Turing Machines

Robert Keller
November 2008

Mathematical Machines

- "Mathematical" (as opposed to mechanical) machines
  - Turing Machines (potentially infinite-state)
  - Finite-state machines
  - Other categories (cf. CS 81, 142)

What is a Turing Machine?

- A computational model thought to be *universal* from the viewpoint of **functions** that can be computed
- Proposed by Alan M. Turing as a means of discussing such functions
- Universality generally accepted by computer scientists based on Turing's argument (see text) and other evidence

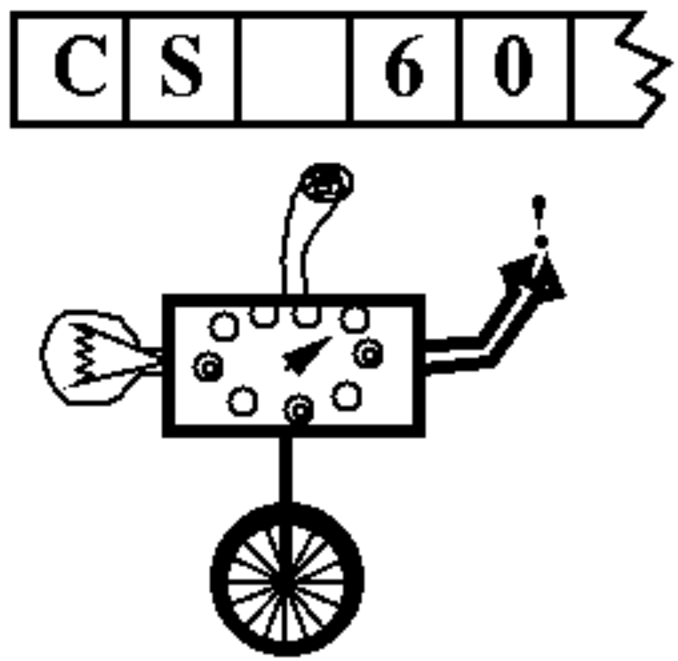
Alan M. Turing (1912-1954)

Turing Milestones

- 1936: Essay on computability
- 1940: Machine ("the Bombe") for decrypting the German *Enigma* code machine
- 1943: Participated in design and construction of an electronic computer (the "Colossus")
- 1949: First paper on proving correctness of programs
- 1950: Paper on AI ("the Turing test")
- 1951: Biological pattern formation ("morphogenesis")

The Enigma (from NSA museum)

Another Enigma?



Play about Turing

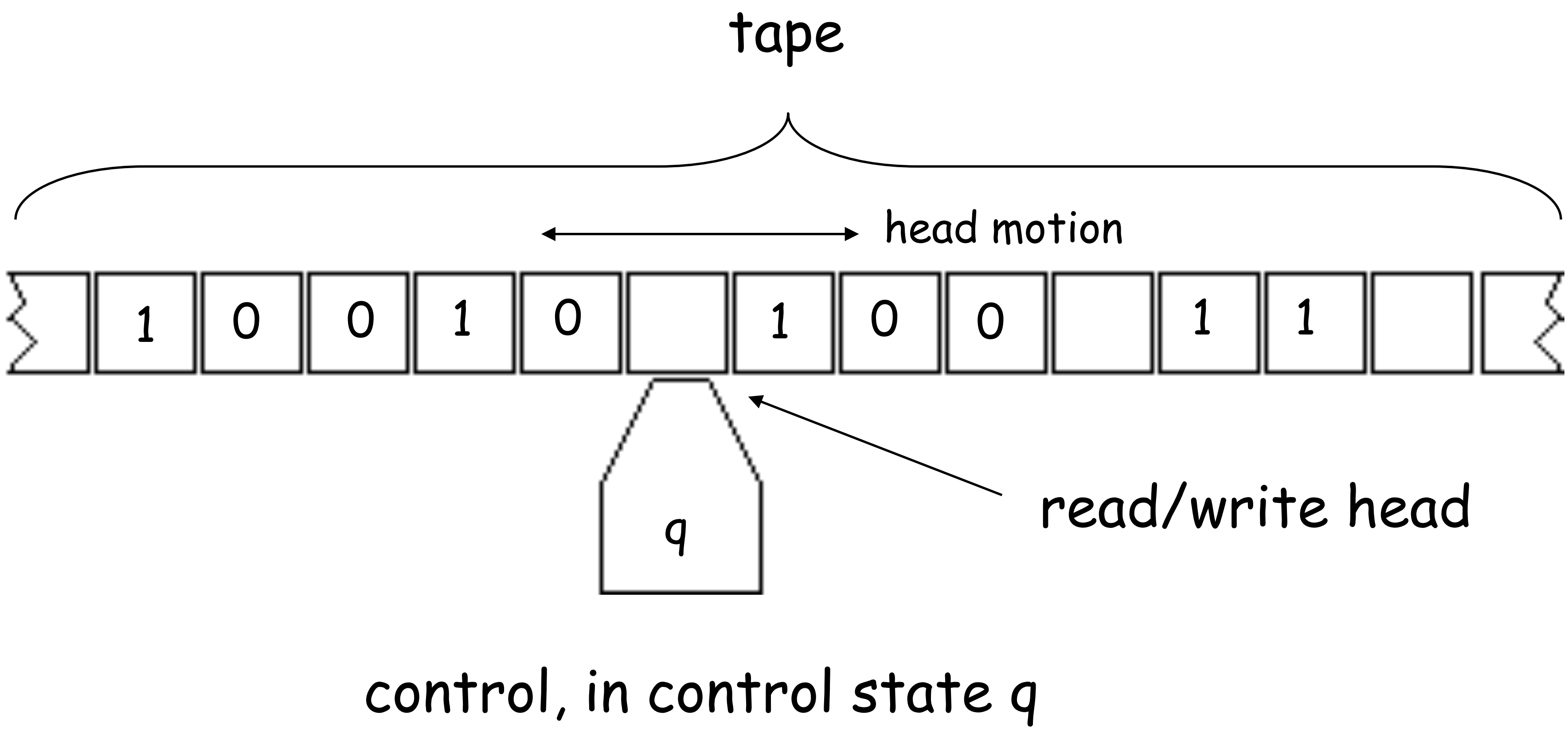
- "Breaking the Code" by Hugh Whitemore
- Played in London, New York, LA
- Public TV version

Turing Machine Details

- The *tape*: an unbounded amount of memory. Consists of *cells*, each containing exactly one of a pre-convened set of characters (such as '0', '1', ' ' blank)

- The control: a finite amount of memory, the *control states*. Defines *control functions*.

Turing Machine



More about the Tape

- Only a finite portion of the tape is "non-blank" at any time.
- New cells are added at either end "as needed".

