Introduction à l'informatique
Algorithmes numériques

Benjamin Monmege

2019/2020

On a vu précédemment comment on pouvait stocker des ensembles ordonnés de données dans un tableau. C'est particulièrement pratique, par exemple pour stocker les informations des différents restaurants à proximité. Si l'on souhaite ensuite calculer la distance entre l'utilisateur et l'un de ces restaurants, il nous faut calculer avec des paires de réels (plus précisément de nombre flottants) représentant des coordonnées cartésiennes.

Cela nous amène à réfléchir à la façon dont une machine compte, c'est-à-dire calcule avec des nombres. On peut par exemple se demander ce que fait une machine lorsqu'on lui demande d'ajouter 1 à une variable n entière, comment une machine ajoute deux nombres entiers, comment elle teste que $n \equiv 4 \mod 27$ (c'est-à-dire que le reste dans la division euclidienne de n par 27 vaut 4), comment elle vérifie si deux entiers a et b sont premiers entre eux (c'est-à-dire n'ont que 1 comme diviseur commun positif), comment elle calcule l'exponentielle de 23, ou le logarithme de 14. Ce chapitre a pour objectif d'éclaircir ces différentes questions.

1 Addition d'entiers

Nous avons déjà vu un algorithme dans le chapitre d'introduction *incrémentant* une variable, c'est-à-dire ajoutant un à cette variable : `n := n+1`. On l'a vu aussi, les entiers sont représentés en binaire dans une machine tel que notre téléphone ou notre ordinateur. Dès lors, passer une variable de 13 à 14, signifie passer son codage en binaire de 1101 à 1110. Nous avions vu alors l'algorithme pour incrémenter la représentation binaire d'une variable :

(i) commencer par le bit de poids faible (celui qui est le plus à droite) ;

(ii) inverser le bit ;

(iii) tant que ce bit est à zéro, recommencer l'étape (ii) avec le bit situé à sa gauche ;

(iv) si on arrive au bout de la représentation binaire, ajouter un bit 1.

À l'époque, nous n'avions pas pu décrire plus précisément cet algorithme, faute de pseudo-code et de structure de données adaptée pour stocker un code binaire. Désormais, nous pouvons stocker le codage binaire de tout entier dans un tableau. Ainsi, on stocke l'entier $n = 207$ par le tableau

1	1	0	0	1	1	1	1

L'incrément peut alors s'écrire de la manière suivante en pseudo-code qui prend en entrée un tableau de bits et le modifie (la fonction ne renvoie donc rien, comme lors du tri par insertion dans le chapitre précédent) :

```
fonction incrémenter(n_en_binaire) :
    i := longueur(n_en_binaire)−1
    Tant que ((i> 0) et (n_en_binaire[i] = 1)) faire
        n_en_binaire[i] := 0
        i := i−1
    FinTantQue
    n_en_binaire[i] := 1
```

Par exemple, sur le tableau précédent, représentant 207, on initialise la variable `i` à 7 (la longueur du tableau valant 8, c'est-à-dire qu'on a codé l'entier sur un octet). On commence par s'apercevoir que la case d'indice 7 héberge un bit à 1, vérifie la condition de la boucle `Tant que` qui permet de passer ce bit à 0 et de passer la valeur de `i` à 6. Le tableau est donc devenu

1	1	0	0	1	1	1	0

On fait de même pour les trois bits à 1 suivants, arrivant à la situation où `i` vaut 3 et le tableau est

1	1	0	0	0	0	0	0

Désormais, la case d'indice `i`= 3 héberge un bit 0 et ne vérifie donc plus la condition de la boucle `Tant que`. On continue avec la suite du pseudo-code, après la boucle. On passe donc la valeur de cette case à 1 et on s'arrête. À la fin de l'algorithme, le tableau `n_en_binaire` est donc

1	1	0	1	0	0	0	0

qui est bien la représentation en binaire de 208.

Vous avez peut-être remarqué qu'on est en train d'utiliser un algorithme pour calculer $n + 1$ à partir de n, et que cet algorithme utilise l'affectation `i:=i`−1 : on pourrait donc naturellement se dire qu'on est en train de tricher. En fait, il n'en est rien car il faut simplement se rendre compte que ce sont des opérations bien différentes :

— l'incrémentation qu'on cherche à effectuer prend en entrée un entier qui pourrait être codé sur la totalité de la mémoire de notre ordinateur, et donc un nombre gigantesque et inconnu ;

— au contraire, l'affectation `i:=i`−1 est simplement un subterfuge pour dire à l'ordinateur de *se déplacer d'une case à gauche dans la mémoire* : c'est une opération de base que l'ordinateur est capable de faire sans rien calculer du tout...

Réitérons désormais l'expérience d'incrémentation avec le tableau suivant en entrée :

1	1	1	1	1	1	1	1

Cette fois-ci, le test `n_en_binaire[i]`= 1 n'échoue jamais, mais le test `i`> 0 finit par échouer, lorsque la variable `i` devient égale à 0. On sort alors du tableau (rempli de 0) et on passe la première case à 1 de sorte qu'on termine avec le tableau suivant :

1	0	0	0	0	0	0	0

Clairement, le résultat n'est pas correct puisque 11111111 est le codage binaire de l'entier $2^8 - 1 = 255$, alors que 10000000 est le codage binaire de l'entier $2^7 = 128$. La raison de l'échec est ce qu'on appelle un *dépassement de capacité* : 11111111 est le plus grand entier

naturel qu'on peut stocker sur 8 bits, de sorte qu'ajouter 1 est impossible sans dépasser les 8 bits autorisés.

