Basics on Complexity Theory

Mika Hirvensalo

Department of Mathematics and Statistics
University of Turku
mikhirve@utu.fi

Thessaloniki, May 2014
About Turku

Established 13th century, capital of Finland until 1812

Population 180,000 (city area) 300,000 (subregion area)
Established 13rd century
About Turku

Established 13rd century, capital of Finland until 1812
About Turku

Established 13th century, capital of Finland until 1812
Population 180,000 (city area) 300,000 (subregion area)
About Turku

Mika Hirvensalo
University of Turku

First established 1640
First established 1640
Current one 1920
University of Turku

First established 1640
Current one 1920
22,000 students in seven faculties
Discrete mathematics:
- Words and Automata
- Complex Systems and Computing
- Coding Theory and Cryptography
- FiDiPro group in Combinatorics on Words

Analysis

Applied Mathematics

Statistics
Computational Problems

- **Product** $m, n \mapsto mn$
  - An instance of **Product**: Input $(3, 5)$ (output 15)

- **Factorization** $m \mapsto p$ (smallest prime factor of $m$)
  - An instance of **Factorization**: Input 15 (output 3)

- **Primality** $n \mapsto 0/1$. 1 if $n$ prime, 0 otherwise
  - An instance of **Primality**: Input 7 (output 1)
  - Another instance of **Primality**: Input 8 (output 0)

Factorization seems harder than Product. Primality appears hard. Factorization is at least as hard as Primality (reduction).
**PRODUCT**

\[ m, n \mapsto mn \]

An instance of Product:
- Input (3, 5) (output 15)

**Factorization**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

An instance of Factorization:
- Input 15 (output 3)

**Primality**

\[ n \mapsto 0/1. 1 \text{ if } n \text{ prime, } 0 \text{ otherwise} \]

An instance of Primality:
- Input 7 (output 1)
- Another instance of Primality: Input 8 (output 0)

Factorization seems harder than Product
Primality appears hard
Factorization is at least as hard as Primality (reduction)
**Product**

\[ m, n \mapsto mn \]

- An *instance* of **Product**: Input (3, 5) (output 15)

Factorization seems harder than **Product**.

**Primality**

\[ n \mapsto 0/1. \] 1 if \( n \) prime, 0 otherwise

- An instance of **Primality**: Input 7 (output 1) 
- Another instance of **Primality**: Input 8 (output 0)

Factorization is at least as hard as **Primality** (reduction).
<table>
<thead>
<tr>
<th><strong>PRODUCT</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$m, n \mapsto mn$</td>
</tr>
</tbody>
</table>

- An *instance* of **PRODUCT**: Input $(3, 5)$ (output 15)

<table>
<thead>
<tr>
<th><strong>FACTORORIZATION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \mapsto p$ (smallest prime factor of $m$)</td>
</tr>
</tbody>
</table>

Factorization seems harder than Product.

Primality appears hard.

Factorization is at least as hard as Primality (reduction).
Computational Problems

**PRODUCT**

\[ m, n \mapsto mn \]

- An *instance* of **PRODUCT**: Input (3, 5) (output 15)

**FACTORORIZATION**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

- An instance of **FACTORIZATION**: Input 15 (output 3)

Factorization seems harder than Product, Primality appears hard, Factorization is at least as hard as Primality (reduction).
### Computational Problems

**PRODUCT**

\[ m, n \rightarrow mn \]

- An *instance* of PRODUCT: Input \((3, 5)\) (output 15)

**FACTORORIZATION**

\[ m \rightarrow p \text{ (smallest prime factor of } m) \]

- An instance of FACTORIZATION: Input 15 (output 3)

**PRIMALITY**

\[ n \rightarrow 0/1. 1 \text{ if } n \text{ prime, } 0 \text{ otherwise} \]
**Product**

\[ m, n \mapsto mn \]

- An instance of **Product**: Input (3, 5) (output 15)

**Factorization**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

- An instance of **Factorization**: Input 15 (output 3)

**Primality**

\[ n \mapsto 0/1. \text{ 1 if } n \text{ prime, 0 otherwise} \]

- An instance of **Primality**: Input 7 (output 1)
**Computational Problems**

**PRODUCT**

\[ m, n \mapsto mn \]

- An instance of **PRODUCT**: Input \((3, 5)\) (output 15)

**FACTORORIZATION**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

- An instance of **FACTORIZATION**: Input 15 (output 3)

**PRIMALITY**

\[ n \mapsto 0/1. \text{ 1 if } n \text{ prime, 0 otherwise} \]

- An instance of **PRIMALITY**: Input 7 (output 1)
- Another instance of **PRIMALITY**: Input 8 (output 0)

Factorization seems harder than Product. Primality appears hard. Factorization is at least as hard as Primality (reduction).
Computational Problems

**PRODUCT**

\[ m, n \mapsto mn \]

- An instance of **PRODUCT**: Input (3, 5) (output 15)

**FACTORIZATION**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

- An instance of **FACTORIZATION**: Input 15 (output 3)

**PRIMALITY**

\[ n \mapsto 0/1. 1 \text{ if } n \text{ prime, } 0 \text{ otherwise} \]

- An instance of **PRIMALITY**: Input 7 (output 1)
- Another instance of **PRIMALITY**: Input 8 (output 0)

**FACTORIZATION** seems harder than **PRODUCT**
Computational Problems

**Product**

\[ m, n \mapsto mn \]

- An instance of **Product**: Input \((3, 5)\) (output 15)

**Factorization**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

- An instance of **Factorization**: Input 15 (output 3)

**Primality**

\[ n \mapsto 0/1. \text{ 1 if } n \text{ prime, 0 otherwise} \]

- An instance of **Primality**: Input 7 (output 1)
- Another instance of **Primality**: Input 8 (output 0)
- **Factorization** seems harder than **Product**
- **Primality** appears hard
Computational Problems

**PRODUCT**

\[ m, n \mapsto mn \]

- An instance of **PRODUCT**: Input \((3, 5)\) (output 15)

**FACTORORIZATION**

\[ m \mapsto p \text{ (smallest prime factor of } m) \]

- An instance of **FACTORORIZATION**: Input 15 (output 3)