More about Blank

- Blank counts as symbol.
- It may be regarded as 0, or it could be separate from 0.
- It all depends on the convention being used.

The Complete State of a TM

- The complete state of a TM is determined by:
  - The control state
  - The symbol currently under the head
  - The sequence of symbols to the right of the head
  - The sequence of symbols to the left of the head

Control Functions

- The control determines the following, given any combination of control state (q) and symbol under the head (s):
  - A new control state (q')
  - A new symbol to be written (s')
  - A head motion m (Left, Right, or None)

- Call the control partial-function f, so that
  $$f(q, s) = (q', s', m)$$

Caution: "state" is an overloaded term

- The **true state** of a TM is the combination of:
  - The control state
  - The tape to the left of the head
  - The tape under and to the right of the head.
- However, "state" is often used in referring to just the control state.
- The ambiguity is resolved by context.

Sequential Operation

- The machine begins in a specified starting control state, with initial tape contents, and the head positioned at a standard place with respect to the contents.
- The machine goes through a sequence of states until it arrives at a halting state.

Halting Convention

- If f(q, s) is unspecified, then the TM is said to have ***halted*** in the current state.

5-tuple notation

- The control partial-function f, so that

  $$f(q, s) = (q', s', m)$$

  is often written as a set of 5-tuples of the form:

  $$(q, s, q', s', m)$$

TM Simulator in Scheme

- Refer to file

Various TM Categories

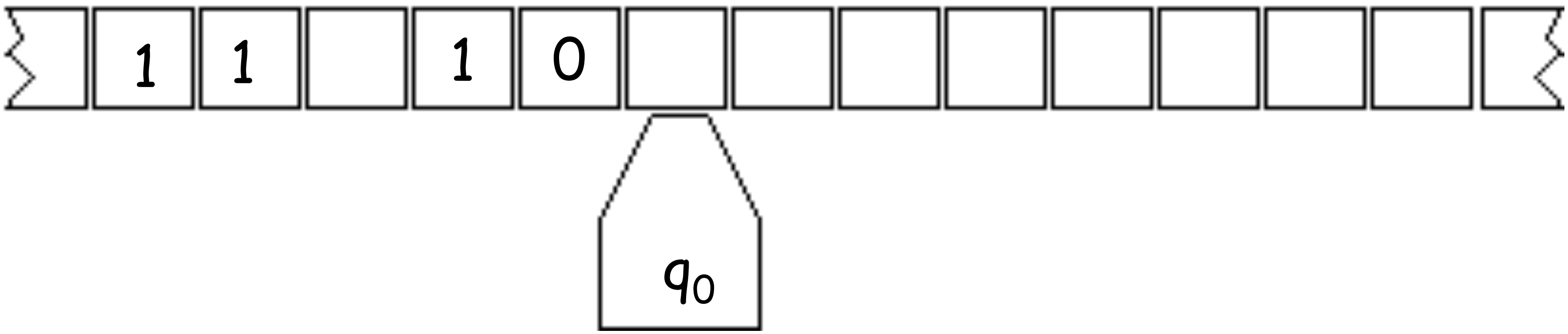
- **Transducer**: Starting with the initial tape contents, produce a new tape contents
- **Acceptor or Classifier**: Starting with the initial tape contents, halt in either an accepting state or a rejecting state
- **Generator**: Starting with an empty tape, generate the elements of some sequence on the tape.

Examples of TM Categories

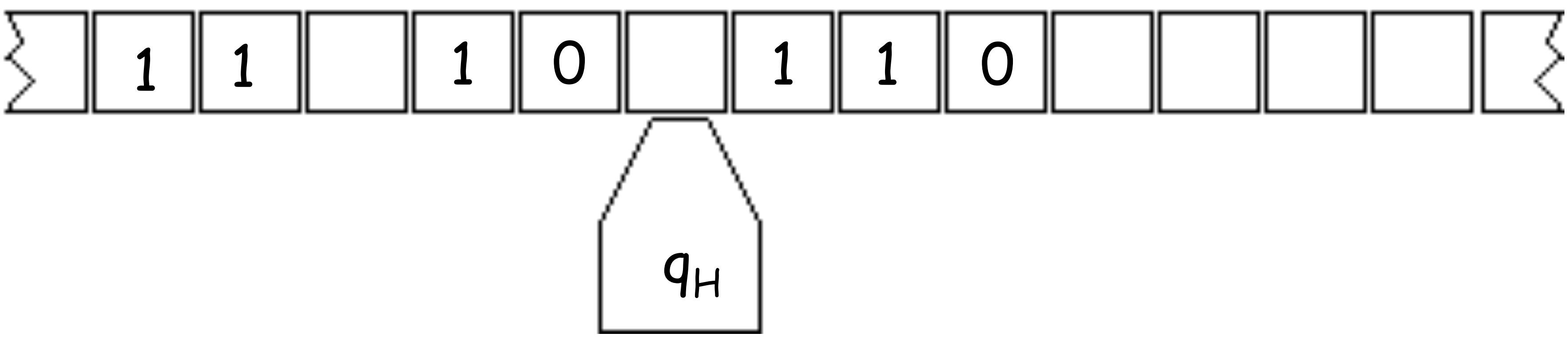
- **Transducer**: Multiply two binary numerals
- **Acceptor or Classifier**: Determine whether or not a binary numeral is prime
- **Generator**: Starting with an empty tape, generate numerals for the primes, each separated by a blank

