

TP08

February 17, 2022

1 Division euclidienne

On rappelle que si $a, b \in \mathbb{N}$ avec $b \neq 0$, il existe un unique couple (quotient, reste) $(q, r) \in \mathbb{N}^2$ tel que $a = bq + r$ et $0 \leq r < b$. On appelle cette opération une *division euclidienne*.

Un algorithme pour calculer ce couple consiste à remarquer que $a = b \times 0 + a = b \times 1 + (a - b) = b \times 2 + (a - 2b) = \dots$

```
[1]: def division_euclidienne(a,b):  
    '''Renvoie le couple (q,r)'''  
    # BEGIN SOLUTION  
    q = 0  
    r = a  
    while r >= b:  
        q = q + 1  
        r = r - b  
    return q, r  
    # END SOLUTION
```

```
[7]: from civ import *  
init()  
test(division_euclidienne, 10, 3, (3, 1))  
test(division_euclidienne, 3, 5, (0, 3))  
test(division_euclidienne, 153, 7, (21, 6))  
test(division_euclidienne, 10, 1, (10, 0))  
valide()
```

```
OK!: division_euclidienne(10, 3) vaut (3, 1)  
OK!: division_euclidienne(3, 5) vaut (0, 3)  
OK!: division_euclidienne(153, 7) vaut (21, 6)  
OK!: division_euclidienne(10, 1) vaut (10, 0)  
Tous les tests sont valides
```

2 PGCD et algorithme d'Euclide

On rappelle que si $(a, b) \in \mathbb{N}^2 \setminus \{(0, 0)\}$, alors le **PGCD** de a et de b , noté $a \wedge b$ est le plus grand diviseur commun à a et à b .

On va donner ici deux versions de l'algorithme de calcul du PGCD dû à Euclide.

La première, appelée méthode des soustractions successives, revient à remarquer :

- que $a \wedge 0 = 0 \wedge a = a$
- que si $a \geq b$ alors $a \wedge b = (a - b) \wedge b$

On en déduit alors une manière récursive de calculer un PGCD :

$$24 \wedge 40 = 24 \wedge 16 = 8 \wedge 16 = 8 \wedge 8 = 0 \wedge 8 = 8$$

Pour simplifier, on pourra remarquer que $a \wedge b = b \wedge a$ et donc qu'on peut remettre les nombres dans l'ordre qu'on souhaite.

```
[37]: def pgcd_soustraction(a, b):  
    '''Implémenter ce calcul du pgcd'''  
    # BEGIN SOLUTION  
    if b == 0:  
        return a  
    if a < b:  
        return pgcd_soustraction(b, a)  
    return pgcd_soustraction(a-b, b)  
    # END SOLUTION
```

```
[38]: from civ import *  
init()  
test(pgcd_soustraction, 24, 40, 8)  
test(pgcd_soustraction, 24, 0, 24)  
test(pgcd_soustraction, 40, 24, 8)  
valide()
```

OK!: pgcd_soustraction(24, 40) vaut 8

OK!: pgcd_soustraction(24, 0) vaut 24

OK!: pgcd_soustraction(40, 24) vaut 8

Tous les tests sont valides

Cette méthode peut être raffinée à l'aide de la division euclidienne : si $a = bq + r$ alors $a \wedge b = b \wedge r$. En déduire l'algorithme usuel d'Euclide utilisant la division euclidienne directement plutôt que des soustractions (*note: la division utilise les soustractions, elles n'ont pas disparues.*).

```
[39]: def pgcd(a,b):  
    '''Algorithme d'Euclide du PGCD'''  
    # BEGIN SOLUTION  
    if b == 0:  
        return a  
    q, r = division_euclidienne(a, b)  
    return pgcd(b, r)  
    # END SOLUTION
```

```
[40]: from civ import *  
init()
```

```
test(pgcd, 24, 40, 8)
test(pgcd, 24, 0, 24)
test(pgcd, 40, 24, 8)
valide()
```

OK!: pgcd(24, 40) vaut 8
OK!: pgcd(24, 0) vaut 24
OK!: pgcd(40, 24) vaut 8
Tous les tests sont valides

3 Test de primalité naïf

Écrire une fonction `est_premier` qui vérifie si un nombre n est premier testant sa divisibilité par des entiers $1 < d < n$.

```
[25]: def est_premier(n):
      # BEGIN SOLUTION
      for d in range(2,n):
          if n % d == 0:
              return False
      return n != 1
      # END SOLUTION
```

```
[26]: from civ import *
      init()
      for i in [1,4,6,10]:
          test(est_premier, i, False)
      for i in [2,3,5,11]:
          test(est_premier, i, True)
      valide()
```

OK!: est_premier(1) vaut False
OK!: est_premier(4) vaut False
OK!: est_premier(6) vaut False
OK!: est_premier(10) vaut False
OK!: est_premier(2) vaut True
OK!: est_premier(3) vaut True
OK!: est_premier(5) vaut True
OK!: est_premier(11) vaut True
Tous les tests sont valides

4 Crible d’Eratosthène

On cherche ici à déterminer tous les nombres premiers $\leq N$ où N est une limite qu’on se fixe à l’avance.

L’idée du crible d’Eratosthène consiste à aligner tous les nombres de 2 à N , puis d’effectuer le processus suivant :

. Déterminer le plus petit entier n non barré et non encore étudié. . Barrer tous les multiples de n qui sont $> n$. . S'il reste des nombres non barrés $> n$ retourner à l'étape 1. . Les nombres non barrés sont premiers.

