Midterm review

Midterm Announcements

- Topics on midterm:

  Beginning of the course through today's lecture
- Exam Coverage
  - Lecture 1-12
  - Discussion 1-6
  - HW 1-3

- You may bring one double-sided 8.5 x 11 study sheet, that you prepare

- Thursday 5/30:
  - No Lecture
  - Midterm 7-9 pm

Techniques/concepts

Algorithmic techniques

- Potential method
- Divide-and-Conquer + Master Theorem
- Dynamic Programming
- Greed + Induction/Exchange

Models of Computation:

- DFAs
- Turing machines + Church-Turing thesis
- Terminology: countable vs uncountable, language, (un)decidable

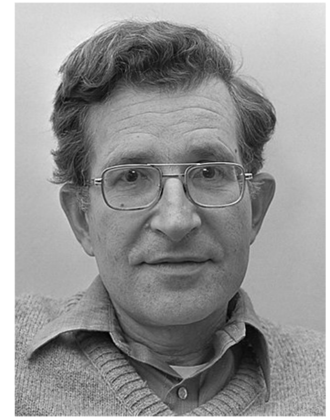
Techniques for proving undecidability

- Diagonalization/paradox
- Reduction

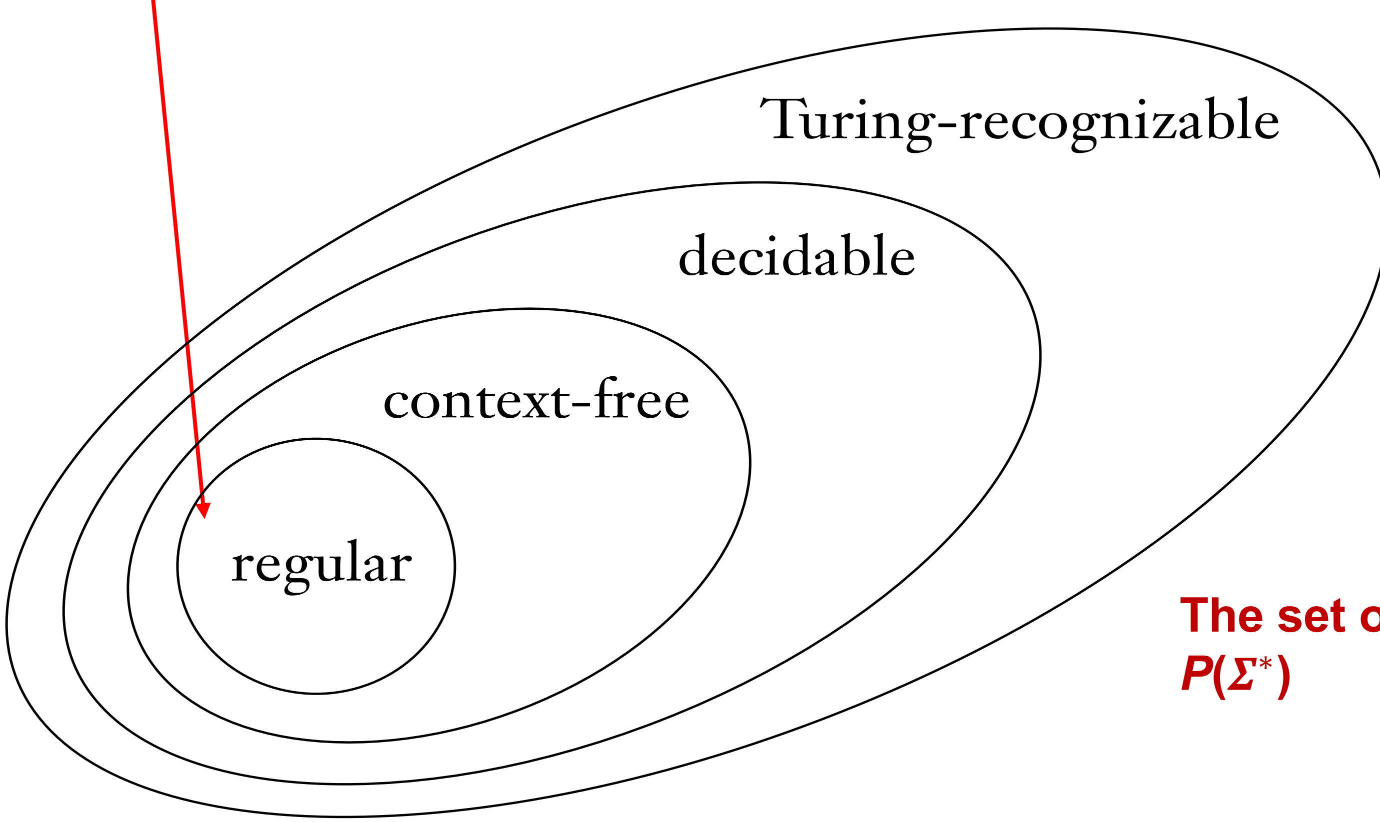
Reminder of problems + algorithms from class

- **Potential method:** GCD (Euclid)
- **Divide-and-conquer:** sorting (mergesort), closest pair, integer multiplication (Karatsuba)
- **Dynamic programming:** weighted task selection, LIS, LCS, knapsack, SSSP (Bellman-Ford), APSP (Floyd-Warshall)
- **Greedy:** unweighted task selection, MST (Kruskal)
- **Countable vs uncountable sets:** integers, rationals, reals, TMs, TM inputs, the set of all languages on a given alphabet
- **Undecidable languages:** L_{BARBER}, L_{ACC}, L_{HALT}

The Chomsky Hierarchy (1956)



“Regular Language”: Language decidable by some DFA



The set of all languages
***P*(Σ^*)**

More powerful “memory system” →

Some reference slides copied from past lectures

Potential Method

Intuitively, a **potential function argument** says:
If I start with a finite amount of water in a leaky bucket, water eventually must stop leaking out.

