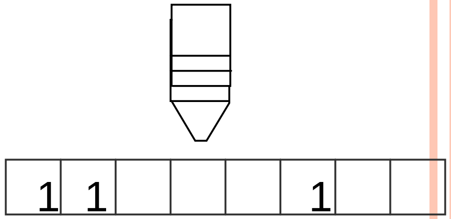
Computability

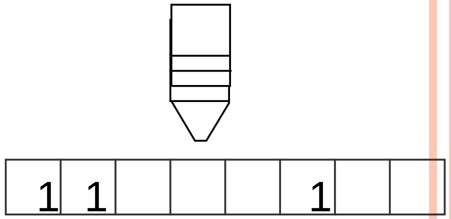
Brief introduction to computable functions, recursive functions, etc.

Jeanine Meyer



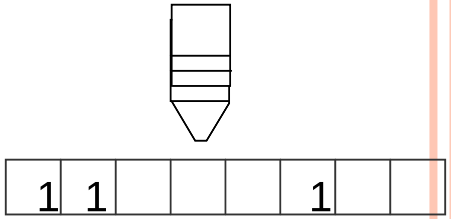
Background

- The study of computability preceded the invention of computers.
  - What does it mean for something (a function, a set of numbers) to be *computed/calculated/evaluated* in a mechanical, procedural way?
  - Related to a problem posed by Hilbert in 1900
- Solved by Alan Turing, and others (Post, Church) in the early 1930s.



QUESTION

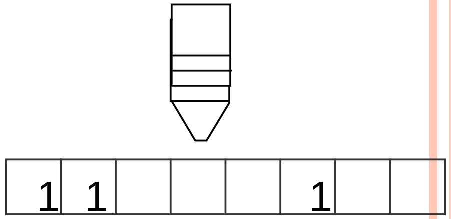
- How old are computers?



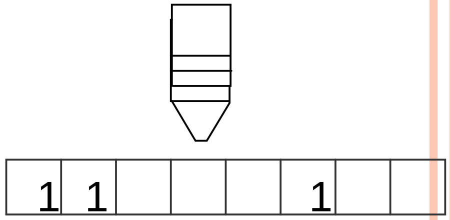
Answer(s)

- Special purpose machines existed
  - Jacquard Loom
  - Enigma (encoded and decoded messages for Germans in WWII)
  - Bombe developed in Bletchley Park by Turing and others helped to find settings for a captured Enigma to decode the codes
- ENIAC developed by the U.S. Army, ready 1946, mainly for ballistics

Turing work



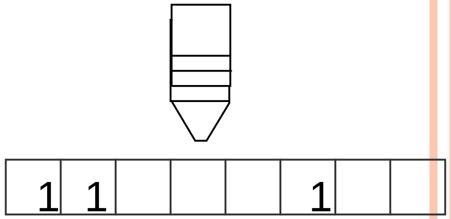
- Defined an abstract machine (is this a contradiction in terms?): came to be called a Turing Machine for computing specific functions…
- ALSO defined set of functions called the recursive functions, made up by a starter set of functions with certain ways of building on functions to get new functions
  - *Note: this definition of a recursive function is more general than "a function that calls itself". Primitive recursion is one of the building methods.*
- Proved any function computed by a Turing Machine was a recursive function AND any recursive function can be computed by a Turing Machine.
  - Also proved these two definitions equivalent to one made by Post
  - Many other functions proved equivalent



TURING MACHINE

Imagine

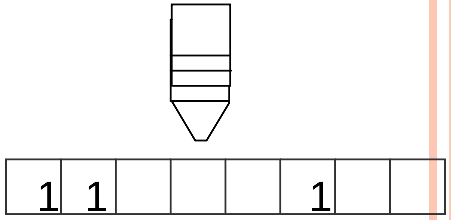
- An [infinite] tape with cells or slots. Each cell can hold a symbol (from a finite alphabet) or a blank
- A specific cell is under the machine head where it can be read
- The machine is in one of a finite number of states
- For each state reading a given symbol, there is a fixed set of actions:
  - Optionally, erase and write a new symbol +
  - Optionally, change to another state +
  - Move left, right, or stay fixed (halting)



TURING MACHINE FUNCTION

- A given Turing machine *computes* a function F(input) ⇨output by
  - representing the input on the tape and placing the head to the left of the input
  - Starting the TM in a special state called the Start state
  - If the function halts, then the content on the tape represents the output using an agreed upon representation.

