8: Intro. to Turing Machines

Problems that Computers Cannot Solve

It is important to know whether a program is correct, namely that it does what we expect.

It is easy to see that the following C program

```
main()
{
        printf(``hello, world\n'');
}
```

prints `hello, world` and terminates.

But what about the program in Figure 8.2 on page 309 of the textbook?!

Given an input n, it prints `hello, world` only if the equation

$$x^n + y^n = z^n$$

has a solution where x, y, and z are integers.

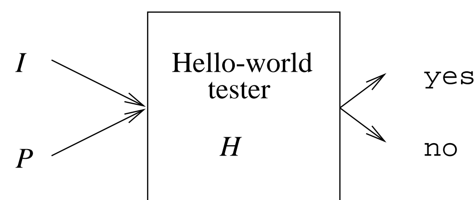
We know nowadays that it will print `hello, world` for $n = 2$, and loop forever for $n > 2$.

It took mathematicians 300 years to prove this so-called "Fermat's last theorem".

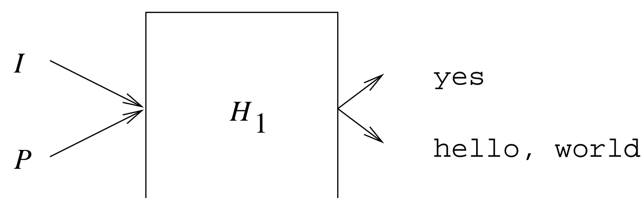
Can we expect to write a program H that solves the general problem of telling whether any given program P, on any given input I, eventually prints `hello, world` or not?

The Hypothetical "hello, world" Tester H

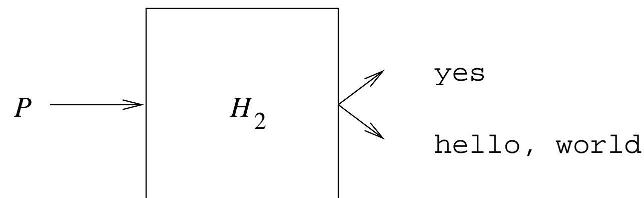
Proof by contradiction that H is impossible to write. Suppose that H exists:



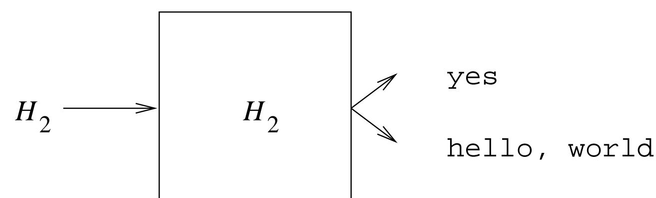
We modify the response **`no`** of H to **`hello, world`**, getting a program H_1:



We modify H_1 to take P and I as a single input, getting a program H_2:



We provide H_2 as input to H_2:



If H_2 prints **`yes`**,
then it should have printed **`hello, world`**.

If H_2 prints **`hello, world`**,
then it should have printed **`yes`**.

So H_2 and hence H cannot exist.

Hence we have an *undecidable* problem. It is similar to the language L_d we will see later.

Undecidable Problems

A problem is *undecidable* if no program can solve it.

Here: problem = deciding on the membership of a string in a language.

Languages over an alphabet are *not* enumerable.

Programs (finite strings over an alphabet) *are* enumerable: order them by length, and then lexicographically.

Hence there are infinitely more languages than programs.

Hence there must be undecidable problems (Gödel, 1931).

Problem Reduction

If we already know problem P_1 to be undecidable, can we use this fact to show that another problem P_2 is undecidable?

Assume there exists a program that decides whether its input instance of problem P_2 is or is not in the language of P_2.

