Turing Machines

"Most General" computer?

- DFAs are simple model of computation.
  - Accept only the regular 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 languages)
    (a) countably infinite
    (b) uncountably infinite
  - {P : P is a finite length computer program} is
    (a) countably infinite
    (b) uncountably 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*

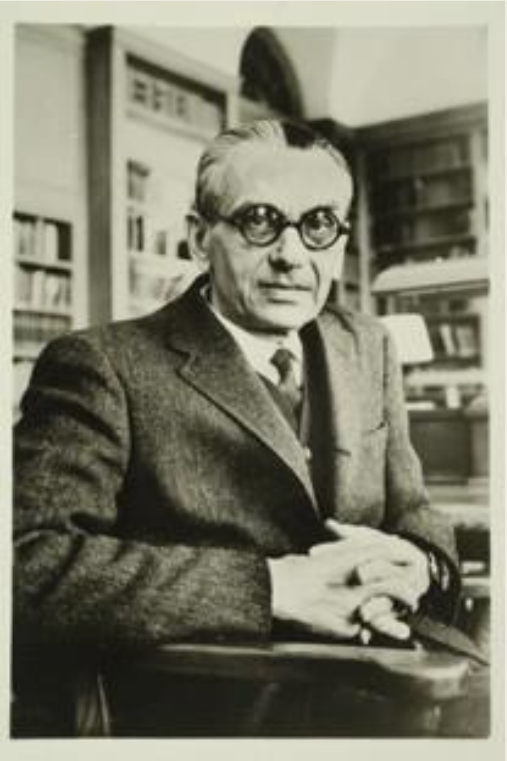
Kurt Gödel

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

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

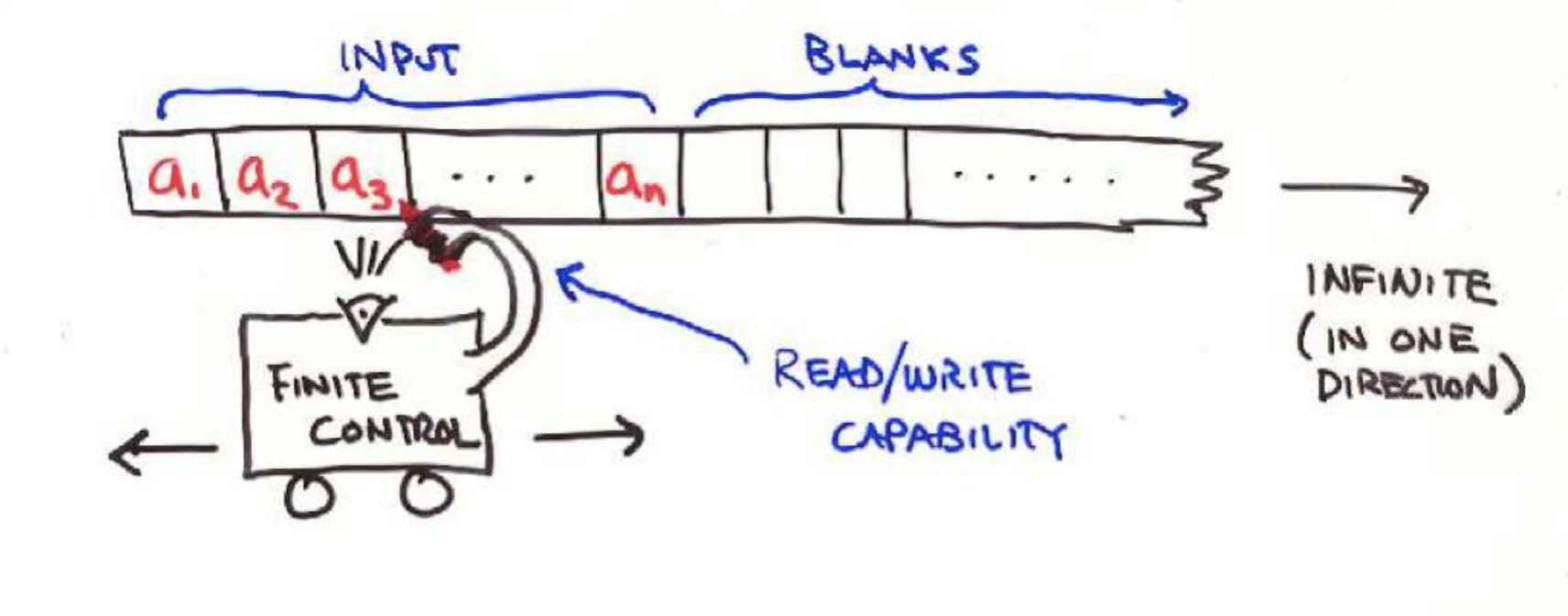
(i.e., "no" to Hilbert)

- 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*:

