Summary last week

- divide and conquer algorithms, e.g. mergesort
- have asymptotic complexities given by recurrences $T(n) = \ldots T(< n) \ldots$
- may find a closed-form solution for a recurrence by:
- self-substitution and looking for pattern; or
- guessing and verifying; or
- generating functions (not this course); or
- master theorem: $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ if $n = b^k$ for $k > 0$, otherwise c:

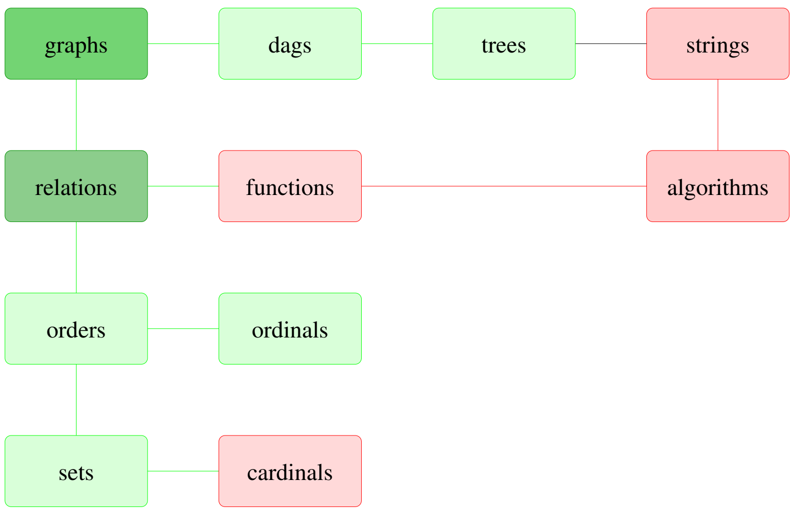
$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^s \\ \Theta(n^s \log n) & \text{if } a = b^s \\ \Theta(n^s) & \text{if } a < b^s \end{cases}$$

for T increasing, $a \geqslant 1$, $b > 1$, $c > 0$, and $f \in \Theta(n^s)$ with $s \geqslant 0$.

Course themes

- directed and undirected graphs
- relations and functions
- orders and induction
- trees and dags
- finite and infinite counting
- elementary number theory
- Turing machines, algorithms, and complexity
- decidable and undecidable problem

Discrete structures



Limitations of algorithms (recall from 3rd lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;

Limitations of algorithms (recall from 3rd lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;
- No algorithms for checking interesting properties of programs (TMs) themselves; termination (halting problem), reachability (unreachable code), ... No interesting property of programs can be programmed.

Limitations of algorithms (recall from 3rd lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;
- No algorithms for checking interesting properties of programs (TMs) themselves; termination (halting problem), reachability (unreachable code), ... No interesting property of programs can be programmed.

- No algorithm for checking whether a formula in first-order logic is universally valid (Entscheidungsproblem).

Limitations of algorithms (recall from 3rd lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;
- No algorithms for checking interesting properties of programs (TMs) themselves; termination (halting problem), reachability (unreachable code), ... No interesting property of programs can be programmed.
- No algorithm for checking whether a formula in first-order logic is universally valid (Entscheidungsproblem).

- No algorithm for checking whether Diophantine equations have a solution (Hilbert's 10th problem).

Limitations of algorithms (recall from 3rd lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;
- No algorithms for checking interesting properties of programs (TMs) themselves; termination (halting problem), reachability (unreachable code), ... No interesting property of programs can be programmed.
- No algorithm for checking whether a formula in first-order logic is universally valid (Entscheidungsproblem).
- No algorithm for checking whether Diophantine equations have a solution (Hilbert's 10th problem).

- ...

Limitations of algorithms (recall from 3rd lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;
- No algorithms for checking interesting properties of programs (TMs) themselves; termination (halting problem), reachability (unreachable code), ... No interesting property of programs can be programmed.
- No algorithm for checking whether a formula in first-order logic is universally valid (Entscheidungsproblem).
- No algorithm for checking whether Diophantine equations have a solution (Hilbert's 10th problem).
- ...