Ingredients of the argument:

1. Define the "unit of time" e.g. one iteration of an algorithm
2. Define how we measure the amount of water in the bucket. This is the **potential function S_i** ⟵ amount of water in bucket at timestep i
3. Prove that the S_0 is finite and S_i can never be negative
4. Prove that the bucket "leaks quickly". I.e. that S_i decreases by at least some fixed amount per unit time.
5. Use this to upper bound the total number of units of time.

Divide and Conquer

Overview: Divide-and-Conquer Algorithms

Main Idea:

1. **Divide** the input into smaller sub-problems
2. **Conquer:** solve each sub-problem recursively and combine their solutions

Designing the Algorithm + Proving Correctness: an "art"

- Depends on problem structure, ad-hoc, creative

Running time Analysis: "mechanical"

- Express runtime using a recurrence
- Can often solve using the "Master Theorem"

Integer Multiplication

- **Problem:** Given two n-bit numbers N_1 and N_2, compute $N_1 \times N_2$
- **Long Multiplication:**
  - Reduce problem to n additions of 2n-bit numbers
  - Do each addition in O(n) time
- **Runtime:** $O(n^2)$ in total!
- **Example:** What is 59 x 42?

$$
\begin{array}{rccccccccccccl}
 & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \leftarrow 59 \\
 & & & & & & \times & 1 & 0 & 1 & 0 & 1 & 0 & \leftarrow 42 \\
\hline
 = & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & & 59<<1 \\
 + & & & & 1 & 1 & 1 & 0 & 1 & 1 & & & & 59<<3 \\
 + & & 1 & 1 & 1 & 0 & 1 & 1 & & & & & & 59<<5 \\
\hline
2478 \rightarrow \; = & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 &
\end{array}
$$

Another example of divide and conquer:

Integer Multiplication

n digits

Divide: $x =$ | x_L | x_R |

$y =$ | y_L | y_R |

Conquer: $x \cdot y = (x_L \cdot 10^{n/2} + x_R)(y_L \cdot 10^{n/2} + y_R)$

$$= x_L y_L \cdot 10^n + (x_L y_R + x_R y_L) \cdot 10^{n/2} + x_R y_R$$

Recurrence:

Solving Recurrences

The Master Theorem

Formally: Consider the recurrence relation $T(n) = kT(n/b) + O(n^d)$, when $k, b > 1$. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } (k/b^d) < 1 \\ O(n^d \log n) & \text{if } (k/b^d) = 1 \\ O(n^{\log_b k}) & \text{if } (k/b^d) > 1 \end{cases}$$

$$T(1) = O(1)$$

Recall: k, b, and d are constants

(Earlier, Gauss used the same trick in a different context)

Karatsuba's idea!

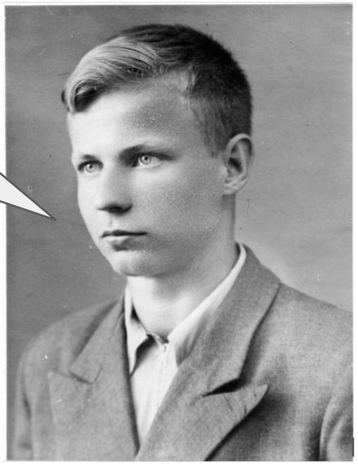
$O(n^2)$

Around 1956, the famous Soviet mathematician Andrey Kolmogorov conjectured that this is the *best possible way* to multiply two numbers together.

Just a few years later, Kolmogorov's conjecture was shown to be spectacularly wrong.

In 1960, Anatoly Karatsuba, a 23-year-old mathematics student in Russia, discovered a sneaky algebraic trick that reduces the number of multiplications needed.

We only need 3 recursive calls rather than 4!



Karatsuba's idea!

n digits

Divide:

x =	x_L	x_R
y =	y_L	y_R

Conquer: $x \cdot y = x_L y_L \cdot 10^n + \cancel{(x_L y_R + x_R y_L)} \cdot 10^{n/2} + x_R y_R$

Recurrence $(x_L+x_R)(y_L+y_R) - x_L y_L - x_R y_R$

$O(n^{\log_2 3})$

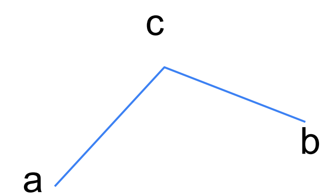
Formally: Consider the recurrence relation $T(n) = kT(n/b) + O(n^d)$, when $k, b > 1$. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } (k/b^d) < 1 \\ O(n^d \log n) & \text{if } (k/b^d) = 1 \end{cases}$$

Dynamic Programming

Dynamic Programming

High Level Idea: Break a complex problem into smaller (easier) subproblems subject to:



1. Principal of optimality (optimal substructure) –
   a substructure of an optimal structure is itself optimal
   Example: A subpath of any shortest path is itself a shortest path.

2. Overlapping sub-problems: “many” smaller subproblem are actually the “same” problem
   Example: When computing the Fibonacci sequence using the rule:
   $F_n = F_{n-1} + F_{n-2}$, “many” numbers are repeated.

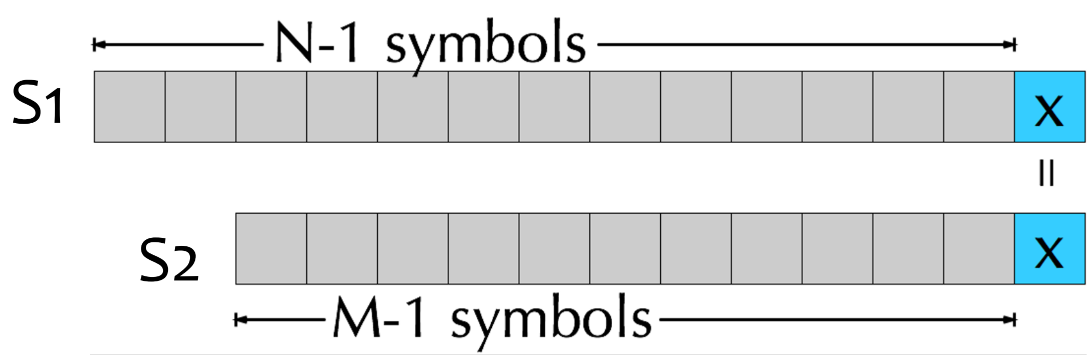
The DP Recipe

1. Write recurrence ← usually the trickiest part
2. Size of table: How many dimensions? Range of each dimension?
3. What are the base cases?
4. To fill in a cell, which other cells do I look at? In which order do I fill the table?
5. Which cell(s) contain the final answer?
6. Running time = (size of table) · (time to fill each entry)
7. To reconstruct the solution (instead of just its size) follow arrows from final answer to base case

LCS Recurrence

Part 1: Suppose the last character of S1 and S2 are the same i.e. S1[N] = S2[M]

Claim. There exists an optimal solution that matches S1[N] and S2[M].
Proof.

