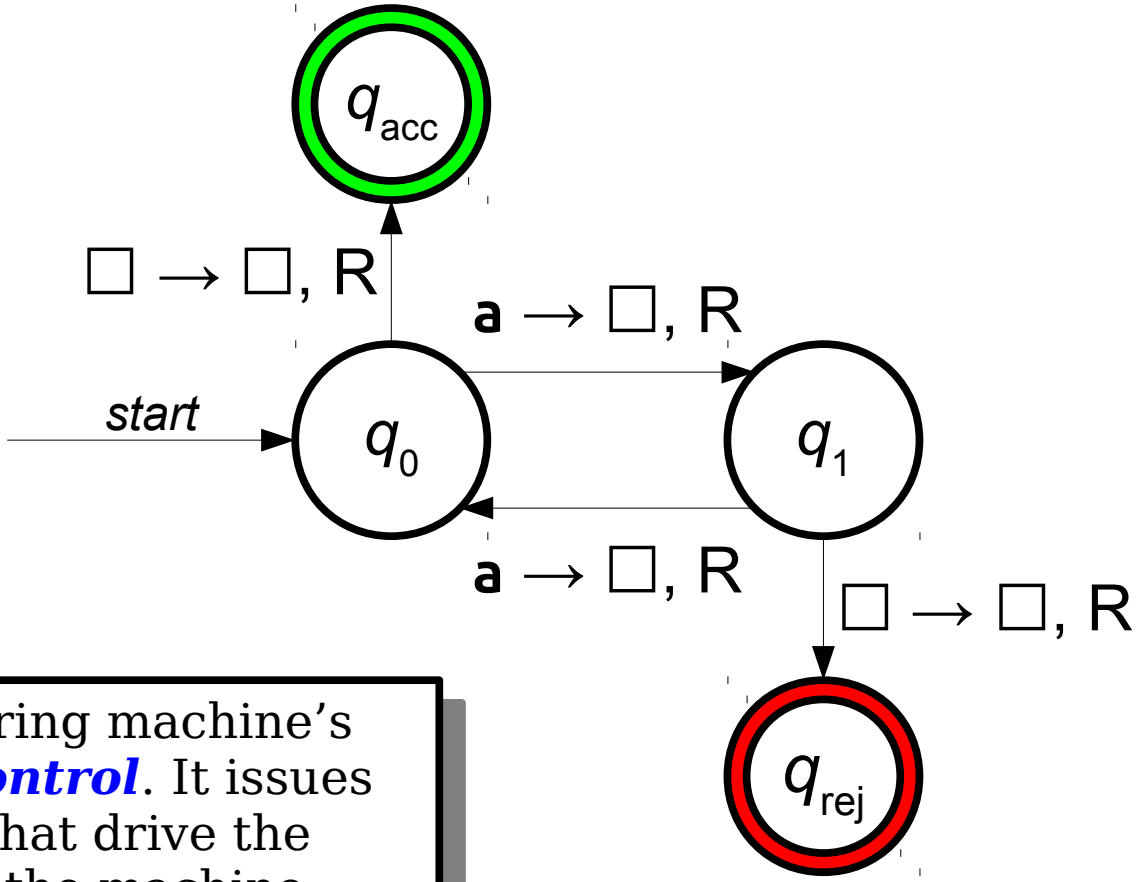
Turing Machines

Part Two

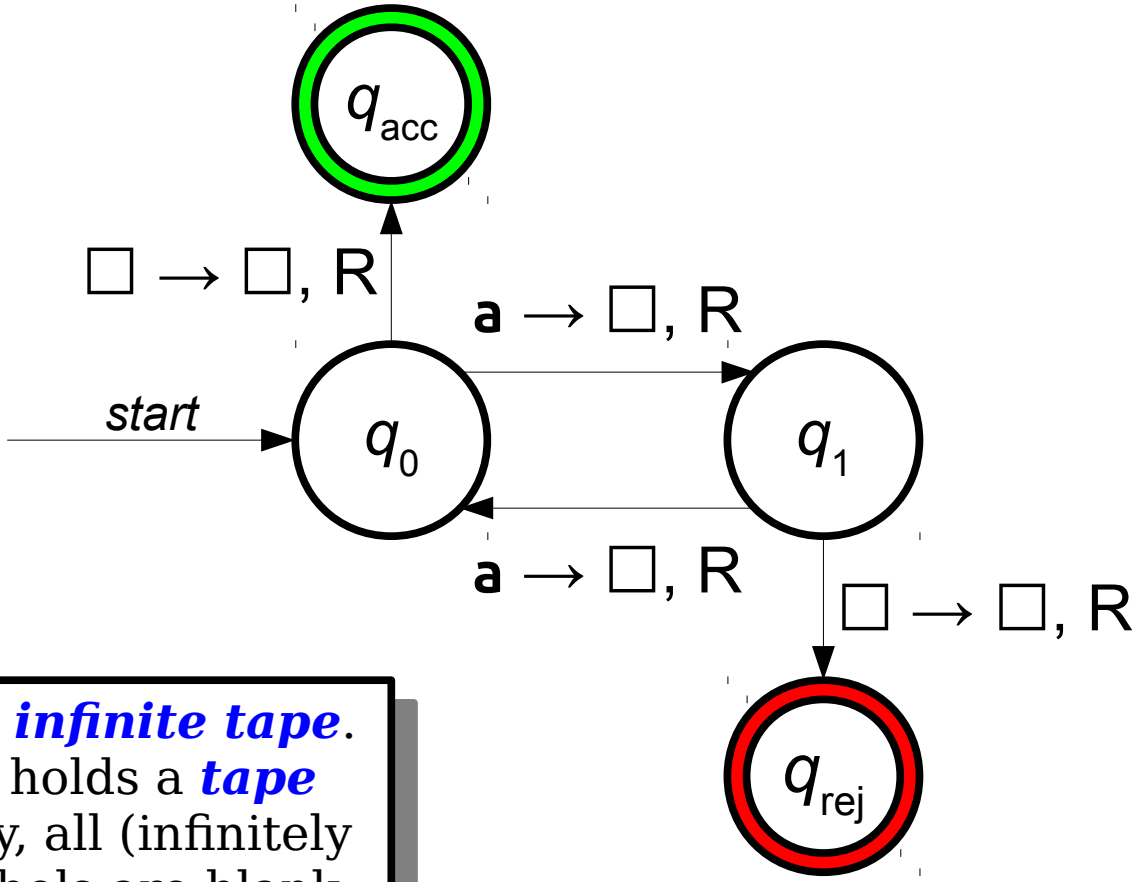
Recap from Last Time

Our First Turing Machine

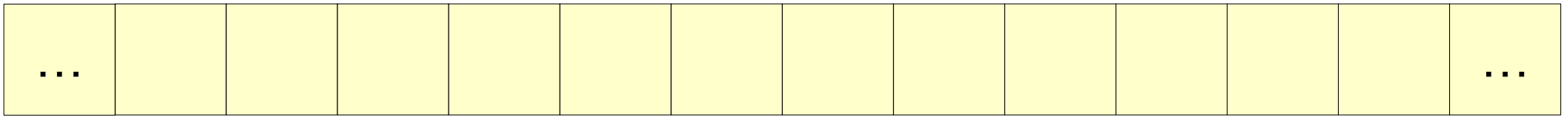


This is the Turing machine's ***finite state control***. It issues commands that drive the operation of the machine.

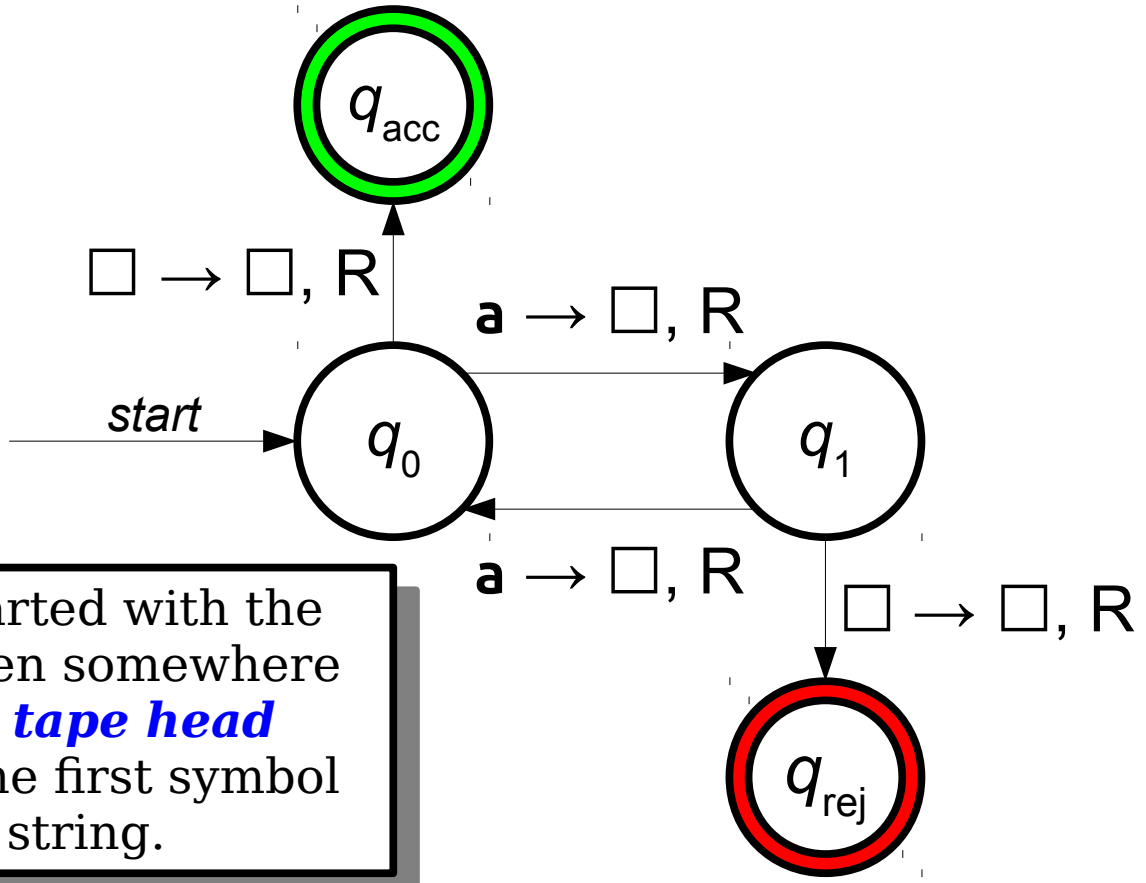
Our First Turing Machine



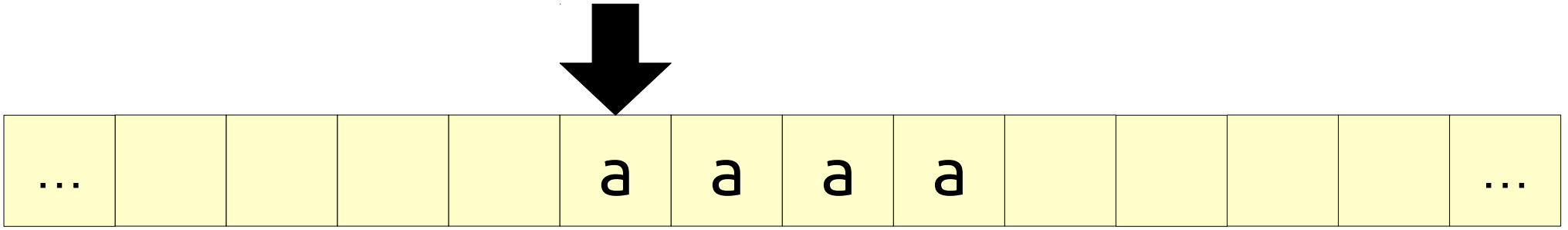
This is the TM's ***infinite tape***. Each tape cell holds a ***tape symbol***. Initially, all (infinitely many) tape symbols are blank.



