• 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 its number of bits, 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

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

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_j; it is *exponential time* if the complexity is $\mathcal{O}(n_1^{a_1} \cdot \ldots \cdot n_s^{a_s})$, for some fixed $a_j > 0$, independent of n_j; it is *subexponential time* if the complexity 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 (or subexponential) 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 required to produce the output is multiplied by a constant factor, *independent of the size of the input*. For example, if the complexity is $\mathcal{O}(\log^k n)$, then

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

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. The complexity of the arithmetic operations in $\mathbf{Z}$ is polynomial in the size of the input. Let $n, m \in \mathbf{N}$, with $n \geq m > 0$. 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 $n + m$

$$\begin{array}{r} a_k a_{k-1} \ldots\ldots a_1 a_0 + \\ b_h b_{h-1} \ldots b_1 b_0 \end{array}$$

is obtained by at most h sums and k carryings (we sum the bits in base 2, by the rules $0+0=0$, $1+0=0+1=1$ and $1+1=(10)$). Hence it has complexity $\mathcal{O}(\log n + \log m) = \mathcal{O}(\log n)$.

(b)

(c)

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

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

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

The remainders form a decreasing sequence of non negative integers

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

the smallest positive element of which 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 $\gcd(n, m)$.

Fact. *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 it follows that $r_{k+2} \leq \frac{1}{2} r_k$, as claimed.

The above fact 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 would decrease of 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. 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)

(b)

(c)

Example. (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 output x^a has size

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

Writing the output is already exponential in $\log a$. This gives a lower bound for the complexity of the whole calculation, which then 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 has complexity $\mathcal{O}((\log n)^2)$, the complexity of the above computation is $\mathcal{O}(\log a (\log n)^2)$. The complexity of multiplying all of them together has also complexity $\mathcal{O}(\log a (\log n)^2)$. 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 $\log a$.