A language is *undecidable* if there is no TM that accepts that language, and in addition, always halts and rejects any string that is *not* in the language.
A language is \textit{undecidable} if there is no TM that accepts that language, and in addition, always halts and rejects any string that is \textit{not} in the language.

Next:

- construct an undecidable language.
- show that it really is undecidable
- this show that “recursively enumerable” is not the same as “recursive”
- look at various other undecidable languages - technique of \textit{reduction} allows us to use a previously-obtained undecidability result to deduce that some new language is undecidable.
The general idea (which we are about to do in detail): Choose a sensible way to encode Turing machines as strings of symbols over a fixed alphabet. The language we construct will consist of encodings of TMs that halt on their input.
An undecidable language

The general idea (which we are about to do in detail): Choose a sensible way to encode Turing machines as strings of symbols over a fixed alphabet. The language we construct will consist of encodings of TMs that halt on their input.

Define a “universal” Turing machine that accepts this language — it executes (or simulates) the encoded TM on some input. (Analogy with various programs being executable on the same computer)
A universal TM takes input representing (or encoding) any TM together with an additional input, and simulates the represented TM on the given input.
A universal TM takes input representing (or encoding) any TM together with an additional input, and simulates the represented TM on the given input. Each transition of the encoded machine will correspond to a sequence of transitions of the “universal” machine, the one that accepts encodings of machines that halt.
A universal TM takes input representing (or encoding) any TM together with an additional input, and simulates the represented TM on the given input. Each transition of the encoded machine will correspond to a sequence of transitions of the “universal” machine, the one that accepts encodings of machines that halt.

Notation: $L_{halt}$ denotes representations of TMs/inputs where the represented TM halts given that input.

A universal TM accepts members of $L_{halt}$ but may loop for ever on non-members.
The purpose of the universal TM will be to show that $L_{halt}$ is accepted by a Turing machine. After that, we show that $L_{halt}$ is not accepted by any TM that always halts on non-members of its language.
The purpose of the universal TM will be to show that $L_{halt}$ is accepted by a Turing machine. After that, we show that $L_{halt}$ is not accepted by any TM that always halts on non-members of its language. This shows a distinction between TMs that are guaranteed to halt, and TMs that aren’t.
The purpose of the universal TM will be to show that $L_{halt}$ is accepted by a Turing machine. After that, we show that $L_{halt}$ is \textit{not} accepted by any TM that always halts on non-members of its language. This shows a distinction between TMs that are guaranteed to halt, and TMs that aren’t. (Later, we can construct a formal language that is even “worse” than $L_{halt}$, in that no TM can recognise it, or its complement.)
Defining $L_{halt}$

Given TM with alphabet $\{0, 1, \bar{b}\}$, encode it using alphabet $\{0, 1, B, *\}$ as follows:

States are encoded as members of $\{0, 1\}^k$

- initial state: $0^k$
- single accepting state: $1^k$

Encode direction $L$ as 0, $R$ as 1

Encode transitions $\delta(p, x) = (q, y, D)$ as concatenation of strings for $p, x, q, y, D$

List the transitions, separated by *
Example of encoding

\( M_{\text{pred}} \) computes predecessor of input
1. \( \delta(i, 0) = (i, 0, R) \)
2. \( \delta(i, 1) = (i, 1, R) \)
3. \( \delta(i, \overline{b}) = (p, \overline{b}, L) \)
4. \( \delta(p, 1) = (t, 0, R) \)
5. \( \delta(p, 0) = (p, 1, L) \)
Encode states \( i \) as \( 00 \), \( p \) as \( 01 \), \( t \) as \( 11 \)

\[ 0000001*0010011*00B01B0*0111101*0100110 \]
Input word to universal TM is

$$0^{k+1} \langle M \rangle \# w$$

$M_{pred}$ (with input 1011) becomes:

$$000*0000001*0010011*00B01B0*0111101*0100110\#1011$$
$M_u$ simulates $M$ using everything to right of right-hand $\$\$ as working tape for $M$.
$M_u$ executes a sequence of cycles, where each cycle simulates a single step of $M$.

1. **Read symbol scanned by $M$**
   - Find #
   - Store symbol to right of #
   - Move to left-hand *
   - Print symbol to left of left-hand *
2. **Find next transition to use**
   
   Find sequence of symbols to right of * which matches symbols between initial $ and *

   Halt if no matching string is found
   Accept if LHS of tape is $1^k$

3. **Fetch the new state and symbol**
   
   We have found correct transition to use. Copy the $k + 1$ symbols corresponding to new state and symbol, over to LHS of tape (between initial $ and *).
4. **Printing the new symbol**
Enter state that encodes new symbol and direction to move. Find # and replace symbol to its right with new symbol.

