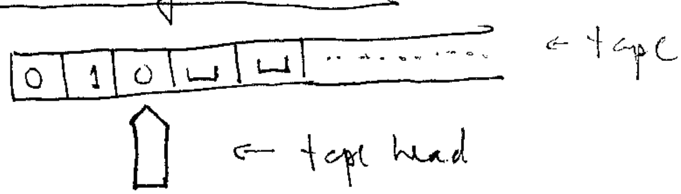
Turing machines

<u>Task</u> look at a more realistic model of a computer than a finite state acceptor.

Turing machines were first proposed by Alan Turing in 1936 in order to explore the theoretical limits of computation. We shall see that certain problems cannot be solved even by a Turing machine and are thus beyond the limits of computation.

A Turing machine is similar to a finite state acceptor but has unlimited memory given by an infinite tape (we mean countably infinite here). The infinite tape is divided into cells each of which holds character of a tape alphabet. The Turing machine is equipped with a tape head that can read and write symbols on the tape and move left (back) or right (forward) on the tape. Initially, the tape contains only the input string and is blank everywhere else. To store information, the Turing machine can write this information on the tape. To read information that it has written, the Turing machine can move its head back over it. The Turing machine continues computing until it decides to produce an output. The outputs "accept" and "rejects" are obtained by entering accepting or rejecting states respectively. It is also possible for the Turing machine to go on forever never stopping if it does not enter either an accepting or a rejecting state.

<u>Illustration of a Turing machine</u>



␣ the blank symbol is part of the tape alphabet

<u>Example</u> Let $A = \{0, 1\}$ and let $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$.
We know L is not a regular language, so there is no finite state acceptor that can recognize it, but there is a Turing machine that can.

<u>Initial state of the tape</u>: input string of 0's and 1's, then infinitely many blanks

<u>Idea of this Turing machine</u> Change a 0 to an X, and then a 1 to a Y until either
- all 0's and 1's have been matched, hence <u>ACCEPT</u>
- the 0's and 1's do not match or the string does not have the form $0^* 1^*$, hence REJECT.

<u>Algorithm</u>

The tape head is initially positioned over the first cell.
1. If anything other than 0 is in the first cell, then REJECT.
2. If 0 is in the cell, then change 0 to X.
3. Move right to the first 1. If none, then REJECT.
4. Change 1 to Y.
5. Move left to the leftmost 0. If none, move right looking for either a 0 or a 1. If either 0 or 1 is found before the first blank symbol, then REJECT; otherwise, ACCEPT.
6. Go to step 2.

Let's process some strings:

<u>Input</u>

```
0 0 1 1 ⊔
X 0 1 1 ⊔
X 0 1 1 ⊔
X 0 Y 1 ⊔
X 0 Y 1 ⊔
```

we continue here

```
X X Y 1 ⊔
X X Y 1 ⊔
X X Y Y ⊔
X X Y Y ⊔
X X Y Y ⊔
```

<u>Outcome</u> ACCEPT (step 5)

Input 0 0 1 ⊔

X 0 1 ⊔

X 0 1 ⊔

X 0 Y ⊔

X 0 Y ⊔

X X Y ⊔

X X Y ⊔ Outcome REJECT (step 3)

Input 0 1 1 ⊔

X 1 1 ⊔

X 1 1 ⊔

X Y 1 ⊔

X Y 1 ⊔

X Y 1 ⊔ Outcome REJECT (step 5)

Input 0 1 0 ⊔

X 1 0 ⊔

X 1 0 ⊔

X Y 0 ⊔

X Y 0 ⊔

X Y 1 ⊔ Outcome REJECT (step 5)

Note that we have the following:

$A = \{0, 1\}$ the input alphabet

$\sqcup \notin A$, where $\sqcup$ is the blank symbol.

$\tilde{A} = \{0, 1, X, Y, \sqcup\}$ is the tape alphabet

S a set of states.

Note also that the tape head is moving right or left, so we also need to have a set $\{L, R\}$ w/ L for left and R for right specifying where the tape head goes.

Recall that a finite state acceptor was given by (S, A, i, t, F)

set of states — alphabet — initial state — transition mapping — set of finishing states

with the transition mapping being given by $t : S \times A \to S$.

By contrast, for a Turing machine the transition mapping is of the form $t : S \times \tilde{A} \to S \times \tilde{A} \times \{L, R\}$

indicates the Turing machine can write

indicates the Turing machine's head can move left or right.

Def A Turing machine is a 7-tuple $(S, A, \tilde{A}, t, i, s_{accept}, s_{reject})$, where $S, A, \tilde{A}$ are finite sets and

(a) S is the set of states

(b) A is the input alphabet not containing the blank symbol $\sqcup$

(c) $\tilde{A}$ is the tape alphabet, where $\sqcup \in \tilde{A}$ and $A \subseteq \tilde{A}$.

(d) $t : S \times \tilde{A} \to S \times \tilde{A} \times \{L, R\}$ is the transition mapping

(e) i is the initial state of the machine.

(f) $s_{accept} \in S$ is the accept state.

(g) $s_{reject} \in S$ is the reject state and $s_{accept} \neq s_{reject}$.

Remarks about the definition

1) Since A does not contain the blank symbol $\sqcup$, the first blank on the tape marks the end of the input string.

2) If the Turing machine is instructed to move left, and it has reached the first cell of the tape, then it stays at the first cell.

3) The Turing machine continues to compute until it enters either the accept or reject states at which point it halts. If it doesnot enter either, then it goes on forever.

