Devoir surveillé 01

- Correction -

Questions diverses

1. $(365)_{10} = (1\,0110\,1101)_2 = (16D)_{16}$.
2. Suite de Catalan :
   (a) $c_2 = c_0c_1 + c_1c_0 = 1 + 1 = 2$
   (b)
```
def Catalan(n):
    listec=[1] #initialisation de la liste des termes
    for i in range(1,n+1):
        s=0
        for j in range(i):
            s+=listec[j]*listec[i-1-j] #Calcul de la somme en utilisant les termes précédents
        listec.append(s) #On ajoute le nouveau terme à la liste
    return listec[-1] #On retourne le dernier terme
```
   (c) Le coût et la complexité de cette fonction peuvent être évalués en comptant le nombre d'opérations unitaires : $C(n) = 2 + 2n + (1 + 2 + 3 + \ldots + n - 1)) = 2 + 2n + \frac{n(n-1)}{2} = \boxed{\mathcal{O}(n^2)}$
3. Dichotomie
   (a)
   - Avant d'entrer dans la boucle : g=0, d=9
   - Après le premier while : m=4, g=4, d=9, **[7,7,9,13,15,17]** ;
   - Après le deuxième while : m=6, g=4,d=6, **[7,7,9]** ;
   - Après le troisième while : m=5, g=5,d=6, **[7,9]**.

   On sort alors de la boucle. La condition L[g] est alors vraie, et on renvoie l'indice 5, qui est l'indice le plus élevé du nombre 7 dans la liste L.
   (b) Soit P la propriété : « x est dans l'intervalle $[\mathtt{L[g]}, \mathtt{L[d]}[$ ».
   - Initialisation : Le test `if x < L[g] or L[d] < x: return None` permet de s'assurer que x est compris dans $[\mathtt{L[g]}, \mathtt{L[d]}[$ juste avant d'entrer dans la boucle.
   - Hérédité : Supposons P vraie à la fin d'une étape de boucle. Dans ce cas, au cours de l'étape suivante soit L[m]>x et L[m] devient L[d], donc x<L[d], soit L[m]≥x et dans ce cas L[m] devient L[g] et x≥L[g]. Finalement, à la fin de la boucle suivante, on a toujours x∈ $[\mathtt{L[g]}, \mathtt{L[d]}[$
   - Par récurrence, la propriété P est vraie à chacune la fin de toutes les étapes de boucle : c'est un invariant de boucle.

   (c) Au cours des étapes de boucle, on divise la longueur de l'intervalle de recherche (d-g) par 2 à chaque étape, en s'assurant que le nombre cherché (x) reste toujours à l'intérieur de l'intervalle de recherche $[\mathtt{L[g]}, \mathtt{L[d]}[$ (invariant de boucle). La longueur de l'intervalle de recherche est donc une suite strictement décroissante, donc il existe un nombre d'étapes fini tel que cette longueur atteigne 1, soit d-g=1. Le condition d'entrée dans la boucle while devient alors fausse, et on sort de la boucle. On renvoie alors l'indice correspondant si on a trouvé x dans le seul terme restant dans l'intervalle de recherche, et `None` sinon.

Exercice 1 : Loi de Benford

1.
```
#Après ouverture du fichier, on lit chaque ligne que l'on découpe en deux termes
#via split. On affecte alors à deux listes (nom de ville et population).
commune=open('communes.txt', 'r')
liste=commune.readlines()

lcom=[]
lpop=[]
for k in range(len(liste)):
    temp=liste[k].split(',')
    lcom.append(temp[0])
    lpop.append(int(temp[1]))#conversion en entier, car str sinon !
```

2.
```
def pop_sup(lpop,n):
    #On balaye la liste lpop en comparant la population à n
    villes=0
    for pop in lpop:
        if pop>=n:
            villes+=1
    return villes
```

3.
```
def villes_mortes(lcom,lpop):
    mortes=[]
    for k in range(len(lcom)):
        if lpop[k]==0:
            mortes.append(lcom[k])
    return mortes
```