L'addition de deux entiers codés en binaire s'effectue de manière similaire. On a vu en TD 1 comment effectuer de telles additions binaires à la main et on peut donc imaginer aisément un algorithme effectuant cela à notre place :

$$\begin{array}{cccccccccc} & 0 & 0 & {}^{1}0 & {}^{1}1 & {}^{1}0 & 1 & 0 & 0 \\ + & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

On suppose donc à partir de maintenant qu'on sait effectuer des additions et des soustractions d'entiers. On s'abstrait dans la suite du chapitre de la représentation en binaire des entiers.

2 Divisibilité

2.1 Division euclidienne

Une autre opération très courante qu'on exécute souvent consiste à calculer le quotient q et le reste r dans la division euclidienne d'un entier naturel n par un entier naturel m, c'est-à-dire l'unique paire $(q, r) \in \mathbf{N} \times \{0, 1, \ldots, m-1\}$ tel que

$$n = m \times q + r$$

Cette fois-ci, par souci de simplicité, on donne un algorithme qui ne suit pas la méthode *à la main* qu'on exécute habituellement. Par exemple, si on cherche à diviser 382 par 27, généralement, voilà comment on procède :

— d'abord on se demande combien de fois on peut mettre 27 dans 38 : c'est une fois donc on *pose* 1, et il reste 11 ;

— on fait descendre le chiffre 2 des unités dans 382 et on se demande combien de fois on peut mettre 27 dans 112 : c'est quatre fois donc on *pose* 4, et il reste 4.

On a donc le diagramme suivant :

$$\begin{array}{cccc|cc} & 3 & 8 & 2 & 2 & 7 \\ \cline{5-6} - & 2 & 7 & & 1 & 4 \\ \cline{1-3} & 1 & 1 & 2 & & \\ - & 1 & 0 & 8 & & \\ \cline{1-4} & & & 4 & & \end{array}$$

C'est une façon compacte et efficace de mener à bien une division euclidienne : elle est cependant plus délicate à écrire sous forme de pseudo-code. On utilise à la place une méthode plus élémentaire, mais moins efficace. Elle consiste à soustraire de 382 le nombre 27 autant de fois que nécessaire jusqu'à trouver un entier strictement inférieur à 27 : le nombre de fois qu'il

a fallu soustraire 27 est le quotient et l'entier restant finalement est le reste. Par exemple,

$$\begin{aligned} 382 - 27 &= 355 & 193 - 27 &= 166 \\ 355 - 27 &= 328 & 166 - 27 &= 139 \\ 328 - 27 &= 301 & 139 - 27 &= 112 \\ 301 - 27 &= 274 & 112 - 27 &= 85 \\ 274 - 27 &= 247 & 85 - 27 &= 58 \\ 247 - 27 &= 220 & 58 - 27 &= 31 \\ 220 - 27 &= 193 & 31 - 27 &= 4 \end{aligned}$$

On retrouve donc bien la division euclidienne $382 = 27 \times 14 + 4$.

Cette méthode par soustraction successive est aisée à écrire, dans une fonction qui prend en entrée deux entiers (le dividende n et le diviseur m), et qui renvoie une paire de deux entiers, le quotient et le reste dans la division euclidienne de n par m.

```
fonction division_euclidienne(dividende, diviseur) :
    quotient := 0
    Tant que (dividende ≥ diviseur) faire
        quotient := quotient + 1
        dividende := dividende − diviseur
    FinTantQue
    reste := dividende
    retourner(quotient, reste)
```

Dans la suite, on se permet d'utiliser

— le quotient dans la division euclidienne de n par m, qu'on notera $\lfloor n/m \rfloor$ (la notation $\lfloor x \rfloor$ étant la partie entière du réel x) et qui se note `n//m` en Python ;

— le reste dans la division euclidienne de n par m, qu'on notera $n \mod m$ et qui se note `n%m` en Python.

2.2 Calcul du plus grand diviseur commun : algorithme d'Euclide

La division euclidienne a de multiples applications : l'une d'entre elles est de tester si deux entiers positifs a et b sont premiers entre eux, c'est-à-dire si leur seul diviseur positif commun est 1. Par exemple, 14 et 15 sont premiers entre eux, mais 14 et 21 ne le sont pas puisqu'ils sont tous les deux divisibles par 7. Si on note $\text{pgcd}(a, b)$ le plus grand diviseur commun de a et b, on a

$$a \text{ et } b \text{ sont premiers entre eux } \iff \text{pgcd}(a, b) = 1$$

Pour tester si deux entiers sont premiers entre eux, il suffit donc de calculer leur pgcd. Pour cela, le fameux *algorithme d'Euclide* s'applique. Il consiste à faire des divisions euclidiennes successives en remplaçant le dividende par le diviseur et le diviseur par le reste, jusqu'à ce que ce ne soit plus possible car le reste devient nul. Par exemple, pour trouver le pgcd de 21 et 14, on effectue la division euclidienne de 21 et 14 :

$$21 = 14 \times 1 + 7$$

puis on remplace le dividende par 14 et le diviseur par 7 pour obtenir la deuxième division euclidienne :

$$14 = 7 \times 2 + 0$$

Le reste devient nul et on en déduit alors que le pgcd de 21 et 14 est le dernier reste non nul, à savoir 7 : $\text{pgcd}(21, 14) = 7$.

On peut représenter visuellement l'algorithme d'Euclide avec le diagramme en FIGURE 1.

