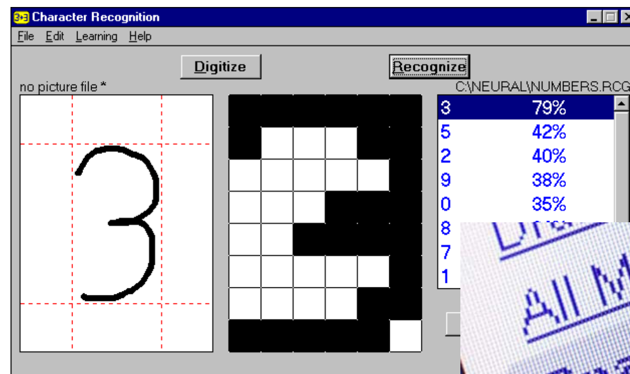
Algorithmic Learning Theory

JENNIFER CHUBB

UNIVERSITY OF SAN FRANCISCO

Context

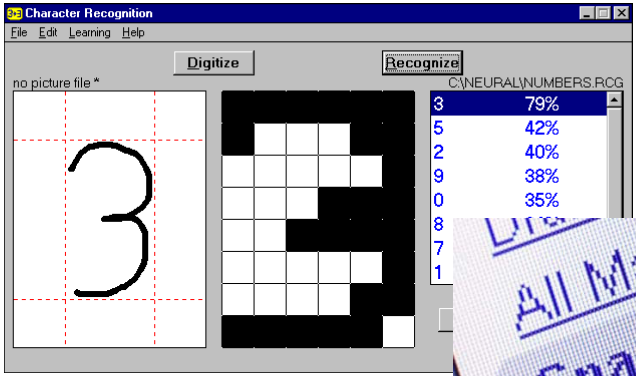
- Machine learning is a huge field.
- Computer science, artificial intelligence, statistics, mathematics.
- Machines learn from the data, they don't just process it.
  - Optical character recognition.
  - Learning to identify "spam" from "not spam."
  - Robots driving.

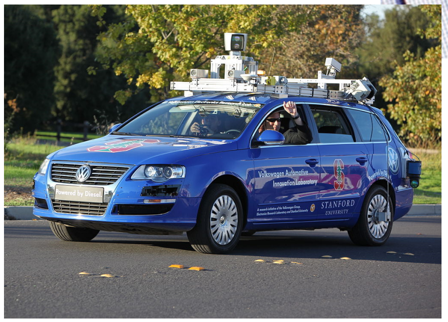


Context

- Computational learning theory is a branch of computer science devoted to analysis of machine learning algorithms.
  - Resource bounds (time & space)
  - Accuracy
  - Theoretical capabilities
    - Algorithmic learning theory

This is us.



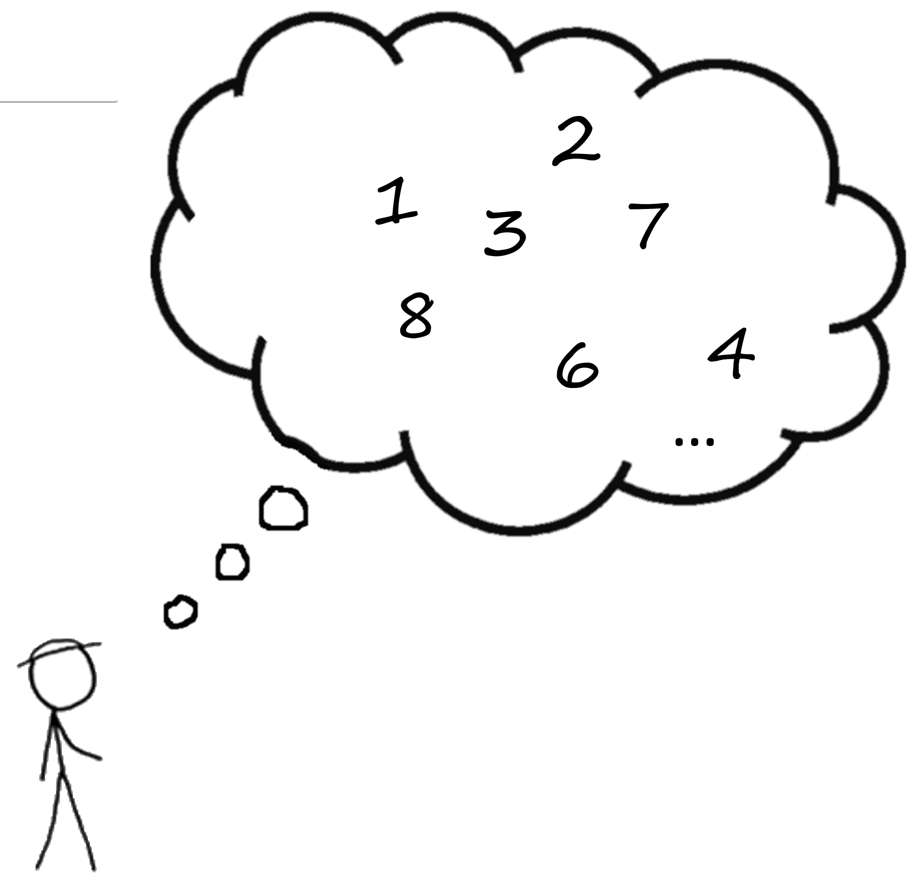


A guessing game

I am thinking of a set. Can you guess what it is?
(I will give you some clues, but I will never tell you if you are right.)

1. It is a set of positive, whole numbers.
2. It contains all but one number.
3. It contains the number 1. (Now guess.)
4. It contains the number 3. (Guess.)
5. It contains the number 4. (Guess.)
6. It contains the number 2. (Guess.)
7. It contains the number 6. (Guess.)

…

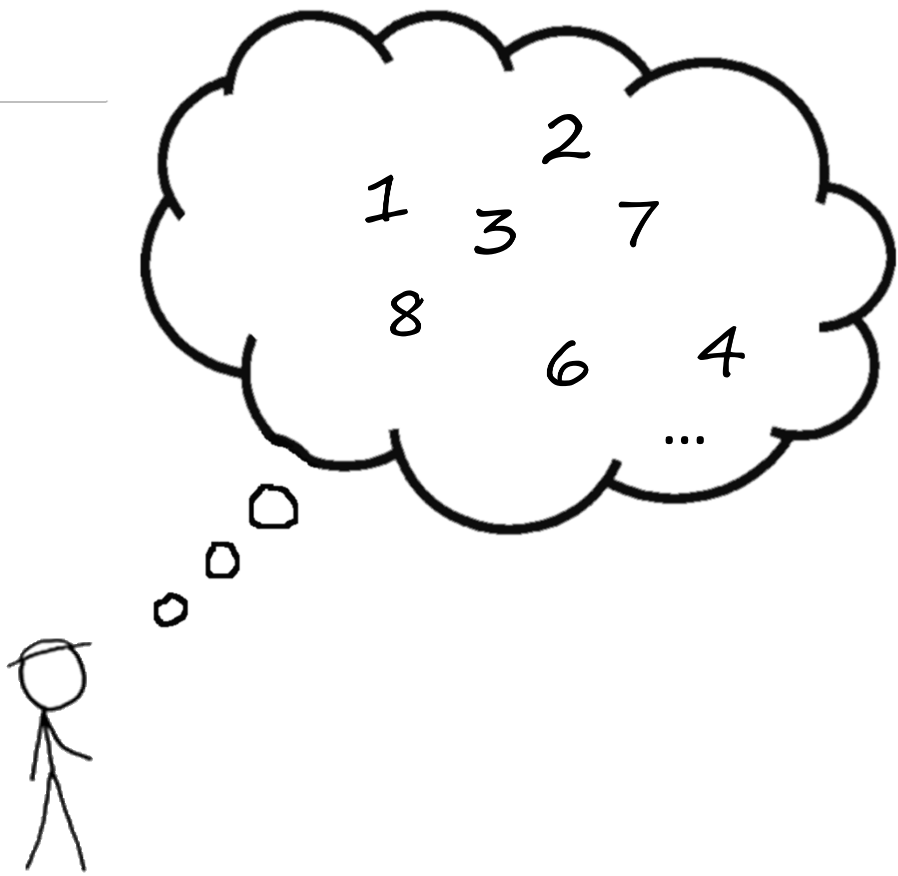


A guessing game

I am thinking of a set. Can you guess what it is? (I will give you some clues, but I will never tell you if you are right.)

Now some questions:

1. Are you confident about your latest guess? What is a possible next clue that would lead you to repeat your guess? To change it?
2. Were your guesses just random or were they made according to some "guessing strategy" that you can formulate?
3. What should count as "winning" this game?