5. **Move the tape head (i.e. $M$’s pointer to its tape)**
If direction (above) is R, switch # with symbol to its right, else switch # with symbol to its left. (Unless that’s a $\$$, in which case halt and accept/reject depending on state of $M$)
The Halting Problem

Given a TM together with input word, will the resulting computation halt? *Note: we don’t care whether it accepts or why it halts*
The Halting Problem

Given a TM together with input word, will the resulting computation halt? *Note: we don’t care whether it accepts or why it halts*

We know how to encode TMs/inputs as strings

Hence we have associated language recognition problem: Does a word represent a TM + input on which it halts?

If so, the language should be accepted by some halting Turing machine $M$. 

$L_{\text{halt}} = \{ \langle N \rangle : u \in \{0,1\}^* \text{ and } N \text{ is a TM which halts on input } u \}$

Note: This is really a result about languages and computation, not Turing machines.
The Halting Problem

Given a TM together with input word, will the resulting computation halt? *Note: we don’t care whether it accepts or why it halts*

We know how to encode TMs/inputs as strings
Hence we have associated language recognition problem: Does a word represent a TM + input on which it halts?

If so, the language should be accepted by some halting Turing machine $M$.

$L_{\text{halt}} = \{ \langle N \rangle u : u \in \{0, 1\}^* \text{ and } N \text{ is a TM which halts on input } u \}$

*Note: This is really a result about languages and computation, not Turing machines.*
Think of a computer program that is supposed to take an initial input and tell you something about it. Given a particular input the program may

- Say that it has some property of interest
- fail to do so
- go into an infinite loop

Can we decide *automatically* when we will get the infinite loop?

You can’t just run the program.

But you might be able to tell in advance, eg some infinite loops in programs are obvious.
“Model of computation” (e.g. choice of programming language) is not important to this question. We may consider programs in some language of our choice.

You cannot automatically identify when an infinite loop will occur. Every program can be assigned a unique number. So can every input.
Entry to table says what happens when any input is given to any program (or, TM)

<table>
<thead>
<tr>
<th>input</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>N</td>
<td>?</td>
<td>N</td>
<td>Y</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now try to define a “program” that accepts a number $n$ whenever the $n$th program in our list fails to accept the $n$th input...
We find that such a program cannot exist.
We find that such a program cannot exist.

However, if we could solve the Halting problem we could in fact use the solution to construct such a program as follows:
We find that such a program cannot exist.

However, if we could solve the Halting problem we could in fact use the solution to construct such a program as follows:

- Use HP solver to decide whether program $n$ halts on input $n$
We find that such a program cannot exist.

However, if we could solve the Halting problem we could in fact use the solution to construct such a program as follows:

- Use HP solver to decide whether program \( n \) halts on input \( n \).
- If not, accept number \( n \).
We find that such a program cannot exist.

However, if we could solve the Halting problem we could in fact use the solution to construct such a program as follows:

- Use HP solver to decide whether program $n$ halts on input $n$.
- If not, accept number $n$.
- If so, simulate program $n$ on input $n$, and reverse the answer given.
Prove by contradiction: Take the imaginary halting TM $M$ that accepts $L_{\text{halt}}$; we will use $M$ to construct an input (to itself) that leads to a contradiction.

Construct $M'$ from $M$: given any input, $M'$ behaves exactly like $M$, but if $M$ accepts, $M'$ is designed to loop for ever instead.
Define machine $M''$ as follows.  
$M''$ with binary input string $v$ starts by replacing it with $v\$v$ and then behaves like $M'$ on the “input” $v\$v$.  
So if a Turing machine $M$ really exists (as specified) then it should be possible to construct $M''$ from it.
$M''$ halts on input $\langle M'' \rangle \iff M'\text{ rejects } \langle M'' \rangle$
$M''$ halts on input $\langle M'' \rangle \iff M'$ rejects $\langle M'' \rangle \$\langle M'' \rangle$

LHS says $\langle M'' \rangle \$\langle M'' \rangle \in L_{halt}$
\( M'' \) halts on input \( \langle M'' \rangle \) \iff \( M' \) rejects \( \langle M'' \rangle \$\langle M'' \rangle \)

LHS says \( \langle M'' \rangle \$\langle M'' \rangle \in L_{halt} \)