TM Multiplying Example

Initial



Final

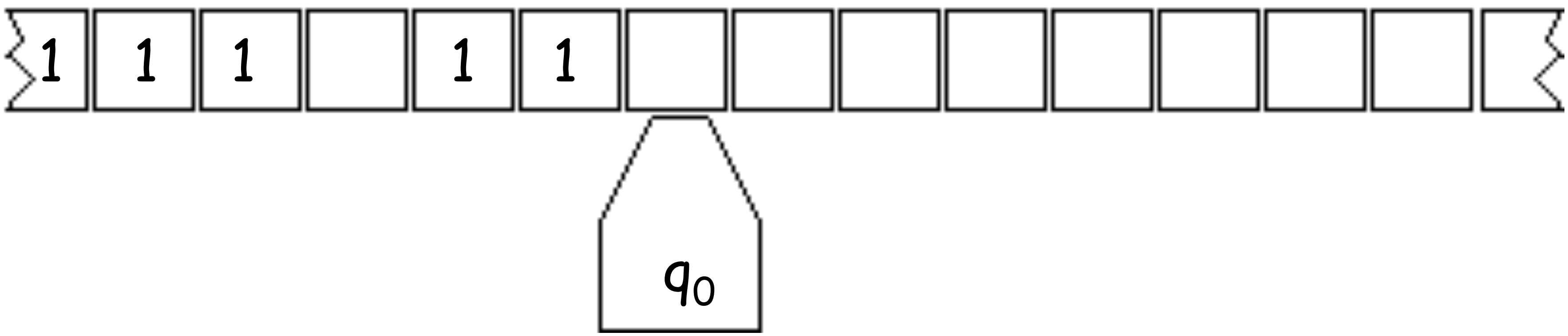


Simplifying TM Programming

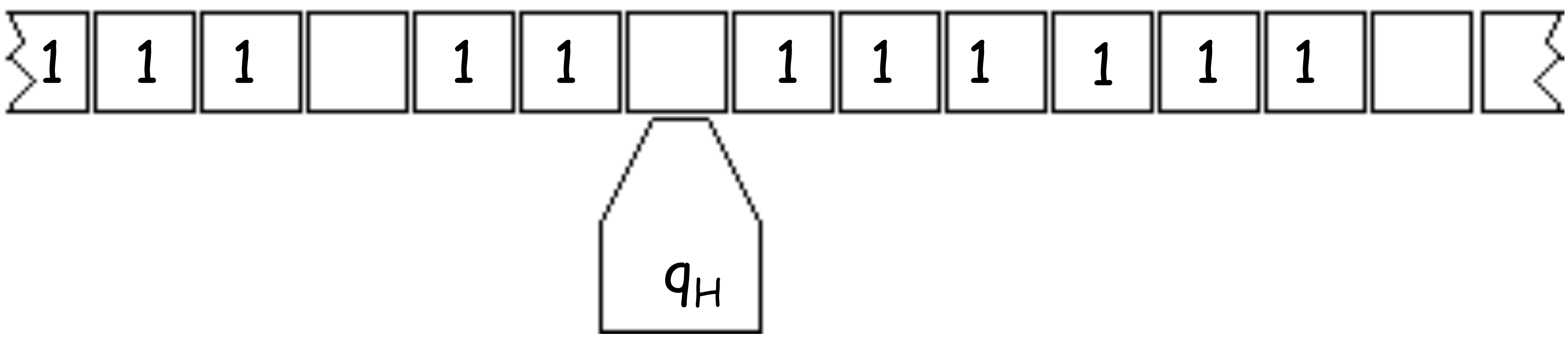
- Allow symbols to be erased
- Use extra symbols: can always convert to fewer symbols later (by encoding the larger set)
- Use symbols with/without **markers**: these in effect are just a larger symbol set
- Use special encodings, such as 1-adic encoding (number n is n 1's)