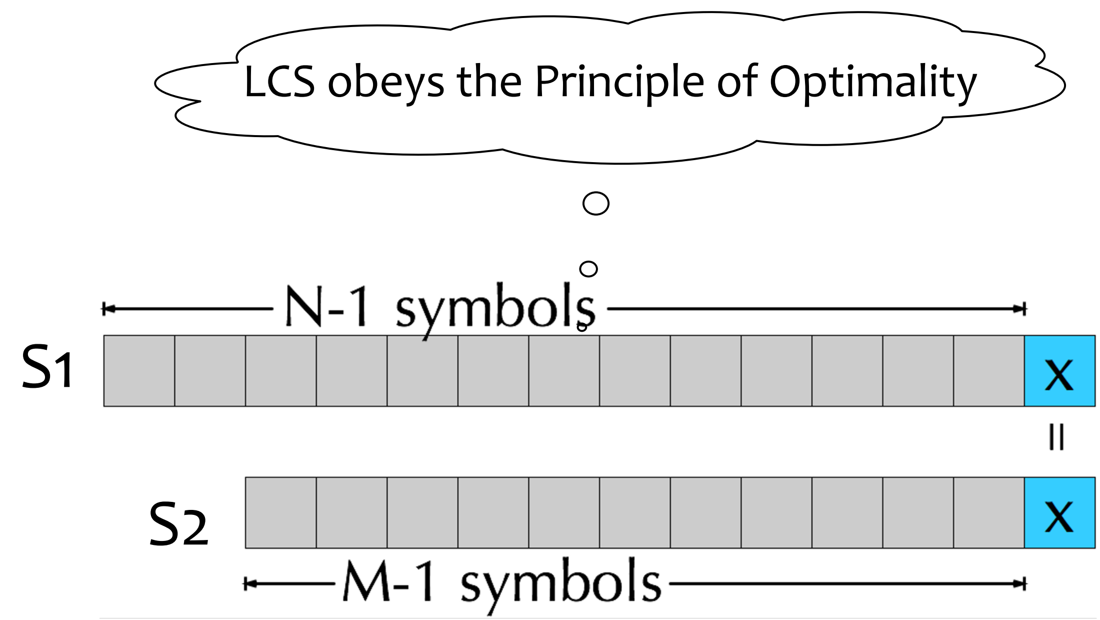


LCS Recurrence

Case 1: Suppose the last character of S1 and S2 are the same
i.e. S1[N] = S2[M]

Claim. There exists an optimal solution that matches S1[N] and S2[M].

$$\mathrm{LCS}(\ S1[1..N]\ ,\ S2[1..M]\ ) = \mathrm{LCS}(\ S1[1..N\text{-}1]\ ,\ S2[1..M\text{-}1]\ ) + 1$$



Since there's an optimal solution matching S1[N] and S2[M], we can **safely** add that match to our solution!

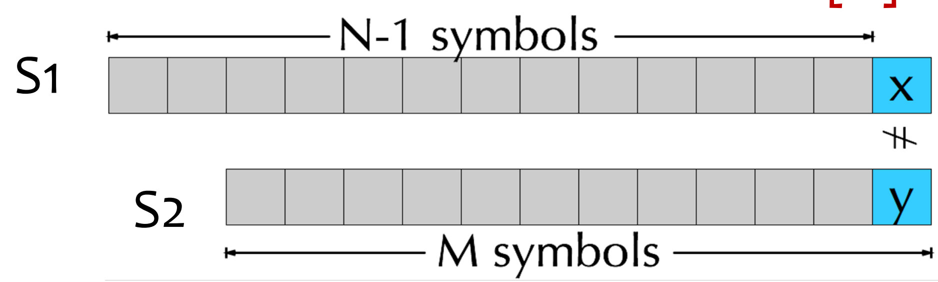
LCS Recurrence

Case 2: The last character of S1 and S2 are **not** the same

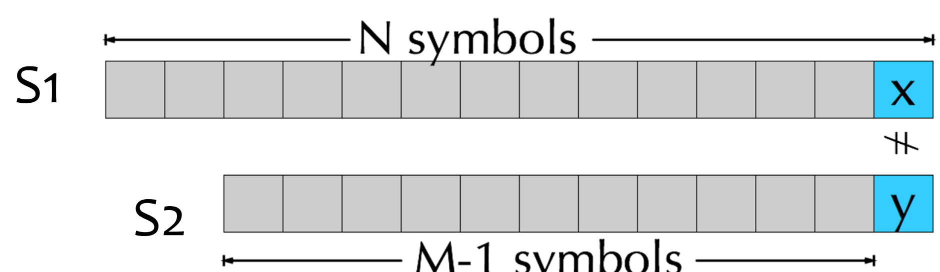
OPT doesn't have at least one of S1[N] and S2[M] **("lose it or lose it")**

"Lose S1[N] "

LCS(S1[1..N-1] , S2[1..M])



LCS(S1[1..N] , S2[1..M-1])



"Lose S2[M]"

Full Recurrence for LCS

$$\text{LCS}(S1[1..N], S2[1..M]) = \begin{cases} \text{LCS}(S1[1..N-1], S2[1..M-1]) + 1 & \text{if } S1[N] = S2[M] \\ \max\{\ \text{LCS}(S1[1..N-1], S2[1..M]), \\ \qquad \text{LCS}(S1[1..N], S2[1..M-1])\ \} & \text{otherwise} \end{cases}$$

