Complexity Classes

Complexity of Problems

- It seems that different (algorithmic) problems are of different difficulty.
- More difficult are those problems that require more time and space to be solved.
- We would like to analyze somehow the difficultness of problems
  - absolutely — how much time and space do we need for their solution,
  - relatively — by how much is their solution harder or simpler with respect to other problems.
- Why do we not succeed in finding efficient algorithms for some problems?
  Can there exist an efficient algorithm for a given problem?
- What are practical boundaries of what can be achieved?

Complexity of Problems

It is necessary to distinguish between a **complexity of an algorithm** and a **complexity of a problem**.

If we for exaple study the time complexity in the worst case, informally we could say:

- **complexity of an algorithm** — a function expressing maximal running time of the given algorithm on inputs of size n
- **complexity of a problem** — what is the time complexity of the "most efficient" algorithm for the given problem

A formal definition of a notion "complexity of a problem" in the above sense leads to some technical difficulties. So the notion "complexity of a problem" is not defined as such but it is bypassed by a definition of **complexity classes**.

Complexity Classes

Complexity classes are subsets of the set of all (algorithmic) **problems**.

A certain particular complexity class is always characterized by a property that is shared by all the problems belonging to the class.

A typical example of such a property is a property that for the given problem there exists some algorithm with some restrictions (e.g., on its time or space complexity):

- Only a problem for which such algorithm exists belongs to the given class.
- A problem for which such algorithm does not exist does not belong to the class.

Remark: In the following discussion, we will concentrate almost exclusively on classes of **decision** problems.

Complexity Classes

Definition

For every function $f : \mathbb{N} \to \mathbb{N}$ we define $\mathrm{DTIME}(f(n))$ as the class containing exactly those decision problems for which there exists an algorithm with time complexity $\mathcal{O}(f(n))$.

Example:

- $\mathrm{DTIME}(n)$ – the class of all decision problems for which there exists an algorithm with time complexity $\mathcal{O}(n)$
- $\mathrm{DTIME}(n^2)$ – the class of all decision problems for which there exists an algorithm with time complexity $\mathcal{O}(n^2)$
- $\mathrm{DTIME}(n \log n)$ – the class of all decision problems for which there exists an algorithm with time complexity $\mathcal{O}(n \log n)$

Complexity Classes

Definition

For every function $f : \mathbb{N} \to \mathbb{N}$ we define $\mathrm{DSPACE}(f(n))$ as the class containing exactly those decision problems for which there exists an algorithm with space complexity $\mathcal{O}(f(n))$.

Example:

- $\mathrm{DSPACE}(n)$ – the class of all decision problems for which there exists an algorithm with space complexity $\mathcal{O}(n)$
- $\mathrm{DSPACE}(n^2)$ – the class of all decision problems for which there exists an algorithm with space complexity $\mathcal{O}(n^2)$
- $\mathrm{DSPACE}(n \log n)$ – the class of all decision problems for which there exists an algorithm with space complexity $\mathcal{O}(n \log n)$

Complexity Classes

Remark:

Note that for classed DTIME(f) and DSPACE(f) it depends which problems belong to the class on the used computational model (if it is a RAM, a one-tape Turing machine, a multitape Turing machine, ...).

Complexity Classes

Using classes DTIME($f(n)$) and DSPACE($f(n)$) we can define classes PTIME and PSPACE as

$$\text{PTIME} = \bigcup_{k \geq 0} \text{DTIME}(n^k) \qquad \text{PSPACE} = \bigcup_{k \geq 0} \text{DSPACE}(n^k)$$

- PTIME is the class of all decision problems for which there exists an algorithm with polynomial time complexity, i.e., with time complexity $\mathcal{O}(n^k)$ where k is a constant.
- PSPACE is the class of all decision problems for which there exists an algorithm with polynomial space complexity, i.e., with space complexity $\mathcal{O}(n^k)$ where k is a constant.

Complexity Classes

Since all (reasonable) computational models are able to simulate each other in such a way that in this simulation the number of steps does not increase more than polynomially, the definitions of classes PTIME and PSPACE are not dependent on the used computational model.
For their definition we can use any computational model.

We say that these classes are **robust** – their definitions do not depend on the used computational model, For all "reasonable" sequential models of computation, this class contains the same problems.

Remark: As "resonable" sequential models of computation are considered those that can be simulated by Turing machines in such a way that the running time increases only polynomially in such simulation.

"Reasonable" Sequential Models of Computation

Examples of models of computation considered to be "reasonable" from this point of view:

- variants of Turing machines (one-tape, multi-tape, . . .)
- RAMs with the use of logarithmic measure
- RAMs without operations for multiplication and division with the use of unit measure
- RAMs that have operations for multiplication and division with the use unit measure if it is ensured for the given RAM that during a computation each memory cell contains a number whose size is bounded by some polynomial

Models of computation that are not "reasonable"

Examples of models of computation that are not "reasonable" from this point of view:

- RAMs with operations form multiplication and division using the unit measure (without restrictions on the size of numbers, on which arithmetic operations can be performed in one step) — they can perform in one step perform arithmetic operations on numbers that have exponential number of bits
- Minsky machines — they are too slow, execution of simple operations takes too much time; in a simulation of a computation of a Turing machine, the time grows exponentially with respect to the original Turing machine

Complexity Classes

Other classes are introduced analogously:

EXPTIME – the set of all decision problems for which there exists an algorithm with time complexity $2^{\mathcal{O}(n^k)}$ where k is a constant

EXPSPACE – the set of all decision problems for which there exists an algorithm with space complexity $2^{\mathcal{O}(n^k)}$ where k is a constant

LOGSPACE – the set of all decision problems for which there exists an algorithm with space complexity $\mathcal{O}(\log n)$

Remark: Instead of $2^{\mathcal{O}(n^k)}$ we can also write $\mathcal{O}(c^{n^k})$ where c and k are constants.

