TM VARIANTS AND ENUMERATOR

COMP 4200 – Formal Language

How to tell when a TM is at the left end of the tape?

- One Approach: Mark it with a special symbol. ex. $, 20
- Alternative method:
  - remember current symbol
  - overwrite it with special symbol
  - move left
  - if special symbol still there, head is at start of tape
  - otherwise, restore previous symbol and move left.

VARIANTS OF TM

- Multitape TM or Non-deterministic TM
- The original model and its reasonable variants all have the same power—they recognize the same class of languages.
- There are multiple ways to check the robustness of TM, one way is to change the transition function by including the tape head to Stay put.
- Might this feature allow Turing machines to recognize additional languages, thus adding to the power of the model? Of course not, because we can convert any TM with the "stay put" feature to one that does not have it.
- This small change shows how TM are robust in nature.

MULTI-TAPE TM

- TM with more than one tape.
- Each tape has its own head for reading and writing.
- Initially the input appears on tape 1, and the others start out blank.
- The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously.

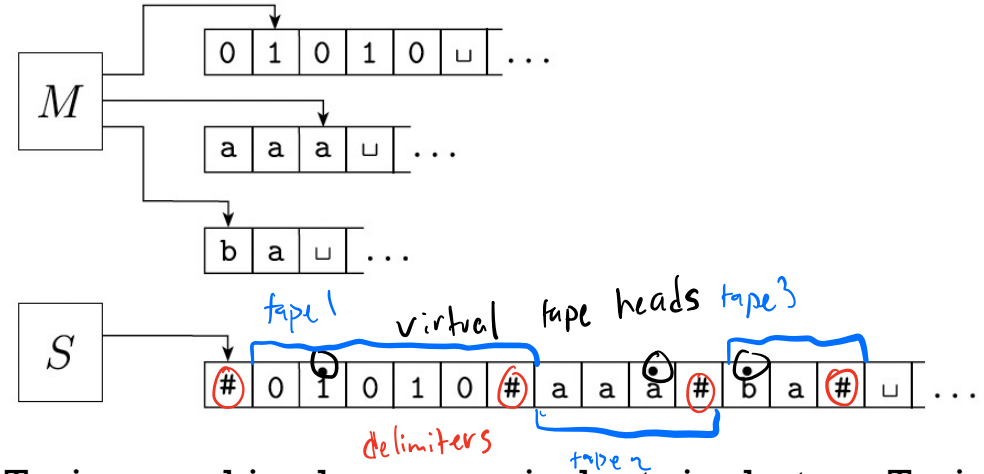
$$\delta: Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k$$

where k is the number of tapes.

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathrm{L}, \mathrm{R}, \ldots, \mathrm{L})$$

means that if the machine is in state qi and heads 1 through k are reading symbols a1 through ak, the machine goes to state qj, writes symbols b1 through bk, and directs each head to move left or right, or to stay put, as specified.

Multi - tape Turing machines appear to be more powerful than ordinary Turing machines, but we can show that they are equivalent in power.



(k+1) #
b/c add extra # at beginning of tape

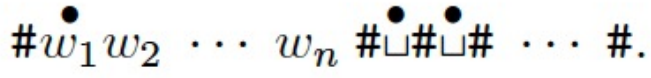
Every multitape Turing machine has an equivalent single-tape Turing machine.

- Say that M has k tapes.
- Then S simulates the effect of k tapes by storing their information on its single tape.
- It uses the new symbol # as a delimiter to separate the contents of the different tapes.
- In addition to the contents of these tapes, S must keep track of the locations of the heads.
- It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be.
- Think of these as "virtual" tapes and heads.
- As before, the "dotted" tape symbols are simply new symbols that have been added to the tape alphabet.

S = “On input w = w1 · · · wn:

- First S puts its tape into the format that represents all k tapes

of M. The formatted tape contains



To simulate a single move, S scans its tape from the first #,

- which marks the left-hand end, to the (k + 1)st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M’s transition function dictates.
- If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before.”

NONDETERMINISTIC TURING MACHINES

more possibilities

- A non-deterministic Turing machine is defined in the expected way. At any point in a computation, the machine may proceed according to several possibilities.
- The transition function for a nondeterministic Turing machine has the form

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

- The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine.
- If some branch of the computation leads to the accept state, the machine accepts its input.

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

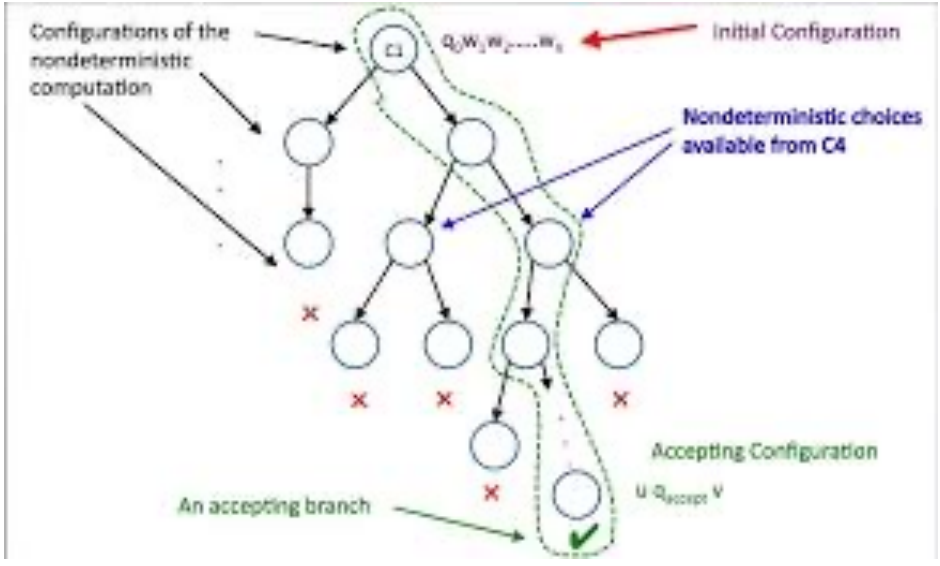
NFA

DFA

Idea:

- We can simulate any nondeterministic TM N with a deterministic TM D.
- The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation.
- If D ever finds the accept state on one of these branches, D accepts.
- Otherwise, D's simulation will not terminate.