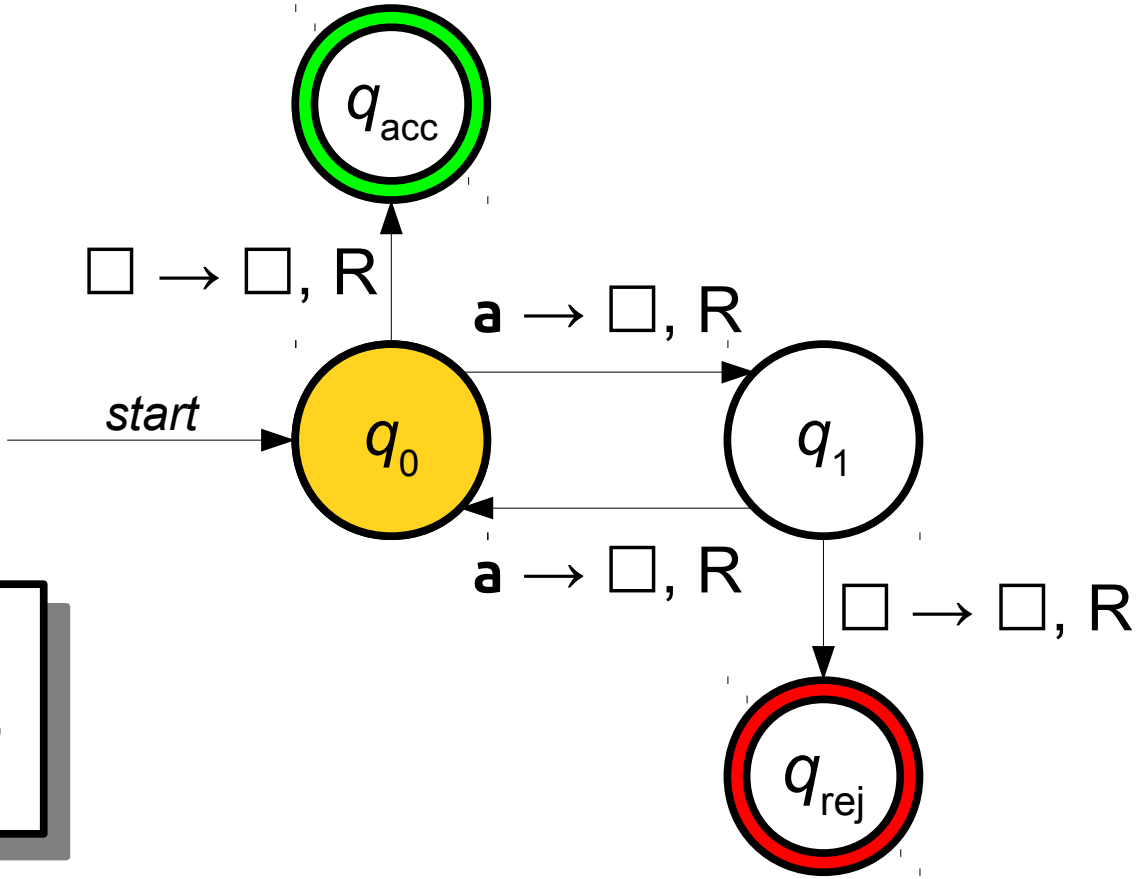
Our First Turing Machine



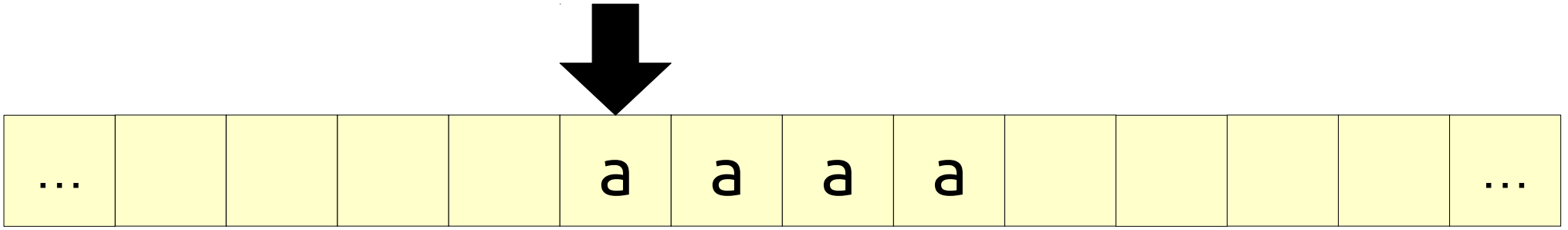
The machine is started with the ***input string*** written somewhere on the tape. The ***tape head*** initially points to the first symbol of the input string.



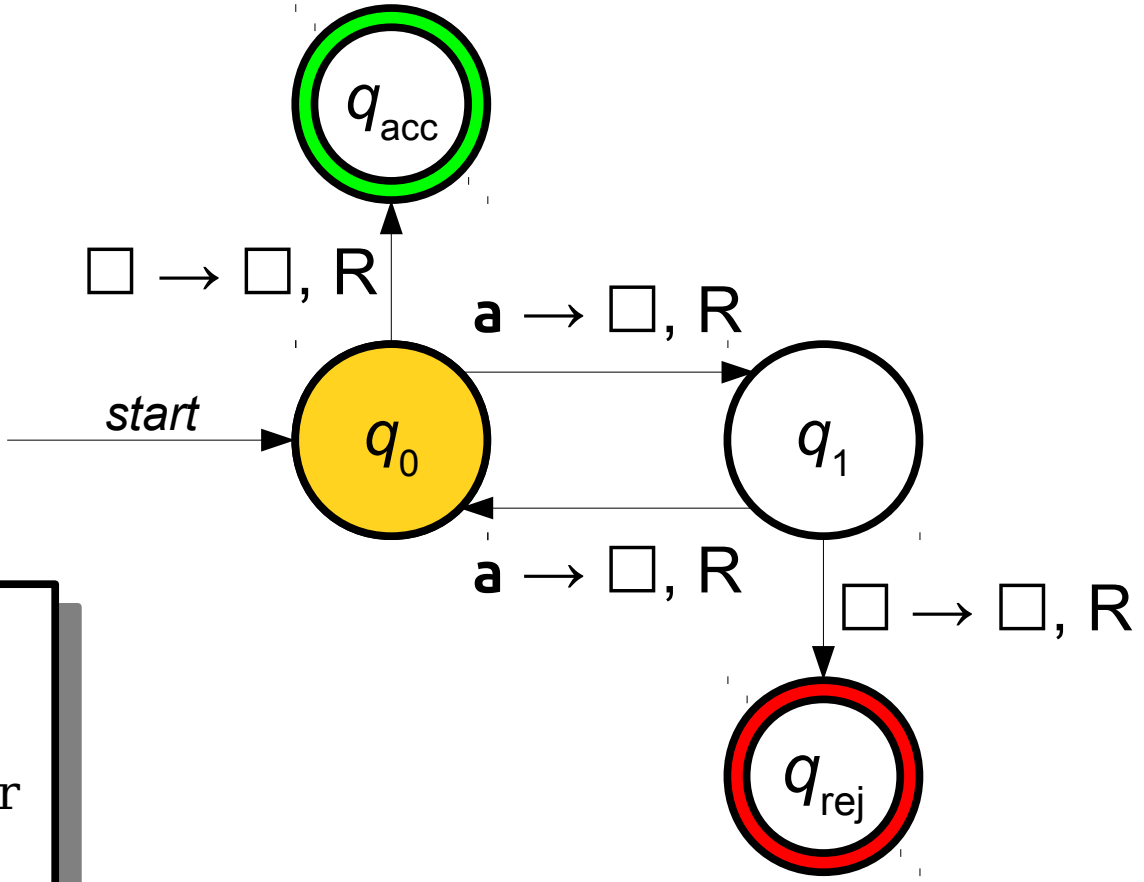
Our First Turing Machine



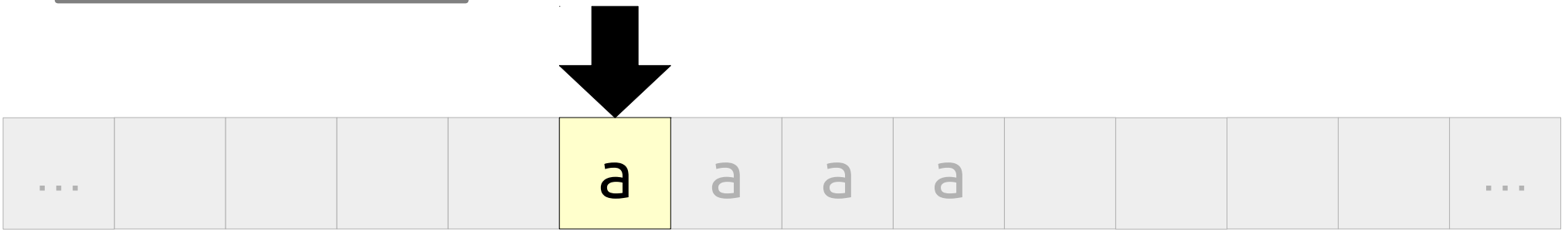
Like DFAs and NFAs, TMs begin execution in their ***start state***.



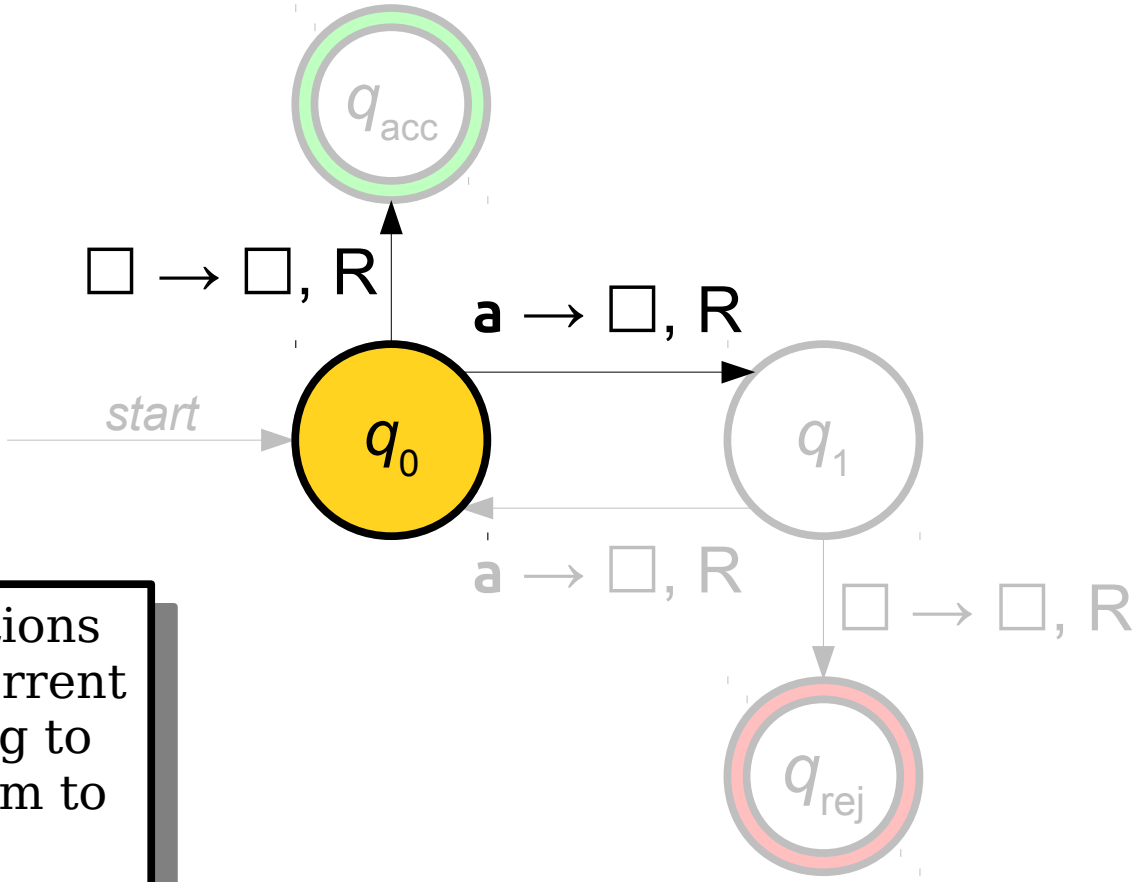
Our First Turing Machine



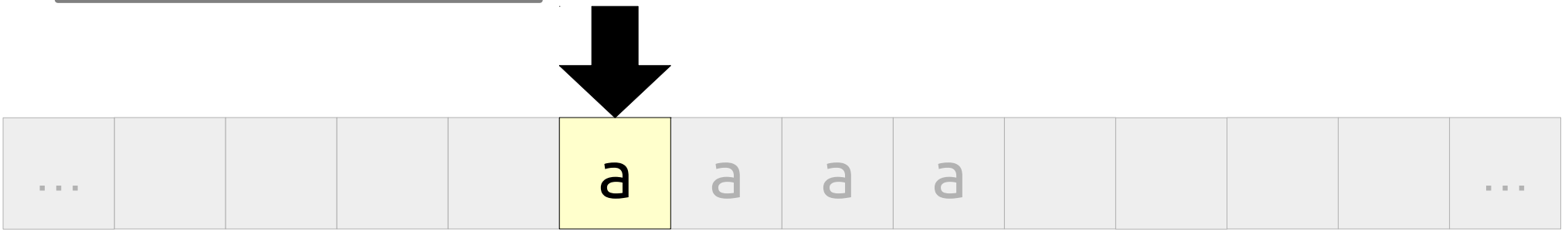
At each step, the TM only looks at the symbol immediately under the ***tape head***.



