Algèbre, arithmétique (Mat309)

Bernard.Parisse@univ-grenoble-alpes.fr

2019

Table des matières

Index

1 Motivations : cryptographie, codes correcteurs

La transmission d'informations par les réseaux sous forme numérique pose deux problèmatiques importantes

- la confidentialité ou l'authentification des données échangées, qu'il s'agisse de donner un numéro de carte bancaire ou de signer numériquement sa déclaration d'impots. C'est le domaine de la cryptographie.
- la détection et si possible la correction d'erreurs de transmission. La correction est souhaitable si on veut éviter de demander de renvoyer un message, en particulier s'il faut du temps pour transmettre un message (par exemple une donnée provenant d'une sonde martienne située en moyenne à 76 millions de km de la Terre met 4 minutes 20 secondes à nous parvenir, une réémission demande alors presque 10 minutes).

Ces deux problématiques ont des solutions à la frontière entre informatique et mathématiques. Dans ce cours intitulé Algèbre et arithmétique, on présente plusieurs structures algébriques (groupes, anneaux, corps, espaces vectoriels) ainsi que des algorithmes d'arithmétique dans ces structures, et on étudiera leur efficacité, qui assure la confidentialité des données transmises ou permet de détecter et corriger rapidement des erreurs de transmission (pas trop nombreuses).

1.1 Cryptographie

Historiquement, les militaires Romains pratiquaient déjà la cryptographie, avec le code dit de César qui consistait à décaler de plusieurs positions les lettres constituant un message dans l'alphabet, de manière cyclique. Par exemple si on décale de 4 positions, un A devient un E, un B devient un F, etc et un Z devient un D. Ce procédé est facile à implémenter sur une chaine de caractère numérisée (on additionne la clef de cryptage modulo 26) mais il est trop simple, car le nombre de possibilités de cryptage est trop limité (voir feuille d'exercices 1).
Plusieurs variations permettent de rendre ce système plus robuste. Par exemple on peut choisir une permutation quelconque des 26 lettres de l'alphabet. Cette méthode est toutefois sujette à une attaque statistique, toutes les lettres de l'alphabet n'ont pas la même fréquence dans un message (c'est pour cela qu'elles ne valent pas le même nombre de points au scrabble !).
Une autre méthode utilisée pendant la seconde guerre mondiale par les services de renseignements utilise une clef variable, on fait toujours une addition modulo 26, mais la clef change à chaque lettre, on utilise par exemple la position d'une série de lettre prise dans un livre, popularisé par *The Key to Rebecca* (*Le Code Rebecca*) de K. Follett basée sur l'histoire de l'espion nazi Johannes Eppler et l'opération Salaam. Ce type de code est incassable sans connaissance de la clef.
Une méthode plus élaborée était mise en oeuvre par la machine Enigma utilisée par les militaires nazis toujours pendant la seconde guerre mondiale, code qui a été cassé par Alan Turing et son équipe à Bletchley Park, popularisé par le film *The Imitation Game* et qui marque les débuts de l'informatique moderne (machine de Turing universelle).

On peut modéliser ces systèmes cryptographiques par la donnée d'une bijection E d'un ensemble d'entiers et de sa bijection réciproque D. Par exemple pour le cryptage de César, l'ensemble d'entiers est $[0, 25]$ (A=0, B=1, etc.), n est le décalage fixe et

$$E(x) = x + n \bmod 26, \quad D(x) = x - n \bmod 26$$

On peut imaginer des variations par exemple le codage dit linéaire

$$E(x) = ax + b \bmod 26$$

qui est bijectif si a est premier avec 26. Ces méthodes de cryptage sont dites à clef secrète (ou symétrique), car la connaissance des paramètres de E permet de déterminer facilement D. Il est nécessaire d'échanger la clef (par exemple par une rencontre physique) avant de pouvoir échanger des données.
L'essor des communications a nécessité l'invention de mécanismes permettant de sécuriser l'échange de données sans échanger au préalable des clefs. La méthode RSA (inventée en 1977 par Ronald Rivest, Adi Shamir et Leonard Adleman) très populaire aujourd'hui utilise le calcul de puissances d'entiers modulo

le produit de deux grands nombres premiers, nous allons maintenant la présenter rapidement et nous y reviendrons plus en détails plus loin lorsque nous aurons tous les outils nécessaires pour l'analyser. On se donne donc n, produit de deux grands nombres premiers p et q, n est public, alors que p et q sont secrets. La sécurité repose sur le fait que multiplier de grands entiers est rapide (moins d'une seconde pour 1 millions de digits) alors que factoriser un entier (avec les algorithmes connus actuellement) prend beaucoup de temps dès qu'on dépasse la centaine de chiffres et est actuellement impossible au-delà d'environ 200 chiffres. Exemple avec p et q pas trop grands :

```
@@p:=nextprime(rand(10^30)); q:=nextprime
(rand(10^23));n:=p*q;
```

$$930321054538491682835526499513, 360248812302365203486 33, 335147054957375551976527360292157074306529248647 15$$

```
@@ifactor(n);
```

$$930321054538491682835526499513 \cdot 360248812302365203486 33$$

On se donne également e et d deux entiers qui vérifient la relation

$$ed \bmod (p-1)(q-1) = 1 \quad \Leftrightarrow \quad ed = m(p-1)(q-1)+1 \ (m \in \mathbb{N}) \tag{1}$$

e est public, alors que d est secrète.
Exemple de paire de clefs :

```
@@E:=65537; phi:=(p-1)*(q-1);d:=inv(E mod
 phi) mod 0;if 0>d then d:=d+phi; fi;irem
(d*E,phi);
```

$$65537, 335147054957375551976518057081251440577398528178675 84, -101939688931753425593175014874920521483587 32$$

Une donnée à sécuriser est représentée par un entier $a < n$, la fonction de codage est

$$b = E(a) = a^e \bmod n$$

ses paramètres sont publics donc n'importe qui peut coder une donnée. On montrera que la fonction de décodage est

$$a = D(b) = b^d \bmod n$$

Seul le détenteur de d peut décoder. On a donc la possibilité d'échanger des données de manière sécurisée sans avoir besoin d'échanger des clefs auparavant.
Illustration : on code les caractères ASCII du mot Bonjour puis on décode (attention, dans la réalité il ne faut surtout pas coder caractère par caractère de cette manière)

```
@@l:=asc("Bonjour");msg:=powmod(l,E,n);dec:=powmod
(msg,d,n);char(dec);
```

$$[66, 111, 110, 106, 111, 117, 114] , [319786588855163696926635889708620479819479369374138 75, 264669639124782169832 42$$

Le fait que les fonctions D et E soient réciproques l'une de l'autre est une conséquence du :
Petit théorème de Fermat : soit p est un nombre premier, et a un entier, alors $a^p - a$ est divisible par p.

```
@@p:=23;[powmod(a,p,p)-a for a in range(p)]
```

$$23, [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$

On a $a^p - a = a(a^{p-1} - 1)$, donc si a n'est pas un multiple de p, on en déduit que $a^{p-1} - 1$ est divisible par p puisque p est premier. Donc $a^{k(p-1)} - 1$ est divisible par p car $x^k - 1 = (x-1)(x^{k-1} + ... + 1)$ (prendre $x = a^{p-1}$). En multipliant par a, on conclut que pour tout a entier, $a^{k(p-1)+1} - a$ est divisible par p, donc $a^{ed} - a$ est divisible par p (prendre $k = m(q-1)$ dans (1)). Pour les mêmes raisons $a^{ed} - a$ est divisible par q, donc il est divisible par $n = pq$ puisque p et q sont des nombres premiers distincts, CQFD.

Le petit théorème de Fermat peut se montrer par récurrence sur a. Rappelons d'abord la formule du binôme :

$$(a+b)^p = \sum_{k=0}^{p} \frac{p!}{k!(p-k)!} a^k b^{p-k}$$

En effet quand on développe le produit, il y a autant de termes $a^k b^{p-k}$ que de choix de k a dans une liste de p éléments, sans tenir compte de l'ordre. Si on tient compte de l'ordre, choisir k éléments parmi p laisse

$$p(p-1)...(p-k+1) = \frac{p!}{(p-k)!}$$

possibilités (p choix pour le premier élément, $p-1$ pour le second, etc.). Il faut ensuite diviser par les ordres possibles entre les k éléments choisis, i.e. $k!$.

Pour $a = 0$, $a^p - a = 0$ est bien divisible par p. Pour montrer l'hérédité, on développe $(a+1)^p$ avec la formule du binôme,

$$(a+1)^p = \sum_{k=0}^{p} \frac{p!}{k!(p-k)!} a^k$$

tous les termes de la somme d'indices $k = 1$ à $k = p-1$ sont divisibles par p, donc $(a+1)^p - (a^p + 1)$ est divisible par p, l'hypothèse de récurrence nous donne $a^p - a$ divisible par p, on additionne et on conclut.

Remarque : si on développe $(a+b)^{p+1} = (a+b)^p(a+b)$ en appliquant la formule du binôme des deux cotés, on obtient une formule donnant les coefficients binomiaux de la ligne $p+1$ en fonction de ceux de la ligne p

$$\binom{p+1}{k} = \binom{p}{k} + \binom{p}{k-1}$$

qui permet de calculer les coefficients binomiaux par un algorithme itératif simple.
Exercice : Implémenter cet algorithme, et vérifier pour quelques valeurs que tous les coefficients sauf le premier et le dernier d'une ligne d'indice p premier sont divisibles par p

```
@@Pascal(11);
```

Pascal (11)

La sécurité du cryptage est liée au temps de calcul de la factorisation de l'entier n, produit de deux grands nombres premiers p et q relativement au temps de calcul du produit de ces deux nombres premiers p et q. Encore faut-il s'assurer qu'on peut coder et décoder en un temps raisonnable, c'est à dire calculer efficacement a^e mod n (ou b^d mod n c'est le même algorithme). Pour cela on utilise l'algorithme dit de la puissance modulaire (ou exponentiation modulaire) rapide. Il repose sur l'observation suivante :

- si m est nul, alors $a^m = 1$
- si m est pair, alors $a^m = (a^{m/2})^2$
- si m est impair, alors $a^m = a * (a^{(m-1)/2})^2$

Attention, pour rendre cet algoritme récursif efficace, il faut veiller à faire les calculs intermédiaires modulo n. On effectue alors des multiplications d'entiers plus petits que n, et le nombre de multiplications est de l'ordre du nombre de bits de m au lieu de m, c'est ce qui fait toute la différence (un millier d'opérations contre 10^{300}), comme on le voit sur le calcul ci-dessous (avec m beaucoup plus petit que 10^{300} pour que l'algorithme lent se termine)

```
@@n:=10^5; a:=1234; m:=567890;time(b:=powmod
(a,m,n)); c:=1; time(for j in range(m) do
 c:=irem(c*a,n); od); b-c;
```

$$100000, 1234, 567890, \left[1.1 \times 10^{-6}, 1.08538394 \times 10^{-6}\right], 1, [0.33, 0.307993352], 0$$

Dans la suite de ce cours, nous allons "industrialiser" ces preuves et méthodes, en étudiant l'arithmétique des entiers, des entiers modulo un entier, on parlera aussi de polynômes, de groupes commutatifs, d'anneaux euclidiens, de corps finis premiers, et un peu de temps de calcul.

1.2 Codes

La théorie de l'information et du codage est beaucoup plus récente que la cryptographie (dont l'intérêt militaire est évident), elle date des années 50 (Shannon, Hamming). Il ne s'agissait pas à l'époque de télécommunications mais de pallier à la fiabilité des cartes perforées.

L'idée est d'ajouter de la redondance, par exemple un code très simple consiste à répéter 3 fois chaque bit. Si en réception les 3 bits sont identiques, on considère que la valeur commune est la bonne, sinon il y a eu une erreur, et on peut la corriger sous l'hypothèse qu'il y a eu au plus une erreur, en votant à la majorité. Mais ce système est peu efficace si la probabilité d'erreur sur 1 bit est faible, car il utilise 3 fois plus de bande passante qu'il n'est nécessaire pour transmettre l'information sur un canal 100% fiable.

On verra vers la fin du cours des méthodes permettant de détecter et corriger des erreurs avec une efficacité meilleure, ces méthodes ont besoin de structure qui sera apportée par les espaces vectoriels dont les vecteurs ont des coordonnées dans $\{0,1\}$ au lieu de $\mathbb{Q}, \mathbb{R}$ ou $\mathbb{C}$ (on se limite à $\mathbb{Z}/2\mathbb{Z}$ dans le cadre de ce cours). On parle de *codes linéaires*. Par exemple, le code de Hamming binaire 7,4 encode un quartet (i.e. un 4-uplet de bits) x par un 7-uplet $y = Gx$, en multipliant modulo 2 par la matrice G le vecteur x :

```
@@G:=[[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1
],[1,1,0,1],[1,0,1,1],[0,1,1,1]];x:=[0,1,1,0
];y:=G*x mod 2;
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}, [0,1,1,0], [0\,\%\,2, 1\,\%\,2, 1\,\%\,2, 0\,\%\,2, 1\,\%\,2, 1\,\%\,2, 0\,\%\,2]$$

On observe que les 4 premières lignes de G forment la matrice identité ce qui permet facilement de reconstituer x si y a été transmis correctement `x=y[0:3]`

Pour détecter une erreur, on vérifie que y est dans le noyau de la matrice

```
@@H:=[[1,1,0,1,1,0,0],[1,0,1,1,0,1,0],[0,1,1,1,0,0,1
]];H*y;
```

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}, [0\,\%\,2, 0\,\%\,2, 0\,\%\,2]$$

S'il n'y a pas eu d'erreur détectable, alors $Hy = 0$ (modulo 2). S'il y a une erreur au plus, alors Hy est à une permutation près l'écriture en base 2 du numéro de composante de y à modifier.

```
@@p:=[2,4,5,6,0,1,3]; pinv:=perminv(p);
```

$$[2,4,5,6,0,1,3], [4,5,0,6,1,2,3]$$

Par exemple, on simule une erreur de transmission en changeant le bit d'indice n :

```
@@y:=G*x mod 2:; n:=4;y[n]:=1-y[n]:;s:=H*y;s:=s
 mod 0; s:=convert(s,base,2)-1;pinv[s];
```

"Done", 4, "Done", [1 % 2, 0 % 2, 0 % 2] , [1, 0, 0] , 0, 4

Ici `s mod 0` a pour effet d'enlever les `%` pour avoir une liste de 0 et de 1, que l'on convertit en entier et à qui on applique la permutation inverse de p, on retrouve bien l'indice de l'erreur (vous pouvez le vérifier en changeant la valeur de n).

2 Arithmétique des entiers et polynômes.

Définition 2.1 *Si a et b sont deux entiers naturels, $b \neq 0$, on définit le* **quotient euclidien** *de a par b comme le plus grand entier naturel q tel que $a - bq \geq 0$ (l'ensemble des entiers naturels tels que $a - bq \geq 0$ est borné par a donc admet un élément maximal), On définit r le* **reste euclidien** *de a par b par $r = a - bq$. On vérifie que $r \in [0, b-1]$ (sinon $a - b(q+1)$ serait positif ou nul).*

2.1 Petits et grands entiers

Les microprocesseurs sont capables d'effectuer des opérations arithmétiques de base (toujours $+, -$, pratiquement toujours $\star$ et la division euclidienne) sur des petits entiers compris dans l'intervalle $[-2^{t-1}, 2^{t-1}-1]$ (entier signé) ou $[0, 2^t - 1]$ (entier non signé), pour $t = 8$, $t = 16$, $t = 32$ et $t = 64$ (parfois aussi pour $t = 128$). En cas de dépassement, les calculs sont faits modulo 2^t. Les logiciels de calcul doivent pouvoir travailler avec des entiers de taille plus grande, il faut les stocker en mémoire et implémenter les opérations arithmétiques de base en se ramenant à des petits entiers. Pour faire cela, on va écrire des entiers dans des bases adéquates.

2.1.1 Ecriture en base b

Il s'agit de généraliser l'écriture des entiers qui nous est familière en base 10 à une base quelconque, les ordinateurs préférant la base 2 ou une puissance de 2. Ainsi écrire 1602 signifie que

$$1602 = 1 \times 10^3 + 6 \times 10^2 + 0 \times 10^1 + 2$$

La taille d'un nombre est le nombre de chiffres qu'il contient, ainsi 1602 s'écrit avec 4 chiffres (en base 10). Les nombres à 4 chiffres sont compris entre $1000 = 10^3$ et $10000 - 1 = 10^4 - 1$, plus généralement, un nombre compris entre 10^{n-1} et $10^n - 1$ sécrit avec n chiffres. Si on change pour une base $b \geq 2$ quelconque, on a alors :

Théorème 2.2 *On se fixe une fois pour toutes un entier $b \geq 2$. Soit N un entier naturel non nul, on peut alors écrire N de manière unique sous la forme*

$$N = \sum_{i=0}^{n} N_i b^i, \quad N_i \in [0, b-1], N_n \neq 0 \tag{2}$$

Les N_i sont appelés bits si $b = 2$ ou digits si $b = 10$ (parfois aussi pour $b > 2$, par exemple $b = 16$).
On a $n =$`floor(`$\frac{\log(N)}{\log(b)}$`)` où `floor` est la partie entière (plus grand entier inférieur ou égal). (Attention, le nombre de chiffres de N est $n + 1$).

On remarque que le nombre n de digits pour écrire N en base b est `floor(`$\log_b(N)$`)`, en effet d'une part :

$$b^n \leq N_n b^n \leq N$$

et d'autre part :

$$N < \sum_{i=0}^{n} N_i b^i \leq (b-1) \sum_{i=0}^{n} b^i = b^{n+1} - 1 < b^{n+1}$$

puis on prend les log.

Pour montrer l'unicité de l'écriture, on suppose qu'il y a deux écritures distinctes de ce type, et on regarde le coefficient d'indice maximal qui est différent (appelons le j), on a alors

$$(\tilde{N}_j - N_j)b^j = \sum_{i=0}^{j-1}(N_i - \tilde{N}_i)b^i$$

quitte à changer de signe, on peut supposer que le membre de gauche est strictement positif, on a alors

$$(N_j - \tilde{N}_j)b^j \geq b^j$$

Mais le membre de droite se majore par

$$\sum_{i=0}^{j-1}(b-1)b^j = b^j - 1$$

absurde.

L'existence se prouve par l'algorithme suivant qui utilise la division euclidienne (on a utilisé `irem(N,b)` au lieu de `N % b` pour calculer le reste de division euclidienne pour des raisons de compatibilité avec LaTeX)

