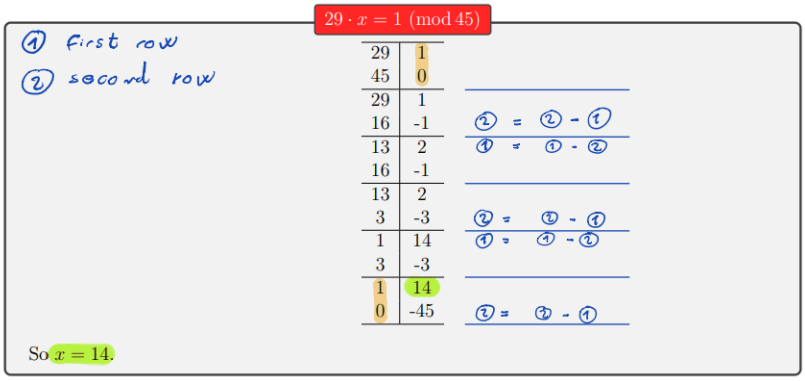
1 How to compute the inverse a · x = 1 (mod n)

In this lesson is presented a recap of how to compute the inverse in a given ring. Are shown three methods derived from the maths theory plus a fourth one which is an algorithmic one (is not only one not computationally feasible). For all the methods is shown the example : 29 · x = 1 (mod 45)

1.1 Gauss-Jordan like method

The first method described is the Gauss-Jordan like method, that consists in considering the modulo equation as column of a matrix. To do the inverse you have to associate an identity-vector-column and try to get the identity instead of your initial column, at the end on the right side, as first element of the vector, you obtain the inverse. The method is described in the figure below.



1.2 Kuttaka method

Here is presented an example of usage of the Kuttaka method.

The problem is:

$$29x = 1(mod 45)$$

Step 1: Mutual divisions

- Divide 45 by 29 to get quotient 1 and remainder 16 (q=1 r=16)
- Divide 29 by 16 to get quotient 1 and remainder 13 (q=1 r=16)
- Divide 16 by 13 to get quotient 1 and remainder 3 (q=1 r=16)
- Divide 13 by 3 to get quotient 4 and remainder 1 (q=1 r=16)
- Divide 3 by 1 to get quotient 3 and remainder 0 (q=1 r=16)
- The process of mutual division stops here.

Quotients = 1, 1, 1, 4, 3
Number of quotients = 4(an even integer) (excluding the first quotient)

Step 2: Computation of last minus one quotient Let us assume that the last minus one quotient obtained is called c. The right c to be inserted in the next step of the computation should be equal to compute as follow: if (Number of quotients is an even number) the new c = (c+1) else (so the number of quotient is an odd number) the new c = (c-1)
In our case c = 4 = 3 + 1

Step 3: Computation of successive numbers

Write elements of 1st column	: 1, 1, 1, 4, 4, 1
Compute elements of 2nd column	: 1, 1, 1, 17, 4
Compute elements of 3rd column	: 1, 1, 21, 17
Compute elements of 4th column	: 1, 38, 21
Compute elements of 5th column	: 59, 38

Where 59 (mod 45) = 14, which is our result

The computational procedure is shown below:

Quotient 0 : 1 1 1 1 59
(38×1 + 31 = 59)↗
Quotient 1 : 1 1 1 38 38
(21×1 + 17 =38)↗
Quotient 2 : 1 1 21 21
(17×1 + 4 = 21)↗
Quotient 3 : 4 17 17
(4×4 + 1 = 17)↗
Quotient 4 : 4 4

1

1.3 Extended Euclidean Algorithm

The Extended Euclidean Algorithm is based on the concept of the Great Common Divisior, in particular is true that : if

$$b = aq + r$$

then

$$gcd(a, b) = gcd(r, a).$$

The method follows two rules :

- Forward rule :

$$(a, b) \xrightarrow{q} (r, a)$$

where r = b (mod a) and q = a // b (// is the integer division)