- We represent N's computation on an input w as a tree.
- Each branch of the tree represents one of the branches of the nondeterminism.
- Each node of the tree is a configuration of N.
- The root of the tree is the start configuration.
- The TM D searches this tree for an accepting configuration.
- What type of search will we use?



NTM
one single path is DTM
many DTM in 1 NTM

MORE ON TM

- The depth-first search strategy goes all the way down one branch before backing up to explore other branches.
- If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch.
- So best way to explore the tree by using breadth-first search.
- This strategy explores all branches to the same depth before going on to explore any branch to the next depth.
- This method guarantees that D will visit every node in the tree until it encounters an accepting configuration.

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

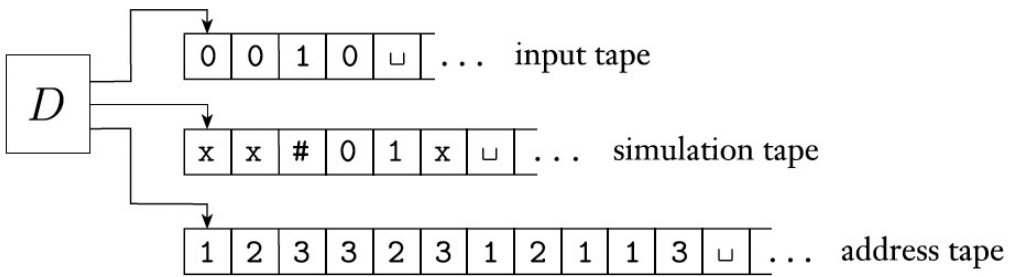
The simulating deterministic TM D has three tapes. The arrangement is equivalent to having a single tape.

The machine D uses its three tapes in a particular way.

Tape 1 always contains the input string and is never altered.

Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation.

Tape 3 keeps track of D's location in N's nondeterministic computation tree.



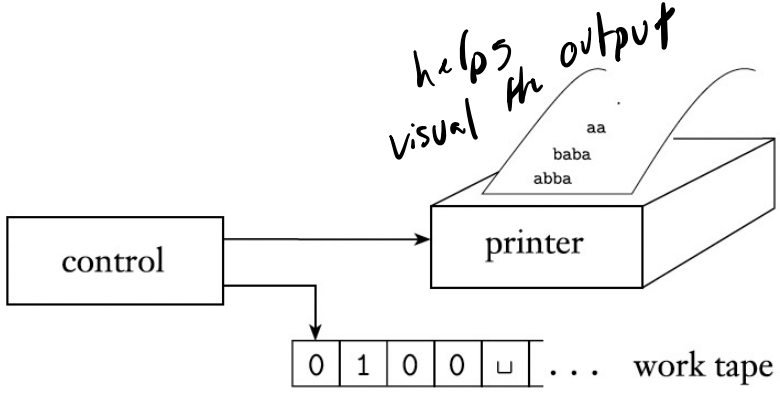
- Consider the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function.
- To every node in the tree we assign an address that is a string over the alphabet $\Gamma_b = \{1, 2, \ldots, b\}$.
- We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child.
- Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation.
- Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration.
- In that case, the address is invalid and doesn't correspond to any node.
- Tape 3 contains a string over Γ_b. It represents the branch of N's computation from the root to the node addressed by that string unless the address is invalid.
- The empty string is the address of the root of the tree.

Working of D:

- Initially, tape 1 contains the input w, and tapes 2 and 3 are empty.
- Copy tape 1 to tape 2 and initialize the string on tape 3 to be ε.
- Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N, consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function.
- If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, accept the input.
- Replace the string on tape 3 with the next string in the string ordering. Simulate the next branch of N's computation by going to stage 2.

ENUMERATORS

- The term, recursively enumerable language originates from a type of Turing machine variant called an enumerator.
- Loosely defined, an enumerator is a Turing machine with an attached printer.
- The Turing machine can use that printer as an output device to print strings.
- Every time the Turing machine wants to add a string to the list, it sends the string to the printer.



ENUMERATORS

- An enumerator E starts with a blank input on its work tape.
- If the enumerator doesn't halt, it may print an infinite list of strings.
- The language enumerated by E is the collection of all the strings that it eventually prints out.
- Moreover, E may generate the strings of the language in any order, possibly with repetitions.

A language is Turing-recognizable if and only if some enumerator enumerates it.

PROOF: First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A. The TM M works in the following way.

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."

Clearly, M accepts those strings that appear on E's list.

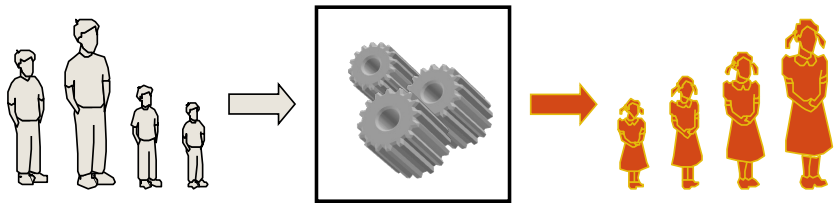
Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that s1, s2, s3, . . . is a list of all possible strings in Σ^*.

E = "Ignore the input.

1. Repeat the following for i = 1, 2, 3, ...
2. Run M for i steps on each input, s1, s2, . . . , si.
3. If any computations accept, print out the corresponding sj ."

If M accepts a particular string s, eventually it will appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running M in parallel on all possible input strings.