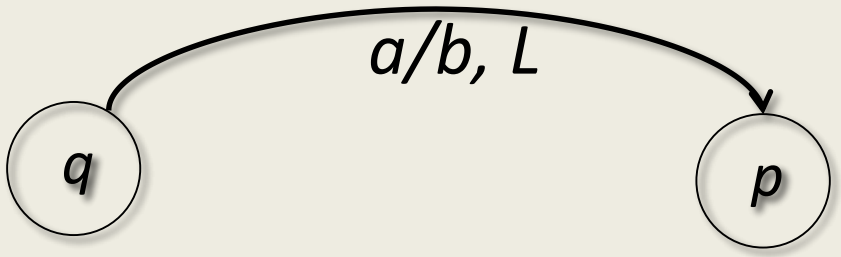
go to state *p*

write *b*

move head Left

Graphical Representation

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

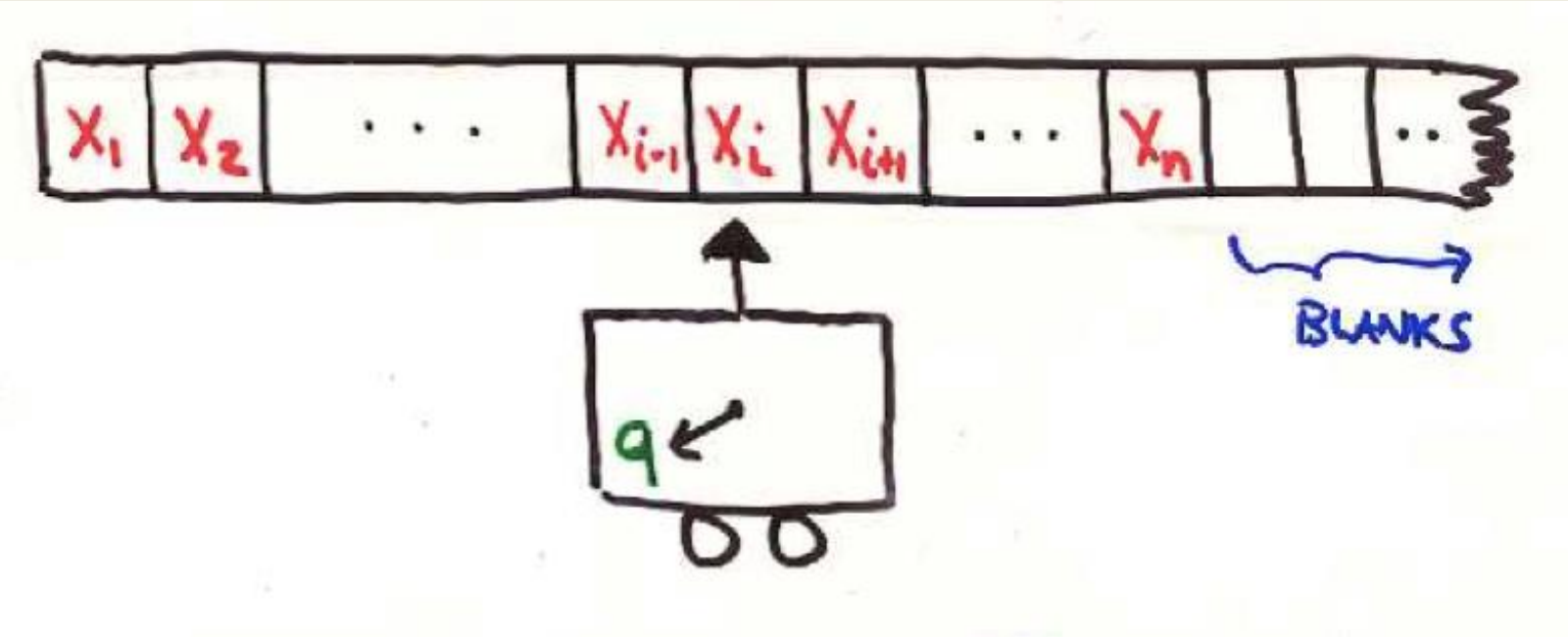


Note: we allow $\delta(q,a)$ to be undefined for some choices of *q, a* (in which case, *M* "crashes")

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)

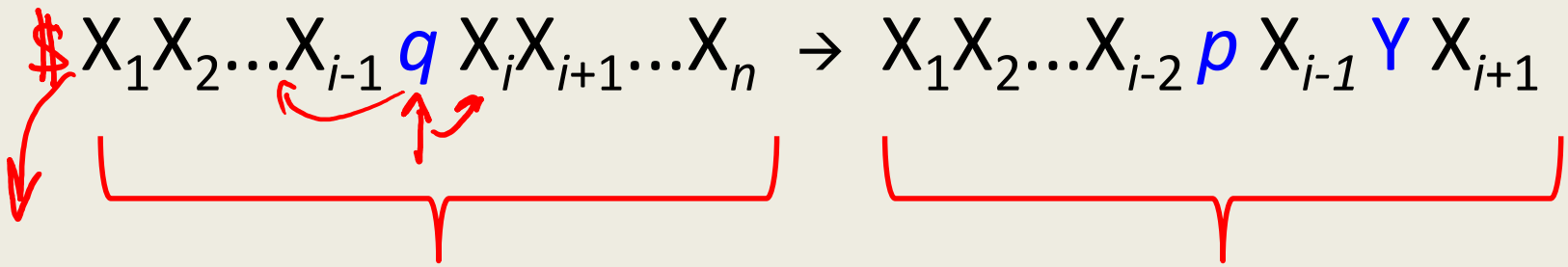
ID: Instantaneous Description



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

Relation "→" on IDs

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



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, $ID_1 \rightarrow^* ID_2$ iff *M*, when run from ID_1, necessarily reaches ID_2 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 $\{0^n 1^n \mid n \geq 1\}$

