Turing Machines

"Most General" computer?

- DFA/PDAs are simple models of computation.
  - Accept only the regular/CF languages.

- Is there a kind of computer that can accept *any* language, or compute *any* function?

- Recall counting argument:
  - $\{ L \mid L \subseteq \{0,1\}^* \}$ (just the set of languages)

    uncountably infinite
  - {P : P is a finite length computer program} is

    countably infinite

Most General Computer

- If not all functions are computable, *which are?*
- Is there a "most general" model of computer?
- What languages can they recognize?

David Hilbert

- Early 1900s – crisis in math foundations
  - – attempts to formalize resulted in paradoxes, etc.
- 1920, Hilbert's Program:

  "mechanize" mathematics
- Finite axioms, inference rules

  turn crank, determine truth

  needed: axioms *consistent* & *complete*

Kurt Gödel

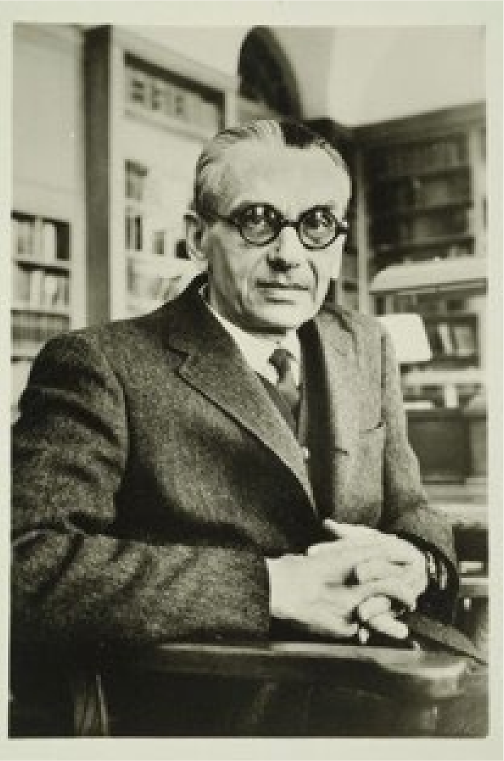
- German logician, at age 25 (1931) proved:

"There are true statements that can't be proved"

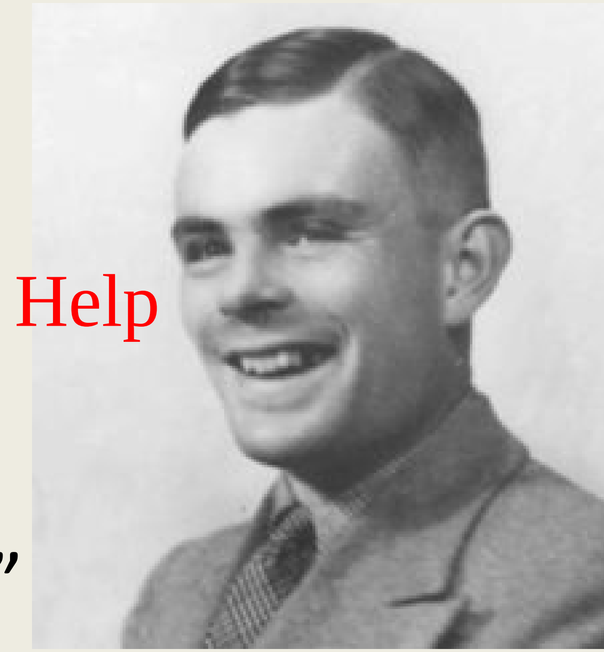
(i.e., "no" to Hilbert)

