Devoir surveillé 01

- Correction -

Questions simples (ou pas)

1. Il s'agit de `import numpy as np`.
2. Le dernier élément s'écrit `L[len(L)-1]` ou `L[-1]` : donc `print(L[-1])` convient.
3. On utilise le code suivant (on pouvait ou non vectoriser la fonction, le choix a été ici fait d'utiliser la fonction racine de numpy) :

```
import matplotlib.pyplot as plt
def f(x):
    return 1/(np.sqrt(x**2-1))

x=np.linspace(2,5,200)
y=f(x)
plt.plot(x,y)
plt.show()
```

4. On trouve $n=8$ car $n=\sum_{i=0}^{3}\left(\sum_{j=0}^{1} i-j\right)=\sum_{i=0}^{3} i+\sum_{i=0}^{3}(i-1)=2\sum_{i=0}^{3} i-4=2\times 6-4=8.$

5.
```
def suite(n):
    if n==0:
        return 1
    if n==1:
        return 3
    L=[1,3]
    for k in range(2,n+1):#attention au choix d'indice, on doit aller jusqu'à k=n
        L.append(2*L[k-1] - L[k-2])
    return L[n]
```

Problème 1 : annuaire téléphonique

1. On ouvre le fichier, on le lit entièrement, et pour chaque ligne, on utilise la fonction split. Attention au fait que le ' n' doit être enlevé, par exemple avec une utilisation de la tranche :

```
fichier=open('annuaire.txt', 'r')
liste=fichier.readlines()
liste_num=[]
for ligne in liste:
    temp=ligne.split(';')
    liste_num.append([temp[0], temp[1][0:10]])
ficher.close()
```

2. On effectue un balayage complet de la liste, principe de la recherche séquentielle vue en cours.

```
def cherche_numero(num):
    for k in range(len(liste_num)):
        if liste_num[k][1]==num:
            return k
    return None
```

3. Dans le meilleur des cas, le numéro recherché est le premier de la liste, donc la complexité est en $\mathcal{O}(1)$. Dans le pire des cas on a dû balayer la liste en entier, la boucle est de coût $\alpha \times N$ avec α entier, donc la complexité est en $\mathcal{O}(N)$.
4. Comme on utilise plus d'une fois la valeur de l'indice associé à `num`, il est préférable de poser une variable associée à `cherche_numero(num)` pour éviter de faire deux fois le calcul :

```
def nom_du_num(num):
    pos=cherche_numero(num)
    if pos==None:
        return None
    return(liste_num[pos][0])
```

5. On utilise à nouveau la fonction de tranche de listes, utilisable également sur les chaînes de caractères :

```
def nat_to_inter(num):
    return '+33'+num[1:10]
```

6. On applique la fonction précédente :

```
for k in range(len(liste_num)):
    liste_num[k][1]=nat_to_inter(liste_num[k][1])
```

7. Il s'agit de l'invariant `liste_num[g][1]<num<=liste_num[d][1]`. D'après la fin du code de dichotomie, au moment où il ne reste que deux termes dans la restriction de la liste, on renvoie celui de droite. D'autre part le test initial permet de s'assurer que le tout premier terme de la liste n'est pas celui recherché, pour respecter l'invariant avant de rentrer dans la boucle while.

8. On complète, comme dans le cours.

```
def nom_du_num_dicho(num,liste_num):
    g=0
    d=len(liste_num)-1
    nb=0
    if liste_num[0][1]==num: #permet de respecter l'invariant de boucle
        return liste_num[0][0]
    while d-g>1:#tant que la restriction de la liste contient plus de 2 termes
        m=(g+d)//2
        if liste_num[m][1]<num:
            g=m
        else:
            d=m
        nb+=1
    if liste_num[d][1]==num:
        return liste_num[d][0]
    else:
        return None
```

9. À chaque tour de boucle (et donc d'incrémentation de la variable nb), on divise l'intervalle par 2. À la fin, lorsque l'on sort de la boucle while, il ne reste que deux éléments, donc nb vérifie $1 = \frac{10^6}{2^{\text{nb}}}$ soit $2^{(\text{nb})} = 10^6$ soit en composant par $\log_2$: $\text{nb} = 6log_2(10) = 6 * 3,3 = 19,8$. Ainsi $\text{nb} = 20$ (lorsqu'on divise un intervalle de longueur impaire, on arrondit toujours le résultat à l'excès : exemple 3 éléments, $3/2 = 1.5$ arrondi à 2).