Example (considered again) $A = \{0, 1\}$ $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$

We need to be able to write down the transition mapping hence the set of states S. Recall that what we gave was an algorithm, and using that algorithm we processed strings to convince ourselves that the corresponding Turing machine behaved correctly.

Here is the algorithm again:

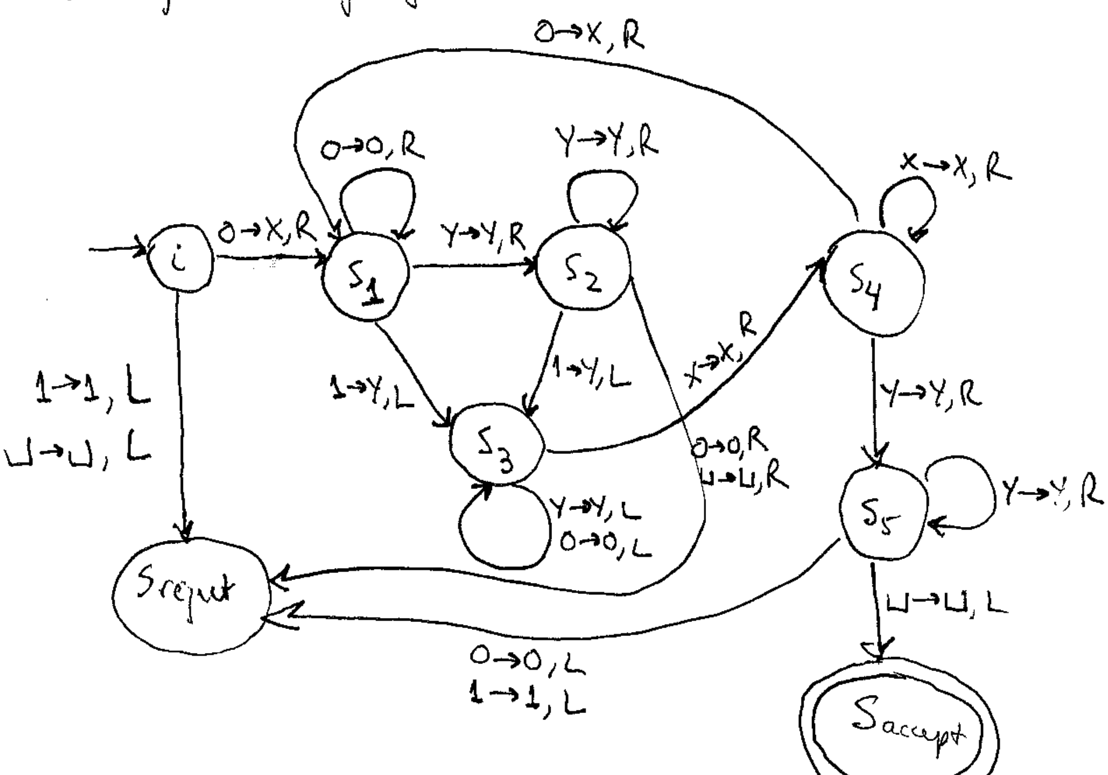
The tape head is initially positioned over the first cell.

1. If anything other than 0 is in the first cell, then REJECT.
2. If 0 is in the cell, then change 0 to X.
3. Move right to the first 1. If none, then REJECT.
4. Change 1 to Y.

5. Move left to the leftmost 0. If none, move right looking for either a 0 or a 1. If either 0 or 1 is found before the first blank symbol, then REJECT; otherwise, ACCEPT.

6. Go to step 2.

Before we can write down the set of states S or the transition mapping t, let us draw a <u>transition diagram</u> which is the Turing machine equivalent to drawing a finite state acceptor when we looked at regular languages.



$i \to S_{reject}$ represents step 1 of the algorithm

$i \to S_1$ and $S_4 \to S_1$ represent step 2 of the algorithm ($i \to S_1$ at the first pass through the string; $S_4 \to S_1$ at subsequent passes)

$S_1 \to S_1$, $S_1 \to S_2$, $S_2 \to S_2$ represent the first part of step 3.

$S_2 \to S_{reject}$ represents the second part of step 3.

$S_1 \to S_3$ and $S_2 \to S_3$ represent step 4.

$S_3 \to S_3$ and $S_3 \to S_4$ represent the first sentence in step 5.

$S_4 \to S_4$, $S_4 \to S_5$, $S_5 \to S_5$ represent the second sentence in step 5.

$S_5 \to S_{reject}$ is the first half of the third sentence in step 5.

$S_5 \to S_{accept}$ is the second half of the third sentence in step 5.

$S_4 \to S_1$ represents step 6.

We have accounted for all pieces of our algorithm. Therefore, we have written down a Turing machine where $A = \{0,1\}$, $\tilde{A} = \{0,1,x,y,\sqcup\}$

blank symbol

$S = \{i, s_{accept}, s_{reject}, s_1, s_2, s_3, s_4, s_5\}$

i is the initial state; $s_{accept} \in S$ is the accept state; $s_{reject} \in S$ is the reject state.

