Feuille de TD n° 3 -

La fonction `assert` sert à tester si les variables en entrée d'un algorithme respectent les conditions attendues (entiers, flottants, listes, tuple, liste non-vide, entier positif, entier non-nul,...). Par exemple, `assert( type(n)==int)` teste si la variable `n` est un entier.
Si `type(n)==int)` est vrai, rien ne se passe et l'algorithme continue.
Si `type(n)==int)` est faux, la fonction est interrompue et Python renvoie une erreur.
Cela permet de s'assurer qu'une fonction n'est pas exécutée avec des variables pour lesquelles elle n'est pas prévue à la base. C'est l'équivalent en informatique du domaine de définition d'une fonction f.

Exercice 1.

1. Ecrire un test avec `assert` qui vérifie si **t** est un tuple.
2. Ecrire un test avec `assert` qui vérifie si `n` est un entier non-nul.
3. Donner un exemple de fonction usuelle dont l'implémentation informatique nécessite d'utiliser `assert`.
4. Ecrire un test avec `assert` qui vérifie si **L** est une liste non-vide.
5. Ecrire un test avec `assert` qui vérifie si **a,b,c** sont trois réels tels que $a \leq b \leq c$.
6. Ecrire un test avec `assert` qui vérifie si `n` est un entier qui est un carré. On pourra s'aider de `math.sqrt`.

Pour obtenir des informations sur une fonction dans un module (domaine de définition, variables possibles, forme du résultat), on utilise la commande `help(module.fonction)`. Elle affiche les instructions enregistrées pour la fonction associée.
On peut de même créer des instructins pour une fonction Python. Il faut pour cela écrire les instructions entre `"""` et `"""` au tout début de la fonction.

Exercice 2. L'algorithme d'Euclide repose sur le fait que pour $a = bq + r$, on a $pgcd(a, b) = pgcd(b, r)$.

1. Compléter l'algorithme suivant pour qu'il soit fonctionnel.

```
def euclide(a,b):
    """Entrée : deux entiers naturels a et b tels que a >= b Sortie : pgcd(a,b)"""
    assert(a >= b)
    while b != 0:
        a, b =
    return a
```

2. Que renvoie `help(euclide)` ?
3. Déterminer une fonction de terminaison pour la boucle `while`.
4. Notons a_k, b_k les valeurs de a, b après k exécutions de la boucle `while`.
   Déterminer HR_k un invariant de boucle pour la boucle `while`.
5. Lorsque la boucle `while` se termine (après n exécutions), qu'obtient-on pour a_n et b_n ?
   Prouver la correction de l'algorithme.

Exercice 3 (Primalité : Algorithme naïf)**.**

1. Écrire une fonction `premier` qui prend en entrée un nombre entier positif n et qui renvoie `True` s'il est premier, et `False` sinon.
2. Tester cette fonction sur les entiers de 1 à 20.

```
def premier(n):
    ...

for i in range(1,21):
    if (premier(i)):
        print(i,end=' ')
print() # pour le retour à la ligne
```

Proposition : Soit $n \in \mathbb{N}^*$. Si n n'est pas premier alors il possède un diviseur entre 2 et $\lfloor \sqrt{n} \rfloor$.

Exercice 4 (Primalité : Algorithme moins naïf)**.**

1. En utilisant la proposition précédente, écrire une fonction `est_premier` qui prend en entrée un nombre entier positif et qui renvoie `True` s'il est premier, et `False` sinon.
   Cette fonction effectuera moins de calculs que la fonction `premier`.
2. Tester cette fonction sur les entiers de 1 à 20.

```
def est_premier(n):
    ...

for i in range(1,21):
    if (est_premier(i)):
        print(i,end=' ')
print() # pour le retour à la ligne
```

3. Combien de nombres premiers sont inférieurs à 1000 ?
4. Déterminer le plus petit nombre premier supérieur à 1000. On utilisera une boucle `while`.

La méthode du crible d'Ératosthène permet de déterminer tous les nombres premiers inférieurs à N, pour N un entier fixé. Pour ceci on écrit tous les nombres entiers de 0

à N, et on barre petit à petit les nombres qui ne sont pas premiers :
On barre 0 et 1 qui ne sont pas premiers.
Le nombre 2 n'est pas barré, donc il est premier. On barre tous les multiples stricts de 2 qui eux ne sont pas premiers.
Le nombre 3 n'est pas barré, donc il est premier. On barre tous les multiples stricts de 3.
Le nombre 4 est barré, il n'est pas premier, on passe au nombre 5. 5 est premier, on barre ses multiples stricts, etc.
Une fois l'algorithme terminé, les nombres non barrés sont exactement les nombres premiers inférieurs à N.