```
def ecriturebase(N,b):
    L=[]
    while N>0:
        L.append(irem(N, b))
        N=N//b
    return L
```

```
@@ecriturebase(1234,16);convert(1234,base,16);
```

$$[2, 13, 4], [2, 13, 4]$$

i.e. tant que $N > 0$, on calcule le reste de N par b, qui donne le coefficient de poids le plus faible de l'écriture de N en base b, on ajoute lécriture en base b du quotient de N par b. L'algorithme s'arrête au bout de partie entière de $\log_b(N)+1$ itérations. Réciproquement, on vérifie que l'écriture obtenue convient en développant

$$N_0 + b(N_1 + b(... + b(N_{n-1} + b(N_n))...)) \tag{3}$$

On observe au passage que l'écriture de N sous la forme ci-dessus nécessite n additions et n multiplications, et est donc plus efficace que le calcul sous la forme développée 2. C'est la méthode dite de Hörner.

Exemple : en base $b = 2$. Pour écrire $N = 12$ en base 2, on calcule le reste de 12 par 2 donc $N_0 = 0$, le quotient de 12 par 2 vaut 6, on divise par 2, reste 0 donc $N_1 = 0$, quotient 3, on divise par 2, reste $N_2 = 1$, quotient 1, on divise par 2, reste $N_3 = 1$ quotient 0 on s'arrête, donc `12=0b1100`. Réciproquement on a bien

$$0 + 2 \times (0 + 2 \times (1 + 2 \times (1))) = 12$$

Exercice : la commande `b:=convert(N,base,b)` de Xcas effectue la conversion d'un entier N en la liste des coefficients de son écriture en base b, la réciproque étant `convert(L,base,b)` ou `horner(L,b)`. Implémenter des fonctions équivalentes dans votre langage de programmation préféré.

Exercice : comment passe-t-on simplement de la représentation d'un nombre en base 2 à un nombre en base 16 et réciproquement ?

2.1.2 Lien avec les polynômes de $A[X]$ avec $A = \mathbb{Z}$.

A tout entier naturel N, on peut associer un polynôme à coefficients entiers qui est son écriture en base b, les coefficients du polynôme sont dans l'intervalle $[0, b-1]$. Réciproquement, si les coefficients d'un polynôme sont des entiers compris entre $[0, b-1]$ alors ils correspondent à l'écriture d'un entier en base b.

On va voir que les opérations arithmétiques de base sur les grands entiers reviennent à effectuer des opérations de base sur les polynômes, avec une petite difficulté supplémentaire, il faut tenir compte de retenues. Les algorithmes naifs pour additionner et multiplier deux polynômes correspondent précisément aux algorithmes enseignés à l'école primaire pour effectuer l'addition ou la multiplication de deux entiers.

On travaille sur ordinateur avec une base b qui est une puissance de 2, par exemple on pourrait prendre $b = 2^8$ et dans ce cas les entiers s'écrivent avec des chiffres qui sont des octets. Ainsi `\nAa` représente

$$13 * 256^2 + 65 * 256 + 97$$

En pratique, sur des processeurs 32 bits, on utilise $b = 2^{32}$ les chiffres sont des double-mots (dword), sur des processeurs 64 bits $b = 2^{64}$ les chiffres sont des quadruple mots (qword).

2.2 Opérations arithmétiques de base

2.2.1 Addition, soustraction

Si $N = \sum_{i=0}^{n} N_i x^i$ et $M = \sum_{i=0}^{m} M_i x^i$ sont deux polynômes, alors

$$N + M = \sum_{i=0}^{\max(n,m)} (N_i + M_i) x^i,$$

Si $N = \sum_{i=0}^{n} N_i b^i$ et $M = \sum_{i=0}^{m} M_i b^i$ sont deux entiers écrits en base b, alors

$$N + M = \sum_{i=0}^{\max(n,m)} (N_i + M_i) b^i,$$

il faut donc additionner les digits de l'écriture en base b comme pour additionner deux polynômes. On a $N_i + M_i \in [0, 2b-2]$, si $N_i + M_i < b$, il n'y a rien à faire, sinon on remplace par $N_i + M_i - b$ et on ajoute 1 (retenue) à $N_{i+1} + M_{i+1}$ qui appartient à $[0, 2b-2+1]$, etc. Le nombre d'opérations à effectuer est de l'ordre du maximum du nombre de bits/chiffres/digits de N et M.

2.2.2 Multiplication

Si $N = \sum_{i=0}^{n} N_i b^i$ et $M = \sum_{j=0}^{m} M_j b^i$, alors on se ramène à la multiplication de deux polynômes :

$$N \times M = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i M_j b^{i+j}$$

Si $b > 2$, par exemple $b = 10$ ou $b = 16$, $N_i M_j$ sera très souvent supérieur à b, il y aura souvent des retenues ! On peut regrouper les termes b^{i+j} ayant le même exposant, en utilisant des décalages pour tenir compte de b^i (c'est l'algorithme de multiplication posée en base 10) ou additionner au fur et à mesure $N_i M_j$ au coefficient actuel de b^{i+j} (exercice de la feuille de TD). En base 2, si on pose la multiplication comme en base 10, il est conseillé d'effectuer les additions une par une, sinon la prise en compte des retenues est délicate.

Le nombre de multiplications et d'additions est de l'ordre de nm (on peut montrer que c'est aussi le cas en tenant compte des retenues, car si une retenue se propage de r positions dans la boucle intérieure en j, elle ne pourra pas se propager de plus de 2 positions pour les $r-2$ itérations suivantes de la boucle intérieure).

Taille du résultat : si on multiplie un nombre à 3 chiffres par un nombre à 2 chiffres, le résultat aura 4 ou 5 chiffres, penser par exemple à 412×11 ou 412×31. Dans le cas général,

$$b^n \leq N < b^{n+1}, \quad b^m \leq M < b^{m+1} \quad \Rightarrow b^{n+m} \leq NM < b^{n+m+2}$$

donc le produit NM a pour taille $n+m+1$ ou $n+m+2$ bits/chiffres/digits.

Si n et m sont proches, par exemple égaux, le résultat a pour taille $2n$ alors que l'algorithme présenté nécessite n^2 opérations. On peut donc espérer l'existence d'algorithmes plus efficaces (un peu comme les algorithmes de tri...). Nous allons en présenter un dans le cadre plus simple de la multiplication des polynômes (où on n'a pas à gérer les retenues).

2.2.3 Diviser pour régner : multiplication de Karatsuba.

Soient P, Q deux polynômes de degrés strictement inférieur à $2n$. On suppose que le cout d'une opération arithmétique dans l'ensemble des coefficients vaut 1 et on néglige les autres opérations (on suppose par exemple que l'ensemble des coefficients est fini). On écrit

$$P = A + x^n B, \quad Q = C + x^n D$$

avec A, B, C, D de degrés strictement inférieur à n, on a alors :

$$PQ = AC + x^n(AD + BC) + x^{2n} BD$$

Il y a 4 produits de polynômes de degrés $< n$, mais au prix d'additions intermédiaires, on peut se ramener à 3 produits, en effet

$$(A + B)(C + D) - AC - BD = AD + BC$$

donc pour calculer le facteur de x^n il suffit de soustraire à $(A + B)(C + D)$ les produits AC et BD que l'on doit calculer par ailleurs. Au premier ordre, le temps nécessaire pour multiplier les 2 polynômes de degré $< 2n$ est multiplié par 3, au lieu de 4.

Plus précisément, soit $M(n)$ le temps nécessaire pour calculer le produit de 2 polynômes par cette méthode, on a alors

$$M(2n) = 3M(n) + 8n$$

où $8n$ représente le nombre d'additions ou de soustractions pour former $A + B$, $C + D$, soustraire AC et BD, et tenir compte du fait que les termes de degré $\geq n$ de AC se combinent avec ceux de degré $< 2n$ de $AD + BC$ et les termes de degré $< 3n$ de $x^{2n} BD$ avec ceux de degré $\geq 2n$ de $AD + BC$. On en déduit

$$u_n = M(2^n), \quad u_{n+1} = 3u_n + 8 \times 2^n$$

cette récurrence se résoud facilement par la commande

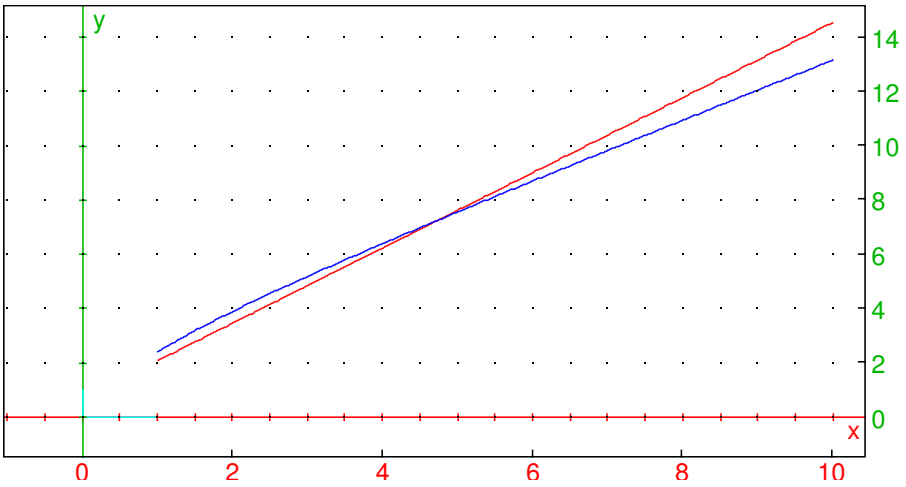
```
@@purge(n):; rsolve(u(n+1)=3*u(n)+8*2^n,u
(n),u(0)=1)
```

$$\text{"Done"}, [-8 \cdot 2^n + 9 \cdot 3^n]$$

ce qui donne $M(2^n) = u_n = -8 \cdot 2^n + 9 \cdot 3^n$.

Asymptotiquement, $M(2^n) \approx 9 \cdot 3^n$ ce qui est bien meilleur que la multiplication naive en $2 \cdot 4^n$, mais pour de petites valeurs de n, la multiplication naive est plus rapide comme l'illustre le graphe ci-dessous (en rouge la multiplication naive, en bleu Karatsuba complet) :

```
@@plotfunc([log(2*4^n),log(-8*2^n+9*3^n)
],n=1..10,display=[red,blue])
```



On utilise Karatsuba (récursivement) uniquement pour des valeurs de n suffisamment grandes (théoriquement lorsque $8n$, le surcout dû aux additions est plus petit que la multiplication économisée, soit $8n < 2n^2$ soit $n > 4$, en pratique plutôt pour n de l'ordre de quelques dizaines selon les implémentations, car nous n'avons tenu compte que des opérations arithmétiques).

Exercice : vérifier que la multiplication des entiers longs en langage Python se comporte pour n grand comme la multiplication de Karatsuba.

Les logiciels de calcul intensif utilisent la plupart du temps des algorithmes plus efficaces lorsque n est encore plus grand, qui se rapprochent de $n\ln(n)$. C'est en particulier le cas de tous les logiciels de calcul formel qui utilisent la libraire GMP écrite en langage C (Xcas, Sagemath, Maple, Mathematica, ...). Ces algorithmes utilisent la transformation de Fourier rapide (FFT) en calcul exact. Voir par exemple (en anglais) ce lien.

2.3 Opérations de base sur les petits entiers.

Les petits entiers sont représentés en base $b = 2$, les opérations de base sont directement faites par le microprocesseur en se ramenant aux calculs en base 2 :

- addition : 0+0=0=1+1 (avec retenue), 0+1=1=1+0 qui se traduisent enou exclusif logique hors retenues. Les opérations bit à bit se parallélisent facilement, conduisant à des temps d'éxecution de 1 cycle.

```
  01001111
+ 01101011
----------
  10111010
```

- multiplication : 0*0=0*1=1*0=0, 1*1=1 qui se traduit en et logique. Les opérations de multiplication se parallélisent aussi, conduisant à des temps d'exécution de un ou quelques cycles sur la plupart des processeurs actuels (mais ce n'était pas vrai il y a une vingtaine d'années).
- l'algorithme de division euclidienne en base 2 peut se faire avec l'algorithme de la potence, il consiste à chercher les bits de l'écriture en base 2 du quotient en commençant par le bit de poids fort. La situation est simplifiée par rapport à un calcul en base 10 parce que le bit du quotient à déterminer ne peut être que 0 ou 1, il suffit de comparer avec le reste en cours. Le temps d'exécution est significativement plus élevé que celui d'une multiplication (plusieurs dizaines de cycle si $t = 32$ ou $t = 64$), car q n'est pas connu quand on calcule bq, alors que a et b sont connus quand on calcule ab.

```
 110000    | 101
-101       |---------
----       | 1001
  010      |
   100     |
   1000    |
  - 101    |
  -----    |
    011    |
```

2.4 Opérations de base sur les grands entiers

Les grands entiers naturels sont représentés dans une base qui est une puissance de 2, $b = 2^{t'}$, les coefficients de l'écriture étant de petits entiers, ils sont stockés en mémoire à des adresses consécutives. On utilise le plus souvent $t' = t$ la taille des régistres du microprocesseur, parfois une valeur légèrement inférieure pour simplifier la gestion des retenues.

Les implémentations de la multiplication font intervenir des algorithmes plus efficaces (comme Karatsuba), ainsi que des découpages par blocs visant à minimiser les opérations de lecture/écriture en mémoire au profit de l'utilisation de régistres et de l'optimisation du stockage dans la mémoire cache (L1). Avec les processeurs modernes, le cout du calcul d'un produit de 2 grands entiers par la méthode naive est dominé par les accès mémoire en RAM et non par les opérations arithmétiques sur les coefficients.

Expérience avec un langage compilé, ici C++, pour la multiplication de deux polynômes par la méthode naïve.

```
#include <iostream>
#include <vector>
using namespace std;

template<class T> ostream & operator << (ostream & os,const vector<T> & v){
  if (v.empty()) return os << "[ ]";
  const T * ptr=&v.front(),*ptrend=ptr+v.size();
  os << "[";
  for (;ptr!=ptrend;++ptr){
    os << *ptr << " ";
  }
  return os << "]";
}

template<class T>
void mul1(vector<T> & A,const vector<T> & M,const vector<T> & N){
  A=vector<T>(M.size()+N.size()-1);
  for (size_t i=0;i<M.size();++i){
    T Mi(M[i]);
    for (size_t j=0;j<N.size();++j){
      A[i+j] += Mi*N[j]; // 1 lecture et 1 ecriture
    }
  }
}

template<class T>
void mul2(vector<T> & A,const vector<T> & M,const vector<T> & N){
  A=vector<T>(M.size()+N.size()-1);
  size_t i;
  for (i=0;i+3<M.size();i+=4){
    T M0=M[i], M1=M[i+1], M2=M[i+2], M3=M[i+3];
    T * Aptr=&A[i];
    size_t j=0;
    for (j=0;j+3<N.size();Aptr+=4,j+=4){
      T N0=N[j];
      T N1=N[j+1];
      T N2=N[j+2];
      T N3=N[j+3];
      *Aptr +=   M0*N0;
      Aptr[1] += M1*N0+M0*N1;
      Aptr[2] += M2*N0+M1*N1+M0*N2;
      Aptr[3] += M3*N0+M2*N1+M1*N2+M0*N3;
      Aptr[4] +=       M3*N1+M2*N2+M1*N3;
      Aptr[5] +=             M3*N2+M2*N3;
      Aptr[6] +=                   M3*N3;
    }
    for (;j<N.size();++Aptr,++j){
      T Nj=N[j];
      *Aptr += M0*Nj;
      Aptr[1] += M1*Nj;
      Aptr[2] += M2*Nj;
      Aptr[3] += M3*Nj;
    }
  }
```

```
  // fin du produit
  for (;i<M.size();++i){
    T Mi(M[i]);
    for (size_t j=0;j<N.size();++j){
      A[i+j]+=Mi*N[j];
    }
  }
}

int main(int argc,char ** argv){
  if (argc<2){
    cout << "Syntaxe " << argv[0] << " <entier> [methode 1 ou 2]" << endl;
    return 1;
  }
  int n=atoi(argv[1]);
  int algo=argc==3?atoi(argv[2]):1;
  vector<int> A,M(n),N(n);
  // on remplit M et N
  for (int i=0;i<n;++i){
    M[i]=i+2;
    N[i]=i+1;
  }
  cout << M << "*" << N << endl;
  if (algo==1)
    mul1(A,M,N);
  else
    mul2(A,M,N);
  cout << A << endl;
  return 0;
}
```

Compilation (sans utiliser les optimisations SSE) avec `c++ -O2 -mno-sse prog1.cc`. Appel de l'algorithme non optimisé, puis optimisé, puis vérification par exemple par

