Algorithmes sur les nombres

François Schwarzentruber

1 Suite de Fibonacci

Définition 1 *La suite de Fibonacci* $(\mathcal{F}_n)_{n\in\mathbb{N}}$ *est définie par :*

$$\mathcal{F}_0 = 0;\ \mathcal{F}_1 = 1\ ;\ \textit{pour tout}\ n \geq 2,\ \mathcal{F}_n := \mathcal{F}_{n-1} + \mathcal{F}_{n-2}$$

La suite de Fibonacci est intéressante car il y a plusieurs algorithmes pour en calculer le n-ème terme ; c'est une très jolie histoire.

1.1 Algorithme récursif

L'algorithme suivant calcule le n-ème terme de la suite de Fibonacci.

```
function fib(n)
  if n ≤ 1 then
    return n
  else
    return fib(n − 1) + fib(n − 2)
```

On note $\tau(n)$ le nombre d'étapes élémentaires de l'appel $fib(n)$ (on considère ici une addition comme une opération élémentaire, bien que cela ne soit pas strictement correct car le nombre de chiffres dans un nombre est arbitrairement grand, on reviendra sur ce problème plus tard). Nous avons

$$\tau(n) := \tau(n-1) + \tau(n-2) + O(1).$$

Autrement dit, la complexité de $fib(n)$ est plus grande que la suite de Fibonacci elle-même dont la croissante est exponentielle :

$$\tau(n) > F_n \sim \varphi^n$$

où $\varphi :== \frac{1+\sqrt{5}}{2}$. Autrement dit, l'algorithme est en temps exponentiel. Le soucis vient du fait que le résultat d'un appel est calculé plusieurs fois (trop de fois !).

1.2 Algorithme en temps linéaire (euh quadratique)

Pour éviter de calculer plusieurs fois un résultat, nous utilisons un tableau T pour stocker les valeurs de la suite.

```
function fib(n)
  | créer un tableau T[0, ..., n]
  | T[0] := 0
  | T[1] := 1
  | for i := 2 à n do
  |   | T[i] := T[i − 1] + T[i − 2]
  | endFor
  | return T[n]
```

L'appel $fib(n)$ réalise $O(n)$ additions. Mais F_n possède $O(n)$ chiffres (pourquoi ?). Et ça coûte $O(n)$ de réaliser une addition ! Donc :

Théorème 1 *L'algorithme $fib(n)$ est en temps quadratique $O(n^2)$.*

1.3 Algorithme avec représentation matricielle

Dans cette sous-section, nous allons améliorer la complexité. Au lieu d'une complexité quadratique $O(n^2)$, nous allons proposer une complexité en $O(n^k)$ où $O(n^k)$ est la complexité pour faire une multiplication de deux nombres à n chiffres, avec $k < 2$ (oui, oui, nous allons voir plus tard un meilleur algorithme que l'algorithme naïf).

L'algorithme repose sur l'équation matricielle pour calculer la suite de Fibonacci :

$$\begin{pmatrix} \mathcal{F}_n \\ \mathcal{F}_{n+1} \end{pmatrix} := \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \mathcal{F}_{n-1} \\ \mathcal{F}_n \end{pmatrix}$$

En itérant cela donne :

$$\begin{pmatrix} \mathcal{F}_n \\ \mathcal{F}_{n+1} \end{pmatrix} := \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} \mathcal{F}_0 \\ \mathcal{F}_1 \end{pmatrix}$$

Le calcul de $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$ se réalise à l'aide de l'exponentiation rapide de matrices. Au lieu de réaliser $A^9 = A \times A \times A \times A \times A \times A \times A \times A \times A$, il vaut mieux faire $A^8 = A \times ((A^2)^2)^2$, non ? Voici l'algorithme de l'exponentiation rapide, que l'on note *puiss*.

```
function puiss(A, n)
  | if n = 0
  |   | return ( 1 0 ; 0 1 )
  | else
  |   | z := puiss(A, ⌊n/2⌋)
  |   | if n est pair then
  |   |   | return A²
  |   | else
  |   |   | return A × A²
```

Du coup, nous obtenons un algorithme pour le calcul du n-ème terme de la suite de Fibonacci.

```
function fib(n)
  | return première coordonnée de puiss(( 0 1 ; 1 1 ), n) × ( F0 ; F1 )
```

Proposition 1 *Le nombre d'appels récursifs imbriqués de $puiss(A, n)$ est $O(\log_2 n)$.*

PROOF.
Nous notons $a(n)$ le nombre d'appels récursifs de $puiss(A, n)$. Nous avons

$$a(n) := 1 + a(\lfloor n/2 \rfloor)$$

Donc $a(n) = O(\log_2 n)$. ■

Le produit de deux matrices demande de faire 4 additions et 8 multiplications. L'addition de deux nombres de n bits est un nombre de $n+1$ bits (par exemple $11+11 = 101$). La multiplication de deux nombres de n bits est un nombre de $2n$ bits (par exemple $11 \times 11 = 1001$).

