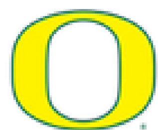
Turing Machines

Part III

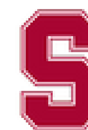
Yesterday, 6:00 PM (PT)
Stanford Stadium, Stanford, California



2 Oregon
Ducks
(8-1)

20 - 26
Final

Stanford 6
Cardinal
(8-1)



	1	2	3	4	Total
Oregon	0	0	0	20	20
Stanford	7	10	6	3	26

Review: **RE** Languages

- The language of a TM M (denoted $\mathscr{L}(M)$) is the set of all strings M accepts.
- A language L is called **recognizable** (equivalently, $L \in$ **RE**) iff L is the language of some TM.
- Intuitively: **RE** languages are languages where for each string in the language, there is some “proof” thatshows the string is in the language.
  - The TM for L can work by searching all possible proofs to see if any of them work.
  - The TM will definitely accept w if $w \in L$ after finding the proof.
  - There is not guarantee the TM will halt and produce an answer if $w \notin L$.

Review: NTMs

- A **nondeterministic Turing machine** (or **NTM**) is a Turing machine where there can be multiple transitions defined for a given state/symbol combination.
- An NTM N accepts a string w iff there exists a choice of transitions N can follow when run on w that causes it to accept.
- Intuition: guess-and-check:
  - **Nondeterministically** guess some information that will help prove w is in the language.
  - **Deterministically** verify that the information proves that w is in the language.

The Story So Far

- We now have two different models of solving search problems:
  - Build a worklist and explicitly step through all options.
  - Use a nondeterministic Turing machine.
- Are these two approaches equivalent?
- That is, are NTMs and DTMs equal in power?

DFAs and NFAs

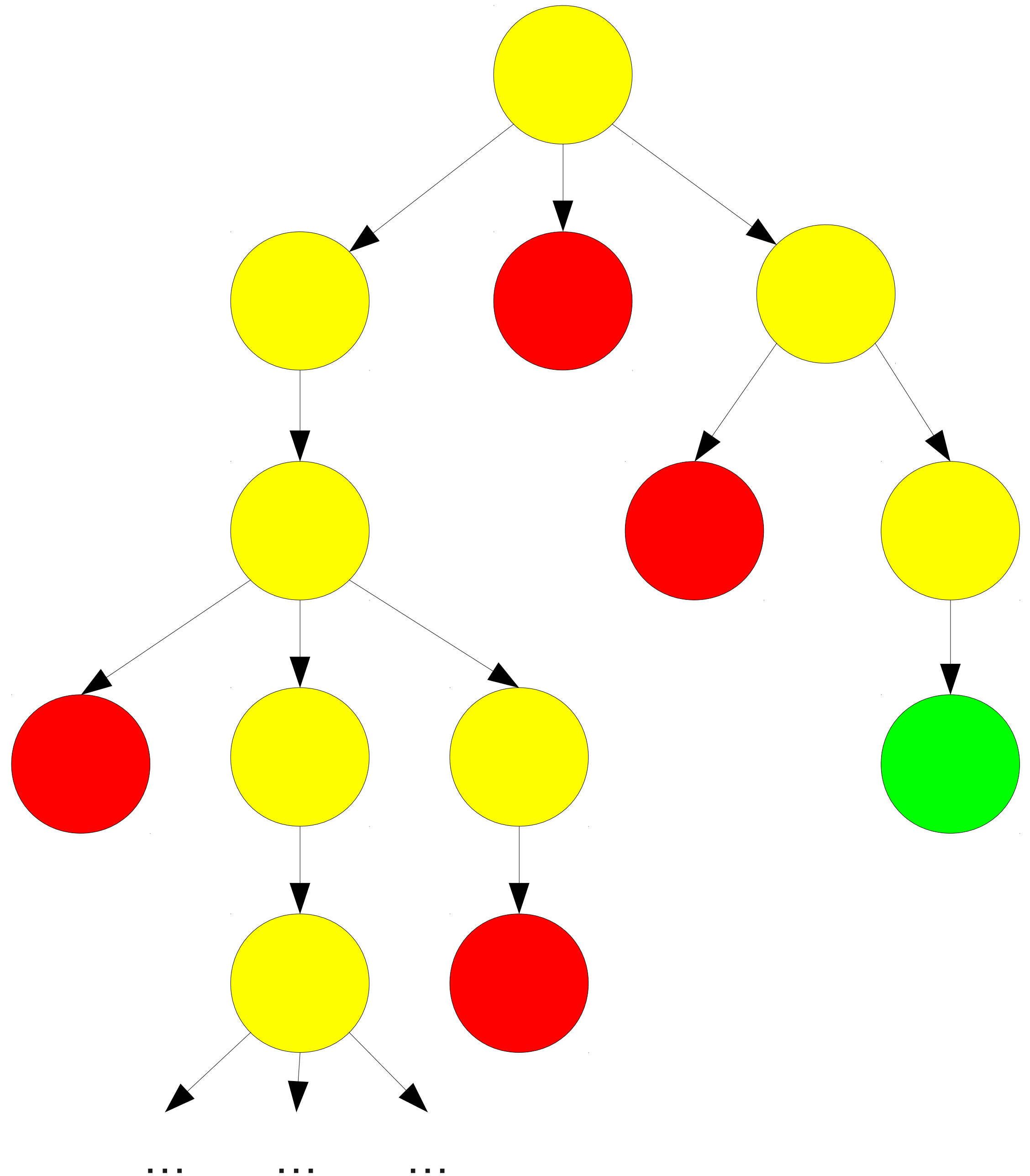
- Earlier, we proved that any NFA can be converted into a DFA using the subset construction.
  - Can exponentially increase the size of the NFA; an n-state NFA turns into a DFA with at most 2^n states.
- Why did this work?
  - Only memory in an NFA is what set of states is currently active.
  - Only finitely many sets of states possible.
- Can we directly apply this construction to Turing machines?
  - **No:** Memory also includes tape, and there are infinitely many possible tapes!

