The Grand Unified Theory of Computation

Vahan Mkrtchyan

1Lane Department of Computer Science and Electrical Engineering
West Virginia University

April 20, 2015
Outline

1. Babbage’s Vision and Hilbert’s Dream

Computational Complexity
Outline

1. Babbage’s Vision and Hilbert’s Dream
2. Universality and Undecidability
Outline

1. Babbage’s Vision and Hilbert’s Dream
2. Universality and Undecidability
3. Building Blocks: Recursive Functions
Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input \( x \), and we are asked to check if a certain property \( P \) is true.

TSP

Given a complete graph \( K_n \), together with an edge-weight function \( c : E(K_n) \rightarrow \mathbb{N} \) and a bound \( B \), the goal is to check whether there is a Hamiltonian cycle of weight at most \( B \).
Problems

Formulation of an Algorithmic Problem
In a typical algorithmic problem (decision problem), we are given a certain input $x$, ...
Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input $x$, and we are asked to check if a certain property $P$ is true.
Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input $x$, and we are asked to check if a certain property $P$ is true.

TSP
Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input \( x \), and we are asked to check if a certain property \( P \) is true.

TSP

Given a complete graph \( K_n \), together with an edge-weight function \( c : E(K_n) \rightarrow \mathbb{N} \) and a bound \( B \),
Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input $x$, and we are asked to check if a certain property $P$ is true.

TSP

Given a complete graph $K_n$, together with an edge-weight function $c : E(K_n) \to \mathbb{N}$ and a bound $B$, the goal is to check whether there is a Hamiltonian cycle of weight at most $B$. 
In this course, we have dealt with problems that are solvable with some algorithm. We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (NP-completeness, NP-hardness).

What is the algorithm that solves TSP?

How many Hamiltonian cycles $K_n$ has?

What is the description of the algorithm that solves the general decision problem?
An Algorithm solving the Problem
In this course, we have dealt with problems that are solvable with some algorithm.
An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm. We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist.
An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (NP-completeness, NP-hardness).
In this course, we have dealt with problems that are solvable with some algorithm. We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (NP-completeness, NP-hardness).

What is the algorithm that solves TSP?
In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (NP-completeness, NP-hardness).

What is the algorithm that solves TSP? How many Hamiltonian cycles $K_n$ has?
An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (**NP-completeness**, **NP-hardness**).

What is the algorithm that solves TSP? How many Hamiltonian cycles $K_n$ has?

What is the description of the algorithm that solves the general decision problem?
Linear Programming
Linear Programming

Does the same trick imply that Linear Programming is solvable?
Linear Programming

Does the same trick imply that Linear Programming is solvable?

Figure: The feasible region
Babbage's Vision and Hilbert's Dream
Universality and Undecidability
Building Blocks: Recursive Functions

Question
Are there algorithmic problems that are unsolvable?

Definition
An algorithmic problem is decidable, computable, or solvable, if there is an algorithm that solves it in some finite amount of time.

Remark
We place no bounds whatsoever on how long the algorithm takes, we just know that it will halt eventually.
Question

Are there algorithmic problems that are unsolvable?

Definition

An algorithmic problem is decidable or computable or solvable, if there is an algorithm that solves it in some finite amount of time.

Remark

We place no bounds whatsoever on how long the algorithm takes, we just know that it will halt eventually.
Question

Are there algorithmic problems that are unsolvable?
Question

Are there algorithmic problems that are unsolvable?

Definition

An algorithmic problem is *decidable* or *computable* or *solvable*, if there is an algorithm that solves it in some finite amount of time.
Question

Are there algorithmic problems that are unsolvable?

Definition

An algorithmic problem is *decidable* or *computable* or *solvable*, if there is an algorithm that solves it in some finite amount of time.

Remark

*We place no bounds whatsoever on how long the algorithm takes,*
Question
Are there algorithmic problems that are unsolvable?

Definition
An algorithmic problem is *decidable* or *computable* or *solvable*, if there is an algorithm that solves it in some finite amount of time.

Remark
*We place no bounds whatsoever on how long the algorithm takes, we just know that it will halt eventually.*
Babbage's Vision and Hilbert's Dream
Universality and Undecidability
Building Blocks: Recursive Functions

Babbage was probably the first, who attempted to construct a mechanical computer. A mechanical device that could calculate the value of a polynomial at any point. Since he was aware of Taylor series, he was expecting to compute the value of any function approximately.
Babbage was probably the first, who attempted to construct a mechanical computer. A mechanical device that could calculate the value of a polynomial at any point. Since he was aware of Taylor series, he was expecting to compute the value of any function approximately.
Babbage's ideas

Babbage was probably the first, who attempted to construct a mechanical computer.
Babbage’s ideas

Babbage was probably the first, who attempted to construct a mechanical computer. A mechanical device that could calculate the value of a polynomial at any point.
Babbage’s ideas

Babbage was probably the first, who attempted to construct a mechanical computer. A mechanical device that could calculate the value of a polynomial at any point. Since he was aware of Taylor series, he was expecting to compute the value of any function approximately.
Hilbert's ideas

Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia. The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century. In 1900, David Hilbert delivered an address to the International Congress of Mathematicians, and asked for the solution of the following problem:

**Problem**

Specify a procedure which, in a finite number of operations, enables one to determine whether a given Diophantine equation (a polynomial equation with integer coefficients) with an arbitrary number of variables has an integer solution.
Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia. The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.

In 1900, David Hilbert delivered an address to the International Congress of Mathematicians, and asked for the solution of the following problem:

**Problem**

Specify a procedure which, in a finite number of operations, enables one to determine whether a given Diophantine equation (a polynomial equation with integer coefficients) with an arbitrary number of variables has an integer solution.
Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.
Hilbert’s ideas

Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia. The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.
Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.

The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.

In 1900, David Hilbert delivered an address to the International Congress of Mathematicians, and asked for the solution of the following problem:
Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.

The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.

In 1900, David Hilbert delivered an address to the International Congress of Mathematicians, and asked for the solution of the following problem:

Specify a procedure which, in a finite number of operations, enables one to determine whether a given Diophantine equation (a polynomial equation with integer coefficients) with an arbitrary number of variables has an integer solution.
An example

A Diophantine equation

\[ 3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0. \]

A consequence

Were there such an algorithm, we could have asked it to solve Fermat's Last Theorem for each fixed value of \( n \geq 3 \):

\[ x^n + y^n = z^n. \]
An example

A Diophantine equation

A consequence
Were there such an algorithm, we could have asked it to solve Fermat's Last Theorem for each fixed value of $n \geq 3$:

$$x^n + y^n = z^n.$$
An example

A Diophantine equation

\[ 3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0. \]
An example

A Diophantine equation

\[ 3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0. \]

A consequence
### An example

#### A Diophantine equation

\[3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot 2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0.\]

#### A consequence

Were there such an algorithm, we could have asked it to solve Fermat's Last Theorem for each fixed value of \(n \geq 3\):
An example

A Diophantine equation

\[ 3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0. \]

A consequence

Were there such an algorithm, we could have asked it to solve Fermat’s Last Theorem for each fixed value of \( n \geq 3 \):

\[ x^n + y^n = z^n. \]
Hilbert’s optimism

Hilbert showed even more optimism about the power of algorithms in 1928, when he challenged his fellow mathematicians with the Entscheidungsproblem (in English, the decision problem):

**Problem**
The Entscheidungsproblem is solved if one knows a procedure that allows one to decide the validity of a given logical expression by a finite number of operations.
Hilbert’s optimism

Entscheidungsproblem
Hilbert showed even more optimism about the power of algorithms in 1928, when he challenged his fellow mathematicians with the Entscheidungsproblem (in English, the decision problem):
Hilbert’s optimism

Entscheidungsproblem

Hilbert showed even more optimism about the power of algorithms in 1928, when he challenged his fellow mathematicians with the Entscheidungsproblem (in English, the decision problem):

The Entscheidungsproblem is solved if one knows a procedure that allows one to decide the validity of a given logical expression by a finite number of operations.
An example

 Were there such an algorithm, we could have asked it to decide whether Fermat's Last Theorem is true:

$$\exists x, y, z \in \mathbb{Z} \setminus \{0\}, x^n + y^n = z^n.$$
An example

A consequence

Were there such an algorithm, we could have asked it to decide whether Fermat's Last Theorem is true:

$$\exists x, y, z \in \mathbb{Z} \setminus \{0\}, x^n + y^n = z^n.$$
An example

A consequence

Were there such an algorithm, we could have asked it to decide whether Fermat’s Last Theorem is true:

\[ \exists x, y, z \in \mathbb{Z} - \{0\}, x^n + y^n = z^n. \]
An example

A consequence

Were there such an algorithm, we could have asked it to decide whether Fermat’s Last Theorem is true:

$$\exists x, y, z \in \mathbb{Z}\backslash\{0\}, x^n + y^n = z^n.$$
Mathematics and its axioms

Mathematicians have been proving theorems without asking these questions since Ancient Greek. Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics. Their goal was to reduce all of mathematics to set theory and logic, creating a formal system powerful enough to prove all the mathematical facts we know. At the turn of the century, several paradoxes shook these foundations, showing that a naive approach to set theory could lead to contradictions.
Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.

Their goal was to reduce all of mathematics to set theory and logic, creating a formal system powerful enough to prove all the mathematical facts we know.

At the turn of the century, several paradoxes shook these foundations, showing that a naive approach to set theory could lead to contradictions.
Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement?
Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?
Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?
Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.
Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.
Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.

Their goal was to reduce all of mathematics to set theory and logic, creating a formal system powerful enough to prove all the mathematical facts we know.
Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.

Their goal was to reduce all of mathematics to set theory and logic, creating a formal system powerful enough to prove all the mathematical facts we know.

At the turn of the century, several paradoxes shook these foundations, showing that a naive approach to set theory could lead to contradictions.
Russell's paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set \( R \) defined as follows:

\[
R = \{ S : S \not\in S \}.
\]

Remark It can be easily seen that \( R \in R \) if and only if \( R \not\in R \).
Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers. So, it seems reasonable to ask which sets are elements of themselves, and which are not. Consider the set $R$ defined as follows:

$$R = \{S : S \not\in S\}.$$ 

Remark It can be easily seen that $R \in R$ if and only if $R \not\in R$. 

Russell's paradox
Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets,
Russell’s paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.
Paradoxes in Mathematics

Russell’s paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.
Paradoxes in Mathematics

Russell’s paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set $R$ defined as follows:

$$R = \{ S : S \not\in S \}.$$
Russell’s paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set $R$ defined as follows:

$$R = \{ S : S \notin S \}.$$
Paradoxes in Mathematics

Russell’s paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set $R$ defined as follows:

$$R = \{S : S \notin S\}.$$

Remark

It can be easily seen that

$$R \in R \text{ if and only if } R \notin R.$$
Paradoxes in Mathematics

In order to specify a natural number \( n \geq 1 \), we need some number of words in English. For each \( n \geq 1 \), there exists a smallest number \( h(n) \), so that any specification of \( n \) requires at least \( h(n) \) words in English.

Consider the smallest number \( k \) which requires at least 1000 words for its specification.
In order to specify a natural number $n \geq 1$, we need some number of words in English. For each $n \geq 1$, there exists a smallest number $h(n)$, so that any specification of $n$ requires at least $h(n)$ words in English.

Consider the smallest number $k$ which requires at least 1000 words for its specification.
A number theoretic paradox

In order to specify a natural number \( n \geq 1 \), we need some number of words in English.
A number theoretic paradox

In order to specify a natural number \( n \geq 1 \), we need some number of words in English.

For each \( n \geq 1 \), there exists a smallest number \( h(n) \), so that any specification of \( n \) requires at least \( h(n) \) words in English.
A number theoretic paradox

In order to specify a natural number \( n \geq 1 \), we need some number of words in English.

For each \( n \geq 1 \), there exists a smallest number \( h(n) \), so that any specification of \( n \) requires at least \( h(n) \) words in English.

