CS3102 Theory of Computation

Warm up:

Think of a yes/no question one could ask about a string.

- finite alphabet
- is it at least 7 characters long
- is it a java program
- is it a java program that will print "hi"
- does it have some char?
- is there a character 3x in a row?
- is it empty
- is it a palindrome
- is it finite?
- is it a digit

Logistics

- Homework released tomorrow
  - See submission page for deadlines (I’m still processing your quiz 3)
- Quiz will be released Thursday, due Tuesday

Last Time

- Exam

What do we compute (redux)

- Input: String (over some alphabet Σ)

finite

Σ^*

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

- So far:
  - We compute a function $f: \Sigma^* \to \Sigma^*$
    - For circuits: $\{0,1\}^n \to \{0,1\}^m$
- Other ideas:
  - Decision problem: $f: \Sigma^* \to \{0,1\}$
    - Does this string have some property?
  - Language: $L \subseteq \Sigma^*$
    - The set of all strings with some property

Function vs Decision vs Language

Name	Decision Problem	Function	Language
XOR 1011101	Are there an odd number of 1's?	$f(b) = \begin{cases} 0 & \text{number of 1s is even} \\ 1 & \text{number of 1s is } odd \end{cases}$	$\{b \in \Sigma^* \mid b \text{ has and odd number of 1s}\}$ $\Sigma = \{0,1\}$
Majority	Are there more 1s than 0s?	$f(b) = \begin{cases} 0 & \text{more 0s than 1s} \\ 1 & \text{more 1s than 0s} \end{cases}$	$\{b \in \Sigma^* \mid b \text{ has more 1s than 0s}\}$
Thing to compute	yes/no question about strings	$f: \Sigma^* \to \{0,1\}$ ans = F ans = T	$\{b \in \Sigma^* \mid f(b) = 1\}$

$.^*000.^*$

Finite vs. Infinite Functions

- Boolean Circuits have a drawback:
  - Fixed number of inputs
- What we want:
  - A single recipe which can take infinitely many inputs

Example: XOR

We can define XOR to take an unbounded number of input~~s~~ bits

$$XOR: \underbrace{\{0,1\}^*} \to \underbrace{\{0,1\}}$$

decision/ language

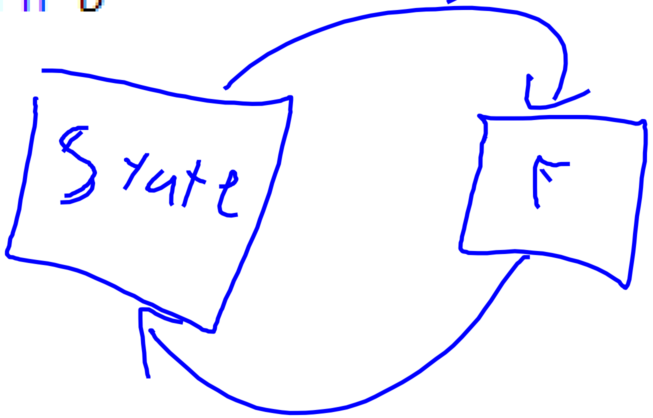
Returns 1 if there are an odd number of 1s in the input

We need a new model

- As a programming language:
  – Add loops!

```
def XOR(x):
    b = 0
    i = 0
    while i < len(x):
        b = XOR(b, x[i])
        i = i + 1
    return b
```

