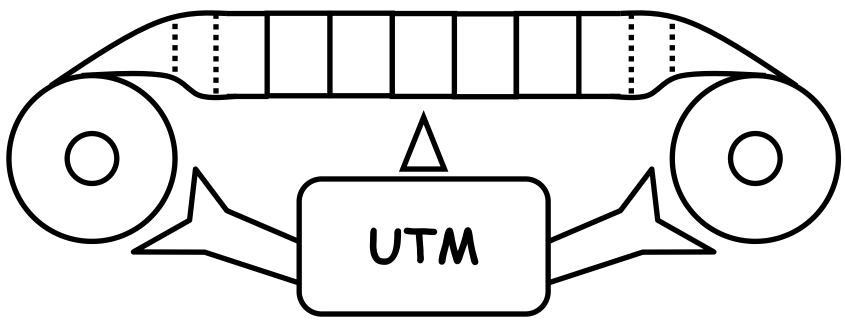
Computational Complexity



Hiroki Sayama
sayama@binghamton.edu

Information and computation

- So far, complexity of a system has been characterized by the amount of information it carries

- Today we will review other approaches that characterize the complexity from a perspective of "computational mechanisms"

Computation

Math vs. computation

- Math is a set of static logical truths
  - It describes logical statements and their relationships
  - It provides pathways leading to other statements, but doesn't tell where to go
- Computation is a dynamic process
  - It executes specific operations in order
  - Computer (either machine or human) has its own internal states, I/Os, etc.

Computation?

A sequence of rewritings applied to a symbolic representation of something

- Rewriting rules are predefined and fixed so that:
  - Process of computation is efficient
  - Final result is accurate and useful

Complexity of Languages

A "language" is a set of grammatically valid expressions

- my dog ate my homework
- my homework my dog ate
- my homework ate my dog
- homework my dog ate my
- dog my my homework ate
- my dog eated my homework
- my god my homework late

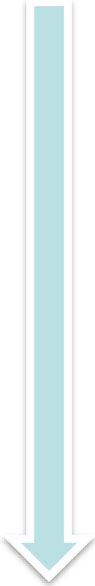
Formal languages and automata

- Formal language: A set of strings that are grammatically valid
  - Several distinct classes known
- Complexity of a formal language can be characterized by the class of "automata" that can recognize it
  - After "hearing" a string, the automaton's internal state decides whether that string is in the language or not

Chomsky's hierarchy

- Regular languages
  - Finite state automata can recognize
- Context-free languages
  - Pushdown automata can recognize
- Context-sensitive languages
  - Linear bounded automata can recognize
- Recursively enumerable languages
  - Turing machines can recognize

Low complexity



High complexity

Examples of formal languages

- Regular languages:
  - { $(01)^n$ }, { bit strings in which 0's and 1's exist in even number each }
- Context-free languages:
  - { 0^n1^n }, { bit strings in which 0's and 1's exist in the same number }, most programming languages
- Context-sensitive languages:
  - { $0^n1^n2^n$ }, most natural languages (weakly context-sensitive)

Automaton (pl.: automata)

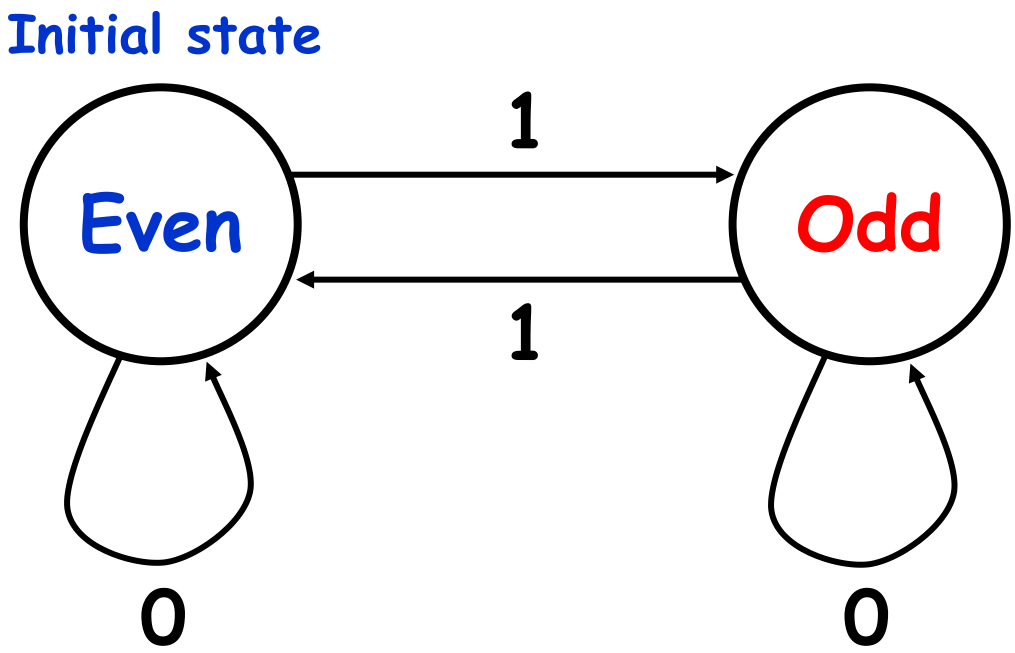
- A formal representation of dynamic, computational behavior of machines
- Has internal states
- Changes its states and produces outputs over time according to predefined rules that refer to its own states and inputs received
- States and time are usually discrete

