Deciphering the Mysteries

SoS 2023(Cryptography)

Arya Vishe
21D070018, IIT-B

Mentor- Nilabha Saha

20/06/2023

Contents

Author's Note

Hello! Here are a few things which I would like you, the reader, to know before you start with the report-

- This report is written as a beginner's guide to Cryptography. I have tried to ensure that the initial learning curve is gentler. In doing so, a bit of mathematical rigour and some technical aspects are glossed over. In a lot of places, informal language was also used to break up some breathing space in between all of the jargon.
- [1] was a heavy influence in writing this report. I have mostly stuck to the format of the reference textbook in the later sections of this report but if time permits, I would like to explore some more topics outside of the actual book.
- The colour "teal" has exclusively been used for hyperlinks to ensure that the reader is able to read the further sections without having to skim through every concept discussed before that. (Only the table of contents has hidden hyperlinks)
- It is highly recommended that the reader goes through the first chapter (*Mathematical Base to Cryptography*) before continuing to other sections. But it is understandable that s/he might get bored by just theoritical knowledge without seeing any link to cryptography so they can free go to any of the further sections. Wherever possible, I have added hyperlinks, in the later chapters, to the earlier concepts to ensure that the first chapter can act as a look-up table for readers.

Acknowledgements

This work was written as a part of the Summer Of Science, 2023 by the MnP Club, IIT Bombay. I read in its entirety [6] to get a surface level introduction to the topic and then mostly followed the reference book [1] and referred to online resoruces like [2] for a deeper dive.

Some of the sections in the report are heavily inspired by the above resources and at several points might simply be a paraphrasing of the reference books. Nevertheless, I have tried to compress the contents of these sources wherever needed, explored some minor topics on my own to a deeper depth, and tried my best to break down too much of difficult mathematics into easier chunks while still preserving sufficient depth.

I would like to acknowledge the help of my mentor, Nilabha Saha, for his help and support.

He was extremely helpful and friendly and his enthusiasm for the topic definitely rubbed off on me and gave me enough encouragement to devote the hours I have put into this report. He cleared my doubts at every stage at the latest and that allowed me to get through some of the most difficult sections of this report.

Some topics in the end had to be skipped as they were only explored partially and so I opted to remove them completely in order not to ruin the structure of this report.

Mathematical Base to Cryptography

Introduction to Cryptography

1.1.1 Some terminologies

- *Plaintext,* The message, usually alphabetic in this context, that you wish to send. Depicted using small letters a, b, c, $\cdots$
- *Ciphertext,* The message that will be transmitted after enciphering. Depicted using captial letters A, B, C, $\cdots$
- *Key,* The word/ phrase/ alphabetic order using which we encrypted our *Plaintext.*
- *Monoalphabetic Substitution Cipher,* A cipher key which uses only one set of alphabet to encrypt the *Plaintext*

- *Polyalphabetic Substitution Cipher,* A key that consists of multiple alphabetic orders/ keywords to encrypt the message.
- *Alice, Bob and Eve,* The quintessential trio studied in Cryptology examples wherein *Alice* and *Bob*, are trying to communicate secretly which *Eve* tries to intercept and decrypt the messages.

1.1.2 Early Cryptography

The earliest forms of monoalphabetic[1] ciphers include the *Caesarean/Shift Cipher*. This involves shifting of the alphabet by a few (historically 3) letter to encrypt the *Plain text.*
This yields us with only 26 possible cipher keys which can be easily cycled through by Eve, once she is sure that Caeser Cipher has been used, hence providing us with next none security.

Of course, there exist a very large number of random monoalphabetic cipher keys ($26! \approx 4 \times 10^{26}$ to be precise) but monoalphabetic ciphers can be easily broken into using statistical analysis of the cipher text. Tools employed by cryptanalysts include-

- Frequency of letters
- Frequency of Bigrams (pairs of letters that occur together)
- Presence of vowels

Another classical cipher, which used polyalphabetic cipher keys, *Vigenère* cipher will be analysed in later weeks.

Some Mathematical Pre-requisites for Modular Arithematic

1.2.1 GCD

Definition. GCD *of any 2 integers is the largest positive integer that divides each of the integers, provided both integers aren't 0.*

The standard procedure to find $\gcd(a, b)$ is to list out factors of a and b, and choose the largest factor common to both. But an even faster algorithm exists called the *Euclid's algorithm.* It uses the fact that $\gcd(a, b) = \gcd(a - k \cdot b, b), \forall a, b, k \in \mathbb{Z}$ such that $a > b$[a]

Thus to find $\gcd(A, B)$, we can use this algorithm

[1] Consisting of only 1 key

1. Take $a = A$, $b = B$
2. If $b > a$, swap them.
3. $\gcd(a, b) = \gcd(a - k \cdot b, b)$ so replace a with $a - k \cdot b$ such that $(k + 1) \cdot b > a$ and $k \cdot b < a$
4. If $a = 0$, $\gcd(A, B) = b$
   else swap a and b, and repeat from Step 3.

Some interesting results-

- $r_{n+2} < \frac{r_n}{2}$ where r_n is the n^{th} remainder[b].
- it takes at most $2 \log_2 b + 2$ steps[c].
- We can prove that $\gcd(a, b) = u \cdot a + v \cdot b$ where $u, v \in \mathbb{Z}$. These u, v values can be determined using a simple algorithm if we know the quotients at each step.
  We can even find the original a and b if we have all the quotients.
- The above statement also means that if $\gcd(a, b) = 1$, we can write any number using linear combinations of a and b.

[a]Proof: Suppose you have 2 numbers a, b such that $\gcd(a, b) = g$ thus we can write $a = p \cdot g$ and $b = q \cdot g$. Obviously then, if we defined $c = a - b = (p - q) \cdot g$, will have g as a divisor.
Futhermore, let us denote $\gcd(b, c) = h$. Thus we can also write $b = m \cdot h$ and $c = n \cdot h$. Upon rearranging, we get $a = b + c = (m + n) \cdot h$ which tells us clearly that h will have to be a divisor of a.
Now, at this point, all of a, b and c have both g and h as factors. So the only way $\gcd(a, b)$ does not exceed the initially set g is because $g = h$. Tada!

[b]Proof- Assuming 2 cases (1. $r_{n+1} < \frac{r_n}{2}$ is obvious,2. $r_{n+1} > \frac{r_n}{2}$ ensures that $r_n = 1.r_{n+1} + r_{n+2} \Rightarrow r_{n+2} = r_n - r_{n+1} < \frac{r_n}{2}$)

[c]We can use the fact that if $b \in [2^{n-1}, 2^n)$ and $r_{2k+1} < \frac{b}{2^k}$, we can easily see that this is true since $2n + 1 = 2(n - 1) + 2 < 2 \log_2 b + 2$

Modular Arithematic

1.3.1 Format

$$a \equiv b \bmod (m) \text{ where } a - b \text{ is divisible by } m$$

1.3.2 Properties

- If $a \cdot b \equiv 1 \bmod (m)$ $\exists b \in \mathbb{Z} \iff \gcd(a, m) = 1$, we say that b is the *modular inverse* of a modulo m.
  Note- *Although multiple values of b will exist if* $\gcd(a, m) = 1$*, all will have the same value of* $b \bmod (m)$*. Hence we used the term* ***the inverse.***
- If $a_1 \equiv b_1 \bmod (m)$ and $a_2 \equiv b_2 \bmod (m)$, then, $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \bmod (m)$

- We can denote *inverse* of a as a^{-1} such that, $a^{-1} \equiv b \bmod (m)$.
  Hence $7^{-1} \equiv 8 \bmod (11)$. This means if we can write $\frac{4}{7} \equiv 7 \cdot 8 \bmod (11) \equiv 1 \bmod (11)$ as $4 \equiv 7 \bmod (11)$.
  The interesting thing to note here is that it isn't easy to find the modular inverses of numbers as m becomes larger.
- Since we can write $a \cdot b \equiv 1 \bmod (m)$ as $a \cdot b = q \cdot m + 1$, from the result obtained in last section, we can be sure that finding a value of b will take no. of steps $\sim \log_2 m$.

You might think that finding the modulo of products of large numbers might take the same amount of time as it takes for calculating the product but you can easily cut down time on it.
Example- $170242 \times 261494 \bmod (10) \equiv 2 \times 4 \equiv 8 \bmod (10)$. Hence it is ***FAST***.

1.3.3 Ring theory

As we can see in the above definitions, it is obvious that in $a \equiv r \bmod (m)$, $r \in [0, m-1] \cap \mathbb{Z}$. We can then say that $r \in \mathbb{Z}/m\mathbb{Z}$ where $\mathbb{Z}/m\mathbb{Z}$ is the ring of integers with modulo m.

For addition and multiplication in $\mathbb{Z}/m\mathbb{Z}$ space, we just divide the answer with m and use the remainder as final value.
Units modulo m- If $\gcd(a, m) = 1$, OR a^{-1} exists for modulo m, we say that a is units modulo m.

- If a_1, a_2 are units modulo m, $a_1 \cdot a_2$ will also be a units modulo m.
- If a_1, a_2 are units modulo m, $a_1 + a_2$ need not be a units modulo m.

The set $*(\mathbb{Z}/m\mathbb{Z})$ is called the group of units modulo m and can also be defined as-
$*(\mathbb{Z}/m\mathbb{Z}) = a \in \mathbb{Z}/m\mathbb{Z}, \gcd(a, m) = 1$ If we took a subset of units modulo $m = \mathbf{P}$, then-

- If $k_{i,j} = a_i \cdot a_j$, where $a_i, a_j \in \mathbf{P}$, then $k_{i,j}$ also belongs to $\mathbf{P}$.
- If $k_{i,j} = a_i + a_j$, where $a_i, a_j \in \mathbf{P}$, then $k_{i,j}$ need not belong to $\mathbf{P}$.

Euler's Totient/ Phi Function- $\phi(m) = \# * (\mathbb{Z}/m\mathbb{Z})$ OR $\phi(m)$ counts the number of $a \in [1, m-1]$, such that, $\gcd(a, m) = 1$ i.e. number of values of $a < m$ such that a and m are co-prime
e.g. $\phi(24) = 8, \phi(7) = 6$
Note- Notice how $\phi(p)$ where p is a prime number, will always be $p - 1$

1.3.4 Shift Ciphers and Modulo Arithematic

We can assign numbers from 0 through 25 to all the letters in the alphabet and then use modulo arithematic for encryption and decryption-

- For a shift of k, we can encrypt any letter whose number is denotes by p to a letter $(p + k)$ mod (26)
- To decrypt the cipher text (p'), you subtract in the ring via $(p' - k)$ mod (26).

1.3.5 Fast Powering algorithm

To calculate g^A mod (m), where $g, A \in \mathbb{N}$, it can take $\sim 2^n)$ multiplications if $A \in [2^n, 2^{(n+1)})$. We can, however, use the fact that g can we written as a linear combination of powers of 2 to reduce the number of steps we take.

- Express $A = \sum_{n=0}^{k} a_r 2^r$ where $2^k \leq A < 2^{(k+1)}$
- Let $g_r = g^{2^r}$. We can find g_1 using $g \cdot g$ and for subsequent values of r, we can just use $g_{r+1} = g_r \cdot g_r$
- Now we have $g^A = \sum_{r=0}^{k} g^{a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 + \cdots} = \sum_{r=0}^{k} g_0^{a_0} + g_1^{a_1} + \cdots$
- x^2 mod $(N) = (x$ mod $(N))^2$ mod (N)

Keep in mind that a_r can only take the values 0 and 1 thus we can say that this method takes $2r$ multiplication steps at most. (Including the exponential expansion of A).

Prime numbers and Finite Fields

1.4.1 Division modulo m

We can usually do multiplication, subtraction, addition modulo m but division modulo m is only possible if the divisior(k) is coprime with m.

Divison modulo m means- If $a = p$ mod (m) *and* $b = q$ mod (m),

$$c = \frac{a}{b} = \frac{p'}{q'}$$

where p mod $(m) = p'$ mod (m) and q mod $(m) = q'$ mod (m) and $\frac{p'}{q'} \in \mathbb{Z}$. It is only possible if $\gcd(b, m) = 1$ as otherwise you cannot get a unique value.

Example- $a = 11 \bmod (5)$, $b = 4 \bmod (5)$ then $\frac{a}{b} = \frac{11 \bmod (5)}{4 \bmod (5)} = \frac{16 \bmod (5)}{4 \bmod (5)} = 4$
Note- In the above example, 4 is the only integral value possible of $\frac{a}{b}$. Values such as $\frac{11 \bmod (5)}{4 \bmod (5)} = \frac{1}{4}$ are ignored.