Proposition 2 *Les nombres dans A^n ont au plus $O(n)$ bits.*

PROOF.
Soit u_n une majoration du nombres de bits d'un nombre dans A_n que l'on définit par $u_0 := 1$ et $u_n := 2 + 2 \times u_{\lfloor n/2 \rfloor}$. Du coup, $u_n := O(n)$. ■

Théorème 2 *L'algorithme total $fib(n)$ est en $O(n^2 \times \log_2 n)$.*

PROOF.
A chaque appel récursif de $puiss$, nous avons un nombre constant de multiplications, qui sont en $O(n^2)$. Il y a $\log_2 n$ appels récursifs imbriquées. D'où le théorème. ■

On verra (bientôt, bientôt !) un algorithme pour faire la multiplication de deux nombres de n bits en $O(n^k)$ pour un k dans $]1, 2[$, i.e. un meilleur algorithme pour multiplier deux nombres.

Théorème 3 *Si on fait la multiplication de deux nombres de n bits en $O(n^k)$, l'algorithme total $fib(n)$ est en $O(n^k \times \log_2 n)$.*

En fait, nous avons été très grossier : nous avons fait comme si le nombre de chiffres des nombres de *toutes* les matrices étaient de n. On peut faire une analyse un peu plus subtile, et obtenir le théorème suivant.

Théorème 4 *L'algorithme $fib3$ en $O(n^k)$ si la multiplication en $O(n^k)$.*

PROOF.
Le nombre de chiffres des nombres de A^n est n. Mais le nombre de chiffres des nombres de $A^{n/2}$ est $n/2$... Prenons cette information en compte. Notons $T(n)$ la complexité d'un appel à $puiss(A, n)$. Le cas de base $T(0)$ coûte $O(1)$. Pour des valeurs de $n \geq 1$, on comptabilise l'appel $puiss(A, \lfloor n/2 \rfloor)$ puis un calcul qui coûte $O((n/2)^k) = O(n^k)$. Ainsi :

$$T(n) := T(\lfloor n/2 \rfloor) + O(n^k)$$

Autrement dit :

$$T(n) = n^k + (n/2)^k + (n/4)^k ... = n^k \times (1 + (1/2^k) + (1/2^k)^2 + \dots)$$

On reconnaît les termes d'une série qui converge. L'algorithme est bien en $O(n^k)$. ■

1.4 Algorithme direct avec formule de Binet

On peut implémenter directement un calcul avec la formule de Binet est :

$$\mathcal{F}_n = \frac{1}{\sqrt{5}}(\varphi^n - \varphi'^n), \qquad \text{avec} \qquad \varphi = \frac{1+\sqrt{5}}{2} \qquad \text{et} \qquad \varphi' = -\frac{1}{\varphi}$$

Les soucis sont alors les erreurs d'arrondis et les imprécis des flottants. Les flottants sont codés sur 64 bits, ou 128 bits... Bref, il n'y a pas suffisamment de bits pour stocker des nombres avec n bits où n est arbitrairement grand.

2 Opérations arithmétiques

2.1 Addition

L'algorithme d'addition naïf vu à l'école primaire est en $O(n)$.

	1	2	3	4
+		5	6	7
	1	8	0	1

Exercice 1 *Écrire l'algorithme d'addition naïf en Python, ainsi que, si vous le souhaitez un rendu graphique (comme si vous aviez posé l'addition sur une feuille).*

2.2 Multiplication

2.2.1 Algorithme naïf

L'algorithme de multiplication naïf vu à l'école primaire est en $O(n^2)$.

		1	2	3	4
	×		5	6	7
		8	6	3	8
	7	4	0	4	
6	1	7	0		
6	9	9	6	7	8

Exercice 2 *Écrire l'algorithme de multiplication naïve en Python, ainsi que, si vous le souhaitez un rendu graphique (comme si vous aviez posé la multiplication).*

On peut réécrire l'algorithme de multiplication naïf sous forme récursive.

```
pre : deux nombres x et y avec n chiffres
post : le produit xy
function multiplier(x, y)
    if y = 0
        return 0
    else
        z := multiplier(x, ⌊y/2⌋)
        if y est pair then
            return 2z
        else
            return x + 2z
```

Théorème 5 *L'algorithme $multiplier(x, y)$ est $O(n^2)$.*

Proof.

