Computational Complexity; slides 2, HT 2022
Turing machines, undecidability (review/recall, for general context)

Prof. Paul W. Goldberg (Dept. of Computer Science, University of Oxford)

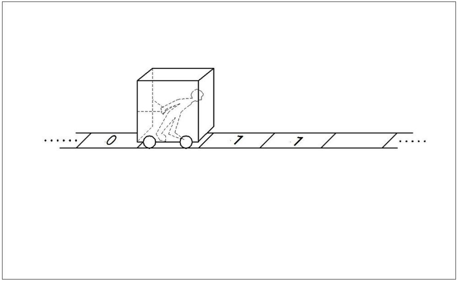
HT 2022

Computation

Alan Turing considered qn. of "What is computation?" in 1936.

He argued, that *any* computation can be done using the following steps (writing on a sheet of paper):

- Concentrate on one part of the problem (one symbol on the paper)
- Depending on what you read there
  - Change into a new state (memorise a finite amount of information)
  - Modify this part of the problem
  - Move to another part of the input
- Repeat until finished



Key points

Why we care about TMs:

- precise notion of "runtime", "memory usage"
- well-defined operations on algorithms (when represented as TMs) — (operations such as pass output of Alg 1 to Alg 2, etc)
- variants of TM (e.g. NTM) define important classes of problems

Sometimes we'll use pseudocode but with understanding that there's an equivalent TM

Next: detailed definition

Deterministic Turing Machines

Definition. (one of many variants, all "equivalent")
A (deterministic) k-tape *Turing machine* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is *input alphabet* – a finite alphabet of *symbols*
- $\Gamma \supseteq \Sigma \cup \{\Box\}$ is *working tape alphabet* (finite)
- δ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of final states

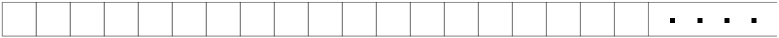
Deterministic Turing Machines

Definition. (one of many variants, all "equivalent")
A (deterministic) k-tape *Turing machine* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is *input alphabet* – a finite alphabet of *symbols*
- $\Gamma \supseteq \Sigma \cup \{\square\}$ is *working tape alphabet* (finite)
- δ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of final states

Tape. Infinite tape, bounded to the left.

Each cell contains one symbol from Γ ($\square$: special "blank" symbol)

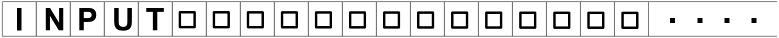


Definition. (one of many variants, all "equivalent")
A (deterministic) k-tape *Turing machine* is a 6-tuple
$(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is *input alphabet* – a finite alphabet of *symbols*
- $\Gamma \supseteq \Sigma \cup \{\Box\}$ is *working tape alphabet* (finite)
- δ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of final states

Tape. Infinite tape, bounded to the left.

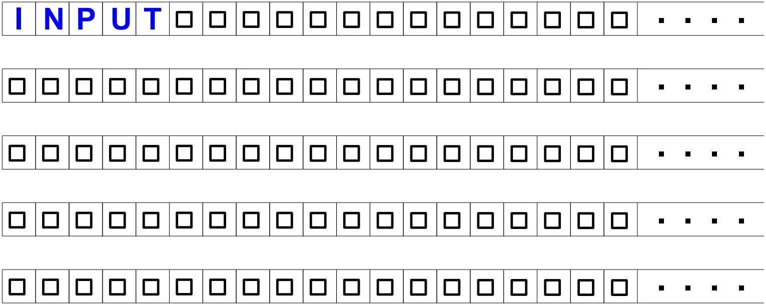
Each cell contains one symbol from Γ ($\Box$: special "blank" symbol)



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$

(-1: "left" 0: "stay put" 1: "right")



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{-1, 0, 1\}^k$

(-1: "left" 0: "stay put" 1: "right")

Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$

(-1: "left" 0: "stay put" 1: "right")

Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{-1, 0, 1\}^k$

(-1: "left" 0: "stay put" 1: "right")

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{-1, 0, 1\}^k$

(-1: "left" 0: "stay put" 1: "right")