Finite state automaton

- Has only finite number of states
  - Has no infinite memory
    - Therefore, all physically built computational systems are in this class in principle
  - Can recognize regular languages
  - Often used in complex systems modeling
    - E.g. cellular automata

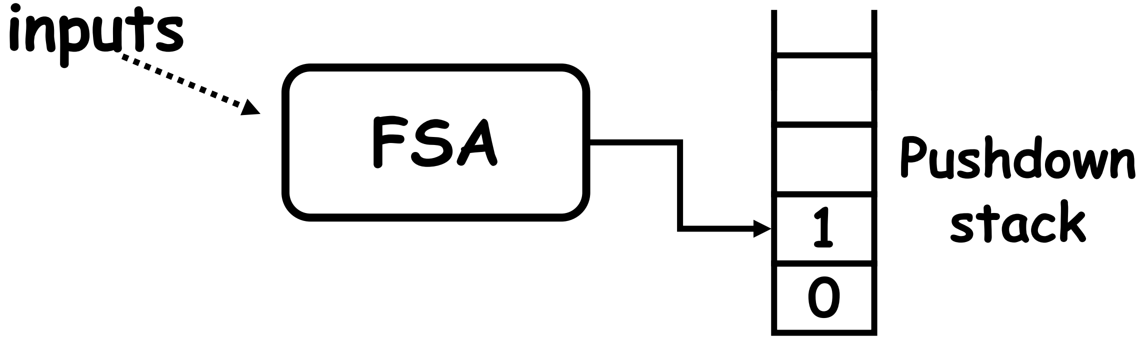
Example

- Parity checker
  - Tells whether the number of 1's included in an input (bit string) is even or odd



Pushdown automaton

- Finite state automaton with an infinitely long pushdown stack
  - Has infinite LIFO memory
  - Non-deterministic PDA can recognize context-free languages
    - Can handle nested structures

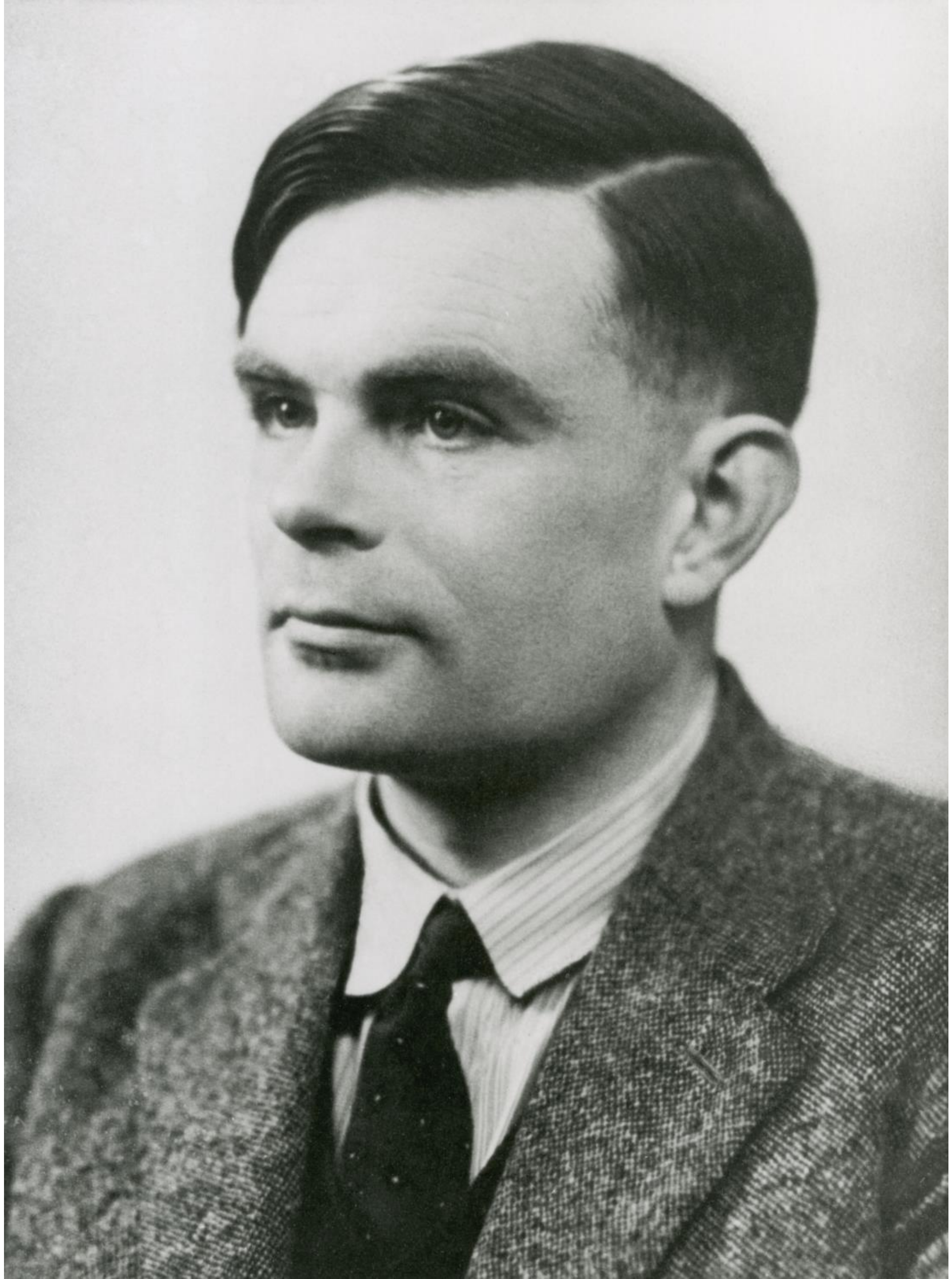


Linear bounded automaton

- (Non-deterministic) Turing machine whose tape length is a linear function of the length of input
  - Has infinite memory but its accessible range is bounded according to input size
  - Can recognize context-sensitive languages

