Cours Turing

Module sur la cryptographie

Olivier Lévêque

Table des matières

1 Cryptographie à clé secrète : un peu d'histoire **3**

1.1 Préliminaire : l'addition modulo 26 3

1.2 Le chiffre de César (Ier siècle av. J.-C.) 4

1.3 Le chiffre par substitution (monoalphabétique) 5

1.4 Le chiffre de Vigenère (XVIe siècle) 6

1.5 Enigma, Alan Turing et la naissance de l'informatique moderne 7

2 Cryptographie à clé secrète "moderne" **11**

2.1 Préliminaire : représentation binaire et addition modulo 2 (XOR) 11

2.2 Clé à usage unique (1917) 13

2.3 Data Encryption Standard (DES, 1977) 15

3 Générateurs de nombres (pseudo-)aléatoires **19**

3.1 La méthode des carrés tronqués (Von Neumann, vers 1950) 19

3.2 Les "générateurs à congruence linéaire" (Lehmer, vers 1950) 19

3.3 Opérations binaires sur des nombres entiers en Python 21

3.4 Algorithme Xorshift (Marsaglia, 2003) 22

3.5 Et encore. 22

4 La recherche de grands nombres premiers **22**

4.1 Le crible d'Eratosthène (environ 200 avant J.-C.) 23

4.2 Le théorème des nombres premiers
(Hadamard et de la Vallée Poussin, 1896) 23

5 Trouver des grands nombres premiers **24**

5.1 Arithmétique modulaire 25

5.2 Le petit théorème de Fermat 25

5.3 Bis repetita. 26

5.4 Exponentiation rapide ("square-and-multiply") 28

5.5 Factoriser des grands nombres entiers 30

6 Cryptographie à clé publique **31**

6.1 Le protocole d'échange de clé de Diffie-Hellman-Merkle : principe 31

6.2 Fonctions à sens unique 32

6.3 Le vrai protocole de Diffie-Hellman-Merkle 34

6.4 Défauts du système 35

6.5 Le même protocole avec des courbes elliptiques ! 35

7 Cryptographie à clé publique : suite **36**

7.1 Protocole de Rivest-Shamir-Adleman (RSA) 36

7.2 Signature digitale (DSA) 39

1 Cryptographie à clé secrète : un peu d'histoire

Dans cette première section, nous allons passer en revue quelques méthodes de chiffrement historiques, qui ne sont plus utilisées aujourd'hui, essentiellement à cause de la puissance des outils informatiques actuels, qui en permettent un décryptage quasi-instantané... Néanmoins, il est intéressant de se plonger dans les principes des premiers systèmes de cryptographie, car les systèmes modernes s'en inspirent fortement.

Tout au long de ce nouveau chapitre, trois personnages-clés vont nous accompagner : Alice, Bob et Eve, dans le scénario suivant : Alice cherche une méthode pour communiquer un message à Bob, tout en évitant qu'Eve, qui écoute leur conversation, soit capable de décrypter le message.

1.1 Préliminaire : l'addition modulo 26

Dans ce qui suit, nous allons manipuler l'alphabet latin, composé de 26 lettres, en nous restreignant pour simplifier aux majuscules. Sur cet alphabet, il est possible de définir une règle d'addition en identifiant chaque lettre à sa position dans l'alphabet (en commençant avec le A en position 0) :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Ainsi, on a par exemple :

```
A + A = A, A + B = B, B + B = C, B + C = D, ..., E + F = J, ...
```

car

```
0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, ..., 4 + 5 = 9, ...
```

Mais alors, qu'advient-il par exemple de Y + D ou encore P + S ? En effet, $24 + 3 = 27$ et $15 + 18 = 33$, or il n'y a que 26 lettres dans l'alphabet latin. C'est pour cela qu'on définit une *addition modulo 26* sur l'ensemble des lettres, ce qui revient à dire que si le résultat de l'addition est plus grand que 25, on lui soustrait 26.

Exemples :

Y + D = B, car $24 + 3 = 27$ et $27 - 26 = 1$

P + S = H, car $15 + 18 = 33$ et $33 - 26 = 7$

L'avantage de cette règle est que l'addition de n'importe quelle paire de lettres redonne toujours une lettre.

1.2 Le chiffre de César (Ier siècle av. J.-C.)

La première idée pour chiffrer un message remonte à loin... Pour Alice, il s'agit simplement de choisir une des 26 lettres de l'alphabet, disons K pour l'exemple, et d'additionner celle-ci à chaque lettre du message qu'elle désire envoyer. Par exemple, pour chiffrer le message BONJOUR, elle effectue les additions :

```
   B  O  N  J  O  U  R
+  K  K  K  K  K  K  K
---------------------
=  L  Y  X  T  Y  E  B
```

et envoie donc le message chiffré LYXTYEB à Bob. Deux questions se posent alors :

1. Bob est-il capable de déchiffrer son message, et si oui, comment ?
2. Et si par hasard Eve intercepte le message envoyé par Alice, que peut-elle en faire ?

1. Pour que Bob puisse déchiffrer facilement le message transmis par Alice, il importe que celle-ci lui ait transmis *au préalable* la lettre K qu'elle a utilisé pour effectuer les additions (on appelle cette lettre la *clé secrète*). Bob peut alors effectuer les soustractions correspondantes (toujours modulo 26, ce qui veut dire qu'il rajoute 26 lorsque le résultat de la soustraction est plus petit que 0) pour retrouver le message d'origine :

```
   L  Y  X  T  Y  E  B
-  K  K  K  K  K  K  K
---------------------
=  B  O  N  J  O  U  R
```

Notez que Bob pourrait de manière équivalente *additionner* une lettre à chaque lettre du message reçu pour retrouver le message d'origine : voyez-vous laquelle ?

2. Que peut faire Eve de son côté si elle intercepte le message LYXTYEB ? Elle ne connaît pas la lettre secrète choisie par Alice, mais d'un autre côté, l'alphabet latin ne comporte que 26 lettres... Rien n'empêche donc Eve d'effectuer les mêmes soustractions que Bob, chaque fois avec une lettre différente, jusqu'à ce qu'elle tombe sur un message cohérent. Eve effectue ici ce qu'on appelle une *attaque par force brute*, ce qui veut dire qu'elle teste toutes les possibilités de chiffrement. Notez qu'avec beaucoup de malchance, il se pourrait qu'Eve tombe sur deux messages cohérents avec deux choix différents de lettre (et ne puisse donc pas décrypter le message d'origine), mais cette probabilité tombe rapidement à zéro pour de longs messages.

Alice et Bob doivent donc trouver autre chose pour assurer le secret de leurs communications...

1.3 Le chiffre par substitution (monoalphabétique)

Une variante du chiffre de César est la suivante : plutôt que d'additionner toujours la même lettre à chacune des lettres du message, Alice pourrait décider de substituer chaque lettre de l'alphabet avec une autre, par exemple :

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
| | | | | | | | | | | | | | | | | | | | | | | | | |
B K Z Y M F E G A R S W C Q J O N V T H L I D U X P
```

Remarques :

- Certaines lettres peuvent rester inchangées avec ce système, comme ici le F.

- Le chiffre de César est un cas particulier du chiffre de substitution.

En utilisant ce tableau (de haut en bas) pour chiffrer le message BONJOUR, Alice obtient le message chiffré KJQRJLV, qu'elle envoie à Bob.

Comment Bob fait-il dans ce cas pour déchiffrer le message d'Alice ? A nouveau, il est nécessaire qu'Alice lui ait communiqué au préalable une clé secrète, qui n'est ici rien d'autre que le tableau ci-dessus (dans son intégralité). A l'aide du tableau, Bob peut retrouver le message d'Alice en utilisant celui-ci de bas en haut.

Se pose maintenant à nouveau la question de savoir ce qu'Eve peut faire si elle intercepte le message KJQRJLV ? La première chose à remarquer est qu'ici le nombre de clés possibles est *beaucoup* plus élevé qu'avec le chiffre de César ! Ce nombre est égal au nombre de permutations possibles des 26 lettres de l'alphabet, qui vaut $26 \cdot 25 \cdot 24 \cdots 3 \cdot 2 \cdot 1 = 26! \simeq 4 \cdot 10^{26}$. Il est donc complètement impossible pour Alice de tester toutes les possibilités en un temps raisonnable, même avec le plus performant des ordinateurs actuels... Une attaque par force brute est infaisable ici (et serait du reste bien inutile : voyez-vous pourquoi ?).

Alice et Bob auraient-ils donc trouvé le système de chiffrement idéal ? Pas si sûr... En effet, si le message envoyé par Alice est relativement long, une attaque possible d'Eve, dite aussi *attaque par analyse fréquentielle*, est la suivante : il s'agit de compter le nombre d'occurences de chaque lettre dans le message chiffré et de diviser ce nombre par le nombre total de lettres du message, pour finalement obtenir la fréquence d'apparition de chaque lettre dans le message.

Or il se trouve que dans un long texte en français, la fréquence d'apparition de chaque lettre est relativement stable et surtout connue : par exemple, le E apparaît beaucoup plus souvent que toutes les autres lettres, et les lettres A, I, S apparaissent beaucoup plus fréquemment que X, Y ou Z, par exemple. Ainsi, en calculant les fréquences d'apparition de chaque lettre dans le message chiffré, et en comparant celles-ci aux fréquences connues des lettres en français, Eve peut essayer d'établir une correspondance pour retrouver quelle lettre a été remplacée par quellle autre. Bien sûr, ce système n'est pas parfait, mais une fois que certaines lettres ont été identifiées (surtout les voyelles), il devient relativement facile pour Eve de deviner les autres lettres par déduction, en identifiant certains mots, et ainsi de retrouver le message d'origine.

Caramba, encore raté ! Essayons autre chose...

1.4 Le chiffre de Vigenère (XVIe siècle)

L'idée suivante est de choisir non pas une lettre, comme dans le chiffre de César, mais une suite de lettres, comme par exemple la suite "MDR", et d'utiliser celle-ci de manière répétée pour chiffrer un message. Voici ce que cela donne sur un exemple :

