• An algorithm is a sequence of steps which, starting from an input, produces an output.

• It is important to estimate the *complexity* ("running time") of an algorithm, namely the amount of bit operations which produce the output, in terms of the size of the input. This estimate only depends on the algorithm and not on the computer used to run it. Generally we are interested in the asymptotic behaviour of the complexity when the size of the input "goes to infinity".

• In our context the input generally consists of positive integers $n_1, \ldots, n_s$. The size of a positive integer $n$ is measured by the number of bits of its binary expansion $a_k a_{k-1} \ldots a_0$, which is given approximately by $\log n := \log_2 n$. In fact, one has

$$2^k \leq n = 2^k a_k + \ldots + 2a_1 + a_0 \leq 2^{k+1}, \qquad a_j \in \{0, 1\}, a_k \neq 0,$$

from which it follows that

$$k \leq \log n \leq k + 1.$$

The number of bits of $n$ is about 3 times the number of decimal digits of $n$: $\log_2 n \sim 3 \log_{10} n$.

• An algorithm is *polynomial time* if the complexity is bounded by a polynomial function in the size of the input, i.e. it is $\mathcal{O}(\log n_1^{a_1} \cdot \ldots \cdot \log n_s^{a_s})$, for some fixed $a_j \in \mathbf{N}$ (independent of $n_1, \ldots, n_s$); it is *exponential time* if the complexity is $\mathcal{O}(n_1^{a_1} \cdot \ldots \cdot n_s^{a_s})$, for some fixed $a_j > 0$; it is *subexponential time* if the complexity is $o(n_1^{a_1} \cdot \ldots \cdot n_s^{a_s})$. An example is $\mathcal{O}(2^{(\log n_1)^{\alpha_1}} \cdot \ldots \cdot 2^{(\log n_s)^{\alpha_s}})$, for some fixed $0 < \alpha_j < 1$.

• The crucial difference between polynomial and exponential algorithms is the following. Suppose the input is a positive integer $n$. In a *polynomial time* algorithm, if we double the size of the input, then the amount of calculations to obtain the output is multiplied by a constant factor, *independent of the size of the input*. For example, if the complexity is $\mathcal{O}((\log n)^k)$, then

$$(2 \log n)^k = 2^k (\log n)^k.$$

in an *exponential time* algorithm, if we double the size of the input, then the amount of calculations is multiplied by factor which is *exponential in the size of the input*. For example, if the complexity is $\mathcal{O}(2^{k \log n})$, for $k \in \mathbf{N}_{\geq 1}$, then

$$2^{k\, 2 \log n} = 2^{k \log n} \cdot 2^{k \log n}.$$

Consequently, the amount of calculations explodes, when $\log n \to \infty$.

• The algorithms which we describe involve mainly arithmetic operations in $\mathbf{Z}$ or in $\mathbf{Z}_n$.

**Example 1**. The complexity of the arithmetic operations in $\mathbf{Z}$ is polynomial in the size of the input. Let $n, m \in \mathbf{N}$, with $n \geq m > 0$. Let $a_k a_{k-1} \ldots a_0$ and $b_h b_{h-1} \ldots b_0$ be the binary expansions of $n$ and $m$, respectively. Then

$(a)$ $n \pm m$ $\quad \mathcal{O}(\log n + \log m) = \mathcal{O}(\log n)$;

$(b)$ $n \cdot m$ $\quad \mathcal{O}(\log n \log m) = \mathcal{O}((\log n)^2)$;

$(c)$ $n : m$ $\quad \mathcal{O}(\log n \log m) = \mathcal{O}((\log n)^2)$;

*Proof*: (a) The sum $n+m$

$$\begin{array}{r} a_k a_{k-1} \dots\dots a_0 + \\ b_h b_{h-1} \dots b_0 \end{array}$$

is obtained by $\log m$ sums and at most $\log n$ carryings (we sum the bits by the rules $0+0=0$, $1+0=0+1=1$ and $1+1=10$). Hence it has complexity $\mathcal{O}(\log n + \log m) = \mathcal{O}(\log n)$. The opposite $-n$ is obtained by switching 0's and 1's in the binary expansion of $n$ and then adding 1, and difference $n-m$ is obtained summing $n$ and $-m$. Consequently, both $-n$ and $n-m$ are $\mathcal{O}(\log n)$.

(b) The product $n \cdot m$

$$\begin{array}{r} a_k a_{k-1} \dots\dots a_0 \times \\ b_h b_{h-1} \dots b_0 = \\ - - - - - - - - - - - - - \\ l_k l_{k-1} \dots\dots l_0 \\ m_k m_{k-1} \dots\dots m_0 - \\ \dots\dots\dots\dots\dots\dots - \ - \end{array}$$

is obtained by adding $\log m$ strips of length $\log n$, each shifted by one place to the left (each strip either coincides with $a_k a_{k-1} \dots\dots a_0$ or it consists of 0's). Hence the complexity of the product is

$$O(\log n \cdot \log m) = O((\log n)^2).$$

(c) If $n = mq + r$, the quotient $q = c_p c_{p-1} \dots c_0$ is obtained by $\log q$ subtractions

$$\begin{array}{l} a_k a_{k-1} \dots\dots a_1 a_0 \ : \ b_h b_{h-1} \dots b_0 \ = \ c_p c_{p-1} \dots \\ b_h b_{h-1} \dots b_0 \\ \quad \dots \quad \dots \\ b_h b_{h-1} \dots b_0 \\ \quad \dots \quad \dots \end{array}$$

Since $\log q \le \log n$ and $\log m \le \log n$, the complexity of the division is

$$\mathcal{O}(\log q(\log n + \log m)) = \mathcal{O}(\log n \log m) = \mathcal{O}((\log n)^2).$$

**Example 2**. The complexity of the Euclidean algorithm to compute the $\gcd(n,m)$, with $n \ge m > 0$, is $\mathcal{O}((\log n)^3)$.

*Proof*: Let $n \ge m > 0$ be positive integers. The Euclidean algorithm consists of a sequence of divisions with remainder:

$$\begin{array}{rll} m &= r_0 & \\ n &= q_0 m + r_1, & 0 < r_1 < m \\ r_0 &= q_1 r_1 + r_2, & 0 < r_2 < r_1 \\ r_1 &= q_2 r_2 + r_3, & 0 < r_3 < r_2 \\ \vdots &= \ \vdots & \\ r_{k-1} &= q_k r_k + r_{k+1}, & 0 < r_{k+1} < r_k \\ \vdots &= \ \vdots & \end{array}$$

The remainders form a decreasing sequence of non negative integers

$$0 \leq \ldots r_{k+1} < r_k < r_{k-1} < \ldots r_1 < r_0 = m,$$

whose smallest positive element is the greatest common divisor $gcd(n, m)$.

The first division requires about $\log n \log m$ operations, the second $\log m \log r_1$, etc ... Each division requires at most $\log n \log m$ operations. It remains to estimate how many divisions are necessary to obtain $\gcd(n, m)$.

