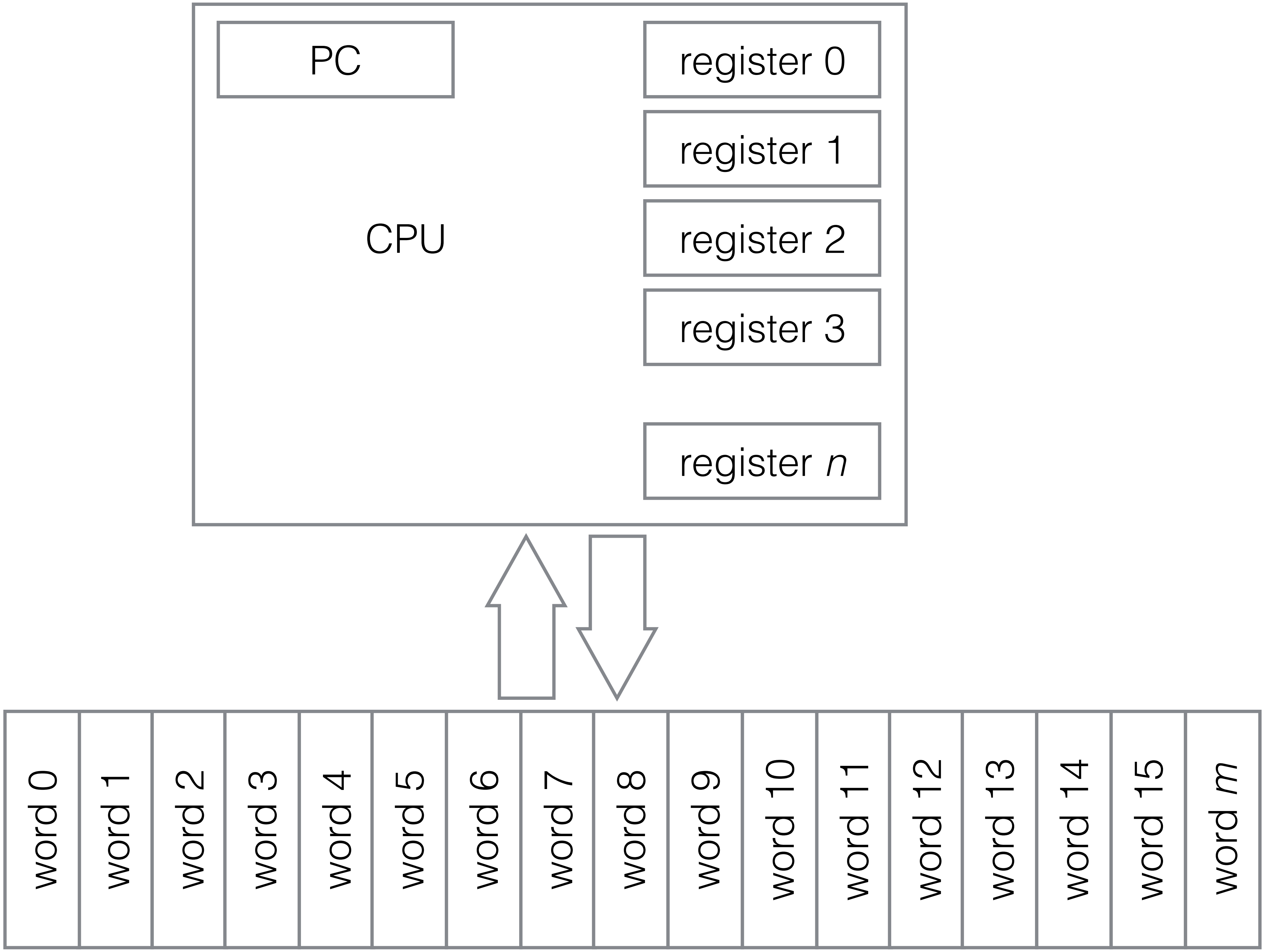
Turing Machines & Computability

02-201 / 02-601

The Conceptual Architecture of a real Computer



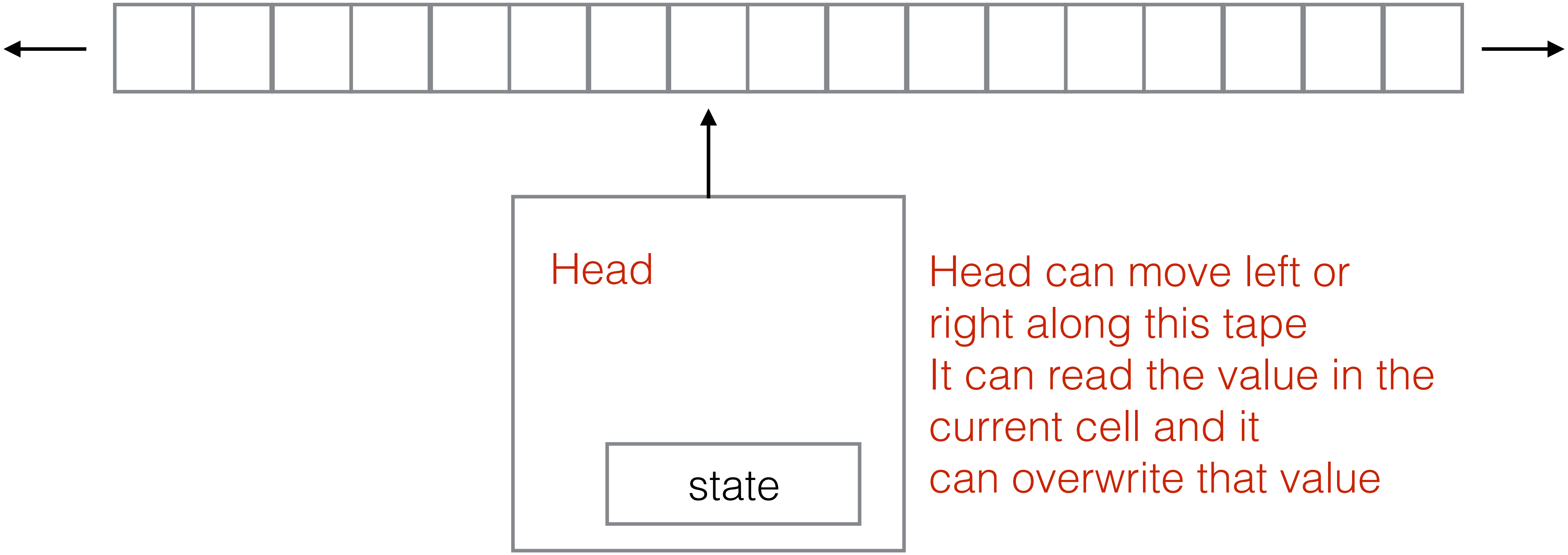
What kinds of problems can this computer solve?

What kinds of problems can't it solve?

The field of computational complexity tries to answer these questions

Turing Machines

Real computers are hard to work with mathematically, so Alan Turing proposed a simplified model:



Infinitely long *tape* of cells:

Cells can hold 0 or 1

Head can move left or right along this tape
It can read the value in the current cell and it
can overwrite that value

Head contains a single register called the “state”

A Turing Machine Program:

```
for true {
    if symbol == b && state == X {
        write [0 or 1]
        set state to Y
        move [left or right]
    } else

    if symbol == b && state == X {
        write [0 or 1]
        set state to Y
        move [left or right]
    } else

                    .
                    .
                    .

    if symbol == b && state == X {
        write [0 or 1]
        set state to Y
        move [left or right]
    } else

    if state == “HALT” {
       stop
    }
 }
```

- A Turing Machine repeatedly executes steps.
- At each step, what the machine does depends only on the symbol *b* on the current tape position and the value of the state register.
- The machine then:
    - writes a value to the current cell,
    - changes the value of the state variable to something, and
    - then moves left or right.

Another View

A Turing Machine program is a function:

$$t : (\underset{\substack{\uparrow\\ \text{Set of possible}\\ \text{states (finite)}}}{S}, \overset{\substack{\text{Symbol on}\\ \text{current position}\\ \downarrow}}{\{0, 1\}}) \rightarrow (\underset{\substack{\uparrow\\ \text{Set of possible}\\ \text{states (finite)}}}{S}, \overset{\substack{\text{Symbol}\\ \text{to write}\\ \downarrow}}{\{0, 1\}}, \overset{\substack{\text{Which}\\ \text{way to}\\ \text{move}\\ \downarrow}}{\{L, R\}})$$

For example t(7, 0) might equal (7, 0, R), meaning stay in state 7, write 0 on the tape, and move one cell to the right