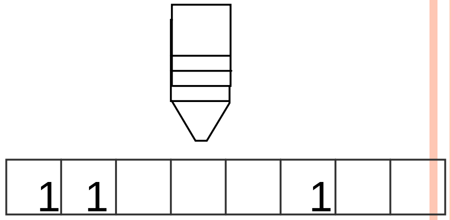
- Alternative: a TM *recognizes a set of numbers or a set of strings of symbols (aka a language)* if the TM halts in a state designated as an accepting state on each number/string in the set AND halts in a state designated as a rejecting state on numbers/strings not in the set.



TURING MACHINE

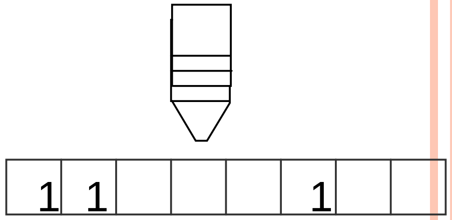
A Turing machine is [specified by]

- a finite number of states
- For each S and each symbol B in the (finite) alphabet, there is a specification of
  - New state
  - New symbol
  - Direction



EXAMPLE: ADDING 1

- Let the alphabet be B (for blank) and 1
- Encode positive integers (0, 1, 2,…) by N+1 1s for input (so zero can be represented)
- Encode (interpret) output by counting up all the 1s on the tape (alternative: require one more 1.)
- States are Start state S, W, F for final
- Start head at start of input

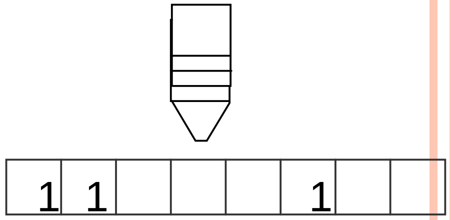


EXAMPLE, ADDING 1

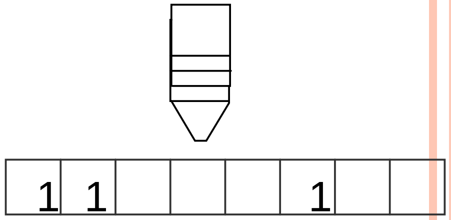
Three states: S, W, F

- State S: sees a 1, moves to the right and changes to State W. sees a blank, does nothing
- State W: sees a 1, moves to the right. Sees a blank, changes to State F.
- State F: does nothing

Example: adding 1, cont.

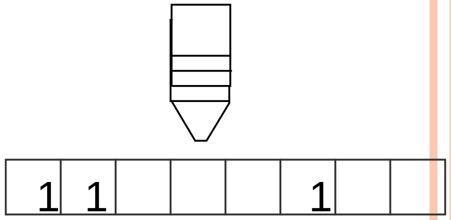


- Start state S: reading a 1, changes to state W and moves to the right.
- W state: reading a 1, moves to the right.
- W state: reading a blank, changes to final state F.
  - [Alternative encoding (1 more 1 for any number): reading a blank, write a 1 and then change to final state F



EXAMPLE:
ADDING TWO INTEGERS

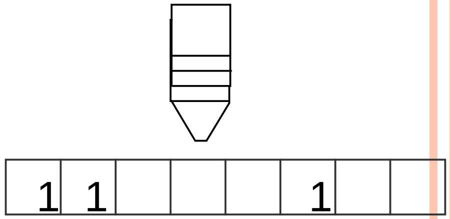
- Encode the two integers N and M by placing N+1 1s followed by 1 blank followed by M+1 1s
- Make output be (N+M)+1 1s contiguously on the tape
  - Alternative: use the encoding of exactly N+M 1s anywhere on the tape.