The reason for $\gcd(4, 5) = 1$ being necessary in the above examples is that we can write $\frac{11 \bmod (5)}{4 \bmod (5)} = 11 \cdot 4^{-1} \bmod (5)$ and 4^{-1} can only exist if $\gcd(4, 5) = 1$. (Here $4^{-1} = 4$ thus our answer $= 44 \bmod (5) = 4$)

1.4.2 Division in rings

As seen earlier, if $\gcd(a, m) = 1$, we can easily divide modulo m by a any integer. If we now studied rings of form $\mathbb{Z}/p\mathbb{Z}$, where $p \in prime$ we will get that any integer $b \in \mathbb{Z}/p\mathbb{Z}$ can be used in division as obviously $b < p$ AND $p \in prime$ thus $\gcd(b, p) = 1$

Note- If $a \in *\mathbb{Z}/m\mathbb{Z}$, a can be used as a divisor modulo m. ALSO $\mathbb{Z}/p\mathbb{Z} = *\mathbb{Z}/p\mathbb{Z}$.
We can calculate a^{-1} *using-* $au + pv = 1$ *(as* $\gcd(a, p) = 1$*), wherein solving for* u *yields us* $a^{-1} \bmod (p)$.

The last line is the **Extended Euclidean Theorem**. Unlike the normal Euclid's theorem, the easiest way to describe its working is via a recursive C code.

1.4.3 Finite Fields

Since we can compute division, addition, multiplication and subtraction in $\mathbb{Z}/p\mathbb{Z}$, we can call it a field just like $\mathbb{R}/\mathbb{Z}/$ etc$\cdots$
Since $\mathbb{Z}/p\mathbb{Z}$ only has a finite number of elements, we call it a finite field and can even denote it by $(F)_p$ to show the finitedness.

Fermat's "Little" Theorem

The theorem states that-

$$a^{p-1} \equiv 1 \bmod (p) \text{ for } a \neq k \cdot p$$

Proof:
Since $a \neq k \cdot p$, $a, 2a, 3a, 4a, \cdots (p-1)a \bmod (p) \neq 0$. Now if we considered $k \in [1, p-1]$,

$$k_1 \cdot a \equiv j_1 \bmod (p) \text{ AND } k_2 \cdot a \equiv j_2 \bmod (p)$$

$$\Rightarrow (k_1 - k_2) \cdot a \equiv (j_1 - j_2) \bmod (p)$$

Since $k_1 - k_2$ cannot be a multiple of p, $j_1 - j_2 \neq 0$ and so for every $k \in [1, p-1]$, we get a distinct $j \in [1, p-1]$. Hence we get a bijective function of $[1, p-1] \to [1, p-1]$. So-

$$\prod_{k=1}^{p-1} k \cdot a \equiv \prod_{k=1}^{p-1} k \bmod (p) \Rightarrow (p-1)! a^{p-1} \equiv (p-1)! \bmod (p)$$

$$\Rightarrow a^{p-1} = 1 \bmod (p)$$

We can use this to find the inverse of any number modulo p-

$$a \cdot a^{p-2} \equiv 1 \bmod (p) \Rightarrow a^{-1} \equiv a^{p-2} \bmod (p)$$

Voila!

We just found an inutitive, new way to check if a number is composite or not

If we wanted to check if a number n was prime or not-

- If $n \in even$, obviously composite
- If $n \in odd$, take $a = 2$. Now if $2^{n-1} \not\equiv 1 \bmod (n)$, we can be sure that it is definitely composite. [2]

Definition. We define *order* as the smallest number k such that

$$a^k \equiv 1 \bmod (p)$$

We can also be sure that $p - 1$ is divisible by k.

1.5.1 Primitive Roots Theorem

The theorem states that *for a prime number p, there always exists a number $g \in \mathbb{F}_p$, such that every element of $\mathbb{F}_p^*$ can be expressed as a power of g*

Number of primitive roots of $p = \phi(p-1)$

[2] As is with all good things in life, this too has an asterisk- you cannot judge if a number is prime or not solely based on this. There is a subset of composite numbers called the "Carmicheal Numbers" which satisfy $b^{(n-1)} \equiv 1 \bmod (n)$ despite their lack of primality. Just random gifts of joy Mathematics bestows upon you. It is better discussed in a later section.

Example- Let us study $\mathbb{F}_7^*$. Since $7 \in$ primes, $\mathbb{F}_7^* \equiv \mathbb{F}_7$.
3 is considered a primitive root of this finite field so-

$$1, 3, 3^2, 3^3, 3^4, 3^5, 3^6 \bmod (7) \equiv 1, 3, 2, 6, 4, 5, 1$$

Even 5 is considered as so-

$$1, 5, 5^2, 5^3, 5^4, 5^5, 5^6 \bmod (7) \equiv 1, 5, 4, 6, 2, 3, 1$$

However 6 is not one-

$$1, 6, 6^2, 6^3, 6^4, 6^5, 6^6 \bmod (7) \equiv 1, 6, 1, 6, 1, 6, 1$$

(Note how despite 6 despite not being a primitive root, still satisfies $6^{p-1} \bmod (p) \equiv 1$.)

$\phi(6) = \phi(2 \times 3) = 1 \times 2 = 2$ thus number of primitive roots of $7 \equiv \phi(7-1) = 2$

Symmetric Ciphers

1.6.1 Notations

If you have a message $m \in \mathcal{M}$ (where $\mathcal{M}$ is the list of all possible messages), you can choose a key $k \in \mathcal{K}$ (again, where $\mathcal{K}$ is the space of all keys), to get a cipher $c \in \mathcal{C}$. (At this point if you cannot guess what $\mathcal{C}$ stands for, I don't think you should read further).

We thus need invertible functions $e_k(m)$ and $d_k(m)$ as -

$$e_k(m_i) = c_i \text{ AND } d_k(e_k(m)) = m$$

1.6.2 Principles of Succesful Ciphers

Kerckhoff's principle says that the security of a cryptosystem should depend only on the secrecy of the key, and not on the secrecy of the encryption algorithm itself.

So keeping that in mind, we design a few principles-

1. For any plaintext m and key k, it must be easy to calculate $e_k(m)$.
2. For any ciphertext c and key k, it must be easy to calculate $d_k(m)$.
3. If you have ciphertexts $c_1, c_2, c_3 \cdots c_\alpha$ encrypted using the key k, it should be really difficult to find $d_k(c_1), d_k(c_2), \cdots d_k(c_\alpha)$ without knowing k.
4. If you have $c_1, c_2, c_3, \cdots c_\beta$ and the corresponding messages $m_1, m_2, m_3, \ldots m_\beta$, it should still be difficult to decipher a c not in this set.

1.6.3 Encoding of Cipher Blocks

We can represent any plaintext in the form of bits $m_{B-1}m_{B-2}\cdots m_2m_1m_0$ where $m_i \in 0, 1 \forall i \in [0, B)$. Then we can break it into bytes or blocks of some other size and encode each block separately. We prefer to choose a $\mathcal{K}$ such that $k \in [0, 2^{B_k})$ and $B_k \geq 80$ bits or 160 bits based on the message type. (This number is chosen based on the computing capacity of a modern computer since you can brute-force check each and every key $k \in \mathcal{K}$, if B_k is not large enough.)

1.6.4 Examples of Symmetric Ciphers

Choose $\mathcal{K} = \mathcal{C} = \mathcal{M} = \mathbb{F}_p^*$ for some prime number p and then you can define-

$$e_k(m) \equiv k \cdot m \bmod (p)$$

Using that you can also find k' (in $2\log_2(p) + 2$ steps) such that-

$$d_k(c) \equiv k' \cdot c \bmod (p) \text{ to yield us } d_k(c) \equiv m$$

Although this method follows the first 3 principles, it fails the fourth principle of successful ciphering as-

$$k \equiv m^{-1} \cdot c \bmod (p)$$

The above formula gives you the exact key as soon as you find one pair of ciphertext and plaintext thus is not preferable at all.

Simple multiplicative cipher

If instead of using a modulo p, if you just used $e_k(m) = k \cdot m$, you can easily find $\gcd(c_1, c_2, c_3 \cdots c_\alpha)$ which can give you the value of k outright. (Calculating GCD of multiple numbers is not labour intensive for a computer)

Affine Cipher

Here we define ciphertext using both multiplication and addition modulo p-

$$e_k \equiv k_1 \cdot m + k_2 \bmod (p)$$

AND

$$d_k(c) \equiv k_1' \cdot (c - k_2) \bmod (p)$$

Note- Caesarean Shift Cipher is just a special case of this cipher with $k_1 = 1$

Hill Cipher

In this cipher, we first denote every message of length n as a $n \times 1$ column matrix. Then we can define a key $k = k_{1,1}k_{1,2}k_{1,3}\cdots k_{n,n}$ as-

$$k = \begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & \dots & k_{1,n} \\ k_{2,1} & k_{2,2} & k_{2,3} & \dots & k_{1,n} \\ k_{3,1} & k_{3,2} & k_{3,3} & \dots & k_{1,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ k_{n,1} & k_{n,2} & k_{n,3} & \dots & k_{n,n} \end{bmatrix}$$

We can calculate the inverse of $k = k^{-1}$ and using the both of them, our cipher looks like-

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & \dots & k_{1,n} \\ k_{2,1} & k_{2,2} & k_{2,3} & \dots & k_{1,n} \\ k_{3,1} & k_{3,2} & k_{3,3} & \dots & k_{1,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ k_{n,1} & k_{n,2} & k_{n,3} & \dots & k_{n,n} \end{bmatrix} \cdot \begin{bmatrix} m_{1,1} \\ m_{2,1} \\ m_{3,1} \\ \vdots \\ m_{n,1} \end{bmatrix} = \begin{bmatrix} c_{1,1} \\ c_{2,1} \\ c_{3,1} \\ \vdots \\ c_{n,1} \end{bmatrix}$$

Unfortunately both Affine and Hill Ciphers are both susceptible to plain text attacks. Hence we cannot use either one of these despite how dapper they look.

XOR Cipher

XOR $\oplus$ of key and message $c \equiv m \oplus k$ seems like a good option but it has the following flaw if you have access to 2 ciphers made using the same key-

$$c' \oplus c \equiv (m' \oplus k) \oplus (m \oplus k) \equiv m' \oplus m$$

$m \oplus m'$	Possible m	Possible m'
0	0	0
0	1	1
1	0	1
1	1	0

Thus we can easily find information about m from just 2 ciphers. (Although still m exactly isn't visible)

1.6.5 Random bit sequences

To circumvent the above issue, we can use different "keys" for different messages that we encrypt. One way to do that is to use a key k and a pseudo-random[3] number generator $\mathcal{R}$ such that-

$$\mathcal{R} : \mathcal{K} \times \mathbb{Z} \to 0, 1$$

Thus we can use this to construct a sequence of n-bits (the XOR key) $j_1 j_2 j_3 \cdots j_n$ such that $j_i = \mathcal{R}(k, i)$ if message to be encrypted has n bits as well. Then we can just sequentially go through every bit of the message and apply $\oplus$ with bits of the XOR key.

[3]Not called "random" as we are using some algorithm to generate it after all.

Asymmetric Ciphers

One flaw with symmetric ciphers is that the key used for encryption is same as the one used for decryption. Hence for *Alice* to send a message to *Bob*, she has to first send him a copy of the key she used for encryption. However, in the information age, that key is equally susceptible to being intercepted by *Eve* hence undermining the privacy of *Alice* and *Bob*.

One effective way to combat this is having a public key used for encryption (k_{public}) and a private key for decryption ($k_{private}$). As the names suggest, the former is available freely and can be used by anyone while sending information to *Alice* but only *Alice* has the latter key.
Some important details of this method-

- $e_{k_{public}}(m_i) = c_i$ and $d_{k_{private}}(c_i) = d_{k_{private}}(e_{k_{public}}(m_i)) = m_i$.
- It should be infeasible to calculate $k_{private}$ using just knowledge of k_{public}.

Probability Theory

1.8.1 Basics

Well the basics are luckily... pretty basic. All of the stuff we learnt already in JEE. Some of the interesting concepts are in the next few sections.

1.8.2 Monty Hall Problem

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?

