Carl Kingsford, 02-201, Fall 2015

Lecture 24: Computability and Computational Complexity

The field of computational complexity tries to answer:

- What kinds of problems can a computer solve?
- What kinds of problems can't a computer solve (efficiently)?

Uncomputable functions

We'd really like to write the following function, which checks whether our program, stored in `filename` will ever stop when run on the data in `inputDataFile`. This will tell us if we have a infinite loop bug or not:

```
func doesTerminate(filename string, inputDataFile string) bool {
    // checks whether the Go program in filename stops when run
    // on inputDataFile
}
```

Checking whether a program will ever stop on a given input is called the Halting problem since you are checking whether your program will halt.

Surprisingly, writing such a function is **not possible**! No matter how clever you are, you cannot write it. Now we will see why.

Consider the following function `paradox`:

```
func paradox(filename string) {
    if doesTerminate(filename, filename) {
        for true {
            // loop forever, i.e. do not terminate
        }
    }
}
```

This function does the opposite of what the program in `filename` does: if `filename` halts on the data in `filename`, then `paradox` runs forever. If `filename` runs forever, then `paradox` stops.

What happens when you put the `paradox` function into a file as a program and run:

```
paradox("paradox.go")
```

If `paradox` terminates: then `paradox` runs forever.

If `paradox` runs forever: then `paradox` terminates.

So `paradox` can neither run forever nor halt, which means that the `paradox` function can't exist. This means that `doesTerminate` can't exist.

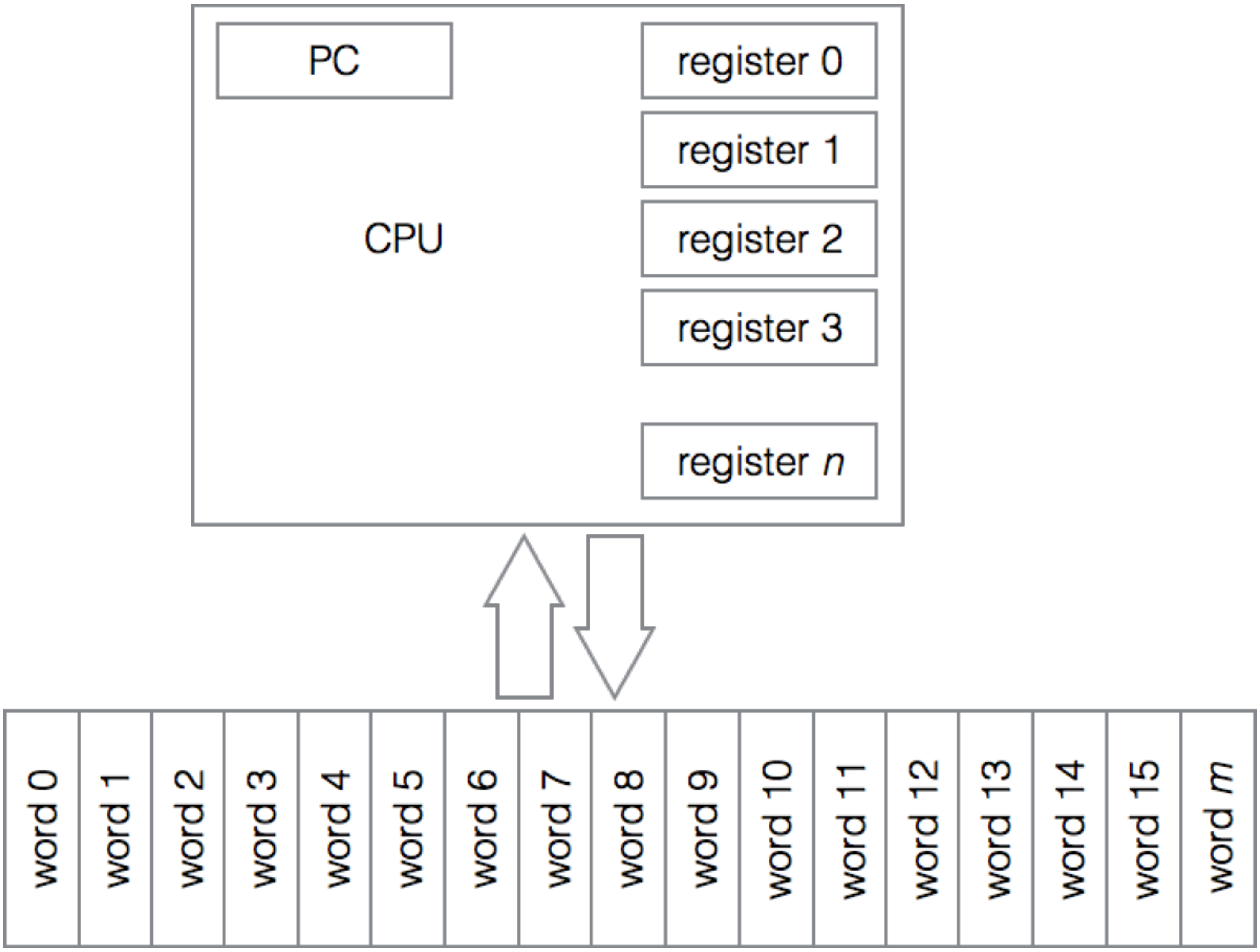
This situation is pretty surprising: Every program either halts on a given input or not. But it’s not possible to write a program to determine this in general. This shows that there are problems that computers cannot solve. In fact, most questions about programs are undecidable! This is not a unique situation: there are may questions about programs that programs can't answer! (This makes writing an autograder a pretty hard problem... since we can’t have a point category "will halt").