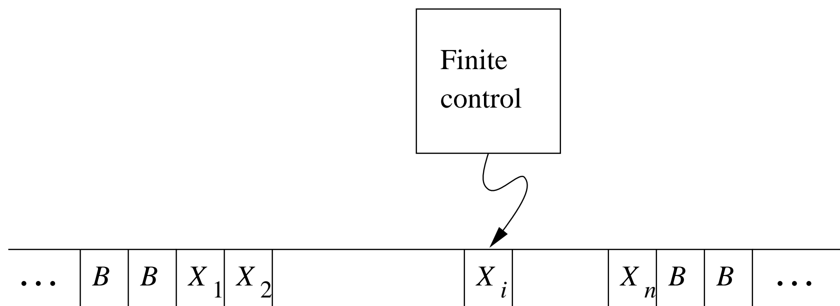
Reduce the known undecidable problem P_1 to P_2: Convert instances of P_1 into instances of P_2 that have the *same* answer.

But we would then have an algorithm for deciding P_1! Contradiction. Hence the assumed program for deciding P_2 does not exist and P_2 is in fact undecidable.

Thereby, we proved the statement "if P_2 is decidable, then P_1 is decidable" and exploited its contrapositive.

Careful: To prove P_2 undecidable, we must *not* reduce P_2 to some known undecidable problem P_1 (by converting instances of P_2 into instances of P_1 that have the same answer), as we would then prove the vacuously true and thus useless statement "if P_1 is decidable, then P_2 is decidable".

Turing Machines (1936)



A *move* of a Turing machine (TM) is a function of the state of the finite control and the tape symbol just scanned.

In one move, the Turing machine will:

1. Change state.

2. Write a tape symbol in the cell scanned.

3. Move the tape head left or right.

Formally, a *Turing machine* is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where:

- Q is the finite set of *states* of the finite control.

- Σ is the finite set of *input symbols*.

- Γ is the finite set of *tape symbols*; $\Sigma \subset \Gamma$.

- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*, which is a partial function.

- $q_0 \in Q$ is the *start state*.

- $B \in \Gamma$ is the *blank symbol*; $B \notin \Sigma$.

- $F \subseteq Q$ is the set of *final* or *accepting* states.

Instantaneous Descriptions for TMs

A Turing machine changes its configuration upon each move.

We use instantaneous descriptions (IDs) for describing such configurations.

An *instantaneous description* is a string of the form

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n$$

where

1. q is the state of the Turing machine.

2. The tape head is scanning the ith symbol from the left.

3. $X_1 X_2 \cdots X_n$ is the portion of the tape between the leftmost and rightmost nonblanks.

The Moves and Language of a TM

We use $\underset{M}{\vdash}$ to designate a move of a Turing machine M from one ID to another.

If $\delta(q, X_i) = (p, Y, L)$, then:

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash}$$
$$X_1 X_2 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n$$

If $\delta(q, X_i) = (p, Y, R)$, then:

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash}$$
$$X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$$

The reflexive-transitive closure of $\underset{M}{\vdash}$ is denoted by $\overset{*}{\underset{M}{\vdash}}$.

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *accepts* the language

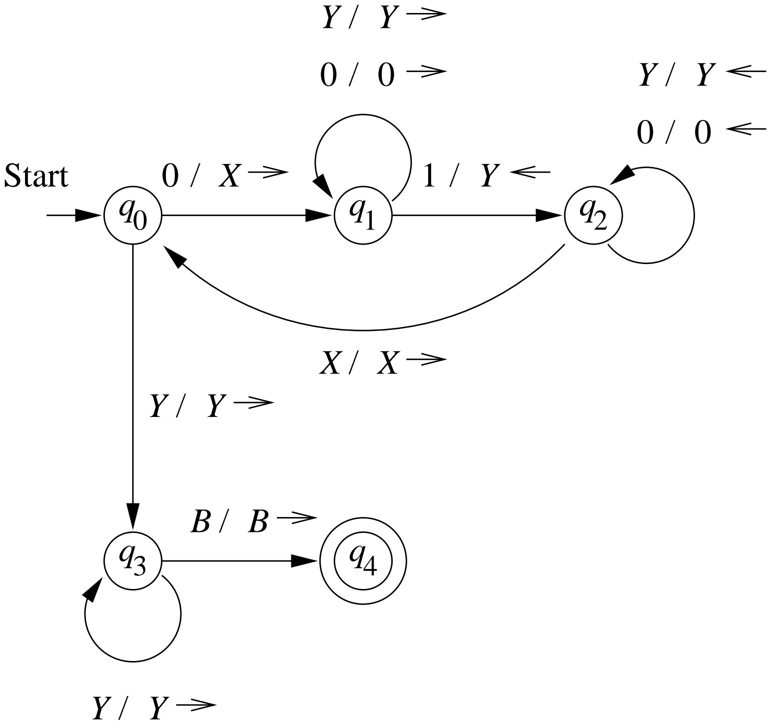
$$L(M) = \{w \in \Sigma^* : q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta, p \in F, \alpha, \beta \in \Gamma^*\}$$

