Turing Machines

Part II

Problem Set Five due in the box up front using a late day.

Hello Condensed Slide Readers!

This lecture is almost entirely animations that show how each Turing machine would be built and how the machine works. I've tried to condense the slides here, but I think a lot got lost in the conversion.

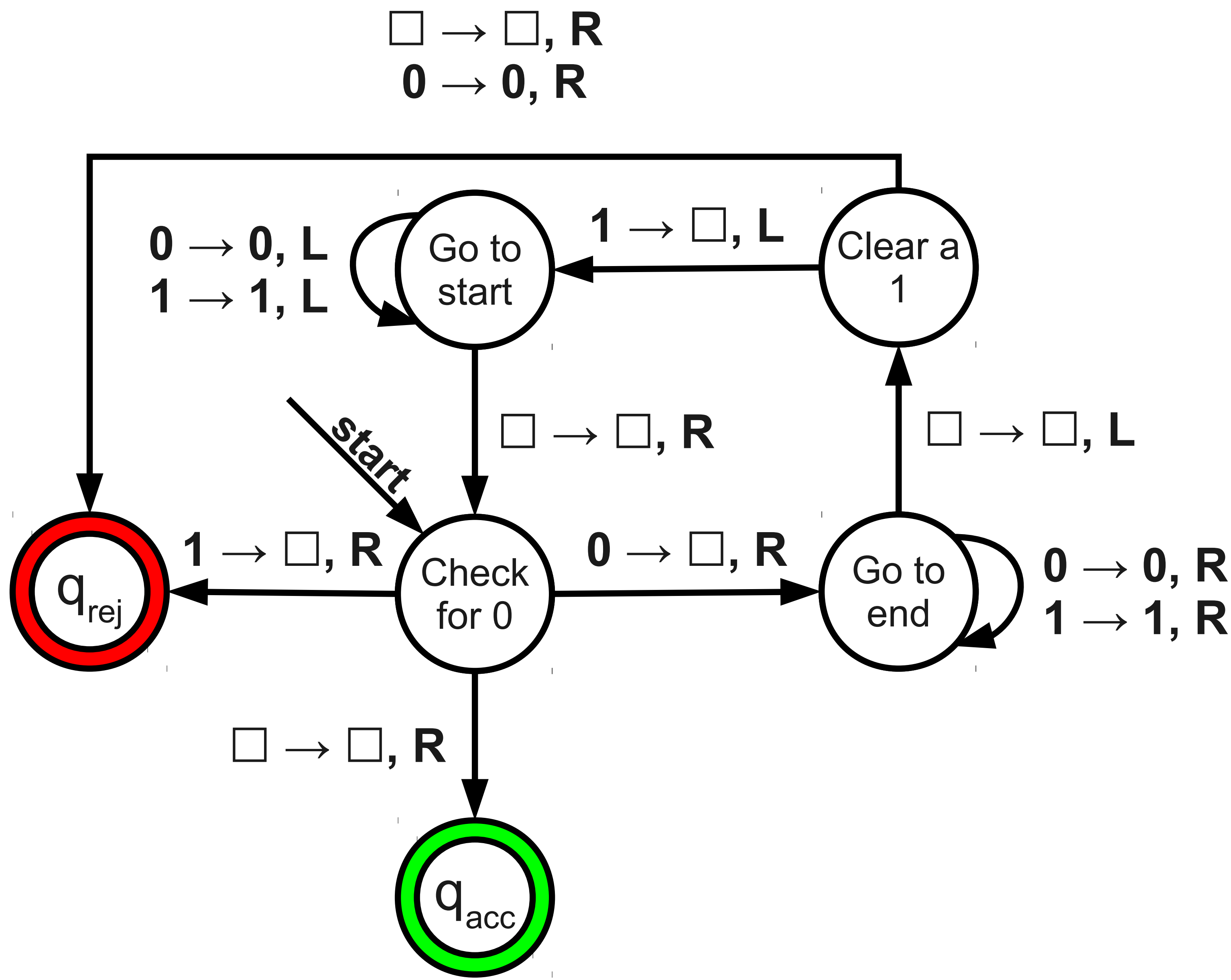
I would recommend reviewing the full slide deck to walk through each animation and see how the overall constructions work.