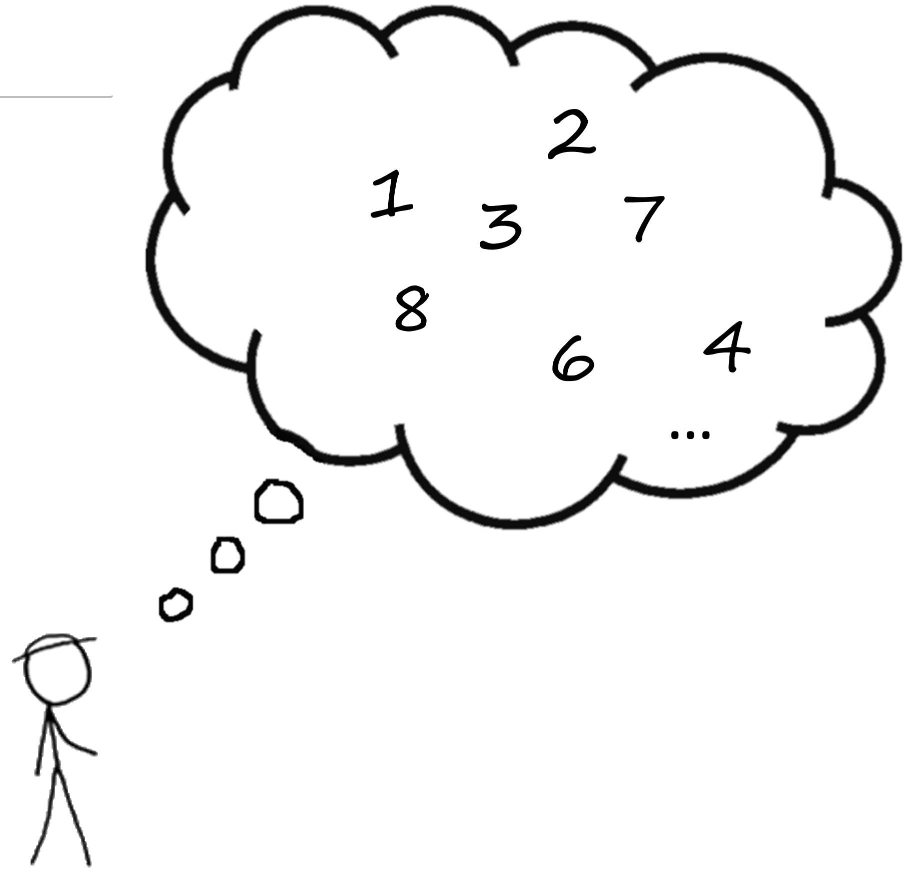


A guessing game

I am thinking of a set. Can you guess what it is? (I will give you some clues, but I will never tell you if you are right.)

Now some questions:

4. Say “winning” means that you eventually guess right, and then never change your guess after that (you win *in the limit*). Is it possible to win in the limit even after 100 wrong guesses?
5. Can you come up with a guessing strategy that will always win in the limit?



Another guessing game

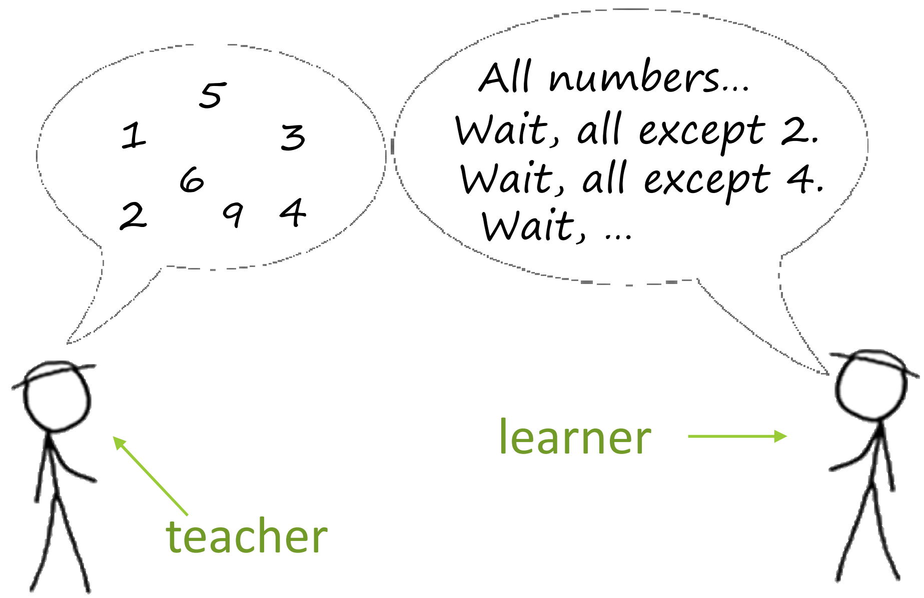
The *teacher* thinks of a set that is either

- All the whole numbers, or
- All but one of the whole numbers.

The teacher gives clues to the *learner* as in the previous game, and the learner has to try to guess the set.

Same criterion for winning --- eventually guess right *in the limit*, i.e., from some point on.

Question: Is there a winning strategy for the learner?



Yet another guessing game

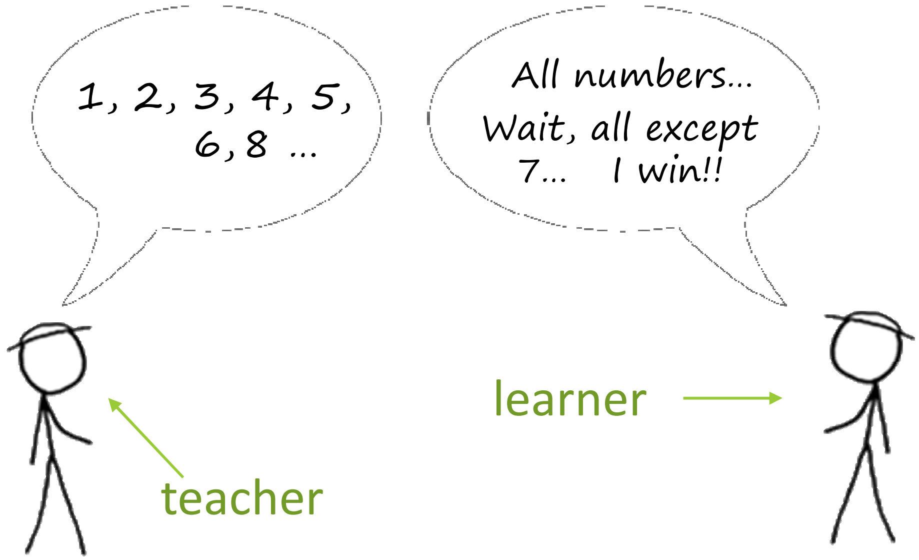
The *teacher* thinks of a set that is either

- All the whole numbers, or
- All but one of the whole numbers.

The teacher gives clues to the *learner* as in the previous game, and the learner has to try to guess the set.

Additional rule: If *i* and *j* are both in the set, and *i* is less than *j*, then the clue ***The set contains i*** will come before the clue ***The set contains j****.*

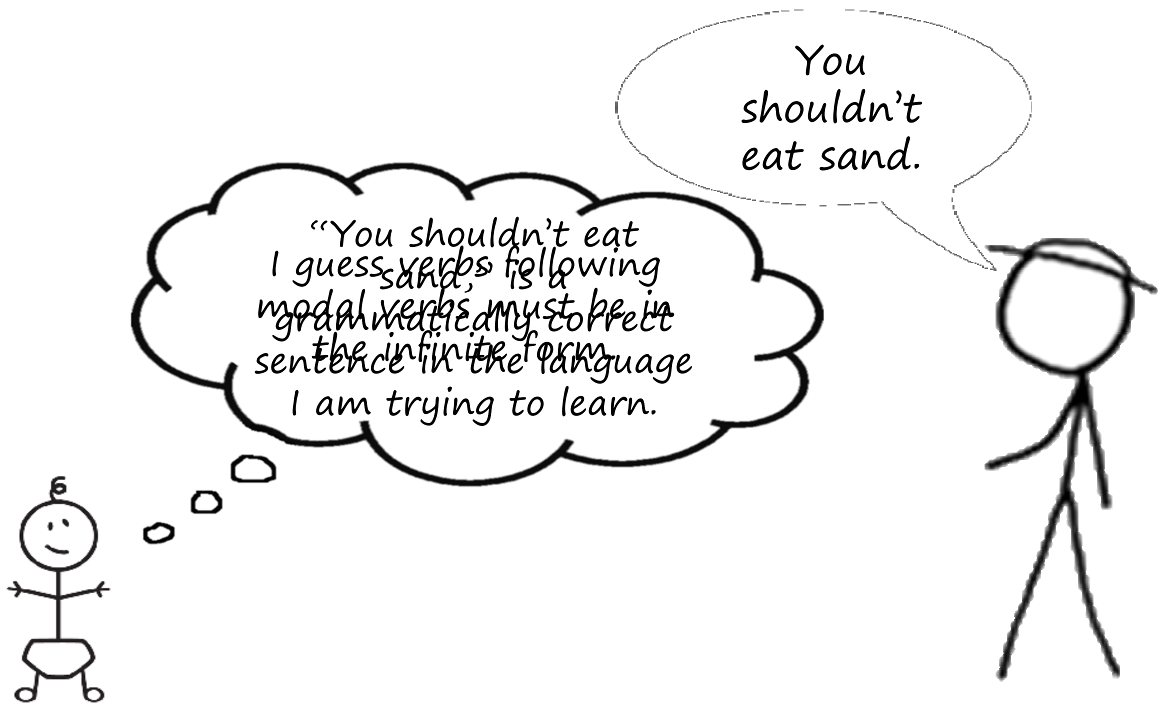
Question: Is there a winning strategy for Player II in this game?