Turing Machine operation

1. At each step of operation the machine is in one state $q \in Q$
2. Initially:
   - Machine is in state $q_0 \in Q$
   - the input is contained on tape 1
   - all other tape symbols are $\square$
3. The machine is reading one symbol on each tape: $s_1 \ldots s_k$
4. To execute one step, the machine looks up

$$\delta(q, s_1, \ldots, s_k) := (q', (s'_1, \ldots, s'_k), (m_1, \ldots, m_k))$$

5. The machine:
   - changes to state q'
   - replaces each s_i by s'_i
   - moves the heads on the individual tapes according to m_i (1 = move right, -1 = move left, 0 = stay)
   - Execution stops when a final state is reached.
   - In this case, the content of the last tape k contains the output.

more general points

I assume you've seen examples of TMs already.

- TM: general-purpose notion of "algorithm", "computational procedure"
- equivalence of alternative defs of TM assure us of above
- Algorithm pseudocode is readable, usually we use it to describe algorithms, tacit assumption: can be converted to TM
- TMs $\rightsquigarrow$ precise notion of runtime/space. Used in various theorems in this course.

Configurations (definition, notation)

For $M := (Q, \Sigma, \Gamma, \delta, q_0, F)$, what's going on is described by

- the current state
- the contents of all tapes
- the position of all its heads

$\big(q,\ (w_1, \ldots, w_k),\ (p_1, \ldots, p_k)\big)$ where $q \in Q, w_i \in \Gamma^*,\ p_i \in \mathbb{N}$

where Γ^* denotes words over alphabet Γ

Start configuration on input w: $\big(q_0, (w, \varepsilon, \ldots, \varepsilon), (0, \ldots, 0)\big)$

where ε denotes empty word

Stop (or, halt) configuration:
Configuration $\big(q, (w_1, \ldots, w_k), (p_1, \ldots, p_k)\big)$ such that $q \in F$.

Notation:

- $C \vdash_M C'$ if M can change from configuration C to C' in one step.
- $C \vdash_M^* C'$ if M can change from configuration C to C' in arbitrarily many steps.

Notation:

- $C \vdash_M C'$ if M can change from configuration C to C' in one step.
- $C \vdash_M^* C'$ if M can change from configuration C to C' in arbitrarily many steps.

The *computation* of a TM M on input $w \in \Sigma^*$ is either

- an infinite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \dots$ of configurations, or
- a finite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \cdots \vdash_M C_n$.

  In the latter case we say that M *halts* on input w.

  Notation: $T_M(w) := n$ number of steps upon input w.

C_n: stop configuration C_0: start config of M on input w.

Computation

Notation:

- $C \vdash_M C'$ if M can change from configuration C to C' in one step.
- $C \vdash_M^* C'$ if M can change from configuration C to C' in arbitrarily many steps.

The *computation* of a TM M on input $w \in \Sigma^*$ is either

- an infinite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \ldots$ of configurations, or
- a finite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \cdots \vdash_M C_n$.

  In the latter case we say that M *halts* on input w.

  Notation: $T_M(w) := n$ number of steps upon input w.

C_n: stop configuration C_0: start config of M on input w.

A TM *halts on input* w (and generates output o) if the computation of M on w terminates in configuration

$$(q, (w_1, \ldots, w_{k-1}, o), (p_1, \ldots, p_k)) \quad \text{with} \quad q \in F.$$

Run-time of a TM

Let M be a Turing machine with alphabet Σ
$f : \Sigma^* \to \Sigma^*$
$g : \mathbb{N} \to \mathbb{N}$

M *computes* f *in time* $g(n)$ if for every $w \in \Sigma^*$ M halts on input w after at most $g(|w|)$ steps with $f(w)$ on its output (last) tape.

