SUITE DES PUISSANCES MODULO m

Partie I — Construction de la liste

Dans l'algorithme, la variable y parcourt les puissances de x : elle vaut successivement x^1 (modulo m), x^2 (modulo m), x^3 (modulo m), etc.. On continue la boucle tant que la puissance n'est pas dans la liste. La condition de continuation est donc `not DéjàVues[`y`]`, qui est vrai si et seulement si `DéjàVues[`y`]` est faux, c'est-à-dire la valeur y n'a pas encore été rencontrée.

```
def Puissances(m, x) :
    DéjàVues = [False] * m
    L = [] ; y = x
    while not DéjàVues[y] :
        L.append(y) ; DéjàVues[y] = True
        y = x * y % m
    L.append(y)
    return L
```

Partie II — Générateurs modulo m

Lorsque x est un générateur (modulo m), la liste de ses puissances contient tous les nombres entre 1 et $m-1$. Comme cette liste ne contient *que des éléments distincts* sauf le dernier (on a arrêté la construction au premier « double »), elle est alors de longueur $m-1+1=m$. On a donc la caractérisation suivante : x est un générateur (modulo m) si et seulement si la longueur de la liste `Puissances(`m`, x)` est égale à m.

```
def TrouverGénérateur(m) :
    for x in range(1, m) :
        L = Puissances(m, x)
        if len(L) == m :
            return x
    return None
```

Comme suggéré, faisons des essais.

```
>>> [(m, TrouverGénérateur(m)) for m in range(2, 30 + 1)]
[(2, 1), (3, 2), (4, None), (5, 2), (6, None), (7, 3), (8, None),
 (9, None), (10, None), (11, 2), (12, None), (13, 2), (14, None),
 (15, None), (16, None), (17, 3), (18, None), (19, 2), (20, None),
 (21, None), (22, None), (23, 5), (24, None), (25, None), (26, None),
 (27, None), (28, None), (29, 2), (30, None)]
```

Une conjecture s'impose : il y a un générateur modulo m si et seulement si m est un nombre premier.

Pour trouver un m pour lequel le plus petit générateur est au moins égal à g, on utilise le programme `TrouverGénérateur` sur tous les m à partir de $m=2$. Si x désigne le résultat de `TrouverGénérateur(`m`)`, on ne peut pas directement tester x `<` g, car si x vaut `None`, la comparaison n'a pas de sens...

```
def PlusPetitGénérateurAuMoins(g) :
    m = 2 ; x = TrouverGénérateur(m)
    while x == None or x < g :
        m += 1 ; x = TrouverGénérateur(m)
    return m
```

...en revanche on peut écrire x `!=` g : soit x vaut `None` et alors il est différent de g, soit c'est un nombre et on peut le comparer à g. C'est ainsi en Python : on peut faire une comparaison avec `==` ou `!=` entre deux objets qui n'ont pas forcément le même type, alors qu'on ne peut pas faire une comparaison avec `<`, `>`, `<=` ou `>=` lorsque les objets ne sont pas du même type.

```
def PlusPetitGénérateurÉgal(g) :
    m = 2 ; x = TrouverGénérateur(m)
    while x != g :
        m += 1 ; x = TrouverGénérateur(m)
    return m
```