ALGORITHM

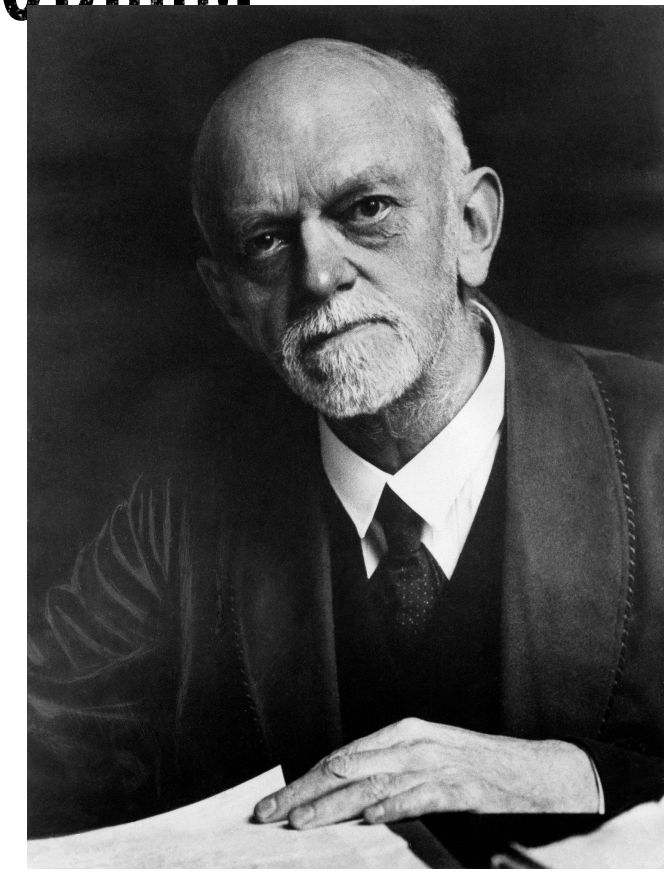


Input Algorithm Output

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

HILBERT'S PROBLEM

- In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris.
- In his lecture, he identified 23 mathematical problems and posed them as a challenge for the coming century.
- The tenth problem on his list concerned algorithms.



HILBERT'S 10TH PROBLEM

- A polynomial is a sum of terms, where each term is a product of certain variables and a constant, called a coefficient. For example,

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

is a term with coefficient 6, and

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

is a polynomial with four terms, over the variables x, y, and z.

For this discussion, we consider only coefficients that are integers.

HILBERT'S 10TH PROBLEM

- A root of a polynomial is an assignment of values to its variables so that the value of the polynomial is 0.
- This polynomial has a root at $x = 5$, $y = 3$, and $z = 0$.
- This root is an integral root because all the variables are assigned integer values.
- Some polynomials have an integral root and some do not.
- Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root.

HILBERT'S 10TH PROBLEM

- No algorithm exists for this task; it is algorithmically unsolvable.
- For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible.
- The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task.
- Proving that an algorithm does not exist requires having a clear definition of algorithm.

CHURCH TURING THESIS

- The definition came in the 1936 papers of Alonzo Church and Alan Turing.
- Church used a notational system called the λ-calculus to define algorithms.
- Turing did it with his "machines."
- These two definitions were shown to be equivalent.
- This connection between the informal notion of algorithm and the precise definition has come to be called the Church–Turing thesis.

CHURCH TURING THESIS

- So, what is Church Turing Thesis in simple words?
  - It states that any function that can be computed algorithmically can be computed by a Turing machine, and vice versa.
  - The Church-Turing thesis suggests that any problem that can be solved by an algorithm can also be solved by a Turing machine, which serves as a theoretical model of a general-purpose computer.

HILBERT'S 10TH PROBLEM

Let D = {p| p is a polynomial with an integral root}

Hilbert's tenth problem → the set D is decidable. The answer is negative.

In contrast, we can show that D is Turing-recognizable.

Considering a simpler problem, polynomials that have only a single variable, such as $4x^3 - 2x^2 + x - 7$.

Let D1 = {p| p is a polynomial over x with an integral root}

Here is a TM M1 that recognizes D1:

M1 = "On input ⟨p⟩: where p is a polynomial over the variable x.

- Evaluate p with x set successively to the values 0, 1, −1, 2, −2, 3,

−3, ... If at any point the polynomial evaluates to 0, accept ."

- If p has an integral root,M1 eventually will find it and accept. If p does not have

an integral root, M1 will run forever.

For the multivariable case, we can present a similar TMM that recognizes D.

Here, M goes through all possible settings of its variables to integral values.

Both M1 and M are recognizers but not deciders.

We can convert M1 to be a decider for D1 because we can calculate bounds within which the roots of a single variable polynomial must lie and restrict the search to these bounds.

We can show that the roots of such a polynomial must lie between the values

$$\pm k \frac{c_{\max}}{c_1},$$

where k is the number of terms in the polynomial, c_{max} is the coefficient with the largest absolute value, and c_1 is the coefficient of the highest order term.

If a root is not found within these bounds, the machine rejects.

HOW DO WE DESCRIBE A TM?

What is the right level of detail to give when describing such algorithms?

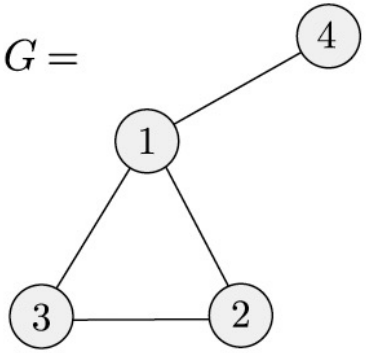
There are three possibilities.

- The first is the formal description that spells out in full the Turing machine's states, transition function, and so on.
- the lowest → most detailed level of description.
- higher level of description → implementation description, in which we use English prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function.
- The high-level description, we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head.