Hope this helps!

-Keith

The Turing Machine

- A Turing machine consists of three parts:
  - A **finite-state control** that issues commands,
  - an **infinite tape** for input and scratch space, and
  - a **tape head** that can read and write a single tape cell.
- At each step, the Turing machine
  - writes a symbol to the tape cell under the tape head,
  - changes state, and
  - moves the tape head to the left or to the right.



□ → □, R
0 → 0, R
0 → 0, L
1 → 1, L
Go to start
1 → □, L
Clear a 1
start
□ → □, R
□ → □, L
q_rej
1 → □, R
Check for 0
0 → □, R
Go to end
0 → 0, R
1 → 1, R
□ → □, R
q_acc

Key Idea: Subroutines

- A **subroutine** of a Turing machine is a small set of states in the TM such that performs a small computation.
- Usually, a single entry state and a single exit state.
- Many very complicated tasks can be performed by TMs by breaking those tasks into smaller subroutines.

Turing Machines and Math

- Turing machines are capable of performing
  - Addition
  - Subtraction
  - Multiplication
  - Integer division
  - Exponentiation
  - Integer logarithms
  - **Plus a whole lot more...**

Outline for Today

- **List Processing**
  - Turing machines that operate on sequences.
- **Exhaustive Search**
  - A fundamentally different approach to designing Turing machines.
- **Nondeterministic Turing Machines**
  - What does a Turing machine with Magic Superpowers look like?
- **The Church-Turing Thesis (ITA)**
  - Just how powerful *are* Turing machines?

List Processing

- Suppose we have a list of strings represented as

$$w_1 : w_2 : \ldots : w_n :$$

- What sorts of transformations can we perform on this list using a Turing machine?

Example: Take Odds

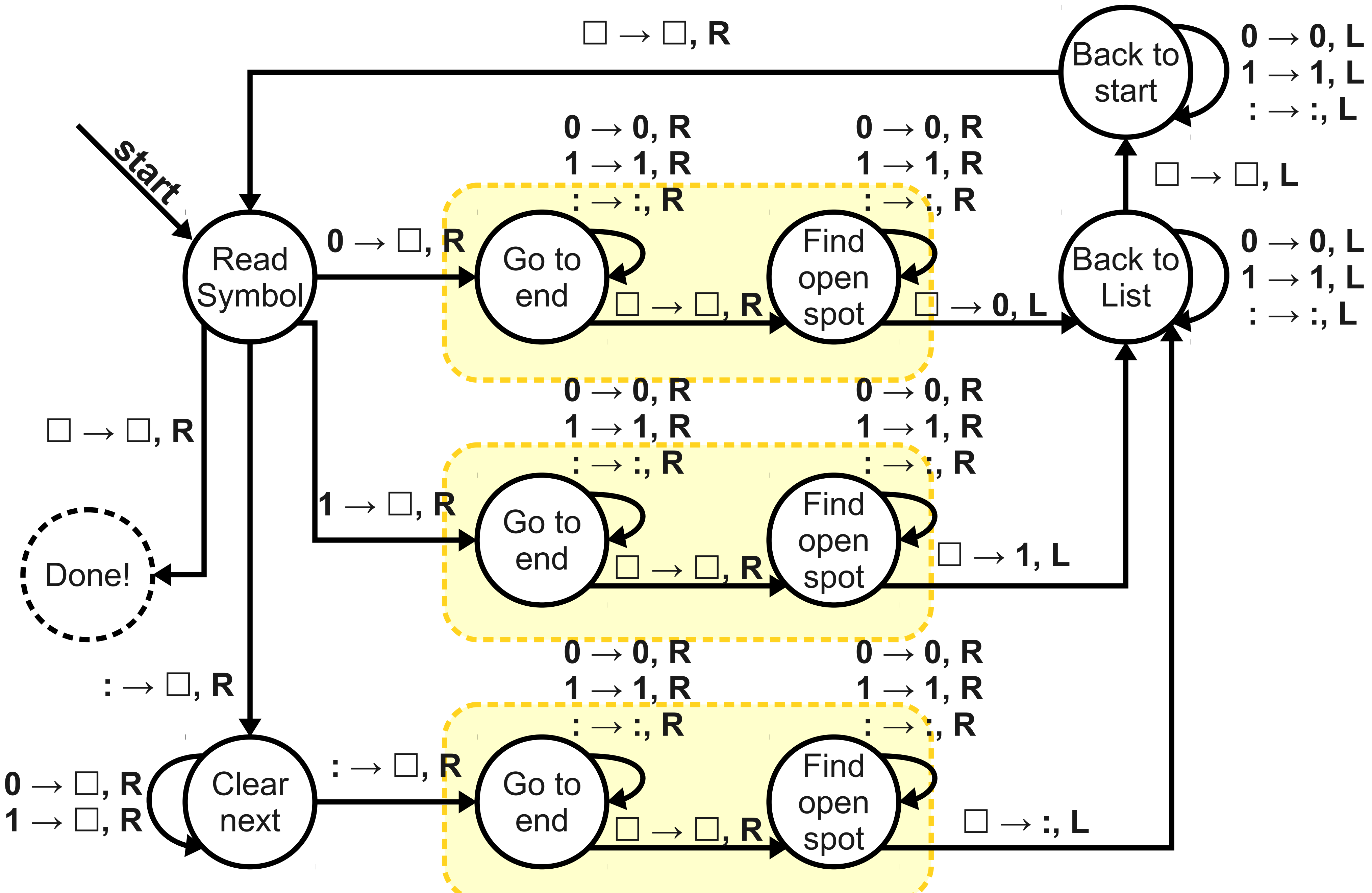
- Given a list of 2*n* strings encoded as follows:

$$w_1 : w_2 : \ldots : w_{2n} :$$

  filter the list to get back just the odd-numbered entries:

$$w_1 : w_3 : \ldots : w_{2n-1} :$$

- How might we do this with a Turing machine?



□ → □, R
Back to start
0 → 0, L
1 → 1, L
: → :, L
start
Read Symbol
0 → □, R
Go to end
0 → 0, R
1 → 1, R
: → :, R
□ → □, R
Find open spot
0 → 0, R
1 → 1, R
: → :, R
□ → 0, L
Back to List
0 → 0, L
1 → 1, L
: → :, L
□ → □, L
□ → □, R
Done!
1 → □, R
Go to end
0 → 0, R
1 → 1, R
: → :, R
□ → □, R
Find open spot
0 → 0, R
1 → 1, R
: → :, R
□ → 1, L
: → □, R
Clear next
0 → □, R
1 → □, R
: → □, R
Go to end
0 → 0, R
1 → 1, R
: → :, R
□ → □, R
Find open spot
0 → 0, R
1 → 1, R
: → :, R
□ → :, L

Turing Machine Memory

- Turing machines often contain many seemingly replicated states in order to store a finite amount of extra information.
- A Turing machine can remember one of k different constants by copying its states k times, once for each possible value, and wiring those states appropriately.

