CSE 311 Foundations of Computing I

Lecture 26
Computability: Turing machines, Undecidability of the Halting Problem
Spring 2013

Announcements

- Reading
  - 7th edition: p. 201 and 13.5
  - 6th edition: p. 177 and 12.5
- Topic list and sample final exam problems have been posted
- Final exam, Monday, June 10
  - 2:30-4:20 pm MGH 389.

Last lecture highlights

- Cardinality
- A set S is *countable* iff we can write it as $S=\{s_1, s_2, s_3, \ldots\}$ indexed by $\mathbb{N}$
- Set of rationals is countable
  - "dovetailing"

1/1	1/2	1/3	1/4	1/5	1/6	1/7	1/8	...
2/1	2/2	2/3	2/4	2/5	2/6	2/7	2/8	...
3/1	3/2	3/3	3/4	3/5	3/6	3/7	3/8	...
4/1	4/2	4/3	4/4	4/5	4/6	4/7	4/8	...
5/1	5/2	5/3	5/4	5/5	5/6	5/7	...	
6/1	6/2	6/3	6/4	6/5	6/6	...		
7/1	7/2	7/3	7/4	7/5	....			
...	...	...	...	...				

- Σ^* is countable
  - $\{0,1\}^* = \{\lambda,0,1,00,01,10,11,000,001,010,011,100,101,\ldots\}$
- Set of all (Java) programs is countable

Last lecture highlights

- The set of real numbers is not countable
  - "diagonalization"

		1	2	3	4	5	6	7	8	9	...
r_1 D=	0. 0.	5^{1}	0	0	0	0	0	0	0	...	...
r_2	0.	3	3^{5}	3	3	3	3	3	3	...	...
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...
r_8	0.	6	1	8	0	3	3	9	4^{5}	...	...
...	....	...	....	....	...	...	...	...	...	......	

  - Why doesn't this show that the rationals aren't countable?

Last lecture highlights

- There exist functions that cannot be computed by any program
  - The set of all functions $f : \mathbb{N} \rightarrow \{0,1,\ldots,9\}$ is not countable
  - The set of all (Java/C/C++) programs is countable
  - So there are simply more functions than programs

Do we care?

- Are any of these functions, ones that we would actually want to compute?
  - The argument does not even give any example of something that can't be done, it just says that such an example exists
- We haven't used much of anything about what computers (programs or people) can do
  - Once we figure out, we'll be able to show that some of these functions are really important

Before Java…more from our Brief History of Reasoning

- 1930's
  - How can we formalize what algorithms are possible?
    - Turing machines (Turing, Post)
      - basis of modern computers
    - Lambda Calculus (Church)
      - basis for functional programming
    - μ-recursive functions (Kleene)
      - alternative functional programming basis

All are equivalent!

Turing Machines

Church-Turing Thesis

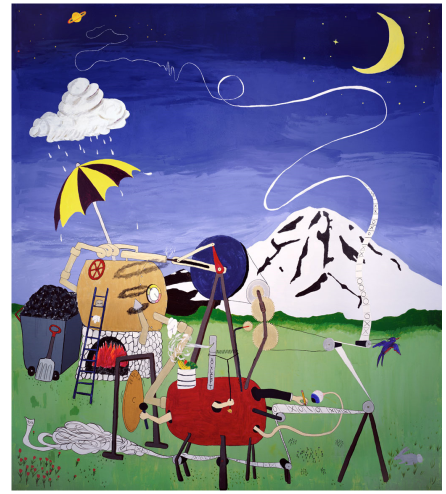
Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

- Evidence
  - Intuitive justification
  - Huge numbers of equivalent models to TM's based on radically different ideas

Components of Turing's Intuitive Model of Computers

- Finite Control
  - Brain/CPU that has only a finite # of possible "states of mind"
- Recording medium
  - An unlimited supply of blank "scratch paper" on which to write & read symbols, each chosen from a finite set of possibilities
  - Input also supplied on the scratch paper
- Focus of attention
  - Finite control can only focus on a small portion of the recording medium at once
  - Focus of attention can only shift a small amount at a time

What is a Turing Machine?



What is a Turing Machine?