We just have to write down the transition mapping $t: S \times \tilde{A} \to S \times \tilde{A} \times \{L,R\}$

$$\left.\begin{aligned} t(i,0) &= (s_1, X, R) \\ t(i,1) &= (s_{reject}, 1, L) \\ t(i,\sqcup) &= (s_{reject}, \sqcup, L) \end{aligned}\right\}$$

These are the only 3 transitions possible out of state i, but $t: S \times \tilde{A} \to S \times \tilde{A} \times \{L,R\}$ so technically, to write down the full transition mapping, we must assign triplets in $S \times \tilde{A} \times \{L,R\}$ even to inputs from $\tilde{A}$ that cannot occur when in i:

$$t(i,x) = (s_{reject}, x, L)$$

← we assign s_{reject}, same element of $\tilde{A}$, and one of the allowable tape head directions

$$t(i,y) = (s_{reject}, y, L)$$

Technically, the Turing machine halts when it enters either an accepting state (s_{accept}) or a rejecting state (s_{reject}), so in practice we can define

$$\tilde{S} = \{i, s_1, s_2, s_3, s_4, s_5\} = \underbrace{S \setminus \{s_{accept}, s_{reject}\}}_{\text{set of nonhalting states}}$$

and $t: \tilde{S} \times \tilde{A} \to S \times \tilde{A} \times \{L,R\}$, so we avoid writing down the transitions from s_{accept} and s_{reject}.

We only have states s_1, s_2, s_3, s_4, and s_5 left.

$$\left.\begin{aligned} t(s_1,0) &= (s_1, 0, R) \\ t(s_1,Y) &= (s_2, Y, R) \\ t(s_1,1) &= (s_3, Y, L) \end{aligned}\right\} \text{on the diagram}$$

$$\left.\begin{aligned} t(s_1,X) &= (s_{reject}, X, R) \\ t(s_1,\sqcup) &= (s_{reject}, \sqcup, R) \end{aligned}\right\} \text{not on the diagram; cannot occur, so added for completeness}$$

$$\left.\begin{aligned} t(s_2, Y) &= (s_2, Y, R) \\ t(s_2, 1) &= (s_3, Y, L) \\ t(s_2, 0) &= (s_{reject}, 0, R) \\ t(s_2, \sqcup) &= (s_{reject}, \sqcup, R) \end{aligned}\right\} \text{ on the diagram; can occur}$$

$t(s_2, X) = (s_{reject}, X, R)$ ← not on the diagram; cannot occur; added for completeness

$$\left.\begin{aligned} t(s_3, Y) &= (s_3, Y, L) \\ t(s_3, 0) &= (s_3, 0, L) \\ t(s_3, X) &= (s_4, X, R) \end{aligned}\right\} \text{ on the diagram; can occur}$$

$$\left.\begin{aligned} t(s_3, \sqcup) &= (s_{reject}, \sqcup, R) \\ t(s_3, 1) &= (s_{reject}, 1, R) \end{aligned}\right\} \text{ not on the diagram; cannot occur; added for completeness}$$

$$\left.\begin{aligned} t(s_4, X) &= (s_4, X, R) \\ t(s_4, Y) &= (s_5, Y, R) \\ t(s_4, 0) &= (s_1, X, R) \end{aligned}\right\} \text{ on the diagram; can occur}$$

$$\left.\begin{aligned} t(s_4, 1) &= (s_{reject}, 1, R) \\ t(s_4, \sqcup) &= (s_{reject}, \sqcup, R) \end{aligned}\right\} \text{ not on the diagram; cannot occur; added for completeness}$$

$$\left.\begin{aligned} t(s_5, Y) &= (s_5, Y, R) \\ t(s_5, \sqcup) &= (s_{accept}, \sqcup, L) \\ t(s_5, 0) &= (s_{reject}, 0, L) \\ t(s_5, 1) &= (s_{reject}, 1, L) \end{aligned}\right\} \text{ on the diagram; can occur}$$

$t(s_5, X) = (s_{reject}, X, L)$ ← not on the diagram; cannot occur; added for completeness.

Moral of the story

The transition mapping is a very inefficient way of specifying a Turing machine as a lot of transitions cannot occur unlike what we saw for a finite state acceptor, where the input alphabet was exactly the alphabet of the language. Here $A \subset \bar{A}$. Therefore, we will specify a Turing machine via either an algorithm or the transition diagram <u>only</u>.

To figure out which languages are recognized by a Turing Machine, we need to introduce the notion of a configuration. As a Turing machine goes through its computations, changes take place in

1. The state of the machine
2. the tape contents
3. the tape head location

a setting of these three items is called a <u>configuration</u>

<u>Representing configurations</u> We represent a configuration as us_iv, where u, v are strings in the tape alphabet $\tilde{A}$ and s_i is the current state of the machine. The tape contents are then the string uv and the current location of the tape head is on the first symbol of v. The assumption here is that the tape contains only blanks after the last symbol in v.

<u>Example</u> $\varepsilon i 001$ is the configuration

0	0	1	⊔	...

⇧ tape head

as we start examining the string 001 in our previous example of a Turing machine.

<u>Def</u> Let C_1, C_2 be two configurations of a given Turing machine. We say that the configuration C_1 <u>yields</u> the configuration C_2 if the Turing machine can go from C_1 to C_2 in one step.

<u>Example</u> If s_i, s_j are states, u and v are strings in the tape alphabet $\tilde{A}$, and $a, b, c \in \tilde{A}$. A configuration $C_1 = uas_ibv$ yields a configuration $C_2 = us_jacv$ if the transition mapping t specifies a transition $t(s_i, b) = (s_j, c, L)$. In other words, the Turing machine is in state s_i, it reads character b, writes character c in its place, enters state s_j, and its head moves left.

<u>Types of configurations</u>

- <u>initial configuration</u> with input u is iu, which indicates that the machine is in the initial state i with its head at the leftmost position on the tape (which is the reason why this configuration has no string left of the state).
- <u>accepting configuration</u> $us_{accept}v$ for $u, v \in \tilde{A}^*$ (u, v string in $\tilde{A}$),

namely the machine is in the accept state.

