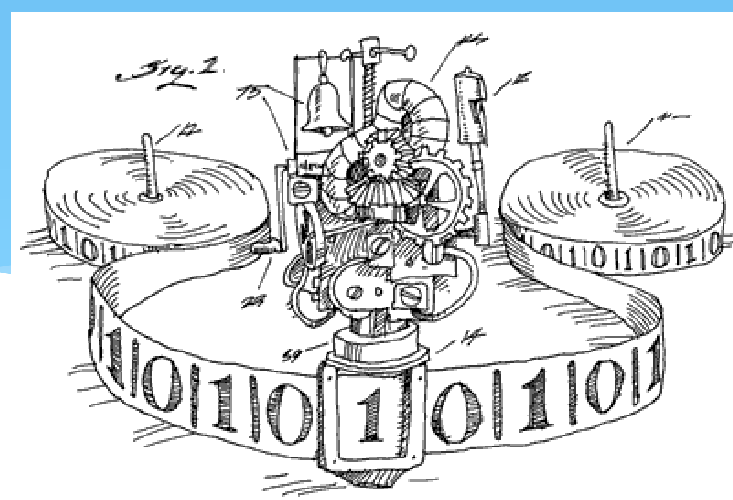
EECS 376: Foundations of Computer Science

Euiwoong Lee



Quote of The Day

"Every Complex Problem has a Clear, Simple and Wrong Solution."

– **H. L. Mencken**

Design & Analysis of Algorithms

- **Algorithm Design:** A set of methods to create algorithms for certain types of problems
- **Examples:** Dynamic Programming, Divide and Conquer, Greedy Algorithms, …
- **Algorithm Analysis:** A set of methods to prove correctness of algorithms and determine the amount of resources (e.g. time, memory) necessary to execute them
- **Examples:** Master Theorem, Potential Method, …
- **Reminder:** We describe algorithms in "Pseudo-Code".

Greatest Common Divisor

- **Definition:** Let $x, y \in \mathbb{N}$. The **Greatest Common Divisor (gcd)** of x and y is the largest $z \in \mathbb{N}$ that divides both x and y.
- If $\gcd(x, y) = 1$ then x and y are **coprime.**
- **Examples:**
  - $\gcd(21{,}9) = 3$
  - $\gcd(121{,}5) = 1$
- **Algorithm 1:** For $z = 1 \dots x$ test if z divides both x and y
- **Runtime:** $O(x)$ operations
- **Question:** Can we do better?

The Euclidean Algorithm

* **Algorithm 2:** $Euclid(x, y)$: (when $x > y \geq 0$)
  * if $y = 0$ return x
  * If $y = 1$ return 1
  * return $Euclid(y, x \bmod y)$
* **Question:** How many iterations can we have?
* **Analysis:** We use the potential method.
* Let s_i = the value of $x + y$ at iteration i. We show:

1. $s_0 = x + y$
2. for all i: $s_i \geq 1$
3. for all i: $s_{i+1} \leq \frac{3}{4} s_i$

* **Conclusion 1:** In ith iteration: $1 \leq s_i \leq \left(\frac{3}{4}\right)^i (x + y)$.
* **Conclusion 2:** $i \leq \log_{4/3}(x + y) = O(\log(x + y))$.

Euclid, 300 BCE

Quote of The Day

"Divide et impera"
(divide and conquer)

– **Philip II**

Divide and Conquer Algorithms

Main Idea:

1. Divide the problem into smaller subproblems
2. Solve each subproblem recursively
3. Combine the solutions of the subproblems in a "meaningful" way

Runtime Analysis:

* Tools to solve recurrence relations
* The "Master Theorem"

The Master Theorem

Story: Divide-and-conquer algorithm breaks a problem of size n into:

* k smaller problems
* each one of size n/b
* with cost of $O(n^d)$ to combine the results together

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

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

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 59×42?