- The input to a Turing machine is always a string.
- If we want to provide an object other than a string as input, we must first represent that object as a string.
- Strings can easily represent polynomials, graphs, grammars, automata, and any combination
- of those objects.
- A Turing machine may be programmed to decode the representation so that it can be interpreted in the way we intend.
- Our notation for the encoding of an object O into its representation as a string is ⟨O⟩.
- If we have several objects O1,O2, . . .,Ok, we denote their encoding into a single
- string ⟨O1,O2, . . .,Ok⟩.
- The encoding itself can be done in many reasonable ways.

GRAPH ENCODING

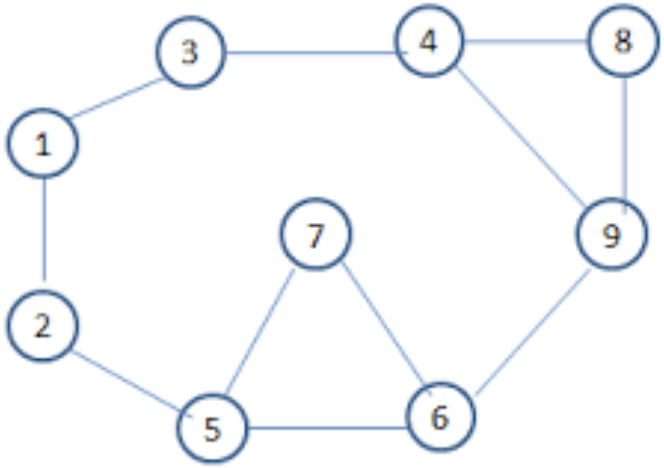
$G =$



$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

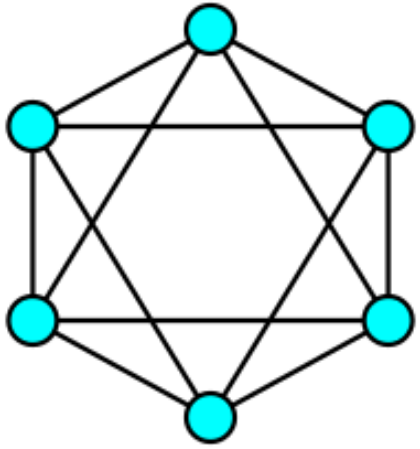
EXAMPLE

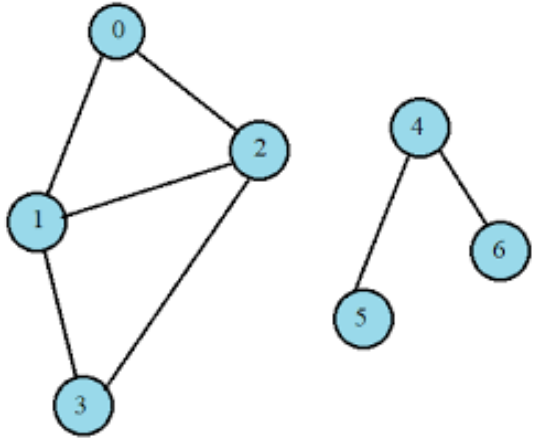


(1, 2, 3, 4, 5, 6, 7, 8, 9) ✓ (1,2), (1,3), (2,5), (5,7), (5,6)
(7,6), (6,9), (9,8), (9,4), (8,4), (4,3)

Let A be the language consisting of all strings representing undirected graphs that are connected. Here the graph is connected if every node can be reached from every other node by traveling along the edges of the graph.

A = {⟨G⟩ | G is a connected undirected graph}.





- A high-level description of a TM M that decides A.

M = "On input ⟨G⟩, the encoding of a graph G:

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, accept; otherwise, reject ."

DECIDABILITY

only short answers/multiple choice

WHY SHOULD YOU STUDY UNSOLVABILITY?

- When a problem is algorithmically unsolvable is useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution, Halting, Complex, NP Hard
- What is a decidable language? (if you can halt & not loop forever)
  - a language for which there exists an algorithm (a Turing machine, for example) that can determine whether any given input string belongs to the language or not.
  - In simple terms, a language is decidable if there exists a procedure that can always halt and correctly answer "yes" or "no" for any input string.

DECIDABLE PROBLEMS FOR REGULAR LANG

DFA

A_{DFA} is a decidable language.

A_{DFA} = {⟨B,w⟩| B is a DFA that accepts input string w}.

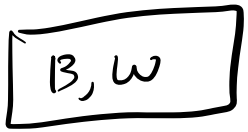
We simply need to present a TM M that decides A_{DFA}.

M = "On input ⟨B,w⟩, 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, accept. If it ends in a nonaccepting state, reject ."

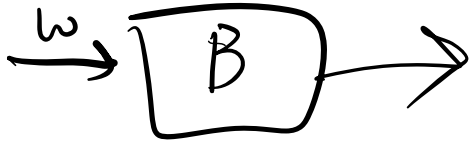
NFA

A_{NFA} is a decidable language.



A_{NFA} = {⟨B,w⟩ | B is an NFA that accepts input string w}.

We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: Have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.



N = "On input ⟨B,w⟩, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M on input ⟨C,w⟩.
3. If M accepts, accept; otherwise, reject ."

$A_{REX} = \{\langle R,w\rangle \mid R \text{ is a regular expression that generates string } w\}$.

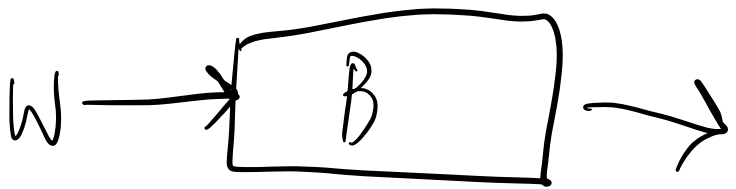
A_{REX} is a decidable language.

PROOF: The following TM P decides AREX.

P = "On input ⟨R,w⟩, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input ⟨A,w⟩.
3. If N accepts, accept; if N rejects, reject ."