```
  B O N J O U R A L A N T U R I N G
+ M D R M D R M D R M D R M D R M D
-----------------------------------
= N R F V R L D D C M Q K G U Z Z J
```

Remarques :

- On a évité d'utiliser des espaces ici pour se restreindre aux 26 lettres latines, mais il est tout à fait possible d'ajouter l'espace à l'ensemble des lettres utilisées.

- Cette méthode est appelée un chiffrement par substitution *polyalphabétique*, car contrairement au cas précédent, une lettre donnée n'est pas systématiquement remplacée par une même autre lettre ; la substitution s'applique à des motifs de 3 lettres consécutives.

Muni de la clé secrète MDR, Bob est capable de déchiffrer le message reçu en effectuant les soustractions correspondantes (il faut certes qu'Alice et Bob s'accordent bien sur la position du début du message). Ce système a clairement l'avantage de la simplicité.

Et si Eve intercepte le message chiffré, que peut-elle faire ?

- Une attaque par force brute ?

Si la clé secrète comporte d lettres, le nombre de clés possibles vaut $\underbrace{26 \cdot 26 \cdot 26 \cdots 26}_{d \text{ fois}} = 26^d$,

ce qui grandit (très) vite avec d (p. ex., si $d = 10$, alors $26^d \simeq 10^{14}$). Il faut essayer autre chose...

- Une attaque par analyse fréquentielle ?

Comme mentionné plus haut, ce système de chiffrement a pour propriété qu'une lettre donnée n'est pas systématiquement remplacée par un même autre lettre. Une analyse naïve des fréquences des lettres dans le message chiffré ne donne donc rien ici. Mais une analyse des fréquences des motifs de d lettres consécutives pourrait quant à elle donner quelque chose. Le problème pour Eve est qu'elle ne connaît même pas la longueur d de la clé...

Ceci dit, au XIXe siècle (donc trois siècles plus tard...), Friederich Kasiski publie un test qui pemet de deviner la valeur de d ; ce test met fin à l'utilisation du chiffre de Vigenère.

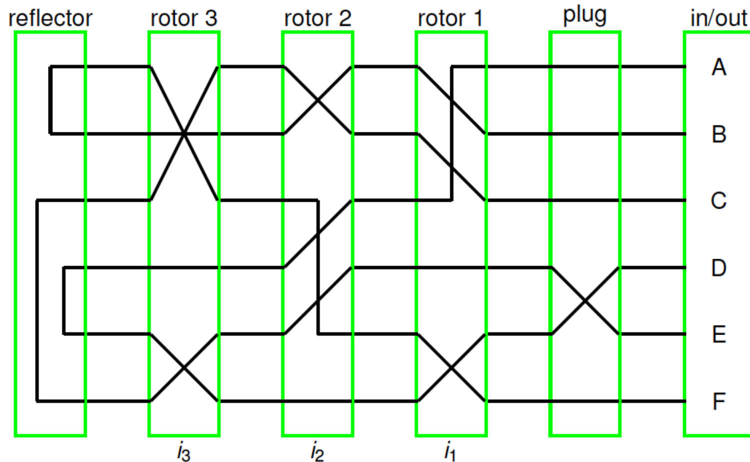
1.5 Enigma, Alan Turing et la naissance de l'informatique moderne

Le dernier exemple de système cryptographique que nous allons passer en revue aujourd'hui est celui de la machine Enigma, inventée à la fin de la première guerre mondiale et utilisée intensivement par l'armée allemande lors de la seconde. Vous en voyez ci-dessous un modèle, exposé au Musée de la Science et de la Technologie de Milan :

Le principe de base de la machine est le suivant : celle-ci est composée (entre autres) de deux claviers, et chaque frappe d'une touche du clavier du bas déclenche l'allumage d'une autre touche du clavier du haut, par un système de connexions électriques que nous allons détailler. Par exemple, pour un réglage donné, en écrivant le mot BONJOUR sur le clavier du bas, on voit s'allumer la séquence de lettres AIOZAPN sur le second clavier.

Rien qu'avec cet exemple, on voit que le chiffrement à l'œuvre ici est plus qu'un simple chiffre par substitution monoalphabétique : la lettre O est une fois remplacée par la lettre I et une autre fois par la lettre A. De même, le A dans le mot chiffré remplace une fois la lettre B et une fois la lettre O. La force du système réside dans l'introduction de 3 rotors, chacun équipé de son propre système de connexions électriques, qui tournent au fur et à mesure que le message est écrit ; ainsi, *à chaque lettre frappée sur le clavier, un autre chiffrement par substitution est utilisé !*

Voyons plus en détail comment le système fonctionne sur le schéma de la page suivante, qui suppose pour simplifier que le clavier ne comporte que les 6 lettres A, B, C, D, E et F.



Lorsqu'une lettre est frappée sur le clavier à droite (disons la lettre A), un signal électrique est envoyé à travers les connexions successives des 3 rotors (laissons tomber pour l'instant l'étape "plug" ; nous y reviendrons), pour finir dans le réflecteur sur la gauche, qui renvoie à son tour le signal à travers les trois rotors, et finit par illuminer la lettre D. Comme déjà mentionné, avant que la prochaine lettre ne soit frappée, les rotors changent de position, ce qui change les connexions ci-dessus.

L'avantage d'introduire un réflecteur est de rendre le système symétrique : ainsi, pour déchiffrer le message reçu, il suffit de placer les rotors dans la même position initiale que lors du chiffrement, et d'écrire le message chiffré sur le clavier du bas pour voir s'allumer le message d'origine sur le clavier du haut. Vous pouvez essayer par vous-même sur le site web ci-dessous, qui offre une simulation du comportement de la machine :

`https://www.101computing.net/enigma-machine-emulator/`

Reste à expliquer un dernier détail, à savoir ce que signifie le "plug" ci-dessus. Ceci fait référence au *tableau de connexions* que vous voyez aussi au bas de l'image de la page précédente : ce tableau permettait de relier deux lettres entre elles par un fil électrique pour créer une permutation de celles-ci. Ainsi, dans le schéma ci-dessus, lorsque la lettre E est frappée au clavier, le tableau de connexions effectue d'abord une permutation avec la lettre D, puis le signal suit les connexions pour finir par atteindre la lettre C. Sans cette permutation initiale, la lettre E serait chiffrée par la lettre A. En connectant de nombreuses paires de lettres ensemble (ce nombre de paires pouvait aller jusqu'à 10), le chiffrement gagne grandement en complexité, comme nous allons le voir. Bien sûr, il est alors nécessaire d'effectuer les mêmes connexions pour déchiffrer le message. Vous pouvez aussi ajouter de telles connexions sur la machine simulée.

Forces et faiblesses d'Enigma

Voyons tout d'abord quelles sont les forces de ce système. La principale force est le nombre de combinaisons offertes par un tel système !

1. Les rotors : dans la version de base, il existe 5 rotors différents, dont 3 sont choisis pour être placés dans la machine (l'ordre choisi importe), et chaque rotor a autant de positions possibles que de lettres dans l'alphabet, soit 26. Ceci donne lieu en tout à

$$5 \cdot 4 \cdot 3 \cdot 26 \cdot 26 \cdot 26 \simeq \text{un million de combinaisons possibles } (10^6)$$

2. Le tableau de connexions : le fait de pouvoir choisir 10 paires de lettres à permuter dans le tableau de connexions donne lieu quant à lui à un nombre encore plus grand de combinaisons, environ égal à $1,5 \cdot 10^{14}$.

Donc au total, le nombre de combinaisons de la machine est de l'ordre de $1,5 \cdot 10^{20}$; un chiffre absolument faramineux ! Le système de chifffrement semble juste parfait...

Oui, mais... Plusieurs défauts se sont aussi révélés au fil du temps (sans entrer dans trop de détails) :

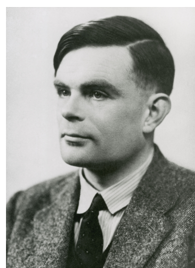
1. Le premier défaut a été de croire que le fonctionnement de la machine resterait caché des Alliés, ce qui ne fut pas le cas. Les Français, avec l'aide des Polonais, interceptent des plans de cablâge de la machine dès 1933, et c'est le mathématicien polonais Marian Rejewski qui parvient le premier à reproduire le fonctionnement de la machine.

2. Pour chaque jour, il faut que les unités communiquant entre elles se mettent d'accord sur la position des rotors et du tableau de connexions à adopter : ces positions sont consignées dans de gros cahiers transportés dans chaque unité. Certains de ces cahiers furent aussi interceptés par les Alliés pendant la guerre.

3. Dans les messages échangés entre les unités, de nombreux mots reviennent fréquemment, comme des formules de salutation au début et à la fin des messages, ou des bulletins météo. L'utilisation de ces messages répétés a pu être utilisée pour le décryptage de la machine. De plus, la fatigue et les conditions difficiles d'utilisation ont mené à certaines erreurs de transmission, ce qui a laissé échapper de précieuses informations.

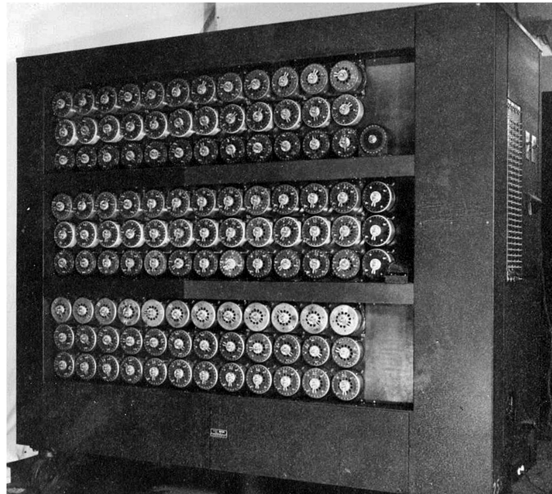
4. L'utilisation du réflecteur, si elle permet un déchiffrage aisé, a aussi pour défaut qu'une lettre n'est jamais chiffrée par elle-même, ce qui réduit (marginalement) le nombre de possibilités.

C'est en utilisant ces diverses faiblesses (entre autres) que l'unité travaillant en secret à *Bletchley Park* en Angleterre, emmenée par *Alan Turing*, a pu venir à bout du chiffrement d'Enigma.



La “bombe” d’Alan Turing

Encore une fois sans entrer dans trop de détails, mentionnons juste ici un élément clé du décryptage d’Enigma : Alan Turing et son équipe, au prix de nombreuses astuces, élaborent une méthode qui permet de faire abstraction du tableau de connexions pour le décryptage. Ainsi, le nombre de combinaisons possibles, au lieu d’être de l’ordre de 10^{20}, est ramené à un million, ce qui reste un très grand nombre de combinaisons, mais pas insurmontable... Pour essayer toutes les combinaisons possibles, Alan Turing construit une machine, qu’il appelle la *bombe*, qui permet de tester toutes ces combinaisons de manière systématique :



Sur la photo ci-dessus, vous voyez que la bombe est composée de plusieurs copies des 3 rotors d’Enigma. En un temps raisonnable, il est ainsi possible de parcourir toutes les combinaisons jusqu’à trouver la bonne.

Avec cette machine, le premier ordinateur est né : le reste appartient à l’histoire...

Les principes de Kerckhoffs et la cryptographie moderne

Mentionnons finalement une leçon retenue de cette épisode : en 1883, *Auguste Kerckhoffs* avait édicté quelques principes que tout cryptosystème devrait vérifier pour être efficace et fiable :

Le premier principe disait qu’il ne fallait pas que la sécurité du système repose sur le *secret du système en lui-même.* Ce premier principe n’a pas été respecté par Enigma... avec le résultat que l’on sait. Dans les prochaines semaines, nous verrons des systèmes de cryptographie moderne qui respectent ce premier principe !

2 Cryptographie à clé secrète "moderne"

Dans cette section nous poursuivons sur ce thème de la cryptographie à clé secrète, en nous focalisant sur deux sujets :

- la *clé à usage unique* : système démontré inviolable, mais pas sans défauts...

- le *système DES* (pour "Data Encryption Standard"), ancêtre du systèms AES (pour "Advanced Encryption Standard"), encore utilisé aujourd'hui !

2.1 Préliminaire : représentation binaire et addition modulo 2 (XOR)

Représentation binaire des nombres entiers positifs

Voici un bref rappel : pour représenter un nombre entier positif N en binaire, il importe d'écrire celui-ci comme une somme de puissances de 2, de la même façon que la représentaion décimale de ce même nombre utilise les puissances de 10. Ainsi, nous avons :

$$1984 = 1000 + 900 + 80 + 4 = \mathbf{1} \cdot 10^3 + \mathbf{9} \cdot 10^2 + \mathbf{8} \cdot 10^1 + \mathbf{4} \cdot 10^0$$

d'où la représentation décimale "1984". Voici maintenant la même chose en binaire :

$$\begin{aligned} 1984 &= 1024 + 512 + 256 + 128 + 64 \\ &= \mathbf{1} \cdot 2^{10} + \mathbf{1} \cdot 2^9 + \mathbf{1} \cdot 2^8 + \mathbf{1} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{0} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{0} \cdot 2^0 \end{aligned}$$

d'où la représentation binaire "11111000000".

Avec n bits, il est possible de représenter tous les nombres entiers allant de 0 (000...0) à $2^n - 1$ (111...1). En particulier, avec 8 bits (équivalent à 1 octet), on peut représenter tous les nombres de 0 à 255.

Représentation binaire des chaînes de caractères

Dans le code ASCII étendu, un caractère est encodé sous la forme d'un nombre entier allant de 0 à 255 (et donc représentable sous forme binaire par 1 octet) :

ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Pour représenter une chaîne de n caractères sous forme binaire, n octets sont donc nécessaires.

Addition modulo 2 (opération XOR)

Une opération qui va se révéler très utile dans la suite est l'*addition modulo 2* de deux bits, dite aussi *opération XOR* (pour "eXclusive OR" en anglais), dénotée par le symbole $\oplus$ et définie ainsi :

$$0 \oplus 0 = 0, \quad 1 \oplus 0 = 1, \quad 0 \oplus 1 = 1, \quad 1 \oplus 1 = 0$$

Cette opération correspond à l'addition classique sur les nombres 0 et 1, *sauf* pour le dernier calcul, où on ne retient que le bit des unités de l'addition 1+1 (qui rappelez-vous vaut 10 en binaire). Comme nous allons le voir dans la section suivante, il peut être très utile de réaliser des opérations XOR sur des *séquences* de bits. Par exemple, on obtient :

$$\begin{aligned} & 01101001 \\ \oplus\ & 11010011 \\ =\ & 10111010 \end{aligned}$$

(à noter que cette opération n'a *rien à voir* avec l'addition "classique" en colonnes de deux nombres entiers représentés en binaire)

En Python, cette opération s'écrit **a^b** pour deux nombres entiers **a** et **b** (int).

2.2 Clé à usage unique (1917)

Ce système de chiffrement est aussi connu sur le nom de "masque jetable" ou "chiffre de Vernam" (du nom de son inventeur), ou encore "one-time pad" dans sa version anglaise. Pour le décrire, supposons que la clé secrète K en possession d'Alice et Bob soit une séquence de n bits (nous verrons encore plus en détail comment celle-ci doit être générée). Pour chiffrer un message M encodé sous la forme d'une autre séquence de n bits, Alice effectue l'opération suivante :

$$C = M \oplus K, \quad \text{ce qui est une notation abrégée pour :} \quad C_i = M_i \oplus K_i, \quad i = 0, \dots, n-1$$

et envoie le message chiffré C à Bob. Celui-ci, également en possession de la clé K, effectue à son tour la *même opération* sur le message reçu : $D = C \oplus K$, et retrouve ainsi le message envoyé M, car

$$D = C \oplus K = (M \oplus K) \oplus K = M \oplus (K \oplus K) = M$$

En effet :

- l'opération XOR est associative, comme l'addition standard ;

- la séquence $K \oplus K$ est une séquence de 0, car nous avons vu que $0 \oplus 0 = 1 \oplus 1 = 0$, donc en effectuant l'opération XOR de la séquence de bits K avec elle-même, on obtient simplement une séquence de 0.

Sécurité de ce système

Alice et Bob ont ainsi trouvé une façon de communiquer secrètement. Pour autant, ce système est-il 100% sûr ? C'est ici qu'il importe de décrire plus précisément comment la clé K doit être générée. Pour que tout fonctionne bien, il faut trois ingrédients :

- chaque bit K_i de la clé K doit être tiré *uniformément* au hasard, c'est-à-dire que les probabilités que $K_i = 1$ et $K_i = 0$ valent toutes deux $\frac{1}{2}$, pour toute valeur de i allant de 0 à $n-1$;

- les bits $K_0, K_1, \dots, K_{n-1}$ doivent être tirés *indépendamment les uns des autres*, comme des dés qu'on lancerait sur une grande table ;

- le tirage aléatoire de la clé K doit être aussi effectué *indépendamment du message M à envoyer*.

Pourquoi toutes ces conditions ? Pour la bonne raison suivante : si maintenant Eve intercepte le message chiffré C, que peut-elle en déduire sur le message d'origine M ?

Pour chaque bit C_i, la probabilité que $C_i = 0$ est la même que la probabilité que $M_i \oplus K_i = 0$, ce qui revient à dire que $K_i = M_i$. Or $K_i = 1$ ou $K_i = 0$ avec la même probabilité $\frac{1}{2}$. Donc pour toute valeur de M_i, la probabilité que $K_i = M_i$ vaut aussi $\frac{1}{2}$. Et donc la probabilité que $C_i = 0$ vaut $\frac{1}{2}$ (et donc, il en va de même pour la probabilité que $C_i = 1$). Pour Eve, qui ne connaît ni M, ni K, la séquence de bits C apparaît donc comme une séquence de bits complètement aléatoire ! Elle ne peut donc strictement rien déduire de cette séquence sur le message M d'origine : le système est sûr à 100%.

Quiz