- Backward rule :

$$(a - qr, r) \overset{q}{\leftarrow} (r, a)$$

In our example : $29 \cdot \mathrm{x} = 1 \pmod{45}$

$$(29, 45) \overset{1}{\rightarrow} (16, 29) \overset{1}{\rightarrow} (13, 16) \overset{1}{\rightarrow} (3, 13) \overset{4}{\rightarrow} (1, 3) \overset{3}{\rightarrow} (0, 1)$$

$$(14, -9) \overset{1}{\leftarrow} (-9, 5) \overset{1}{\leftarrow} (5, -4) \overset{1}{\leftarrow} (-4, 1) \overset{4}{\leftarrow} (1, 0) \overset{3}{\leftarrow} (0, 1)$$

Again 14 is our result

1.4 Python scrypt

6.2.1 Inverse in Python

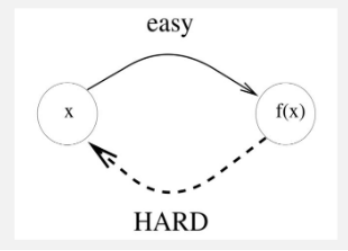
```
''' inverse(r,N) = x ,
          such that r x = 1 (mod N).

'''
def inverse(r,N):
    x=1
    while (r*x)%N != 1 and  x<N:
            x+=1
    if  (r*x)%N == 1:
        return x
    else: return 'non invertible'
```

2 One-way functions and trapdoors

2.1 One-way property (or preimage resistance)

A function $f : D \rightarrow \mathrm{T}$ is one-way if for $x \in D$ the value $f(x)$ can be computed efficiently, but for any $y \in T$, it is not computationally feasible to find $x \in D$ such that $f(x) = y$



In crypto are not used function that are not invertible at all, this concept of one-way is based on conjectures.

2.1.1 Trapdoor block-cipher

One-way functions f can be obtained by using a secure block-cipher (Enc,Dec).

Example 1) :

$$f : K \to C$$

from the key space to the cipher-text space is defined as follows: pick a random plain-text P_0 and put

$$f(k) := Enc_k(P_0)$$

So, since encryption should be efficient we get that computing $f(k)$ is easy. But finding k for a given C is crypto analysis which should be hard.

Example 2) :

$$f : P \to C$$

from plain-text space to cipher-text space defined as

$$f(P) := Enc_k(P)$$

Key **k** is regarded as a **trapdoor** which is the special information which makes easy the inversion.

Well-known examples are RSA and Rabin functions $f(x) = x^e \mod n$ and $f(x) = x^2 \mod n$. The trapdoor is the factorization of n i.e. the two prime numbers p, q such that $n = pq$. For the Rabin's function keep in mind the role of CRT (Chinese Remainder Theorem) + Tonelli-Shanks' algorithm.

If we take for example the Rabin function and we suppose to know the factorization of $n = pq$, is computationally feasible to compute x such as

$$\begin{cases} x^2 = y \mod p \\ x^2 = y \mod q \end{cases}$$

using the Cipolla's method, or the Tonelli's one, the with CRT computing the inverse.

2.2 Discrete Logarithm

A typical example of one-way function is the power map$f : \mathbb{Z}_n \to < \alpha >$

$$\gamma \to \alpha^\gamma$$

where α has order n. Usually, the best way to solve $f(\gamma) = y$ is related to the Birthday's Paradox, an algorithm of complexity O(n): If $\beta = \alpha^\gamma$ and

$$\alpha^a \beta^b = \alpha^A \beta^B \text{ in } < \alpha >$$

(The computation above is regarded as a collision in which our have same result but different exponent) then:

you should find a,b,B,A in $\mathbb{Z}_n$

$$\alpha^a \alpha^{\gamma b} = \alpha^A \alpha^{\gamma B}$$

$$\alpha^{a+\gamma b} = \alpha^{A+\gamma B}$$

$$a + \gamma b = A + \gamma B \mod n$$

$$\gamma = \frac{(A-a)}{(b-B)} \mod n$$

A,B,a,b are exponents and belongs to $\mathbb{Z}_n$ and not to $< \alpha >$, so the belongs to the domain of the transformation and not to the co-domain.