Remark

These limitations will be addressed in the last few weeks of course (i.e. now)

Function defined by a TM (recall from 3rd lecture)

Definition

a TM M

- accepts $x \in \Sigma^*$, if $\exists\, y, n$:

$$(s, \vdash x \sqcup^\infty, 0) \xrightarrow[M]{*} (t, y, n)$$

- rejects $x \in \Sigma^*$, if $\exists\, y, n$:

$$(s, \vdash x \sqcup^\infty, 0) \xrightarrow[M]{*} (r, y, n)$$

- halt on input x, if x is accepted or rejected
- does not halt on input x, if x is neither accepted nor rejected
- is total, if M halts on all inputs

Function defined by a TM (recall from 3rd lecture)

Definition

a TM M

- accepts $x \in \Sigma^*$, if $\exists\, y, n$:

$$(s, \vdash x \sqcup^\infty, 0) \xrightarrow[M]{*} (t, y, n)$$

- rejects $x \in \Sigma^*$, if $\exists\, y, n$:

$$(s, \vdash x \sqcup^\infty, 0) \xrightarrow[M]{*} (r, y, n)$$

- halt on input x, if x is accepted or rejected
- does not halt on input x, if x is neither accepted nor rejected
- is total, if M halts on all inputs

Definition

A function $f : A \to B$ is defined by a TM M for every $x \in A$, M accepts input x with $f(y)$ on the tape (and does not halt or rejects on inputs $x \notin A$).

Computable functions

Idea of computability

$f : \mathbb{N} \to \mathbb{N}$ computable if there is an effective procedure to compute $f(n)$ for input n

Computable functions

Idea of computability

$f : \mathbb{N} \to \mathbb{N}$ computable if there is an effective procedure to compute $f(n)$ for input n

Definition (computability via TM)

$f : \mathbb{N} \to \mathbb{N}$ computable if it can be defined by a TM

Examples of computable functions

remark

computability equivalently defined via models of computation: μ-recursive functions, λ-calculus, register machines, term rewriting, ...

Examples of computable functions

remark

computability equivalently defined via models of computation: μ-recursive functions, λ-calculus, register machines, term rewriting, ...

Example

- any function programmable in some programming language
  square root, counting the number of 3s, compression, etc.
- effective $\neq$ efficient
  factorial, Ackermann function (complexity far worse than exponential)

Examples of computable functions

remark

computability equivalently defined via models of computation: μ-recursive functions, λ-calculus, register machines, term rewriting, ...

Example

- any function programmable in some programming language
  square root, counting the number of 3s, compression, etc.
- effective $\neq$ efficient
  factorial, Ackermann function (complexity far worse than exponential)
- unbounded search functions
  the least number that has property P (need not exist)

Examples of computable functions

remark

computability equivalently defined via models of computation: μ-recursive functions, λ-calculus, register machines, term rewriting, ...

Example

- any function programmable in some programming language
  square root, counting the number of 3s, compression, etc.
- effective $\neq$ efficient
  factorial, Ackermann function (complexity far worse than exponential)
- unbounded search functions
  the least number that has property P (need not exist)
- functions defined by finite cases
  $f(n) = n$ if n odd, otherwise n^2

Limits of computability

Lemma

there exist functions that are not computable (more functions than programs)

Proof.

Limits of computability

Lemma

there exist functions that are not computable

Proof.

- any program may be encoded by a finite bit-string

Limits of computability

Lemma

there exist functions that are not computable

Proof.

- any program may be encoded by a finite bit-string
- $\Rightarrow$ there are countably many programs; (recall $\bigcup_i \{0,1\}^i$ is countable)

Limits of computability

Lemma

there exist functions that are not computable

Proof.

- any program may be encoded by a finite bit-string
- $\Rightarrow$ there are countably many programs; (recall $\bigcup_i \{0,1\}^i$ is countable)
- there are uncountably many functions $\mathbb{N} \to \mathbb{N}$ (recall $\mathbb{N} \to \{0,1\}$ is uncountable)

Limits of computability

Lemma

there exist functions that are not computable

Proof.

- any program may be encoded by a finite bit-string
- $\Rightarrow$ there are countably many programs; (recall $\bigcup_i \{0,1\}^i$ is countable)
- there are uncountably many functions $\mathbb{N} \to \mathbb{N}$ (recall $\mathbb{N} \to \{0,1\}$ is uncountable)
- $\Rightarrow$ some function $\mathbb{N} \to \mathbb{N}$ is not computable ■

Theorem

concrete non-computable functions (diagonalise away from TM behaviours)

Limits of computability

Lemma

there exist functions that are not computable

Proof.

