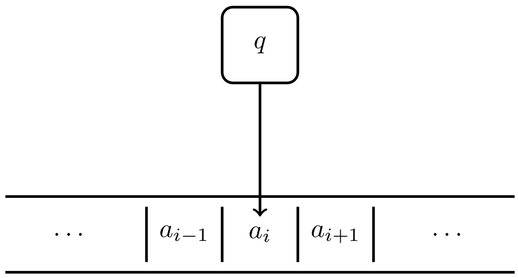
P and NP

1. *Turing Machines*

A *Turing machine* $\mathfrak{M}$ consists of an *external alphabet* Σ (in this alphabet the input and output is formulated), an *internal alphabet* $\Gamma \supseteq \Sigma$ (during the computation process, the machine is allowed to use some extra letters), a finite set of *states* Q with designated initial state q_0 and final state q_F and, most importantly, a finite set of *rules* Δ.

Each rule in Δ is of the form $\langle p, a \rangle \to \langle q, b, d \rangle$, where $p, q \in Q$, $a, b \in \Gamma$, and $d \in \{L, R, N\}$.

The Turing machine operates as follows. At each step of its run, the machine keeps one of the states (from Q) in its internal memory, and observes one of the cells of an infinite *tape.* The tape is considered infinite; however, at each moment of time only a finite part of it is filled with data. For convenience, we suppose that the rest of the tape is padded by a "blank" symbol $B \in \Gamma - \Sigma$.



Rules of $\mathfrak{M}$ are interpreted as follows. If there is a rule $\langle p, a \rangle \to \langle q, b, d \rangle$ in Δ, the machine keeps state p in its internal memory and observes a cell with a written there, then $\mathfrak{M}$ is allowed to perform the following move:

1. replace a with b in the cell;
2. replace state p with state q in the internal memory;
3. perform the movement according to d: if $d = L$, move one cell left, if $d = R$, move one cell right; if $d = N$, stay on the same cell.

A Turing machine is *deterministic,* if for any $p \in Q$ and $a \in \Gamma$ there exists at most one rule of the form $\langle p, a \rangle \to \ldots$ A deterministic machine always "knows what to do."

Once a machine runs into state q_F, it *successfully stops* (computation finished). We suppose that the resulting word on the tape is in the Σ alphabet; this is the result of computation. If there is no rule to apply, the machine halts unsuccessfully. There is also a possibility for infinite execution.

For non-deterministic Turing machines, more than one execution trajectory is possible, and some of them could be successful (but possibly with different results), while others are not.

2. P *and* NP

Here we consider only a specific class of algorithmic questions, namely *decision problems.* In a decision problem, the answer is either "yes" or "no." Equivalently, a decision problem can be represented as a set A of possible inputs (i.e., $A \in \Sigma^*$) on which the answer is "yes."

Definition. A given decision problem A is *polynomial-time decidable* (notation: $A \in \mathsf{P}$), if there exists a deterministic Turing machine $\mathfrak{M}$ which solves the decision problem, and there exists a polynomial p such that for any input x the running time of $\mathfrak{M}$ on x does not exceed $p(|x|)$.

(Here and further $|x|$ means the length of x, in bits.)

Definition. A given decision problem A belongs to NP if it is solvable in polynomial time by a non-deterministic Turing machine $\mathfrak{M}$, in the following sense: $x \in A$ if and only if there *exists* a successful execution of $\mathfrak{M}$ on x, which yields "yes," with no more than $p(|x|)$ steps.

One can easily see that $\mathsf{P} \subseteq \mathsf{NP}$: any deterministic Turing machine can be also considered as a non-deterministic one.

The NP class can be also equivalently defined in terms of *hints:*

Definition. Decision problem A belongs to NP, if there is a (deterministically) polynomially decidable binary relation R, such that $x \in A$ iff there *exists* a polynomial-size hint y such that $R(x, y)$ is true (the algorithm for R yields "yes").