Problème 2 : recherche de la médiane d'une liste

1. On a 11 termes, donc il faut prendre le 6e terme, donc 8, si on ordonne par ordre croissant tab.

2.
```
def nbPlusPetit(tab,val):
    compteur=0
    for nombre in tab :
        if val>nombre:
            compteur+=1
    return compteur
```

3. Attention, il n'existe pas que le cas où `nbPlusPetit(tab[k]) =nbPlusGrand(tab[k])`, s'il y a plusieurs fois la valeur de la médiane dans la liste, et selon que la longueur de la liste est paire ou impaire. Le plus simple est d'écrire :

```
def mediane(tab):
    """On va se servir des fonctions nbPlusPetit et nbPlusGrand pour déterminer le terme qui est la val
    c'est-à-dire la valeur telle que nbPlusPetit(tab[k]) et nbPlusGrand(tab[k]) <=len(tab)//2"""
    for k in range(len(tab)):
        if abs(nbPlusGrand(tab,tab[k])<=len(tab)//2 and nbPlusPetit(tab,tab[k]))<=len(tab)//2 :
            return tab[k]
```

On pouvait aussi proposer :

```
def mediane(tab):
    """On va se servir des fonctions nbPlusPetit et nbPlusGrand pour déterminer le terme qui est la val
    c'est-à-dire la valeur telle que nbPlusPetit(tab[k]) =nbPlusGrand(tab[k]) (-1 si len(tab) est pair)
    for k in range(len(tab)):
        if abs(nbPlusGrand(tab,tab[k])-nbPlusPetit(tab,tab[k]))<=1 :
            return tab[k]
```

4. Complexité en $\mathcal{O}(n^2)$ car `nbPlusPetit` est de complexité $\mathcal{O}(n)$ car on balaie toute la liste, et `mediane` s'en sert au maximum n fois (boucle for).

5. Attention, i_1 et i_2 doivent tenir compte des valeurs initiales a et b, donc $i_1 \neq$ `len(debut)` par exemple.

```
def partition(tab,a,b,indicePivot):
    debut,milieu,fin=[],[],[]
    i1=a
    i2=b
    for i in range(a,b):
        if tab[i]<tab[indicePivot]:
            debut.append(tab[i])
            i1+=1
        elif tab[i]>tab[indicePivot]:
            fin.append(tab[i])
            i2-=1
        else:
            milieu.append(tab[i])
    tab[a:b]=debut+milieu+fin #on modifie tab par effet secondaire
    return i1,i2
```

6. Le tableau exemple étant $[3, 14, 5, 6, 1, 14, 21, 8, 9, 2, 8]$, le premier pivot est d'indice `n=11//2 = 5`, donc 14.
   - À l'issue du premier tour de boucle `while`, la liste est modifiée par `partition` et devient :
     $$[3, 5, 6, 1, 8, 9, 2, 8, 14, 14, 21]$$
     Donc $i_1 = 8$ et $i_2 = 10$. Comme $i_1 > 5$, $b = 8$ et $a = 0$.
   - À l'issue du deuxième tour de boucle, on part de la liste modifiée (impératif ! !). Le nouveau pivot est d'indice 4 donc 8 :
     $$[3, 5, 6, 1, 2, 8, 8, 9, 14, 14, 21]$$
     Donc $i_1 = 5$, $i_2 = 7$. Les deux premiers tests ne sont pas vérifiés, donc $a = 5$, $b = 7$ et le programme retourne `tab[5]` c'est-à-dire 8.

7. On réalise une démonstration par récurrence :
   - **Initialisation** : « `tab[0:len(tab)]` contient l'ensemble des valeurs de `tab`, donc contient nécessairement la médiane.

- **Hérédité** : on suppose l'invariant vérifié au précédent tour de boucle `while`. On note a et b les indices obtenus au tour de boucle précédent On doit alors distinguer trois cas (dans l'ordre des tests de la boucle), connaissant le fonction de la fonction `partition` :
  - soit i_1 est supérieur à `len(tab)//2`, auquel cas la liste `tab[a:i1]` contient nécessairement la médiane comme $I_{i_1} > \lfloor \frac{n}{2} \rfloor$ en notant `n=len(tab)` ; d'où le choix $b = i_1$;
  - soit i_2 est inférieur à `len(tab)//2`, auquel cas la liste `tab[i2:b]` contient nécessairement la médiane comme $S_{i_2} > \lfloor \frac{n}{2} \rfloor$ (`len(tab[i2,len(tab)])>len(tab)//2`) ; d'où le choix $a = i_2$;
  - si les deux conditions précédentes ne sont pas vérifiées, on a donc à la fois $I_{i_1} \leq \lfloor \frac{n}{2} \rfloor$ et $S_{i_2} \leq \lfloor \frac{n}{2} \rfloor$, donc toutes les valeurs comprises entre les indices i_1 et i_2 sont des médianes, on peut renvoyer `tab[a]`.
- **Terminaison** : comme chaque tour de boucle restreint l'intervalle d'étude (soit a croît, soit b décroît), on arrive nécessairement au 3e cas lorsqu'on isole la médiane, avec un garde-fou (le test de la boucle `while`) que $a < b - 1$, c'est-à-dire qu'on garde au plus un seul terme dans `tab[a:b]`.