Complexity Classes

For definition of LOGSPACE class we specify more exacly what we consider as a space complexity of an algorithm.

For example, let us consider a Turing machine with three tapes:

- An **input tape** on which the input is written at the beginning.
- A **working tape** which is empty at the start of the computation. It is possible to read from this tape and to write on it.
- An **output tape** which is also empty at the start of the computation. It is only possible to write on it.

The amount of used space is then defined as the number of cells used on the working tape.

Complexity Classes

Other examples of complexity classes:

2-EXPTIME – the set of all problems for which there exists an algorithm with time complexity $2^{2^{\mathcal{O}(n^k)}}$ where k is a constant

2-EXPSPACE – the set of all problems for which there exists an algorithm with space complexity $2^{2^{\mathcal{O}(n^k)}}$ where k is a constant

ELEMENTARY – the set of all problems for which there exists an algorithm with time (or space) complexity

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{\mathcal{O}(n^k)}}}}}}}}}$$

where k is a constant and the number of exponents is bounded by a constant.

Relationships between Complexity Classes

If a Turing machine performs m steps then it visits at most m cells on the tape.

This means that if there exists an algorithm for some problem with time complexity $\mathcal{O}(f(n))$, the space complexity of this algorithm is (at most) $\mathcal{O}(f(n))$.

So it is obvious that the following relationship holds.

Observation

For every function $f : \mathbb{N} \to \mathbb{N}$ is $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$.

Remark: We can analogously reason in the case of a RAM.

Relationships between Complexity Classes

Based on the previous, we see that:

$$\begin{aligned} \text{PTIME} &\subseteq \text{PSPACE} \\ \text{EXPTIME} &\subseteq \text{EXPSPACE} \\ \text{2-EXPTIME} &\subseteq \text{2-EXPSPACE} \\ &\vdots \end{aligned}$$

Since polynomial functions grow more slowly than exponential and logarithmic more slowly than polynomial, we obviously have:

$$\text{PTIME} \subseteq \text{EXPTIME} \subseteq \text{2-EXPTIME} \subseteq \cdots$$

$$\text{LOGSPACE} \subseteq \text{PSPACE} \subseteq \text{EXPSPACE} \subseteq \text{2-EXPSPACE} \subseteq \cdots$$

Relationships between Complexity Classes

For analyzing relationships between complexity classes it is useful to consider **configurations**.

A configuration is a global state of a machine during one step of a computation.

- For a Turing machine, a configuration is given by the state of its control unit, the content of the tape (resp. tapes), and the position of the head (resp. heads).
- For a RAM, a configuration is given by the content of the memory, by the content of all registers (including IP), by the content of the input and output tapes, and by positions of their heads.

Relationships between Complexity Classes

It should be clear that configurations (or rather their descriptions) can be written as words over some alphabet.

Moreover, we can write configurations in such a way that the length of the corresponding words will be approximately the same as the amount of memory used by the algorithm (i.e., the number of cells on the tape used by a Turing machine, the number of number of bits of memory used by a RAM, etc.).

Remark: If we have an alphabet Σ where $|\Sigma| = c$ then:

- The number of words of length n is c^n, i.e., $2^{\Theta(n)}$.
- The number of words of length at most n is

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}$$

  i.e., also $2^{\Theta(n)}$.

Relationships between Complexity Classes

It is clear that during a computation of an algorithm there is no configuration repeated, since otherwise the computation would loop.

Therefore, if we know that the space complexity of an algorithm is $\mathcal{O}(f(n))$, it means that the number of different configurations that are reachable during a computation is $2^{\mathcal{O}(f(n))}$.

Since configurations do not repeat during a computation, also the time complexity of the algorithm is at most $2^{\mathcal{O}(f(n))}$.

Observation

For every function $f : \mathbb{N} \to \mathbb{N}$ it holds that if a problem P is solved by an algorithm with space complexity in $\mathcal{O}(f(n))$, then the time complexity of this algorithm is in $2^{\mathcal{O}(f(n))}$.

So if a problem P is in class $\mathrm{DSPACE}(f(n))$, then it is also in class $\mathrm{DTIME}(2^{c \cdot f(n)})$ for some $c > 0$.

Relationships between Complexity Classes

The following results can be drawn from the previous discussion:

$$\begin{aligned} \text{LOGSPACE} &\subseteq \text{PTIME} \\ \text{PSPACE} &\subseteq \text{EXPTIME} \\ \text{EXPSPACE} &\subseteq \text{2-EXPTIME} \\ &\vdots \end{aligned}$$

Summary:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq \\ \subseteq \text{2-EXPTIME} \subseteq \text{2-EXPSPACE} \subseteq \cdots \subseteq \text{ELEMENTARY}$$

Upper and Lower Bounds on Complexity of Problems

An **upper bound** on a complexity of a problem means that the complexity of the problem is not greater than some specified complexity.