4.
```
def commune_plus_peuplee(lcom,lpop):
    m=0 #on initialise la position du max
    for k in range(len(lpop)): #on boucle sur les populations
        if lpop[k]>lpop[m]: #on teste qui est le maximum
            m = k #on affecte le nouveau maximum
    return lcom[m] #on retourne le nom de la ville la plus peuplée
```

5.
```
def lprems(L):
    liste=[]
    for nb in L:
        if nb!=0: #on ne traite pas les valeurs nulles.
            nb=str(nb) #Conversion en str pour l'extraction du premier chiffre
            liste.append(nb[0])
    return liste
```

6.
```
def proportion(L):
    liste=lprems(L)
    n=len(liste)
    listeprop=[0]*9
    for k in liste: #on décompte entre 1 et 9 dans la liste
        listeprop[k-1]+=1/n #k-1 car l'index de la liste commence à 0
                                    #et /n pour que ce soit normalisé
    return listeprop
```

7. L'énoncé nous précise que $p(c) = \log(1 + 1/c)$, donc $10^{p(c)} \times (c)$ (calculé à la deuxième ligne des instructions de l'énoncé), doit être égal à $c(1 + 1/c) = c + 1$. C'est bien ce que l'on obtient : on a une droite de coefficient directeur 1, et d'ordonnée à l'origine 1 (en interpolant).

Exercice 2 : Hauteurs de marées

1. `data[2][1]` renvoie `'00:20'`. (Il renvoie pour la troisième sous-liste le deuxième élément).

2.
```
h_17mai16=[]
h_juin16=[]
dates1=[]
dates2=[]
for x in data:
    if x[0]=="17/05/2016":
        h_17mai16.append(x[2])
        dates1.append(int(x[1][0:2])+int(x[1][3:5])/60) #Conversion de la date en heures
    if x[0][3:10]=="06/2016":
        h_juin16.append(x[2])
        dates2.append(int(x[0][0:2])-1 + int(x[1][0:2])/24+int(x[1][3:5])/(24*60))
        #Conversion de la date en jours en partant de 0
```

J'ai beaucoup utilisé les tranches de listes pour ce programme, mais il était aussi possible d'appliquer `.split('/')` à la date ou `.split(':')` à l'heure afin d'extraire les chiffres.

3.
```
plt.plot(dates1,h_17mai16)
plt.show()

plt.plot(dates2,h_juin)
plt.show()
```

On peut aussi proposer d'utiliser subplots si on veut les mettre côte à côte (on rappelle : ligne, colonne, numéro de graphe) (non demandé ici) :

```
plt.subplot(121)
plt.plot(dates1,h_17mai16)
plt.subplot(122):
plt.plot(dates2,h_juin)
plt.show()
```

4.
```
def moyenne(liste):
    s=0
    n=len(liste)
    for k in range(n):
        s+=liste[k]
    return s/n
```

5. On peut par exemple programmer explicitement la boucle, ou faire une simple liste par compréhension : `height=[x[2] for x in data]`

6. La fonction complétée s'écrit :

```
def construction_successeurs(height):
    m=moyenne(height)
    liste_PMM=[]
    for i in range(1,len(height))):
        if height[i-1]<m and height[i]>=m:
            liste_PMM.append(i)
    return liste_PMM
```

7. Un moyen très simple est d'utiliser la liste des passages par le niveau moyen en montée pour « découper » des sous listes de `height`.

```
def decompose_maree(height):
    lmaree=[]
    listePMM=construction_successeurs(height)
    for i in range(len(listePMM)-1):
        lmaree.append(height[listePMM[i]:listePMM[i+1]])
    return lmaree
```

8.
```
def marnage(height):
    listeamp=[]
    lmaree=decompose_maree(height)
    for maree in lmaree:
        listeamp.append(max(maree)-min(maree))
    return listeamp
```

9. Cette fonction a une complexité égale à celle de `decompose_maree` car l'extraction des amplitudes est moins coûteuse (le nombre de marées est nécessairement plus faible que le nombre de hauteurs de `height`). Or `decompose_maree` parcourt la liste `height` deux fois successivement : une fois pour extraire les PMM, une autre fois pour effectuer la décomposition en marées. Le complexité de cette fonction est donc **linéaire** en fonction du nombre de termes de `height`.