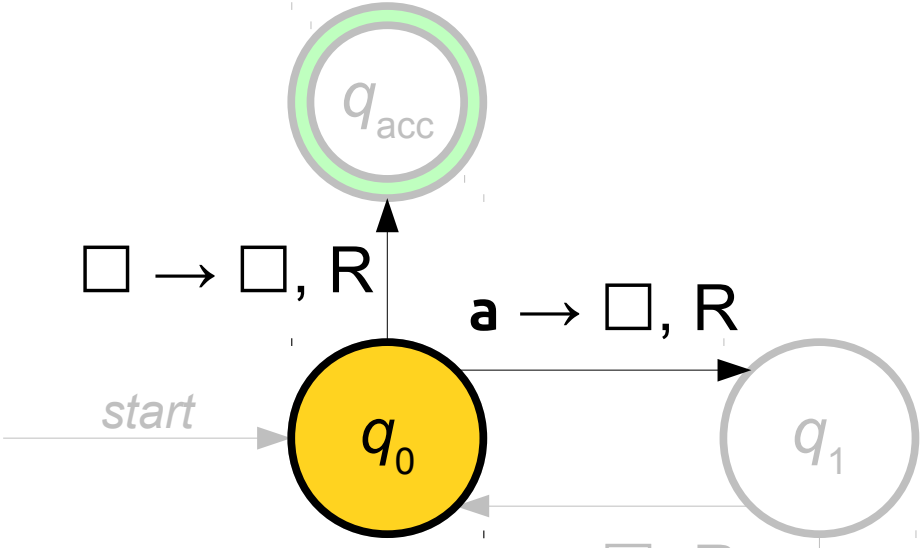
Our First Turing Machine



These two transitions originate at the current state. We're going to choose one of them to follow.



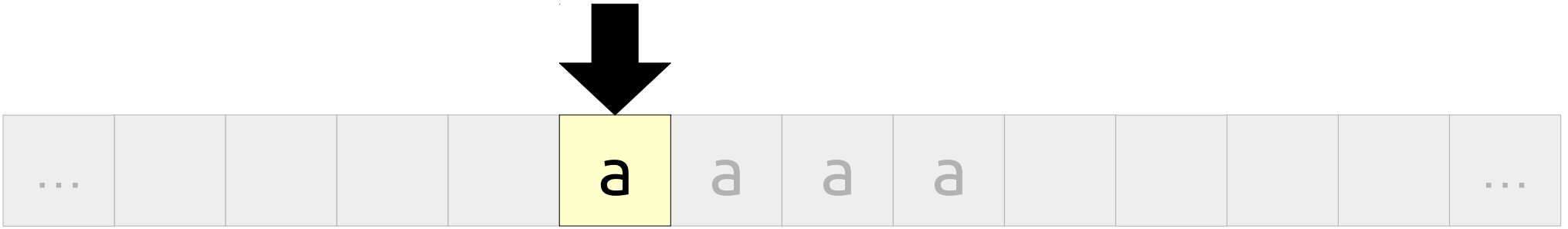
Our First Turing Machine



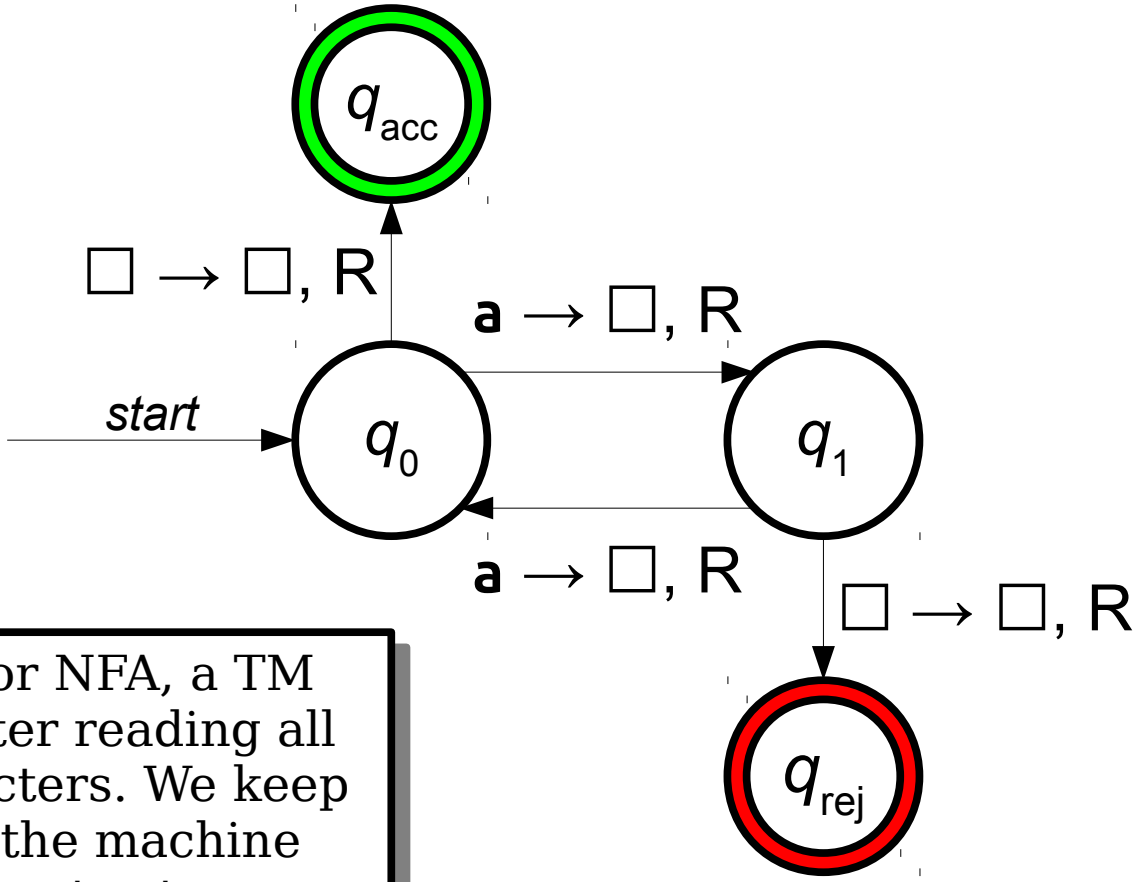
Each transition has the form

read → write, dir

and means "if symbol ***read*** is under the tape head, replace it with ***write*** and move the tape head in direction ***dir*** (L or R). The □ symbol denotes a blank cell.

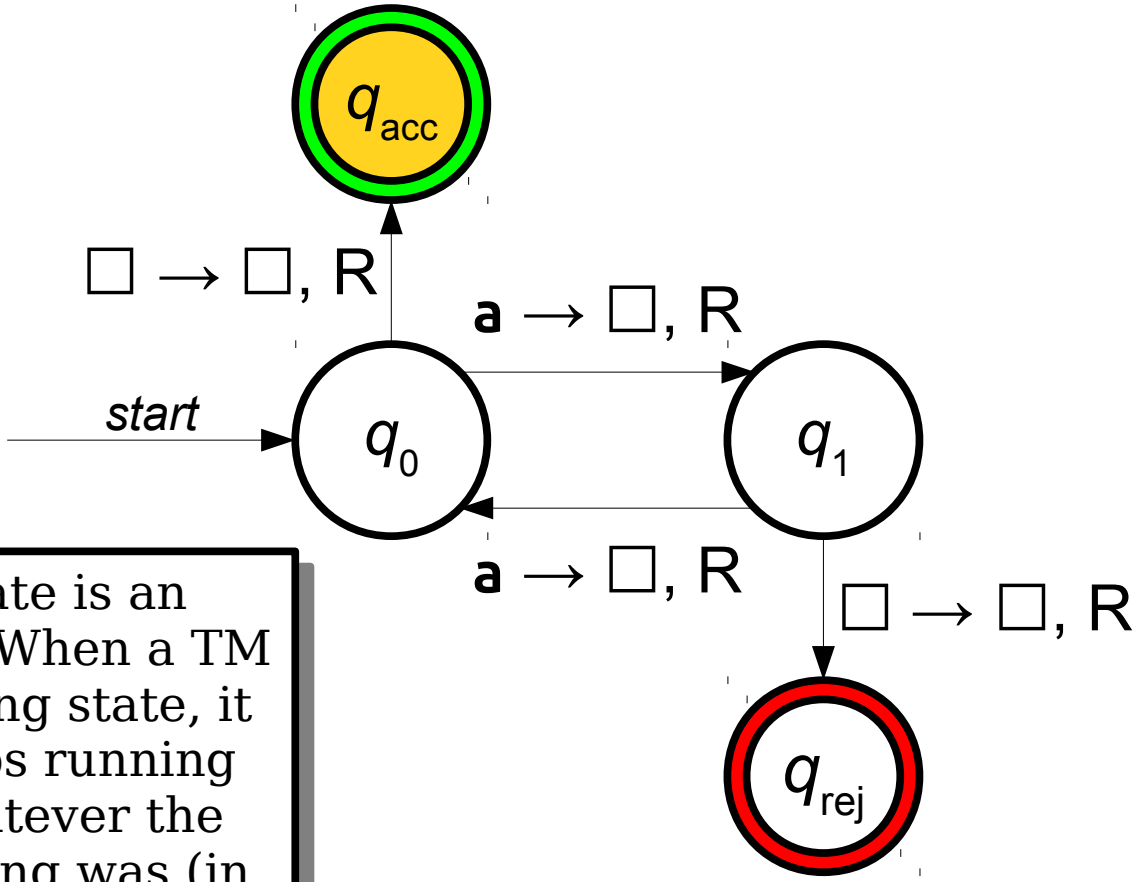


Our First Turing Machine



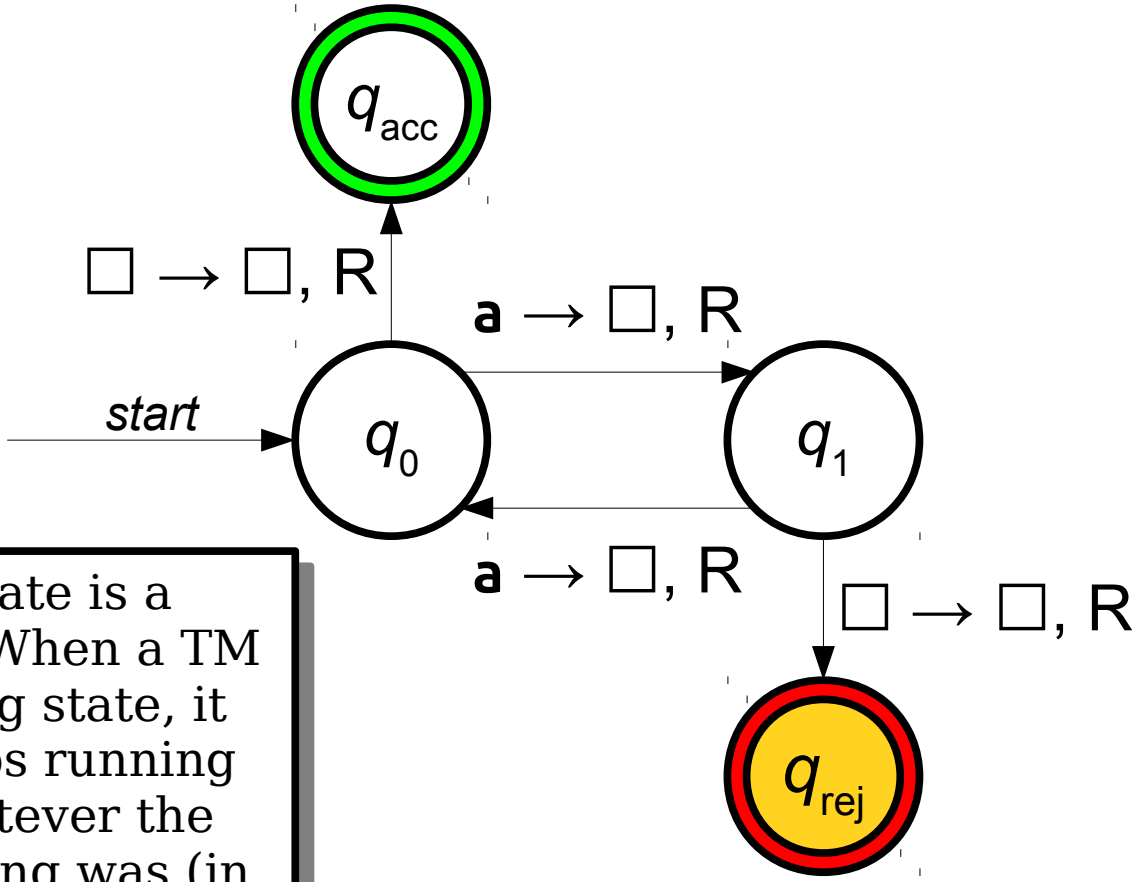
Unlike a DFA or NFA, a TM doesn't stop after reading all the input characters. We keep running until the machine explicitly says to stop.