Usually it is formulated so that the problem belongs to a particular complexity class.

Examples of propositions dealing with upper bounds on the complexity:

- The problem of reachability in a graph is in PTIME.
- The problem of equivalence of two regular expressions is in EXPSPACE.

If we want to find some upper bound on the complexity of a problem it is sufficient to show that there is an algorithm with a given complexity.

Upper and Lower Bounds on Complexity of Problems

A **lower bound** on a complexity of a problem means that the complexity of the problem is at least as big as some specified complexity.

In general, proving of (nontrivial) lower bounds is more difficult than proving of upper bounds.

To derive a lower bound we must prove that **every** algorithm solving the given problem has the given complexity.

Upper and Lower Bounds on Complexity of Problems

Problem "Sorting"

Input: Sequence of elements $a_1, a_2, \ldots, a_n$.

Output: Elements $a_1, a_2, \ldots, a_n$ sorted from the smallest to the greatest.

It can be proven that every algorithm, that solves the problem "Sorting" and that has the property that the only operation applied on elements of a sorted sequence is a comparison (i.e., it does not examine the content of these elements), has the time complexity in the worst case $\Omega(n \log n)$ (i.e., for every such algorithm there exist constants $c > 0$ and $n \geq n_0$ such that for every $n \geq n_0$ there is an input of size n, for which the algorithm performs at least $cn \log n$ operations.)

Nodeterministic Algorithms and Complexity Classes

Nondeterminism

So far, we have considered only deterministic algorithms.

We can also consider **nondeterministic** algorithms performed by nondeterministic variants of various kinds of machines:

- Turing machines
- RAMs
- ...

In general, nondeterministic algorithms are algorithms where:

- In every step, the algorithm can choose from several possibilities of a next instruction that will be performed.
- If at least one of computations of such a machine on a given input ends with the answer YES, then the answer is YES.
- If all computations end with the answer NO then the answer is NO.

Nondeterminism

For example, for one-tape Turing machine, the only difference between the deterministic and nondeterministic variant is in the definition of transition function δ:

- **deterministic**: $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$
- **nondeterministic**: $\delta : (Q - F) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{-1, 0, +1\})$

Nondeterministic RAM:

- Its definition is very similar to that of a deterministic RAM.
- Moreover, it has an instruction

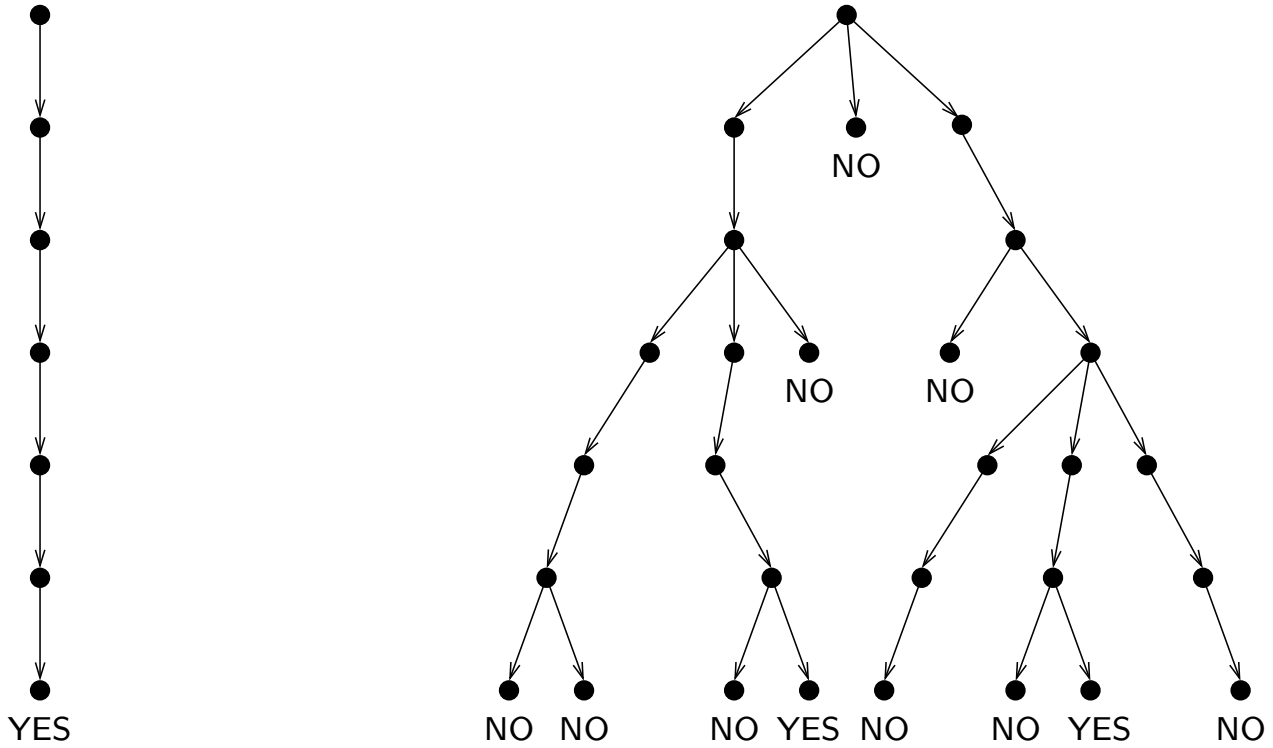
  nd_goto ℓ_1, ℓ_2

  that allows it to choose the next instruction from two possibilities.