**Lemma.** *Every two steps the remainder is cut in half:* $r_{k+2} \leq \frac{1}{2} r_k$.

*Proof*: If $r_{k+1} \leq \frac{1}{2} r_k$, then automatically $r_{k+2} < r_{k+1} \leq \frac{1}{2} r_k$ holds true. Suppose now that $r_{k+1} > \frac{1}{2} r_k$. From the equation $r_k = q_{k+1} r_{k+1} + r_{k+2}$ one deduces that $q_{k+1} = 1$. Then from $r_k = r_{k+1} + r_{k+2} > \frac{1}{2} r_k + r_{k+2}$, it follows that $r_{k+2} \leq \frac{1}{2} r_k$, as claimed.

The above lemma is very important. It implies that $\gcd(n, m)$ is obtained by at most $2 \log m$ divisions with remainder. Consequently the complexity of the Euclidean algorithm can be estimated as

$$\mathcal{O}(2 \log m \cdot \log n \log m) = \mathcal{O}((\log n)^3).$$

**Remark.** If the remainders were only decreasing by one unit at each step, then it would take $m = 2^{\log m}$ divisions to obtain $\gcd(n, m)$. In that case, the complexity of the algorithm would be exponential in $\log m$.

**Example 3**. The complexity of the arithmetic operations in $\mathbf{Z}_n$ is polynomial in $\log n$, for any input $x$, $y$ in $\mathbf{Z}_n$.

$(a)$ $x \pm y$ $\quad \mathcal{O}(\log n)$;

$(b)$ $x \cdot y$ $\quad \mathcal{O}((\log n)^2)$;

$(c)$ $x^{-1}$ $\quad \mathcal{O}((\log n)^3)$.

*Proof*: (a) and (b) follow directly from Example 1.

(c) For computing $x^{-1}$ in $\mathbf{Z}_n$, one uses the extended Euclidean algorithm, which both checks that $\gcd(x, n) = 1$, i.e. that $x$ is invertible modulo $n$, and produces integers $a$ and $b$ satisfying $ax+bn = 1$. Then $a \equiv x^{-1} \bmod n$ is the required inverse. With some work..., one can check that $|a| \leq n$, which implies that computing the residue class of $a$ modulo $n$ is $\mathcal{O}((\log n)^2)$. In total, the complexity of determining $x^{-1}$ in $\mathbf{Z}_n$ is

$$\mathcal{O}((\log n)^3) + \mathcal{O}((\log n)^2) = \mathcal{O}((\log n)^3).$$

**Example 4**. (computing powers by successive squarings, in $\mathbf{Z}$ and in $\mathbf{Z}_n$).

(a) Let $x$, $a \in \mathbf{N}$. The complexity of computing $x^a$ in $\mathbf{Z}$ is exponential in $\log a$.

(b) The complexity of computing $x^a$ in $\mathbf{Z}_n$ is given by

$$\mathcal{O}(\log a \, (\log x)^2 \, ).$$

Hence it is polynomial both in $\log x$ and $\log a$.

*Proof.* (a) The size of the power $x^a$ is

$$\log x^a = a \log x.$$

Writing down the output is already exponential in $\log a$. This gives a lower bound for the complexity of the whole calculation, which is therefore exponential in $\log a$.

To prove (b), we describe an efficient method of computing exponentiations in $\mathbf{Z}_n$. Let

$$a = 2^h a_h + \ldots + 2a_1 + a_0, \qquad \text{with } a_j \in \{0,1\},\ a_h \neq 0,$$

be the binary expansion of the exponent, where $h$ is approximately $\log a$. Then

$$x^a = x^{a_h 2^h + a_{h-1} 2^{h-1} + \ldots + a_1 2 + a_0} = \prod_{i=0, a_i \neq 0}^{h} x^{2^i}.$$

Since $x^{2^i} = x^{2^{i-1}} \cdot x^{2^{i-1}}$, all the powers up to $x^{2^h}$ are obtained by $h$ successive squarings modulo $n$

$$x^2 = x \cdot x \bmod n, \qquad x^{2^2} = x^2 \cdot x^2 \bmod n, \quad \ldots, \quad x^{2^h} = x^{2^{h-1}} \cdot x^{2^{h-1}} \bmod n.$$

Since each squaring modulo $n$ is $\mathcal{O}((\log n)^2)$, the complexity of the above computation is $\mathcal{O}(\log a (\log n)^2)$. Multiplying all of the powers together is $\mathcal{O}(\log a (\log n)^2)$. Consequently, the whole process has complexity

$$\mathcal{O}(\log a (\log n)^2 + \log a (\log n)^2) = \mathcal{O}(\log a (\log n)^2),$$

and it is *polynomial* both in $\log x$ and in $\log a$.

**The trial division or "naif" factoring algorithm.**

Let $n \in \mathbf{N}$ be a composite number. Then $n$ has at least a prime factor $p \leq \sqrt{n}$. The trial division factoring algorithm consists of dividing $n$ by all the primes $s \leq \sqrt{n}$, until the smallest prime factor $p$ is found (assuming that the list of primes up to $\sqrt{n}$ is already available).

By the Prime Number Theorem, $\pi(p)$, the number of primes $\leq p$, can be approximated with $\frac{p}{\ln p}$. To obtain the smallest prime factor $p$ of $n$, it takes $\frac{p}{\ln p}$ divisions with remainder $n : s$, for $s \leq p$. Then the complexity of this algorithm can be estimated as

$$\mathcal{O}(\frac{p}{\ln p} \log n \log p) = \mathcal{O}(p \log n) = \mathcal{O}(2^{\log p} \log n).$$

Hence, it is exponential in $\log p$, where $p$ is the smallest factor.

**Example 5.** Let $n = p * q$ be the product of two primes $p$ and $q$, with $p \ll q$. The above formulas show that if $p$ is increased by one digit, then the complexity is multiplied by a factor of 10. In the table below $n$ is the product of the 40 digits prime $q = 1000000000000098787979879687687687608810 7$ and a prime $p$ of 2, 3, ..., up to 19 digits.

```
[1, 1100000000000108666777867656545645636969177]
[2, 1010000000000099775859678484564564484898807]
[3, 1009000000000099677071698604876876797289996 3]
[4, 1000700000000098857131465603469068990136867 49]
[5, 1000030000000098790943519084078318239438964321]
[6, 1000003000000098788276243627326750671873526432 1]
[7, 1000001900000098788167576849459094215417156740 33]
[8, 1000000070000098787986794846279265746948832616749]
[9, 1000000007000098787980571203546845422624513261674 9]
[10, 1000000001900098787980067384849459015417306456740 33]
[11, 1000000000030098787979882651327083999441330628264321]
[12, 1000000000039098787979883540418902916630519816743617 3]
[13, 1000000000003798787979880053203213163655144444152599 59]
[14, 1000000000000408787979879718311961371513883183158731317]
[15, 1000000000000135787979879691342842864359144444441525995 9]
[16, 1000000000000104887979879688290294286076794894894413745 27]
[17, 1000000000000098817979879687690651248207090630630628264321]
[18, 1000000000000098790979879687687983972750339063063062826432 1]

1: 11 * 1000000000000098787979879687687687608810 7. time:0 ms
2: 101 * 1000000000000098787979879687687687608810 7. time:0 ms
3: 1009 * 1000000000000098787979879687687687608810 7. time:0 ms
4: 10007 * 1000000000000098787979879687687687608810 7. time:4 ms
5: 100003 * 1000000000000098787979879687687687608810 7. time:32 ms
6: 1000003 * 1000000000000098787979879687687687608810 7. time:175 ms
7: 10000019 * 1000000000000098787979879687687687608810 7. time:1,646 ms
8: 100000007 * 1000000000000098787979879687687687608810 7. time:17,397 ms
9: 1000000007 * 1000000000000098787979879687687687608810 7. time:3min, 22,293 ms

10: 10000000019 * 1000000000000098787979879687687687608810 7. time:33min, 24,023 ms
```

If it takes about 17.000 milliseconds to find $p$ of 9 digits (on a laptop), then to find $p$ of 19 digits it is likely to take more than 5 years. On the other hand, if we fix the smallest prime $p$ and increase the larger prime $q$ by one digit, then the running time remains almost the same.