DTMs and NTMs

- ***Theorem:*** For any NTM N, there is a DTM D such that $\mathscr{L}(N) = \mathscr{L}(D)$.
- Nondeterminism does not add any more power than determinism!
- **Proof idea:** Build a DTM that simulates the behavior of the NTM.
- Challenges:
  - Each branch of the NTM has its own tape; how can we store this?
  - Need to build states in a way that simulates the operation of the NTM; how to simulate nondeterminism with determinism?

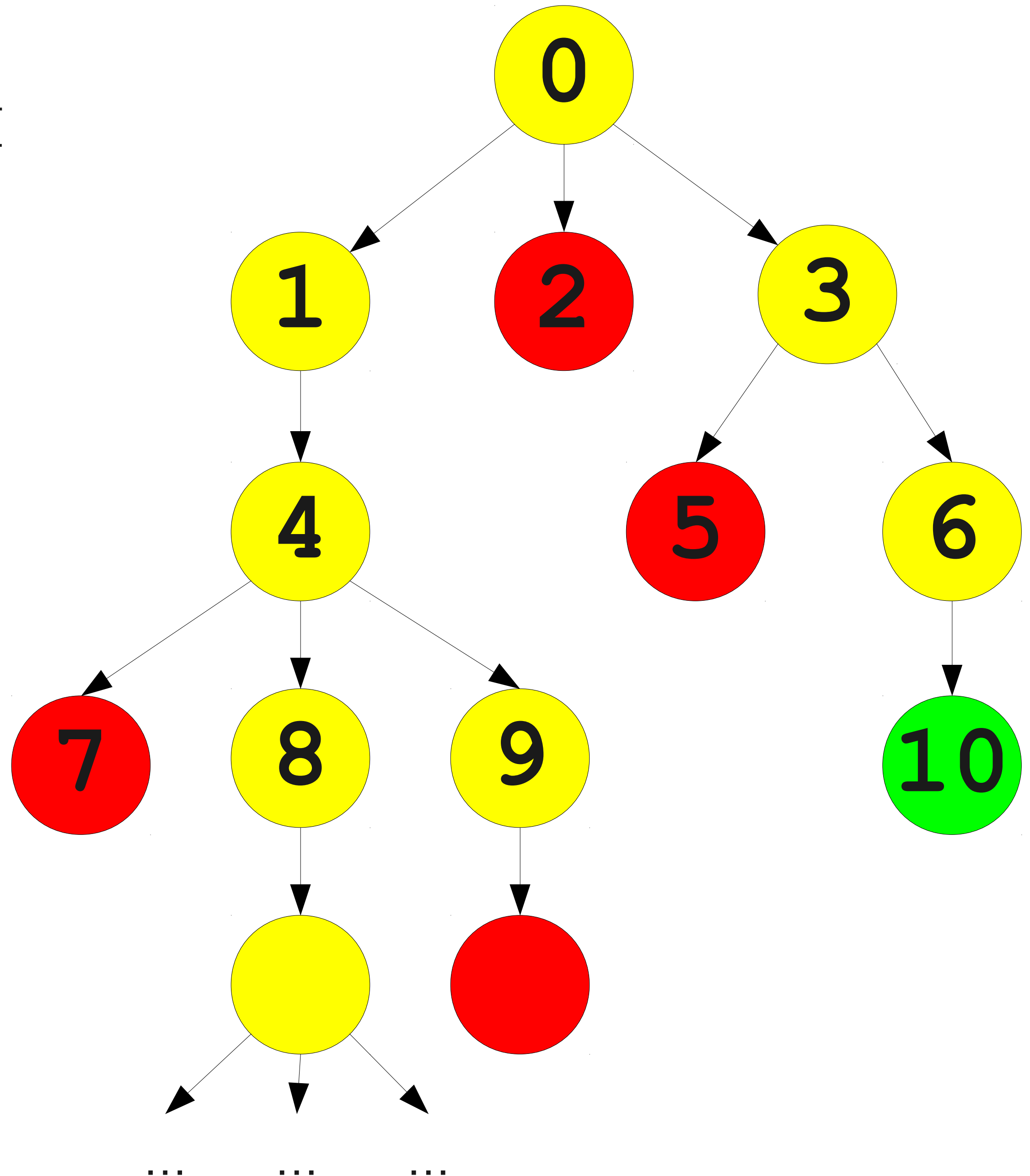
Review: Tree Computation

- One interpretation of nondeterminism is as a **tree computation**.
- Each node in the tree has children corresponding to each possible choice for the computation.
- The computation accepts if *any* node in tree enters an accepting state.