Turing Machines and Lists

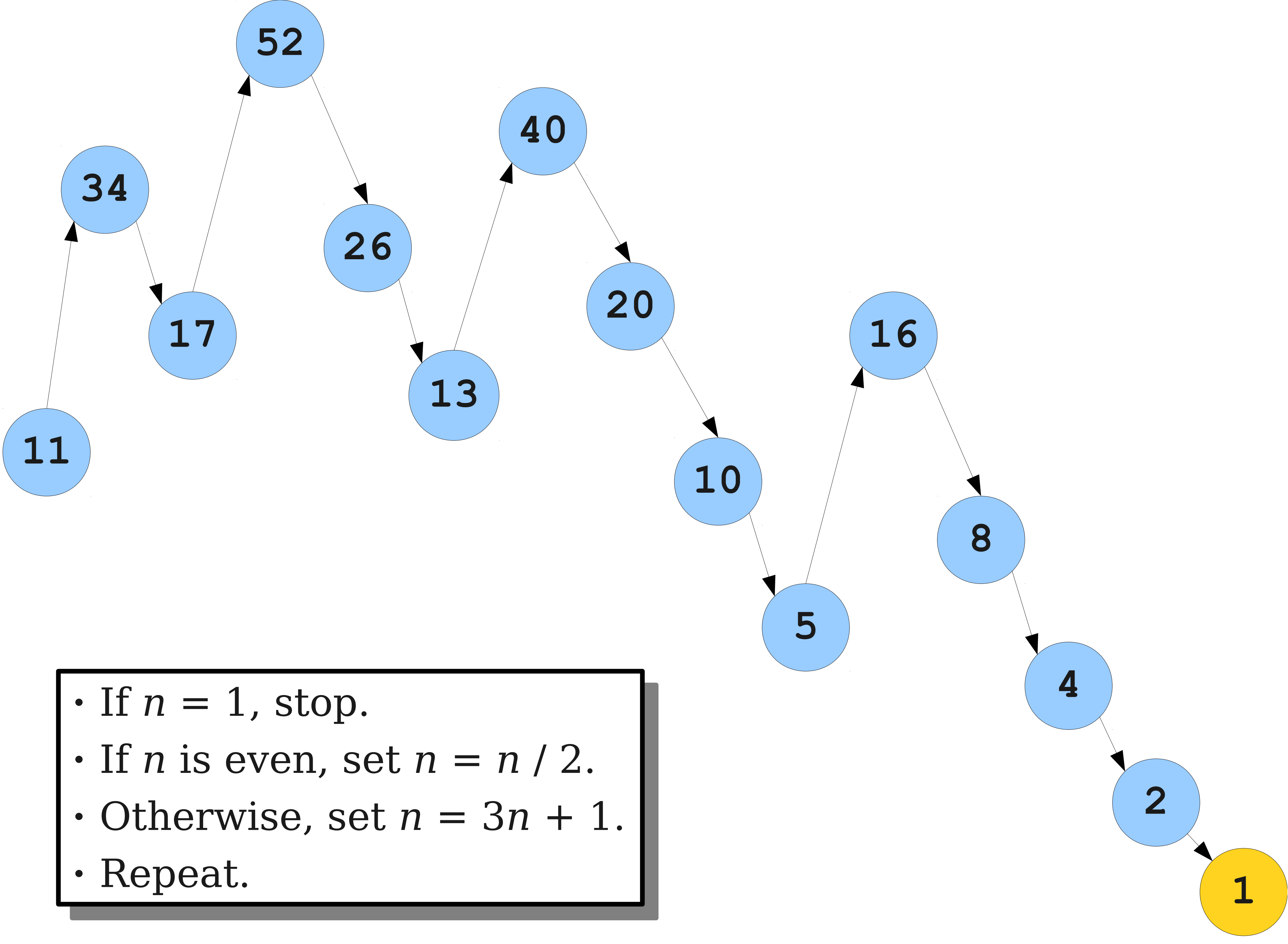
- Turing machines can perform many operations on lists:
  - Concatenate two lists.
  - Reverse a list.
  - Sort a list.
  - Find the maximum element of a list.
  - **And a whole lot more!**

The Power of Turing Machines

- Turing machines can
  - Perform standard arithmetic operations (addition, subtraction, multiplication, division, exponentiation, etc.)
  - Manipulate lists of elements (searching, sorting, reversing, etc.)
- What else can Turing machines do?

The Hailstone Sequence

- Consider the following procedure, starting with some $n \in \mathbb{N}$, where $n > 0$:
  - If $n = 1$, you are done.
  - If n is even, set $n = n / 2$.
  - Otherwise, set $n = 3n + 1$.
  - Repeat.
- Question: Given a number n, does this process terminate?



- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- Otherwise, set $n = 3n + 1$.
- Repeat.

The Hailstone Sequence

- Let $\Sigma = \{\mathbf{1}\}$ and consider the language

$$L = \{ \mathbf{1}^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}.$$

- Could we build a TM for L?

