4/10

Poly time algs via greedy, dynamic prog, flow.

NP-complete problems – believe not solvable in poly time.

- Clearly can be solved, just slowly.

Before all this (1920s, 1930s): Are there (math) problems that can't be solved by algorithms?

- Actually there are – this and next few lectures.

What kinds of problems?

1920s – David Hilbert

- Can we automate all of mathematics?
  - ~ Is there an algorithm that can take a mathematical statement, run for some finite steps, and output whether it's true or false?

    (The Entscheidungsproblem)
  - The answer turned out to be (basically) no.

A question that turns out to be related:

Program equivalence.

Given two programs P_1, P_2, each takes
a natural number n as input,
and outputs $P_1(n)$ and $P_2(n)$ respectively.

Q: Does $P_1(n) = P_2(n)$ for all n, or
is there an n where $P_1(n) \neq P_2(n)$?

Example of two programs (to suggest this is hard)

Pick a conjecture in math:

e.g. Goldbach Conjecture: Every even number n
can be written as the sum of two primes.

Program $P_1(n)$: {If n is odd, output "Input must be even"

Else: For $k=2$ to $n-2$:

if k and $n-k$ are both prime
then output "yes".

Endfor

Output "no"

}

Program $P_1(n)$: { If n is odd, output "Input must be even"

Else: For $k=2$ to $n-2$:

if k and $n-k$ are both prime

then output "yes".

Endfor

Output "no"

}

Program $P_2(n)$: { If n is odd, output "Input must be even"

Else: output "yes" }

P_1 and P_2 are algorithms that run in finite time & take a single n and test if n can be written as sum of two primes.

Question: Is there a program P^* that takes P_1, P_2 and decides if equivalent?

Later: No: there is no algorithm that on every pair P_1, P_2 runs in finite steps and then outputs the correct answer.

If we wanted & show program equivalence

Can't be solved by an algorithm (running in finite time, always correct)

first we need to know: what is an algorithm?

1930s: Church λ-calculus, Post rewriting systems
Turing machine, ...

And as powerful as python, C, modern PL.

- All turned out to equivalently powerful, so any of them can be used as the definition of an algorithm.

Can translate from one to other.

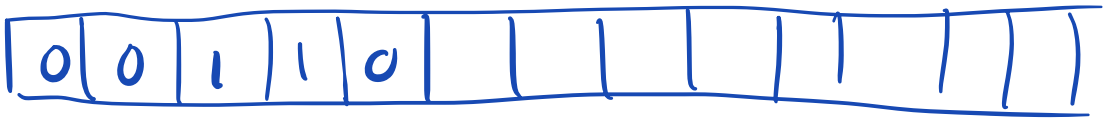
→ Church-Turing thesis: these formalisms are what we mean by "algorithm"

Formally: define a Turing Machine.

- Input: a string of symbols over some alphabet.
- Output: a decision: binary, yes/no.

How does a Turing Machine work?

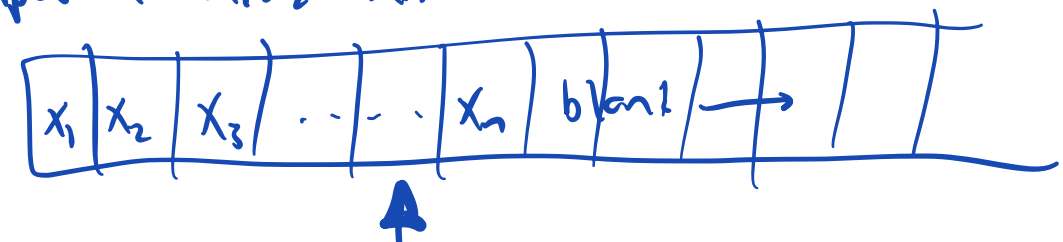
Input: 00110



⟶ blank

Infinite input tape, input itself starts at left, finite sequence of symbols, infinite blank tape square after.

Input $x = x_1 x_2 \cdots x_n$



Control unit:

- Looks at a tape square
- Reads what's there.
- Remembers some constant amount of info
- Writes a new symbol
- Moves left or right.

like a finite automaton that can:

- move in both directions and
- write notes on input tape.

Formally: Turing machine consists of:

- A finite state set Q (internal states)
- An input alphabet Σ
- A tape alphabet Γ $(\Sigma \subseteq \Gamma)$
- A special "blank" symbol $\sqcup \in \Gamma - \Sigma$
- A special "left end of tape" symbol $\vdash \in \Gamma - \Sigma$
- A start state s, an accept state t, a reject state r (and Q can have other states)

- A state transition function δ

$$\delta(q, x) = (q', x', L \text{ or } R)$$

state — symbol — new state — new symbol — move left or right

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}.$$

Basic rules:

$$\delta(q, \vdash) = \delta(q', \vdash, R)$$

$$\delta(t, x) = \delta(t, x', L \text{ or } R)$$

$$\delta(r, x) = \delta(r, x', L \text{ or } R)$$

No guarantee the Turing machine ever reaches its accept or reject state (might not halt)