Learning Languages

A child learning a language is playing the role of the learner in a more complex variant of this game.

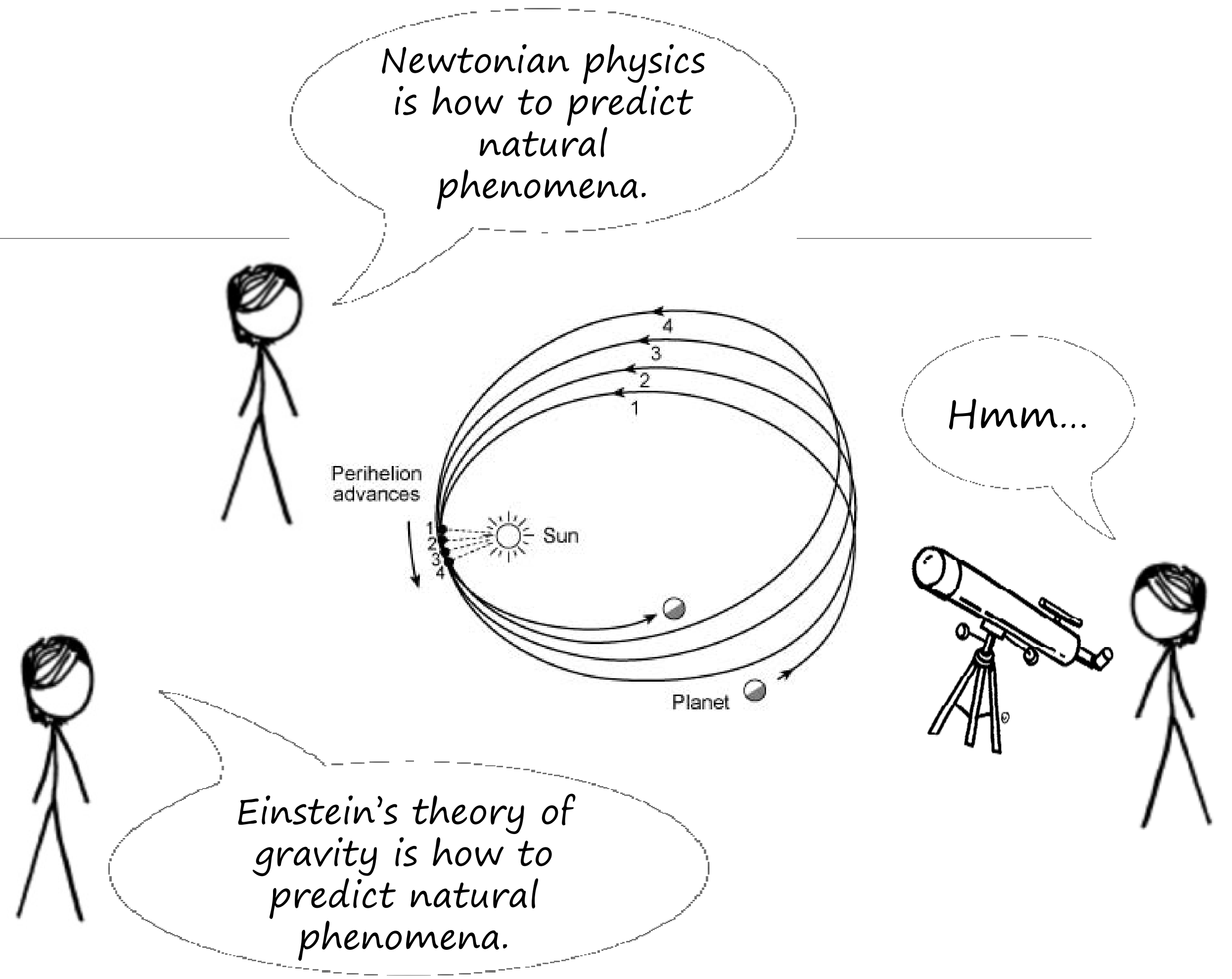
- A language is a set of grammatically correct sentences.
- The child will be given clues in the form of sentences that are in that set.
- The child will try to figure out what is in that set (by guessing the grammatical rules of the language).



Doing science

A scientist trying to understand Nature is a learner in a game like this.

- We can think of Nature as the set of all natural phenomena.
- The scientist tries to figure out how to predict natural phenomena.
- Nature provides examples of phenomena, but never “tells” the scientist if she’s got it right.



Formalization of Learning

We want to formalize learning in a tractable way so that we can study it and prove things about it. We'll take as the underlying motivating scenario a child learning a language in an idealized setting.

A *learning paradigm* consists of the following:

- A learner.
- A thing to be learned.
- A teacher or presentation of the thing to be learned.
- The hypotheses about the thing to be learned postulated by the learner.
- A criterion for success or *identification*.

Formalization of Learning

Human language is messy and full of words. We are going to make some simplifying assumptions so that we can make a rigorous study. Here are two of the big ones:

1. a language ≡ a set of natural numbers

2. rules of grammar ≡ computer program

Language = set of numbers

How does this work? Let's just start with English and do some encoding of just words…

- We can assign to each letter of the alphabet a natural number:

$$\mathrm{A}\to 0,\ \mathrm{B}\to 1,\ \mathrm{C}\to 2,\ \text{etc.}$$

- So, each word can be written as a string of numbers:

$$\mathrm{HELLO} \to 7,\ 4,\ 11,\ 11,\ 14$$

- We can use the fundamental theorem of arithmetic to turn this string of numbers into a single unique number representing that word:

$$7,\ 4,\ 11,\ 11,\ 14 \to 2^7 \cdot 3^4 \cdot 5^{11} \cdot 7^{11} \cdot 11^{14} \sim 3.801 \times 10^{35}$$

Language = set of numbers

Now, sentences:

- We have a number for each word.

$$\langle \mathrm{HELLO} \rangle \sim 3.801 \times 10^{35}$$
$$\langle \mathrm{WORLD} \rangle \sim 4.028 \times 10^{37}$$

- Now, we can use the same FTA trick to construct sentences:

$$\mathrm{HELLO\ WORLD} \rightarrow 2^{\langle HELLO \rangle} \cdot 3^{\langle WORLD \rangle} \sim \text{humongous}$$

- So each sentence* has a unique number assigned to it via this scheme.
- The collection of grammatically correct sentences in the English language is a set of numbers.

* in fact, any string of letters and spaces…

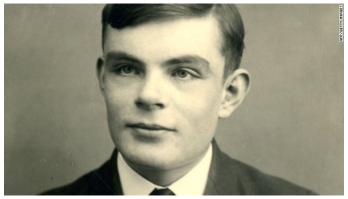
Grammar = Program

The motivation for this assumption is the following:

- Real languages aren't just arbitrary strings of symbols in some alphabet.
- There are rules of grammar and spelling, for example.
- *Learning a language* doesn't mean memorizing all the grammatically correct sentences; this is impossible anyway --- languages (as sets of sentences) are infinite!
- *Learning a language* means learning the rules of grammar, vocabulary, and spelling so that sentences can be constructed to express what we feel, think, want to ask, etc.
- We really only want to think about how to learn what might be a "real" language by figuring out the rules of the grammar that govern that language.

To completely understand this assumption, we will need to make a foray into the world of *computability theory*.

Turing Machines

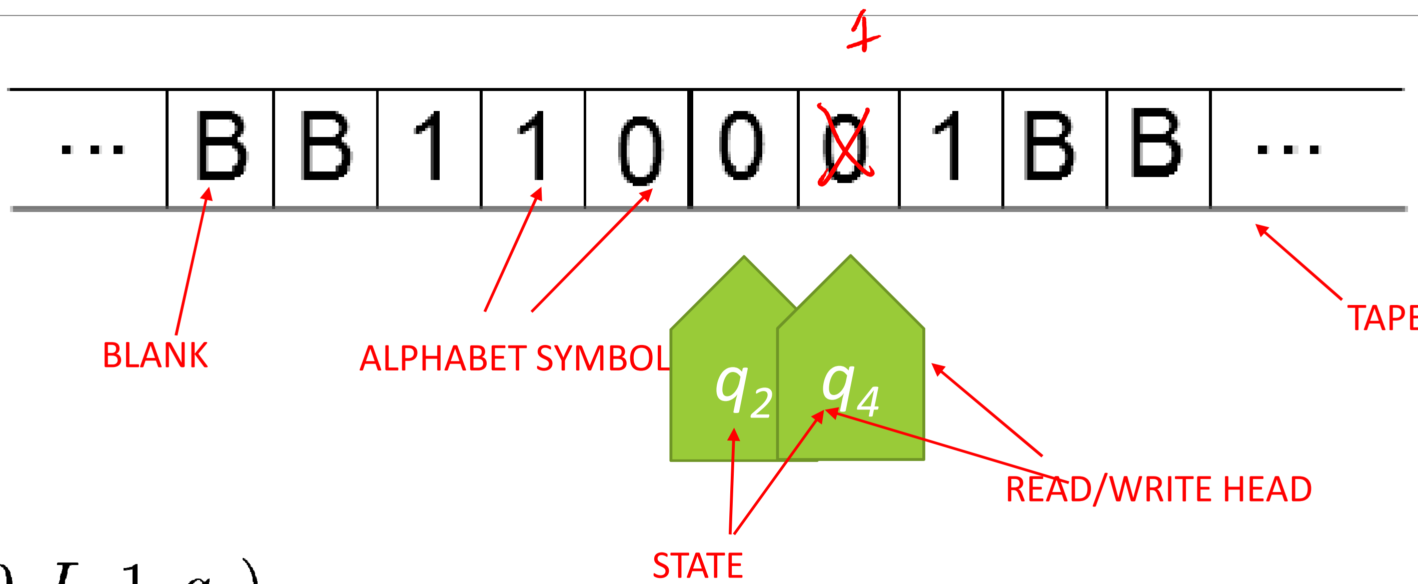


A *Turing Machine* is a theoretical computer formulated by Alan Turing in the 1930's.