Next, we'll turn to some more positive results: questions that computers *can* answer efficiently.

An idealized computer: Turing Machines

Real computers

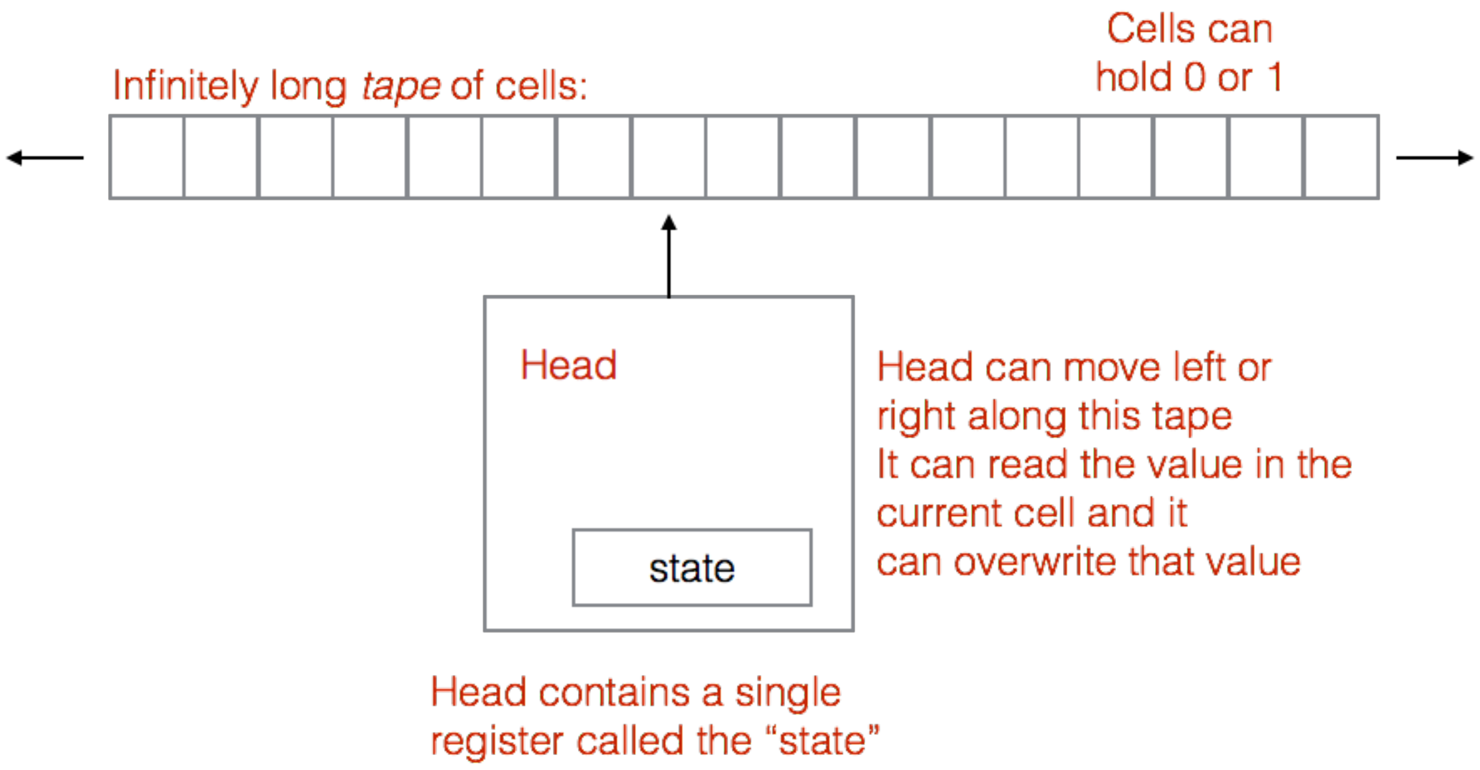
The conceptual architecture of a modern computer looks like:



You've seen this before. This kind of computer can be somewhat difficult to reason about: there are registers, RAM, and lots of different CPU instructions. Some of the computer is also based on limits of our current technology: how much RAM? how many registers?

Turing machines

Alan Turing proposed an idealized version of a computer (before, in fact, computers really existed). This is now called a Turing machine:

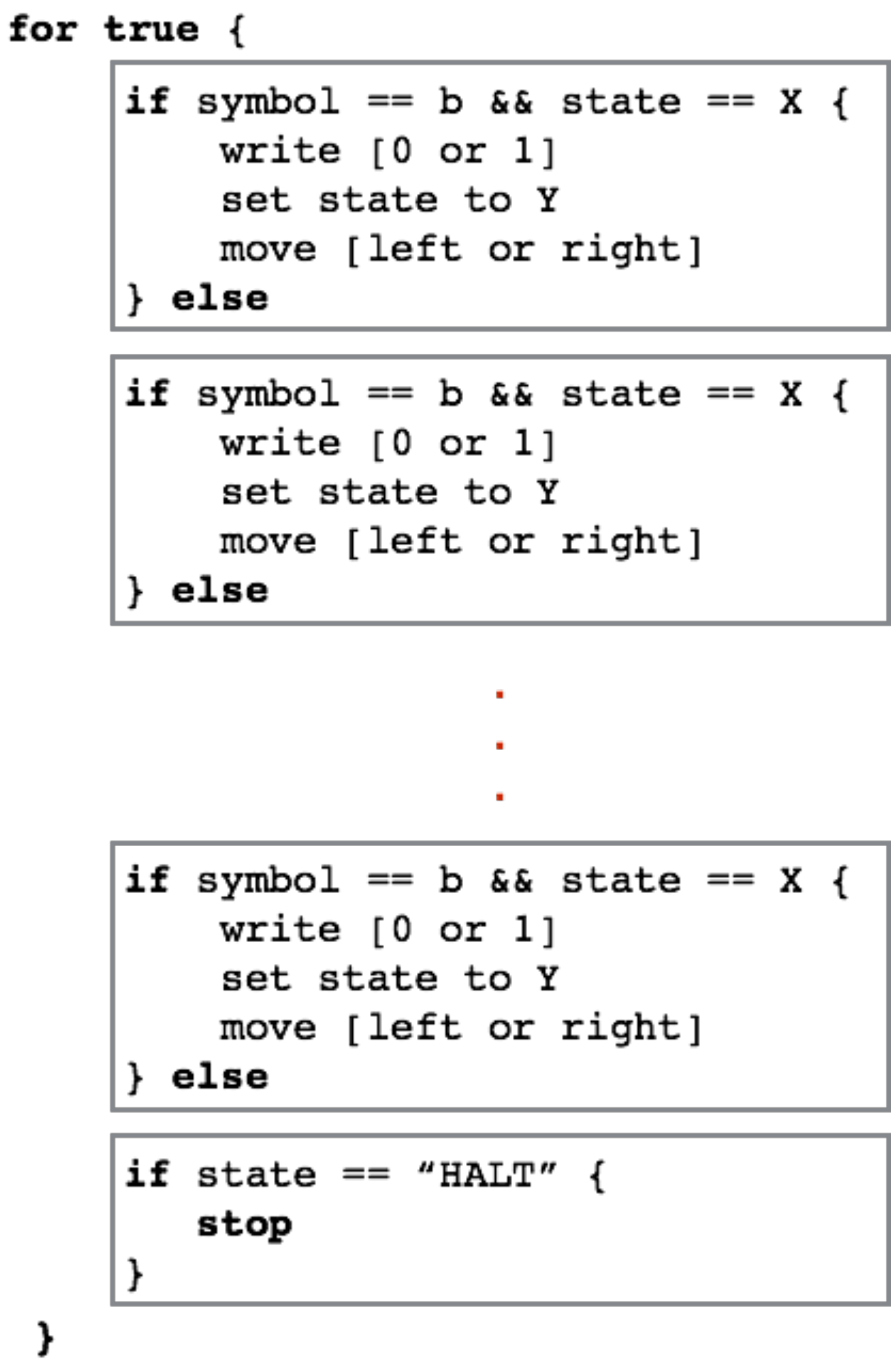

Infinitely long *tape* of cells:
Cells can hold 0 or 1
Head
state
Head can move left or right along this tape
It can read the value in the current cell and it can overwrite that value
Head contains a single register called the "state"

A Turing Machine repeatedly executes steps. At each step, the machine decides what to do based only on the symbol b on the current tape position and the value of the state register.

At each step, the machine:

- writes a value to the current cell,
- changes the value of the state variable to something, and
- then moves left or right.

In other words, a Turing machine (TM) executes a program such as this:



```
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
    }
 }
```

Instead of thinking of a TM as a program, we can think of it as a function that gets called repeatedly:

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

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

Turing machines can solve anything a current real computer can. We won't prove this, but intuitively it should be clear that a TM can do anything a current 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:

The Church-Turing thesis is: "Everything that is efficiently computable is efficiently computable on a Turing Machine."