1-adic Multiplying

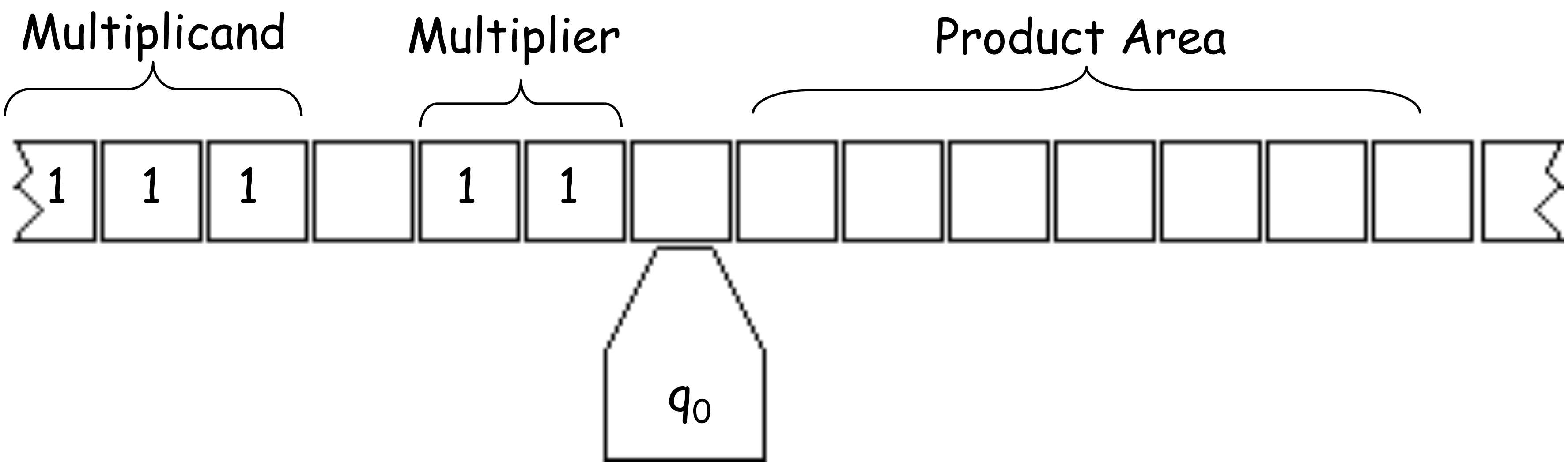
Initial



Final



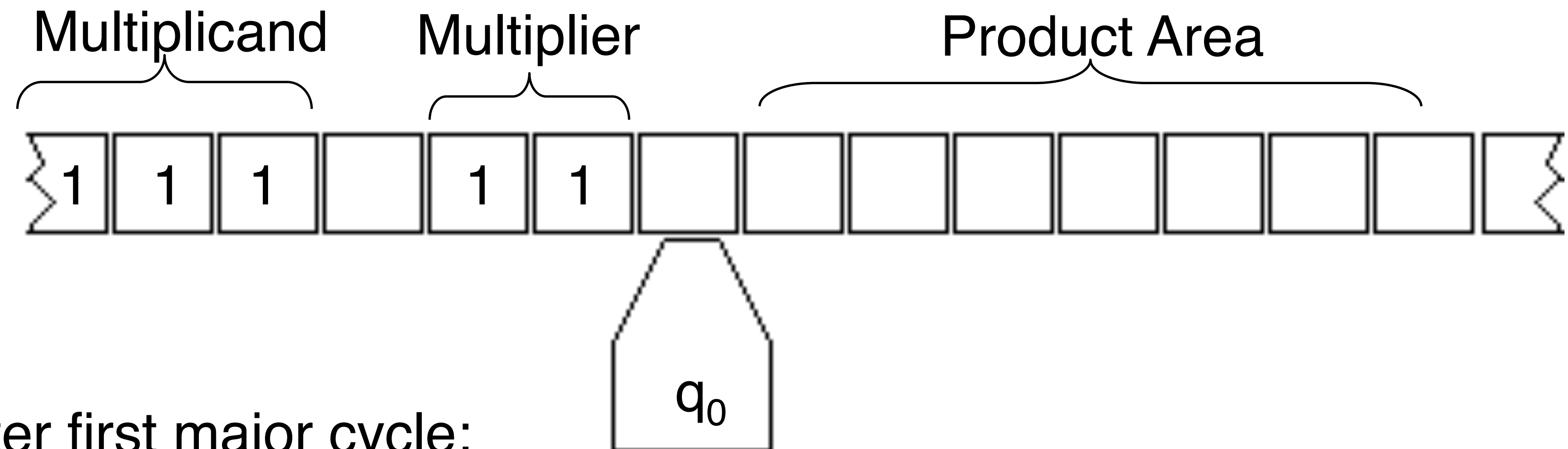
1-adic Multiplier Structure



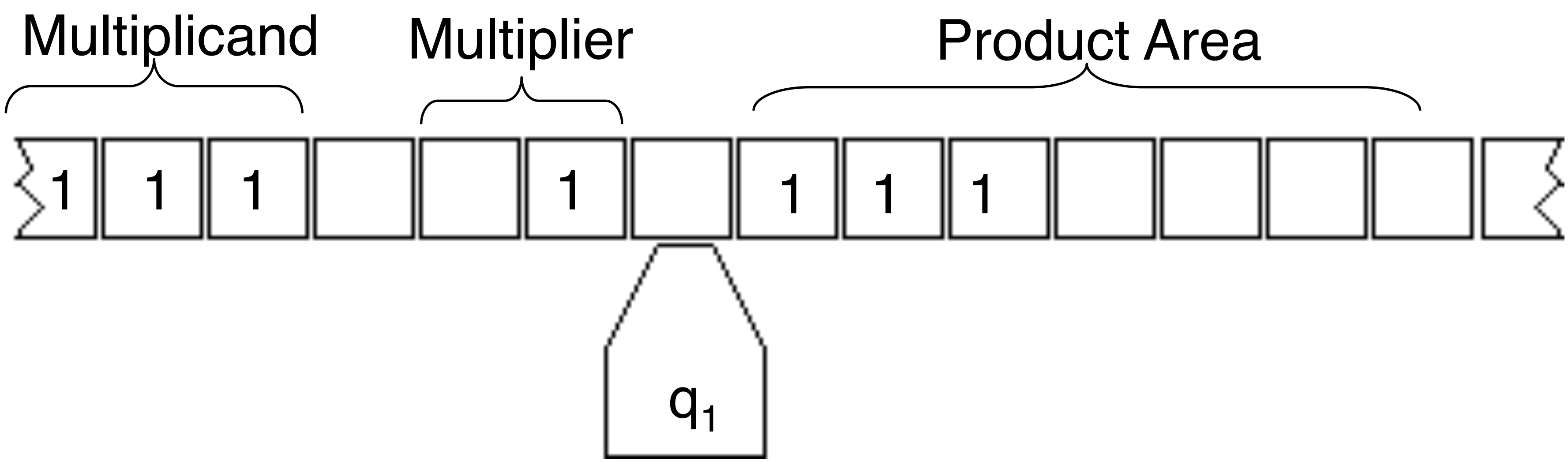
1-adic Multiplier Plan

- Check whether the multiplicand is 0 (no 1's); if so, return to home position and halt.
- For each 1 in the multiplier:
  - Copy the multiplicand to the right of the accumulated product
  - Erase the leftmost 1 in the multiplier
- until all multiplier 1's have been erased
- Then restore the multiplier 1's and halt.

1-adic Multiplier In Operation

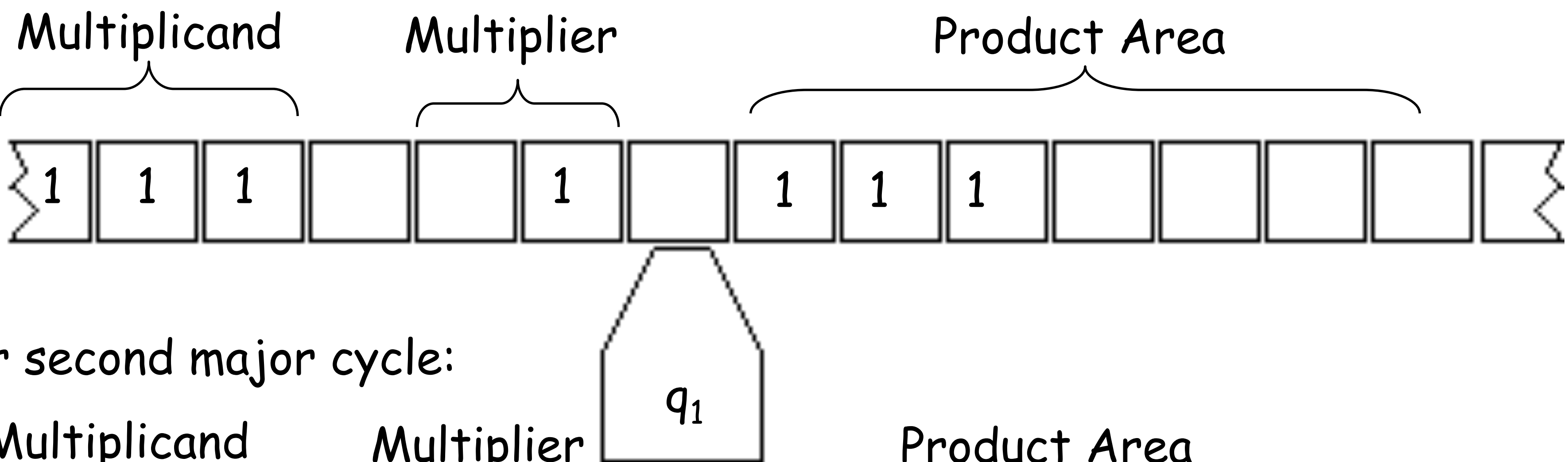


After first major cycle:

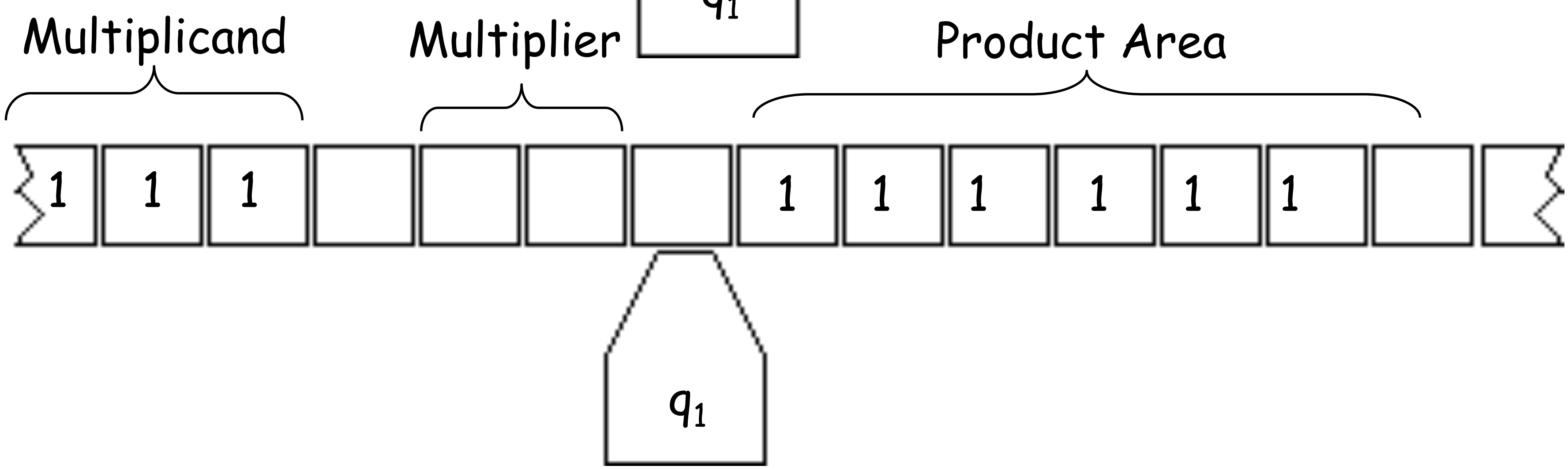


1-adic Multiplier In Operation

After first major cycle:



After second major cycle:



How to tell when done?

- Each time the multiplicand is copied, the leftmost 1 of the multiplier will be set to blank (it will be restored at the end).
- Moving left from q_1, if there is a 1 then the multiplier has not been decimated.

How to copy the multiplicand?

- This is tricky, because the multiplicand can be arbitrarily long; we cannot "count" arbitrarily-high in the control of the machine alone.
- During copying, make each 1 of the multiplicand into a 0. At the end of copying, turn all 0's back to 1's.
- The machine is done copying when there are no 1's left.

Turing's Hypothesis

- Turing's Hypothesis is that for every computable function there is some Turing machine that computes it.
- Turing's argument was detailed and based on a direct appeal to intuition.

Practical Use of Turing's Hypothesis

- If we can state an algorithm for doing something, there is a way to program a Turing machine to perform that algorithm.

Impossibility of proving Turing's hypothesis

- In order to *give* a sound proof of the hypothesis, it would be necessary to characterize precisely what it means to be "computable".
- This would entail presenting another convincing notion of computability, which would have to be argued to Turing's original argument.
- Most computer scientists and mathematicians accept Turing's notion as the notion.

Possibility of disproving Turing's hypothesis

- Disproof is possible, although not likely.

- A disproof would involve finding an example of a function that is clearly intuitively computable, then proving that no Turing machine can compute it.

Turing's Hypothesis

- Many other notions of computability have been proposed, some more natural than others:
  - Partial recursive functions
  - General recursive functions
  - Lambda calculus
  - Markov algorithms
  - Uniform register machines
  - . . .
- All such notions have been proved equivalent to Turing machines through appropriate encodings.

Divergence

- A Turing machine is said to **diverge** on an input if it **never halts**.
- Divergence is like a program that never terminates, e.g. either due to an infinite loop or a search that can never yield an answer (but maybe we don't know that).
- If a machine diverges, the partial function it computes is **undefined** for this input.

Examples of Divergence

- If a TM, starting on a given input, returns to a complete state a second time, then it will return to that state an infinite number of times, and thus diverge.

- If a TM "searches" for a number having a certain property, but there is no such number, then it will diverge.

Example of Properties for which divergence is not known

Let P(n) be the property:

"2n is expressible as the sum of two primes"

Is there an $n > 1$ for which P(n) is false? It is not known at present (1 December 2008).

P(n) is easily testable by a program.

It is similarly easy to construct a program that will determine an n for which P(n) is false, if there is such an n.

Example of Properties for which divergence is not known

- The assertion that P(n) is true for all n > 1, i.e.

  "Every even integer > 2 is expressible as the sum of two primes"

  is known as "Goldbach's Conjecture" (1742).

- Examples: 4 = 2+2
  6 = 3+3
  8 = 3 + 5
  . . .
  14 = 3 + 11
  . . .
- It has been checked for all $n \leq 10^{18}$.

Encoding Turing Machines into a fixed alphabet

- Why do we want to encode?
  - For input to a Universal machine.
  - To prove interesting results.

Encoding

- Every TM and its tapes can be encoded into the alphabet {1, _}.

- First the tapes.
  - Suppose the tape alphabet is {b, a, c, d}, where b plays the role of blank.
  - Encode each symbol as follows:

    b as _ _
    a as _ 1
    c as _ 1 1
    d as _ 1 1 1

Encoded tape

- Suppose our tape is:

  a c d a a b . . .

- Then the encoding is:

  _ 1 _ 1 1 _ 1 1 1 _ 1 _ 1 _ _ …

Encoding the Machine

- Encode the state as:
  _1, _11, _111, _1111, … to whatever number of states we have.

- Encode the moves as:

  L is _1

  R is _ 1 1

  N is _ 1 1 1

Encoded tuples

- Suppose the 5-tuple is:
  (s1 a c L s2)

  The encoding would be:
  _ 1 _ 1 _ 111 _11 _11

  Encode a list of 5-tuples by concatenating their encodings.

Interpretation of Encodings as Numerals

- For convenience, we can think of strings of _ and 1 as numerals (representations of numbers).
- 0 = _
- 1 = 1
- 2 = 1 _
- 3 = 1 1

etc.

Default TM

- For an arbitrary number n, its numeral may or may not represent an encoded TM as described.

- For those n that do not, adopt the convention that they represent a special **default machine**, that does nothing.

Default Tape

- Similarly, some numbers represent valid encoded tapes, while others do not.
- Consider the ones that do not to be encodings of the all-blank tape, by convention.

Enumeration

- Because the strings over {_, 1} can be enumerated in a **natural** order, as given by the numbers they encode, there is a Turing machine T_n for each number n, and a tape x_n for each n as well.

Universal Turing Machine

- Consider a TM U constructed to operate on the encoding alphabet {1, _}. When this machine start with its tape having an encoding of machine T_n and an encoding of some tape x_m, it **simulates** the moves on , in the sense that
  - U halts on <T_n, x_m> (the encoding of T_n and x_m) iff T_n halts on x_m.
- A universal TM can simulate any TM, including itself.

Non-Computable Functions

- There are more partial functions, say, from the natural numbers to themselves, than there are Turing machines:
  - The infinite set of functions N -> N is *not* countable (can't be enumerated).
  - Even the set of functions N -> {0, 1} is not countable (Cantor, 1891).
  - The infinite set of Turing machines *is* countable (can be enumerated).
  - So there are functions in N -> {0, 1} that are not computable.