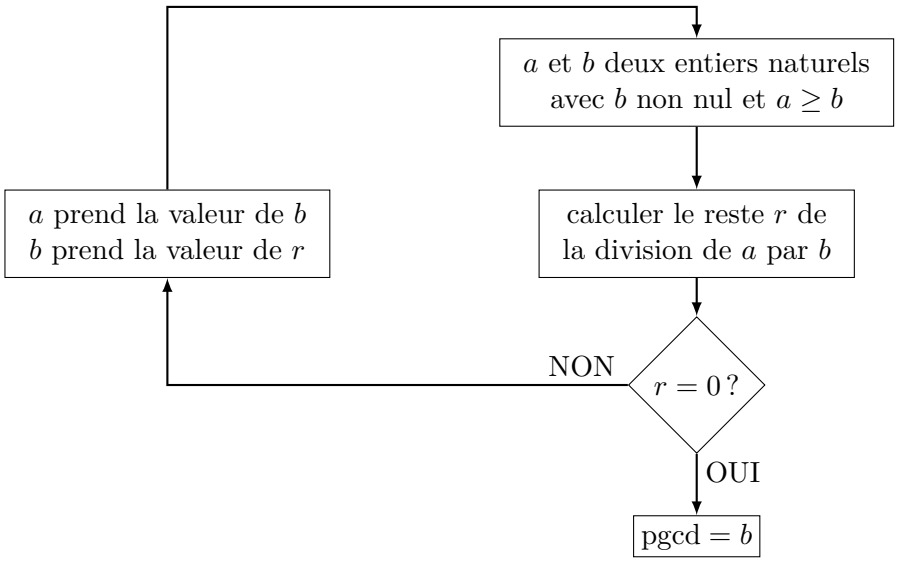

FIGURE 1 – Algorithme d'Euclide

Cette représentation graphique illustre qu'on doit faire une action (donner à a la valeur de b et à b la valeur de r, puis calculer le reste r de la division euclidienne de a par b) *tant que* r est non nul. On peut donc écrire tout naturellement le pseudo-code suivant :

```
fonction pgcd(a, b) :
    # a > b deux entiers non nuls
    r := a mod b
    Tant que (r > 0) faire
        a := b
        b := r
        r := a mod b
    FinTantQue
    retourner(b)
```

La correction de cet algorithme provient du fait qu'à tout moment dans l'algorithme le pgcd des variables a et b reste inchangé, ce qui tient du fait que si r est le reste dans la division euclidienne de a par b on a

$$\text{pgcd}(a, b) = \text{pgcd}(b, r)$$

La preuve de ce théorème est simple : il suffit de démontrer que les diviseurs communs de a et b sont les mêmes que ceux de b et r. Notons q le quotient dans la division euclidienne de a par b de sorte que $a = b \times q + r$.

— Soit d un diviseur commun de a et b. Alors d divise a et d divise b, donc d divise $a - b \times q = r$. Donc d est un diviseur commun de b et r.

— Soit d un diviseur commun de b et r. Alors d divise $b \times q + r = a$ donc d est un diviseur commun de a et b.

La terminaison de l'algorithme est également simple à démontrer puisque le long d'une itération de la boucle `Tant que` la valeur de la variable r passe à $b \mod r$ c'est-à-dire le reste

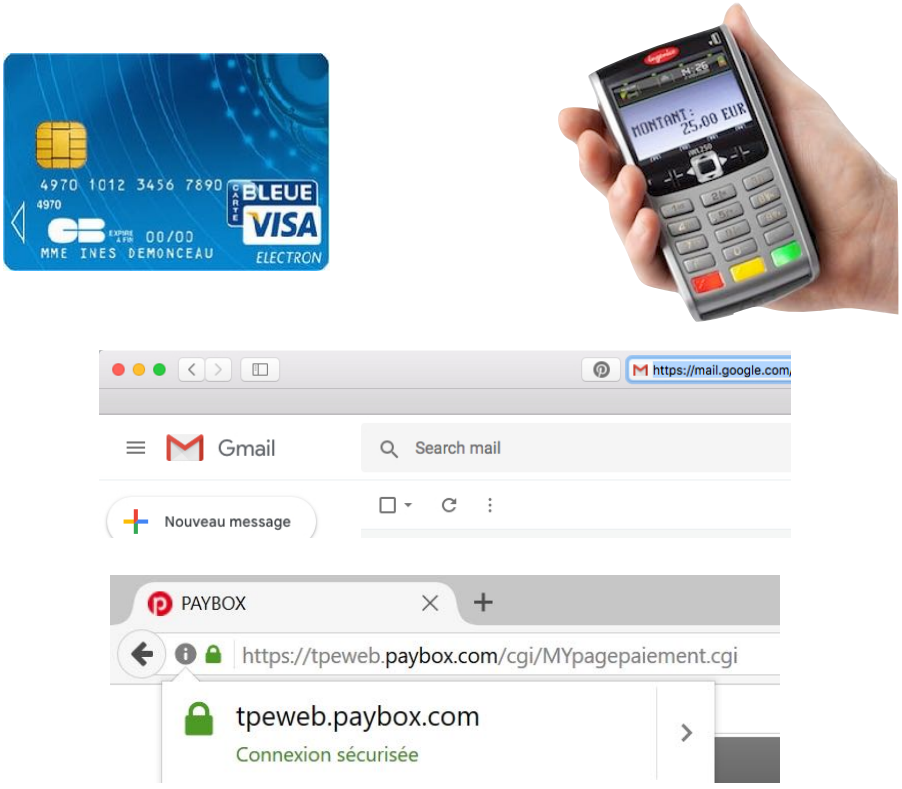

FIGURE 2 – Applications du test de primalité réciproque

dans la division euclidienne de b par r, qui par définition est strictement inférieur à r. Puisque la boucle `Tant que` s'arrête dès lors que r devient négatif ou nul, on est sûr qu'on ne peut boucler qu'un nombre fini de fois.