Our First Turing Machine



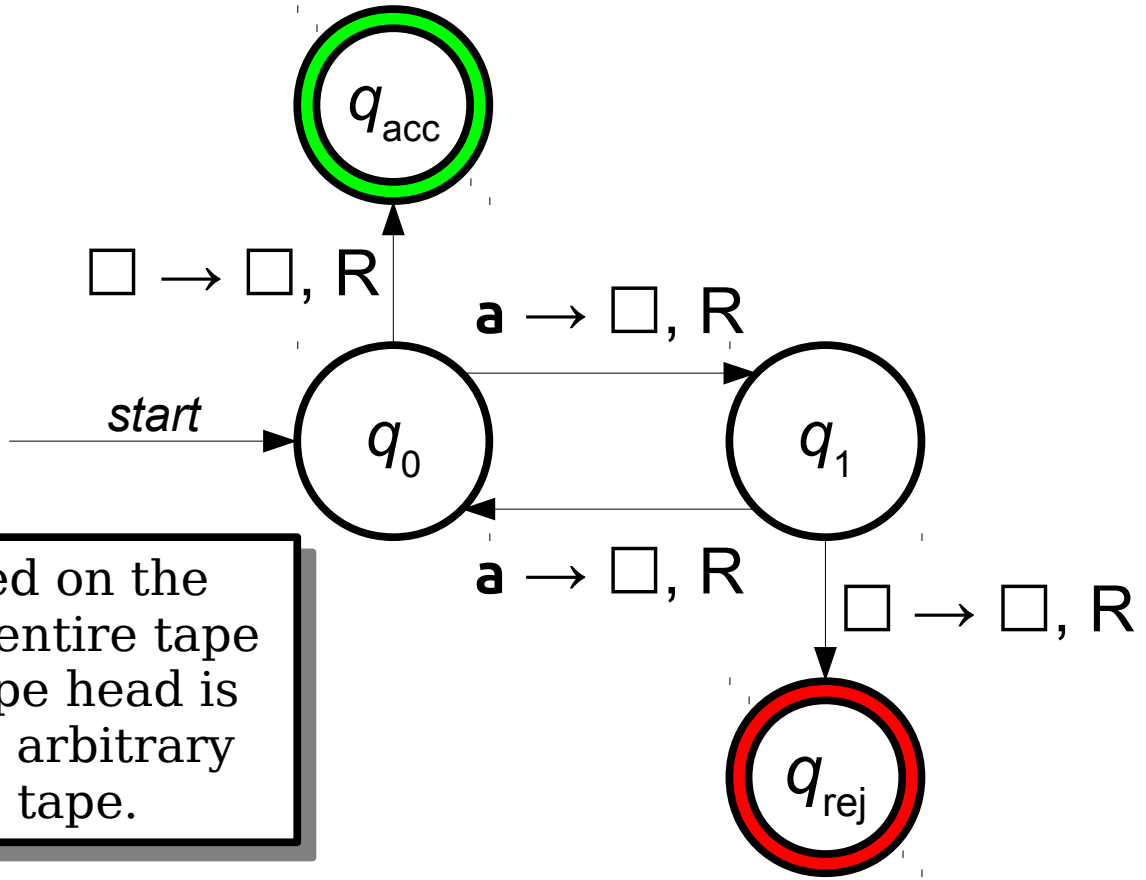
This special state is an ***accepting state***. When a TM enters an accepting state, it *immediately* stops running and accepts whatever the original input string was (in this case, **aaaa**).

Our First Turing Machine

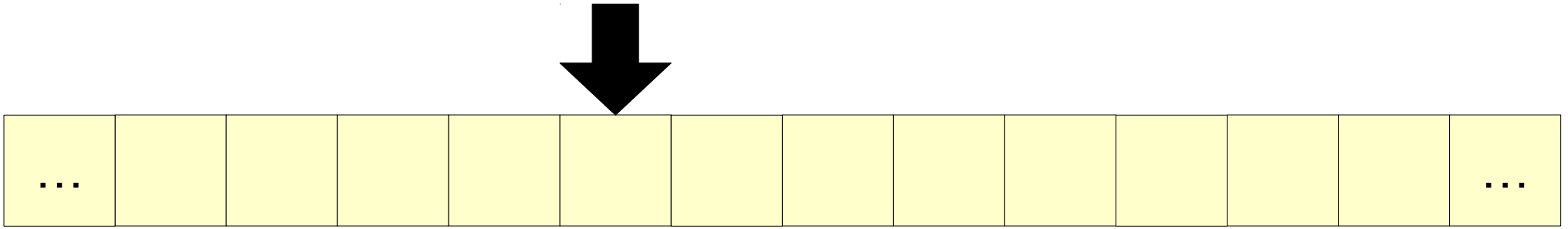


This special state is a ***rejecting state***. When a TM enters a rejecting state, it *immediately* stops running and rejects whatever the original input string was (in this case, **aaaaa**).

Our First Turing Machine

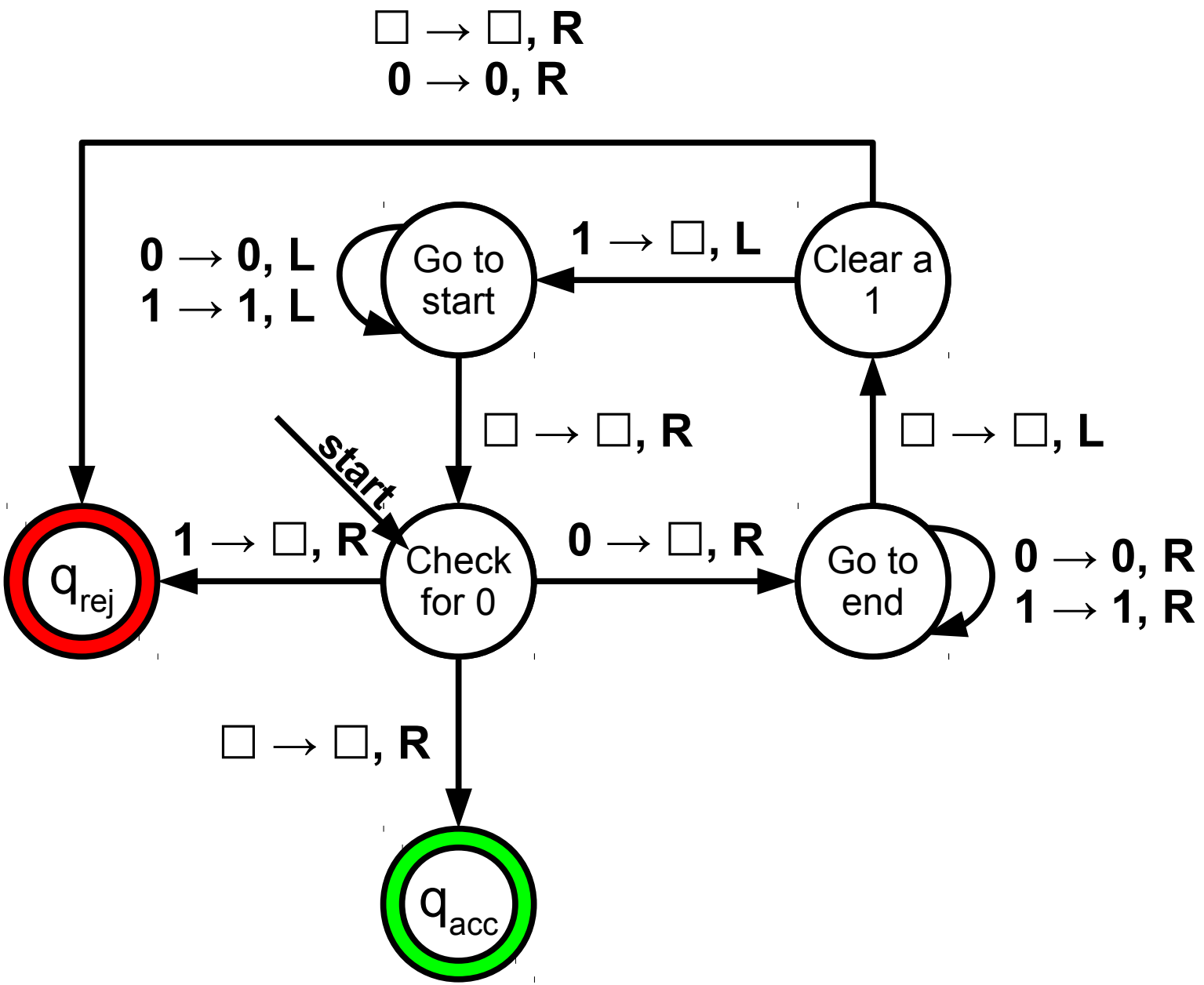


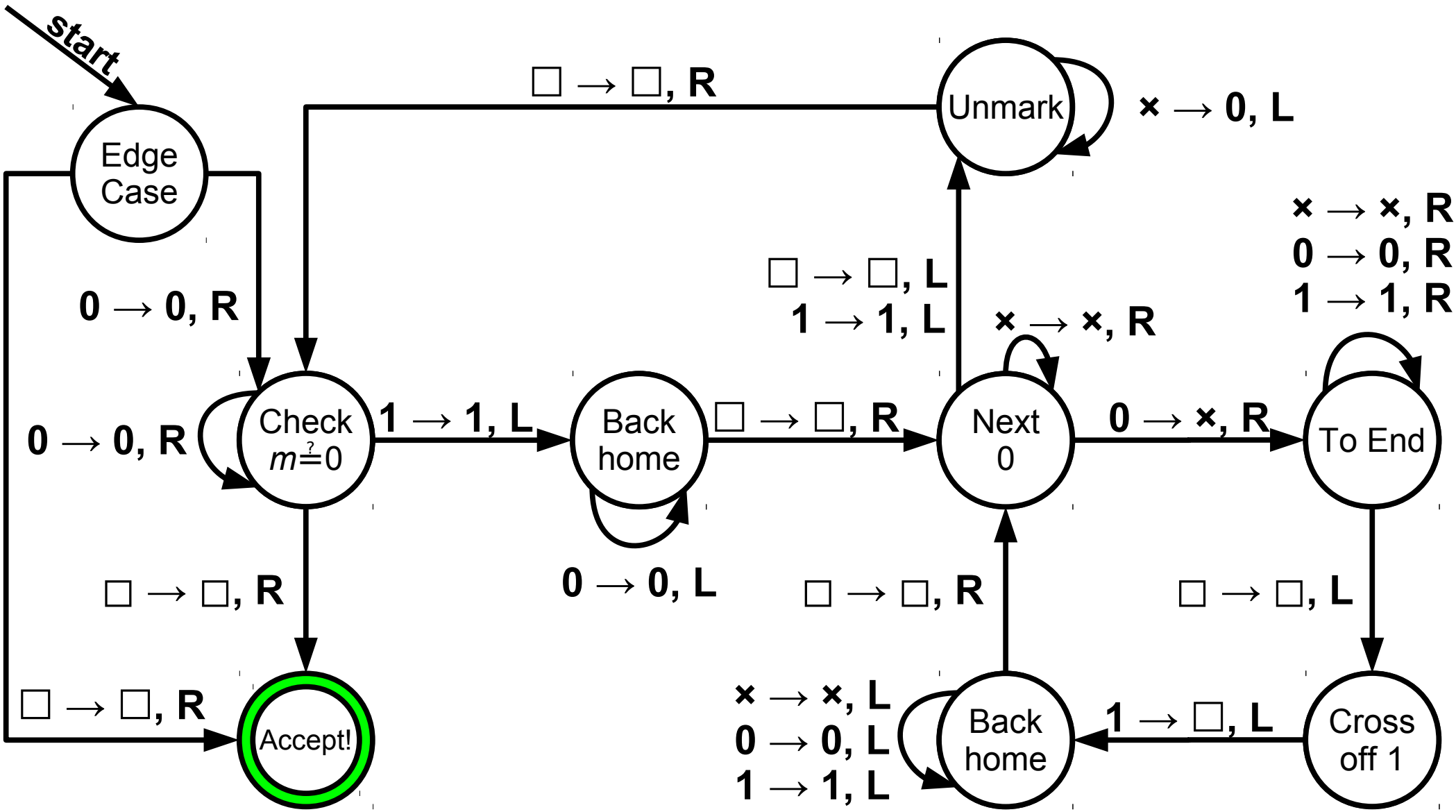
If the TM is started on the empty string ε, the entire tape is blank and the tape head is positioned at some arbitrary location on the tape.