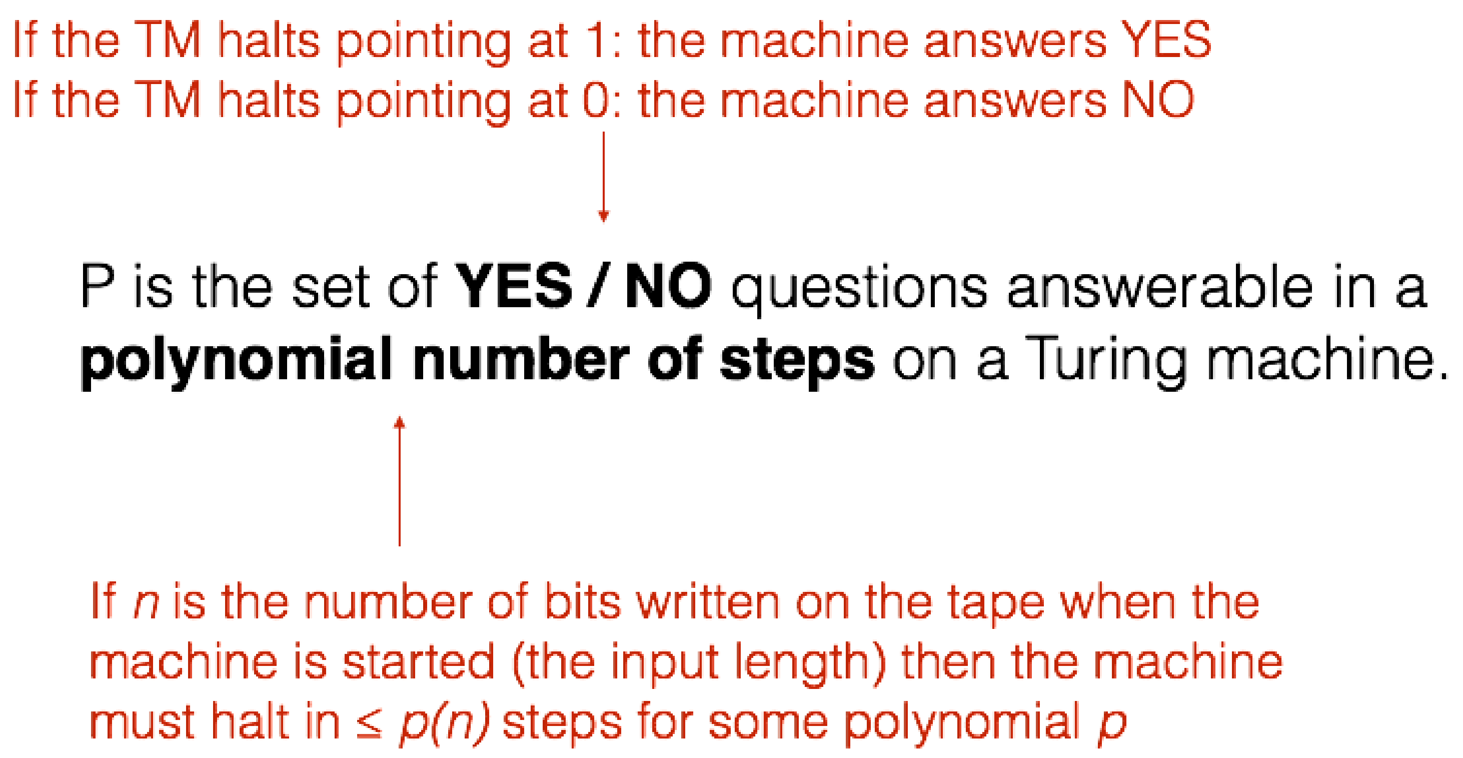
Notice that this is much stronger than the (true) statement that Turing machines can solve anything your laptop can.

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

As such, it's not something that can really be "proven", but it seems intuitively reasonable.

The set P

The set P is the set of computational questions that can be answered in a polynomial number of steps.



If the TM halts pointing at 1: the machine answers YES
If the TM 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*

Polynomial efficiency is a reasoanble choice to define "efficient" since polynomials like $p(n) = n^5 + 6n^3 + 2n + 3$ grow much much slower than exponential functions like $e(n) = 2^{3n+2}$.

Nondeterministic Turing Machines

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

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

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

At every step, the TM 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 (NDTM). It's non-deterministic since it has choices at each step.

Another way to think about NDTM: at each step, the NDTM tries *both* versions of the functions. Notice this splits the computation into two totally different histories.

The set NP

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

P = NP?

P=NP? is the question: does this all powerful being telling us which function to use at each step help?

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 program.

On the other hand, no one can prove that non-deterministic TMs can do more than regular ones.

Proving $P \neq NP$ is one of the (if not *the*) major open question in mathematics and computer science. Showing that $P = NP$ would immediately give faster algorithms for thousands of problems. Showing that $P \neq NP$ would show that many problems do not have good algorithms, no matter what we do.

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

There is a huge amount currently not known about what kinds of problems can be solved in polynomial time.

Summary

- There are problems that no computer can solve. The Halting problem is one of them.
- Turing machines are an idealized version of real computers that a somewhat easier to reason about. We think Turing machines can simulate any classical computer we will ever invent. (Quantum computers are a different story!)
- Non-determinstic Turing machines are an unrealistic model of computers. It seems like such a powerful computer can never be built (Quantum?).
- P = set of problems solvable in polynomial number of steps on a TM. NP = set of problems solvable in polynomial number of steps on a NDTM.
- No one can prove that $P \neq NP$, even though it really seems like NP is much bigger than P.

Glossary

- P: set of problems solvable in polynomial number of steps on a TM.
- NP: set of problems solvable in polynomial number of steps on a NDTM.
- Halting problem: The computational question asking whether a program ever stops running when given a particular input.
- Turing machine: An idealized model of real computers.
- Nondeterministic Turing machine: An idealized model of computation that seems much more powerful than real computers.