Turing Machines


2014 © The Imitation Game



Alan Turing
(1912-1954)

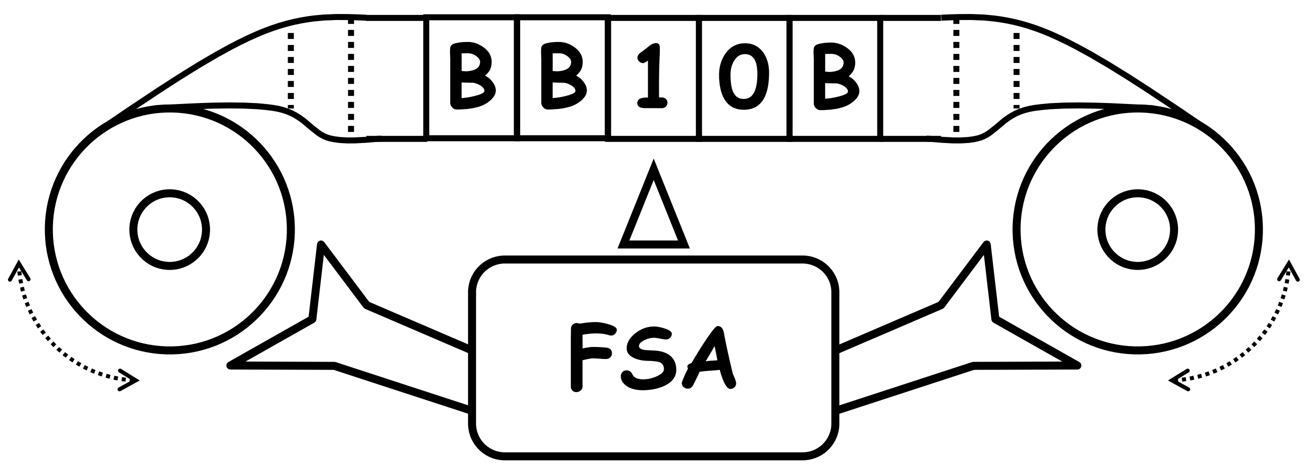
"Computer" back then

(Image from Wikipedia)

(Image from Hidden Figures (2016))

Turing machine (TM)

- Finite state automata with an infinitely long memory tape
  - Has a read/write head that can move on the tape left and right



Mathematical definition

- Turing machine M:

$$M = \langle S, \Sigma, f, q_0, H \rangle$$

S: A finite set of states

Σ: A finite set of tape symbols

- Includes "B" for blank

f: State-transition function

- $S \times \Sigma \rightarrow S \times \Sigma \times \{R, L, N\}$ (motion of head)

q_0: Initial state (in S)

H: A set of halting states (subset of S)

Rule table

f: State-transition function

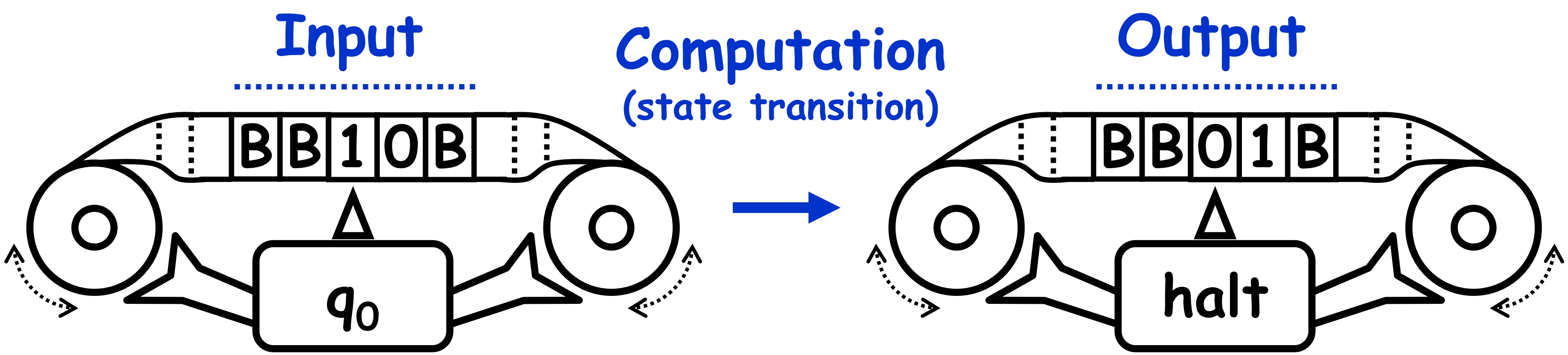
- $S \times \Sigma \rightarrow S \times \Sigma \times \{R, L, N\}$ (motion of head)

Often written as a set of "$s\sigma s'\sigma' m$"

- s: Current state of FSA
- σ: Symbol read from the tape
- s': Next state of FSA
- σ': Symbol to be written to the tape
- m: Direction of motion of the head

Computation by a Turing machine

- **Input**: Initial contents of the tape
- **Output**: Contents of the tape when the TM halts
  - TM halts when the FSA reaches one of its halting states