**PRIMALITY**

\[ n \mapsto 0/1. 1 \text{ if } n \text{ prime, 0 otherwise} \]

- An instance of **PRIMALITY**: Input 7 (output 1)
- Another instance of **PRIMALITY**: Input 8 (output 0)
- **FACTORIZATION** seems harder than **PRODUCT**
- **PRIMALITY** appears hard
- **FACTORIZATION** is at least as hard as **PRIMALITY** (reduction)
Input and output are strings over some alphabet \( \Sigma = \{a_1, \ldots, a_k\} \)
Input and output are strings over some alphabet
\( \Sigma = \{a_1, \ldots, a_k\} \)

Encoding \( a_i \) (and hence input and output) in binary is always possible
Computational Problems

- Input and output are strings over some alphabet
  \( \Sigma = \{a_1, \ldots, a_k\} \)
- Encoding \( a_i \) (and hence input and output) in binary is always possible

Decision problems: Output \( \in \{0, 1\} \)
Computational Problems

- Input and output are strings over some alphabet $\Sigma = \{a_1, \ldots, a_k\}$
- Encoding $a_i$ (and hence input and output) in binary is always possible

**Decision problems:** Output $\in \{0, 1\}$

- A general problem can be presented as a sequence of decision problems: 1:st bit of the output? 2:nd bit of the output? etc.
Input and output are strings over some alphabet 
\[ \Sigma = \{ a_1, \ldots, a_k \} \]

Encoding \( a_i \) (and hence input and output) in binary is always possible

Decision problems: Output \( \in \{ 0, 1 \} \)

A general problem can be presented as a sequence of decision problems: 1:st bit of the output? 2:nd bit of the output? etc.

Computation: Input \( \rightarrow \) Output
Computational Problems

- Input and output are strings over some alphabet \( \Sigma = \{a_1, \ldots, a_k\} \)
- Encoding \( a_i \) (and hence input and output) in binary is always possible

Decision problems: Output \( \in \{0, 1\} \)

- A general problem can be presented as a sequence of decision problems: 1:st bit of the output? 2:nd bit of the output? etc.

Computation: Input \( \xrightarrow{A} \) Output

- What does it take to compute \( A \)?
Input and output are strings over some alphabet
\[ \Sigma = \{a_1, \ldots, a_k\} \]

Encoding \( a_i \) (and hence input and output) in binary is always possible

Decision problems: Output \( \in \{0, 1\} \)

A general problem can be presented as a sequence of decision problems: 1:st bit of the output? 2:nd bit of the output? etc.

Computation: Input \( \xrightarrow{A} \) Output

What does it take to compute \( A \)? How much time?
Computational Problems

- Input and output are strings over some alphabet \( \Sigma = \{a_1, \ldots, a_k\} \)
- Encoding \( a_i \) (and hence input and output) in binary is always possible

Decision problems: Output \( \in \{0, 1\} \)

- A general problem can be presented as a sequence of decision problems: 1:st bit of the output? 2:nd bit of the output? etc.

Computation: Input \( A \rightarrow \) Output

- What does it take to compute \( A \)? How much time? How much space?

Mika Hirvensalo  Basics on Complexity Theory  7 of 39
What is computation?
What is computation?

How to measure the complexity of computation?
What is computation?

How to measure the complexity of computation?

Time / Space?
Computational Complexity – Preliminaries

What is computation?

How to measure the complexity of computation?

Time / Space? Physical time (in seconds) not useful
What is computation?

Gottfried Wilhelm Leibniz (1646–1716)

Scientia Universalis: Characteristica universalis
Calculus ratiocinator

"I think that some selected men could finish the matter in five years"
What is computation?

Gottfried Wilhelm Leibniz (1646–1716)
What is computation?

Gottfried Wilhelm Leibniz (1646–1716)

Scientia Universalis:
What is computation?

Gottfried Wilhelm Leibniz (1646–1716)

Scientia Universalis:

- Characteristica universalis
What is computation?

Gottfried Wilhelm Leibniz (1646–1716)

**Scientia Universalis:**
- Characteristica universalis
- Calculus ratiocinator
What is computation?

Gottfried Wilhelm Leibniz (1646–1716)

Scientia Universalis:
- Characteristica universalis
- Calculus ratiocinator

"I think that some selected men could finish the matter in five years"
What is computation?

Kurt Gödel (1906–1978)
What is computation?

Kurt Gödel (1906–1978)

Incompleteness theorems
What is computation?

Incompleteness theorems
⇔ Algorithmic undecidability

Kurt Gödel (1906–1978)
Christos H. Papadimitriou: Computational Complexity

(Addison-Wesley 1994)
S. Barry Cooper & Jan van Leeuwen: Alan Turing: His Work and Impact

(Elsevier 2013)
Alan Turing (1912–1954)

Theoretical model of computer, Turing Machine (1937)

Tape →

State set → (program)

$p, q, r, \ldots$
Turing Machine

Tape →

Input

Read-write head

State set →
(program)

State $p$:
- Reading $a$, write $b$ ($b$ depends on $p$ and $a$)
- More read-write head (direction depends on $p$ and $a$)
- Move to state $q$ ($q$ depends on $p$ and $a$)
Turing Machine

Tape →

I N P U T

← Read-write head

State set →

(p, q, r, ...)

State set (program)

State p:

- Reading a, write b (b depends on p and a)
- More read-write head (direction depends on p and a)
- Move to state q (q depends on p and a)

Transition function \( \delta(p, a) = (q, b, D) \) (computational step)
INPUT $\xrightarrow{\tau}$ OUTPUT

Turing machine has a starting state $q_0$ and final state(s) $q_f$.

In the beginning, INPUT is written on the tape, read-write head set to read the first symbol, and the state is $q_0$.

Computation is carried on by applying the transition function $\delta$ again and again until a final state is reached.

When $q_f$ is reached, the computation stops and the tape content is interpreted as OUTPUT.

On decision problems, it is enough to have two ending states $q_y$ (yes) and $q_n$ (no), and tape content can be ignored.

Notation: $T(INPUT) = OUTPUT$.
Turing Machine

INPUT $\xrightarrow{T}$ OUTPUT

- Turing machine has a starting state $q_0$ and final state(s) $q_f$
Turing machine has a starting state $q_0$ and final state(s) $q_f$.

