Computability Theory

Barbara Csima

May 6, 2026

EMAIL: csima@uwaterloo.ca
Department of Pure Mathematics
University of Waterloo

Intuitive Computability

Intuitively, we would say a set is computable if we could program a computer (with a finite program), such that running the program on any reasonable input would halt after finitely many steps and say "yes" if the input was in the set, and "no" otherwise.

Intuitive Computability

Intuitively, we would say a set is computable if we could program a computer (with a finite program), such that running the program on any reasonable input would halt after finitely many steps and say "yes" if the input was in the set, and "no" otherwise.

Example

The set of names of people in this room is computable.

Intuitive Computability

Intuitively, we would say a set is computable if we could program a computer (with a finite program), such that running the program on any reasonable input would halt after finitely many steps and say "yes" if the input was in the set, and "no" otherwise.

Example

The set of names of people in this room is computable.

Example

The set of names of descendants of people in this room may not be computable.

Computability

A Turing machine is a computer which uses finite programs, and takes as input natural numbers. A program on given input may either output a natural number, or fail to halt.

Computability

A Turing machine is a computer which uses finite programs, and takes as input natural numbers. A program on given input may either output a natural number, or fail to halt.

Definition

To each program, we can then associate a partial function from $\mathbb{N}$ to $\mathbb{N}$. Such a function is called partial computable.

Computability

A Turing machine is a computer which uses finite programs, and takes as input natural numbers. A program on given input may either output a natural number, or fail to halt.

Definition

To each program, we can then associate a partial function from $\mathbb{N}$ to $\mathbb{N}$. Such a function is called partial computable.

Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is computable if it is partial computable and has a value on every $n \in \mathbb{N}$.

Computability

Definition

A set $A \subseteq \mathbb{N}$ is computable if its characteristic function is computable.

Definition

A set $A \subseteq \mathbb{N}$ is computable if its characteristic function is computable.

That is, A is computable if there is a program that on every input n, has output 1 if $n \in A$, and output 0 if $n \notin A$.

Why use partial computable functions?

Note that there is no effective list of all the computable functions.

Why use partial computable functions?

Note that there is no effective list of all the computable functions. That is, there is no computable way of listing out all the computable functions.

Why use partial computable functions?

Note that there is no effective list of all the computable functions. That is, there is no computable way of listing out all the computable functions.

Indeed, suppose $f_0, f_1, f_2, \ldots$ were an effective listing of all computable functions.

Why use partial computable functions?

Note that there is no effective list of all the computable functions. That is, there is no computable way of listing out all the computable functions.

Indeed, suppose $f_0, f_1, f_2, \ldots$ were an effective listing of all computable functions. Then the function $g(e) = f_e(e) + 1$ would also be computable, but not appear on the list, a contradiction.

Why use partial computable functions?

Note that there is no effective list of all the computable functions. That is, there is no computable way of listing out all the computable functions.

Indeed, suppose $f_0, f_1, f_2, \ldots$ were an effective listing of all computable functions. Then the function $g(e) = f_e(e) + 1$ would also be computable, but not appear on the list, a contradiction.

Since Turing machines use finite programs, all possible Turing programs can be listed.

Why use partial computable functions?

Note that there is no effective list of all the computable functions. That is, there is no computable way of listing out all the computable functions.

Indeed, suppose $f_0, f_1, f_2, \ldots$ were an effective listing of all computable functions. Then the function $g(e) = f_e(e) + 1$ would also be computable, but not appear on the list, a contradiction.

Since Turing machines use finite programs, all possible Turing programs can be listed. Hence, we have an effective listing of all partial computable functions.

Why use partial computable functions?

Note that there is no effective list of all the computable functions. That is, there is no computable way of listing out all the computable functions.

Indeed, suppose $f_0, f_1, f_2, \ldots$ were an effective listing of all computable functions. Then the function $g(e) = f_e(e) + 1$ would also be computable, but not appear on the list, a contradiction.