- rejecting configuration $u s_{reject} v$ for $u, v \in \tilde{A}^*$, namely the machine is in the reject state.
- halting configurations yield no further configurations; no transitions are defined out of their states. Accepting and rejecting configurations are examples of halting configurations.

Def A Turing machine M accepts input $w \in A^*$ (string over the input alphabet A) if $\exists$ sequence of configurations $C_1, C_2, \ldots, C_k$ such that:

1. C_1 is the start configuration with input w.
2. Each C_i yields C_{i+1} for $i = 1, \ldots, k-1$.
3. C_k is an accepting configuration.

Def Let M be a Turing machine. $L(M) = \{ w \in A^* \mid M \text{ accepts } w \}$ is the language recognized by M.

Def A language $L \subset A^*$ is called Turing-recognizable if $\exists M$ a Turing machine that recognizes L, i.e. $L = L(M)$.

NB Some textbooks use the terminology recursively enumerable language (RE language) instead of Turing-recognizable.

Turing-recognizable is not necessarily as strong a notion as we might need because a Turing machine can
accept
reject
loop

Looping is any simple or complex behaviour that does not lead to a halting state. The problem with looping is that the user does not have infinite time. It can be difficult to distinguish between looping or taking a very long time to compute. We thus prefer deciders.

Def A decider is a Turing machine that enters either an accept state or a reject state for every input in A^*.

Def A decider that recognizes some language $L \subset A^*$ is said to decide that language.

Def A language $L \subset A^*$ is called <u>Turing-decidable</u> if $\exists$ a Turing machine M that decides L.

NB Some textbooks use the terminology <u>recursive language</u> instead of Turing-decidable.

Example $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$ is Turing-decidable because the Turing machine we built that recognized it was in fact a <u>decider</u> (check again to convince yourself that machine did not loop.).

Turing-decidable $\Rightarrow$ Turing-recognizable, but the converse is not true: Turing-recognizable $\not\Rightarrow$ Turing-decidable. We will hopefully have time to cover an example of a language that is Turing-recognizable, but <u>NOT</u> Turing-decidable before the end of the term.

Variants of Turing machines

<u>Task</u> Explore variants of the original set-up of a Turing machine and show they do not enlarge the set of Turing-recognizable languages

(A) Add "stay put" to the list of allowable directions

Say instead of allowing just $\{L, R\}$ (the tape head moves left or right), we also allow the "stay put" option (no change in the position of the tape head). Thus, the transition mapping is defined as $t: S \times \hat{A} \to S \times \hat{A} \times \{L, R, N\}$ where N is for "no movement" (stay put) instead of $t: S \times \hat{A} \to S \times \hat{A} \times \{L, R\}$. We realize N is the same as $L + R$ or $R + L$ (move the tape head left by one cell, then right by one cell or the other way around) $\Rightarrow$ variant (A) yields no increase in computational power.

(B) Multitape Turing machine

We allow the Turing machine to have several tapes, each with its own tape head for reading and writing. Initially, the input is on tape 1, and the others are blank. The transition mapping then must allow for reading, writing, and moving the tape heads on some or all of the tapes simultaneously. If k is the number of tapes, then the transition mapping is defined as $t: S \times \underbrace{\tilde{A}^k}_{\substack{k\text{-fold Cartesian product} \\ \underbrace{\tilde{A} \times \dots \times \tilde{A}}_{k \text{ times}}}} \to S \times \underbrace{\tilde{A}^k}_{\substack{k\text{-fold Cartesian product} \\ \underbrace{\tilde{A} \times \dots \times \tilde{A}}_{k \text{ times}}}} \times \underbrace{\{L, R, N\}^k}_{\substack{k\text{-fold Cartesian product} \\ \underbrace{\{L,R,N\} \times \dots \times \{L,R,N\}}_{k \text{ times}}}}$

Since one of the tape heads or more might not move for some transitions, we make use of the option N ("no movement") besides left and right.

Multitape Turing machines seem more powerful than ordinary (single-tape) ones, but that is not the case.

<u>Def</u> We call two Turing machines M_1 and M_2 <u>equivalent</u> if $L(M_1) = L(M_2)$, namely if they recognize the same language.

<u>Theorem</u> Every multitape Turing machine has an equivalent single-tape Turing machine.

<u>Sketch of proof</u> Let $M^{(k)}$ be a Turing machine w/ k tapes. We will simulate it with a single-tape Turing machine $M^{(1)}$ constructed as follows. We add # to the tape alphabet $\tilde{A}$ and use it to separate the contents of the different tapes. $M^{(1)}$ also needs to keep track of the locations of the tape heads of $M^{(k)}$. It does so by adding a dot to the character to which a tape head is pointing. We thus only need to enlarge the tape alphabet $\tilde{A}$ by allowing a version with a dot above for every character in $\tilde{A}$ apart from # and the blank symbol $\sqcup$.

<u>Corollary</u> A language L is Turing-recognizable (T.-rec.) $\iff$ some multitape Turing machine recognizes L.

<u>Proof</u> "$\Rightarrow$" A language L is Turing-recognizable if $\exists$ M a single-tape

Turing machine that recognizes it. A single-tape Turing machine is a special type of a multitape Turing machine, so we are done.
"$\Longleftarrow$" follows from the previous Theorem! (q.e.d.)