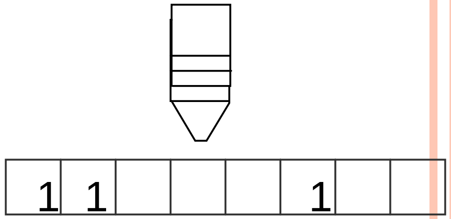
CLASS WORK!

- Strategy:
  - Move until TM finds the blank in the middle and write a 1 where the blank was
  - Keep moving until you find the next blank (after the second set of 1s) and erase the last 2 1s
    - For alternative output encoding: erase the last 3 1s
  - Double check for N and/or M being zero!
- Work in groups and make this precise (list the states and the actions)

Language Recognition TM



- Suppose an alphabet with two letters {a,b}
  - Actually, more formally {a, b, blank}
- A string of letters from the alphabet is called a word: a, ab, aab, abab, etc.
- A set of words is called a language.
- A TM recognizes a language if it stops in a state designated as accepting (may be more than one accepting state), etc.

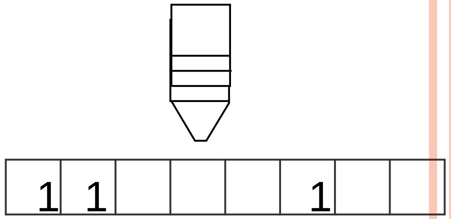


EXAMPLES

Construct a TM that accepts the language

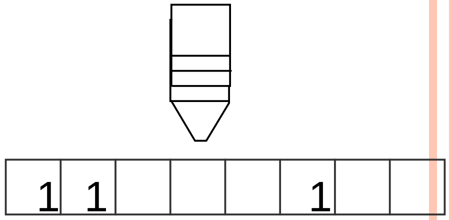
- {a, aa, aaa, …}
- {ab, aab, aaab, aaaab, ….}
- {a, aa, aaa, … } UNION {b, bb, bbb, …}

- Make up a language (that is, a set of patterns}



PALINDROMES

- What would be a TM for a palindrome (words from alphabet {a, b}
- Brain storm strategy

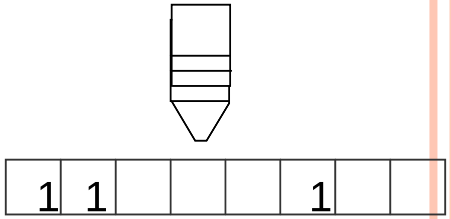


Computable functions

- The functions that can be implemented by a TM are the **computable functions.**

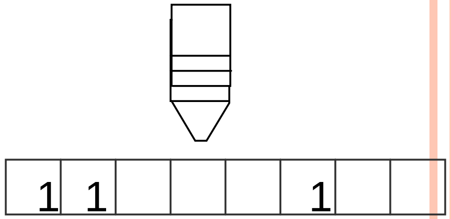
- Reminder: the notion here wasn't an actual machine, but to express a procedure in a effective manner. Think of the TM as being an aid to a human.

Switch gears (so to speak)

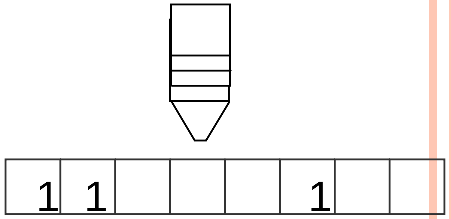


- Define a set of functions (to be called the recursive functions) on integers (finite vectors of integers) in the following way
- Starter (aka initial) functions
  - $C_0(n) \equiv 0$ (constant)
  - $S(n) \equiv n+1$ (successor function)
  - $P^n_i(x_1, x_2, x_3, \ldots x_n) \equiv x_i$ (projections)
    - Special case $Id(x) \equiv x$
- Grow the set of functions using 3 ways to combine functions (aka building methods)