Diagonalization

- We know that for each natural number n, there is a Turing machine T_n and a tape x_n.
- Conceptually create the following infinite array:
  - The rows correspond to T_0, T_1, T_2, …
  - The columns correspond to x_0, x_1, x_2, …
  - There is a 1 in row i column j iff T_i accepts x_j, otherwise there is a 0.
  - Consider the **diagonal** of this array. Flip the 1's and 0's. The corresponding sequence cannot be a row of the array. Hence, the flipped diagonal describes a function not computed by **any** Turing machine.

The Divergence Problem

- Let T_0, T_1, T_2, … be an enumeration of all Turing machines.
- Similarly let x_0, x_1, x_2, … be an enumeration of all tapes.
- Define the partial function D as follows:

$$D(x_i) = \begin{cases} 1 \text{ if } T_i \text{ diverges on input } x_i \\ \text{diverges if } T_i \text{ halts on } x_i \end{cases}$$

- (x_i encodes the natural number i, as described before.)

The Divergence Problem

$$D(x_i) = \begin{cases} 1 \text{ if } T_i \text{ diverges on input } x_i \\ \text{diverges if } T_i \text{ halts on } x_i \end{cases}$$

- Claim: D is not computable by any Turing machine (in the sense that D(i) is the result of the machine T_i on x_i).
- Suppose instead that k is such that T_k computes D. Then

$$D(x_k) = \begin{cases} 1 \text{ if } T_k \text{ diverges on input } x_k \\ \text{diverges if } T_k \text{ halts on } x_k \end{cases}$$

- Does this look ok so far?

The Divergence Problem

- The thrust of the divergence problem is to state concretely a (partial) function that is not computable.

- It really is diagonalization expressed in a slightly different way.

A Similar-Looking Problem

$$H(x_i) = \begin{cases} 1 \text{ if } T_i \text{ halts on input } x_i \\ \text{diverges if } T_i \text{ diverges on } x_i \end{cases}$$

- Is this partial function computable?

The Halting Problem

- The "Halting Problem" is that of devising a computable function that will tell whether or not (yes or no) Turing machine T_n halts on tape x_m, i.e. whether H is computable:

$$H(n, m) = \begin{cases} 1 \text{ if } T_n \text{ halts on } x_m \\ 0 \text{ otherwise} \end{cases}$$

- If the Halting Problem were solvable, so would the Divergence problem be. We say the Divergence Problem **reduces to** the Halting Problem.

DP reduces to HP

- Simply note that:

$$D(i) = \begin{cases} 1 \text{ if } H(i, i) = 0 \text{ (i.e. } T_i \text{ fails to halt on } x_i) \\ \text{diverge otherwise} \end{cases}$$

So if we could compute H, we could compute D by embedding a call to H with just a little more "glue logic":

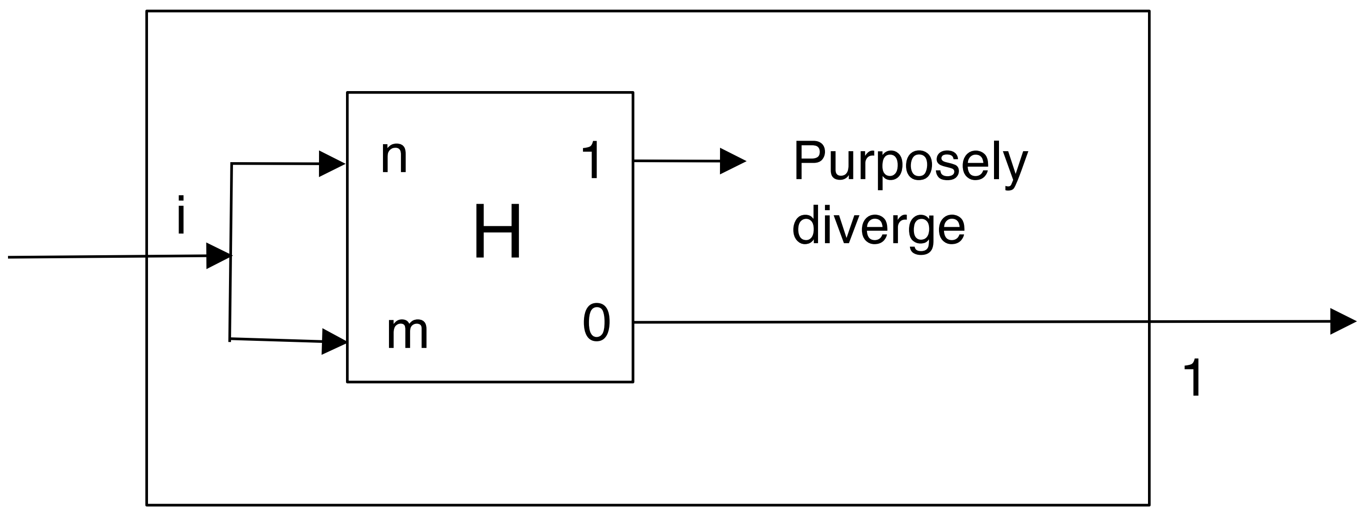
Replicating the argument i of D.

Checking whether the result of H(i, i) is 0.

DP reduces to HP

- How to implement D, given H:

D:

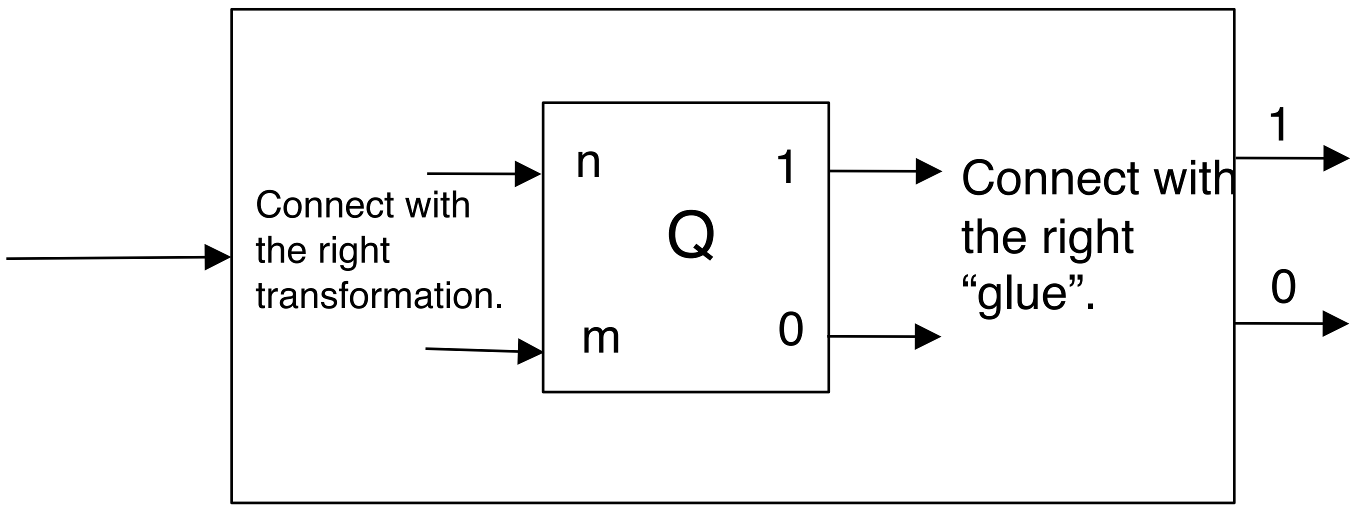


Since D can't really exist, H can't either.