- It has a bi-infinite tape of cells, almost all of which are blank. Those that are not blank contain symbols from some finite set of symbols (or *alphabet*), $\mathcal{A}$.
- It has a movable "read/write head" capable of reading the contents of a tape cell and writing a symbol from $\mathcal{A}$ into a cell.
- At each moment, the machine head is in one of some finite collection of *states*, $\mathcal{Q}$.
- The operation of the machine is governed by a finite set of instructions called *quints*.
- A quint looks like this:

$$(q_i, s_i, M, s_f, q_f).$$

Turing Machines



QUINT

$$(q_4, 0, L, 1, q_2)$$

For the TM, the quint is an instruction: *If I am in state* q_4 *reading symbol* 0, *I will write symbol* 1 *into the cell, change my state to* q_2, *and make move* L.

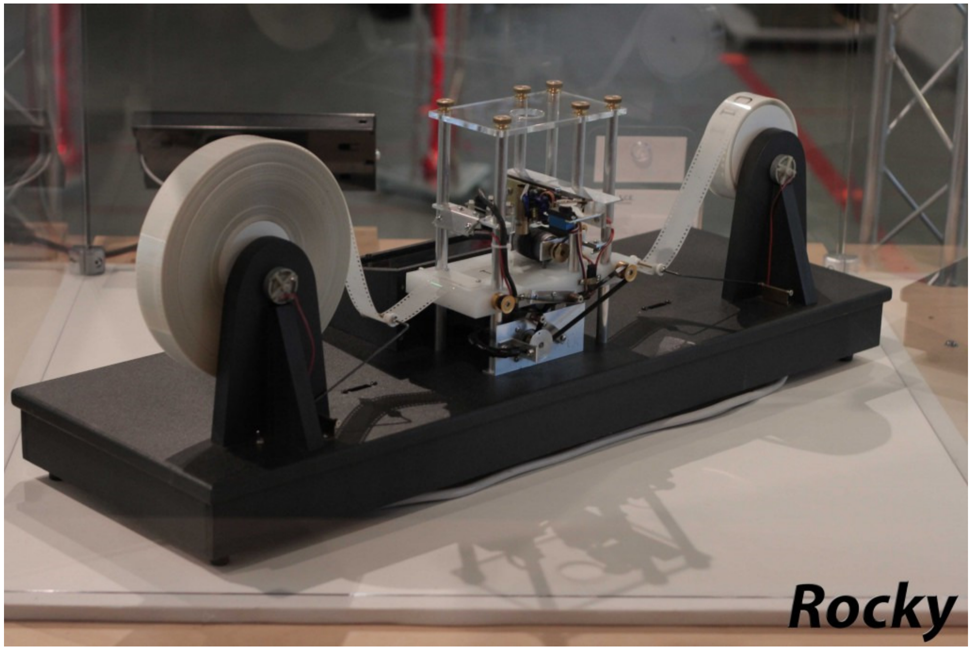
Turing Machines: the fine print

The symbols.

- There are two kinds of symbols, those from the alphabet, $\mathcal{A}$, and the (special) blank symbol, B.
- The alphabet is finite:

$$\mathcal{A} = \{s_1, s_2, \ldots, s_n\}.$$

- The input given to a TM is always some finite string of symbols from the alphabet.



Turing Machines: the fine print

The states.

- There are finitely many possible states a TM might be in at any given time. Generically, we write $\mathcal{Q} = \{q_0, q_1, q_2, \ldots, q_m, q_H\}$.
- Two of these states are special: q_0 and q_H.
- A TM is in state q_0, called the *initial state*, only at the beginning of the first step of the computation.
- The state q_H is called the *halting state*. If a TM ever enters this state, it stops (halts) and the computation is over.

The Turing Machine is the father of all computers

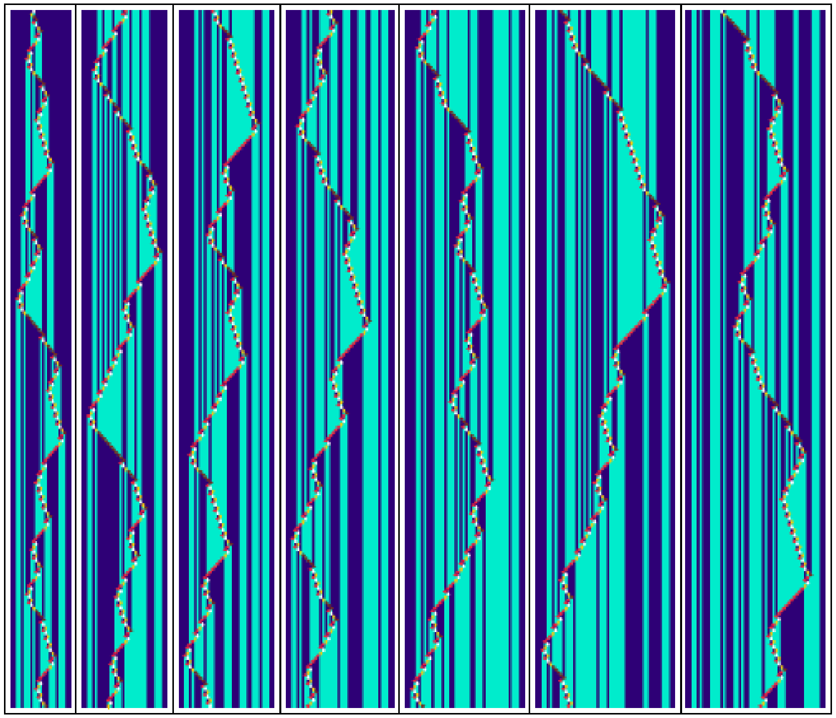
Turing Machines: the fine print

The quints.

- A TM has finitely many quints of the form

$$(q_i, s_i, M, s_j, q_j).$$

- Think: In state q_i, reading s_i, write s_j, change to state q_j, and move M.
- No quint may have q_0 as the last entry.
- No quint may have q_H as the first entry.
- No two quints may have the same first two entries.



Turing Machines: the fine print

How the computation works.

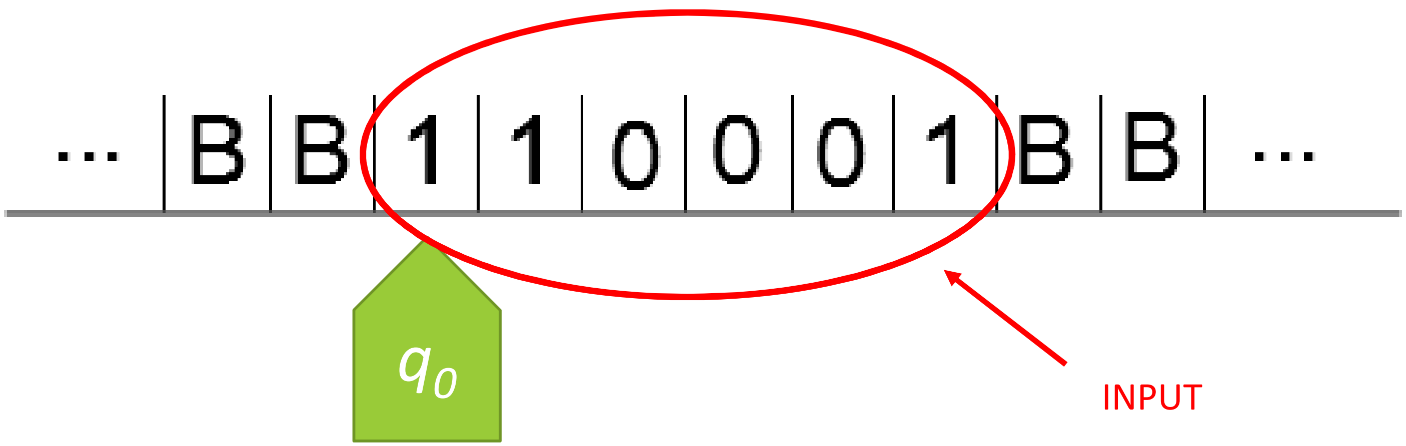
0. Initialize.
1. Check to see which (if any) quint matches the current configuration of the TM.
2. Execute the instruction in that quint, or, if no matching quint exists, the machine has crashed and the computation fails.
3. Repeat 1 and 2 as needed.
4. If the TM enters state q_H, the computation halts (successfully) and the output of the computation is whatever is on the tape.