The equivalence is established as follows: if we have a non-deterministic machine, we can just guess the correct value of y and then run the algorithm for R. For the other direction, suppose that all non-deterministic branching points are binary (choice of two possible rules). The total number of such branching points on any computation path is bounded by $p(|x|)$. Then let our hint y include $p(|x|)$ bits, and each time we need to do non-deterministic choice, we take the next bit of y for choosing.

3. NP*-hardness*

The question whether the classes P and NP coincide ($\mathsf{P} =? \mathsf{NP}$) is one of the most challenging questions in computer science. If the answer happens to be positive, then any NP problem would be deterministically solvable in polynomial time. There is a consensus in the computer science community that probably $\mathsf{P} \neq \mathsf{NP}$.

Since, however, there is no proof of $\mathsf{P} \neq \mathsf{NP}$, one cannot *prove,* for a particular NP problem, that it cannot be solved polynomially. However, the theory of NP-hardness provides a way to obtain *conditional results.* Namely, one can prove for some particular problems that they cannot be solve polynomially, *provided that* $\mathsf{P} \neq \mathsf{NP}$.

Definition. A decision problem A is *polynomially* m*-reducible* to another decision problem B, if there exists a polynomially computable function f, defined on possible inputs of A, such that $x \in A \iff f(x) \in B$. Notation: $A \leq_m^P B$.

(Notice that m here is not a natural parameter, it is just a name for this particular type of reduction.)

It is easy to see the following: if $B \in \mathsf{P}$ and $A \leq_m^P B$, then $A \in \mathsf{P}$. By contraposition we deduce that if $A \notin \mathsf{P}$ and $A \leq_m^P B$, then B is also not in P.

This shows the usage of *forward and backwards reductions.* If we wish to show that a problem A is *easy,* we can do this by reducing it to an easy problem B (forward reduction). Conversely, if we wish to show that B is *hard,* we do this by reducing a hard problem A to B (backwards reduction).

Definition. A decision problem B is NP*-hard,* if $A \leq_m^P B$ for any problem A in NP.

If it happens that B is NP-hard and at the same time belongs to P, then any $A \in \mathsf{NP}$ would also belong to P, which means $\mathsf{P} = \mathsf{NP}$. Thus, if $\mathsf{P} \neq \mathsf{NP}$, then an NP-hard problem has no polynomial time solution.

Definition. A problem is NP*-complete,* if it belongs to NP and is NP-hard.

(There could also be NP-hard problems beyond NP.)

By transitivity of $\leq_m^P$, if A is NP-hard and $A \leq_m^P B$, then B is also NP-hard (backwards reduction!).

4. NP*-completeness of* SAT

The *satisfiability problem,* denoted by SAT, is formulated as follows: given a Boolean formula φ, determine whether it is satisfiable. Clearly, SAT $\in \mathsf{NP}$: the satisfying assignment is the hint y, and the fact that y is indeed a satisfying assignment for φ can be checked in polynomial time.

Theorem 1 (S. Cook, L. Levin)**.** SAT *is* NP*-complete.*

We give only a sketch of proof. In order to prove NP-hardness of SAT, we have to provide a reduction of an arbitrary problem $A \in \mathsf{NP}$ to SAT. Suppose A is solvable by a non-deterministic Turing machine $\mathfrak{M}$. For convenience, we suppose that the tape of $\mathfrak{M}$ can grow infinitely only to the right. Moreover, since on a successful run $\mathfrak{M}$ could perform $\leq p(|x|)$ steps, we can suppose that the length of the tape, when running on input x, is bounded by $p(|x|)$.