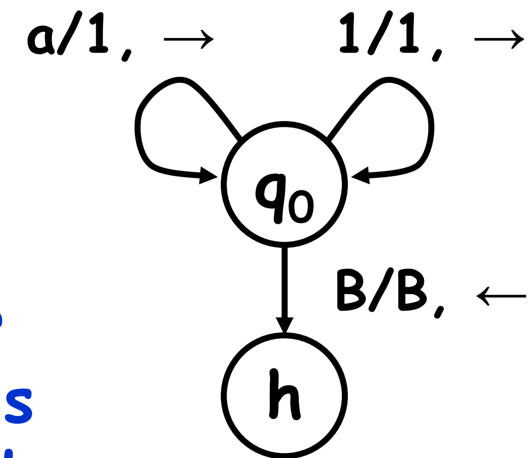


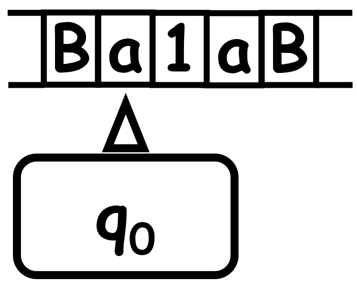
Example

$S = \{ q_0, h \}$, $\Sigma = \{ B, a, 1 \}$

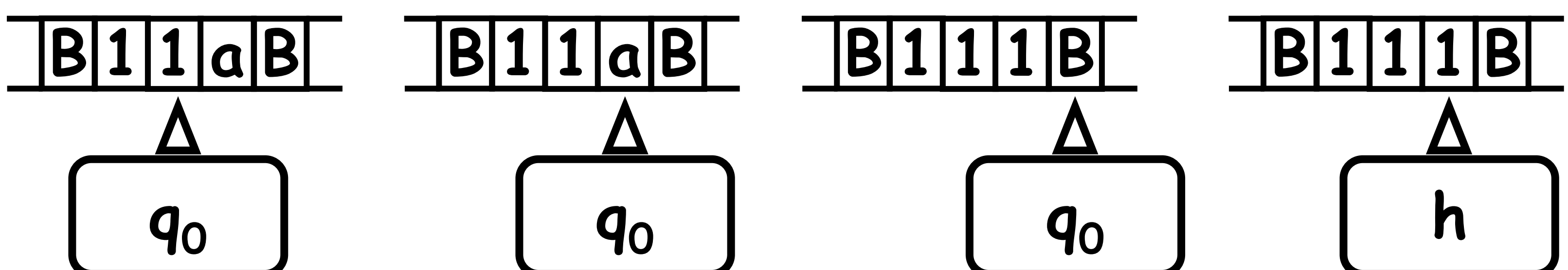
$f = \{ q_0 a q_0 1R, q_0 1 q_0 1R, q_0 B h B L \}$

$H = \{ h \}$





Writing 1's over non-blank symbols moving rightward until it reaches a "B"



Exercise

- Figure out what the following TM does:

$$
\begin{aligned}
S &= \{\ q_0,\ e,\ o,\ h\ \} \\
\Sigma &= \{\ B,\ 1,\ E,\ O\ \} \\
f &= \{\ q_0 1 o 1 R,\ q_0 B h E N, \\
&\qquad e 1 o 1 R,\ e B h E N, \\
&\qquad o 1 e 1 R,\ o B h O N\ \} \\
H &= \{\ h\ \}
\end{aligned}
$$

Exercise

- Design a Turing machine that moves its head rightward and keeps inverting bits written in its tape until its head reaches a blank symbol

Computational universality of TMs

- Church-Turing thesis:

  "Every mathematical function that is naturally regarded as computable is computable by a Turing machine"

  - Not a rigorous theorem or hypothesis, but an empirical "thesis" widely accepted

$\Downarrow$

TMs are considered "computationally universal"

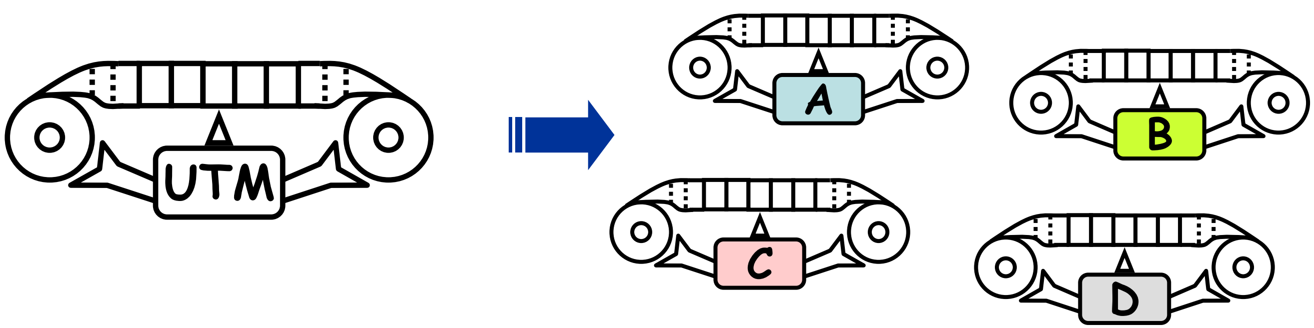
Universal Turing machines (UTM)

- More important fact shown by Turing:

  There are TMs that can emulate behaviors of any other TMs if instructions are given (software)

  ⬇

  A single TM can be computationally universal just by itself!