- Shook the foundations of

  - mathematics
  - philosophy
  - science
  - everything

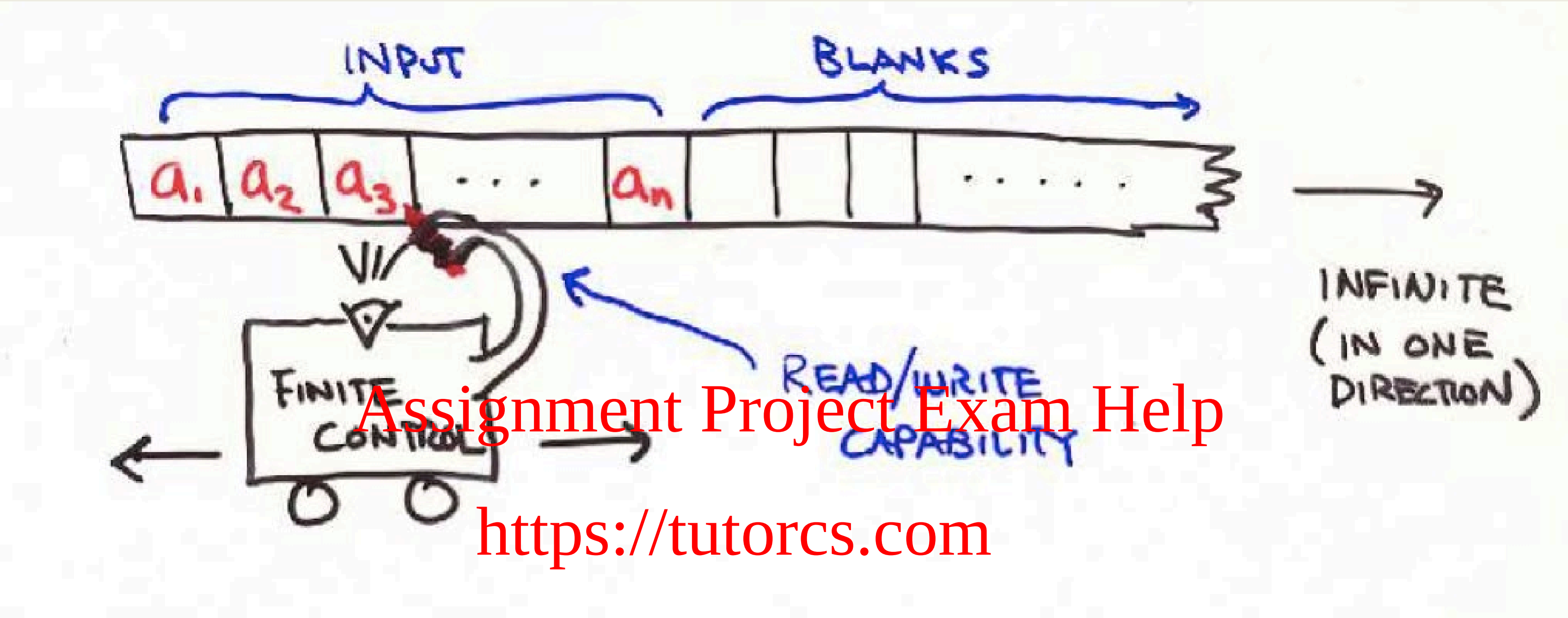


Alan Turing



- British mathematician
  - cryptanalysis during WWII
  - arguably, father of AI, Theory
  - several books, movies
- Defined "*computer*", "*program*"

and (1936) at age 23 provided foundations for investigating fundamental question of what is computable, what is not computable.



- DFA with (infinite) tape.
- One move: read, write, move, change state.

High-level Goals

- **Church-Turing thesis:** TMs are the most general computing devices. So far no counter example

- Every TM can be represented as a string. Think of TM as a program but in a very low-level language.

- Existence of **Universal Turing Machine** which is the model/inspiration for stored program computing. UTM can simulate any TM

- Implications for what can be computed and what cannot be computed

Formal Definition

$M = (Q, \Sigma, \Gamma, \delta, q_{0,} B, q_{accept}, q_{reject})$, where:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ as defined on next page
- Γ is a finite tape alphabet. (Σ a subset of Γ)
- q_0 is the initial state (in Q)
- B in $\Gamma - \Sigma$ is the blank symbol
- q_{accept}, q_{reject} are unique accept, reject states in Q

Transition Function

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

current state	symbol scanned	new state	symbol to write	direction to move on tape

$$\delta(q,a) = (p, b, L)$$

from state q, on reading a:

