Undecidable Problems

Algorithmically Solvable Problems

Let us assume we have a problem P.

If there is an algorithm solving the problem P then we say that the problem P is **algorithmically solvable**.

If P is a decision problem and there is an algorithm solving the problem P then we say that the problem P is **decidable (by an algorithm)**.

If we want to show that a problem P is algorithmically solvable, it is sufficient to show some algorithm solving it (and possibly show that the algorithm really solves the problem P).

Algorithmically Unsolvable Problems

A problem that is not algorithmically solvable is **algorithmically unsolvable**.

A decision problem that is not decidable is **undecidable**.

Surprisingly, there are many (exactly defined) problems, for which it was proved that they are not algorithmically solvable.

Halting Problem

Let us consider some general programming language $\mathcal{L}$.

Futhermore, let us assume that programs in language $\mathcal{L}$ run on some idealized machine where a (potentially) unbounded amount of memory is available — i.e., the allocation of memory never fails.

Example: The following problem called the **Halting problem** is undecidable:

Halting problem

Input:	A source code of a $\mathcal{L}$ program P, input data x.
Question:	Does the computation of P on the input x halt after some finite number of steps?

Halting Problem

Let us assume that there is a program that can decide the Halting problem.

So we could construct a subroutine H, declared as

Bool H(String code, String input)

where $H(P, x)$ returns:

- true if the program P halts on the input x,
- false if the program P does not halt on the input x.

Remark: Let us say that subroutine $H(P, x)$ returns false if P is not a syntactically correct program.

Halting Problem

Using the subroutine H we can construct a program D that performs the following steps:

- It reads its input into a variable x of type String.
- It calls the subroutine $H(x, x)$.
- If subroutine H returns true, program D jumps into an infinite loop

  ```
  loop: goto loop
  ```

  In case that H returns false, program D halts.

What does the program D do if it gets its own code as an input?

Halting Problem

If D gets its own code as an input, it either halts or not.

- If D halts then $H(D, D)$ returns true and D jumps into the infinite loop. A contradiction!
- If D does not halt then $H(D, D)$ returns false and D halts. A contradiction!

In both case we obtain a contradiction and there is no other possibility. So the assumption that H solves the Halting problem must be wrong.

Semidecidable Problems

A problem is **semidecidable** if there is an algorithm such that:

- If it gets as an input an instance for which the answer is YES, then it halts after some finite number of steps and writes `"YES"` on the output.
- If it gets as an input an instance for which the answer is NO, then it either halts and writes `"NO"` on the input, or does not halt and runs forever.

It is obvious that for example HP (Halting Problem) is semidecidable.

Some problems are not even semidecidable.

Post's Theorem

The **complement** problem for a given decision problem P is a problem where inputs are the same as for the problem P and the question is negation of the question from the problem P.

Post's Theorem

If a problem P and its complement problem are semidecidable then the problem P is decidable.

Reduction between Problems

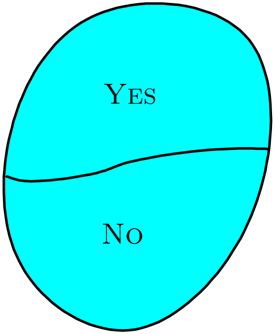
If we have already proved a (decision) problem to be undecidable, we can prove undecidability of other problems by reductions.

Problem P_1 can be **reduced** to problem P_2 if there is an algorithm Alg such that:

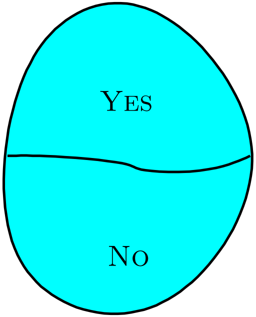
- It can get an arbitrary instance of problem P_1 as an input.
- For an instance of a problem P_1 obtained as an input (let us denote it as w) it produces an instance of a problem P_2 as an output.
- It holds i.e., the answer for the input w of problem P_1 is $\textsc{Yes}$ iff the answer for the input $Alg(w)$ of problem P_2 is $\textsc{Yes}$.