We can define nondeterministic variants of arbitrary other models of computation in a similar way.

Nondeterminism

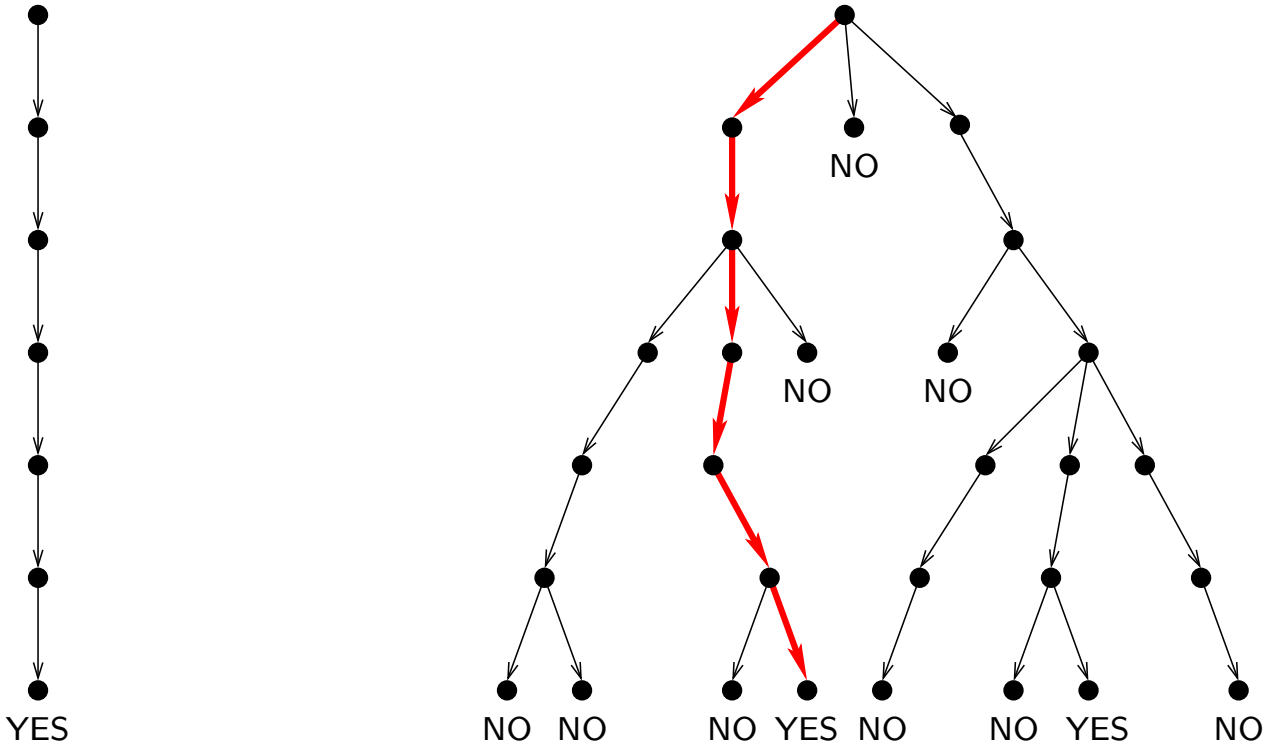
A nondeterministic algorithm gives the answer YES for a given input x if there **exists** at least one computation of this machine that gives answer YES.



- The time required for a computation of a nondeterministic RAM (or other nondeterministic machine) on a given input is defined as the length of the longest computation on the input.

Nondeterminism

A nondeterministic algorithm gives the answer YES for a given input x if there **exists** at least one computation of this machine that gives answer YES.



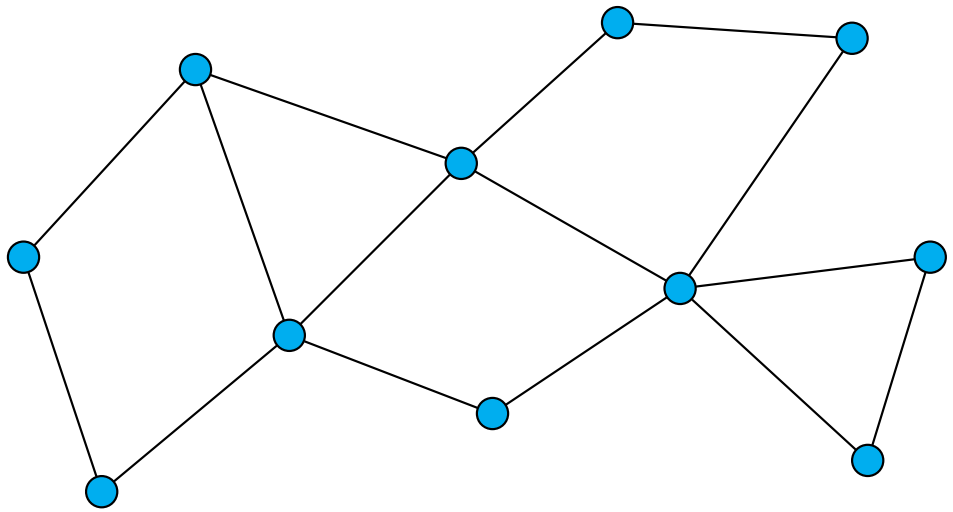
- The time required for a computation of a nondeterministic RAM (or other nondeterministic machine) on a given input is defined as the length of the longest computation on the input.

