcourt la liste des entiers de 1 à n
    for i in range (1,n+1) :
        m=tempsVol(i) #On calcule le temps de vol
        #Si le temps de vol maximal est dépassé, on met à jour le max
         et le terme initial
        if m>max :
            max=m
            p=i
    return p
```

Exercice n° 5

1.
```
def DivNT(n) :
    L=[] #On créé une liste vide.
    #On parcourt la liste des entiers de 2 à n-1.
    for i in range (2,n) :
        if n%i==0 :
            L.append(i) #On ajoute i à la liste s'il divise n
    return L #A la fin de la boucle, on a la liste des diviseurs non
     triviaux.
```

2.
```
def sommeCarresDivNT(n) :
    L=DivNT(n) #On créé la liste des diviseurs non triviaux.
    somme=0 #On initialise une variable pour calculer la somme.
    #On parcourt la liste L.
    for i in range (len(L)) :
        somme+=L[i]**2 #On ajoute le carré du diviseur d'index i.
    return somme #A la fin de la boucle, on a la somme des carrés
     des diviseurs non triviaux.
```

3.
```
def ListeEgalite(n) :
    L=[] #On créé une liste vide.
    #On parcourt la liste des entiers de 2 à n.
    for i in range (2,n+1) :
        if i==sommeCarresDivNT(i) :
            L.append(i) #On ajoute i à la liste si la somme des carrés
             des diviseurs non triviaux est égale à i.
    return L #A la fin de la boucle, on a la liste des entiers
     recherchés.
```

4. On peut conjecturer que les entiers égaux à la somme des carrés de leurs diviseurs non triviaux sont les carrés des nombres premiers.

Exercice n° 6

1. Le codage de L est $\boxed{[2,2,1,3,4,1,1,-5,2,0].}$
   La liste codée par C est $\boxed{[1,1,-1,-1,-1,0,0,0,0,3].}$

2.
```
def decoder(C) :
    L=[] #On initialise L par une liste vide
    #On réalise la boucle len(C)//2 fois
    for i in range (len(C)//2) :
        for  j in range (C[2*i]) :
            L.append(C[2*i+1])
    return L
```

3.
```
def longueurBloc(L,i) :
    c=0 #On initialise un compteur c
    j=i #On initialise un index j à i
    #On parcourt la liste tant que c'est le terme L[i] et que l'on
     n'est pas arrivé au bout
    while j<len(L) and L[j]==L[i] :
        c=c+1
        j=j+1
    return c
return L
```

4.
```
def codage(L) :
    C=[] #On initialise une liste L
    i=0 #On initialise un index i à 0
    #On parcourt la liste tant que l'on n'est pas arrivé au bout
    while i<len(L) :
        j=longueurBloc(L,i) #On détermine le nombre d'éléments
         identiques à partir de l'index i
        C.append(j) #On ajoute le nombre de répétitions.
        C.append(L[i]) #On ajoute l'élément qui a été répété.
        i=i+j #On met à jour l'index pour aller au terme différent
         suivant
    return C
```

Exercice nº 7

1.
```
def estADN(s) :
    i=0
    #Tant qu'on a un caractère valide
    while i<len(s) and (s[i]=="A" or s[i]=="T" or
     s[i]=="C" or s[i]=="G") :
        i += 1 #On passe au suivant
    return i==len(s) #Si on a fini s, tous les caractère
     sont valides
```

2.
```
def baseComp(base) :
    if base=="A" :
        return "T"
    elif base=="C" :
        return "G"
    elif base=="G" :
        return "C"
    else:
        return "A"
```

3.
```
def estDoubleHelice(s1,s2) :
    #Si les deux chaînes n'ont pas la même longueur
    if not len(s1)==len(s2) :
        return False
    else :
        for i in range (len(s1)) :
            #Si le complémentaire de s1[i] n'est pas la ième
             lettre de s2 en partant de la fin
            if not baseComp(s1[i])==s2[len(s2)-1-i] :
                return False
        return True
```

4.
```
def seqCompInv(s) :
    res = "" #Séquence complémentaire inversée
    for i in range (len(s)) :
        res = baseComp(s[i]) + res #On ajoute la base
         complémentaire devant
    return res
```

5.
```
def transcrit(s) :
    res = "" #Séquence ARN
    for i in range (len(s)) :
        #Un T est remplacé par un U
        if s[i]=="T" :
            res += "U"
        #Les autres bases sont
         inchangées
        else :
            res += s[i]
    return res
```

6.
```
def OccurrencesCodon(s,cod) :
    arn = transcrit(s) #Séquence ARN
     correspondant à s
    cpt = 0 #Compteur de codons cod
    i=0
    while i<len(arn) :
        #Si le prochain codon (triplet) est cod
        if arn[i:i+3]==cod :
            cpt += 1
        i +=3 #On passe au triplet suivant
    return cpt
```

7.
```
def aPourCodons(acides_amines,codons,a) :
    i=0
    #On parcourt acides_amines jusqu'à trouvé a (qui est
     dans la liste)
    while acides_amines[i] != a :
        i += 1
    return codons[i] #Liste des codons codant l'acide
     aminé d'indice i
```

8.
```
def codeAcideAmine(acides_amines,codons,cod) :
    for i in range (len(codons)) :
        for j in range (len(codons[i])) :
            #Si on a trouvé cod
            if codons[i][j]==cod :
                return acides_amines[i] #On renvoie l'acide
                 aminé correspondant
```

9.
```
def proteine(acides_amines,codons,s) :
    arn = transcrit(s) #On transcrit l'ADN en ARN
    res = [] #Protéine
    i=0
    while i<len(arn) :
        #On ajoute l'acide aminé codé par le (i+1)ème codon
        res.append(codeAcideAmine(acides_amines,codons,arn[i:i+1]))
        i += 3
    #On vérifie si l'ARN termine par un codon "Stop"
    if res[-1]==acides_amines[0] :
        return res
    else :
        return False
```