Since Turing machines use finite programs, all possible Turing programs can be listed. Hence, we have an effective listing of all partial computable functions. Let $\varphi_0, \varphi_1, \varphi_2, \ldots$ be such a listing.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The set $\emptyset'$ is not computable.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The set $\emptyset'$ is not computable. Indeed, consider the function

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in \emptyset', \\ 0 & \text{if } x \notin \emptyset'. \end{cases} \tag{1}$$

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The set $\emptyset'$ is not computable. Indeed, consider the function

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in \emptyset', \\ 0 & \text{if } x \notin \emptyset'. \end{cases} \tag{1}$$

If $\emptyset'$ were computable, then g would be computable.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The set $\emptyset'$ is not computable. Indeed, consider the function

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in \emptyset', \\ 0 & \text{if } x \notin \emptyset'. \end{cases} \tag{1}$$

If $\emptyset'$ were computable, then g would be computable. But then there would be some e such that $g = \varphi_e$.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The set $\emptyset'$ is not computable. Indeed, consider the function

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in \emptyset', \\ 0 & \text{if } x \notin \emptyset'. \end{cases} \tag{1}$$

If $\emptyset'$ were computable, then g would be computable. But then there would be some e such that $g = \varphi_e$. Since g is computable, we must have $e \in \emptyset'$.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e) \downarrow\}$.

The set $\emptyset'$ is not computable. Indeed, consider the function

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in \emptyset', \\ 0 & \text{if } x \notin \emptyset'. \end{cases} \tag{1}$$

If $\emptyset'$ were computable, then g would be computable. But then there would be some e such that $g = \varphi_e$. Since g is computable, we must have $e \in \emptyset'$. But then $g(e) = \varphi_e(e) + 1 = g(e) + 1$, a contradiction.

The Halting Problem

The halting problem is to decide whether a given program halts on given input.

Definition

Let $\emptyset' = \{e \in \mathbb{N} \mid \varphi_e(e)\downarrow\}$.

The set $\emptyset'$ is not computable. Indeed, consider the function

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in \emptyset', \\ 0 & \text{if } x \notin \emptyset'. \end{cases} \tag{1}$$

If $\emptyset'$ were computable, then g would be computable. But then there would be some e such that $g = \varphi_e$. Since g is computable, we must have $e \in \emptyset'$. But then $g(e) = \varphi_e(e) + 1 = g(e) + 1$, a contradiction.

Hence there is no decision procedure to decide the halting problem.

Relative Computability

Definition

For sets A and B, we say A is Turing reducible to B, $A \leq_T B$, if there is a program which computes A, using B as an oracle.

Relative Computability

Definition

For sets A and B, we say A is Turing reducible to B, $A \leq_T B$, if there is a program which computes A, using B as an oracle.

We say $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$. A Turing degree is an equivalence class under $\equiv_T$.

Relative Computability

Definition

For sets A and B, we say A is Turing reducible to B, $A \leq_T B$, if there is a program which computes A, using B as an oracle.

We say $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$. A Turing degree is an equivalence class under $\equiv_T$.

Definition

We let $\mathbf{0}$ denote the collection of computable sets. This is the least Turing degree.

Computably Enumerable sets

Definition

A set A is computably enumerable (c.e.) if it is the domain of a partial computable function.

Computably Enumerable sets

Definition

A set A is computably enumerable (c.e.) if it is the domain of a partial computable function.

Equivalently, a set A is computably enumerable if there is some computer that enumerates the members of A.

Computably Enumerable sets

Definition

A set A is computably enumerable (c.e.) if it is the domain of a partial computable function.

Equivalently, a set A is computably enumerable if there is some computer that enumerates the members of A.

Example

The halting set, $\emptyset'$, is computably enumerable.

Computably Enumerable sets

Definition

A set A is computably enumerable (c.e.) if it is the domain of a partial computable function.

Equivalently, a set A is computably enumerable if there is some computer that enumerates the members of A.

Example

The halting set, $\emptyset'$, is computably enumerable.

In fact, $\emptyset'$ is the most complicated c.e. set, in the sense that any other c.e. set is Turing reducible to it.

Breaking into Requirements

Priority arguments are a common proof technique in Computability Theory. A theorem is broken down to being equivalent to a list of requirements. These requirements are given a priority order, and a strategy is devised to meet all the requirements, making use of the priority order.

Breaking into Requirements

Priority arguments are a common proof technique in Computability Theory. A theorem is broken down to being equivalent to a list of requirements. These requirements are given a priority order, and a strategy is devised to meet all the requirements, making use of the priority order.

The property we just considered, that we can nicely list all the partial computable functions, but not all computable ones, is a common theme in Computability Theory, and is why the organization into requirements with priority works so nicely.

An easy example with requirements

In Computable Structure Theory, we often ask whether effective versions of classical theorems still hold.

An easy example with requirements

In Computable Structure Theory, we often ask whether effective versions of classical theorems still hold.

Theorem

Every infinite binary branching tree has an infinite path.

An easy example with requirements

In Computable Structure Theory, we often ask whether effective versions of classical theorems still hold.

Theorem

Every infinite binary branching tree has an infinite path.