**The Pollard $\rho$ factoring algorithm.**

The Pollard $\rho$ factoring algorithm was introduced by John M. Pollard in 1975. In 1980 a variation of this algorithm was successfully used by R.P. Brent and J.M. Pollard to factor the 78-digit Fermat number $F_8 = 2^{2^8} + 1 = 2^{256} + 1$

$$F_8 = 1238926361552897 \cdot p_{62}.$$

The Pollard $\rho$ algorithm is based on the birthday paradox and it is very efficient in determining the small factors of a given number.

**The birthday paradox**. *Let $X$ be a set with $N$ elements. Pick a random element from $X$. If this action is repeated $k$ times, then the probability $p(k)$ that some element is picked twice can be estimated as*

$$p(k) > 1 - e^{-k(k-1)/2N}.$$

*Proof.* The probability $p(k)$ that *some* element is picked twice is given by

$$p(k) = 1 - \frac{N(N-1)\dots(N-k+1)}{N^k} = 1 - (1 - \frac{1}{N}) \cdot \ldots \cdot (1 - \frac{k-1}{N}) = 1 - \prod_{i=1}^{k-1}(1 - \frac{i}{N}).$$

From the inequality $1 - x < e^{-x}$, for $x \neq 0$, one has

$$\prod_{i=1}^{k-1}(1 - \frac{i}{N}) < \prod_{i=1}^{k-1} e^{-\frac{i}{N}} = e^{-\sum_{i=1}^{k-1}\frac{i}{N}} = e^{-k(k-1)/2N},$$

from which the required estimate follows

$$p(k) > 1 - e^{-k(k-1)/2N}.$$

**Remark.** In particular, one obtains the estimates

$$p(k) > \frac{1}{2}, \quad \text{for } k > 1,177\sqrt{N}; \qquad p(k) > 1 - e^{-5} \sim 0.9932, \quad \text{for } k > 3,16\sqrt{N}.$$

In short, we may say that if we pick an element out of a set of $N$ elements $k$ times, then there is a positive probability to pick the same element twice.

Conversely, if we picked $k$ times an element out of a set of a unknown cardinality $N$ and we always got distinct elements, then $N$ was probably larger than $k^2$.

**The birthday example.** Let $X$ the set of the days of the year, of cardinality $\#X = 366$. The above estimates say that in a group of 60 people, the probability that two of them have the same birthday is $p(60) > 1 - e^{-354/73} \sim 0.9921$.

*How the birthday paradox is applied to the factorisation of an integer $n$?*

In this case the set $X$ is $\mathbf{Z}_n$, the group of residue classes modulo $n$. Let $\{x_j\}_{j=1}^k$ be a random sequence of length $k$ in $\mathbf{Z}_n$. For every divisor $p$ of $n$, the sequence descends to a random sequence in $\mathbf{Z}_p$.

• If $p < k^2$ then, by the birthday paradox, with high probability two elements of the sequence coincide in $\mathbf{Z}_p$. The smaller $p$ is with respect to $k^2$, the more probable is a coincidence to occur in $\mathbf{Z}_p$.

• Even *without knowing the factorisation of* $n$, a coincidence in some $\mathbf{Z}_p$ can be detected by taking

$$\gcd(x_i - x_j \bmod n, n), \quad i, j = 1, \ldots, k.$$

Indeed, one has $\gcd(x_i - x_j \bmod n, n) > 1$ if and only if there is a divisor $p$ of $n$ for which $x_i = x_j \bmod p$. *In this way* $\gcd(x_i - x_j \bmod n, n)$ *will provide a non trivial factor of* $n$. (it is unlikely that $\gcd(x_i - x_j \bmod n, n) = n$, unless $k \gg \sqrt{n}$).

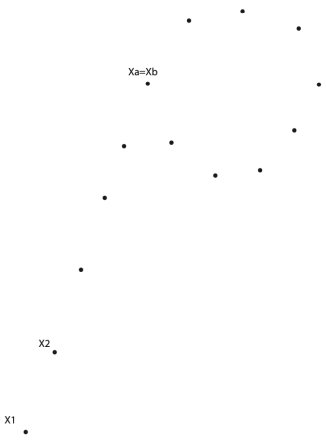
• A crucial idea in the Pollard $\rho$ algorithm is to produce a random sequence in $\mathbf{Z}_n$ by evaluating the iterates of some function $f\colon \mathbf{Z}_n \to \mathbf{Z}_n$ on a random element $x_0 \in \mathbf{Z}_n$

$$x_1 = f(x_0), \qquad x_2 = f^{(2)}(x_0) = f \circ f(x_0), \qquad \ldots, \qquad x_k = f^{(k)}(x_0) = f \circ f \ldots \circ f(x_0).$$

Usually the function is $f(x) = x^2 + 2 \bmod n$. Once a coincidence occurs

$$x_a = x_b, \quad \text{for some } b > a \geq 1, \tag{$*$}$$

then the sequence becomes periodic of period $b - a$, describing the letter $\rho$ which gives the name to the algorithm.




*The letter* $\rho$ *described by the sequence.*

• Another crucial idea of the algorithm is the following observation, which reduces the number of gcd's to be performed from $k^2$ to $k$.

**Lemma.** *(Floyd's trick) If for some* $1 \leq a < b$ *one has*

$$x_a \equiv x_b \mod p,$$

*for some divisor* $p$ *of* $n$*, then there exists* $a \leq m \leq b$ *such that*

$$x_m \equiv x_{2m} \mod p.$$

*Dim.* Suppose that there exist $a$ and $b > a$ such that $x_b \equiv x_a \bmod p$. From then on the sequence becomes periodic of period $b-a$. We claim that there exist $a \leq m \leq b$ and $s \in \mathbf{N}$ for which

$$m + s(b-a) = 2m \quad \Leftrightarrow \quad m = s(b-a).$$

Let's check that $s = [\frac{a}{(b-a)}] + 1$ works. Set $m = ([\frac{a}{(b-a)}] + 1)(b-a)$. One has indeed

$$m = ([\frac{a}{(b-a)}] + 1)(b-a) \geq \frac{a}{(b-a)}(b-a) = a$$

$$m = ([\frac{a}{(b-a)}] + 1)(b-a) \leq (\frac{a}{(b-a)} + 1)(b-a) = b.$$

As a result, one has

$$x_m \equiv x_{2m} \bmod p,$$

as claimed.