Base cases:

$\text{LCS}(S1[1..i], \emptyset) = 0$ for all i

$\text{LCS}(\emptyset, S2[1..j]) = 0$ for all j

Let’s Follow the DP Recipe

S1 = GAC   S2 = AGCAT

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0					
A	0					
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , Ø) = 0 for all i

LCS(Ø , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC   S2 = AGCAT

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	0				
A	0					
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , Ø) = 0   for all i

LCS(Ø , S2[1..j]) = 0   for all j

Let’s Follow the DP Recipe

S1 = GAC S2 = AGCAT						
	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	0	1			
A	0					
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , Ø) = 0 for all i

LCS(Ø , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC S2 = AGCAT

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	0	1	1		
A	0					
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , Ø) = 0 for all i

LCS(Ø , S2[1..j]) = 0 for all j

Let’s Follow the DP Recipe

S1 = GAC S2 = AGCAT						
	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	0	1	1	1	
A	0					
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , Ø) = 0 for all i

LCS(Ø , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC   S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0					
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0   for all i

LCS(∅ , S2[1..j]) = 0   for all j

Let’s Follow the DP Recipe

S1 = GAC S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1				
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0 for all i

LCS(∅ , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC   S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1			
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0   for all i

LCS(∅ , S2[1..j]) = 0   for all j

Let's Follow the DP Recipe

S1 = GAC S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1		
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0 for all i

LCS(∅ , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC   S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	
C	0					

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0   for all i

LCS(∅ , S2[1..j]) = 0   for all j

Let's Follow the DP Recipe

S1 = GAC   S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1			

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1   if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) }   otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0   for all i

LCS(∅ , S2[1..j]) = 0   for all j

Let's Follow the DP Recipe

S1 = GAC S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2		

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0 for all i

LCS(∅ , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0 for all i

LCS(∅ , S2[1..j]) = 0 for all j

Let's Follow the DP Recipe

S1 = GAC S2 = AGCAT

	∅	A	G	C	A	T
∅	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2

LCS(S1[1..N] , S2[1..M]) =

LCS(S1[1..N-1] , S2[1..M-1]) + 1 if S1[N]=S2[M]

max { LCS(S1[1..N-1] , S2[1..M]),
LCS(S1[1..N] , S2[1..M-1]) } otherwise

Base cases:

LCS(S1[1..i] , ∅) = 0 for all i

LCS(∅ , S2[1..j]) = 0 for all j

Greedy algorithms

General strategy commonly used for analyzing greedy algorithms:

Proof by induction using an "**exchange**" argument

The idea: Show that we can transform any **optimal solution** into the **solution given by our algorithm** by **exchanging** each piece of it out one-by-one without increasing the cost.

Key part of proof: **Exchange** shows that my greedy choice is **safe** i.e. it is in some optimal solution.

Induction formalizes the idea that *each successive choice* is **safe**.

DFAs and Turing Machines

String notation

Alphabet: **A nonempty finite set Σ of symbols.**

$\Sigma = \{0,1\}$ is a popular choice.

String: **A finite sequence of 0 or more symbols.**
(or "word")

The empty string is denoted ε.

For any $a \in \Sigma$:

- a^k means k a's
- a^* means ≥0 a's
- a^+ means ≥1 a's

Σ^k means all strings over Σ of length k.

Σ^* means **all** (finite) strings over Σ.

Σ^+ means all nonempty (finite) strings over Σ

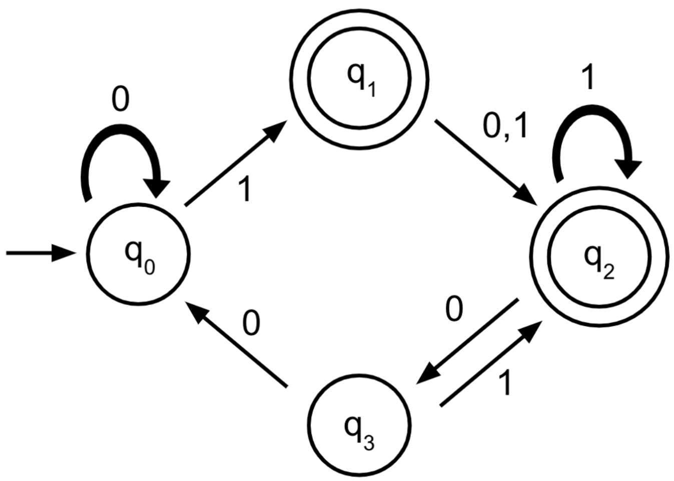
For any $a,b \in \Sigma$: $a|b$ means a OR b

Language: **A collection of strings.**

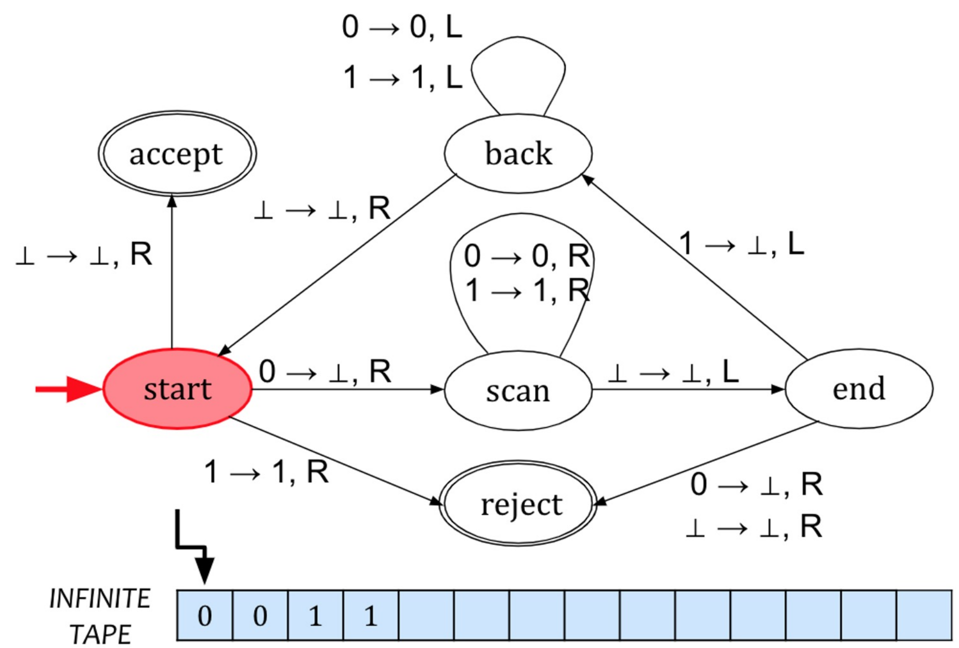
I.e. any subset $L \subseteq \Sigma^*$.