- Pourquoi une attaque par force brute ne permettrait-elle pas de casser le système de clé à usage unique (en supposant donc qu'Eve dispose de la puissance de calcul nécessaire pour essayer toutes les clés possibles) ?

- (Question reliée) Vu que la clé K est tirée au hasard, il y a une probabilité (petite, certes, mais non-nulle) que tous les bits de K valent exactement 0. Dans ce cas, le message chiffré C est égal au message d'origine M : est-ce grave ?

Une clé malheureusement non réutilisable (d'où son nom...)

Ceci dit, le plus gros défaut de ce système est le suivant : supposons qu'Alice désire réutiliser la clé K pour envoyer deux messages M_1 et M_2, chacun de même longueur n que la clé K. Ainsi, Alice envoie les deux messages chiffrés C_1 et C_2 suivants :

$$C_1 = M_1 \oplus K \qquad\qquad C_2 = M_2 \oplus K$$

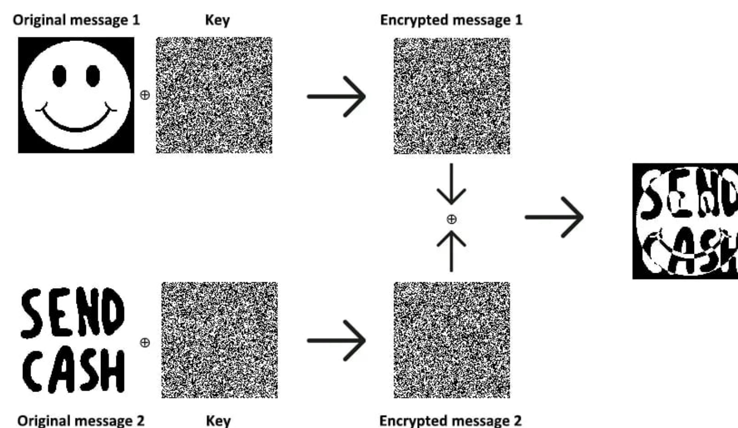
Individuellement, chaque message est bien protégé, comme nous venons de le voir. Mais si Eve intercepte les deux messages chiffrés C_1 et C_2, rien ne l'empêche d'effectuer une opération XOR sur ceux-ci, pour obtenir :

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus K \oplus M_2 \oplus K = (M_1 \oplus M_2) \oplus (K \oplus K)$$

car l'opération XOR est non seulement associative, mais aussi *commutative* (comme l'addition standard). D'autre part, nous avons vu aussi que $K \oplus K = 0$, et donc, finalement :

$$C_1 \oplus C_2 = M_1 \oplus M_2$$

ce qui veut dire que la clé K a maintenant totalement disparu ! Certes, il peut être difficile pour Eve de déduire quelque chose à propos des messages M_1 et M_2 à partie du seul $M_1 \oplus M_2$, mais la sécurité du chiffrement n'est plus du tout assurée ! Pensez par exemple au cas où M_1 contient une longue suite de 0... Le problème est bien illustré sur le schéma ci-dessous : si deux images sont chiffrées avec la même clé, alors un XOR des deux images chiffrées (où noir et blanc jouent le rôle de 0 et 1) donne l'image à droite :



Cette impossibilité de réutiliser de la clé est clairement un très grave défaut du système ! Si Alice désirait chiffrer un gros fichier de données avec ce système, elle devrait produire une clé de même taille, ce qui en soi est déjà coûteux (car produire de longues séquences de bits aléatoires ne s'improvise pas), mais il y a bien pire : il faudrait d'abord partager la clé secrètement avec Bob avant de communiquer. Or si une telle occasion de communiquer secrètement se présentait, pourquoi ne pas utiliser celle-ci pour communiquer directement le fichier ?

Tentons donc autre chose pour réutiliser la clé K pour le chiffrement de plusieurs messages...

2.3 Data Encryption Standard (DES, 1977)

Revenons au chiffrement d'un seul message M avec une clé K de même longueur, disons n. Un premier essai serait le suivant : plutôt que d'effectuer une opération XOR de M et K, pourquoi ne pas effectuer le XOR du message M avec une *fonction* de la clé K et du message M ? Ceci donnerait le message chiffré suivant :

$$C = M \oplus f(K, M)$$

où f est donc une fonction qui à deux séquences de n bits K et M fait correspondre une autre séquence de n bits $f(K, M)$. Si la fonction f est suffisamment compliquée (pour être précis, on parle de fonction "non-linéaire"), cette opération combine le message M et la clé K d'une façon beaucoup plus intriquée que le système de clé à usage unique, et permet donc potentiellement de réutiliser la clé K pour chiffrer un autre message, sans qu'il soit cette fois-ci possible pour Eve d'effectuer une opération simple pour se débarasser de la clé K.

Oui, mais... Ce nouveau système a un gros défaut : Bob, même s'il connaît la clé K, n'est pas non plus capable de retrouver le message M ! Voilà donc une fausse bonne idée. Ceci dit, tout n'est pas à jeter. C'est Horst Feistel qui découvre le moyen de rendre cette idée applicable en pratique au moyen de l'algorithme suivant :

1. Commençons par découper le message M et la clé K chacun en deux parties égales :

$$M = (M_a, M_b) \quad \text{et} \quad K = (K_a, K_b)$$

Pour simplifier les notations, nous dirons que le message M et la clé K sont maintenant composés chacun de $2n$ bits, de sorte que M_a, M_b, K_a et K_b sont toutes des séquences de n bits.

2. Nous allons maintenant réutiliser la même fonction f que ci-dessus, mais sur chacune des deux parties du message M, pour obtenir successivement :

$$C_a = M_a \oplus f(K_a, M_b) \quad \text{puis} \quad C_b = M_b \oplus f(K_b, C_a)$$

Notez bien que ces deux opérations doivent s'effectuer dans cet ordre, car le résultat C_a de la première opération est utilisé pour calculer C_b.

3. Alice envoie le message chiffré $C = (C_a, C_b)$ à Bob. Si Eve intercepte le message C sans connaître la clé K, elle est toujours bien embêtée pour décrypter celui-ci. Mais Bob peut-il le déchiffrer, qui lui connaît la clé K ?

4. C'est là toute l'ingéniosité du système : Bob effectue à nouveau les *mêmes opérations qu'Alice, mais dans l'ordre inverse* :

$$D_b = C_b \oplus f(K_b, C_a) \quad \text{puis} \quad D_a = C_a \oplus f(K_a, D_b)$$

Qu'obtient-il ? Tout d'abord, observez que

$$D_b = C_b \oplus f(K_b, C_a) = (M_b \oplus f(K_b, C_a)) \oplus f(K_b, C_a) = M_b \oplus (f(K_b, C_a) \oplus f(K_b, C_a)) = M_b$$

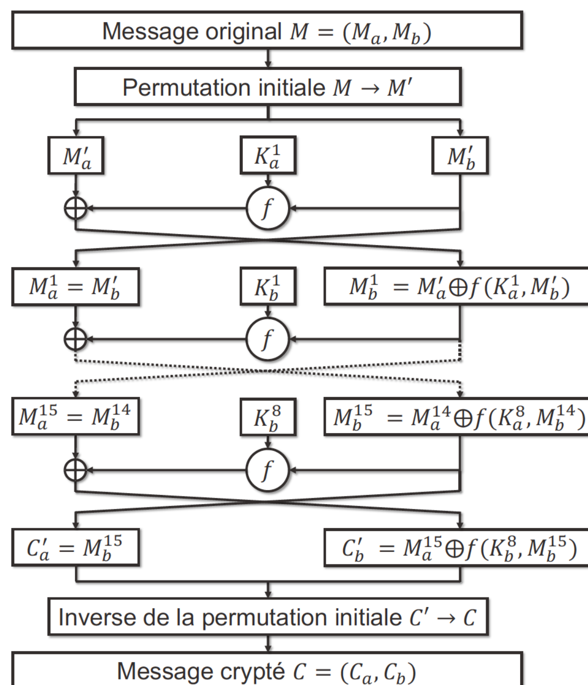
pour les mêmes raisons que lors de l'utilisation de la clé à usage unique. Aussi compliquée que soit la fonction f, elle disparaît avec l'opération XOR ! Donc la deuxième partie du message déchiffré D_b correspond bien à la deuxième partie du message d'origine M_b. Nous pouvons maintenant passer à la dernière étape :

$$\begin{aligned} D_a &= C_a \oplus f(K_a, D_b) = C_a \oplus f(K_a, M_b) \quad \text{(par ce qui vient d'être dit)} \\ &= (M_a \oplus f(K_a, M_b)) \oplus f(K_a, M_b) = M_a \oplus (f(K_a, M_b) \oplus f(K_a, M_b)) = M_a \end{aligned}$$

par le même mécanisme que précédemment. Ainsi Bob peut parfaitement déchiffrer le message envoyé par Alice, tandis qu'Eve ne peut rien faire.

L'avantage par rapport à la clé à usage unique est qu'Alice peut réutiliser ici ce système avec la même clé K pour chiffrer successivement deux messages M_1 et M_2, et qu'il sera très difficile pour Eve de se débarrasser de la clé K dans ce contexte, à cause de la fonction non-linéaire f.

Le système DES utilise quant à lui l'algorithme ci-dessus de manière répétée (8 fois, plus précisément), avec en plus une permutation de la séquence en entreée et son inverse en sortie. Il est illustré sur le schéma ci-dessous :



Notez que pour aussi complexe qu'il soit, le système DES a le même avantage que la machine Enigma et beaucoup d'autres systèmes cryptographiques : pour déchiffrer le message, il suffit de faire la même suite d'opérations (dans l'ordre inverse) que lors du chiffrement !

Finalement, notez qu'en pratique, $n = 32$, ce qui mène à une longueur totale de clé de 64 bits, ou 8 octets, pour la clé K ; cependant, dans chaque octet, un bit de est utilisé comme *bit de parité*, ce qui réduit la taille de la clé à $8 \cdot 7 = 56$ bits.

Attaque par force brute de la clé

Le système DES a finalement pu être décrypté par une attaque par force brute de sa clé K, d'une longueur de 56 bits. Combien d'opérations cela représente-t-il ?

En général, le nombre de possibilités pour construire une clé de longueur n, où chaque symbole peut être choisi dans un alphabet de k symboles, est donné par $\mathbf{k^n}$: veuillez noter que ce nombre augmente extrêmement rapidement avec n. Exemples :

- il y a 2^n possibilités pour une clé composée de n bits 0 ou 1 ;

- il y a 10^n possibilités pour une clé composée de n chiffres de 0 à 9 ;

- il y a 26^n possibilités pour une clé composée de n lettres majuscules de A à Z ;

- il y a 52^n possibilités pour une clé composée de n lettres minuscules et majuscules.

Voyons ce que ça donne concrètement pour certaines valeurs de n :

- clé composée de 6 chiffres : 10^6 possibilités

- clé composée de 6 lettres majuscules : $26^6 \simeq 3 \cdot 10^9$ possibilités

- clé composée de 6 lettres minuscules et majuscules : $52^6 \simeq 2 \cdot 10^{10}$ possibilités

- clé composée de 15 chiffres : 10^{15} possibilités

- clé composée de 56 bits : $2^{56} \simeq 7 \cdot 10^{16}$ possibilités

En faisant le parallèle entre ces clés et les mots de passe que vous utilisez tous les jours, ceci vous donne une idée du nombre d'opérations qu'il faut effectuer pour attaquer ce mot de passe par force brute. Le tableau de la page suivante résume bien la situation.

TEMPS REQUIS POUR DÉCHIFFRER UN MOT DE PASSE

NOMBRE DE CARACTÈRES	CHIFFRES SEULEMENT	LETTRES MINUSCULES	LETTRES MINUSCULES ET MAJUSCULES	CHIFFRES, LETTRES MINUSCULES ET MAJUSCULES	SYMBOLES, CHIFFRES, LETTRES MINUSCULES ET MAJUSCULES
4	Instantanément	Instantanément	Instantanément	Instantanément	Instantanément
5	Instantanément	Instantanément	Instantanément	Instantanément	Instantanément
6	Instantanément	Instantanément	Instantanément	1 seconde	5 secondes
7	Instantanément	Instantanément	25 secondes	1 minute	6 minutes
8	Instantanément	5 secondes	22 minutes	1 heure	8 heures
9	Instantanément	2 minutes	19 heures	3 jours	3 semaines
10	Instantanément	58 minutes	1 mois	7 mois	5 ans
11	2 secondes	1 jour	5 ans	41 ans	400 ans
12	25 secondes	3 semaines	300 ans	2000 ans	34k ans
13	4 minutes	1 an	16k années	100k ans	2M ans
14	41 minutes	51 ans	800k années	9M ans	200M ans
15	6 heures	1k ans	43M ans	600M ans	15G ans
16	2 jours	34k ans	2G ans	37G ans	1T ans
17	4 semaines	800k ans	100G ans	2T ans	93T ans
18	9 mois	23M ans	2T ans	100T ans	$7(10^{48})$ ans

Traduction libre des données recueillies par Hive Systems

Guiddy

Sur ce tableau (et par ce qui précède, également), on voit que ce n'est pas tant en augmentant le nombre de symboles différents k dans l'alphabet qu'on rend un mot de passe plus sûr, mais bien en augmentant la taille n du mot de passe lui-même. A l'époque où le système DES a été conçu (dans les années 70), il était raisonnable de penser que 56 bits serait une longueur de clé suffisante pour résister à une telle attaque, mais tel n'était plus le cas vingt ans plus tard... Ce système a donc été remplacé par le système AES, qui utilise des clés de 128, 192 ou même 256 bits.

3 Générateurs de nombres (pseudo-)aléatoires

[1] La semaine dernière, nous avons vu le système de clé à usage unique et comment sa sécurité repose sur la génération d'une clé K aléatoire. Il existe un grand nombre d'applications en informatique, et plus particulièrement en cryptographie, qui reposent sur la génération de nombres aléatoires. Mais comment générer des nombres aléatoires à partir d'un ordinateur, qui est fondamentalement une machine déterministe ?

Pour simuler le hasard, une première idée est d'utiliser la représentation binaire du temps indiqué par l'horloge de l'ordinateur (par exemple, le samedi 18 novembre 2023 à 9 heures, 3 minutes, 45 secondes, 17 centièmes et...). Ceci fournit une séquence de bits qu'on peut considérer comme plus ou moins aléatoire (en considérant au besoin une sous-séquence de celle-ci) et qui représente un nombre entier donné. Mais comment choisir ensuite d'autres nombres entiers aléatoires ? Si on consulte l'horloge de l'ordinateur à intervalles réguliers pour trouver d'autres nombres, forcément qu'un motif répétitif fera irruption dans la séquence et endommagera la côté aléatoire de celle-ci.

On peut cependant retenir le premier nombre ainsi trouvé comme *graine* (ou *seed* en anglais) à fournir à un algorithme qui, à partir de là, génère une séquence de nombres qu'on espère "les plus aléatoires possibles". Encore une fois, vu que les ordinateurs sont des machines déterministes, l'algorithme, quel qu'il soit, ne pourra pas fournir de vrais nombres aléatoires : on parle donc de générateurs *pseudo*-aléatoires. En voici quelques exemples.

3.1 La méthode des carrés tronqués (Von Neumann, vers 1950)

Une première idée pour générer des séquences de (grands) nombres aléatoires est la suivante. A partir d'un nombre à 8 chiffres, par exemple $x = 30472901$, calculons son carré :

$$x^2 = (0)928597695355801$$

et ne retenons que les 8 chiffres du milieu de celui-ci (en rouge) pour le prochain nombre, et ainsi de suite... Cette idée, pour ingénieuse qu'elle soit, génère cependant une séquence de nombres qui est loin d'être aléatoire : au bout d'un moment, la séquence ainsi produite tombe sur un nombre qui se répète (comme par exemple $x = 60 \to x^2 = 3600$) ou effectue une boucle à travers quelques valeurs seulement (comme par exemple $x = 57 \to x^2 = 3249 \to (0)576$).

3.2 Les "générateurs à congruence linéaire" (Lehmer, vers 1950)

La deuxième idée (qui a un nom bien savant...) est la suivante : on choisit tout d'abord deux nombres a et b fixés, ainsi qu'un nombre m (tous des nombres entiers positifs), et à partir d'un nombre x, on calcule la nombre suivant en effectuant l'opération :

$$(a \cdot x + b) \pmod{m}$$

1. Cette section reprend des éléments du rapport "Générateurs de nombres aléatoires", par J.-C. Barros, Y. Dethurrens, D. Kessler et J.-F. Ravoux, juillet 2021

(où pour rappel, $z \pmod{m}$ désigne le reste de la division de z par m, donc un nombre compris entre 0 et $m-1$). Par exemple, en choisissant $a = 4$, $b = 3$ et $m = 9$, on obtient la suite de nombres

$$x = 1 \to 7 \pmod{9} = 7 \to 31 \pmod{9} = 4 \to 19 \pmod{9} = 1$$

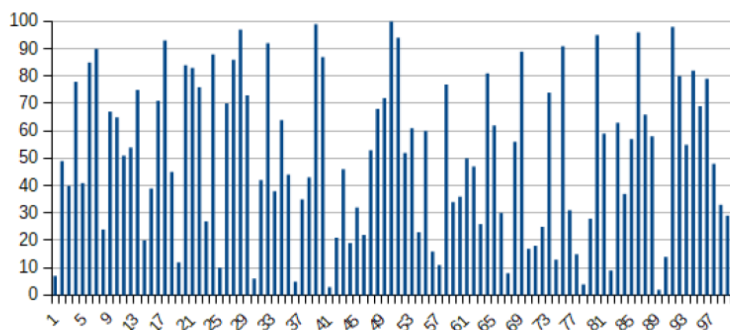
ou encore

$$x = 5 \to 23 \pmod{9} = 5$$

Comme on le voit, cette méthode a aussi des défauts ! Mais notez qu'en choisissant $a = 4$, $b = 2$ et $m = 9$, on obtient le cycle le plus long possible pour cette valeur de m :

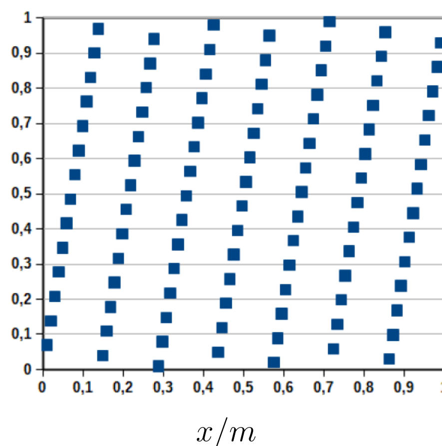
$$x = 1 \to 6 \to 8 \to 7 \to 3 \to 5 \to 4 \to 0 \to 2 \to 1$$

Il en va de même si on choisit $m = 101$, $a = 7$ et $b = 0$: la séquence des nombres ainsi générée explore toutes les valeurs possibles entre 0 et 100 avant de revenir au point de départ, comme le montre le graphique suivant :



et les nombres semblent explorés dans un ordre aléatoire. Mais si on représente maintenant la succession de ces nombres en deux dimensions :

$$\frac{(a \cdot x + b) \pmod{m}}{m}$$



on note alors le caractère grandement prévisible de cette séquence de nombres !

En choisissant des valeurs (nettement !) plus grandes de m, comme par exemple $m = 2^{32}$ (et $a = 129$, $b = 907633385$), on peut bien sûr obtenir des séquences de nombres plus intéressantes, mais le caractère prévisible observé ci-dessus ne disparaît pas.

3.3 Opérations binaires sur des nombres entiers en Python

Pour introduire la prochaine idée, nous avons besoin de quelques opérations binaires sur les nombres entiers en Python. Rappelons tout d'abord différentes représentations des nombres entiers : si x est un nombre entier (de type int), alors bin(x) et hex(x) permettent d'afficher (au format str) les représentations binaires et hexadécimales, respectivement, de ce même nombre entier. Par exemple, si $x = 156$, alors

$$\begin{cases} \text{bin}(x) = \text{“}0b10011100\text{''} & \text{car } 156 = 128 + 16 + 8 + 4 = 2^7 + 2^4 + 2^3 + 2^2 \\ \text{hex}(x) = \text{“}0x9c\text{''} & \text{car } 156 = 9 \cdot 16 + 12 \end{cases}$$

A noter pour toutes les opérations listées ci-dessous, il n'y a pas besoin de convertir les nombres en binaire avant d'effectuer ces opérations : Python utilise directement la représentation binaire des nombres dans ce cas !

1. L'opération XOR (notée x^y en Python) que nous avons vue la semaine dernière, s'effectue bit à bit sur la représentation binaire des deux nombres entiers x et y. Ainsi, si par exemple, $x = 156$ et $y = 17$, nous obtenons pour $z = x$^y :