Le nombre d'appels récursifs est majoré par le nombre de bits dans y, et donc par n. On a potentiellement une addition à chaque appel, i.e. du $O(n)$ à chaque appel. Ca donne du $O(n^2)$ en tout. ■

2.2.2 Algorithme de Karatsuba

On peut construire un meilleur algorithme pour multiplier deux nombres, par exemple l'algorithme de Karatsuba qui est $O(n^{\log_2 3})$. Voici son pseudo-code.

```
- Entrée : deux entiers x et y de n bits
- Sortie : le produit xy
function karatsuba(x, y)
  if n = 1 return xy
  x_G, x_D := ⌈n/2⌉ bits les plus à gauche, ⌊n/2⌋ bits les plus à droite de x
  y_G, y_D := ⌈n/2⌉ bits les plus à gauche, ⌊n/2⌋ bits les plus à droite de x
  A := karatsuba(x_G, y_G)
  B := karatsuba(x_D, y_D)
  C := karatsuba(x_G + x_D, y_G + y_D)
  return A2^n + (C − A − B)2^⌊n/2⌋ + B
```

Théorème 6 *La complexité de Karatsuba est en $O(n^{\log_2 3})$.*

PROOF.

On note $\tau(n)$ la complexité de l'appel lorsque les deux nombres ont n chiffres. On a :

$$\tau(n) = 3 \times \tau(\lceil n/2 \rceil) + O(n).$$

Supposons que $n = 2^k$. On a alors :

$$\begin{aligned}\tau(n) &= n + n3/2 + n3^2/2^2 + ... + n3^k/2^k \text{ où } k = \text{hauteur de l'arbre} \\ &= n(1 + 3/2 + 3^2/2^2 + ...3^k/2^k) \\ &= n\frac{(3/2)^{k+1} - 1}{3/2 - 1}\end{aligned}$$

La hauteur de l'arbre est $k = \log_2 n$ donc :

$$\tau(n) = O(n(3/2)^{\log_2 n}) = O(n(2^{\log_2(3/2)})^{\log_2 n}) = O(n \times n^{\log_2(3/2)}) = O(n^{\log_2 3}).$$

Ces résultats asymptotiques sont vraies quand n est une puissance de 2. Pour montrer que $\tau(n) = O(n^{\log_2 3})$, il suffit de remarquer que τ est croissante. Pour montrer la croissance, on peut introduire une propriété (vous en voyez une ?) et la démontrer par récurrence.

■

Exercice 3 *Écrire l'algorithme de multiplication de Karatsuba en Python. Comparer son efficacité avec celle de l'algorithme naïf.*

2.3 Division euclidienne

La division euclidienne vu à l'école primaire est en $O(n^2)$.

1	5	2	3	1	1	
	4	2		1	3	8
		9	3			
			5			

Exercice 4 *Écrire l'algorithme de division naïf en Python.*

On peut écrire la division récursivement comme suit.

```
pre : deux nombres x et y avec n chiffres, avec y ≥ 1
post : le quotient et le reste de la division euclidienne de x par y
function diviser(x, y)
  if x = 0 then
    return (0, 0)
  else
    (q, r) := diviser(⌊x/2⌋, y)
    q := 2q
    r := 2r
    if x est impair then
      r := r + 1
    if r ≥ y then
      r := r − y
      q := q + 1
    return (q, r)
```

Théorème 7 *L'algorithme* $diviser(x, y)$ *où* x *et* y *sont des nombres avec* n *chiffres est en* $O(n)$*.*

3 Calcul du PGCD

Comme nous allons le voir, le calcul du PGCD de deux nombres avec n chiffres peut se réaliser avec l'algorithme d'Euclide qui est en $O(n^3)$ car n appels récursifs $\times$ complexité en $O(n^2)$ de la division euclidienne.

```
- Entrée : a et b de n bits
- Sortie : pgcd(a, b)
function euclide(a, b)
  if b = 0
    return a
  else
    return euclide(b, a mod b)
```

```
- Entrée : a et b de n bits
- Sortie : (x, y, d) avec xa + yb = d = pgcd(a, b)
function euclideEtendue(a, b)
  if b = 0
    return (1, 0, a)
  else
    (x', y', d) := euclideEtendue(b, a mod b)
    return (y', x' − ⌊a/b⌋y', d)
```

Théorème 8 *L'algorithme d'Euclide est correct.*

Proof.