Ex. $\mathbb{Z}_n^* = \{1,2,3,4,5,6\}$, $p = 7$. If we take $\alpha = 2$ then $< \alpha >= \{2,4,1\}$ with $n = 3$

$$f : \mathbb{Z}_n^* \to \mathbb{Z} < \alpha > \subset \mathbb{Z}^*{}_p$$

This is the one-way function used by Diffie-Hellman in his key exchange cryptosystem.

2.3 Factorization and CRT

Another one-way is the map which takes two prime numbers p, q and gets the product $N = pq$.Namely, it is computationally efficient to multiply pq but is computationally expensive to compute both factors p and q from N.

This is the one-way function that together with the CRT it is used in RSA.

2.4 Computing with encrypted data

Another way of using the one-way function is with encrypted data. To understand this thing we have to put in this scenario:

Alice has a particular value x and wants to compute $f(x)$ but either her computer is broken or it has not enough computational power.Bob is willing to compute $f(x)$ for her, but Alice is not keen on letting Bob know her x. In such situation Bob behaves as an **oracle** i.e. he answer questions.

A way of encrypting the for the oracle is using the discrete logarithm as described in the figure below.

7.1.5 Computing with DL encrypted data

Let p a prime number, let g a generator and $f(x)$ be the corresponding discrete logarithm. Namely, if $x = g^e \pmod p$ then

$$f(x) = e$$

To get e without revealing x **Alice** generate a random $r \in \{1, \cdots, p-1\}$, computes $\tilde{x} = x \cdot g^r$ and ask **Bob** $f(\tilde{x}) = \tilde{e}$.

Then **Alice** recover e from

$$e + r = \tilde{e} \pmod{p-1}$$

From the figure above it is easy to understand how Alice is to obtain her computation without letting the oracle know her data.

3 Hash functions

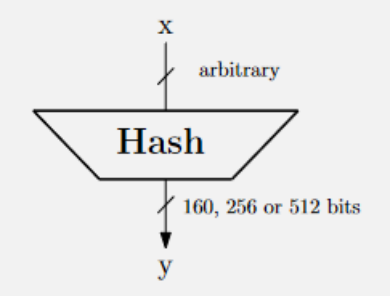
A cryptographic hash function (CHF) is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function which is practically infeasible to invert.

$$\textbf{Hash} : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$$

$\mathbb{Z}_2^*$ contains any possible finite set of bits (so any sequence of bits of any length)

$\mathbb{Z}_2^n$ can be considered as the product of $\mathbb{Z}_2$ for n times. It represents the set of all strings of bits of length n,

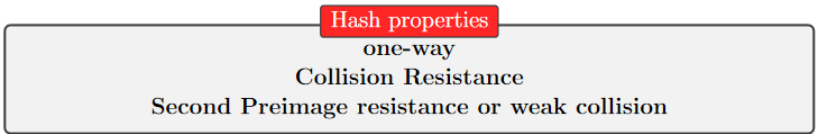
The digest is usually of size $n = 160, 256, 512$ bits. The hash value is computed as $y = \textbf{Hash}(x)$.



"Hash functions-such as MD5, SHA-1, SHA-256, SHA-3, and BLAKE2 - comprise the cryptographer's Swiss Army Knife: they are used in digital signatures, public-key encryption,integrity verification, message authentication, password protection, key agreement protocols, and many other cryptographic protocols" (from Aumasson18, Chapter 6)

Hash functions with full (co)domain or with a parameter for the length of the digest can be also useful in applications e.g.SHAKE128(M,d) and SHAKE256(M,d).