```
  1 0 0 1 1 1 0 0 (x)
^ 0 0 0 1 0 0 0 1 (y)
--------------------
= 1 0 0 0 1 1 0 1 (z)
```

et donc $z = 128 + 8 + 4 + 1 = 141$.

2. De même, nous pouvons définir les opérations binaires ET (notée &) et OU (notée |) qui donnent respectivement sur les mêmes nombres $x = 156$ et $y = 17$ que ci-dessus :

```
  1 0 0 1 1 1 0 0        1 0 0 1 1 1 0 0
& 0 0 0 1 0 0 0 1      | 0 0 0 1 0 0 0 1
----------------       ----------------
= 0 0 0 1 0 0 0 0      = 1 0 0 1 1 1 0 1
```

donc $x \;\&\; y = 16$ et $x \mid y = 128 + 16 + 8 + 4 + 1 = 157$.

3. Finalement, nous aurons encore besoin de deux autres opérations, notées $<<$ (“décalage à gauche”) et $>>$ (“décalage à droite”) dont la fonction est de décaler (comme leur nom l'indique…) les bits qui composent un nombre d'un certain nombre de positions.

Par exemple, $x << 3$ veut dire décaler de 3 positions vers la gauche les bits qui composent le nombre x (en rajoutant des 0). Si par exemple $x = 156 = 0b10011100$ en binaire, alors $x << 3 = 0b10011100000 = 1248 = 156 \cdot 8$, vu que chaque décalage vers la gauche en binaire revient à multiplier le nombre par 2.

De même, $x >> 3$ décale de 3 positions vers la droite les bits du nombre x, en laissant éventuellement tomber les 1 qui se trouveraient aux 3 dernières places. Par exemple, si $x = 156 = 0b10011100$, alors $x >> 3 = 0b10011 = 19$ (qui n'est rien d'autre que le quotient de 156 par 8, où le reste de la division est négligé).

3.4 Algorithme Xorshift (Marsaglia, 2003)

Dans sa version la plus simple, l'idée de l'algorithme est la suivante : partir d'une "graine" d'une longueur de 32 bits, et effectuer un XOR de celle-ci avec une version décalée vers la gauche d'elle-même pour générer le prochain nombre. Ceci ne suffit toutefois pas (vous comprendrez pourquoi en faisant l'exercice correspondant) ; il faut encore effectuer à deux reprises une opération similaire pour obtenir une suite de nombres qui soit proche d'une suite aléatoire. Ceci donne l'algorithme suivant (en supposant qu'une valeur initiale a été attribuée à x) :