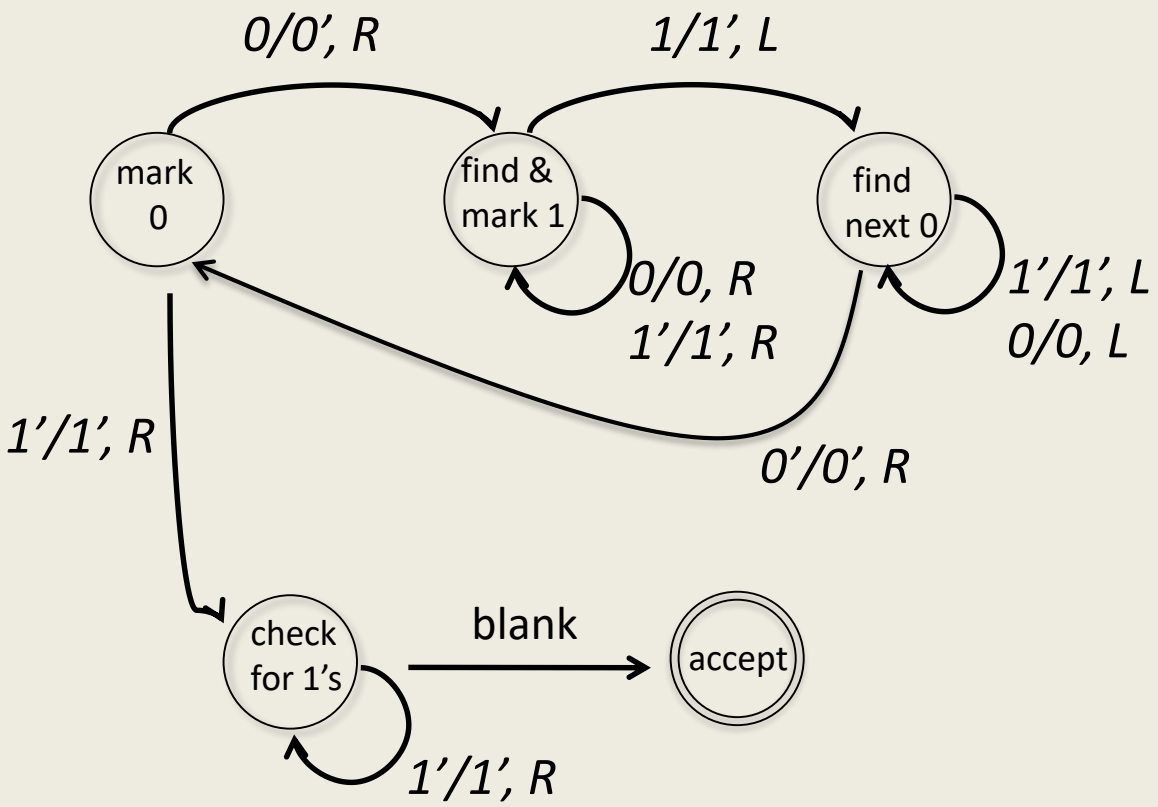
$0^* 1^*$

$0^n 1^n \in L$

$0^m 1^n \quad n > m$

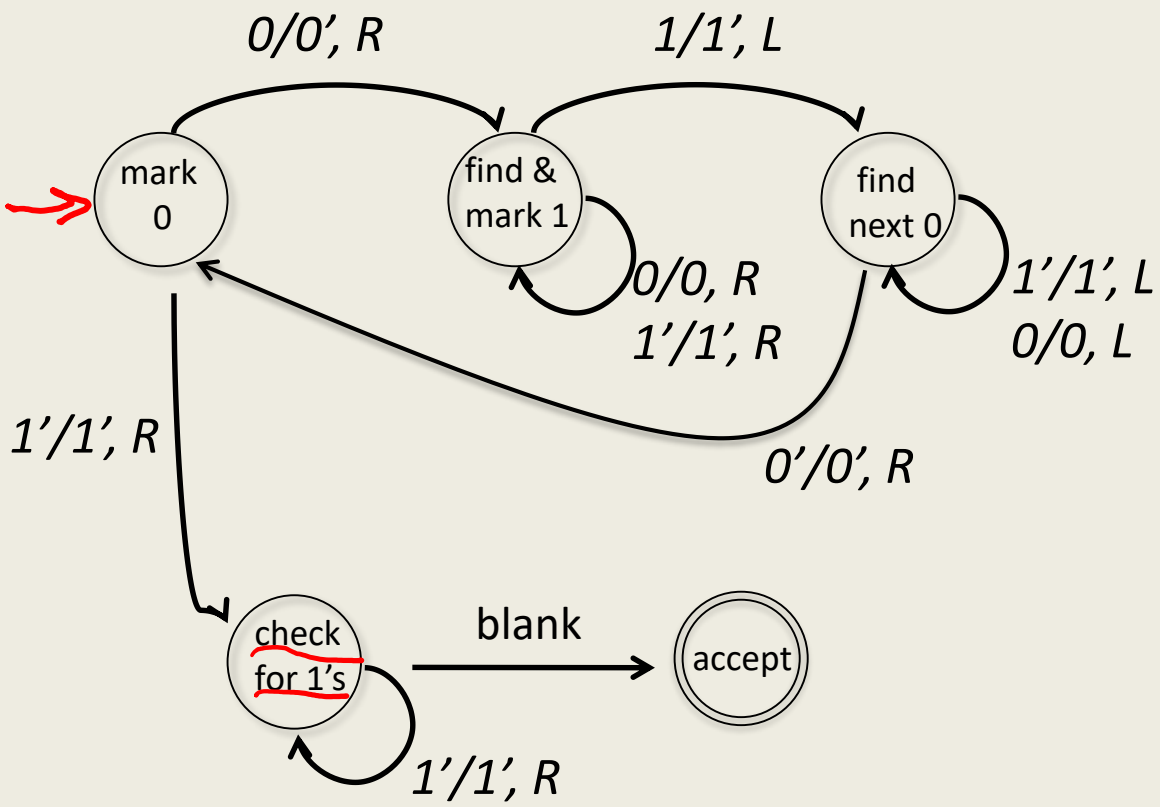
$0^m 1^n \quad n < m$

$w \in 0^* 1^*$



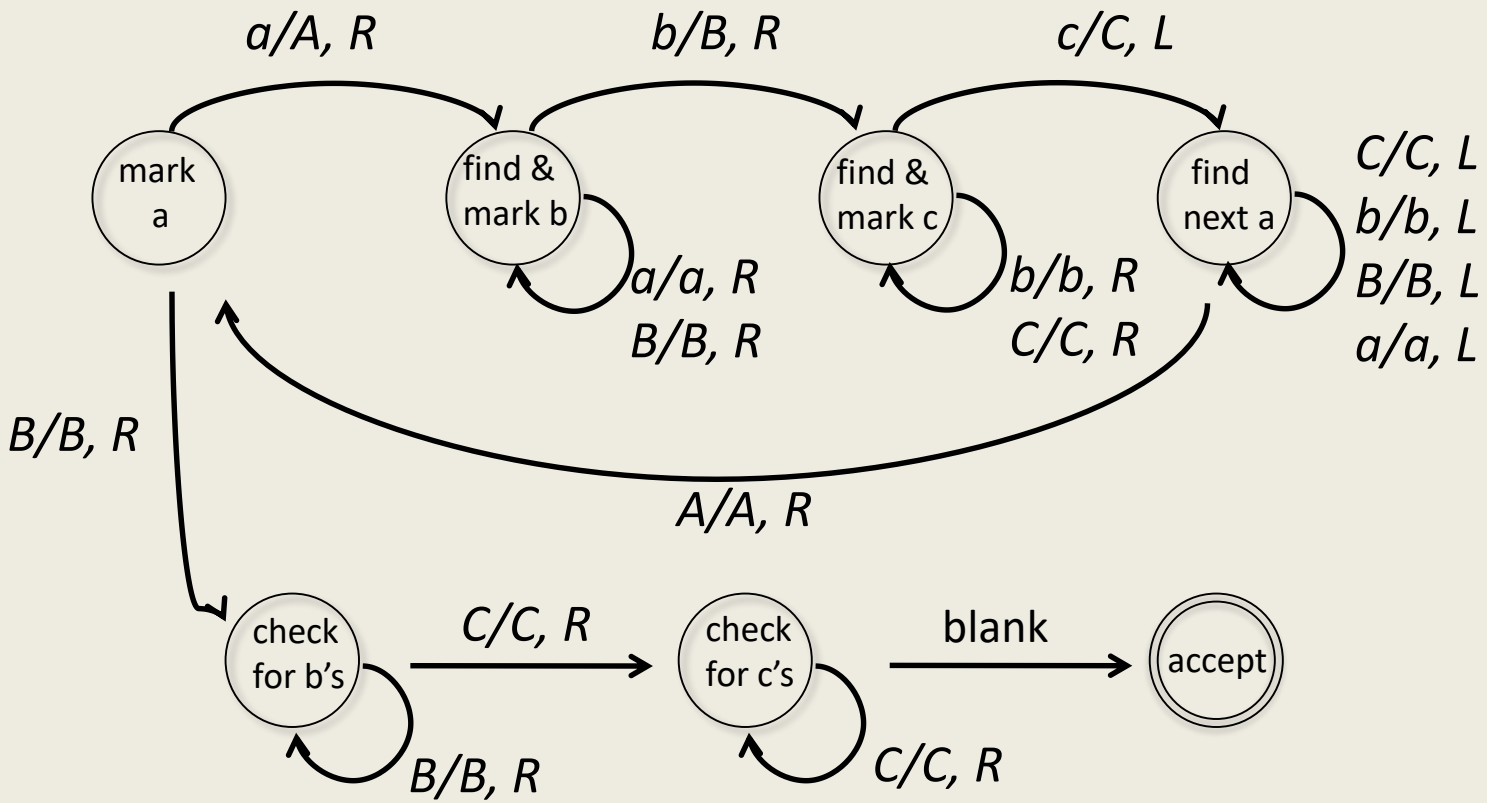
(This technique is known as "checking off symbols")

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



(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’

$1’0’0’1’0#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

Computing Functions with TMs

- number *n* represented in unary by 0^n

  (well, really 0^{n+1} so we can represent 0…)

- *M(n)* definition: "output of *M* on input *n*":

  IF $q_0 0^n \rightarrow^* q_{halt} 0^m$ then $M(n) = m$.

- Every TM *M* computes some function

  f_M : N → N U {undefined}.

- Functions with multiple inputs and outputs:

  $M(x,y,z) = (r,s)$ means $q_0 0^x \# 0^y \# 0^z \rightarrow^* q_{halt} 0^r \# 0^s$

"Easily" Computed Functions

- addition
- multiplication
- subtraction (max {0, x-y})
- any function you have an "algorithm" for...

Computable Functions

- A (partial) function f: N → N U {undefined} is said to be *computable* iff for some TM M,