• Because of the above Lemma, one may form two parallel sequences

$$x_i = f^{(i)}(x_0), \quad \text{and} \quad y_i = f^{(2i)}(x_0),$$

and take

$$\gcd(x_i - y_i \bmod n, n), \qquad \text{for} \quad i = 1, \ldots k.$$

By the birthday paradox this procedure is likely to detect any non trivial factor $p$ of $n$ of size $\leq k^2$.

• *The Pollard $\rho$ algorithm has probabilistic complexity*

$$\mathcal{O}(\sqrt{p}(\log n)^3) = \mathcal{O}(2^{\frac{1}{2}\log p}(\log n)^3).$$

Indeed, each step of the algorithm consists of three squarings and a subtraction followed by a gcd. Hence it is

$$\mathcal{O}(3(\log n)^2 + \log n) + \mathcal{O}((\log n)^3) = \mathcal{O}((\log n)^3).$$

If $p$ is the smallest factor of $n$, by the birthday paradox, it is *probably* found by $\sqrt{p}$ steps. This is why the complexity is called probabilistic: finding the factor by the above amount of calculations is probable, but not guaranteed.

**Remark**. Without the Floyd's trick, the Pollard $\rho$ algorithm would be $\mathcal{O}(p \log n)$, like trial division, with the disadvantage of being probabilistic, instead of deterministic. The Pollard $\rho$ algorithm is still exponential in $\log p$, however with a better constant.

**Example 6**. If we process the same set of integers as in Example 5, we can see the difference in the performances of the two algorithms.

```
[1, 11000000000001086667778676564564563696917 7]
[2, 10100000000000997758596784845645644848988 07]
[3, 10090000000000996770716986048768767972899963]
[4, 10007000000000988571314656034690689901368 6749]
[5, 10000300000000987904351908407831823943896 4321]
[6, 10000030000000987882762436273267506718735264321]
[7, 10000019000000987881675768494590942154171567 4033]
[8, 10000000700000987879867948462792657469488326 16749]
[9, 10000000070000987879805712035468454226245132616749]
[10, 10000000019000987879800673848494590154173064567 4033]
[11, 10000000000300987879798826513270839994413306282643 21]
[12, 10000000000390987879798835404189029166305198167436173]
[13, 10000000000037987879798800532032131636551444441525 9959]
[14, 10000000000004087879798797183119613715138831831587 31317]
[15, 10000000000013578797987969134284286435914444444152 59959]
[16, 10000000000010488797987968829029428607679489489441374527]
[17, 10000000000000988179798796876906512482070906306306282643 21]
[18, 10000000000000987909798796876879839727503390630630628264321]

1: 11 * 10000000000000987879798796876876876088107. time:0 ms
2: 101 * 10000000000000987879798796876876876088107. time:0 ms
3: 1009 * 10000000000000987879798796876876876088107. time:0 ms
4: 10007 * 10000000000000987879798796876876876088107. time:1 ms
5: 100003 * 10000000000000987879798796876876876088107. time:0 ms
6: 1000003 * 10000000000000987879798796876876876088107. time:5 ms
7: 10000019 * 10000000000000987879798796876876876088107. time:5 ms
8: 100000007 * 10000000000000987879798796876876876088107. time:48 ms
9: 1000000007 * 10000000000000987879798796876876876088107. time:46 ms
10: 10000000019 * 10000000000000987879798796876876876088107. time:232 ms
11: 100000000003 * 10000000000000987879798796876876876088107. time:807 ms
12: 1000000000039 * 10000000000000987879798796876876876088107. time:1,567 ms
13: 10000000000037 * 10000000000000987879798796876876876088107. time:16,409 ms
14: 100000000000031 * 10000000000000987879798796876876876088107. time:10,778 ms
15: 1000000000000037 * 10000000000000987879798796876876876088107. time:40,311 ms
16: 10000000000000061 * 10000000000000987879798796876876876088107. time:4min, 50,415 ms
17: 100000000000000003 * 10000000000000987879798796876876876088107. time:24min, 59,276 ms
18: 100000000000000003 * 10000000000000987879798796876876876088107. time:20min, 28,101 ms
```

```
[1, 11000000000001086667778676564564563696917 7]
[2, 10100000000000997758596784845645644848988 07]
[3, 10090000000000996770716986048768767972899963]
[4, 10007000000000988571314656034690689901368 6749]
[5, 10000300000000987904351908407831823943896 4321]
[6, 10000030000000987882762436273267506718735264321]
[7, 10000019000000987881675768494590942154171567 4033]
[8, 10000000700000987879867948462792657469488326 16749]
[9, 10000000070000987879805712035468454226245132616749]
[10, 10000000019000987879800673848494590154173064567 4033]
[11, 10000000000300987879798826513270839994413306282643 21]
[12, 10000000000390987879798835404189029166305198167436173]
[13, 10000000000037987879798800532032131636551444441525 9959]
[14, 10000000000004087879798797183119613715138831831587 31317]
[15, 10000000000013578797987969134284286435914444444152 59959]
[16, 10000000000010488797987968829029428607679489489441374527]
[17, 10000000000000988179798796876906512482070906306306282643 21]
[18, 10000000000000987909798796876879839727503390630630628264321]

1: 11 * 10000000000000987879798796876876876088107. time:0 ms
2: 101 * 10000000000000987879798796876876876088107. time:0 ms
3: 1009 * 10000000000000987879798796876876876088107. time:0 ms
4: 10007 * 10000000000000987879798796876876876088107. time:1 ms
5: 100003 * 10000000000000987879798796876876876088107. time:0 ms
6: 1000003 * 10000000000000987879798796876876876088107. time:6 ms
7: 10000019 * 10000000000000987879798796876876876088107. time:5 ms
8: 100000007 * 10000000000000987879798796876876876088107. time:49 ms
9: 1000000007 * 10000000000000987879798796876876876088107. time:45 ms
10: 10000000019 * 10000000000000987879798796876876876088107. time:242 ms
11: 100000000003 * 10000000000000987879798796876876876088107. time:845 ms
12: 1000000000039 * 10000000000000987879798796876876876088107. time:1,654 ms
13: 10000000000037 * 10000000000000987879798796876876876088107. time:17,708 ms
14: 100000000000031 * 10000000000000987879798796876876876088107. time:11,767 ms
15: 1000000000000037 * 10000000000000987879798796876876876088107. time:43,567 ms
16: 10000000000000061 * 10000000000000987879798796876876876088107. time:5min, 113 ms
17: 100000000000000003 * 10000000000000987879798796876876876088107. time:25min, 511 ms
18: 100000000000000003 * 10000000000000987879798796876876876088107. time:21min, 44,343 ms
```

Note that the above two tables are not identical: at every attempt different random sequences may be taken in $\mathbf{Z}_n$. This is why the complexity of the algorithm is probabilistic.