(C) A nondeterministic Turing machine

Just like a nondeterministic finite state acceptor, a nondeterministic Turing machine may proceed according to different possibilities, so its computation is a tree, where each branch corresponds a different possibility. The transition mapping of such a nondeterministic Turing machine is given by $t : S \times \widehat{A} \to \mathcal{P}(S \times \widehat{A} \times \{L, R\})$

shows we have different possibilities on how to proceed.

Theorem Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Idea of the proof We construct a deterministic Turing machine that simulates the nondeterministic one by trying out all possible branches. If it finds an accept state on one of these computational branches, it accepts the input; otherwise, it loops.

Corollary A language is Turing-recognizable $\Longleftrightarrow$ some nondeterministic Turing machine recognizes it.

Proof "$\Longrightarrow$" A deterministic Turing machine is a nondeterministic one, so this direction is obvious.
"$\Longleftarrow$" follows from the previous Theorem.

(D) Enumerators

As we saw, a Turing-recognizable language is called in some textbooks a recursively enumerable language. The term comes from a variant of a Turing machine called an enumerator. Loosely, an enumerator is a Turing machine with an attached printer.

The enumerator print out the language L it accepts as a sequence of strings. Note that the enumerator can print out the strings of the language in any order and possibly with repetitions.

<u>Theorem</u> A language L is Turing-recognizable $\Leftrightarrow$ some enumerator enumerates (outputs) L.

<u>Proof</u> "$\Leftarrow$" Let E be the enumerator. We construct the following Turing machine M:

M = on input w

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, accept w.

Thus, M accepts exactly those strings that appear on E's list and no others, hence exactly L.

"$\Rightarrow$" Let M be a Turing machine that recognizes L. We would like to construct an enumerator E that outputs L. Let A be the alphabet of L, i.e. $L \subset A^*$. In the unit on countability, we proved A^* is countably infinite (note that the alphabet A is always assumed to be finite), so A^* has an enumeration as a sequence $A^* = \{w_1, w_2, \dots\}$

E = Ignore the input

1. Repeat the following for $i = 1, 2, 3, \dots$
2. Run M for i steps on each input $w_1, w_2, \dots, w_i$
3. If any computations accept, print out the corresponding w_j.

Every string accepted by M will eventually appear on the list of E, and once it does, it will appear infinitely many times because M runs from the beginning on each string for each repetition of step 1. Note that each string accepted by M is accepted in some finite number of steps, say k steps, so this string will be printed on E's list for every $i \geq k$.

(q.e.d.)

Moral of the story

The single-tape Turing machine we first introduced is as powerful as any variants we can think of.

Algorithms

Task Use Hilbert's 10^{th} problem to give an example of something that is Turing-recognizable but not Turing-decidable.

We saw that the Continuum Hypothesis of Cantor was the 1^{st} of Hilbert's 23 problems in 1900 at the International Congress of Mathematicians.

Hilbert's 10^{th} problem

Find a procedure that tests whether a polynomial in several variables with integer coefficients has integer roots.

Example. $p(x,y) = 2x^2 - xy - y^2$ is a polynomial in 2 variables (x and y) with integer coefficients $(2,-1,-1)$ that has integer roots

$p(1,1) = 2\cdot 1^2 - 1\cdot 1 - 1^2 = 0$ so $x = 1 = y$, $1 \in \mathbb{Z}$ is a solution.

Hilbert's problem asked how to find integer roots via a set procedure.

In 1936 independently Alonzo Church invented λ-calculus to define algorithms, while Alan Turing invented Turing machines. Church's definition was shown to be equivalent to Turing's. This equivalence says

$$\boxed{\text{Intuitive notion of algorithms}} = \boxed{\text{Turing machine algorithms}}$$

and is known as the Church-Turing Thesis. It led to the formal definition of an algorithm and eventually to resolving in the negative Hilbert's 10^{th} problem. Using previous work by Martin Davis, Hilary Putnam, and Julia Robinson, Yuri Matijasevič proved in 1970 that there is no algorithm which can decide whether a polynomial has integer roots. As we shall see now, Hilbert's 10^{th} problem is

An example of a problem that is Turing-recognizable but not Turing-decidable. Let $D = \{ p \mid p$ is a polynomial with an integer root$\}$. Hilbert's 10^{th} problem is asking whether D is decidable.

Let us simplify the problem to the one variable case:

$$D_1 = \{ p \mid p \text{ is a polynomial in variable } x \text{ with an integer root} \}.$$

We can easily write down a Turing machine M_1 that recognizes D_1:

M_1 = On input p, where p is a polynomial in x

1. Evaluate p with x set successively to the values $0, 1, -1, 2, -2, \ldots$ If at any value the polynomial evaluates to 0, accept.

If p does indeed have an integer root, M_1 will eventually find it and accept p. If p does not have an integer root, then M_1 will run forever.

Principle behind M_1: $\mathbb{Z} \sim \mathbb{N}$, i.e. $\mathbb{Z}$ is countably infinite, so we can write $\mathbb{Z}$ as a sequence (enumerate it) $\mathbb{Z} = \{ s_1, s_2, \ldots \} = \{ s_i \}_{i=1,2,\ldots}$ $= \{ 0, 1, -1, 2, -2, \ldots \}$

Now, consider polynomials of n variables $p(x_1, \ldots, x_n)$. We want to find $(x_1, \ldots, x_n) \in \mathbb{Z}^n$ such that $p(x_1, \ldots, x_n) = 0$, so in general Hilbert's 10^{th} problem is asking us to build a decider for

$$D_n = \{ p(x_1, \ldots, x_n) \mid \exists (x_1, \ldots, x_n) \in \mathbb{Z}^n \text{ such that } p(x_1, \ldots, x_n) = 0 \}.$$

We can easily build a Turing machine M_n that recognizes D_n using the principle behind M_1: $\mathbb{Z}^n$ is countably infinite because it is the Cartesian product of a countably infinite set with itself n times. Since $\mathbb{Z}^n$ is countably infinite, we can enumerate it, namely write it as a sequence $\mathbb{Z}^n = \{ c_1, c_2, \ldots \}$, where $c_i = (x_1^{(i)}, \ldots, x_n^{(i)})$.