One important thing to keep in mind which makes the final solution seem non-intuitive → The host knows which door has the Car behind it. He will never open the door with the car. Let us try to rationalise the problem based on Bayes' Theorem.
Assume for now that whatever door we choose is called $Door_1$, the host opens either $Door_2$ or $Door_3$

Case 1: $Car \rightarrow Door_1$
In that case $P(keep \rightarrow Car) = 1$ and $P(swap \rightarrow Car) = 0$

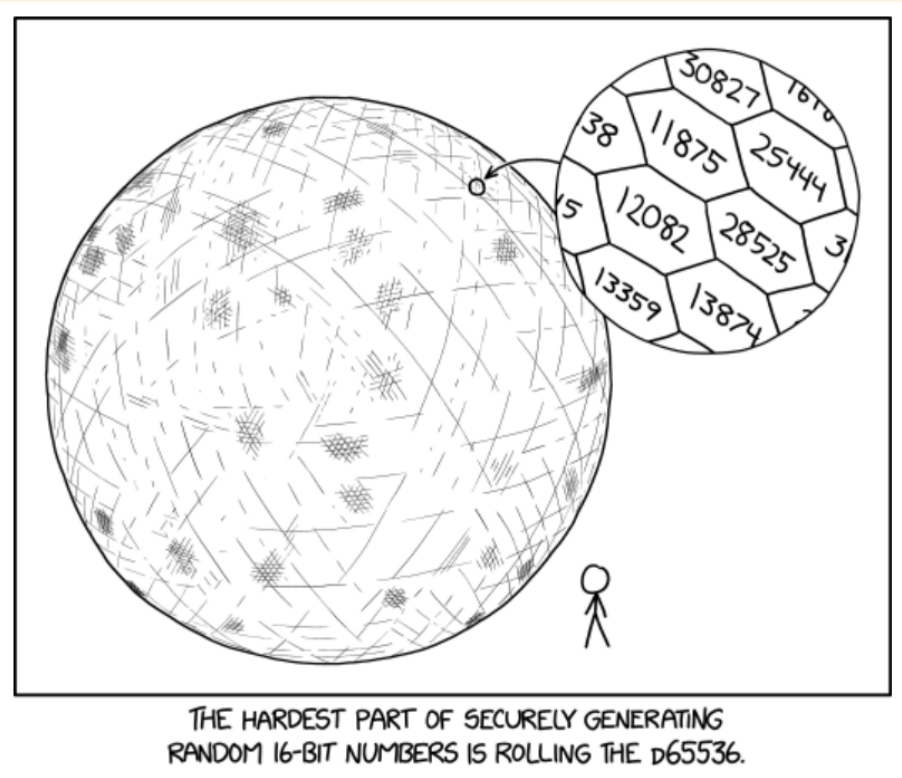
Case 2: $Car \rightarrow Door_2$
In that case $P(keep \rightarrow Car) = 0$ and $P(swap \rightarrow Car) = 1$

Case 3: $Car \rightarrow Door_3$

In that case $P(keep \rightarrow Car) = 0$ and $P(swap \rightarrow Car) = 1$

Using Total probability theorem, it is obvious that $P(keep \rightarrow Car) = \frac{1}{3}$ and $P(swap \rightarrow Car) = \frac{2}{3}$

Randomness and Obfuscation



Random Numbers

2.1.1 Why do we need random numbers

A lot of the encryption techniques used in the later weeks will require some randomness to increase the strength of the cipher. We had already covered their importance in general in Random Bit Sequences but in specific-

- RSA cryptosystems, despite being completely deterministic, requires some randomness in the plaintext for better security.
- Public key cryptosystems depend almost completely on the using random prime numbers to design the keys.

- Probabilistic Encryption systems definitely require incorporation of randomness into the encryption process.

2.1.2 Is it even possible to construct an actually random number?

Quantum Theory states that the decay of specific atom is completely probabilistic. We can be sure that for a radioactive atom, there is a T time such that after T seconds, a undecayed atom has a *50%* chance of decaying but we cannot say with 100% surety if a particular atom will decay or not.

Thus to construct the first bit of a random number, we just study a radioactive atom for T seconds and if it decays, $bit = 1$ else $bit = 0$.

Unfortunately, this is a very impractical way to construct a large random string daily use. Hence we go for pseudo random bit generators.

2.1.3 Pseudo-Random Number Generator

PRNG is just a function that takes in an initial seed S which is truly random to algorithmically create a pseudo-random number[4].

The two properties that a PRNG should have for it to be considered cryptographically secure are-

1. If *Eve* has the first k bits of the PRN, she should not be able to predict the next bit with a more than 50% chance.
2. If *Eve* has some the bits in the PRN such as $R_t, R_{t+1}, R_{t+2} \cdots$, she should not be able to predict any of $R_1, R_2, \cdots, R_{T-1}$ bits with a more than 50% surety.

To continue further, some knowledge of Hash Functions is necessary.

Hash Functions

Hash Algorithmns are like a summary of the original document. It is difficult to figure out the original document from the hash output but it acts as a signature that ensures accuracy. Due to the pigeonhole problem, it is obvious that Hash collisions would occur- but a good algorithm ensures that those are incredibly difficult to happen by happenstance and equally so to artificially bring about
- Tom Scott

Hash functions take in arbitrarily long documents D and output a shorter bit strong H. Their main objective is for authentication purposes. Some important characteristics are-

[4]Called so because of the irony of having a well defined function that can create supposedly random numbers.

- Computation of $Hash(D)$ should be fast $\rightarrow$ *linear time*
- Inversion of Hash(D) should be difficult $\rightarrow$ *exponential time* i.e. finding **any** D given a hash output H of the form $Hash(D) = H$
- Ideally we want Hash functions to have collision resistance i.e. it should be difficult for 2 non-identical documents $D1$ and $D2$ to give the $Hash(D1) = Hash(D2)$. This property is important for the sake of unambiguity in differentiating between 2 possible inputs when the inteded recepient receives the Hash output.

Note-

a. There is a considerable difference between Authentication/ Verification and Inversion. Hash functions need to be easy to compute as well as verify but should be resistant from inversion.

b. Hash is not an encryption- it cannot (ideally) be decrypted back to the original message. It is just a signature.

c. Hash outputs are "seemingly" random so even a small flipping of bits in the input should completely change the hash.

We ideally would base Hash functions on mathematically difficult problems to solve such as the Discrete Logaritms and Prime Factoring to ensure that inversion is difficult but that is impractical for even the encryption speeds today's world needs. So we prefer ad-hoc mixing methods such as the *Secure Hash Algorithm* SHA (or SHA-1) which is the universally used algorithm.

2.2.1 How does SHA-1 work?

SHA outputs 160 bits no matter what you input in; if the input is a multiple of 160 bits, we just xor the first result of the kth output with the transformation of the $(k+1)$th mixing to get the $k+1$th output. If the file isn't a multiple or is smaller than 160 bits, we just append 0s till we get a proper multiple.

1. Choose 5 32-bit numbers $h_0, h_1, h_2, \ldots, h_4$ randomly.
2. Divide the file into chunks of 512 bits and then redivide those chunks individually into 16 blocks of 32-bits (or words) $w_0, w_1, w_2 \ldots w_15$.
3. "Rotate" the words (i.e. shift the bits circularly arbitrarily **between** the words[a] (not inside the words individually)) to create 80 words $w_0, w_1, w_2, \ldots, w_{79}$.
4. Start the loop with i as control variable incrementing from 0 to 79.
5. Now just create temporarily variables $a = h_0$, $b = h_1$, $\ldots e = h_4$ and construct f from some arbitrary combination of a, b, c, d, e using AND and XOR.
6. Arbitrarily mix a, b, c, d, e using some rotations, shifts and whatever you feel like adding on a lazy Wednesday afternoon.
7. Add f and w_i to a and then equate $h_0 = h_0 + a, h_1 = h_1 + b, \ldots, h_4 = h_4 + e$.
8. End the loop once you reach i=79. (i.e. complete 80 rounds)
9. Do the same for all chunks just making sure that the values of $h_0, h_1 \ldots h_4$ are carried forward after each chunk.
10. Concatenate $h_0, h_1 \ldots h_4$ to get the final 160-bit hash.

[a]Essentially if you have 1000, 1101, 0111, 1010 and you decided to rotate the 3rd bit in each 4-bit number, you can get a 1000, 1111, 0111, 1000 i.e. the 3rd bit of each number is extracted to form the array 0,0,1,1 and this is rotated as the algorithm is designed

Thus now to create a pseudo-random number R_i, we just select a specific $Hash$ function and input into it S (A truly random number) concatenated with i to get-

$$Hash(i \mid\mid S) = R_i$$

Public Standard for making PRNGs

Start of with a random seed S and a key k for the cryptosystem. Let E_k be the encryption function associated with that k.
Every time a random number is required, create a D from some CPU parameters (temperauture and/or date and time and/or cpu usage) to get-

$$C = E_k(D)$$

Thus your random number R will be-

$$R = E_k(C \text{ xor } S)$$

You can then update S to -

$$S = E_k(R \text{ xor } C)$$

Stream Ciphers

This is an encryption technique which heavily depends on the PRNGs we defined above.
If you wanted to encrypt a 10-bit plaintext using Stream Ciphers, you will require a 10-bit keystream as well. If k_i is the ith bit of the keystream and m_i is the ith bit of the plaintext,

$$c_i \equiv k_i \oplus m_i$$

This keystream should be as random as possible to prevent codebreakers from predicting properties about the keystream thus getting information about the plaintext.

But it is infeasible to make n-bit keystreams for large values of n hence we use PRNGs. We can thus have a k bit purely random seed S with which we can create $R_1, R_2, R_3 \ldots, R_n$ using the above given algorithm. (Initial seed S is updated after every use as was defined in the last section to strengthen the security of the system)

BTW the keystream used with Stream Ciphers is known as *One Time Pad*. A n-bit ciphertext made from a truly random n-bit one time pad is mathematically impossible to decipher.

2.4.1 Deciphering Stream Ciphers

xor ($\oplus$) has an interesting property $\rightarrow$

A	B	$A \oplus B$	$(A \oplus B) \oplus B$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Hence we can say with surety that-

$$k_i \oplus c_i \equiv k_i \oplus (k_i \oplus m_i) \equiv m_i$$

Giving us a clear cut way to use this private key symmetric encryption system.

ALSO since the generation of R_n does not require any subsequent $R_{k>n}$, we can process each bit as it the CPU reads it instead of having to store it

Block Cipher

Unlike Stream Cipher, Block Cipher is a lot simpler and less secure. It divides the input into chunks of bits and evaluates them separately.[5]

2.5.1 Electronic Code Book Method

Essentially if you have a k bit codebook and a n bit plain text, you divide the plain text into chunks of k bits (some padding scheme is employed in case $n \% k \neq 0$). Then you go through each chunk and find the corresponding ciphertext in the ECB (Electronic Code Book). Thus you encrypt the entire file.

Plaintext	*Ciphertext*
00	11
01	01
10	00
11	10

2-bit Electronic Code Book

Some minor itty bitty issues- It preserves large scale structures hence it is completely useless to the most degree in obfuscating files with a large size if there is a lot of repetition.

[5] A lot of these concepts and pictures are sourced from "https://www.hypr.com/black-cipher/"

Imagine a 10×10 picture where the brightness of each pixel is defined by a 2-bit digit

Plaintext	*Ciphertext*
0000	1001
0001	1000
0010	0100
0011	0101
0100	0011
0101	0111
0110	1101
0111	1111
1000	0110
1001	1011
1010	0001
1011	1100
1100	0010
1101	1100
1110	1010
1111	0000

4-bit Electronic Code Book

$$\begin{bmatrix} 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 01 & 01 & 01 & 01 & 01 & 01 & 00 & 00 \\ 00 & 00 & 01 & 10 & 10 & 10 & 10 & 01 & 00 & 00 \\ 00 & 00 & 01 & 10 & 11 & 11 & 10 & 01 & 00 & 00 \\ 00 & 00 & 01 & 10 & 11 & 11 & 10 & 01 & 00 & 00 \\ 00 & 00 & 01 & 10 & 10 & 10 & 10 & 01 & 00 & 00 \\ 00 & 00 & 01 & 01 & 01 & 01 & 01 & 01 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix} \xrightarrow{\text{ECB}} \begin{bmatrix} 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 \\ 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 \\ 10 & 01 & 01 & 11 & 01 & 11 & 01 & 11 & 10 & 01 \\ 10 & 01 & 11 & 01 & 00 & 01 & 10 & 11 & 10 & 01 \\ 10 & 01 & 11 & 01 & 00 & 00 & 10 & 11 & 10 & 01 \\ 10 & 01 & 11 & 01 & 00 & 00 & 10 & 11 & 10 & 01 \\ 10 & 01 & 11 & 01 & 00 & 01 & 10 & 11 & 10 & 01 \\ 10 & 01 & 01 & 11 & 01 & 11 & 01 & 11 & 10 & 01 \\ 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 \\ 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 & 10 & 01 \end{bmatrix}$$

In the above illustration, despite the codebook using more bits than the individual pixels in the image, it still cannot obfuscate efficiently the initial file hence giving a lot of information away like-

1. The length of the codebook string.
2. Presence of repeated structures such as the brightspot in the middle and the darkspace in the periphery.
3. General lack of self respect for having used such a unsophisticated method.