A_{DFA}, A_{NFA}, A_{REX} all decidable



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

E_{DFA} is a decidable language.

A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible.

T = "On input ⟨A⟩, where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states 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, accept; otherwise, reject ."

Opposite from the rest

The theorem states that determining whether two DFAs recognize the same language is decidable. Let

EQ_{DFA} = {⟨A,B⟩ | A and B are DFAs and L(A) = L(B)}.

EQ_{DFA} is a decidable language.

We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

This expression is sometimes called the symmetric difference of L(A) and L(B).

Here, $\overline{L(A)}$ is the complement of L(A).

The symmetric difference is useful here because L(C) = ∅ iff L(A) = L(B).

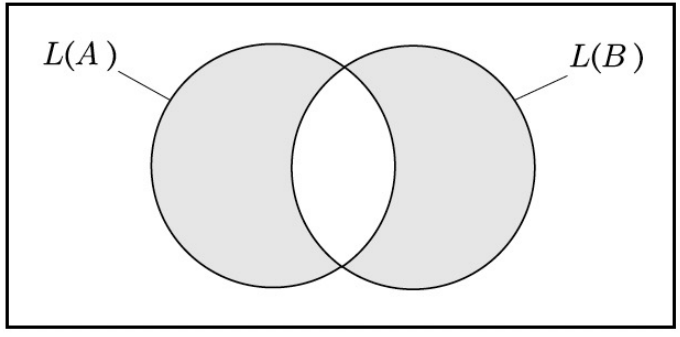
Symmetric Difference

F = "On input ⟨A,B⟩, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input ⟨C⟩.
3. If T accepts, accept. If T rejects, reject."

only accept if at a time

The intersection part will not be accepted



$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

DECIDABLE PROBLEMS CONCERNING CONTEXT-FREE LANGUAGES

A_{CFG} = {⟨G,w⟩| *G is a CFG that generates string w*}.

A_{CFG} is a decidable language.

PROOF The TM S for ACFG follows.

S = "On input ⟨G,w⟩, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form. CNF (reduce # of deviations)
2. List all derivations with 2n−1 steps, where n is the length of w; except if n = 0, then instead list all derivations with one step.
3. If any of these derivations generate w, accept; if not, reject."

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

E_{CFG} is a decidable language.

empty language

PROOF

R = "On input ⟨G⟩, where G is a CFG:

✓1. Mark all terminal symbols in G.

2. Repeat until no new variables get marked:

3. Mark any variable A where G has a rule $A \rightarrow U1U2 \cdots Uk$ and each symbol $U_1, \ldots, U_k$ has already been marked.

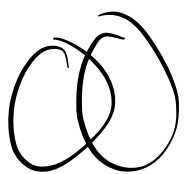
4. If the start variable is not marked, accept; otherwise reject.

opposite

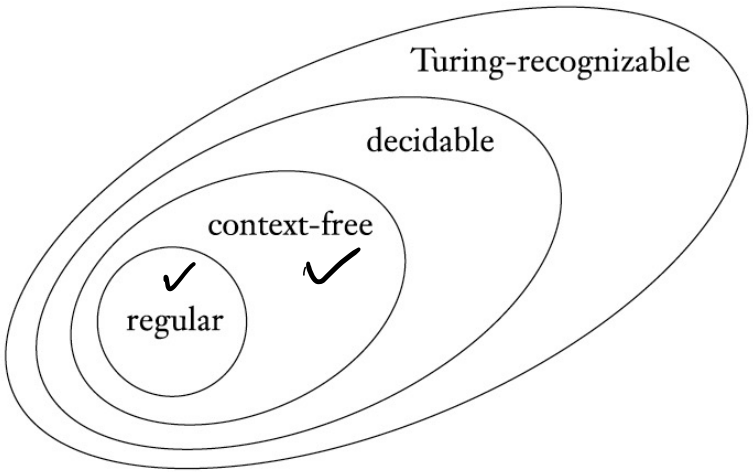
$EQ_{CFG} = \{\langle G,H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$.

- We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable.
- Because E_{CFG} also is decidable, you might think that we can use a similar strategy to prove that EQ_{CFG} is decidable.
- But something is wrong with this idea! → but why?
- The class of context-free languages is not closed under complementation or intersection

Can't use



RELATIONSHIP AMONG LANGUAGES



WHAT IS UNDECIDABILITY?

loop forever

What sorts of problems are unsolvable by computer?

Even some ordinary problems that people want to solve turn out to be computationally unsolvable.

In one type of unsolvable problem, you are given a computer program and a precise specification of what that program is supposed to do (e.g., sort a list of numbers).

double check →

You need to verify that the program performs as specified (i.e., that it is correct).

Because both the program and the specification are mathematically precise objects, you hope to automate the process of verification by feeding these objects into a suitably programmed computer.

The general problem of software verification is not solvable by computer.

$A_{TM} = \{\langle M,w\rangle | \ M \text{ is a TM and } M \text{ accepts } w\}$.

A_{TM} is undecidable.

First A_{TM} is Turing-recognizable.

Requiring a TM to halt on all inputs restricts the kinds of languages that it can

recognize. The following Turing machine U recognizes ATM.

U = "On input ⟨M,w⟩, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject." or can loop

3 choices

Note that this machine loops on input ⟨M,w⟩ if M loops on w, which is why this machine does not decide ATM. If the algorithm had someway to determine that M was not halting on w, it could reject in this case.

THE DIAGONALIZATION METHOD

- The proof of the undecidability of ATM uses a technique called diagonalization, discovered by mathematician Georg Cantor in 1873.
- Cantor was concerned with the problem of measuring the sizes of infinite sets.
- If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size?
- For finite sets, of course, answering these questions is easy.
- We simply count the elements in a finite set, and the resulting number is its size.
- But if we try to count the elements of an infinite set, we will never finish!
- So we can't use the counting method to determine the relative sizes of infinite sets.