```
time ./a.out 100000 1 >& log1
time ./a.out 100000 2 >& log2
diff -q log1 log2
```

On gagne presque un facteur 2 en temps.

La fonction `mul1` effectue une lecture et une incrémentation par produit de coefficients, donc nm lecture et nm incrémentation où n et m désignent le nombre de coefficients des polynômes N et M. La fonction `mul2` stocke dans des régistres 4 coefficients de M et de N pour faire leur produit, elle effectue moins de la moitié d'opérations de lecture/écriture en RAM (asymptotiquement 4 lectures et 7 incrémentations pour 16 produits de coefficients donc $nm/4$ lectures et $7nm/16$ incrémentations). Le nombre d'opérations arithmétiques sur les coefficients est identique.

2.5 Points à retenir absolument

- L'algorithme pour écrire un nombre en base b, et le calcul inverse.
- Le nombre de chiffres de a est proportionnel au log de a.
- Addition et soustraction se font avec l'algorithme vu à l'école primaire, le cout est proportionnel au max du nombre de chiffres de a et b
- Le produit de deux entiers a pour taille la somme des tailles à un près, l'algorithme école primaire a un cout proportionnel au produit des tailles. Il existe des algorithmes plus rapides.

3 Euclide

Dans cette section, on se concentre sur tout ce qui tourne autour de la division euclidienne.

3.1 Division euclidienne.

On a vu la définition de la division euclidienne pour les entiers naturels $a = bq + r, r \in [0, b-1]$. Comment calcule-t-on la représentation dans une base (que l'on notera β) de q et r ? Si $a < b$ alors $b = 0$ et $r = a$. Si $a \geq b$, le calcul de q et r en base β peut se faire par l'algorithme dit de la potence.

On suppose donc $a \geq b$. Posons

$$a = \sum_{j=0}^{n} a_j \beta^j, \quad b = \sum_{j=0}^{m} b_j \beta^j, \quad n \geq m, \quad q = \sum_{j=0}^{l} q_j \beta^j$$

On observe que l'écriture de q en base β n'a pas de digit d'indice l supérieur strict à $n-m$, car $a - b\beta^{n-m+1} < 0$. La valeur du digit q_{n-m} d'indice $n-m$ de q (digit éventuellement nul) est le plus grand entier $q_{n-m} \in [0, \beta - 1]$ tel que $a - bq_{n-m}\beta^{n-m} \geq 0$. Comme

$$\sum_{j=0}^{n-m-1} a_j \beta^j < \beta^{n-m}$$

cela revient à chercher le plus grand digit tel que

$$\sum_{j=n-m}^{n} a_j \beta^j - bq_{n-m}\beta^{n-m} \geq 0$$

on regarde donc les $m+1$ digits de a les plus significatifs et on calcule le quotient euclidien par b. Si $q_{n-m} \neq 0$, on effectue alors la soustraction

$$\tilde{a} = \sum_{j=0}^{n} a_j \beta^j - bq_{n-m}\beta^{n-m} = \sum_{j=0}^{n-m-1} a_j \beta^j + \beta^{n-m}(\sum_{j=n-m}^{n} a_j \beta^{j-(n-m)} - bq_{n-m})$$

Pour déterminer le digit q_{n-m-1}, on recommence la procédure ci-dessus, avec les au plus $m+2$ digits de $\tilde{a}$ d'indices supérieurs ou égaux à $n-m-1$.

Le cout de l'algorithme de la potence est proportionnel au produit $(n-m+1)(m+1)$, le cas le pire correspond à m proche de $n/2$ qui donne un cout proportionnel à n^2 (dans ce cas, on peut accélérer par un algorithme diviser pour régner de type Karatsuba).

Pour les entiers relatifs, d'autres conventions de signe sont possibles pour q et r. Ainsi, la division euclidienne en langage C renvoie q tronqué vers 0. Si $b > 0$, `a % b` sera du signe de a. Pour obtenir le reste dans $[0, b[$ lorsque $a < 0$ (convention de signe de reste positif), il faudra ajouter b à r et diminuer q de 1.

Optimisation : sur les microprocesseurs modernes, on évite si possible l'utilisation de tests pour optimiser le temps de calcul. En C, si on divise par $b > 0$, et si a est de signe quelconque, tous deux des entiers 31 bits, alors `unsigned(a)>>31` vaut 0 si $a \geq 0$ et 1 sinon, on peut donc écrire :

```
inline int reste(int a,int b){
  return a
}
```

On peut aussi adopter la convention dite du reste symétrique, c'est-á-dire que $r \in]-\frac{|b|}{2}, \frac{|b|}{2}]$. En général, on peut supposer que $b > 0$, ce qui donne le programme suivant :

```
def rsym(a,b):
    r = irem(a, b)
```

```
    if r<0:
        r += b
    if r<=b//2:
        return r
    return r-b

@@rsym(5,7); rsym(3,7)
```

$$-2, 3$$

Optimisation :

```
int rsym(int r,int m){
  r
  r += (unsigned(r)>>31)*m; // rend r positif
  return r-(unsigned((m>>1)-r)>>31)*m;
}
```

Le reste de division euclidienne (`%`) est ici l'opération la plus couteuse, il existe heureusement parfois des méthodes qui permettent de l'éviter, par exemple on verra plus bas l'algorithme de PGCD binaire ou lorsqu'on fait des calculs modulo un entier fixé.

Pour les polynômes en une variable, si A et B sont deux polynômes à coefficients dans un corps K, il existe un unique polynôme Q et un unique polynôme R tels que :

$$A = BQ + R, \quad \deg(R) < \deg(B)$$

L'unicité vient du fait que $R - R' = B(Q - Q')$ est incompatible avec les degrés si $Q \neq Q'$. L'existence se prouve par récurrence sur la différence de degré entre A et B, si elle est négative $Q = 0, R = A$, si elle est nulle Q est le quotient des coefficients dominants A_a de A de degré a et B_b de B de degré b, sinon, on pose $Q = x^{a-b}A_a/B_b + ...$, ce qui annule le coefficient dominant de A et on est ramené à une différence de degré diminuée de 1 (au moins).

Remarque 1 : si A et B sont à coefficients entiers et si B est unitaire, alors Q et R sont à coefficients entiers. Sinon, on peut définir la pseudo-division de A par B en multipliant A par le coefficient dominant de B à la puissance la différence des degrés + 1.

Remarque 2 : pour les polynômes à plusieurs variables, il n'existe malheureusement pas de division euclidienne.

3.2 Anneau euclidien

L'ensemble des entiers relatifs $\mathbb{Z}$ est muni des opérations + et $\star$ qui en font un anneau commutatif unitaire intègre. On rappelle (ou on définit) que

- un **groupe** possède une loi associative, avec un élément neutre, tel que tout élément possède un symétrique. Si la loi est commutative, on parle de groupe commutatif et on note habituellement la loi +.
- un **anneau** A dispose de deux lois, + et $\star$, telles que A est un groupe commutatif pour +, et que la loi $\star$ est associative et distributive par rapport à +.
- si la loi $\star$ est commutative, on parle d'**anneau commutatif**
- si la loi $\star$ admet un élément neutre (1), on parle d'**anneau unitaire**.
- si la relation $ab = 0$ entraine $a = 0$ ou $b = 0$, on parle d'**anneau intègre**. On dit aussi qu'il n'y a pas de diviseurs de zéro.

L'ensemble des entiers relatifs est donc un anneau commutatif unitaire intègre, mais il a une propriété supplémentaire, il possède une opération de division euclidienne pour $a \in \mathbb{Z}$ et b non nul, il existe q et r (uniques) tels que $a = bq + r$ avec $r \in [0, b[$. Cette propriété est également vraie pour les polynômes à coefficients dans un corps avec une condition de degré $A = BQ + R$ avec degree(R)<degree(B). Plus généralement, on parle de **stasthme** pour désigner l'équivalent du degré pour un polynôme ou de la valeur absolue de l'entier dans $\mathbb{Z}$.

3.3 Divisibilité

Définition 3.1 *On dit que b divise a (ou que a est divisible par b) si le reste de la division de a par b est 0.*

Ceci s'applique aussi bien aux entiers qu'aux polynômes. On observe que l'entier 0 est divisible par tous les entiers non nuls, et le polynôme nul par tous les polynômes non nuls. Si a et b sont des entiers naturels non nuls et si a est divisible par b alors $a \geq b$. En effet $a = bq$ avec $q \geq 1$ puisque $a \neq 0$. Si A et B sont deux polynômes non nuls et si A est divisible par B alors le degré de A est supérieur ou égal au degré de B.

Observation : 1 n'est divisible que par 1 et -1, les seuls entiers ayant un inverse pour la multiplication dans $\mathbb{Z}$ sont 1 et -1. Pour les polynômes à coefficients dans un corps K, le polynôme 1 est divisible par les polynômes constants non nuls (ceci est une conséquence du degré d'un produit qui est égal à la somme des degrés des facteurs), les polynômes constants sont les seuls polynômes ayant un inverse pour la multiplication dans les polynômes. Ces éléments, appelés inversibles de l'anneau, forment un groupe, ils jouent un role important, en effet si b divise a, i.e. $a = bq$ alors pour tout élément inversible i, bi divise a puisque $a = (bi)i^{-1}q$. Dans le cas des entiers, cela signifie qu'on peut négliger le signe pour étudier la divisibilité, autrement dit on peut se limiter aux entiers naturels. Dans le cas des polynômes, on peut multiplier par n'importe quel coefficient non nul, autrement dit on peut se limiter aux polynômes unitaires (i.e. dont le coefficient dominant est 1).

La relation de divisibilité dans l'ensemble des entiers naturels non nuls vérifie les propriétés suivantes :

- réflexivité : a divise a
- transitivité : si a est divisible par b et b par c alors a est divisible par c, en effet $a = bq$ et $b = cq'$ alors $a = cqq'$.
- antisymétrie : si a est divisble par b et b par a, alors $a = bq$ et $b = aq'$ donc $a = aqq'$ et $qq' = 1$ donc $q = q' = 1$ et $b = a$.

Ces propriétés sont aussi vérifiées par la relation $\leq$ sur les entiers (ou les réels). Une relation réflexive (aRa), antisymétrique (aRb et bRa entraine $a = b$) et transitive (aRb et bRc entraine aRc) est appelée **relation d'ordre**. Il s'agit ici d'une relation d'ordre **partiel**, il y a des éléments qui ne peuvent être comparés, par exemple 2 et 3. Lorsque deux éléments sont toujours comparables, on parle de relation d'ordre total (par exemple $\leq$ sur les entiers ou les réels ou l'ordre lexicographique sur les chaines de caractères ou les listes).

La divisibilité sur les polynômes unitaires est aussi une relation d'ordre partiel.

On verra plus loin un autre type de relation, les relations d'équivalence, où on remplace l'antisymétrie par la symétrie ($a\ R\ b$ entraine $b\ R\ a$).

3.4 Le PGCD

Soient a et b deux entiers. On se ramène au cas des entiers naturels en enlevant les signes.

Définition 3.2 *On définit le PGCD (plus grand commun diviseur) de deux entiers naturels a et b non simultanément nuls comme le plus grand entier naturel d qui divise simultanément a et b. Si a et b sont nuls, on peut adopter comme convention que leur PGCD est nul.*

Si l'un des deux éléments est nul, disons b, alors la réponse est a. Sinon, comme 1 divise a et de b, l'ensemble des diviseurs communs à a et b est non vide, et son plus grand élément est inférieur ou égal au minimum de a et de b, le PGCD existe donc bien.

On verra dans la suite que le PGCD intervient souvent en arithmétique. On peut déjà citer une de ses applications qui est la réduction des fractions sous forme irréductible.

Pour le calculer, on observe que le PGCD de a et de b est le même que celui de b et de r, le reste de la division euclidienne de a par b. En effet si d est un diviseur commun à a et b, alors $a = da'$ et $b = db'$ donc $r = a - bq = d(a' - qb')$. Réciproquement si d est un diviseur commun à b et r, alors $b = db'$ et $r = dr'$ donc $a = bq + r = d(b'q + r')$. Cet algorithme, appelé **algorithme d'Euclide**, se traduit en un programme récursif (on a utilisé `irem(a,b)` pour le reste euclidien au lieu de `a % b` pour raison de compatibilité avec LaTeX) :

```
def pgcdr(a,b):
     if b==0:
          return a
     return pgcdr(b,irem(a,b))
```

```
@@pgcdr(25,35); pgcdr(24,35);
```

$$5, 1$$

A chaque appel récursif, la valeur du reste diminue au moins de 1 puisque $r < b$. Au bout d'un nombre fini d'appels, $b = 0$ et on a le PGCD. Si on utilise la valeur absolue du reste symétrique au lieu du reste euclidien positif, le reste perd (au moins) un bit dans son écriture en base 2 à chaque itération, le nombre d'itérations N est au plus égal au minimum du nombre de bits de a et b.

Notons $r_0 = a, r_1 = b, .., r_N$ la suite des restes de l'algorithme d'Euclide (avec $r_N = 0$ le dernier reste), on a

$$r_k = q_k r_{k+1} + r_{k+2}$$

avec un cout pour cette division proportionnel à $\#q_k \#r_{k+1}$ où $\#a$ est le nombre de digits de l'écriture de a. Or à une unité près $\#q_k = \#r_k - \#r_{k+1}$, on a donc un cout total pour l'algorithme d'Euclide proportionnel à

$$\sum_{k=0}^{N-1} (\#r_k - \#r_{k+1} + 1)\#r_{k+1} \leq \#b(N + \sum_k (\#r_k - \#r_{k+1}) \leq \#b(N + \#a)$$

Donc le cout est en $O(n^2)$ si le nombre de bits de a et b est plus petit que n, en majorant le nombre d'étapes N par n avec la version restes symétriques d'Euclide. C'est encore vrai à une constante près sans prendre les restes symétrique mais un peu plus long à justifier, on peut par exemple utiliser le bonus de l'exercice sur Fibonacci F_n de la feuille de TD et un équivalent de F_n pour n grand. On peut aussi voir que si le reste de la division de a par b est supérieur à $b/2$, alors à l'étape suivante, le quotient vaudra 1, donc le reste suivant sera $b - r < b/2$, donc le nombre d'étapes d'Euclide classique est au plus 2 fois le nombre de bits de l'écriture en base 2 de b.

Exercice : traduire l'algorithme d'Euclide en un programme non récursif.
Solution avec reste symétrique, cf. A.2

Le même algorithme fonctionne pour les polynômes, cette fois on cherche un polynôme de degré le plus grand possible qui divise A et B. L'algorithme ci-dessus s'adapte en remplaçant la division euclidienne des entiers par celle des polynômes, et le nombre d'appels récursifs est fini car le degré du reste qui diminue de 1 au moins à chaque itération. On obtient alors un PGCD, les autres PGCD sont des multiples non nuls de ce PGCD.

Il existe une version adaptée à l'écriture en base 2 des entiers sur machine, appelée PGCD binaire, qui évite de devoir faire des divisions euclidiennes (les divisions par 2 sont des décalages)

- si a ou b est nul on renvoie l'autre,
- si a et b sont pairs pgcd(a,b)=2 pgcd($a/2$,$b/2$), (ne peut se produire qu'au début)
- si a est pair, b impair, pgcd(a,b)=pgcd($a/2$,b),
- le cas symétrique (ne peut se produire qu'au début)
- si les 2 sont impairs pgcd(a,b)=pgcd($|a - b|/2$,min(a,b))

Chaque itération nécessite au plus le nombre n de bits de l'écriture en base 2 de a et b, et il y a au plus $2n$ itérations (car on diminue de 1 le nombre de bits d'au moins un des 2 opérandes). Cet algorithme a donc un coût d'au plus $O(n^2)$, mais la constante devant n^2 est significativement plus petite que celle de l'algorithme d'Euclide car on évite les divisions qui sont des opérations arithmétiques de base couteuses. Sa justification repose sur des propriétés du PGCD qui se déduisent eux-mêmes de l'identité de Bézout.

3.5 L'identité de Bachet-Bézout

On va construire deux entiers relatifs u et v tels que

$$au + bv = d = \text{pgcd}(a, b)$$

On considère la matrice

$$\begin{pmatrix} L_1 & 1 & 0 & a \\ L_2 & 0 & 1 & b \end{pmatrix}$$

qu'il faut interpréter ligne par ligne comme $a\times$1er coefficient+$b\times$ 2ème coefficient=3ième coefficient. L'algorithme de construction de u et v est une version arithmétique de l'algorithme du pivot de Gauss, où on souhaite créer un zéro dans la dernière colonne de la matrice. Mais seules les manipulations de lignes du type $L_{n+2} = L_n - qL_{n+1}$ avec q **entier** sont admises. Le mieux que l'on puisse faire est donc de créer $L_3 = L_1 - qL_2$ avec q quotient euclidien de a par b. Il faudra donc plusieurs manipulations de lignes avant d'avoir un 0, et la dernière colonne sera la liste des restes successsifs de l'algorithme d'Euclide pour calculer le PGCD de a et b. La ligne précédent la ligne avec un 0 en dernière colonne se termine par le PGCD (dernier reste non nul) et les deux premières colonnes donnent les coefficients de Bézout. Exemple avec $a = 125$ et $b = 45$

$$\begin{pmatrix} L_1 & 1 & 0 & 125 \\ L_2 & 0 & 1 & 45 \\ L_3 = L_1 - 2L_2 & 1 & -2 & 35 \\ L_4 = L_2 - L_3 & -1 & 3 & 10 \\ L_5 = L_3 - 4L_4 & 4 & -11 & 5 \\ L_6 = L_4 - 2L_5 & -9 & 25 & 0 \end{pmatrix}$$

Algorithme :

- Initialiser deux listes l_1 et l_2 à $[1, 0, a]$ et $[0, 1, b]$.
- Tant que $l_2[2] \neq 0$ (comme les indices commencent à 0, $l_2[2]$ est le dernier nombre de la liste l_2), faire
  - $q =$ quotient euclidien de $l_1[2]$ par $l_2[2]$,
  - l_1, l_2 prend la valeur $l_2, l_1 - ql_2$,
- Renvoyer l_1

Exemple : exécutez la commande ci-dessous qui calcule l'identité de Bézout pour 125 et 45 en affichant les étapes de l'algorithme dans la partie basse de la page HTML