Car $pgcd(a, b) = pgcd(b, a \mod b)$. Formellement, la correction se démontre par récurrence (vous voyez laquelle ?).

L'algorithme étendue lui est correct car :

$$\begin{array}{ll} \text{Si} & x'b + y'(a \mod b) = d \\ \text{alors} & x'b + y'(a - \lfloor a/b \rfloor b) = d \\ \text{i.e.} & y'a + (x' - \lfloor a/b \rfloor y')b = d. \end{array}$$

■

Exemple 1

$euclide(120, 23)$	$120 = 23 \times 5 + 5$	$1 = (-9) \times 120$	$+$	47×23
$euclide(23, 5)$	$23 = 5 \times 4 + 3$	$1 = 2 \times 23$	$+$	$(-9) \times 5$
$euclide(5, 3)$	$5 = 3 \times 1 + 2$	$1 = (-1) \times 5$	$+$	2×3
$euclide(3, 2)$	$3 = 2 \times 1 + 1$	$1 = 1 \times 3$	$+$	$(-1) \times 2$
$euclide(2, 1)$	$2 = 1 \times 2 + 0$	$1 = 0 \times 2$	$+$	1×1
$euclide(1, 0)$		$1 = 1 \times 1$	$+$	0×0

Théorème 9 *Si les nombres a et b ont n chiffres, alors le nombre d'appels est $O(n)$.*

Proof.
Si $a < b$, alors l'algorithme "retourne le sens" de a et b pour avoir $a > b$ une fois pour toujours. Puis après, on a toujours $b \leq a$, pour tous les appels suivants.

Fait 1 *Si $b \leq a$ alors $a \mod b < a/2$.*

Proof.
En effet :
— si $b \leq a/2$, alors $a \mod b < b \leq a/2$;
— sinon, si $b > a/2$, $a \mod b = a - b < a/2$.
■
Ainsi, on a moins de $2n + 1$ appels (vous voyez pourquoi ?).
■

Théorème 10 (théorème de Lamé) *Pour tout entier $k \geq 1$, si on effectue au moins k appels récursifs pour $euclide(a, b)$ avec $a > b$ alors $a \geq \mathcal{F}_{k+1}$ et $b \geq \mathcal{F}_k$.*

Proof.
On démontre par récurrence sur k. Posons la propriété :

$\mathcal{P}(k)$: 'si on effectue au moins k appels récursifs pour $euclide(a, b)$ avec $a > b$ alors $a \geq \mathcal{F}_{k+1}$ et $b \geq \mathcal{F}_k$.'

Cas de base Pour $k = 1$, si on effectue au moins un appel récursif, cela signifie que $a > b \geq 1$ et donc $a \geq \mathcal{F}_2$ et $b \geq \mathcal{F}_1$.

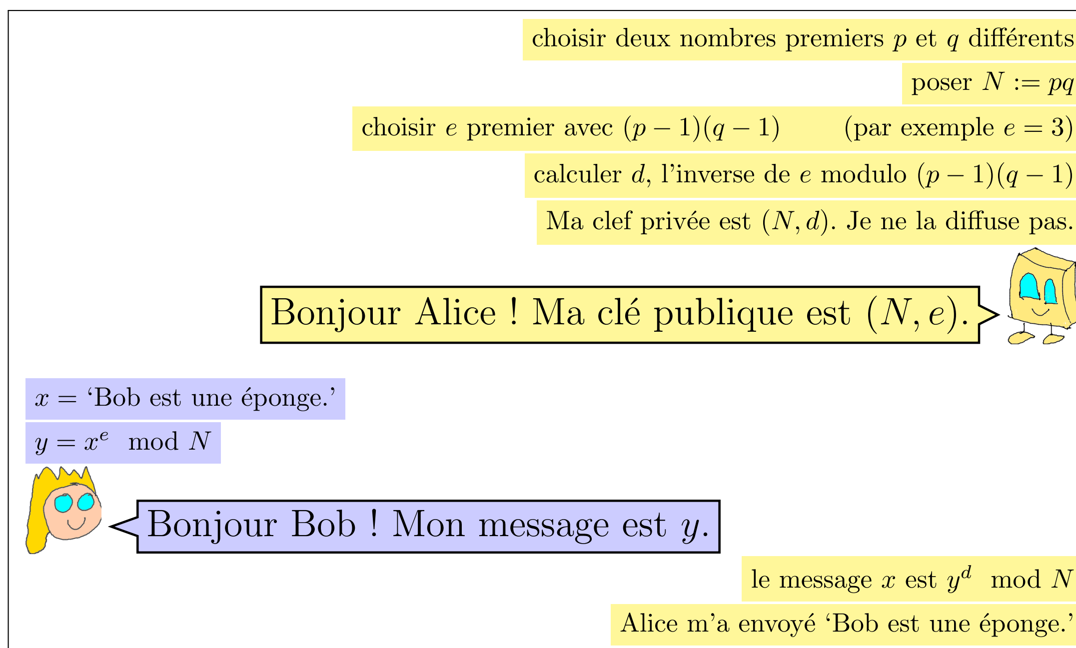
Cas inductif Supposons que la propriété $\mathcal{P}(k-1)$ et montrons que $\mathcal{P}(k)$. Considérons a et b pour lesquelles l'algorithme effectue plus de k appels. L'algorithme effectue $k - 1$ appels pour b et $a \mod b$. Par hypothèse de récurrence, cela signifie que $b \geq \mathcal{F}_k$ et que $a \mod b \geq \mathcal{F}_{k-1}$. Mais $a \geq b + (a \mod b) \geq \mathcal{F}_k + \mathcal{F}_{k-1} = \mathcal{F}_{k+1}$.
■