Input and Tape Alphabets

- A Turing machine has two alphabets:
  - An ***input alphabet*** Σ. All input strings are written in the input alphabet.
  - A ***tape alphabet*** Γ, where Σ ⊆ Γ. The tape alphabet contains all symbols that can be written onto the tape.
- The tape alphabet Γ can contain any number of symbols, but always contains at least one ***blank symbol***, denoted □. You are guaranteed □ ∉ Σ.
- At startup, the Turing machine begins with an infinite tape of □ symbols with the input written at some location. The tape head is positioned at the start of the input.


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


start
Edge Case
Check $m \stackrel{?}{=} 0$
Back home
Unmark
Next 0
To End
Back home
Cross off 1
Accept!
□ → □, R
0 → 0, R
1 → 1, L
□ → □, L
1 → 1, L
× → 0, L
× → ×, R
0 → 0, R
1 → 1, R
0 → ×, R
0 → 0, L
× → ×, L
0 → 0, L
1 → 1, L
1 → □, L

New Stuff!

Main Question for Today:

Just how powerful are Turing machines?

Another TM Design

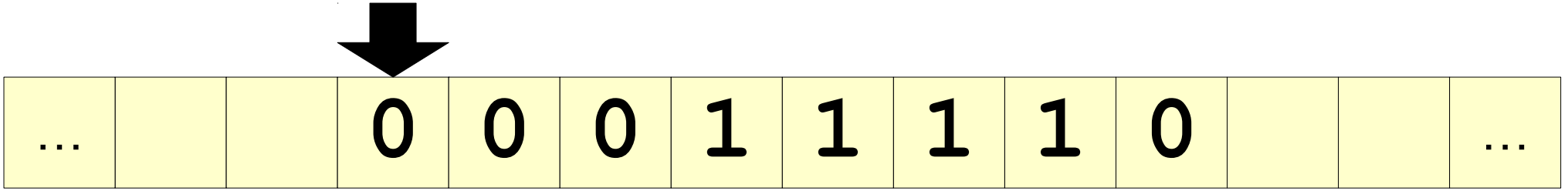
- Last time, we designed a TM for this language over $\Sigma = \{$**0**, **1**$\}$:

  $L = \{ w \in \Sigma^* \mid w$ has the same number of **0**s and **1**s $\}$

- Let's do a quick review of how it worked.

A Different Idea

A Different Strategy

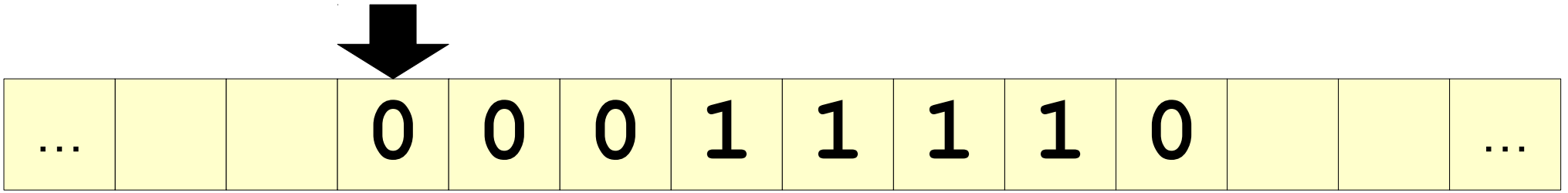


Last time, we built a machine that checks whether a string has the form 0^n1^n.

That machine <u>almost</u> solves this problem, but requires that the characters have to be in order.

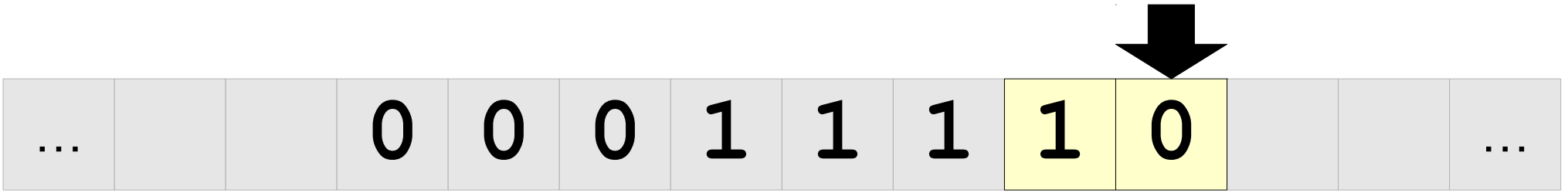
What if we sorted the input?

A Different Strategy



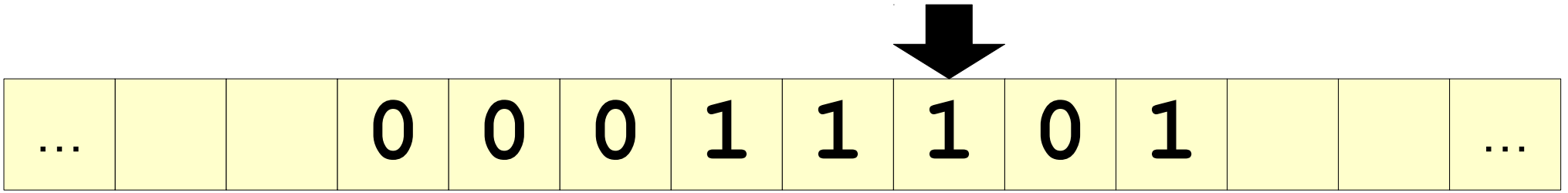
Observation 1: A string of 0s and 1s is sorted if it matches the regex 0*1*.

A Different Strategy



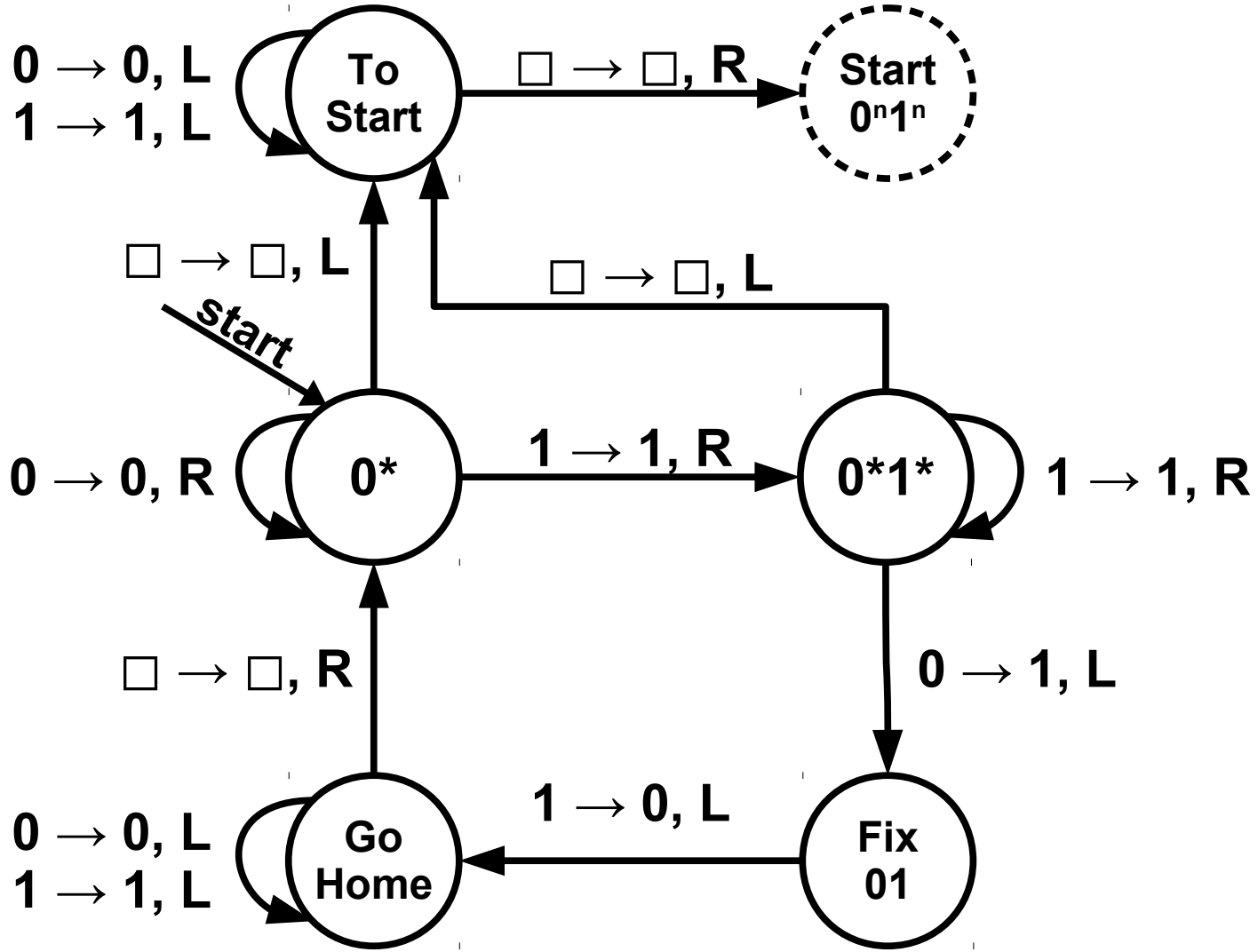
Observation 2: A string of 0s and 1s is not sorted if it contains 10 as a substring.

A Different Strategy



Idea: Repeatedly find a copy of 10 and replace it with 01.

Let's Build It!


0 → 0, L
1 → 1, L
To Start
□ → □, R
Start
0n1n
□ → □, L
□ → □, L
start
0 → 0, R
0*
1 → 1, R
0*1*
1 → 1, R
□ → □, R
0 → 1, L
1 → 0, L
0 → 0, L
1 → 1, L
Go Home
Fix 01

TM Subroutines

- A ***TM subroutine*** is a Turing machine that, instead of accepting or rejecting an input, does some sort of processing job.
- TM subroutines let us compose larger TMs out of smaller TMs, just as you'd write a larger program using lots of smaller helper functions.
- Here, we saw a TM subroutine that sorts a sequence of 0s and 1s into ascending order.

TM Subroutines

- Typically, when a subroutine is done running, you have it enter a state marked “done” with a dashed line around it.
- When we're composing multiple subroutines together – which we'll do in a bit – the idea is that we'll snap in some real state for the “done” state.

What other subroutines can we make?

TM Arithmetic

- Let's design a TM that, given a tape that looks like this:

…				1	3	7		4	2			…

ends up having the tape look like this:

…				1	7	9		0	0			…

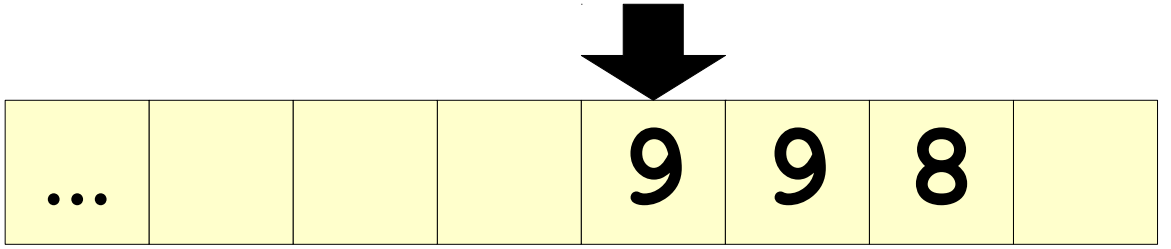
- In other words, we want to build a TM that can add two numbers.

TM Arithmetic

- There are many ways we could in principle design this TM.
- We're going to take the following approach:
  - First, we'll build a TM that increments a number.
  - Next, we'll build a TM that decrements a number.
  - Then, we'll combine them together, repeatedly decrementing the second number and adding one to the first number.