- any program may be encoded by a finite bit-string
- $\Rightarrow$ there are countably many programs; (recall $\bigcup_i \{0,1\}^i$ is countable)
- there are uncountably many functions $\mathbb{N} \to \mathbb{N}$ (recall $\mathbb{N} \to \{0,1\}$ is uncountable)
- $\Rightarrow$ some function $\mathbb{N} \to \mathbb{N}$ is not computable ■

Theorem

concrete non-computable functions

rest of this lecture, details of the above: coding, diagonalising way

Recursive/recursively enumerable languages

Definition

A language L (or, more generally, a set) is

- recursively enumerable, if there exists a TM M such that $L = \mathrm{L}(M)$
  i.e. L is the set of strings accepted by M
- recursive, if there exists a total TM M, such that $L = \mathrm{L}(M)$
  i.e. M is required to halt (accept or reject) on all strings

Recursive/recursively enumerable languages

Definition

A language L (or, more generally, a set) is

- recursively enumerable, if there exists a TM M such that $L = \mathrm{L}(M)$
  i.e. L is the set of strings accepted by M
- recursive, if there exists a total TM M, such that $L = \mathrm{L}(M)$
  i.e. M is required to halt (accept or reject) on all strings

Church–Turing-Thesis

Every problem that is algorithmically solvable is solvable by a Turing machine

Recursive/recursively enumerable languages

Definition

A language L (or, more generally, a set) is

- recursively enumerable, if there exists a TM M such that $L = \mathrm{L}(M)$
  i.e. L is the set of strings accepted by M
- recursive, if there exists a total TM M, such that $L = \mathrm{L}(M)$
  i.e. M is required to halt (accept or reject) on all strings

Church–Turing-Thesis

Every problem that is algorithmically solvable is solvable by a Turing machine

Computable function vs. recursive sets

Partial function $f : \mathbb{N} \to \mathbb{N}$ is computable iff $L_f = \{x\#f(x) \mid x \in \mathbb{N}\}$ is recursively enumerable. Total f is computable iff L_f is recursive.

Theorem

Let $L \subseteq \Sigma^*$ be a recursive language over some alphabet Σ; then $\sim L$ is recursive.

Theorem

Let $L \subseteq \Sigma^*$ be a recursive language over some alphabet Σ; then $\sim L$ is recursive.

Proof.

Because L is recursive, there exists a total TM M such that $L = \mathrm{L}(M)$. Let the TM M' be obtained from M by exchanging its accepting and rejecting states. Because M is total, so is M'. Therefore, M' accepts a word iff M rejects it, hence $\sim L = \mathrm{L}(M')$, i.e. $\sim L$ is recursive. ∎

Theorem

Let $L \subseteq \Sigma^*$ be a recursive language over some alphabet Σ; then $\sim L$ is recursive.

Proof.

Because L is recursive, there exists a total TM M such that $L = \mathrm{L}(M)$. Let the TM M' be obtained from M by exchanging its accepting and rejecting states. Because M is total, so is M'. Therefore, M' accepts a word iff M rejects it, hence $\sim L = \mathrm{L}(M')$, i.e. $\sim L$ is recursive. ∎

Theorem

Every recursive set is recursively enumerable, but not every recursively enumerable set is recursive.

Theorem

Let $L \subseteq \Sigma^*$ be a recursive language over some alphabet Σ; then $\sim L$ is recursive.

Proof.

Because L is recursive, there exists a total TM M such that $L = \mathrm{L}(M)$. Let the TM M' be obtained from M by exchanging its accepting and rejecting states. Because M is total, so is M'. Therefore, M' accepts a word iff M rejects it, hence $\sim L = \mathrm{L}(M')$, i.e. $\sim L$ is recursive. ∎

Theorem

Every recursive set is recursively enumerable, but not every recursively enumerable set is recursive.

Proof.

The first part of the theorem follows from the definitions; the second part we will show later ∎

Theorem

If both L and $\sim L$ are recursively enumerable, then L is recursive.

Theorem

If both L and $\sim L$ are recursively enumerable, then L is recursive.

Proof.

- $\exists$ TM M_1, M_2 with $L = \mathrm{L}(M_1)$ and $\sim(L) = \mathrm{L}(M_2)$

Theorem

If both L and $\sim$L are recursively enumerable, then L is recursive.

Proof.

- $\exists$ TM M_1, M_2 with $L = \mathrm{L}(M_1)$ and $\sim(L) = \mathrm{L}(M_2)$
- define TM M', such that its tape has two 'halves' (or a TM with 2-tapes):

b	$\hat{b}$	a	b	a	a	a	a	b	a	a	a	...
c	c	c	d	d	d	c	$\hat{c}$	d	c	d	c	

Theorem

If both L and $\sim$L are recursively enumerable, then L is recursive.

Proof.

- $\exists$ TM M_1, M_2 with $L = \mathrm{L}(M_1)$ and $\sim(L) = \mathrm{L}(M_2)$
- define TM M', such that its tape has two 'halves' (or a TM with 2-tapes):

b	$\hat{b}$	a	b	a	a	a	a	b	a	a	a	...
c	c	c	d	d	d	c	$\hat{c}$	d	c	d	c	

- M_1 is simulated on the upper tape and M_2 on the lower tape

Theorem

If both L and $\sim$L are recursively enumerable, then L is recursive.