Reductions between Problems

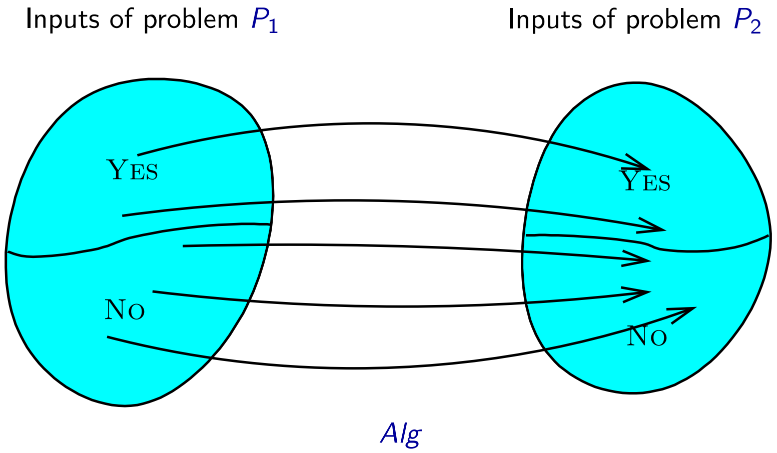
Inputs of problem P_1



Inputs of problem P_2



Reductions between Problems



Reductions between Problems

Let us say there is some reduction Alg from problem P_1 to problem P_2.

If problem P_2 is decidable then problem P_1 is also decidable.

Solution of problem P_1 for an input x:

- Call Alg with x as an input, it returns a value $Alg(x)$.
- Call the algorithm solving problem P_2 with input $Alg(x)$.
- Write the returned value to the output as the result.

It is obvious that if P_1 is undecidable then P_2 cannot be decidable.

Other Undecidable Problems

By reductions from the Halting problem we can show undecidability of many other problems dealing with a behaviour of programs:

- Is for some input the output of a given program YES?
- Does a given program halt for an arbitrary input?
- Do two given programs produce the same outputs for the same inputs?
- ...

Halting Problem

For purposes of proofs, the following version of Halting problem is often used:

Halting problem

Input:	A description of a Turing machine $\mathcal{M}$ and a word w.
Question:	Does the computation of the machine $\mathcal{M}$ on the word w halt after some finite number of steps?

Other Undecidable Problems

We have already seen the following example of an undecidable problem:

Problem

Input: Context-free grammars $\mathcal{G}_1$ and $\mathcal{G}_2$.

Question: Is $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$?

respectively

Problem

Input: A context-free grammar generating a language over an alphabet Σ.

Question: Is $\mathcal{L}(\mathcal{G}) = \Sigma^*$?

Other Undecidable Problems

An input is a set of types of cards, such as:

abb	a	bab	baba	aba
bbab	aa	ab	aa	a

The question is whether it is possible to construct from the given types of cards a non-empty finite sequence such that the concatenations of the words in the upper row and in the lower row are the same. Every type of a card can be used repeatedly.

a	abb	abb	baba	abb	aba
aa	bbab	bbab	aa	bbab	a

In the upper and in the lower row we obtained the word `aabbabbbabaabbaba`.

Other Undecidable Problems

Undecidability of several other problems dealing with context-free grammars can be proved by reductions from the previous problem:

Problem

Input: Context-free grammars $\mathcal{G}_1$ and $\mathcal{G}_2$.

Question: Is $\mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2) = \emptyset$?

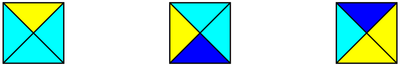
Problem

Input: A context-free grammar $\mathcal{G}$.

Question: Is $\mathcal{G}$ ambiguous?