Theorem

There exists an infinite computable binary branching tree with no computable path (paths are infinite).

An easy argument with requirements

Theorem

There exists an infinite computable binary branching tree with no computable path (paths are infinite).

An easy argument with requirements

Theorem

There exists an infinite computable binary branching tree with no computable path (paths are infinite).

We meet for all $e \in \omega$ the requirement:

An easy argument with requirements

Theorem

There exists an infinite computable binary branching tree with no computable path (paths are infinite).

We meet for all $e \in \omega$ the requirement:

R_e: If φ_e is a total computable function, then φ_e does not code a path on the tree.

An easy argument with requirements

Theorem

There exists an infinite computable binary branching tree with no computable path (paths are infinite).

We meet for all $e \in \omega$ the requirement:

R_e: If φ_e is a total computable function, then φ_e does not code a path on the tree.

We build the computable tree by stages, always extending previous branches. However, if at some stage s, $\varphi_{e,s}(e) = 0$ or 1, we no longer extend any branches of the tree that agree with this.

An easy argument with requirements

This example was fairly straightforward. Each requirement only "acted" once. The actions of the requirements were disjoint, so that the tree constructed was infinite, while it was easy to see that each requirement was met.

$\emptyset'$ can do it

Theorem

Every infinite computable binary branching tree has a $\emptyset'$-computable path (paths are infinite).

$\emptyset'$ can do it

Theorem

Every infinite computable binary branching tree has a $\emptyset'$-computable path (paths are infinite).

Let T be an infinite computable binary branching tree. Use a $\emptyset'$-oracle to build a path through T by stages as follows:

$\emptyset'$ can do it

Theorem

Every infinite computable binary branching tree has a $\emptyset'$-computable path (paths are infinite).

Let T be an infinite computable binary branching tree. Use a $\emptyset'$-oracle to build a path through T by stages as follows:

Stage 0: Let σ_0 be the empty string.

$\emptyset'$ can do it

Theorem

Every infinite computable binary branching tree has a $\emptyset'$-computable path (paths are infinite).

Let T be an infinite computable binary branching tree. Use a $\emptyset'$-oracle to build a path through T by stages as follows:

Stage 0: Let σ_0 be the empty string.

Stage $s+1$: Consider the question
$(\exists n)(\forall \tau \in 2^n)[\tau \succ \sigma_s 0 \rightarrow \tau \notin T]$?

$\emptyset'$ can do it

Theorem

Every infinite computable binary branching tree has a $\emptyset'$-computable path (paths are infinite).

Let T be an infinite computable binary branching tree. Use a $\emptyset'$-oracle to build a path through T by stages as follows:

Stage 0: Let σ_0 be the empty string.

Stage $s+1$: Consider the question $(\exists n)(\forall \tau \in 2^n)[\tau \succ \sigma_s 0 \rightarrow \tau \notin T]$? This is a uniformly (in s) $\emptyset'$-computable question.

$\emptyset'$ can do it

Theorem

Every infinite computable binary branching tree has a $\emptyset'$-computable path (paths are infinite).

Let T be an infinite computable binary branching tree. Use a $\emptyset'$-oracle to build a path through T by stages as follows:

Stage 0: Let σ_0 be the empty string.

Stage $s+1$: Consider the question $(\exists n)(\forall \tau \in 2^n)[\tau \succ \sigma_s 0 \rightarrow \tau \notin T]$? This is a uniformly (in s) $\emptyset'$-computable question. If the answer is "yes", set $\sigma_{s+1} = \sigma_s 1$. If the answer is "no", set $\sigma_{s+1} = \sigma_s 0$.

The first priority argument

In the early days of the subject, a big question (Post's Problem - 1944) was whether there were any non-computable c.e. sets that were not Turing equivalent to the halting set.

The first priority argument

In the early days of the subject, a big question (Post's Problem - 1944) was whether there were any non-computable c.e. sets that were not Turing equivalent to the halting set.

The solution, from Friedberg (1957) and Muchnik (1956), was to construct a pair of Turing incomparable c.e. sets. They each used a priority argument to construct c.e. sets A and B, meeting for each $e \in \mathbb{N}$ the requirements:

The first priority argument

In the early days of the subject, a big question (Post's Problem - 1944) was whether there were any non-computable c.e. sets that were not Turing equivalent to the halting set.