$$
\begin{array}{rccccccccccccl}
 & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \longleftarrow 59 \\
 & & & & & & \times & 1 & 0 & 1 & 0 & 1 & 0 & \longleftarrow 42 \\
\hline
 & = & & & & & 1 & 1 & 1 & 0 & 1 & 1 & & 59\text{<<}1 \\
 & + & & & & 1 & 1 & 1 & 0 & 1 & 1 & & & 59\text{<<}3 \\
 & + & & 1 & 1 & 1 & 0 & 1 & 1 & & & & & 59\text{<<}5 \\
\hline
2478 \longrightarrow & = & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 &
\end{array}
$$

Divide and Conquer Multiplication

- **Input:** N_1 and N_2, two n-digit numbers (assume n is a power of 2)
- Split N_1 and N_2 into $n/2$ low-order digits & $n/2$ high-order digits:
  - $N_1 = a \cdot 10^{n/2} + b$
  - $N_2 = c \cdot 10^{n/2} + d$

	$\leftarrow n/2$ digits $\rightarrow$	$\leftarrow n/2$ digits
N_1	a	b
N_2	c	d

- Compute $N_1 \times N_2 = a \times c \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
  - $m_1 = (a + b) \times (c + d)$ time: $O(n) + T(n/2)$
  - $m_2 = a \times c$ time: $T(n/2)$
  - $m_3 = b \times d$ time: $T(n/2)$
  - **Return:** $m_2 \cdot 10^n + (m_1 - m_2 - m_3) \cdot 10^{n/2} + m_3$. time: $O(n)$
- $T(n)$ = time to multiply two n-digit numbers
  - $T(n) = 3T(n/2) + O(n) \Longrightarrow k = 3, b = 2 \Longrightarrow$
    $T(n) = O\left(n^{\log_2 3}\right) = O(n^{1.585})$.

Quote of The Day

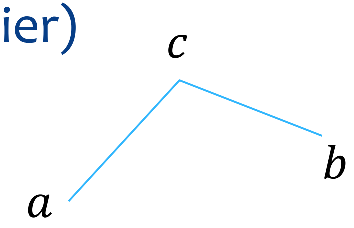
"If you can solve it, it is an exercise; otherwise it is a research problem"

– **Richard E. Bellman,**

The Inventor of Dynamic Programming

Dynamic Programming

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



1. Principle 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 subproblems are actually the "same" problem.

   Example: When computing the Fibonacci sequence using the rule: $F_n = F_{n-1} + F_{n-2}$, "many" recursive calls will be repeated.

Implementation Strategies

* Recurrence for Fibonacci:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

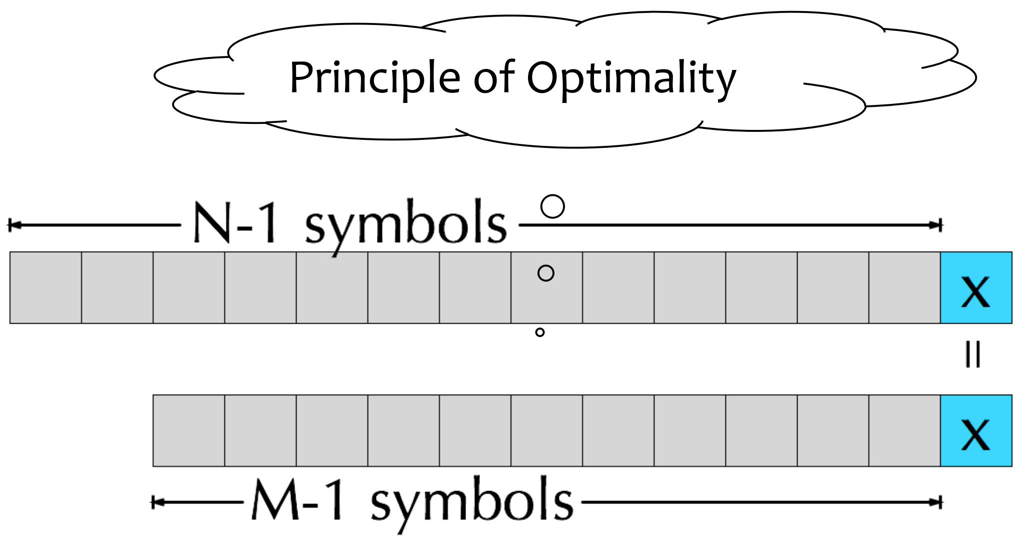
* Once we've determined the recurrence relation, we have a choice of three implementation strategies:
* **Top-down Recursive (Naïve):** Start at desired result, compute recursively down to the base case
* **Top-down Memoization:** Same as naïve, but save results as we compute them and reuse already-computed results
* **Bottom-up Table:** Start from base case(s), work our way up to the desired result