```
x = x^(x << 13)
x = x^(x >> 17)
x = x^(x << 5)
```

(à répéter n fois)

A noter que de nombreuses autres versions de l'algorithme existent, qui ont pour but de perfectionner celui-ci. Notamment, plus l'algorithme est sophistiqué, plus celui-ci passe un nombre important de tests qui vérifient le caractère aléatoire de la séquence ainsi produite : mais rentrer dans ces détails nous emmènerait un peu trop loin…

3.5 Et encore…

Pour produire des séquences de nombres pseudo-aléatoires, de nombreuses autres méthodes ont été inventées : citons notamment le "Mersenne twister", ainsi que d'autres algorithmes reposant sur la théorie du chaos, ou même le bruit atmosphérique ! (cf. `random.org`). Mais le générateur ultime de "vrais" nombres aléatoires reste à l'heure actuelle celui qui repose sur les propriétés quantiques de la matière au niveau microscopique, se basant en particulier sur la polarisation des photons, qui par nature se comporte de manière aléatoire lorsqu'on la mesure… A noter que cette dernière méthode reste beaucoup plus lente (et aussi plus coûteuse) que les précédentes.

4 La recherche de grands nombres premiers

Bon nombre d'algorithmes de cryptographie à clé publique, que nous découvrirons dans les prochaines leçons, reposent sur un ingrédient essentiel, à savoir l'utilisation de grands nombres premiers (pour rappel, un nombre premier est un nombre qui n'admet que deux diviseurs distincts : 1 et lui-même). Par "grands nombres premiers", on entend ici de *vraiment grands* nombres premiers, par exemple des nombres à 100 chiffres. La tâche de trouver de tels nombres est un peu ardue, mais nous allons progresser pas à pas.

4.1 Le crible d'Eratosthène (environ 200 avant J.-C.)

Cette (première !) méthode pour trouver des nombres premiers est la suivante : l'idée est de dresser la liste de tous les nombres, puis de rayer systématiquement tous les nombres qui sont des multiples de 2, de 3, de 5, etc. (notez que 4 étant un multiple de 2, il n'y pas besoin de rayer les multiples de 4, qui sont donc tous aussi des multiples de 2) : le nombres qui restent sont les nombres premiers. Voici une illustration de cette méthode, où les nombres premiers sont ceux colorés en rose (auxquels il faut encore ajouter les premiers nombres de chaque autre couleur) :

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	107
109	113		

Cette méthode est certes très intéressante pour établir la liste des nombres premiers existants, mais devient de plus en plus lente lorsqu'on cherche de plus en plus grands nombres : ça n'est donc pas ainsi qu'il faut procéder pour trouver des nombres premiers à 100 chiffres...

4.2 Le théorème des nombres premiers (Hadamard et de la Vallée Poussin, 1896)

Pour de plus grands nombres, une question qui vient tout de suite à l'esprit est la suivante : est-ce que des nombres premiers à 100 chiffres existent ? (car si tel n'est pas le cas, mieux vaut arrêter tout de suite...). A priori, vu qu'un nombre à 100 chiffres est de l'ordre de 10^{100}, il a donc aussi de l'ordre de 10^{100} diviseurs potentiels : il est donc justifié de se poser la question ! C'est cependant un fait qui est connu depuis Euclide : il existent une infinité de nombres premiers.

Mais vient maintenant la deuxième question : vu que plus un nombre grandit, plus il a de diviseurs potentiels, il doit donc être vrai que les nombres premiers se *raréfient* plus on va vers de grands nombres. Par exemple :

- parmi les 10 premiers nombres, on trouve 4 nombres premiers (donc 40%) ;

- parmi les 100 premiers nombres, on trouve 25 nombres premiers (donc 25%) ;

- parmi les 1000 premiers nombres, on trouve 168 nombres premiers (donc 17% environ) ;

- etc.

Si les nombres premiers se raréfiaient trop lorsque qu'on va vers de grandes valeurs, la recherche de grands nombres premiers serait problématique... Heureusement, le *théorème des nombres premiers* nous dit que cette raréfaction est plutôt lente, à savoir :

$\pi(N)$, le nombre de nombres premiers plus petits ou égaux à un nombre donné N,
est approximativement donné par $\pi(N) \simeq \frac{N}{\ln(N)}$

où $\ln(N)$ est le *logarithme népérien de* N. Ceci veut dire que la *proportion* de nombres premiers plus petits ou égaux à N est de l'ordre de $\frac{1}{\ln(N)}$.

La fonction $\ln(N)$ est une fonction qui grandit très lentement avec N (par exemple, $\ln(1000) \simeq 6.9$, tandis que $\ln(1000000) \simeq 13.8$), donc la proportion $\frac{1}{\ln(N)}$ décroît très lentement avec N.

A nouveau, si N est un nombre à 100 chiffres, qu'implique ce résultat ? Dans ce cas, $N \simeq 10^{100}$ et on peut calculer que $\ln(N) \simeq 230$: même si N est gigantesque, $\ln(N)$ a une valeur assez faible : ceci veut dire en pratique que si on tire complètement au hasard un nombre à 100 chiffres, on a environ une chance sur 230 de tomber sur un nombre premier ; c'est une probabilité certes assez faible, mais suffisamment grande pour ne pas se décourager, car en effectuant 230 essais, on obtient une probabilité non-négligeable de tomber au moins une fois sur un nombre premier. Reste maintenant la question épineuse de trouver une manière efficace de *tester* si un nombre donné N est premier...

5 Trouver des grands nombres premiers

Comme nous l'avons vu précédemment, il n'est a priori pas du tout aisé de vérifier qu'un nombre N à 100 chiffres est un nombre premier... Par exemple, vérifier que N n'est divisible par aucun nombre plus petit que lui (excepté 1 et lui-même) prend de l'ordre de 10^{100} divisions, ce qui est plus que le nombre estimé d'atomes dans l'univers !

On peut réduire sensiblement ce nombre d'opérations en observant la chose suivante : si un nombre N n'est pas premier et est donc le produit de deux nombres P et Q, c'est forcément le cas que soit P soit Q est plus petit ou égal à $\sqrt{N}$ (sinon, on obtiendrait que $P \cdot Q > \sqrt{N} \cdot \sqrt{N} = N$). Et donc, pour tester si un nombre est premier, il suffit de tester tous ses diviseurs potentiels de 2 jusqu'à $\sqrt{N}$, et non de 2 jusqu'à N. Alors certes, $\sqrt{N}$ est beaucoup plus petit que N, mais lorsque $N = 10^{100}$, on obtient que $\sqrt{N} = 10^{100/2} = 10^{50}$, ce qui représente un nombre encore extrêmement grand !

Il existe heureusement une autre façon beaucoup plus efficace de procéder ! Mais elle demande un peu de travail...

5.1 Arithmétique modulaire

Commençons par une définition : étant donné un nombre entier positif N, on dit que "A modulo N est égal à B" et on écrit :

$$A \pmod N = B$$

si B est le reste de la division de A par N. B est donc un nombre compris entre 0 et $N-1$. Lorsqu'on effectue des opérations modulo N, on peut se limiter à ces nombres, ce qui, comme nous allons le voir, n'est pas sans intérêt !

Voici un exemple avec $N = 11$ et $A = 64$, on a $A = 5 \cdot 11 + 9$ donc $A \pmod{11} = 9$.

Et voici comment effectuer des additions et multiplications modulo N (toujours avec $N = 11$) :

- Si $A = 64$ et $B = 17$, alors $(A + B) \pmod{11} = 81 \pmod{11} = 4$. Mais notez qu'on aurait pu aussi calculer d'abord :

$A \pmod{11} = 64 \pmod{11} = 9$ et $B \pmod{11} = 17 \pmod{11} = 6$

et effectuer ensuite $(9 + 6) \pmod{11} = 15 \pmod{11} = 4$ pour arriver au même résultat.

- Toujours avec $A = 64$ et $B = 17$, on peut effectuer directement $(A \cdot B) \pmod{11} = 1088 \pmod{11} = 10$, mais on peut aussi se simplifier la vie en utilisant à nouveau le fait que $A \pmod{11} = 9$, $B \pmod{11} = 6$, puis en calculant $(9 \cdot 6) \pmod{11} = 54 \pmod{11} = 10$.

Nous avons utilisé ici le fait que les opérations d'addition et de multiplication commutent chacune avec l'opération modulo (ce qui signifie qu'on peut effectuer ces opérations dans l'ordre qu'on désire). C'est très pratique, car cela permet de ne jamais faire de calculs impliquant des nombres plus grands que N.

5.2 Le petit théorème de Fermat

Maintenant que nous avons vu quelques opérations de base en arithmétique modulaire, en quoi donc cela peut-il nous être utile pour trouver des nombres premiers ? La première relation avec les nombres premiers est la suivante, aussi connue sous le nom de *petit théorème de Fermat*[2] :

Si N est un nombre premier, alors $A^{N-1} \pmod N = 1$ pour tout nombre entier $1 \leq A < N$.

Malheureusement, l'assertion ne va pas dans le bon sens ! En effet, si on trouve un nombre $1 \leq A < N$ tel que $A^{N-1} \pmod N = 1$, ceci n'implique pas a priori que N est un nombre premier. Par contre, on peut utiliser la *contraposée* de ce théorème :

S'il existe un nombre entier $2 \leq A < N$ tel que $A^{N-1} \pmod N \neq 1$, alors N n'est *pas* un nombre premier (noter qu'il est impossible ici que $A = 1$, car $1^{N-1} \pmod N = 1$ pour tout N).

2. Attention à ne pas confondre ici avec le *grand* ou *dernier* théorème de Fermat, qui est resté un probléme ouvert pendant de nombreuses années, plus précisément de 1637, date de son énoncé, à 1995, date de la parution de sa démonstration par Andrew Wiles.

Ainsi, on a un outil pour dénicher des grands nombres qui ne sont *pas* premiers. “A quoi bon ?” direz-vous, car ce qui nous intéresse, ce sont les grands nombres premiers... Il se trouve qu'une extension du petit théorème de Fermat va nous aider. Celle-ci dit la chose suivante :

S'il existe un nombre $2 \leq A \leq N-1$ tel que $\mathrm{PGDC}(A, N) = 1$ et $A^{N-1} \pmod N \neq 1$, alors $B^{N-1} \pmod N \neq 1$ pour au moins la moitié des autres nombres B compris entre 2 et $N-1$.

Ce que dit ce résultat pour l'essentiel (si on fait abstraction d'un petit détail ; voir la 2e remarque ci-dessous), c'est : *si on choisit un nombre A uniformément au hasard entre 2 et $N-1$ et qu'on trouve que $A^{N-1} \pmod N = 1$, alors il y a au moins 50% de chances pour que N soit un nombre premier.* En effet, puisque si N n'était pas premier, au moins la moitié des nombres A entre 2 et $N-1$ donneraient $A^{N-1} \pmod N \neq 1$). Nous allons maintenant voir en détail quelle utilisation faire de ce résultat pour transformer l'essai, car il est clair qu'on ne peut pas se satisfaire d'un algorithme qui se trompe une fois sur deux !

5.3 Bis repetita...

L'idée est maintenant de répéter l'expérience précédente, en tirant plusieurs nombres $A_1, A_2, \ldots, A_k$ au hasard, chacun entre 2 et $N-1$, et indépendamment les uns des autres (d'où l'importance d'avoir un bon générateur de nombres aléatoires). Pour tous ces nombres, on teste si

$$A_1^{N-1} \pmod N = 1, \quad A_2^{N-1} \pmod N = 1, \quad \ldots, \quad A_k^{N-1} \pmod N = 1$$

- Si on n'obtient pas le résultat 1 pour un des nombres $A_1, A_2, \ldots, A_k$, alors on sait par le petit théorème de Fermat (plus précisément, par sa contraposée), que N n'est pas premier. Fin de la discussion : il faut essayer une nouvelle valeur de N.

- Si par contre c'est le cas qu'on obtient le résultat 1 pour chacun des nombres $A_1, A_2, \ldots, A_k$, alors qu'est-ce que ça nous dit ? Par ce qui précède, *si N n'est pas un nombre premier*, alors il y a au moins 50% de chances, pour chaque nombre A, que $A^{N-1} \pmod N \neq 1$. Donc dans ce cas, il faudrait être vraiment malchanceux pour obtenir le résultat 1 pour chacun des nombres $A_1, A_2, \ldots, A_k$. Plus précisément, la chance que ça arrive vaut au plus :

$$p = \underbrace{\frac{1}{2} \cdot \frac{1}{2} \cdots \frac{1}{2}}_{k \text{ fois}} = \frac{1}{2^k}$$