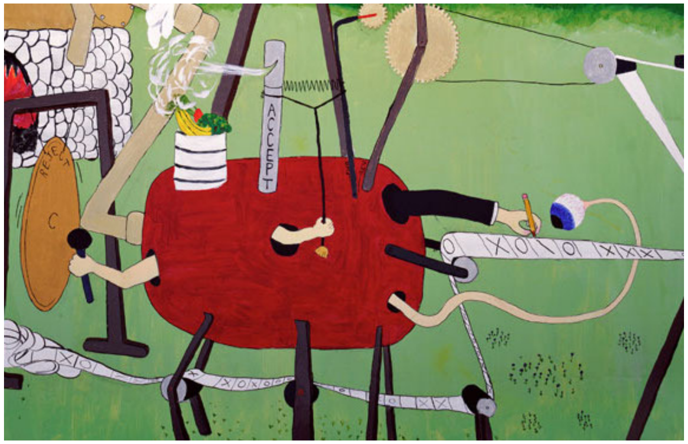
- Recording Medium
  - An infinite read/write "tape" marked off into cells
  - Each cell can store one symbol or be "blank"
  - Tape is initially all blank except a few cells of the tape containing the input string
  - Read/write head can scan one cell of the tape - starts on input
- In each step, a Turing Machine
  - Reads the currently scanned symbol
  - Based on state of mind and scanned symbol
    - Overwrites symbol in scanned cell
    - Moves read/write head left or right one cell
    - Changes to a new state
- Each Turing Machine is specified by its finite set of rules

Sample Turing Machine

	_	0	1
s_1	$(1,s_3)$	$(1,s_2)$	$(0,s_2)$
s_2	(H,s_3)	(R,s_1)	(R,s_1)
s_3	(H,s_3)	(R,s_3)	(R,s_3)

_	_	1	1	0	1	1	_	_

What is a Turing Machine?



Turing Machine ≡ Ideal Java/C Program

- Ideal C/C++/Java programs
  - Just like the C/C++/Java you're used to programming with, except you never run out of memory
    - constructor methods always succeed
    - **malloc** never fails
- Equivalent to Turing machines except a lot easier to program !
  - Turing machine definition is useful for breaking computation down into simplest steps
  - We only care about high level so we use programs

Turing's idea: Machines as data

- Original Turing machine definition
  - A different "machine" **M** for each task
  - Each machine **M** is defined by a finite set of possible operations on finite set of symbols
    - **M** has a finite description as a sequence of symbols, its "code"
- You already are used to this idea:
  - We'll write **<P>** for the code of program **P**
  - i.e. **<P>** is the program text as a sequence of ASCII symbols and **P** is what actually executes

Turing's Idea: A Universal Turing Machine

- A Turing machine interpreter **U**
  - On input **<P>** and its input **x**, **U** outputs the same thing as **P** does on input **x**
  - At each step it decodes which operation **P** would have performed and simulates it.
- One Turing machine is enough
  - Basis for modern stored-program computer
    - Von Neumann studied Turing's UTM design

Halting Problem

- **Given:** the code of a program **P** and an input **x** for **P**, i.e. given (<**P**>,**x**)
- **Output:** **1** if **P** halts on input **x**
  0 if **P** does not halt on input **x**

Theorem (Turing): There is no program that solves the halting problem
"The halting problem is undecidable"

Proof by contradiction

- Suppose that **H** is a Turing machine that solves the Halting problem

Function **D**(**x**):
- if **H**(**x**,**x**)=**1** then
  - **while** (true); /* loop forever */
- else
  - **no-op**; /* do nothing and halt */
- endif

- What does **D** do on input <**D**>?
  - Does it halt?

Does **D** halt on input <**D**>?

Function **D**(**x**):
- if **H**(**x**,**x**)=**1** then
  - **while** (true); /* loop forever */
- else
  - **no-op**; /* do nothing and halt */
- endif

D halts on input <**D**>

$\Leftrightarrow$ **H** outputs **1** on input (<**D**>,<**D**>)

[since **H** solves the halting problem and so **H**(<**D**>,**x**) outputs **1** iff **D** halts on input **x**]

$\Leftrightarrow$ **D** runs forever on input <**D**>

[since **D** goes into an infinite loop on **x** iff **H**(**x**,**x**)=**1**]

That's it!

- We proved that there is no computer program that can solve the Halting Problem.
- This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have