Longest Common Subsequence

- **Definition:** A ***subsequence*** of a string s is a subset of the characters of s with respect to their original order.
  - **Example:** for s = "Fibonacci sequence"
    - "Fun"
    - "seen"
    - "cse"
    - …
- Given strings $X[1..n]$ and $Y[1..m]$
- **Goal:** Find the length of a ***longest common subsequence*** of X and Y.
  - Largest string obtainable from X *and* Y by deleting chars
- **Example:** "Gole" is an LCS of "Google" and "Go Blue".
- **Q:** What's a brute force solution?
  - Each character of X and Y is either deleted or not.

Longest Common Subsequence

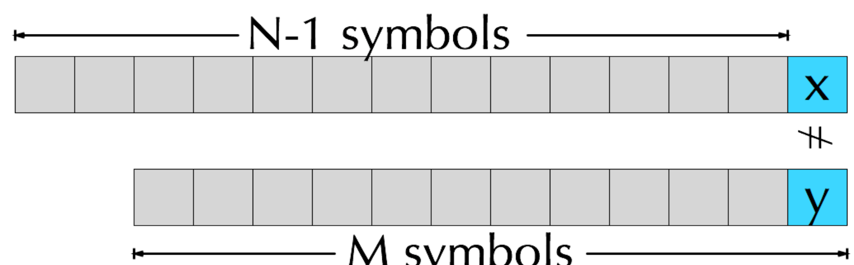
* **Idea:** Let X and Y be two strings of length n and m, respectively.
* If the last characters are equal: $(X[n] = Y[m])$:
* $LCS(X[1..n], Y[1..m]) = LCS(X[1..n-1], Y[1..m-1]) + 1$



Longest Common Subsequence

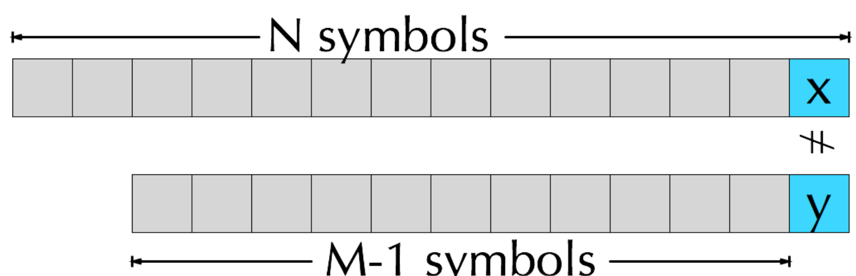
- **Idea:** Let X and Y be two strings of length n and m, respectively.
- If the last characters are **not** equal: $(X[n] \neq Y[m])$:
- $LCS(X[1..n], Y[1..m]) =$ Maximum of

$LCS(X[1..n-1], Y[1..m])$



and

$LCS(X[1..n], Y[1..m-1])$



Recurrence for LCS

* Let $LCS(i, j)$ denote the length of a longest common subsequence of $X[1..i]$ and $Y[1..j]$.
* Then:

$$LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & X[i] = Y[j] \\ \max\begin{Bmatrix} LCS(i-1, j), \\ LCS(i, j-1) \end{Bmatrix} & X[i] \neq Y[j] \end{cases}$$

* **Naïve Implementation:** Exponential runtime!
* **Observation:** There are $O(nm)$ distinct values: $LCS(i, j)$ for $0 \leq i \leq n$ and $0 \leq j \leq m$ (overlapping sub-problems)

https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

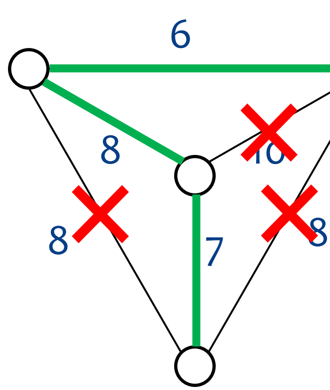
Quote of The Day

"Greed is not a financial issue.
It's a heart issue."