  for all x in N, $f(x) = M(x)$ when $f(x)$ is defined

  "f is a (partial) recursive function)"

- A function f: N → N is a *total recursive function* iff for all x in N, $f(x) = M(x)$ for every x.

- If M computes a partial recursive function, it may not halt on some inputs. If M computes a total recursive function, it must halt for all inputs.

Why only f: N → N ?

Q: What about negatives, rationals, reals?

A: We can encode anything as a natural number.

- -5: 100000
- *p/q:* 0^p110^q, *(p/q) + (a/b) as* $0^{pb+aq}110^{bq}$
- reals to given precision, or symbolically
- Intuition: *ANY* function can be coded as a function from N to N

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:

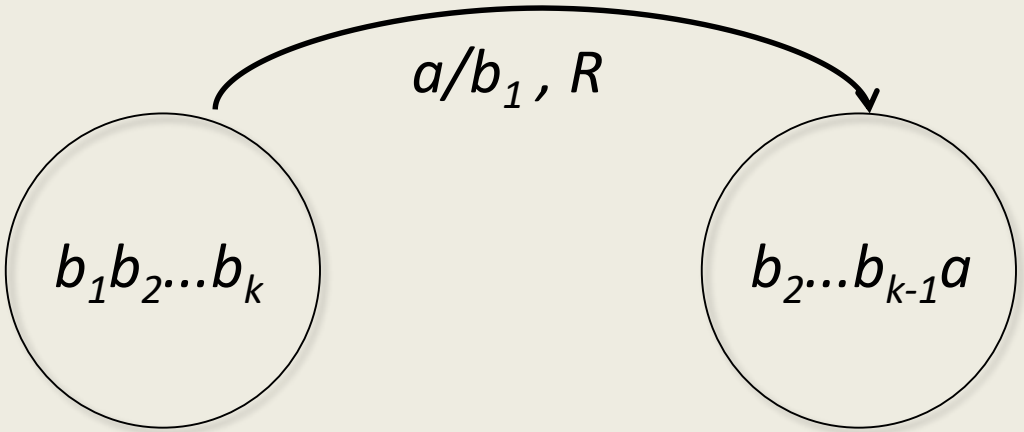
Convince you of the power of the basic model

Checking off symbols

- Use additional tape symbols to represent a "checked-off" character.
- E.g., for each symbol *a* in Γ, also include "$a_{\surd}$" to represent a marked *a.*
- We essentially did this using *A* for checked *a,* or 0' for checked 0, in our previous examples.