**Remark.** The Pollard $\rho$ algorithm can be also be applied to solve the DLP in a cyclic group $G$. In order to apply the birthday paradox one must be able to produce suitable random sequences in $G$. For example, if $G$ is the group of points of an elliptic curve over $\mathbf{Z}_p$, with $p$ prime, then the construction of $f$ can be seen in L. Washington's book on elliptic curves.

**The Pollard $p-1$ factoring algorithm.**

The Pollard $p-1$ algorithm, is a factoring algorithm introduced by John Pollard in 1974. It takes its name from the fact that it detects the prime factors $p$ of an integer $n$, with the property that $p-1$ is a *smooth number*, i.e. it factors into relatively small primes. Note that $p-1$ is the order of the group $\mathbf{Z}_p^*$.

Despite its limited probability of success, the Pollard $p-1$ algorithm is important because in Lenstra's elliptic curve method for factoring (ECM), the same principles are applied to the group of points of an elliptic curve over $\mathbf{Z}_p$.

Integers all of whose prime factors are $\leq B$, are called *$B$-smooth.* They become very sparse, when the ratio $u = \log p / \log B$ grows.

Let $n \in \mathbf{N}$ be the integer to be factored. Fix a smoothness bound $B \in \mathbf{N}$ and define

$$\mathbf{k} := \prod_{\substack{p \leq B \\ p^\alpha \leq B}} p^\alpha.$$

We may think of $\mathbf{k}$ as "the mother of all $B$-smooth numbers", in the sense that heuristically we may assume that "any" $B$-smooth number divides $\mathbf{k}$. The Pollard $p-1$ algorithm consists of taking a random class $a \in \mathbf{Z}_n^*$ and computing

$$\gcd(a^{\mathbf{k}} - 1 \bmod n, n).$$

When is the above *gcd* going to be $> 1$?

*Heuristically, we can say that* $\gcd(a^{\mathbf{k}} - 1 \bmod n,\, n) > 1$ *if and only if $n$ admits some prime factor $p$ with the property that $p - 1 = \#\mathbf{Z}_p^*$ is $B$-smooth.*

*Proof.* If $p$ is a prime factor of $n$ with the property that $p-1$ is $B$-smooth, then we may assume that $p-1$ divides $\mathbf{k}$. From the Fermat Little Theorem applied to $\mathbf{Z}_p^*$, one deduces that $a^{\mathbf{k}} = 1 \bmod p$ and that $p$ divides $\gcd(a^{\mathbf{k}} - 1 \bmod n, n)$.
Conversely, if a prime $p$ divides $\gcd(a^{\mathbf{k}} - 1 \bmod n, n)$, then $p$ is a factor of $n$ and $a^{\mathbf{k}} = 1 \bmod p$. This implies that the order of $a$ in $\mathbf{Z}_p^*$ divides $\mathbf{k}$ and therefore it is $B$-smooth. On the other hand, in the cyclic group $\mathbf{Z}_p^*$, the order of an element is typically given by $\frac{p-1}{r}$, for some small integer $r$. Hence $p-1$ is $B$-smooth.

**Remark.** If a smoothness bound $B$ is fixed, then the $\gcd(a^{\mathbf{k}} - 1 \bmod n, n)$ coincides with *the product of all prime factors $p$* of $n$ with the property that $p-1$ is $B$-smooth.

The complexity of the algorithm is dominated by the complexity of the calculation

$$a^{\mathbf{k}} \bmod n,$$

which is

$$\mathcal{O}(\log \mathbf{k} (\log n)^2) = \mathcal{O}(B (\log n)^2),$$

since the size of $\mathbf{k}$ is roughly $\exp B$. *

Once a smoothness bound $B$ is fixed, the $B$-smooth numbers become more and more sparse as their size grows. Consequently the factors $p$ with the property that $p-1$ is $B$-smooth are rare.

* One has $\prod_{\substack{p \leq B, \\ p \text{ prime}}} p \sim e^B$

If the algorithm fails to find a factor with a given smoothness bound $B$, the only option is to increase $B$. On the other hand, increasing $B$ is very expensive from a computational point of view (the complexity of the algorithm is exponential in $\log B$). This is why this algorithm has limited probability of success.

**Example 7.** Consider $n = 11951438413903$ and take for example $a = 57 \in \mathbf{Z}_n^*$.
For $B = 50$, we get $\gcd(a^{\mathbf{k}} - 1 \bmod n, n) = 1$;
for $B = 100$, we get $\gcd(a^{\mathbf{k}} - 1 \bmod n, n) = 1$;
for $B = 150$, we get $\gcd(a^{\mathbf{k}} - 1 \bmod n, n) = 108769$.
Indeed $n = p * q$, with $p = 108769$ and $q = 109879087$, and

$$p - 1 = 2^5 \cdot 3 \cdot 11 \cdot 103, \qquad q - 1 = 2 \cdot 3 \cdot 29 \cdot 373 \cdot 1693.$$

Since $p - 1$ is 150-smooth, while $q - 1$ is not, the algorithm detected $p$ and not $q$.

**Exercise.** In the above example, repeat the procedure with $a = 2,\ 3,\ 5 \in \mathbf{Z}_n^*$.

**Exercise.** Consider $n = p \cdot q = 687442130387521$, where $p = 686989$ and $q = 1000659589$.
One has $p - 1 = 2^2 \cdot 3^3 \cdot 6361$ and $q - 1 = 2^2 \cdot 3 \cdot 59 \cdot 1413361$.
How big should one choose $B$ so that algorithm detects the non-trivial factor $p$?
What happens if one takes $B > 1413361$?

Now we apply the same principle of Pollard's $p - 1$ algorithm to the group of points of an "elliptic curve" $E$ over $\mathbf{Z}_n$. In this situation the algorithm detects the prime factors $p$ of $n$ for which the elliptic curve $E$ over $\mathbf{Z}_p$ has $B$-smooth order. The next two examples illustrate what can happen.

**Example 9.** Let $n = 129789496891$. The equation

$$E: \qquad Y^2 = X^3 + X + 107926713632$$

has discriminant $\Delta \equiv 28709649864 \in \mathbf{Z}_n^*$. Hence it defines an elliptic curve over $\mathbf{Z}_n$ and over $\mathbf{Z}_p$, for every prime divisor $p$ of $n$. The point $P = (111692269326, 48777146469)$ is on $E(\mathbf{Z}_n)$. Fix the smoothness bound $B = 50$. Then $\mathbf{k} = 3099044504245996706400$.
Computing $\mathbf{k} \cdot P$ on $E(\mathbf{Z}_n)$, we get

$$\mathbf{k} \cdot P = (35973533882, 48777146469) \in E(\mathbf{Z}_n).$$

We can check that

$$n = p \cdot q, \quad p = 1297, \quad q = 100069003,$$
$$\#E(\mathbf{Z}_p) = 100065626 = 2 \cdot 73 \cdot 685381, \qquad \#E(\mathbf{Z}_q) = 1036 = 2 \cdot 653.$$

Neither $E(\mathbf{Z}_p)$ nor $E(\mathbf{Z}_q)$ have $B$-smooth order. When this is the case, the curve $E$ does not produce any factor of $n$.