(i.e. $T_M(w) \leq g(|w|)$)

Example (TM as transducer)

The following 2-tape Turing machine

$$M := \big(\{q_0, q_1, q_f\}, \{a, b\}, \{a, b, \Box\}, \delta, q_0, \{q_f\}\big)$$

where

$$\delta := \left\{ \begin{array}{l} \left(q_0, \binom{a}{-}, \binom{a}{-}, \binom{1}{0}, q_0\right) \\ \left(q_0, \binom{b}{-}, \binom{b}{-}, \binom{1}{0}, q_0\right) \\ \left(q_0, \binom{\Box}{-}, \binom{\Box}{-}, \binom{-1}{0}, q_1\right) \\ \left(q_1, \binom{a}{-}, \binom{\Box}{a}, \binom{-1}{1}, q_1\right) \\ \left(q_1, \binom{b}{-}, \binom{\Box}{b}, \binom{-1}{1}, q_1\right) \\ \left(q_1, \binom{\Box}{-}, \binom{\Box}{-}, \binom{0}{0}, q_f\right) \end{array} \right\}$$

computes the *reverse*-function *reverse*$(a_1 \dots a_n) := a_n \dots a_1$ in time $g(n) = 2n + 2 = \mathcal{O}(n)$.

For various alternative definitions of TM, including changes to alphabet, runtimes needed are polynomially related.

Decision problems as languages; Turing acceptors

Example

Travelling Salesman Problem (TSP): Given pairwise distances between cities, we ask for the shortest tour, or the length of the shortest tour

Decision version: given the pairwise distances and a number k, is there a tour of length at most k?

General point: ability to solve the decision version is "good enough" (why?).

For decision problem D, $\mathcal{L}(D)$ denotes the *yes-instances* of D (needs an agreed-on encoding).

TM M solves a decision problem if the language accepted by M (M as a *language acceptor*) is the yes-instances of the decision problem.

Recall: decidable languages

Definition/notation

The language $\mathcal{L}(M) \subseteq \Sigma^*$ *accepted* by a Turing acceptor $M := (Q, \Sigma, \Gamma, \delta, q_0, F)$ is defined as

$$\{w \in \Sigma^* : M \text{ accepts } w\}.$$

(Note that we do not require M to halt on rejected inputs.)

A language $\mathcal{L} \subseteq \Sigma^*$ is *recursively enumerable*, if there is an acceptor M such that $\mathcal{L} = \mathcal{L}(M)$.

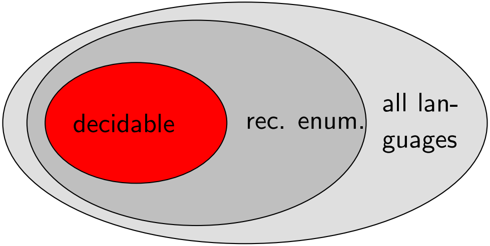
A language $\mathcal{L} \subseteq \Sigma^*$ is *decidable* (or, "recursive") if there is an acceptor M such that for all $w \in \Sigma^*$:

$$w \in \mathcal{L} \implies M \text{ halts on input } w \text{ in an accepting state}$$
$$w \notin \mathcal{L} \implies M \text{ halts on input } w \text{ in a rejecting state}$$

Decidable and Enumerable Languages

Recall:

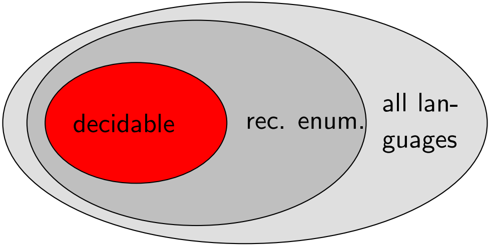
1. If a language $\mathcal{L}$ is *decidable* then it is *recursively enumerable*
2. If $\mathcal{L}$ and $\Sigma^* \setminus \mathcal{L}$ are *recursively enumerable* then $\mathcal{L}$ is decidable.



Decidable and Enumerable Languages

Recall:

1. If a language $\mathcal{L}$ is *decidable* then it is *recursively enumerable*
2. If $\mathcal{L}$ and $\Sigma^* \setminus \mathcal{L}$ are *recursively enumerable* then $\mathcal{L}$ is decidable.