Intuitive reason

- C-T thesis: TMs can compute any naturally computable process
- It is possible to emulate the behavior of a certain TM using paper and pencil

  → Emulation of TMs is a naturally computable process

  → It must be done by a TM too

Computational Complexity

Computational complexity

- Complexity measurement for an "algorithm" – a finite sequence of operations given to a universal Turing machine (or any universal computer)
- How much time and/or space the machine takes to solve a problem or produce a system
  - Evaluated for worst or average cases

Time complexity

- How many steps does your algorithm need to take to produce a solution?

- Number of steps is represented as a function of the input size, $f(n)$
- Then its dominant term is extracted as the "order" of computational complexity

Big-O notation

- $O(***)$: "Order" ***

- $O(\log n) < O(n)$
- $O(n) < O(n^2) < O(n^3) < O(n^4)$ ...
- $O(n^k) < O(k^n)$ $(k > 1)$
- $O(k^n) < O(n!)$

etc...

Example

- $p(x) = a_0 + a_1x + a_2x^2 + \ldots a_nx^n$

- For a naïve algorithm:

  $f(n) = n(n+3)/2 \quad \Rightarrow O(n^2)$

- For Horner's algorithm:

  $f(n) = 2n \quad \Rightarrow O(n)$

Why do we care only dominant terms?

- When n is small, the speed of computation will likely be determined *not* by the algorithm but by other parts (e.g., inputs/outputs)

- The primary reason why we want an efficient algorithm is that we want to quickly process a large amount of data (i.e., large n)

Exercise

- Find the order of computational complexity for each of the following:

$f(n) = n + 2 \log n + 0.8^n$

$f(n) = n^5 + 2^{n/5}$

$f(n) = n^{2.5} + n^2 \log n$

Exercise: Simple search

- Consider a task to search for a name of a student from his/her ID
  - Available data to be searched:
    A long list of "(ID, name)" data entries
  - Input: ID
  - Output: Name that corresponds to the given ID

Exercise: Simple search

- Design an algorithm for each of the following cases and then evaluate the order of its computational complexity
  - When the list is in random order
  - When the list is sorted according to ID's numerical values

Exercise

- Evaluate the computational complexity of some of the codes you have recently written
- Is there any way you can reduce its computational complexity?

Exercise

- Assume there are 6 algorithms with the following complexity orders:

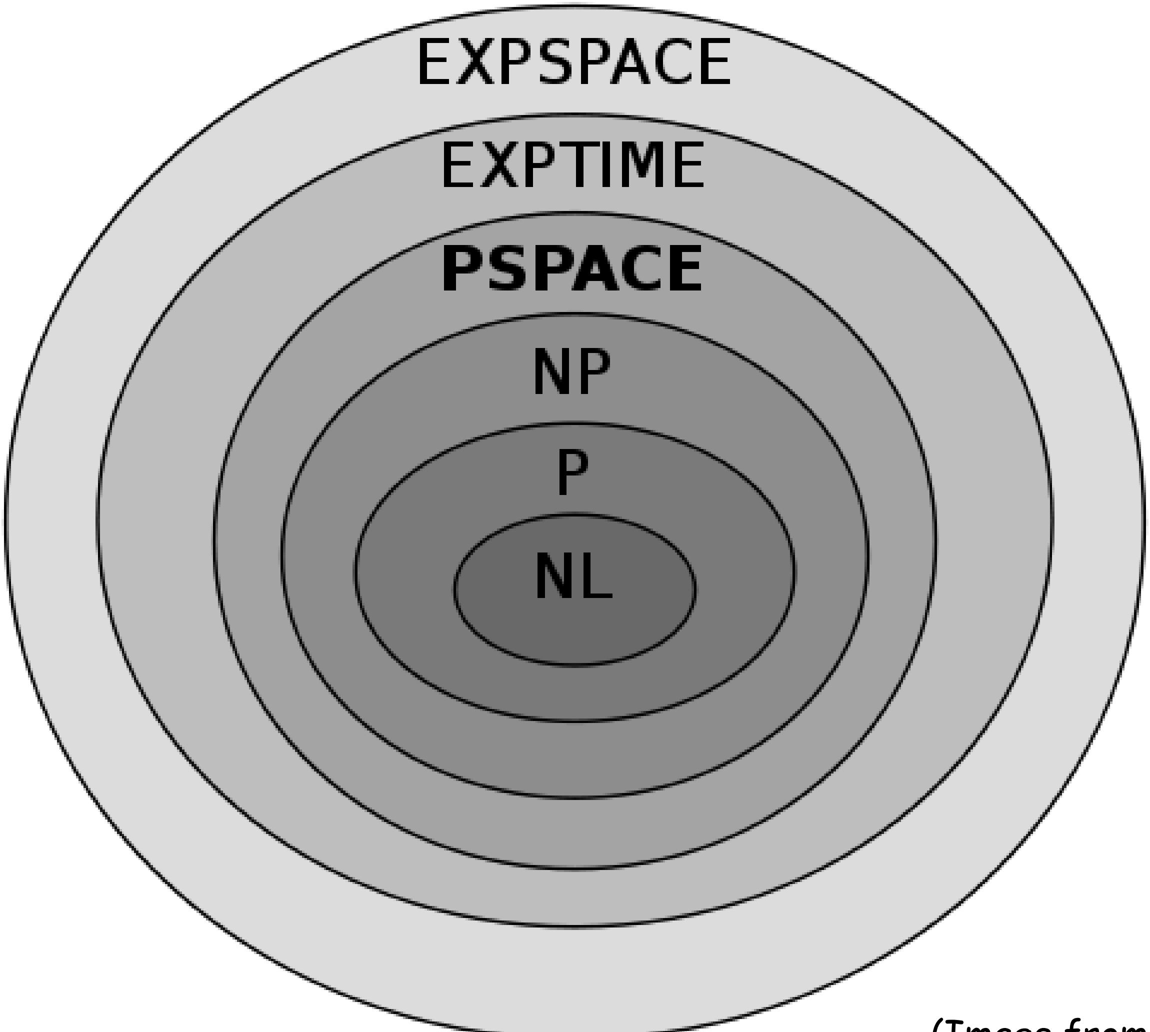
  $n, \; n \log n, \; n^2, \; n^3, \; 2^n, \; n!$