Pour une valeur même assez petite de k, cette probabilité p est très proche de 0. Par exemple : si $k = 10$, alors $p \simeq 0,001$; si $k = 20$, alors $p \simeq 0,000001$, et si $k = 30$, alors $p \simeq 0,000000001$. En conclusion : si on n'obtient que des résultats 1 aux k tests ci-dessus, on peut déclarer avec confiance que N est un nombre premier (à noter que si N est un nombre premier, alors par le petit théorème de Fermat, on sait qu'on obtiendra toujours le résultat 1).

Remarques

- Il peut sembler assez troublant que pour vérifier une propriété déterministe d'un nombre entier, à savoir ici sa *primalité*, on ait recours au hasard. Par ce qui précède, vous voyez cependant que ce hasard peut être réduit “autant qu'on veut", en assez peu d'étapes. En pratique, c'est tout à fait acceptable ! Il existe d'ailleurs une pléthore d'algorithmes qui font appel au hasard pour résoudre des problèmes déterministes. Sans le hasard, nos ordinateurs modernes seraient beaucoup moins puissants !

- Un détail a été omis dans la présentation qui précède : il existe quelques nombres entiers N (pas si nombreux, mais quand-même), qui possèdent la propriété étrange de ne pas être premiers tout en vérifiant $A^{N-1} \pmod N = 1$ pour *toutes* les valeurs de A entre 2 et $N-1$. Ces nombres sont appelés les *nombres de Carmichael*. Le plus petit d'entre eux est $N = 561 = 17 \cdot 33$. Pour gérer ce problème, un autre algorithme est nécessaire : l'*algorithme de Miller-Rabin*.

Et pour finir, reste une question cruciale : est-ce qu'avec tout ce travail, nous avons gagné quoi que ce soit par rapport à l'algorithme qui teste tous les diviseurs de N allant de 2 à $\sqrt{N}$? ? ? Faisons un peu les comptes pour voir. . .

Comme mentionné au début de ce chapitre, l'algorithme classique demande d'effectuer de l'ordre de 10^{50} divisions pour vérifier qu'un nombre à 100 chiffres est premier. De plus, comme déjà vu la semaine dernière, en tirant au hasard un nombre à 100 chiffres, on a environ une chance sur 230 de tirer un nombre premier. Au total, on trouvera donc avec cette méthode un nombre premier après

$$230 \cdot 10^{50} \text{ opérations}$$

en moyenne ; autrement dit, mieux vaut être patient !

Comparons maintenant avec la méthode vue plus haut : certes, on n'échappe pas au fait qu'il faut tirer de l'ordre de 230 nombres N au hasard pour finalement tomber sur un nombre premier. Mais combien d'opérations coûte à chaque fois le test proposé ci-dessus ? Pour fixer les idées, disons qu'on effectue le test avec $k = 30$ nombres A tirés au hasard (ce qui rappelons-le mène à une probabilité d'erreur inférieure ou égale à 0,000000001) : pour chaque nombre, ceci consiste à calculer

$$A^{N-1} \pmod N \tag{1}$$

Heureusement, comme nous allons le voir ci-dessous, pour un nombre N à 100 chiffres, ce calcul ne demande pas plus de deux millions d'opérations. Donc au total, le nombre d'opérations à effectuer pour trouver un nombre premier à 100 chiffres avec cette méthode est de l'ordre de

$$230 \cdot 30 \cdot 2 \text{ millions} \simeq 10 \text{ milliards}$$

Or 10 milliards d'opérations, avec un ordinateur moderne, ça se fait très vite (en tout cas beaucoup plus vite que $230 \cdot 10^{50}$ opérations !).

Reste à comprendre pourquoi l'opération (1) ne demande pas plus de deux millions d'opérations. Pour cela, il nous faut (clairement. . .) revenir un moment à l'arithmétique modulaire.

5.4 Exponentiation rapide ("square-and-multiply")

Nous avons vu au début de ce chapitre que "prendre des modulos" permet de simplifier de additions et multiplications, mais c'est encore plus vrai lorsqu'on effectue une opération d'*exponentiation* (qui commute également avec l'opération modulo) : pour calculer $A^B \pmod N$, on peut bien sûr d'abord calculer A^B, puis prendre la reste de la division de ce nombre par N pour trouver le résultat, mais il est clairement plus facile d'utiliser le fait que

$$A^B \pmod N = (A \pmod N)^B \pmod N$$

Voyons ça sur un exemple avec $N = 11$, $A = 64$ et $B = 6$: pour calculer $64^6 \pmod{11}$, calculons d'abord $64 \pmod{11} = 9$, puis

$$64^6 \pmod{11} = 9^6 \pmod{11} = 531'441 \pmod{11} = 9$$

? ? ? A vous entendre, vous ne semblez pas forcément convaincus… Certes, il est plus facile de calculer 9^6 que 64^6, mais ça reste quand-même un calcul ardu ! En fait, on peut faire mieux que ça. Regardez plutôt :

$$9^6 = 9^{4+2} = 9^4 \cdot 9^2 = (9^2)^2 \cdot 9^2 \quad \text{donc} \quad 9^6 \pmod{11} = (9^2 \pmod{11})^2 \pmod{11} \cdot (9^2 \pmod{11})$$

Pour effectuer ce calcul, on calcule d'abord $9^2 \pmod{11} = 81 \pmod{11} = 4$, puis $4^2 \pmod{11} = 16 \pmod{11} = 5$, et finalement la multiplication $(5 \cdot 4) \pmod{11} = 20 \pmod{11} = 9$, qui nous donne le résultat voulu : $64^6 \pmod{11} = 9$, sans avoir eu à effectuer de multiplications plus compliquées que des livrets appris à l'école primaire.

Remarque : Attention ! Pour calculer $A^B \pmod N$, on peut remplacer A par $A \pmod N$, mais on ne peut *pas* remplacer B par $B \pmod N$. Par exemple :

$$64^{17} \pmod{11} \neq 64^6 \pmod{11}$$

Voici la preuve (avec à nouveau un calcul qui ne demande pas de multiplications plus complexes que celles apprises à l'école primaire) :

$$\begin{aligned} 64^{17} \pmod{11} &= (64 \pmod{11})^{17} \pmod{11} = 9^{17} \pmod{11} \\ &= 9^{16+1} \pmod{11} = (((((9^2)^2)^2)^2) \cdot 9^1) \pmod{11} \end{aligned}$$

Calculons succesivement :

$$\begin{array}{cccc} & 9^2 \pmod{11} = 81 \pmod{11} = 4 & \to & 4^2 \pmod{11} = 16 \pmod{11} = 5 \\ \to & 5^2 \pmod{11} = 25 \pmod{11} = 3 & \to & 3^2 \pmod{11} = 9 \end{array}$$

donc nous obtenons finalement :

$$64^{17} \pmod{11} = (((((9^2)^2)^2)^2) \cdot 9^1) \pmod{11} = (9 \cdot 9) \pmod{11} = 81 \pmod{11} = 4$$

alors que nous avions obtenu précédemment $64^6 \pmod{11} = 9$.

En général, voyons maintenant comment cette méthode fonctionne : pour calculer A^B tout d'abord (en oubliant mod N pour l'instant), on utilise la décomposition binaire de B, c'est-à-dire qu'on écrit B comme une somme de puissances de 2. Par exemple, pour $B = 43$, écrivons $B = 43 = 32 + 8 + 2 + 1$, et donc

$$A^B = A^{32+8+2+1} = A^{32} \cdot A^8 \cdot A^2 \cdot A$$

Ainsi, en calculant successivement A, A^2, $A^4 = (A^2)^2$, $A^8 = (A^4)^2$, $A^{16} = (A^8)^2$ et $A^{32} = (A^{16})^2$, on n'effectue que des carrés, et pour obtenir le résultat A^B, il suffit de multiplier les quatre termes A, A^2, A^8 et A^{32}. Cette méthode, qui s'appelle en anglais "square-and-multiply", permet d'éviter le calcul direct de A^{43}, qui est fastidieux.

Viennent maintenant s'ajouter les modulos : comme déjà mentionné ci-dessus, l'avantage d'effectuer des opérations modulo permet de se cantonner aux nombres compris entre 0 et $N - 1$. Ainsi, si A^B est potentiellement un (très) grand nombre, $A^B \pmod N$ sera par contre toujours un nombre compris entre 0 et $N - 1$. Donc on peut refaire tout ce qu'on vient de faire en prenant chaque fois systématiquement le résultat modulo N. Ainsi, on sait qu'on n'effectuera que des carrés ou des multiplications avec des nombres compris entre 0 et $N - 1$.

Comptons maintenant le nombre d'opérations effectuées, en supposant que A, B, N sont tous des nombres à 100 chiffres :

- effectuer le carré d'un nombre à 100 chiffres ou la multiplication de deux nombres à 100 chiffres coûte $100 \cdot 100 = 10'000$ opérations environ (penser à la multiplication en colonnes) ;

- si B est un nombre à 100 chiffres, sa decómposition binaire sera également composée d'une centaine de termes (à peu près) ;

- et donc pour calculer la série A, A^2, A^4, A^8... jusqu'à $A^{2^{100}}$ (le tout modulo N), il faudra effectuer environ 100 carrés, et encore 100 autres multiplications (au pire) des termes dont on a besoin pour obtenir le résultat final A^B. Ceci représente au total

$$2 \cdot 100 \cdot 10'000 = 2 \text{ millions d'opérations}$$

comme annoncé plus haut ! Nous voilà donc arrivés à notre but, à savoir : effectuer un nombre raisonnable d'opérations pour trouver un (très) grand nombre premier.

5.5 Factoriser des grands nombres entiers

Maintenant que nous avons trouvé un algorithme efficace pour trouver des grands nombres premiers, il semble logique de passer à la prochaine étape et essayer de trouver un algorithme efficace pour exprimer un grand nombre en produit de facteurs premiers. En effet, c'est un fait établi qui que tout nombre entier peut toujours s'écrire comme un produit de facteurs premiers. Voici quelques exemples :

$98 = 2 \cdot 7 \cdot 7$

$99 = 3 \cdot 3 \cdot 11$

$100 = 2 \cdot 2 \cdot 5 \cdot 5$

$101 = 101$

$102 = 2 \cdot 3 \cdot 17$

etc.

Pour autant, existe-t-il un algorithme efficace permettant de calculer la factorisation d'un grand nombre N, par exemple un nombre à 100 chiffres ? La réponse honnête à cette question est : "A l'heure actuelle, on ne sait pas" ! Mais une chose est sûre : jusqu'à présent, on n'a pas trouvé d'algorithme efficace [3].

Donc pour aussi étonnant que cela puisse paraître, nous avons ici affaire à deux problèmes de même nature (le problème de la primalité et celui de la factorisation), qui sont en fait complètement différents, puisqu'une solution a été trouvée pour le premier, tandis que le second résiste encore... Cette différence fondamentale entre ces deux problèmes est même utilisée comme principe de base du système de cryptographie à clé publique RSA, que nous verrons prochainement.

3. Il existe en fait un algorithme efficace : l'algorithme de Shor, mais celui-ci requiert l'utilisation d'un ordinateur quantique... pas encore construit !

6 Cryptographie à clé publique

Dans cette section, nous allons voir le protocole d'échange de clé de Diffie-Hellman-Merkle : c'est un protocole astucieux qui permet à Alice et Bob d'échanger une clé secrète en ne faisant qu'échanger des messages qui peuvent être écoutés par tout le monde ! (pour ensuite utiliser cette clé pour communiquer avec un système de cryptographie à clé secrète, comme DES).

Bien sûr, il peut sembler a priori complètement impossible qu'Alice et Bob parviennent à se mettre d'accord sur une clé secrète s'ils ne peuvent qu'échanger des messages dits *publics*, à savoir des messages qu'Eve écoute en permamence. Et dans l'absolu, c'est vrai : c'est impossible...

Mais il se trouve que ça devient possible si on ajoute l'hypothèse qu'Eve n'a pas une puissance de calcul infinie, et qu'Alice et Bob disposent de ce qu'on appelle une *opération à sens unique*, c'est-à-dire une opération qu'il est facile d'effectuer (i.e., qu'il est possible d'effectuer en un temps raisonnable), mais très difficile d'inverser (par *très difficile*, on entend ici que le temps nécessaire pour effectuer cette inversion est beaucoup trop long). Voyons ceci en détail.

6.1 Le protocole d'échange de clé de Diffie-Hellman-Merkle : principe

En guise d'introduction, nous allons d'abord voir une version simplifée de ce protocole. Pour ce faire, nous allons supposer que l'opération à sens unique mentionnée ci-dessus est la multiplication. Vous objecterez qu'il est à l'heure actuelle très facile d'effectuer l'opération inverse, à savoir la division, ce qui est vrai ; mais on peut quand-même dire qu'il est en général plus difficile de diviser que de multiplier. Par extension, imaginons donc un monde où il soit *très difficile* d'effectuer des divisions.

1. Pour commencer, Alice et Bob se mettent d'accord sur un nombre entier commun M, qu'ils communiquent "en clair" à tout le monde (et donc en particulier à Eve, qui écoute tout).
