I) Questions de cours

1. Pour afficher le contenu d'une variable on utilise la fonction *print*.
2. Une boucle *for* est un cas particulier d'une boucle *while*. La boucle *for* est généralement utilisée lorsqu'on connaît le nombre d'itérations a priori ; dans le cas contraire une boucle *while* est plus souvent utilisée.
3. Pour écrire un commentaire dans le code *Python* on utilise le symbole #.
4. On écrit $O(n)$.
5. On a une complexité de $O(n^2)$.
6. Tri par insertion, tri par sélection, tri à bulles.
   Le tri par insertion consiste à insérer à la « bonne place » parmi les éléments déjà triés l'élément suivant. Pour le tableau initial [3,2,5,1] :
   - le sous-tableau [3] est déjà trié et on place l'élément suivant, c'est-à-dire 2 : [2,3,5,1] ;
   - on place l'élément 5 dans le sous-tableau [2,3] : [2,3,5,1] ;
   - on place l'élément 1 dans le sous-tableau [2,3,5] : [1,2,3,5].

II) Préliminaires

1. On peut écrire les deux instructions suivantes :
   - `from math import log, sqrt, floor, ceil`
   - `print(log(0.5))`
2. On peut écrire le code suivant :

```
def sont_proches(x,y):
    atol=1e-5
    rtol=1e-8
    return abs(x-y)<= atol+abs(y)*rtol
```

3. La commande *mystere(1001,10)* retourne 3, la plus grande puissance de 10 inférieure ou égale à 1001.
4. La fonction retourne $\left\lfloor \frac{\ln x}{\ln b} \right\rfloor$.

III) Génération de nombres premiers

5. La structure de boucle principale est itérée $\lfloor \sqrt{N} \rfloor$ fois. Pour chaque itération on marque les multiples de l'élément i à traiter ; ce nombre de multiple est majoré par N/i qui est lui même majoré par N. La complexité est donc majorée par un $O(N\sqrt{N})$.
   On peut remarquer que cette majoration est très grossière.
6. Il suffit de traduire en langage *Python* l'algorithme proposé. On suppose bien sûr que les fonctions *sqrt* et *floor* ont été importé depuis le module *math*.

```
def crible_erato(n):
    liste_bool=[True]*n
    liste_bool[0]=False # 1 n'est pas un nombre premier
    for i in range(2,floor(sqrt(n))+1):
        if liste_bool[i-1]: # attention, il y a un décalage entre l'indice et l'entier tra
            for k in range(i+i,n+1,i):
                liste_bool[k-1]=False
    return liste_bool
```

IV) Compter les nombres premiers

7. On peut écrire la fonction suivante :

```
def pi(n):
    liste_bool=crible_erato(n)
    L=[[1,0]]
    for i in range(2,n+1):
        L.append([i,L[i-2][1]+int(liste_bool[i-1])])
    return L
```

8. On peut écrire le code suivant :

```
def verif_pi(N):
    L=pi(N)
    for n in range(5393,N+1):
        if n/(log(n)-1) >= L[n-1][1]:
            return False
    return True
```

V) Vérification

9. Comme on va utiliser la fonction `crible_erato` on a besoin de connaître la plus grande valeur de la liste `L`. Pour cela on peut utiliser une fonction *Python* déjà existante ou bien écrire notre propre fonction. Je vous propose d'utiliser notre fonction.

```
def mon_max(L):
    m=L[0]
    for e in L:
        if e>m:
            m=e
    return m

def est_liste_premier(L):
    N=mon_max(L):
    l_premier=crible_erato(N)
    for e in L:
        if not l_premier[e-1]:
            return False
    return True
```

10. La fonction suivante utilise le tri par insertion pour effectuer le travail.

```
def trier_liste(L):
    n=len(L)
    for i in range(1,n):
        k=i
        e=L[i]
        while k>0 and L[k-1]>e:
            L[k]=L[k-1]
            k=k-1
        L[k]=e
```

11. On peut écrire le code suivant :

```
def complete(L,N):
    l_premier=crible_erato(N)
    trier_liste(L)
    compteur_premier=0
    for i in range(N):
        if l_premier[i]:
            if L[compteur_premier]==i+1: # il y a toujours un décalage
                compteur_premier+=1
            else:
                return False
    return True
```