The properties of hash function are illustrated in the figure below :



In particular :

- One-way : should be computationally infeasible to invert
- Collision resistance : given $x, y \in \mathbb{Z}_2^*$ should be computationally infeasible to have $\mathbf{Hash}(x) = \mathbf{Hash}(y)$
- Second Preimage resistance or weak collision : if you have $\mathbf{Hash}(x) = y$ should be computationally infeasible to have $\tilde{x}$ such as $\mathbf{Hash}(\tilde{x}) = y$

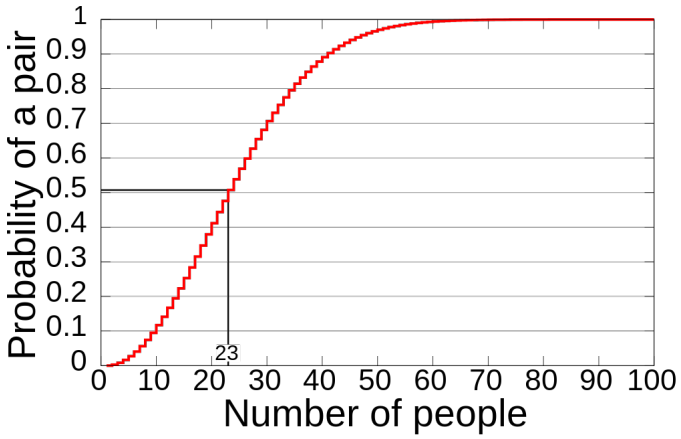
3.1 Collision: Birthday attack

The so called Birthday attack is based on the Birthday paradox. In probability theory, the birthday problem or birthday paradox concerns the probability that, in a set of n randomly chosen people, some pair of them will have the same birthday. In a group of 23 people, the probability of a shared birthday exceeds 50%, while a group of 70 has a 99.9% chance of a shared birthday. (By the pigeonhole principle, the probability reaches 100% when the number of people reaches 367, since there are only 366 possible birthdays, including February 29.)

These conclusions are based on the assumption that each day of the year is equally probable for a birthday. Actual birth records show that different numbers of people are born on different days. In this case, it can be shown that the number of people required to reach the 50% threshold is 23 or fewer.

The birthday problem is a veridical paradox: a proposition that at first appears highly counter intuitive, but is in fact true. While it may seem surprising that only 23 individuals are required to reach a 50% probability of a shared birthday, this result is made more intuitive by considering that the comparisons of birthdays will be made between every possible pair of individuals. With 23 individuals, there are $23 \times 22 \ / \ 2 = 253$ pairs to consider, which is well over half the number of days in a year (182.5 or 183).

The representative curve is shown below:



The computation of our interest is when a collision occurs with a probability greater than 50%. Called Γ_N the probability of having a collision with N

elements, we want to know when $\Gamma_N = \frac{1}{2}$. To study this is necessary first to underline that our domain is composed by N elements whereas our co-domain is composed by 365 elements (days in a year). Defined $1 - \Gamma_n$ as the probability of not having a collision, we can proceed as follows:

$$1 - \Gamma_n = N! \frac{\binom{365}{N}}{365^N}$$

In the previous fraction at the numerator can be interpreted as from 365 days I take N days and I consider all the possible reordering, whereas at the denominator the number of possible maps from N to 365.

Doing some computation we arrive at the result:

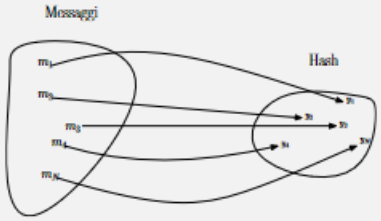
$$N = \sqrt{365 \times 2 \times \log 2} = 23$$

This number tells us that with 23 people the probability that 2 people has the same birthday if 50%

All the general computations for the birthday attack are presented in the figure below:

Collisions: Birthday attack

Assume a digest of n bit. With N messages $x_1, \cdots x_N$ we can form $\frac{N(N-1)}{2}$ pairs which are candidates for a collision. So it is natural to guess that the probability λ_N of a collision between two of these $\frac{N(N-1)}{2}$ pairs of messages would be related to the numbers $2^{\frac{n}{2}}$ and 2^n.