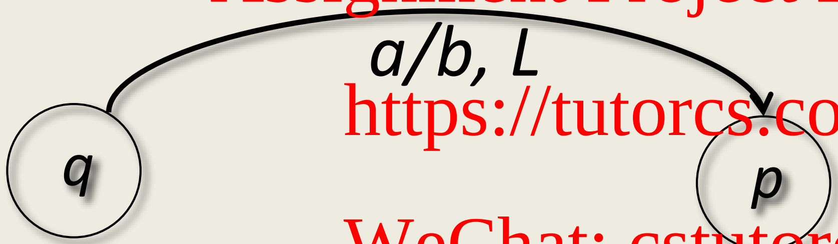
go to state p

write b

move head Left

Graphical Representation

$\delta(q,a) = (p, b, L)$

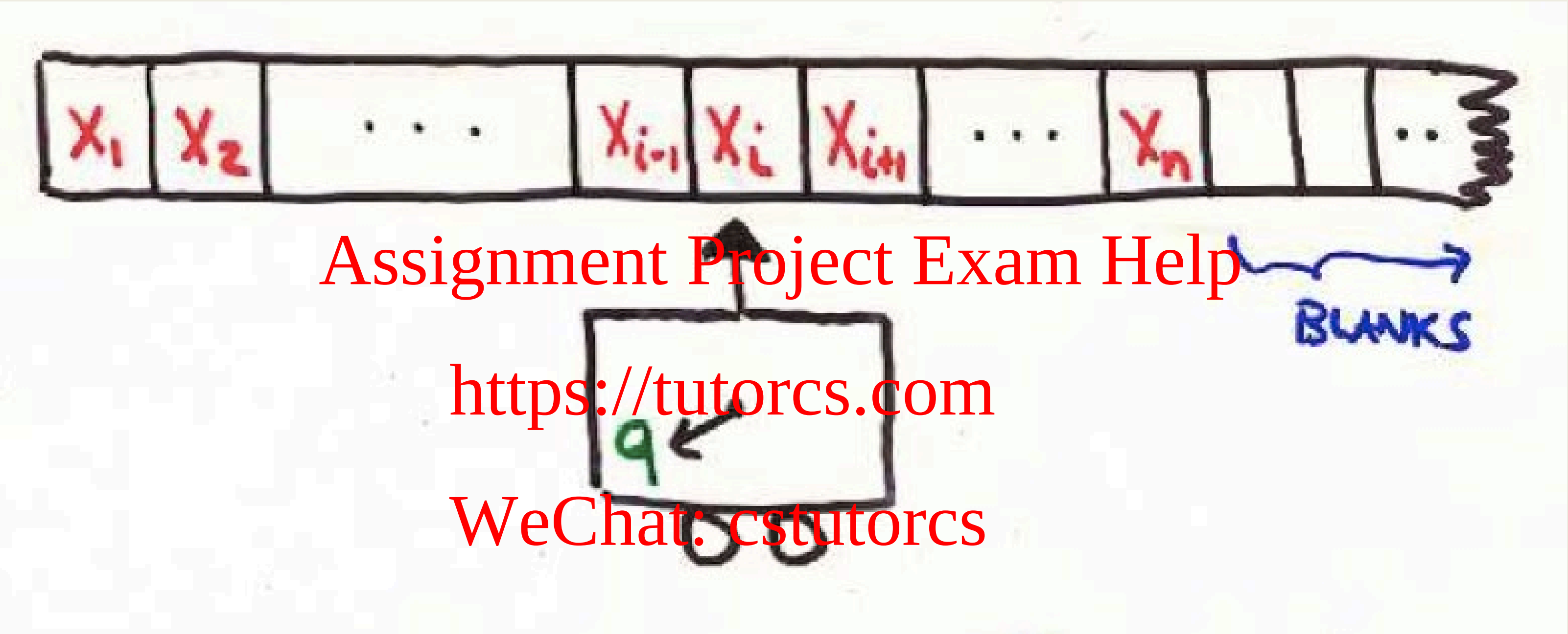


Note: we allow $\delta(q,a)$ to be undefined for some choices of q, a (in which case, M "crashes")

ID: Instantaneous Description

- Contains all necessary information to capture "state of the computation"
- Includes
  - state *q* of *M*
  - location of read/write head
  - contents of tape from left edge to rightmost nonblank (or to head, whichever is rightmost)

ID: Instantaneous Description



ID: $X_1X_2\ldots X_{i-1}\ q\ X_iX_{i+1}\ldots X_n$ (q in Q, X_i in Γ)

Relation “→” on IDs

If $\delta(q, X_i) = (p, Y, L)$, then

$$\underbrace{X_1X_2\ldots X_{i-1}\; q\; X_iX_{i+1}\ldots X_n}_{\text{current ID}} \rightarrow \underbrace{X_1X_2\ldots X_{i-2}\; p\; X_{i-1}\; Y\; X_{i+1}}_{\text{next ID}}$$

If $\delta(q, X_i)$ is undefined, then there is no next ID

If *M* tries to move off left edge, there is no next ID

(in both cases, the machine “crashes”)

Capturing many moves...

Define $\rightarrow^*$ as the reflexive, transitive closure of $\rightarrow$

Thus, ID $\rightarrow^*$ ID' iff *M*, when run from ID, necessarily reaches ID' after some finite number of moves.

Initial ID: $q_0 w$ (more often, assume ... $\$q_0 w$)

Accepting ID: $\alpha_1 \, q_{accept} \, \alpha_2$ for any α_1, α_2 in Γ^*

(reaches the accepting state with any random junk left on the tape)

Definition of Acceptance

M accepts *w* iff for some α_1, α_2 in Γ^*,

$$q_0 w \rightarrow^* \alpha_1 q_{accept} \alpha_2$$

M accepts if at *any* time it enters the accept state

Regardless of whether or not

- it has scanned all of the input
- it has moved back and forth many times
- it has completely erased or replaced *w* on the tape

$$L(M) = \{w \mid M \text{ accepts } w\}$$

Non-accepting computation

M doesn't accept *w* if any of the following occur:

- *M* enters q_{reject}
- *M* moves off left edge of tape
- *M* has no applicable next transition
- *M* continues computing forever

If *M* accepts – we can tell: it enters q_{accept}

If *M* doesn't accept – we may not be able to tell

(c.f. "Halting problem" – later)

"Recursive" vs "Recursively Enumerable"