```
@@step_infolevel:=1:;iegcd(125,45);step_infolevel:=0:;
```

"Done", $[4, -11, 5]$, "Done"

Explication : A chaque itération, l'élément `l[2]` d'indice 2 de la liste l_1 (ou l_2) prend comme valeur l'un des restes successifs de l'algorithme d'Euclide, et on a un invariant de boucle `a*l[0]+b*l[1]=l[2]` qui donne l'identité de Bézout lorsque `l[2]` est le dernier reste non nul. Cet invariant se prouve facilement par récurrence.

Regardons maintenant la taille de u et de v. On peut supposer que $a \geq b$ quitte à échanger a et b et on ignore les cas triviaux où a et/ou b valent 1.

Proposition 3.3 *Si $a \geq b \geq 2$, alors $|u| \leq b$ et $|v| \leq a$.*

Preuve (abrégée) : On considére les trois suites u_n (coefficients de la colonne d'indice 0), v_n (colonne d'indice 1) et r_n (colonne d'indice 2), ces suites vérifient la même relation de récurrence :

$$u_{n+2} = u_n - q_n u_{n+1}, \quad v_{n+2} = v_n - q_n v_{n+1}, \quad r_{n+2} = r_n - q_n r_{n+1}$$

où q_n est le quotient de la division de r_n par r_{n+1}. On montre par récurrence

$$v_n r_{n+1} - v_{n+1} r_n = (-1)^{n+1} a, \quad (-1)^{n+1} v_n \text{ croissante (stricte pour } n \geq 2)$$

au cran après Bézout, on a

$$|v_{N+1}|\text{pgcd}(a, b) = a$$

donc la suite $|v_n|$ est majorée par a et $|v| \leq a$ On en déduit $|u| \leq b$ puisque $u = (1 - bv)/a$.

On montre que le cout de l'algorithme est proportionnel à n^2, comme pour l'algorithme d'Euclide.

Cet algorithme fonctionne à l'identique avec les polynômes à coefficients dans un corps, en utilisant la division euclidienne des polynômes.

L'identité de Bézout a de nombreuses conséquences en arithmétique. En voici une première :

Proposition 3.4 *Tout diviseur commun à a et b est un diviseur du PGCD de a et b.*

En effet, si c divise a et b, il divise $au + bv$ le PGCD de a et b.

3.6 Nombres premiers entre eux

Deux entiers sont dits premiers entre eux si leur PGCD vaut 1. Par exemple 22 et 15 sont premiers entre eux. Plus généralement, si a et b ont comme PGCD d, alors a/d et b/d sont premiers entre eux, car leur PGCD multiplié par d divise a et b.

Théorème 3.5 *(Lemme de Gauss) :*
Si a et b sont premiers entre eux et si a divise bc alors a divise c.

Preuve : par hypothèse, il existe un entier q tel que $bc = qa$ et par Bézout, deux entiers u, v tels que $au + bv = 1$, on élimine b entre ces deux équations ce qui donne $qav = bcv = bvc = (1 - au)c$ er $c = a(qv + uc)$ est bien multiple de a.

Proposition 3.6 *Si a et b sont premiers entre eux et divisent tous deux c alors ab divise c.*

En effet, il existe des entiers q, q', u, v tels que

$$c = qa, \quad c = q'b, \quad au + bv = 1$$

Donc

$$c = c(au + bv) = acu + bvc = aq'bu + bvqa = ab(q'u + qv)$$

Ces résultats restent vrais avec des polynômes à coefficients dans un corps (où premiers entre eux signifie que le PGCD est constant).

Application : On en tire une méthode de résolution de l'équation

$$ax + by = c$$

d'inconnues x et y. Soit d le PGCD de a et b. Alors $ax + by$ est divisible par d donc l'équation n'a pas de solution si c n'est pas divisible par d. Si c est divisible par d, alors une solution particulière est $x = u\frac{c}{d}, y = v\frac{c}{d}$. Les autres solutions vérifient

$$ax + by = c = ax' + by' \;\Rightarrow\; a'(x - x') = b'(y' - y), \quad a' = \frac{a}{d}, b' = \frac{b}{d}$$

Comme a' et b' sont premiers entre eux, a' divise $y' - y$ donc

$$y' = y + qa', x' = x - qb', \quad q \in \mathbb{Z}$$

Application : décomposition d'une fraction $\frac{n}{ab}$ avec a et b premiers entre eux (entiers ou polynômes). Il existe alors u et v tels que $au + bv = 1$, donc U et V tels que $n = aU + bV$ et

$$\frac{n}{ab} = \frac{aU + bV}{ab} = \frac{aU}{ab} + \frac{bV}{ab} = \frac{U}{b} + \frac{V}{a}$$

Exemple 1 : $\frac{2}{3\times 5}$, on a $1 = 2 \times 3 - 1 \times 5$ donc $2 = 4 \times 3 - 2 \times 5$ et

$$\frac{2}{15} = \frac{4 \times 3 - 2 \times 5}{3 \times 5} = \frac{4}{5} - \frac{2}{3}$$

Exemple 2 : $\frac{2}{x^2-1} = \frac{2}{(x-1)(x+1)}$ Les polynômes $x - 1$ et $x + 1$ sont premiers entre eux, on a l'identité de Bézout

$$x - 1 - (x + 1) = -2$$

donc

$$\frac{2}{(x-1)(x+1)} = \frac{(x+1) - (x-1)}{(x-1)(x+1)} = \frac{1}{x-1} - \frac{1}{x+1}$$

Ce qui permet de calculer une primitive de $\frac{2}{x^2-1}$.

$$\int \frac{2}{x^2 - 1} = \ln(|x - 1|) - \ln(|x + 1|)$$

3.7 Les restes chinois

L'identité de Bézout a une application très importante en calcul formel, elle permet de reconstruire un entier dont on connait le reste de la division par des entiers premiers entre eux :

Théorème 3.7 *(restes chinois)*
Soit m et n deux entiers naturels premiers entre eux, alors pour tout a et b entiers relatifs, il existe un entier c tel que $c-a$ est divisible par m et $c-b$ est divisible par n.

Il existe une infinité de solutions, deux solutions c et c' distinctes diffèrent par un multiple de nm. Il y a donc existence et unicité dans n'importe quel intervalle contenant nm entiers.

Remarque : lorsqu'on reconstruit un entier naturel plus petit que nm, on choisit l'unique valeur de c telle que $c \in [0, nm[$ (c'est le reste de la division par nm de n'importe quelle solution), lorsqu'on reconstruit un entier relatif de valeur absolue plus petite que $nm/2$, on choisit c tel que $|c| \leq nm/2$ (on prend le reste symétrique).

Preuve : Si on a deux solutions c et c' alors $c-c'$ est divisible par n et m qui sont premiers entre eux, donc $c-c'$ est divisible par nm. Cherchons une solution, cela revient à chercher U et V tels que

$$c-a = mU, \quad c-b = nV$$

On a donc

$$a+mU = b+nV \quad \Leftrightarrow \quad -mU+nV = b-a$$

Or m et n sont premiers entre eux, donc par Bézout, il existe u et v tels que

$$mu+nv = 1$$

il suffit de multiplier par $b-a$ pour trouver un couple de valeurs U, V qui convient.

On sait que le couple U, V n'est pas unique, si U', V' est une autre solution, alors $b-a = -mU+nV = -mU'+nV'$ donc

$$n(V-V') = m(U-U')$$

donc n divise $U-U'$ et m divise $V-V'$ avec le même quotient. Pour déterminer efficacement une valeur de c qui convient, on calculera v tel que $|v| \leq m$ par l'algorithme de Bézout, on multipliera par $b-a$ et on prendra pour V le reste de $V(b-a)$ par m.

Exemple : trouver $c \in [0, 13 \times 7 = 91[$ tel que le reste de c par 13 est 11 et le reste de c par 7 est 3. On veut

$$c = 11+13U = 3+7V$$

donc

$$13U-7V = -8$$

On a l'identité de Bézout :

$$(-1) \times 13 + 7 \times 2 = 1$$

on multiplie par -8 donc on peut prendre $V = 16$ ($U = 8$), on préfère prendre le reste par 13 donc $V = 3$, $c = 24$ (et $U = 1$).

Remarque : en calcul formel, on prend $m = p_1...p_{k-1}$ un produit de nombres premiers distincts, et $n = p_k$ un nombre premier distinct des précédents, on calcule la valeur de c modulo chaque nombre premier p_j et dès qu'on a assez de nombres premiers p_k (i.e. si $|c| < p_1...p_k/2$) on en déduit c. On parle d'algorithme modulaire. Par exemple, on peut calculer le déterminant d'une matrice à coefficients entiers de cette manière.

```
@@a:=ranm(3,3);det(a);p1:=nextprime(200);p2:=nextprime
(p1);p3:=nextprime(p2);p1*p2*p3;d1:=det(a
 mod p1);d2:=det(a mod p2);d3:=det(a mod
 p3);ichinrem(d1,d2,d3);
```

$$\begin{pmatrix} 63 & -49 & -86 \\ -64 & -30 & 70 \\ 22 & 42 & 56 \end{pmatrix}, -367728, 211, 223, 227, 10681031, 45\%211, (-1)\%223, 12\%227, (-367728)\%10681031$$

Cette méthode s'applique aussi aux polynômes, mais elle est moins utile, car l'interpolation polynomiale de Lagrange est plus efficace lorsque les nombres premiers p_i sont remplacés par des $x - \alpha_i$

3.8 Nombres premiers, factorisation

Définition 3.8 *Un entier naturel est premier s'il est divisible seulement par deux entiers distincts : 1 et lui-même.*

Exemple : 17 est premier, mais pas 18. 1 n'est pas premier.

Un nombre non premier admet un diviseur différent de 1 et lui-même donc une factorisation $n = pq$ avec $p \neq 1$ et $q \neq 1$. On peut supposer que $p \leq q$, pour savoir si un nombre est premier on peut donc tester que tous les entiers plus petits ou égaux à $\sqrt{n}$ ne divisent pas n.

On a la

Proposition 3.9 *Si p premier ne divise pas un entier a, alors a et p sont premiers entre eux. En particulier si p et q sont deux nombres premiers distincts, alors ils sont premiers entre eux.*
Si p premier divise ab alors p divise a ou p divise b.

Si p ne divise pas a, alors la liste des diviseurs de a ne contient pas p, donc la liste des diviseurs communs à a et p est réduite à 1 donc a et p sont premiers entre eux.
Si p divise ab, soit p divise a, soit p est premier avec a donc par le lemme de Gauss il divise b.

On en déduit :

Proposition 3.10 *Si $q_1, .., q_k$ sont des nombres premiers tous distincts de p premier, alors p et $a = q_1...q_k$ sont premiers entre eux.*

Sinon, p divise $a = q_1...q_n$ donc p divise l'un des facteurs q_i donc est égal à q_i puisque q_i est premier.

Théorème 3.11 *Un entier naturel $n > 1$ s'écrit de manière unique (à permutation près) comme produit de nombres premiers.*

$$n = \prod_i p_i^{m_i}$$

L'exposant m_i est la multiplicité de p_i.

Preuve de l'existence par récurrence. Pour $n = 2$, on a $2 = 2$. Pour $n > 2$, si n est premier, alors $n = n$, sinon il admet un plus petit diviseur strictement supérieur à 1, on a $n = pq$ avec $p < n$ premier et $q < n$ à qui on applique l'hypothèse de récurrence.

Exercice : traduire cette preuve en un algorithme.

Unicité : supposons $p_1...p_k = p'_1...p'_l$. Donc p_j divise $p'_1...p'_l$. Comme p_j est premier, il doit être égal à un des p'_l. On divise par ce facteur commun et on recommence.

Remarques :

- La liste des diviseurs d'un entier se déduit de sa factorisation. En effet, un diviseur a de b a comme liste de facteurs premiers un sous-ensemble de la liste des facteurs premiers de b (en raison de la proposition 3.10) avec des multiplicités inférieures ou égales.
- Le PGCD de deux entiers se déduit de la factorisation de ces entiers. En raison de ce qui précède, on reprend les facteurs premiers communs de a et b avec comme multiplicité la plus petite des deux. Ceci a surtout un intérêt théorique, car factoriser deux entiers est beaucoup plus couteux que de leur appliquer l'algorithme d'Euclide.

Des résultats analogues s'appliquent pour les polynômes à coefficients dans un corps : on parle de facteur irréductible (analogue de nombre premier), i.e. polynome divisible seulement par des polynomes constants et des polynomes multiples non nuls d'eux-mêmes. Par exemple tous les polynômes de degré 1 sont irréductibles. Un polynôme s'écrit de manière unique à permutation près et à inversibles près comme produit de facteurs irréductibles.

3.9 Prolongement : idéaux

Un idéal est un sous-ensemble d'un anneau qui est stable pour l'addition (la somme de deux éléments de l'idéal est encore dans l'idéal), et stable par multiplication par tout élément de l'anneau. Par exemple si on veut résoudre un système de plusieurs équations en une inconnue, l'ensemble des polynômes en une variable qui s'annulent en un point est un idéal.

Dans les anneaux euclidiens, tous les idéaux sont principaux, i.e. engendrés par un élément, on prend un plus petit entier en valeur absolue pour les entiers ou un plus petit polynôme en degré pour les polynômes, la division euclidienne par cet élément doit renvoyer 0 comme reste sinon on aurait un élément dans l'idéal plus petit. Dans l'exemple ci-dessus, on peut se ramener à une équation polynomiale en une variable (c'est un PGCD des polynômes des équations).

Dans les anneaux non euclidiens, comme par exemple les polynômes à plusieurs variables, les idéaux ne sont pas forcément engendrés par un seul élément, ce qui rend la résolution de systèmes polynomiaux à plusieurs inconnues beaucoup plus délicate qu'en une inconnue. Il est en général difficile de vérifier qu'un élément de l'anneau appartient ou non à l'idéal (alors que c'est juste une division dans le cas à une variable). Heureusement, il existe des familles particulières qui engendrent l'idéal pour lesquelles la vérification se fait par une généralisation de la division euclidienne. La recherche efficace de telles familles, appelées bases de Groebner, est un domaine de recherche actif !

4 L'anneau $\mathbb{Z}/n\mathbb{Z}$ et le corps $\mathbb{Z}/p\mathbb{Z}$

4.1 Définition

4.1.1 Congruences

On dit que deux entiers a et b sont congrus modulo n si $a-b$ est divisible par n. On vérifie immédiatement les propriétés suivantes qui caractérisent une **relation d'équivalence** :

- (réflexivité) a est toujours congru à a modulo n
- (symétrie) si a est congru à b modulo n alors b est congru à a modulo n
- (transitivité) si a est congru à b modulo n et si b est congru à c modulo n, alors a est congru à c modulo n

L'ensemble des entiers congrus à a modulo n est appelé classe de a modulo n et est notée $\overline{a}$. Par exemple la classe de 3 modulo 11,

$$\overline{3} = \{..., -19, -8, 3, 14, 25, ...\}$$

Les propriétés ci-dessus montrent que $\overline{a} = \overline{b}$ si et seulement si a et b sont congrus modulo n (ou encore si et seulement si $b \in \overline{a}$). Un élément de de $\overline{a}$ est appelé représentant de la classe. Deux classes distinctes ont une intersection vide.

On peut définir les opérations de somme et de produit sur les classes, car ces opérations sont compatibles avec la divisibilié par n, ainsi

$$\overline{a} + \overline{b} = \overline{a+b}, \overline{a}\overline{b} = \overline{ab}$$

et le résultat ne dépend du choix du représentant dans la classe de a et de b. En effet si a' et b' sont deux autres représentants, on a $a' = a + \alpha n$ et $b' = b + \beta n$ donc

$$a' + b' = a + b + (\alpha + \beta)n, \quad a'b' = ab + (\alpha b + \beta a + \alpha\beta)n$$

Comme il y a n reste possibles lorsqu'on divise un entier par n, il y a n classes distinctes, l'ensemble de ces classes est appelé $\mathbb{Z}/n\mathbb{Z}$ et on vient de voir qu'il possède une structure d'anneau. Nous allons dans la suite de cette section nous intéresser aux propriétés de la multiplication dans cet anneau.

Optimisation : lorsqu'on travaille modulo un entier fixé, on peut optimiser les divisions euclidiennes par cet entier en utilisant un algorithme dû à Granlund et Montgomery. Le lecteur intéressé pourra rechercher des informations sur le site d'Agner Fog (volume 2, section 16.9).

4.1.2 Généralisation : relations d'équivalence, classes d'équivalence

On peut faire la même chose dans tout ensemble E possédant une relation d'équivalence, i.e. une relation binaire $\mathcal{R}$ vérifiant les propriétés de réflexivité, symétrie et transitivité, on obtient ainsi que E est réunion disjointe de classes d'équivalence $E/\mathcal{R}$. Si E est muni d'une loi de groupe + compatible avec la relation d'équivalence (i.e. $a\mathcal{R}a'$ et $b\mathcal{R}b'$ entraine que $(a+b)\mathcal{R}(a'+b')$), alors l'ensemble des classes d'équivalence $E/\mathcal{R}$ muni de la loi + est un groupe. De même on obtient un anneau si on a compatibilité avec les deux lois + et $\times$ d'un anneau E.

Par exemple, sur les polynômes à coefficients dans un corps K, étant donné un polynôme M on peut définir la congruence modulo M, i.e. $A\mathcal{R}B$ si le reste de la division euclidienne de $A-B$ par M est nul. On vérifie que cette relation est une relation d'équivalence, compatible avec l'addition et la multiplication des polynômes.

Autre exemple : si on a un anneau commutatif unitaire intègre, on peut construire son corps des fractions : il est constitué de couples (n,d), deux couples sont équivalents si $nd' = n'd$. Si on a un anneau euclidien, on peut alors construire un représentant privilégié qui est une fraction n/d telle que n et d soient premiers entre eux. En ajoutant des conditions pour tenir compte des inversibles de l'anneau, on peut construire un représentant unique. Ainsi pour le corps des fractions de $\mathbb{Z}$, n et d sont premiers entre eux et $d>0$. Pour celui des polynômes à coefficients rationnels, on peut montrer qu'on a un représentant sous forme de fraction de deux polynômes à coefficients entiers primitifs (le PGCD des coefficients vaut 1) de coefficient dominant strictement positif multiplié par une fraction irréductible d'entiers.

4.2 Le groupe des inversibles

L'ensemble des éléments inversibles pour la multiplication d'un anneau forme un groupe multiplicatif. En effet le produit de deux inversibles est encore inversible $(ab)^{-1} = b^{-1}a^{-1}$, le neutre de la multiplication est inversible (son inverse est lui-même), et tout élément admet un symétrique pour le produit puisqu'il est supposé inversible.
Exemple : les inversibles modulo 8 sont 1, 3, 5 et 7. Pour obtenir la table de multiplication du groupe, on peut valider la commande