Nondeterminism

Problem "Coloring of a graph with k colors"

Input: An undirected graph G and a natural number k.

Question: Is it possible to color the nodes of the graph G with k colors in such a way that no two nodes connected with an edge are colored with the same color?



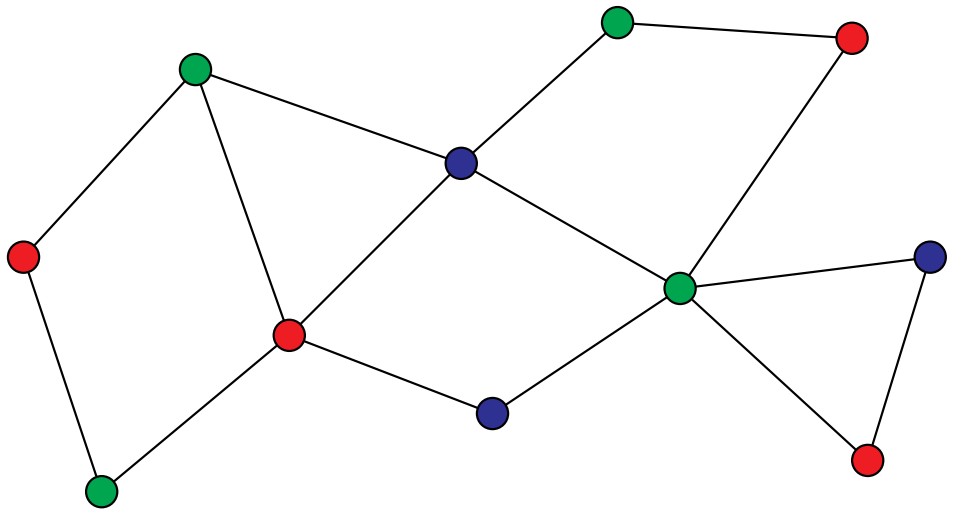
$k = 3$

Nondeterminism

Problem "Coloring of a graph with k colors"

Input: An undirected graph G and a natural number k.

Question: Is it possible to color the nodes of the graph G with k colors in such a way that no two nodes connected with an edge are colored with the same color?



$k = 3$

Nondeterminism

Problem "Coloring of a graph with k colors"

Input: An undirected graph G and a natural number k.

Question: Is it possible to color the nodes of the graph G with k colors in such a way that no two nodes connected with an edge are colored with the same color?

A nondeterministic algorithm works as follows:

1. It assignes nondeterministically to every node of G one of k colors.
2. It goes through all edges of G and for each of them verifies that its endpoints are colored with different colors. If this is not the case, it halts with the answer NO.
3. If it has verified for all edges that their endpoints are colored with different colors, it halts with the answer YES.

Nondeterminism

Problem "Graph isomorphism"

Input: Undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

Question: Are graphs G_1 and G_2 isomorphic?

Remark: Graphs G_1 and G_2 are isomorphic if there exists some bijection $f : V_1 \to V_2$ such that for every pair of nodes $u, v \in V_1$ is $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$.

A nondeterministic algorithm works as follows:

1. It nondeterministically chooses values of the function f for every $v \in V_1$.
2. It (deterministically) verifies that f is a bijection and that the above mentioned condition is satisfied for all pairs of nodes.
3. If some of the conditions is violated, it halts with the answer No. Otherwise it halts with the answer Yes.

Nondeterminism

Boolean formula φ is **satisfiable** if there exists some truth valuation ν, for which the formula φ is true, i.e., has truth value 1.

SAT (boolean satisfiability problem)

Input: Boolean formula φ.

Question: Is φ satisfiable?

Example:

Formula $\varphi_1 = x_1 \wedge (\neg x_2 \vee x_3)$ is satisfiable:
e.g., for valuation ν where $\nu(x_1) = 1$, $\nu(x_2) = 0$, $\nu(x_3) = 1$, the formula φ_1 is true.

Formula $\varphi_2 = (x_1 \wedge \neg x_1) \vee (\neg x_2 \wedge x_3 \wedge x_2)$ is not satisfiable:
it is false for every valuation ν.

Nondeterminism

A nondeterministic algorithm solving the SAT problem in polynomial time:

- It reads a formula φ.
- It cycles through all variables $x_1, x_2, \ldots, x_k$ occurring in the formula φ.

  For each of these variables, it nondeterministically chooses if it assigns value 0 or value 1 to it.
- It evaluates the truth value of formula φ for the generated valuation.

  It halts with answer YES if the formula φ is true for the give valuation.

  Otherwise, it halts with answer NO.