The empty language is denoted Ø.

DFA



Turing Machine



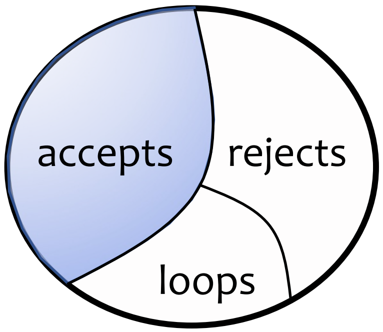
Undecidability

Undecidability and Reductions

Question: What are the possible outcomes of a TM **M**?

Answer: **M** either (i) accepts, (ii) rejects, *or* (iii) it *"loops" (forever)*

The language of a TM is the set of strings it accepts:
L(M) = {x : M accepts x}



Definition: A Turing Machine **M** ***decides*** a language L if it:

1. *<u>accepts</u>* every string in L, and
2. *<u>rejects</u>* every string not in L

   (and never loops forever)

A *language* L is ***decidable*** if there is a TM that decides L.

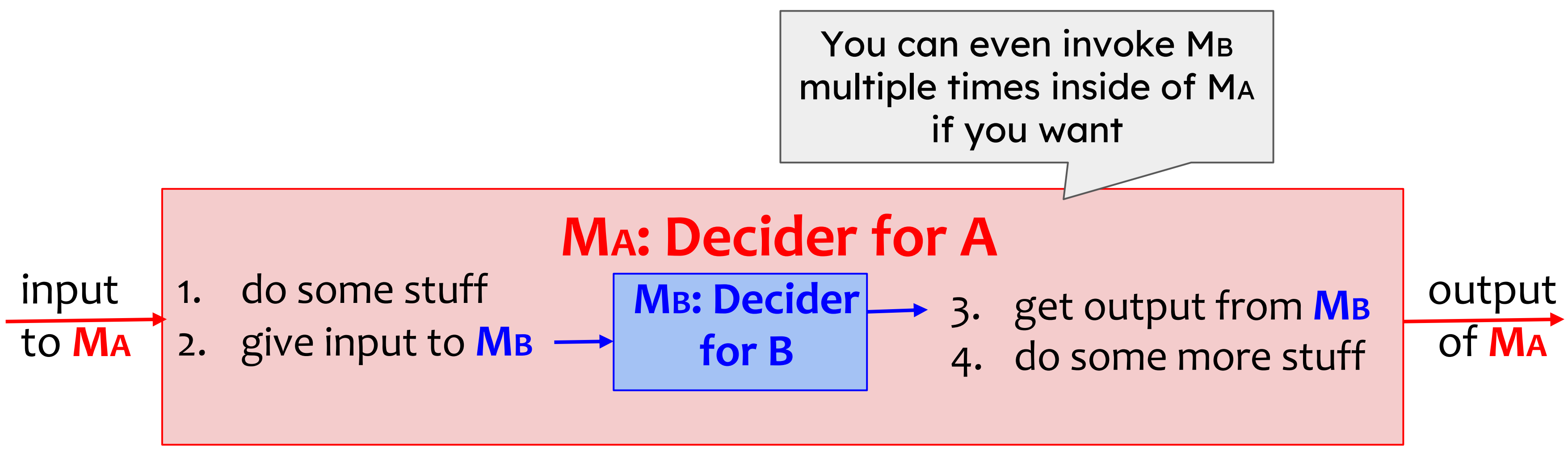
Otherwise L is ***undecidable***.

Turing Reduction from **A** to **B** (denoted **A** $\leq_T$ **B**):

"We can use a black-box decider for **B**
as a subroutine to decide **A**."

What it implies:

1. If **B** is decidable then **A** is decidable.
2. Contrapositive: If **A** is undecidable then **B** is undecidable.



"Problem **B** is at least as hard as Problem **A**"

New technique:

constructing new machines inside reductions

Another Undecidable Language:
ε-Halting Problem

Input: Turing Machine **M**

Output: Does **M** halt when given input **ε**?

Language: $L_{\varepsilon\text{-HALT}} = \{\langle M \rangle : M \text{ halts on input } \varepsilon\}$

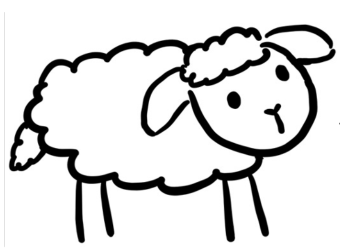
Here's a reduction from $L_{\varepsilon\text{-HALT}}$ to L_{HALT}, showing $L_{\varepsilon\text{-HALT}}$ is undecidable!