Consider the smallest number \( k \) which requires at least 1000 words for its specification.
Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality. They can carry out any program we give to them. In particular, there are programs that run other programs. A computer's operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an interpreter or universal program, a program that takes the source code of another program as input, and runs it step-by-step, keeping track of its variables and which instruction to perform next. Symbolically, we can define this universal program like this:

\[ U(\Pi, x) = \Pi(x) \]
The most basic fact about modern computers is their universality. They can carry out any program we give to them. In particular, there are programs that run other programs. A computer’s operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an interpreter or universal program, a program that takes the source code of another program as input, and runs it step-by-step, keeping track of its variables and which instruction to perform next. Symbolically, we can define this universal program like this:

\[ U(\Pi, x) = \Pi(x) \]
### Universal Programs

The most basic fact about modern computers is their universality.
Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.
Universal Programs

The most basic fact about modern computers is their universality. They can carry out any program we give to them. In particular, there are programs that run other programs.
Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer’s operating system is a program that runs and manages many programs at once.
Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer’s operating system is a program that runs and manages many programs at once.

Interpreters
**Universal Programs**

The most basic fact about modern computers is their universality. They can carry out any program we give to them. In particular, there are programs that run other programs. A computer’s operating system is a program that runs and manages many programs at once.

**Interpreters**

In any programming language, one can write an *interpreter* or *universal program*,...
Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer’s operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an *interpreter* or *universal program*, a program that takes the source code of another program as input,
Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer’s operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an interpreter or universal program, a program that takes the source code of another program as input, and runs it step-by-step, keeping track of its variables and which instruction to perform next.
Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer’s operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an interpreter or universal program, a program that takes the source code of another program as input, and runs it step-by-step, keeping track of its variables and which instruction to perform next.

Symbolically, we can define this universal program like this:

\[ U(\Pi, x) = \Pi(x). \]
Some programs cannot halt. Let \( U(\Pi, x) \) be a universal program. Consider the special case where \( x = \Pi \). Then, we will have the following:

\[
U(\Pi, \Pi) = \Pi(\Pi).
\]

Now suppose, for simplicity, that the programs in question return a Boolean value, true or false. Then we can define a new program \( V(\Pi) = \Pi(\Pi) \).

Now, if we feed \( V \) its own source code, an apparent contradiction arises, since \( V(V) = V(V) \).

The only way to resolve this paradox is if \( V(V) \) is undefined. In other words, when given its own source code as input, \( V \) runs forever, and never returns any output.
Let $U(\Pi, x) be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question return a Boolean value, true or false. Then we can define a new program $V$ which runs $\Pi$ on itself, and negates the result:

$$V(\Pi) = \Pi(\Pi).$$

Now, if we feed $V$ its own source code, an apparent contradiction arises, since $V(V) = V(V)$.

The only way to resolve this paradox is if $V(V)$ is undefined. In other words, when given its own source code as input, $V$ runs forever, and never returns any output.
Some programs cannot halt

Let $U(\Pi, x)$ be a universal program.
Some programs cannot halt

Let \( U(\Pi, x) \) be a universal program. Consider the special case where \( x = \Pi \). Then, we will have the following:

\[
U(\Pi, \Pi) = \Pi(\Pi).
\]
Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false.
Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program $V$ which runs $\Pi$ on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$
Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program $V$ which runs $\Pi$ on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Now, if we feed $V$ its own source code, an apparent contradiction arises, since

$$V(V) = \overline{V(V)}.$$
Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program $V$ which runs $\Pi$ on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Now, if we feed $V$ its own source code, an apparent contradiction arises, since

$$V(V) = \overline{V(V)}.$$

The only way to resolve this paradox is if $V(V)$ is undefined.
Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program $V$ which runs $\Pi$ on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Now, if we feed $V$ its own source code, an apparent contradiction arises, since

$$V(V) = \overline{V(V)}.$$

The only way to resolve this paradox is if $V(V)$ is undefined. In other words, when given its own source code as input, $V$ runs forever, and never returns any output.
Universality and Undecidability

Building Blocks: Recursive Functions

Diagonalization and Halting

Universality implies non-halting programs. This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs. In brief, universality implies non-halting programs. Thus any reasonable definition of computable functions includes partial functions, which are undefined for some values of their input, in addition to total ones, which are always well-defined.
Universality implies non-halting programs
Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program
Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.
Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

In brief, universality implies non-halting programs.
Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

In brief, universality implies non-halting programs.

Thus any reasonable definition of computable functions includes partial functions, which are undefined for some values of their input,
Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

In brief, universality implies non-halting programs.

Thus any reasonable definition of computable functions includes partial functions, which are undefined for some values of their input, in addition to total ones, which are always well-defined.
Diagonalization and Cantor

Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.

Natural numbers and odd numbers are equicardinal.

Definition

If $C$ is a set, let $2^C$ denote its power set, that is, $2^C = \{ D : D \subseteq C \}$. 

The Grand Unified Theory of Computation
Computational Complexity
Definition

Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.
Diagonalization and Cantor

**Definition**

Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

**Equicardinal Sets**
Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Natural numbers and even numbers are equicardinal.
**Diagonalization and Cantor**

**Definition**

Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

**Equicardinal Sets**

Natural numbers and even numbers are equicardinal.

Natural numbers and odd numbers are equicardinal.
Definition

Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.
Natural numbers and odd numbers are equicardinal.

Definition

If $C$ is a set, let $2^C$ denote its power set,
Diagonalization and Cantor

Definition

Two sets $A$ and $B$ are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.

Natural numbers and odd numbers are equicardinal.

Definition

If $C$ is a set, let $2^C$ denote its power set, that is,

$$2^C = \{ D : D \subseteq C \}.$$
The set of natural numbers, $\mathbb{N}$ and its power set $2^\mathbb{N}$ are not equicardinal.

Proof

Assume that we have some enumeration $f$ of $2^\mathbb{N}$.

Consider $C = \{x \in \mathbb{N} : x \not\in f(x)\}$.

Since $C$ is a subset of $\mathbb{N}$, we have that there is a $c \in \mathbb{N}$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \not\in C$.

If $c \not\in C$, then $c \not\in f(c)$, hence $c \in C$.

We have that $c \not\in C$ if and only if $c \in C$, which is a contradiction.
The set of natural numbers, $\mathbb{N}$ and its power set $2^\mathbb{N}$ are not equicardinal.
The set of natural numbers, $\mathbb{N}$ and its power set $2^{\mathbb{N}}$ are not equicardinal.

Proof

Assume that we have some enumeration $f$ of $2^{\mathbb{N}}$.

Consider $C = \{x \in \mathbb{N} : x \notin f(x)\}$.

Since $C$ is a subset of $\mathbb{N}$, we have that there is a $c \in \mathbb{N}$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$.

We have that $c \notin C$ if and only if $c \in C$, which is a contradiction.
Diagonalization and Cantor

**Theorem**

The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.

**Proof**

Assume that we have some enumeration $f$ of $2^N$. 

Consider $C = \{x \in N : x \notin f(x)\}$. Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$. If $c \in C$, then $c \in f(c)$, hence $c \notin C$. If $c \notin C$, then $c \in f(c)$, hence $c \in C$. We have that $c \notin C$ if and only if $c \in C$, which is a contradiction.
The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.

Proof
Assume that we have some enumeration $f$ of $2^N$.
Consider $C = \{x \in N : x \notin f(x)\}$. 
The set of natural numbers, \( N \) and its power set \( 2^N \) are not equicardinal.

Assume that we have some enumeration \( f \) of \( 2^N \).

Consider \( C = \{ x \in N : x \notin f(x) \} \).

Since \( C \) is a subset of \( N \), we have that there is a \( c \in N \), such that \( f(c) = C \).
The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.

Proof

Assume that we have some enumeration $f$ of $2^N$.

Consider $C = \{ x \in N : x \notin f(x) \}$.

Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, ...
Diagonalization and Cantor

**Theorem**

The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.

**Proof**

Assume that we have some enumeration $f$ of $2^N$.

Consider $C = \{ x \in N : x \notin f(x) \}$.

Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \notin f(c)$,
Diagonalization and Cantor

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof</th>
</tr>
</thead>
</table>
| Assume that we have some enumeration $f$ of $2^N$.  

Consider $C = \{ x \in N : x \notin f(x) \}$.  

Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$.  

If $c \in C$, then $c \in f(c)$, hence $c \notin C$. |
The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.

Proof

Assume that we have some enumeration $f$ of $2^N$.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$,
The set of natural numbers, \( N \) and its power set \( 2^N \) are not equicardinal.

**Proof**

Assume that we have some enumeration \( f \) of \( 2^N \).

Consider \( C = \{ x \in N : x \notin f(x) \} \).

Since \( C \) is a subset of \( N \), we have that there is a \( c \in N \), such that \( f(c) = C \).

If \( c \in C \), then \( c \in f(c) \), hence \( c \notin C \).

If \( c \notin C \), then \( c \notin f(c) \),
Diagonalization and Cantor

**Theorem**

The set of natural numbers, $N$ and its power set $2^N$ are not equicardinal.

**Proof**

Assume that we have some enumeration $f$ of $2^N$.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$. 
The set of natural numbers, $\mathbb{N}$ and its power set $2^\mathbb{N}$ are not equicardinal.

Proof

Assume that we have some enumeration $f$ of $2^\mathbb{N}$.

Consider $C = \{ x \in \mathbb{N} : x \notin f(x) \}$.

Since $C$ is a subset of $\mathbb{N}$, we have that there is a $c \in \mathbb{N}$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$.

We have that $c \notin C$ if and only if $c \in C$. 
The set of natural numbers, $N$, and its power set $2^N$ are not equicardinal.

**Proof**

Assume that we have some enumeration $f$ of $2^N$.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since $C$ is a subset of $N$, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$.

We have that $c \notin C$ if and only if $c \in C$, which is a contradiction.
Idea
Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem
Given a program $\Pi$ and an input $x$, determine whether $\Pi$ will halt when $x$ is given as the input.

Theorem
The Halting Problem is undecidable.
Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

**Problem**
Given a program $\Pi$ and an input $x$, determine whether $\Pi$ will halt when $x$ is given as the input.

**Theorem**
The Halting Problem is undecidable.
Since some programs halt and others do not,
Idea

Since some programs halt and others do not, it would be nice to be able to tell which is which.
Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:
The Halting Problem

Idea
Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem
Given a program $\Pi$ and an input $x$, 
The Halting Problem

Idea
Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem
Given a program $\Pi$ and an input $x$, determine whether $\Pi$ will halt when $x$ is given as the input.
The Halting Problem

Idea
Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem
Given a program $\Pi$ and an input $x$, determine whether $\Pi$ will halt when $x$ is given as the input.

Theorem
The Halting Problem is undecidable.
The Halting Problem

Proof
Assume that there is a program $A$ that solves the Halting Problem. $A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows:
if $A(\Pi, \Pi) = \text{TRUE}$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = \text{FALSE}$, hence $B(B)$ does not halt.
If $B(B)$ does not halt, then $A(B, B) = \text{TRUE}$, hence $B(B)$ halts.
In both cases we have a contradiction, hence $A$ cannot exist.
Proof

Assume that there is a program $A$ that solves the Halting Problem. $A$ returns TRUE if $\Pi$ halts on $x$, and FALSE otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = \text{TRUE}$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = \text{FALSE}$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = \text{TRUE}$, hence $B(B)$ halts.

In both cases we have a contradiction, hence $A$ cannot exist.
Proof

Assume that there is a program $A$ that solves the Halting Problem.

If $B(B)$ halts, then $A(B,B) = \text{FALSE}$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B,B) = \text{TRUE}$, hence $B(B)$ halts.

In both cases we have a contradiction, hence $A$ cannot exist.
Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$,
The Halting Problem

Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.
Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows:
### The Halting Problem

#### Proof

Assume that there is a program $A$ that solves the Halting Problem.

A returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop,
The Halting Problem

Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.
The Halting Problem

Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns $TRUE$ if $\Pi$ halts on $x$, and $FALSE$, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns $TRUE$.

If $B(B)$ halts,
Proof

Assume that there is a program $A$ that solves the Halting Problem. $A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$. 

If $B(B)$ does not halt, then $A(B, B) = TRUE$, hence $B(B)$ halts.

In both cases we have a contradiction, hence $A$ cannot exist.
The Halting Problem

Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.
The Halting Problem

Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt,
The Halting Problem

Proof

Assume that there is a program \( A \) that solves the Halting Problem.

\( A \) returns TRUE if \( \Pi \) halts on \( x \), and FALSE, otherwise.