Théorème 11 *$euclide(\mathcal{F}_{k+1}, \mathcal{F}_k)$ réalise k appels récursifs.*

Proof.
$euclide(\mathcal{F}_{k+1}, \mathcal{F}_k) = euclide(\mathcal{F}_k, \mathcal{F}_{k-1}) = ...euclide(\mathcal{F}_1, \mathcal{F}_0)$. ■

4 RSA

Le protocole RSA permet de faire une communication cryptée. Bob souhaite initialiser la conversation. Il crée donc une clé privée (qu'il garde) et une clé publique qu'il envoie à Alice. Alice utilise la clé publique pour crypter son message que seul Bob, détenteur de la clé privé, pourra déchiffrer. Le schéma suivant montre comment s'utilise le protocole RSA.



Théorème 12 *Soit p et q deux nombres premiers différents et posons $N = pq$. Pour tout nombre e premier avec $(p-1)(q-1)$, l'application*

$$\begin{array}{rccl} enc: & \{0,\dots,N-1\} & \to & \{0,\dots,N-1\} \\ & x & \mapsto & x^e \end{array}$$

est une bijection. De plus, en posant d égal à l'inverse de e modulo $(p-1)(q-1)$, alors pour tout $x \in \{0,\dots,N-1\}$, on a :

$$(x^e)^d \equiv x \mod N.$$

PROOF.
Comme d est l'inverse de e modulo $(p-1)(q-1)$, nous avons $ed = 1 \mod (p-1)(q-1)$. En d'autres termes, il existe un entier k tel que $ed = 1 + k(p-1)(q-1)$. Ainsi, $x^{ed} = x(x^{(p-1)(q-1)})^k$.

D'après le petit théorème de Fermat (voir Théorème 13), $x^{p-1} = 1 \mod p$. Ainsi, $x^{ed} - x$ est divisible par p. D'après le petit théorème de Fermat, $x^{q-1} = 1 \mod q$. Ainsi, $x^{ed} - x$ est divisible par q. Le nombre $x^{ed} - x$ est divisible par p et par q, qui sont deux nombres premiers différents, donc $x^{ed} - x$ est divisible par N. C'est ce qu'il faut montrer. ■

5 Opérations modulo N

Dans cette section, on s'intéresse aux opérations arithmétiques modulo N. On note n le nombre de chiffres de N. On a $n = O(\log N)$. Ainsi, tous les nombres manipulés dans cette section ont n chiffres.

5.1 Addition modulo N

$O(n)$ aussi.

5.2 Multiplication modulo N

en $O(n^2)$: multiplication normale + division euclidienne par N.