```
@@matrix(4,4,apply((x,y)->x*y mod 8,set[1,3,5,7
]*set[1,3,5,7]))
```

$$\begin{pmatrix} 1\,\%\,8 & 3\,\%\,8 & (-3)\,\%\,8 & (-1)\,\%\,8 \\ 3\,\%\,8 & 1\,\%\,8 & (-1)\,\%\,8 & (-3)\,\%\,8 \\ (-3)\,\%\,8 & (-1)\,\%\,8 & 1\,\%\,8 & 3\,\%\,8 \\ (-1)\,\%\,8 & (-3)\,\%\,8 & 3\,\%\,8 & 1\,\%\,8 \end{pmatrix}$$

On vérifie bien que chaque ligne contient un 1, donc chaque élément est inversible.

Dans un groupe (avec une loi notée multiplicativement), on a le

Théorème 4.1 *(Lagrange)*
Soit G un groupe ayant un nombre fini g d'éléments, alors pour tout $x \in G$, $x^g = x \times ... \times x = 1_G$ l'élément neutre du groupe (dans le produit avec les ..., il y a g occurences de x et $g-1$ signes $\times$).
De plus le plus petit entier $k>0$ tel que $x^k = 1_G$, appelé **ordre** *de l'élement x, divise g.*

Exemple : pour les inversibles modulo 8, on a

$$1^1 \pmod 8 = 1, 3^2 = 1 \pmod 8, 5^2 = 1 \pmod 8, 7^2 = 1 \pmod 8$$

Preuve dans le cas où G est commutatif.
Soit $x \in G$ fixé. L'application de G dans G qui à $y \in G$ associe xy est une bijection (de réciproque $y \to x^{-1}y$). Donc le produit de tous les éléments de G est égal au produit de leurs images. Comme le groupe est supposé commutatif, c'est le produit des images multiplié par x^g, on en déduit que $x^g = 1_G$.
Montrons que k divise g : soit r le reste de la division de g par k, on a $g = kq + r$ donc $x^g = (x^k)^q x^r$ donc $x^r = 1_G$ donc $r = 0$ car $r < k$ et k est le plus petit entier non nul tel que $x^k = 1_G$.

Remarque : ce résultat est encore vrai si G n'est pas commutatif, avec une preuve différente (cf. par exemple wikipedia).

Revenons aux entiers et aux entiers modulo n. Soit donc $n > 1$ un entier. Un entier a est inversible modulo n (ou une classe de congruence $\overline{a}$ est inversible dans $\mathbb{Z}/n\mathbb{Z}$) si et seulement si il existe un entier u tel que $au - 1$ est divisible par n, donc si et seulement si il existe deux entiers u et v tels que $au - 1 = vn$, i.e. $au - vn = 1$. On retrouve l'identité de Bézout. Les inversibles de $\mathbb{Z}/n\mathbb{Z}$ sont les classes dont les représentants sont premiers avec n et le calcul de l'inverse d'un entier a modulo n est obtenu par le coefficient de a dans l'identité de Bézout pour a et n.

Définition 4.2 *On appelle* **indicatrice d'Euler** *le nombre d'entiers $m \in [0, n[$ qui sont premiers avec n, on le note $\varphi(n)$.*

C'est le nombre d'éléments inversibles de $\mathbb{Z}/n\mathbb{Z}$. Par exemple si $n = p$ est premier, $\varphi(p) = p - 1$. Autre exemple, si $n = 15$, les inversibles sont

$$\{1, 2, 4, 7, 8, 11, 13, 14\}$$

et $\varphi(15) = 8$.

Le théorème de Lagrange donne alors :

Corollaire 4.3

$$\textit{si } a \textit{ est premier avec } n, \quad a^{\varphi(n)} = 1 \pmod{n}$$

Exemple : $2^8 = 16^2 = (16-1) \times (16+1) + 1 = 1 \pmod{1}5$.

Remarque :On retrouve des résultats de la section motivation, si $n = p$ est premier et $a \in [1, p[$, alors $a^{p-1} = 1 \pmod{p}$.

Calcul de $\varphi(n)$ connaissant la factorisation de n

- Si $n = p$ est premier, $\varphi(p) = p - 1$
- si $n = p^k$ est une puissance d'un nombre premier p. Comptons les nombres non premiers avec n de $[0, n[$, comme p est premier, ils sont divisibles par p, ce sont donc des multiples de p inférieurs stricts à $n = p^k$, i.e. des nombres de la forme ap inférieurs stricts à p^k il y en a p^{k-1}, donc $\varphi(p^k) = p^k - p^{k-1}$
- Si $n = ab$ est le produit de deux entiers a et b premiers entre eux alors $\varphi(n) = \varphi(a)\varphi(b)$.
  En effet, soit $c \in [0, n = ab[$, alors c et ab sont premiers entre eux si et seulement si c et a sont premiers entre eux d'une part **et** si c et b sont premiers entre eux d'autre part (considérer un diviseur premier commun). Ce qui revient au reste de la division de c par a premier avec a **et** au reste de la division de c par b premier avec b. Il y a donc $\varphi(a)$ possibilités pour $c \pmod{a}$ et $\varphi(b)$ possibilités pour $c \pmod{b}$ donc par le théorème des restes chinois, il y a $\varphi(a)\varphi(b)$ possibilités pour $c \in [0, ab[$.

Exemple :

$$\varphi(15) = \varphi(3 \times 5) = \varphi(3)\varphi(5) = 2 \times 4 = 8$$

Si $n = pq$ est le produit de deux nombres premiers distincts, on a

$$\varphi(pq) = (p-1)(q-1)$$

Plus généralement, on a pour tout a premier avec $n = pq$ produit de deux nombres premiers distincts :

$$a^{(p-1)(q-1)} = 1 \pmod{n}$$

résultat déjà obtenu dans la section motivations.

Remarques :

- on a utilisé le théorème des restes chinois dans le calcul de $\varphi(ab)$ pour a et b premiers entre eux. On peut traduire le théorème des restes chinois en terme de $\mathbb{Z}/n\mathbb{Z}$ de la manière suivante :
  Si a et b sont premiers entre eux, alors il existe une bijection entre $\mathbb{Z}/ab\mathbb{Z}$ et $\mathbb{Z}/a\mathbb{Z} \times \mathbb{Z}/b\mathbb{Z}$. Elle consiste à prendre un représentant x de $\mathbb{Z}/ab\mathbb{Z}$ et lui associer le couple (classe de x dans $\mathbb{Z}/a\mathbb{Z}$, classe de x $\mathbb{Z}/b\mathbb{Z}$). Cela ne dépend pas du représentant x choisi. C'est une bijection à cause du théorème des restes chinois. De plus cette bijection est compatible avec les opérations d'addition et de multiplication dans les anneaux, on parle d'isomorphisme d'anneaux. Elle préserve les inversibles.

Exemple : $\mathbb{Z}/12\mathbb{Z}$ et $\mathbb{Z}/4\mathbb{Z} \times \mathbb{Z}/3\mathbb{Z}$ sont isomorphes, on peut construire la table de correspondance

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ (0,0) & (1,1) & (2,2) & (3,0) & (0,1) & (1,2) & (2,0) & (3,1) & (0,2) & (1,0) & (2,1) & (3,2) \end{pmatrix}$$

les inversibles de la ligne du dessus correspondent aux couples d'inversibles en-dessous, donc le premier élément du couple doit être 1 ou 3, le deuxième 1 ou 2, ce qui donne 4 possibilités $(1,1), (1,2), (3,1), (3,2)$ correspondant à $1, 5, 7, 11$.

- connaitre la factorisation de $n = pq$ ou la valeur de $\varphi(n)$ est équivalent. En effet si on connait n et

$$\varphi(n) = pq - (p+q) + 1, \quad n = pq$$

on en déduit la somme $S = p + q$ et le produit $P = pq$ puis p et q en résolvant l'équation du second degré $x^2 - Sx + P = 0$.

- si on connait la factorisation de n, on peut calculer $\varphi(n)$ par récurrence :

$$n = \prod_{i=1}^{k} p_k^{r_k} \quad \Rightarrow \quad \varphi(n) = \prod_{i=1}^{k} p_k^{r_k - 1}(p_k - 1) = n \prod_{i=1}^{k} (1 - \frac{1}{p_k})$$

4.3 L'algorithme d'exponentiation rapide

Il s'agit de calculer efficacement $a^k \pmod n$. On a vu dans la section motivation un algorithme récursif qui permet de le faire efficacement. On commence par remplacer a par $a \pmod n$ puis

- si $k = 0$ on renvoie 1
- si k est pair, on calcule $b = a^{k/2} \pmod n$, on renvoie $b^2 \pmod n$
- si k est impair, on calcule $b = a^{(k-1)/2} \pmod n$, on renvoie $ab^2 \pmod n$.

Il y a au plus ceil$(2\log_2(k))$ multiplications, qui font intervenir des entiers plus petits que n donc le coût est en $O(\log_2(k)\log_2(n)^2)$ (avec multiplication naive pour les entiers).

Il existe une version itérative de l'exponentiation rapide, utilisant la décomposition en base 2 de l'exposant que l'on calcule simultanément.

On initialise $A \leftarrow a$, $b \leftarrow 1$, puis tant que $k \neq 0$

- si k est impair, $b \leftarrow A \times b \pmod n$,
- $k \leftarrow$ quotient euclidien de k par 2,
- $A \leftarrow A \times A \pmod n$

et on renvoie b. En effet après i itérations, A vaut $a^{2^i} \pmod n$ et

$$a^k = a^{\sum_i k_i 2^i} = \prod_{i/k_i=1} a^{2^i}$$

4.4 Le corps $\mathbb{Z}/p\mathbb{Z}$

Lorsque $n = p$ est premier, tous les éléments de l'anneau $\mathbb{Z}/p\mathbb{Z}$ sauf $\overline{0}$ sont inversibles, on est alors en présence d'un corps ayant p éléments, et dont le groupe multiplicatif des inversibles possède $p-1$ éléments.

Une propriété très importante des corps est que le nombre de racines d'un polynôme de degré d est au plus d. Cela résulte du fait qu'il n'y a pas de diviseurs de 0 dans un corps, donc si r est racine de P, alors on peut écrire $P = (X - r)Q$ où Q est de degré $d - 1$, les autres racines de P sont exactement les racines de Q. Cette propriété n'est pas vraie lorsqu'on considère un polynôme à coefficients dans un anneau qui n'est pas un corps, par exemple $x^2 = 4 \pmod 1 5$ admet 4 solutions 2, 7, 8, 13.

On rappelle que par le théorème de Lagrange, $x^{p-1} = 1 \pmod p$ pour x inversible modulo p. On a donc

$$X^{p-1} - 1 = \prod_{i=1}^{p-1} (X - i) \pmod p$$

car ces deux polynômes ont les mêmes $p - 1$ racines distinctes et même coefficient dominant.

On peut alors se demander si l'ensemble des puissances de x est égal à $\mathbb{Z}/p\mathbb{Z}$. On peut se poser la même question sur $\mathbb{C}$ pour les racines complexes de $z^{p-1} = 1$ qui sont de la forme

$$e^{\frac{2ik\pi}{p-1}}, \quad k \in [0, p-2]$$

la réponse sur $\mathbb{C}$ est que c'est le cas si k est premier avec $p-1$, par exemple $k = 1$. Ici, on a le

Théorème 4.4 *Il existe un générateur des inversibles de $\mathbb{Z}/p\mathbb{Z}$, i.e. un élément x tel que*

$$\mathbb{Z}/p\mathbb{Z} = \{0, 1, x, x^2, ..., x^{p-2}\}$$

x est appelé racine primitive $p-1$-ième de 1 (modulo p), parce que $x^{p-1} = 1$, mais $x^k \neq 1$ pour $k < p-1$. Les autres générateurs des inversibles sont obtenus en prenant x^k pour k premier avec $p-1$, il y en a donc $\varphi(p-1)$.

On donne la preuve de ce théorème pour être complet, étant entendu qu'elle se situe au-delà du niveau de ce cours. On factorise $p-1 = \prod_i p_i^{m_i}$ en produit de facteurs premiers, les p_i étant deux à deux distincts. Montrons que pour tout i, il existe un élément x_i de $\mathbb{Z}/p\mathbb{Z}^*$ d'ordre $p_i^{m_i}$. En effet, le polynôme

$$P(X) = X^{p_i^{m_i-1} \prod_{j\neq i} p_j^{m_j}} - 1$$

n'est pas identiquement nul sur $\mathbb{Z}/p\mathbb{Z}^*$ car son degré est strictement inférieur à $p-1$. Soit $a \neq 0$ non racine de P, on prend $x_i = a^{\prod_{j\neq i} p_j^{m_j}}$, on a $x_i^{p_i^{m_i-1}} = P(a) + 1 \neq 1$ et $x_i^{p_i^{m_i}} = a^{p-1} = 1$. On pose alors $x = \prod_i x_i$. On vérifie que x est bien d'ordre $p-1$ avec le critère ci-dessous.

Critère : Pour savoir si x est d'ordre $p-1$, comme l'ordre d'un élément divise $p-1$, si on connait les facteurs premiers p_i de $p-1 = \prod_i p_i^{m_i}$, il suffit de vérifier que

$$x^{\frac{p-1}{p_i}} \neq 1 \quad \text{pour } p_i \text{ facteur premier de } p-1$$

Pour trouver un générateur de $\mathbb{Z}/p\mathbb{Z}^*$, on teste alors si $x = 2, 3, ...$ convient en utilisant l'algorithme d'exponentiation rapide. La probabilité de trouver un générateur en tirant un entier au hasard dans $[1, p[$ vaut :

$$\frac{\varphi(p-1)}{p-1} = \frac{\prod_i (p_i^{m_i} - p_i^{m_i-1})}{\prod_i p_i^{m_i}} = \prod_i (1 - \frac{1}{p_i})$$

Remarque : la recherche de racines primitives est un ingrédient essentiel pour faire fonctionner la transformée de Fourier rapide exacte, plus précisément on cherche des racines primitives 2^k-ième de 1 modulo p avec p premier de la forme $p = 1 + 2^k m$.

4.5 L'équation $ax = b \pmod p$

Si p est premier, l'équation est du premier degré dans un corps, donc

– si $a \neq 0 \pmod p$, elle admet une solution unique modulo p :

$$x = (a \pmod p)^{-1} b$$

– si $a = 0 \pmod p$, deux cas : si $b = 0$, tout x est solution, si $b \neq 0$ il n'y a pas de solutions.

Prolongement : $ax = b \pmod n$

On peut faire le même raisonnement que ci-dessus lorsque a est inversible modulo n, on a alors une solution unique. Sinon, on écrit l'équation sous la forme

$$AX + nY = B$$

où A est un représentant de la classe de a et n, B de b. Si le pgcd de A et n (qui est indépendant du représentant choisi) vaut $d > 1$, il faut que B soit divisible par d pour qu'il y ait des solutions (ceci ne dépend pas des représentants choisis dans la classe déquivalence), que l'on sait alors calculer en partant de l'identité de Bézout pour A/d et n/d. Il y a alors d solutions modulo n, en effet pour $k \in [0, d-1]$ on a :

$$a(x + k\frac{n}{d}) = b \pmod n$$

Si on connait la factorisation de n, on peut un peu gagner en efficacité sur les calculs intermédiaires.

– Si $n = p^k$ est une puissance d'un nombre premier ($k \geq 2$) et si $a \neq 0 \pmod p$. Notons c l'inverse de a modulo p dans l'intervalle $[0, p[$. Soit $x_0 = cb$, on a $ax_0 = acb = b \pmod p$. Si x est une solution de $ax = b \pmod p^k$, alors $ax = b \pmod p$ donc $x = x_0 \pmod p$. On pose alors $x = x_0 + px_1$ et on cherche x_1. On a $ax - b = ax_0 - b + pax_1 \pmod{p^k}$ donc

$$pax_1 = b - ax_0 \pmod{p^2} \quad \Rightarrow \quad ax_1 = \frac{b - ax_0}{p} \pmod p \quad \Rightarrow \quad x_1 = c\frac{b - ax_0}{p} \pmod p$$

(car $b - ax_0$ est bien un multiple de p). On continue ainsi de proche en proche pour déterminer lécriture de x en base p. C'est ce qu'on appelle une méthode p-adique. Son intérêt est de faire des calculs intermédiaires avec des entiers plus petits que p au lieu de p^k. On fait ainsi k étapes avec un cout en $\log(p)^2$, i.e. $k\log(p)^2$, il faut ajouter le cout de la décomposition en base p de a et b qui est en $k^2\log(p)^2$, au lieu d'une inversion modulaire avec un cout en $\log(p^k)^2 = k^2\log(p)^2$ mais avec une constante de proportionnalité plus grande.
En pratique cette méthode p-adique est surtout utilisée pour résoudre des grands systèmes linéaires à coefficients entiers.
– Si n n'est pas premier, on peut se ramèner au cas précédent par le théorème des restes chinois (méthode modulaire).

4.6 Prolongement : l'équation $x^2 = a \pmod p$

Si $a = 0$, il y a une solution unique $x = 0$. Si $a \neq 0$, si l'équation a une solution, alors $a^{(p-1)/2} = x^{p-1} = 1 \pmod p$. Réciproquement, on si $a^{(p-1)/2} = 1 \pmod p$ alors on montre qu'il y a bien 2 solutions opposées, en comptant les carrés non nuls (il y en a $(p-1)/2$ qui est le degré de $X^{(p-1)/2} - 1$, ils forment donc l'ensemble des solutions de $X^{(p-1)/2} = 1 \pmod p$).

– Si $p = 3 \pmod 4$, ces solutions s'obtiennent par l'algorithme d'exponentiation rapide $x = b = \pm a^{(p+1)/4}$.
– Si $p = 1 \pmod 4$, il existe un algorithme probabiliste qui fonctionne au moins une fois sur deux. Soit b tel que $b^2 = a \pmod p$. L'idée est de tiret un entier r au hasard entre 0 et p et de calculer le PGCD unitaire des polynômes $(X+r)^{(p-1)/2} - 1 \pmod p$ et $X^2 - a \pmod p$, on s'arrête lorsque le PGCD est de degré 1, ce sera alors $X - b$ ou $X + b$.
En effet le PGCD est de degré 2 si $X - b$ et $X + b$ sont facteurs de $(X+r)^{(p-1)/2} - 1$ donc si b et $-b$ sont racines de $(X+r)^{(p-1)/2} - 1$ donc si $(b+r)^{(p-1)/2} = (-b+r)^{(p-1)/2} \pmod p = 1$. Le PGCD est de degré 0, si b et $-b$ sont racines de $(X+r)^{(p-1)/2} + 1$ donc si $(b+r)^{(p-1)/2} = (-b+r)^{(p-1)/2} \pmod p = -1$. Dans les deux cas, $(b+r)^{(p-1)/2} = (-b+r)^{(p-1)/2} \pmod p$. Cette équation en r est de degré $(p-1)/2 - 1$ donc possède (au plus) $(p-1)/2 - 1$ solutions. Pour r non solution, donc pour $(p-1)/2 + 1$ valeurs de r (au moins), le PGCD sera de degré 1.

Exemple 1 : $p = 7, a = 2$. On a $a^{(p-1)/2} = 2^3 = 8 = 1 \pmod 7$ donc 2 est un carré modulo 7, et les racines carrées sont $b = \pm a^{(p+1)/4} = \pm 2^2 = \pm 4 \pmod 7 = \mp 3 \pmod 7$
Exemple 2 : $p = 5, a = 2$. On a $a^{(p-1)/2} = 2^2 = 4 = -1 \pmod 5$ donc 2 n'est pas un carré modulo 5.
Exemple 3 : $p = 5, a = 4$. On a $a^{(p-1)/2} = 4^2 = 16 = 1 \pmod 5$ donc 4 est un carré modulo 5. Comme 4 est un carré parfait, les deux racines carrées sont 2 et -2. Voici les PGCD de $(X+r)^2 - 1$ et $X^2 - 4$ modulo 5 :