Shifting over

- Sometimes need extra cells
- Can shift-over by any number of cells
  - Shift-by-k: Use states to remember previous $|\Gamma|^k$ symbols:



plus states to begin and end the process

Implication of Shifting

- Can assume without loss of generality that input is in the form $w where $ is a special symbol at start of input and not used anywhere else
- With above assumption can avoid the issue of crashing because of moving off the left of the tape

Using finite control

- just like DFAs
- can use tuples to store different types of info
- E.g., $\{a^n b^n c^n \mid n = 1 \pmod 4 \text{ and } n = 2 \pmod 7\}$

  States: (q, i, j) where:

  q: state from TM for $\{a^n b^n c^n \mid n \geq 1\}$

  i: counter mod 4

  j: counter mod 7

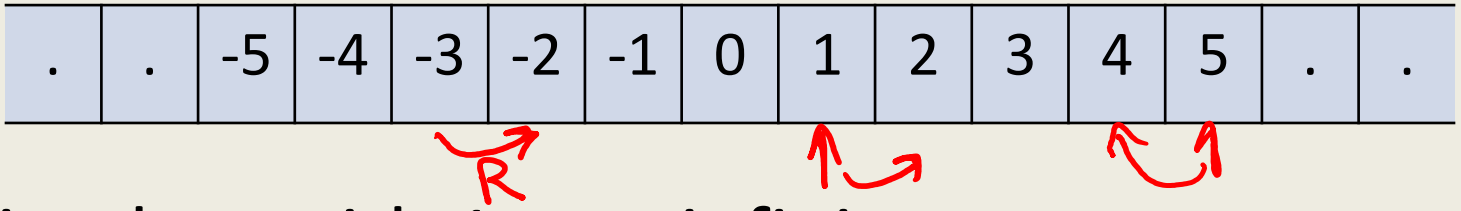
  etc.

In general, can store *any finite information* in states

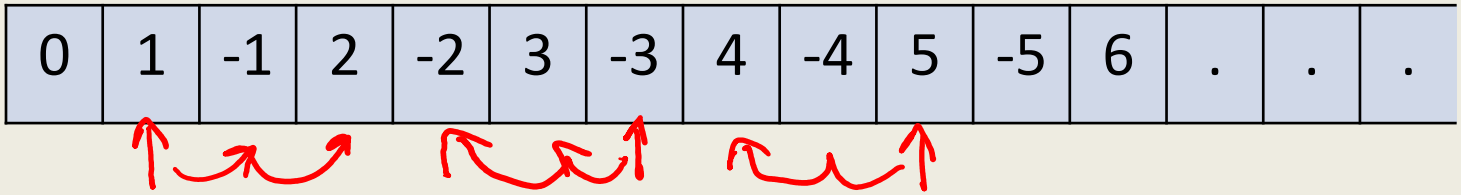
Subroutine calls

- soon

"Extensions" of TMs: 2-way infinite tape



Simulate with 1-way infinite tape...



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

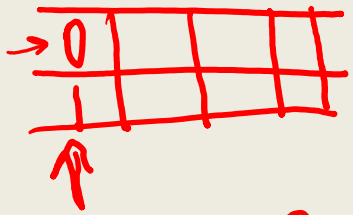
Extension: multiple tracks

4 tracks

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

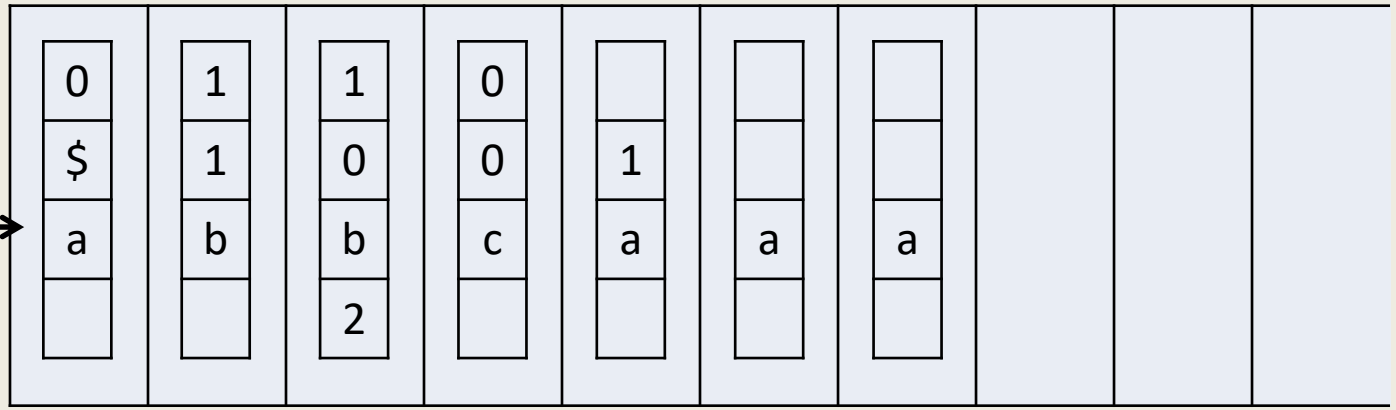
M can address any particular track in the cell it is scanning

infinite tape →



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

single character →



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' $\delta(q,$ [x, 0, y, z] $) = (p,$ [x, 0, y, 1] $, R)$

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

$$\delta(q, a_1, a_2, \ldots a_k) = (p, b_1, b_2, \ldots b_k, D_1, D_2, \ldots D_k)$$

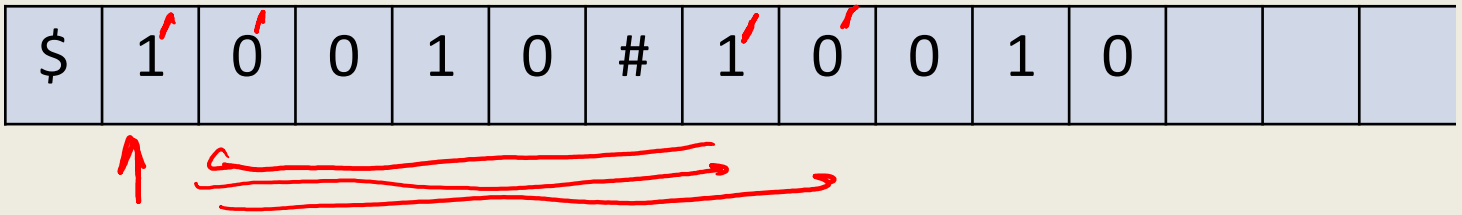
Symbols scanned on the *k* different tapes