Consider a program \( B \) that is defined as follows: if \( A(\Pi, \Pi) = \text{TRUE} \), then \( B \) goes to an infinite loop, otherwise \( B \) returns TRUE.

If \( B(B) \) halts, then \( A(B, B) = \text{FALSE} \), hence \( B(B) \) does not halt.

If \( B(B) \) does not halt, then \( A(B, B) = \text{TRUE} \),
Proof

Assume that there is a program $A$ that solves the Halting Problem. $A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = TRUE$, hence $B(B)$ halts.
The Halting Problem

Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = TRUE$, hence $B(B)$ halts.

In both cases we have a contradiction,
Proof

Assume that there is a program $A$ that solves the Halting Problem.

$A$ returns TRUE if $\Pi$ halts on $x$, and FALSE, otherwise.

Consider a program $B$ that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then $B$ goes to an infinite loop, otherwise $B$ returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = TRUE$, hence $B(B)$ halts.

In both cases we have a contradiction, hence $A$ cannot exist.
The 42 Problem

We have one undecidable problem.

We can prove that other problems are undecidable by reducing The Halting Problem to them.

Consider the following problem:

Problem

Given a program $\Pi$. Is there an input $x$, such that $\Pi(x)$ halts and returns 42?

Theorem

The $42$ Problem is undecidable.
The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them.

Consider the following problem:

Problem

Given a program $\Pi$. Is there an input $x$, such that $\Pi(x)$ halts and returns 42?

Theorem

The 42 Problem is undecidable.
The 42 Problem

Idea

We have one undecidable problem.
The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them.
The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:
The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem

Given a program $\Pi$. 

Theorem

The 42 Problem is undecidable.
The 42 Problem

Idea
We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem
Given a program $\Pi$. Is there an input $x$,
The 42 Problem

Idea
We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem
Given a program $\Pi$. Is there an input $x$, such that $\Pi(x)$ halts and returns 42?
The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem

Given a program $\Pi$. Is there an input $x$, such that $\Pi(x)$ halts and returns 42?

Theorem

The 42 Problem is undecidable.
The 42 Problem

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too. But we know that the Halting Problem is undecidable, hence the 42 Problem must be undecidable as well.
The 42 Problem

Proof

Given a program \( \Pi \) and an input \( x \), we can convert them to a program \( \Pi' \) which ignores its input, runs \( \Pi(x) \) instead, and returns 42 if it halts.

If \( \Pi(x) \) halts, then \( \Pi'(x') \) returns 42.

If \( \Pi(x) \) does not halt, then neither does \( \Pi' \), no matter what input \( x' \) we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too. But we know that the Halting Problem is undecidable, hence the 42 Problem must be undecidable as well.
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts. If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42. If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it. Thus, if the 42 Problem were decidable, the Halting Problem would be too. But we know that the Halting Problem is undecidable, hence the 42 Problem must be undecidable as well.
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead,
Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.
Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$,
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it.
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it.

Thus, if the 42 Problem were decidable,
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too.
The 42 Problem

Proof

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too.

But we know that the Halting Problem is undecidable,
The 42 Problem

**Proof**

Given a program $\Pi$ and an input $x$, we can convert them to a program $\Pi'$ which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does $\Pi'$, no matter what input $x'$ we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too.

But we know that the Halting Problem is undecidable, hence the 42 Problem must be undecidable as well.
The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same. It shows that the 42 Problem is at least hard as the Halting Problem, that is

\[ \text{The Halting Problem} \leq \text{The 42 problem}. \]

The reductions that we used in the proof are computable reductions. That is, a reduction can be any function from instances of \( A \) to instances of \( B \) that we can compute in finite time. In this case, \( A \leq B \) implies that if \( B \) is decidable then \( A \) is decidable, and conversely, if \( A \) is not decidable, then \( B \) is undecidable, too.
The idea

The mapping that we have just constructed maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same. It shows that the 42 Problem is at least as hard as the Halting Problem, that is $\text{Halting Problem} \leq \text{42 Problem}$. The reductions that we used in the proof are computable reductions. That is, a reduction can be any function from instances of $A$ to instances of $B$ that we can compute in finite time. In this case, $A \leq B$ implies that if $B$ is decidable then $A$ is decidable, and conversely, if $A$ is not decidable, then $B$ is undecidable, too.
The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.
The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

The Halting Problem \( \leq \) The 42 problem.
The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

\[
\text{The Halting Problem} \leq \text{The 42 problem.}
\]

The reductions that we used in the proof are computable reductions.
The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

\[
\text{The Halting Problem} \leq \text{The 42 problem.}
\]

The reductions that we used in the proof are *computable* reductions.

That is, a reduction can be any function from instances of $A$ to instances of $B$ that we can compute in finite time.
The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

\[ \text{The Halting Problem} \leq \text{The 42 problem}. \]

The reductions that we used in the proof are *computable* reductions.

That is, a reduction can be any function from instances of \(A\) to instances of \(B\) that we can compute in finite time.

In this case, \(A \leq B\) implies that if \(B\) is decidable then \(A\) is decidable, and conversely, if \(A\) is not decidable, then \(B\) is undecidable, too.
The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability. If the instance $(\Pi, x)$ is a YES instance, then we can learn this fact in a finite amount of time, by simulating $\Pi$ until it halts.

In other words, the Halting Problem can be represented as:

$$\text{Halts}(\Pi, x) = \exists t: \text{HaltsInTime}(\Pi, x, t),$$

where $\text{HaltsInTime}(\Pi, x, t)$ is the property that $\Pi$, given $x$ as input, halts on its $t$th step.

$\text{HaltsInTime}(\Pi, x, t)$ is decidable?

Simulate $\Pi$ for $t$ steps.

Thus $\text{Halts}(\Pi, x)$ is a combination of a decidable problem with a single $\exists$. 

The Grand Unified Theory of Computation

Recursive Enumerability
The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability. If the instance $(\Pi, x)$ is a YES instance, then we can learn this fact in a finite amount of time, by simulating $\Pi$ until it halts.

In other words, the Halting Problem can be represented as:

$$\text{Halts}(\Pi, x) = \exists t : \text{HaltsInTime}(\Pi, x, t),$$

where $\text{HaltsInTime}(\Pi, x, t)$ is the property that $\Pi$, given $x$ as input, halts on its $t$th step.

$\text{HaltsInTime}(\Pi, x, t)$ is decidable?

Simulate $\Pi$ for $t$ steps.

Thus $\text{Halts}(\Pi, x)$ is a combination of a decidable problem with a single $\exists$. 

The Grand Unified Theory of Computation
Computational Complexity
The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.
While the Halting Problem is undecidable, it has kind of one-sided decidability. If the instance \((\Pi, x)\) is a YES instance, then we can learn this fact in a finite amount of time, by simulating \(\Pi\) until it halts.
While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance \((\Pi, x)\) is a YES instance, then we can learn this fact in a finite amount of time, by simulating \(\Pi\) until it halts.

In other words, the Halting Problem can be represented as:
The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance \((\Pi, x)\) is a YES instance, then we can learn this fact in a finite amount of time, by simulating \(\Pi\) until it halts.

In other words, the Halting Problem can be represented as:

\[
\text{Halts}(\Pi, x) = \exists t : \text{HaltsInTime}(\Pi, x, t),
\]
While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance $(\Pi, x)$ is a YES instance, then we can learn this fact in a finite amount of time, by simulating $\Pi$ until it halts.

In other words, the Halting Problem can be represented as:

$$\text{Halts}(\Pi, x) = \exists t : \text{HaltsInTime}(\Pi, x, t),$$

where $\text{HaltsInTime}(\Pi, x, t)$ is the property that $\Pi$, given $x$ as input, halts on its $t$th step.
The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance \((\Pi, x)\) is a YES instance, then we can learn this fact in a finite amount of time, by simulating \(\Pi\) until it halts.

In other words, the Halting Problem can be represented as:

\[
Halts(\Pi, x) = \exists t : HaltsInTime(\Pi, x, t),
\]

where \(HaltsInTime(\Pi, x, t)\) is the property that \(\Pi\), given \(x\) as input, halts on its \(t\)th step.

\(HaltsInTime(\Pi, x, t)\) is decidable?
The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance \((\Pi, x)\) is a YES instance, then we can learn this fact in a finite amount of time, by simulating \(\Pi\) until it halts.

In other words, the Halting Problem can be represented as:

\[
\text{Halts}(\Pi, x) = \exists t : \text{HaltsInTime}(\Pi, x, t),
\]

where \(\text{HaltsInTime}(\Pi, x, t)\) is the property that \(\Pi\), given \(x\) as input, halts on its \(t\)th step. \(\text{HaltsInTime}(\Pi, x, t)\) is decidable? Simulate \(\Pi\) for \(t\) steps.
Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance $(\Pi, x)$ is a YES instance, then we can learn this fact in a finite amount of time, by simulating $\Pi$ until it halts.

In other words, the Halting Problem can be represented as:

\[ \text{Halts}(\Pi, x) = \exists t : \text{HaltsInTime}(\Pi, x, t), \]

where $\text{HaltsInTime}(\Pi, x, t)$ is the property that $\Pi$, given $x$ as input, halts on its $t$th step.

$\text{HaltsInTime}(\Pi, x, t)$ is decidable? Simulate $\Pi$ for $t$ steps.

Thus $\text{Halts}(\Pi, x)$ is a combination of a decidable problem with a single $\exists$. 

Recursive Enumerability

Definition
Let $\text{RE}$ denote the class of problems that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with $P$ and $NP$
In some ways, this is analogous to the relationship between $P$ and $NP$.
Decidable problems are the analogues of $P$.
Recall that a property $A$ is in $NP$, if it can be written as $A(x) = \exists w : B(x, w)$, where $B$ is in $P$.
In other words, $x$ is a YES instance of $A$, if some witness $w$ exists, and the property $B(x, w)$, that $w$ is a valid witness for $x$, can be checked in polynomial time.
Similarly, $t$ is a witness that $\Pi$ halts, and we can check the validity of this witness in finite time.

The Grand Unified Theory of Computation
Computational Complexity
Recursive Enumerability

Definition

Let $\text{RE}$ denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$. 

Analogy with $\text{P}$ and $\text{NP}$

In some ways, this is analogous to the relationship between $\text{P}$ and $\text{NP}$.

Decidable problems are the analogues of $\text{P}$.

Recall that a property $A(x)$ is in $\text{NP}$, if it can be written as $A(x) = \exists w : B(x, w)$, where $B$ is in $\text{P}$.

In other words, $x$ is a YES instance of $A$, if some witness $w$ exists, and the property $B(x, w)$, that $w$ is a valid witness for $x$, can be checked in polynomial time.

Similarly, $t$ is a witness that $\Pi$ halts, and we can check the validity of this witness in finite time.
Recursive Enumerability

Definition

Let \( \text{RE} \) denote the class of problems, that can be represented as a combination of a decidable problem with a single \( \exists \).

Analogy with \( \text{P} \) and \( \text{NP} \)

In some ways, this is analogous to the relationship between \( \text{P} \) and \( \text{NP} \). Decidable problems are the analogues of \( \text{P} \).

Recall that a property \( A(x) \) is in \( \text{NP} \), if it can be written as \( A(x) = \exists w : B(x, w) \), where \( B \) is in \( \text{P} \).

In other words, \( x \) is a YES instance of \( A \), if some witness \( w \) exists, and the property \( B(x, w) \), that \( w \) is a valid witness for \( x \), can be checked in polynomial time.

Similarly, \( t \) is a witness that \( \Pi \) halts, and we can check the validity of this witness in finite time.
Definition

Let \( \text{RE} \) denote the class of problems, that can be represented as a combination of a decidable problem with a single \( \exists \).

Analogies with \( \mathbb{P} \) and \( \mathbb{NP} \)

In some ways, this is analogous to the relationship between \( \mathbb{P} \) and \( \mathbb{NP} \).
Recursive Enumerability

**Definition**

Let $\text{RE}$ denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

**Analogy with $P$ and $NP$**

In some ways, this is analogous to the relationship between $P$ and $NP$.

Decidable problems are the analogues of $P$. 
Recursive Enumerability

Definition

Let \textbf{RE} denote the class of problems, that can be represented as a combination of a decidable problem with a single \exists.

