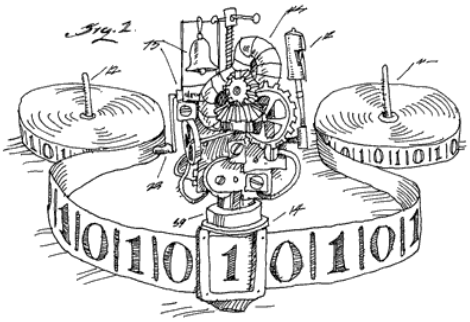
What problems can our computers solve?
To answer this question, we need to understand the *machines* we can use to solve problems.

Turing Machines.



<u>A Turing machine has:</u>

- A *finite* list of states, Q
- A *finite* alphabet A={_, 0, 1} of symbols
- A *finite* list δ of "transition rules"
  - <u>Inputs</u>: current state & read-in symbol
  - <u>Outputs</u>: new state, write-out symbol, direction to move ($\leftarrow$, $\rightarrow$, 0)
- An *infinite* tape to read and write – the Turing machine's memory.

<u>Test Yourself #1</u>

If Q has 5 states and A={_, 0, 1}, then how big is the list δ of transition rules? _____

(Hint: How many states are there? How many possible read-in symbols are there?
So how many possible transition rules are there?)

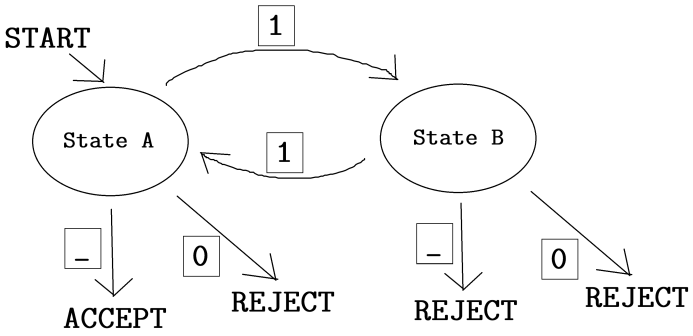
<u>Using the Turing machine to solve problems</u>

Usually, we are running the Turing machine with *something already written on its tape*. This is called the **input.** We will be thinking about Turing machines that read the input, do some computing, and then come to one of three states:

1. they **halt** and **accept** the input
2. they **halt** and **reject** the input
3. they **loop** forever, never accepting nor rejecting

<u>Example of a Turing machine</u>

This is a Turing machine, EVEN_TM, which outputs **accept** if its input has an even number of "1"s, and outputs **reject** if its input has any "0"s or an odd number of "1"s.



This particular Turing machine always moves forward ($\rightarrow$) and never writes to its tape, so I didn't bother writing down the *Write:* and *Move:* rules on each arrow. (But they're important too.)

What does it mean to solve a problem, anyways?

Instead of using the word "problem", we'll be using the word "language" to represent something we might want to calculate. And instead of using the word "solves", we'll be using the word "decides" to describe a machine that solves a problem.

A **language** is a *list of Good Inputs.* For example, the language EVEN={"", "11", "1111", ...} represents all inputs that have a multiple-of-2 "1"s. The problem is "does the input has an even number of "1"s?"

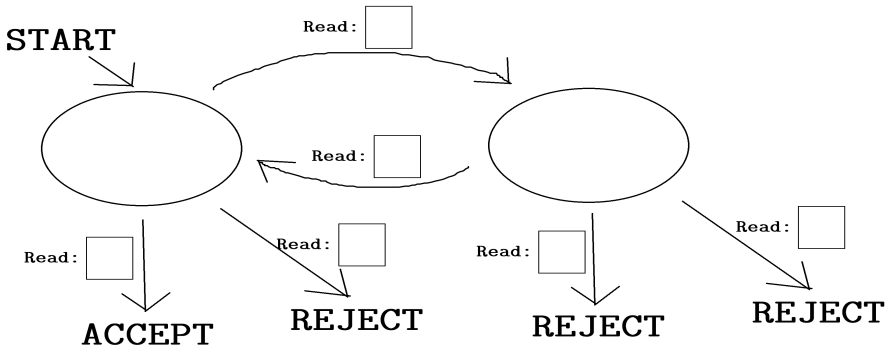
A Turing machine that **decides** some language L is one that accepts every input within L, and rejects every input that is not in L. For example, EVEN_TM (our example above) decides EVEN.

We can create more complicated languages to describe more complicated problems we might need to solve. For example, the language PRIMES={"10", "11", "101", "111", "1011", ...} represents the problem of calculating whether a number (written in binary) is prime.

To abbreviate "the output of the Turing machine M when run on the input x," we write *M(x).*

Test Yourself #2

1. Fill in the transition rules for this Turing machine which is meant to decide the language TENS={"", "10", "1010", "101010", ...}. It should accept any input which has any number of "10"s. The Turing machine always moves ($\rightarrow$) after each read, and never needs to write anything.



2. Draw a Turing machine that decides the language ONE={"1", "01", "001", "0001", ...}. It should accept any input which has any number of zeroes, followed by a single one. Your Turing machine should always move ($\rightarrow$) after each read, and never needs to write anything.