2.3 Applications de la divisibilité

Mais finalement, est-ce vraiment utile de savoir si deux entiers n et m sont premiers entre eux ? En fait, non seulement c'est utile mais c'est même indispensable dans notre vie de tous les jours. C'est une des opérations de base que l'on exécute très souvent comme illustré en FIGURE 2.

— Lorsqu'on paie avec une carte de crédit dans un commerce ou qu'on retire de l'argent à un distributeur, des informations personnelles critiques transitent sur Internet et la banque envoie l'autorisation de transaction : il faut donc que le terminal de paiement ou le distributeur de billets soit sûr qu'il est bien en train de communiquer avec la banque de manière sécurisée.

— Lorsqu'on se connecte à sa boîte mail, sur un réseau social ou qu'on paie une transaction en ligne, on envoie son mot de passe ou son numéro de carte bancaire sur Internet. On veut donc garantir que personne n'est capable de récupérer facilement ces informations privées. On utilise alors le protocole de communication *https*, la lettre *s* signifiant *sécurisé*.

Plus généralement, la tâche où s'applique le test de primalité réciproque concerne la cryp-

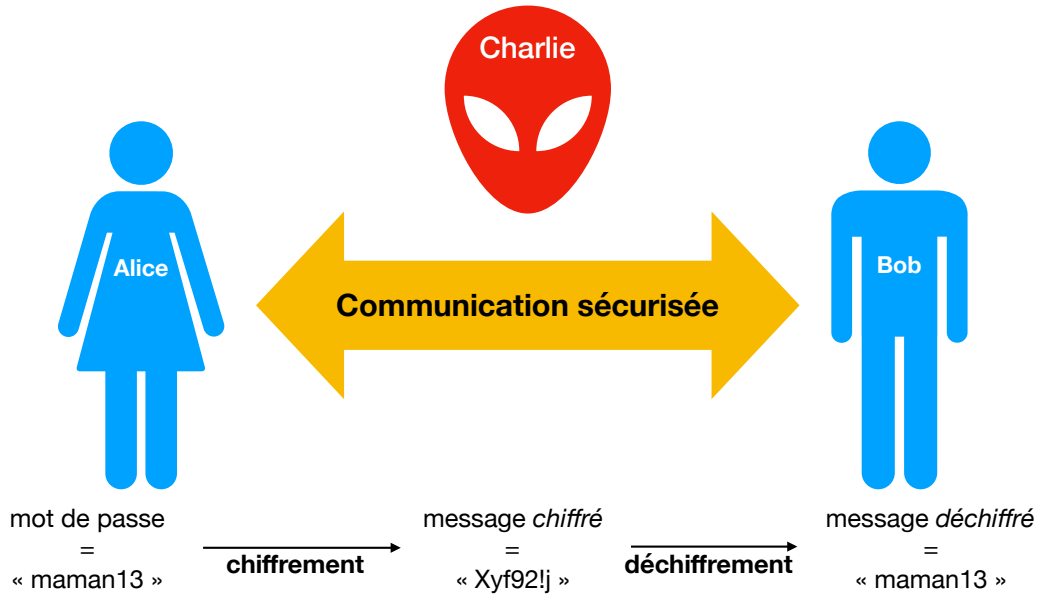

FIGURE 3 – Communication sécurisée entre deux entités

tographie. Alice et Bob veulent ainsi communiquer des informations de manière sécurisée, c'est-à-dire sans qu'un intrus, Charlie, puisse récupérer les informations. Par exemple, Alice peut vouloir se connecter à son compte Twitter en envoyant son mot de passe à Bob (qui est alors ici le serveur de Twitter...), comme en FIGURE 3.

Nous avons déjà vu dans les exercices des chapitres précédents des méthodes de chiffrement, que ce soit le chiffrement de César ou de Vigenère. À chaque fois, on s'est aperçu que ces techniques de chiffrement souffraient d'un lourd défaut : on peut rapidement casser le chiffrement en réalisant une cryptoanalyse (en essayant toutes les clés possibles ou bien en guidant sa recherche via l'utilisation de statistiques sur la langue du message à chiffrer). On n'utilise donc pas ces méthodes en réalité. À la place, on utilise des systèmes basés sur l'arithmétique, tels que le cryptosystème RSA (du nom de ses créateurs Ronald Rivest, Adi Shamir et Leonard Adleman). Il s'agit d'un système de chiffrement asymétrique, c'est-à-dire qu'Alice et Bob n'ont pas la même clé de chiffrement/déchiffrement (contrairement aux codes de César ou Vigenère où les différents participants partagent la même clé de décalage). C'est un système basé sur l'utilisation de grands entiers premiers qui implique que les messages eux-mêmes soient d'abord transformés en entiers : c'est facile en utilisant par exemple la table ASCII transformant chaque caractère en entier. La clé d'Alice est une paire d'entiers (e, n), celle de Bob une paire d'entiers (d, n). La procédure de chiffrement, illustrée en FIGURE 4, consiste, pour Alice, à partir d'un message M et de le chiffrer grâce à l'opération $M^e \mod n$. Symétriquement, la procédure de déchiffrement d'un chiffré C consiste, pour Bob, à calculer $C^d \mod n$.

Mais comment sont choisis les clés d'Alice et Bob pour rendre robuste et correct ce protocole de chiffrement ?

1. Tout d'abord, on choisit deux grands entiers premiers distincts p et q.
2. On calcule ensuite leur produit $n = p \times q$.
3. On calcule également $\varphi(n) = (p-1)(q-1)$.