HOW?

- How do we compare the relative size of the infinite set?

HOW?

pairs

- How do we compare the relative size of the infinite set?
- Cantor, proposed a solution to this problem.
- He observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set.
- This method compares the sizes without resorting to counting. We can extend this idea to infinite sets.

CARDINALITY

Cantor dealt with questions like:

How many **natural numbers** are there? Infinity!

How many **real numbers** are there? Infinity!

Does the **amount of natural numbers equal to** the **amount of real numbers?**

How is the size of infinite sets measured?

CARDINALITY

Cantor's answer to these question was the notion of ***Cardinality***.

The ***cardinality*** of a set is a property marking its size. relative size

Two sets has the same cardinality if there is a ***correspondence*** between their elements

some relationship between 2 sets will have the same cardinality

The Size of Infinity

How to compare the relative sizes of infinite sets?

Cantor (~1890s) had the following idea.

Defn: Say that set A and B have the same size if there is a one-to-one and onto function $f: A \to B$

$x \neq y \to f(x) \neq f(y)$ "injective"

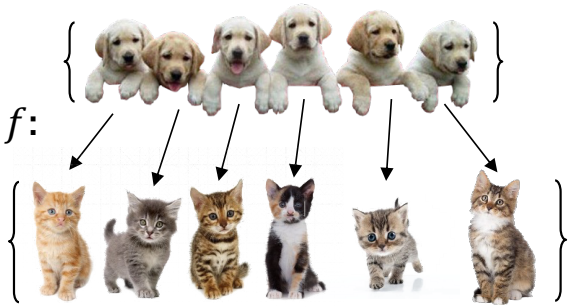
Range $(f) = B$ "surjective"

We call such an f a 1-1 correspondence

Informally, two sets have the same size if we can pair up their members.

This definition works for finite sets.

Apply it to infinite sets too.



CANTOR'S DEFINITION

Let N be the set of natural numbers {1, 2, 3, . . .} and let E be the set of even natural numbers {2, 4, 6, . . .}. Using Cantor's definition of size, we can see that N and E have the same size. The correspondence f mapping N to E is simply f(n) = 2n.

n	$f(n)$
1	2
2	4
3	6
⋮	⋮

$n = 1$ $f(n) = 2$

$f(n) = 2n$

There is a relationship here

EXAMPLE

So let us try to create a correspondence between the ***natural numbers*** the ***even natural numbers***?

$$N = \{1,2,3,4,\ldots,n,\ldots\} \qquad EN = \{2,4,6,8,\ldots,2n,\ldots\}$$

N	1	2	3	...	n	...
EN	2	4	6	...	$2n$	...

Indeed, ***defines*** the wanted correspondence between the 2 sets.

$$f(n) = 2n$$

EXAMPLE

$$N = \{1,2,3,4,\ldots,n,\ldots\}$$

$$EN = \{2,4,6,8,\ldots,2n,\ldots\}$$

N	1	2	3	...	n	...
EN	2	4	6	...	$2n$	...

So the ***cardinality*** of N is ***equal to*** the ***cardinality*** of EN.

COUNTABLE SETS

This last example suggests the notion of ***Countable Sets***:

A set A is ***countable*** if it is either ***finite*** or its cardinality is equal to the cardinality of N.

A cool way of looking at countable sets is:

"A set is countable if a list of its elements can be created".

COUNTABLE SETS

"A set is countable if a list of its elements can be created".

Note: This list does not have to be finite, but for each natural number i, one should be able to specify the i-th element on the list.

For example, for the set EN the i-th element on the list is $2i$.

COUNTABLE SETS

We just proved that EN, the set of even natural numbers is countable. What about the set of ***rational numbers***?

Is the set Q of rational numbers ***countable***?

Can its elements be ***listed***?

THE SET OF RATIONALS IS COUNTABLE

Theorem

The set of ***rational numbers*** is countable.

Proof

In order to prove this theorem we have to show how a complete list of the rational numbers can be formed.

Countable Sets

Let $\mathbb{N} = \{1,2,3, \dots\}$ and let $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$

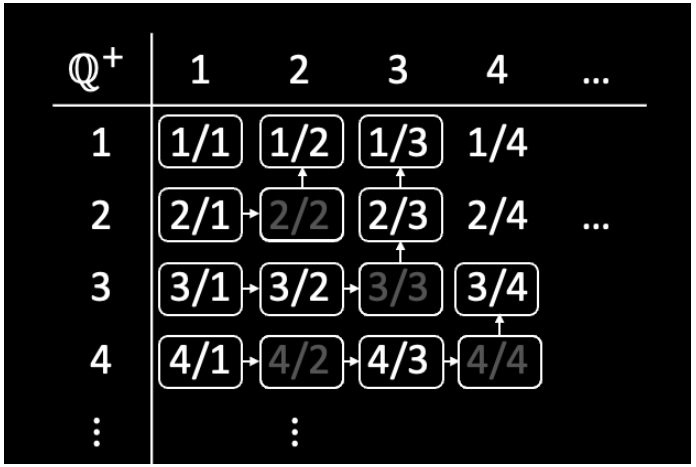
Show $\mathbb{N}$ and $\mathbb{Z}$ have the same size

n	$f(n)$
$\mathbb{N}$	$\mathbb{Z}$

Let $\mathbb{Q}^+ = \{ {}^m/_n \mid m, n \in \mathbb{N}\}$

Show $\mathbb{N}$ and $\mathbb{Q}^+$ have the same size

n	$f(n)$
$\mathbb{N}$	$\mathbb{Q}^+$



Defn: A set is countable if it is finite, or it has the same size as $\mathbb{N}$.

Both $\mathbb{Z}$ and $\mathbb{Q}^+$ are countable.

THE SET OF RATIONALS IS COUNTABLE

Matrix

Recall that each natural number is defined by a pair of natural numbers.