```
@@p:=5; seq([r,gcd((X+r)^((p-1)/2)-1 mod
 p,X^2-4 mod p)],r,0,p);
```

$$5, \begin{bmatrix} 0 & 1 \,\%\, 5 \\ 1 & (1 \,\%\, 5)\, X + 2 \,\%\, 5 \\ 2 & (1 \,\%\, 5)\, X + (-2) \,\%\, 5 \\ 3 & (1 \,\%\, 5)\, X + 2 \,\%\, 5 \\ 4 & (1 \,\%\, 5)\, X + (-2) \,\%\, 5 \\ 5 & 1 \,\%\, 5 \end{bmatrix}$$

Vous pouvez essayer d'autres valeurs de p premier pour vérifier que tirer au hasard r donne un PGCD de degré 1 environ une fois sur 2.

On peut alors résoudre les équations du second degré si $p \neq 2$ avec la formule habituelle du discriminant.

Remarque
Comme pour l'équation du premier degré, on peut résoudre $x^2 = a \pmod n$ lorsque n n'est pas premier. On commence par le cas d'une puissance de nombre premier, où on utilise un algorithme p-adique comme dans la section précédente. Le cas général il se traite avec le théorème des restes chinois. Ainsi pour $n = pq$ produit de deux premiers distincts, on a 4 racines carrées en cas d'existence. Par exemple, 4 admet 2, 7, 8, 13 comme racines carrées modulo 15.

4.7 Prolongement : $\mathbb{F}_p[X] = \mathbb{Z}/p\mathbb{Z}[X]$

L'anneau des polynômes $\mathbb{F}_p[X]$ à coefficients dans $\mathbb{Z}/p\mathbb{Z}[X]$ est un anneau euclidien, on peut en effet effectuer des divisions euclidiennes comme sur $\mathbb{Q}[X]$. On peut donc définir le PGCD de 2 polynômes non simultanément nuls de $\mathbb{F}_p[X]$ (normalisé par coefficient dominant 1), et les calculs sont beaucoup plus simples que sur $\mathbb{Q}$, et c'est l'ingrédient principal des algorithmes efficaces de calcul de PGCD sur $\mathbb{Q}[X]$.

4.8 Test de primalité

Test de Fermat : Si p est premier et a premier avec p, alors $a^{p-1} = 1 \pmod p$. Donc si on trouve un entier a premier avec n tel que $a^{n-1} \neq 1 \pmod n$ alors n n'est pas premier.
Exemple : $a = 2$ et $n = 1345677$

```
@@n:=1345677; pow(2,n-1,n)
```

$$1345677, 220021$$

Il est important de voir qu'on a alors prouvé que n n'est pas premier *sans* déterminer un facteur non trivial de n. Le temps de calcul d'un test de Fermat est proportionnel à $\ln(n)^3$, alors que la factorisation naïve de n peut couter $\sqrt{n}\ln(n)^2$.

Attention, si $a^{n-1} = 1 \pmod n$ pour deux entiers a et n premiers entre eux, cela ne prouve pas que n est premier ! Il existe d'ailleurs des entiers non premiers pour lesquels cette propriété est vraie pour tout a premier avec n, ce sont les nombres de Carmichael, le plus petit est 561.

Exercice : Implémenter un programme déterminant tous les nombres de Carmichael plus petits que n.

Prolongement : test de Miller-Rabin : c'est une version améliorée du test précédent. Il part de l'observation suivante : soit a premier avec p nombre premier impair, si $a^{p-1} = 1 \pmod p$ alors $a^{(p-1)/2} = \pm 1 \pmod p$ (1 a exactement deux racines carrées dans $\mathbb{Z}/p\mathbb{Z}$). Si $a^{(p-1)/2} = 1 \pmod p$ et $(p-1)/2$ est pair, on peut recommencer le même raisonnement. Plus précisément, soit on trouve un -1, soit on ne peut plus diviser par 2 l'exposant. En raisonnant dans l'autre sens, si $p - 1 = 2^t s$ avec s impair, alors $a^s = \pm 1 \pmod p$ ou l'un des carrés successifs doit valoir -1 modulo p (si on tombe sur un 1 le test échoue et le nombre n'est pas premier)

```
def millerrabin(a,n):
  t=0
  s=n-1
  while s
     t += 1
     s = s // 2
  a=pow(a,s,n)
  if a==1 or a==n-1: return True
  for j in range(t):
    a=(a*a)
    if a==1: return False
    if a==n-1: return True
  return False
```

```
@@millerrabin(2,1345677);millerrabin(2,1345691
);millerrabin(3,1345691)
```

False, False, False

Comme pour le test de Fermat, si le test renvoie faux, on est sur que le nombre n'est pas premier (a est appelé témoin de Miller), s'il renvoie vrai on ne peut rien prouver. Mais on peut montrer que si n est non premier, le nombre d'entiers $a < n$ qui renvoient `True` est au plus $n/4$, ce qu'on peut plus ou moins traduire par si on tire k entiers a "au hasard" et que le test de Miller-Rabin renvoie vrai à chaque fois, il y a moins d'une chance sur 4^k que n ne soit pas premier. Par exemple pour $k = 20$ et en prenant les 20 premiers nombres premiers, $1/4^{20}$ est de l'ordre de `1e-12`, la probabilité de considérer premier un nombre qui ne l'est pas est très faible. On parle de nombre pseudo-premier.

Remarques :

- On peut montrer qu'un entier 64 bits est premier si et seulement si il passe le test de Miller-Rabin pour les 12 premiers nombres premiers (de 2 à 37).
- Si l'hypothèse de Riemann généralisée (voir par exemple wikipedia) est vraie, et n est composé, on peut montrer qu'il existe un témoin de Miller plus petit que $2\ln(n)^2$, et le test de Miller-Rabin pour tous les entiers plus petit que $2\ln(n)^2$ devient un test déterministe de primalité (en $O(\ln(n)^5)$ opérations avec multiplication naïve).

4.9 Application : cryptographie RSA

4.9.1 Interprétation du codage et du décodage.

Nous revenons sur la cryptographie RSA, en utilisant les notions vue dans les sections précédentes. On rappelle qu'on se donne un entier $n = pq$ produit de deux nombres premiers p et q. Les entiers p et q sont secrets alors que n est public. On choisit ensuite deux entiers d et e tels que $de = 1 + k(p-1)(q-1)$, ou encore d et e sont inverses modulo l'indicatrice d'Euler de n :

$$de = 1 \pmod{\varphi(n)}$$

Le codage et le décodage se font avec l'algorithme de la puissance modulaire rapide :

$$a \to b = a^e \pmod{n} \to c = b^d \pmod{n}$$

le cout est de $O(\ln(e)\ln(n)^2)$ et $O(\ln(d)\ln(n)^2)$ opérations.

On peut maintenant facilement montrer que ces deux fonctions sont réciproques l'une de l'autre lorsque a est premier avec n. En effet, on a $a^{\varphi(n)} = 1 \pmod{n}$ donc :

$$c = a^{de} \pmod{n} = a^{1+k\varphi(n)} \pmod{n} = a(a^{\varphi(n)})^k \pmod{n} = a \pmod{n}$$

4.9.2 Comment générer des clefs

On choisit p et q en utilisant le test de Miller-Rabin. Par exemple

```
@@p:=nextprime(randint(10^150));q:=nextprime
(randint(10^200));n:=p*q;
```

45363489186110371903439757062676833871921966457658440406969023473911123951034170365970715033263202480209

On choisit un couple de clefs privée-publique en utilisant l'identité de Bézout (ou inverse modulaire). Par exemple

```
@@E:=65537; gcd(E,(p-1)*(q-1));d:=iegcd(E,
(p-1)*(q-1))[0];
```

$65537, 1, -1402893705019690012574667458457607703154591032160295012036479431233504278707979799487174851317755$

Ici, on a pris `E` de sorte que l'exponentiation modulaire rapide à la puissance `E` nécessite peu d'opérations arithmétiques (17), comparé au calcul de la puissance `d`, ceci permet de faire l'opération "publique" plus rapidement (encodage ou vérification d'une signature) si le microprocesseur a peu de ressources (par exemple puce d'une carte bancaire).

```
@@a:=randint(123456789);b:=powmod(a,E,n);c:=powmod
(b,d,n);
```

$18731026, 139824656627401897600052668600630684353324140104368201748264521030533805682009024559367106488211$

On peut utiliser un programme tel que `ssh-keygen` pour générer des clefs de taille industrielle. Par exemple la commande suivante

```
ssh-keygen -t rsa -b 2048 -m PEM -f rsa2048.key
```

génère une clef 2048 bits stockée dans le fichier `rsa2048.key`, on peut afficher en clair les valeurs de n, p, q avec la commande

```
openssl rsa -in rsa2048.key -text -inform PEM -noout
```

On obtient une sortie analogue à ce qui suit

```
Private-Key: (2048 bit)
modulus:
    00:a1:e8:58:f4:81:34:c7:0c:e6:4e:12:a3:22:f3:
    2b:49:c4:7a:e0:c0:b6:11:07:3f:ea:f3:23:eb:d2:
    3d:97:99:35:65:ce:42:d6:9a:e9:53:3e:60:a2:90:
    28:7a:bf:23:1f:84:b0:10:4a:15:8c:dd:e1:29:43:
    53:cb:74:22:4f:f6:ae:ec:62:35:1e:61:0b:66:2e:
    fe:21:41:c1:56:6d:68:70:51:8b:42:6a:db:ca:13:
    e1:06:1d:db:da:33:82:96:70:0c:df:d6:4f:51:c2:
    51:57:ae:ec:a8:6d:06:f3:a3:02:b7:5d:79:3e:58:
    1b:e2:a2:e6:d8:97:58:c9:7f:f6:81:8d:ab:1e:9b:
    22:0a:4f:77:41:49:e6:5c:71:03:ee:70:9a:60:22:
    28:76:c1:53:a7:99:06:2a:22:9c:0b:9d:68:cc:2f:
    36:5a:84:bd:b1:03:36:cc:be:07:4b:17:8c:85:84:
    ef:d5:1e:31:e0:82:32:0e:52:85:d0:8e:40:0d:a8:
    ac:c7:ff:c1:c0:e9:fe:6c:49:a5:c9:ff:cc:15:f1:
    2b:1c:c6:70:29:97:77:da:3e:7c:f7:3f:bf:6f:17:
    91:ee:b6:5b:90:d8:ba:49:07:96:1b:a7:a9:5e:37:
    8f:4c:be:de:c4:07:ef:12:e6:78:a0:4d:95:42:13:
    2e:3b
publicExponent: 65537 (0x10001)
privateExponent:
    69:0e:97:f2:07:88:d4:84:15:48:a1:a5:43:7f:60:
    1e:5c:a4:93:03:d8:df:d1:c1:72:d5:d4:00:28:0a:
    99:3c:eb:be:24:89:90:31:32:a7:36:39:84:22:60:
    71:cd:66:a0:03:fc:2e:85:b3:d8:14:fd:0e:46:46:
    b0:24:aa:43:12:c1:4c:57:29:3a:8e:23:d4:69:37:
    b3:22:b4:ae:3d:0d:e0:9b:b8:ee:1e:e2:81:0c:47:
    1e:2d:ef:c3:75:5b:0d:fc:a5:0d:f5:44:c0:bb:83:
    06:8f:55:b6:b0:10:2b:b5:21:85:13:dd:21:4c:10:
    c4:0d:8a:17:0e:95:a9:21:1b:91:17:86:07:de:74:
    d5:3f:05:21:e7:57:67:3b:32:de:68:cf:a5:9b:61:
    6f:f7:89:c3:db:a4:75:53:71:b8:77:ab:b5:22:e7:
    dc:23:43:30:26:3b:d0:0f:6a:46:d0:b6:51:88:49:
    4b:c8:25:3e:b6:eb:32:cf:56:63:db:7a:79:43:35:
```

```
    0f:f7:e5:9e:fa:60:66:c7:78:f3:e8:a9:4e:c2:ed:
    3d:04:a9:c3:7d:ae:23:86:25:e9:8d:a9:5c:33:4a:
    db:16:31:85:88:b6:c1:5b:bd:02:4b:41:73:eb:c1:
    9c:d1:0b:ee:d8:74:4d:a4:ca:ab:0a:97:4d:a0:26:
    59
prime1:
    00:d7:99:be:f5:71:ee:79:b1:a0:28:77:f3:eb:10:
    ab:83:0a:f4:00:b9:76:ca:4c:d1:1a:00:97:37:de:
    32:3f:b1:5c:fb:8d:e5:ec:9b:ec:c6:10:06:2d:2e:
    13:38:97:4a:f4:f8:5d:1a:3b:9b:e2:cb:74:4d:b7:
    31:f5:22:f2:f4:c1:25:78:2d:df:ee:ca:00:0f:65:
    16:5b:fe:ea:e3:49:02:61:05:52:cd:d9:66:42:34:
    b9:d0:09:23:62:ff:c5:51:2f:b7:fb:bf:62:ed:02:
    a5:7c:85:c6:e4:db:9f:6f:4a:44:ca:41:01:40:c9:
    8b:a1:53:0f:f5:b2:4c:a0:a5
prime2:
    00:c0:3e:f9:f1:c9:a9:ed:23:7c:ae:f3:41:97:0f:
    4a:83:b6:df:e0:1f:e1:b7:4c:b8:d6:3f:79:0c:41:
    29:e9:5b:f8:95:48:8d:7d:10:8c:8b:a0:b7:09:91:
    e2:aa:35:31:60:b9:d3:88:0f:d1:33:61:0c:b3:55:
    4f:dc:ed:18:1d:d9:7a:7e:1a:e4:61:11:2c:4a:2d:
    03:3d:02:b3:e2:48:ca:e9:08:e0:bd:da:90:a4:00:
    f5:cc:ed:9b:90:5e:c0:17:e1:4e:aa:4c:94:5d:14:
    79:7a:b7:c0:39:9f:36:45:df:38:48:7d:32:48:5e:
    17:d7:ba:ea:d5:96:f7:7d:5f
```

Exercice : vérifiez que le produit de `prime1` par `prime2` donne bien `modulus` et que l'indicatrice d'Euler est correcte, ainsi que la clef secrète.

Ci-dssous un programme C de cryptage/décryptage utilisant ces clefs. Il utilise la librairie GMP pour les opérations sur les entiers de grande taille et se compile par la commande

```
gcc rsa.c -lgmp
```

Bien entendu, un vrai programme de cryptage/décryptage ne doit pas contenir de clé(s) privée(s) dans son code source, car le code objet les contiendra, il doit les lire dans un fichier.

```
#include <gmp.h>
#include <stdio.h>

const unsigned char rsa_n_tab[]={
  0xa1,0xe8,0x58,0xf4,0x81,0x34,0xc7,0x0c,0xe6,0x4e,0x12,0xa3,0x22,0xf3,
 0x2b,0x49,0xc4,0x7a,0xe0,0xc0,0xb6,0x11,0x07,0x3f,0xea,0xf3,0x23,0xeb,0xd2,
 0x3d,0x97,0x99,0x35,0x65,0xce,0x42,0xd6,0x9a,0xe9,0x53,0x3e,0x60,0xa2,0x90,
 0x28,0x7a,0xbf,0x23,0x1f,0x84,0xb0,0x10,0x4a,0x15,0x8c,0xdd,0xe1,0x29,0x43,
 0x53,0xcb,0x74,0x22,0x4f,0xf6,0xae,0xec,0x62,0x35,0x1e,0x61,0x0b,0x66,0x2e,
 0xfe,0x21,0x41,0xc1,0x56,0x6d,0x68,0x70,0x51,0x8b,0x42,0x6a,0xdb,0xca,0x13,
 0xe1,0x06,0x1d,0xdb,0xda,0x33,0x82,0x96,0x70,0x0c,0xdf,0xd6,0x4f,0x51,0xc2,
 0x51,0x57,0xae,0xec,0xa8,0x6d,0x06,0xf3,0xa3,0x02,0xb7,0x5d,0x79,0x3e,0x58,
 0x1b,0xe2,0xa2,0xe6,0xd8,0x97,0x58,0xc9,0x7f,0xf6,0x81,0x8d,0xab,0x1e,0x9b,
 0x22,0x0a,0x4f,0x77,0x41,0x49,0xe6,0x5c,0x71,0x03,0xee,0x70,0x9a,0x60,0x22,
 0x28,0x76,0xc1,0x53,0xa7,0x99,0x06,0x2a,0x22,0x9c,0x0b,0x9d,0x68,0xcc,0x2f,
 0x36,0x5a,0x84,0xbd,0xb1,0x03,0x36,0xcc,0xbe,0x07,0x4b,0x17,0x8c,0x85,0x84,
 0xef,0xd5,0x1e,0x31,0xe0,0x82,0x32,0x0e,0x52,0x85,0xd0,0x8e,0x40,0x0d,0xa8,
 0xac,0xc7,0xff,0xc1,0xc0,0xe9,0xfe,0x6c,0x49,0xa5,0xc9,0xff,0xcc,0x15,0xf1,
 0x2b,0x1c,0xc6,0x70,0x29,0x97,0x77,0xda,0x3e,0x7c,0xf7,0x3f,0xbf,0x6f,0x17,
 0x91,0xee,0xb6,0x5b,0x90,0xd8,0xba,0x49,0x07,0x96,0x1b,0xa7,0xa9,0x5e,0x37,
 0x8f,0x4c,0xbe,0xde,0xc4,0x07,0xef,0x12,0xe6,0x78,0xa0,0x4d,0x95,0x42,0x13,
```