We can clearly see the issue here- inability of the encryption system to convert the same number to 2 different numbers if they occur at 2 different places. Hence we use *Cipher Block Chaining (CBC)*.

2.5.2 Cipher Block Chaining

In this encryption method, the outcome of the previous encryption decides the outcome of the next encryption. Just like ECB mode, we do have a codebook we refer to for each chunk.

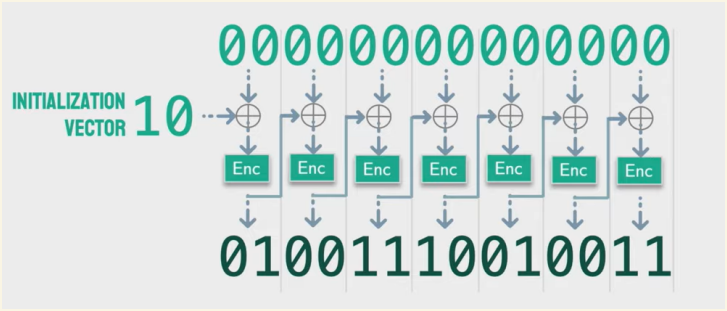


Figure 2.1: We can clearly see that the first IV (intialisation vector) is xor-ed with the first chunk. The cipher text of the first chunk is then used similarly. Notice how repeated structures are not visible anymore.

Now another strength of this method is the factor that the same codebook and plaintext can give a wildly different cipher if you just changed the initial seed (or initialisation vector).

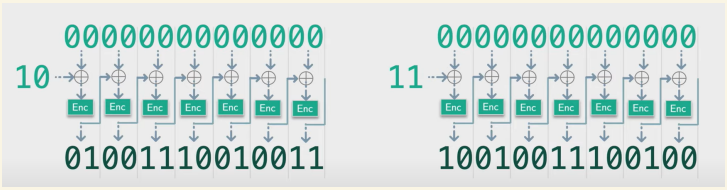


Figure 2.2: This is so nice to look at. Relish the randomness

An interesting point- The I.V. (or initial seed) is not at all a secret. It is in fact given along with the ecrypted message as it is necessary for decryption. Another property - it should not be predictable or used more than once (to ensure resistance from plaintext attacks)

2.5.2.1 How to decrypt CBC

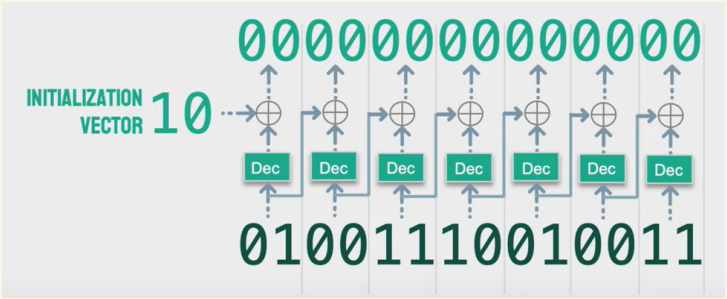


Figure 2.3: Decryption exploits the reversibility of $\oplus$

2.6 Counter Mode (CTR)

Another method of Block Ciphering is Counter Mode wherein you are actually encrypting (i.e. using the codebook) on the counter and not the plaintext. The plain text is merely xor-ed with the encryption of the counter. Initial seed decides the starting value of the counter which is incremented.

To put it algebraically, if we called the k-bit counter $C(i)$, and the code book as $ECB : \{0,1\}^k \to \{0,1\}^k$ and the ith k-bit chunk of plaintext as m_i then-

$$cipher_i = m_i \oplus ECB(C(i))$$

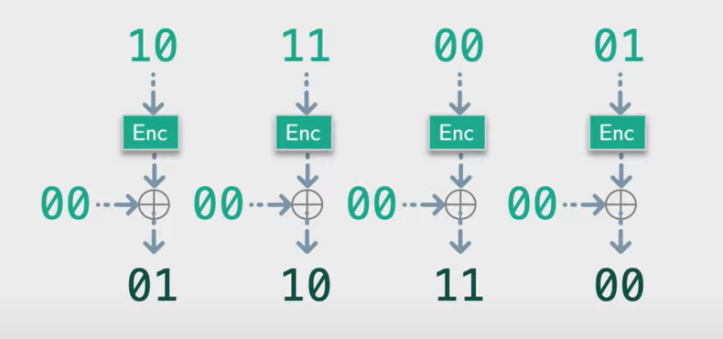


Figure 2.4: A very logical and simple encryption process. The counter is on the top and the plaintext is xored in the middle

2.6.1 Decryption

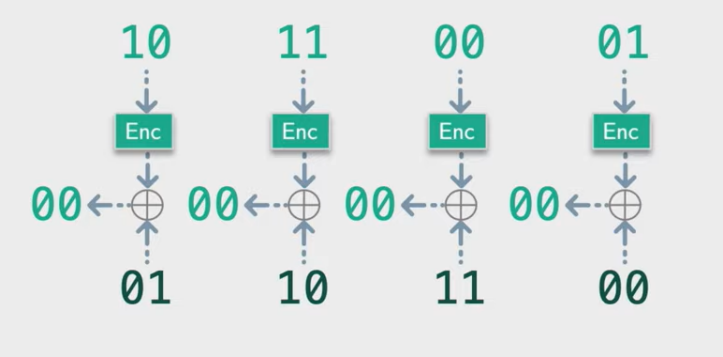


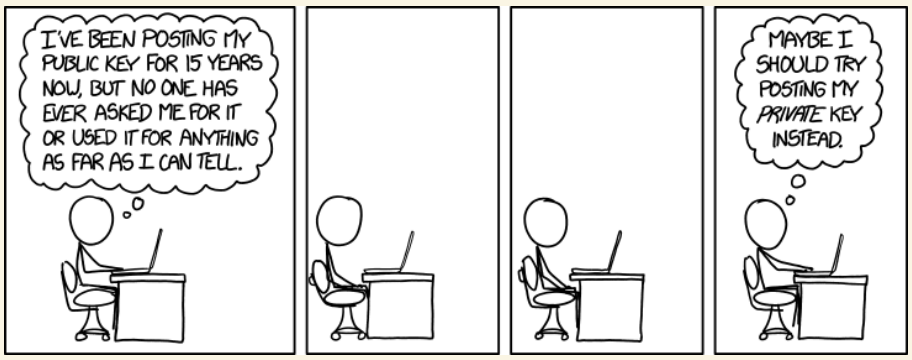
Figure 2.5: It is very similar in the sense that we interchange one

In algebraic terms, we can call it-

$$m_i = cipher_i \oplus ECB(C(i))$$

Advantages over CBC	Disadvantage over CBC
Simpler implementation Resistant to attacks terms as “padding oracle”	Not safe for small block lengths (<128 bit)

Diffie Hellman and Discrete Logarithms



Introduction to Diffie Hellman

Finally now we will study a public key cryptosystem in detail. Diffie and Hellman were some of the earliest cryptographers working to create a one-way trapdoor function - easy to compute, difficult to invert unless you have some trapdoor information.

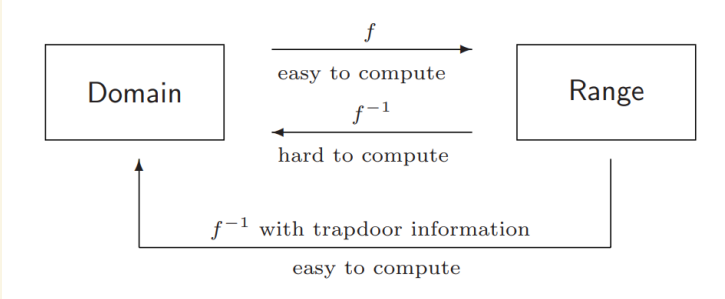


Figure 3.6: Schematic of a trapdoor function

These functions are of a particular importance. As was explained in Asymmetric Ciphers section, we make sure of k_{public}, $e_k()$ for encryption which is made public knowledge whereas $k_{private}$, $d_k()$ are kept private as they are used for decryption. Since k_{public} is constructed from $k_{private}$, it is important that the function that decides their relation is unidirectional else finding $k_{private}$ will be possible for *Eve* from just the information that is publically available.

Why exactly is the use of public key cryptography even?

Banks in the internet age would require to hand deliver keys to their clients which would massively slow down communications and created massive weaknesses in the line of communication. Hence finding a solution to this problem was important.

But another thing to keep in mind is that if *Bob* is receiving the message from *Alice*, he should be able to decrypt it since he has that extra information.

Although ultimately Diffie and Hellman failed to create a public key cryptosystem (PKC), they did manage to create a system that can securely transfer certain data by utilising the Discrete Logarithm problem.

Discrete Logarithm

Refer to Prime numbers and Finite Fields for the background information in this section. According to the Primitive Roots Theorem, in every $\mathbb{F}_p$, there exists a g such that $1, g^1, g^2, \ldots, g^{p-1}$ contains all the elements present in $\mathbb{F}_p*$ and $g^{p-1} \equiv 1 \bmod (p)$ (Fermat's Little Theorem).

Discrete Logarithm Problem (DLP) is the problem of finding x such that-

$$g^x \equiv h \bmod (p)$$

for some arbitrary h and x is called the *discrete logarithm of h to the base g* $= \log_g h$.

$$x = \log_g h \bmod (p) = \text{Discrete logarithm}(h) \quad \text{for } x, h \in \mathbb{F}_p^*$$

Note- We use the term "index" to sometimes refer to the discrete logarithm i.e. $x = \text{ind}_g(h)$. This is to ensure that the usual log and discrete log are not confused for each other.

$x' = x + (p-1)k$ would also satisfy our condition thus giving up ∞ly many solutions x. Hence x is only defined modulo (p-1). Hence we say that

$$\text{ind}_g : \mathbb{F}_p* \to (\mathbb{Z}/(p-1)\mathbb{Z})$$

3.2.1 Why is it called logarithm?

Well you can easily see that

$$\text{ind}_g(ab) = \text{ind}_g(a) + \text{ind}_g(b)$$

and that gives us enough of a reason to call it so.

Illustration

If our prime number $p = 56509$, and we know with surety that $g = 2$ is a primitive root modulo p, we can be sure that 2 is a "generator" of $(\mathbb{Z}/p\mathbb{Z}*)$ hence the discrete logarithm of every number $< p$ to the base 2 exists.
In that case, if we have to calculate $\text{ind}_2 38679$, we will have to cycle through **every** one of $1, 2, 2^2, 2^3, \cdots 2^{56508}$ mod (56509), until we get a value that matches 38679. This is exactly why the discrete logarithm problem is a formidable one.

How did we know that 2 is a primitive root modulo 56509?

If we wanted to check if a number a is a primitive root modulo p, the first approach that comes to mind might be to first evaluate all powers of a modulo p from 0 to $p-2$ and see if all numbers from 1 to $p-1$ appear exactly once. But this is too resource consuming so we use a better approach. Since we know that if a is really a primitive root, then all the powers of a modulo p would be a bijective function with the set $1, 2, 3, \cdots, p-1$, we can be sure that 1 would appear once in the powers other than the $p-1$th power. (If 1 does not appear even once in between 0 and $p-1$, we can be sure that all the values will **have** to be unique hence making a a primitive root). So we have

$$a^{p-1} \equiv 1 \bmod (p)$$

So we just have to calculate the prime factors $(q_1, q_2, ...)$ of $p-1$ and if

$$a^{\frac{p-1}{q_i}} \not\equiv 1 \bmod (p) \quad \forall i \text{ such that } q_i \text{ divides } p-1 \Rightarrow a \text{ is a primitive root modulo } p$$

Is it necessary for g to always be a primitive root modulo p?

No. As long as 2 things are ensured, we can always write $\text{ind}_g h$-

1. The discrete logarithm exists.
   e.g. In this example, $\text{ind}_6 2$ is just not defined.
2. The value is unique (in ring $\mathbb{Z}/p\mathbb{Z}$)
   e.g. In the same example, we can see that $\text{ind}_6 6$ will have multiple values (1, 3, 5).

The strongest point of our encryption using Discrete Logarithm is that there is a lot of variation and a seeming randomness in the values of the discrete logarithms in our data.

h	$\log_g(h)$	h	$\log_g(h)$
1	0	11	429
2	183	12	835
3	469	13	279
4	366	14	666
5	356	15	825
6	652	16	732
7	483	17	337
8	549	18	181
9	938	19	43
10	539	20	722

Figure 3.7: Table of values of h for $g = 627$ and $h = 941$

Using Discrete Logarithm for Key Exchange

Let us say that *Bob* and *ALice* want to begin communication across an insecure channel. If they can agree upon a key, the communication can begin smoothly using any of the block or stream cipher ways we have already discussed. But how do you send the key, very well knowing that any and all information across the channel can easily get intercepted?