One way to look at the Rational

is by listing them

in an infinite

Rectangle.

$$
\begin{array}{ccccccc}
1/1 & 1/2 & 1/3 & 1/4 & 1/5 & \cdots & \\
2/1 & 2/2 & 2/3 & 2/4 & 2/5 & \cdots & \\
3/1 & 3/2 & 3/3 & 3/4 & 3/5 & \cdots & \\
4/1 & 4/2 & 4/3 & 4/4 & 4/5 & \cdots & \\
5/1 & 5/2 & 5/3 & 5/4 & 5/5 & \cdots & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots &
\end{array}
$$

THE SET OF RATIONALS IS COUNTABLE

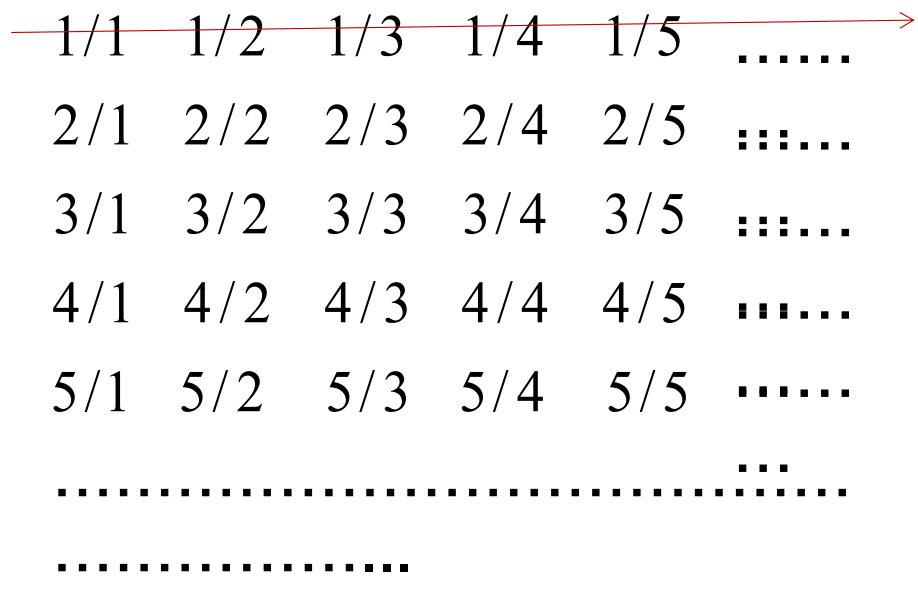
How can we form a list including all these numbers?

If we first list

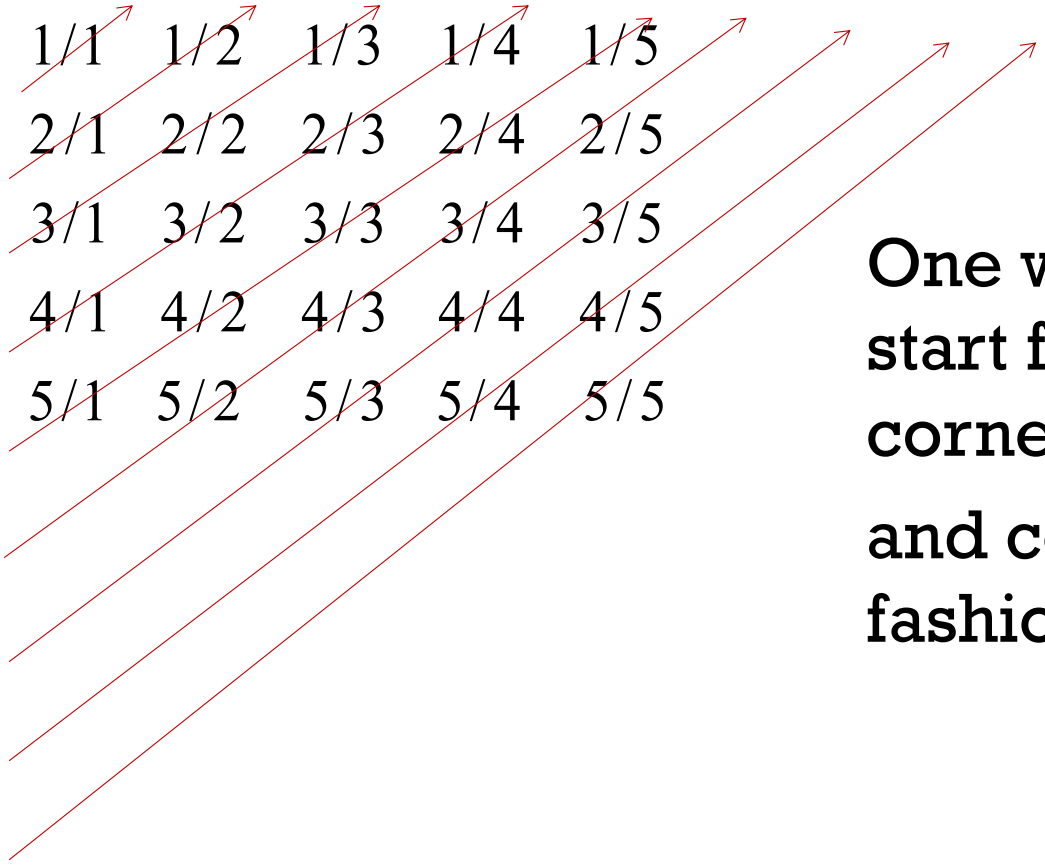
The first row –

We will never

reach the second.



THE SET OF RATIONALS IS COUNTABLE



One way to do it is to start from the upper left corner,

and continue in this fashion

THE SET OF RATIONALS IS COUNTABLE

Note that some rational numbers appear more than once. For example: all numbers on the main diagonal are equal to 1, so this list is not final.

In order to compute the actual place of a given rational, we need to erase all duplicates, but this is a technicality…

SO PERHAPS ALL SETS ARE COUNTABLE

Can you think of any infinite set whose elements cannot be listed in one after the other?