A Template for Uncomputability Proofs

- The preceding diagrammatic reduction can be modified for a variety of similar proofs.

U: (Known to be uncomputable)



Q's computability is in question.

The transformation might not be obvious!

The Blank-Tape Halting Problem (BT)

- What about the ostensibly-easier case of computing:

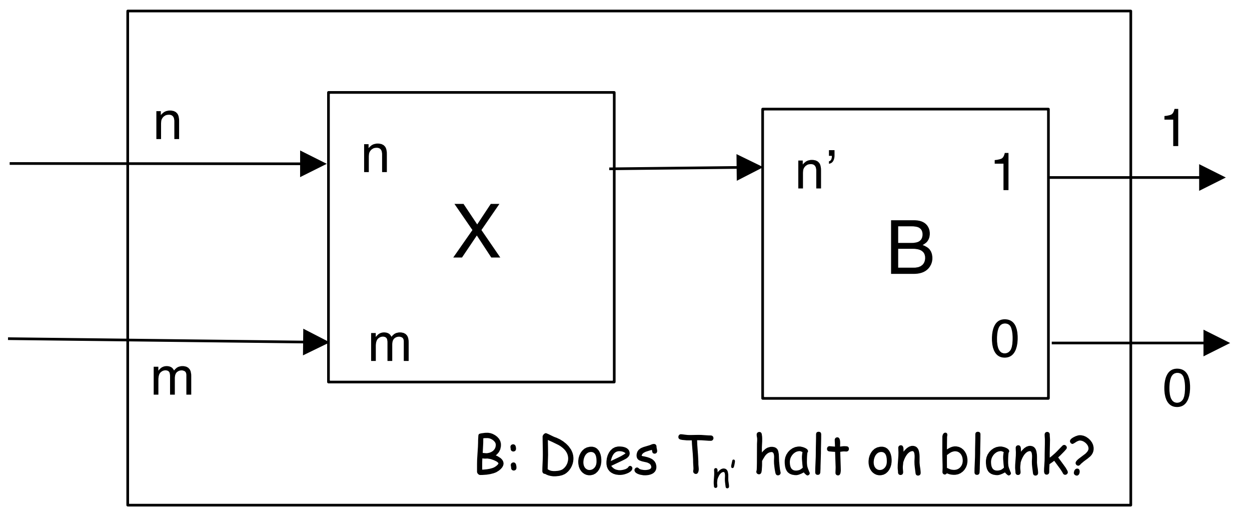
$$B(i) = \begin{cases} 1 \text{ if } T_i \text{ halts on a completely blank tape} \\ 0 \text{ otherwise} \end{cases}$$

Remember that the argument is an encoding of an arbitrary Turing machine. That encoding can be treated as data, to create an encoding of another machine, if desired.

HP reduces to BT

- How to implement H (known uncomputable) given B:

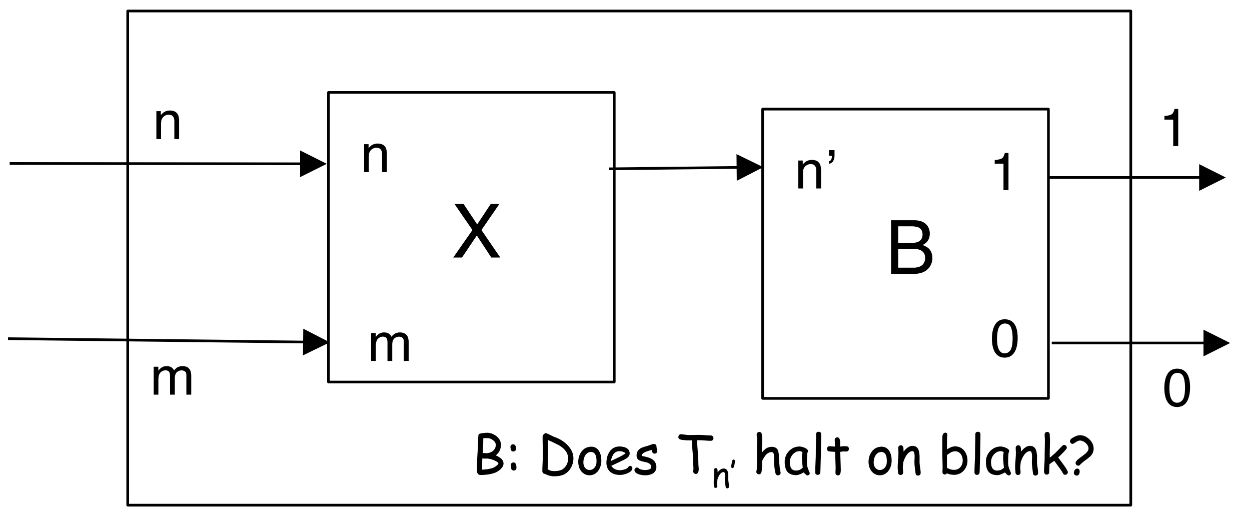
H: Does T_n halt on x_m?



What is X?

What is X?

- X: with input n and m, constructs a new machine n′: n′ writes x_m on the tape, then behaves like T_n .
- Thus T_n halt on x_m iff $T_{n'}$ halts on a blank tape.



X adds to the tuples of T_n a prolog of tuples that writes x_m.

Try this

- What about the still-easier case of computing:

$$S(i) = \begin{cases} 1 \text{ if } T_i \text{ halts on } \textbf{some} \text{ tape} \\ 0 \text{ otherwise} \end{cases}$$

Cautions

- The template might not apply to every uncomputable problem.
- The machines that are constructed by transformations X don't necessarily get **run** in the process.

The arguments are not specific to Turing machines

- The same argument could be made for any universal programming language (Scheme, Java, ISCAL, ...).

Turing Machine vs. Finite-State

- Discuss the best characterization of the **computing power** of a practical computer, such as a laptop, desktop, or pda:
  - Finite-State Machine
  - Turing Machine
  - Something else?