Example: A TM for $\{0^n1^n : n \geq 1\}$

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0,1\}, \{0,1,X,Y,B\}, \delta, q_0, B, \{q_4\})$ where δ is given by the following table:

	0	1	X	Y	B
$\rightarrow q_0$	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
$\star\ q_4$					

We can also represent M by the following *transition diagram*:



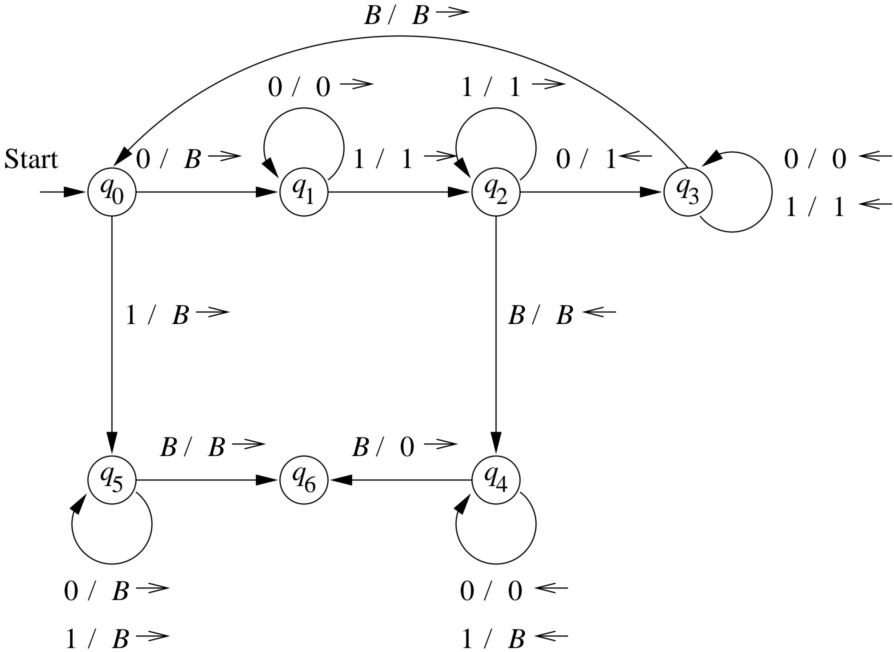
Example: A TM With "Output"

The following Turing machine computes

$$m \dot{-} n = \max(m - n, 0)$$

	0	1	B
$\rightarrow q_0$	(q_1, B, R)	(q_5, B, R)	
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	
q_2	$(q_1, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
$\star\ q_6$			

The transition diagram is as follows:



Acceptance by Halting

A Turing machine *halts* if it enters a state q, scanning a tape symbol X, and there is no move in this situation, i.e., $\delta(q, X)$ is undefined.

We can always assume that a Turing machine halts if it accepts, as we can make $\delta(q, X)$ undefined whenever q is an accepting state.

Unfortunately, it is not always possible to require that a Turing machine halts even if it does not accept.

Recursive language: there is a TM, corresponding to the concept of *algorithm*, that halts eventually, whether it accepts or not.

Recursively enumerable language: there is a TM that halts if the string is accepted.

Decidable problem: there is an algorithm for solving it.

Alternative Models for Turing Machines

Turing-machine programming techniques: storage in the state, multiple tape tracks, subroutines, . . .

Extensions: multiple tapes, non-determinism, . . .

Restrictions: semi-infinite tape, multiple stacks, counters, . . .

All these models are equivalent: they accept the recursively enumerable languages (Church-Turing thesis, 1936).

Turing Machines and Computers

Simulating a Turing machine by a computer: it suffices to have enough memory to simulate the infinite tape.