Analogy with \textbf{P} and \textbf{NP}

In some ways, this is analogous to the relationship between \textbf{P} and \textbf{NP}.

Decidable problems are the analogues of \textbf{P}.

Recall that a property \( A \) is in \textbf{NP},

\[ A(x) = \exists w : B(x, w), \]

where \( B \) is in \textbf{P}.

In other words, \( x \) is a YES instance of \( A \), if some witness \( w \) exists, and the property \( B(x, w) \), that \( w \) is a valid witness for \( x \), can be checked in polynomial time.

Similarly, \( t \) is a witness that \( \Pi \) halts, and we can check the validity of this witness in finite time.
### Recursive Enumerability

#### Definition

Let \( \text{RE} \) denote the class of problems, that can be represented as a combination of a decidable problem with a single \( \exists \).

#### Analogy with \( \mathbf{P} \) and \( \mathbf{NP} \)

In some ways, this is analogous to the relationship between \( \mathbf{P} \) and \( \mathbf{NP} \).

Decidable problems are the analogues of \( \mathbf{P} \).

Recall that a property \( A \) is in \( \mathbf{NP} \), if it can be written as

\[
A(x) = \exists w : B(x, w),
\]

where \( B \) is in \( \mathbf{P} \).
Definition

Let $\text{RE}$ denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with $\text{P}$ and $\text{NP}$

In some ways, this is analogous to the relationship between $\text{P}$ and $\text{NP}$.

Decidable problems are the analogues of $\text{P}$.

Recall that a property $A$ is in $\text{NP}$, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where $B$ is in $\text{P}$.
Recursive Enumerability

**Definition**

Let \( \text{RE} \) denote the class of problems, that can be represented as a combination of a decidable problem with a single \( \exists \).

**Analogy with \( \mathbb{P} \) and \( \mathbb{NP} \)**

In some ways, this is analogous to the relationship between \( \mathbb{P} \) and \( \mathbb{NP} \).

Decidable problems are the analogues of \( \mathbb{P} \).

Recall that a property \( A \) is in \( \mathbb{NP} \), if it can be written as

\[
A(x) = \exists w : B(x, w),
\]

where \( B \) is in \( \mathbb{P} \).

In other words, \( x \) is a YES instance of \( A \), if some witness \( w \) exists,
Recursive Enumerability

Definition

Let \( \text{RE} \) denote the class of problems, that can be represented as a combination of a decidable problem with a single \( \exists \).

Analogy with \( \text{P} \) and \( \text{NP} \)

In some ways, this is analogous to the relationship between \( \text{P} \) and \( \text{NP} \).

Decidable problems are the analogues of \( \text{P} \).

Recall that a property \( A \) is in \( \text{NP} \), if it can be written as

\[
A(x) = \exists w : B(x, w),
\]

where \( B \) is in \( \text{P} \).

In other words, \( x \) is a YES instance of \( A \), if some witness \( w \) exists, and the property \( B(x, w) \), that \( w \) is a valid witness for \( x \), can be checked in polynomial time.
## Recursive Enumerability

### Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

### Analogy with $\textbf{P}$ and $\textbf{NP}$

In some ways, this is analogous to the relationship between $\textbf{P}$ and $\textbf{NP}$.

Decidable problems are the analogues of $\textbf{P}$.

Recall that a property $A$ is in $\textbf{NP}$, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where $B$ is in $\textbf{P}$.

In other words, $x$ is a YES instance of $A$, if some witness $w$ exists, and the property $B(x, w)$, that $w$ is a valid witness for $x$, can be checked in polynomial time.

Similarly, $t$ is a witness that $\Pi$ halts,
**Recursive Enumerability**

**Definition**

Let $\text{RE}$ denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

**Analogy with $\mathbf{P}$ and $\mathbf{NP}$**

In some ways, this is analogous to the relationship between $\mathbf{P}$ and $\mathbf{NP}$.

Decidable problems are the analogues of $\mathbf{P}$.

Recall that a property $A$ is in $\mathbf{NP}$, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where $B$ is in $\mathbf{P}$.

In other words, $x$ is a YES instance of $A$, if some witness $w$ exists, and the property $B(x, w)$, that $w$ is a valid witness for $x$, can be checked in polynomial time.

Similarly, $t$ is a witness that $\Pi$ halts, and we can check the validity of this witness in finite time.
Recursion and Enumerability

More analogy with \( P \) and \( \text{NP} \)

Recall that \( \text{coNP} \) stands for the class of problems for which NO instances have witnesses, whose validity can be verified in polynomial time.

Similarly, the class \( \text{coRE} \) stands for the class of problems, whose NO instances are in \( \text{RE} \).
Recall that $\text{coNP}$ stands for the class of problems for which NO instances have witnesses, whose validity can be verified in polynomial time. Similarly, the class $\text{coRE}$ stands for the class of problems, whose NO instances are in $\text{RE}$.
More analogy with $P$ and $NP$

Recall that $coNP$ stands for the class of problems for which NO instances have witnesses,
More analogy with P and NP

Recall that \textbf{coNP} stands for the class of problems for which NO instances have witnesses, whose validity can be verified in polynomial time.
More analogy with $P$ and $NP$:

Recall that $\text{coNP}$ stands for the class of problems for which NO instances have witnesses, whose validity can be verified in polynomial time.

Similarly, the class $\text{coRE}$ stands for the class of problems, whose NO instances are in $\text{RE}$. 
Recursive Enumerability and the P vs. NP Problem

Unlike the polynomial world, where the P vs. NP question remains unsolved, we know that RE, coRE, and Decidable are different. Show that Decidable = RE ∩ coRE. In other words, if both S and ¯S are in RE, then S is decidable. From this one can conclude that RE ≠ coRE.

In contrast, the questions whether NP ≠ coNP and P = NP ∩ coNP are still open.
Unlike the polynomial world, where the $P$ vs. $NP$ question remains unsolved, we know that $RE$, $coRE$, and $Decidable$ are different.

Show that $Decidable = RE \cap coRE$.

In other words, if both $S$ and $\overline{S}$ are in $RE$, then $S$ is decidable.

From this one can conclude that $RE \neq coRE$.

In contrast, the questions whether $NP \neq coNP$ and $P = NP \cap coNP$ are still open.
Some relations among the classes

Unlike the polynomial world, where the P vs. NP question remains unsolved, we know that RE, coRE, and Decidable are different. Show that Decidable = RE ∩ coRE. In other words, if both S and ¯S are in RE, then S is decidable. From this one can conclude that RE ≠ coRE. In contrast, the questions whether NP ≠ coNP and P = NP ∩ coNP are still open.
Some relations among the classes

Unlike the polynomial world, where the P vs. NP question remains unsolved, we know that RE, coRE and Decidable are different.
Some relations among the classes

Unlike the polynomial world, where the P vs. NP question remains unsolved, we know that \textit{RE, coRE} and \textit{Decidable} are different.

Show that

\[
\text{Decidable} = \text{RE} \cap \text{coRE}.
\]
Some relations among the classes

Unlike the polynomial world, where the \( P \) vs. \( NP \) question remains unsolved, we know that \( RE \), \( coRE \) and \( Decidable \) are different.

Show that

\[
\text{Decidable} = RE \cap coRE.
\]

In other words, if both \( S \) and \( \bar{S} \) are in \( RE \),
Recursive Enumerability and the $P$ vs. $NP$ Problem

Some relations among the classes

Unlike the polynomial world, where the $P$ vs. $NP$ question remains unsolved, we know that $RE$, $coRE$ and $Decidable$ are different.

Show that

$$Decidable = RE \cap coRE.$$ 

In other words, if both $S$ and $\overline{S}$ are in $RE$, then $S$ is decidable.
Some relations among the classes

Unlike the polynomial world, where the $P$ vs. $NP$ question remains unsolved, we know that $RE$, $coRE$ and $Decidable$ are different.

Show that

$$\text{Decidable} = RE \cap coRE.$$

In other words, if both $S$ and $\overline{S}$ are in $RE$, then $S$ is decidable.

From this one can conclude that $RE \neq coRE$. 
Some relations among the classes

Unlike the polynomial world, where the \( P \) vs. \( NP \) question remains unsolved, we know that \( RE, \ coRE \) and \( Decidable \) are different.

Show that

\[
\text{Decidable} = RE \cap coRE.
\]

In other words, if both \( S \) and \( \bar{S} \) are in \( RE \), then \( S \) is decidable.

From this one can conclude that \( RE \neq coRE \).

In contrast, the questions whether \( NP \neq coNP \) and \( P = NP \cap coNP \) are still open.
Polynomial Hierarchy

Definition

Let $D$ be a class of problems.
Definition

Let $D$ be a class of problems. A problem $L$ is in $P^D$, 

Definition

Let $D$ be a class of problems. A problem $L$ is in $P^D$, if there exists a problem $L' \in D$, such that $L$ can be solved in polynomial time by an oracle program using an $L'$ oracle.
Polynomial Hierarchy

Definition

The polynomial hierarchy is the following sequence of classes:

1. $\Delta^P_0 = \Sigma^P_0 = \Pi^P_0 = P$
2. $\Delta^P_{i+1} = \Sigma^P_{i+1} = \Pi^P_{i+1}$
3. $\Sigma^P_{i+1} = \text{NP}$
4. $\Pi^P_{i+1} = \text{coNP}$

For all $i \geq 0$.

We also define the collective class $\text{PH} = \bigcup_{i \geq 0} \Sigma^P_i$.

Observations

Note that because $\Sigma^P_0 = \text{P}$, we have that $\Sigma^P_1 = \text{NP}$, $\Delta^P_1 = \text{P}$, and $\Pi^P_1 = \text{coNP}$.

At each level the classes are believed to be distinct and are known to hold the same relationship as $\text{P}$, $\text{NP}$, and $\text{coNP}$.

Also, each class at each level includes all classes at the previous levels.
The polynomial hierarchy is the following sequence of classes:

1. $\Sigma^P_0 = P$
2. $\Delta^P_1 = P$
3. $\Pi^P_1 = \text{coNP}$
4. $\Sigma^P_1 = \text{NP}$
5. $\Delta^P_i = \text{P}$
6. $\Pi^P_i = \text{coNP}$

For all $i \geq 0$.

We also define the collective class $\text{PH} = \bigcup_{i \geq 0} \Sigma^P_i$.

Observations

Note that because $\Sigma^P_0 = P$, we have that $\Sigma^P_1 = \text{NP}$, $\Delta^P_1 = \text{P}$, and $\Pi^P_1 = \text{coNP}$.

At each level the classes are believed to be distinct and are known to hold the same relationship as $\text{P}$, $\text{NP}$, and $\text{coNP}$.

Also, each class at each level includes all classes at the previous levels.
The polynomial hierarchy is the following sequence of classes:
The polynomial hierarchy is the following sequence of classes:

1. \( \Delta_0 \mathbf{P} = \Sigma_0 \mathbf{P} = \Pi_0 \mathbf{P} = \mathbf{P} \)
Definition

The polynomial hierarchy is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = P$
2. $\Delta_{i+1} P = P^{\Sigma_i P}$
The polynomial hierarchy is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = P$
2. $\Delta_{i+1} P = P^{\Sigma_i P}$
3. $\Sigma_{i+1} P = NP^{\Sigma_i P}$

For all $i \geq 0$.

We also define the collective class $PH = \Sigma_i P$. 