- As "hardware":
  – Automata



$\boxed{\text{Finite State}}$ Automaton

Finite # options

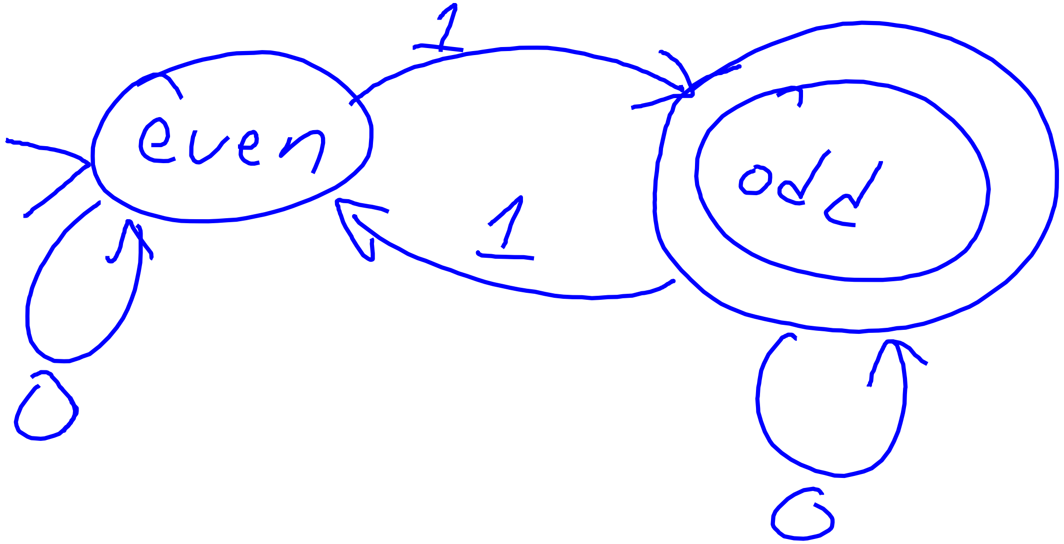
- Implementation:
  - Finite number of states Q
  - One start state $q_0 \in Q$
  - "Final" states $F \subseteq Q$
  - Transitions (function mapping state-character pairs to states)
- Execution:
  - Start in the initial "state"
  - Read each character once, in order (no looking back)
  - Transition to a new state once per character (based on current state and character)
  - Give output depending on which state you end in

$\in F \rightarrow ans = 1$

$o.w. \rightarrow ans = 0$

Computing Infinite XOR

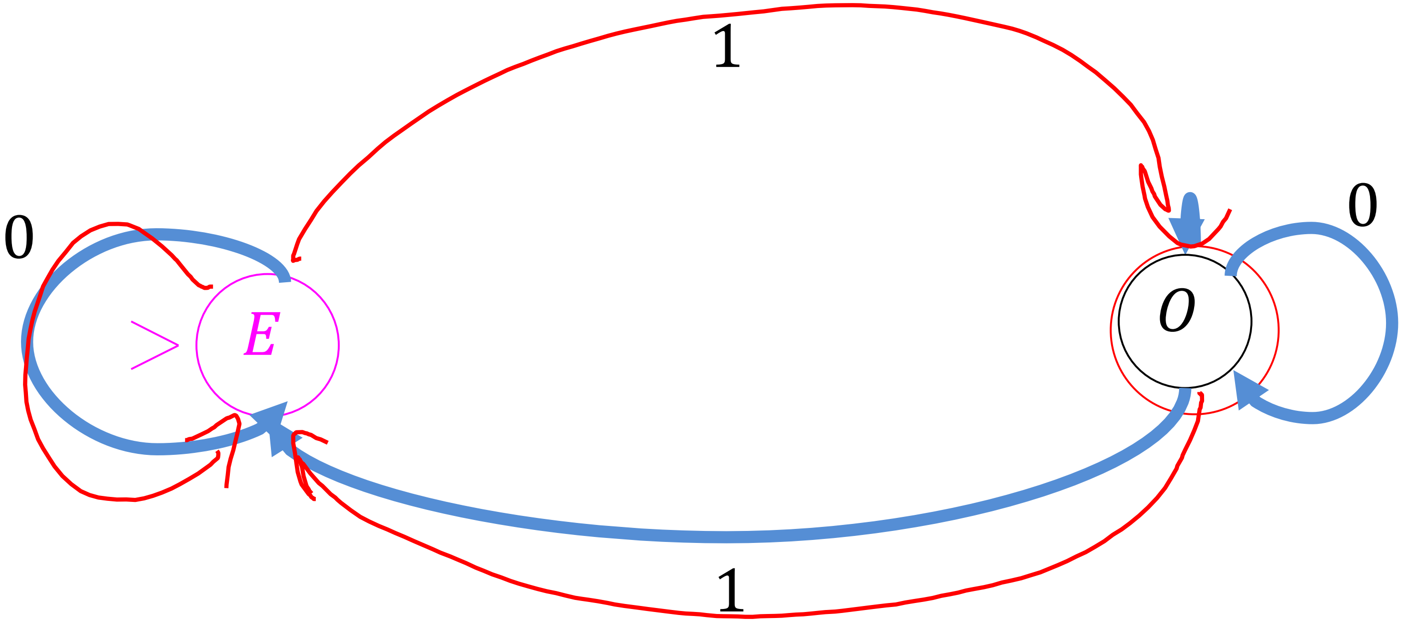
- The "state" we start in:
  - Even number of 1's seen
- The "state" in which we return 1
  - Odd number of 1's seen
- Reading one bit at a time:
  - If we're currently in "Even":
    - Switch to "Odd" when we see a 1
    - Stay in even when we see a 0
  - If we're currently in "Even":
    - Switch to "Odd" when we see a 1
    - Stay in even when we see a 0