Well, there are many:

Theorem

The set of infinite binary sequences is not countable.

$\mathbb{R}$ is Uncountable – Diagonalization

Real #s

Let $\mathbb{R}$ = all real numbers (expressible by infinite decimal expansion)

Theorem: $\mathbb{R}$ **is uncountable**

Proof by contradiction via diagonalization: Assume $\mathbb{R}$ is countable

So there is a 1-1 correspondence $f: \mathbb{N} \to \mathbb{R}$

n	$f(n)$
1	
2	
3	
4	
5	
6	
7	
⋮	

Diagonalization

Demonstrate a number $x \in \mathbb{R}$ that is missing from the list.

$$x = 0.8516182\ldots$$

differs from the n^{th} number in the n^{th} digit
so cannot be the n^{th} number for any n.

Hence x is not paired with any n. It is missing from the list.

Therefore f is not a 1-1 correspondence.

AN UNDECIDABLE LANGUAGE

A_{TM} = {⟨M,w⟩| M is a TM and M accepts w}.

We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM}.

On input ⟨M,w⟩, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In otherwords, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} accept & \text{if } M \text{ accepts } w \\ reject & \text{if } M \text{ does not accept } w. \end{cases}$$

Now we construct a new Turing machine D with H as a subroutine.

This new TM calls H to determine what M does when the input to M is its own description ⟨M⟩.

Once D has determined this information, it does the opposite.

That is, it rejects if M accepts and accepts if M does not accept.

The following is a description of D.

D = "On input ⟨M⟩, where M is a TM:

1. Run H on input ⟨M,⟨M⟩⟩.
2. Output the opposite of what H outputs. That is, if H accepts, reject; and if H rejects, accept ."

$$D(\langle M \rangle) = \begin{cases} accept & \text{if } M \text{ does not accept } \langle M \rangle \\ reject & \text{if } M \text{ accepts } \langle M \rangle. \end{cases}$$

What happens when we run D with its own description $\langle D \rangle$ as input? In that case, we get

$$D(\langle D \rangle) = \begin{cases} accept & \text{if } D \text{ does not accept } \langle D \rangle \\ reject & \text{if } D \text{ accepts } \langle D \rangle. \end{cases}$$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist.

Assume that a TM H decides A_{TM}. Use H to build a TM D that takes an input ⟨M⟩, where D accepts its input ⟨M⟩ exactly when M does not accept its input ⟨M⟩. Finally, run D on itself. Thus, the machines take the following actions, with the last line being the contradiction.

- H accepts $\langle M, w \rangle$ exactly when M accepts w.
- D rejects $\langle M \rangle$ exactly when M accepts $\langle M \rangle$.
- D rejects $\langle D \rangle$ exactly when D accepts $\langle D \rangle$.

Entry i, j is accept if Mi accepts ⟨Mj⟩

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

if M3 does not accept input ⟨M2⟩, the entry for row M3 and column ⟨M2⟩ is reject because H rejects input ⟨M3, ⟨M2⟩⟩.

Entry i, j is the value of H on input ⟨Mi, ⟨Mj⟩⟩

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

By our assumption, H is a TM and so is D. Therefore, it must occur on the list M1, M2, . . of all TMs.

Note that D computes the opposite of the diagonal entries. The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

REDUCIBILITY

INTRODUCTION

In the context of undecidability: If we want to prove that a certain language L is undecidable. We assume by way of contradiction that L is decidable, and show that a decider for L, can be used to devise a decider for A_{TM}. Since A_{TM} is undecidable, so is the language L.

INTRODUCTION

Using a decider for L to construct a decider for A_{TM}, is called ***reducing*** L ***to*** A_{TM}.

Note: Once we prove that a certain language L is undecidable, we can prove that some other language, say L', is undecidable, by reducing L' to L.

PROCESS

1. We know that A is undecidable.
2. We want to prove B is undecidable.
3. We assume that B is decidable and use this assumption to prove that A is decidable.
4. We conclude that B is undecidable.

Note: The reduction is ***from*** A ***to*** B.

DEMONSTRATION

1. We know that A is undecidable.
   The only undecidable language we know, so far, is A_{TM} whose undecidability was proven directly. So we pick A_{TM} to play the role of A.
2. We want to prove B is undecidable.

DEMONSTRATION

2. We want to prove B is undecidable.
   We pick $HALT_{TM}$ to play the role of B that is: We want to prove that $HALT_{TM}$ is undecidable.

3. We assume that B is decidable and use this assumption to prove that A is decidable.

DEMONSTRATION

3. We assume that B is decidable and use this assumption to prove that A is decidable.
   In the following slides we assume (towards a contradiction) that $HALT_{TM}$ is decidable and use this assumption to prove that A_{TM} is decidable.
4. We conclude that B is undecidable.

THE "REAL" HALTING PROBLEM

Consider

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on } w\}$$

Theorem

$HALT_{TM}$ is undecidable.

Proof

By reducing $HALT_{TM}$ to A_{TM}.

DISCUSSION

Assume by way of contradiction that $HALT_{TM}$ is decidable.

Recall that a decidable set has a ***decider*** R: A TM that halts on every input and either accepts or rejects, but ***never loops!***.

We will use the assumed decider of $HALT_{TM}$ to devise a decider for A_{TM}

DISCUSSION

Recall the definition of A_{TM} :

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts } w\}$$

Why is it impossible to decide A_{TM} ?

Because as long as M runs, we cannot determine whether it will eventually halt.

Well, now we can, using the ***decider*** R for $HALT_{TM}$.

PROOF

Assume by way of contradiction that $HALT_{TM}$ is decidable and let R be a TM deciding A_{TM} it. In the next slide we present TM S that uses R as a subroutine and decides A_{TM}. Since A_{TM} is undecidable this constitutes a contradiction, so R does not exist.