Decidability

- A **decision problem** is one of devising an algorithm that will tell us "yes" or "no" for any given input.

- Thus a decision problem dichotomizes all possible inputs into one of two categories.

Presentation of Languages

- We need a way of presenting languages, i.e. representing them finitely.
- DFAs and regular expressions provide a way, but are not very general.
- Turing machines acting as acceptors allow us to represent a much broader set of languages, the **computable** languages.

Undecidability

- If it is provably-impossible to devise an algorithm for a decision problem, the problem is called undecidable.

- Decision problems can be cast as language-recognition problems, given an appropriate encoding of problem instances: They recognize members of the language for which a "yes" answer is given, and reject members for which a "no" is given.

Functional Properties

- A property of (a machine accepting) a Turing-acceptable language is called "functional" if it only depends on the language itself, and not on the characteristics of the specific machine that accepts the language.

Example of a Non-Functional Property

- Let P(n) be the property T_n halts within n steps when started on a blank tape.
- The language of all n for which P(n) is true is non-functional: Different machines computing the same language may have or not have the property.

Trivial Properties

- A property P of languages is called "trivial" if either:
  - P holds for all languages, or
  - P holds for no language.

Examples of Non-Trivial Properties of languages L

- L is empty
- L is all strings over the alphabet
- L is regular
- L is context-free

   . . .

Rice's Theorem

- The only decidable functional properties are the trivial ones.
- In other words, there is no algorithm that will decide of the properties of languages accepted by Turing machines, other than the two trivial properties.

More Caution

- Consider a language L such as:

$$L = \begin{cases} \text{empty, if Goldbach's conjecture is true,} \\ \text{all strings in } \{0, 1\}^* \text{ otherwise.} \end{cases}$$

- **L is decidable**, because L is one of two decidable sets. We just don't know which set (yet).

- (In fact, L is trivial in the sense of Rice's Theorem.)

Proof of Rice's Theorem

- We'll first do the proof for a special case to make it more concrete, then observe that the method is completely general.
- Suppose we want to show the property of being **regular** is undecidable, i.e. there is no way to test whether an arbitrary Turing machine accepts a regular language.

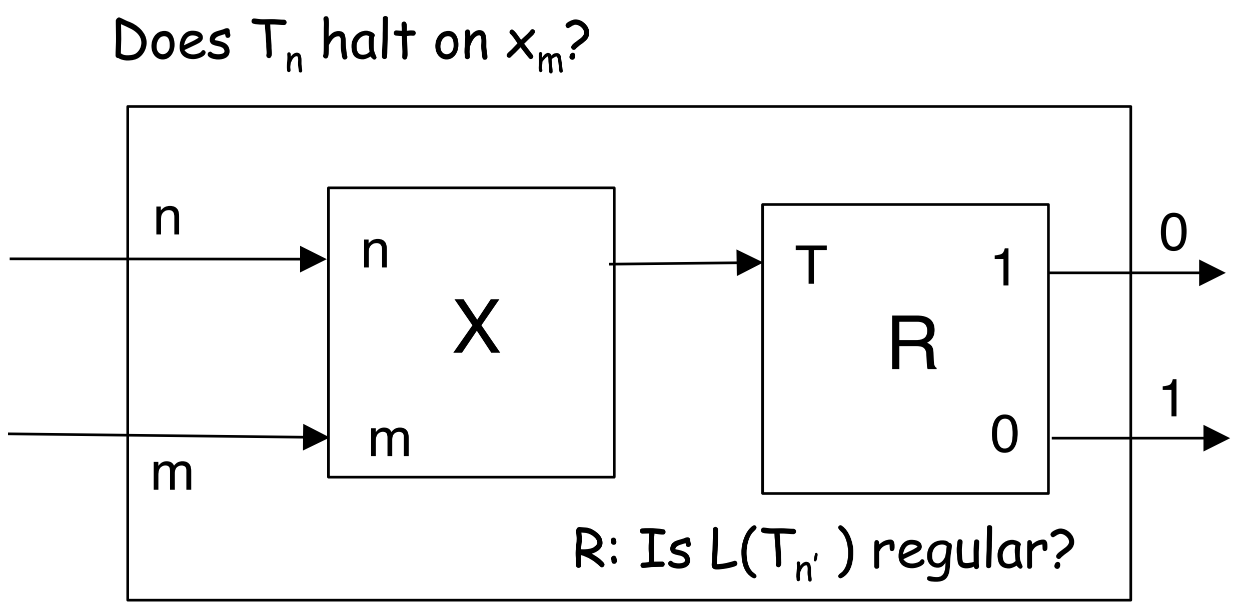
Proof of Rice

- We have to ask this question first:
  - Does the empty language have the property?
    - For the property of being regular, the answer is ____.
    - If so, let S be some Turing-acceptable language that does **not** have the property. For the question of regularity, we could let S be ________.

Rice Proof, continued

- Let Q be a Turing machine accepting S, the non-regular language.
- Now assume that we can determine whether the language of an arbitrary TM is regular. Let R be a machine that decides this.
- We'll bring our template back again, to construct an uncomputable function.

Rice Template



Note label switch!

What is X?

X:

- With input n and m, X constructs a Turing machine T that behaves as follows:
  - T puts its original input aside (on a separate tape track for later recall).
  - T then writes x_m on the main tape track and behaves as T_n on x_m.
    - **If this behavior halts**, then T **next behaves as Q** on the original tape of T that was set aside.
    - **If this behavior does not halt** (which can't be detected), T does not halt either.

Picture of T in action

Tape:

original input to T
x_m

2 tracks

Control:

Save input, write x_m on main track	Behave as T_n on x_m	Behave as Q on original input

T

- The **language** of T is exactly one of two things:
  - The **language of Q**, i.e. S, if T_n halts on x_m.
  - The **empty language**, if T_n does not halt on x_m.

T

- The language of T is one of two things:
  - S, i.e. **non-regular**,
    if T_n halts on x_m.
  - Empty, i.e. **regular**,
    if T_n does not halt on x_m.

Rice Conclusion

- Once T is constructed, it is passed to R, the regularity tester.
- If R says that L(T) **is** regular (which is equivalent to L(T) being empty), it is essentially saying that T_n does not halt on x_m.
- If R says that L(T) is **not** regular (which is equivalent to L(T) being S), it is saying that T_n halts on x_m.
- So if R exists, we could use it to construct a halt-checker.
- Thus R **cannot exist**.

The General Case

- We worked with the property of being **regular**. What about other properties?
- For any property, we must first ask whether the **empty language** has the property or not.
- If it does, we proceed exactly as before, with S being a computable language **not** having the property.
- If it does not, we let S be a language **having** the property. In this case, we **don't switch the labels** on the outer box in the template.