Let’s Draw It!

Let's Draw It!

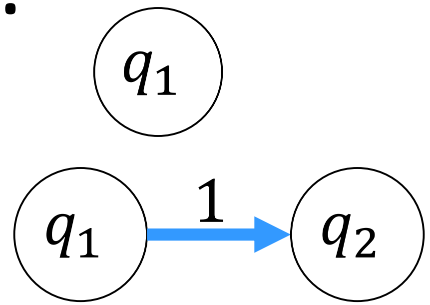
State:



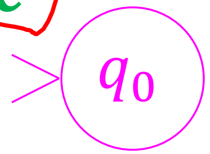
return 1

Finite State Automata

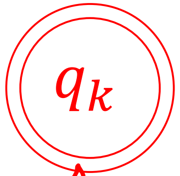
- Basic idea: a FA is a "machine" that changes states while processing symbols, one at a time.



- Finite set of states: $Q = \{q_0, q_1, \ldots q_n\}$
- Transition function: $\delta : Q \times \Sigma \rightarrow Q$



- Initial state: $q_0 \in Q$
- Final states: $F \subseteq Q$



- Finite state automaton is $M = (Q, \Sigma, \delta, q_0, F)$
- Return 1 if we end in a Final state, otherwise return 0

Computing with a FSA

state $q = q_0$

for each bit b in the input:

    $q = \delta(q, b)$

return whether $q \in F$

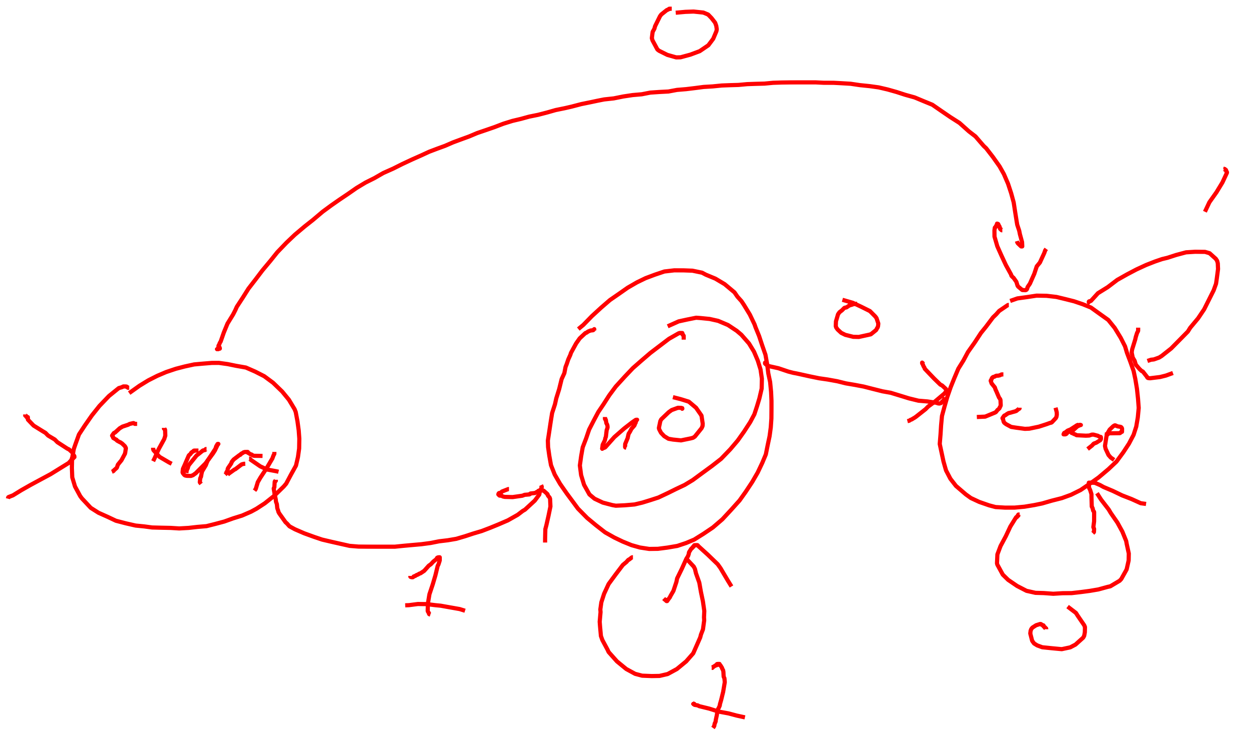
Example: AND

Example: AND

$Q = \{noOs, someOs, start\}$

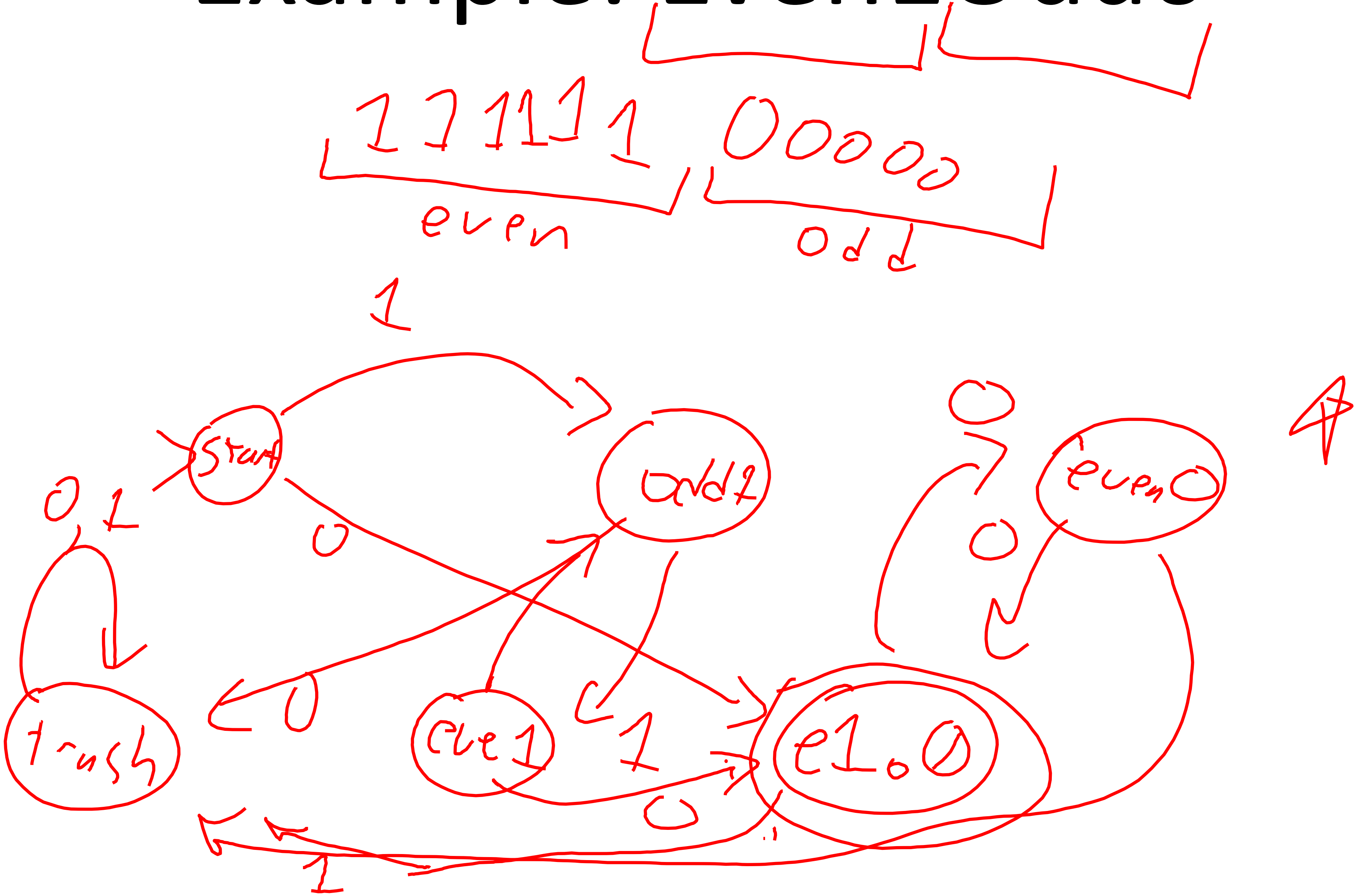
$q_0 = start$

$F = \{noOs,$



Example: AND

Example: Even1Odd0



Example: Even1Odd0

Example: Even1Odd0

FSA are strictly more powerful than NAND circuits

- How can we show this?
  - Show that there is at least one function we can do with FSA but not NAND-CIRC
    - Done! (infinite XOR)
  - Show anything we can do with NAND-CIRC can also be done with FSA
    - How?
    - We need to be able to compute any finite function

Computing any finite function with NAND-CIRC

- Summary:
  - "Manually Precompute" the output for every (finitely-many) possible input
  - When we receive the actual input, do a "lookup"
- Our proof before:
  - Make a variable to represent each possible input, assigning its value to match the correct output
  - Use LOOKUP to return the proper variable for the given input