– **Andy Stanley**

Kruskal's Algorithm

Kruskal(G)**:** // G is a weighted, undirected graph
$T \leftarrow \emptyset$ // invariant: T has no cycles
for each edge e in *increasing order of weight*:
  if $T + e$ is acyclic: $T \leftarrow T + e$
return T



Quote of The Day

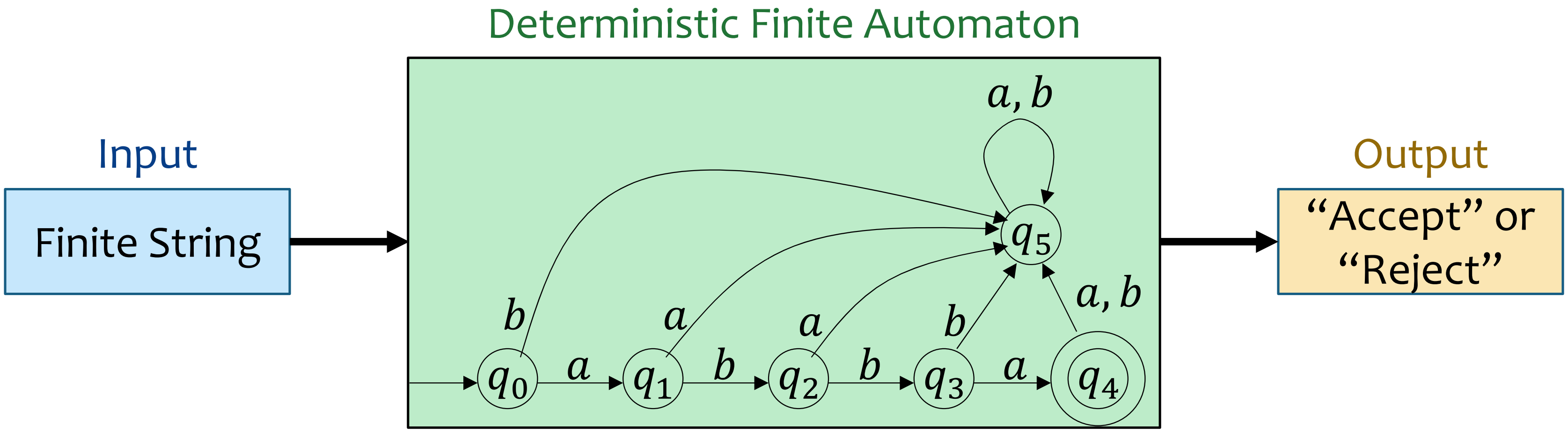
"I believe that the question:
'Can machines think?'
is too meaningless to deserve discussion."

– **Alan Turing**

Computability: Review

- **Question:** Which problems are solvable by a computer?
- **Answer:** Depends on what a *problem* is, what *solvable* is, and what a *computer* is.
- **Problem:** A language $L \subseteq \Sigma^*$
  - Set of strings whose output is YES/accept.

