Arithmétique dans $\mathbb{Z}$: algorithmes

1 Algorithme d'Euclide

Implémentations en Python et Caml :

```python
def pgcd_rec(a, b):
    """renvoie le pgcd de deux
entiers a et b"""
    if b == 0:
        return a
    else:
        return pgcd_rec(b, a % b)
```

```
let rec pgcd a b =
  match b with
  | 0 -> a
  | _ -> pgcd b (a mod b)
;;
pgcd : int -> int -> int = <fun>
```

Algorithme récursif efficace car pas de retour nécessaire sur la pile d'évaluation (récursivité terminale) !

```python
def pgcd(a, b):
    """renvoie le pgcd de deux
entiers a et b"""
    r, r1 = a, b
    while r1 != 0:
        r, r1 = r1, r % r1
    return r
```

À la fin de la k^{e} étape, `r` contient r_{k-1} et `r1` contient r_k (avec $r_{-1} = a$ et $r_0 = b$).

```
let pgcd a b =
  let r = ref a in
  let r1 = ref b in
  while !r1 <> 0 do
    let temp = }r in
    r := !r1;
    r1 := temp mod }r1
  done;
  !r
;;
pgcd : int -> int -> int = <fun>
```

ce qui s'écrit encore plus simplement si on se permet de renommer r_1, r en a, b :

```python
def pgcd(a, b):
    "renvoie le pgcd de deux entiers a et b"
    while b != 0:
        a, b = b, a % b
    return a
```

Le théorème de Lamé affirme que si $a \geqslant b$, le nombre de divisions dans l'algorithme d'Euclide est de l'ordre de $\left\lfloor \dfrac{\ln b}{\ln \varphi} \right\rfloor$ où $\varphi = \dfrac{1+\sqrt{5}}{2}$ est le nombre d'or. Ainsi, sa complexité est

$T(b) = O(\ln b)$.

2 Algorithme d'Euclide étendu

On peut tirer des cœfficients de Bezout [1] de l'algorithme d'Euclide en remontant les étapes pour obtenir à chaque fois $a \wedge b$ comme combinaison linéaire de r_k et r_{k-1}.

Si on a déjà $a \wedge b = r_k \cdot U + r_{k+1} \cdot V$, comme $r_{k-1} = r_k \cdot q + r_{k+1}$, on a alors

$$a \wedge b = r_{k-1} \cdot V + r_k \cdot (U - qV).$$

Algorithme d'Euclide étendu :
C'est facile en récursif :

```python
def euclide_rec(a, b):
    """renvoie (d, u, v) où
    d=pgcd(a, b) et au + bv = d"""
    if b == 0:
        return a, 1, 0
    else:
        q, r = divmod(a, b)
        d, u, v = euclide_rec(b, r)
        return d, v, u - q * v
```

```
let rec euclide a b =
  match b with
  | 0 -> a, 1, 0
  | _ ->
    let q = a / b in
    let r = a mod b in
    let d, u, v = euclide b r in
    d, v, u - q * v
;;
euclide:
  int -> int -> int * int * int = <fun>
```

Pour une version itérative, c'est plus compliqué.

- On peut comme on le fait à la main, garder en mémoire (pile) les quotients successifs puis les parcourir de nouveau pour les calculs successifs de u et v (cela nécessite donc un double parcours) :

```python
def euclide(a, b):
    """renvoie (d, u, v) où d = pgcd(a, b) et au + bv = d"""
    r, r1, quotients = a, b, []
    while r1 != 0: # Algorithme d'Euclide avec mém. des quot.
        quotients.append(r // r1)
        r, r1 = r1, r % r1
        # k_e tour : r = r[k-1], r1 = r[k], quotients = [q[0],...,q[k]]
    u, v = 1, 0
    while quotients != []: # Parcours de la liste des quotients
        q = quotients.pop()
        u, v = v, u - q * v
    return r, u, v
```

1. 

Étienne Bezout (Nemours 1730 - Avon 1783) Éminent mathématicien, adjoint mécanicien à l'Académie des sciences en mars 1758, il fut nommé en 1763 professeur et examinateur des gardes-marine et composa pour eux un Cours de mathématiques en 4 volumes. Membre de l'Académie de marine, il est l'auteur de nombreux ouvrages dont un Traité de navigation (1769) et une Théorie générale des équations algébriques (1779). Bézout contribua beaucoup à orienter dans un sens mathématique la formation des jeunes officiers pour les rendre aptes aux calculs astronomiques les plus savants. Un tel système, où la théorie l'emportait trop souvent sur la pratique, contrairement à ce qui se faisait en Angleterre, provoqua d'assez vives polémiques.