The solution, from Friedberg (1957) and Muchnik (1956), was to construct a pair of Turing incomparable c.e. sets. They each used a priority argument to construct c.e. sets A and B, meeting for each $e \in \mathbb{N}$ the requirements:

$R_e : \exists x_e \, \Phi_e^A(x_e) \neq B(x_e)$

$S_e : \exists y_e \, \Phi_e^B(y_e) \neq A(y_e)$

The first priority argument

To do a priority argument, one needs a plan of how to meet a single requirement in such a way as to also allow the plans for the other requirements to succeed.

The first priority argument

To do a priority argument, one needs a plan of how to meet a single requirement in such a way as to also allow the plans for the other requirements to succeed.

Here is the plan for a single requirement of the form $R_e : \exists x_e \, \Phi_e^A(x_e) \neq B(x_e)$.

The first priority argument

To do a priority argument, one needs a plan of how to meet a single requirement in such a way as to also allow the plans for the other requirements to succeed.

Here is the plan for a single requirement of the form $R_e : \exists x_e \, \Phi_e^A(x_e) \neq B(x_e)$.

Choose some x_e that is not enumerated yet into B, so $B(x_e) = 0$. Promise not to enumerate x_e into B unless something happens. Wait until there is a time when we see that $\Phi_e^A(x_e) \downarrow = 0$ (using the current approximation to A). If this never happens, then R_e is met. If at some stage $\Phi_e^A(x_e) \downarrow = 0$, then enumerate x_e into B. So now $B(x_e) = 1 \neq 0 = \Phi_e^A(x_e)$.

The first priority argument

To do a priority argument, one needs a plan of how to meet a single requirement in such a way as to also allow the plans for the other requirements to succeed.

Here is the plan for a single requirement of the form $R_e : \exists x_e \, \Phi_e^A(x_e) \neq B(x_e)$.

Choose some x_e that is not enumerated yet into B, so $B(x_e) = 0$. Promise not to enumerate x_e into B unless something happens. Wait until there is a time when we see that $\Phi_e^A(x_e) \downarrow = 0$ (using the current approximation to A). If this never happens, then R_e is met. If at some stage $\Phi_e^A(x_e) \downarrow = 0$, then enumerate x_e into B. So now $B(x_e) = 1 \neq 0 = \Phi_e^A(x_e)$.

What could go wrong? Nothing for just one R_e, but try to do the analogue for some S_e, and you might need to enumerate into A, which means that you may have had the wrong value for $\Phi_e^A(x_e)$.

Finite injury priority argument

The solution to the possible problem is to order the requirements according to a priority order, say $R_0 < S_0 < R_1 < S_1 < \cdots$, where appearing earlier in the order corresponds to having higher priority.

Finite injury priority argument

The solution to the possible problem is to order the requirements according to a priority order, say $R_0 < S_0 < R_1 < S_1 < \cdots$, where appearing earlier in the order corresponds to having higher priority.

When the procedure begins, the sets A and B are empty, and one can begin with $x_e = y_e = e$. Now if at some stage, $\Phi_e^A(x_e) \downarrow = 0$ for the first time, then look at the *use* of the computation (the part of the oracle consulted), and *reset* the values of y_i for $i \geq e$ to be larger than that use.

Finite injury priority argument

The solution to the possible problem is to order the requirements according to a priority order, say $R_0 < S_0 < R_1 < S_1 < \cdots$, where appearing earlier in the order corresponds to having higher priority.

When the procedure begins, the sets A and B are empty, and one can begin with $x_e = y_e = e$. Now if at some stage, $\Phi_e^A(x_e) \downarrow = 0$ for the first time, then look at the *use* of the computation (the part of the oracle consulted), and *reset* the values of y_i for $i \geq e$ to be larger than that use.

This means that only y_i with $i < e$ will be allowed to be enumerated into A below the use. If that happens it will *injure* the requirement R_e. But this will happen at most finitely often, so each requirement is eventually met.

Jump Operator

Definition

For $A \subset \mathbb{N}$, let $A' = \{e \in \mathbb{N} \mid \varphi_e^A(e) \downarrow\}$. We call this the jump of A.

Note that if $A \leq_T \emptyset'$, then $\emptyset' \leq_T A' \leq_T \emptyset''$.

Low and High

Definition

A set A is low if $A' \equiv_T \emptyset'$.

Low and High

Definition

A set A is low if $A' \equiv_T \emptyset'$.

Definition

A set $A \leq_T \emptyset'$ is high if $A' \equiv_T \emptyset''$.

Low and High

Definition

A set A is low if $A' \equiv_T \emptyset'$.

Definition

A set $A \leq_T \emptyset'$ is high if $A' \equiv_T \emptyset''$.

We think of low sets as close to computable, and high sets as close to the halting problem.