$M_{\varepsilon\text{-HALT}}(\langle M \rangle)$:

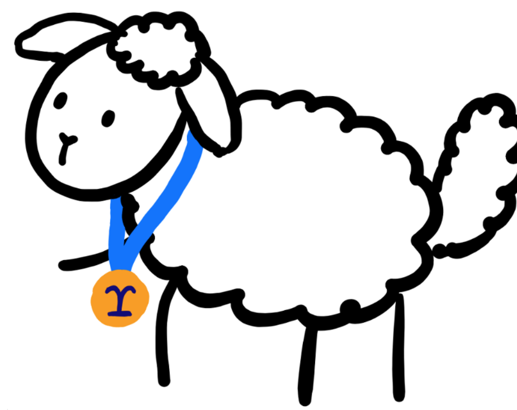
Run $M_{HALT}(\langle M \rangle, \varepsilon)$

If it accepts: accept

If it rejects: reject



Something is off…



Reduction from L_{HALT} to $L_{\varepsilon\text{-}HALT}$ (i.e. $L_{HALT} \leq_T L_{\varepsilon\text{-}HALT}$)

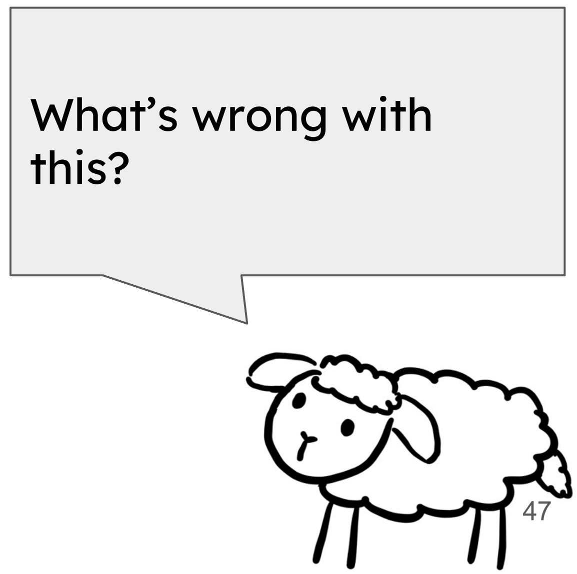
We need to implement:
M_{HALT} takes two inputs: **⟨M⟩, x**
M halts on input **x** ⇒ **M_{HALT}** accepts
M loops on input **x** ⇒ **M_{HALT}** rejects

Suppose we have:
$M_{\varepsilon\text{-}HALT}$ takes one input: **⟨M'⟩**
M' halts on input **ε** ⇒ **$M_{\varepsilon\text{-}HALT}$** accepts
M' loops on input **ε** ⇒ **$M_{\varepsilon\text{-}HALT}$** rejects

We need to specify the pseudocode:

M_{HALT}(⟨M⟩, x):

Run **$M_{\varepsilon\text{-}HALT}$(⟨M⟩)** and answer as **$M_{HALT}$**



Reduction from L_{HALT} to $L_{\varepsilon\text{-}HALT}$ (i.e. $L_{HALT} \leq_T L_{\varepsilon\text{-}HALT}$)

We need to implement:
M_{HALT} takes two inputs: **⟨M⟩, x**
M halts on input **x** ⇒ **M_{HALT}** accepts
M loops on input **x** ⇒ **M_{HALT}** rejects

Suppose we have:
$M_{\varepsilon\text{-}HALT}$ takes one input: **⟨M'⟩**
M' halts on input **ε** ⇒ **$M_{\varepsilon\text{-}HALT}$** accepts
M' loops on input **ε** ⇒ **$M_{\varepsilon\text{-}HALT}$** rejects

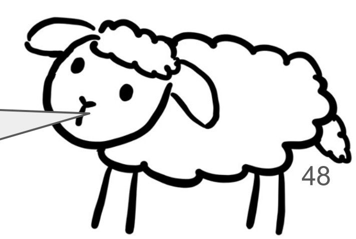
We need to specify the pseudocode:
M_{HALT}(⟨M⟩, x):
Let M_x be a TM that ignores its input and runs M(x)
What is next ??

M_x(w):
Run M(x) and answer as M does

Note: We will not run M_x, we just constructed it.
Why can't we run M_x?

Key idea: Construct new machine

We "hardcode" string x into the "hardware" of the TM M_x.



Reduction from L_{HALT} to $L_{\varepsilon\text{-}HALT}$ (i.e. $L_{HALT} \le_T L_{\varepsilon\text{-}HALT}$)

We need to implement:
M_{HALT} takes two inputs: $\langle M \rangle$, x
M halts on input $x \Rightarrow M_{HALT}$ accepts
M loops on input $x \Rightarrow M_{HALT}$ rejects

Suppose we have:
$M_{\varepsilon\text{-}HALT}$ takes one input: $\langle M' \rangle$
M' halts on input $\varepsilon \Rightarrow M_{\varepsilon\text{-}HALT}$ accepts
M' loops on input $\varepsilon \Rightarrow M_{\varepsilon\text{-}HALT}$ rejects

We need to specify the pseudocode:
$M_{HALT}(\langle M \rangle, x)$:
Let M_x be a TM that ignores its input and runs $M(x)$
What is next ??

$M_x(w)$:
Run $M(x)$ and answer as M

We are allowed to use $M_{\varepsilon\text{-}HALT}(\langle M' \rangle)$ as a subroutine, with the input of our choice

Reduction from L_{HALT} to $L_{\varepsilon\text{-}HALT}$ (i.e. $L_{HALT} \leq_T L_{\varepsilon\text{-}HALT}$)

We need to implement:
M_{HALT} takes two inputs: $\langle M \rangle$, x
M halts on input $x \Rightarrow M_{HALT}$ accepts
M loops on input $x \Rightarrow M_{HALT}$ rejects

Suppose we have:
$M_{\varepsilon\text{-}HALT}$ takes one input: $\langle M' \rangle$
M' halts on input $\varepsilon \Rightarrow M_{\varepsilon\text{-}HALT}$ accepts
M' loops on input $\varepsilon \Rightarrow M_{\varepsilon\text{-}HALT}$ rejects

We need to specify the pseudocode:

$M_{HALT}(\langle M \rangle, x)$:

Let M_x be a TM that ignores its input and runs $M(x)$

Run $M_{\varepsilon\text{-}HALT}(\langle M_x \rangle)$ and answer as $M_{\varepsilon\text{-}HALT}$

$M_x(w)$:

Run $M(x)$ and answer as M

Analysis:
M halts on x → $M_x(w)$ halts for all w including $w = \epsilon$ → $M_{\epsilon-halt}(M_x)$ accepts
M loops on x → $M_x(w)$ loops for all w including $w = \epsilon$ → $M_{\epsilon-halt}(M_x)$ rejects

Another Undecidable Language:
Empty Language Problem

Input: Turing Machine **M**

Output: Does **M** accept any input string at all?