RHS says \( \langle M'' \rangle \$\langle M'' \rangle \notin L_{halt} \)
\[ M'' \text{ halts on input } \langle M'' \rangle \iff M' \text{ rejects } \langle M'' \rangle \dollar \langle M'' \rangle \]

LHS says \[ \langle M'' \rangle \dollar \langle M'' \rangle \in L_{halt} \]

RHS says \[ \langle M'' \rangle \dollar \langle M'' \rangle \not\in L_{halt} \]
Contradiction!
We can use the undecidability of the halting problem to show that various other problems are undecidable. A *reduction* from $L_1$ to $L_2$ is a TM-computable function $f$ such that

\[
\forall w \in L_1 \quad f(w) \in L_2 \\
\forall w \notin L_1 \quad f(w) \notin L_2
\]

If $L_2$ is recursive then $L_1$ is recursive

*equivalently:*

If $L_1$ is not recursive then $L_2$ is not recursive.
Uniform Halting Problem

Given a TM, does it halt on all inputs?

\[ L_{uhalt} = \{ \langle M \rangle : M \text{ halts on all inputs} \} \]

Find reduction \( L_{halt} \) to \( L_{uhalt} \)
reduction: $L_{\text{halt}}$ is reducible to $L_{\text{uhalt}}$

Given $\langle M \rangle\, w$, construct $M'$ as follows.
reduction: $L_{halt}$ is reducible to $L_{uhalt}$

Given $\langle M \rangle w$, construct $M'$ as follows.

$M'$ first of all writes word $w$ on tape (use $|w|$ states to do this).
reduction: $L_{halt}$ is reducible to $L_{uhalt}$

Given $\langle M \rangle w$, construct $M'$ as follows. $M'$ first of all writes word $w$ on tape (use $|w|$ states to do this). Then $M'$ deletes every non-blank symbol to the right of $w$ (two more states).
Reduction to uniform H.P.

reduction: \( L_{\text{halt}} \) is reducible to \( L_{\text{uhalt}} \)

Given \( \langle M \rangle \$w \), construct \( M' \) as follows.

\( M' \) first of all writes word \( w \) on tape (use \( |w| \) states to do this).

Then \( M' \) deletes every non-blank symbol to the right of \( w \) (two more states)

Then \( M' \) returns to LHS of input tape (another \( |w| \) states can be used)
reduction: $L_{halt}$ is reducible to $L_{uhalt}$

Given $\langle M \rangle, w$, construct $M'$ as follows.

$M'$ first of all writes word $w$ on tape (use $|w|$ states to do this).
Then $M'$ deletes every non-blank symbol to the right of $w$ (two more states)
Then $M'$ returns to LHS of input tape (another $|w|$ states can be used)
Then $M'$ imitates $M$ (using however many states $M$ had).
reduction: $L_{halt}$ is reducible to $L_{uhalt}$

Given $\langle M \rangle \, w$, construct $M'$ as follows. $M'$ first of all writes word $w$ on tape (use $|w|$ states to do this). Then $M'$ deletes every non-blank symbol to the right of $w$ (two more states) Then $M'$ returns to LHS of input tape (another $|w|$ states can be used) Then $M'$ imitates $M$ (using however many states $M$ had). The construction is TM-computable, and constructs $M'$ that accepts all its inputs if and only if $M$ accepts $w$. 
Define $L_{accept}$ to be the language

$$L_{accept} = \{ \langle M \rangle$w : M accepts input w $$

This problem is also undecidable. Reduce from $L_{halt}$
reduction: $L_{halt}$ is reducible to $L_{accept}$

Given $\langle M \rangle \$ w$, construct $\langle M' \rangle \$ w$ as follows. We want $M'$ to accept input $w$ if and only if $M$ halts on input $w$. 

1. Introduce a new accepting state $t$. For all undefined transitions of $M$ (which would lead to halt and reject) introduce a transition to $t$.
2. In any accepting state of $M$, transfer automatically to $t$.
3. Finally, we need to catch the case when $M$ moves off the left-hand side of the tape. Make $M'$ place a new character at the left-hand side initially and copy input 1 square to the right, then come back to the 2nd square of the tape and simulate $M$, accepting if that new character is ever scanned.
reduction: \( L_{halt} \) is reducible to \( L_{accept} \)

Given \( \langle M \rangle \$w \), construct \( \langle M' \rangle \$w \) as follows. We want \( M' \) to accept input \( w \) if and only if \( M \) halts on input \( w \).