Symbols to be written on the *k* different tapes

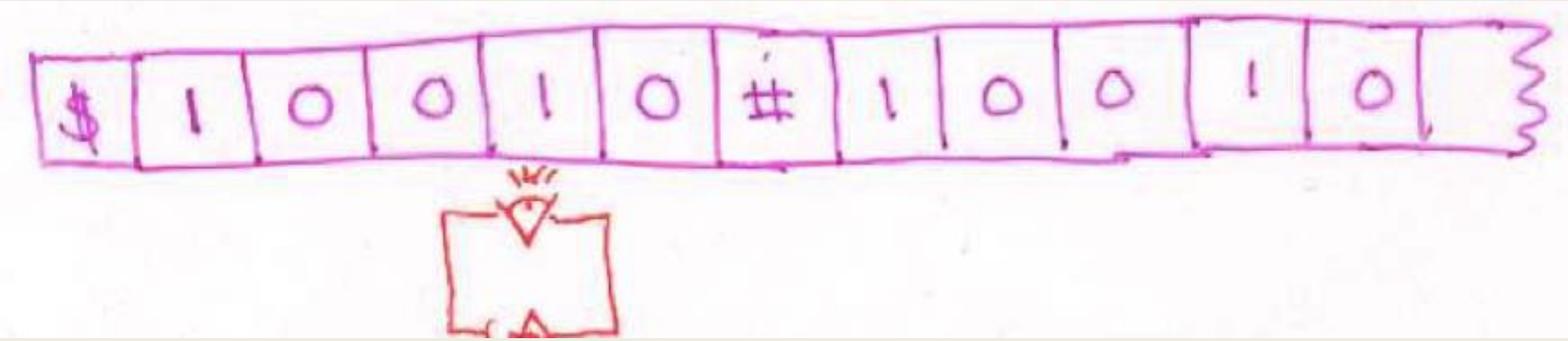
Directions to be moved (D_i is one of L, R, S)

Utility of multiple tapes

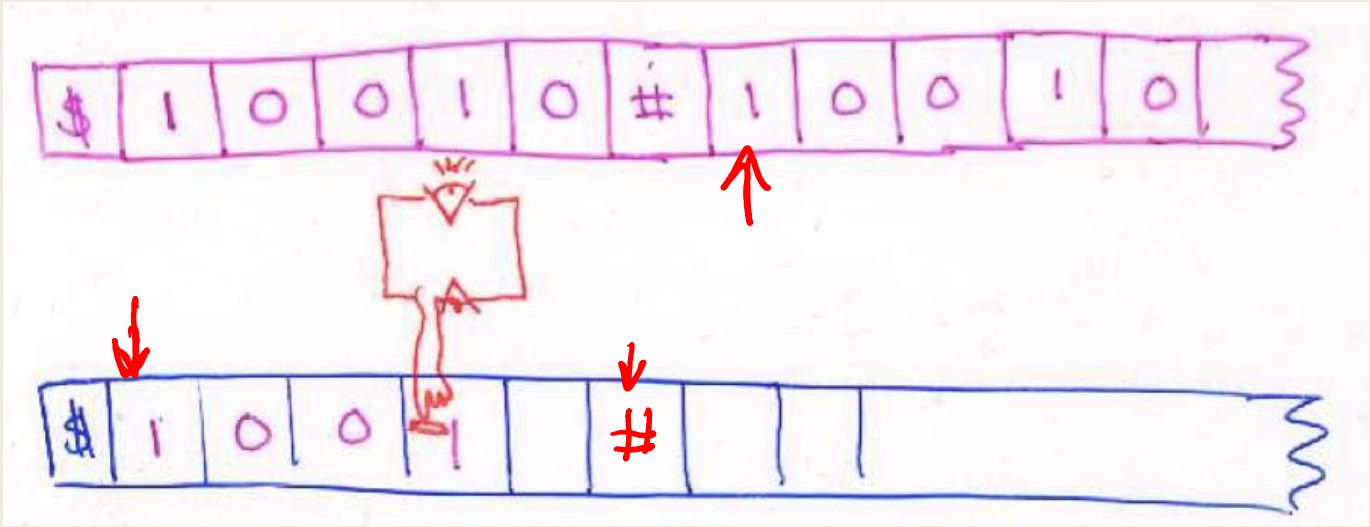
makes programming a whole lot easier



is input string of form w#w ?



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

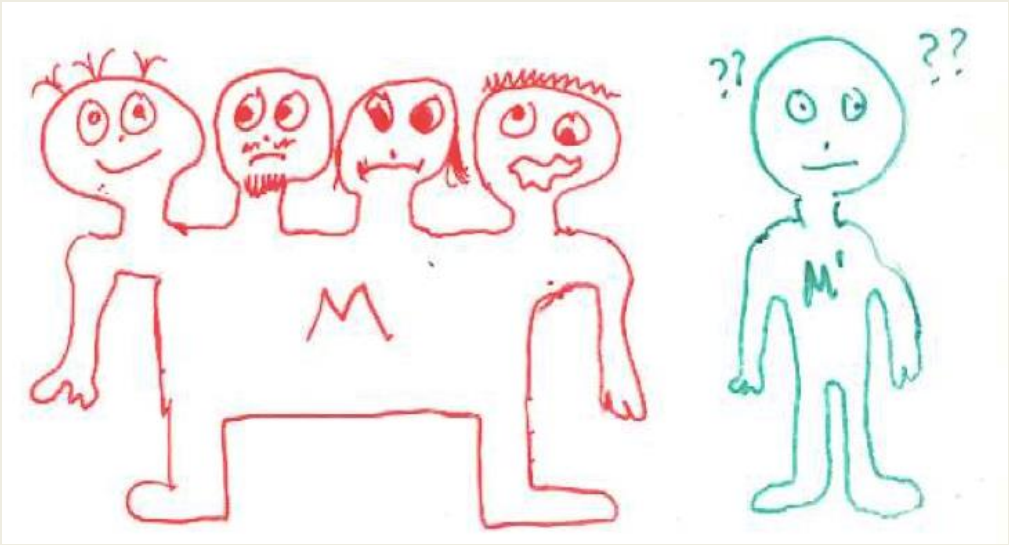


≈ 3n/2 steps easily programmed

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'.