(Patience, young Padawan… the time for examples is near.)

Turing Machines: the fine print

The initial state of a Turing Machine has

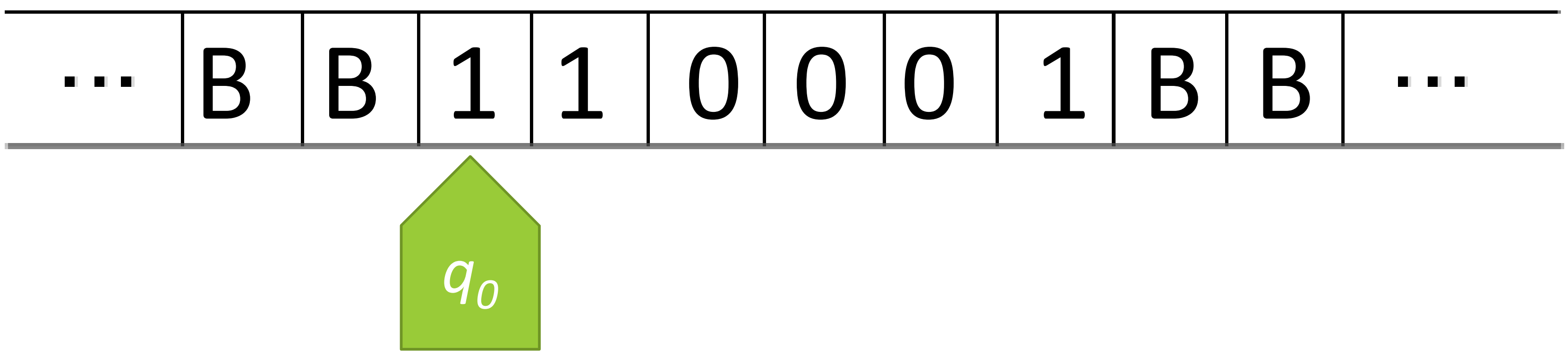
- the input written on the tape,
- the machine head at the first non-blank cell,
- the machine head in the initial state, q_0.



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

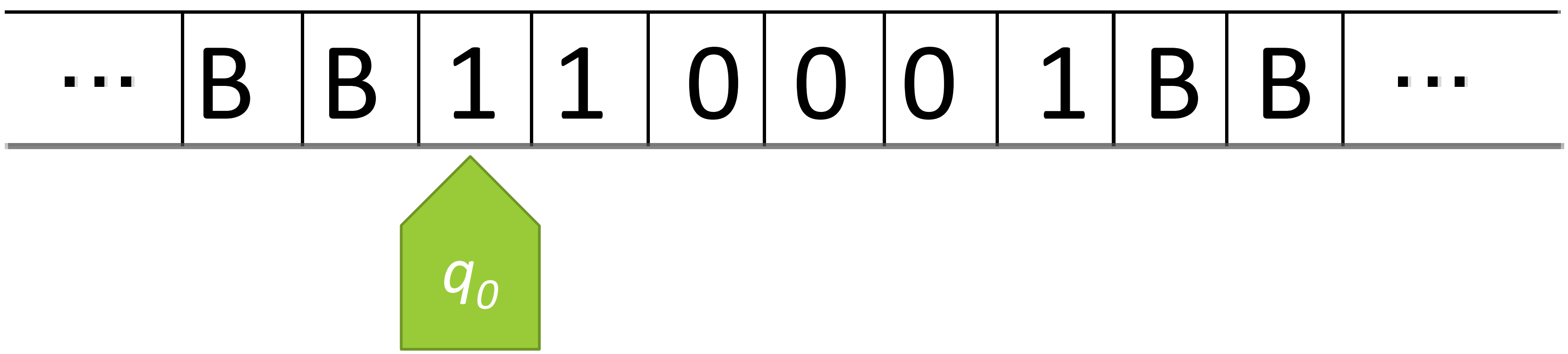
$$
\begin{array}{l}
(q_0, 0, R, 1, q_1) \\
(q_0, 1, R, 0, q_1) \\
(q_1, 0, R, 1, q_1) \\
(q_1, 1, R, 0, q_1) \\
(q_1, B, S, B, q_H)
\end{array}
$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

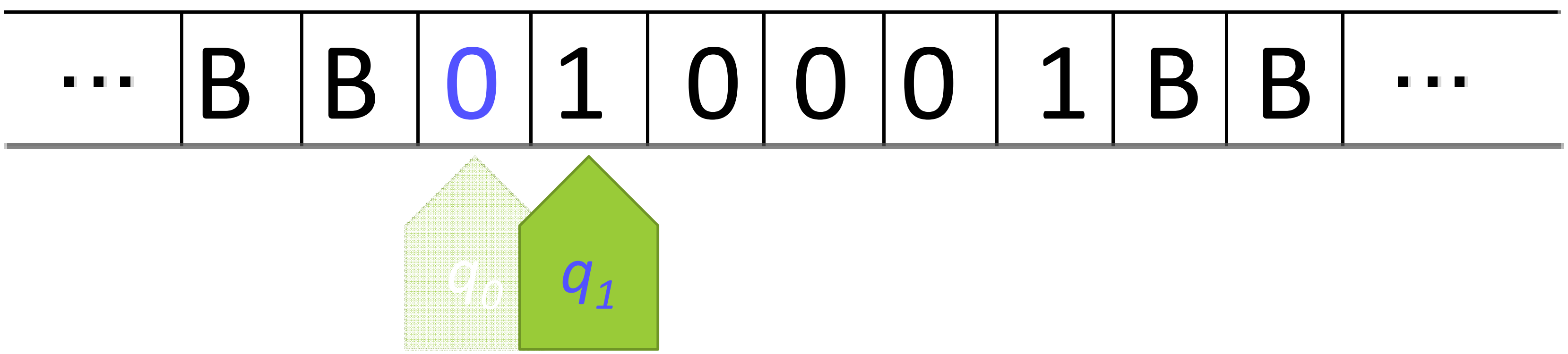
$$(q_0, 0, R, 1, q_1)$$
$$(q_0, 1, R, 0, q_1)$$
$$(q_1, 0, R, 1, q_1)$$
$$(q_1, 1, R, 0, q_1)$$
$$(q_1, B, S, B, q_H)$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

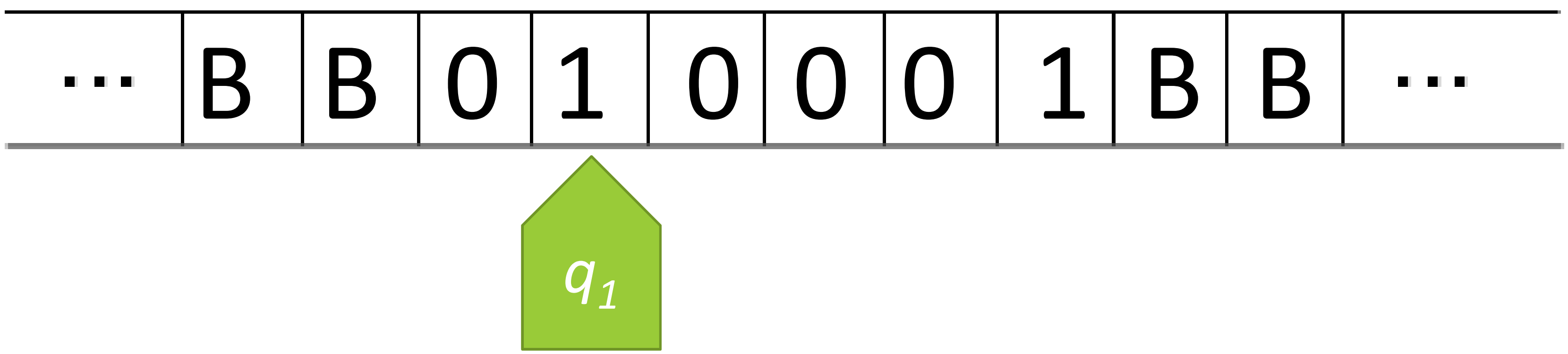
$$
\begin{array}{l}
(q_0, 0, R, 1, q_1) \\
(q_0, 1, R, 0, q_1) \\
(q_1, 0, R, 1, q_1) \\
(q_1, 1, R, 0, q_1) \\
(q_1, B, S, B, q_H)
\end{array}
$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