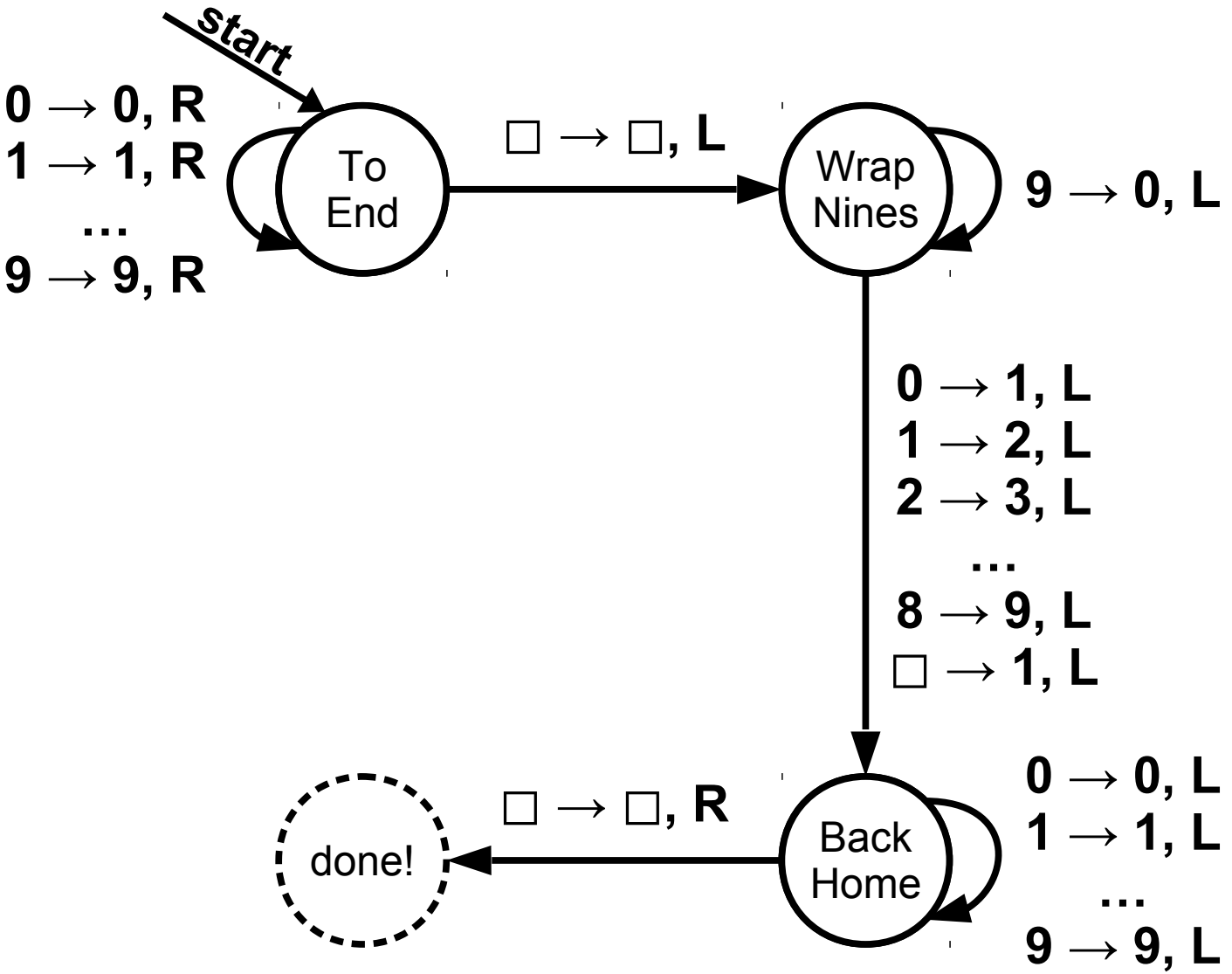
Incrementing Numbers

- Let's begin by building a TM that increments a number.
- We'll assume that
  - the tape head points at the start of a number,
  - there is are at least two blanks to the left of the number, and
  - that there's at least one blank at the start of the number.
- The tape head will end at the start of the number after incrementing it.



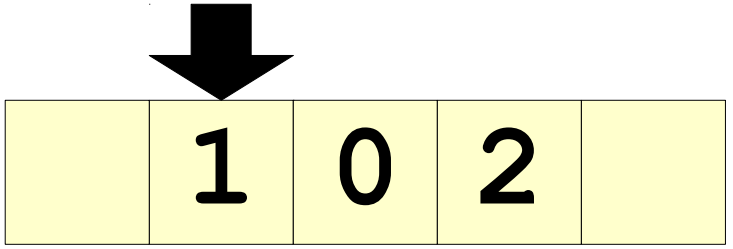
Incrementing Numbers

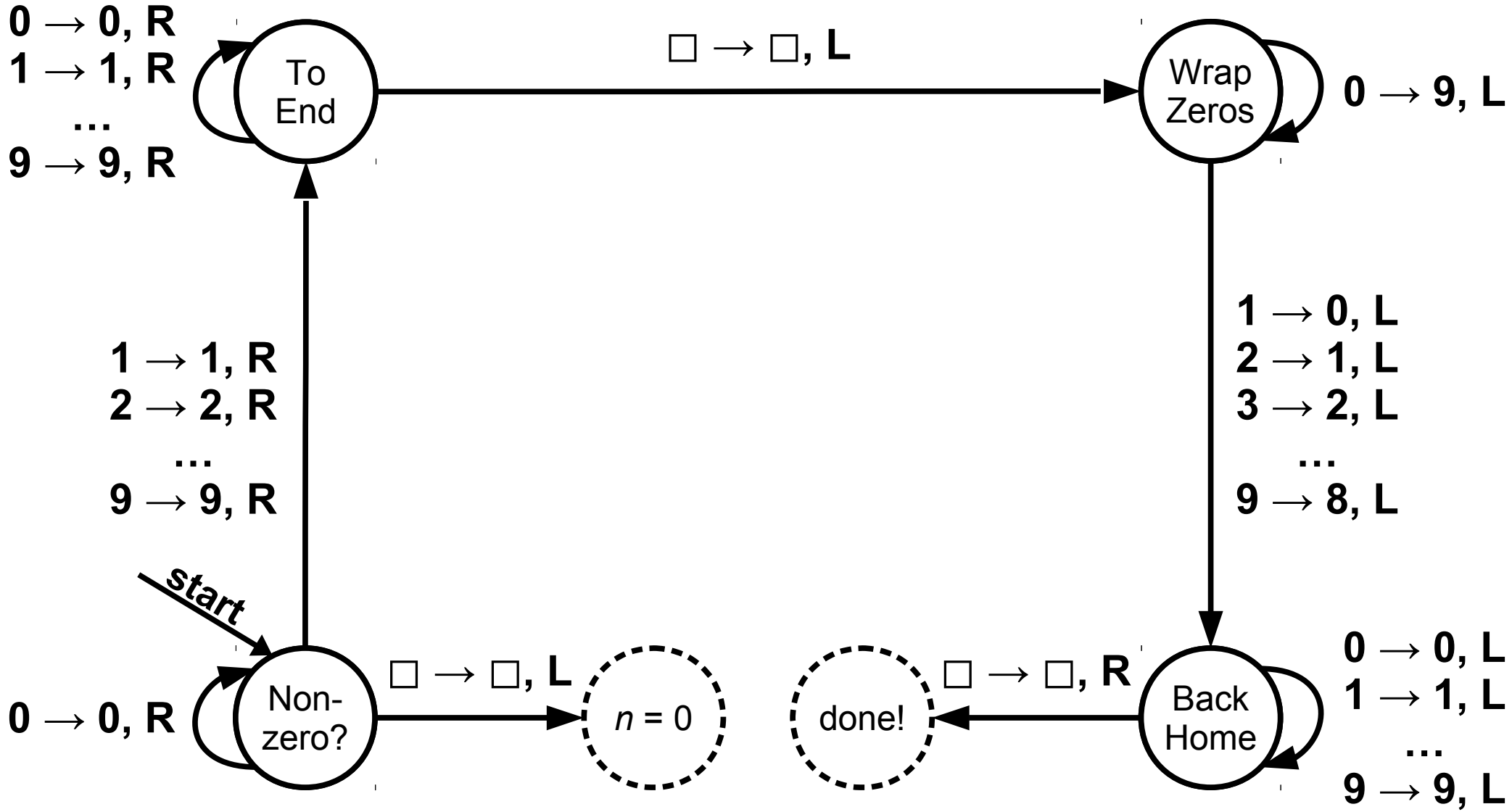
```
go to the end of the number;
while (the current digit is 9) {
    set the current digit to 0;
    back up one digit;
}
increment the current digit;
go to the start of the number;
```


start
0 → 0, R
1 → 1, R
…
9 → 9, R
To End
□ → □, L
Wrap Nines
9 → 0, L
0 → 1, L
1 → 2, L
2 → 3, L
…
8 → 9, L
□ → 1, L
Back Home
0 → 0, L
1 → 1, L
…
9 → 9, L
□ → □, R
done!

Decrementing Numbers

- Now, let's build a TM that decrements a number.
- We'll assume that
  - the tape head points at the start of a number,
  - there is at least one blank on each side of the number.
- The tape head will end at the start of the number after decrementing it.
- If the number is 0, then the subroutine should somehow signal this rather than making the number negative.



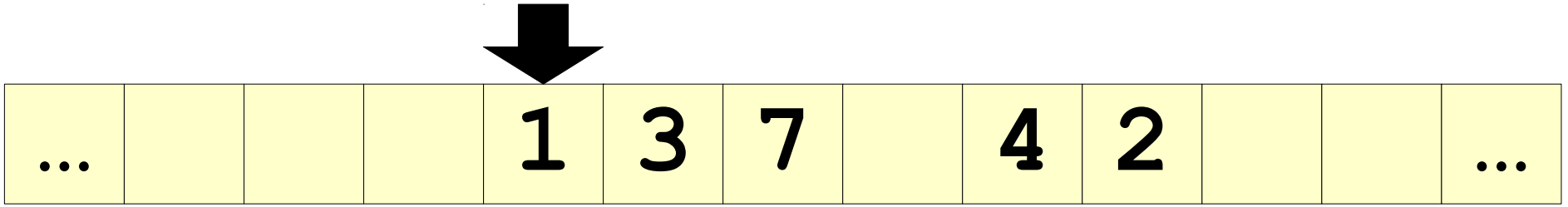

0 → 0, R
1 → 1, R
…
9 → 9, R
To End
□ → □, L
Wrap Zeros
0 → 9, L
1 → 0, L
2 → 1, L
3 → 2, L
…
9 → 8, L
1 → 1, R
2 → 2, R
…
9 → 9, R
start
Non-zero?
0 → 0, R
□ → □, L
n = 0
done!
□ → □, R
Back Home
0 → 0, L
1 → 1, L
…
9 → 9, L

TM Subroutines

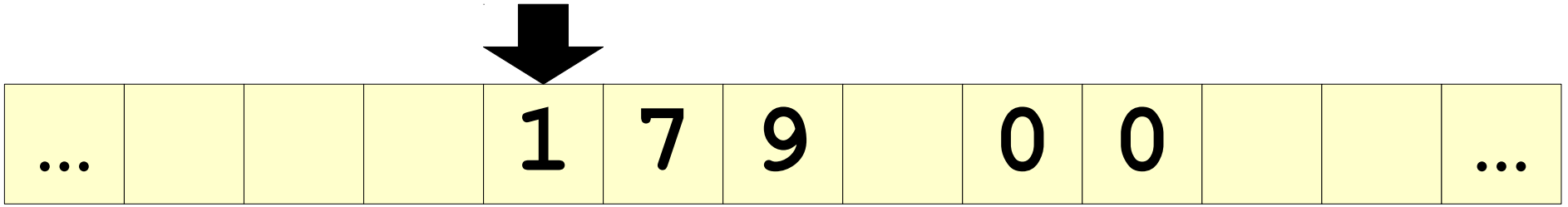
- Sometimes, a subroutine needs to report back some information about what happened.
- Just as a function can return multiple different values, we'll allow subroutines to have different “done” states.
- Each state can then be wired to a different state, so a TM using the subroutine can control what happens next.