- *Recursively Enumerable (r.e.) Languages:*

  = {L | there is a TM M such that L(M) = L}

- *Recursive Languages* *(also called "decidable")*

  = {L | there is a TM M that halts for all w in Σ^*

  and such that L(M) = L }

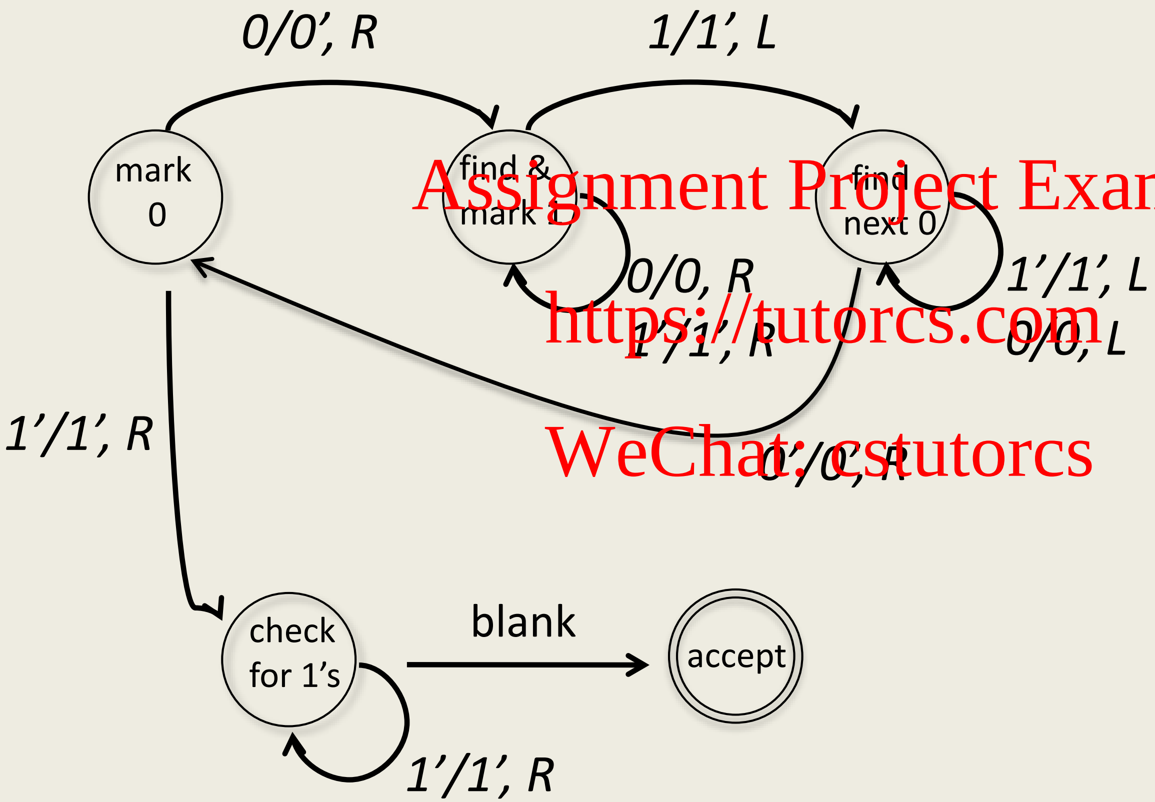
Recursive languages: nice; run *M* on *w* and it will eventually enter either q_{accept} or q_{reject}

R.E. languages: not so nice; can know if *w* in *L*, but not necessarily if *w* is not in *L.*

Fundamental Questions

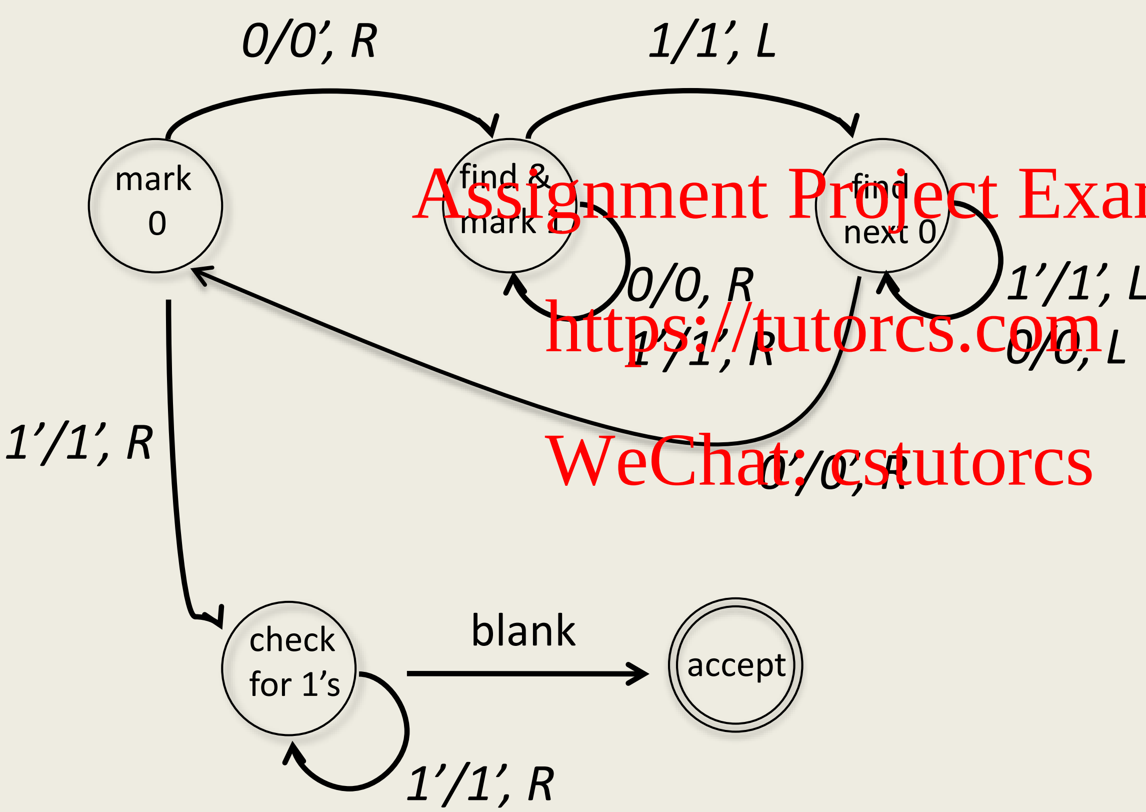
- Which languages are R.E.?
- Which are recursive?
- What is the difference?
- What properties make a language decidable?