```
 0x2e,0x3b
};

int public_key=65537;

const unsigned char pri_n_tab[]={
  0x69,0x0e,0x97,0xf2,0x07,0x88,0xd4,0x84,0x15,0x48,0xa1,0xa5,0x43,0x7f,0x60,
  0x1e,0x5c,0xa4,0x93,0x03,0xd8,0xdf,0xd1,0xc1,0x72,0xd5,0xd4,0x00,0x28,0x0a,
  0x99,0x3c,0xeb,0xbe,0x24,0x89,0x90,0x31,0x32,0xa7,0x36,0x39,0x84,0x22,0x60,
  0x71,0xcd,0x66,0xa0,0x03,0xfc,0x2e,0x85,0xb3,0xd8,0x14,0xfd,0x0e,0x46,0x46,
  0xb0,0x24,0xaa,0x43,0x12,0xc1,0x4c,0x57,0x29,0x3a,0x8e,0x23,0xd4,0x69,0x37,
  0xb3,0x22,0xb4,0xae,0x3d,0x0d,0xe0,0x9b,0xb8,0xee,0x1e,0xe2,0x81,0x0c,0x47,
  0x1e,0x2d,0xef,0xc3,0x75,0x5b,0x0d,0xfc,0xa5,0x0d,0xf5,0x44,0xc0,0xbb,0x83,
  0x06,0x8f,0x55,0xb6,0xb0,0x10,0x2b,0xb5,0x21,0x85,0x13,0xdd,0x21,0x4c,0x10,
  0xc4,0x0d,0x8a,0x17,0x0e,0x95,0xa9,0x21,0x1b,0x91,0x17,0x86,0x07,0xde,0x74,
  0xd5,0x3f,0x05,0x21,0xe7,0x57,0x67,0x3b,0x32,0xde,0x68,0xcf,0xa5,0x9b,0x61,
  0x6f,0xf7,0x89,0xc3,0xdb,0xa4,0x75,0x53,0x71,0xb8,0x77,0xab,0xb5,0x22,0xe7,
  0xdc,0x23,0x43,0x30,0x26,0x3b,0xd0,0x0f,0x6a,0x46,0xd0,0xb6,0x51,0x88,0x49,
  0x4b,0xc8,0x25,0x3e,0xb6,0xeb,0x32,0xcf,0x56,0x63,0xdb,0x7a,0x79,0x43,0x35,
  0x0f,0xf7,0xe5,0x9e,0xfa,0x60,0x66,0xc7,0x78,0xf3,0xe8,0xa9,0x4e,0xc2,0xed,
  0x3d,0x04,0xa9,0xc3,0x7d,0xae,0x23,0x86,0x25,0xe9,0x8d,0xa9,0x5c,0x33,0x4a,
  0xdb,0x16,0x31,0x85,0x88,0xb6,0xc1,0x5b,0xbd,0x02,0x4b,0x41,0x73,0xeb,0xc1,
  0x9c,0xd1,0x0b,0xee,0xd8,0x74,0x4d,0xa4,0xca,0xab,0x0a,0x97,0x4d,0xa0,0x26,
  0x59
};

void tabunsignedchar2mpz(const unsigned char tab[],int len,mpz_t & res){
  mpz_set_si(res,0);
  for (int i=0;i<len;++i){
    mpz_mul_si(res,res,256); // res=256*res;
    mpz_add_ui(res,res,tab[i]); // res+=tab[i];
  }
}

// ./rsa crypt|decrypt filein fileout
int main(int argc,const char ** argv){
  if (argc<4){
    printf("Usage:
    return 1;
  }
  mpz_t z,p,n;
  mpz_init(z); mpz_init(p); mpz_init(n);
  tabunsignedchar2mpz(rsa_n_tab,sizeof(rsa_n_tab),n);
  if (argv[1][0]=='c'){
    printf("crypting\n");
    mpz_set_si(p,public_key);
  }
  else {
    tabunsignedchar2mpz(pri_n_tab,sizeof(pri_n_tab),p);
    printf("decrypting\n");
    mpz_out_str(stdout,10,n);
    printf("\n");
    mpz_out_str(stdout,10,p);
    printf("\n");
```

```
  }
  FILE * f=fopen(argv[2],"r");
  int res=mpz_inp_str(z,f,0);
  fclose(f);
  mpz_powm(z,z,p,n);
  f=fopen(argv[3],"w");
  mpz_out_str(f,10,z);
  fclose(f);
  mpz_clear(z); mpz_clear(p); mpz_clear(n);
}
```

4.9.3 Sur quoi repose la sécurité de RSA.

- Difficulté de factoriser n.
  Si on arrive à factoriser n, tous les autres calculs se font en temps polynomial en $\ln(n)$ donc la clef est compromise. Il faut bien choisir p et q pour que certains algorithmes de factorisation ne puissent pas s'appliquer. Par exemple choisir

  ```
  @@p:=nextprime(10^100):;q:=nextprime(10^101
  ):;n:=p*q;
  ```

  "Done", "Done", 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

  n'est pas une bonne idée car n a beaucoup trop de 0 dans son écriture décimale, on peut donc facilement le factoriser, en essayant les diviseurs de 801 ajoutés à 10^{100}.
- Difficulté de trouver d connaissant n et e, si on ne connait pas la factorisation de n.
  On peut tenter de trouver d sans chercher à factoriser n. Notons que ce n'est pas avec l'identité de Bézout qu'on procèdera, car connaitre l'indicatrice d'Euler de $n = pq$ permet de trouver p et q immédiatement. Une attaque de ce type pour p et q presque egaux et d petit devant n est décrite par exemple ici.
- Difficulté de trouver a connaissant $b = a^e \pmod n$. Il n'y a pas d'algorithme connu permettant de le faire dans tous les cas. Mais il faut quand même faire attention au choix de e.
  Par exemple prendre $e = 3$ pour minimiser le nombre d'étapes de la puissance modulaire n'est pas judicieux, car si un message commun est envoyé à 3 personnes ayant des clefs publiques $(n_1, 3)$, $(n_2, 3)$, $(n_3, 3)$, la connaissance de

  $$a^3 \pmod{n_1}, \quad a^3 \pmod{n_2}, \quad a^3 \pmod{n_3}$$

  permet de retrouver $a^3 \pmod{n_1 n_2 n_3}$ par le lemme chinois si n_1, n_2, n_3 sont premiers entre eux 2 à deux, puis a^3 puis a.
  Il faut aussi que l'ordre o de e dans les inversibles de $(\mathbb{Z}/\varphi(n)\mathbb{Z}, \times)$ ne soit pas trop petit, sinon un attaquant peut itérer $o - 1$ la fonction de cryptage sur b et obtenir a.

Une référence sur l'utilisation de RSA pour la sécurisation des transactions par carte bancaire.

4.10 Pour aller plus loin

Les polynômes de $\mathbb{Z}/p\mathbb{Z}[X]$ modulo un polynôme irréductible de degré n forment un corps fini de cardinal une puissance d'un nombre premier p^n. On montre que les corps de cardinal fini sont tous de cette forme.

La recherche de solutions $a^2 = b^2 \pmod n$ est un objectif de pas mal d'algorithmes de factorisation, par exemple cfrac (fractions continues), le crible quadratique, et même l'algorithme de Shor (pour ordinateur quantique).

5 Algèbre linéaire

5.1 Systèmes linéaires sur $\mathbb{Z}/p\mathbb{Z}$, pivot de Gauss

Les systèmes linéaires sur $\mathbb{Z}/p\mathbb{Z}$ se résolvent comme les systèmes linéaires à coefficients réels ou complexes, par combinaison linéaire d'équations. On peut encoder par une matrice, en créant une ligne de matrice par équation et en y stockant les coefficients des inconnues. Les combinaisons linéaires d'équations correspondent alors à des combinaisons linéaires de lignes de la matrice. L'algorithme du pivot de Gauss consiste à triangulariser la matrice de manière équivalente (i.e. avec des opérations réversibles).

Algorithme du pivot de Gauss (sur un corps)
Soit M une matrice ayant L lignes et C colonnes. On numérote à partir de 0.

1. initialiser les indices du pivot ligne l et c à 0.
2. Tant que $l < L$ et $c < C$ faire :
   (a) Parcourir la colonne c à partir de la ligne l et chercher un coefficient non nul.
   Si tous les coefficients sont nuls, incrémenter c de 1 et passer à l'itération suivante de la boucle Tant que.
   (b) Si nécessaire, échanger la ligne ayant un coefficient non nul avec la ligne l
   (c) Maintenant $M_{l,c} \neq 0$. Si $M_{l,c} \neq 1$ on effectue :
   $$L_l \leftarrow \frac{1}{M_{l,c}} L_l$$
   Cette opération est réversible car on multiplie la ligne par un coefficient non nul.
   (d) Pour j de $l+1$ à $L-1$ effectuer la manipulation de ligne sur la matrice M (dont la ligne l a été modifiée) :
   $$L_j \leftarrow L_j - M_{j,c} L_l,$$
   Cette opération crée un 0 à la ligne j en colonne c sans détruire les 0 existants en ligne j pour des indices de colonne strictement inférieurs à c.
   Cette opération est réversible d'inverse $L_j \leftarrow L_j + M_{j,c} L_l$ (avec le coefficient de la matrice *avant* l'opération)
   (e) Incrémenter l et c de 1.

Toutes les opérations sont réversibles, donc le système obtenu est équivalent au système initial. Mais il est plus simple, car triangulaire supérieur, il y a des 0 en-dessous d'une "diagonale généralisée" (normalement c'est une diagonale, mais elle peut se décaler vers la droite s'il y a des colonnes où on n'a pas trouvé de 0). On résoud alors le système de bas en haut.

On peut aussi faire l'étape 2d pour j de 0 jusque $L-1$ et $j \neq L$ et obtenir une matrice échelonnée, avec des 0 de part et d'autre de la diagonale, plus précisément des 0 dans toutes les colonnes correspondant à des coefficients non nuls sur la diagonale généralisée.

```
def piv(M):
    L,C=dim(M)
    l,c=0,0
    while l<L and c<C:
        for j in range(l,L):
            if M[j,c]!=0:
                break
        if j==L:
            c += 1
            continue
        if j!=l:
            M[j],M[l]=M[l],M[j]
        if M[l,c]!=1:
```

```
            M[l]=inv(M[l,c])*M[l]
        for j in range(l+1,L):
            M[j] -= M[j,c]*M[l]
        c += 1
        l += 1
    return M
```

```
@@M:=[[2,2,3,4],[1,-2,1,2],[-2,2,3,5]] mod
 11; piv(M)
```

$$\begin{bmatrix} 2\,\%\,11 & 2\,\%\,11 & 3\,\%\,11 & 4\,\%\,11 \\ 1\,\%\,11 & (-2)\,\%\,11 & 1\,\%\,11 & 2\,\%\,11 \\ (-2)\,\%\,11 & 2\,\%\,11 & 3\,\%\,11 & 5\,\%\,11 \end{bmatrix}, \begin{bmatrix} 1\,\%\,11 & 1\,\%\,11 & (-4)\,\%\,11 & 2\,\%\,11 \\ 0\,\%\,11 & 1\,\%\,11 & 2\,\%\,11 & 0\,\%\,11 \\ 0\,\%\,11 & 0\,\%\,11 & 1\,\%\,11 & 1\,\%\,11 \end{bmatrix}$$

Donc :

$$\begin{cases} 2x & +2y & +3z & = 4[11] \\ x & -2y & +z & = 2[11] \\ -2x & +2y & +3z & = 5[11] \end{cases} \Leftrightarrow \begin{cases} x & +y & -4z & = 2[11] \\ & y & +2z & = 0[11] \\ & & z & = 1[11] \end{cases}$$

sa solution s'obtient en résolvant l'équation correspondant à la dernière ligne de la matrice réduite, donc $z = 1[11]$, puis l'avant-dernière ligne donne $y = -2z = -2[11]$ puis la première équation donne $x = -y + 4z + 2 = 8[11] = -3[11]$.
On vérifie