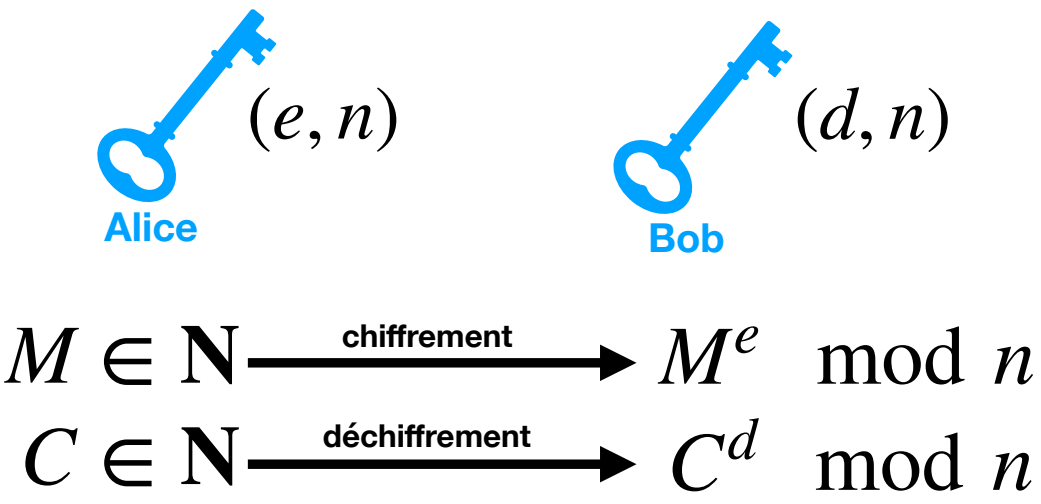

FIGURE 4 – Cryptosystème RSA

4. On choisit alors un entier e premier avec $\varphi(n)$.
5. On calcule l'entier d inverse de e modulo $\varphi(n)$, c'est-à-dire un entier d tel que $d \times e \equiv 1 \mod \varphi(n)$.

Il est très facile de calculer l'entier d à l'aide de l'algorithme d'Euclide (étendu). En effet, le fait que e et $\varphi(n)$ soient premiers entre eux (c'est-à-dire que leur pgcd soit égal à 1) garantit l'existence d'une paire de Bézout (d, k) tels que $de + k\varphi(n) = 1$. L'algorithme d'Euclide, légèrement enrichi, permet de calculer une telle paire (d, k).

Une fois crées les clés (e, n) et (d, n), on peut oublier p et q. La propriété clé qui garantit que le cryptosystème RSA est correct est le théorème suivant, qu'on admet dans ce cours (mais dont vous pourrez facilement trouver une preuve sur Internet si cela vous intéresse) :

Théorème 1. *Pour tout message $M \in \mathbf{N}$, $(M^e)^d \equiv M \mod n$, c'est-à-dire que si on déchiffre le message chiffré obtenu à partir du clair M, on retombe sur le message M initial.*

On le voit, l'arithmétique des entiers premiers (et premiers entre eux) est donc indispensable pour chiffrer efficacement et sûrement des messages qu'on s'échange à longueur de journée.

3 Exponentiation

Une question se pose cependant au vu de la procédure de chiffrement et de déchiffrement RSA qu'on vient de découvrir : comment la machine fait-elle pour calculer $M^e \mod n$? Plus généralement, comment la machine fait-elle pour calculer l'élévation à une puissance entière positive, par exemple pour calculer 21^{32} ou e^{23} ?

Étant donné un nombre x (cela peut-être un entier ou un réel, qu'on représente alors en machine à l'aide d'un flottant) et un entier naturel n, on souhaite donc calculer le nombre

$$x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$$

Si on suppose que la machine sait réaliser une multiplication (ce qui est du même genre que le calcul d'une addition qu'on a étudié plus tôt), on a donc un algorithme très simple qui consiste à multiplier n fois le nombre x avec lui-même. La fonction suivante réalise ces multiplications.

```
fonction puissance(x, n) :
    résultat := x
    Pour i de 2 à n faire
        résultat := résultat × x
    FinPour
    retourner(résultat)
```

L'opération coûteuse dans ce calcul est clairement la multiplication. Combien en effectue-t-on au cours de cet algorithme ? Autant qu'il y a de tours de la boucle `Pour`, c'est-à-dire $n-1$. Peut-on faire mieux ?

Par exemple, pour calculer x^8, l'algorithme précédent revient aux 7 multiplications

$$x^8 = x \times x \times x \times x \times x \times x \times x \times x$$

Mais on peut clairement faire mieux en remarquant que $x^8 = x^4 \times x^4$, puis que $x^4 = x^2 \times x^2$ puis que $x^2 = x \times x$. En seulement 3 multiplications (qu'on représente ci-dessous par des élévations au carré), on obtient donc

$$x^8 = ((x^2)^2)^2$$

Si on veut calculer x^{13}, on peut de même utiliser l'algorithme précédent réalisant 12 multiplications :

$$x^{13} = x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x$$

ou bien on peut remarquer que $x^{13} = x^6 \times x^6 \times x$. Si on sait calculer x^6, il faut alors ajouter deux multiplications supplémentaires pour obtenir x^{13}. De même, $x^6 = x^3 \times x^3$ et $x^3 = x^2 \times x$, avec $x^2 = x \times x$. On obtient donc x^{13} à l'aide de 5 multiplications seulement

$$x^{13} = ((x^2 \times x)^2)^2 \times x$$

On voit qu'on ne réalise donc pas le même découpage selon que la puissance n est paire ou impaire, ce qui nous amène à considérer l'algorithme suivant, qu'on appelle d'*exponentiation rapide*.

```
fonction puissance_rapide(x, n):
    a := 1
    b := x
    m := n
    Tant que (m > 0) faire
        Si (m ≡ 0 mod 2) alors
            m := m/2
        Sinon
            m := (m − 1)/2
            a := a × b
        FinSi
        b := b × b
    FinTantQue
    retourner(a)
```