Machine accepting $\{0^n 1^n \mid n \geq 1\}$



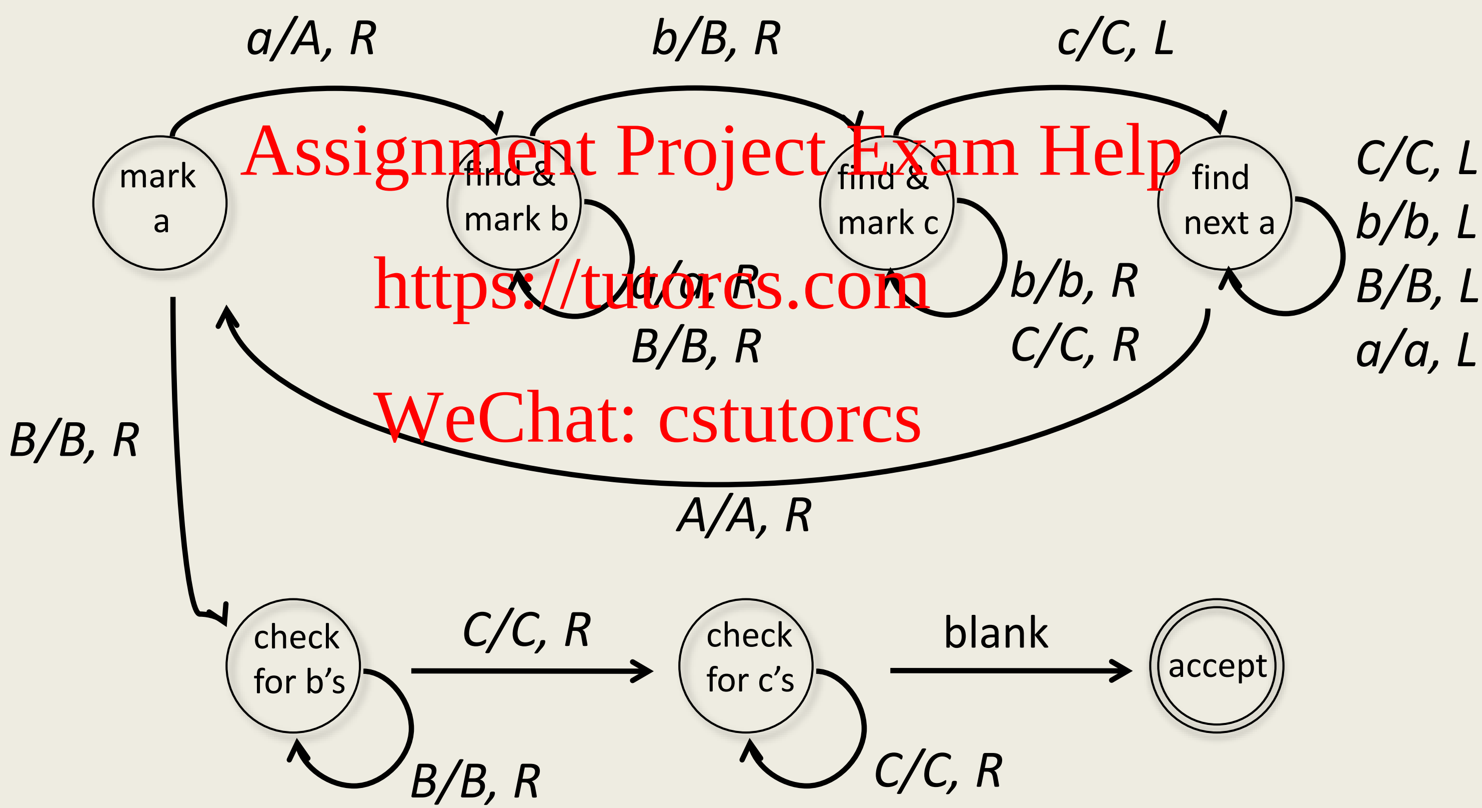
(This technique is known as “checking off symbols”)

Machine accepting $\{0^n1^n \mid n \geq 1\}$



(This technique is known as “checking off symbols”)

Machine accepting $\{a^n b^n c^n \mid n \geq 1\}$



(This technique is known as "checking off symbols")

Machine to add two n-bit numbers

("high-level" description)

- Assume input is $\$a_1a_2...a_n\#b_1b_2...b_n$
- Pre-process phase

  – sweep right, replacing 0 with 0' and 1 with 1'

- Main loop:

  – erase last bit b_i, and remember it
  – move left to corresponding bit a_i
  – add the bits, plus carry, overwrite a_i with answer
  – remember carry, move right to next (last) bit b_{i-1}

Program Trace (some missing steps)

$10011#11001

$1’0’0’1’1’#1’1’0’0’1’

$1’0’0’1’1’#1’1’0’0’1’   *b = 1*, *c = 0*

$1’0’0’1’1’#1’1’0’0’

$1’0’0’1’1’#1’1’0’0’

$1’0’0’1’1’#1’1’0’0’

$1’0’0’1’1’#1’1’0’0’

$1’0’0’1’1’#1’1’0’0’

$1’0’0’1’1’#1’1’0’0’

$1’0’0’1’0#1’1’0’0’   *c = 1*

$1’0’0’1’0#1’1’0’0’

$1’0’0’1’0#1’1’0’0’

$1’0’0’1’0#1’1’0’0’