Par exemple pour $N = 9$:

- initialisation : 2 3 4 5 6 7 8 9
- étape 1 – $n = 2$ on barre donc 4, 6, 8 : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9
- étape 2 – $n = 3$ on barre donc 6, 9 (mais 6 est déjà barré) : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
- étape 3 – $n = 5$ mais il n'y a pas de multiples : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
- étape 4 – $n = 7$ mais il n'y a pas de multiples : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
- étape 5 – il n'y a plus de nombres non barrés, on a donc trouvé les premiers ≤ 9 : 2 3 5 7

Pour réaliser cet algorithme on va utiliser un tableau `premier` de booléens tel que `premier[i]` soit à `True` ssi i est premier. Pour cela, il va falloir créer un tableau de $N + 1$ valeurs contenant initialement `True` pour tous les nombres sauf 0 et 1 qui sont les deux seuls qu'on peut barrer *a priori*.

On rappelle que pour créer un tableau contenant 10 fois la valeur `True`, il suffit d'écrire `[ True ] * 10`.

```
[8]: def initialise_crible(N):
    '''Renvoie un tableau de N+1 correspondant à l'état initial du crible.'''
    # BEGIN SOLUTION
    t = [ True ] * (N+1)
    t[0] = t[1] = False
    return t
    # END SOLUTION
```

```
[9]: from civ import *
init()
test(initialise_crible, 3, [False, False, True, True])
test(initialise_crible, 5, [False, False, True, True, True, True])
valide()
```

OK!: `initialise_crible(3)` vaut `[False, False, True, True]`

OK!: `initialise_crible(5)` vaut `[False, False, True, True, True, True]`

Tous les tests sont valides

Écrire une fonction `barre_multiple` qui prend en entrée un tableau `t` de booléens donné par la fonction précédente et un entier `n` et passe à `False` tous les indices multiples stricts (i.e. $\neq n$) de n . On évitera de faire des multiplications à répétitions car on peut se contenter d'additions. On rappelle que $N + 1 = \text{len}(t)$ ici.

```
[10]: def barre_multiple(t,n):
    # BEGIN SOLUTION
    m = 2*n
    while m < len(t):
        t[m] = False
        m = m + n
```

```
# END SOLUTION
```

```
[11]: from civ import *
init()
t = [False, False, True, True, True, True, True, True, True]
def test(t, n):
    return t == [False, False, True, True, False, True, False, True, False]
test_condition(barre_multiple, test, t, 2)
t = [False, False, True, True, True, True, True, True, True]
def test(t, n):
    return t == [False, False, True, True, True, True, False, True, True]
test_condition(barre_multiple, test, t, 3)
valide()
```

OK!: barre_multiple([False, False, True, True, True, True, True, True, True], 2)

a fait ce qu'on attendait

OK!: barre_multiple([False, False, True, True, True, True, True, True, True], 3)

a fait ce qu'on attendait

Tous les tests sont valides

Écrire une fonction `crible` qui prend en entrée N et qui renvoie le tableau de booléens correspondant à l'application de l'algorithme.

```
[14]: def crible(N):
      # BEGIN SOLUTION
      v = initialise_crible(N)
      for n in range(N):
          if v[n]:
              barre_multiple(v, n)
      return v
      # END SOLUTION
```

```
[17]: from civ import *
init()
test(crible, 10, [False, False, True, True, False, True, False, True, False,
↪False, False])
test(crible, 15, [False, False, True, True, False, True, False, True, False,
↪False, False, True, False, True, False, False])
valide()
```

OK!: crible(10) vaut [False, False, True, True, False, True, False, True, False, False, False]

OK!: crible(15) vaut [False, False, True, True, False, True, False, True, False, False, False, True, False, True, False]

Tous les tests sont valides

Écrire une fonction `premiers` qui utilise `crible` pour déterminer la liste des nombres premiers $\leq N$. On pourra utiliser `l.append(x)` pour rajouter l'élément x à `l`.

```
[18]: def premiers(N):
      # BEGIN SOLUTION
      p = []
      l = crible(N)
      for i in range(N+1):
          if l[i]:
              p.append(i)
      return p
      # END SOLUTION
```

```
[21]: from civ import *
      init()
      test(premiers, 5, [2,3,5])
      test(premiers, 100, [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
      ↪ 53, 59, 61, 67, 71, 73, 79, 83, 89, 97])
      valide()
```

OK!: premiers(5) vaut [2, 3, 5]

OK!: premiers(100) vaut [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Tous les tests sont valides

5 Conjecture de Goldbach

En 1742, Christian Goldbach énonce cette conjecture célèbre :

“Tout nombre entier pair ≥ 3 peut s’écrire comme la somme de deux nombres premiers.”

Écrire une fonction `decompose` qui prend en entrée un entier n pair et ≥ 3 et renvoie un couple de nombre premiers (p_1, p_2) tel que $n = p_1 + p_2$ et $p_1 \leq p_2$.

```
[35]: def decompose(n):
      # BEGIN SOLUTION
      for i in range(2, n//2+1):
          if est_premier(i) and est_premier(n-i):
              return (i, n-i)
      # END SOLUTION
```

```
[33]: from civ import *
      init()
      test(decompose, 6, (3, 3))
      test(decompose, 10000, (59, 9941))
      valide()
```

OK!: decompose(6) vaut (3, 3)

OK!: decompose(10000) vaut (59, 9941)

Tous les tests sont valides

6 Il vous reste du temps ?

C'est le bon moment d'utiliser ce que vous avez appris sur le site <https://projecteuler.net/> !