The Key Idea

- **Idea:** Simulate an NTM with a DTM by exhaustively searching the computation tree!
- Start at the root node and go layer by layer.
- If an accepting path is found, accept.
- If all paths reject, reject.
- Otherwise, keep looking.



Exploring the Tree

- Each node in this tree consists of one possible configuration that the NTM might be in at any time.
- What does this consist of?
  - The contents of the tape.
  - The position of the tape head.
  - The current state.

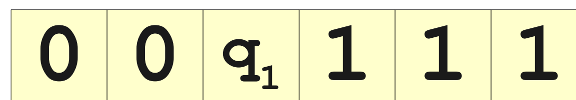
Instantaneous Descriptions

- At any instant in time, only finitely many cells on the TM's tape may be non-blank.
- An **instantaneous description** (ID) of a Turing machine is a string representation of the TM's tape, state, and tape head location.
  - Only store the “interesting” part of the tape.
- There are many ways to encode an ID; we'll see one in a second.

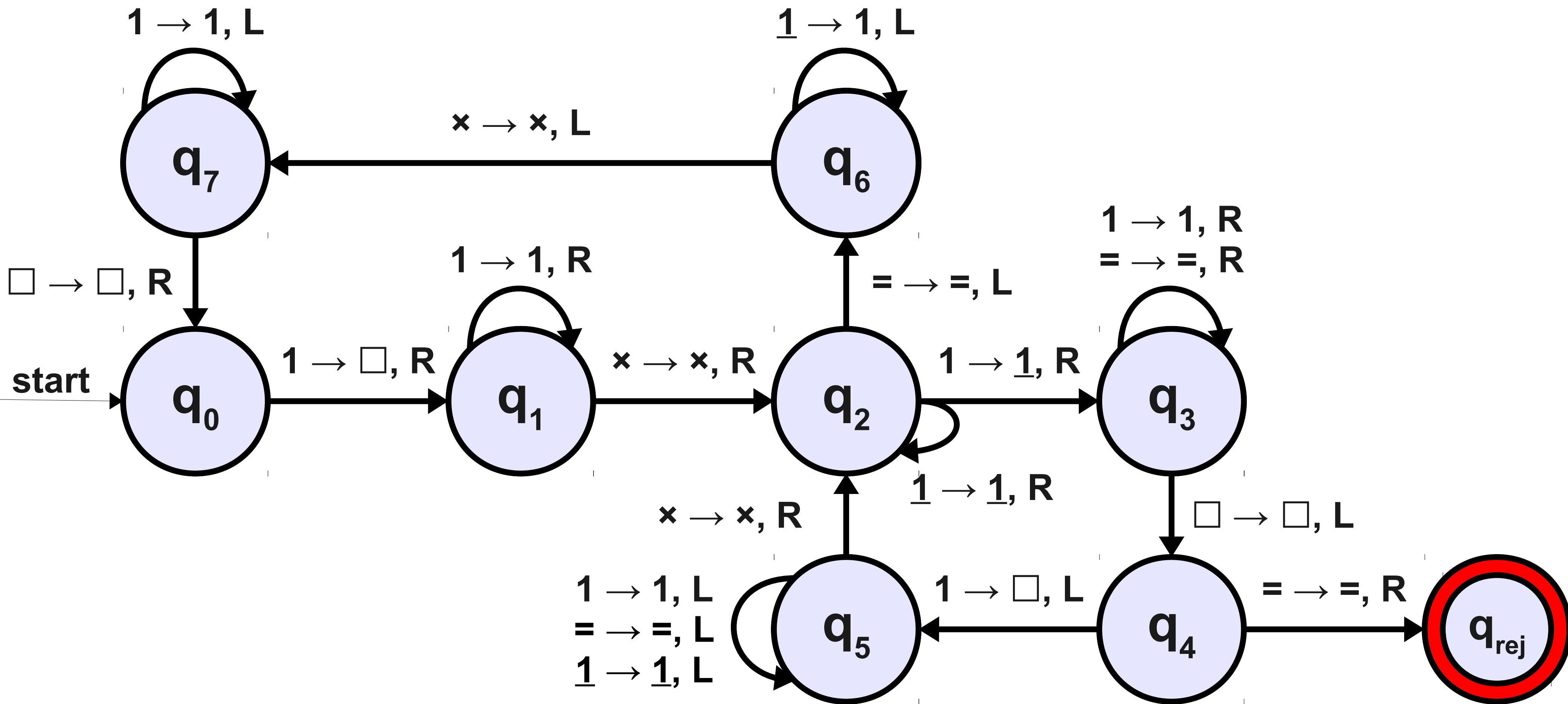
Building an ID

- Start with the contents of the tape.
- Trim "uninteresting" blank symbols from the ends of the tape (though remember blanks under the tape head).
- Insert a marker for the tape head position that encodes the current state.