On représente cela par une liste de booléens : `C=[True,True,...]`. Au début la liste ne contient que la valeur `True` (personne n'est barré). À la fin de l'algorithme, chaque terme `C[n]` vaut `True` si n est premier et `False` sinon.

Exercice 5 (Crible d'Eratosthène)**.**

1. Ecrire une fonction `Crible` qui prend en entrée un nombre `N` et qui renvoie la liste `C` obtenue avec le crible d'Eratosthène.
2. Vérifier que la fonction est correcte pour les nombres premiers inférieurs à 20.

```
def Crible(N):
    ...

C1=Crible(20)
for n in range(20):
    if C1[n]:  #Si C1[n]==True, càd si n est premier
        print(n,end=' ')
print() # Pour le retour à la ligne
```

3. Retrouver le nombre de nombres premiers inférieurs à 1000.

Les trois algorithmes traitent de la même question mais de deux façons très différentes. Qui est le plus efficace ?
Une façon de mesurer l'efficacité entre deux fonctions Python est de regarder le temps qu'il faut à chacune d'entre elles pour être exécutées (leur temps d'exécution).
On mesure le temps grâce à la fonction time du module time. Le résultat est donné en secondes, il s'agit du temps de travail du processeur.

```
from time import time
t1=time()

... # Exécution de l'algorithme

t2=time()
print("Temps :",t2-t1)
```

Il faut tout d'abord s'assurer que les fonctions fournissent exactement la même information.
Le tableau `C` déterminé par le crible d'Ératosthène peut être obtenu avec la fonction `premier` de la façon suivante :

```
def premier2(n):
    C=[]
    for n in range(N+1):
        C.append(premier(n))
    return C
```

On écrit de même une fonction `est_premier2` qui renvoir une liste, à partir de la fonction `est_premier`.

Exercice 6.

1. Importer le module `time`.
   Créer une fonction `TestTemps(N)` qui mesure le temps de calcul de la liste C selon N pour chacune des trois méthodes, et qui renvoie les trois temps obtenus.
2. Comparer les temps d'exécution pour $N = 100$, $N = 10^4$, $N = 10^5$.
   Que remarque-t-on ?

Exercice 7 (Conjecture de Goldbach)**.** Cette conjecture affirme que tout nombre pair supérieur à 3 est somme de deux nombres premiers. Elle n'a pas été démontrée à l'heure actuelle.

1. Ecrire une fonction `Goldbach` qui renvoie la liste de tous les nombres entiers inférieurs à n, pairs, et somme de deux nombres premiers.
   On utilisera la fonction `Crible`.
2. Vérifier que la conjecture de Goldbach est vraie pour tous les nombres pairs inférieurs à 2^{20}.
   Tester d'abord avec 2^{10} et 2^{13} pour s'assurer que la fonction écrite ne prend pas trop de temps. Si la fonction n'est pas bien optimisée, le test jusqu'à 2^{20} peut prendre quelques jours (contre quelques minutes sinon).

Exercice 8 (Indicatrice d'Euler)**.** On définit l'indicatrice d'Euler d'un entier n par $\phi(n) = Card(\{k \in \{1, \dots, n\} \text{ t.q. } pgcd(k, n) = 1\})$.

1. Ecrire une fonction `Euler` qui renvoie $\phi(n)$ l'indicatrice d'Euler de n.
2. (Oral MP 2015) Expliquer pourquoi l'algorithme suivant fonctionne en exhibant un invariant de boucle c'est-à-dire une propriété H_i qui est vérifiée à chaque étape i de la boucle principale.

```
def premAvec(n):
    """Renvoie la liste des entiers 0 <= k < n premiers avec n"""
    if n == 1:
        return [0]
```

```
table = [0] + [1 for i in range(1,n)] # objectif final: table[i] == 1 ssi pgcd(n,i) == 1
for i in range(2, n//2 + 1):
    if table[i] == 1 and n%i == 0:
        for k in range(1, (n-1)//i + 1):
            table[k*i] = 0
return [i for i in range(1,n) if table[i] == 1]
```