Let us encode states and letters of the internal alphabet of $\mathfrak{M}$ by words ("bytes") of 0's and 1's, of sufficient constant length m. Moreover, suppose that the code of each state starts with 1. Then the current configuration of $\mathfrak{M}$ can be encoded as follows: we write down codes of all letters on the tape, from the first to the $p(|x|)$-th, and prepend each letter with the code of the state, if it is the letter in the cell which is now observed, or by $0^m = 00\ldots 0$ (m times) otherwise:

$$0^m \quad a_1 \quad \ldots \quad 0^m \quad a_{i-1} \quad q \quad a_i \quad 0^m \quad a_{i+1} \quad \ldots$$

Next, the sequence of configurations can be represented as a *matrix* of size $(m \cdot p(|x|)) \times p(|x|)$. Let this matrix be $\left(b_{ij}\right)_{i \leq m \cdot p(|x|), j \leq p(|x|)}$.

Given the input x, one can efficiently (i.e., by a polynomial time algorithm) construct a Boolean formula φ_x, with variables b_{ij}, which expresses the fact that the sequence of configurations encoded by the $\left(b_{ij}\right)$ matrix is a protocol of successful execution of $\mathfrak{M}$ on x. Namely, φ_x should include the following claims (as a conjunction):

1. the first line represents the configuration with x on the tape, $\mathfrak{M}$ observing its first letter;
2. each next line is obtained from the previous one by one of the rules of $\mathfrak{M}$;
3. the last line includes state q_F.

Now we see that $x \in A$ iff there is a succesful run of $\mathfrak{M}$ on x, which is if and only if φ_x is satisfiable. This establishes the necessary reduction $A \leq_m^P$ SAT.

5. NP*-completeness of* 3-SAT

It happens that a particular case of SAT is as hard as SAT itself. We are talking about 3-SAT, the satisfiability problem for formulae in 3-CNF. The reduction is based on the following theorem.

Theorem 2 (G. Tseitin)**.** *For any Boolean formula φ there exists, and can be computed in polynomial time, an equisatisfiable formula ψ in 3-CNF.*

Note that it is not always possible to construct an *equivalent* 3-CNF for a given Boolean formula, and even constructing and equivalent CNF could lead to exponential blowup. However, equsatisfiability is a weaker notion. The formula φ is translated to ψ by *Tseitin transformations.* For each subformula ξ_i of φ, we introduce a new variable t_i and replace each connective application with the corresponding equivalence: for example, instead of $(\xi_i \wedge \xi_j)$, which is subformula ξ_k of φ, we write down an equivalence $t_k \leftrightarrow (t_i \wedge t_j)$. (If a subformula is just a variable, then we put it instead of the corresponding t_i.)

For example, formula $(p \to q) \vee (q \to (p \to r))$ gets translated as follows:

$$\begin{aligned} t_1 &\leftrightarrow (p \to q) \\ t_2 &\leftrightarrow (p \to r) \\ t_3 &\leftrightarrow (q \to t_2) \\ t_4 &\leftrightarrow (t_1 \vee t_3) \end{aligned}$$

Let t_n (in this example $n = 4$) correspond to the whole formula φ.

It is easy to see that φ is equisatisfiable with the conjunction of all these equivalences plus formula t_n.

Finally, each of these equivalences can be equivalently rewritten as several 3-CNF clauses:

$t_k \leftrightarrow (t_i \wedge t_j)$	$(\neg t_i \vee \neg t_j \vee t_k) \wedge (t_i \vee \neg t_k) \wedge (t_j \vee \neg t_k)$
$t_k \leftrightarrow (t_i \vee t_j)$	$(t_i \vee t_j \vee \neg t_k) \wedge (\neg t_i \vee t_k) \wedge (\neg t_j \vee t_k)$
$t_k \leftrightarrow (t_i \rightarrow t_j)$	$(\neg t_i \vee t_j \vee \neg t_k) \wedge (t_i \vee t_k) \wedge (\neg t_j \vee t_k)$
$t_k \leftrightarrow \neg t_i$	$(t_i \vee t_k) \wedge (\neg t_i \vee \neg t_k)$

This gives a polynomial size 3-CNF $\psi = f(\varphi)$ which is equisatisfiable with φ. Thus, SAT $\leq_m^P$ 3-SAT, whence 3-SAT is NP-hard.