□ → □, R
0 → 0, R
0 → 0, L
1 → 1, L
1 → □, L
q_3
q_2
start
□ → □, R
□ → □, L
1 → □, R
0 → □, R
q_{acc}
q_0
q_1
0 → 0, R
1 → 1, R
□ → □, R
q_{acc}



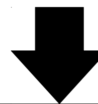
Rehydrating IDs

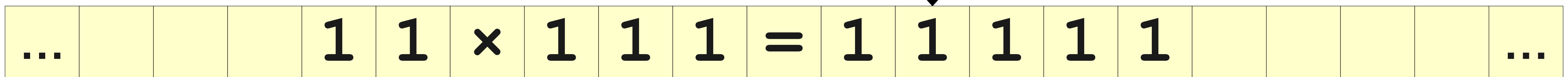


1	1	×	1	1	1	=	1	q_3	1	1	1	1

Rehydrating IDs

1 → 1, L
$\underline{1}$ → 1, L
× → ×, L
q7
q6
1 → 1, R
= → =, R
□ → □, R
1 → 1, R
= → =, L
start
q0
1 → □, R
q1
× → ×, R
q2
1 → $\underline{1}$, R
q3
$\underline{1}$ → $\underline{1}$, R
× → ×, R
□ → □, L
1 → 1, L
= → =, L
$\underline{1}$ → $\underline{1}$, L
q5
1 → □, L
q4
= → =, R
qrej





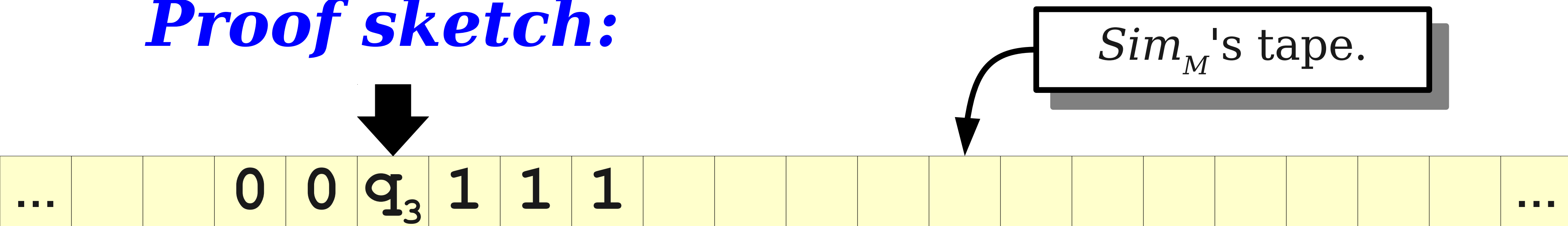
Manipulating IDs

Theorem: For any TM M, it is possible to build a second TM Sim_M that, given a a tape containing an ID of M, simulates one step of M's computation.

Manipulating IDs

Theorem: For any TM M, it is possible to build a second TM Sim_M that, given a a tape containing an ID of M, simulates one step of M's computation.