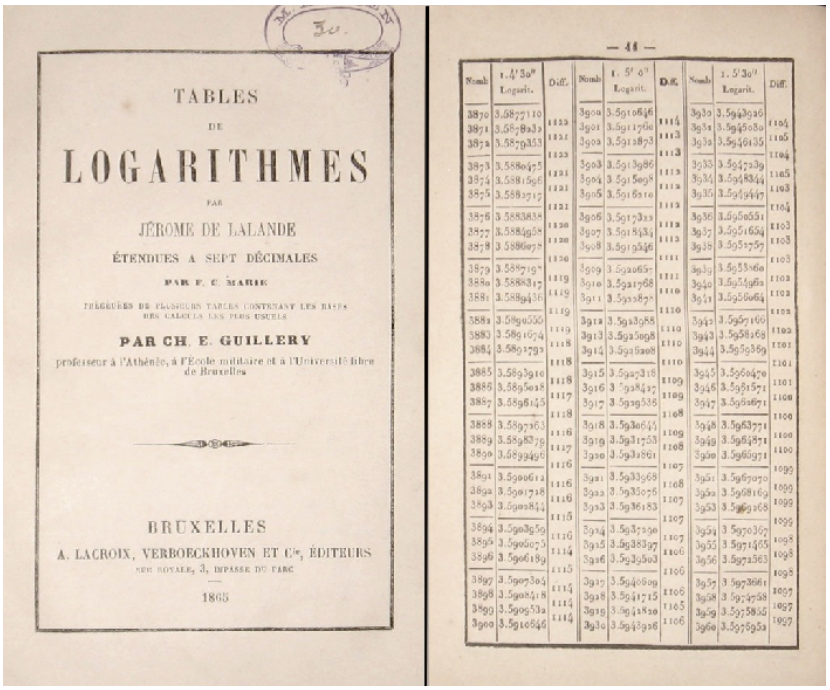
TABLES
DE
LOGARITHMES
PAR
JÉROME DE LALANDE
ÉTENDUES A SEPT DÉCIMALES
PAR F. C. MARIE
PRÉCÉDÉES DE PLUSIEURS TABLES CONTENANT LES BASES DES CALCULS LES PLUS USUELS
PAR CH. E. GUILLERY
professeur à l'Athénée, à l'École militaire et à l'Université libre de Bruxelles
BRUXELLES
A. LACROIX, VERBOECKHOVEN ET Cie, ÉDITEURS
RUE ROYALE, 3, IMPASSE DU PARC
1865
— 44 —
Nomb.	1.4'30" Logarit.	Diff.	Nomb.	1. 5' 0" Logarit.	Diff.	Nomb.	1. 5'30" Logarit.	Diff.
3870	3.5877110	1122	3900	3.5910846	1114	3930	3.5943936	1104
3871	3.5878232	1121	3901	3.5911960	1113	3931	3.5945040	1105
3872	3.5879353	1122	3902	3.5913073	1113	3932	3.5946145	1104
3873	3.5880475	1121	3903	3.5914186	1113	3933	3.5947249	1105
3874	3.5881596	1121	3904	3.5915299	1112	3934	3.5948354	1103
3875	3.5882717	1121	3905	3.5916411	1113	3935	3.5949457	1104
3876	3.5883838	1120	3906	3.5917524	1112	3936	3.5950561	1103
3877	3.5884958	1120	3907	3.5918634	1112	3937	3.5951664	1103
3878	3.5886078	1120	3908	3.5919746	1111	3938	3.5952767	1103
3879	3.5887197	1119	3909	3.5920857	1111	3939	3.5953869	1102
3880	3.5888317	1119	3910	3.5921968	1110	3940	3.5954971	1102
3881	3.5889436	1119	3911	3.5923078	1110	3941	3.5956074	1102
3882	3.5890555	1119	3912	3.5924188	1110	3942	3.5957176	1102
3883	3.5891674	1118	3913	3.5925298	1110	3943	3.5958278	1101
3884	3.5892792	1118	3914	3.5926408	1110	3944	3.5959379	1101
3885	3.5893910	1118	3915	3.5927518	1109	3945	3.5960479	1101
3886	3.5895028	1117	3916	3.5928627	1109	3946	3.5961581	1100
3887	3.5896145	1118	3917	3.5929736	1108	3947	3.5962681	1100
3888	3.5897263	1116	3918	3.5930844	1109	3948	3.5963781	1100
3889	3.5898379	1117	3919	3.5931953	1108	3949	3.5964881	1100
3890	3.5899496	1116	3920	3.5933061	1107	3950	3.5965981	1099
3891	3.5900612	1116	3921	3.5934168	1108	3951	3.5967079	1099
3892	3.5901728	1116	3922	3.5935276	1107	3952	3.5968179	1099
3893	3.5902844	1115	3923	3.5936383	1107	3953	3.5969278	1098
3894	3.5903959	1116	3924	3.5937490	1107	3954	3.5970376	1098
3895	3.5905075	1114	3925	3.5938597	1106	3955	3.5971475	1098
3896	3.5906189	1115	3926	3.5939703	1106	3956	3.5972573	1098
3897	3.5907304	1114	3927	3.5940809	1106	3957	3.5973671	1097
3898	3.5908418	1114	3928	3.5941915	1105	3958	3.5974768	1097
3899	3.5909532	1114	3929	3.5943020	1106	3959	3.5975865	1097
3900	3.5910646		3930	3.5944126		3960	3.5976962	

FIGURE 5 – Tables de logarithmes

4 Recherche d'un zéro d'une fonction

Terminons ce chapitre en nous intéressant au calcul de logarithmes, particulièrement pertinent par exemple pour calculer une valeur en décibel d'atténuation de signal (en acoustique par exemple). Développé par John Napier au début du XVIIème siècle, le logarithme *népérien* avait initialement pour objectif de simplifier des calculs, en particulier en remplaçant des multiplications par des sommes grâce à la propriété

$$\forall a \quad \forall b \qquad \ln(a \times b) = \ln(a) + \ln(b)$$

En l'absence de calculatrices, en effet, les sommes sont bien plus simples à réaliser à la main que des multiplications. En revanche, cela demandait à savoir passer aisément d'une valeur à son logarithme et vice versa. Pendant trois siècles, cela s'est fait à l'aide de tables de logarithmes qui étaient regroupées dans des petits livres, comme l'illustre la FIGURE 5.