Other Undecidable Problems

An input is a set of types of tiles, such as:

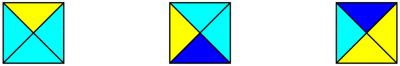


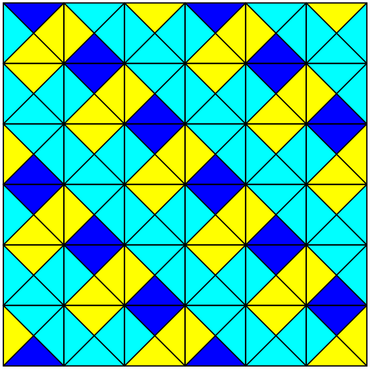
The question is whether it is possible to cover every finite area of an arbitrary size using the given types of tiles in such a way that the colors of neighboring tiles agree.

Remark: We can assume that we have an infinite number of tiles of all types.

The tiles cannot be rotated.

Other Undecidable Problems





Other Undecidable Problems

Problem

Input: A closed formula of the first order predicate logic where the only predicate symbols are $=$ and $<$, the only function symbols are $+$ and $*$, and the only constant symbols are 0 and 1.

Question: Is the given formula true in the domain of natural numbers (using the natural interpretation of all function and predicate symbols)?

An example of an input:

$$\forall x \exists y \forall z ((x * y = z) \land (y + 1 = x))$$

Remark: There is a close connection with Gödel's incompleteness theorem.

Other Undecidable Problems

It is interesting that an analogous problem, where real numbers are considered instead of natural numbers, is decidable (but the algorithm for it and the proof of its correctness are quite nontrivial).

Also when we consider natural numbers or integers and the same formulas as in the previous case but with the restriction that it is not allowed to use the multiplication function symbol $*$, the problem is algorithmically decidable.

Other Undecidable Problems

If the function symbol $*$ can be used then even the very restricted case is undecidable:

Hilbert's tenth problem

Input: A polynomial $f(x_1, x_2, \ldots, x_n)$ constructed from variables $x_1, x_2, \ldots, x_n$ and integer constants.

Question: Are there some natural numbers $x_1, x_2, \ldots, x_n$ such that $f(x_1, x_2, \ldots, x_n) = 0$?

An example of an input: $5x^2y - 8yz + 3z^2 - 15$

I.e., the question is whether

$$\exists x \exists y \exists z (5 * x * x * y + (-8) * y * z + 3 * z * z + (-15) = 0)$$

holds in the domain of natural numbers.

Other Undecidable Problems

Also the following problem is algorithmically undecidable:

Problem

Input:	A closed formula φ of the first-order predicate logic.
Question:	Is $\models \varphi$?

Remark: Notation $\models \varphi$ denotes that formula φ is logically valid, i.e., it is true in all interpretations.

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 $\mathcal{T}(f(n))$ as the class containing exactly those decision problems for which there exists an algorithm with time complexity $O(f(n))$.

Example:

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

Complexity Classes

Definition

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

Example:

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

Complexity Classes

Remark:

Note that for classed $\mathcal{T}(f)$ and $\mathcal{S}(f)$ 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 $\mathcal{T}(f(n))$ and $\mathcal{S}(f(n))$ we can define classes PTIME and PSPACE as

$$\text{PTIME} = \bigcup_{k \geq 0} \mathcal{T}(n^k) \qquad\qquad \text{PSPACE} = \bigcup_{k \geq 0} \mathcal{S}(n^k)$$

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

Complexity Classes

Remark: 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.

Complexity Classes

Other classes are introduced analogously:

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

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

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

Remark: Instead of $2^{O(n^k)}$ we can also write $O(c^{n^k})$ 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 $2^{2^{O(n^k)}}$ where k is a constant

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

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

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

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 $O(f(n))$, the space complexity of this algorithm is (at most) $O(f(n))$.

So it is obvious that the following relationship holds.

Observation

For every function $f : \mathbb{N} \to \mathbb{N}$ is $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$.