Observations

Note that because $\Sigma_0 P = P$, we have that $\Sigma_1 P = NP$, $\Delta_1 P = P$, and $\Pi_1 P = \text{coNP}$. At each level the classes are believed to be distinct and are known to hold the same relationship as $P$, $NP$ and $\text{coNP}$. Also, each class at each level includes all classes at the previous levels.
Definition

The *polynomial hierarchy* is the following sequence of classes:

1. \( \Delta_0 P = \Sigma_0 P = \Pi_0 P = P \)
2. \( \Delta_{i+1} P = P^{\Sigma_i P} \)
3. \( \Sigma_{i+1} P = NP^{\Sigma_i P} \)
4. \( \Pi_{i+1} P = coNP^{\Sigma_i P} \)
Definition

The polynomial hierarchy is the following sequence of classes:

1. \( \Delta_0 P = \Sigma_0 P = \Pi_0 P = P \)
2. \( \Delta_{i+1} P = P^{\Sigma_i P} \)
3. \( \Sigma_{i+1} P = NP^{\Sigma_i P} \)
4. \( \Pi_{i+1} P = coNP^{\Sigma_i P} \)

For all \( i \geq 0 \).
Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0^P = \Sigma_0^P = \Pi_0^P = \text{P}$
2. $\Delta_{i+1}^P = \text{P}^{\Sigma_i^P}$
3. $\Sigma_{i+1}^P = \text{NP}^{\Sigma_i^P}$
4. $\Pi_{i+1}^P = \text{coNP}^{\Sigma_i^P}$

For all $i \geq 0$. We also define the collective class $\text{PH} = \bigcup_{i\geq 0} \Sigma_i^P$. 
Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 \mathbb{P} = \Sigma_0 \mathbb{P} = \Pi_0 \mathbb{P} = \mathbb{P}$
2. $\Delta_{i+1} \mathbb{P} = \mathbb{P}^{\Sigma_i \mathbb{P}}$
3. $\Sigma_{i+1} \mathbb{P} = \text{NP}^{\Sigma_i \mathbb{P}}$
4. $\Pi_{i+1} \mathbb{P} = \text{coNP}^{\Sigma_i \mathbb{P}}$

For all $i \geq 0$. We also define the collective class $\text{PH} = \bigcup_{i \geq 0} \Sigma_i \mathbb{P}$.

Observations

Note that because $\Sigma_0 \mathbb{P} = \mathbb{P}$, we have that $\Sigma_1 \mathbb{P} = \text{NP}$, $\Delta_1 \mathbb{P} = \mathbb{P}$, and $\Pi_1 \mathbb{P} = \text{coNP}$. 
### Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = P$
2. $\Delta_{i+1} P = P^{\Sigma_i P}$
3. $\Sigma_{i+1} P = \text{NP}^{\Sigma_i P}$
4. $\Pi_{i+1} P = \text{coNP}^{\Sigma_i P}$

For all $i \geq 0$. We also define the collective class $\text{PH} = \bigcup_{i \geq 0} \Sigma_i P$.

### Observations

Note that because $\Sigma_0 P = P$, we have that $\Sigma_1 P = \text{NP}$, $\Delta_1 P = P$, and $\Pi_1 P = \text{coNP}$.

At each level the classes are believed to be distinct and are known to hold the same relationship as $P$, $\text{NP}$ and $\text{coNP}$. 

---

**The Grand Unified Theory of Computation**

**Computational Complexity**
### Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = P$
2. $\Delta_{i+1} P = P^{\Sigma_i P}$
3. $\Sigma_{i+1} P = NP^{\Sigma_i P}$
4. $\Pi_{i+1} P = coNP^{\Sigma_i P}$

For all $i \geq 0$. We also define the collective class $PH = \bigcup_{i \geq 0} \Sigma_i P$.

### Observations

Note that because $\Sigma_0 P = P$, we have that $\Sigma_1 P = NP$, $\Delta_1 P = P$, and $\Pi_1 P = coNP$.

At each level the classes are believed to be distinct and are known to hold the same relationship as $P$, $NP$ and $coNP$.

Also, each class at each level includes all classes at the previous levels.
**Arithmetical Hierarchy**

The arithmetical hierarchy is the following sequence of classes:

1. $\Delta_0^D = \Sigma_0^P = \Pi_0^P = \text{Decidable}$
2. $\Delta_{i+1}^D = \text{Decidable}$
3. $\Sigma_{i+1}^D = \text{RE}$
4. $\Pi_{i+1}^D = \text{coRE}$

For all $i \geq 0$.

We also define the collective class $\text{AH} = \bigcup_{i \geq 0} \Sigma_i^D$.

What's Known

Unlike the polynomial hierarchy, it is known that the levels of the arithmetical hierarchy are distinct.
Definition

The arithmetical hierarchy is the following sequence of classes:

1. $\Delta_0^D = \Sigma_0^P = \Pi_0^P = \text{Decidable}$
2. $\Delta_{i+1}^D = \text{Decidable}$
3. $\Sigma_{i+1}^D = \text{RE}$
4. $\Pi_{i+1}^D = \text{coRE}$

For all $i \geq 0$.

We also define the collective class $\mathcal{AH} = \bigcup_{i \geq 0} \Sigma_i^D$.

What's Known

Unlike the polynomial hierarchy, it is known that the levels of the arithmetical hierarchy are distinct.
Definition

The *arithmetical hierarchy* is the following sequence of classes:
The arithmetic hierarchy is the following sequence of classes:

1. $\Delta_0 D = \Sigma_0 P = \Pi_0 P = \text{Decidable}$

What's Known

Unlike the polynomial hierarchy, it is known that the levels of the arithmetic hierarchy are distinct.

The Grand Unified Theory of Computation

Computational Complexity
**Definition**

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 D = \Sigma_0 P = \Pi_0 P = \text{Decidable}$
2. $\Delta_{i+1} D = \text{Decidable}^{\Sigma_i D}$
The arithmetical hierarchy is the following sequence of classes:

1. $$\Delta_0 D = \Sigma_0 P = \Pi_0 P = \text{Decidable}$$
2. $$\Delta_{i+1} D = \text{Decidable}^{\Sigma_i D}$$
3. $$\Sigma_{i+1} D = \text{RE}^{\Sigma_i D}$$
### Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 D = \Sigma_0 P = \Pi_0 P = \text{Decidable}$
2. $\Delta_{i+1} D = \text{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1} D = \text{RE}^{\Sigma_i D}$
4. $\Pi_{i+1} D = \text{coRE}^{\Sigma_i D}$

For all $i \geq 0$.

We also define the collective class $AH = \sum_{i \geq 0} \Sigma_i D$. What’s Known

Unlike the polynomial hierarchy, it is known that the levels of the arithmetical hierarchy are distinct.
The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 D = \Sigma_0 P = \Pi_0 P = \text{Decidable}$
2. $\Delta_{i+1} D = \text{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1} D = \text{RE}^{\Sigma_i D}$
4. $\Pi_{i+1} D = \text{coRE}^{\Sigma_i D}$

For all $i \geq 0$. 

We also define the collective class $\text{AH} = \sum_{i \geq 0} \Sigma_i D$. 

What’s Known 

Unlike the polynomial hierarchy, it is known that the levels of the arithmetical hierarchy are distinct.
Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 \mathbb{D} = \Sigma_0 \mathbb{P} = \Pi_0 \mathbb{P} = \text{Decidable}$
2. $\Delta_{i+1} \mathbb{D} = \text{Decidable}^{\Sigma_i \mathbb{D}}$
3. $\Sigma_{i+1} \mathbb{D} = \text{RE}^{\Sigma_i \mathbb{D}}$
4. $\Pi_{i+1} \mathbb{D} = \text{coRE}^{\Sigma_i \mathbb{D}}$

For all $i \geq 0$. We also define the collective class $\mathbb{AH} = \bigcup_{i \geq 0} \Sigma_i \mathbb{D}$. 

The arithmetical hierarchy is the following sequence of classes:

1. $\Delta_0^D = \Sigma_0^P = \Pi_0^P = \text{Decidable}$
2. $\Delta_{i+1}^D = \text{Decidable}^{\Sigma_i^D}$
3. $\Sigma_{i+1}^D = \text{RE}^{\Sigma_i^D}$
4. $\Pi_{i+1}^D = \text{coRE}^{\Sigma_i^D}$

For all $i \geq 0$. We also define the collective class $\text{AH} = \bigcup_{i \geq 0} \Sigma_i^D$. 

What’s Known
Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 D = \Sigma_0 P = \Pi_0 P = \text{Decidable}$
2. $\Delta_{i+1} D = \text{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1} D = \text{RE}^{\Sigma_i D}$
4. $\Pi_{i+1} D = \text{coRE}^{\Sigma_i D}$

For all $i \geq 0$. We also define the collective class $\text{AH} = \bigcup_{i \geq 0} \Sigma_i D$. 

What’s Known

Unlike the polynomial hierarchy, it is known that the levels of the arithmetical hierarchy are distinct.
Formal Systems

A formal system has a finite set of axioms including rules of inference such as modus ponens:

\[ A \land A \rightarrow B \implies B. \]

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement \( T \) such that both \( T \) and \( \neg T \) are theorems.

A formal system is complete, if for each statement \( T \), at least one of \( T \) and \( \neg T \) is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way.∃ - there exists, ∧ - and, so on, and assuming that its variables refer to a specific set of mathematical objects such as integers.
Formal Systems

The idea
Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens.
The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. $A$ and $A \rightarrow B$ implies $B$. 
The idea

A \textit{formal system} has a finite set of axioms including rules of inference such as modus ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.
Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent,
Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. \( A \) and \( A \rightarrow B \) implies \( B \).

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement \( T \) such that both \( T \) and \( \neg T \) are theorems.
A formal system has a finite set of axioms including rules of inference such as modus
ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from
the axioms.

A formal system is consistent, if there is no statement $T$ such that both $T$ and $\neg T$ are
theorems.

A formal system is complete,
The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement $T$ such that both $T$ and $\neg T$ are theorems.

A formal system is complete, if for each statement $T$, at least one of $T$ and $\neg T$ is a theorem.
The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. \( A \) and \( A \rightarrow B \) implies \( B \).

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement \( T \) such that both \( T \) and \( \neg T \) are theorems.

A formal system is complete, if for each statement \( T \), at least one of \( T \) and \( \neg T \) is a theorem.

We can define a statement as true or false,
## The idea

A **formal system** has a finite set of axioms including rules of inference such as modus ponens. \( A \) and \( A \rightarrow B \) implies \( B \).

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement \( T \) such that both \( T \) and \( \bar{T} \) are theorems.

A formal system is complete, if for each statement \( T \), at least one of \( T \) and \( \bar{T} \) is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way.
The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement $T$ such that both $T$ and $\neg T$ are theorems.

A formal system is complete, if for each statement $T$, at least one of $T$ and $\neg T$ is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way. $\exists$ - there exists,
A **formal system** has a finite set of axioms including rules of inference such as modus ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement $T$ such that both $T$ and $\overline{T}$ are theorems.

A formal system is complete, if for each statement $T$, at least one of $T$ and $\overline{T}$ is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way. $\exists$ - there exists, $\land$ - and,
The idea

A *formal system* has a finite set of axioms including rules of inference such as modus
ponens. $A$ and $A \rightarrow B$ implies $B$.

A theorem is a statement that can be proved, with some finite chain of reasoning, from
the axioms.

A formal system is consistent, if there is no statement $T$ such that both $T$ and $\bar{T}$ are
theorems.

A formal system is complete, if for each statement $T$, at least one of $T$ and $\bar{T}$ is a
theorem.

We can define a statement as true or false, by interpreting the symbols of the formal
system in some standard way. $\exists$ - there exists, $\land$ - and, so on, and assuming that its
variables refer to a specific set of mathematical objects such as integers.
David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems. Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics. It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert's hopes. He proved that no sufficiently powerful system is both consistent and complete. He did this by constructing a self-referential statement, which can be interpreted as:

This statement cannot be proved.
David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems. Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics. It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert's hopes. He proved that no sufficiently powerful system is both consistent and complete. He did this by constructing a self-referential statement, which can be interpreted as:

“This statement cannot be proved.”
David Hilbert

The ideal system would be consistent and complete,
David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.
David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.
David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert’s hopes.

He proved that no sufficiently powerful system is both consistent and complete.

He did this by constructing a self-referential statement, which can be interpreted as:

This statement cannot be proved.
David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

---
David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert’s hopes.
David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert’s hopes.

He proved that no sufficiently powerful system is both consistent and complete.
David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert’s hopes.

He proved that no sufficiently powerful system is both consistent and complete.

He did this by constructing a self-referential statement, which can be interpreted as:
David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert’s dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert’s hopes.

He proved that no sufficiently powerful system is both consistent and complete.

He did this by constructing a self-referential statement, which can be interpreted as:

This statement cannot be proved.
Proof of Gödel’s Theorem

Idea

If this statement is false,
then it can be proved,
and it would violate the consistency.
Thus, it must be true,
and it is unprovable,
therefore there are truths that cannot be
proved.