The Hailstone Turing Machine

- Intuitively, we can build a TM for the hailstone language as follows: the machine M does the following:
  - If the input is ε, reject.
  - While the input is not `1`:
    - If the input has even length, halve the length of the string.
    - If the input has odd length, triple the length of the string and append a `1`.
  - Accept.

Does this Turing machine always accept?

The Collatz Conjecture

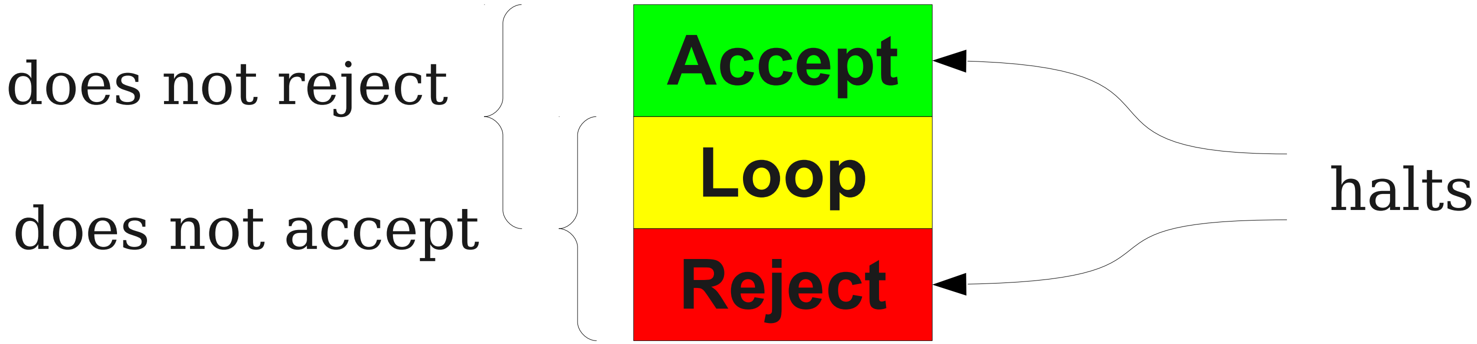
- It is *unknown* whether this process will terminate for all natural numbers.
- In other words, ***no one knows whether the TM described in the previous slides will always stop running!***
- The conjecture (claim) that this always terminates is called the **Collatz Conjecture**.

An Important Observation

- Unlike the other automata we've seen so far, Turing machines choose for themselves whether to accept or reject.
- It is therefore possible for a TM to run forever without accepting or rejecting.

Some Important Terminology

- Let M be a Turing machine.
- M **accepts** a string w if it enters the accept state when run on w.
- M **rejects** a string w if it enters the reject state when run on w.
- M **loops infinitely** (or just **loops**) on a string w if when run on w it enters neither the accept or reject state.
- M **does not accept w** if it either rejects w or loops infinitely on w.
- M **does not reject w** w if it either accepts w or loops on w.
- M **halts on w** if it accepts w or rejects w.



The Language of a TM

- The language of a Turing machine M, denoted $\mathscr{L}(M)$, is the set of all strings that M accepts:

$$\mathscr{L}(\boldsymbol{M}) = \{\ \boldsymbol{w} \in \boldsymbol{\Sigma}^* \mid \boldsymbol{M} \textbf{ accepts } \boldsymbol{w}\ \}$$

- For any $w \in \mathscr{L}(M)$, M accepts w.
- For any $w \notin \mathscr{L}(M)$, M does not accept w.
    - It might loop forever, or it might explicitly reject.
- A language is called **recognizable** iff it is the language of some TM.
- Notation: **RE** is the set of all recognizable languages.

$$L \in \textbf{RE} \quad \text{iff} \quad L \text{ is recognizable}$$

Time Out For Announcements!

Office Hours Schedule

- We've update our office hours schedule to shift office hours more toward Monday.
- Check the website for the updated schedule!

Your Questions!

“What is your favorite 103 topic and why?”

Stay tuned...
we're about to
get there!