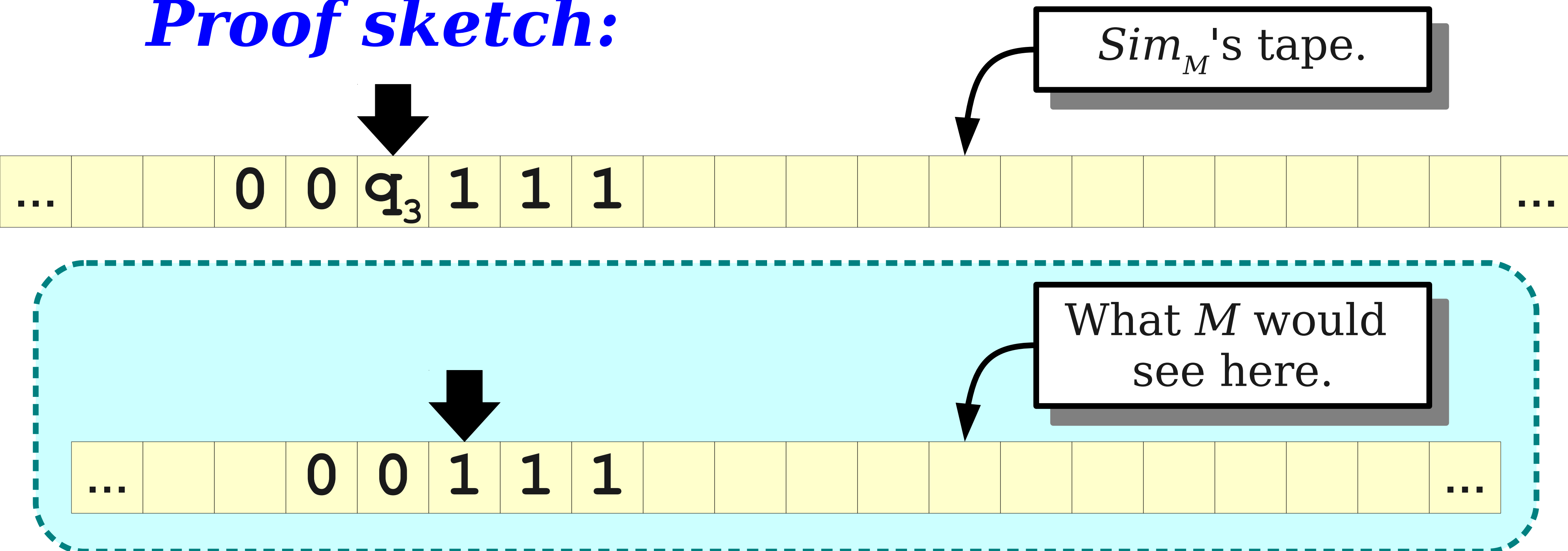
Proof sketch:



Manipulating IDs

Theorem: For any TM M, it is possible to build a second TM Sim_M that, given a a tape containing an ID of M, simulates one step of M's computation.

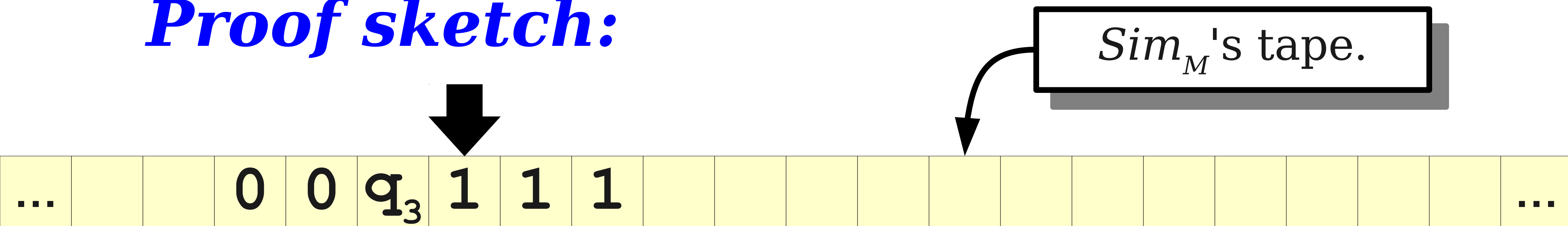
Proof sketch:



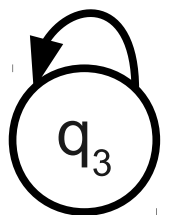
Manipulating IDs

Theorem: For any TM M, it is possible to build a second TM Sim_M that, given a a tape containing an ID of M, simulates one step of M's computation.

Proof sketch:



Sim_M now knows that **M** is in state **q_3** and reading **1**.

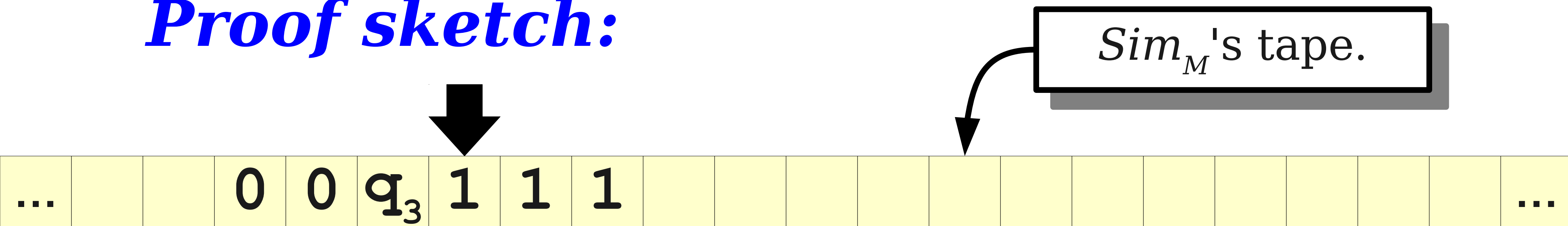


(Hypothetically, suppose that this is the state **q_3** in the machine **M**)

Manipulating IDs

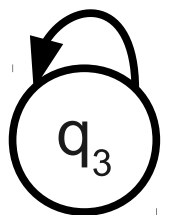
Theorem: For any TM M, it is possible to build a second TM Sim_M that, given a a tape containing an ID of M, simulates one step of M's computation.