**Example 10.** Let $n = 129789496891$, as in the previous example. The equation

$$F: \qquad Y^2 = X^3 + X + 76757924901$$

has discriminant $\Delta \equiv 43794887955 \in \mathbf{Z}_n^*$. Hence it defines an elliptic curve over $\mathbf{Z}_n$ and over $\mathbf{Z}_p$, for every prime divisor $p$ of $n$. The point $Q = (98961650999, 56306439329)$ is on $F(\mathbf{Z}_n)$. Fix $B$ and $\mathbf{k}$ as in the previous example. This time, $\mathbf{k} \cdot Q$ is the point at infinity on $F(\mathbf{Z}_p)$ but not on $F(\mathbf{Z}_q)$. Indeed, the order of the curve $F$ over $\mathbf{Z}_p$ is given by $\#F(\mathbf{Z}_p) = 1288 = 2^3 \cdot 7 \cdot 23$, which is $B$-smooth. On the other hand $\#F(\mathbf{Z}_q) = 100085195 = 5 \cdot 7^2 \cdot 607 \cdot 673$ is not $B$-smooth and $\mathbf{k} \cdot Q$ is a point on $F(\mathbf{Z}_q)$, different from the point at infinity.

In practice, this means that when we try to compute $\mathbf{k} \cdot Q$ we encounter an impossible inverse modulo $n$: some *gcd* with $n$ coincides with the factor $p$ of $n$.

**The Miller-Rabin theorem and the Miller-Rabin primality test**.

The Miller-Rabin primality test is a very efficient algorithm to detect whether a positive integer $n \in \mathbf{N}$ is composite or possibly prime. It is based on the Miller-Rabin Theorem (MRT in short), which is a refinement of the Fermat Little Theorem (FLT in short).

**Theorem.** (Fermat Little Theorem). *Let $n \in \mathbf{N}$ be a prime number and let $a$ be an integer satisfying* $\gcd(a, n) = 1$. *Then*

$$a^{n-1} \equiv 1 \bmod n.$$

The FLT provides a necessary condition for an integer $n$ to be prime: if $a^{n-1} \not\equiv 1 \bmod n$, then we are sure that $n$ is composite. On the other hand, if $a^{n-1} \equiv 1 \bmod n$, then we cannot conclude anything about the primality of $n$. Even worse, there infinitely many composite integers for which $a^{n-1} \equiv 1 \bmod n$, for every $a$ with $\gcd(a, n) = 1$. These are the the *Carmichael numbers.** The MRT combines the FLT with the following fact:

*Let $p$ be a prime number. If $p$ divides $x^2 - 1$, then either $x \equiv 1 \bmod p$ or $x \equiv -1 \bmod p$.*

**Theorem.** (Miller-Rabin Theorem). *Let $n \in \mathbf{N}$ be a prime number and let $a$ be an integer satisfying* $\gcd(a, n) = 1$. *Write $n-1 = m2^k$, for some $k$, $m \in \mathbf{N}$ with $m$ odd, and set $b := a^m \bmod n$. Then either $b \equiv 1 \bmod n$ or there is $1 \leq s \leq k$ such that $b^{2^{s-1}} = -1 \bmod n$.*

*Proof.* By FLT one has

$$a^{n-1} = (a^m)^{2^k} \equiv 1 \bmod n.$$

Now we examine how such a power of $a$ can become to 1, modulo $n$.
One way is that already $b := a^m \equiv 1 \bmod n$.
Otherwise, if $b := a^m \not\equiv 1 \bmod n$, then the power $a^{n-1}$ is obtained from $b$ by successive squarings modulo $n$

$$b^2 = b \cdot b, \qquad b^{2^2} = b^2 \cdot b^2, \quad \ldots \quad b^{2^k} = b^{2^{k-1}} \cdot b^{2^{k-1}}.$$

Let $s$ be the smallest positive in integer, $1 \leq s \leq k$, for which

$$b^{2^s} = b^{2^{s-1}} \cdot b^{2^{s-1}} \equiv 1 \bmod n.$$

Then the element $b^{2^{s-1}}$ is a zero of the polynomial $X^2 - 1$. Since $n$ is prime, it follows that either $b^{2^{s-1}} \equiv 1 \bmod n$ or $b^{2^{s-1}} \equiv -1 \bmod n$. Since $s$ is the *smallest* integer for which $b^{2^s} \equiv 1 \bmod n$, then it is necessarily

$$b^{2^{s-1}} \equiv -1 \bmod n,$$

as claimed.

An integer $n$ is called $a$-pseudoprime, provided that it satisfies the Miller-Rabin theorem for some class $a \in \mathbf{Z}_n^*$. Like the FLT, also the MRT only provides a necessary condition for an integer $n$ to be prime: if $a^{n-1} \not\equiv 1 \bmod n$, then $n$ is certainly composite. On the other hand, given an *odd composite integer* $n > 9$, then

$$\#\{a \in \mathbf{Z}_n^* \mid n \text{ is } a\text{-pseudoprime}\} \leq \varphi(n)/4,$$

* Carmichael numbers are composite integers $n$ which satisfy $a^{n-1} \equiv 1 \bmod n$, for all $a \in \mathbf{Z}_n^*$. They are characterized by Korselt's criterion: *A positive composite integer $n$ is a Carmichael number if and only if it is square-free, and for all prime divisors $p$ of $n$, one has $p - 1 | n - 1$.*

where $\varphi$ denotes the Euler $\varphi$ function (for a proof of this fact, see Schoof R., Four primality tests).

This means in particular that if $n$ is $a$-pseudoprime for $k$ different classes $a \in \mathbf{Z}_n^*$, the probability that it is composite is $\leq \frac{1}{4^k}$. This fact combined with the Prime Number Theorem enables one to determine integers $n$, with probability of being primes arbitrarily close to 1, in polynomial time.

**Example 10.** Let $n = 825265$. As $n$ is a Carmichael number, it passes the FLT test for every $a \in \mathbf{Z}_n^*$. Let's check that it does not pass the MR test. Write $n - 1 = 825264 = 51579 \cdot 2^4$, take $13 \in \mathbf{Z}_n^*$ and compute

$$13^{n-1} = 13^{51579 \cdot 2^4} = (13^{51579})^{2^4} \bmod n.$$

We find

$b = 13^{51579} \equiv 341942 \not\equiv 1 \bmod n$

Now we compute $b^{2^4}$ by successive squarings $\bmod n$

$b^2 = 341942^2 \equiv 786164 \not\equiv 1$
$b^{2^2} = b^2 \cdot b^2 \equiv 497421 \not\equiv 1$
$b^{2^3} = b^{2^2} \cdot b^{2^2} \equiv 1.$

Since $b^{2^2} \not\equiv -1 \bmod n$, then $n$ is not prime. Indeed $n = 5 \cdot 7 \cdot 17 \cdot 19 \cdot 73$.

**The Baby-Step-Giant-Step algorithm.**

The Baby-Step-Giant-Step algorithm was introduced by Dan Shanks in 1969. It can be applied to solve the discrete logarithm problem in an arbitrary finite cyclic group.