Then M_n = On input p, where p is a polynomial in $x_1, \ldots, x_n$

1. Evaluate p with $(x_1, \ldots, x_n)$ set successively to the values $c_1, c_2, \ldots$. If at any value $c_i = (x_1^{(i)}, \ldots, x_n^{(i)})$, $p(x_1^{(i)}, \ldots, x_n^{(i)}) = 0$, accept p.

If p has an integer root $(x_1^{(i)}, \ldots, x_n^{(i)}) \in \mathbb{Z}^n$, then the Turing machine accepts; otherwise, it goes on forever (it loops) just like M_1. It turns out M_1 can be converted into a decider because if $p(x)$ of one variable has a root, then that root must fall between certain bounds, so the checking of possible values can be made to terminate when those bounds are reached. By contrast, no such bounds exist when the polynomial is of two variables or more $\Rightarrow$ M_n for $n \geq 2$ CANNOT be converted into a decider. This is what Matijasević proved.

$\boxed{\text{Decidable languages}}$

Task: Explore whether certain languages are decidable that come from our study of formal languages and grammars.

① The acceptance problem for deterministic finite state acceptors (DFA's)

Test whether a given deterministic finite state acceptor (DFA) B accepts a given string w.

We can rewrite the acceptance problem as a language:

$$L_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$$

Theorem: L_{DFA} is a Turing-decidable language.

Proof: We construct a Turing machine M that decides L_{DFA} as follows: M = on input $\langle B, w \rangle$, where B is a DFA and w is a string

1. Simulate B on input w.
2. If the simulation ends in an accept state of B, accept $\langle B, w \rangle$. If it ends in a non-accepting state of B, reject $\langle B, w \rangle$.

We need to provide more details on the input $\langle B, w \rangle$. B is a finite state acceptor, which we defined as a 5-tuple (S, A, i, t, F) w/ S the set of states, A the alphabet, i the initial state, t the transition mapping $t: S \times A \to S$, and F the set of finishing states. The string w is over the alphabet A, so the pair $\langle B, w \rangle$ as input for our Turing machine is in fact $(S, A, i, t, F; w)$. The Turing machine M starts in the configuration $\varepsilon i w$ (remind yourself what a configuration is). If $w = uv$, where $u \in A$ is the first character in the word w and if $t(i,u) = s$, then the next configuration of the Turing machine M is usv, i.e. the new state corresponds to the state s in which B enters from the initial state i upon receiving input character u and the tape head has moved right past u ready to examine the second character of w. Once the string w has been completely processed, then the configuration of the Turing machine is $w s_w \varepsilon$. If the final state s_w where we ended up is an accepting state, i.e. $s_w \in F$, then we accept $\langle B, w \rangle$; otherwise, we reject $\langle B, w \rangle$. (q.e.d.)

② <u>The acceptance problem for nondeterministic finite state acceptors (NFA's)</u>

Test whether a given nondeterministic finite state acceptor B accepts a given string w.

Rewrite this acceptance problem as a language:

$L_{NFA} = \{ \langle B, w \rangle \mid B$ is a NFA that accepts input string $w \}$.

<u>Theorem</u> L_{NFA} is a Turing-decidable language.

<u>Proof</u> This result is in fact a corollary to the previous theorem. As we showed in our unit on formal languages and grammars, given any NFA B, $\exists$ a deterministic finite state acceptor (DFA) B' that corresponds to it (with potentially many more states). Therefore, to any pair $\langle B, w \rangle \in L_{NFA}$, there corresponds a pair $\langle B', w \rangle \in L_{DFA}$. Since L_{DFA} is a Turing-decidable language, L_{NFA} is Turing-decidable as well. (q.e.d.).

③ <u>The acceptance problem for regular expressions</u>

Test whether a regular expression R generates a string w.

We rewrite this acceptance problem as the language
$L_{REX} = \{\langle R, w\rangle \mid R$ is a regular expression that generates string $w\}$.

Theorem L_{REX} is a Turing-decidable language.

Proof Recall that a language L is regular $\Leftrightarrow$ L is accepted by a deterministic or nondeterministic finite state acceptor $\Leftrightarrow$ L is given by a regular expression. There exists an algorithm to construct a nondeterministic finite state acceptor from any given regular expression $\Rightarrow \forall \langle R, w\rangle \in L_{REX}$, $\exists \langle B, w\rangle \in L_{NFA}$ that corresponds to it. Since L_{NFA} is Turing-decidable, L_{REX} is Turing decidable.

(q.e.d.)

④ Emptiness testing for the language of an automaton

Given a DFA B, figure out whether the language recognized by B, $L(B)$ is empty or not, i.e. whether $L(B) \neq \emptyset$ or $L(B) = \emptyset$.
Rewrite the emptiness testing problem as a language:

$$E_{DFA} = \{\langle B\rangle \mid B \text{ is a DFA and } L(B) = \emptyset\}.$$

Theorem E_{DFA} is a Turing-decidable language.