In that case, *Bob* and *Alice* publicly agree upon g and p. *Alice* and *Bob* then privately construct some a and b respectively without letting it be known to the outside world.

Bob then sends *Alice* $A = g^a \bmod (p)$ and *Alice* sends back $B = g^b \bmod (p)$. Due to the nature of the insecure channel, *Eve* is able to safely intercept both A and B.

Finally *Alice* calculates $B^a \bmod (p)$ and *Bob* calculates $A^b \bmod (p)$. It is obvious that even in $\mathbb{F}_p$, $A^b = g^{ab} = B^a \bmod (p)$ hence that becomes the key the both parties use for secure communication henceforth. *Eve*, despite knowing A, B, g and p, is still unable to calculate $g^{ab} \bmod (p)$ trivially because of her not knowing the values a and b.

Information *Bob* has	Information *Alice* has	Information *Eve* has
Before exchange		
$b,\ g,\ p,\ B\ (= g^b \bmod (p))$	$a,\ g,\ p,\ A\ (= g^a \bmod (p))$	$g,\ p$
After exchange		
$b,\ g,\ p,\ B,\ A$	$a,\ g,\ p,\ A,\ B$	$g,\ p,\ A,\ B$
Key $(=A^b = B^a = g^{ab})$		No information about the key

Does this mean that finding the key is a Discrete Logarithm Problem (DLP)?

No.
If the only way for *Eve* to find the key was to find out a from A and g, p (or similarly b from B and g, p), then admittedly, this problem would have been a classic DLP problem.
However, *Eve* actually has both A and B and thus actually has to find g^{ab} using g^a and g^b which is definitely a simpler problem. This problem is termed as the **Diffle Hellman Problem** (DHP) and is not harder than the DLP.

We called DHP to be "not more difficult than" DLP instead of saying DHP is easier than DLP. The reason for it is that although we know if DLP was computable, DHP would be trivially solvable, the converse hasn't been proven.

For the modern standards of computing power, p needs to be ≈ 1000 bits for brute-force to not be a feasible option in *Eve*'s hands. Also, $g \approx p/2$ ensures most computational difficulty in the inversion of Discrete Logarithm.

But wait!

Doesn't this mean we can effectively create a *One Time Pad* (OTP) and hence an indecipherable means of communication is feasible using DHP?

Yes, but also no.[6]
Although it is very much possible to have a system wherein,

- If *Alice* and *Bob* agree on only sharing 160-bit messages, they publicly choose a large enough ($\approx$ 160 bit)p, g.
- Then both of them use PRNGs privately to construct a, b of a suitable size so as to ensure that the final g^{ab} has a length larger than 160-bits.
- If the key has more than 160 bits, chop off the last few bits until you get a key k' of exactly 160 bit size.
- Use $\oplus$ (xor) on k' and m and then if *Alice* sends this encrypted message to *Bob*, *Bob* can just calculate $k' \oplus c \equiv m$ to get the original message.
- *Alice* and *Bob* change keys thus a and b for every message sent across the channel thus ensuring that they create a feasible One-Time Pad encrypted network.

Although the above method is really great, it is not a secure system in today's times. The flaws this suffers from can best be explained at the end of this chapter.

3.3.1 The anti-climactic conclusion of Diffie-Hellman

As you will come to see in the further sections, certain cases of Diffie Hellman are feasible to solve really quick and others can be solved with more efficient algorithms. There also exist some other flaws which prevent Diffie Hellman, despite all its glory, from becoming the cryptosystem standard it deserved to be. Rest in Peace :(

ElGamal Public Key Cryptography

Fear not, for now we have a different approach to use the Discrete Logarithm Problem without having to sacrifice either speed or security (we don't want it to be any simpler to solve than the **DHP**).
The steps for setting up an ElGamal Cryptosystem is as follows-

1. *Alice* and *Bob* agree on a g and a large enough p.
2. *Alice* randomly[7] chooses an a and publicly displays $A \equiv g^a \bmod (p)$.
3. *Bob* then randomly[8] creates a *ephemeral key* k (which will be used for this message only).
4. *Bob* can only encrypt a message $2 < m < p$. He does so by calculating the 2 following values $c_1 \equiv g^k \bmod (p)$ and $c_2 \equiv m \cdot A^k \bmod (p)$.
5. He then sends c_1 and c_2 to *Alice* who then can calculate $m \equiv (c_1^a)^{-1} \cdot c_2 \bmod (p)$.

[6] At this point, the reader should just get used to seeing every question mark and brace themselves for a heartless "No."
[7] Of course, pseudo-randomly using a suitable PRNG or such
[8] Again the same misnomer as above.

Proof of the above-

$$c_1^k \equiv g^{ak} \Rightarrow (c_1^k)^{-1} \equiv (g^{ak})^{-1} \bmod (p)$$

AND

$$c_2 \equiv m \cdot (g^a)^k \equiv m \cdot g^{ak} \bmod (p)$$

Hence

$$(c_1^a)^{-1} \cdot c_2 \bmod (p) \equiv (g^{ak})^{-1} \cdot m \cdot (g^{ak}) \equiv m \bmod (p)$$

Since c_1 and c_2 are of the same range of bits as m, in order to send one bit of data, we send 2 bits of encrypted bits hence it is called a *2-to-1 message expansion.*

ElGamal Cryptography is as secure as the Diffie Hellman problem - i.e. in order to decrypt a ElGamal encrypted code, you need to solve the Diffie Hellman problem.[9]

Mathematical Pre-requisite: Group Theory

3.5.1 General properties of a Group

A group consists of a set $\mathbf{G}$ and a rule, which we denote by $\star$, for combining two elements $a, b \in \mathbf{G}$ to obtain an element $a \star b \in \mathbf{G}$. The composition operation $\star$ is required to have the following three properties:

- **Identity law-** There exists an $e \in \mathbf{G}$ such that, for every $a \in \mathbf{G}$,
$$a \star e = e \star a = a$$
- **Inverse law-** There exists a unique $a^{-1} \in \mathbf{G}$ for every $a \in \mathbf{G}$ such that,
$$a \star a^{-1} = a^{-1} \star a = e$$
- **Associative law-** For every $a, b, c \in \mathbf{G}$,
$$a \star (b \star c) = (a \star b) \star c$$
- **Commutative law-** (optional- if group follows this as well, it is called a *Commutative* or *Abelian Group*) For every $a, b \in \mathbf{G}$,
$$a \star b = b \star a$$

[9] An intuitive proof would be- assume you have a machine that can decrypt any ElGamal encrypted data if you supplied it the necessary c_1 and c_2. In that case, providing $c_1 = B$ and $c_2 = 1$ just gives us $(c_1^a)^{-1} \cdot c_2 \equiv (g^{ab})^{-1} \bmod (p)$ hence giving us the Diffie Hellman key essentially.

Order of a group

$|\mathbf{G}|$ or $\#\mathbf{G}$ is the *order of group* $\mathbf{G}$ and it is the number of elements present in $\mathbf{G}$. $\mathbf{G}$ is called a *finite group* if $\#\mathbf{G} \in \text{finite}\mathbb{N}$.

Examples-

1. $\mathbb{F}_p^*$ is a finite group, with order $(p-1)$ with composition operation $\star = \times$ (multiplication). $e = 1$ and inverses do exist.
2. $\mathbb{Z}$ has $\star = +$ (addition) and is an infinite group. $e = 0$ and inverse of $a \Rightarrow -a$
3. $\mathbb{Z}$ with $\star = \times$ is not a group since inverses do not exist inside the group for every element.
4. A non commutative group can be $\mathbf{G} = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad ; (ad - bc \neq 0) \right.$. It has $e = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and since $\det(A) \neq 0$ where A is a matrix $\in \mathbf{G}$, an inverse also exists.

If g is an element of $\mathbf{G}$, g^x denotes

$$\underbrace{g \star g \star g \star \cdots \star g}_{\text{x times}}$$

\# $g^0 = e$

\# If $x < 0$, we define g^x as $(g^{-1})^{|x|}$

\# Even though it may seem non-intuitive, g^x for $\begin{cases} \mathbf{G} = \mathbb{Z} \\ \star = + \end{cases}$ is actually $g + g + g + \cdots = x \cdot g$

If $a \in \mathbf{G}$, and if d is the smallest positive integer such that,

$$a^d = e$$

then d is called **order of a**. If no such d exists, a is said to have **infinite order**.

3.5.2 Generalisation of Fermat's "Little" Theorem

If $\mathbf{G}$ is a finite group, then every element $\in \mathbf{G}$ will have finite order.
Furthermore, if $a \in \mathbf{G}$ and has order d, and[a]

$$a^k = e \Rightarrow d \mid k$$

Lagrange's Theorem

Let $\mathbf{G}$ be a finite group and let $a \in \mathbf{G}$. Then the order of a divides the order $\mathbf{G}$.[a]

[a]Proof: If $a \in \mathbf{G} = \{g_1, g_2, g_3, \cdots, g_n\}$, we can construct a new group $\mathbf{S_a} = \{a \star g_1, a \star g_2, \cdots, a \star g_n\}$. Now using the definition of group (If $a, b \in \mathbf{G}$, $a \star b \in \mathbf{G}$ as well) we can be sure that every element of $\mathbf{S_a}$ is an element of $\mathbf{G}$ as well since $a \in \mathbf{G}$. Keeping that in mind,

$$a \star g_1 \times a \star g_2 \times \cdots \times a \star g_n = g_1 \times g_2 \times \cdots \times g_n$$

Now we assume our group is Commutative (more general proof is slightly more rigorous) hence we can say that

$$a^n = e$$

In other words, if $|\mathbf{G}| = n$, then

$$a^n = e \text{ AND } d \mid n$$

[a]Trivial proof: Since d is the smallest positive integer which ensures $a^d = e$, if we wrote $k = q \cdot d + r$, where $0 < r < d$, then obviously $a^k \neq e$

How difficult is the DLP?

Up until now, we have claimed that the DLP is a very difficult problem. Now it is time to quantify the difficulty using big-$\mathcal{O}$ notation.

3.5.1 Big- $\mathcal{O}$ notation

Formal definition. If $f(x)$ and $g(x)$ are 2 positive functions, $f(x)$ is called "big-$\mathcal{O}$ of $g(x)$" iff there exists a positive C, k such that

$$|f(x)| \leq C \cdot |g(x)| \forall x > k$$

In other words, $f(x) = \mathcal{O}(g(x))$ iff

$$\lim_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| \text{ exists and is finite}$$

Some examples-

- $x^2 + 3x + 2981 = \mathcal{O}(x^2)$
- $(\log x)^{375} = \mathcal{O}(x^{0.001})$
- $x^2 \cdot 2^x = \mathcal{O}(e^x)$

Although this is the mathematically pure definition, we will use a slightly different definition somewhat loosely while we talk of difficulty in the proceeding sections -

"Definition". If we made a proper list of "complexities" $(0, 1, \log n, n, n^2, n^3, \ldots 2^n, \ldots, n!, n^n \ldots)$, and if $f(x) = \mathcal{O}(g(x))$, $g(x)$ has to be the "least complex" function which satisfies the previous formal definition.

In other words, $f(x) = \mathcal{O}(g(x))$ iff

$$\lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| \text{ exists and is non-zero finite}$$

If an algorithm has an input of k-bits, then if output has order-

- $\mathcal{O}(k^a)$ where $a \in$ positive constant: Polynomial order $\rightarrow$ "Easy" Problems
- $\mathcal{O}(e^{ax})$ where $a \in$ positive constant: Exponential order $\rightarrow$ "Hard" Problems

There also exists a "Sub-exponential order" can be best described as "grows faster than any polynomial function but slower than **any** exponential function of the form $b^x \quad \forall b > 1$."

3.5.2 Computing $\mathcal{O}$ for DLP

$$g^x = h \text{ where } g \in \mathbf{G} = \mathbb{F}_p^* \text{ and } \star = \times \Rightarrow \text{Finding } x, \text{ given } g, p, h \quad \cdots (\text{DLP})$$

If $p \in (2^k, 2^{k+1})$, then obviously p, g, h all will take individually k-bits at most. Thus we can say that input/data we are dealing with is $\mathcal{O}(k)$.

Using trial-and-error method, we will, in the worst case scenario, be evaulating all of the $(p-1)$ numbers from 1 to $p-1$ hence order= $\mathcal{O}(p) = \mathcal{O}(2^k)$ hence exponential $\rightarrow$ difficult.

But that is not the only approach we have in hand.

* For some primes, where $p-1$ can be factored into a product of small primes, we can use the *Pohlig-Hellman* algorithm which makes the DLP for them quite easy.
* *Collision algorithm* works for all primes and takes $\mathcal{O}(\sqrt{p}\log p)$ steps (still exponential but easier than $\mathcal{O}(p)$)
* Using index calculus, we can bring it down to $\mathcal{O}(e^{c\sqrt{(\log p)(\log\log p)}})$ which is sub-exponential.