Proof sketch:



Sim_M now knows that **M** is in state **q_3** and reading **1**.

0 → 1, L
1 → 0, L

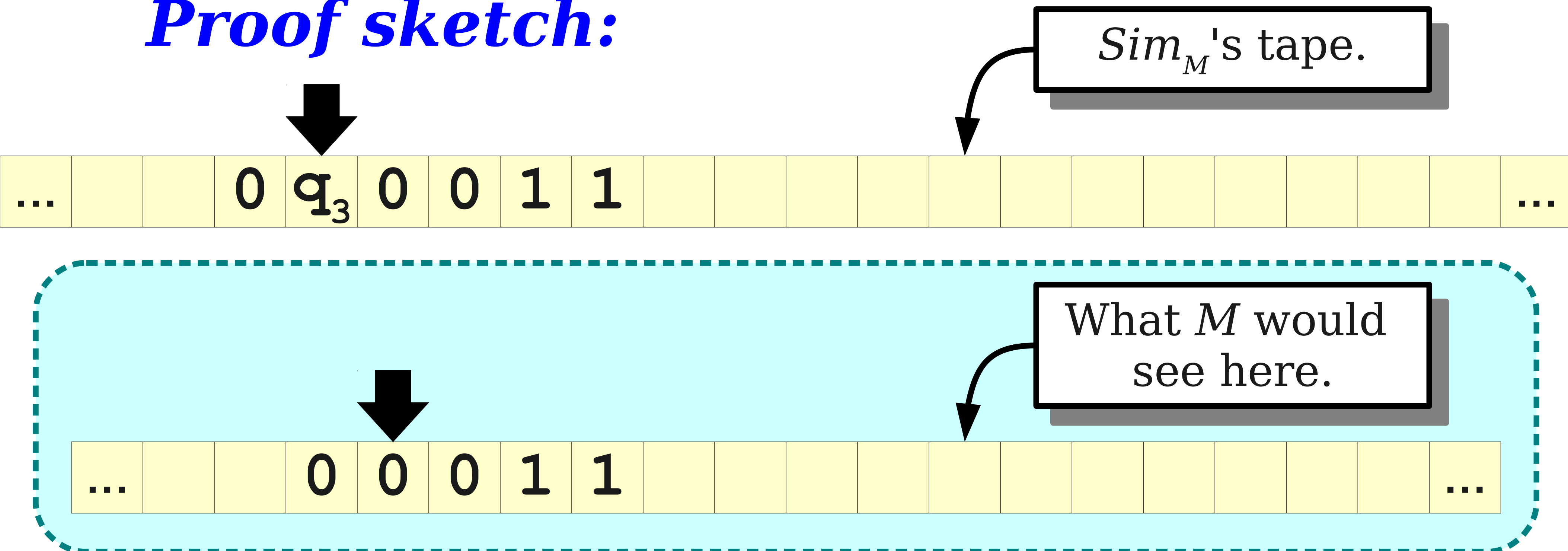


Sim_M can then go simulate writing a **0** and moving left.

Manipulating IDs

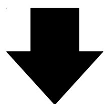
Theorem: For any TM M, it is possible to build a second TM Sim_M that, given a a tape containing an ID of M, simulates one step of M's computation.

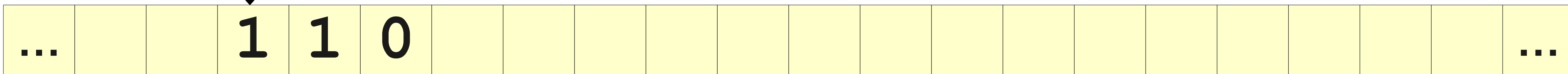
Proof sketch:

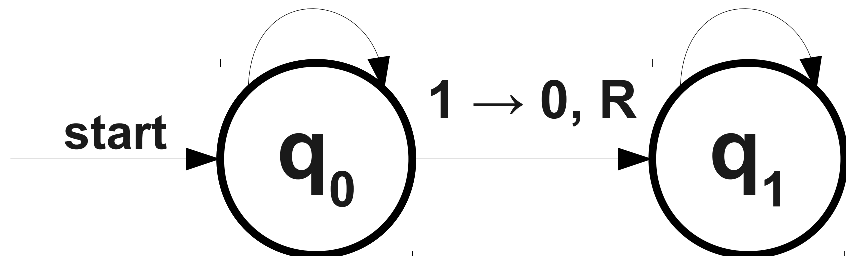


Putting Everything Together

- ***Theorem:*** For any NTM N, there exists a DTM D such that $\mathscr{L}(N) = \mathscr{L}(D)$.
- ***Proof sketch:*** D uses a worklist to exhaustively search over N's computation tree.