Language: L_E = { <M> | M is a Turing machine and $L(M) = \emptyset$}

Reduction from L_{ACC} to L_E (i.e. $L_{ACC} \leq_T L_E$)

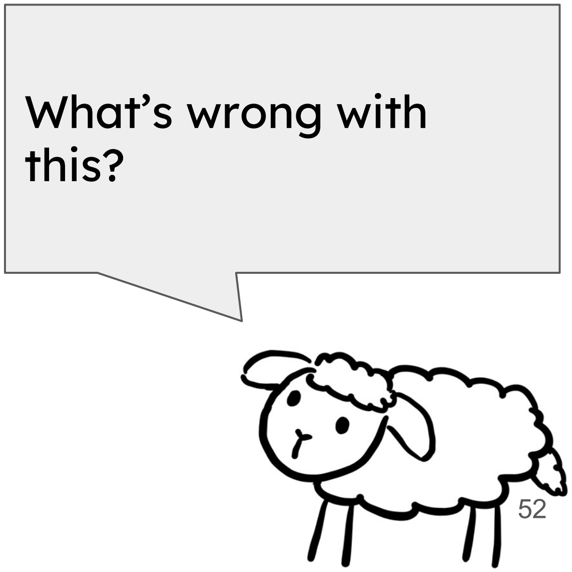
We need to implement:
M_{ACC} takes two inputs: $\langle M \rangle$, x
M accepts input $x \Rightarrow M_{ACC}$ accepts
M rejects input $x \Rightarrow M_{ACC}$ rejects

Suppose we have:
M_E takes one input: $\langle M' \rangle$
$L(M') = \emptyset \Rightarrow M_E$ accepts
$L(M') \neq \emptyset \Rightarrow M_E$ rejects

We need to specify the pseudocode:

$M_{ACC}(\langle M \rangle, x)$:

Run $M_E(\langle M \rangle)$ and answer as M_{ACC}



Reduction from L_{ACC} to L_E (i.e. $L_{ACC} \leq_T L_E$)

We need to implement:
M_{ACC} takes two inputs: ⟨M⟩, x
M accepts input x ⇒ M_{ACC} accepts
M rejects input x ⇒ M_{ACC} rejects

Suppose we have:
M_E takes one input: ⟨M'⟩
L(M') = ∅ ⇒ M_E accepts
L(M') ≠ ∅ ⇒ M_E rejects

We need to specify the pseudocode:
M_{ACC}(⟨M⟩, x):

M_x(w):
Reject if $w \neq x$
else Run M(x) and answer as M does

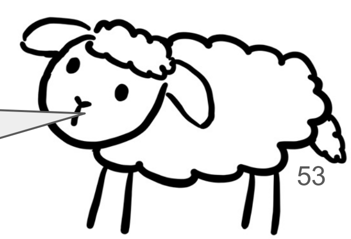
Let M_x be a TM that rejects all inputs except x and runs M(x)

What is next ??

Note: We will not run M_x, we just constructed it.
Why can't we run M_x?

Key idea: Construct new machine

We "hardcode" string x into the "hardware" of the TM M_x.



Reduction from L_{ACC} to L_E (i.e. $L_{ACC} \leq_T L_E$)

We need to implement:

M_{ACC} takes two inputs: $\langle M \rangle$, x

M accepts input $x \Rightarrow M_{ACC}$ accepts

M rejects input $x \Rightarrow M_{ACC}$ rejects

Suppose we have:

M_E takes one input: $\langle M' \rangle$

$L(M') = \emptyset \Rightarrow M_E$ accepts

$L(M') \neq \emptyset \Rightarrow M_E$ rejects

We need to specify the pseudocode:

$M_{ACC}(\langle M \rangle, x)$:

$M_x(w)$:
Reject if $w \neq x$
else Run $M(x)$ and answer as M does

Let M_x be a TM that rejects all inputs except x and runs $M(x)$

What is next ??

We are allowed to use $M_E(\langle M' \rangle)$ as a subroutine, with the input of our choice

Reduction from L_{ACC} to L_E (i.e. $L_{ACC} \leq_T L_E$)

We need to implement:

M_{ACC} takes two inputs: **⟨M⟩, x**

M accepts input **x** ⇒ **M_{ACC}** accepts

M rejects input **x** ⇒ **M_{ACC}** rejects

Suppose we have:

M_E takes one input: **⟨M'⟩**

L(M') = ∅ ⇒ **M_E** accepts

L(M') ≠ ∅ ⇒ **M_E** rejects

M_x(w):

Reject if **w ≠ *x***

else Run **M**(**x**) and answer as **M** does

We need to specify the pseudocode:

M_{ACC}(**⟨M⟩, x**):

Let **Mx** be a TM that rejects all inputs except **x** and runs **M**(**x**)

Run **M_E**(**⟨M_x⟩**) and answer as the opposite of **M_E**

Analysis:

M accepts x → $M_x(w)$ rejects all w except $w = x$ → $M_E(M_x)$ rejects

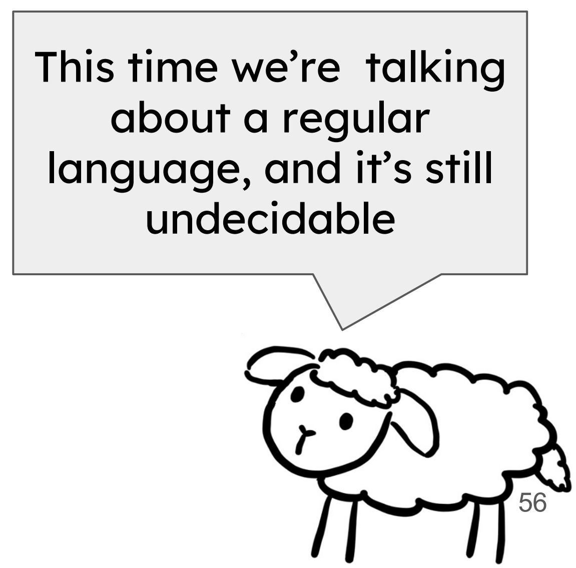
M rejects x → $M_x(w)$ rejects all w including $w = x$ → $M_E(M_x)$ accepts

Another Undecidable Language:
Regular Language Problem

Input: Turing Machine **M**

Output: Does **M** accept a regular language?