Note: recursively enumerable a.k.a. *semi-decidable*, *partially decidable*

Problems as languages

Main points:

- decision problems viewed as language recognition problems We can use "decision problem" and "language" interchangeably
- We're allowed to be vague about encoding of problems (e.g. CLIQUE, TSP) — we will see that details of encoding don't affect the problem classifications of interest. Details of alphabet also unimportant (but *unary* alphabet is too big a restriction!). ("standard encoding", should be sensible.)

Undecidable Languages

Aim of this section

- Recursion theory — a brief reminder
- 2 techniques: *diagonalisation* and *reductions* — variants appear in complexity-theory classification of problems

A counting argument (sketch):

- The number of Turing machines is infinite but *countable*
- The number of different languages is infinite but *uncountable*; diagonalisation
- Therefore, there are "more" languages than Turing machines

It follows that there are languages that are not decidable.
Indeed some aren't even semi-decidable.

The Halting Problem

previous argument shows that there are undecidable languages.

Can we find a concrete example?

Halting problem (HALT)

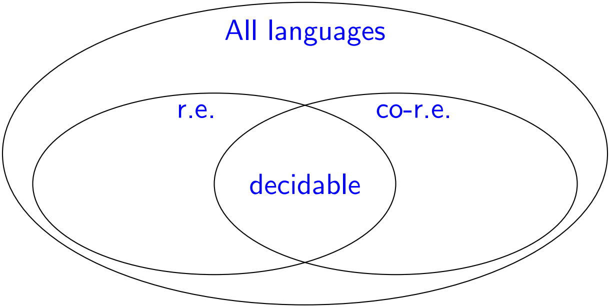
Input: A Turing machine M and an input string w
Question: Does M halt on w?

Again, undecidability of HALT is proved by diagonalisation: consider effective listing of TMs, new TM that differs from all in listing
details in e.g. Sipser Chapter 4.2

Classification of Languages

Definition. A language $\mathcal{L} \subseteq \Sigma^*$ is *co-recursively enumerable*, or *co-r.e.*, if $\Sigma^* \setminus \mathcal{L}$ is recursively enumerable.

Example: $\mathcal{L}(\overline{\text{HALT}})$ is co-r.e (but not r.e.).



Looking ahead, relationship between NP and co-NP is more complicated...

Reductions

A major tool in analysing and classifying problems is the idea of "reducing one problem to another"

As you expect — or have already seen — use undecidability of HALT to prove undecidability of variants, e.g. TM acceptance problem.

- Informally, a problem $\mathcal{A}$ is *reducible* to a problem $\mathcal{B}$ if we can use methods to solve $\mathcal{B}$ in order to solve $\mathcal{A}$.
- We want to capture the idea, that $\mathcal{A}$ is "no harder" than $\mathcal{B}$.

  (as we can use $\mathcal{B}$ to solve $\mathcal{A}$.)

Turing Reductions

Informally, problem $\mathcal{A}$ is *Turing reducible* to $\mathcal{B}$ if we can solve $\mathcal{A}$ using a program solving $\mathcal{B}$ as sub-program.

We write $\mathcal{A} \leq_T \mathcal{B}$.

Example: $\overline{\text{HALT}}$ is Turing reducible to HALT.

take a Turing acceptor accepting HALT as sub-program and reverse its output

Turing Reductions

Informally, problem $\mathcal{A}$ is *Turing reducible* to $\mathcal{B}$ if we can solve $\mathcal{A}$ using a program solving $\mathcal{B}$ as sub-program.

We write $\mathcal{A} \leq_T \mathcal{B}$.

Example: $\overline{\text{HALT}}$ is Turing reducible to HALT.

take a Turing acceptor accepting HALT as sub-program and reverse its output

Turing reductions are free/unrestricted; sometimes too much so for our purposes.

$\rightsquigarrow$ **Many-One Reductions** (Sipser: "mapping reduction") *are more informative*: $\mathcal{A} \leq_T \mathcal{B}$ relates (un)decidability of problems; use $\mathcal{A} \leq_m \mathcal{B}$ (next slide) to find out if a problem (or its complement) is recursively enumerable.