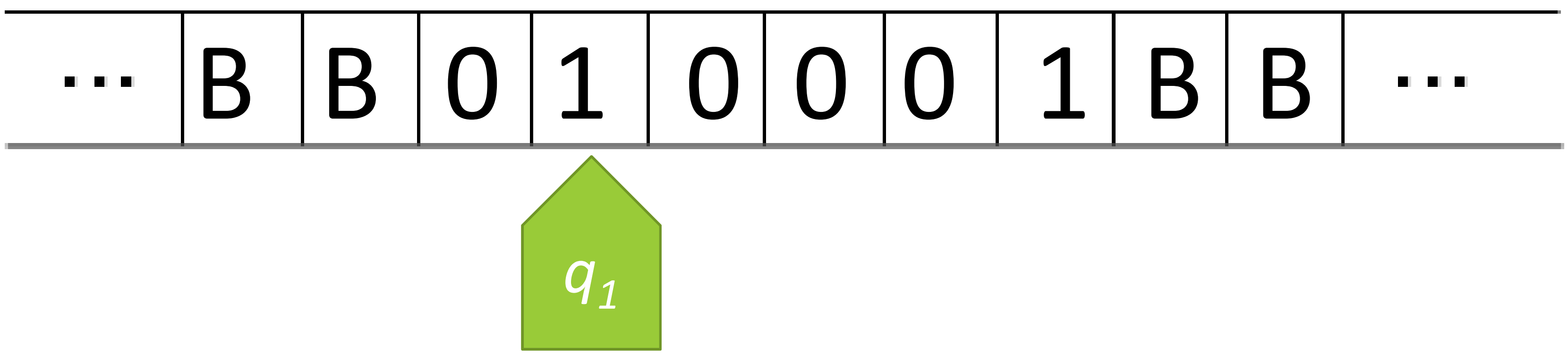
$$
\begin{array}{l}
(q_0, 0, R, 1, q_1) \\
(q_0, 1, R, 0, q_1) \\
(q_1, 0, R, 1, q_1) \\
(q_1, 1, R, 0, q_1) \\
(q_1, B, S, B, q_H)
\end{array}
$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

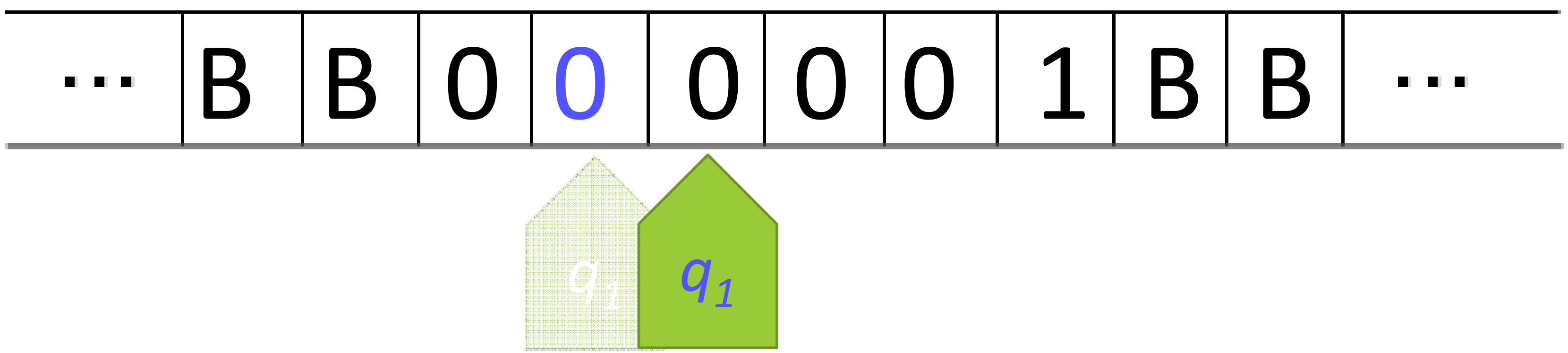
$$(q_0, 0, R, 1, q_1)$$
$$(q_0, 1, R, 0, q_1)$$
$$(q_1, 0, R, 1, q_1)$$
$$(q_1, 1, R, 0, q_1)$$
$$(q_1, B, S, B, q_H)$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

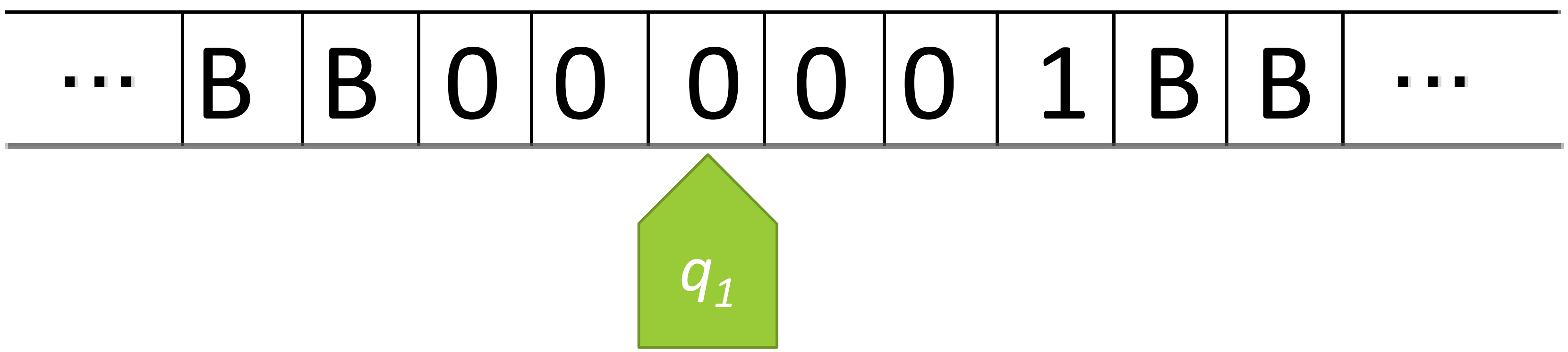
$$(q_0, 0, R, 1, q_1)$$
$$(q_0, 1, R, 0, q_1)$$
$$(q_1, 0, R, 1, q_1)$$
$$(q_1, 1, R, 0, q_1)$$
$$(q_1, B, S, B, q_H)$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

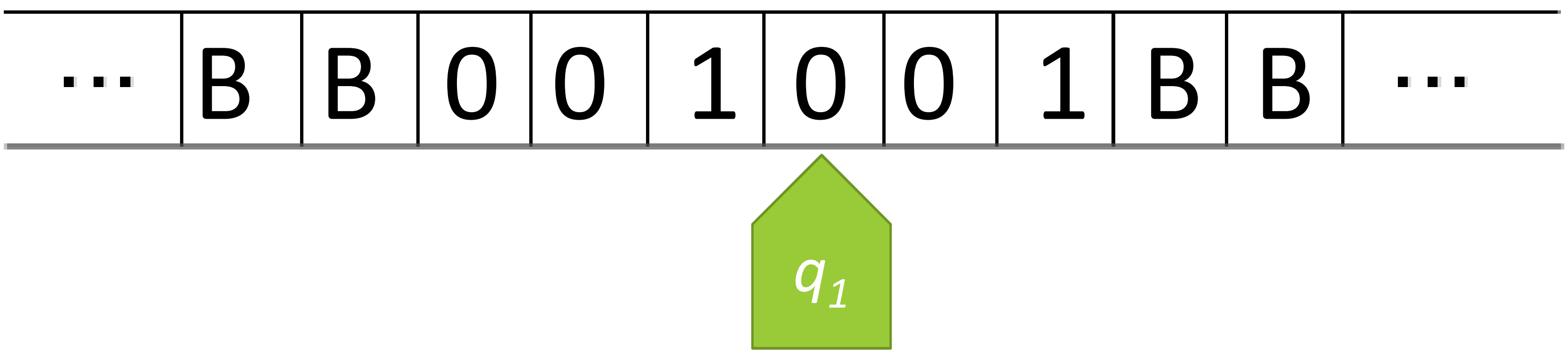
$$
\begin{array}{l}
(q_0, 0, R, 1, q_1) \\
(q_0, 1, R, 0, q_1) \\
\color{blue}{(q_1, 0, R, 1, q_1)} \\
(q_1, 1, R, 0, q_1) \\
(q_1, B, S, B, q_H)
\end{array}
$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

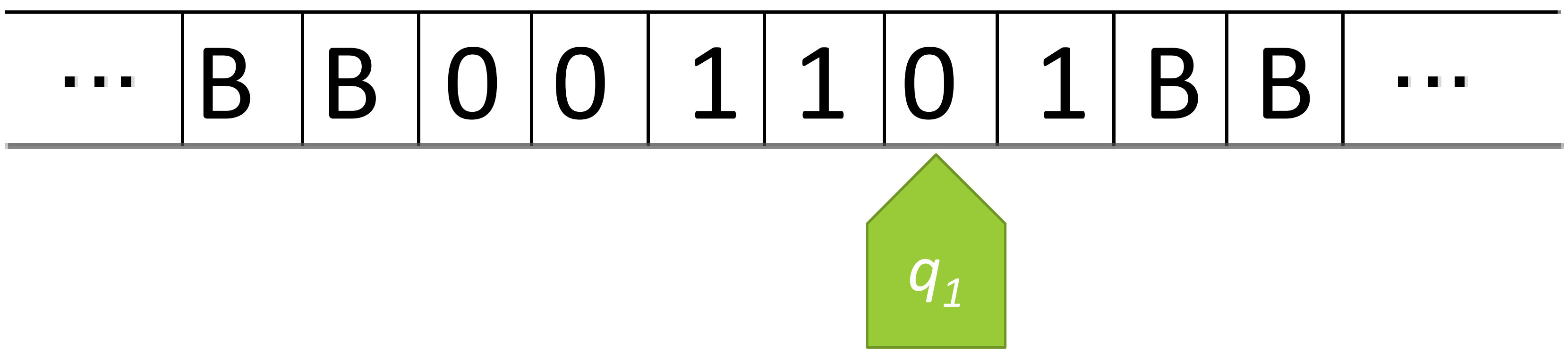
$$
\begin{array}{l}
(q_0, 0, R, 1, q_1) \\
(q_0, 1, R, 0, q_1) \\
(q_1, 0, R, 1, q_1) \\
(q_1, 1, R, 0, q_1) \\
(q_1, B, S, B, q_H)
\end{array}
$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