Proof.

- $\exists$ TM M_1, M_2 with $L = \mathrm{L}(M_1)$ and $\sim(L) = \mathrm{L}(M_2)$
- define TM M', such that its tape has two 'halves' (or a TM with 2-tapes):

b	$\hat{b}$	a	b	a	a	a	a	b	a	a	a	...
c	c	c	d	d	d	c	$\hat{c}$	d	c	d	c	

- M_1 is simulated on the upper tape and M_2 on the lower tape
- if M_1 accepts x, then M' accepts x
- if M_2 accepts x, then M' rejects x

Theorem

If both L and $\sim$L are recursively enumerable, then L is recursive.

Proof.

- $\exists$ TM M_1, M_2 with $L = \mathrm{L}(M_1)$ and $\sim(L) = \mathrm{L}(M_2)$
- define TM M', such that its tape has two 'halves' (or a TM with 2-tapes):

b	$\hat{b}$	a	b	a	a	a	a	b	a	a	a	...
c	c	c	d	d	d	c	$\hat{c}$	d	c	d	c	

- M_1 is simulated on the upper tape and M_2 on the lower tape
- if M_1 accepts x, then M' accepts x
- if M_2 accepts x, then M' rejects x ■

Decidable/semi-decidable properties

Definition

Let Σ be an alphabet. A property P of words over Σ is

- decidable if the set $\{x \in \Sigma^* \mid x \text{ has property } P\}$ is recursive
- semi-decidable if the set $\{x \in \Sigma^* \mid x \text{ has property } P\}$ is recursively enumerable

Decidable/semi-decidable properties

Definition

Let Σ be an alphabet. A property P of words over Σ is

- decidable if the set $\{x \in \Sigma^* \mid x \text{ has property } P\}$ is recursive
- semi-decidable if the set $\{x \in \Sigma^* \mid x \text{ has property } P\}$ is recursively enumerable

Example

Let $P(x) := x$ is a palindrome of even length; then P is decidable

Decidable/semi-decidable properties

Definition

Let Σ be an alphabet. A property P of words over Σ is

- decidable if the set $\{x \in \Sigma^* \mid x \text{ has property } P\}$ is recursive
- semi-decidable if the set $\{x \in \Sigma^* \mid x \text{ has property } P\}$ is recursively enumerable

Example

Let $P(x) := x$ is a palindrome of even length; then P is decidable

Example

Every decidable problem is semi-decidable

Remark

A problem P is

- semi-decidable, if there exists a TM M whose language is the set of words having property P;

Remark

A problem P is

- semi-decidable, if there exists a TM M whose language is the set of words having property P;
- decidable, if there exists a total TM M that accepts exactly the words having property P

Encoding TMs

TMs can be encoded by representing all necessary information as words over $\{0, 1\}$:

1. Number of states
2. transition function
3. input and tape alphabet
4. ...

Encoding TMs

TMs can be encoded by representing all necessary information as words over $\{0, 1\}$:

1. Number of states
2. transition function
3. input and tape alphabet
4. ...

Example

Let $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r)$ be a TM; encoding over $\{0, 1\}$

$$0^n \ 1 \ 0^m \ 1 \ 0^k \ 1 \ 0^s \ 1 \ 0^t \ 1 \ 0^r \ 1 \ 0^u \ 1 \ 0^v \ 1 \quad \cdots$$

represents $Q = \{0, \ldots, n-1\}$, $\Gamma = \{0, \ldots, m-1\}$, $\Sigma = \{0, \ldots, k-1\}$, $(k \leqslant m)$, s initial state, t accepting state, r rejecting state, u left-end marker, v blank symbol

Encoding TMs

TMs can be encoded by representing all necessary information as words over $\{0, 1\}$:

1. Number of states
2. transition function
3. input and tape alphabet
4. ...

Example

Let $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r)$ be a TM; encoding over $\{0, 1\}$

$$0^n \ 1 \ 0^m \ 1 \ 0^k \ 1 \ 0^s \ 1 \ 0^t \ 1 \ 0^r \ 1 \ 0^u \ 1 \ 0^v \ 1 \quad \cdots$$

represents $Q = \{0, \ldots, n-1\}$, $\Gamma = \{0, \ldots, m-1\}$, $\Sigma = \{0, \ldots, k-1\}$, $(k \leqslant m)$, s initial state, t accepting state, r rejecting state, u left-end marker, v blank symbol; the symbol 1 is used as separator in the encoding