Proof A DFA B accepts a certain string w if we are in an accepting state when the last character of w has been processed. We design a Turing machine M to test this condition as follows:

M = On input $\langle B\rangle$, where B is a DFA:

1. Mark the initial state of B.
2. Repeat until no new states of B get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, then accept; otherwise, reject.

We have thus marked all states of B where we can end up given an input string. If no such state is an accepting state, then B will not accept any string, i.e. $L(B) = \emptyset$ as needed.

(q.e.d.)

(5) Checking whether two given DFA's accept the same language

Given B_1, B_2 DFA's, test whether $L(B_1) = L(B_2)$.

We rewrite this problem as the language

$$EQ_{DFA} = \{\langle B_1, B_2 \rangle \mid B_1 \text{ and } B_2 \text{ are DFA's and } L(B_1) = L(B_2)\}.$$

<u>Theorem</u> EQ_{DFA} is a Turing-decidable language.

<u>Proof</u> Given two sets Γ and Σ, $\Gamma \neq \Sigma$ if $\exists\, x \in \Gamma$ such that $x \notin \Sigma$ (i.e. $\Gamma \setminus \Sigma \neq \emptyset$) or $\exists\, x \in \Sigma$ such that $x \notin \Gamma$ (i.e. $\Sigma \setminus \Gamma \neq \emptyset$). Recall from our unit on set theory that $\Gamma \setminus \Sigma = \Gamma \cap \overline{\Sigma}$, Γ intersect the complement of Σ. Similarly, $\Sigma \setminus \Gamma = \Sigma \cap \overline{\Gamma}$. Therefore, $\Gamma \neq \Sigma$ $\Longleftrightarrow (\Gamma \cap \overline{\Sigma}) \cup (\Sigma \cap \overline{\Gamma}) \neq \emptyset$. This expression is called the <u>symmetric difference</u> of sets Γ and Σ in set theory. Now, returning to our problem, note that B_1 and B_2 are DFA's $\Rightarrow L(B_1)$ and $L(B_2)$ are regular languages. Furthermore, we showed the set of regular languages is closed under union, intersection, and the taking of complements $\Rightarrow \left(L(B_1) \cap \overline{L(B_2)}\right) \cup \left(L(B_2) \cap \overline{L(B_1)}\right)$ is a regular language $\Rightarrow \exists\, C$ a DFA that recognizes the symmetric difference of $L(B_1)$ and $L(B_2)$ $\left(L(B_1) \cap \overline{L(B_2)}\right) \cup \left(L(B_2) \cap \overline{L(B_1)}\right)$. $L(B_1) = L(B_2)$ if this symmetric difference is empty $\Rightarrow \forall \langle B_1, B_2 \rangle \in EQ_{DFA}$ $\exists \langle C \rangle \in E_{DFA}$, the language corresponding to the emptiness testing problem. Since E_{DFA} is Turing-decidable, EQ_{DFA} is Turing-decidable.

q.e.d.)

Next, we look at context-free grammars (CFG's) that we studied last term.

(6) $L_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG and } w \text{ is a string}\}$

<u>Theorem</u> L_{CFG} is a Turing-decidable language.

<u>Sketch of proof</u> We could try to go through all possible applications of production rules allowable under G to see whether we can generate w, but infinitely many derivations may need to be tried. Therefore,

if G does not generate w, our algorithm would not halt. We would thus have a Turing machine that is a recognizer but not a decider. To get a decider, we have to put G into a special form called a Chomsky normal form that takes $2n-1$ steps to generate a string w of length n. We do not need to know what a Chomsky normal form is, just that one exists in order to write down our decider M:

M = on input $\langle G, w \rangle$, where G is a context-free grammar and w is a string.

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w if $n>0$. If $n=0$, list all derivations with one step.
3. If any of these derivations generates w, then accept; otherwise, reject.

q.e.d.

(7) Emptiness testing for context-free grammars

Given a context-free grammar G, figure out whether the language it generates L(G) is empty or not.

Rewrite as a language $E_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \}$

Theorem E_{CFG} is a Turing-decidable language.

Proof We use a similar marking argument as we did to show E_{DFA} was Turing-decidable. We outline the Turing machine as

M = on input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G
2. Repeat until no new variables get marked:
3. Mark any nonterminal $\langle T \rangle$ if G contains a production rule $\langle T \rangle \to u_1 \cdots u_k$, and each symbol (terminal or non-terminal) $u_1, \dots, u_k$ has already been marked.

4. If the start symbol $\langle S \rangle$ is not marked, accept; otherwise, reject.

As we can see from step 4, if $\langle S \rangle$ is marked, then the context-free grammar will end up generating at least one string as all terminals have already been marked in step 1. Therefore, $L(G) \neq \emptyset$, and we reject G.

(q.e.d.)

⑧ <u>Equivalence problem for context-free grammars</u>

Given two context-free grammars, G_1 and G_2, determine whether they generate the same language, i.e. $L(G_1) = L(G_2)$.

Rewrite this problem as a language:

$$EQ_{CFG} = \{ \langle G_1, G_2 \rangle \mid G_1 \text{ and } G_2 \text{ are CFG's and } L(G_1) = L(G_2) \}.$$

To solve the equivalence problem for DFA's, we used the symmetric difference and the fact that the emptiness problem for DFA's is Turing-decidable. In this case, the emptiness problem for CFG's is Turing-decidable as we just proved, but the symmetric difference argument does <u>NOT</u> work as the set of languages produced by context-free grammars is <u>NOT</u> closed under complements or intersection so the following result is true instead:

<u>Proposition</u> EQ_{CFG} is not a Turing-decidable language.