- On peut chercher à chaque étape u_k, v_k tels que $au_k + bv_k = r_k$:
  - ⋆ $r_{-1} = a$ donc $(u_{-1}, v_{-1}) = (1, 0)$,
  - ⋆ $r_0 = b$ donc $(u_0, v_0) = (0, 1)$,
  - ⋆ puis comme $r_{k-1} = r_k \cdot q_{k+1} + r_{k+1}$,

$$\begin{aligned} r_{k+1} &= r_{k-1} - q_{k+1} \cdot r_k \\ &= a \cdot (u_{k-1} - q_{k+1} \cdot u_k) + b \cdot (v_{k-1} - q_{k+1} \cdot v_k), \end{aligned}$$

donc

$$\begin{cases} u_{k+1} = u_{k-1} - q_{k+1} \cdot u_k \\ v_{k+1} = v_{k-1} - q_{k+1} \cdot v_k \end{cases}$$

```python
def euclide2(a, b):
    """renvoie (d, u, v) où d = pgcd(a, b) et au + bv = d"""
    r, r1 = a, b
    u, v = 1, 0
    u1, v1 = 0, 1
    while r1 != 0:
        q = r // r1
        r, r1 = r1, r % r1
        u, u1 = u1, u - q * u1
        v, v1 = v1, v - q * v1
    return r, u, v
```

À la fin de la k^e étape, `r` contient r_{k-1} et `r1` contient r_k, et `u, v, u1, v1` sont tels que $a \cdot u + b \cdot v = r$ et $a \cdot u1 + b \cdot v1 = r1$.

3 Tester la primalité

Pour qu'un nombre entier n soit premier, il faut et il suffit qu'il n'ait pas de diviseur entre 2 et $\sqrt{n}$. D'où le test basique de primalité :

```python
def est_premier(n):
    """teste si n est un nombre premier."""
    if n < 2:
        return False
    k = 2
    premier = True
    while k ** 2 <= n and premier:
        premier = (n % k != 0) # passe à False si k | n
        k += 1
    return premier
```

4 Crible d'Eratosthène

Pour déterminer les nombres premiers $\leqslant n$, il suffit de dessiner dans un tableau contenant tous les entiers de 2 à n, puis barrer successivement les multiples (stricts) de 2, puis de 3, etc. jusqu'à $\sqrt{n}$.

		2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19
~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29
~~30~~	31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~
~~40~~	41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~
~~50~~	~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59
~~60~~	61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~
~~70~~	71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79
~~80~~	~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89
~~90~~	~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~

Implémentation en Python :

```
def crible(N):
    """Liste des nombres premiers <= N"""
    estPremier = [True for _ in range(N + 1)]
    estPremier[0:2] = [False, False]
    # A la fin de l'algo., on veut que
    # k soit premier ssi estPremier[k] contient True
    k = 2
    while k ** 2 <= N:
        if estPremier[k]:  # Si k est un nombre premier
            m = k * k # premier multiple potentiellement non encore rencontré
            while m <= N:    # Suppression des multiples de k
                estPremier[m] = False
                m += k
        k += 1
    return [k for k in range(N + 1) if estPremier[k]]
    # on pourrait aussi renvoyer estPremier pour tester en O(1) si un nombre
    # donné est premier.
```

Il n'y a pas de problème de terminaison et un invariant de sortie de la première boucle est : « Les multiples stricts de tous les nombres premiers au plus égaux à `k` sont à `False` dans `estPremier` », ce qui permet bien de prouver la correction de la fonction.

La complexité spatiale est évidemment en $O(N)$ et la complexité temporelle vérifie :

$$T(N) = \sum_{k \text{ premier} \leqslant \sqrt{N}} \sum_{i=k}^{\lfloor N/k \rfloor} O(1) = \sum_{k \text{ premier} \leqslant \sqrt{N}} O(\lfloor N/k \rfloor) = O\left(N \cdot \sum_{k \text{ premier} \leqslant \sqrt{N}} \frac{1}{k} \right)$$

soit $T(N) = O(N \cdot \ln(\ln N))$ car on peut montrer [1] que $\sum_{p \text{ premier} \leqslant n} \frac{1}{p} \sim \ln(\ln n)$.

1. Mais ce n'est pas facile !