Example (Continued)

consider M and encode $\delta(p, a) = (q, b, d)$, where $c = 0$ if $d = \mathrm{L}$ and $c = 1$ if $d = \mathrm{R}$

$$0^{p}\ 1\ 0^{a}\ 1\ 0^{q}\ 1\ 0^{b}\ 1\ 0^{c}\ 1$$

Example (Continued)

consider M and encode $\delta(p, a) = (q, b, d)$, where $c = 0$ if $d = \mathrm{L}$ and $c = 1$ if $d = \mathrm{R}$

$$0^{p}\ 1\ 0^{a}\ 1\ 0^{q}\ 1\ 0^{b}\ 1\ 0^{c}\ 1$$

Example

We encode $M' = (\{s, p, t, r\}, \{0, 1\}, \{0, 1, \vdash, \sqcup\}, \vdash, \sqcup, \delta, s, t, r)$ by

	$\vdash$	0	1	$\sqcup$
s	$(s, \vdash, \mathrm{R})$	$(s, 0, \mathrm{R})$	$(s, 1, \mathrm{R})$	$(p, \sqcup, \mathrm{L})$
p	$(t, \vdash, \mathrm{R})$	$(t, 1, \mathrm{L})$	$(p, 0, \mathrm{L})$	$\cdot$

We obtain

$$\underbrace{0000}_{n=4}1\underbrace{0000}_{m=4}1\underbrace{00}_{k=2}1\underbrace{\epsilon}_{s}1\underbrace{00}_{t}1\underbrace{000}_{r}1\underbrace{00}_{\vdash}1\underbrace{000}_{\sqcup}1\cdots$$

and, for example, $\delta(p, \vdash) = (t, \vdash, \mathrm{R})$ yields $010^{2}10^{2}10^{2}101$

Definition

A TM U is universal (UTM), if for input of

Definition

A TM U is universal (UTM), if for input of

- the code $\ulcorner M \urcorner$ of a TM M, and
- the code $\ulcorner x \urcorner$ of an input x for M

Definition

A TM U is universal (UTM), if for input of

- the code $\ulcorner M \urcorner$ of a TM M, and
- the code $\ulcorner x \urcorner$ of an input x for M

the TM U simulates the TM M on x

Definition

A TM U is universal (UTM), if for input of

- the code $\ulcorner M \urcorner$ of a TM M, and
- the code $\ulcorner x \urcorner$ of an input x for M

the TM U simulates the TM M on x that is

$$L(U) = \{\ulcorner M \urcorner \# \ulcorner x \urcorner \mid x \in L(M)\}$$

Definition

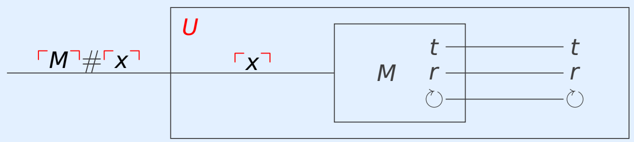
A TM U is universal (UTM), if for input of

- the code $\ulcorner M \urcorner$ of a TM M, and
- the code $\ulcorner x \urcorner$ of an input x for M

the TM U simulates the TM M on x that is

$$L(U) = \{\ulcorner M \urcorner \# \ulcorner x \urcorner \mid x \in L(M)\}$$

UTM schematically



Simulation by a universal Turing machine

Notation

To avoid notational clutter, we often omit the ‘coding corners’:

$$L(U) = \{M \# x \mid x \in L(M)\}$$

Simulation by a universal Turing machine

Notation

To avoid notational clutter, we often omit the 'coding corners':

$$\mathsf{L}(U) = \{M\#x \mid x \in \mathsf{L}(M)\}$$

Simulation

1. UTM U checks correctness of the encodings; if incorrect, U rejects

Simulation by a universal Turing machine

Notation

To avoid notational clutter, we often omit the 'coding corners':

$$\mathsf{L}(U) = \{M\#x \mid x \in \mathsf{L}(M)\}$$

Simulation

1. UTM U checks correctness of the encodings; if incorrect, U rejects
2. U simulates M using 3 tapes, with input x
   - Tape 1 contains the encoding of the TM M
   - Tape 2 contains the encoding of the input word x
   - Tape 3 contains the simulated tape of M

Simulation by a universal Turing machine

Notation

To avoid notational clutter, we often omit the 'coding corners':

$$\mathsf{L}(U) = \{M\#x \mid x \in \mathsf{L}(M)\}$$

Simulation

1. UTM U checks correctness of the encodings; if incorrect, U rejects
2. U simulates M using 3 tapes, with input x
   - Tape 1 contains the encoding of the TM M
   - Tape 2 contains the encoding of the input word x
   - Tape 3 contains the simulated tape of M
3. If M accepts, then U accepts; if M rejects, then U reject

Lemma

Let U be a UTM and M an arbitrary TM. Then there exists a specialisation of U, called U_M, that simulates M on all inputs.