Let $G$ be a cyclic group with $n$ elements, and let $a \in G$ be a generator of the group. It means that $G = \{a,\ a^2,\ \ldots,\ a^n = e\}$. In particular, every $x \in G$ can be written as $x = a^s$, for some $s \in \mathbf{Z}$. The exponent $s$, which by Lagrange's theorem it is only well defined modulo $n$, is *by definition* the discrete logarithm of $x$ in base $a$

$$s := \log_a(x) \bmod n.$$

The Baby-Step-Giant-Step algorithm exploits the fact that every element $x \in G$ can be written as

$$x = a^{j+mi}, \tag{1}$$

for integers $m,\ i,\ j$ satisfying $m = [\sqrt{n}] + 1$, and $0 \leq i,\ j \leq m$. Equation (1) can be rewritten as $a^i = xa^{-mj}$. Then the logarithm $\log_a(x)$ is obtained by comparing two lists: the baby steps $a^i$ and the giant steps $xa^{-mj}$, for $0 \leq i, j \leq m$. When a coincidence is found between the two lists, namely one has $a^{i_0} = xa^{-mj_0}$ for some $i_0$ and $j_0$, then

$$\log(x)_a = i_0 + mj_0.$$

By the naif method one could possibly have to compute up to $p$ powers modulo $p$, before obtaining the desired logarithm. By BSGS, one obtains the desired logarithm by computing at most $2m \sim 2\sqrt{p}$ powers modulo $p$, one inverse modulo $p$, and by comparing two lists of length at most $m$. In total, the complexity of BSGS in $\mathbf{Z}_p^*$ is

$$\mathcal{O}(\sqrt{p}(\log p)^3).$$

**Example 11.** Fix $p = 433$ and let $a = 7$ be a primitive root in $\mathbf{Z}_p^*$. We want to calculate the discrete logarithm of $x = 166$ in base $a$. In this case, $m = 21 = [\sqrt{433}] + 1$.
We first produce the list of the **Baby-Steps**:

$$a^i \bmod p, \qquad \text{for } 0 \leq i \leq m$$

$a^0 \equiv 1$
$a^1 \equiv 7$
$a^2 \equiv 49$
$a^3 \equiv 343$
$a^4 \equiv 236$
$a^5 \equiv 353$
$a^6 \equiv 306$
$a^7 \equiv 410$
$a^8 \equiv 272$
$a^9 \equiv 172$
$a^{10} \equiv 338$
$a^{11} \equiv 201$
$a^{12} \equiv 108$
$a^{13} \equiv 323$
$a^{14} \equiv 96$

$a^{15} \equiv 239$
$a^{16} \equiv 374$
$a^{17} \equiv 20$
$a^{18} \equiv 140$
$a^{19} \equiv 114$
$a^{20} \equiv 365$
$a^{21} \equiv 390$

$a^{-m} \equiv a^{-21} \equiv 292$

Next we produce the list of the **Giant-Steps**:

$$xa^{-mj} \bmod p, \quad \text{for } 0 \leq j \leq m$$

and each time we check whether the value the new Giant-Step already appears in the list of the Baby-Steps. When that is the case, we are done.

$x \cdot a^0 \equiv 166$
$x \cdot a^{-21} \equiv 409$
$x \cdot a^{-42} \equiv 353$ !!!

We found a coincidence: $a^5 \equiv x \cdot a^{-42}$. This means that

$$x \equiv a^{5+42} \equiv a^{47} \quad \text{and} \quad \log_7(166) \equiv 47 \bmod 432.$$

Indeed one can check that $7^{47} \equiv 166 \bmod 433$.

**Example 12.** Baby Step Giant Step on an elliptic curve. Let $p = 101$. The equation

$$E: \qquad Y^2 = X^3 + X + 55$$

defines an elliptic curve over $\mathbf{Z}_p$, of prime order $\#E(\mathbf{Z}_p) = 109$. Consequently the group $(E(\mathbf{Z}_p), +)$ is cyclic and every point different from $\infty$ is a generator. For example, we fix $P = (60, 51) \in E(\mathbf{Z}_p)$.

Now let $Q = (62, 40) \in E(\mathbf{Z}_p)$. Then the logarithm $\log_P(Q)$ of $Q$ in base $P$ is by definition the integer $\mathbf{k}$ mod 109 such that

$$Q = \mathbf{k} \cdot P.$$

In this case, $m = [\sqrt{109}] + 1 = 11$
We first produce the list of the **Baby-Steps**:

$$iP, \quad \text{for } 0 \leq i \leq m.$$

$\infty$
$P = (60, 51)$
$2P = (17, 95)$
$3P = (5, 36)$
$4P = (92, 78)$
$5P = (66, 26)$
$6P = (12, 52)$
$7P = (13, 89)$

$8P = (49, 54)$
$9P = (48, 10)$
$10P = (18, 42)$
$11P = (72, 33)$
$-11P = (72, 68)$

Next we produce the list of the **Giant-Steps**:

$$Q - 11jP, \quad \text{for } 0 \leq j \leq m-1$$

and each time we check whether the value the new Giant-Step already appears in the list of the Baby-Steps. When that is the case, we are done.
$Q = (62, 40)$
$Q - 11P = (92, 78)$

Since $4P = Q - 11P$, we obtain the desired logarithm

$$\log_P(Q) = 15 \bmod 109.$$

One can check that $15P = Q$ in $E(\mathbf{Z}_p)$.

**The Pohlig-Hellman algorithm.**
Let $G$ be a cyclic group of order $N$ and suppose that $N = \prod_i q_i^{e_i}$ is the product of small distinct primes $q_i$, for $i = 1, \ldots, s$. Then $G \cong G_1 \times \ldots \times G_s$, with

$$\#G_i = q_i^{e_i} \quad \text{and} \quad G_i \cong \mathbf{Z}_{q_i^{e_i}}.$$

By the Chinese Remainder Theorem the discrete logarithm problem in $G$ can be reduced to the discrete problem in the smaller groups $G_i$. Hence *the essential case is* $G = \mathbf{Z}_{q^e}$*, for* $q$ *odd prime and* $e \geq 1$.

Let $P$ be a generator of $G$ and let $Q$ be a given element. Then $Q = kP$, for some integer $k \bmod q^e$. We want to determine $k$, which by definition is the discrete logarithm of $Q$ in base $P$. Recall that the subgroups of $G$ are linearly ordered

$$0 = q^e G \subset q^{e-1} G \subset \ldots \subset qG \subset G,$$

where $q^m G$ is the $q^{e-m}$-torsion subgroup, for $m = 0, 1, \ldots, e$.

The Pohlig-Hellman algorithm provides a method to solve the DLP in $G$. Write $k$ in base $q$, as $k = k_0 + k_1 q + \ldots + k_s q^s$, for $k_j \in \{0, \ldots, q^e - 1\}$. Then

$$Q = kP = k_0 P + k_1 qP + \ldots + k_s q^s P, \qquad (*)$$

where the summand $k_m q^m P$ is an element in the $q^{e-m}$-torsion subgroup of $G$, for $m = 0, 1, \ldots, e$.