Nondeterminism

- For decidability of problems, the nondeterministic algorithms are not more powerful than deterministic ones:
  If a problem can be solved by a nondeterministic RAM or TM, it can be also solved by a deterministic RAM or TM that successively tries all possible computations of the nondeterministic machine on a given input.
- Nondeterminism is useful primarily in the study of a complexity of problems.

Nondeterminism

- In the straightforward simulation of a nondeterministic algorithm by a deterministic, described above, where the deterministic algorithm systematically tries all possible computations, the time complexity of the deterministic algorithm is exponentially bigger than in the nondeterministic algorithm.
- For many problems, it is clear that there exists a nondeterministic algorithm with a polynomial time complexity solving the given problem but it is not clear at all whether there also exists a deterministic algorithm solving the same problem with a polynomial time complexity.

Nondeterminism

Nondeterminism can be viewed in two different ways:

1. When a machine should nondeterministically choose between several possibilities, it "guesses" which of these possibilities will lead to the answer YES (if there is such a possibility).

2. When a machine should choose between several possibilities, it splits itself into several copies, each corresponding to one of the possibilities. These copies continue in the computation in parallel.

   The answer is YES iff at least one of these copies halts with the answer YES.

None of these possibilities is something that could be efficiently realistically implemented.

Nondeterminism

Other possible view of the nondeterminism:

- A kind of an algorithm that does not solve the given problem but using an additional information — called **witness** — can **verify** that the answer for the given instance is YES.

  Let us assume that in the original problem the input is some x from the set of instances In and the question is whether this x has some specified property P.

  For the given input x, there is a corresponding set $\mathcal{W}(x)$ of **potential witnesses** with the property that x has the property P iff there exists an **actual witness** $y \in \mathcal{W}(w)$ of the fact that x really has property P.

  There is a **deterministic** algorithm Alg that expects as input a pair (x, y) (where $y \in \mathcal{W}(x)$) and that checks that y is a witness of the fact that x has property P.

Nondeterminism

Example: The problem "Graph Colouring with k colours":

- *Input*: An undirected graph $G = (V, E)$ and number k.
- *Potential witnesses*: All possible colourings of nodes of graph G with k colours, i.e., all functions c of the form $c : V \to \{1, \ldots, k\}$.
- *Actual witnesses*: Those colourings c where for each edge $(u, v) \in E$ holds that $c(u) \neq c(v)$.

Nondeterminism

- For each **deterministic** algorithm Alg that can verify for a given pair (x, y) that y is a witness of the fact that x has property P, we can easily construct a corresponding **nondeterministic** algorithm that solves the original problem:
  - For a given $x \in In$ it generates nondeterministically a potential witness $y \in \mathcal{W}(x)$.
  - Then it uses the (deterministic) algorithm Alg as a subroutine to check that y is an actual witness.

Nondeterminism

- On the contrary, for every **nondeterministic** algorithm, we can also easily construct a **deterministic** algorithm for checking witnesses:
  - A potential witness will be a sequence specifying for each nondeterministic step of the original algorithm, which possibility should be chosen in the given step.
  - The deterministic algorithm then simulates one particular computation (one branch of the tree) of the original algorithm where in those steps where several choices are possible, it does not guess but continues according to the sequence given as a witness.

Nondeterminism

We will concentrate particularly to those cases where the time complexity of the algorithm for checking a witness is polynomial with respect to the size of input x.

This also means that a given witness y, witnessing that the answer for x is YES, must be of a polynomial size.

So by a nondeterministic algorithm with a polynomial time complexity we can solve those decision problems where:

- for a given input x there exists a corresponding (polynomially big) witness iff the answer for x is YES,
- it is possible to check using a deterministic algorithm in polynomial time that a given potential witness is really a witness.

Nondeterminism

In many cases, the existence of such polynomially big witnesses and deterministic algorithms checking them is obvious and it is trivial to show that they exist — e.g., for problems like "Graph Colouring with k Colours", "Graph Isomorphism", or the following problem:

Testing that a number is composite

Input: A natural number x.

Question: Is the number x composite?

Remark: Number x is **composite** if there exist natural numbers a and b such that $a > 1$, $b > 1$, and $x = a \cdot b$.

For example, number 15 is composite because $15 = 3 \cdot 5$.

So the number $x \in \mathbb{N}$ is composite iff $x > 1$ and x is not a prime.

Existence of such polynomially big witnesses of course does not automatically mean that it is easy to find them.

Nondeterminism

For some problems, a proof of existence of such polynomially bounded witnesses, which can be checked deterministically in a polynomial time, rather nontrivial result.

An example can be the following problem:

Primality Testing

Input: A natural number x.

Question: Is number x a prime?

Using some nontrivial results from number theory, there can be shown existence of such witnesses even for this problem — those witnesses here are rather complicated recursively defined data structures.

Remark: This result was shown by V. Pratt in 1975.

Much later it was shown that "Primality Testing" is in PTIME (Agrawal–Kayal–Saxena, 2002).

Nondeterministic Complexity Classes

Definition

For a function $f : \mathbb{N} \to \mathbb{N}$ we define the **time complexity class** $\mathrm{NTIME}(f)$ as the set of all problems that are solved by nondeterministic RAMs with a time complexity in $\mathcal{O}(f(n))$.