Schematically

Workspace	Worklist of IDs

To simulate the NTM N with a DTM D, we construct D as follows:

- On input w, D converts w into an initial ID for N starting on w.
- While D has not yet found an accepting state:
  - D finds the next ID for N from the worklist.
  - D copies this ID once for each possible transition.
  - D simulates one step of the computation for each of these IDs.
  - D copies these IDs to the back of the worklist.

Why All This Matters

- All of the TM design techniques we've seen have come into play here:
  - Building and manipulating lists.
  - Exhaustively searching over an infinite space.
  - Storing constant information in the finite-state control.
  - Simulating one TM with another.
- TMs can do a *huge* number of complex computations!

Just how powerful ***are*** Turing machines?

Effective Computation

- An **effective method of computation** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
- This is not a formal definition. Rather, it's a set of properties we want to have.

The **Church-Turing Thesis** states that

Every effective method of computation is either equivalent to or weaker than a Turing machine.

This is not a mathematical fact – it's a hypothesis about the nature of computation.

Regular Languages

CFLs

Problems Solvable by *Any Feasible Computing Machine*

All Languages

What problems are out here?

Regular Languages

CFLs

RE

All Languages

Time-Out For Announcements!

Friday Four Square!

Today at 4:15PM outside Gates

Your Questions

“Do you do research? If so, what kind of research do you do?”

"What are some of the famous open questions computer theorists are trying to prove (within the scope of the material that we have covered so far)?"

When converting an NTM to a DTM, how much more time will the DTM take relative to the NTM to determine whether to accept? (More on that later in CS103…)

If a problem can be solved by an NTM, can it be solved by a TM that makes *random* choices about which transitions to take rather than making *nondeterministic* choices about which transitions to take? (Take CS254!)

“What's the significance of the number 137? It's always cropping up in examples.”

$$\alpha = \frac{e^2}{4\pi\epsilon_0 \hbar c} \approx \frac{1}{137}$$

“I wanna introduce mathematical theory to martial arts. I think math is the language of science and engineering including martial arts. And I wanna make martial arts an formal official course just like computer science. Do you think it is possible?”

High-Level Descriptions

The Church-Turing Thesis

- The Church-Turing thesis states that all effective models of computation are equivalent to or weaker than a Turing machine.
- As as a result, we can start to be less precise with our TM descriptions.

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:

  Do something with x."

- Example:

  M = "On input x:

  Repeat the following:

  If $|x| \leq 1$, accept.

  If the first and last symbols of x aren't the same, reject.

  Remove the first and last characters of x."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Construct y, the reverse of x.

  If $x = y$, accept.

  Otherwise, reject."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  If x is a palindrome, accept.

  Otherwise, reject."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Check that x has the form $\mathtt{0}^n\mathtt{1}^m\mathtt{2}^p$.

  If not, reject.

  If $nm = p$, accept.

  Otherwise, reject."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Check that x has the form $\mathbf{1}^n\mathbf{+1}^m\overset{?}{=}\mathbf{1}^p$.

  If not, reject.

  If so, if $n + m = p$, accept.

  Otherwise, reject."

Formatted Input

- Many languages require the input to be in some particular format.
- We can encode this directly into our TMs:

M = "On input $\mathbf{1}^n\mathbf{+1}^m\overset{?}{\mathbf{=}}\mathbf{1}^p$:

If $n + m = p$, accept.

Otherwise, reject."

- Machines of this form implicitly reject any inputs that don't have the right format.

What's Allowed?

- Rule of thumb:

You can include *anything* in a high-level description, as long as you could write a computer program for it.

- A few exceptions: can't get input from the user, make purely random decisions, etc.
- Unsure about what you can do? Try building the TM explicitly, or ask the course staff!

Encodings

Computing over Objects

- Turing machines always compute over strings.
- We have seen examples of automata that can *essentially* compute over other objects:
  - Walking your Dog: Compute over paths.
  - Multiplication: Compute over numbers.
  - IDs: Compute over TM configurations.
- We have always said how we will encode objects:
  - e.g. $\{ \mathbf{1}^m\mathbf{\times 1}^n\mathbf{=1}^{mn} \mid m, n \in \mathbb{N} \}$

A Multitude of Encodings

- There can be many ways of encoding the same thing.
- Example: the natural number 13:
  - In unary: `1111111111111`
  - In binary: `1101`
  - In decimal: `13`
  - In hexadecimal: `D`
  - In Roman numerals: `XIII`
  - ...
- ***Claim:*** Turing machines are sufficiently powerful to transform any one of these representations into any other of these representations.

An Abstract Idea of Encodings

- For simplicity, from this point forward we will make the following assumption:

  For any finite, discrete object *O*, it is always possible to find some way of encoding *O* as a string.

- When working with Turing machines, it really doesn't matter *how* we do the encoding. A TM can convert any reasonable encoding scheme into any other encoding scheme.