Remark

What we did demonstrates that the problem is in English.
Gödel did something more, he showed that one can get similar statements in mathematics.
Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English. Gödel did something more, he showed that one can get similar statements in mathematics. Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.
Proof of Gödel’s Theorem

Idea

If this statement is false,

What we did demonstrates that the problem is in English. Gödel did something more, he showed that one can get similar statements in mathematics. Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved,
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true,
Proof of Gödel’s Theorem

**Idea**

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable,
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English. Gödel did something more, he showed that one can get similar statements in mathematics. Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English. Gödel did something more, he showed that one can get similar statements in mathematics. Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English.
Proof of Gödel’s Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English.

Gödel did something more, he showed that one can get similar statements in mathematics.
Proof of Gödel’s Theorem

Idea
If this statement is false, then it can be proved, hence it would violate the consistency. Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark
What we did demonstrates that the problem is in English.
Gödel did something more, he showed that one can get similar statements in mathematics.
Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.
Proof of Gödel’s Theorem

Proof

Let \( \text{Theorem}(T) \) be the property that a statement \( T \) is provable.

Then it can be written as:

\[
\text{Theorem}(T) = \exists P : \text{Proof}(P, T),
\]

where \( \text{Proof}(P, T) \) is the property that \( P \) is a valid proof of \( T \).

\( \text{Proof}(P, T) \) is decidable, because we can check the proof line by line.

Thus, the set of theorems is in \( \text{RE} \).

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like \( \forall \) and \( \exists \).

We assume that the theory can express statements like \( \text{Halts}(\Pi, x) \).

We assume that the axioms of the theory are strong enough to derive each step of a computation from the previous one.
Proof of Gödel’s Theorem

Proof

Let $\text{Theorem}(T)$ be the property that a statement $T$ is provable. Then it can be written as:

\[
\text{Theorem}(T) = \exists P: \text{Proof}(P, T),
\]

where $\text{Proof}(P, T)$ is the property that $P$ is a valid proof of $T$. $\text{Proof}(P, T)$ is decidable, because we can check the proof line by line. Thus, the set of theorems is in $\text{RE}$. We assume that our formal system is powerful enough to talk about computation. We assume that it includes quantifiers like $\forall$ and $\exists$. We assume that the theory can express statements like $\text{Halts}(\Pi, x)$. We assume that the axioms of the theory are strong enough to derive each step of a computation from the previous one.
Proof of Gödel’s Theorem

Proof

Let $Theorem(T)$ be the property that a statement $T$ is provable.
Proof of Gödel’s Theorem

Proof

Let $Theorem(T)$ be the property that a statement $T$ is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$
Proof of Gödel’s Theorem

Proof

Let \( \text{Theorem}(T) \) be the property that a statement \( T \) is provable.

Then it can be written as:

\[
\text{Theorem}(T) = \exists P : \text{Proof}(P, T),
\]

where \( \text{Proof}(P, T) \) is the property that \( P \) is a valid proof of \( T \).
Proof

Let $\text{Theorem}(T)$ be the property that a statement $T$ is provable.

Then it can be written as:

$$\text{Theorem}(T) = \exists P : \text{Proof}(P, T),$$

where $\text{Proof}(P, T)$ is the property that $P$ is a valid proof of $T$.

$\text{Proof}(P, T)$ is decidable,
Proof of Gödel’s Theorem

Proof

Let $Theorem(T)$ be the property that a statement $T$ is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that $P$ is a valid proof of $T$.

$Proof(P, T)$ is decidable, because we can check the proof line by line.
Proof of Gödel’s Theorem

Proof

Let $Theorem(T)$ be the property that a statement $T$ is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that $P$ is a valid proof of $T$.

$Proof(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in $\text{RE}$. 
Proof of Gödel’s Theorem

Proof

Let $Theorem(T)$ be the property that a statement $T$ is provable.

Then it can be written as:

$$Theorem(T) = \exists P : \text{Proof}(P, T),$$

where $\text{Proof}(P, T)$ is the property that $P$ is a valid proof of $T$.

$\text{Proof}(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in $\text{RE}$.

We assume that our formal system is powerful enough to talk about computation.
Proof of Gödel’s Theorem

Proof

Let $Theorem(T)$ be the property that a statement $T$ is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that $P$ is a valid proof of $T$.

$Proof(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in $\text{RE}$.

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like $\forall$ and $\exists$. 
Proof of Gödel’s Theorem

Proof

Let \( \text{Theorem}(T) \) be the property that a statement \( T \) is provable.

Then it can be written as:

\[
\text{Theorem}(T) = \exists P : \text{Proof}(P, T),
\]

where \( \text{Proof}(P, T) \) is the property that \( P \) is a valid proof of \( T \).

\( \text{Proof}(P, T) \) is decidable, because we can check the proof line by line.

Thus, the set of theorems is in \( \text{RE} \).

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like \( \forall \) and \( \exists \).

We assume that the theory can express statements like \( \text{Halts}(\Pi, x) \).
Proof of Gödel’s Theorem

Proof

Let \( \text{Theorem}(T) \) be the property that a statement \( T \) is provable.

Then it can be written as:

\[
\text{Theorem}(T) = \exists P : \text{Proof}(P, T),
\]

where \( \text{Proof}(P, T) \) is the property that \( P \) is a valid proof of \( T \).

\( \text{Proof}(P, T) \) is decidable, because we can check the proof line by line.

Thus, the set of theorems is in \( \text{RE} \).

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like \( \forall \) and \( \exists \).

We assume that the theory can express statements like \( \text{Halts}(\Pi, x) \).

We assume that the axioms of the theory are strong enough to derive each step of a computation from the previous one.
Proof of Gödel’s Theorem

We will get a truth about halting and non-halting programs that cannot be proved. If \( \Pi(x) \) halts on its \( t \)-th step, then its computation is a proof of this fact, about \( t \) lines long. Thus, if \( \text{Halts}(\Pi, x) \) is true, then it is provable.

What if \( \Pi \) does not hold? Assume that all statements of the form \( \text{Halts}(\Pi, x) \) are provable. Then we can solve the Halting Problem by doing two things in parallel: run \( \Pi(x) \) to see if it halts, and looking for the proof that it will not. Since the Halting Problem is undecidable, there must exist a statement of the form \( \text{Halts}(\Pi, x) \) that is not provable. In this case, neither \( \text{Halts}(\Pi, x) \) nor \( \neg \text{Halts}(\Pi, x) \) is a theorem. It is independent of the axioms.
Proof of Gödel’s Theorem

**Proof**

We will get a truth about halting and non-halting programs that cannot be proved. If \( \Pi(x) \) halts on its \( t \)-th step, then its computation is a proof of this fact, about \( t \) lines long.

Thus, if \( \text{Halts}(\Pi, x) \) is true, then it is provable.

What if \( \Pi \) does not hold? Assume that all statements of the form \( \text{Halts}(\Pi, x) \) are provable. Then we can solve the Halting Problem by doing two things in parallel: run \( \Pi(x) \) to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form \( \text{Halts}(\Pi, x) \) that is not provable. In this case, neither \( \text{Halts}(\Pi, x) \) nor \( \neg \text{Halts}(\Pi, x) \) is a theorem. It is independent of the axioms.
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step,
Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step, then its computation is a proof of this fact, about $t$ lines long.
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step, then its computation is a proof of this fact, about $t$ lines long.

Thus,

$$\text{if } Halts(\Pi, x) \text{ is true, then it is provable.}$$
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step, then its computation is a proof of this fact, about $t$ lines long.

Thus,

$$\text{if } \text{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if $\Pi$ does not hold?
We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step, then its computation is a proof of this fact, about $t$ lines long.

Thus,

if $\text{Halts}(\Pi, x)$ is true, then it is provable.

What if $\Pi$ does not hold? Assume that all statements of the form $\overline{\text{Halts}(\Pi, x)}$ are provable.
Proof of Gödel’s Theorem

**Proof**

We will get a truth about halting and non-halting programs that cannot be proved.

If \( \Pi(x) \) halts on its \( t \)-th step, then its computation is a proof of this fact, about \( t \) lines long.

Thus,

\[
\text{if } \text{Halts}(\Pi, x) \text{ is true, then it is provable.}
\]

What if \( \Pi \) does not hold? Assume that all statements of the form \( \overline{\text{Halts}}(\Pi, x) \) are provable.

Then we can solve the Halting Problem by doing two things in parallel:
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step, then its computation is a proof of this fact, about $t$ lines long.

Thus,

$$\text{if } \text{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if $\Pi$ does not hold? Assume that all statements of the form $\overline{\text{Halts}(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts,
Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If \( \Pi(x) \) halts on its \( t \)-th step, then its computation is a proof of this fact, about \( t \) lines long.

Thus,

\[
\text{if } \text{Halts}(\Pi, x) \text{ is true, then it is provable.}
\]

What if \( \Pi \) does not hold? Assume that all statements of the form \( \overline{\text{Halts}}(\Pi, x) \) are provable.

Then we can solve the Halting Problem by doing two things in parallel: run \( \Pi(x) \) to see if it halts, and looking for the proof that it will not.
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its $t$-th step, then its computation is a proof of this fact, about $t$ lines long.

Thus,

\[
\text{if } \text{Halts}(\Pi, x) \text{ is true, then it is provable.}
\]

What if $\Pi$ does not hold? Assume that all statements of the form $\overline{\text{Halts}}(\Pi, x)$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form $\overline{\text{Halts}}(\Pi, x)$ that is not provable.
Proof of Gödel’s Theorem

**Proof**

We will get a truth about halting and non-halting programs that cannot be proved.

If Π(χ) halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

\[ \text{if } \text{Halts}(\Pi, \chi) \text{ is true, then it is provable.} \]

What if Π does not hold? Assume that all statements of the form \( \overline{\text{Halts}}(\Pi, \chi) \) are provable.

Then we can solve the Halting Problem by doing two things in parallel: run Π(χ) to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form \( \text{Halts}(\Pi, \chi) \) that is not provable.

In this case, neither \( \text{Halts}(\Pi, \chi) \) nor \( \overline{\text{Halts}}(\Pi, \chi) \) is a theorem.
Proof of Gödel’s Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If \( \Pi(x) \) halts on its \( t \)-th step, then its computation is a proof of this fact, about \( t \) lines long.

Thus,

if \( \text{Halts}(\Pi, x) \) is true, then it is provable.

What if \( \Pi \) does not hold? Assume that all statements of the form \( \overline{\text{Halts}}(\Pi, x) \) are provable.

Then we can solve the Halting Problem by doing two things in parallel: run \( \Pi(x) \) to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form \( \overline{\text{Halts}}(\Pi, x) \) that is not provable.

In this case, neither \( \text{Halts}(\Pi, x) \) nor \( \overline{\text{Halts}}(\Pi, x) \) is a theorem. It is independent of the axioms.
Possible Remedy

Idea

How about adding $\text{Halts}(\Pi, x)$ to our list of axioms?

Then the fact that $\Pi$ does not hold on input $x$ becomes, trivially, a theorem of the system.

But then there will be another program $\Pi'$ and an input $x'$, such that $\text{Halts}(\Pi', x')$ is true, but not provable in the new system, and so on.

No finite set of axioms captures all the non-halting programs.

For any formal system, there will be some truth that it cannot prove.
Possible Remedy

Idea

How about adding $\text{Halts}(\Pi_x)$ to our list of axioms? Then the fact that $\Pi$ does not hold on input $x$ becomes, trivially, a theorem of the system. But then there will be another program $\Pi_{x'}$ and an input $x_{x'}$ such that $\text{Halts}(\Pi_{x'}, x_{x'})$ is true, but not provable in the new system, and so on. No finite set of axioms captures all the non-halting programs. For any formal system, there will be some truth that it cannot prove.
Idea

How about adding $\text{Halts}(\Pi, x)$ to our list of axioms?
Idea

How about adding $\text{Halts}(\Pi, x)$ to our list of axioms?

Then the fact that $\Pi$ does not hold on input $x$ becomes, trivially, a theorem of the system.
Possible Remedy

Idea

How about adding $\text{Halts}(\Pi, x)$ to our list of axioms?