Simulating a computer by a Turing machine: multiple tapes (memory, instruction counter, memory address, computer's input file, and scratch) plus simulation of the instruction cycle.

The simulating multitape Turing machine needs an amount of steps that is at most some polynomial, namely n^3, in the number n of steps taken by the simulated computer.

From now on: computer = Turing machine.

9: Undecidability

Goal: Prove undecidable the recursively enumerable language L_u consisting of pairs (M, w) such that:

- M is a Turing machine (suitably coded, in binary) with input alphabet $\{0, 1\}$.

- w is a string of 0s and 1s.

- M accepts input w.

If this problem with binary inputs is undecidable, then surely the more general problem, where the Turing machines may have any alphabet, is undecidable.

First step: codify a Turing machine as a string of 0s and 1s, and exhibit a language that is not even recursively enumerable, namely L_d.

Codes for Turing Machines

We need to assign integers to all the binary strings so that each integer corresponds to one string and vice versa: ϵ is the first string, 0 the second, 1 the third, 00 the fourth, 01 the fifth, and so on.

Equivalently, strings are ordered by length, and strings of equal length are ordered lexicographically.

We will refer to the ith string as w_i.

We now want to represent Turing machines with input alphabet $\{0, 1\}$ by binary strings, so that we can identify Turing machines with integers and refer to the ith Turing machine as M_i.

To represent a Turing machine

$$M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F\}$$

as a binary string, we must first assign integers to the states, tape symbols, and directions L and R:

- Assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state is q_1, and the only accepting state is q_2.

- Assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. Then: $0 = X_1$, $1 = X_2$, and $B = X_3$.

- $L = D_1$ and $R = D_2$.

Encode the transition rule
$\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ by $0^i10^j10^k10^\ell10^m$.
Note that there are no two consecutive 1s.

Encode an entire Turing machine by concatenating, in any order, the codes C_i of its transition rules, separated by 11: $C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$.

Ex.: $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$ where δ is defined by: $\delta(q_1, 1) = (q_3, 0, R)$, $\delta(q_3, 0) = (q_1, 1, R)$, $\delta(q_3, 1) = (q_2, 0, R)$, and $\delta(q_3, B) = (q_3, 1, L)$.

Codes for the transition rules:
0100100010100
0001010100100
00010010010100
0001000100010010

Code for M:
010010001010011000101010010011
00010010010100110001000100010010

Given a Turing machine M with code w_i, we can now associate an integer to it: M is the ith Turing machine, referred to as M_i.

Many integers do no correspond to any Turing machine at all. Examples: 11001 and 001110.

If w_i is not a valid TM code, then we shall take M_i to be the Turing machine (with one state and no transitions) that immediately halts on any input. Hence $L(M_i) = \emptyset$ if w_i is not a valid TM code.

The Diagonalisation Language L_d

The *diagonalisation language* L_d is the set of strings w_i such that $w_i \notin L(M_i)$.

That is, L_d contains all strings w such that the Turing machine M with code w does not accept w.

Consider the matrix with Turing machine indices i in the rows and string indices j in the columns, where the cell for row i and column j tells whether M_i accepts w_j, "yes" being denoted by 1 and "no" by 0. The diagonal values tell whether M_i accepts w_i. The strings of L_d correspond to the 0s of the diagonal.

Is it possible that the diagonal complement is a row? No, because the diagonal complement disagrees with every row in some column.

Hence L_d is not recursively enumerable and cannot be accepted by any Turing machine.

Recursive Languages

A language L is recursive if $L = L(M)$ for some Turing machine M such that:

- If $w \in L$, then M accepts w (and halts).
- If $w \notin L$, then M does not accept w but eventually halts.

Such a Turing machine corresponds to our informal notion of an "algorithm".

The problem (of acceptance of L) is *decidable* if L is recursive, and *undecidable* otherwise.

Classes of Languages

- Recursive = decidable:
  their Turing machine always halt.

- Recursively enumerable but not recursive:
  their Turing machines halt if they accept.
  Example: L_u.

- Non recursively enumerable (non-RE):
  there are no Turing machines for them.
  Example: L_d.