Notation for Encodings

- For any object O, we will denote a string encoding of O by writing O in angle brackets: O is encoded as $\boldsymbol{\langle O \rangle}$.
- This makes it much easier to specify languages.
- Examples:

$$\{ \langle R \rangle \mid R \text{ is a regular expression that matches } \varepsilon \}$$

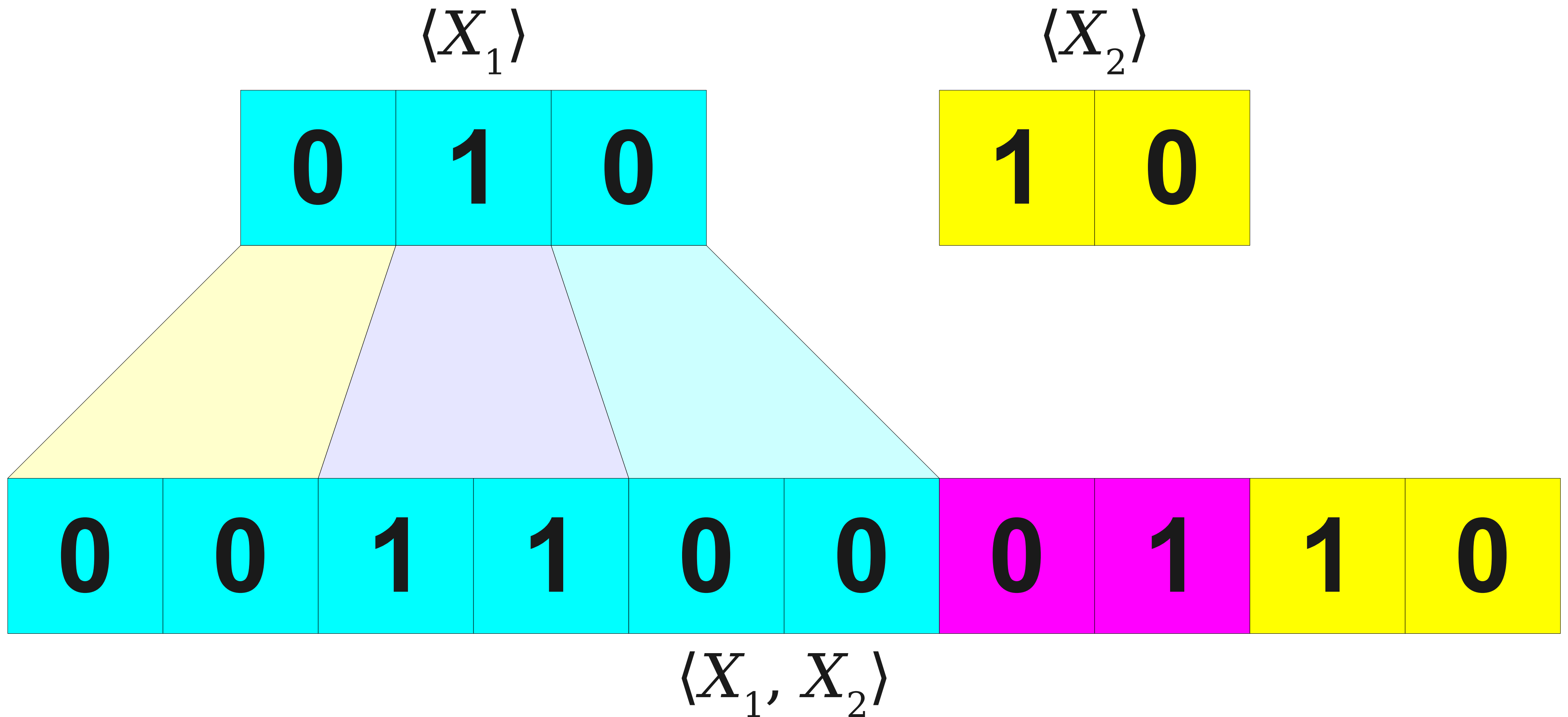
$$\{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n. \}$$

- The encoding scheme can make a difference when trying to determine whether a language is regular or context-free because of the relative weakness of DFAs and CFGs.

Encoding Multiple Objects

- Suppose that we want to provide an encoding of multiple objects.
  - Two natural numbers and their product.
  - A graph and a path in the graph.
  - “I just met you” and “this is crazy.”
- We can get encodings of each individual object.
- Can we make one string encoding all of these objects?

One Encoding Scheme



Encoding Multiple Objects

- Given several different objects $O_1, \ldots, O_n$, we can represent the encoding of those n objects as $\boldsymbol{\langle O_1, O_2, \ldots, O_n \rangle}$.
- Examples:
  - $\{ \langle m, n, mn \rangle \mid m, n \in \mathbb{N} \}$
  - $\{ \langle G, w \rangle \mid G$ is a context-free grammar that generates $w \}$

Next Time

- **The Universal Turing Machine**
  - One machine to run them all?
- **An Unsolvable Problem**
  - Finally... something we honestly can't solve!
- **More Unsolvable Problems**
  - What other languages are not **RE**?
- **Decidability**
  - How do we formalize the definition of an algorithm?
  - What problems can we learn the answer to?