Representing TM functions

$$t : (S, \{0,1\}) \rightarrow (S, \{0,1\}, \{L, R\})$$

We can represent *t* as a long (but finite) string of bits:

`0,0:0,0,L;0,1:1,1,R;2,1:10,1,L …`

↓

`string of bits representing these characters`

Turing Machines Can Solve Anything A Real Computer Can

- We won’t prove this, but intuitively it should be clear that a TM can do anything a real computer can.
- Both have finite state (registers in a computer, the “state” in a TM)
- Both have memory (RAM in a computer, the tape in a TM)
- The TM is slower because it doesn’t have random access, but if the TM wants to access cell *i*, it just has to move left or right until it is at cell *i*.

Church-Turing Thesis

Everything that is efficiently computable is efficiently computable on a Turing Machine.

Notice that this is much strong than the (true) statement that Turing machines can solve anything your laptop can.

It talks about all past, current, and future kinds of computers (quantum computers, kerfloble computers (as yet uninvented), whatever…)

The Set P

You may of heard of the famous P = NP question.

P and NP are sets of computational problems

If it halts pointing at 1: the machine answers YES
If it halts pointing at 0: the machine answers NO

P is the set of **YES / NO** questions answerable in a **polynomial number of steps** on a Turing machine.

If *n* is the number of bits written on the tape when the machine is started (the input length) then the machine must halt in ≤ $p(n)$ steps for some polynomial p

The Halting Problem

Halting Problem: Given a TM *t* and an initial contents *I* on the tape determine if *t* running on *I* ever halts (i.e. reaches the "halt" state within a finite number of steps)

- We can represent *t* as a long (but finite) string of bits.
- We can represent any Go program as a long (but finite) string of bits.

- The Halting Problem is itself a computational problem: it's input is a description of *t* and its output is YES or NO.
- It's a super important one: Will my program hang? Will it get stuck and never stop?

Is the Halting Problem in P?

The Halting Problem is Uncomputable

There is NO program that can solve the halting problem.

No matter how much finite time you give your computer, you can’t write a program that will check whether an arbitrary program will stop.

Let’s prove that now.

Proof

Suppose there *is* a program that solves the halting problem: `halt(t, I)`

Consider all the possible inputs of halt:

All the possible inputs I to a TM
(an infinite, but countable, number of them)

All the possible programs t

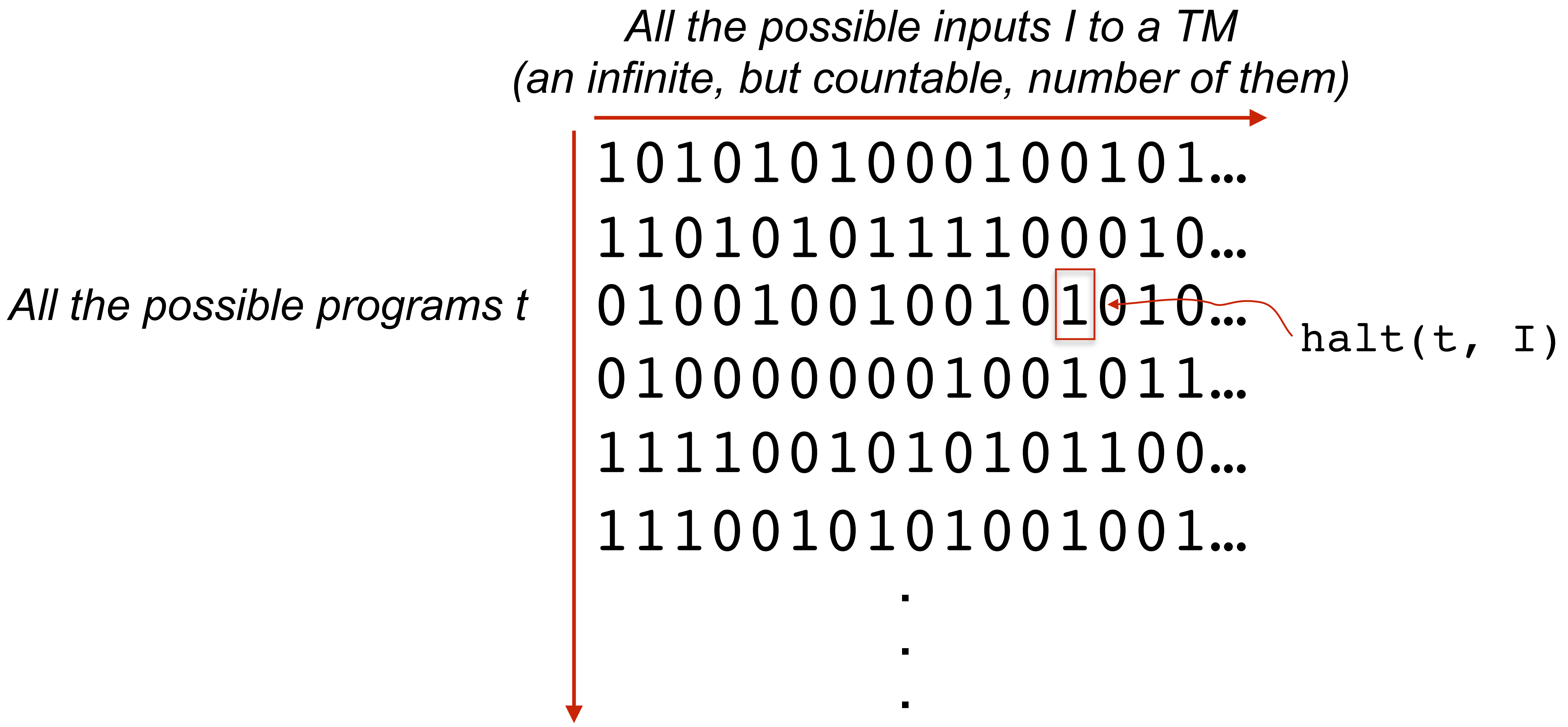
1010101000100101…
1101010111100010…
0100100100101010…
0100000001001011…
1111001010101100…
1110010101001001…