- If these are the actual numbers of steps and if each step takes 10^{-8} seconds, how does the computational time grows as n increases?
- Calculate time length for $n = 10^1 \sim 10^5$

Polynomial vs. exponential

- Polynomial-time algorithms
  - Time complexity: $O(n^k)$, $O(\log n)$ etc.
  - "Practical algorithms"
  - Faster computers do help solve larger problems
- Exponential-time algorithms
  - Time complexity: $O(k^n)$, $O(n!)$, $O(n^n)$ etc.
  - "Impractical algorithms"
  - Faster computers do NOT help solve larger problems!

Hierarchy of computational complexities



P≠NP?

(Image from Wikipedia)

Exercise

- Assume there are 6 algorithms with the following complexity orders:

  $n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n, \quad n!$

- If a computer becomes 100 times faster, how much larger a problem can each algorithm solve within the same given time period?
  - Assume large n

Uncomputable Problems

Uncomputable problems (1)

- Problems whose computational complexity exceeds the physical limit of computational power of Earth (or the Universe, or whatever)

Bremermann's limit

- Maximal information processing speed:

  1.36×10^{50} bits/sec/kg

- Maximal amount of information:

  10^{93} bits

  - Amount of information that can be processed using the entire mass of Earth within a time period of its age

Lloyd's estimate

- Lloyd, S. (2000) Computational capacity of the Universe. Phys. Rev. Lett. 88, 237901

"The Universe can have performed 10^{120} ops on 10^{90} bits (10^{120} bits including gravitational degrees of freedom)."

Transcomputational problems

- Many real-world problems are "transcomputational"
  - Problems that can't be solved under the physical limit of Earth/the Universe if solutions are sought exhaustively
  - Traveling salesman problem
  - Integrated circuit testing
  - Cracking cryptographic keys (for 512-bit keys)

Uncomputable problems (2)

- Problems for which no natural procedure of computation exists

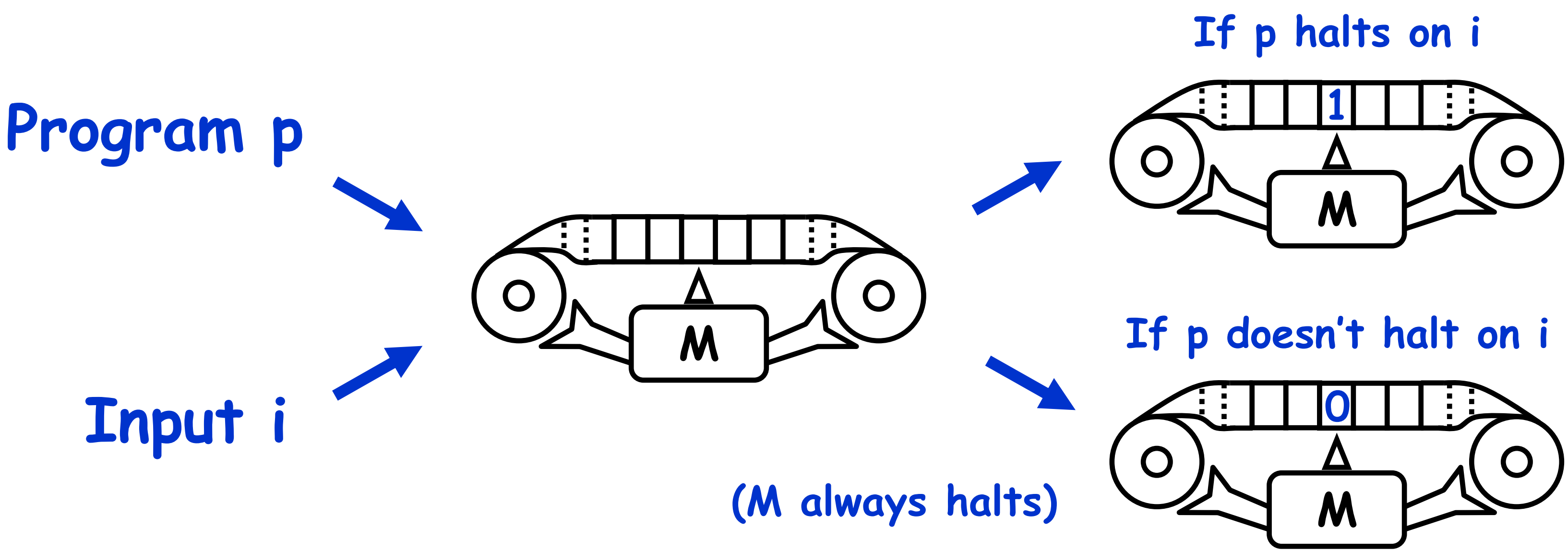
Example: The halting problem

- Given a description of a computer program and an initial input it receives, determine whether the program eventually finishes computation and halts on that input
- Is there a general procedure to solve this problem for any arbitrary programs and inputs?