Definition

For a function $f : \mathbb{N} \to \mathbb{N}$ we define the **space complexity class** $\mathrm{NSPACE}(f)$ as the set of all problems that are solved by nondeterministic RAMs with a space complexity in $\mathcal{O}(f(n))$.

Remark: Of course, the definitions given above can also use Turing machines or some other model of computation instead of RAMs.

Class NPTIME

Definition

$$\text{NPTIME} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$$

- NPTIME (sometimes we write just NP) is the class of all problems, for which there exists a nondeterministic algorithm with polynomial time complexity.
- The class NPTIME contains those problems for which it is possible to verify in polynomial time that the answer is YES if somebody, who wants to convince us that this is really the case, provides additional information.

Classes NPSPACE, NEXPTIME, NEXPSPACE, ...

Other classes can be defined similarly:

NPSPACE – the set of all decision problems, for which there exists a nondeterministic algorithm with polynomial space complexity

NEXPTIME – the set of all decision problems, for which there exists an algorithm with time complexity $2^{\mathcal{O}(n^k)}$ where k is a constant

NEXPSPACE – the set of all decision problems for which there exists an algorithm with space complexity $2^{\mathcal{O}(n^k)}$ where k is a constant

NLOGSPACE – the set of all decision problems for which there exists an algorithm with space complexity $\mathcal{O}(\log n)$

Names of complexity classes

Remarks: The following shorter names are commonly used in literature.

These shorter names with be sometimes used also in this course:

L	–	LOGSPACE
NL	–	NLOGSPACE
P	–	PTIME
NP	–	NPTIME
EXP	–	EXPTIME
NEXP	–	NEXPTIME

Relationships between Complexity Classes

It is clear that deterministic algorithms can be viewed as a special case of nondeterministic algorithms.

Therefore it obviously holds that:

$$\begin{aligned} \text{LOGSPACE} &\subseteq \text{NLOGSPACE} \\ \text{PTIME} &\subseteq \text{NPTIME} \\ \text{PSPACE} &\subseteq \text{NPSPACE} \\ \text{EXPTIME} &\subseteq \text{NEXPTIME} \\ \text{EXPSPACE} &\subseteq \text{NEXPSPACE} \\ &\vdots \end{aligned}$$

Relationships between Complexity Classes

It is also obvious that for both deterministic and nondeterministic algorithms, an algorithm can not use considerably bigger number of memory cells than what is the number of steps executed by the algorithm.

A space complexity of an algorithm is therefore always at most as big as its time complexity.

From this follows that:

$$\begin{aligned} \text{PTIME} &\subseteq \text{PSPACE} \\ \text{NPTIME} &\subseteq \text{NPSPACE} \\ \text{EXPTIME} &\subseteq \text{EXPSPACE} \\ \text{NEXPTIME} &\subseteq \text{NEXPSPACE} \\ &\vdots \end{aligned}$$

Relationships between Complexity Classes

Theorem

A **nondeterministic** algorithm with **time** complexity $\mathcal{O}(f(n))$ can be simulated by a **deterministic** algorithm with **space** complexity $\mathcal{O}(f(n))$.

Proof: Consider a **nondeterministic** algorithm with **time** complexity $\mathcal{O}(f(n))$.

The deterministic algorithm will simulate its behaviour by systematically going through all its computations:

- It goes through the tree of all computations using a depth-first search.
- It will use a stack to store information necessary for returning to the previous configuration.
  — to allow to go back to a previous configuration α from a following configuration α', it is sufficient to store a constant amount of information — only those things that were changed in th transition from α to α'

Relationships between Complexity Classes

This simulating algorithm needs the following memory:

- a memory to store a current configuration of the simulated machine — its size is $\mathcal{O}(f(n))$ (since if this simulated machine performs at most $\mathcal{O}(f(n))$ steps then its configurations will use at most $\mathcal{O}(f(n))$ memory cells)
- a memory to store a stack that will be used to allow returning to previous configurations

  Since the length of branches is $\mathcal{O}(f(n))$, the amount of memory needed for the stack is $\mathcal{O}(f(n))$.

So in total, the deterministic algorithm uses in this simulation an amount of memory, which is at most $\mathcal{O}(f(n))$.

Relationships between Complexity Classes

It follows from this that:

$$\begin{aligned} \text{NPTIME} &\subseteq \text{PSPACE} \\ \text{NEXPTIME} &\subseteq \text{EXPSPACE} \\ &\vdots \end{aligned}$$

Relationships between Complexity Classes

Consider a **nondeterministic** algorithm with a **space** complexity $\mathcal{O}(f(n))$:

- Let us recall that the total number of configurations of size at most $\mathcal{O}(f(n))$ is $\mathcal{O}(c^{f(n)})$, where c is a constant, so this can be written as $2^{\mathcal{O}(f(n))}$.
- So the number of steps of the nondeterministic algorithm in one branch of computation could be at most $2^{\mathcal{O}(f(n))}$.

  (Remark: No configuration can be repeated during a computation since otherwise computations could be infinite.)
- So the simulation done this way would have time complexity $2^{2^{\mathcal{O}(f(n))}}$.