The Church-Turing Thesis.

More Symbols: A Turing machine with symbols {0,1} can solve no fewer problems than a Turing machine with symbols {0,1,2,3} can.

(Why?)

More Tapes: A Turing machine with access to 1 tape can solve no fewer problems than a Turing machine with access to 2 tapes can.

(Why?)

The Church-Turing Thesis:

A simple Turing machine can solve as many problems – or more – as compared to all other sorts of computing machines we can imagine.

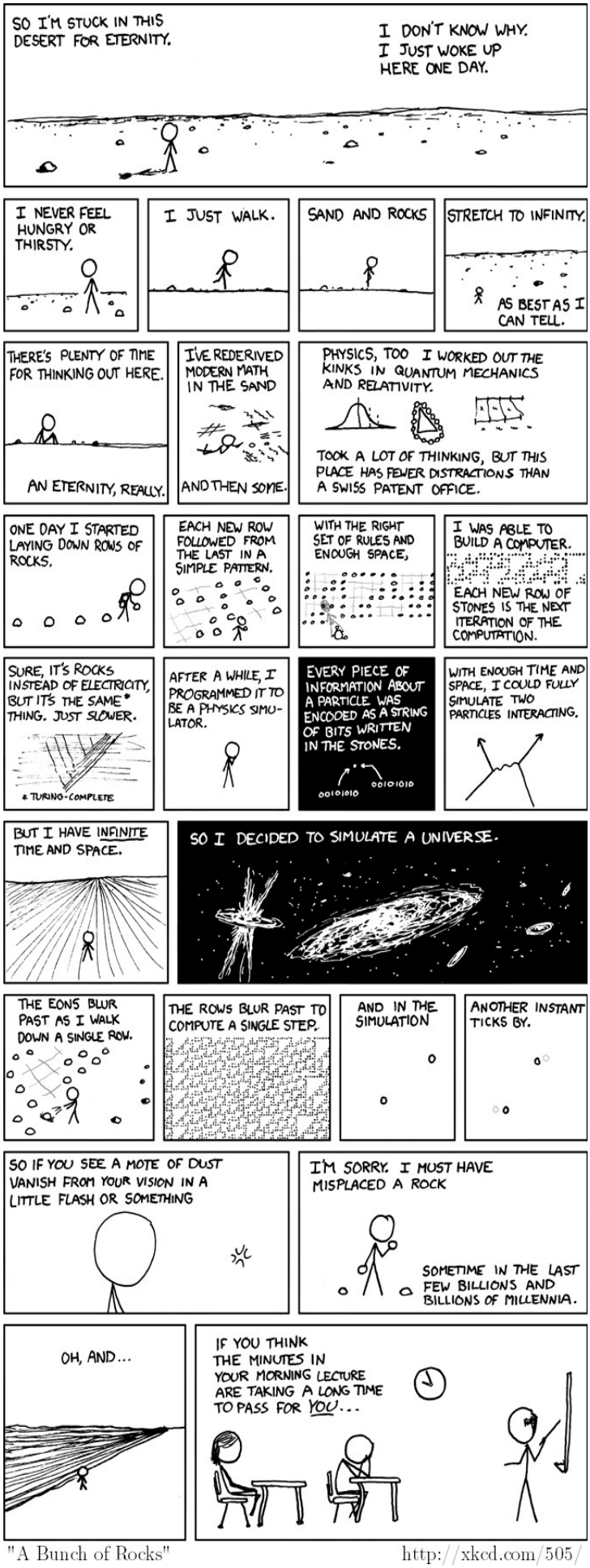
(*Is it true?* We're not sure! *Quantum computing* is an exciting new field. It might turn out that we can build quantum computers to solve more problems than Turing machines can.)

"Turing-complete": Anything which has equal computational power to a Turing machine. It must allow potentially-infinite memory and potentially-infinite computation time. If you can use it to simulate any Turing machine, it's Turing-complete.

Test Yourself #3

Is it Turing-complete?

Turing machines	Y / N
Minecraft circuits (in an (*infinite* Minecraft world)	Y / N
This desert (→)	Y / N
Your computer (which has non-infinite memory)	Y / N



"A Bunch of Rocks" http://xkcd.com/505/

Turing machines within Turing machines!

Descriptions of Turing Machines

It is possible to write down any Turing machine as a sequence of zeros and ones. This sequence describes everything about the Turing machine that we mentioned earlier – *Q, A*, and *δ.*

We call this the **description** of the Turing machine.

We abbreviate "the description of Turing machine M" as just "<M>".

Universal Turing Machines:

Alan Turing proved the existence of a "**Universal Turing Machine**" (let's call it U). The Universal Turing Machine *simulates other Turing machines.*

U takes as its input the description of *any* Turing machine, <M>, followed by *any* input x. We call U the "Universal Turing Machine" because it can then simulate what M would do when M runs on the input x. U(<M>, x) produces exactly the same output as M(x).

$$U(<M>, x) = M(x)$$