2. Puis Alice choisit en secret un nombre entier A_1 et effectue la multiplication $A_2 = M \cdot A_1$; Bob fait de même de son côté en choisissant un nombre secret B_1 et en effectuant la multiplication $B_2 = M \cdot B_1$.
3. Alice communique ensuite le nombre A_2 à Bob, et Bob communique le nombre B_2 à Alice. Eve, qui a tout écouté (pour rappel, Alice et Bob ne peuvent rien communiquer secrètement), a donc maintenant entendu les valeurs des nombres M, A_2 et B_2 .
4. Puis Alice reprend son nombre secret A_1 et effectue, toujours en secret, la multiplication $A_3 = B_2 \cdot A_1$; de même, Bob effectue en secret la multiplication $B_3 = A_2 \cdot B_1$.

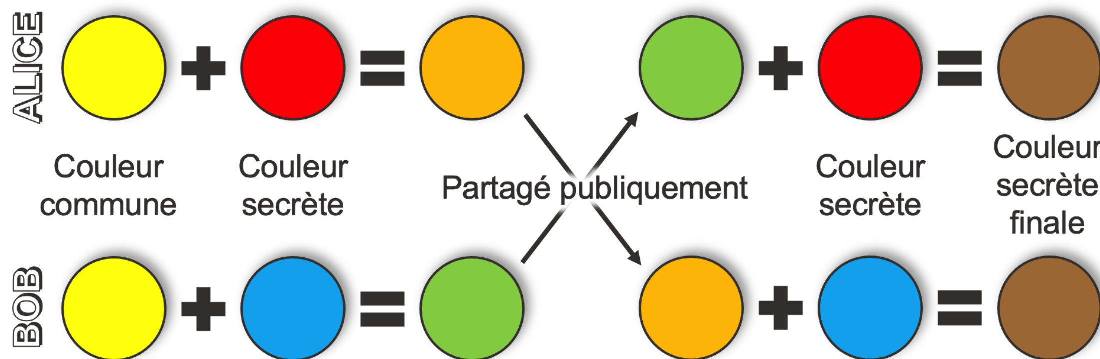
Que valent donc A_3 et B_3 ? $A_3 = B_2 \cdot A_1 = M \cdot B_1 \cdot A_1$ et $B_3 = A_2 \cdot B_1 = M \cdot A_1 \cdot B_1$. Vu que la multiplication est commutative, ces deux nombres sont égaux : $K = A_3 = B_3$ est donc la clé secrète partagée entre Alice et Bob.

Bien entendu, il faut encore vérfifier qu'Eve ne puisse pas trouver la clé K. Mais vous pouvez vérifier en effet qu'avec seulement les valeurs de M, A_2 et B_2 à disposition, Eve est incapable de trouver la clé K si elle ne sait pas effectuer des divisions (ou ne peut pas effectuer ces divisions

en un temps raisonnable, ce qui revient au même). K est donc bien le secret partagé d'Alice et Bob.

Finalement, il importe d'observer encore une chose : Alice et Bob vivent dans le même monde qu'Eve et ne savent donc pas diviser non plus, par hypothèse. Ceci veut dire que même s'ils ont pu se mettre d'accord sur une clé secrète commune K, ni l'un ni l'autre n'est capable de retrouver le nombre secret initial choisi par l'autre, même à la fin du protocole. Ainsi, Alice ne connaît pas B_1, ni Bob ne connaît A_1. Ces deux nombres restent *privés*, tandis que A_2 et B_2 sont eux *publics*. Et c'est la combinaison de A_1 et B_2 d'une part, et de B_1 et A_2 d'autre part, qui permet de trouver le secret commun K.

Une autre façon encore plus visuelle (voir la figure ci-dessous) d'envisager ce protocole est de penser à des pots de peinture de différentes couleurs (qu'il est facile de mélanger, mais très difficile de séparer). Alice et Bob se mettent d'accord au départ sur une couleur commune (disons le jaune), et choisissent chacun en secret une autre couleur (le rouge pour Alice, le bleu pour Bob). Alice mélange le jaune et le rouge, obtenant ainsi de l'orange, et transmet cette couleur à Bob ; Bob mélange de son côté le jaune et le bleu, obtenant ainsi du vert, et transmet cette couleur à Alice. Puis Bob mélange la couleur reçue d'Alice, l'orange, avec sa couleur secrète, le bleu, et obtient du brun. Alice de son côté mélange le vert reçu de Bob avec sa couleur secrète, le rouge, et obtient le même brun. Mais Eve, qui n'a vu passer que les pots de couleur jaune, orange et vert, n'est pas en mesure d'obtenir le même brun qu'Alice et Bob :



6.2 Fonctions à sens unique

Dans la suite, nous allons parler de *fonctions à sens unique*, plutôt que d'opérations à sens unique. *Grosso modo* [4], une fonction à sens unique est une fonction F telle que :

1. Pour toute valeur de A, il est facile de calculer $F(A)$.
2. Si on nous donne une valeur de B (faisant partie de l'image de F), il est alors très difficile de trouver une valeur de A telle que $F(A) = B$.

Cette deuxième ligne donne une condition plus stricte à respecter que la condition d'être "difficile à inverser". Voyez plutôt :

4. Notez bien qu'en théorie, on n'est jamais totalement sûr qu'une fonction soit vraiment à sens unique...

- La fonction F telle que $F(A) = 0$ pour toute valeur de A est la fonction la plus difficile à inverser qui soit ! Pour autant, ça n'est pas une fonction à sens unique, car il est facile de trouver une valeur de A telle que $F(A) = 0$ (en effet, n'importe quelle valeur de A fonctionne !)

- De même, pour un entier N donné, la fonction $F(A) = A \pmod N$ définie sur tous les nombres entiers A n'est pas à sens unique, car pour une valeur B entre comprise entre 0 et $N - 1$, il suffit de choisir $A = B$ pour trouver $F(A) = B$.

- Par contre, il se trouve que la *fonction de Rabin* $F(A) = A^2 \pmod N$ est une fonction à sens unique si $N = P \cdot Q$, avec P et Q des grands nombres premiers (et qu'on connaît seulement la valeur de N, et pas les facteurs P et Q). C'est sur cette fonction que se base le *cryptosystème de Rabin*, mais celui-ci reste un peu technique à étudier...

Dans la paragraphe suivant, vous découvrirez la fonction à sens unique utilisée dans le protocole de Diffie-Hellman-Merkle.

Pour faire le lien avec ce dont nous avons parlé il y a deux semaines, notez au passage que ces fonctions à sens unique sont non seulement intéressantes pour la cryptographie, mais aussi pour générer des suites de nombres aléatoires !

Le problème du "logarithme discret"

En arithmétique classique, si quelqu'un vous donne les valeurs de deux nombres A et C et vous dit que $A^B = C$, il est facile de calculer la valeur de B en utilisant la formule

$$B = \log_A(C)$$

Bon, il est vrai que calculer un logarithme à la main n'est pas si facile, mais toute calculatrice digne de ce nom vous permettra de faire ça rapidement.

En arithmétique modulaire, rien n'est moins vrai ! En effet, si on nous donne : N un grand nombre premier, A et C deux nombres compris entre 2 et $N - 1$, et qu'on nous dit que $A^B \pmod N = C$, il est alors très difficile de retrouver la valeur de B. On appelle ça le *problème du logarithme discret* (par analogie à la relation trouvée ci-dessus pour la valeur de B dans le cas continu).

Par contre, nous avons vu la dernière fois qu'il est possible d'effectuer rapidement l'exponentiation modulaire $A^B \pmod N$. De ces deux constatations, nous déduisons que la fonction F suivante est une fonction à sens unique :

$$F(B) = A^B \pmod N$$

où N est un grand nombre premier et $2 \leq A \leq N - 1$.

6.3 Le vrai protocole de Diffie-Hellman-Merkle

Nous allons maintenant tirer parti de cette fonction à sens unique pour permettre à Alice et Bob de se mettre d'accord sur un secret commun, en supposant qu'Eve n'a pas la puissance de calcul nécessaire pour résoudre le problème du logarithme discret (ce qui est somme toute une hypothèse très réaliste). Voici le protocole :

1. Alice et Bob se mettent d'accord sur la valeur d'un grand nombre premier N, ainsi que sur un autre nombre M compris entre 2 et $N-1$. Eve, qui écoute tout, connaît donc également les valeurs de M et N.
2. Alice choisit secrètement un nombre $2 \leq A_1 < N$ et effectue l'opération $A_2 = M^{A_1} \pmod N$. De son côté, Bob fait la même chose : il choisit $2 \leq B_1 < N$ et effectue $B_2 = M^{B_1} \pmod N$.
3. Alice communique la valeur de A_2 à Bob et Bob communique la valeur de B_2 à Alice. Eve, qui encore une fois écoute tout, apprend donc les valeurs de A_2 et B_2.
4. Alice calcule (de son côté) $A_3 = B_2^{A_1} \pmod N$ et Bob calcule (aussi de son côté) $B_3 = A_2^{B_1} \pmod N$.

Nous allons maintenant vérifier les deux affirmations suivantes :

1. $A_3 = B_3 = K$: Alice et Bob ont donc réussi à se mettre d'accord sur un secret commun.
2. Avec les informations dont elle dispose (et en supposant qu'elle n'a pas une puissance de calcul infinie), Eve ne peut pas deviner la valeur de K, qui sera donc le secret commun d'Alice et de Bob.

En effet, on voit que

$$
\begin{aligned}
A_3 = B_2^{A_1} \pmod N = (M^{B_1} \pmod N)^{A_1} \pmod N = (M^{B_1})^{A_1} \pmod N = M^{B_1 \cdot A_1} \pmod N \\
B_3 = A_2^{B_1} \pmod N = (M^{A_1} \pmod N)^{B_1} \pmod N = (M^{A_1})^{B_1} \pmod N = M^{A_1 \cdot B_1} \pmod N
\end{aligned}
$$

ce qui confirme le premier point : les valeurs de A_3 et B_3 sont égales (car $B_1 \cdot A_1 = A_1 \cdot B_1$). Quelles sont donc maintenant les informations dont dispose Eve ? Elle connaît :

$$
N, \quad M, \quad A_2 = M^{A_1} \pmod N \quad \text{ainsi que} \quad B_2 = M^{B_1} \pmod N
$$

A moins d'être capable de résoudre le problème du logarithme discret, Eve ne peut donc pas retrouver les valeurs de A_1 ou B_1, ni pas conséquent la valeur de $A_3 = B_3 = K$: Alice et Bob ont donc bien trouvé un secret commun K, qu'ils peuvent utiliser pour communiquer secrètement grâce au protocole DES, par exemple (rappelez-vous que vu que K est un grand nombre, sa représentation binaire est une longue suite de bits, certes pas complètement aléatoire, mais c'est un moindre défaut).

A noter que, comme dans l'exemple avec les multiplications et les divisions, les nombres A_1 et B_1 restent secrets tout au long du protocole. Alice et Bob n'ont pas besoin de révéler ces nombres respectifs à leur partenaire ; le protocole fonctionne sans ça.

6.4 Défauts du système

Pour finir, mentionnons que ce système a un défaut principal : c'est justement celui d'être d'un protocole d'échange de clé, qui nécessite après coup l'utilisation d'un système de cryptographie à clé secrète pour échanger des messages. La semaine prochaine, nous verrons d'autres protocoles de cryptographie à clé publique, qui permettent d'échanger directement des messages de manière confidentielle.

L'autre défaut de ce système est celui déjà mentionné plus haut : il n'existe aucune garantie théorique que le problème du logarithme discret soit un problème *vraiment* difficile à résoudre. Qui sait, peut-être que dans quelques années, quelqu'un trouvera un algorithme efficace pour résoudre ce problème ? Ceci remettrait en question bien des choses... En prévision de cela, certaines personnes s'intéressent à de nouveaux systèmes cryptographiques, comme celui évoqué dans le paragraphe suivant.

6.5 Le même protocole avec des courbes elliptiques !

En général, lorsqu'on entend parler de "courbes elliptiques", on pense tout de suite à quelque chose de substantiellement complexe, donc difficile à approcher (par exemple, les courbes elliptiques apparaissent dans la démonstration du grand théorème de Fermat évoqué la semaine dernière). Il se trouve cependant que celles-ci peuvent être utilisées pour une version modifiée du protocole d'échange de clé de Diffie-Hellman-Merkle, et que cette utilisation est très bien expliquée dans l'excellent blog de Fang-Pen Lin (en anglais) :

`https://fangpenlin.com/posts/2019/10/07/elliptic-curve-cryptography-explained/`

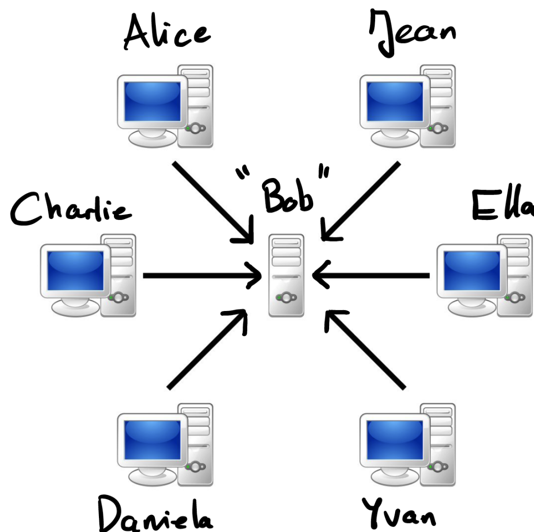
Nous éviterons de bêtement paraphraser ce blog dans le présent chapitre, mais nous passerons à travers les idées principales de celui-ci pendant le cours.

7 Cryptographie à clé publique : suite

Dans cette dernière section, nous allons voir un nouveau type de cryptographie à clé publique, où le but n'est pas de partager une clé commune pour ensuite échanger des messages chiffrés, mais de directement communiquer un message chiffré ! Nous allons voir comment cette magie est possible avec un des protocoles les plus célèbres de cryptographie à clé publique : le *protocole de Rivest-Shamir-Adleman* (RSA). Puis nous verrons comment on peut avec le même principe certifier un document avec une *signature digitale* (DSA).

7.1 Protocole de Rivest-Shamir-Adleman (RSA)

Le scénario auquel s'applique ce protocole est un peu différent de celui où deux personnes essayent de communiquer secrètement des messages entre elles. Il faut plutôt penser à Alice comme une utilisatrice et à Bob comme un serveur central, auquel s'adressent de nombreuses personnes (typiquement, un site web auquel Alice désire se connecter). La contrainte maintenant est d'établir une(des) connection(s) sécurisée(s) sans qu'il soit nécessaire pour Bob de faire autre chose que de publier une clé qu'Alice et d'autres personnes puissent utiliser pour chiffrer directement leurs messages :