5.3 Exponentiation modulo N

$O(n^3)$ car n d'appel récursif $\times$ multiplication

```
- Entrée : trois entiers x, y et N de n bits
- Sortie : x^y mod N
function expomod(x, y, N)
  | if y = 0
  |   | return 1
  | z := expomod(x, ⌊y/2⌋), N)
  | if y est pair then
  |   | return z^2 mod N
  | else
  |   | return x × z^2 mod N
```

5.4 Inverse modulo N

$O(n^3)$: calcul de l'inverse de a mod N

```
- Entrée : deux entiers a, N de n bits
- Sortie : a^-1 dans Z/NZ
function inv(a, N)
  | (x, y, d) := euclideEtendue(a, N)
  | if d = 1 then
  |   | return x
  | else
  |   | ERREUR 'a n'a pas d'inverse'
```

6 Test de primalité

6.1 Algorithme naïf

```
- Entrée : un entier N de n bits
- Sortie : oui si N est premier, non sinon
function estPremier(N)
    for i = 2 à N − 1
        if i divise N then
            return non
    return oui
```

L'algorithme est en $O(N)$. Mais *c'est la taille qui compte*, pas l'entier N. La complexité est bien mesurée en la taille de l'entrée, c'est-à-dire n le nombre de chiffres dans l'entier N, et pas en l'entier N lui-même. Ainsi, l'algorithme est en $O(2^n)$.

6.2 Test de primalité de Fermat

Le test de primalité de Fermat s'appuie sur le petit théorème de Fermat.

Théorème 13 (petit théorème de Fermat) *Si N est premier alors*

(*) pour tout entier $a \in \{2, \ldots, N-1\}$, $a^{N-1} \equiv 1 \mod N$.

Proof.

Si N est premier, $(\mathbb{Z}/N\mathbb{Z}^*, \times)$ est un groupe. D'après le théorème de Lagrange, l'ordre x de a divise $N-1$. Donc $a^{N-1} \equiv a^{x\frac{N-1}{x}} \equiv 1 \mod N$. ■

Le test de Fermat est *probabiliste* et ne donne pas toujours la bonne réponse. En tout cas, on peut lui faire confiance à 100% si la réponse est non : si la réponse est non alors N est composé.

```
- Entrée : un entier N de n bits
- Sortie : si la réponse est non, N est composé.
function testFermat(N)
    choisir a ∈ {2, ..., N − 1} au hasard
    if a^(N−1) ≡ 1 mod N then
        return oui
    else
        return non
```

Exemple 2 *Calcul de* $2^{30} \mod 31$.

Quand la réponse est oui, il y a trois cas :

1. le nombre est premier (même si on en est pas sûr)
2. le nombre est composé mais pourtant l'algorithme répond oui tout le temps : le nombre est un *nombre de Carmichael*.
3. le nombre est composé mais on peut relancer le test plusieurs fois et il répondra bien non pour bonne valeur de a (avec une grande probabilité)

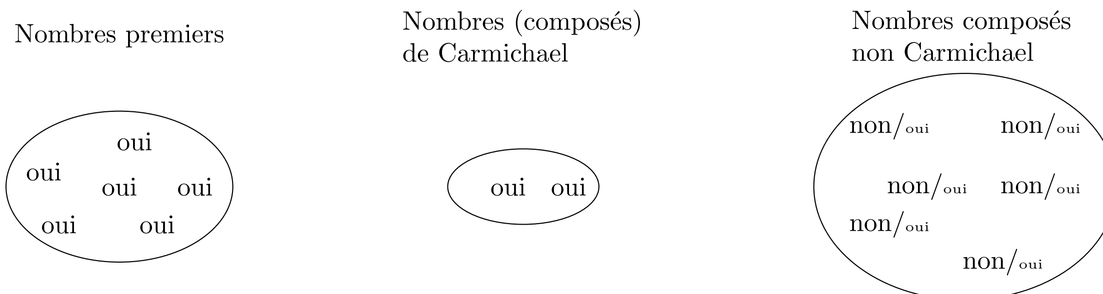


FIGURE 1 – Réponses du test de Fermat.

6.3 Premier souci : nombres de Carmichael

Définition 2 *Les entiers composés qui vérifient (*) s'appelle des nombres de Carmichael.*

Exemple 3 *Nombres de Carmichael : 561, 1105, 1729, 2465, 2821, 6601, 8911, etc.*

Théorème 14 (1994, Alford, Granville et Pomerance) *Il y a une infinité de nombres de Carmichael.*

Remarque 1 — *Il y a 105 212 nombres de Carmichael entre 1 et* 10^{15}.
- — *un nombre de 512 bits déclarés premier par l'algorithme avec* $a = 2$*, a 1 chance sur* 10^{20} *d'être composé (i.e. d'être pseudo-premier de base 2) ;*
- — *un nombre de 1024 bits déclarés premier par l'algorithme avec* $a = 2$ *a 1 chance sur* 10^{41} *d'être composé (i.e. d'être pseudo-premier de base 2).*

Pour aller plus loin : test de primalité de Miller-Rabin

6.4 Deuxième souci : on ne teste pas toutes les valeurs de a

Théorème 15 *Si N est composé et non Carmichael, alors la probabilité que l'algorithme réponde oui est plus petite que* $\leq \frac{1}{2}$.