.
.
.

Proof

Suppose there *is* a program that solves the halting problem: `halt(t, I)`

Consider all the possible inputs of halt:



All the possible inputs I to a TM
(an infinite, but countable, number of them)

All the possible programs t

1010101000100101…
1101010111100010…
0100100100101010…
0100000001001011…
1111001010101100…
1110010101001001…
.
.
.

`halt(t, I)`

Note that **every *t* is also a valid input *I***: both are just strings of bits, so each *t* appears someplace as a column:

			t_1		t_3			t_5			t_2				t_6	t_4	
t_1	1	0	1	0	1	0	1	0	0	0	1	0	0	1	0	1	…
t_2	1	1	0	1	0	1	0	1	1	1	1	0	0	0	1	0	…
t_3	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	…
t_4	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	1	…
t_5	1	1	1	1	0	0	1	0	1	0	1	0	1	1	0	0	…
t_6	1	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	…

⋮

Let q be the following program:
func q(t) { **if** halt(t, t)==1 { **for** {} } }

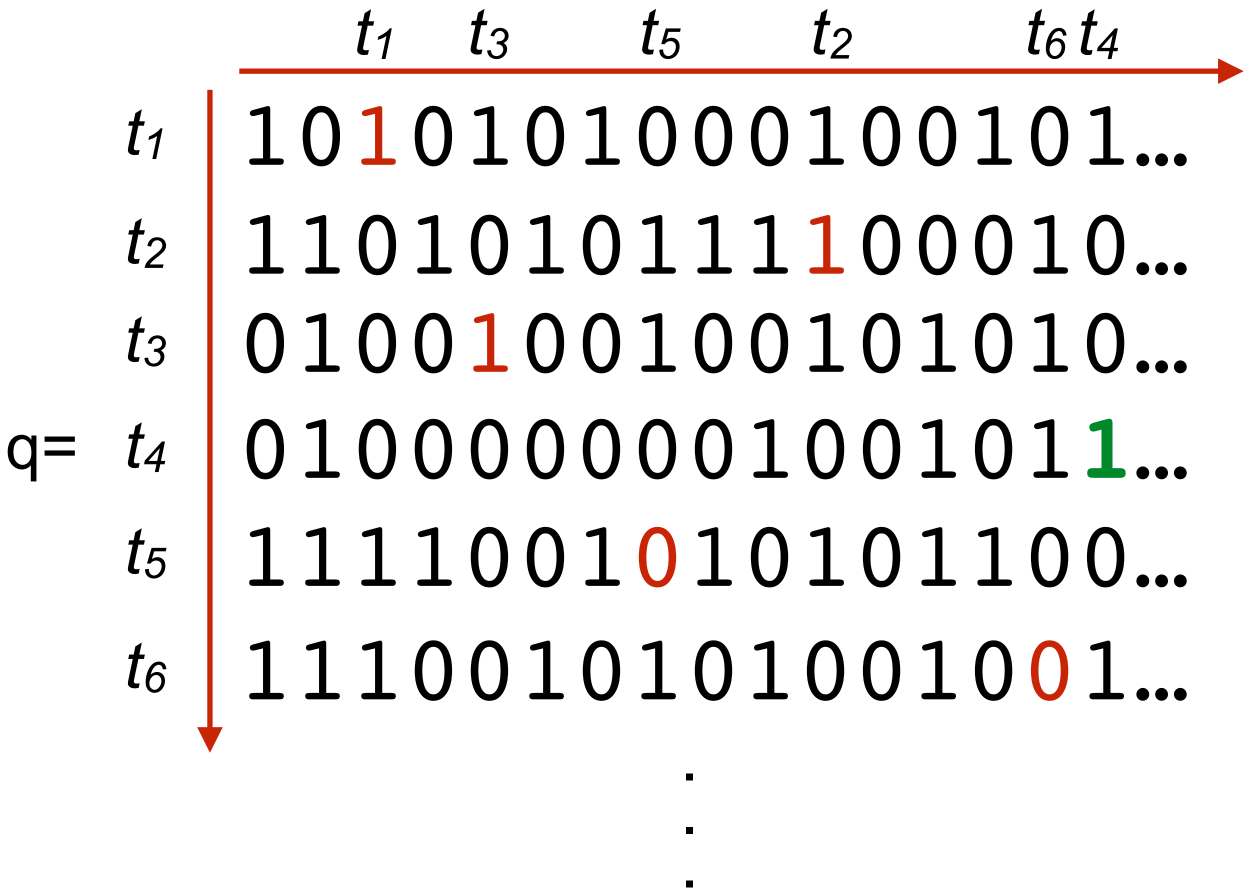
q is a TM that takes as input a program t, and if t halts on input t, q **doesn't** halt, otherwise q halts

Since q is a program, **it must appear someplace in the list of programs** (rows).

Where is q?

Let q be the following program:
func q(t) { **if** halt(t, t)==1 { **for** {} } }

Suppose q = t_4 (say)



What does q(q) do (i.e. $t_4(t_4)$)?

- If q halts (as determined by halt(q,q)), q does the opposite: it runs forever.
- If q doesn't halt, q halts.