Then the fact that $\Pi$ does not hold on input $x$ becomes, trivially, a theorem of the system.

But then there will be another program $\Pi'$ and an input $x'$, such that $\text{Halts}(\Pi', x')$ is true, but not provable in the new system, and so on.
Possible Remedy

Idea

How about adding $\text{Halts}(\Pi, x)$ to our list of axioms?

Then the fact that $\Pi$ does not hold on input $x$ becomes, trivially, a theorem of the system.

But then there will be another program $\Pi'$ and an input $x'$, such that $\text{Halts}(\Pi', x')$ is true, but not provable in the new system, and so on.

No finite set of axioms captures all the non-halting programs.
Possible Remedy

Idea

How about adding $\text{Halts}(\Pi, x)$ to our list of axioms?

Then the fact that $\Pi$ does not hold on input $x$ becomes, trivially, a theorem of the system.

But then there will be another program $\Pi'$ and an input $x'$, such that $\text{Halts}(\Pi', x')$ is true, but not provable in the new system, and so on.

No finite set of axioms captures all the non-halting programs.

For any formal system, there will be some truth that it cannot prove.
What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too. These simpler functions are defined in turn in terms of even simpler ones, and so on. With this we reach a set of basic functions, for which no further explanation is necessary. These basic functions form the atoms of computation. In terms of programming, they are the elementary operations that we can carry out in a single step.

The Grand Unified Theory of Computation
What is an algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too. These simpler functions are defined in turn in terms of even simpler ones, and so on. With this we reach a set of basic functions, for which no further explanation is necessary. These basic functions form the atoms of computation. In terms of programming, they are the elementary operations that we can carry out in a single step.
What is an algorithm?

How can we give a clear definition of the algorithm?
What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.
What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.
What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

With this we reach a set of basic functions, for which no further explanation is necessary.
What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

With this we reach a set of basic functions, for which no further explanation is necessary.

These basic functions form the atoms of computation.
Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

With this we reach a set of basic functions, for which no further explanation is necessary.

These basic functions form the atoms of computation.

In terms of programming, they are the elementary operations that we can carry out in a single step.
Basic functions

The constant 0 and the successor function

The first basic function is: \( 0(x) = 0 \).

The second basic function is: \( S(x) = x + 1 \).

Remark
Strictly speaking, in order to use \( x \) on the right-side we also need to include the identity function \( I(x) = x \).
Basic functions

The constant 0 and the successor function
Basic functions

The constant 0 and the successor function

The first basic function is:

\[ 0(x) = 0. \]
Basic functions

The constant 0 and the successor function

The first basic function is:

\[ 0(x) = 0. \]

The second basic function is:

\[ S(x) = x + 1. \]
Basic functions

The constant 0 and the successor function

The first basic function is:

\[ 0(x) = 0. \]

The second basic function is:

\[ S(x) = x + 1. \]

Remark

Strictly speaking, in order to use \( x \) on the right-side we also need to include the identity function \( I(x) = x \).
Basic functions

The constant 0 and the successor function

The first basic function is:

$$0(x) = 0.$$  

The second basic function is:

$$S(x) = x + 1.$$  

Remark

Strictly speaking, in order to use $x$ on the right-side we also need to include the identity function $I(x) = x.$
We need some schemes by which we can construct new functions from old ones.

**Composition**

If \( f \) and \( g \) are already defined, we can define a new function \( h = f \circ g \) by

\[
h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n))
\]

More generally, we allow functions to access each of their variables. For instance, if \( f(x_1, x_2) \), \( g(x_1, x_2) \), and \( m(x_3, x_1) \) are already defined, we can define

\[
h(x_1, x_2, x_3) = f(g(x_1, x_2), m(x_3, x_1))
\]

**Remark**

In terms of programming, composition lets us call previously defined functions as subroutines, using the output of one as the input of the other.
Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If $f$ and $g$ are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n)).$$

More generally, we allow functions to access each of their variables. For instance, if $f(x_1, x_2)$, $g(x_1, x_2)$, and $m(x_3, x_1)$ are already defined, we can define $h(x_1, x_2, x_3) = f(g(x_1, x_2), m(x_3, x_1)).$

Remark

In terms of programming, composition lets us call previously defined functions as subroutines, using the output of one as the input of the other.
Generating new functions

We need some schemes by which we can construct new functions from old ones.
Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition
Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If $f$ and $g$ are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n)).$$
Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If $f$ and $g$ are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n)).$$

More generally, we allow functions to access each of their variables.
Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If $f$ and $g$ are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n)).$$

More generally, we allow functions to access each of their variables. For instance, if $f(x_1, x_2), g(x_1, x_2)$ and $m(x_1, x_2)$ are already defined, we can define

$$h(x_1, x_2, x_3) = f(g(x_1, x_2), m(x_3, x_1)).$$
Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If \( f \) and \( g \) are already defined, we can define a new function \( h = f \circ g \) by

\[
h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n)).
\]

More generally, we allow functions to access each of their variables. For instance, if \( f(x_1, x_2), g(x_1, x_2) \) and \( m(x_1, x_2) \) are already defined, we can define

\[
h(x_1, x_2, x_3) = f(g(x_1, x_2), m(x_3, x_1)).
\]

Remark

*In terms of programming, composition lets us call previously defined functions as subroutines, using the output of one as the input of the other.*
Schemes

Babbage's Vision and Hilbert's Dream
Universality and Undecidability

Building Blocks: Recursive Functions

Schemes

Universal Functions

Primitive Recursion

If \( f(x_1, \ldots, x_n) \) and \( g(x_1, \ldots, x_n, y, z) \) are already defined, we can define a new function \( h(x_1, \ldots, x_n, y) \) as follows:

\[
\begin{align*}
    h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n), \\
    h(x_1, \ldots, x_n, y + 1) &= g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)).
\end{align*}
\]

Remark

In terms of programming, this corresponds to a \texttt{for} loop, when one iterates through the values of \( y \).
If \( f(\ldots, x_n \ldots) \) and \( g(\ldots, x_n, y, z \ldots) \) are already defined, we can define a new function \( h(\ldots, y \ldots) \) as follows:

\[
h(\ldots, 0 \ldots) = f(\ldots, \ldots),
\]

and

\[
h(\ldots, y + 1 \ldots) = g(\ldots, y, h(\ldots, y \ldots))
\]

Remark: In terms of programming, this corresponds to a for loop, when one iterates through the values of \( y \).
Schemes

Primitive recursion

If $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n, y, z)$ are already defined,
If $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n, y, z)$ are already defined, we can define a new function $h(x_1, \ldots, x_n, y)$ as follows:
If \( f(x_1, ..., x_n) \) and \( g(x_1, ..., x_n, y, z) \) are already defined, we can define a new function \( h(x_1, ..., x_n, y) \) as follows:

\[
    h(x_1, ..., x_n, 0) = f(x_1, ..., x_n),
\]

Remark

In terms of programming, this corresponds to a for loop, when one iterates through the values of \( y \).
### Primitive recursion

If $f(x_1, ..., x_n)$ and $g(x_1, ..., x_n, y, z)$ are already defined, we can define a new function $h(x_1, ..., x_n, y)$ as follows:

$$h(x_1, ..., x_n, 0) = f(x_1, ..., x_n), \text{ and } h(x_1, ..., x_n, y + 1) = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y)).$$
If \( f(x_1, ..., x_n) \) and \( g(x_1, ..., x_n, y, z) \) are already defined, we can define a new function \( h(x_1, ..., x_n, y) \) as follows:

\[
h(x_1, ..., x_n, 0) = f(x_1, ..., x_n), \quad \text{and} \quad h(x_1, ..., x_n, y + 1) = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y)).
\]

**Remark**

In terms of programming, this corresponds to a `for` loop, when one iterates through the values of \( y \).
Examples

The addition function is computable:

\[ \text{add}(x, 0) = x \]
\[ \text{add}(x, y + 1) = S(\text{add}(x, y)) \]

In standard language this will look as follows:

\[ x + 0 = x \]
\[ x + (y + 1) = (x + y) + 1 \]

The multiplication function is computable:

\[ \text{mult}(x, 0) = 0 \]
\[ \text{mult}(x, y + 1) = \text{add}(\text{mult}(x, y), x) \]
Examples

The addition function is computable

\[
\text{add}(x, 0) = x \\
\text{add}(x, y + 1) = S(\text{add}(x, y))
\]

In standard language this will look as follows:

\[
x + 0 = x \\
x + (y + 1) = (x + y) + 1
\]
The addition function is computable

\[ add(x, 0) = x \]
The addition function is computable

\[ add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)). \]
Examples

The addition function is computable

\[ \text{add}(x, 0) = x \] and \[ \text{add}(x, y + 1) = S(\text{add}(x, y)). \]

In standard language this will look as follows:
Examples

The addition function is computable

\[ add(x, 0) = x \] and \[ add(x, y + 1) = S(add(x, y)). \]

In standard language this will look as follows:

\[ x + 0 = x \]
The addition function is computable

\[ add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)). \]

In standard language this will look as follows:

\[ x + 0 = x \text{ and } x + (y + 1) = (x + y) + 1. \]
The addition function is computable

\[ \text{add}(x, 0) = x \text{ and } \text{add}(x, y + 1) = S(\text{add}(x, y)). \]

In standard language this will look as follows:

\[ x + 0 = x \text{ and } x + (y + 1) = (x + y) + 1. \]

The multiplication function is computable

\[ \text{mult}(x, 0) = 0 \text{ and } \text{mult}(x, y + 1) = \text{add}(\text{mult}(x, y), x). \]
Examples

The addition function is computable

\[ add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)). \]

In standard language this will look as follows:

\[ x + 0 = x \text{ and } x + (y + 1) = (x + y) + 1. \]

The multiplication function is computable

\[ mult(x, 0) = 0 \]
### Examples

#### The addition function is computable

\[
\text{add}(x, 0) = x \quad \text{and} \quad \text{add}(x, y + 1) = S(\text{add}(x, y)).
\]

In standard language this will look as follows:

\[
x + 0 = x \quad \text{and} \quad x + (y + 1) = (x + y) + 1.
\]

#### The multiplication function is computable

\[
\text{mult}(x, 0) = 0 \quad \text{and} \quad \text{mult}(x, y + 1) = \text{add}(\text{mult}(x, y), x).
\]
Primitive recursive functions

Definition
Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive. Addition, multiplication, subtraction, and even prime $(x)$ are primitive recursive.

Are all computable functions primitive recursive? One may wonder whether any computable function is primitive recursive? The answer is NO! Basic functions are defined everywhere. The schemes for construction of new functions do not change this property. The universal function, which is computable, is not defined everywhere. Hence it cannot be primitive recursive.
Primitive recursive functions

Definition

Functions that can be obtained from basic functions \( 0(x) \) and \( S(x) \) by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive. Addition, multiplication, subtraction, and even \( \text{prime}(x) \) are primitive recursive.

Are all computable functions primitive recursive? One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere. The schemes for construction of new functions do not change this property. The universal function, which is computable, is not defined everywhere. Hence it cannot be primitive recursive.
Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.
Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive
Definition
Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive
Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.
### Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

### Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.

### Are all computable functions primitive recursive?

The answer is **NO**! Basic functions are defined everywhere. The schemes for construction of new functions do not change this property. The universal function, which is computable, is not defined everywhere. Hence it cannot be primitive recursive.
Primitive recursive functions

Definition
Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive
Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?
One may wonder whether any computable function is primitive recursive?
### Primitive recursive functions

**Definition**

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

**Many functions are primitive recursive**

Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.

**Are all computable functions primitive recursive?**

One may wonder whether any computable function is primitive recursive? The answer is NO!
**Primitive recursive functions**

**Definition**
Functions that can be obtained from basic functions \(0(x)\) and \(S(x)\) by using composition and primitive recursion are called primitive recursive.

**Many functions are primitive recursive**
Addition, multiplication, subtraction, and even \(prime(x)\) are primitive recursive.

**Are all computable functions primitive recursive?**
One may wonder whether any computable function is primitive recursive? The answer is NO!
Basic functions are defined everywhere.
Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere.

The schemes for construction of new functions do not change this property.
Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere.

The schemes for construction of new functions do not change this property.