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 every pair of real numbers ϵ_1 a ϵ_2 taková, že $0 \leq \epsilon_1 < \epsilon_2$, is

$$\mathcal{S}(n^{\epsilon_1}) \subsetneq \mathcal{S}(n^{\epsilon_2})$$

- LOGSPACE $\subsetneq$ PSPACE
- PSPACE $\subsetneq$ EXPSPACE
- For every pair of real numbers ϵ_1 a ϵ_2 taková, že $0 \leq \epsilon_1 < \epsilon_2$, is

$$\mathcal{T}(n^{\epsilon_1}) \subsetneq \mathcal{T}(n^{\epsilon_2})$$

- PTIME $\subsetneq$ EXPTIME
- EXPTIME $\subsetneq$ 2-EXPTIME

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

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

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 $O(f(n))$, it means that the number of different configurations that are reachable during a computation is $2^{O(f(n))}$.

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

Observation

For every function $f : \mathbb{N} \to \mathbb{N}$ it holds that pokud je nějaký problém P řešený algoritmem s prostorovou složitostí $O(f(n))$, pak časová složitost tohoto algoritmu je v $2^{O(f(n))}$.

Pokud je tedy problém P ve třídě $\mathcal{S}(f(n))$, pak je i ve třídě $\mathcal{T}(2^{c \cdot f(n)})$ pro nějaké $c > 0$.

Relationships between Complexity Classes

The following results can be drawn from the previous discussion:

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

Relationships between Complexity Classes

Summary:

$$\begin{aligned}&\mathrm{LOGSPACE} \subseteq \mathrm{PTIME} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXPSPACE} \subseteq \\ &\qquad \subseteq \text{2-EXPTIME} \subseteq \text{2-EXPSPACE} \subseteq \cdots \subseteq \mathrm{ELEMENTARY}\end{aligned}$$

- $\mathrm{PTIME} \subsetneq \mathrm{EXPTIME} \subsetneq \text{2-EXPTIME} \subsetneq \cdots$
- $\mathrm{LOGSPACE} \subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE} \subsetneq \text{2-EXPSPACE}, \subsetneq \cdots$

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

Nondeterministic RAM:

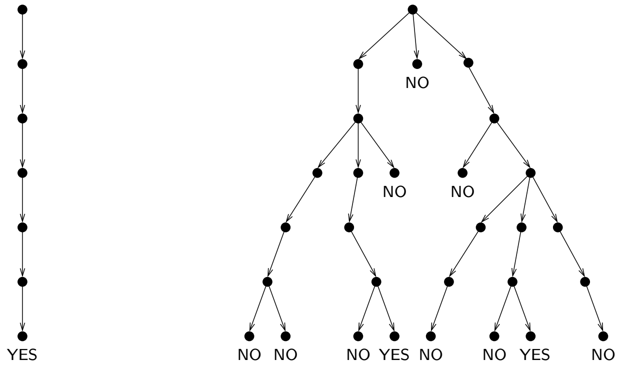
- Its definition is very similar to that of a deterministic RAM.
- Moreover, it has an instruction

  $$\textbf{nd\_goto}\ \ell_1,\ \ell_2$$

  that allows it to choose the next instruction from two possibilities.
- If at least one of computations of such a machine on a given input ends with the answer $\textsc{Yes}$, then the answer is $\textsc{Yes}$.
- If all computations end with the answer $\textsc{No}$ then the answer is $\textsc{No}$.

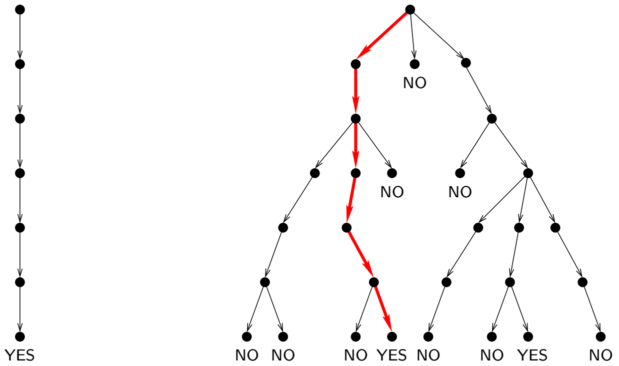
Nondeterministic versions of other computational models (such as nondeterministic Turing machines) are defined similarly.