Relationships between Complexity Classes

In a simulation we can proceed in a more clever way — consider a directed graph where:

- **nodes** — all configurations of the simulated machine whose size is at most $\mathcal{O}(f(n))$
  — the number of such configurations is $2^{\mathcal{O}(f(n))}$
- **edges** — there is an edge between nodes representing configurations α and α' iff the simulated machine can go in one step from configuration α to configuration α'
  — the number of edges going out from each node is bounded from above by some constant — so the number of edges is also $2^{\mathcal{O}(f(n))}$

It is sufficient to be able to find out whether there is a path in this graph from the node corresponding to the initial configuration (for the given input x) to some node corresponding to a final configuration where the machine gives answer YES.

Relationships between Complexity Classes

Existence of such a path can be tested using an arbitrary algorithm for searching a graph — for example by breadth-first search or depth-first search:

- This algorithm needs to store and mark, which configurations have been already visited.
  It also needs a memory to store a queue or a stack, etc.
- The time and space complexity of such algorithm is linear with respect to the size of the graph, i.e., $2^{\mathcal{O}(f(n))}$.

Relationships between Complexity Classes

So we obtain the following:

Theorem

The behaviour of a nondeterministic algorithm whose space complexity is $\mathcal{O}(f(n))$ can be simulated by a deterministic algorithm with time complexity $2^{\mathcal{O}(f(n))}$.

It follows from this that:

$$\begin{aligned} \text{NLOGSPACE} &\subseteq \text{PTIME} \\ \text{NPSPACE} &\subseteq \text{EXPTIME} \\ \text{NEXPSPACE} &\subseteq \text{2-EXPTIME} \\ &\vdots \end{aligned}$$

Relationships between Complexity Classes

Consider once again a **nondeterministic** algorithm with **space** complexity $\mathcal{O}(f(n))$. Now we would like to have the **space** complexity of the simulating deterministic algorithm as small as possible.

Theorem (Savitch, 1970)

The behaviour of a nondeterministic algorithm with space complexity $\mathcal{O}(f(n))$ can be simulated by a deterministic algorithm with space complexity $\mathcal{O}(f(n)^2)$.

Proof idea:

- Consider once again the graph of configurations with $2^{\mathcal{O}(f(n))}$ nodes (and edges).
- The algorithm will try to find out whether there exists a path from the initial configuration to some accepting configuration.

Relationships between Complexity Classes

The most important part is a recursive function $F(\alpha, \alpha', i)$ that for arbitrary configurations α and α' and number $i \in \mathbb{N}$ finds out whether the given graph contains a path from α to α' of length at most 2^i:

- For $i = 0$ it finds out whether there is a path from α to α' of length at most 1:
  - it is either a path of length 0, i.e., $\alpha = \alpha'$,
  - or it is a path of length 1, i.e., it is possible to go from α to α' in one step
- For $i > 0$, it will systematically try all configurations α'' and check whether:
  - there is a path of length at most $2^i/2$ from α to α'' — it calls $F(\alpha, \alpha'', i-1)$ recursively
  - there is a path of length at most $2^i/2$ from α'' to α' — it calls $F(\alpha'', \alpha', i-1)$ recursively

  If both returns $\textsc{True}$, it returns $\textsc{True}$, otherwise it continues with trying the next α''.

Relationships between Complexity Classes

The analysis of the space complexity of the algorithm:

- in one recursive call of the function F, the algorithm needs to store:
  - three configurations α, α', α'' — all of them of size $\mathcal{O}(f(n))$
  - the value of the number i, which is approximately $\mathcal{O}(f(n))$ — so to store this number, $\mathcal{O}(\log f(n))$ bits are sufficient
  - other auxiliary variables whose sizes are negligible compared to the sizes of the values described above
- So the amount of memory needed for one recursive call is $\mathcal{O}(f(n))$.
- The depth of the recursion is also $\mathcal{O}(f(n))$.
- So the total space complexity of the algorithm is $\mathcal{O}(f(n)^2)$.

Relationships between Complexity Classes

It follows from this theorem that:

$$\begin{aligned} \mathrm{NPSPACE} &\subseteq \mathrm{PSPACE} \\ \mathrm{NEXPSPACE} &\subseteq \mathrm{EXPSPACE} \\ &\vdots \end{aligned}$$

Together with the trivial facts that $\mathrm{PSPACE} \subseteq \mathrm{NPSPACE}$, $\mathrm{EXPSPACE} \subseteq \mathrm{NEXPSPACE}$, atd., this implies:

$$\begin{aligned} \mathrm{PSPACE} &= \mathrm{NPSPACE} \\ \mathrm{EXPSPACE} &= \mathrm{NEXPSPACE} \\ &\vdots \end{aligned}$$

Remark: Note that it **does not follow** from this that $\mathrm{LOGSPACE} = \mathrm{NLOGSPACE}$.

Relationships between Complexity Classes

Putting all this together, we obtain the following **hierarchy of complexity classes**:

$$\begin{gathered}
\mathrm{LOGSPACE} \subseteq \mathrm{NLOGSPACE} \subseteq \\
\subseteq \mathrm{PTIME} \subseteq \mathrm{NPTIME} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \\
\subseteq \mathrm{EXPTIME} \subseteq \mathrm{NEXPTIME} \subseteq \mathrm{EXPSPACE} = \mathrm{NEXPSPACE} \subseteq \\
\vdots
\end{gathered}$$