Many-One Reductions

Definition. A language $\mathcal{A}$ is *many-one reducible* to a language $\mathcal{B}$ if there exists a computable function f such that for all $w \in \Sigma^*$:

$$x \in \mathcal{A} \quad \Longleftrightarrow \quad f(x) \in \mathcal{B}.$$

We write $\mathcal{A} \leq_m \mathcal{B}$.

Observation 1. If $\mathcal{A} \leq_m \mathcal{B}$ and $\mathcal{B}$ is decidable, then so is $\mathcal{A}$.

Proof. A many-one reduction is a Turing reduction, so it inherits that functionality

Observation 2. If $\mathcal{A} \leq_m \mathcal{B}$ and $\mathcal{B}$ is recursively enumerable, then so is $\mathcal{A}$.

Many-one reductions can classify problems into:
decidable/r.e./co-r.e/neither.

Properties of Many-One Reductions

1. $\leq_m$ is *reflexive* and *transitive*
   (if $\mathcal{A} \leq_m \mathcal{B}$ and $\mathcal{B} \leq_m \mathcal{C}$ then $\mathcal{A} \leq_m \mathcal{C}$, by composition of functions.)
2. If $\mathcal{A}$ is decidable and $\mathcal{B}$ is *any* language apart from $\emptyset$ and Σ^*, then $\mathcal{A} \leq_m \mathcal{B}$.

   As $\mathcal{B} \neq \emptyset$ and $\mathcal{B} \neq \Sigma^*$ there are $w_a \in \mathcal{B}$ and $w_r \notin \mathcal{B}$.

   For $w \in \Sigma^*$, define $f(w) := \begin{cases} w_a & \text{if } w \in \mathcal{A} \\ w_r & \text{if } w \notin \mathcal{A} \end{cases}$

Hence, many-one reductions are too crude to distinguish between decidable problems. later: "smarter" reductions

Examples of Many-One Reductions

We will show the following chain of reductions:

$$\text{HALT} \leq_m \varepsilon\text{-HALT} \leq_m \text{EQUIVALENCE}$$

ε-HALT: Does M halt on the empty input?

EQUIVALENCE: $\mathcal{L}(M) = \mathcal{L}(M')$?

Hence, all these problems are undecidable.

Proof.

Define function f such that $w \in \text{HALT} \iff f(w) \in \varepsilon\text{-HALT}$

For $w := \langle M, v \rangle$ compute the following Turing machine M_w :

1. Write v onto the input tape.
2. Simulate M.

Clearly, M_w accepts the empty word if, and only if, M accepts v.

Let M_r be a TM that does not halt on the empty input.

Define $f(w) := \begin{cases} M_w & \text{if } w = \langle M, v \rangle \\ M_r & \text{if } w \text{ is not of the correct input form } ^1 \end{cases}$

[1] i.e. doesn't encode a TM with word

ε-HALT $\leq_m$ EQUIVALENCE

Proof.

Define f such that $w \in \varepsilon\text{-HALT} \iff f(w) \in \text{EQUIVALENCE}$

Let M_a be a Turing machine that accepts all inputs.

For a TM M compute the following Turing machine M^* :

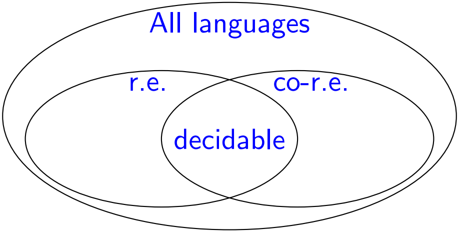
1. Run M on the empty input
2. If M halts, accept.

M^* is equivalent to M_a if, and only if, M halts on the empty input.

Define

$$f(w) := \begin{cases} (\langle M^* \rangle, \langle M_a \rangle) & \text{if } w = \langle M \rangle \\ (w, \langle M_a \rangle) & \text{if } w \text{ is not of the correct input form} \end{cases}$$

Decidable and Enumerable Languages



Recursion Theory:

Study the border between decidable and undecidable languages
Study the fine structure of undecidable languages.

The work of Turing, Church, Post, ... pre-dated modern computational machinery.

Complexity Theory:

Look at the fine structure of decidable languages.