Lemma

Let U be a UTM and M an arbitrary TM. Then there exists a specialisation of U, called U_M, that simulates M on all inputs.

Proof.

- Consider the variation U' of U such that the second tape of U' contains the encoding of the TM to be simulated, and the first tape the (decoded) input
- The desired specialisation U_M is obtained from U' by fixing the code of M on the second tape (hardcoding it)
- By definition, U_M executes all steps of M on the input x

Lemma

Let U be a UTM and M an arbitrary TM. Then there exists a specialisation of U, called U_M, that simulates M on all inputs.

Proof.

- Consider the variation U' of U such that the second tape of U' contains the encoding of the TM to be simulated, and the first tape the (decoded) input
- The desired specialisation U_M is obtained from U' by fixing the code of M on the second tape (hardcoding it)
- By definition, U_M executes all steps of M on the input x ∎

18

Lemma

Let U be a UTM and M an arbitrary TM. Then there exists a specialisation of U, called U_M, that simulates M on all inputs.

Proof.

- Consider the variation U' of U such that the second tape of U' contains the encoding of the TM to be simulated, and the first tape the (decoded) input
- The desired specialisation U_M is obtained from U' by fixing the code of M on the second tape (hardcoding it)
- By definition, U_M executes all steps of M on the input x ∎

Remark

Meta-programming and macros originate with UTMs

Definition

The halting problem and the membership problem for TMs are

$$\text{HP} := \{M\#x \mid M \text{ halts for input } x\}$$

$$\text{MP} := \{M\#x \mid x \in \mathrm{L}(M)\}$$

Definition

The halting problem and the membership problem for TMs are

$$\text{HP} := \{M\#x \mid M \text{ halts for input } x\}$$

$$\text{MP} := \{M\#x \mid x \in \mathrm{L}(M)\}$$

Definition

1. M_x is TM (with input alphabet $\{0,1\}$), whose code (with coding alphabet $\{0,1\}$) is x
2. if x is not the code (of some TM), take M_x arbitrary

Definition

The halting problem and the membership problem for TMs are

$$\text{HP} := \{M\#x \mid M \text{ halts for input } x\}$$

$$\text{MP} := \{M\#x \mid x \in \mathrm{L}(M)\}$$

Definition

1. M_x is TM (with input alphabet $\{0,1\}$), whose code (with coding alphabet $\{0,1\}$) is x
2. if x is not the code (of some TM), take M_x arbitrary

Definition

The halting problem and the membership problem for TMs are

$$\text{HP} := \{M\#x \mid M \text{ halts for input } x\}$$

$$\text{MP} := \{M\#x \mid x \in \mathrm{L}(M)\}$$

Definition

1. M_x is TM (with input alphabet $\{0,1\}$), whose code (with coding alphabet $\{0,1\}$) is x
2. if x is not the code (of some TM), take M_x arbitrary

Enumerating all Turing machines

$$M_\epsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, \dots$$

(ordered with respect to the lexical order)

Definition

The halting problem and the membership problem for TMs are

$$\text{HP} := \{M\#x \mid M \text{ halts for input } x\}$$

$$\text{MP} := \{M\#x \mid x \in \mathrm{L}(M)\}$$

Definition

1. M_x is TM (with input alphabet $\{0,1\}$), whose code (with coding alphabet $\{0,1\}$) is x
2. if x is not the code (of some TM), take M_x arbitrary

Enumerating all Turing machines

$$M_\epsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, \dots$$

(ordered with respect to the lexical order)

Two-dimensional matrix of behaviours (loops ↻ vs. halts !)

indexed by words $w \in \{0,1\}^*$ respectively Turing machines

Two-dimensional matrix of behaviours (loops ↻ vs. halts !)

indexed by words $w \in \{0,1\}^*$ respectively Turing machines

	ϵ	0	1	00	01	10	11	000	001	010	...
M_ϵ	!	↻	↻	!	!	↻	!	↻	!	!	
M_0	↻	↻	!	!	↻	!	!	↻	↻	!	
M_1	↻	!	↻	!	↻	!	!	↻	↻	!	
M_{00}	!	↻	↻	!	!	!	!	↻	↻	!	
M_{01}	!	!	!	!	↻	↻	↻	!	!	↻	...
M_{10}	!	!	↻	!	!	↻	!	!	↻	!	
M_{11}	!	!	↻	↻	!	↻	!	↻	!	↻	
M_{000}	!	!	!	!	↻	!	!	↻	!	↻	
M_{001}	↻	!	!	!	!	↻	!	!	!	!	
$\vdots$						$\vdots$					$\ddots$

Claim

the behaviour *cd* corresponding to the complement of the behaviour at the diagonal, is not the behaviour of any TM

Claim

the behaviour *cd* corresponding to the complement of the behaviour at the diagonal, is not the behaviour of any TM

Proof.