$1’0’0’1’0#1’1’0’0’

$1’0’0’1’0#1’1’0’0’

$1’0’0’1’0#1’1’0’   *b = 0*, *c = 1*

$1’0’0’1’0#1’1’0’

$1’0’0’1’0#1’1’0’

$1’0’0’1’0#1’1’0’

$1’0’0’00#1’1’0’   *c = 1*

$1’0’0’00#1’1’0’   etc

Some TM programming tricks

- checking off symbols
- shifting over
- using finite control memory
- subroutine calls

“Extensions” of TMs

- 2-way infinite tape
- multiple tracks
- multiple tapes
- multi-dimensional TMs
- nondeterministic TMs
- --- bells & whistles

Goal:

Convince you of the power of the basic model

“Extensions” of TMs: 2-way infinite tape

.	.	-5	-4	-3	-2	-1	0	1	2	3	4	5	.	.

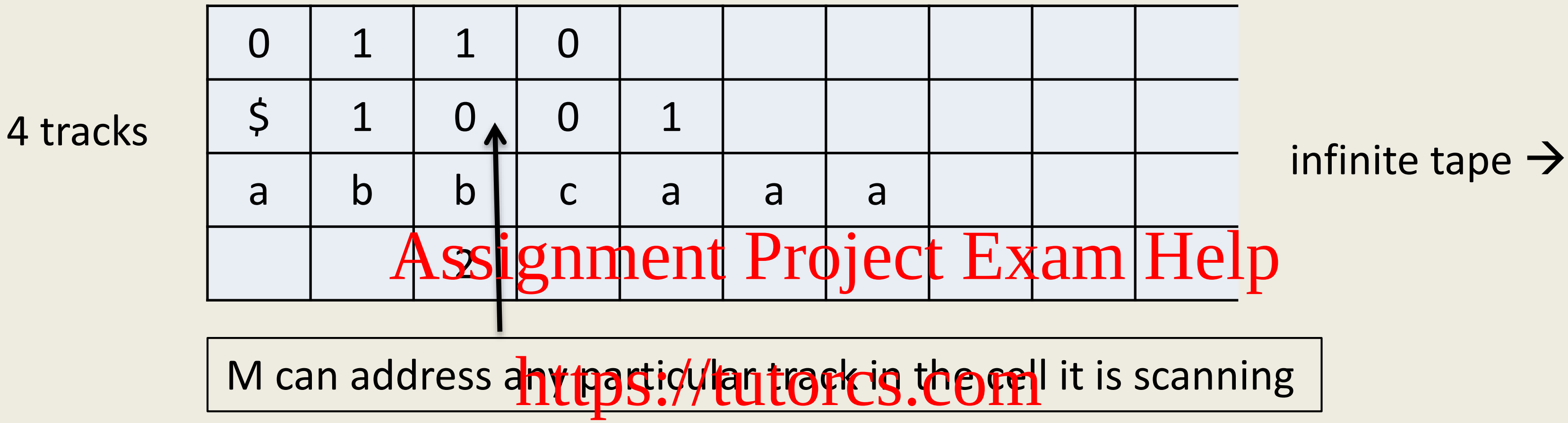
Simulate with 1-way infinite tape…

0	1	-1	2	-2	3	-3	4	-4	5	-5	6	.	.	.

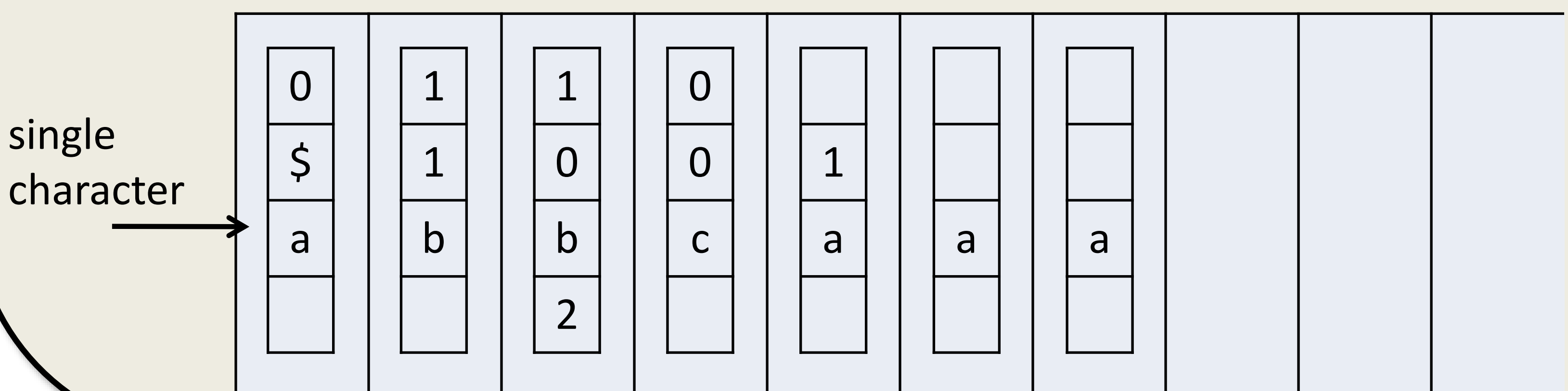
Must modify transitions appropriately

- remember in finite control if negative or positive
- if positive, R → RR; L → LL
- if negative, R → LL; L → RR
- must mark left edge & deal with 0 cell differently