In the beginning, INPUT is written on the tape, read-write head set to read the first symbol, and the state is $q_0$. Computation is carried on by applying the transition function $\delta$ again and again until a final state is reached. When $q_f$ is reached, the computation stops and the tape content is interpreted as OUTPUT.

On decision problems, it is enough to have two ending states $q_y$ (yes) and $q_n$ (no), and tape content can be ignored.
Turing machine has a starting state \( q_0 \) and final state(s) \( q_f \).

In the beginning, INPUT is written on the tape, read-write head set to read the first symbol, and the state is \( q_0 \).

Computation is carried on by applying the transition function \( \delta \) again and again until a final state is reached.
Turing machine has a starting state $q_0$ and final state(s) $q_f$

In the beginning, INPUT is written on the tape, read-write head set to read the first symbol, and the state is $q_0$

Computation is carried on by applying the transition function $\delta$ again and again until a final state is reached

When $q_f$ is reached, the computation stops and the tape content is interpreted as OUTPUT
Turing machine has a starting state $q_0$ and final state(s) $q_f$

In the beginning, INPUT is written on the tape, read-write head set to read the first symbol, and the state is $q_0$

Computation is carried on by applying the transition function $\delta$ again and again until a final state is reached

When $q_f$ is reached, the computation stops and the tape content is interpreted as OUTPUT

On decision problems, it is enough to have two ending states $q_y$ (yes) and $q_n$ (no), and tape content can be ignored
Turing machine has a starting state $q_0$ and final state(s) $q_f$

In the beginning, INPUT is written on the tape, read-write head set to read the first symbol, and the state is $q_0$

Computation is carried on by applying the transition function $\delta$ again and again until a final state is reached

When $q_f$ is reached, the computation stops and the tape content is interpreted as OUTPUT

On decision problems, it is enough to have two ending states $q_y$ (yes) and $q_n$ (no), and tape content can be ignored

Notation: $T(INPUT) = OUTPUT$
Church-Turing Thesis:

Turing machine is the exact mathematical counterpart of the intuitive notion of algorithm.
Church-Turing Thesis:

Turing machine is the exact mathematical counterpart of the intuitive notion of algorithm

- Not provable
Church-Turing Thesis:

Turing machine is the exact mathematical counterpart of the intuitive notion of algorithm

- Not provable
- Not seriously challenged
Church-Turing Thesis:

Turing machine is the exact mathematical counterpart of the intuitive notion of algorithm

- Not provable
- Not seriously challenged
- All *known* algorithms can be converted into Turing machine formalism
Turing machines

Computational time exact (number of steps)
Space needed for computation exact (number of cells)
Useful constructions
Concatenation
Parallel computation
Subroutines
Encoding

TMs are too elementary for practical algorithm design
Very useful theoretically
Computational time exact (number of steps)
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions

- Concatenation
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions

- Concatenation
- Parallel computation
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions

- Concatenation
- Parallel computation
- Subroutines
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions

- Concatenation
- Parallel computation
- Subroutines
- Encoding
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions

- Concatenation
- Parallel computation
- Subroutines
- Encoding

- TMs are too elementary for practical algorithm design
Turing machines

- Computational time exact (number of steps)
- Space needed for computation exact (number of cells)

Useful constructions

- Concatenation
- Parallel computation
- Subroutines
- Encoding

- TMs are too elementary for practical algorithm design
- Very useful theoretically
A description of a Turing machine is a finite table of transitions \( \Rightarrow \) finite string. The state set describes the program. Part of the input can be interpreted as program.

Universal Turing Machine \( U \)

On input \((T, w)\), \( U \) simulates the computation of \( T \) on input \( w \).

 Quite small universal Turing machines exists: 

- 15 states, 2-element alphabet
- 9 states, 3-element alphabet
- 2 states, 18-element alphabet

Mika Hirvensalo

Basics on Complexity Theory 18 of 39
A description of a Turing machine is a finite table of transitions $\Rightarrow$ finite string.
A description of a Turing machine is a finite table of transitions $\Rightarrow$ finite string.

The state set describes the program.
- A description of a Turing machine is a finite table of transitions $\Rightarrow$ finite string.
- The state set describes the program
- Part of the input can be *interpreted* as program
A description of a Turing machine is a finite table of transitions $\Rightarrow$ finite string.

The state set describes the program

Part of the input can be interpreted as program

Universal Turing Machine $\mathcal{U}$

On input $(\mathcal{T}, w)$, $\mathcal{U}$ simulates the computation of $\mathcal{T}$ on input $w$. 
A description of a Turing machine is a finite table of transitions $\Rightarrow$ finite string.

The state set describes the program

Part of the input can be *interpreted* as program

**Universal Turing Machine $U$**

On input $(T, w)$, $U$ simulates the computation of $T$ on input $w$.

The input $w$ of $T$ must be encoded into the alphabet of $U$
A description of a Turing machine is a finite table of transitions $\Rightarrow$ finite string.

The state set describes the program

Part of the input can be interpreted as program

**Universal Turing Machine $U$**

On input $(\mathcal{T}, w)$, $U$ simulates the computation of $\mathcal{T}$ on input $w$.

- The input $w$ of $\mathcal{T}$ must be encoded into the alphabet of $U$

- Quite small universal Turing machines exists:
  - 15 states, 2-element alphabet
  - 9 states, 3-element alphabet
  - 2 states, 18-element alphabet
Gluing alphabet symbols together results in a bigger alphabet and more $\delta$-rules, but smaller computations time (constant improvement)
Gluing alphabet symbols together results in a bigger alphabet and more \( \delta \)-rules, but smaller computations time (constant improvement)

“Increasing hardware makes computation faster”
(constant improvement)
Gluing alphabet symbols together results in a bigger alphabet and more $\delta$-rules, but smaller computations time (constant improvement)

“Increasing hardware makes computation faster” (constant improvement)

Time / Space -resources should be measured only up to multiplicative constant.
Ordo-notation

Measuring computational resources

Computational resources (time/space) should be measured ignoring the multiplicative constants.