Straightline Code for f

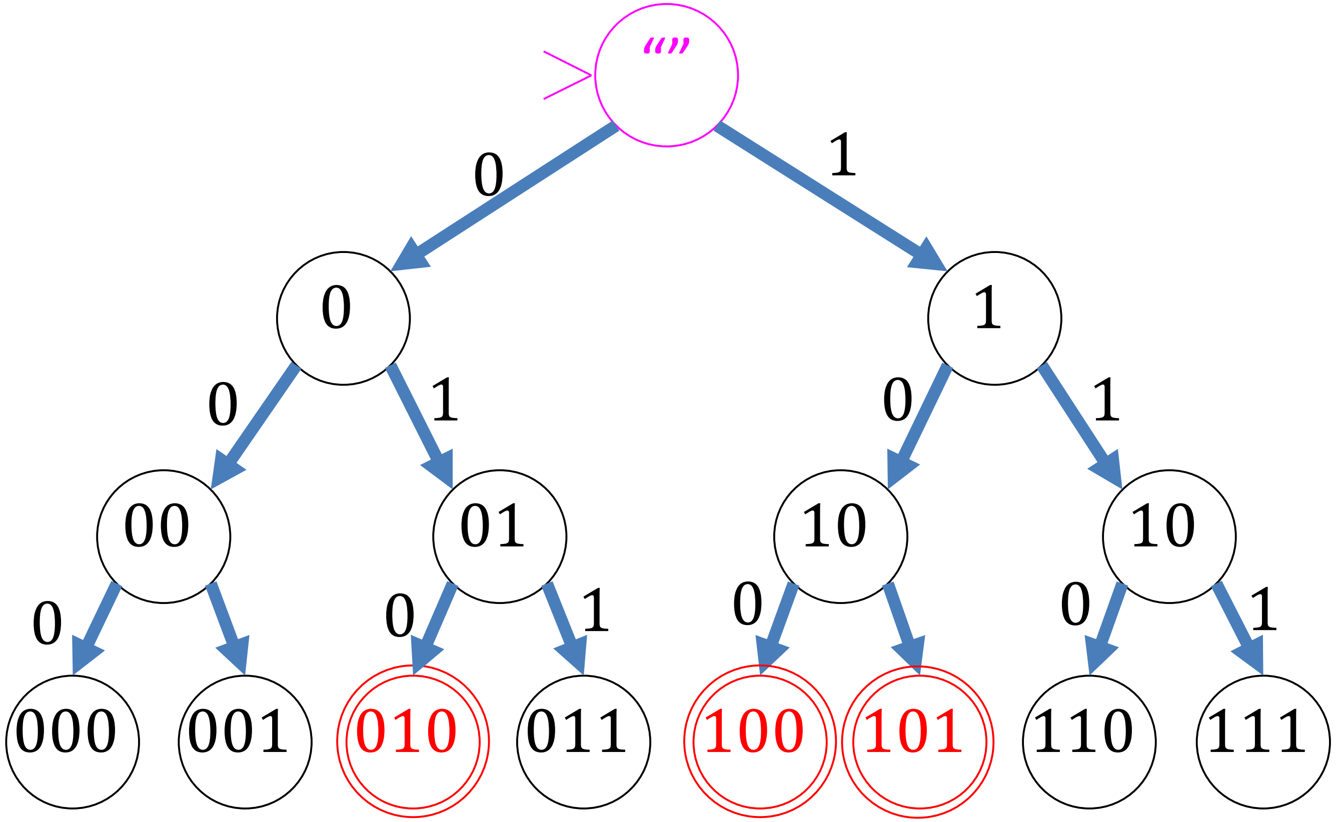
Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

```
def F(x0,x1,x2):
    F000=0
    F001=0
    F010=1
    F011=0
    F100=1
    F101=1
    F110=0
    F111=1
    return LOOKUP3(F000,F001,F010,F011,F100,F101,F110,F111,x0,x1,x2)
```

Computing finite functions with FSA

- Summary:
  - "Manually Precompute" the output for every (finitely-many) possible input
  - When we receive the actual input, do a "lookup"
- Same idea, but with Automata:
  - Make a state for every possible input, determining whether or not it is final depending on the correct output
  - Do a "binary tree traversal" with the given input to navigate to its correct output

FSA for f



Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

Characterizing What's computable

- Things that are computable by FSA:
  - Functions that don't need "memory"
  - Languages expressible as Regular Expressions (next time)
- Things that aren't computable by FSA:
  - Things that require more than finitely many states
  - Intuitive example: Majority

Majority with FSA?

- Consider an inputs with lots of 0s

000...0000 111...1111	000...0000 111...1111	000...0000 111...1111
×49,999 ×50,000	×50,000 ×50,000	×50,000 ×50,001

- Recall: we read 1 bit at a time, no going back!
- To count to 50,000, we'll need 50,000 states!