GROW THE SET OF FUNCTIONS

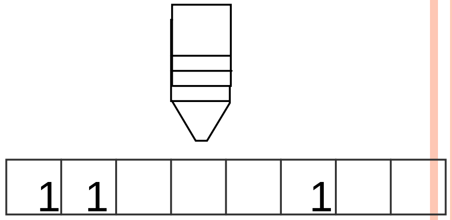


- Composition: if two functions F and G are in the set, so is F(G(input))
- Primitive recursion, if F and G are in the set, so is H where H is defined
  - H(x,0) ≡ F(x)
  - H(x,S(y)) ≡ G(x,H(x,y))
    - Also write this as H(x, y+1) ≡ G(x,H(x,y))
  - NOTE H(x,y) is defined for all x and y



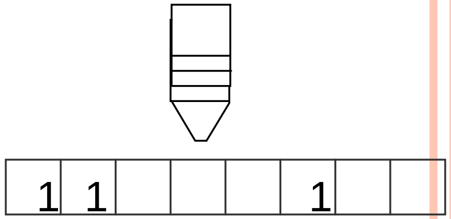
NOTE

- This work assumes certain properties of the integers, arithmetic, etc.



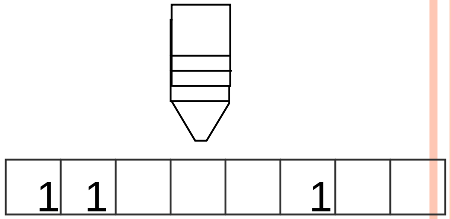
ADDITION

- $A(x,0) \equiv Id(x) \equiv x$
- $A(x, S(y)) \equiv G(x, A(x,y))$ where
  $G(a, b) \equiv S(P_2(a,b)) = S(b) = b + 1$



CONSTANT

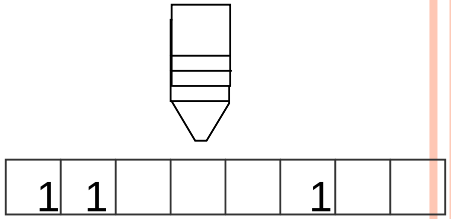
- $C_c(x) \equiv c$ for any constant c is in the set
- $C_c(x) \equiv A(C_0(x),c)$
- Other approaches
  - Since c is a constant, can write out the composition of the successor function starting with 0.



SUBTRACT 1

- Partial function: it is not defined for x = 0
- Since I've used S for successor, use M
- M(S(x)) ≡ x

- Note: can use the next building function (minimization) for an alternate definition



MULTIPLICATION

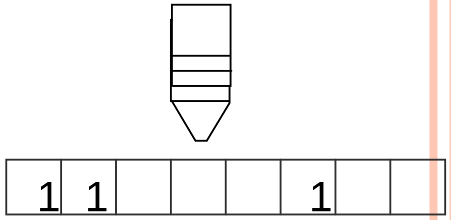
- $H(x,0) \equiv C_0(x) \equiv 0$
- $H(x,S(y)) \equiv G(x, H(x,y))$ where

  $G(a,b) \equiv Id(x) + H(x,y)$

  NOTE: addition has already been established to be in the set

Check

- $H(x,0) \equiv 0 = x * 0$
- $H(x,S(y)) \equiv Id(x)+H(x,y) = x + x*y = x * (S(y))$

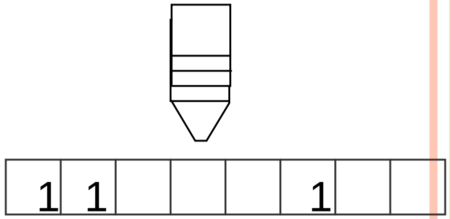


Factorial

- Fac(0) ≡ Fac(1) ≡ 1
- Fac(S(x)) ≡ G(x, Fac(x)) where
  G(a,b) ≡ (a+1) * b so
  Fac(S(x)) = (x+1) * Fac(x)

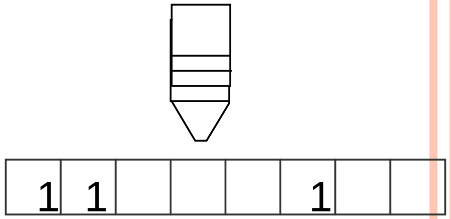
Grow

the Set of Functions, cont.