Encore un peu d'arithmétique modulaire !

Dans ce chapitre, nous aurons besoin d'un nouvel élément d'arithmétique modulaire. Rappelez-vous le petit théorème de Fermat :

$$\text{Si } N \text{ est un nombre premier et } 1 \leq A \leq N-1, \text{ alors } A^{N-1} \pmod{N} = 1.$$

Il existe une version plus générale, le *thèorème d'Euler*[5], donc l'énoncé est le suivant :

5. A noter qu'il existe par ailleurs une démonstration courte et très élégante de ce théorème si on connaît des choses sur les groupes. Voir `https://en.wikipedia.org/wiki/Euler's_theorem`

Si N est un nombre entier et $1 \leq A \leq N-1$ avec $\text{PGDC}(A, N) = 1$, alors $A^{\phi(N)} \pmod{N} = 1$,

où ϕ est la *fonction d'Euler*. Cette fonction est définie ainsi :

$\phi(N)$ est le nombre de nombres entiers $1 \leq A \leq N-1$ tels que $\text{PGDC}(A, N) = 1$.

En particulier, on trouve que :

- si N est premier, alors $\phi(N) = N-1$, car dans ce cas, *tous* les nombres A compris entre 1 et $N-1$ vérifient $\text{PGDC}(A, N) = 1$: on retrouve ici le petit théorème de Fermat.

- si $N = P \cdot Q$, avec P et Q premiers, vous pouvez vérifier que $\phi(N) = (P-1) \cdot (Q-1)$. Par exemple, si $N = 10 = 2 \cdot 5$, alors $\phi(N) = 1 \cdot 4 = 4$; en effet, les nombres A compris entre 1 et 9 tels que $\text{PGDC}(A, 10) = 1$ sont $A = 1, 3, 7, 9$.

Remarque importante : Il est difficile de calculer la valeur de $\phi(N)$ si on ne connaît pas la décomposition de N en produit de facteurs premiers. Si par contre on la connaît, alors le calcul devient facile (il suffit d'effectuer le produit $(P-1) \cdot (Q-1)$ dans l'exemple précédent).

Revenons maintenant au protocole RSA. Voici comment celui-ci fonctionne :

1. Bob génère tout d'abord deux grands nombres premiers P et Q (distincts), et calcule $N = P \cdot Q$. Il publie ce nombre N (mais garde les nombres premiers P et Q secrets).
2. Il choisit ensuite un nombre C entre 2 et $N-1$ tel que $\text{PGDC}(C, \phi(N)) = 1$. Bob publie également ce nombre C.
3. Il calcule alors le nombre D tel que $C \cdot D = 1 \pmod{\phi(N)}$: ce nombre D est donc l'inverse de C modulo $\phi(N)$. On peut montrer que cette opération est facile à effectuer, si on connaît la valeur de $\phi(N)$, bien sûr. Bob garde ce nombre D secret.
4. Puis vient l'étape du *chiffrement* : pour envoyer un message X compris entre 2 et $N-1$ à Bob, Alice calcule d'abord $Y = X^C \pmod{N}$ et envoie Y.
5. Et finalement vient l'étape du *déchiffrement* : Bob, pour retrouver le message envoyé par Alice, effectue l'opération $Y^D \pmod{N}$ et retrouve ainsi X, comme nous allons le voir.

Vérifions que tout ceci fonctionne bien :

1. Quel est le message reçu par Bob ? Calculons :

$$Y^D \pmod N = (X^C \pmod N))^D \pmod N = X^{C \cdot D} \pmod N = X\,?$$

Par construction, $C \cdot D = 1 \pmod{\phi(N)}$; ceci veut dire que $C \cdot D = 1 + K \cdot \phi(N)$ pour un nombre entier K et donc

$$Y^D \pmod N = X^{1+K\cdot\phi(N)} \pmod N$$

- Si $\text{PGDC}(X, N) = 1$, alors c'est aussi vrai que $\text{PGDC}(X^K, N) = 1$, et donc

$$X^{K\cdot\phi(N)} \pmod N = (X^K)^{\phi(N)} \pmod N = 1$$

par le théorème d'Euler, ce qui implique que $Y^D \pmod N = X$, car $2 \leq X \leq N - 1$.

- Si $\text{PGDC}(X, N) \neq 1$, alors une autre démonstration est nécessaire, mais notez que ce cas de figure arrive seulement dans le cas où X est un multiple de P ou de Q, donc très rarement.

2. Et que peut faire maintenant Eve, qui connaît les valeurs de N, C et Y ? Pour retrouver X, elle devrait d'abord retrouver la valeur du nombre D (afin d'effectuer ensuite le même calcul que Bob). Mais pour cela, elle a besoin de connaître le produit $\phi(N) = (P - 1)(Q - 1)$, or elle ne connaît que le nombre $N = P \cdot Q$. Et c'est là que vient s'ajouter la dernière pièce du puzzle, déjà mentionnée ci-dessus : retrouver les nombres P et Q à partir de N revient à *factoriser* ce nombre. Or il se trouve que cette opération est a priori également une opération difficile à réaliser (du moins, personne n'a trouvé jusqu'à maintenant d'algorithme efficace pour résoudre ce problème).

A propos de factorisation...

Pour finir, mentionnons que la sécurité du protocole RSA est compromise à l'heure actuelle par le développement des *ordinateurs quantiques*, qui pourraient bien un jour permettre d'effectuer efficacement des factorisations de grands nombres entiers, grâce à l'*algorithme de Shor*[6]. Lorsque les premiers ordinateurs quantiques ont vu le jour au début des années 2'000, ceux-ci pouvaient essentiellement effectuer des factorisations du type $15 = 3 \cdot 5$, et tout le monde a rigolé... Mais il faut bien avouer qu'aujourd'hui, soit vingt ans plus tard, plus personne ne rigole, car même s'il subsiste des incertitudes, il se pourrait bien que les prochaines années voient fleurir des choses très intéressantes de ce côté-là ! A tel point que le sujet de la *cryptographie post-quantique*, celle qui résisterait aux ordinateurs quantiques, est maintenant tout à fait sérieusement étudiée dans le monde académique[7].

6. Un sujet passionnant, mais qui mériterait rien qu'à lui un cours complet...

7. Signalons ici qu'il existe aussi une *cryptographie quantique*, qui utilise les propriétés intrinsèques des particules de lumière (les photons) pour chiffrer des messages.

7.2 Signature digitale (DSA)

Le but de cette seconde partie est différent de la première, mais utilise le même principe. L'idée est de proposer à Alice un moyen digital pour *signer* un message, afin que Bob puisse être sûr que le message en question provient bien d'Alice et pas de quelqu'un d'autre (Eve, par exemple). Faites attention qu'on oublie donc ici l'idée de transmettre *secrètement* un message !

Avant de présenter le protocole de signature digitale, nous avons besoin d'introduire un nouvel élément : les *fonctions de hachage*.

Fonctions de hachage

A l'origine, une fonction de hachage H a pour but de faire correspondre à un message donné X une *valeur de hachage* $H(X)$ qui satisfasse les propriétés suivantes :

1. $H(X)$ doit être déterministe (i.e., non aléatoire) ;
2. $H(X)$ doit occuper (significativement) moins de place en mémoire que X lui-même ;
3. si X et Y diffèrent légèrement, alors $H(X)$ et $H(Y)$ doivent différer grandement.

Exemple : En identifiant les lettres majuscules à des nombres entre compris 0 et 25, comme nous l'avons déjà fait auparavant, nous pouvons calculer pour un message composé de n lettres :

a) la somme des n lettres modulo 26

b) la somme des n lettres, multipliée chacune par sa position dans le message, modulo 26

et obtenir ainsi une valeur de hachage composée des deux lettres résultantes. Voici un exemple : si le message est BONJOUR, alors

a) $B + O + N + J + O + U + R \pmod{26}$
$= 1 + 14 + 13 + 9 + 14 + 20 + 17 \pmod{26} = 88 \pmod{26} = 10 = K$

b) $1 \cdot B + 2 \cdot O + 3 \cdot N + 4 \cdot J + 5 \cdot O + 6 \cdot U + 7 \cdot R \pmod{26}$
$= 1 \cdot 1 + 2 \cdot 14 + 3 \cdot 13 + 4 \cdot 9 + 5 \cdot 14 + 6 \cdot 20 + 7 \cdot 17 \pmod{26}$
$= 1 + 28 + 39 + 36 + 70 + 120 + 119 \pmod{26} = 413 \pmod{26} = 23 = X$

Donc la valeur de hachage du message BONJOUR est KX. Tandis que vous pouvez vérifier que la valeur de hachage du message BONKOUR est LB : une petite modification dans le message d'origine mène bien à une valeur de hachage substantiellement différente.

Il est possible de faire de nombreux usages des fonctions de hachage :

- Celles-ci peuvent être utilisées pour vérifier qu'un message transmis ne contient pas d'erreur : si la valeur de hachage transmise ne correspond pas à celle attendue, on demande à l'expéditeur de retransmettre le message.

- Elle peuvent aussi être utilisées pour classer des grands jeux de données en économisant de la place. Au lieu d'enregistrer les données telles quelles, on enregistre leurs valeurs de hachage dans une *table de hachage* qui occupe moins de place en mémoire.

- Finalement, elles peuvent être utilisées à des fins cryptographiques, comme nous allons le voir.

Signature digitale

Revenons au problème évoqué plus haut : Alice veut transmettre un message à Bob et aussi lui donner un moyen de vérifier que c'est bien elle qui est l'auteure de ce message.

Une solution naïve serait la suivante : Alice transmet le message X à Bob, et calcule, avec une fonction de hachage H donnée, une valeur de hachage $H(X)$, qu'elle utilise comme signature du message. Cependant, ce système a le grave défaut de supposer qu'Eve ne connaît pas la fonction de hachage H ; il ne respecte donc pas le principe de Kerkhoffs vu au début de ce cours, qui dit qu'il faut toujours supposer qu'Eve connaît le système utilisé. Sous cette hypothèse, rien n'empêcherait Eve d'envoyer un message Y à Bob, et d'y apposer sa signature $H(Y)$ pour faire croire à Bob que ce message vient d'Alice. Il faut donc trouver autre chose.

Pour ce faire, nous allons réutiliser les idées du protocole RSA, mais à l'envers, en quelque sorte. Voici comment procéder :

1. *Alice* génère d'abord deux grands nombres premiers P et Q (distincts) et calcule $N = P \cdot Q$, qu'elle publie.
2. Puis elle choisit un autre nombre C entre 2 et $N-1$ tel que $\text{PGDC}(C, \phi(N)) = 1$, qu'elle garde *secret*.
3. Elle calcule ensuite le nombre D tel que $C \cdot D \pmod{\phi(N)} = 1$ et *publie ce nombre*.
4. C'est ici qu'intervient la fonction de hachage H, publique elle aussi, dont on supposera que l'image est l'ensemble des nombres allant de 0 à $N-1$. Pour envoyer le message X, Alice calcule $H(X)$, puis $S = H(X)^C \pmod{N}$, et envoie finalement X et S à Bob. S est maintenant la vraie signature du message, qu'Eve ne peut pas contrefaire, comme nous allons le voir ci-dessous.
5. Pour vérifier que le message provient bien d'Alice, Bob calcule $H(X)$ et $S^D \pmod{N}$. Si ces deux nombres sont égaux, Bob a alors la garantie que le message vient bien d'Alice.

A nouveau, vérifions que tout ceci fonctionne bien :

1. Si le message X a bien été envoyé par Alice et que celle-ci a calculé la valeur de la signature S comme décrit ci-dessus, alors Bob trouve effectivement que $S^D \pmod{N} = H(X)^{C \cdot D} \pmod{N} = H(X)$ par le théorème d'Euler et le fait que $C \cdot D \pmod{\phi(N)} = 1$.
2. Pour contrefaire la signature d'Alice, Eve devrait connaître la valeur de C, mais elle ne connaît que N, D, X, S et la fonction H. Or elle ne sait pas factoriser N (ce qui lui permettrait de calculer $\phi(N)$, et donc de retrouver la valeur de C), ni résoudre le problème du logarithme discret $S = H(X)^C \pmod{N}$ (ce qui lui permettrait également de retrouver la valeur de C). Elle ne peut donc pas utiliser ce nombre C pour signer son propre message Y et ainsi faire croire à Bob que celui-ci provient d'Alice.

Remarque : Les nombres N et D publiés par Alice doivent être des nombres dont on est sûr a priori qu'ils proviennent d'Alice ! De façon pratique, on peut penser à ces nombres comme un certificat qui établit une fois pour toutes l'identité d'Alice. Ensuite, elle utilise ce même certificat pour envoyer un ou plusieurs messages $X_1, X_2, X_3, \ldots$ et y apposer à chaque fois la signature digitale correspondante $S_1, S_2, S_3 \ldots$