The universal function, which is computable, is not defined everywhere.
Primitive recursive functions

Definition
Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive
Addition, multiplication, subtraction, and even $\text{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?
One may wonder whether any computable function is primitive recursive? The answer is NO!
Basic functions are defined everywhere.
The schemes for construction of new functions do not change this property.
The universal function, which is computable, is not defined everywhere.
Hence it cannot be primitive recursive.
Explicit example

Ackerman's function

\[ A_1(x, y) = x + y, \]
\[ A_n(x, 0) = 1, \]
\[ A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0. \]

Small values of \( n \):

- \( A_2(x, y) = x \cdot y \),
- \( A_3(x, y) = x^y \).

Theorem

For any primitive recursive function \( f(y) \), there is an \( n \), such that \( f(y) < A_n(2, y) \) for all \( y \geq 3 \).
Explicit example

Ackerman’s function

\[ A_1(x, y) = x + y, \]
\[ A_n(x, 0) = 1, \]
\[ A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0. \]

Small values of \( n \) It can be shown that:
\[ A_2(x, y) = x \cdot y, \]
\[ A_3(x, y) = x^y. \]

Theorem For any primitive recursive function \( f(y) \), there is an \( n \), such that \( f(y) < A_n(2, y) \) for all \( y \geq 3 \).
Ackerman’s function

\[ A_1(x, y) = x + y, \]
Explicit example

Ackerman’s function

\[ A_1(x, y) = x + y, \ A_n(x, 0) = 1 \]
Ackerman’s function

\[ A_1(x, y) = x + y, \quad A_n(x, 0) = 1 \quad \text{and} \quad A_n(x, y) = A_{n-1}(x, A_n(x, y - 1)) \quad \text{if} \quad y > 0. \]
Explicit example

**Ackerman’s function**

\[ A_1(x, y) = x + y, \ A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0. \]

**Small values of \( n \)**
### Ackerman’s function

\[
A_1(x, y) = x + y, \quad A_n(x, 0) = 1 \quad \text{and} \quad A_n(x, y) = A_{n-1}(x, A_n(x, y - 1)) \quad \text{if} \quad y > 0.
\]

### Small values of \( n \)

It can be shown that:

\[
A_2(x, y) = x \cdot y,
\]
Explicit example

Ackerman’s function

\[ A_1(x, y) = x + y, \quad A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y - 1)) \text{ if } y > 0. \]

Small values of \( n \)

It can be shown that:

\[ A_2(x, y) = x \cdot y, \quad A_3(x, y) = x^y. \]
Explicit example

Ackerman’s function

\[ A_1(x, y) = x + y, \quad A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_{n}(x, y - 1)) \text{ if } y > 0. \]

Small values of \( n \)

It can be shown that:

\[ A_2(x, y) = x \cdot y, \quad A_3(x, y) = x^y. \]

Theorem
### Explicit example

**Ackerman’s function**

\[
A_1(x, y) = x + y, \quad A_n(x, 0) = 1 \quad \text{and} \quad A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \quad \text{if} \quad y > 0.
\]

**Small values of \( n \)**

It can be shown that:

\[
A_2(x, y) = x \cdot y, \quad A_3(x, y) = x^y.
\]

**Theorem**

*For any primitive recursive function \( f(y) \),*
Explicit example

**Ackerman’s function**

\[ A_1(x, y) = x + y, \ A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0. \]

**Small values of \( n \)**

It can be shown that:

\[ A_2(x, y) = x \cdot y, \ A_3(x, y) = x^y. \]

**Theorem**

*For any primitive recursive function \( f(y) \), there is an \( n \), such that*
Explicit example

**Ackerman’s function**

\[ A_1(x, y) = x + y, \ A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0. \]

**Small values of \( n \)**

It can be shown that:

\[ A_2(x, y) = x \cdot y, \ A_3(x, y) = x^y. \]

**Theorem**

*For any primitive recursive function \( f(y) \), there is an \( n \), such that*

\[ f(y) < A_n(2, y) \text{ for all } y \geq 3. \]
Definition
Given a function $f(x_1, \ldots, x_n, y)$, which is already defined, $\mu$-recursion lets us define a new function $h(x_1, \ldots, x_n)$ as follows:

$$h(x_1, \ldots, x_n) = \mu y f(x_1, \ldots, x_n, y) = \min \{ y : f(x_1, \ldots, x_n, y) = 0 \}.$$  

Remark
$h(x_1, \ldots, x_n)$ returns the smallest solution $y$ to the equation $f(x_1, \ldots, x_n, y) = 0$.

Remark
In programming, this corresponds to the `while` loop.
What is missing?

Definition

Given a function $f(x_1, \ldots, x_n, y)$, which is already defined, $\mu$-recursion lets us define a new function $h(x_1, \ldots, x_n)$ as follows:

$$h(x_1, \ldots, x_n) = \mu y f(x_1, \ldots, x_n, y) = \min \{ y : f(x_1, \ldots, x_n, y) = 0 \}.$$  

Remark

$h(x_1, \ldots, x_n)$ returns the smallest solution $y$ to the equation $f(x_1, \ldots, x_n, y) = 0$.

Remark

In programming, this corresponds to the while loop.
Definition

Given a function $f(x_1, \ldots, x_n, y)$, which is already defined,
Definition

Given a function \( f(x_1, \ldots, x_n, y) \), which is already defined, \( \mu \)-recursion lets us define a new function \( h(x_1, \ldots, x_n) \) as follows:

\[
h(x_1, \ldots, x_n) = \min \{ y : f(x_1, \ldots, x_n, y) = 0 \}.
\]

Remark

\( h(x_1, \ldots, x_n) \) returns the smallest solution \( y \) to the equation \( f(x_1, \ldots, x_n, y) = 0 \).

Remark

In programming, this corresponds to the `while` loop.
Definition

Given a function $f(x_1, \ldots, x_n, y)$, which is already defined, $\mu$-recursion lets us define a new function $h(x_1, \ldots, x_n)$ as follows:

$$h(x_1, \ldots, x_n) = \mu y f(x_1, \ldots, x_n, y)$$
Definition

Given a function \( f(x_1, \ldots, x_n, y) \), which is already defined, \( \mu \)-recursion lets us define a new function \( h(x_1, \ldots, x_n) \) as follows:

\[
h(x_1, \ldots, x_n) = \mu y_f(x_1, \ldots, x_n, y) = \min\{y : f(x_1, \ldots, x_n, y) = 0\}.
\]
Definition
Given a function $f(x_1, \ldots, x_n, y)$, which is already defined, $\mu$-recursion lets us define a new function $h(x_1, \ldots, x_n)$ as follows:

$$h(x_1, \ldots, x_n) = \mu_y f(x_1, \ldots, x_n, y) = \min\{y : f(x_1, \ldots, x_n, y) = 0\}.$$ 

Remark
$h(x_1, \ldots, x_n)$ returns the smallest solution $y$ to the equation $f(x_1, \ldots, x_n, y) = 0$. In programming, this corresponds to the `while` loop.
Definition

Given a function $f(x_1, \ldots, x_n, y)$, which is already defined, $\mu$-recursion lets us define a new function $h(x_1, \ldots, x_n)$ as follows:

$$h(x_1, \ldots, x_n) = \mu_y f(x_1, \ldots, x_n, y) = \min\{y : f(x_1, \ldots, x_n, y) = 0\}.$$ 

Remark

$h(x_1, \ldots, x_n)$ returns the smallest solution $y$ to the equation $f(x_1, \ldots, x_n, y) = 0$. 
What is missing?

**Definition**

Given a function $f(x_1, \ldots, x_n, y)$, which is already defined, $\mu$-recursion lets us define a new function $h(x_1, \ldots, x_n)$ as follows:

$$h(x_1, \ldots, x_n) = \mu_y f(x_1, \ldots, x_n, y) = \min\{y : f(x_1, \ldots, x_n, y) = 0\}.$$

**Remark**

$h(x_1, \ldots, x_n)$ returns the smallest solution $y$ to the equation $f(x_1, \ldots, x_n, y) = 0$.
Definition

Given a function \( f(x_1, \ldots, x_n, y) \), which is already defined, \( \mu \)-recursion lets us define a new function \( h(x_1, \ldots, x_n) \) as follows:

\[
h(x_1, \ldots, x_n) = \mu_y f(x_1, \ldots, x_n, y) = \min \{ y : f(x_1, \ldots, x_n, y) = 0 \}.
\]

Remark

\( h(x_1, \ldots, x_n) \) returns the smallest solution \( y \) to the equation \( f(x_1, \ldots, x_n, y) = 0 \).

Remark

In programming, this corresponds to the **while** loop.
Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and $\mu$-recursion are called partial recursive.

Definition

Partial recursive functions that are defined everywhere are called total recursive.

Three types of recursion

Primitive recursive $\subseteq$ Total recursive $\subseteq$ Partial recursive.
Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and $\mu$-recursion are called partial recursive. Partial recursive functions that are defined everywhere are called total recursive.

Three types of recursion:

- Primitive recursive
- Total recursive
- Partial recursive

The relation between these types is:

$\text{primitive recursive} \subset \text{total recursive} \subset \text{partial recursive}$
Partial Recursive Functions

Definition

Functions that can be obtained from basic functions 0(x) and S(x) by using composition, primitive recursion and μ-recursion are called partial recursive.
Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and $\mu$-recursion are called partial recursive.

Definition

Partial recursive functions that are defined everywhere are called total recursive.

Three types of recursion

primitive recursive $\subset$ total recursive $\subset$ partial recursive.
Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and $\mu$-recursion are called partial recursive.

Definition

Partial recursive functions that are defined everywhere are called total recursive.
Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and $\mu$-recursion are called partial recursive.

Definition

Partial recursive functions that are defined everywhere are called total recursive.

Three types of recursion

- primitive recursive $\subseteq$ total recursive $\subseteq$ partial recursive.
Kleene’s Normal Form Theorem

Theorem

There exist primitive recursive functions \( f \) and \( g \), such that for every partial recursive function \( h \) there exists \( p \), such that

\[
h(x_1, \ldots, x_n) = g(x_1, \ldots, x_n, \mu y f(p, x_1, \ldots, x_n, y)).
\]

Remark

Any partial recursive function can be written with a single use of \( \mu \)-recursion.
Kleene’s Normal Form Theorem

**Number of \( \mu \)-recursions**

What is the maximum number of \( \mu \)-recursions that one needs to get an arbitrary partial recursive function?

**Theorem**

There exist primitive recursive functions \( f \) and \( g \), such that for every partial recursive function \( h \) there exists \( p \), such that

\[
h(x_1, \ldots, x_n) = g(x_1, \ldots, x_n, \mu y f(p, x_1, \ldots, x_n, y)).
\]

**Remark**

Any partial recursive function can be written with a single use of \( \mu \)-recursion.
Kleene’s Normal Form Theorem

Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?
### Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?

### Theorem

There exist primitive recursive functions $f$ and $g$, such that for every partial recursive function $h$, there exists $p$, such that

$$h(x_1, \ldots, x_n) = g(x_1, \ldots, x_n, \mu y f(p, x_1, \ldots, x_n, y)).$$

Remark

Any partial recursive function can be written with a single use of $\mu$-recursion.
Kleene’s Normal Form Theorem

Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?

Theorem

*There exist primitive recursive functions* $f$ and $g$, 
### Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?

### Theorem

*There exist primitive recursive functions $f$ and $g$, such that for every partial recursive function $h$*
Kleene’s Normal Form Theorem

Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions $f$ and $g$, such that for every partial recursive function $h$ there exists $p$, such that
Kleene’s Normal Form Theorem

Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions $f$ and $g$, such that for every partial recursive function $h$ there exists $p$, such that

$$h(x_1, \ldots, x_n) = g(x_1, \ldots, x_n, \mu_y f(p, x_1, \ldots, x_n, y)).$$
Kleene’s Normal Form Theorem

Number of $\mu$-recursions

What is the maximum number of $\mu$-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions $f$ and $g$, such that for every partial recursive function $h$ there exists $p$, such that

$$h(x_1, \ldots, x_n) = g(x_1, \ldots, x_n, \mu y f(p, x_1, \ldots, x_n, y)).$$

Remark

Any partial recursive function can be written with a single use of $\mu$-recursion.
References

Books

References

Books

References

Books