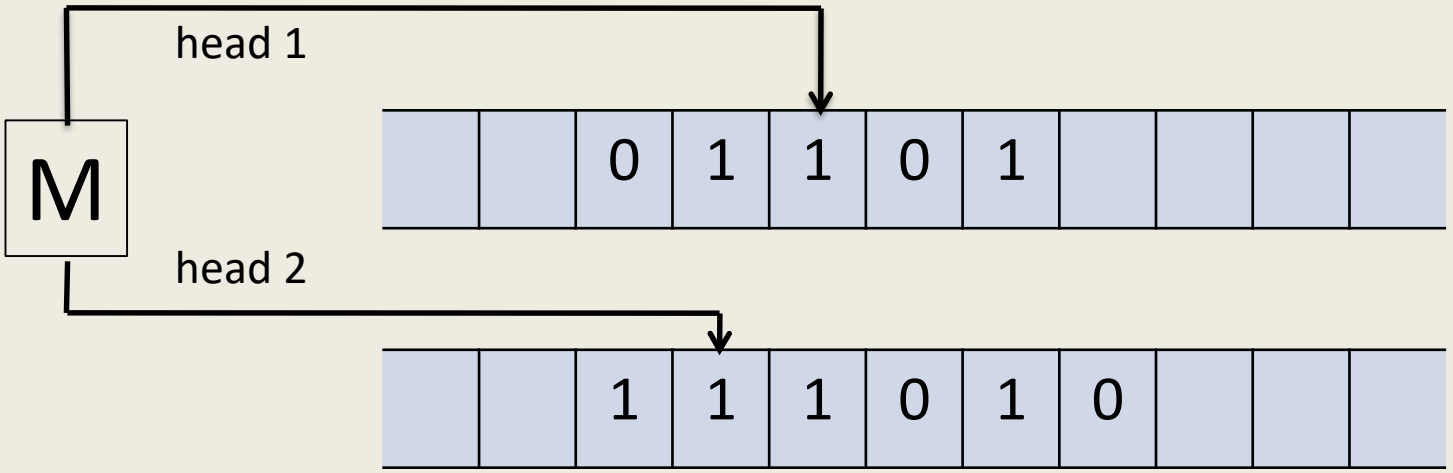
Intuition: M' uses 2k tracks to simulate M

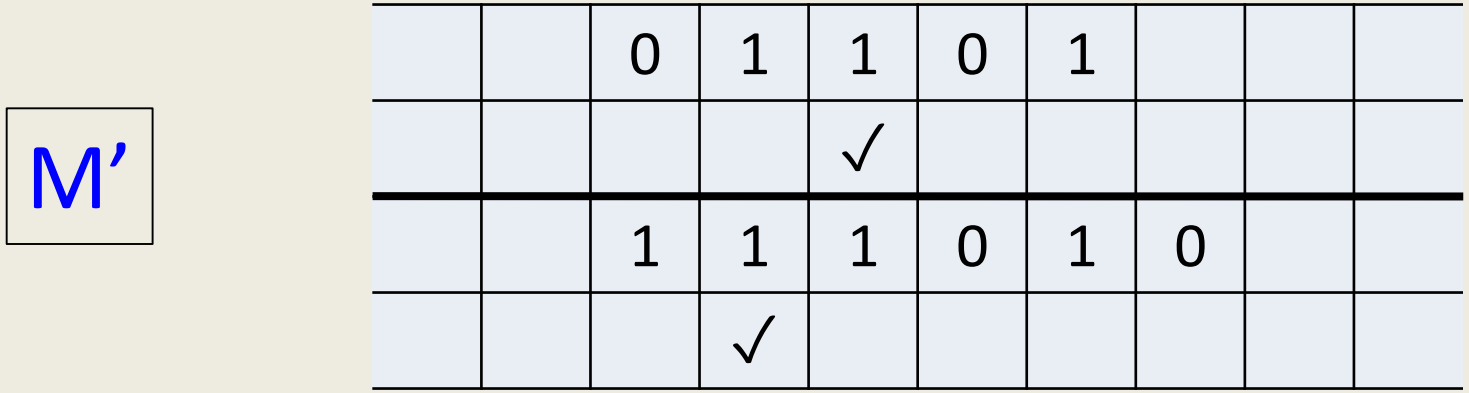


BUT....
M has k heads!

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

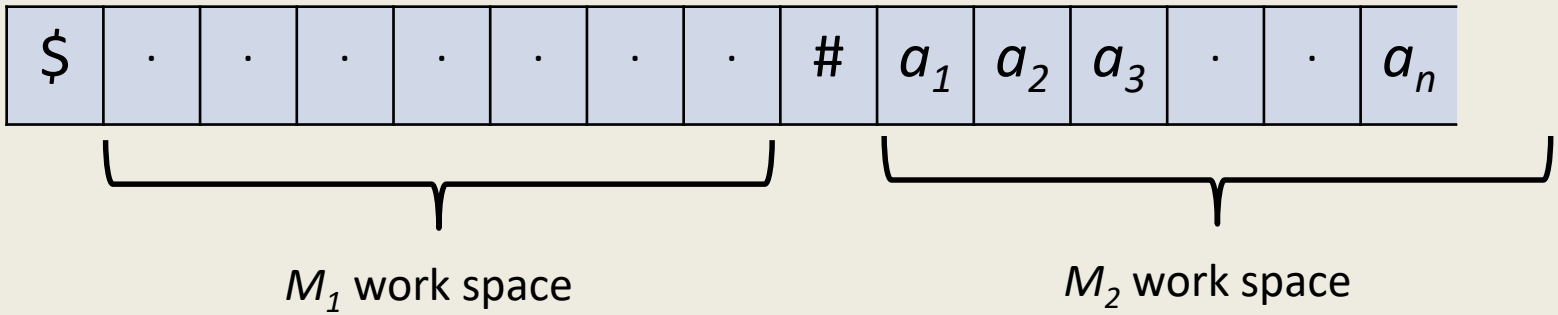
Thus, each move of *M* requires *M'* to do a complete sweep across, and back.