La démonstration du théorème précédent s'appuie sur le lemme suivant.

Lemme 1 *Si* $a^{N-1} \not\equiv 1 \mod N$ *pour un certain a, alors*

$$a^{N-1} \not\equiv 1 \mod N \text{ pour au moitié des valeurs de } a \text{ dans } \{1, \dots, N-1\}.$$

PROOF.
$G := \{b \mid b^{N-1} \equiv 1 \mod N\}$. G est un sous-groupe strict de $(\mathbb{Z}/n\mathbb{Z}^*, \times)$. Donc par le théorème de Lagrange, $|G| \leq (N-1)/2$. ■

Du coup, en répétant le test, on augmente la probabilité d'avoir juste :

- Entrée : un entier N de n bits que l'on suppose non Carmichael
- Sortie :
  — si N est premier, il répond oui ;
  — si N est composé, il répond non avec une probabilité $> 1 - \frac{1}{2^k}$

```
function testFermat(N)
    choisir a_1, ..., a_k ∈ {2, ..., N − 1} au hasard de manière uniforme
    if a_i^{N−1} ≡ 1 mod N pour tout i = 1, ..., k then
        return oui (probablement premier ou nombre de Carmichael)
    else
        return non (nombre composé)
```

7 Génération de nombres premiers

Théorème 16 (de Lagrange des nombres premiers) *Soit $\pi(N)$ le nombre de nombres premiers $\leq N$.*

$$\pi(N) \sim_{N \to +\infty} \frac{N}{\ln N}.$$

Théorème 17 *Pour n assez grand, la probabilité qu'un nombre de n chiffres choisi uniformément soit premier est $> \frac{1}{n}$.*

Proof.
Il faut montrer que

$$\frac{\pi(2^n)}{2^n} > \frac{1}{n}.$$

Fait 2 *Pour N assez grand, $\pi(N) > \frac{N}{\log_2 N}$.*

On a $\ln N = \log_e N = \log_2 N \times \log_e 2$. Comme $\log_e 2 < 1$, on a $\ln N < \log_2 N$ et $\frac{N}{\ln N} > \frac{N}{\log_2 N}$. Le fait 2 découle du théorème de Lagrange des nombres premiers.

Conclusion : le nombre de nombres premiers à n chiffres est $\pi(2^n - 1) = \pi(2^n) > \frac{2^n}{n}$ pour n assez grand.

■

Comme il y a beaucoup de nombres premiers, on peut se contenter de générer au hasard un nombre, tester s'il est premier. Si c'est le cas, on a gagné. Sinon, on recommence.

- Entrée : entier n
- Sortie : un entier premier sur n bits

```
function genererNombrePremier(n)
    do
        N := tirer au hasard un nombre à n chiffres
    while le test de primalité dit que N est premier
    return N
```

8 Factorisation de nombres composés

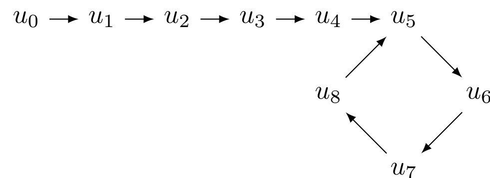
Dans cette section, on s'intéresse au problème suivant :
— entrée : un nombre composé N ;
— sortie : deux facteurs $a > 1$ et $b > 1$ tel que $N = ab$.

Ce problème est plus difficile que de tester si un nombre est premier ou composé. Le test de Fermat ne donne aucun facteur. L'algorithme naïf, oui, mais il est en temps exponentiel. Nous allons discuter l'algorithme rho de Pollard qui marche assez bien en pratique pour trouver deux facteurs. L'algorithme repose sur la détection de cycles dans une suite. Nous présentons d'une la détection de cycles, puis l'algorithme de Floyd qui utilise une mémoire constante. Puis nous terminons sur l'algorithme rho de Pollard.

8.1 Détection de cycles dans une suite $u_{n+1} = f(u_n)$

Définition 3 *On dit qu'une suite $(u_n)_{n\in\mathbb{N}}$ possède un cycle s'il existe μ, $\ell > 0$ tels que $u_\mu = u_{\mu+\ell}$.*

Exemple 4 *Voici une représentation graphique d'une suite avec $u_5 = u_9$.*



Voici un algorithme naïf pour détecter un cycle :

- Entrée : une fonction f, l'élément initial u_0
- Sortie : oui s'il y a un cycle, ne termine pas sinon

function *cycle?*(f, u_0)
  U := liste vide
  $y := f(u_0)$
  while y n'est pas dans la liste U **do**
    ajouter y à U
  return cycle sur y

Le soucis est que cet algorithme utilise une quantité de mémoire folle : on stocke toutes les valeurs précédentes.