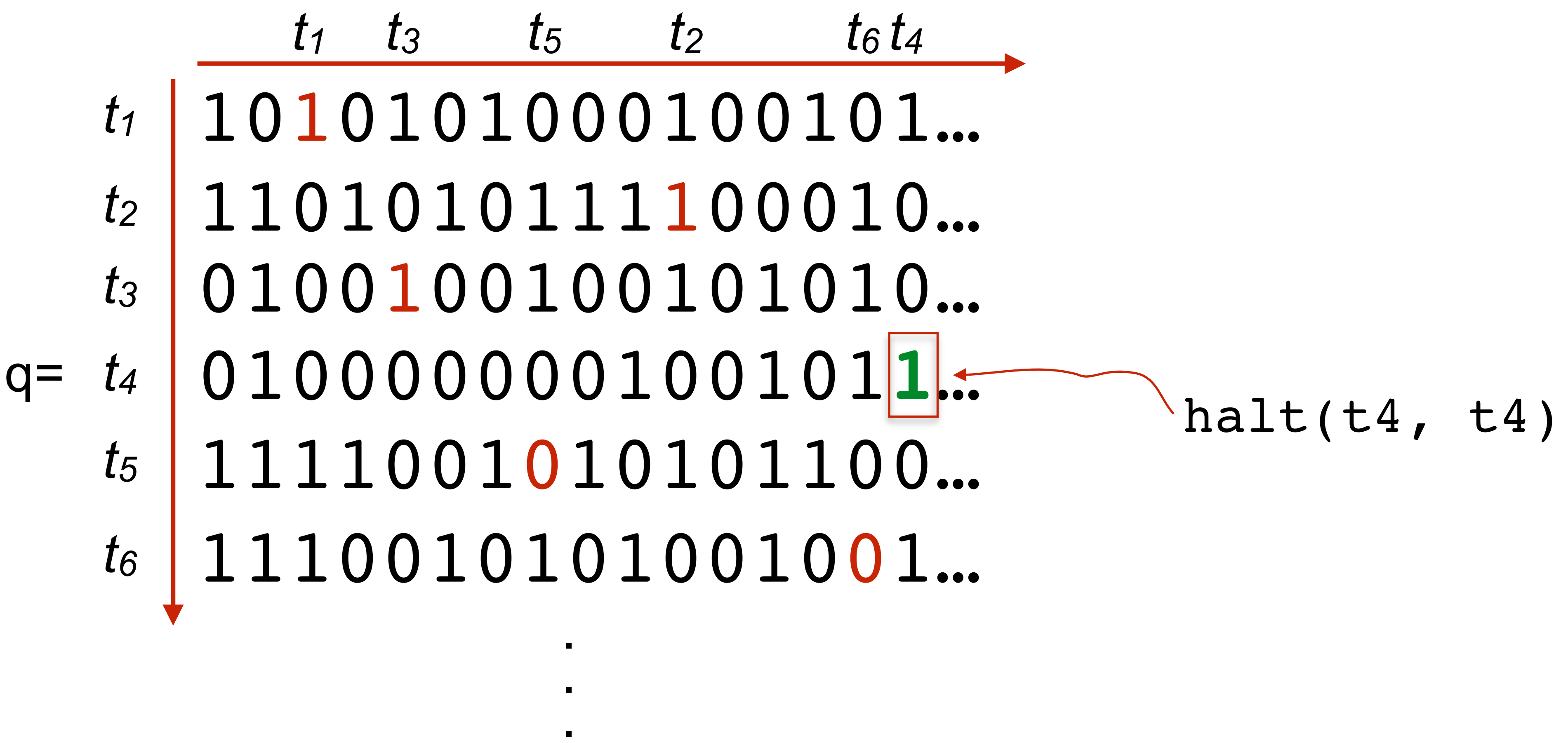
⇒ q does the opposite of q, a contradiction!

⇒ q can't appear in the list of programs

⇒ our assumption that halt(t, I) exists is **false**.

Let q be the following program:
func q(t) { **if** halt(t, t)==1 { **for** {} } }

Suppose q = t_4 (say)



				t_1		t_3			t_5			t_2				t_6	t_4	
	t_1	1	0	1	0	1	0	1	0	0	0	1	0	0	1	0	1	…
	t_2	1	1	0	1	0	1	0	1	1	1	1	0	0	0	1	0	…
	t_3	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	…
q=	t_4	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	1	…
	t_5	1	1	1	1	0	0	1	0	1	0	1	0	1	1	0	0	…
	t_6	1	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	…

`halt(t4, t4)`

⋮

What does q(q) do (i.e. $t_4(t_4)$)?

- If q halts (as determined by halt(q,q)), q does the opposite: it runs forever.
- If q doesn't halt, q halts.

⇒ q does the opposite of q, a contradiction!

⇒ q can't appear in the list of programs

⇒ our assumption that halt(t, I) exists is **false**.

The Limits of Computers

- Every program either halts on *I* or not (true)
- But it’s not possible to write a program to determine this in general.
- There are problems that *computers cannot solve.*
- In fact, most questions *about* programs are undecidable!
  (makes writing an autograder a pretty hard problem… since we can’t have a problem “will halt”)

The set NP

Let's extend what we mean by a TM program to have **two** functions:

$$t_0 : (S, \{0, 1\}) \to (S, \{0, 1\}, \{L, R\})$$
$$t_1 : (S, \{0, 1\}) \to (S, \{0, 1\}, \{L, R\})$$

At every step, it can either use t_0 or t_1 to decide what to do.

Suppose an all powerful being *tells the computer which one to use at each step.*

This is called a **non-deterministic Turing machine.**

NP is the set of **YES / NO** questions answerable in a **polynomial number of steps** on a **non-deterministic** Turing machine.

P=NP? is the question: does this all powerfull being help?

P=NP?

- Intuitively, having an all powerful being give you hints about what to do at every step seems like it should expand the set of questions you can answer efficiently.
- There are many (many, thousands) of problems for which we have efficient non-deterministic programs, but can’t find a regular one.
- On the other hand, no one can prove that non-deterministic TMs can do more than regular ones.

Computational Complexity

- Computational complexity is the field concerned with answering these types of questions.

- Huge amount currently not known about what kinds of problems can be solved in polynomial time.