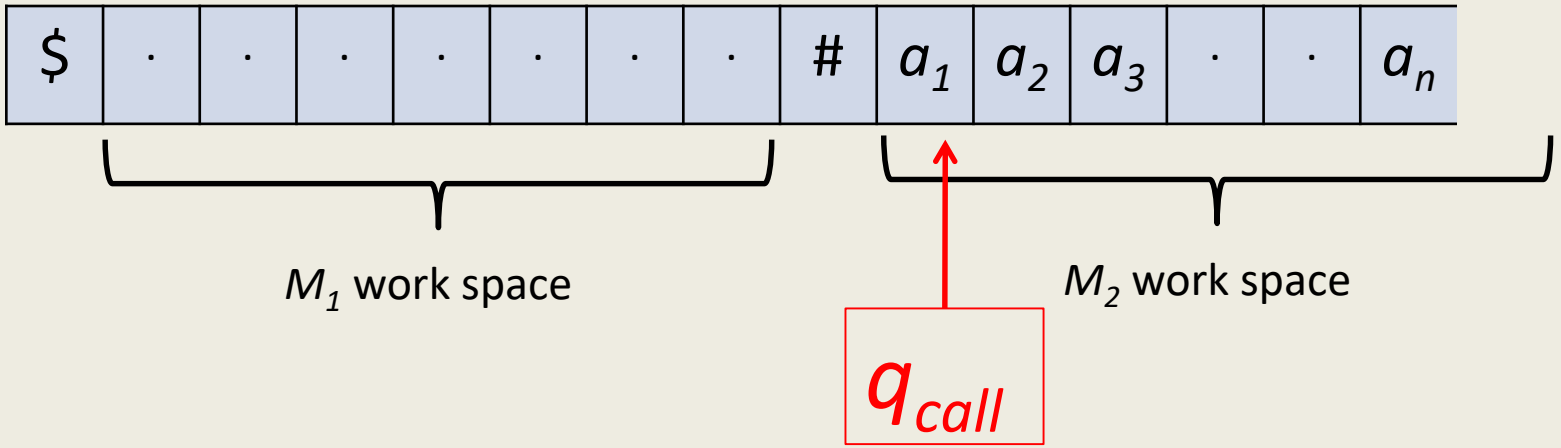
Not hard to show that if *M* takes *t* steps to complete its computation, then *M'* takes $O(t^2)$ steps.

Subroutine calls

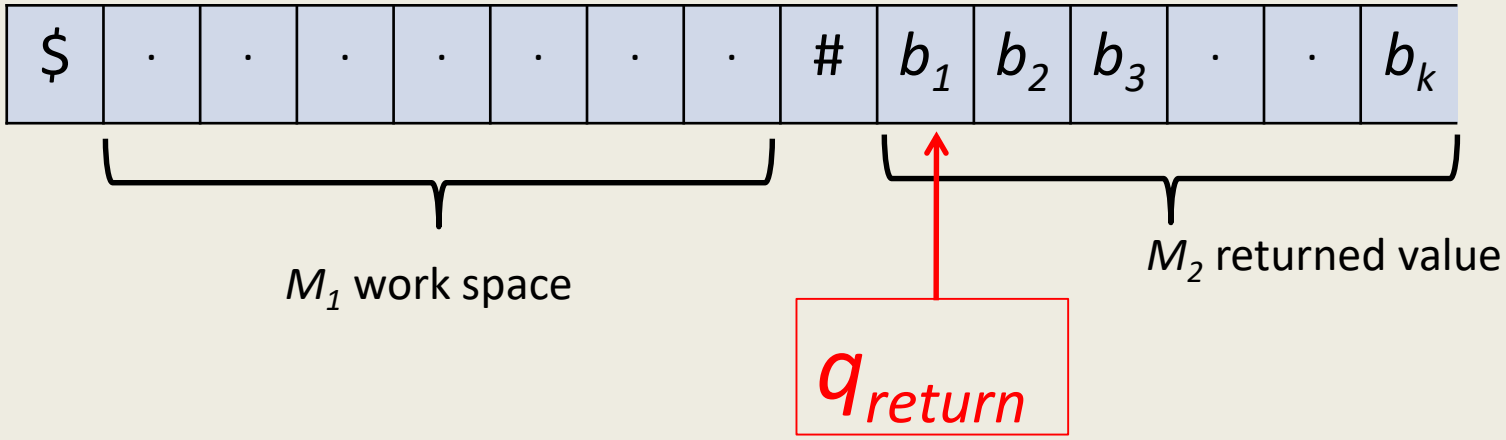
Mechanism for M_1 to "call" M_2 on an argument

- Rename states so that M_1 and M_2 have no common states *except* q_{call} and q_{return}
- Goal: M_1 calls from state q_{call} returns to q_{return}
- Rename init. state of M_2 as q_{call} ,halt state q_{return}
- M_1 sets up argument $a_1a_2...a_n$ for M_2 :

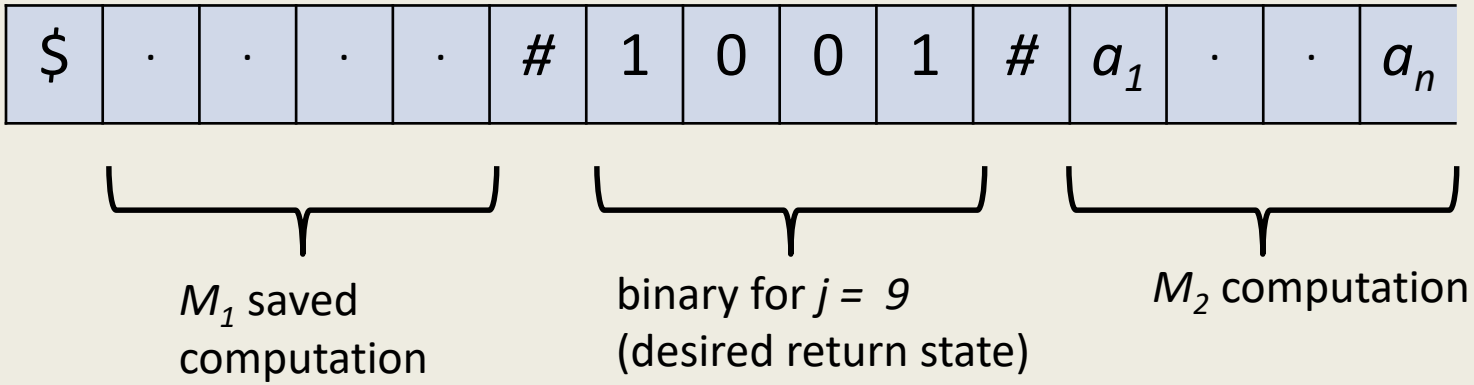




- M_2 runs, and when done:



- Can be more elaborate, and return to specified state q_j



- q_{return} now goes to special sequence of states designed to read binary 1011 (or whatever) and then transition to state 9 (or whatever)

  - This can actually be done with a DFA