Behaviours are functions from finite bit-strings in $\{0,1\}^*$ (inputs) to $\{!, ↻\}$.
Given an enumeration m_ϵ, m_0, m_1, ... of such behaviours, indexed by finite bit-strings

$$\begin{aligned} m_\epsilon &= m_\epsilon(\epsilon)m_\epsilon(0)m_\epsilon(1)m_\epsilon(00)m_\epsilon(01)\dots \\ m_0 &= m_0(\epsilon)m_0(0)m_0(1)m_0(00)m_0(01)\dots \\ m_1 &= m_1(\epsilon)m_1(0)m_1(1)m_1(00)m_1(01)\dots \\ &\vdots \end{aligned}$$

behaviour *cd* defined by

$$cd(x) = \begin{cases} ↻ & \text{if } m_x(x) = ! \\ ! & \text{if } m_x(x) = ↻ \end{cases}$$

is a new behaviour; distinct from each m_x, namely at x: $m_x(x) = \overline{cd(x)} \neq cd(x)$

Claim

the behaviour *cd* corresponding to the complement of the behaviour at the diagonal, is not the behaviour of any TM

Proof.

Behaviours are functions from finite bit-strings in $\{0,1\}^*$ (inputs) to $\{!, \circlearrowright\}$.
Given an enumeration $m_\epsilon, m_0, m_1, \ldots$ of such behaviours, indexed by finite bit-strings

$$m_\epsilon = m_\epsilon(\epsilon)m_\epsilon(0)m_\epsilon(1)m_\epsilon(00)m_\epsilon(01)\ldots$$
$$m_0 = m_0(\epsilon)m_0(0)m_0(1)m_0(00)m_0(01)\ldots$$
$$m_1 = m_1(\epsilon)m_1(0)m_1(1)m_1(00)m_1(01)\ldots$$
$$\vdots$$

behaviour *cd* defined by

$$cd(x) = \begin{cases} \circlearrowright & \text{if } m_x(x) = ! \\ ! & \text{if } m_x(x) = \circlearrowright \end{cases}$$

is a new behaviour; distinct from each m_x, namely at x: $m_x(x) = \overline{cd(x)} \neq cd(x)$ ■

Theorem

HP *is not recursive, but recursively enumerable*

Theorem

HP *is not recursive, but recursively enumerable*

Proof.

we first show non-recursiveness

1. for proof by contradiction, suppose total TM K such that HP $=$ L(K) were to exist

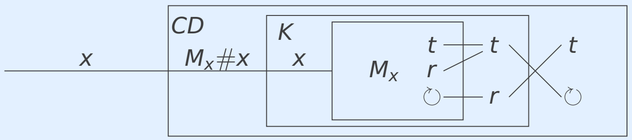
Theorem

HP *is not recursive, but recursively enumerable*

Proof.

we first show non-recursiveness

1. for proof by contradiction, suppose total TM K such that HP $=$ L(K) were to exist
2. then we could construct a TM *CD*, based on K, as follows



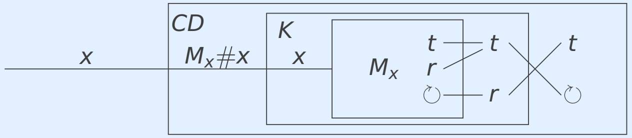
Theorem

HP *is not recursive, but recursively enumerable*

Proof.

we first show non-recursiveness

1. for proof by contradiction, suppose total TM K such that $\mathrm{HP} = \mathrm{L}(K)$ were to exist
2. then we could construct a TM CD, based on K, as follows



3. CD exhibits behaviour cd. For the earlier behaviour matrix we, e.g., have:
   - M_{10} loops on 10, so K rejects $M_{10}\#10$ and CD halts on (accepts) 10; indeed $cd(10) = !$
   - M_{001} halts on 001, so K accepts $M_{001}\#001$ and CD loops on 001; indeed $cd(001) = \circlearrowright$

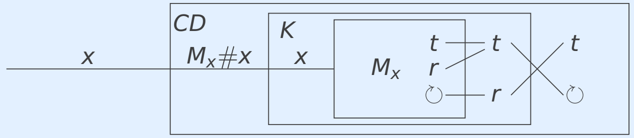
Theorem

HP *is not recursive, but recursively enumerable*

Proof.

we first show non-recursiveness

1. for proof by contradiction, suppose total TM K such that $\mathrm{HP} = \mathrm{L}(K)$ were to exist
2. then we could construct a TM CD, based on K, as follows