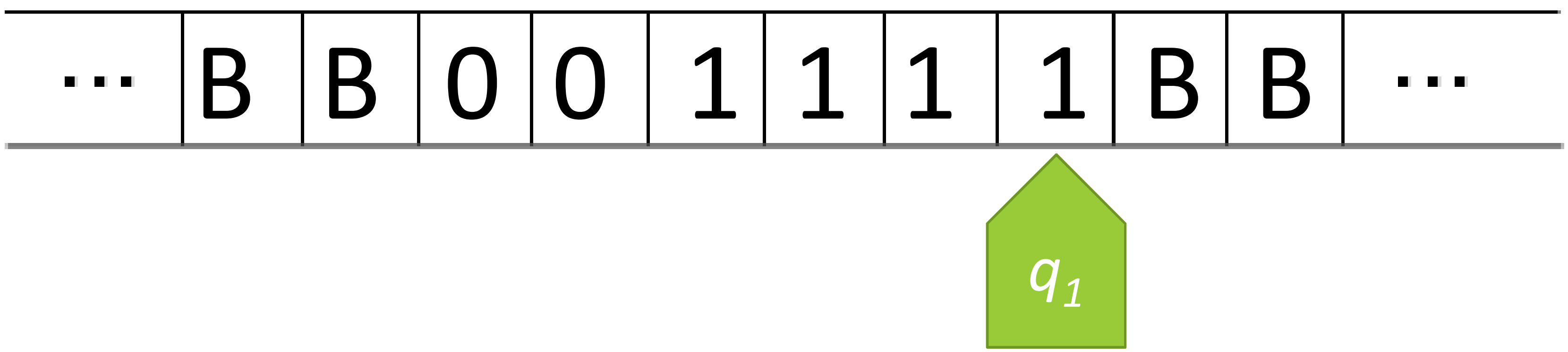
$$
\begin{aligned}
&(q_0, 0, R, 1, q_1)\\
&(q_0, 1, R, 0, q_1)\\
&{\color{blue}(q_1, 0, R, 1, q_1)}\\
&(q_1, 1, R, 0, q_1)\\
&(q_1, B, S, B, q_H)
\end{aligned}
$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

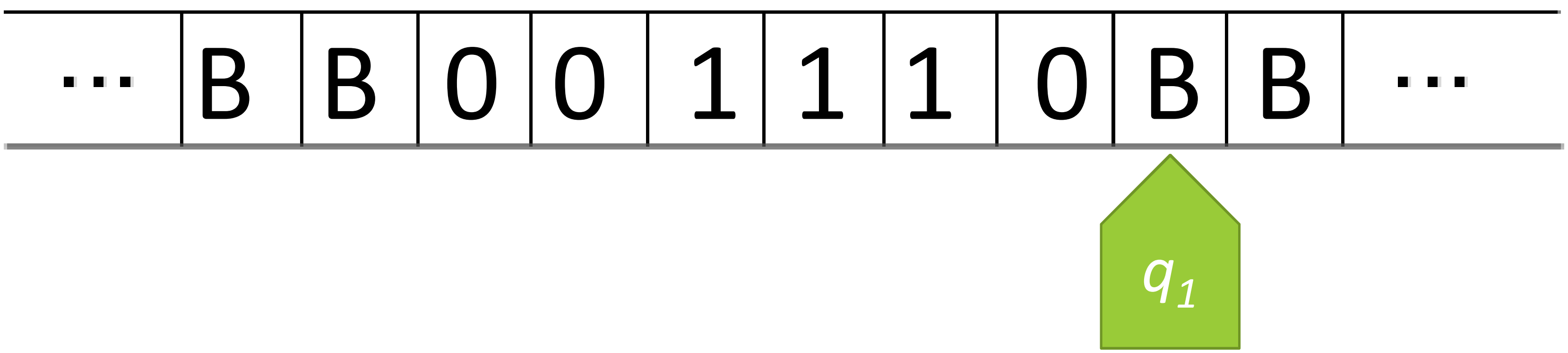
$$(q_0, 0, R, 1, q_1)$$
$$(q_0, 1, R, 0, q_1)$$
$$(q_1, 0, R, 1, q_1)$$
$$(q_1, 1, R, 0, q_1)$$
$$(q_1, B, S, B, q_H)$$



Turing Machines: example 1

$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

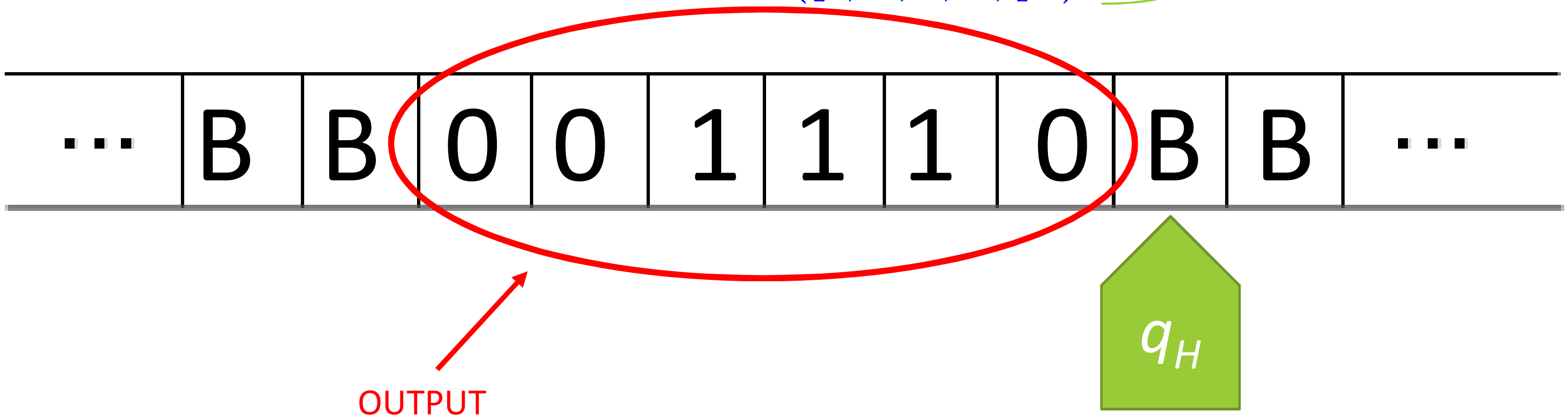
$$
\begin{array}{l}
(q_0, 0, R, 1, q_1) \\
(q_0, 1, R, 0, q_1) \\
(q_1, 0, R, 1, q_1) \\
(q_1, 1, R, 0, q_1) \\
{\color{blue}(q_1, B, S, B, q_H)}
\end{array}
$$



Turing Machines: example 1

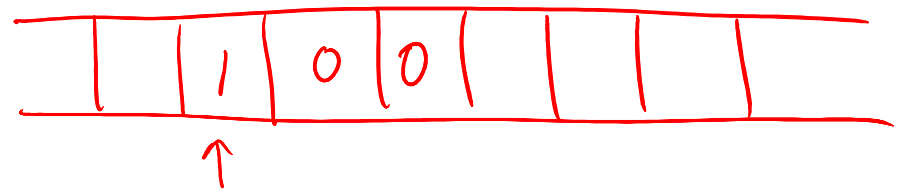
$\mathcal{A} = \{0, 1\}$ and $\mathcal{Q} = \{q_0, q_1, q_H\}$. The quints:

$$\left.\begin{array}{l} (q_0, 0, R, 1, q_1) \\ (q_0, 1, R, 0, q_1) \\ (q_1, 0, R, 1, q_1) \\ (q_1, 1, R, 0, q_1) \\ \color{blue}{(q_1, B, S, B, q_H)} \end{array}\right\} \text{Bit-flip program}$$



Turing Machines: example 2

Here are the details on another machine:



- $\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\mathcal{Q} = \{q_0, q_1, q_2, q_H\}$
- Here are the quints:

$$
\begin{array}{lr}
(q_0, s, R, s, q_1), & s \in \mathcal{A} \\
(q_1, s, R, s, q_1), & s \in \mathcal{A} \\
(q_1, B, L, B, q_2) & \\
(q_2, t, S, t+1, q_H), & t \in \mathcal{A}, t \neq 9 \\
(q_2, 9, L, 0, q_2) & \\
(q_2, B, S, 1, q_H) &
\end{array}
$$

Turing Machines: example 3

Here are the details on one more machine:

- $\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\mathcal{Q} = \{q_0, q_1, q_2, q_H\}$
- Here are the quints:

$$
\begin{array}{lr}
(q_0, s, R, s, q_1), & s \in \mathcal{A} \\
(q_1, s, R, s, q_1), & s \in \mathcal{A} \\
(q_1, B, L, B, q_2) & \\
(q_2, t, L, 1, q_3), & t = 0, 2, 4, 6, 8 \\
(q_2, t, L, 0, q_3), & t = 1, 3, 5, 7, 9 \\
(q_3, s, L, B, q_3), & s \in \mathcal{A} \\
(q_3, B, S, B, q_H) &
\end{array}
$$

Run this machine on input 253.
What is the output?