Language: $L_{REGULAR}$ = { <M> | M is a Turing machine and L(M) is a regular language}



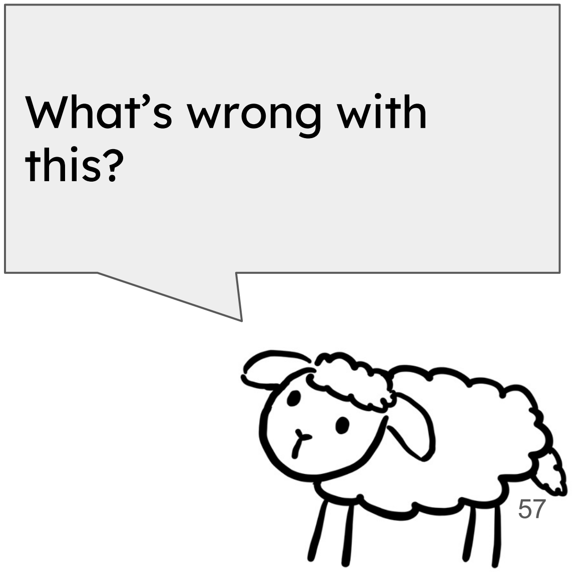
Reduction from L_{ACC} to $L_{REGULAR}$ (i.e. $L_{ACC} \leq_T L_{REGULAR}$)

We need to implement:
M_{ACC} takes two inputs: **⟨M⟩, x**
M accepts input **x** ⇒ **M_{ACC}** accepts
M rejects input **x** ⇒ **M_{ACC}** rejects

Suppose we have:
$M_{REGULAR}$ takes one input: **⟨M'⟩**
L(M') is regular ⇒ **$M_{REGULAR}$** accepts
L(M') is not regular ⇒ **$M_{REGULAR}$** rejects

We need to specify the pseudocode:

M_{ACC}(**⟨M⟩**, **x**):
Run **$M_{REGULAR}$**(**⟨M⟩**) and answer as **M_{ACC}**



Reduction from L_{ACC} to $L_{REGULAR}$ (i.e. $L_{ACC} \leq_T L_{REGULAR}$)

We need to implement:

M_{ACC} takes two inputs: $\langle M \rangle$, x

M accepts input x ⇒ M_{ACC} accepts

M rejects input x ⇒ M_{ACC} rejects

Suppose we have:

$M_{REGULAR}$ takes one input: $\langle M' \rangle$

L(M') is regular ⇒ $M_{REGULAR}$ accepts

L(M') is not regular ⇒ $M_{REGULAR}$ rejects

$M_x(w)$:
Accept if $w = 0^n1^n$
else Run M(x) and answer as M does

We need to specify the pseudocode:

$M_{ACC}(\langle M \rangle, x)$:

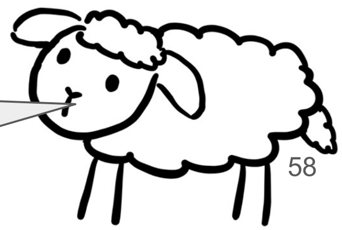
Let M_x be a TM that accepts all inputs 0^n1^n and runs M(x) otherwise

What is next ??

Note: We will not run M_x,
we just constructed it.
Why can't we run M_x?

Key idea: Construct new machine

We "hardcode" string x into the
"hardware" of the TM M_x.



Reduction from L_{ACC} to $L_{REGULAR}$ (i.e. $L_{ACC} \leq_T L_{REGULAR}$)

We need to implement:

M_{ACC} takes two inputs: **⟨M⟩, x**

M accepts input **x** ⇒ **M_{ACC}** accepts

M rejects input **x** ⇒ **M_{ACC}** rejects

Suppose we have:

$M_{REGULAR}$ takes one input: **⟨M'⟩**

L(M') is regular ⇒ **$M_{REGULAR}$** accepts

L(M') is not regular ⇒ **$M_{REGULAR}$** rejects

We need to specify the pseudocode:

M_{ACC}(**⟨M⟩**, **x**):

M_x(**w**):
Accepts if $w = 0^n 1^n$
else Runs **M**(**x**) and answer as **M** does

Let **Mx** be a TM that accepts all inputs $0^n 1^n$ and runs **M**(**x**) otherwise

What is next ??

We are allowed to use **M_E**(**⟨M'⟩**) as a subroutine, with the input of our choice

Reduction from L_{ACC} to $L_{REGULAR}$ (i.e. $L_{ACC} \leq_T L_{REGULAR}$)

We need to implement:

M_{ACC} takes two inputs: ⟨M⟩, x

M accepts input x ⇒ M_{ACC} accepts

M rejects input x ⇒ M_{ACC} rejects

Suppose we have:

$M_{REGULAR}$ takes one input: ⟨M'⟩

L(M') is regular ⇒ $M_{REGULAR}$ accepts

L(M') is not regular ⇒ $M_{REGULAR}$ rejects

We need to specify the pseudocode:

M_{ACC}(⟨M⟩, x):

Let Mx be a TM that accepts all inputs 0^n1^n and runs M(x) otherwise

Run $M_{REGULAR}$(⟨M_x⟩) and answer as $M_{REGULAR}$

M_x(w):

Accepts if $w = 0^n1^n$

else Runs M(x) and answer as M does

Analysis:

M accepts x → $M_x(w)$ accepts all inputs w → $M_{REGULAR}(M_x)$ accepts

M rejects x → $M_x(w)$ rejects all inputs except $w = 0^n1^n$ → $M_{REGULAR}(M_x)$ rejects