No

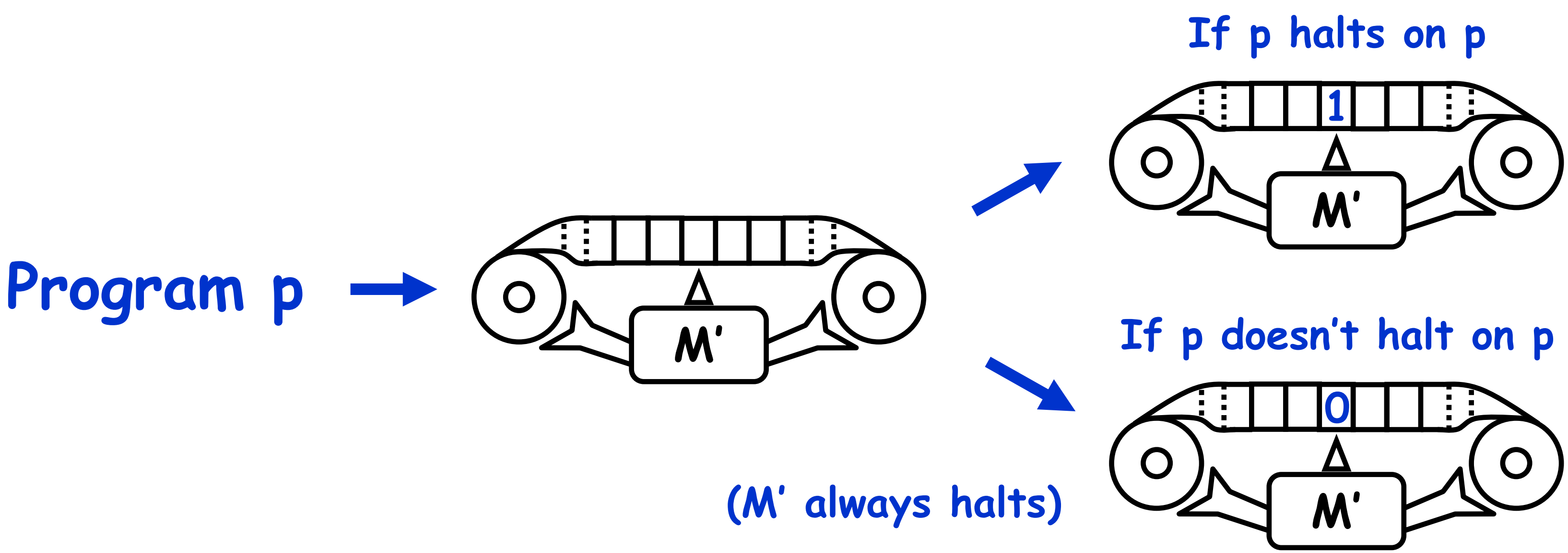
Proof: Reductio ad absurdum (1)

- Assume there is a general algorithm (and a TM, called M) that can solve the halting problem for any program p and input i
  - Output of M: f(p, i) = 1 if program p halts on input i; 0 otherwise



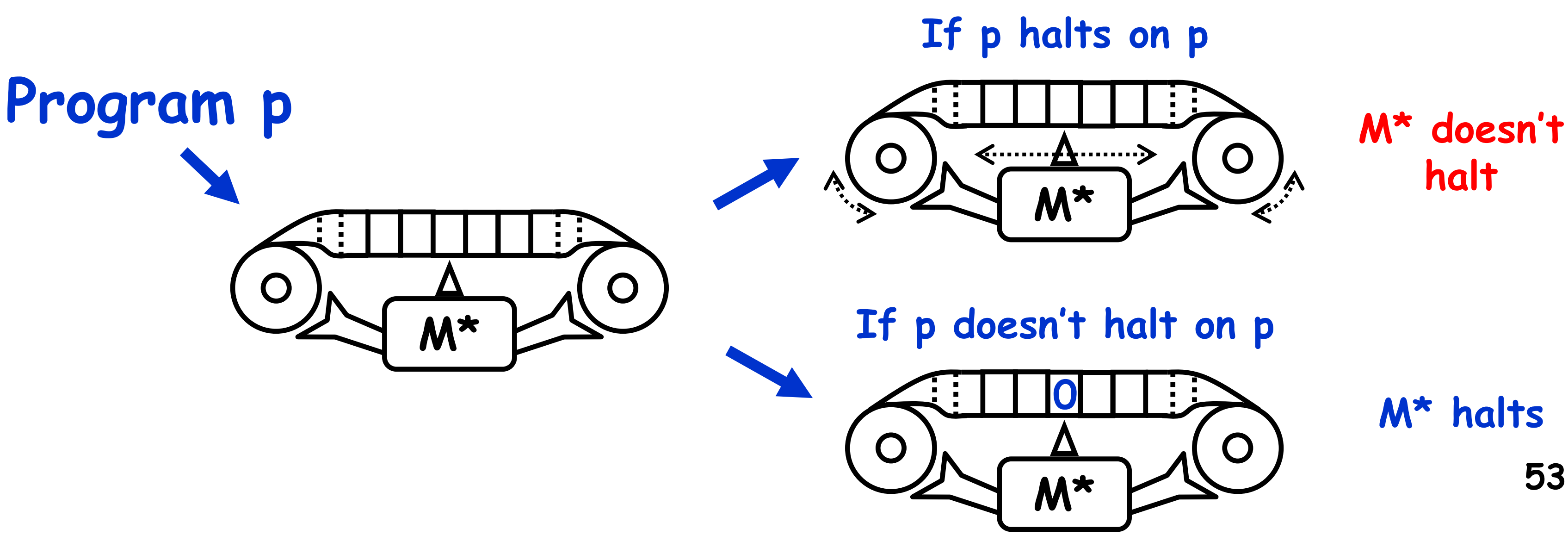
Proof: Reductio ad absurdum (2)