3.5.3 How there is no inherent difficulty in DLP

The last few sections about DLP in $\mathbb{F}_p^*$ might give you an impression that all DLPs of the form $g^x = h \quad g, h \in \mathbf{G}$ are inherently difficult.

In constrast, if $\mathbf{G} = \mathbb{F}_p$ and $\star = +$, our DLP can also be written as

$$x \cdot g = h \bmod (p)$$

Now using Extended Euclidean Theorem, we know that finding g^{-1} in the above setup will take $\mathcal{O}(\log p) = \mathcal{O}(\log 2^k)$ steps thus is solvable in linear time.

With index calculus, DLP for $\mathbf{G} = \mathbb{F}_p^*$ and $\star = \times$ is solvable in sub-exponential time.

If we instead had *Elliptic curves*, our best algorithm for DLP takes exponential time.

Collision Algorithm

Collision Algorithms work by creating 2 lists, of approximately the same size for maximum efficiency and then comparing the 2 lists. This is considerably faster than just trial-and-error method we evaluated until now.

Shank's "Babystep- Giantstep" Theorem

Let $g \in \mathbf{G}$ be an element of order $N > 2$. Then we can employ the following algorithm to get order of $\mathcal{O}(\sqrt{N} \cdot \log N)$[a]-

1. Let $n = 1 + \lfloor \sqrt{N} \rfloor$ i.e. $n > \sqrt{N}$.
2. Then construct 2 lists-
$$\text{List}_1 = e, g, g^2, g^3, \ldots g^n$$
$$\text{List}_2 = h, h \cdot g^{-n}, h \cdot g^{-2n}, \ldots, h \cdot g^{-n^2}$$
3. Now try to find the common entry in both lists. There should exist just one pair such that
$$g^i = h \cdot g^{-j \times n}$$
4. Then $x = i + n \times j$ is the solution to $g^x = h$.

[a]Proof: Making each list takes n multiplications each (fast powering algorithm) and sorting and searching through the list takes another $\mathcal{O}(\log n)$ steps- bringing the total time to $\mathcal{O}(n \log n) = \mathcal{O}(\sqrt{N} \log N)$

Intuitive explanation for how this works:
First of all, the size of $\mathbf{G}$ does not matter since, if h is a valid solution of $g^x = h$, x will have exactly one solution per N per values of x we cycle through. We will only concentrate on the first N values i.e. the first x that satisfies our condition.

Moreover, we can write $x = p \cdot n + q$ and since $x \leq N$ AND $n > \sqrt{N}$, $p < \sqrt{N}$ and by remainder theorem, $q < n \Rightarrow q < \sqrt{N}$. Hence if there was a way to make 2 lists such that one decides p and the other sets q, our inversion algorithm would be successful.

Hence we find $u = (g^{-1})^n$ (which due to the extended Euclidean theorem, is easy to compute) and then just find $h, h \cdot u, h \cdot u^2, \ldots, h \cdot u^n \Rightarrow$ List$_2$ a.k.a. p setter. List$_1$ is just $e, g, g^2, \ldots, g^n$ which is the q setter.

Finally when they correspond, $h \cdot u^p = h \cdot (g^{-1})^{p \cdot n} = g^q \Rightarrow h = g^{p \cdot n + q} = g^x$

Jargon Overload

That was a lot of jargon. Let's take a look at an example for better understanding.

Illustration

In DLP, take $\mathbf{G} = \mathbb{F}_p^*$ and $\star = \times$ with $p = 17389$, $g = 9704$, and $h = 13896$. Find a suitable x.

Order of $g = 9704$ in $\mathbb{F}_{17389}^*$ is 1242.[a]
Using that,

$$n = \lfloor \sqrt{N} \rfloor + 1 = 36$$

Next,

$$u = (9704^{-1})^{36}$$

We find 9704^{-1} using recursion with Extended Eucleadian Theorem in linear time. After that, we just use Fast Powering Algorithm to find u.
That yields us

$$u = 2494$$

Now we constuct the two lists-

k	g^k	$h \cdot u^k$
1	9704	347
2	6181	13357
3	5763	12423
4	1128	13153
5	8431	7928
6	16568	1139
7	**14567**	6259
8	2987	12013

k	g^k	$h \cdot u^k$
9	15774	16564
10	12918	11741
11	16360	16367
12	13259	7315
13	4125	2549
14	16911	10221
15	4351	16289
16	1612	4062

k	g^k	$h \cdot u^k$
17	10137	10230
18	17264	3957
19	4230	9195
20	9880	13628
21	9963	10126
22	15501	5416
23	6854	13640
24	15680	5276

k	g^k	$h \cdot u^k$
25	4970	12260
26	9183	6578
27	10596	7705
28	2427	1425
29	6902	6594
30	11969	12831
31	6045	4754
32	7583	**14567**

As you can clearly see,

$$9704^7 = 14567 = 13896 \cdot 2494^{32} = 13896 \cdot (9704^{-36})^{32} \Rightarrow 13896 = 9704^{1159} \Rightarrow x = 1159$$

[a] How did we get this order?
The standard algorithm is to try and break 17388 into its prime factors and then try them one by one till the smallest factor k yields $g^k = e$. (Reason- Size of the group is 17388 hence using the Langrange's Theorem, we can be sure that the order is a factor of order of the group)

Congruences moduli composite numbers

Up until now, we have only dealt with congruences of the form $\mod(p)$; $p \in$ primes. But, what if we have to do computations in $\mod(m)$; $m \notin$ primes? Well that is where we get a helpful hand from the Chinese Remainder Theorem.

3.7.1 Chinese Remainder Theorem

Suppose you are given-

$$x \equiv a \bmod (p) \text{ AND } x \equiv b \bmod (q); \text{ where } p,\ q \in \text{primes}$$

We can use this information to find the value of $x \bmod (p \cdot q)$.

Algorithm:

1. Since $x \equiv a \bmod (p)$, we can write $x = a + p \cdot y$.
2. Input that in the second congruence to get $p \cdot y \equiv b - a \bmod (q)$. Now since $\gcd(p, q) = 1$ (prime numbers, duh), we know that p^{-1} exists.
3. Hence finally we get $y \equiv p^{-1}(b - a) \bmod (q)$
4. Assuming $y < q$, we input the RHS into $x = a + p \cdot y$ and viola! We just got a value of x in modulo $p \cdot q$ (since, if c_1 and c_2 are solutions of x in this situation, $c_1 = k \cdot (p \cdot q) + c_2$)

Illustration

$$x \equiv 1 \bmod (5) \text{ AND } x \equiv 9 \bmod (11)$$

Thus

$$x = 1 + 5y \Rightarrow 5y \equiv 8 \bmod (11)$$

Now $5^{-1} = 9$. Using that,

$$y \equiv 72 \bmod (11) \equiv 6 \bmod (11)$$

Finally $y = 6 \Rightarrow x = 31$ is a particular solution. The final answer is-

$$x \equiv 31 \bmod (55) \Leftrightarrow x = 31 + 55k$$

Extending it to multiple numbers is just iteratively i.e. if $\gcd(m_1, m_2, m_3, \ldots, m_i) = 1$, $x \equiv a_1 \bmod (m_1)$, $x \equiv a_2 \bmod (m_2)$, $x \equiv a_3 \bmod (m_3)$, ... , $x \equiv a_i \bmod (m_i)$. Now just try to find $x \bmod (m_1 \cdot m_2)$ and then continue accordingly since $\gcd(m_1 \cdot m_2, m_3) = 1$.

And thus we can just decompose any m into a product of primes and then apply Chinese Remainder Theorem.

3.7.2 Application of CRT

A good use⇒ Finding square roots in modulo m. It is actually easy to calculate square roots in modulo prime, and more so in primes of form $4l + 3$.
If $x^2 \equiv a \bmod (p)$, then we can write that $x \equiv a^{\frac{p+1}{4}} \bmod (p)$

How? Well, if we assumed g is the primitive root modulo p, we can write $x \equiv b \equiv g^k$ as a solution. Then,

$$x^2 \equiv a^{\frac{p+1}{2}} \equiv a^{\frac{p-1}{2}} \cdot a \equiv g^{k\cdot(p-1)} \cdot a \equiv 1 \cdot a \bmod (p)$$

Using this, if we had to solve for $x^2 \equiv a \bmod (m)$ where $m = p_1 \cdot p_2$, if they are of the form $4\alpha + 3$, we can easily write,

x can be found by solving the congruences of y and z where $y^2 \equiv a \bmod (p_1)$ and $z^2 \equiv a \bmod (p_2)$

We can solve according to the previously discussed method.

Note: $x^2 \equiv a \bmod (p)$ will have 2 solutions but the above one will have 4 solutions since we can solve 4 different final congruences based on + - changes.

Factorisation of m isn't really easy as m increases in size. In that case, directly evaluating whether or not the roots exist or not becomes really a tedious task. We can use that to construct a trapdoor function- "Goldwasser–Micali" Cryptosystems[a] depend on this function with the trapdoor information being the factors of m.

[a] [7]

Pohlig-Hellman Algorithm

Pohlig-Hellman first starts off by analysing the fact that when we are talking about solutions of $g^x \equiv h \bmod (p)$, $x \in \mathbb{Z}/(p-1)\mathbb{Z}$ since $x = p - 1$ will just give the same value as $x = 0$. So we try to exploit the fact that $p - 1$, an obviously composite number has prime roots and hodge podge a solution using Chinese Remainder Theorem.

Let us first start off with an example instead of the actual theory.

Illustration

Given that 2 is a generator of 211 (a.k.a. primitive root modulo 211), find x such that[a]

$$2^x \equiv 41 \bmod (211)$$

Proof:
Since we know that 2 is a generator, we can be sure that a solution exists. Now, let us suspend belief for a few fleeting seconds and just find the roots of $\phi(211) = 210$. The factorisation yields $210 = 2 \cdot 3 \cdot 5 \cdot 7$.
Again, as a leap of faith just find $a^{\frac{\phi(p)}{7}} \bmod (p)$ (for sake of simplicity, refer to 211 as p and $a \equiv 2^x \equiv 41$)

$$a^{\frac{\phi(p)}{7}} \equiv (2^x)^{\frac{\phi(p)}{7}} \bmod (p)$$

Using remainder theorem, write $x = 7q + r \Rightarrow 2^{q\cdot\phi(p)} \cdot 2^{\frac{r\cdot\phi(p)}{7}} \equiv 1 \cdot 2^{\frac{r\cdot\phi(p)}{7}} \bmod (p)$

$$\text{Hence, } 41^{\frac{\phi(p)}{7}} \equiv 2^{\frac{r\cdot\phi(p)}{7}} \Rightarrow 41^{2\cdot3\cdot5} \equiv 2^{r\cdot2\cdot3\cdot5} \bmod (p)$$

Now the funny thing is, since $r < 7$, we can just check all values of r in $\{0, 1, \ldots, 6\}$ to get finally, $x \equiv 3 \bmod (7)$.

We do the same with 5 (run all values of r from 0 to 4 in $41^{2\cdot3\cdot7} \equiv 2^{r\cdot2\cdot3\cdot7} \bmod (p)$) to get $x \equiv 2 \bmod (5)$

For $3 \Rightarrow x \equiv 2 \bmod (3)$

For $2 \Rightarrow x \equiv 1 \bmod (2)$
Finally, we have 4 equations which we can stitch togther using Chinese Remainder theorem to get

$$x \equiv 17 \bmod (210)$$

Since $210 < 211$, we can safely say that any value we get this way is also the same for modulo 211 hence,

$$2^{17} \equiv 41 \bmod (211)$$

[a] Example was taken from this video.

The power of *Pohlig-Hellman Algorithm* lies in the fact that we need to apply trial-and-error only on small values of r (between 0 and the prime q_i) instead of from 1 to $p - 1$.

Pohlig-Hellman Algorithm

If $g \in \mathbf{G}$ with an order of N and we can write $N = q_1^{e_1} \cdot q_2^{e_2} \cdot \ldots \cdot q_m^{e_m}$ then we can solve the discrete logarithm problem of the form $g^x = h$ can be solved in $\mathcal{O}(\sum_{i=1}^{m} S_{q_i^{e_i}} + \log N)$ steps where $S_{q_i^{e_i}}$ is the number of steps it takes to solve a discrete value problem in modulo $q_i^{e_i}$ if we follow the algorithm.