Nondeterminism



- 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



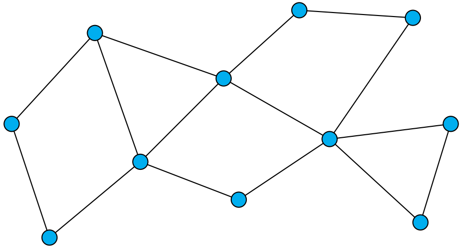
- The time required for a computation of a nondeterministic RAM (or other nondeterministic machine) on a given input is defined as the length of the longest computation on the input.

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?



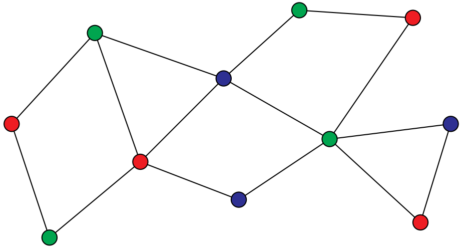
$k = 3$

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

- 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 $W(x)$ of **potential witnesses** with the property that x has the property P iff there exists an **actual witness** $y \in W(w)$ of the fact that x really has property P.

  There is a **deterministic** algorithm Alg that expects as input a pair (x, y) (where $y \in W(x)$) 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 $y \in W(x)$.
  - 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 $\textsc{Yes}$, 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 $\textsc{Yes}$,
- 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** $\mathcal{NT}(f)$ as the set of all problems that are solved by nondeterministic RAMs with a time complexity in $O(f(n))$.

Definition

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

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

Class NPTIME

Definition

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

- 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 – množina všech rozhodovacích problémů, pro které existuje nedeterministický algoritmus s polynomiání prostorovou složitostí

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

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

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

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:

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

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:

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

Relationships between Complexity Classes

Consider a **nondeterministic** algorithm with **time** complexity $O(f(n))$.

A **deterministic** algorithm that will simulate its behaviour by systematically trying all its possible computations (by going through the tree of these computations in a depth-first manner) will need only the following memory:

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

Relationships between Complexity Classes

- Since the length of branches is $O(f(n))$, the amount of memory needed for the stack is $O(f(n))$.
- So in total, the deterministic algorithm uses in this simulation an amount of memory, which is at most $O(f(n))$.

It follows from this that:

$$\begin{gathered}\mathrm{NPTIME} \subseteq \mathrm{PSPACE} \\ \mathrm{NEXPTIME} \subseteq \mathrm{EXPSPACE} \\ \vdots\end{gathered}$$

Relationships between Complexity Classes

Consider a **nondeterministic** algorithm with a **space** complexity $O(f(n))$:

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

  (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 $2^{2^{O(f(n))}}$.

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 $O(f(n))$
  — the number of such configurations is $2^{O(f(n))}$
- **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 $2^{O(f(n))}$

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 $\textsc{Yes}$.

Relationships between Complexity Classes

Existence of such a path can be tested using an arbitrary algorithm for searching a graph — breadth-first search, 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., $2^{O(f(n))}$.

Relationships between Complexity Classes

So we obtain the following:

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

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 $O(f(n))$. 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 $O(f(n))$ can be simulated by a deterministic algorithm with space complexity $O(f(n)^2)$.

Proof idea:

- Consider once again the graph of configurations with $2^{O(f(n))}$ 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 TRUE, it returns 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 $O(f(n))$
    - the value of the number i, which is approximately $O(f(n))$ — so to store this number, $O(\log F(n))$ 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 $O(f(n))$.
- The depth of the recursion is also $O(f(n))$.
- So the total space complexity of the algorithm is $O(f(n)^2)$.

Relationships between Complexity Classes

It follows from this theorem that:

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

Together with the trivial facts that PSPACE ⊆ NPSPACE, EXPSPACE ⊆ NEXPSPACE, . . . this implies:

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

Remark: Note that it **does not follow** from this that LOGSPACE = NLOGSPACE.

Relationships between Complexity Classes

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

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