Example:

Algorithms $A_1$, $A_2$, $A_3$, and $A_4$ consume respectively $2^n_6$, $600n_3$, $40000n$, and $2n^{10}$ steps to accomplish their tasks. Ignoring the multiplicative constants, their running times are around $n_6$, $n_3$, $n$, and $2n$. Hence $A_3$ is the fastest and $A_4$ the slowest.
Measuring computational resources

Computational resources (time/space) should be measured ignoring the multiplicative constants.
Measuring computational resources

Computational resources (time/space) should be measured ignoring the multiplicative constants

Example

Algorithms $A_1$, $A_2$, $A_3$, and $A_4$ consume respectively $2n^6$, $600n^3$, $40000n$, and $\frac{2^n}{10^{10}}$ steps to accomplish their tasks. Ignoring the multiplicative constants, their running times are around $n^6$, $n^3$, $n$, and $2^n$. Hence $A_3$ is the fastest and $A_4$ the slowest.
Definition

\[ f(x) = O(g(x)), \] if there are constants \( K > 0 \) and \( M > 0 \) so that

\[ |f(x)| \leq K |g(x)|, \] whenever \( x \geq M \).

Example

\[ 4x^5 - 2x^3 + 3x + 4 = O(x^5) \]

Example

\[ x^n = O(mx) \] for each \( n \in \mathbb{N} \) and \( m > 1 \).
**Definition**

\[ f(x) = O(g(x)), \text{ if there are constants } K > 0 \text{ and } M > 0 \text{ so that} \]
\[ |f(x)| \leq K |g(x)| , \]

whenever \( x \geq M \).
**Ordo-notation**

**Definition**

\[ f(x) = O(g(x)), \] if there are constants \( K > 0 \) and \( M > 0 \) so that

\[ |f(x)| \leq K |g(x)|, \]

whenever \( x \geq M \).

**Example**

\[ 4x^5 - 2x^3 + 3x + 4 = O(x^5) \]
Ordo-notation

**Definition**

\[ f(x) = O(g(x)), \text{ if there are constants } K > 0 \text{ and } M > 0 \text{ so that} \]
\[ |f(x)| \leq K |g(x)|, \]

whenever \( x \geq M \).

**Example**

\[ 4x^5 - 2x^3 + 3x + 4 = O(x^5) \]

**Example**

\[ x^n = O(m^x) \]

for each \( n \in \mathbb{N} \) and \( m > 1 \).
Algorithmic undecidability

Program STUCK

Finds out if program $P$ with input $x$ gets stuck (runs forever):

$$\text{STUCK}(P, x) = \text{yes/no}.$$ 

Program TEASE

$$\text{TEASE}(\text{input}) = \begin{cases} \text{Stop}, & \text{if STUCK(input, input) = yes} \\ \text{Get stuck}, & \text{if STUCK(input, input) = no} \end{cases}$$

$$\text{TEASE}(\text{TEASE})? \text{Stops if, jos STUCK(TEASE, TEASE) = yes (meaning that TEASE(TEASE) does not stop)}$$

$$\text{Gets stuck, if STUCK(TEASE, TEASE) = no (meaning that TEASE(TEASE) stops)}$$

Contradiction! $\Rightarrow$ no program STUCK exists.
**Algorithmic undecidability**

Program **STUCK**

Finds out if program \( P \) with input \( x \) gets stuck (runs forever):
\[
\text{STUCK}(P, x) = \text{yes/no}.
\]
Program STUCK

Finds out if program P with input x gets stuck (runs forever): STUCK(P,x) = yes/no.

Program TEASE

TEASE(input) = \begin{align*}
\text{Stop,} & \quad \text{if } \text{STUCK(input,input)} = \text{yes} \\
\text{Get stuck,} & \quad \text{if } \text{STUCK(input,input)} = \text{no}
\end{align*}
Algorithmic undecidability

Program STUCK

Finds out if program P with input x gets stuck (runs forever):
\[ \text{STUCK}(P, x) = \text{yes/no}. \]

Program TEASE

\[ \text{TEASE}(\text{input}) = \begin{cases} 
\text{Stop,} & \text{if } \text{STUCK}(\text{input, input}) = \text{yes} \\
\text{Get stuck,} & \text{if } \text{STUCK}(\text{input, input}) = \text{no} 
\end{cases} \]

- \[ \text{TEASE} (\text{TEASE})? \]
Algorithmic undecidability

Program **STUCK**

Finds out if program \( P \) with input \( x \) gets stuck (runs forever):

\[ \text{STUCK}(P, x) = \text{yes/no}. \]

Program **TEASE**

\[
\text{TEASE}(\text{input}) = \begin{cases} 
\text{Stop,} & \text{if STUCK(\text{input}, \text{input})=yes} \\
\text{Get stuck,} & \text{if STUCK(\text{input}, \text{input})=no}
\end{cases}
\]

- **TEASE(TEASE)?**
- Stops if, jos STUCK(TEASE, TEASE) = yes (meaning that TEASE(TEASE) *does not* stop)
### Program STUCK

Finds out if program $P$ with input $x$ gets stuck (runs forever):

$$\text{STUCK}(P, x) = \text{yes}/\text{no}.$$  

### Program TEASE

$$\text{TEASE}(\text{input}) = \begin{cases} 
\text{Stop,} & \text{if STUCK(input, input) = yes} \\
\text{Get stuck,} & \text{if STUCK(input, input) = no}
\end{cases}$$

- TEASE(TEASE)?
- Stops if, jos $\text{STUCK(TEASE, TEASE) = yes}$ (meaning that $\text{TEASE(TEASE)}$ does not stop)
- Gets stuck, if $\text{STUCK(TEASE, TEASE) = no}$ (meaning that $\text{TEASE(TEASE)}$ stops)

Contradiction! $\Rightarrow$ no program $\text{STUCK}$ exists.
Program STUCK

Finds out if program $P$ with input $x$ gets stuck (runs forever):
$\text{STUCK}(P,x) = \text{yes/no}$.