Here the equation relating λ_N and the digest length:

$$N \approx 2^{\frac{n+1}{2}} \cdot \sqrt{\ln\left(\frac{1}{1-\lambda_N}\right)}$$

The above equation follows from the following discussion.
A collision between the N messages is produced when the restriction of the **Hash** function to the set of N messages is no more injective. So the probability $1-\lambda_N$ of **no collision** is the quotient between the number of injective functions and the number of all possible functions:

$$1-\lambda_N = \frac{N! \cdot \binom{2^n}{N}}{(2^n)^N} = \frac{2^n!}{(2^n-N)! \cdot (2^n)^N} = \frac{2^n \cdot (2^n-1) \cdot (2^n-2) \cdots (2^n-(N-1))}{(2^n)^N} =$$

hence

$$\lambda_N = 1 - (1-\frac{1}{2^n}) \cdot (1-\frac{2}{2^n}) \cdots \left(1-\frac{(N-1)}{2^n}\right)$$

by using $e^{-x} \approx 1-x$ for x close to zero we get:

$$\lambda_N \approx 1 - e^{-\frac{1}{2^n}} \cdot e^{-\frac{2}{2^n}} \cdots e^{-\frac{N-1}{2^n}} = 1 - e^{-\frac{N \cdot (N-1)}{2 \cdot 2^n}}$$

and from this we get:

$$N \approx 2^{\frac{n+1}{2}} \cdot \sqrt{\ln\left(\frac{1}{1-\lambda_N}\right)}$$

Here there is an extract from *KatLin15,page 168* on finding meaningful collisions.

Finding meaningful collisions. The algorithm just described may not seem amenable to finding meaningful collisions since it has no control over the elements sampled. Nevertheless, we show how finding meaningful collisions is possible. The trick is to find a collision in the right function!

Assume, as before, that Alice wishes to find a collision between messages of two different "types," e.g., a letter explaining why Alice was fired and a flattering letter of recommendation that both hash to the same value. Then, Alice writes each message so that there are $\ell - 1$ interchangeable words in each; i.e., there are $2^{\ell-1}$ messages of each type. Define the one-to-one function $g : \{0,1\}^\ell \to \{0,1\}^*$ such that the ℓth bit of the input selects between messages of type 0 or type 1, and the ith bit (for $1 \le i \le \ell - 1$) selects between options for the ith interchangeable word in messages of the appropriate type. For example, consider the sentences:

> 0: Bob is a *good/hardworking* and *honest/trustworthy worker/employee.*
> 1: Bob is a *difficult/problematic* and *taxing/irritating worker/employee.*

Define a function g that takes 4-bit inputs, where the last bit determines the type of sentence output, and the initial three bits determine the choice of words in that sentence. For example:

$$g(0000) = \text{Bob is a good and honest worker.}$$
$$g(0001) = \text{Bob is a difficult and taxing worker.}$$
$$g(1010) = \text{Bob is a hardworking and honest employee.}$$
$$g(1011) = \text{Bob is a problematic and taxing employee.}$$

Now define $f : \{0,1\}^\ell \to \{0,1\}^\ell$ by $f(x) \stackrel{\text{def}}{=} H(g(x))$. Alice can find a collision in f using the small-space birthday attack shown earlier. The point here is that any collision x, x' in f yields two messages $g(x), g(x')$ that collide under H. If x, x' is a random collision then we expect that with probability $1/2$ the colliding messages $g(x), g(x')$ will be of different types (since x and x' differ in their final bit with that probability). If the colliding messages are not of different types, the process can be repeated again from scratch.