In simpler words, if we are looking at the equation,

$$g^x \equiv h \bmod (p)$$

where $(p-1)$ can be factorised into $q_1^{e_1} \cdot q_2^{e_2} \cdot \ldots \cdot q_m^{e_m}$, our problem can be solved using Pohlig-Hellman Algorithm in $\mathcal{O}(\sum_{i=1}^{m} S_{q_i^{e_i}} + \log N)$ steps where $S_{q_i^{e_i}}$ is the number of steps it takes to solve the DLP

$$g^{\left\{r\cdot\left(\prod_{j\neq i}^{m} q_j^{e_j}\right)\right\}} \equiv g^{\left\{r\cdot\frac{(p-1)}{q_i^{e_i}}\right\}} \equiv h^{\left\{\frac{(p-1)}{q_i^{e_i}}\right\}} \bmod (p)$$

We can further optimise the algorithm to get $\mathcal{O}(e \cdot S_q)$. Using index calculus, we can cut it down further.

Pohlig-Hellman Algorithm illustrates that p of the form $p = 2q + 1$, where $q \in$ prime is the more secure in comparison to others.

Why don't we use pure DHP or any DLP based cryptosystem?

As discussed above, there do exist a lot of algorithms that can considerably bring down the computation time for inversion of the DHP. But even then, we can carefully select the primes numbers and construct a system that is difficult to invert.

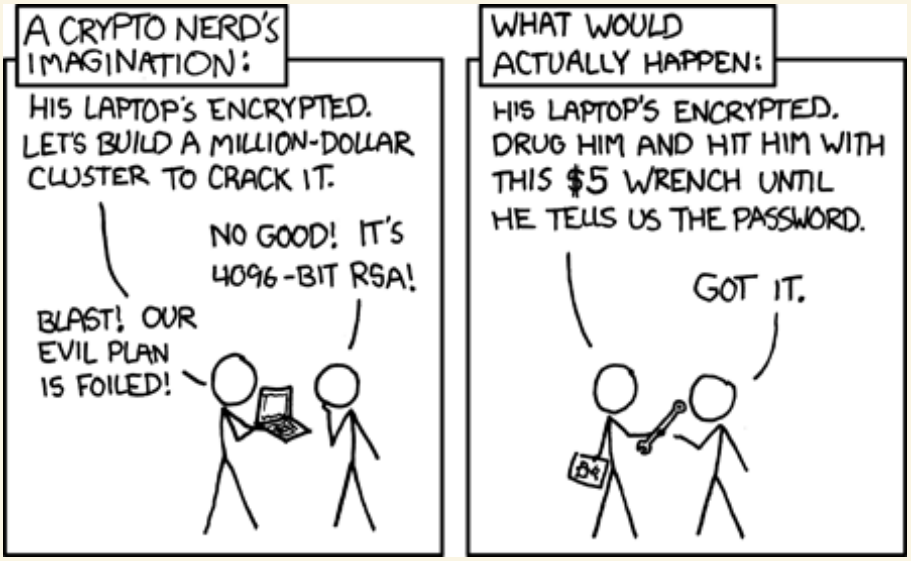
The main issue is a **lack of authentication**.

If *Alice* and *Bob* are talking over an insecure network such that *Eve* is not only able to intercept but also block direct communication between *Alice* and *Bob* between their conversation, she can use the following trick to get information out of both of them without ever even solving the DHP.

Man in the Middle Attack

Eve sets a e of her own. Since g, p are openly available, she can just send E to both *Bob* and *Alice* and now the key becomes A^e when talking with *Alice* and B^e when talking to *Bob*. However, if we were able to attach some sort of authentication method to our system, we can still use it.

RSA and Integer Factorisation



What we have covered until now

We studied the Diffie Hellman and other cryptosystems based on the Discrete Logarithm Problem. Till now we have studied Fermat's Little Theorem,

$$a^{p-1} \equiv 1 \bmod (p)$$

Let us now try to generalise it for some composite number N.

4.1.1 Euler's Formula

If $N = p \cdot q$, where $p, q \in$ primes, then we can write that

$$a^{(p-1)(q-1)} \equiv 1 \bmod (p \cdot q) \equiv 1 \bmod (N)$$

Moreover, if $g = \gcd(p-1, q-1)$, we can even write,

$$a^{\frac{(p-1)(q-1)}{g}} \equiv 1 \bmod (p \cdot q) \equiv 1 \bmod (N)$$

Proof

Firstly, let us prove $a^{\frac{(p-1)(q-1)}{g}} \equiv 1 \bmod (p)$ and $a^{\frac{(p-1)(q-1)}{g}} \equiv 1 \bmod (q)$ separately.
Because of the fact that both $\frac{p}{g}$ and $\frac{q}{g}$ are integers and using the Fermat's Little Theorem, we can trivially prove these to be true.
Since both the equations are true, we can be sure that $q \mid a^{\frac{(p-1)(q-1)}{g}} - 1$ and $p \mid a^{\frac{(p-1)(q-1)}{g}} - 1 \Rightarrow p \cdot q | a^{\frac{(p-1)(q-1)}{g}} - 1$ since $\gcd(p, q) = 1$.

Since we know that $a^{(p-1)\cdot(q-1)} \equiv 1 \bmod (p \cdot q)$, we can write that any exponent x in the form $a^x \equiv 1 \bmod (p \cdot q)$ lies in the ring $\mathbb{Z}/\big((p-1)\cdot(q-1)\big)\mathbb{Z}$ and hence modulo $(p-1)\cdot(q-1)$ since $x = x$ gives the same results as $x = x + (p-1)\cdot(q-1)$.

Actually, we infact, can write it exists in modulo $\frac{(p-1)\cdot(q-1)}{g}$ because of the above formula.

Integer Factorisation Problem

Now let us set the foundations for the RSA cryptosystem which depends on the integer factorisation problem. Until now, we have seen systems where DLP is used which asks you to solve for x when a, b and p and given in the following system-

$$a^x \equiv b \bmod (p)$$

Now we will deal with systems where security depends on the difficulty in finding x when e, c and N are known publicly-

$$x^e \equiv c \bmod (N)$$

4.2.1 Is it really hard to find the eth root modulo N?

4.2.1.1 $N \in$ primes

Assume $N = p \in$ primes. In that case, finding the eth root is actually trivially easy.

If $N = p$, we just need to solve for d in

$$d \cdot e \equiv 1 \bmod (p-1)$$

Reason? We had discussed while studying the Pohlig-Hellman Algorithm that in $a^x \equiv 1 \bmod (p)$, x belongs to the ring $\mathbb{Z}/(p-1)\mathbb{Z}$ hence all calculations relating to it exist in modulo $(p-1)$ space. From the Extended Euclidean Theorem, we know that finding d in the above congruence takes linear time $\rightarrow$ trivially easy.
After that we just need to calculate $c^d \bmod (p)$ as

$$x^e \equiv c \bmod (p) \Rightarrow (x^e)^d \equiv c^d \bmod (p) \Rightarrow x^1 \equiv c^d \bmod (p)$$

Note- We need to choose e and p such that $\gcd(e, p-1) = 1$, else $d \equiv e^{-1} \bmod (p-1)$ won't exist.

4.2.1.2 $N \in$ product of 2 primes

Let us now solve for $N = p \cdot q$, where $p, q \in$ primes-

$$x^e \equiv 1 \bmod (p \cdot q)$$

Now we have to find the inverse d of e such that,

$$x^{d \cdot e} \equiv x^1 \equiv c^d \bmod (p \cdot q)$$

To do that, we refer to Euclid's formula, to state that d would satisfy

$$d \cdot e \equiv 1 \bmod \big((p-1) \cdot (q-1)\big)$$

Actually, to make our calculations easier henceforth (i.e. to get a smaller value of d), we use-

$$d \cdot e \equiv 1 \bmod \left(\frac{(p-1) \cdot (q-1)}{g}\right)$$

Solving for d in this is also very easy since $\frac{\big((p-1) \cdot (q-1)\big)}{g}$ is a composite number which we can solve[10] by finding the prime factors and then evaluating e^{-1} in each one of them and then weaving them together with CRT; all in linear time.

4.2.2 Then how is this cryptosystem even safe?

As of now, we have seen that if we can find the prime factors of any number N, we can easily find the inverse d such that $x \equiv c^d$. Finding individual inverses modulo primes takes linear time which calculating c^d takes $\mathcal{O}(\log d)$ time. So how exactly can we say that this can be used to create a secure cryptosystem?

Well the difficulty actually lies in the factorisation of N itself. The only way to find the inverse of e, is by factorising N which becomes increasingly difficult as N increases.
Why? We need to find $(p-1) \cdot (q-1)$ which is $p \cdot q - (p+q) + 1 = N - (p+q) + 1$. Hence the only way to know $(p-1) \cdot (q-1)$ is if you know $p+q$ which means you know both p and q individually[11] (by the quadratic formula).

RSA Implementation

Named after the inventors (Rivest, Shamir, Adleman), RSA depends on the Integer Factorisation problem.

- *Bob*⇒Chooses a $p, q \in$ primes to construct $N = p \cdot q$. e is then selected such that $\gcd(e, (p-1) \cdot (q-1)) = 1$[12]

[10] Of course, d will only exist if $\gcd(e, \big((p-1) \cdot (q-1)\big)/g) = 1$

[11] There is some concern that this might not be completely true. Although this paper, states that "an oracle for breaking RSA does not help in factoring integers" but they couldn't prove any weakness in RSA because of that.

[12] There are some more nuances in selecting e which are discussed later.

- *Alice*⇒Converts her plaintext to an integer m such that $1 \leq m < N$. She then sends $c \equiv m^e \bmod (N)$ to *Bob*.
- *Bob*⇒Finds d using $d \cdot e \equiv 1 \bmod \big((p-1)\cdot(q-1)\big)$ and calculates $c^d \equiv m^{e\cdot d} \equiv m \bmod (N)$.
- *Eve*⇒This whole while had m^e but inversion required her to factor N which is a difficult task without an easy algorithm as is the case with DLP.

4.3.1 Nuances in choosing e

Keep in mind that choosing e and d actually decides the time it takes for encryption and decryption respectively. In general, choosing a small e gives you a large d, and vice versa (unless of course you go for $e = d = 1$).

- We can choose a small e $\Rightarrow$ large $d \Rrightarrow$ Ensures faster encryption, but slower decryption. →There is some concern that a small e might cause issues so a lot of people prefer to use $e = 2^{16} + 1$[13]. But there is no evidence that $e = 3$[14] is any less secure than a larger e.
- We can choose a small d $\Rightarrow$ large $e \Rrightarrow$ Ensures faster decryption, but slower encryption. →This is actually extremely insecure as $d < N^{\frac{1}{4}}$ can be easily decrypted using theory of continued fractions.

It is not compulsory for *Eve* to factor N to solve a RSA cryptosystem but at the same time, we have yet not found any such alternative method. (Like how we found Collision Algorithms in DHP)

4.3.2 Security issues in RSA

Just like in DHP, there exist some security concerns with the RSA system which can just circumvent having to solve the underlying problem.

RSA Oracle Method

Eve asks *Alice* to authenticate her identity by sending her messages encrypted using the public e, N she has put up and asking her to tell her the decrypted/original message. This request makes some practical sense as it is feasible for *Eve* to be talking to someone who isn't *Alice* in which case this impostor won't be able to send her back the original text[a] If *Alice* decides to do so and sends back whatever gibberish she gets after decryption for a few messages, *Eve* can easily employ the following method to decrypt *Bob*'s messages that were meant for *Alice* but intercepted by *Eve*. *Bob* enrypts messages m as $m' \equiv m^e \bmod (N)$ and *Eve* gets her hands on it. *Eve* then compute $m'' \equiv k^e \cdot m' \bmod (M)$ where $k \in$ random constant and send m'' to *Alice*. She returns back $(m'')^d \equiv k \cdot m \bmod (N)$. Since $k \cdot m$ will be gibberish, *Alice* cannot read the original message and *Eve* can easily just get back m.

[13] Although this looks like a huge exponent for calculating m, it is just 4 squarings and 1 addition
[14] We cannot use $e = 2$ since $\gcd(p-1, q-1)$ for large p, q will be 2.

[a]By original message, we obviously mean *Alice* is expected to send *Eve* whatever she decrypted with her private key and then encrypt it with *Eve's* public key as that is standard practice.

A good practice that *Alice* can follow to not fall prey to such attacks is to not send back decrypted messages to any person for authentication purposes if they don't follow a specific format; especially if they are gibberish.

Multiple encryption exponents for the same modulo

If *Alice* decided to keep different keys for different sets of people in her online life, she can do that by changing both N and e or by just changing e for the same N. The second method is fatal to the security of her system though.
How? Let's say that the 2 exponents are e_1 and e_2. If somehow, there comes a situation wherein

Eve intercepts 2 ciphers c_1 and c_2 of the same message m, she can write[a]-

$$u \cdot e_1 + v \cdot e_2 = \gcd(e_1, e_2)$$

After finding suitable u and v, she can compute-

$$c_1^u \cdot c_2^v \equiv m^{e_1 \cdot u + e_2 \cdot v} \equiv m^{\gcd(e_1, e_2)} \bmod (N)$$

If e_1 and e_2 are relatively prime, we can directly find the plaintext m. Note that if $\gcd(e_1, e_2) \neq 1$,

we still have to solve $x^\alpha \equiv h \bmod (N)$ which is pretty difficult even if $\alpha \ll e_1, e_2$[b].