Program TEASE

$\text{TEASE}(\text{input}) = \begin{cases} 
\text{Stop, } & \text{if } \text{STUCK}(\text{input, input}) = \text{yes} \\
\text{Get stuck, } & \text{if } \text{STUCK}(\text{input, input}) = \text{no}
\end{cases}$

- $\text{TEASE}(\text{TEASE})$?
- Stops if, jos $\text{STUCK}(\text{TEASE, TEASE}) = \text{yes}$ (meaning that $\text{TEASE}(\text{TEASE})$ \textit{does not} stop)
- Gets stuck, if $\text{STUCK}(\text{TEASE, TEASE}) = \text{no}$ (meaning that $\text{TEASE}(\text{TEASE})$ stops) \textbf{Contradiction!} $\Rightarrow$ no program STUCK exists
Algorithmic undecidability

Halting problem is algorithmically undecidable

There is no such program as STUCK

Strings can be encoded in numbers

INPUT \rightarrow N = 73,788,808,584.

Turing Machine operations are interpreted as calculations:

N_1 = 73,788,085,84 \rightarrow N_2 = 72,798,085,84

Implies algorithmic undecidability for many mathematical problems

Hilbert’s 10th problem

Given an polynomial \( p(x_1, x_2, \ldots, x_n) \) over integers, does it have any integer zero? (Undecidable: Yury Matiyasevich 1970)

Matrix problems: Given set \{ M_1, \ldots, M_k \} of integer matrices, can we get the zero matrix multiplicatively?

Mika Hirvensalo

Basics on Complexity Theory 23 of 39
<table>
<thead>
<tr>
<th>Algorithmic undecidability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Halting problem is algorithmically undecidable</strong></td>
</tr>
<tr>
<td>There is no such program as STUCK</td>
</tr>
</tbody>
</table>
Algorithmic undecidability

Halting problem is algorithmically undecidable

There is no such program as STUCK

- Strings can be encoded in numbers $INPUT \ldots \rightarrow N = 7378808584$. 

Turing Machine operations are interpreted as calculations:

$N_1 = 7378808584 \rightarrow N_2 = 7279808584$

Implies algorithmic undecidability for many mathematical problems

Hilbert's 10th problem

Given a polynomial $p(x_1, x_2, \ldots, x_n)$ over integers, does it have any integer zero? (Undecidable: Yury Matiyasevich 1970)

Matrix problems: Given set $\{M_1, \ldots, M_k\}$ of integer matrices, can we get the zero matrix multiplicatively?
Halting problem is algorithmically undecidable
There is no such program as STUCK

- Strings can be encoded in numbers \( INPUT \ldots \rightarrow N = 7378808584 \).
- Turing Machine operations are interpreted as calculations:
  \( N_1 = 73\underline{78}808584 \rightarrow N_2 = 72\underline{79}808584 \)
Algorithmic undecidability

Halting problem is algorithmically undecidable
There is no such program as STUCK

- Strings can be encoded in numbers $INPUT \ldots \rightarrow N = 7378808584$.
- Turing Machine operations are interpreted as calculations: $N_1 = 737808584 \rightarrow N_2 = 7279808584$
- Implies algorithmic undecidability for many mathematical problems
Algorithmic undecidability

Halting problem is algorithmically undecidable

There is no such program as STUCK

- Strings can be encoded in numbers $INPUT \ldots \to N = 7378808584$.
- Turing Machine operations are interpreted as calculations: $N_1 = 73\underline{78}808584 \to N_2 = 72\underline{79}808584$
- Implies algorithmic undecidability for many mathematical problems

Hilbert’s 10th problem

Given an polynomial $p(x_1, x_2, \ldots, x_n)$ over integers, does it have any integer zero? (Undecidable: Yury Matiyasevich 1970)
Algorithmic undecidability

Halting problem is algorithmically undecidable

There is no such program as STUCK

- Strings can be encoded in numbers $INPUT \ldots \rightarrow N = 7378808584$.
- Turing Machine operations are interpreted as calculations: $N_1 = 73\underline{78}808584 \rightarrow N_2 = 72\underline{79}808584$
- Implies algorithmic undecidability for many mathematical problems

Hilbert’s 10th problem

Given an polynomial $p(x_1, x_2, \ldots, x_n)$ over integers, does it have any integer zero? (Undecidable: Yury Matiyasevich 1970)

- Matrix problems: Given set $\{M_1, \ldots, M_k\}$ of integer matrices, can we get the zero matrix multiplicatively?
Notations

A finite set \( \Sigma = \{a_1, \ldots, a_n\} \) is called an alphabet. The set of all strings (words) over alphabet \( \Sigma \) is denoted by \( \Sigma^* \). A formal language over alphabet \( \Sigma \) is a subset of \( \Sigma^* \).

Example
Any Turing machine \( T \) defines a formal language \( L(T) = \{w \in \Sigma^* | \text{The computation of } T \text{ on input } w \text{ stops}\} \).

Definition
A formal language \( L \subseteq \Sigma^* \) is recursively enumerable if it can be accepted by a Turing machine, meaning that \( w \in L \iff T \text{ halts on input } w \). The set of recursively enumerable languages is denoted by \( \text{RE} \).
**Definition**

A finite set $\Sigma = \{a_1, \ldots, a_n\}$ is called an *alphabet*. The set of all strings (words) over alphabet $\Sigma$ is denoted by $\Sigma^*$. A *formal language* over alphabet $\Sigma$ is a subset of $\Sigma^*$. 

**Example**

Any Turing machine $T$ defines a formal language $L(T) = \{w \in \Sigma^* | \text{The computation of } T \text{ on input } w \text{ stops}\}$.
A finite set $\Sigma = \{a_1, \ldots, a_n\}$ is called an *alphabet*. The set of all strings (words) over alphabet $\Sigma$ is denoted by $\Sigma^*$. A *formal language* over alphabet $\Sigma$ is a subset of $\Sigma^*$.

*Example*

Any Turing machine $T$ defines a formal language $L(T) = \{w \in \Sigma^* \mid \text{The computation of } T \text{ on input } w \text{ stops}\}$.
Notations

**Definition**

A finite set $\Sigma = \{a_1, \ldots, a_n\}$ is called an *alphabet*. The set of all strings (words) over alphabet $\Sigma$ is denoted by $\Sigma^*$. A *formal language* over alphabet $\Sigma$ is a subset of $\Sigma^*$.