Extension: multiple tracks



Can simulate 4 tracks with a single track machine, using extra "stacked" characters:



Multiple tracks

4 tracks

0	1	1	0						
$	1	0	0	1					
a	b	b	c	a	a	a			
		2							

infinite tape →

M: $\delta(q, -,0,-,-) = (p, -,-,-,1, R)$

“If in state q reading 0 on second track, then go to state p, write 1 on fourth track, and move right”

Then in M’ $\delta\left(q, \begin{bmatrix} x \\ 0 \\ y \\ z \end{bmatrix}\right) = \left(p, \begin{bmatrix} x \\ 0 \\ y \\ 1 \end{bmatrix}, R\right)$

for *every* x, y, z in Γ

Extension: multiple tapes

k-tape TM

- *k* different (2-way infinite) tapes
- *k* different independently controllable heads
- input initially on tape 1;   tapes 2, 3, ..., *k*, blank.
- single move:
  - read symbols under all heads
  - print (possibly different) symbols under heads
  - move all heads (possibly different directions)
  - go to new state

k-tape TM transition function

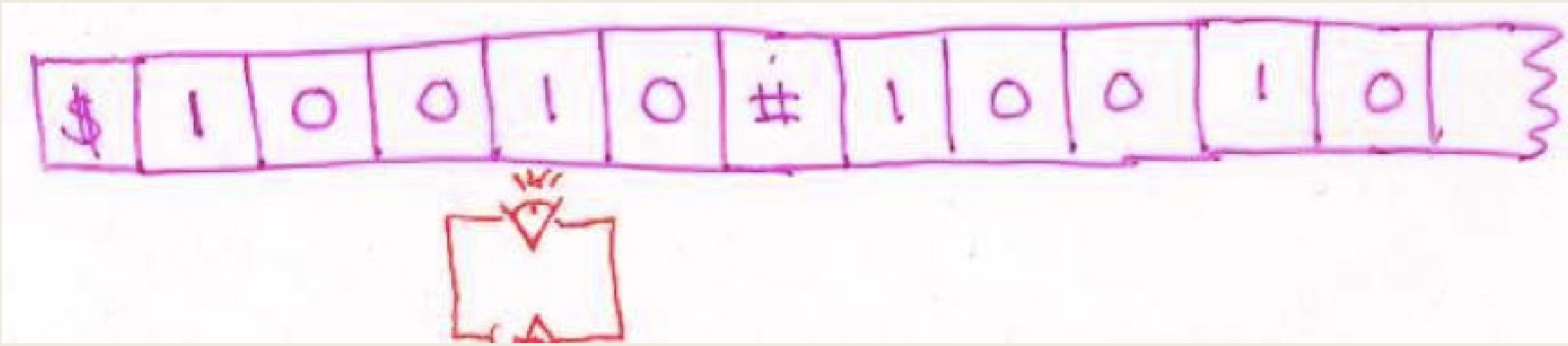
$$\delta(q, \underbrace{a_1, a_2, \ldots a_k}_{\text{Symbols scanned on the } k \text{ different tapes}}) = (p, \underbrace{b_1, b_2, \ldots b_k}_{\text{Symbols to be written on the } k \text{ different tapes}}, \underbrace{D_1, D_2, \ldots D_k}_{\text{Directions to be moved } (D_i \text{ is one of L, R, S})})$$

Utility of multiple tapes

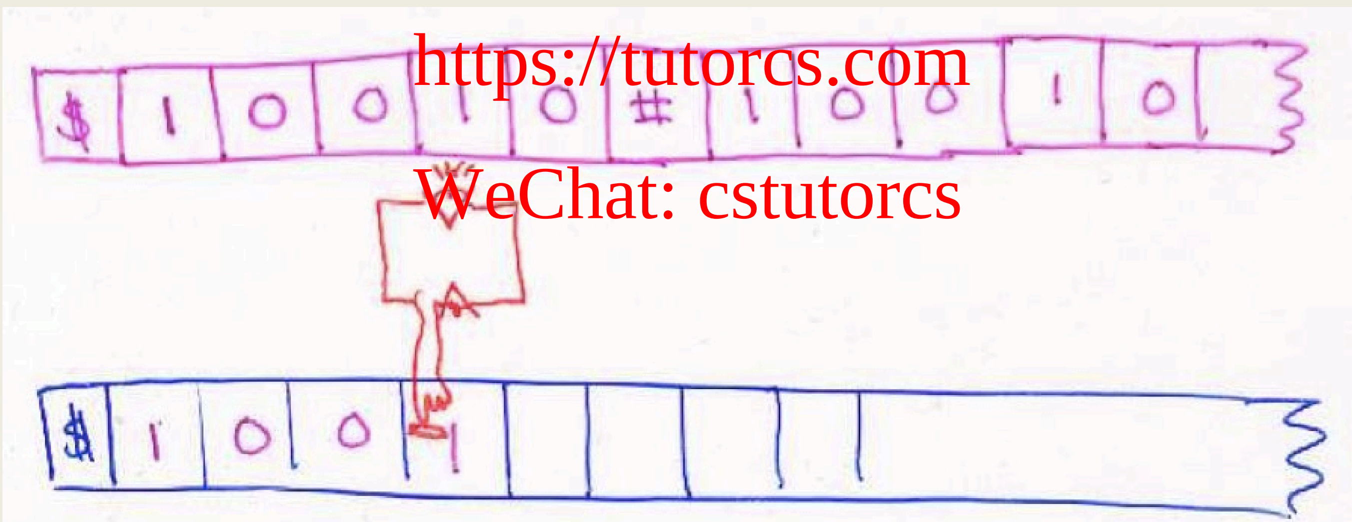
makes programming a whole lot easier

\$	1	0	0	1	0	#	1	0	0	1	0			

is input string of form w#w ?