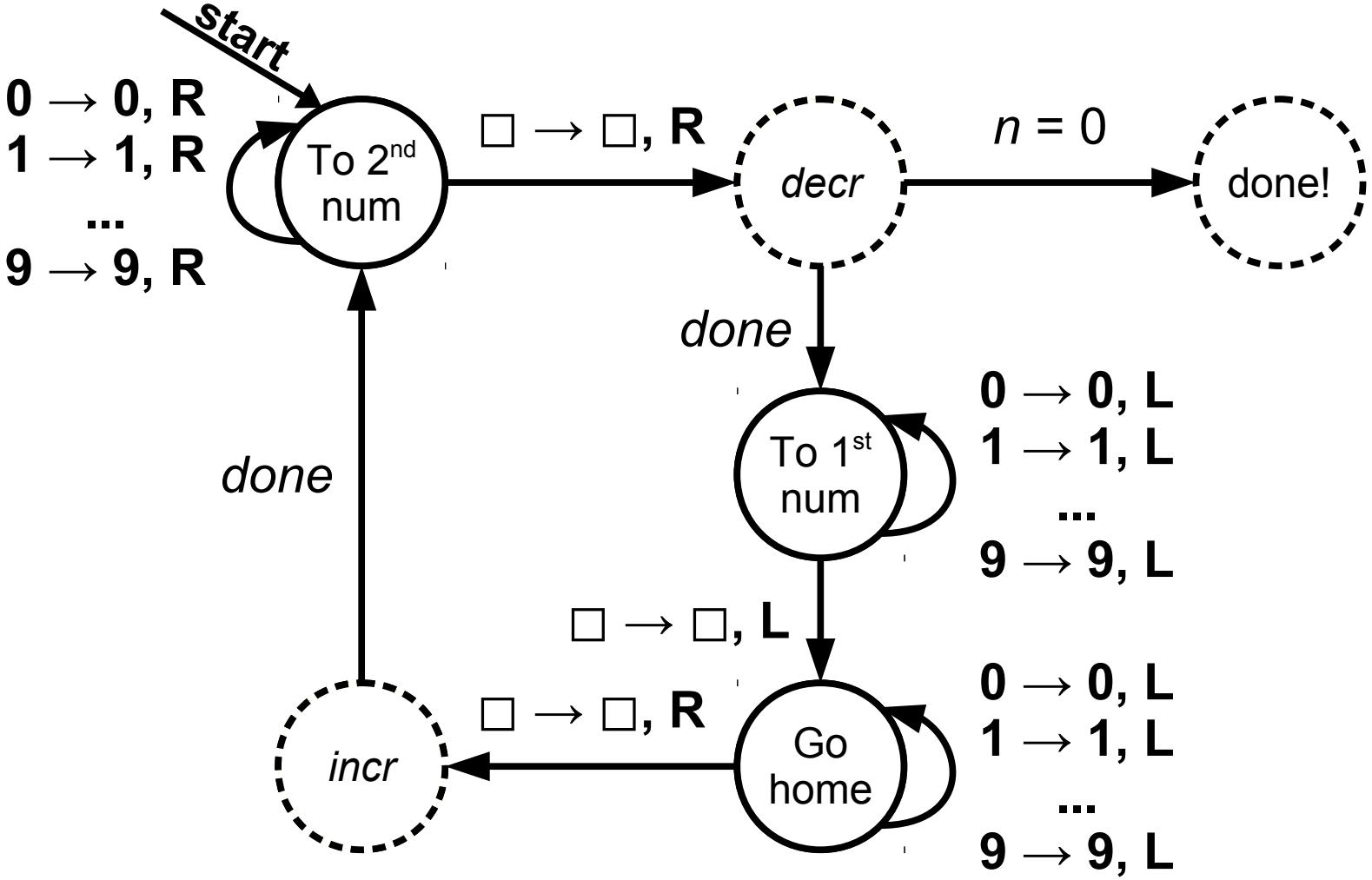
Putting it All Together

- Our goal is to build a TM that, given two numbers, adds those numbers together.
- Before:



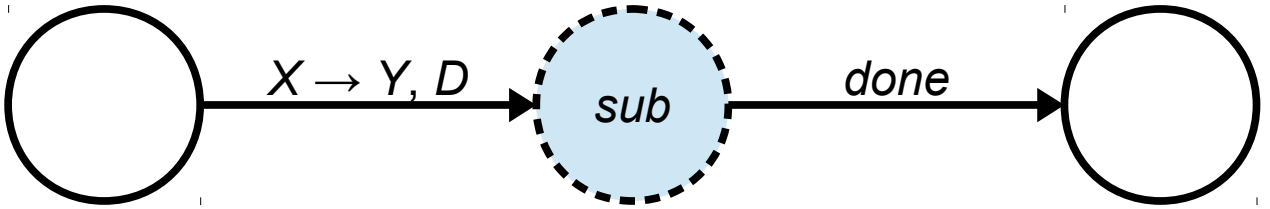
- After:




start
0 → 0, R
1 → 1, R
...
9 → 9, R
To 2nd num
□ → □, R
decr
n = 0
done!
done
To 1st num
0 → 0, L
1 → 1, L
...
9 → 9, L
□ → □, L
Go home
0 → 0, L
1 → 1, L
...
9 → 9, L
□ → □, R
incr
done

Using Subroutines

- Once you've built a subroutine, you can wire it into another TM with something that, schematically, looks like this:

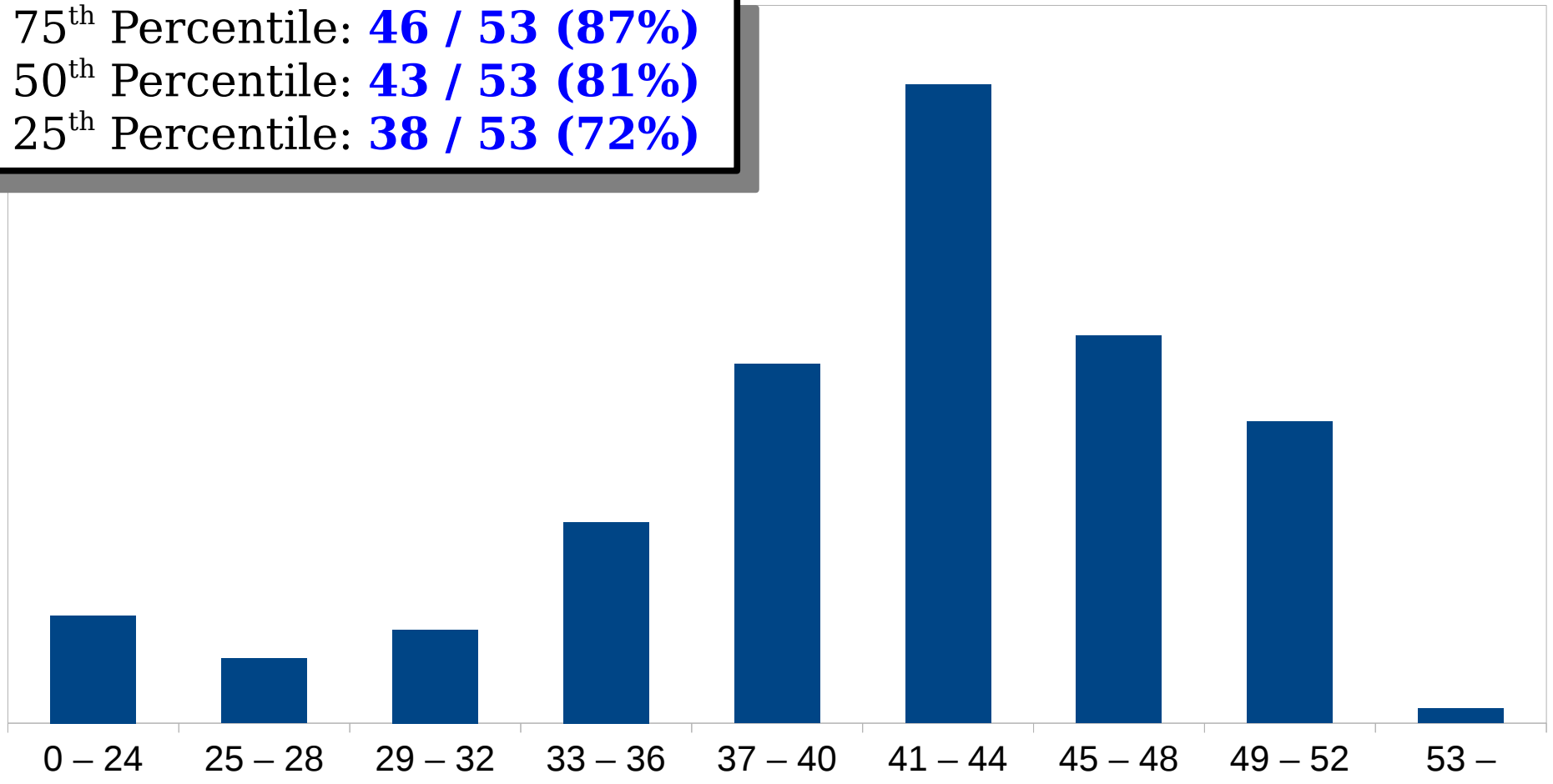


- Intuitively, this corresponds to transitioning to the start state of the subroutine, then replacing the “done” state of the subroutine with the state at the end of the transition.

Time-Out for Announcements!

Problem Set 6 Graded

75th Percentile: **46 / 53 (87%)**
50th Percentile: **43 / 53 (81%)**
25th Percentile: **38 / 53 (72%)**



Problem Sets

- Problem Set Seven was due at 2:30PM today.
  - Using late days, you can extend the deadline to this Monday at 2:30PM.
- Problem Set Eight goes out today. It's due the Friday after Thanksgiving break at 2:30PM.
  - Play around with CFGs and Turing machines!
  - We have online tools you'll use to design, test, and submit your grammars and TMs. Hope this helps!
  - Some of the topics on PS8 will be covered on the Monday we get back – problems referencing these topics are clearly marked.

Extra Practice Problems 3

- We've posted a (massive!) set of practice problems (EPP3) on the course website, with solutions.
- These problems cover all the topics we'll explore this quarter. Questions involving topics we will cover next week are marked with a star.
- Feel free to work through any of the problems that seem interesting and to ask questions!

Thanksgiving Break

- We will not be holding our standard office hours over Thanksgiving Break.
- We will be checking Piazza, though.
- Life advice:
  - If you’re an international student: find a friend and ask if you can join them for Thanksgiving. It’s one of those lovely secular rituals that I think everyone should experience at least once.
  - If you’re from the US: find a friend who isn’t an invite them to join you for Thanksgiving!

Your Questions

“I couldn't answer one of the questions on the midterm and freaked out. I know I don't handle failure well. How do you handle failure?”

Remember that a test is a single, noisy measurement of how you did at a specific point in time in a specific environment. Remember that you are learning. Remember that your abilities are malleable, and that this malleability is under your control.

Don’t confuse “I didn’t do as well as I would have liked” with “I have failed.” Don’t confuse “I still have a ways to go” with “I have failed.” Don’t confuse “People around me are doing better” with “I have failed.” And, please, don’t confuse “whelp, that didn’t work” with “I have failed.”

Put things in context. When something doesn’t work, take the time to see why it didn’t work. When your approach to something isn’t working, think about how you can change it.

The best way I know how to deal with failure is to remember that you are playing the long game. Don’t let a single, small blip burn off the incredible momentum you have going. Remember that you are an incredible person, since otherwise you wouldn’t be here.

“If you could have one Superpower, what would it be?”

Time for a bad joke!

Back to CS103!

Main Question for Today:

Just how powerful are Turing machines?

How Powerful are TMs?

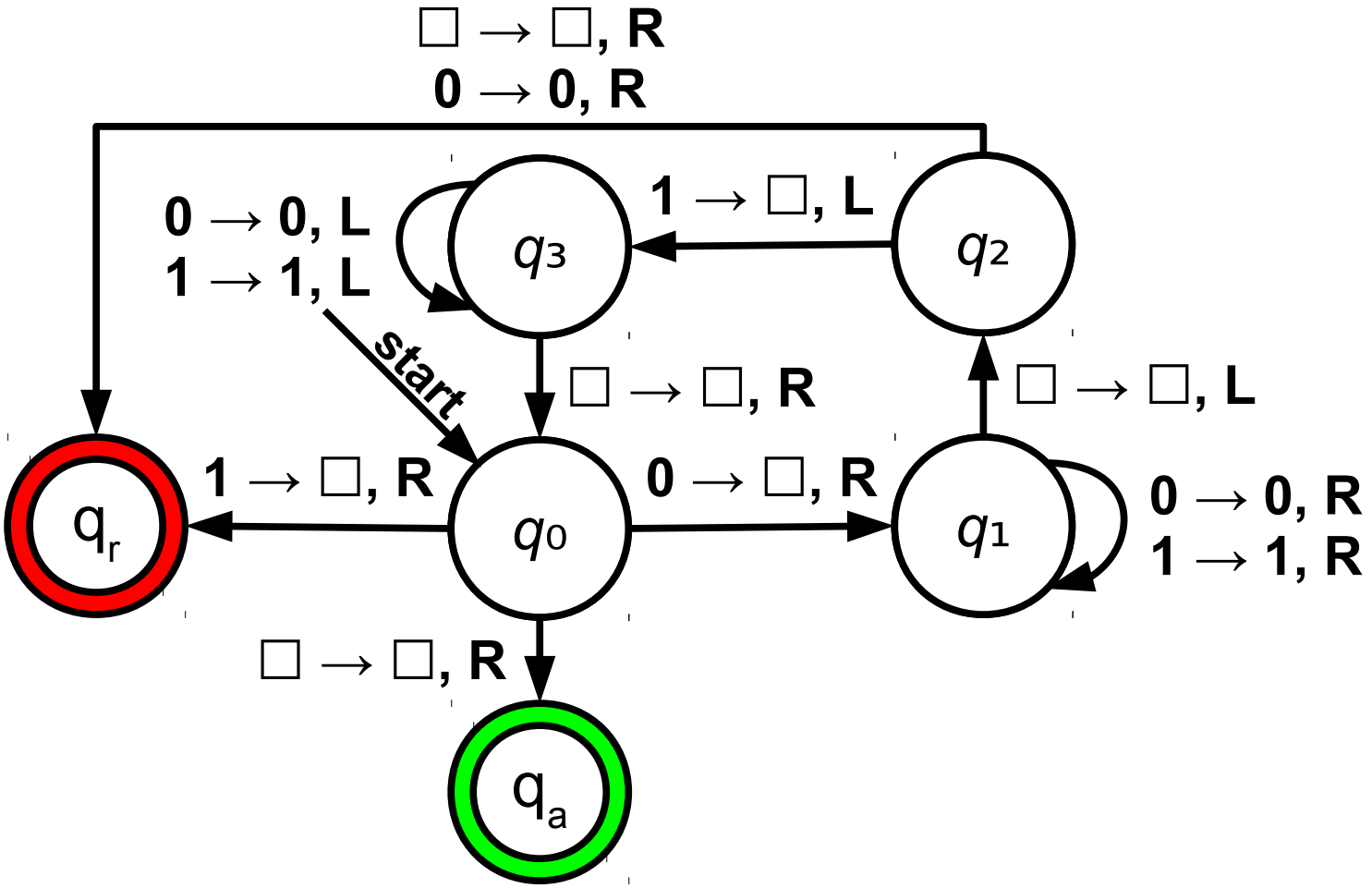
- Regular languages, intuitively, are as powerful as computers with finite memory.
- TMs by themselves seem like they can do a fair number of tasks, but it's unclear specifically what they can do.
- Let's explore their expressive power.