```
@@linsolve(M*[x,y,z,-1],[x,y,z])
```

$$[(-3)\,\%\,11, (-2)\,\%\,11, 1\,\%\,11]$$

Prolongement :
L'algorithme du pivot de Gauss peut s'effectuer directement sur la matrice A d'un système sans son second membre, c'est la factorisation $PA = LU$ d'une matrice, qui permet de se ramener à résoudre deux systèmes triangulaires.

Optimisations

- Si $p = 2$, on peut représenter les lignes comme des suites de bits, donc travailler 64 colonnes simultanément avec une représentation par des entiers non signés sur 64 bits. Une combinaison linéaire non triviale de 2 lignes est alors une addition qui se traduit par un ou exclusif sur les entiers 64 bits de la représentation.
- si on veut faire la réduction complète d'une grande matrice sous forme échelonnée, il est plus rapide de commencer par faire une réduction sous-diagonale complète, puis une réduction au-dessus de la diagonale.
- Pour $p > 2$, l'opération qui est la plus couteuse dans une combinaison linéaire de lignes est le calcul du reste euclidien par p. Il peut être intéressant d'effectuer plusieurs combinaisons linéaires de lignes avec une ligne donée et faire un seul calcul de reste, en prenant garde à éviter les dépassements.
  Ainsi si $p^2 n < 2^{64}$ avec n le nombre d'opérations, on peut utiliser des entiers 64 bits pour toutes les opérations intermédiaires, sans réduction modulo p avant la fin. Si on a seulement $p < 2^{32}$, on peut représenter par des entiers non signés 64 bits, effectuer une opération et enlever p^2 si le résultat dépasse p^2.
  En particulier la réduction au-dessus de la diagonale d'une matrice réduite sous la diagonale se prête à ce genre d'optimisation.
- Il est possible d'utiliser des algorithmes diviser pour régner en faisant des opérations par blocs et en se ramenant à des multiplications de matrice plus rapides que la multiplication naïve.

5.2 Algèbre linéaire sur $\mathbb{Z}/p\mathbb{Z}$

La théorie des espaces vectoriels sur $\mathbb{R}, \mathbb{Q}, \mathbb{C}$ se généralise à un corps fini. Les définitions et propriétés d'espace vectoriel, de dimension finie, de famille libre, génératrice, de bases, d'applications linéaires, de matrice d'applications linéaires sont identiques.

Définition 5.1 *Un* **espace vectoriel** V *sur un corps* K *est un ensemble muni d'une loi d'addition de deux vecteurs (*V *est un groupe commutatif pour +), et d'une multiplication d'un vecteur par un scalaire de* K*, qui est distributive (*$a.(u+v) = a.u + a.v$ *et* $(a+b).u = a.u + b.u$*) et vérifie* $a.(b.v) = (a.b)v$ *et* $1.v = v$*.*

On montre qu'un sous-ensemble W *de* V *stable par addition (si* $w_1, w_2 \in W$ *alors* $w_1 + w_2 \in W$*) et par multiplication par une constante de* K *(si* $w \in W$ *alors* $a.w \in W$*) est un espace vectoriel, on dit que c'est un* **sous-espace vectoriel** *de* V*.*

Dans la suite, on n'écrira plus explicitement le signe de multiplication entre un scalaire et un vecteur.

Exemples d'espaces vectoriels : $(\mathbb{Z}/p\mathbb{Z})^d$ (qui est en bijection avec les entiers naturels plus petits que p^d, mais attention l'addition des entiers ne correspond pas à l'addition dans l'espace vectoriel), les polynômes de degré inférieur à d à coefficients dans $\mathbb{Z}/p\mathbb{Z}$, les matrices à coefficients dans $\mathbb{Z}/p\mathbb{Z}$

Définition 5.2 *Soit* V *un espace vectoriel sur un corps fini* $K = \mathbb{Z}/p\mathbb{Z}$ *et* $F = \{v_1, ..., v_k\}$ *une partie de* k *éléments de* V*.*
Une combinaison linéaire de F *est un vecteur de la forme* $a_1v_1 + ... + a_kv_k$ *avec les* $a_i \in K$*. On vérifie que l'ensemble des combinaisons linéaires de* F *est un sous-espace vectoriel de* V *noté Vect*(F)*.*
On dit que F *est* **libre** *si aucun des vecteurs de* F *n'est combinaison linéaire des autres ce qui revient à dire que la relation* $a_1v_1 + ... + a_kv_k = 0$ *entraine* $a_1 = ... = a_k = 0$*.*
Une famille F *est* **génératrice** *si tout vecteur de* V *peut s'écrire comme combinaison linéaire des éléments de* F*.*
Un espace vectoriel est de dimension finie s'il admet une famille génératrice (ayant un nombre fini d'éléments).

D'une famille génératrice finie F (non réduite à des vecteurs nuls), on peut extraire une famille génératrice et libre. En effet si F est libre, c'est terminé. Sinon, l'un des vecteurs de F est combinaison linéaire des autres, on peut l'enlever de F en conservant l'espace vectoriel engendré. Comme le nombre d'éléments de F est fini, on s'arrête.

Par exemple $\{1, x, 1+x^2, x^2\}$ est une famille génératrice de Vect($\{1, x, 1+x^2, x^2\}$) et on peut éliminer $1 + x^2$ de la famille pour la rendre libre (on peut aussi éliminer 1 ou éliminer x^2).

Définition 5.3 *Un espace vectoriel de dimension finie possède une famille génératrice et libre, on l'appelle* **base** *(dans le cas particulier de* $V = \{0\}$*, la dimension de* V *est nulle et on peut définir la base de* V *comme étant l'ensemble vide). Toutes les bases ont le même nombre d'éléments* d*, appelé* **dimension** *de* V*.* V *possède* p^d *éléments.*

En effet, tout élément de V peut sécrire de manière unique comme combinaison linéaire des éléments de la famille. Les d composantes sur une base fixée peuvent prendre p valeurs chacune, il y a donc p^d éléments dans V. C'est une grande différence avec $\mathbb{R}$ (qui simplifie la preuve que toutes les bases ont le même nombre d'éléments).

Proposition 5.4 *Une famille libre d'un espace vectoriel de dimension* d *possède au plus* d *éléments, si elle en possède* d *c'est une base. Une famille génératrice possède au moins* d *éléments, si elle en possède* d *c'est une base.*

Sinon, on aurait pour les combinaisons linéaires d'une famille libre trop d'éléments, et pour les combinaisons linéaires d'une famille génératrice pas assez.

Définition 5.5 *On appelle* **base canonique** *de* $(\mathbb{Z}/p\mathbb{Z})^d$ *la base* $\{(1, 0..0), (0, 1, 0..), ...(0..., 0, 1)\}$*. On appelle base canonique des polynômes de degré inférieur strict à* d *la base* $\{1, x, ..., x^{d-1}\}$*. Plus généralement, on parle de base canonique pour une base intrinsèque (ou naturelle) d'un espace vectoriel.*

Définition 5.6 *Soient* V *et* W *deux espaces vectoriels. Une* **application linéaire** $\varphi : V \to W$ *est une application qui respecte la linéarité, i.e. telle que* $\varphi(u + v) = \varphi(u) + \varphi(v)$ *et* $\varphi(av) = a\varphi(v)$*.*

Exemples : $(x, y) \to (x + y, x - y)$ la multiplication par x des polynômes de degré inférieur à 2 (P_2) dans les polynômes de degré inférieur à 3 (P_3), ou la dérivée de P_3 dans P_2.

Si V et W sont de dimension finie, V de base $\{v_1, .., v_d\}$ et W de base $\{w_1, .., w_\delta\}$ on peut caractériser φ par les coordonneées des $\varphi(v_j)$ dans la base $\{w_1, ..., w_\delta\}$. On appelle matrice de φ dans ces bases le tableau des coordonnées des $\varphi(v_j)$ en colonnes. On a alors

$$M = (m_{ij})_{1 \leq i \leq \delta, 1 \leq j \leq d}, \quad \varphi(v_j) = \sum_{i=1}^{\delta} m_{i,j} w_j$$

On vérifie que si v a pour coordonnées le vecteur colonne V, alors $\varphi(v)$ a pour coordonnées le vecteur colonne $W = MV$

Exercice : calculer les matrices des exemples ci-dessus.

Soit $v \in V$, les coordonnées de $\varphi(v)$ dans la base $\{w_1, .., w_\delta\}$ s'obtiennent alors en faisant le produit de la matrice A de φ par le vecteur colonne des coordonnées de v dans la base $\{v_1, .., v_d\}$

La somme de 2 matrices correspond alors à la somme de deux applications linéaires ayant ces matrices, la multiplication d'une matrice par un élément de $\mathbb{Z}/p\mathbb{Z}$ correspond à la multiplication de l'application linéaire correspondante par cet élément. Le produit de deux matrices correspond à la composition de deux applications linéaires (c'est d'ailleurs une manière de prouver que le produit de deux matrices est associatif).

Définition 5.7 *On appelle transposée de la matrice $A = (a_{ij})_{1 \leq i \leq n, 1 \leq j \leq m}$ ayant n lignes et m colonnes la matrice $A^* = (a_{ji})_{1 \leq j \leq m, 1 \leq i \leq n}$ ayant m lignes et n colonnes.*

Définition 5.8 **Noyau** *et* **image** *d'une application linéaire $\varphi : V \to W$.*

L'ensemble des vecteurs $v \in V$ tels que $\varphi(v) = 0$ est un sous-espace vectoriel de V appelé noyau de φ et noté Ker(φ). Par abus, on note aussi Ker(A) le noyau de l'application linéaire de matrice A dans les bases canoniques de V et W. Une application linéaire est injective si et seulement si son noyau est réduit au vecteur nul.

L'ensemble des images $\varphi(v), v \in V$ est un sous-espace vectoriel de W appelé image de φ et noté Im(φ). De même, on parle de Im(A) par abus de notation. La dimension de l'image est appelé range de φ (ou de la matrice A).

Théorème 5.9 *(Théorème du rang)*

dim(V)=dim(Ker(φ)) + dim(Im(φ))

Preuve : on utilise le procédé de création d'une base à partir d'une famille génératrice de Im(φ) obtenue en prenant les images des vecteurs d'une base de V, i.e. on transpose la matrice A de l'application linéaire et on applique Gauss. Les combinaisons linéaires sur la base de V de l'algorithme de Gauss forment toujours une base de V. Les lignes nulles de A^* correspondent à des vecteurs qui forment une base de Ker(φ), les lignes non nulles forment une base de Im(φ).

Définition 5.10 *Une application linéaire de V (espace vectoriel de dimension d) dans lui-même $\varphi : V \to V$ est appelé* **endomorphisme***. Si un endomorphisme est injectif alors il est inversible. La matrice A de φ et la matrice B de son application réciproque vérifient alors $AB = BA = I_d$*

L'algorithme du pivot de Gauss permet de traiter tous les aspects calculatoires :
- liberté d'une famille, on résoud le système par le pivot de Gauss,
- extraction d'une famille génératrice, on écrit les coordonnées de la famille dans une matrice en ligne, et on garde les lignes non nulles après le pivot.
- rang d'une matrice, il est conservé par le pivot de Gauss, c'est le nombre de lignes non nulles à la fin
- détermination d'une base de l'image d'une application linéaire de matrice M, il suffit de mettre en ligne les vecteurs colonnes de la matrice M (puisqu'ils forment une famille génératrice), et de faire le pivot de Gauss (puisque les opérations de ligne sont alors des combinaisons linéaires de vecteurs),
- détermination du noyau d'une matrice, on résoud le système correspondant.
  Remarque : il existe une version plus algorithmique de calcul du noyau.

– inverse d'une matrice de taille n. On résout simultanément n systèmes linéaires avec comme second membre les n vecteurs de la base canonique.
– déterminant d'une matrice. Il faut refléter l'effet des opérations sur les lignes sur le déterminant : un échange de ligne change le signe du déterminant, la multiplication de l'étape 2c multiplie le déterminant

5.3 Application : cryptographie de Hill

Au lieu de coder un message caractère par caractère, ce qui est sujet à des attaques par fréquence, on décide de grouper les caractères par d-uplets et on les considère comme des vecteurs v de $(\mathbb{Z}/p\mathbb{Z})^d$ (en fait, il n'est pas indispensable de travailler sur un corps, on peut aussi travailler sur un anneau $\mathbb{Z}/n\mathbb{Z}$). On choisit une matrice carrée A de taille d inversible sur ce corps qui sera la clef secrète de chiffrement. Le message crypté correspondant au message en clair v est alors le message $w = Av$. On retrouve v connaissant le message crypté w et la clef de chiffrement en calculant $v = A^{-1}w$.

Pour assurer la sécurité du cryptage, il faut bien sur prendre p et d assez grands pour que l'ensemble des matrices possibles soit suffisamment grands (il y a p^{d^2} matrices possibles, et elles ne sont pas toutes inversibles, la première colonne ne doit pas être nulle, la deuxième ne doit pas être dans l'espace engendré par la première, etc. il y a donc $(p^d-1)(p^d-p)(p^d-p^2)...(p^d-p^{d-1})$ matrices inversibles). On peut aussi décider d'éviter les attaques par fréquence en prenant une partie de d-uplet dans le message à coder, et une autre partie avec des caractères aléatoires (le message crypté sera plus long mais le cryptage sera plus résistant).

5.4 Application aux codes

Cette section ne sera probablement plus au programme à partir de 2021/22 en raison de la diminution de l'horaire de l'UE
On travaillera dans cette section avec $p = 2$, i.e. sur le corps $K = \mathbb{Z}/2\mathbb{Z}$, mais les définitions et énoncés restent valables dans $\mathbb{Z}/p\mathbb{Z}$ ou dans un corps fini plus général.

On appellera symbole d'information l'unité de base transmise, qu'on supposera appartenir à un corps fini K, par exemple pour un bit un élément de $K = \mathbb{Z}/2\mathbb{Z}$ (pour un octet, il faudrait prendre un élément du corps à 256 éléments $K = F_{256} = F_d$).

On veut coder un message de longueur k avec des possibilités de détection et de correction d'erreurs, pour cela on rajoute des symboles calculés à partir des précédents, on envoie un élément d'un code ayant n symboles.

5.4.1 Code de répétition

Par exemple, on peut décider de répéter 3 fois chaque symbole $n = 3k$. Le récepteur du message peut détecter des erreurs de transmission (sauf si la même altération se produit 3 fois de suite, ce qui est très improbable). Il peut corriger des erreurs par exemple s'il reçoit 2 fois le même symbole a et un 3ième symbole distinct b, il garde le symbole a. Mais ce code simple est assez couteux à transmettre, on envoie 3 fois plus de données qu'il n'y a d'informations. On va présenter des méthodes plus efficaces.

5.4.2 Le bit de parité.

On prend $k = 7$ bits et $n = 8$ bits. On compte le nombre de 1 parmi les 7 bits envoyés, si ce nombre est pair, on envoie 0 comme 8ième bit, sinon 1. Au final le nombre de bits à 1 de l'octet (1 octet=8 bits) est pair. On peut ainsi détecter une erreur de transmission si à la réception le nombre de bits d'un octet est impair, mais on ne peut pas corriger d'erreurs.

5.4.3 Codes linéaires

Définition 5.11 *On multiplie le vecteur $v \in K^k$ des k symboles à gauche par une matrice M à coefficients dans K de taille $n \times k$ et on transmet l'image $Mv \in K^n$.*

L'ensemble des mots de code est l'image de M, c'est un sous-espace vectoriel de K^n. La matrice de M d'un code linéaire s'obtient en calculant les images des vecteurs de la base canonique.

Exemple : le code de répétition est un code linéaire de paramètres $k=1, n=3$, de matrice la matrice colonne $\begin{pmatrix}1\\1\\1\end{pmatrix}$. Le bit de parité est un code linéaire de paramètres $k=7, n=8$, de matrice

```
@@concat(idn(7),[seq(1,7)])
```

$$\begin{pmatrix}1&0&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&1&0&0&0&0\\0&0&0&1&0&0&0\\0&0&0&0&1&0&0\\0&0&0&0&0&1&0\\0&0&0&0&0&0&1\\1&1&1&1&1&1&1\end{pmatrix}$$

Remarque : On utilise ici la convention de l'algèbre linéaire (avec des vecteurs colonnes et multiplication par une matrice à gauche). Certains auteurs utilisent l'autre convention, i.e. des vecteurs lignes et multiplication par la matrice à droite, tout est transposé. Ceci vient peut-être du fait qu'il est plus facile de saisir un vecteur ligne qu'un vecteur colonne avec certains logiciels. Xcas utilise la convention que si on multiplie une matrice par un vecteur ligne, alors le vecteur ligne est automatiquement transformé en vecteur colonne (il n'y a en effet aucune confusion possible).

Pour assurer qu'on peut identifier un antécédent unique à partir d'une image, il faut que M corresponde à une application linéaire injective, ce qui entraine $n \geq k$. On dit qu'un vecteur de n symboles est un mot du code s'il est dans l'image de l'application linéaire.

Pour assurer l'injectivité tout en facilitant le décodage, on utilise très souvent une matrice identité k,k comme sous-bloc de la matrice n,k, par exemple on prend l'identité pour les k premières lignes de M, on ajoute ensuite $n-k$ lignes. On parle de code systématique.

Pour savoir si un vecteur est un mot de code, il faut vérifier qu'il est dans l'image de M. On peut par exemple vérifier qu'en ajoutant la colonne de ses coordonnées à M, on ne change pas le rang de M (qui doit être k), mais c'est assez couteux. On préfère déterminer une matrice de controle H telle que :

$$x \in \mathrm{Im}(M) \Leftrightarrow Hx = 0$$

Proposition 5.12 *Pour un code systématique de matrice*

$$M = \begin{pmatrix}I_k\\C\end{pmatrix}$$

(composée de l'identité I_k et d'une matrice C sur ses $n-k$ dernières lignes), alors la matrice de contrôle est

$$H = (-C, I_{n-k})$$

En effet :

$$Mv = x = \begin{pmatrix}v\\Cv\end{pmatrix}$$

donc :

$$Hx = (-C, I_{n-k})\begin{pmatrix}v\\Cv\end{pmatrix} = 0$$

Exemple, pour le bit de parité, $C = (1,1,1,1,1,1,1)$ et $H = (1,1,1,1,1,1,1,1)$.

5.4.4 Codes polynomiaux

Il s'agit d'un cas particulier de codes linéaires.

Définition 5.13 *On se donne un polynôme $g(x)$ de degré $n-k$, On représente le message de longueur k à coder par un polynôme P de degré $k-1$. On envoie alors Pg.*

Le récepteur peut controler que le reste de la division euclidienne du polynôme reçu par g est nul, et extraire l'information qui est le quotient de la division euclidienne par g. Ce code n'est pas systématique, mais on peut facilement l'adapter pour le rendre systématique.

Définition 5.14 *On multiplie P par x^{n-k}, on calcule le reste R de la division de Px^{n-k} par g. On émet alors $Px^{n-k}-R$ qui est divisible par g.*

Les mots de code sont les polynômes divisibles par g.

Exemple : le bit de parité correspond à $k=7, n-8, g=x+1$. Le polynôme $g=x^7+x^3+1$ ($k=120, n=127$) était utilisé par le Minitel.

5.4.5 Erreurs

Si le mot recu n'est pas dans l'image de l'application linéaire il y a eu erreur de transmission. Sinon, il n'y a pas eu d'erreur *détectable* (il pourrait y avoir eu plusieurs erreurs qui se "compensent").

Plutôt que de demander la réémission du mot mal transmis (ce qui serait par exemple impossible en temps réel depuis un robot en orbite autour de Mars), on essaie d'ajouter suffisamment d'information pour pouvoir corriger des erreurs en supposant que leur nombre est majoré par N. Si les erreurs de transmissions sont indépendantes, la probabilité d'avoir au moins $N+1$ erreurs dans un message de longueur L est $\sum_{k=N+1}^{L}\binom{L}{k}\epsilon^k(1-\epsilon)^{L-k}$, où ϵ est la probabilité d'une erreur de transmission, c'est aussi `1-binomial_cdf(L,epsilon,N)`. Par exemple, pour un message de 10^3 caractères, chacun ayant une probabilité d'erreur de transmission de 10^{-3}, si on prend $N=3$, alors la probabilité d'avoir au moins 4 erreurs est de 0.019 (arrondi par excès) :

```
@@P(N,eps,L):=sum(comb(L,k)*eps^k*(1-eps
)^(L-k),k,N+1,L):;P(3,1e-3,10^3)
```

"Done", 0.0189268334503

ou directement

```
@@1-binomial_cdf(1000,1e-3,3)
```

0.0189268334504

5.4.6 Distance

Définition 5.15 *La* **distance de Hamming** *de 2 mots est le nombre de symboles qui diffèrent. (il s'agit bien d'une distance au sens mathématique, elle vérifie l'inégalité triangulaire).*

Exercice : écrire une procédure de calcul de la distance de Hamming de 2 mots. En Xcas, la fonction s'appelle `hamdist`.

La distance d'un code est la distance de Hamming minimale de 2 mots différents du code. Pour un code linéaire, la distance est aussi le nombre minimal de coefficients non nuls d'un vecteur non nul de l'image. Pour un code polynomial, la distance du code est le nombre minimal de coefficients non nuls d'un multiple de g de degré inférieur à n.

Majoration de la distance du code :

Proposition 5.16 *(borne de Singleton)*
La distance minimale d'un code linéaire est inférieure ou égale à $m+1$ où $m = n-k$ est le nombre de symboles ajoutés.

Preuve : On écrit en ligne les coordonnées des images de la base canonique (ce qui revient à transposer la matrice) et on réduit par le pivot de Gauss, comme l'application linéaire est injective, le rang de la matrice est k, donc la réduction de Gauss crée $k-1$ zéros dans chaque ligne, donc le nombre de coefficients non nuls de ces k lignes (qui sont toujours des mots de code) est au plus de $n-k+1$.

Remarque : la borne n'est pas toujours atteinte, par exemple un code de paramètres $n=7, k=4$ a une distance minimale d'au plus 3 (et non 4).

5.4.7 Correction au mot le plus proche

Une stratégie de correction basée sur la distance consiste à trouver le mot de code le plus proche d'un mot donné. Si la distance d'un code est supérieure ou égale à $2t+1$, et s'il existe un mot de code de distance inférieure à t au mot donné, alors ce mot de code est unique. On corrige alors le mot transmis en le remplaçant par le mot de code le plus proche. Il n'est pas toujours possible de corriger un mot de code, sauf si la propriété suivante est vérifiée :

Définition 5.17 *On dit qu'un code t-correcteur est* **parfait** *si la réunion des boules de centre un mot de code et de rayon t (pour la distance de Hamming) est disjointe et recouvre l'ensemble des mots (K^n).*

Pour $t=1$ et $K=\mathbb{Z}/2\mathbb{Z}$, la boule de rayon 1 de K^n a pour cardinal $1+n$ (on est soit au centre, soit une des n composantes est différente de celle du centre), on doit donc avoir :

$$(1+n)2^k = 2^n \Leftrightarrow n = 2^m - 1, \quad m = n-k$$

On a aussi $n-k+1 \geq 2t+1 = 3$ donc $m \geq 2$. Les codes possibles sont donc :
- $m=2, n=3, k=1$ c'est le code de répétition
- $m=3, n=7, k=4$: voir le code donné dans la section motivations 1.2.
- $m=4, n=15, k=11$
- ...
- $m=7, n=127, k=120$: on peut montrer que le code polynomial de polynôme x^7+x^3+1 est un code de Hamming binaire
- ...

Ces codes 1 correcteurs parfaits sur $\mathbb{Z}/2\mathbb{Z}$ sont appelés **codes de Hamming binaires**. On les construit pour pouvoir corriger facilement une erreur avec la matrice de contrôle. Par exemple pour $m=4$, il y a une seule matrice de controle de taille $(15,4)$ telle que Hx donne la position de l'erreur (en base 2), elle est obtenue en écrivant les entiers de 1 à 15 en base 2

$$H = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

on déplace les colonnes de la matrice identité (colonnes 1, 2, 4, 8) en fin pour écrire $K=(-C, I_4)$, le code correspondant est $\begin{pmatrix} I_{11} \\ C \end{pmatrix}$, il permet de corriger une erreur, on calcule Hx et si le résultat est non nul, on change le bit d'indice Hx en tenant compte de la permutation d'indices sur les colonnes de H.

5.5 Prolongement

On peut construire des corps finis `GF(2,n)` ayant 2^n éléments en prenant les classes de congruence de polynômes à coefficients dans $\mathbb{Z}/2\mathbb{Z}$ pour la division euclidienne par un polynôme fixé de degré n qui n'admet pas de diviseur non trivial (comme pour les entiers premiers). Pour certains polynômes bien choisis, on peut fixer la distance de Hamming du code correspondant à $2t+1$ pour t quelconque et donc corriger jusqu'à t erreurs.

A Programmes

A.1 Triangle de Pascal pour calculer les coefficients binomiaux

```
# on calcule une ligne en fonction de la precedente
def Pascal(n):
    # local tableau,ligne,prec,j,k;
    tableau=[[1]]
    for j in range(1,n+1):
        ligne=[1]
        prec=tableau[j-1]
        for k in range(1,j):
            ligne.append(prec[k-1]+prec[k])
        ligne.append(1)
        tableau.append(ligne);
    return tableau
```

A.2 PGCD itératif avec reste symétrique

```
def pgcd(a,b):
    while b!=0:
         a,b=b,abs(mods(a,b))
    return abs(a) # abs est necessaire si b==0 au debut
```

En Python, il faut aussi définir la fonction `mods` :

```
def mods(a,b):
    if b<0:
        b = -b
    a
    if 2*a>b:
        return a-b
    return a
```