3. CD exhibits behaviour cd. For the earlier behaviour matrix we, e.g., have:
   - M_{10} loops on 10, so K rejects $M_{10}\#10$ and CD halts on (accepts) 10; indeed $cd(10) = !$
   - M_{001} halts on 001, so K accepts $M_{001}\#001$ and CD loops on 001; indeed $cd(001) = \circlearrowright$

4. Behaviour cd distinct from that of any TM M_x, so CD not a TM, so K not a TM.

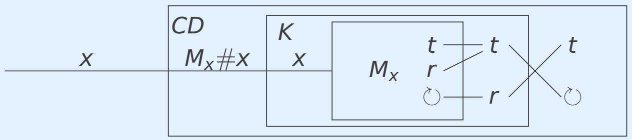
Theorem

HP *is not recursive, but recursively enumerable*

Proof.

we first show non-recursiveness

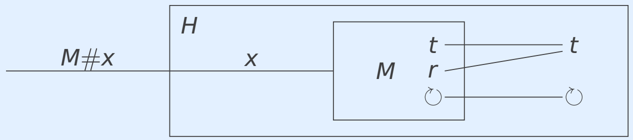
1. for proof by contradiction, suppose total TM K such that $\mathrm{HP} = \mathrm{L}(K)$ were to exist
2. then we could construct a TM CD, based on K, as follows



3. CD exhibits behaviour cd. For the earlier behaviour matrix we, e.g., have:
   - M_{10} loops on 10, so K rejects $M_{10}\#10$ and CD halts on (accepts) 10; indeed $cd(10) = !$
   - M_{001} halts on 001, so K accepts $M_{001}\#001$ and CD loops on 001; indeed $cd(001) = \circlearrowright$
4. Behaviour cd distinct from that of any TM M_x, so CD not a TM, so K not a TM.
   Contradiction. ∎

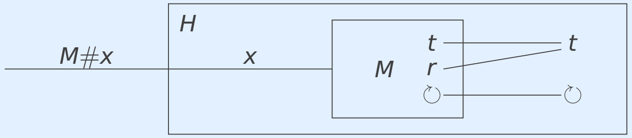
Proof (Continued).

We sketch why HP is recursively enumerable; to that end we construct the following TM H, based on the universal TM



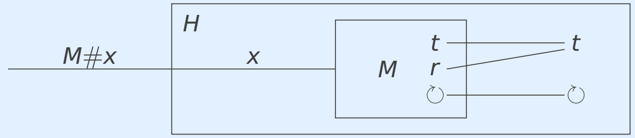
Proof (Continued).

We sketch why HP is recursively enumerable; to that end we construct the following (not necessarily total) TM H, based on the universal TM



Proof (Continued).

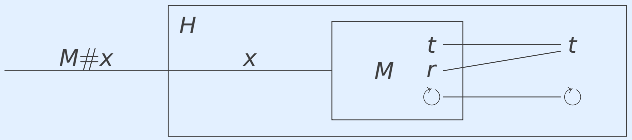
We sketch why HP is recursively enumerable; to that end we construct the following (not necessarily total) TM H, based on the universal TM



∎

Proof (Continued).

We sketch why HP is recursively enumerable; to that end we construct the following (not necessarily total) TM H, based on the universal TM



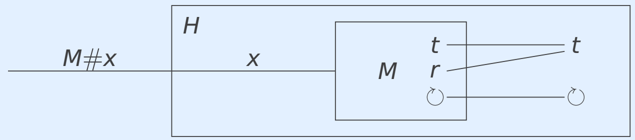
∎

Corollary

The set $\sim$HP *is not recursively enumerable*

Proof (Continued).

We sketch why HP is recursively enumerable; to that end we construct the following (not necessarily total) TM H, based on the universal TM



∎

Corollary

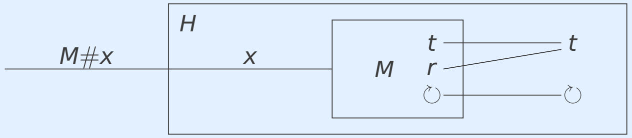
The set $\sim$HP *is not recursively enumerable*

Proof.

Suppose $\sim$HP were recursively enumerable; then both HP and $\sim$HP would be recursively enumerable, hence HP would be recursive. Contradiction

Proof (Continued).

We sketch why HP is recursively enumerable; to that end we construct the following (not necessarily total) TM H, based on the universal TM



■

Corollary

The set $\sim$HP *is not recursively enumerable*

Proof.

Suppose $\sim$HP were recursively enumerable; then both HP and $\sim$HP would be recursively enumerable, hence HP would be recursive. Contradiction ■