- One can easily derive another TM M' from M so that it computes only diagonal components in the p-i space
  - Output of M': $f'(p) = f(p, p)$



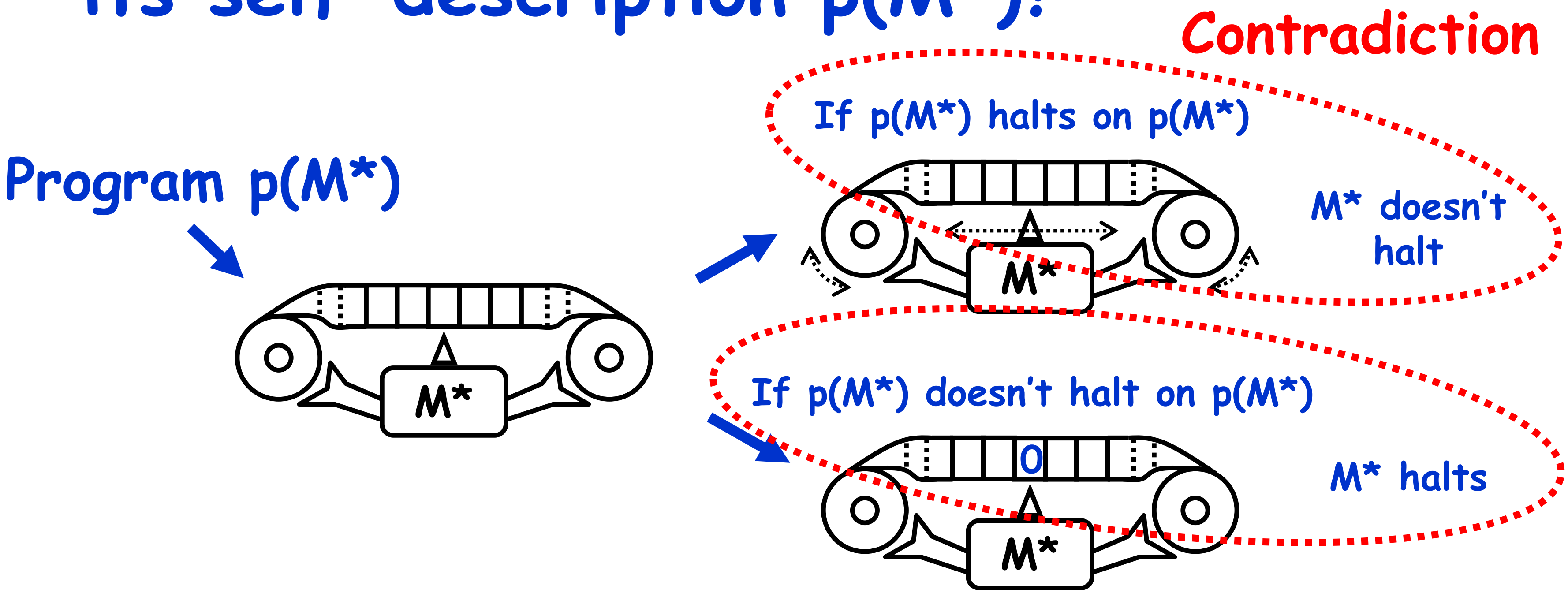
Proof: Reductio ad absurdum (3)

- One can further tweak M' to make another TM M* that falls into an infinite loop if f'(p) = 1
  - Output of M*: f*(p) = 0 if f'(p) = 0; doesn't halt otherwise



Proof: Reductio ad absurdum (4)

- What could happen if M* is given with its self-description p(M*)?



- Our initial assumption must be wrong: No general algorithm exists

Turing's theorem

- The halting problem of Turing machines is *undecidable* by Turing machines
  - Similar to Gödel's incompleteness theorems, informally stated as:

    For any consistent, "powerful enough" axiomatic system, (1) there must be a statement that it can neither prove or disprove, and (2) its own consistency cannot be proven by itself

Example of statements that are neither provable nor disprovable

"This statement is unprovable"

- If the system proves this, it means that the system proves "This statement is unprovable" → contradiction
- If the system disproves this, it means that the system proves that "This statement is unprovable" is provable → contradiction
- Neither is possible!

Self-reference and dynamical systems

- All of these messy things arise from "self-reference"
- Logic is inherently static, but self-reference makes it a dynamic process that develops over "time" (or logical reasoning steps)

For those who know networks…

- A "dynamic" view of logic:
  - An axiomatic system defines an infinitely large network of all possible logical expressions (nodes) connected by logical relationships
  - Truth values are states of nodes and propagates through inference
  - Incompleteness theorems say that the final attractor of this network must be non-stationary if the network includes self-referencing loops

Summary

- Computation = dynamic form of logic, often modeled using automata
- Time & space complexities
- There are limits of computational abilities in any physical systems
- Some problems are inherently uncomputable (could be due to its self-referencing nature)