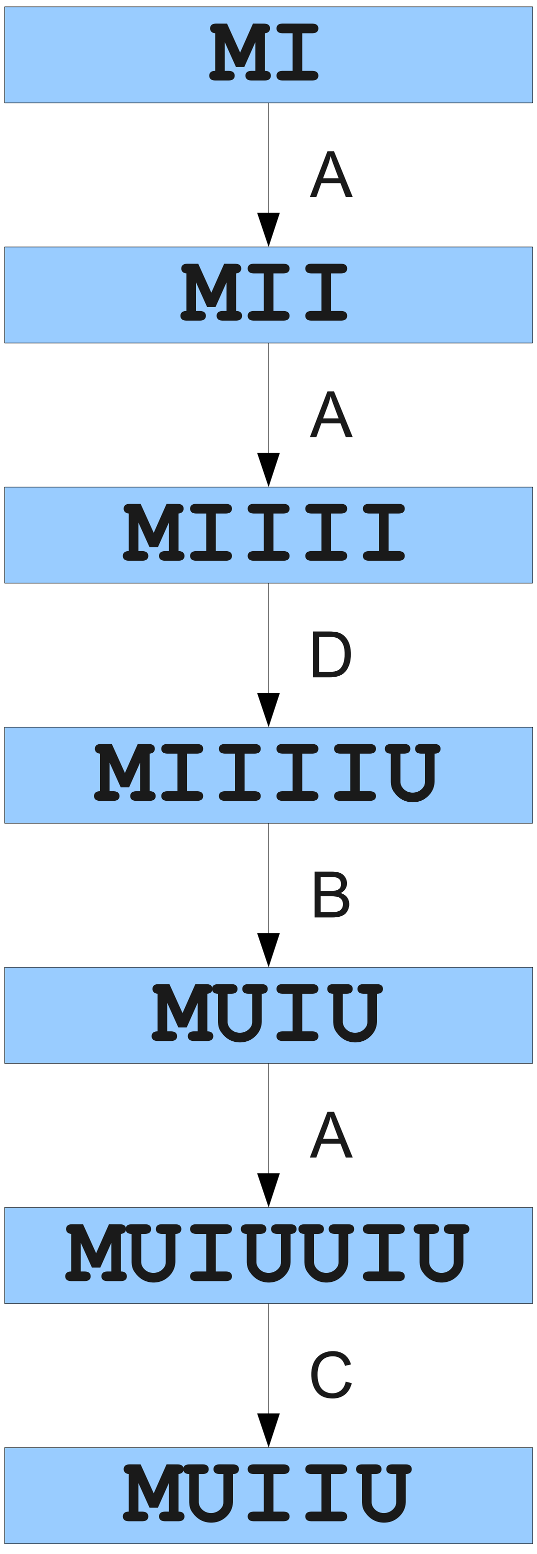
Worklist Algorithms

A) Double the contents of the string after **`M`**.

B) Replace **`III`** with **`U`**.

C) Remove **`UU`**

D) Append **`U`** if the string ends in **`I`**.



A Recognizable Language

- Let Σ = { **M**, **I**, **U** } and consider the language L = { $w \in \Sigma^*$ | Using the four provided rules, it is possible to convert w into **MU** }
- Some strings are in this language (for example, **MU** $\in L$, **MIII** $\in L$, **MUUU** $\in L$).
- Some strings are not in this language (for example, **I** $\notin L$ **MI** $\notin L$ **MIIU** $\notin L$).
- Could we build a Turing machine for L?

TM Design Trick: Worklists

- It is possible to design TMs that search over an infinite space using a worklist.
- Conceptually, the TM
  - Finds all possible options one step away from the original input,
  - Appends each of them to the end of the worklist,
  - Clears the current option, then
  - Grabs the next element from the worklist to process.
- **This Turing machine is not guaranteed to halt.**

The Power of TMs

- The worklist approach makes that all of the following languages are recognizable:
  - Any context-free language: simulate all possible production rules and see if the target string can be derived.
  - Solving a maze – use the worklist to explore all paths of length 0, 1, 2, … until a solution is found.
  - Determining whether a polynomial has an integer zeros: try 0, -1, +1, -2, +2, -3, +3, … until a result is found.