Deterministic Finite Automaton (DFA)



DFA: Formal Definition

* A deterministic finite automaton (DFA) is a 5-tuple:

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

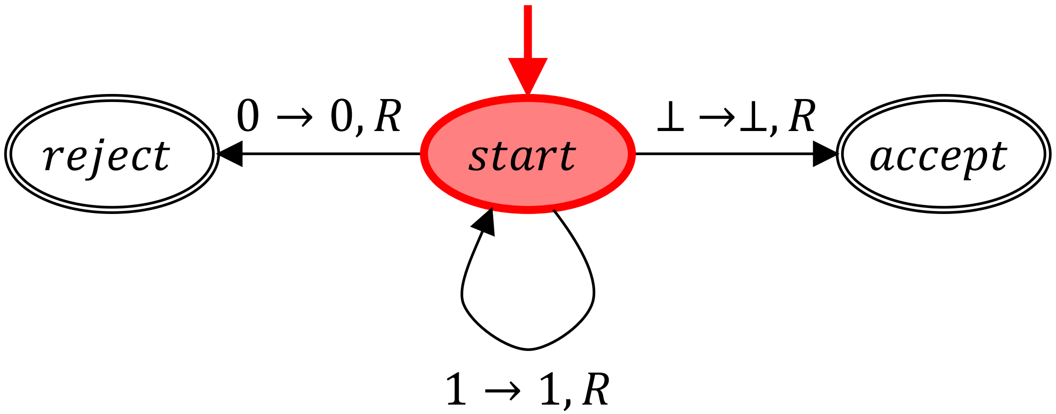
  * Q = (a finite) set of **states**
  * Σ = the (finite) **input alphabet** (often {0,1} but not always)
  * $q_0 \in Q$, = the **initial state** (an element of Q)
  * $F \subseteq Q$, = the set of **final/accepting states** (a subset of Q)
  * $\delta: Q \times \Sigma \to Q$ = the **transition function** (maps a state and input character to a new state)
  * **Definition:** M **accepts** x, if given x as an input, M starts at q_0, makes transitions according to δ, and ends in an accepting state $q \in F$.

Turing Machines

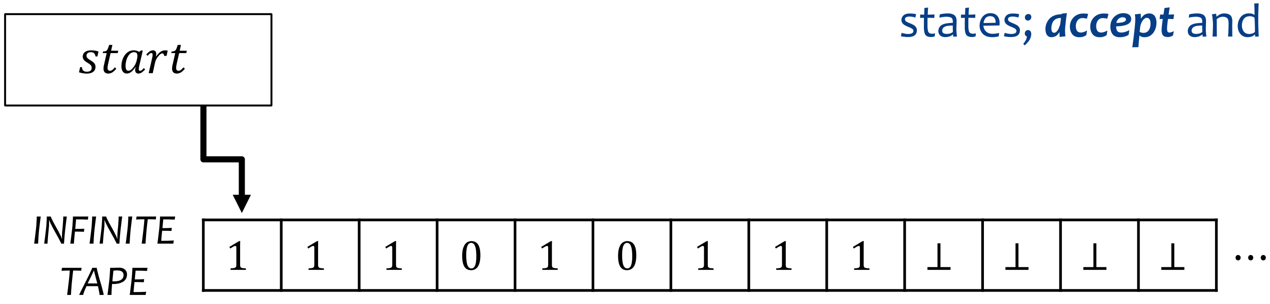
The "brain" of a TM is like a DFA, except it additionally specifies:

- what we write and
- whether move *left* or *right*

Note: "$a \to b, R$" means if the contents of the cell is a, then write b and move right.



There are also *two* special "termination" states; ***accept*** and ***reject***.



Turing Machine: Formal Definition

* A Turing machine is a 7-tuple:

$$M = \langle Q, \Gamma, \Sigma, \delta, q_{start}, q_{accept}, q_{reject} \rangle$$

  * Q = set of **states**
  * Σ = the **input** alphabet (typically {0,1} but not always)
  * $\perp$ = the **blank symbol**
  * Γ = the **tape alphabet** where generally $\Gamma = \Sigma \cup \{\perp\}$
  * $q_{start} \in Q$, = the **initial state**
  * $F = \{q_{accept}, q_{reject}\} \subseteq Q$, = the set of **final states** (one accepting state and one rejecting state)
  * $\delta: (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ = the **transition function**
  * **Definition:** M **accepts/rejects** x if, given x as input, M starts at q_{start} and reaches q_{accept}/q_{reject}, respectively, when making transitions according to δ.