**Example**

Any Turing machine $T$ defines a formal language $L(T) = \{w \in \Sigma^* \mid \text{The computation of } T \text{ on input } w \text{ stops}\}$

**Definition**

A formal language $L \subseteq \Sigma^*$ is *recursively enumerable* if it can be accepted by a Turing machine, meaning that $w \in L \iff T$ halts on input $w$. The set of recursively enumerable languages is denoted by $\text{RE}$. 
A formal language $L \subseteq \Sigma^*$ is recursive if its membership problem is solvable by a Turing machine that halts on every input: $w \in L \iff$ the computation of $T$ on $w$ halts on an accepting state. The set of recursive languages is denoted by $R$.

Example: The halting language $H = \{ (P, x) \mid$ Turing machine $P$ halts on input $x \}$ is recursively enumerable but not recursive (There is no program STUCK).
Notations

Definition

A formal language $L \subseteq \Sigma^*$ is *recursive* if its membership problem is solvable by a Turing machine that halts on every input: $w \in L \iff$ the computation of $T$ on $w$ halts on an accepting state. The set of recursive languages is denoted by $R$. 

Example

The halting language $H = \{ (P, x) \mid$ Turing machine $P$ halts on input $x \}$ is recursively enumerable but not recursive (There is no program STUCK).
A formal language $L \subseteq \Sigma^*$ is **recursive** if its membership problem is solvable by a Turing machine that halts on every input: $w \in L \iff$ the computation of $T$ on $w$ halts on an accepting state. The set of recursive languages is denoted by $\mathbf{R}$.

**Example**

The halting language

$$H = \{(P, x) \mid \text{Turing machine } P \text{ halts on input } x\}$$

is recursively enumerable but not recursive (There is no program STUCK).
Polynomial Time Computation

Polynomial Time Computation

On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.

Can run in parallel with any other TM

Polynomial Time Turing Machines

On any input $w$, the machine stops in at most $n^k$ steps.

The decision (yes/no) depends on the halting state:

- If machine stops because the “time” is up, the answer is “no” (reject)
- If machine finishes its computation in time bound, the answer can be yes (accept) or (no) depending on the input

$k$ is a parameter that can be chosen freely
Polynomial Time Computation

- Straightforward to design: On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.
Polynomial Time Computation

- Straightforward to design: On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.
- Can run in parallel with any other TM
• Straightforward to design: On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.
• Can run in parallel with any other TM
Polynomial Time Computation

- Straightforward to design: On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.
- Can run in parallel with any other TM

Polynomial Time Turing Machines

On any input $w$, the machine stops in at most $n^k$ steps.

- The decision (yes/no) depends on the halting state:
Polynomial Time Computation

- Straightforward to design: On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.
- Can run in parallel with any other TM

Polynomial Time Turing Machines

On any input $w$, the machine stops in at most $n^k$ steps.

- The decision (yes/no) depends on the halting state:
  - If machine stops because the “time” is up, the answer is “no” (reject)
  - If machine finishes its computation in time bound, the answer can be yes (accept) or (no) depending on the input
Polynomial Time Computation

- Straightforward to design: On input $w = a_1 \ldots a_n$ (denote $n = |w|$), machine counts $n^k$ steps and then stops.
- Can run in parallel with any other TM

Polynomial Time Turing Machines

On any input $w$, the machine stops in at most $n^k$ steps.

- The decision (yes/no) depends on the halting state:
  - If machine stops because the “time” is up, the answer is “no” (reject)
  - If machine finishes its computation in time bound, the answer can be yes (accept) or (no) depending on the input
- $k$ is a parameter that can be chosen freely
Polynomial Time Computation

Example
Product is solvable in $O(n^2)$ steps (normal multiplication) $\Rightarrow n^3$ time bound is sufficient

Example
Primality is solvable in $O(n^{6+\epsilon})$ steps (highly nontrivial) $\Rightarrow n^7$ time bound is sufficient

Definition
$P$ is the set of formal languages accepted by polynomial-time Turing Machines (Practical computation with no errors)
Example

**PRODUCT** is solvable in $O(n^2)$ steps (normal multiplication) $\Rightarrow n^3$
time bound is sufficient
Example

**PRODUCT** is solvable in $O(n^2)$ steps (normal multiplication) $\Rightarrow n^3$
time bound is sufficient

Example

**PRIMALITY** is solvable in $O(n^{6+\epsilon})$ steps (highly nontrivial) $\Rightarrow n^7$
time bound is sufficient
Polynomial Time Computation

**Example**

PRODUCT is solvable in $O(n^2)$ steps (normal multiplication) $\Rightarrow n^3$ time bound is sufficient

**Example**

PRIMALITY is solvable in $O(n^{6+\epsilon})$ steps (highly nontrivial) $\Rightarrow n^7$ time bound is sufficient

**Definition**

P is the set of formal languages accepted by polynomial-time Turing Machines (Practical computation with no errors)
Turing Machine generalizations

Nondeterministic
Instead of a single transition \( \delta(p, a) = (q, b, D) \), the machine can choose its action from a finite set \( (p, a) \rightarrow (q_i, b_i, D_i) \) (transition relation).

A nondeterministic Turing machine does not define a function but a relation.

Probabilistic
Same as nondeterministic, but all transitions occur with a given probability \( (p, a) \rightarrow_i (q_i, b_i, D_i) \).

A probabilistic Turing machine does not define a function but a probability distribution over outputs, depending on the inputs.
Nondeterministic

Instead of single transition $\delta(p, a) = (q, b, D)$ the machine can choose its action from a finite set $(p, a) \rightarrow (q_i, b_i, D_i)$ (transition relation).

A nondeterministic Turing machine does not define a function but a relation $INPUT \rightarrow OUTPUT$.

Probabilistic

Same as nondeterministic, but all transitions occur with a given probability $(p, a) \rightarrow (q_i, b_i, D_i)$.

A probabilistic Turing machine does not define a function but a probability distribution over OUTPUTs, depending on the INPUT.
Turing Machine generalizations

Nondeterministic

Instead of single transition \( \delta(p, a) = (q, b, D) \) the machine can choose its action from a finite set \((p, a) \rightarrow (q_i, b_i, D_i)\) (transition relation).

- A nondeterministic Turing machine does not define a function but a relation \( INPUT \xrightarrow{T} OUTPUT \)
Turing Machine generalizations

Nondeterministic

Instead of single transition \( \delta(p, a) = (q, b, D) \) the machine can choose its action from a finite set \((p, a) \xrightarrow{\tau} (q_i, b_i, D_i)\) (transition relation).