This proposition is proven using a technique called reducibility.

An even more general result is true, the equivalence problem for Turing machines is undecidable:

$$EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are Turing machines and } L(M_1) = L(M_2) \}.$$

<u>Proposition</u> EQ_{TM} is not a Turing-decidable language

This proposition follows from another result, namely that the emptiness testing problem for Turing machines is undecidable:

$$E_{TM} = \{ \langle M \rangle \mid M \text{ is a Turing machine and } L(M) = \emptyset \}.$$

<u>Proposition</u> E_{TM} is not a Turing-decidable language.

Returning to context-free grammars, we now know that L_{CFG} and E_{CFG} are Turing-decidable, but EQ_{CFG} is not.
Recall that a language is called context-free if it can be generated by a context-free grammar.

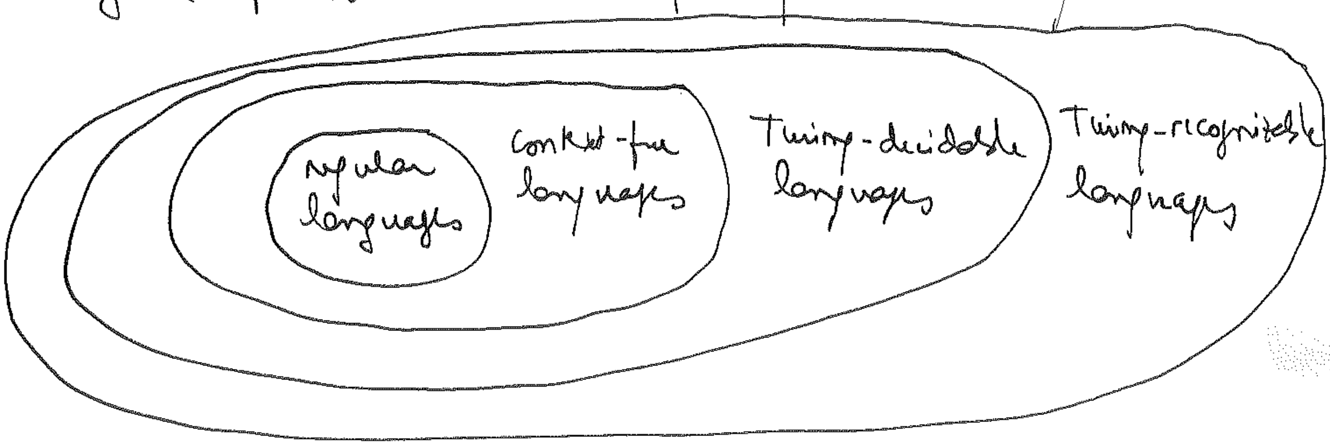
$\boxed{\text{Moral of the story}}$

We now know how the main types of languages relate to each other

{regular languages} $\subset$ {context-free languages} $\subset$ {Turing-decidable languages} $\cap$ {Turing-recognizable languages}

this set includes non regular languages

Visually, we represent the relationship using a Venn diagram:



So Turing machines provide a very powerful computational model. What is surprising is that once we have built a Turing machine to recognize a language, we do not know whether there is a simpler computational model such as a DFA that recognizes the same language. Define

$$\text{REGULAR}_{TM} = \{\langle M\rangle \mid M \text{ is a Turing machine and } L(M) \text{ is a regular language}\}.$$

Theorem REGULAR$_{TM}$ is not a Turing-decidable language.

This Theorem is proven using reducibility.
In fact, even more is true:

Rice's Theorem Any property of the languages recognized by Turing machines is not Turing-decidable.

$\boxed{\text{Undecidability}}$

<u>Task</u> Understand why certain problems are algorithmically unsolvable.

Recall that a Turing machine M is defined as a 7-tuple

$$(S, A, \tilde{A}, t, i, s_{accept}, s_{reject})$$

set of states

input alphabet

tape alphabet

transition mapping $t: S \times \tilde{A} \to S \times \tilde{A} \times \{L, R\}$

initial state

accept state

reject state

<u>Def</u> An <u>encoding</u> $\langle M \rangle$ of a Turing machine M refers to the 7-tuple $(S, A, \tilde{A}, t, i, s_{accept}, s_{reject})$ that defines M and is therefore a finite string.

Recall that earlier in the module we proved the following results:

<u>Theorem</u> If A is a finite alphabet, then the set of all words over A $A^* = \bigcup_{j=0}^{\infty} A^j$ is countably infinite.

<u>Corollary I</u> If A is a finite alphabet, then the set of all languages over A is uncountably infinite.

<u>Corollary II</u> The set of all programs in any programming language is countably infinite.

Recall that we proved Corollary II by realizing that for any programming language, a program is a finite string over the finite alphabet of all allowable characters in that programming language.

<u>Corollary III</u> Given a finite alphabet A, the set of all Turing-recognizable languages over A is countably infinite.

<u>Proof</u> An encoding $\langle M \rangle$ of a Turing machine M is the 7-tuple $(S, A, \tilde{A}, t, i, s_{accept}, s_{reject})$, which is a finite string over a language

B that contains A and is finite. By the theorem, $B^* = \bigcup_{j=0}^{\infty} B^j$ is (72)
countably infinite. Since $\langle M \rangle \in B^*$, there are at most countably
infinitely many Turing machines M that recognize languages over A