Évidemment, il n'est plus question de telles tables de logarithmes aujourd'hui, les calculatrices et ordinateurs faisant le travail pour nous. Mais comment font-ils ? En fait, ils utilisent des techniques mathématiques un peu plus avancées que celles que vous connaissez sans doute à l'heure actuelle (développements limités, séries numériques, résolutions d'équations différentielles...), mais on peut déjà étudier une méthode simple pour faire de tels calculs, en supposant simplement qu'on sait calculer l'exponentielle de n'importe quel nombre flottant (et pas seulement d'un nombre entier positif comme on l'a fait dans la section précédente).

En effet, une fois qu'on sait calculer l'exponentielle, on peut remarquer que

$$x = \ln(14) \iff e^x = 14$$

Si on sait trouver un antécédent par la fonction exponentielle d'un nombre, 14 par exemple, alors on a trouvé son logarithme népérien $\ln(14)$. De manière similaire, pour le logarithme en

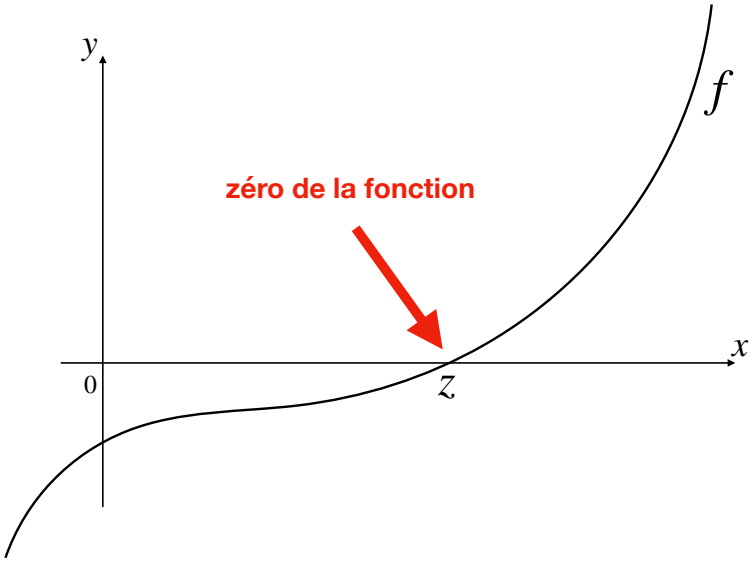

FIGURE 6 – Zéro d'une fonction f : abscisse z telle que $f(z) = 0$

base 2 :

$$x = \log_2(152) \iff 2^x = 152$$

Remarquons désormais que résoudre $e^x = 14$ revient à résoudre $e^x - 14 = 0$. On se ramène donc à rechercher un réel x tel que $e^x - 14 = 0$ ou $2^x - 152 = 0$, c'est-à-dire le zéro d'une fonction mathématique f, comme illustré en FIGURE 6.

Ce problème ressemble beaucoup au problème de la recherche d'un élément dans un tableau, qu'on a étudié dans le chapitre précédent. La différence est qu'on est face à un *tableau infini* ne permettant pas de réaliser la recherche séquentielle consistant à visiter tous les éléments du tableau de gauche à droite. En revanche, rien n'empêche d'utiliser une technique de recherche dichotomique (cf FIGURE 7). On commence par encadrer un des zéros de la fonction qu'on suppose continue : on suppose donc connu un intervalle $[a, b]$ tel que f est continue sur cet intervalle et change de signe, c'est-à-dire $f(a) \times f(b) < 0$. Par le théorème des valeurs intermédiaires, on est assuré de l'existence d'un zéro z dans l'intervalle $[a, b]$. Considérons le milieu de l'intervalle $\frac{a+b}{2}$. Deux cas se présentent :

— si $f((a + b)/2) < 0$ (comme dans la figure), alors on sait, toujours grâce au théorème des valeurs intermédiaires, que la fonction f possède un zéro sur l'intervalle $[\frac{a+b}{2}, b]$;
— si $f((a + b)/2) > 0$, alors on sait que la fonction f possède un zéro sur l'intervalle $[a, \frac{a+b}{2}]$.

On continue donc ainsi à resserrer l'intervalle encadrant un zéro de la fonction f : sur la figure, on considère ensuite m le milieu de l'intervalle $[\frac{a+b}{2}, b]$, puis m' le milieu de l'intervalle $[\frac{a+b}{2}, m]$. Quand s'arrête-t-on ? On s'arrête si on finit par tomber exactement sur un zéro, ou alors après avoir obtenu un intervalle $[x, x']$ suffisamment petit pour avoir obtenu une approximation suffisante du zéro recherché.