- A nondeterministic Turing machine does not define a function but a relation \( INPUT \xrightarrow{T} OUTPUT \)

Probabilistic

Same as nondeterministic, but all transitions occur with a given probability \((p, a) \xrightarrow{p_i} (q_i, b_i, D_i)\)
Turing Machine generalizations

Nondeterministic
Instead of single transition $\delta(p, a) = (q, b, D)$ the machine can choose its action from a finite set $(p, a) \rightarrow (q_i, b_i, D_i)$ (transition relation).

- A nondeterministic Turing machine does not define a function but a relation $INPUT \xrightarrow{T} OUTPUT$

Probabilistic
Same as nondeterministic, but all transitions occur with a given probability $(p, a) \xrightarrow{p_i} (q_i, b_i, D_i)$

- A probabilistic Turing machine does not define a function but a probability distribution over $OUTPUT$s, depending on the $INPUT$. 
Turing Machine generalizations

Definition (Nondeterministic acceptance)
An input word \( w \) is accepted by a nondeterministic Turing machine \( N \) if there is at least one accepting computation. Otherwise, \( w \) is rejected by \( N \).
Definition (Nondeterministic acceptance)

An input word $w \in L$ is accepted by a nondeterministic Turing machine $\mathcal{N}$ if there is at least one accepting computation. Otherwise $w$ is rejected by $\mathcal{N}$. 
Definition (Nondeterministic acceptance)

An input word $w \in L$ is accepted by a nondeterministic Turing machine $N$ if there is at least one accepting computation. Otherwise, $w$ is rejected by $N$. 

Tree of computations
Definition (Monte Carlo Model)

An input word $w$ is accepted by a probabilistic Turing Machine $\mathcal{P}$, if the acceptance probability for $w$ is at least $\frac{2}{3}$. A word is rejected if its acceptance probability is at most $\frac{1}{3}$.
Turing Machine generalizations

Definition (Monte Carlo Model)
An input word $w$ is accepted by a probabilistic Turing Machine $\mathcal{P}$, if the acceptance probability for $w$ is at least $\frac{2}{3}$. A word is rejected, if its acceptance probability is at most $\frac{1}{3}$.

Definition (Las Vegas Model)
An input word $w$ is accepted by a probabilistic Turing machine, if its acceptance probability is 1. A word is rejected, if its acceptance probability is at most $\frac{1}{2}$.
Example (Nondeterministic Factorization Algorithm)

1. Input \( m = n_1 \ldots n_n \) in binary
2. For \( i = 1 \) to \( \frac{n}{2} \) do: guess the \( i \):th digit of a factor \( f = f_1 \ldots f_{\frac{n}{2}} \)
3. Perform division \( m/f \) deterministically
4. If \( f \) divides \( m \), the run is successful, otherwise not
5. In a successful case, the algorithm can be re-run on \( f \)
Example (Nondeterministic Factorization Algorithm)

1. Input $m = n_1 \ldots m_n$ in binary
2. For $i = 1$ to $\frac{n}{2}$ do: guess the $i$:th digit of a factor $f = f_1 \ldots f_{\frac{n}{2}}$
3. Perform division $m/f$ deterministically
4. If $f$ divides $m$, the run is successful, otherwise not
5. In a successful case, the algorithm can be re-run on $f$

For any composite number $m$, there exists a factor $f$, and hence there is at least one successful run of the algorithm.
Example (Nondeterministic Factorization Algorithm)

1. Input $m = n_1 \ldots m_n$ in binary
2. For $i = 1$ to $\frac{n}{2}$ do: guess the $i$:th digit of a factor $f = f_1 \ldots f_{\frac{n}{2}}$
3. Perform division $m/f$ deterministically
4. If $f$ divides $m$, the run is successful, otherwise not
5. In a successful case, the algorithm can be re-run on $f$

- For any composite number $m$, there exists a factor $f$, and hence there is at least one successful run of the algorithm.
- In general, running a nondeterministic algorithm corresponds to guessing and verification
Important complexity classes

**Definition**

- **P** is the class of languages that can be accepted by polynomial time deterministic Turing machines.
- **NP** is the class of languages that can be accepted by nondeterministic polynomial time Turing machines.
- **BPP** is the class of languages that can be accepted by probabilistic polynomial time Turing machines with Monte Carlo acceptance model (practical computability).

Clearly, $P \subseteq NP$ and $P \subseteq BPP$. 
Important complexity classes

**Definition**

- **P** is the class of languages that can be accepted by polynomial time deterministic Turing machines
- **NP** is the class of languages that can be accepted by nondeterministic polynomial time Turing machines
- **BPP** is the class of languages that can be accepted by probabilistic polynomial time Turing machines with Monte Carlo acceptance model (practical computability)

Clearly \( P \subseteq NP \) and \( P \subseteq BPP \).
For any nondeterministic Turing machine, the number of computational choices can be assumed two. Machine "tosses coin" on each step.

Guiding string:
\[ s = s_1 s_2 \ldots s_n \]

It tells which nondeterministic option to take (outcomes of the "coin tosses").

Computing with a guiding string is deterministic: a guiding string determines a path in the tree of computations.

- Computing with a guiding string:
- Computing deterministically

\[ w \in L \iff \text{if there is a guiding string leading to an accepting final state.} \]

Hence:

Nondeterministic computing = guessing the guiding string + computing deterministically.
For any nondeterministic Turing machine, the number of computational choices can be assumed two.

Machine “tosses coin” on each step.
For any nondeterministic Turing machine, the number of computational choices can be assumed two.

Machine “tosses coin” on each step.

**Guiding string:** \( s = s_1s_2 \ldots s_n \); \( s_i \) tells which nondeterministic option to take (outcomes of the “coin tosses”).
For any nondeterministic Turing machine, the number of computational choices can be assumed two.

Machine “tosses coin” on each step.

**Guiding string:** \( s = s_1 s_2 \ldots s_n \); \( s_i \) tells which nondeterministic option to take (outcomes of the “coin tosses”)

Computing with a guiding string is deterministic: a guiding string determines a path in the tree of computations.