Searching and Guessing

Nondeterminism Revisited

- Recall: One intuition for nondeterminism is **perfect guessing**.
  - The machine has many options, and somehow magically knows which guess to make.
- With regular languages, we could generalize DFAs with NFAs.
- What happens if we do this for Turing machines?

Nondeterministic TMs

- A **nondeterministic Turing machine** (or **NTM**) is a variant on a Turing machine where there can be any number of transitions for a given state/tape symbol combination.
  - Notation: “Turing machine” or “TM” refers to a deterministic Turing machine unless specified otherwise. The term **DTM** specifically represents a deterministic TM.
- The NTM accepts iff there is *some possible series of choices* it can make such that it accepts.

Questions for Now

- How can we build an intuition for nondeterministic Turing machines?
- What sorts of problems can we solve with NTMs?
- What is the relative power of NTMs and DTMs?

Designing NTMs

- When designing NTMs, it is often useful to use the approach of guess and check:
  - **Nondeterministically** guess some object that can "prove" that $w \in L$.
  - **Deterministically** verify that you have guessed the right object.
- If $w \in L$, there will be some guess that causes the machine to accept.
- If $w \notin L$, then no guess will ever cause the machine to accept.

Composite Numbers

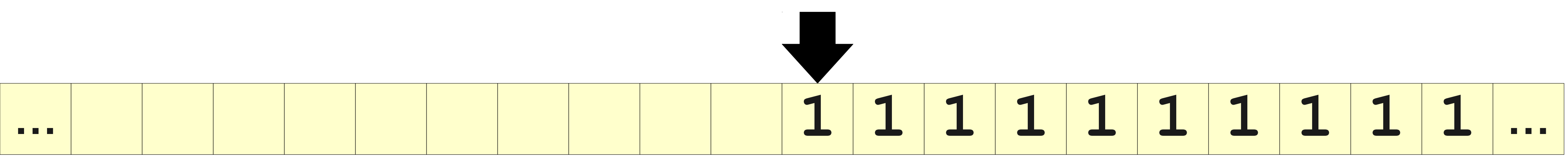
- A natural number $n \geq 2$ is called **composite** iff it has a factor other than 1 and n.
- Equivalently: there are two natural numbers $r \geq 2$ and $s \geq 2$ such that $rs = n$.
- Let $\Sigma = \{\mathtt{1}\}$ and consider the language