Property of Recursive Languages

The recursive languages are closed under complementation:

Theorem 9.3: If L is a recursive language, then $\bar{L}$ is recursive.

Proof: If L is recursive, then $L = L(M)$ for some Turing machine M that always halts. Transform M into M' such that M' accepts what M does not accept, and vice versa. So M' always halts and accepts $\bar{L}$. Hence $\bar{L}$ is recursive.

Consequence: If L is RE, but $\bar{L}$ is not RE, then L cannot be recursive.

Property of RE Languages

Theorem 9.4: If L and $\bar{L}$ are RE, then L is recursive (and so is $\bar{L}$, by Theorem 9.3).

Proof: Let $L = L(M_1)$ and $\bar{L} = L(M_2)$. Construct a Turing machine M that simulates M_1 and M_2 in parallel (using two tapes and two heads). If the input to M is in L, then M_1 accepts it and halts, hence M also accepts it and halts. If the input to M is not in L, then M_2 accepts it and halts, hence M halts without accepting it. Hence M halts on every input and $L(M) = L$, so L is recursive.

L and $\bar{L}$

There are only four ways of placing L and $\bar{L}$:

- Both L and $\bar{L}$ are recursive.

- Neither L nor $\bar{L}$ is RE.

- L is RE but not recursive, and $\bar{L}$ is not RE.

- $\bar{L}$ is RE but not recursive, and L is not RE.

Indeed, it is impossible that one language (L or $\bar{L}$) is recursive and the other is in either of the other two classes (by Theorem 9.3).

It is also impossible that both languages are RE but not recursive (by Theorem 9.4).

The Universal Language

The universal language L_u is the set of binary strings that encode a pair (M, w) (by putting 111 between the code for M and w) where $w \in L(M)$.

There is a Turing machine U, often called the *universal Turing machine*, such that $L_u = L(U)$. It has three tapes: one for the code of (M, w), one for the code of the simulated tape of M, and one for the code of the state of M. Thus U simulates M on w, and U accepts (M, w) if and only if M accepts w. Hence L_u is RE.

Any Turing machine M may not halt when the input string w is not in the language, thus U will have the same behaviour as M on w. Hence L_u is RE but not recursive.

The Halting Problem

Given a Turing machine M, define $H(M)$ to be the set of strings w such that M halts on input w, regardless of whether or not M accepts w.

The *halting problem* is the set of pairs (M, w) such that $w \in H(M)$. This problem (or language) also is recursively enumerable but not recursive.

Closure Properties of Recursive Languages

The recursive languages are closed under the following operations:

- Union.
- Intersection.
- Concatenation.
- Kleene closure.

Closure Properties of RE Languages

The recursively enumerable (RE) languages are closed under the following operations:

- Union.
- Intersection.
- Concatenation.
- Kleene closure.

Recursive and RE Languages

Given a recursive language and a recursively enumerable (RE) language:

- Union: RE.
- Intersection: RE.
- Concatenation: RE.
- Kleene closure: RE.
- If L_1 is recursive and L_2 is RE, then $L_2 - L_1$ is RE and $L_1 - L_2$ is not RE.

Problem Reduction

Recall "Problem Reduction" of Chapter 8.

If P_1 reduces to P_2,
then P_2 is at least as hard as P_1.

Theorem 9.7: If P_1 reduces to P_2, then:

- If P_1 is undecidable, then so is P_2.
- If P_1 is non-RE, then so is P_2.

Examples of Undecidable Problems

Theorem 9.11: All non-trivial properties of RE languages are undecidable. (Rice, 1953)

All problems about Turing machines that involve only the *language* that the TM accepts are undecidable.

Examples:

Is the language accepted by the TM empty?

Is the language accepted by the TM finite?

Is the language accepted by the TM regular?

Is the language accepted by the TM a CFL?

Does the language accepted by the TM contain the string "ab"?

Does the language accepted by the TM contain all even numbers?

Example of Decidable Problems

Fortunately, not everything is undecidable! Some problems about the *states* of the Turing machine, rather than about the language it accepts, *are* decidable.

Examples:

Does the TM have five states?

Is there an input such that the TM makes at least five moves?