8.2 Algorithme de Floyd

L'algorithme de Floyd repose sur cette proposition.

Proposition 3 *S'l existe μ, $\ell \geq 1$ tels que $u_\mu = u_{\mu+\ell}$, alors il existe $n \geq 1$ tel que $u_n = u_{2n}$.*

Proof.
On a alors $u_i = u_{i+\ell}$ pour tout $i \geq \mu$. Soit k tel que $k\ell \geq \mu$. On a :

$$u_{k\ell} = u_{k\ell+\ell} = u_{k\ell+2\ell} = \cdots = u_{2k\ell}.$$

■

Voici le pseudo-code de l'algorithme de Floyd qui utilise une mémoire constante.

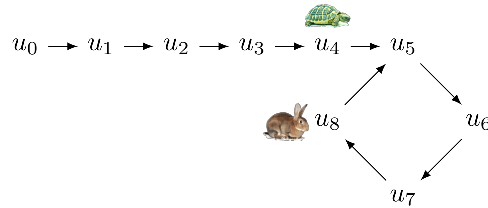


FIGURE 2 – La tortue et le lapin qui se baladent. La tortue est en u_4 et le lapin en u_8.

```
- Entrée : une fonction f, l'élément initial u0
- Sortie : oui s'il y a un cycle, ne termine pas sinon
function floyd(f, u0)
  (tortue, lapin) := (f(u0), f(f(u0)))
  while tortue ≠ lapin do
    (tortue, lapin) := (f(tortue), f(f(lapin)))
  return oui
```

L'algorithme fait marcher une tortue, qui parcourt la suite $u_0, u_1, u_2, \dots$, alors que le lapin parcourt la suite $u_0, u_2, u_4, \dots$ comme le montre la figure 2.

8.3 Algorithme rho de Pollard

L'algorithme rho de Pollard répond au problème de trouver un facteur d'un nombre N, i.e. un diviseur non trivial de N. On pose :

$$f: \begin{array}{rcl} \mathbb{Z}/N\mathbb{Z} & \rightarrow & \mathbb{Z}/N\mathbb{Z} \\ x & \mapsto & x^2 + 1[N]. \end{array}$$

Définition 4 *On pose* $(u_n)_{n\in\mathbb{N}}$ *définie par* $u_0 = 2$, $u_{n+1} = f(u_n)$.

Proposition 4 *La suite* $(u_n)_{n\in\mathbb{N}}$ *admet un cycle.*

PROOF.
Car $\mathbb{Z}/N\mathbb{Z}$ est fini. ■

L'algorithme de rho ne termine pas toujours. Il peut retourner un diviseur non trivial de N ou alors retourner N.

```
- Entrée : un entier N composé
- Sortie : un diviseur non trivial de N ou N, ou alors ne s'arrête pas
function rho(N)
  (tortue, lapin) := (f(2), f(f(2)))
  while pgcd(tortue − lapin, N) = 1 do
    (tortue, lapin) := (f(tortue), f(f(lapin)))
  return pgcd(tortue − lapin, N)
```

Théorème 18 *L'algorithme rho est correct.*

PROOF.

Soit N un nombre composé. Soit p un diviseur non trivial (que l'on connait pas) de N. Regardons la suite $(v_n)_{n\in\mathbb{N}}$ définie par $v_n = u_n[p]$, suite que l'on ne calcule pas explicitement. Cette suite admet aussi un cycle, i.e. il existe n tel que $u_n \equiv u_{2n}[p]$.

Plaçons nous dans un cas qui arrive souvent en pratique : on trouve $u_n \equiv u_{2n}[p]$ mais $u_n \not\equiv u_{2n}[N]$. Dans ce cas :

— $pgcd(u_n - u_{2n}, N) < N$ car $u_n \not\equiv u_{2n}[N]$;

— $pgcd(u_n - u_{2n}, N) > 1$ car $u_n \equiv u_{2n}[p]$

En d'autres termes, nous avons trouver un diviseur non trivial de N : $pgcd(u_n - u_{2n}, N)$.

Si on a juste $u_n \equiv u_{2n}[p]$ alors on sait juste que $pgcd(u_n - u_{2n}, N) > 1$. L'algorithme retourne un diviseur de N non trivial ou alors N lui-même. ■