In order to determine the coefficients $k_m$, we precompute the elements of the $q$-torsion

$$T = \{0, q^{e-1}P, \ldots, (q-1)q^{e-1}P\}.$$

By multiplying both terms of the equation $(*)$ by $q^{e-1}$ we get

$$q^{e-1}Q = k_0 q^{e-1} P,$$

which is an element in the $q$-torsion $T$. By comparing it with the elements of $T$, we determine $k_0$. In general, once we have determined $k_0, \ldots, k_{j-1}$, we obtain $k_j$ as follows: we multiply both terms of the equation

$$Q - k_0 P - \ldots - k_{j-1} q^{j-1} P = k_j q^j P + \ldots + k_s q^s P$$

by $q^{e-j-1}$. The only surviving element on the right hand side is $k_j q^{e-1} P$. By comparing it with the elements of $T$, we determine $k_j$.

**Index calculus**.

Index calculus is an algorithm for solving the discrete logarithm problem in $\mathbf{Z}_p^*$, for $p$ prime. Before explaining it, we start with an example.

**Example 13.** Let $p = 13$. Then $\mathbf{Z}_{13}^*$ is a cyclic group of order 12 and $a = 2$ is a generator. We want to compute $\log_2 5$, which is an integer modulo 12. We already know

$$\log_2 2 \equiv 1, \qquad \log_2 1 \equiv 0, \qquad \log(-1) = \frac{p-1}{2} \equiv 6 \qquad \bmod 12.$$

From the relations mod 13

$$-1 \equiv 12 \equiv 2^2 \cdot 3, \quad 1 \equiv 14 \equiv 2 \cdot 7, \quad 2 \equiv 15 \equiv 3 \cdot 5,$$

we obtain the following linear relations among the respective logarithms modulo 12 (in any base $a$)

$$\log(-1) \equiv 2\log 2 + \log 3, \quad \log 1 \equiv \log 2 + \log 7, \quad \log 2 \equiv \log 3 + \log 5.$$

For $a = 2$, we obtain the following linear system with coefficients in $\mathbf{Z}_{12}$

$$\begin{cases} \log 3 \equiv 4 \\ \log 7 \equiv -1 \\ \log 3 + \log 5 \equiv 1 \end{cases}$$

and

$$\log_2 5 = 1 - 4 \equiv -3 \equiv 9 \bmod 12.$$

One can check that $2^9 \equiv 5 \bmod 13$.

The *Index calculus* is a method which follows the same steps of the above example, in a systematic way. Let $p$ be a prime and let $a$ be a generator of $\mathbf{Z}_p^*$. Fix a positive integer $B$ and set $F := \{s \leq B, s \text{ prime}\}$, the *factor base*. Let $\alpha := \#F$. Suppose we want to compute

$$\log_a x, \qquad \text{for some } x \in \mathbf{Z}_p^*.$$

• The first step of the algorithm consists of determining the logarithms of all the primes in $F$. We produce random powers of the primes in $F$

$$s_1^{e_1} \cdot \ldots \cdot s_\alpha^{e_\alpha}.$$

Every time that the residue class of one such power modulo $p$ factors in the primes of the factor base, i.e.

$$s_1^{e_1} \cdot \ldots \cdot s_\alpha^{e_\alpha} \equiv s_1^{f_1} \cdot \ldots \cdot s_\alpha^{f_\alpha} \bmod p,$$

we obtain a linear relation among the respective logarithms modulo $p-1$, with respect to any base $a$. With enough linear relations, we will be able to determine the logarithms

$$\log_a s_1, \quad \ldots \quad \log_a s_\alpha.$$

• The second step consists of using the relations obtained in the first step to determine the $\log_a x$. We multiply $x$ by random powers of the primes in $F$

$$x s_1^{e_1} \cdot \ldots \cdot s_\alpha^{e_\alpha}.$$

If the residue class of one such product modulo $p$ factors in the primes of the factor base, i.e.

$$xs_1^{e_1} \cdot \ldots \cdot s_\alpha^{e_\alpha} \equiv s_1^{f_1} \cdot \ldots \cdot s_\alpha^{f_\alpha} \mod p,$$

then

$$\log_a x \equiv (f_1 - e_1) \log_a s_1 + \ldots + (f_\alpha - e_\alpha) \log_a s_\alpha \mod p - 1.$$

The complexity of this algorithm mainly depends on the distribution of the $B$-smooth numbers as a function of the ratio $\log p / \log B$. Here we just say that it is subexponential in $\log p$.

**Example 14.** Let $p = 179$ prime. Then $\mathbf{Z}_p^*$ is a cyclic group of order 178. Generators of $\mathbf{Z}_p^*$ are for example $a = 2$ and $a = 11$. Fix $B = 7$. Then $F = \{2, 3, 5, 7\}$. We want to calculate

$$\log_{11} 13.$$

In the first step we determine

$$\log_a 2, \quad \log_a 3, \quad \log_a 5, \quad \log_a 7.$$

Suppose we have determined the following relations $\mod p$

$$2^2 \cdot 3^2 \cdot 5 \equiv 1, \quad 2^3 \cdot 5^2 \equiv 3 \cdot 7 \quad 2^5 \cdot 7 \equiv 3^2 \cdot 5$$

$$3^4 \equiv 5^7, \quad 3 \cdot 7^2 \equiv 7^{12}.$$

Then by taking logarithms, we obtain the linear relations modulo $p - 1 = 178$ (indepent of the choice of a generator of $\mathbf{Z}_p^*$)

$$\begin{cases} 2\log 2 + 2\log 3 + \log 5 \equiv 0 \\ 3\log 2 - \log 3 + 2\log 5 - \log 7 \equiv 0 \\ 5\log 2 - 2\log 3 - \log 5 + \log 7 \equiv 0 \\ 4\log 3 - 7\log 5 \equiv 0 \\ \log 3 - 10\log 7 \equiv 0. \end{cases}$$

If we take $a = 2$, then $\log_2 2 = 1$ and the subset of equations

$$\begin{cases} 2\log_2 3 + \log_2 5 \equiv -2 \\ -\log_2 3 + 2\log_2 5 - \log 7 \equiv -3 \\ -2\log_2 3 - \log_2 5 + \log 7 \equiv -5 \end{cases}$$

is already sufficient to obtain

$$\log_2 2 \equiv 1, \quad \log_2 3 \equiv 108, \quad \log_2 5 \equiv 138, \quad \log_2 7 = 171 \mod 178.$$

In solving the equations, one should remember that $\mathbf{Z}_{178}$ is not a field, and some coefficients may not be invertible modulo 178. Next we compute $\log_2 13$. From the relation

$$13 \cdot 2 \cdot 7 \equiv 3 \mod 179,$$

we have

$$\log_2 13 = -\log_2 2 - \log_2 7 + \log_2 3 \equiv 114 \mod 178.$$

Similarly, form

$$11 \cdot 2^4 \equiv -3 \mod 179,$$

we get

$$\log_2 11 = -4\log_2 2 + \log_2(-1) + \log_2 3 \equiv 15 \mod 178.$$

Finally

$$\log_{11} 13 = 114 \cdot 15^{-1} \equiv 114 \cdot 95 \equiv 150 \mod 178.$$