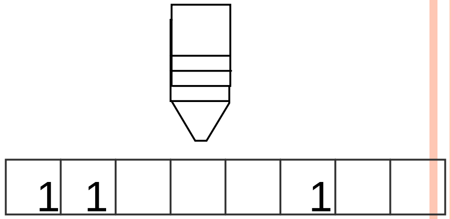
- Minimization (inverse): if F is in the set, then so is G where G is in the set defined as
  If F(x) = y and x is the least integer such that F(x) = y, then
  G(y) ≡ x
- Extend this idea to multiple inputs and outputs
- NOTE: G may not be defined for every y. A function not defined for every input is called a *partial* function

EXERCISE: SUBTRACTION



- Use the minimization/inverse building approach to define
  M(a,b)

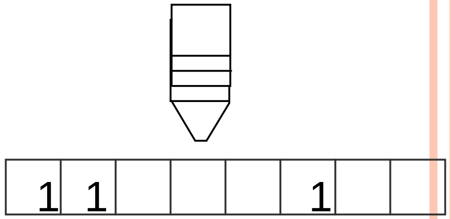
{Computable functions} = {Recursive functions}



- Any function that can be represented by a TM is a recursive function (can be constructed starting with the starter set, using the combining steps)
- Any recursive function can be implemented by a TM

☺☺☺☺ ☺☺☺☺ ☺☺☺☺ ☺☺☺☺ ☺☺☺☺

- Proof makes use of encoding of TM using a technique called Godel numbers.

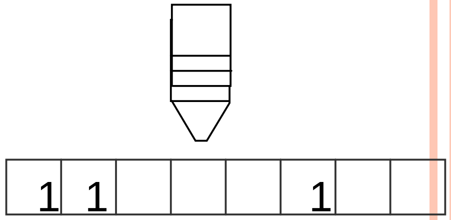


Universal TM

Is a very special Turing Machine (call it U) that takes as input a number representing a TM plus input (input vector of values V) and produces what TM would produce with input V.

- *U is analogous to a general purpose computer with someone standing by supplying more memory as needed.*

- **Halting problem**: Turing proved that there did **not** exist a TM that would accept as input a TM T and input I and answer if the TM would halt on that input.



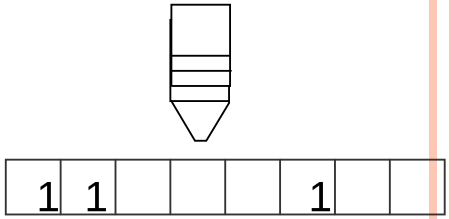
VARIATIONS

in the rules, for example

- Multiple tapes
- Different size alphabets

don't make a difference! What can be done with a TM of that type can be done with a TM of another type.

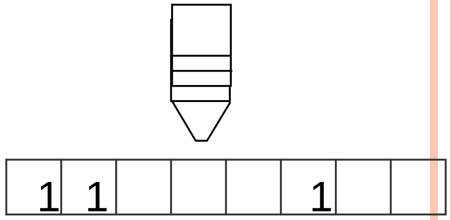
OTHER MACHINES/SYSTEMS



- Finite automata

- Context free grammars

These are not as powerful as TM. That is, there are languages accepted by TM for which there is not CFG and (similarly), there are languages accepted by CFG for which there is no Finite automata



DISCUSSION

- Beautiful piece of mathematics
- Study it!
- Preview: plan to offer course in computability in Fall, 2010