$\Omega(n^2)$ steps provably required



≈ 3n/2 steps easily programmed

Can't compute more with k tapes

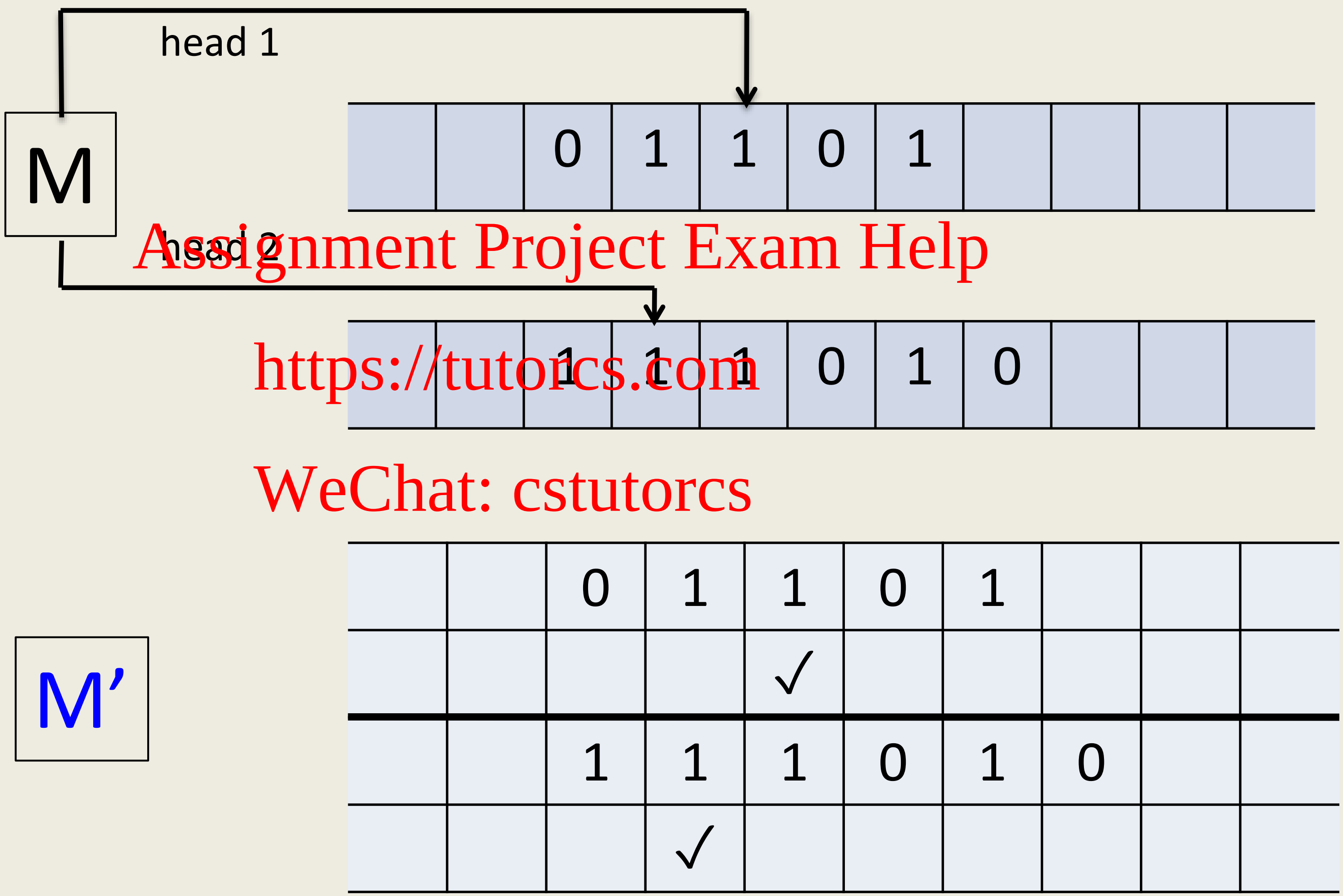
Theorem: If L is accepted by a k-tape TM M, then L is accepted by some 1-tape TM M'.

Intuition: M' uses 2k tracks to simulate M

BUT....
M has k heads!

How can M' be in k places at once?

Snapshot of simulation (k = 2)



Track 2i-1 holds tape i. Track 2i holds position of head i

To make a move, M’ does:

Phase 1: Sweep from leftmost edge to rightmost “✓” on any track, noting symbols ✓’ed, and what track they are on. Save this info in finite control.

Now, M’ knows what move of M to make

Phase 2: Sweep from right to left edge implementing the move of M

Thus, each move of M requires M' to do a complete sweep across, and back.

Not hard to show that if M takes t steps to complete its computation, then M' takes $O(t^2)$ steps.