\( w \in L \iff \) if there is a guiding string \( s \) leading to an accepting final state. Hence:
For any nondeterministic Turing machine, the number of computational choices can be assumed two

- Machine “tosses coin” on each step

**Guiding string:** $s = s_1 s_2 \ldots s_n$; $s_i$ tells which nondeterministic option to take (outcomes of the “coin tosses”)

- Computing with a guiding string is deterministic: a guiding string determines a path in the tree of computations
- $w \in L \iff$ if there is a guiding string $s$ leading to an accepting final state. Hence:

Nondeterministic computing
= guessing the guiding string + computing deterministically
Definition (3-Sat)

Given a Boolean expression \( \varphi(x_1, \ldots, x_n) \) in Conjunctive Normal Form, three literals in each clause, determine if there is a satisfying assignment.

Example

\[ \varphi(x_1, x_2, x_3, x_4, x_5) = (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_2 \lor \neg x_3 \lor \neg x_4) \land (\neg x_3 \lor x_4 \lor x_5) \]

has a satisfying assignment \((x_1, x_2, x_3, x_4, x_5) = (0, 1, 0, 1, 1)\).

Nondeterministic algorithm for 3-Sat

1. For \( i = 1 \) to \( n \) do:
   - Guess value \( v_i \) for \( x_i \)
   - Check whether \( \varphi(v_1, \ldots, v_n) \) has truth value 1.
Definition (3-SAT)

Given a Boolean expression $\phi(x_1, \ldots, x_n)$ in Conjunctive Normal Form, three literals in each clause, determine if there is a satisfying assignment.
Definition (3-SAT)

Given a Boolean expression $\phi(x_1, \ldots, x_n)$ in Conjunctive Normal Form, three literals in each clause, determine if there is a satisfying assignment.

Example

$\phi(x_1, x_2, x_3, x_4, x_5) = (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee x_4 \vee x_5)$

has a satisfying assignment $(x_1, x_2, x_3, x_4, x_5) = (0, 1, 0, 1, 1)$. 
Definition (3-SAT)

Given a Boolean expression $\phi(x_1, \ldots, x_n)$ in Conjunctive Normal Form, three literals in each clause, determine if there is a satisfying assignment.

Example

$\phi(x_1, x_2, x_3, x_4, x_5) = (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_2 \lor \neg x_3 \lor \neg x_4) \land (\neg x_3 \lor x_4 \lor x_5)$

has a satisfying assignment $(x_1, x_2, x_3, x_4, x_5) = (0, 1, 0, 1, 1)$.

Nondeterministic algorithm for 3-SAT

1. For $i = 1$ to $n$ do: Guess value $v_i$ for $x_i$
2. Check whether $\phi(v_1, \ldots, v_n)$ has truth value 1.
Deterministic algorithm for 3-SAT

1. Go through all values $v_1 \ldots v_n \in \{0, 1\}^n$;
2. For each valuation, check whether $\phi(v_1, \ldots, v_n)$ has truth value 1
Deterministic algorithm for 3-SAT

1. Go through all values \( v_1 \ldots v_n \in \{0, 1\}^n \);
2. For each valuation, check whether \( \phi(v_1, \ldots, v_n) \) has truth value 1

The deterministic version requires \( 2^n \) rounds, whereas the nondeterministic version has only one round.
Deterministic algorithm for $3$-SAT

1. Go through all values $v_1 \ldots v_n \in \{0, 1\}^n$;
2. For each valuation, check whether $\phi(v_1, \ldots, v_n)$ has truth value $1$

The deterministic version requires $2^n$ rounds, whereas the nondeterministic version has only one round.

Can you do better (deterministically)?
Deterministic algorithm for 3-SAT

1. Go through all values $v_1 \ldots v_n \in \{0, 1\}^n$;
2. For each valuation, check whether $\phi(v_1, \ldots, v_n)$ has truth value 1

The deterministic version requires $2^n$ rounds, whereas the nondeterministic version has only one round.

Can you do better (deterministically)?

Uwe Schöning:
$O((\frac{4}{3})^n)$ algorithm
Can you solve 3-Sat deterministically in $n^k$ time?

No-one knows 3-Sat is NP-complete. All other NP problems reduce to it.

A polynomial algorithm for 3-Sat will give a P algorithm for all NP problems.

Guessing would be as "hard" as discovering the solution. A polynomial algorithm for 3-Sat would imply that "guessing" is as difficult as "verifying"; NP would be equal to P.
Can you solve 3-SAT deterministically in $n^k$ time?
Can you solve 3-SAT deterministically in $n^k$ time?

No-one knows
Can you solve 3-SAT deterministically in $n^k$ time?  
No-one knows

**3-SAT is NP-complete**

- All other NP problems reduce to it
- P algorithm to 3-SAT will give a P algorithm for all NP problems
- Guessing would be as “hard” as discovering the solution
Can you solve 3-SAT deterministically in $n^k$ time?

No-one knows

**3-SAT is NP-complete**
- All other NP problems reduce to it
- P algorithm to 3-SAT will give a P algorithm for all NP problems
- Guessing would be as “hard” as discovering the solution

A polynomial algorithm for 3-SAT would imply that “guessing” is as difficult as “verifying”; NP would be equal to P.
An **NP** algorithm for finding the proof of Riemann Hypothesis

1. Guess a 10 000 pages long proof
2. Verify that it is correct
### An NP algorithm for finding the proof of Riemann Hypothesis

1. Guess a 10,000 pages long proof
2. Verify that it is correct

If \( P = NP \) there is also a deterministic polynomial-time algorithm for the same task.
An \textbf{NP} algorithm for finding the proof of Riemann Hypothesis

1. Guess a 10 000 pages long proof
2. Verify that it is correct

If $P = NP$ there is also a deterministic polynomial-time algorithm for the same task

Probably $P \neq NP$, but how to prove it?
Clay Mathematics Institute:
The solver of the $P$ vs. $NP$ problem will get

$1,000,000$
Further complexity classes

Complexity Zoo at
https://complexityzoo.uwaterloo.ca/Complexity_Zoo
Further complexity classes

Complexity Zoo at
https://complexityzoo.uwaterloo.ca/Complexity_Zoo

496 complexity classes by now