Real and “Ideal” Computers

- A real computer has memory limitations: you have a finite amount of RAM, a finite amount of disk space, etc.
- However, as computers get more and more powerful, the amount of memory available keeps increasing.
- An ***idealized computer*** is like a regular computer, but with unlimited RAM and disk space. It functions just like a regular computer, but never runs out of memory.

Claim 1: Idealized computers can simulate Turing machines.

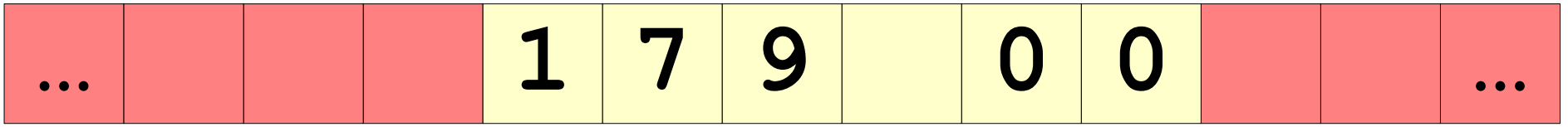
"Anything that can be done with a TM can also be done with an unbounded-memory computer."



	0			1			□		
q_0	q_1	□	R	q_r	□	R	q_a	□	R
q_1	q_1	0	R	q_1	1	R	q_2	□	L
q_2	q_r	0	R	q_3	□	L	q_r	□	R
q_3	q_3	0	L	q_3	1	L	q_0	□	R

Simulating a TM

- To simulate a TM, the computer would need to be able to keep track of
  - the finite-state control,
  - the current state,
  - the position of the tape head, and
  - the tape contents.
- The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
- We only need to store the “interesting” part of the tape (the parts that have been read from or written to so far.)



Claim 2: Turing machines can simulate idealized computers.

"Anything that can be done with an unbounded-memory computer can be done with a TM."

What We've Seen

- TMs can
  - implement loops (basically, every TM we've seen).
  - make function calls (subroutines).
  - keep track of natural numbers (written in unary or in decimal on the tape).
  - perform elementary arithmetic (equality testing, multiplication, addition, increment, decrement, etc.).
  - perform if/else tests (different transitions based on different cases).

What Else Can TMs Do?

- Maintain variables.
  - Have a dedicated part of the tape where the variables are stored.
  - We've seen this before: take a look at our machine for composite numbers, or for increment/decrement.
- Maintain arrays and linked structures.
  - Divide the tape into different regions corresponding to memory locations.
  - Represent arrays and linked structures by keeping track of the ID of one of those regions.

A CS107 Perspective

- Internally, computers execute by using basic operations like
  - simple arithmetic,
  - memory reads and writes,
  - branches and jumps,
  - register operations,
  - etc.
- Each of these are simple enough that they could be simulated by a Turing machine.

A Leap of Faith

- It may require a leap of faith, but anything you can do a computer (excluding randomness and user input) can be performed by a Turing machine.
- The resulting TM might be colossal, or really slow, or both, but it would still faithfully simulate the computer.
- We're going to take this as an article of faith in CS103. If you curious for more details, come talk to me after class.

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

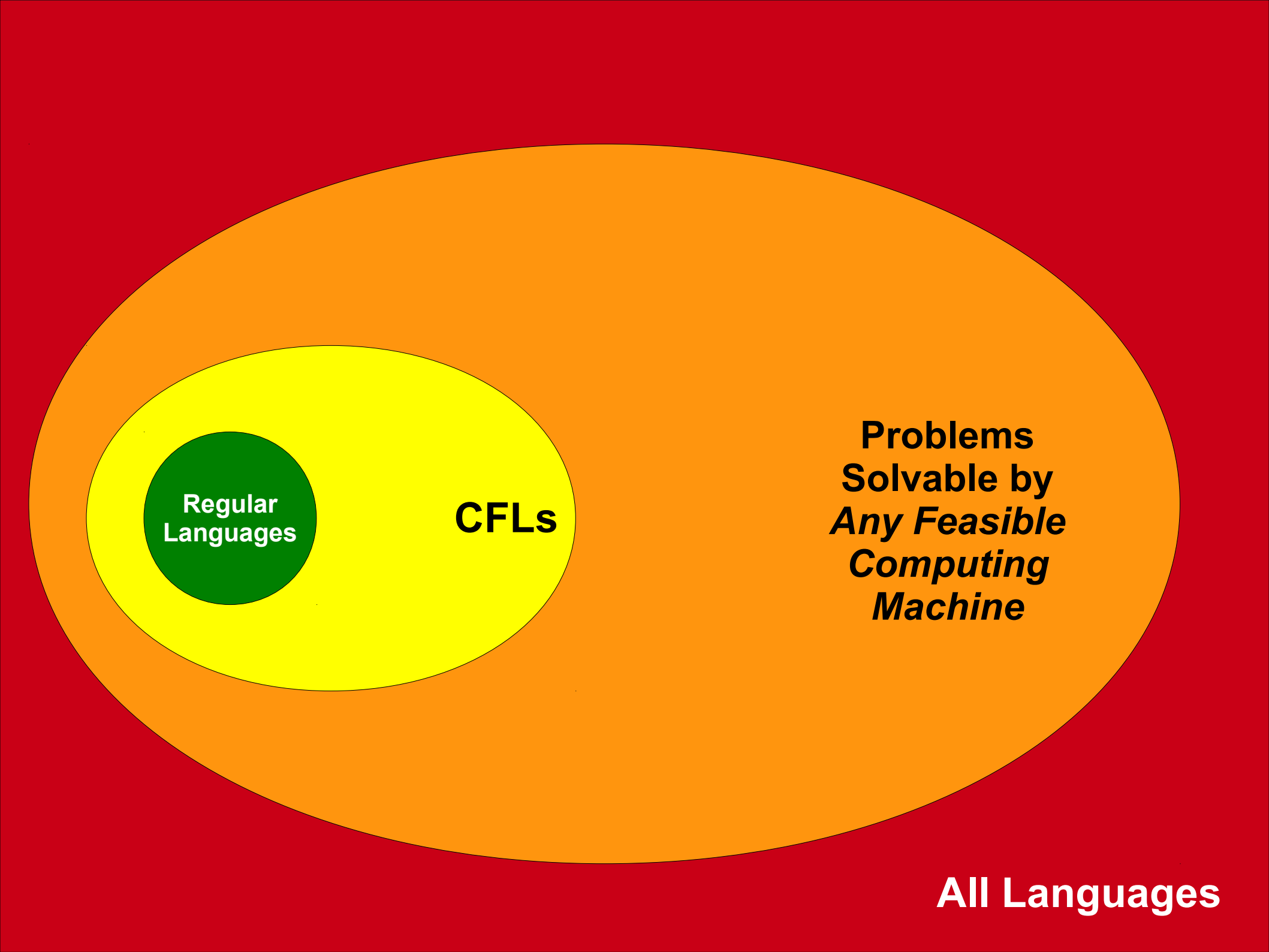
Effective Computation

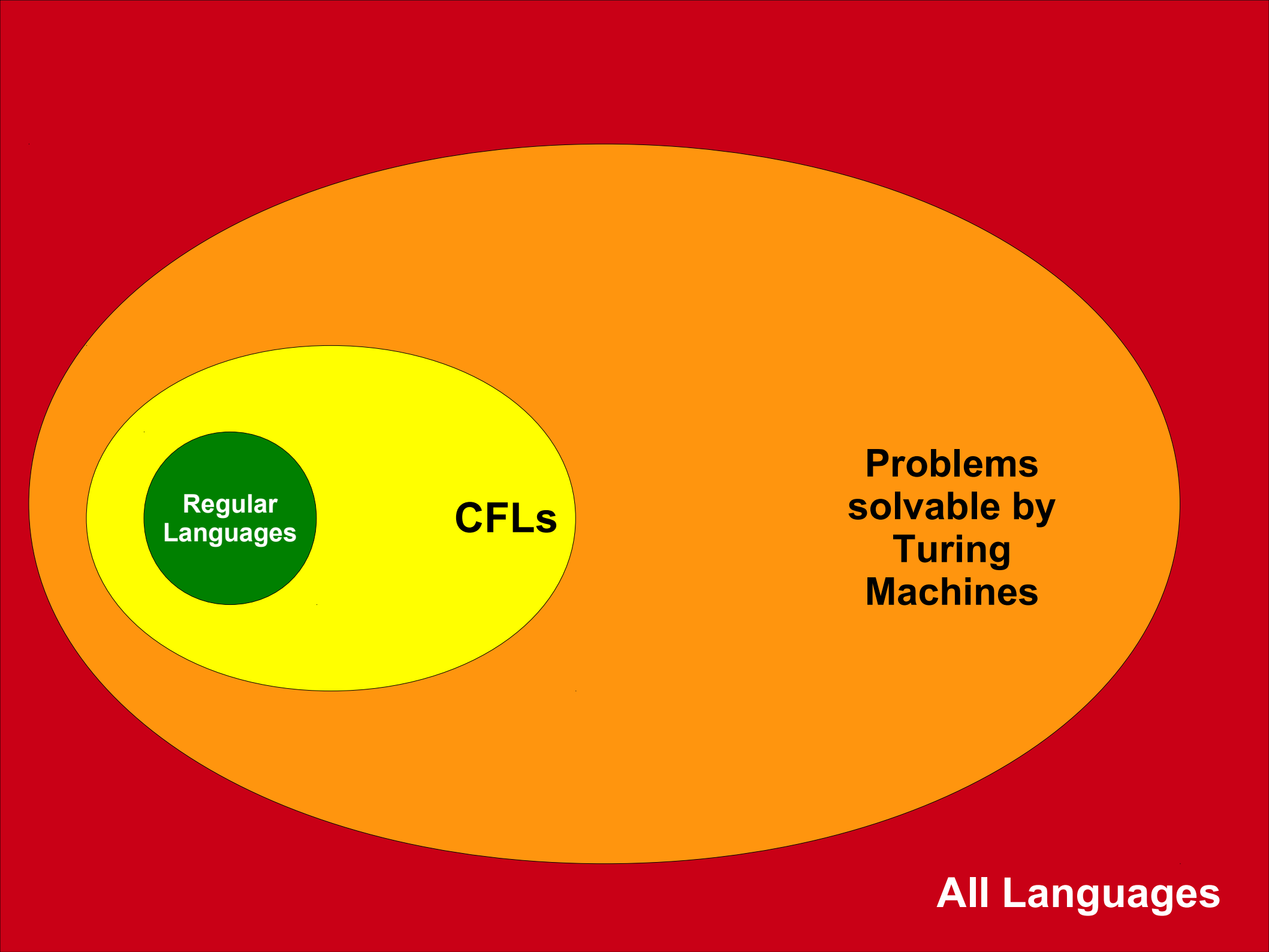
- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation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

The ***Church-Turing Thesis*** claims that

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

> “This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!”
>
> - Ryan Williams


Regular Languages
CFLs
Problems Solvable by *Any Feasible Computing Machine*
All Languages


Regular Languages
CFLs
Problems solvable by Turing Machines
All Languages

TMs ≈ Computers

- Because Turing machines have the same computational powers as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.
- Going forward, we're going to switch back and forth between TMs and computer programs based on whatever is most appropriate.
- In fact, our eventual proofs about the existence of impossible problems will involve a good amount of pseudocode. Stay tuned for details!

Next Time

Enjoy Thanksgiving break!

- ***Then, when we come back…***
  - ***What is a "problem?"***
  - ***What does it mean to "solve" a problem?***
  - ***What is an emergent property?***
  - ***What are the emergent properties of computation?***