Computability: Review

- **Question:** Which problems are solvable by a computer?
- **Answer:** Depends on what *solvable* is and what a *computer* is.
- **Definition:** A program M ***decides*** a language A if given x as input:
  - If $x \in A$, M *accepts* x ("return 1")
  - If $x \notin A$, M *rejects* x ("return 0")
- **Remark:** M is called a ***decider*** and *must* always halt; A is ***decidable***.
- **Definition:** The ***language*** of M, $\boldsymbol{L}(\boldsymbol{M}) = \{x : M \text{ accepts } x\}$
- **Definition:** M ***recognizes*** a language A if $A = L(M)$. In other words:
  - If $x \in A$, M *accepts* x
  - If $x \notin A$, M *either rejects* x *or* loops on x
- **Remark:** M is called a ***recognizer*** and *may not* always halt.

Turing Machine: Need to know

HERE IS WHAT **YOU**
NEED TO KNOW ABOUT TURING MACHINES!

A Turing Machine: Review

* **General:** Everything (PDF file, Photo, C++ code) is a binary string. The application is what makes sense of it.
* Storing/encoding a natural number n, requires $O(\log n)$ bits
* a Turing Machine (TM) M = a Program
* $\langle M \rangle$ – the source code of M
* **Remarks:**
  1. The source code $\langle M \rangle$ of a program is a binary string of **finite** length.
  2. An input x to a program is always of **finite** length.
  3. A source code $\langle M \rangle$ can serve as an input for another program.
  4. We can run a program on its source code.

A Turing Machine: Review

- **Algorithmic description:** Instead of writing C++ code or a 7-tuple, we give a high-level description that can be converted into code.
- **Example:** M on input x: ….
- **In C++:** "bool M(string x);"
- **Fact 1:** There are countably-many programs.
- **Fact 2:** There are uncountably-many languages.
- **Conclusion:** There are "more" languages then programs.
  Therefore, there exist **undecidable** (and in fact) **unrecognizable** languages.
  That is, a language A such that $L(M) \neq A$ for every M.

An Undecidable Language

* Let X be a list of languages $L(M_i)$ for all programs M_i.
* **Claim:** We can construct a language L^* that is not on list X by flipping the diagonal.
* Since X contains the language of every program, L^* is not the language of *any* program – it is undecidable!

X

	ε	0	1	00	01	10	$\cdots$
$L(M_1)$	0	0	0	0	0	0	$\cdots$
$L(M_2)$	0	1	0	1	0	0	$\cdots$
$L(M_3)$	1	1	0	1	0	1	$\cdots$
$L(M_4)$	0	0	1	1	0	1	$\cdots$
$L(M_5)$	1	1	1	0	1	0	$\cdots$
$L(M_6)$	0	1	1	0	1	0	$\cdots$
$\vdots$							$\ddots$

	ε	0	1	00	01	10	$\cdots$
L^*	1	0	1	0	0	1	$\cdots$

A Turing Machine: Review

- A language is the set of "yes" instances for a decision problem.
- **Example:** The Halting problem:
$$L_{\text{HALT}} = \{(\langle M \rangle, x) : M \text{ halts on } x\}$$
- **Question behind the language:** $(\langle M \rangle, x) \in L_{\text{HALT}}$?
- **In English:** "Does the program M halt on input x?"
- **Importance:** Is L_{HALT} decidable?
- **Answer:** No, but it is recognizable: simulate M on x (using the interpreter/Universal TM).
- **Explicit undecidable languages:** $L_{\text{ACC}}, L_{\text{HALT}}, L_{\varepsilon\text{-HALT}}, L_{\emptyset}, L_{\text{EQ}}$

The Barber Paradox

* **Claim:** $L_{\mathrm{BARBER}} = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\}$ is undecidable.
* **Proof:** Assume for contradiction some program B decides L_{BARBER}.
  * It implies $\langle P \rangle \in L_{\mathrm{BARBER}} \Leftrightarrow B$ accepts $\langle P \rangle$.
* **Question:** $\langle B \rangle \in L_{\mathrm{BARBER}}$?
* **Answer:** Suppose P is a program.