Run this machine on input 46.
What is the output?

Pick another number and try it…

What is the machine doing?

Turing Machines: example 4

Here are the details on one more machine:

- $\mathcal{A} = \{a, b\}$
- $\mathcal{Q} = \{q_0, q_1, q_H\}$
- Here are the quints:

$$
\begin{array}{l}
(q_0, a, S, a, q_H) \\
(q_0, b, S, b, q_1) \\
(q_1, b, S, b, q_1) \\
(q_1, B, S, B, q_H)
\end{array}
$$

What is the behavior of this machine?

--Try it on “bba”
--Try it on “abba”
--Try some more words…

What’s it doing?

TMs as functions

We can think of a TM as defining a function on natural numbers. We say that the TM *computes* the function.

- The TM in Example 2 computes $f(n) = n + 1$
- The TM in Example 4 is weird. It computes a *partial function*:
- The TM in Example 3 computes $g(n) = \begin{cases} 1 & n \text{ is even} \\ 0 & n \text{ is odd} \end{cases}$

$$f(w) = \begin{cases} w & w \text{ begins with } a \\ \uparrow & w \text{ begins with } b \end{cases}$$

Function is undefined.

For the rest of our discussion, we'll only consider TMs on numerical alphabets defining functions on the natural numbers.

Indices of programs

Each one of these machines is described by its set of quints.

Each quint is, in the end, just a string of symbols.

We can use the same coding trick we used for words and sentences to encode a Turing Machine as a single number!

This can be done carefully so that each program has a number, and each number corresponds to a (perhaps nonsensical) TM.

The number that encodes a particular TM in this way is called the *index* or *Gödel code* of that program.

The index of Turing machine M is denoted $\langle M \rangle$.

TMs as functions

We can think of a TM as defining a function on natural numbers. We say that the TM *computes* the function.

Ex: TM 2 COMPUTED $f(n) = n+1$.

$\langle TM2 \rangle$ IS AN INDEX FOR $f(n)=n+1$.

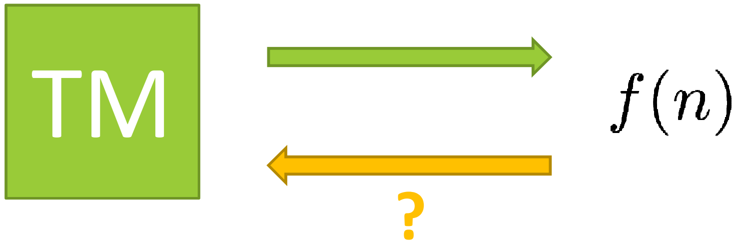
If $f(n)$ is a function on $\mathbb{N}$, and M is a TM that computes $f(n)$, then $\langle M \rangle$ is *an index for the function* $f(n)$.

Natural numbers

The index of the TM.

TMs as functions

1. Does every TM define a function?
2. Does every function have a TM?
3. For the functions that have TMs, are they unique?



Answer to question 1: Yes! (Though possibly not a total function.)
Answer to question 2: No! (We'll come back to this.)
Answer to question 3: No!

Q3: Are TMs unique? No.

This TM computes the function $f(n) = n + 1$.

- $\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\mathcal{Q} = \{q_0, q_1, q_2, q_H\}$
- Here are the quints:

Dummy quint.

$$
\begin{array}{ll}
(q_5, 0, S, 4, q_7) & \\
(q_0, s, R, s, q_1), & s \in \mathcal{A} \\
(q_1, s, R, s, q_1), & s \in \mathcal{A} \\
(q_1, B, L, B, q_2) & \\
(q_2, t, S, t+1, q_H), & t \in \mathcal{A}, t \neq 9 \\
(q_2, 9, L, 0, q_2) & \\
(q_2, B, S, 1, q_H) &
\end{array}
$$

Theorem. *If a function can be computed by a TM, then there are infinitely many TMs that compute that function.*

Q2. Does every function have a TM? No.

- Every TM defines a function.
- The function computed by the TM with index i is denoted φ_i.

FOR EXAMPLE:
$e = \langle TM2 \rangle$ IS AN INDEX FOR $f(n) = n+1$.
$\varphi_e(n) = n+1$.

Consider the function $f(x) = \begin{cases} 0, & \varphi_x(x)\uparrow \\ y+1, & \varphi_x(x)\downarrow = y \end{cases}$

Cannot be $=$ TO φ_e FOR ANY $e \in \mathbb{N}$.

Fact. Not all functions can be computed by a TM.

Question: Is there an e so that $f = \varphi_e$?

$f(e) \neq \varphi_e(e)$

What functions are defined by TMs?

Any function that can be computed on any computer of any kind can be computed by a TM.

Church-Turing Thesis. *The class of functions that can be computed by a TM is the same as the class of "effectively calculable" functions.*

This is very handy… it means that to demonstrate that a function is computable by a TM, all we have to do is give an algorithm for computing it, we don't have to actually find a set of quints to do it!

What functions are defined by TMs?

The following function can be computed by a TM:

$$g(n) = n^2$$

By the CT Thesis, all we need to do is explain how to square a number in an *effective* (i.e., *procedural*) way.

1. `Input` n.
2. `If` $n = 0$ `output 0.`
3. `Set` $c = 0$ `and` $s = 0$.
4. `If` $c < n$, `set` $s = s + n$ `and` $c = c + 1$, `and repeat this step.`
5. `Otherwise, output` s.

What functions are defined by TMs?

The following function can be computed by a TM:

$$p(n) = \begin{cases} 1, & n \text{ is prime} \\ 0, & \text{otherwise} \end{cases}$$

By the CT Thesis, all we need to do is explain how to check if a number is prime in an *effective* way, i.e., we basically need an algorithm for checking primality.

1. `Check if` n `is divisible by 2.`
2. `Check if` n `is divisible by 3.`
3. `Check if` n `is divisible by 4.`
4. $\cdots$ up to $n-1$.
5. Output 0 if a divisor is found, 1 otherwise.

1. `SET` $d = 2$.
2. `IF` n `is divisible by` d, `OUTPUT 0.`
3. `OTHERWISE SET` $d = d + 1$.
4. `IF` $d < n$ `GOTO 2.`
5. `OTHERWISE OUTPUT 1.`

What functions are defined by TMs?

The following function can be computed by a TM:

$$h(n) = \begin{cases} \text{the largest prime smaller than } n, & n > 2 \\ \uparrow, & n \leq 2 \end{cases}$$

1. If $n \leq 2$, loop forever.
2. Check if $n-1$ is prime.
3. Check if $n-2$ is prime.
4. Check if $n-3$ is prime.
5. $\cdots$ down to 2.
6. Output first prime found.

1. IF $n \leq 2$, GOTO 1.
2. SET $k = n$.
3. IF $k-1$ is prime, OUTPUT $k-1$.
4. OTHERWISE SET $k = k-1$ and GOTO 3.

What functions are defined by TMs?

Definition: A function is called *partially computable* if can be computed by a TM.

Definition: A function is called *computable* if it is *total* and can be computed by a TM.

All the functions we've seen are partially computable.

Computable

$f(n) = n + 1$, $g(n) = n^2$, $p(n) = \begin{cases} 1, & n \text{ is prime} \\ 0, & \text{otherwise} \end{cases}$, and

Partially computable

$$h(n) = \begin{cases} \text{the largest prime smaller than } n, & n > 2 \\ \uparrow, & n \leq 2 \end{cases}$$

Which functions are partially computable?

CAN BE COMPUTED BY A TM (OR ANY COMPUTER PROGRAM)

- Basically all natural functions on $\mathbb{N}$ are at least partially computable.
- Thanks to Church's thesis, we can be relatively informal about proving things are partially computable.

1. $f(n) = 3n^3 - 2n + 1.$
2. $h(n) = \begin{cases} n^2, & n \equiv 1 \mod 3 \\ 0, & \text{otherwise} \end{cases}$ n÷3 HAS REMAINDER 1.
3. $g(n) =$ the smallest positive integer root of $x^{11} - 4x^3 + 2n$. $g(2) =$ smallest… of $x^{10} - 4x^3 + 4$
4. $K(n) = \begin{cases} 1 \text{ if } \varphi_n(n)\downarrow. \\ \uparrow \text{ o.w.} \end{cases}$