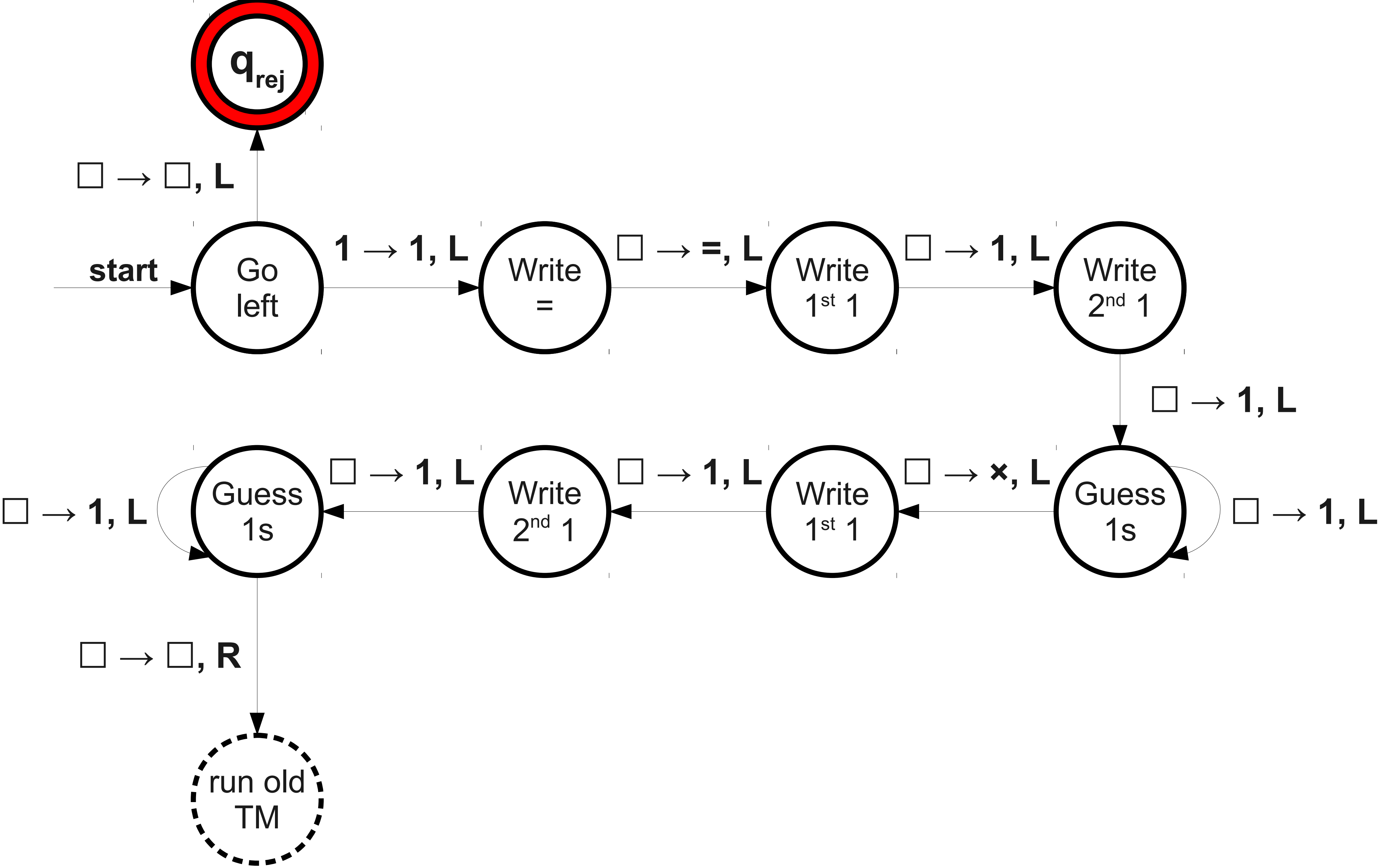
$$L = \{\ \mathtt{1}^n \mid n \text{ is composite }\}$$

- How might we design an NTM for L?

A Sketch of the NTM

- We saw how to build a TM that checks for correct multiplication.
- Have our NTM
  - **Nondeterministically** guess two factors, then
  - **Deterministically** run the multiplication TM.





q_rej
□ → □, L
start
Go left
1 → 1, L
Write =
□ → =, L
Write 1st 1
□ → 1, L
Write 2nd 1
□ → 1, L
Guess 1s
□ → 1, L
□ → ×, L
Write 1st 1
□ → 1, L
Write 2nd 1
□ → 1, L
Guess 1s
□ → 1, L
□ → □, R
run old TM

Nondeterminism and States

- When working with NFAs, we could think of the NFA as being in multiple states at the same time.
- **You cannot think of NTMs this way**.
- In NFAs, the only memory is the current state. In an NTM, memory includes the current state and the tape contents.
- If you're using the “massive parallelism” intuition, think about the machine cloning itself for all possible next steps, with each machine getting its own copy of the tape.

Designing NTMs

- Suppose that we have a CFG G.
- Can we build a TM M where $\mathscr{L}(M) = \mathscr{L}(G)$?
- **Idea:** Nondeterministically guess which productions ought to be applied.
  - Keep the original string on the input tape.
  - Keep guessing productions until no nonterminals remain.
  - Accept if the resulting string matches.

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?

Next Time

- **The Church-Turing Thesis**
  - Just how powerful *are* Turing machines?
- **Encodings**
  - How do we compute over arbitrary objects?
- **The Universal Turing Machine**
  - Can TMs compute over themselves?
- **The Limits of Turing Machines (ITA)**
  - A language not in **RE**.