1. P accepts $\langle P \rangle \Longrightarrow \langle P \rangle \notin L_{\mathrm{BARBER}}$.
2. P does not accept $\langle P \rangle \Longrightarrow \langle P \rangle \in L_{\mathrm{BARBER}}$.

* **Question:** What if $P = B$?

1. $\langle B \rangle \in L_{\mathrm{BARBER}} \Rightarrow B$ accepts $\langle B \rangle \Longrightarrow \langle B \rangle \notin L_{\mathrm{BARBER}}$.
2. $\langle B \rangle \notin L_{\mathrm{BARBER}} \Rightarrow B$ does not accept $\langle B \rangle \Longrightarrow \langle B \rangle \in L_{\mathrm{BARBER}}$.

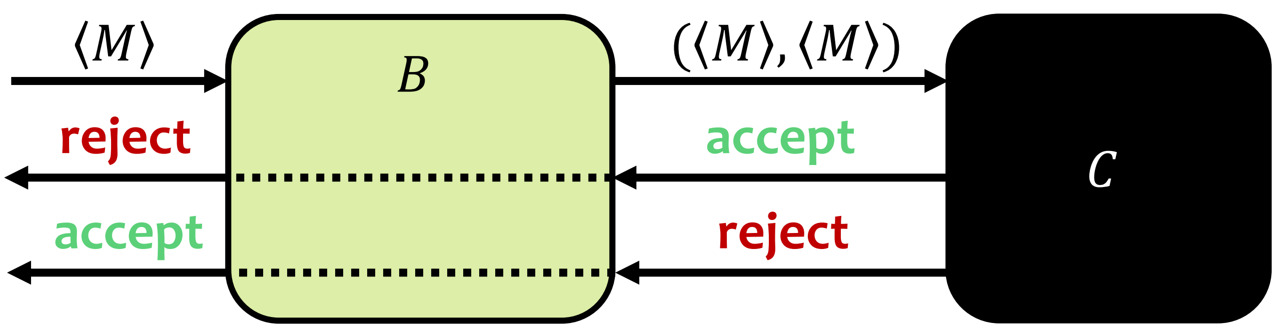
Contradiction!

L_{ACC} is Undecidable

We need to implement:
B is given one input: $\langle M \rangle$
M does not accept $\langle M \rangle \Longrightarrow B$ accepts $\langle M \rangle$
M accepts $\langle M \rangle \Longrightarrow B$ rejects $\langle M \rangle$

We have:
C is given two inputs: $\langle M \rangle$ and x
M accepts $x \Longrightarrow C$ accepts $(\langle M \rangle, x)$
M does not accept $x \Longrightarrow C$ rejects $(\langle M \rangle, x)$

* **Proof:** Assume (for contradiction) that a decider C exists for $L_{\text{ACC}} = \{(\langle M \rangle, x) : M \text{ accepts } x\}$. We can use C to construct a decider B for $L_{\text{BARBER}} = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\}$:



L_{HALT} is Undecidable

We need to implement:

C is given two inputs: $\langle M \rangle$ and x

M accepts $x \implies C$ accepts $(\langle M \rangle, x)$

M does not accept $x \implies C$ rejects $(\langle M \rangle, x)$

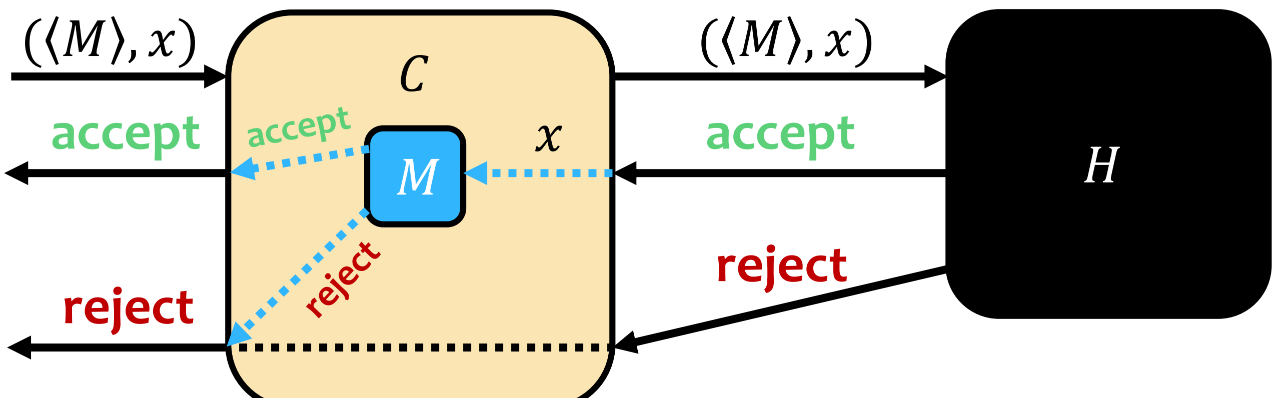
We have:

H is given two inputs: $\langle M \rangle$ and x

M accepts or rejects $x \implies H$ accepts $(\langle M \rangle, x)$

M loops on $x \implies H$ rejects $(\langle M \rangle, x)$

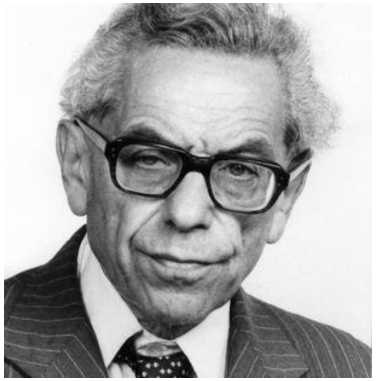
* **Claim:** $L_{\text{HALT}} = \{(\langle M \rangle, x) : M \text{ halts on } x\}$ is undecidable.
* **Proof:** Assume (for contradiction) that a decider H exists for L_{HALT}. We can the construct a decider C for $L_{\text{ACC}} = \{(\langle M \rangle, x) : M \text{ accepts } x\}$:



Conclusion

- The Halting Problem is undecidable although it is a fundamental problem in software and hardware design!
- **Question:** Perhaps the problem is easy for small programs?
- **Collatz Conjecture:** This program halts for every n:

```
int n;
while (n > 1) {
  n = (n%2) ? 3*n+1 : n/2;
}
```



Paul Erdös offered $500 for this problem!

Decidability and Reducibility

- **Question:** How do we show undecidability of a language?
- **Answer:**
  - Directly: L_{BARBER} is undecidable.
  - Indirectly: If L_{HALT} is decidable so is L_{ACC}.
- **Definition:** Language A is ***Turing reducible*** to language B, written $\boldsymbol{A \leq_T B}$, if there exists a program M that decides A using a "*black box*" that decides B.
- **Intuition:** A is "*no harder*" than B to solve.
- **Theorem:** Suppose $A \leq_T B$. Then B is decidable $\Longrightarrow$ A is decidable.
- **Contrapositive:** Suppose $A \leq_T B$. Then A is *undecidable* $\Longrightarrow$ B is *undecidable*.
- **Strategy:** Pick an undecidable language A and show that $A \leq_T B$.

Proving a Language is Unrecognizable

- **Claim:** If a language A and its complement $\overline{A}$ are both *recognizable*, then A is *decidable*.
- **Observation:** If a language A is undecidable, then at least one of A or $\overline{A}$ must be unrecognizable. *(contraposition of the above)*
- **Conclusion:** If A is undecidable but recognizable, then $\overline{A}$ is unrecognizable.
- **Example:** $\overline{L_{\text{ACC}}}$ is unrecognizable
  - L_{ACC} is undecidable *(proof by contradiction)*
  - L_{ACC} is recognizable *(the universal TM U is a recognizer for L_{ACC})*
  - $\Longrightarrow \overline{L_{\text{ACC}}}$ must be unrecognizable

Type of Questions

- Multiple choice
- True / False
- Always / Sometimes / Never
- Short Answer
- Free Response

Good luck!