Outre cet algorithme de recherche dichotomique, d'autres solutions simples existent. L'une d'entre elles est l'*algorithme de Newton* qui consiste à « descendre le long de la courbe ». Cette fois-ci, on suppose de plus que la fonction f est dérivable, de sorte qu'on connait une équation

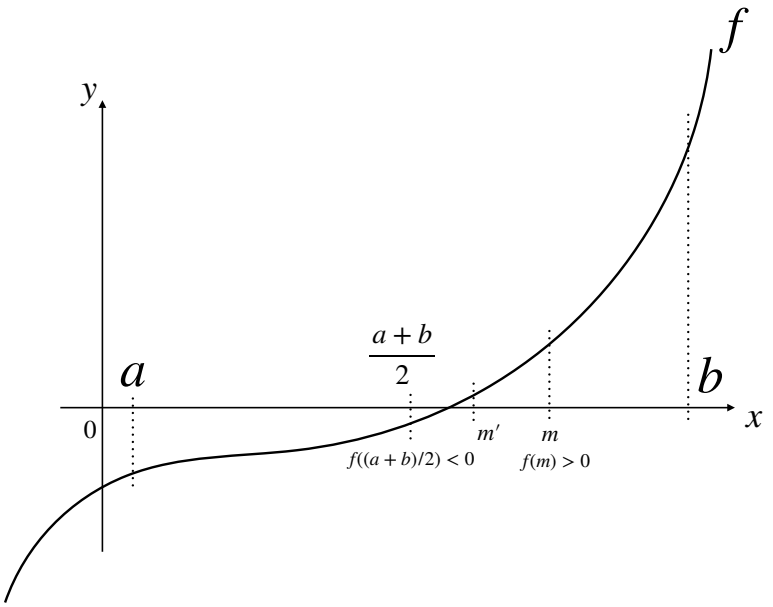

FIGURE 7 – Recherche dichotomique du zéro d'une fonction f

de la tangente à la courbe de f en le point d'abscisse x_0 : $y = f(x_0) + f'(x_0) \times (x - x_0)$. La méthode de Newton se base sur l'idée qu'on peut approcher la courbe d'une fonction par sa tangente : si x est suffisamment proche de x_0 alors $f(x)$ est proche de $f(x_0) + f'(x_0)(x - x_0)$.

En particulier, si on calcule l'abscisse du point d'intersection de la tangente à la courbe en x_0 avec l'axe des abscisses, on espère obtenir une nouvelle abscisse x_1 plus proche du zéro comme le montre la FIGURE 8.

On résout donc l'équation $0 = f(x_0) + f'(x_0)(x - x_0)$ afin de trouver (si la dérivée $f'(x_0)$ est non nulle) l'abscisse $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. Plus généralement, on construit donc la suite $(x_n)_{n\in\mathbb{N}}$ par récurrence, à partir du point x_0, en posant pour tout n

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

tant que la dérivée $f'(x_n)$ est non nulle (sinon la méthode échoue).

En partant de la fonction f associant à x la valeur $f(x) = e^x - 14$, on peut appliquer la méthode de Newton pour trouver une valeur approchée de $\ln(14)$. Il suffit de trouver la dérivée de f qui vaut $f'(x) = e^x$ et de partir de n'importe quelle abscisse, par exemple $x_0 = 0$. De même, si on choisit la fonction f associant à x la valeur $f(x) = 2^x - 152$, alors on a $f'(x) = \ln(2) \times 2^x$. Puisqu'on sait estimer précisément $\ln(2)$ (comme précédemment), on sait calculer la fonction dérivée. À nouveau en partant de $x_0 = 0$, cela permet d'estimer précisément la valeur de $\log_2(152)$.

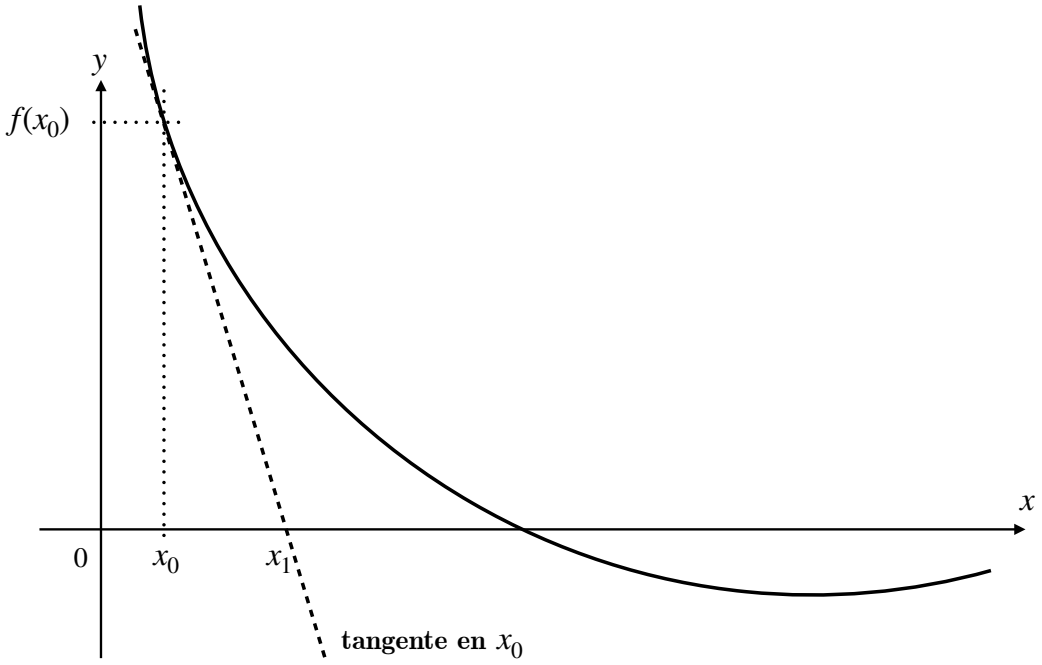

FIGURE 8 – Première étape de la méthode de Newton