So, general idea is: modify \( M \) such that if it halts, it must accept.
reduction: $L_{halt}$ is reducible to $L_{accept}$

Given $\langle M \rangle w$, construct $\langle M' \rangle w$ as follows. We want $M'$ to accept input $w$ if and only if $M$ halts on input $w$. So, general idea is: modify $M$ such that if it halts, it must accept. Introduce new accepting state $t$. For all undefined transitions of $M$ (which would lead to halt and reject) introduce a transition to $t$. In any accepting state of $M$, transfer automatically to $t$. Finally we need to catch the case when $M$ moves off LHS of tape. Make $M'$ place a new character at LHS initially and copy input 1 square to right, then come back to 2nd square of tape and simulate $M$, accepting if that new character is ever scanned.
reduction: \( L_{halt} \) is reducible to \( L_{accept} \)

Given \( \langle M \rangle \)$w, construct \( \langle M' \rangle \)$w as follows. We want \( M' \) to accept input \( w \) if and only if \( M \) halts on input \( w \).

So, general idea is: modify \( M \) such that if it halts, it must accept. Introduce new accepting state \( t \). For all undefined transitions of \( M \) (which would lead to halt and reject) introduce a transition to \( t \). In any accepting state of \( M \), transfer automatically to \( t \). Finally we need to catch the case when \( M \) moves off LHS of tape. Make \( M' \) place a new character at LHS initially and copy input 1 square to right, then come back to 2nd square of tape and simulate \( M \), accepting if that new character is ever scanned.
notation: Given language $L$ over alphabet $A$, its complement $A^* \setminus L$ is denoted $\overline{L}$.

Fact: $L$ is recursive iff $\overline{L}$ is recursive.

(easy to prove)

Definition: Language $L$ is \textit{co-recursively enumerable} provided that its complement $\overline{L}$ is recursively enumerable.

Fact: Language $L$ is recursive iff both $L$ and $\overline{L}$ are recursively enumerable.

from which we can deduce that $L_{halt}$ is not co-recursively enumerable.
**Proof** that \( L \) is recursive iff \( L, \bar{L} \) are r.e:

Easy to prove that \( L \) is recursive \( \Rightarrow \) \( L, \bar{L} \) are r.e.

Prove that \( L \) is recursive \( \Leftarrow \) \( L, \bar{L} \) are r.e:

There exist TMs \( T_1 \) and \( T_2 \) that accept \( L \) and \( \bar{L} \) respectively.

Given an input word \( w \), run \( T_1 \) and \( T_2 \) in parallel - as soon as one or the other accepts, then \( w \) can be accepted or rejected as appropriate.
Observe that $L_{halt}$ is r.e. but not co-r.e.
Also $L_{halt}$ is co-r.e. but not r.e.

**Question:** Is any language neither r.e. nor co-r.e.?
Observe that $L_{halt}$ is r.e. but not co-r.e. Also $\overline{L}_{halt}$ is co-r.e. but not r.e.

**Question:** Is any language neither r.e. nor co-r.e.?

Define $L_{inf}$ to be
\[
\{ \langle M \rangle : M \text{ halts on infinitely many distinct inputs} \}
\]
$L_{inf}$ is neither r.e. nor co-r.e.

Prove this by reducing $L_{halt}$ to $L_{inf}$ and to $\overline{L}_{inf}$. 
reduction: \( L_{\text{halt}} \leq L_{\text{inf}} \)

Convert \( \langle M \rangle \)$w to \( M' \) where \( M' \) writes \( w \) on tape, then goes to LHS and simulates \( M \).
**reduction:** $L_{halt} \leq L_{inf}$

Convert $\langle M \rangle \$w$ to $M'$ where $M'$ writes $w$ on tape, then goes to LHS and simulates $M$.

**reduction:** $L_{halt} \leq \overline{L_{inf}}$

Let $M''$ be a TM which scans input word left-to-right and accepts. Construct $M'$ to simulate $M''$ and $M$ in parallel, and if $M''$ halts (and accepts) first then $M'$ should halt, but if $M$ halts first, then $M'$ should go into an infinite loop. $M'$ halts for all inputs whose length is shorter than the computation of $M$ on input $w$. (Infinitely many of those iff $M$ fails to halt.)
The “busy beaver” function

$BB(x)$ is defined to be the largest number of steps taken by an $x$-state TM which halts, given an empty input tape. Using the undecidability of $L_{halt}$, we can show that $BB$ (and any faster-growing function) is uncomputable. We can say for example that $(((x)!))!$ is asymptotically smaller than $BB(x)$ (i.e. $(((x)!))! = o(BB(x))$.) simply because $(((x)!))!$ is computable.