[a]Using the property of gcd
[b]As was discussed earlier, there is no evidence $e = 3$ is any easier to solve than a larger e.

Although this method requires a lot of things to line up (*Bob* verifiably sending the same message via 2 exponents, e_1 and e_2 being coprime), it is a weakness that *can* manifest unknowingly if *Alice* decided to make even more keys. Hence it is good practice to use a different modulo for every key.

Primality testing

Before *Bob* can begin communication with anyone in the RSA system, he has to choose a $p, q \in$ very very large primes.

- If he goes for small primes, today's computers can easily factorise N.
- If either one of p and q ends up as composite with small prime factors, *Eve* can easily work with CRT to bring down the system's security.
- If either one of p and q is composite with large prime factors, *Bob* will have to spend computation time in trying to find the factors else he won't be able to decrypt *Alice's* messages.

Now for us to satisfy all of these conditions, *Bob* first will choose a fairly large number and then check if it is prime and, if not, do the same with the next number until he finds a prime.

But once he has a number, how can he verify that it is prime?

- ⋆ The first idea that can come to mind is to naively check if all numbers from 2 to $\sqrt{n}$ are coprime with n.
  This however is $\mathcal{O}(\sqrt{n}) \Rightarrow$ Exponential time.

4.4.1 Fermat Primality test

So we go with exploiting Fermat's Little Theorem[15]. We know that for every prime number p,

$$a^p \equiv a \bmod (p)$$

This is **not compulsorily** false for composite numbers.
e.g. $2^{341} \equiv 2 \bmod (341)$ although $341 = 11 \cdot 31$.

We then define the concept of "witnesses".

4.4.1.1 Witnesses

For a number n, any a such that

$$a^n \not\equiv a \bmod (n)$$

is called the "witness" of n.

After checking a large number of candidates $\{k_1, k_2, k_3, \ldots, k_l\}$, if we cannot find a witness for n, we can be reasonably sure that n is prime; but we cannot be 100% certain.

Keep in mind that there exists a subset of numbers called Carmicheal Numbers which do not have any witnesses despite being composite. e.g. 561
There exist ∞ly many such numbers and hence, even if you couldn't find a single witness, it doesn't prove primality.

4.4.2 Miller-Rabin test for Primality

The Miller-Rabin test depends on the following propostion.

Proposition

If $p \in$ odd primes, and we write $p = 2^k \cdot q + 1$, where $q \in$ odd integers, we can say that the following it will satisfy either of the 2 following properties-

[15] We will use a slightly different version. Fermat's "Little" theorem states that for $a \nmid p$, $a^{p-1} \equiv 1 \bmod (p)$. This puts a restrictions which can be lifted by instead denoting it as $a^p \equiv a \bmod (p)$.

1. $a^q \equiv 1 \bmod (p)$

OR

2. One element in the series $a^q, a^{2q}, a^{4q}, \ldots, a^{2^{k-1}q} \bmod (p)$ is congruent to -1.

Proof

Consider $a^{q \cdot 2^k} \bmod (p)$. It is obviously equivalent to $a^{p-1} \equiv 1 \bmod (p)$. Also all of the terms in the series $a^q, a^{2q}, a^{4q}, \ldots, a^{2^k q} \bmod (p)$ are actually in series of squares so if the last term is 1, the only possibility is that $a^{q \cdot 2^{k-1}} \equiv \pm 1 \bmod (p)$.

- If $a^{q \cdot 2^{k-1}} \equiv -1 \bmod (p)$, condition 2 is satisfied.
- If $a^{q \cdot 2^{k-1}} \equiv +1 \bmod (p)$, $a^{q \cdot 2^{k-2}} \equiv \pm 1 \bmod (p)$
  - If $a^{q \cdot 2^{k-2}} \equiv -1 \bmod (p)$, condition 2 is satisfied.
  - If $a^{q \cdot 2^{k-2}} \equiv +1 \bmod (p)$, $a^{q \cdot 2^{k-3}} \equiv \pm 1 \bmod (p)$
    - If $a^{q \cdot 2^{k-3}} \equiv -1 \bmod (p)$, condition 2 is satisfied.
    - If $a^{q \cdot 2^{k-3}} \equiv +1 \bmod (p)$, $a^{q \cdot 2^{k-4}} \equiv \pm 1 \bmod (p)$

$\vdots$

$a^q \equiv 1 \bmod (p)$ and thus condition 1 is satisfied.

Now let us develop the algorithm for using this primality test.

4.4.2.1 Miller-Rabin witnesses

A Miller-Rabin witness (a) will satisfy BOTH the following 2 conditions-

1. $a^q \not\equiv 1 \bmod (p)$
2. $a^{q \cdot 2^i} \not\equiv -1 \bmod (p) \quad \forall i \in [0, k-1]$

Digesting the meaning of witness here

We know from Fermat's primality test, that if $a^{p-1} \equiv 1 \bmod (p)$, then p is probably a prime number. So if either one of the 2 congruences was true, obviously $p \rightarrow$ probably prime.
Hence the definition of a Miller-Rabin Witness means that if the 2 properties are followed (i.e. both congruences are false), the number p is definitely composite.

Algorithm for Miller-Rabin test

1. To check n's primality, choose a value of $2 \leq a \leq n-1$ (candidate for a witness) and find q and k

such that,

$$n = 2^k \cdot q + 1$$

2. Evaluate values of $a^q, a^{2\cdot q}, a^{4\cdot q}, \ldots, a^{2^{k-1}\cdot q}$ mod (p) (use fast exponentiation) and check if a is a valid Miller-Rabin witness.
3. If a-
   (a) is a witness- n is definitely composite
   (b) is not a witness- n is probably prime. If so, choose another value of a and try again for better resolution.

But what makes Miller-Rabin test better than Fermat test?

At least 75% of the bases from 2 to $n-1$ are Miller-Rabin witnesses if $n \in$ composite. Hence even Carmicheal numbers get caught by this algorithm even though Fermat cannot find even a single witness for them.
So if you chose 10 bases and got not a single Miller-Rabin witness, the probabilty of it not being a prime number is $(0.25)^{10} \approx 9.54 \times 10^{-7}$. (If there are 0 witnesses in 100 bases selected, probability of it being composite$= 10^{-60}$)

One interesting fact is that **every** composite number has a $\frac{1}{4}$ chance of passing the Miller-Rabin test as a "probably prime number" for a specific base.
Fermat primality test has the same statistics for **ever non-Carmicheal** composite number.

Illustration

Check if $n = 172947529$ is prime or not

$$n - 1 = 172947528 = 2^3 \cdot 21618441 \quad \rightarrow \quad k = 3 \text{ and } q = 21618441$$

Choose $a = 3 \Rrightarrow$

$$3^{21618441} \equiv -1 \bmod (172947529)$$

So $a = 3$ is not a witness.
Let us try $a = 17$-

$$17^{21618441} \equiv 1 \bmod (172947529)$$

Again not a witness. We might suspect that 172947529 might actually just be a prime. But on further analysis with $a = 23$, we get-

$$23^{21618441} \equiv 40063806 \bmod (172947529)$$

$$23^{2\cdot 21618441} \equiv 2257065 \bmod (172947529)$$

$$23^{4\cdot 21618441} \equiv 1 \bmod (172947529)$$

In this whole sequence, not once did we get congruence of -1 and the first term was not congruent to 1. Hence 23 is actually a witness of 172947529.
But wait.

Isn't $23^{8 \cdot 21618441} \equiv 2257065 \bmod (172947529)$ here?
Yes.

Doesn't that mean that it is "probably prime" according to Fermat primality?
Yes. But that is irrelevant.
All prime numbers don't have either witnesses but **some** composite numbers would give a "false-positive" with Fermat but not so with Miller-Rabin.

Factorisation algorithm

Till now we have discussed how we need to choose large p, q while constructing N and also discussed a bit about how to choose large primes without wasting too much computation in determining if it is prime or not. Yet, choosing large primes, despite all the optimisations, remains a computationally heavy task. In that case, it is of significant use to understand the algorithms that *Eve* can use to factorise N and try to optimise our order of p, q in accordance.

The easiest algorithm we can study is the Pollard's $p-1$ factorisation algorithm.

4.5.1 Pollard's $p-1$ factorisation algorithm

Let $N = p \cdot q \quad p, q \in$ primes.

- Assume for a second that you can choose a L such that $(p-1)|L$ but $(q-1) \nmid L$.
- Then we can write that $a^{p-1} \equiv 1 \bmod (p) \Rightarrow a^L \equiv 1 \bmod (p)$.
- Thus we can say that $p|(a^L - 1)$ but at the same time $q \nmid (a^L - 1)$.
- Since $p \in$ primes, obviously
$$p = \gcd(a^L - 1, N)$$

But how do you find this famed L?
Well you can take the reasonable assumption that $(p-1)$ has a lot of small prime factors. In that case, if we chose a random large n, we can expect $p-1|n!$ thus we can take $L = n!$.

Now obviously, the roots of $(q-1)$ are also allowed to exist in $n!$ by happenstance so in the case that $(q-1) \cdot (p-1)|n!$, we might get $\gcd(a^{n!} - 1, N) = N$[16] which is completely useless.

[16] Essentially $a^{n!} \equiv 1 \bmod (p)$ and $a^{n!} \equiv 1 \bmod (q)$ would be both true simultaneously hence $a^{n!} \equiv 1 \bmod (p \cdot q) \equiv 1 \bmod (N)$

So we will definitely have to vary $n!$ a little bit in the hopes that only one of $p-1$ or $q-1$ is completely able to divide $n!$. But there is a good chance that it will be useless. So in that case, we just vary a[17].

You might be worried that you will have to evaluate gcd of N and a larger number $a^{n!}-1$ but you can actually use Euclid's algorithm for gcd and then find the $\gcd(N, a^{n!}-1 \bmod (p))$. Yay!

[17] How does this change gcd in any meaningful way?
Essentially we can write $a^{n_1!}-1 = \kappa_1(p-1)\cdot(q-1)$. But if we changed n_1 to n_2, since there is a chance the primes factors of both $(p-1)$ and $(q-1)$ are still both present in $n_2!$ i.e. $a^{n_2!}-1 = \kappa_2(p-1)\cdot(q-1)$
However if we changed a_1 to a_2, it might be possible that $(q-1)|a_2$ in which case $a_2^{n!} \equiv 0 \bmod (q)$ hence $a_2^{n!}-1$ is just a multiple of $(p-1)$

Addendum

Recursive algorithm for Extended Eucleadian Theorem

The extended Eucleadian Theorem helps us find modular inverses in linear time. Here is a small code snippet showing that using recursion.

```
// C++ program to find multiplicative modulo
// inverse using Extended Euclid algorithm.
#include <iostream>
using namespace std;

// Function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int* x, int* y);

// Function to find modulo inverse of a
void modInverse(int a, int m)
{
  int x, y;
  int g = gcdExtended(a, m, &x, &y);
  if (g != 1)
    cout << "Inverse doesn't exist";
  else
  {

    // m is added to handle negative x
    int res = (x % m + m) % m;
    cout << "Modular multiplicative inverse is " << res;
  }
}

// Function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int* x, int* y)
{

  // Base Case
  if (a == 0)
  {
    *x = 0, *y = 1;
    return b;
  }

  // To store results of recursive call
  int x1, y1;
  int gcd = gcdExtended(b % a, a, &x1, &y1);

  // Update x and y using results of recursive
  // call
  *x = y1 - (b / a) * x1;
  *y = x1;

  return gcd;
}

// Driver Code
int main()
{
  int a = 3, m = 11;

  // Function call
  modInverse(a, m);
  return 0;
}

// This code is contributed by khushboogoyal499


```

Bibliography

[1] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *An Introduction to Mathematical Cryptography*

[2] Hypr.com (https://www.hypr.com/black-cipher/)

[3] Joshua Holden, *THE MATHEMATICS OF SECRETS*

[4] Jonathan Katz and Yehuda Lindell, *Introduction to Modern Cryptography*

[5] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *HANDBOOK of APPLIED CRYPTOGRAPHY*

[6] Simon Singh, *"The Code Book- HOW TO MAKE IT, BREAK IT, HACK IT, CRACK IT"*

[7] Dheeraj & Nithin, Goldwasser-Micali Cryptosystem

[7] XKCD comics