ACMS Seminar, Fridays 3:30-4:30 Smith 105

- Algorithms Theme
- Today: 3:30-4:30 Smith 105
  - **Primes is in P!**
  - Hot-off-the-newswire talk by Neal Koblitz
    - new (this August) algorithm by Agrawal, Kayal, and Saxena
    - First deterministic polynomial-time algorithm for testing whether a number is prime!

1

Course Staff

- **Instructor:** Paul Beame beame@cs
  - **Office:** Sieg 416 **Hours:** MW 1:20-2:00
  - **Phone:** 543-5114 F 2:00-3:00
- **TAs:**
- Deepak Verma deepak@cs
  - **Office:** Sieg 226A **Hours:** Tu 12:00-1:00
- Michael Nelson nelsonmj@cs
  - **Office:** Sieg 226A **Hours:** Th 3:30-4:30

2

CSE 417: Algorithms and Computational Complexity

Computability: Turing Machines & The Halting Problem

Autumn 2002
Paul Beame

3

Computing & Mathematics

Computers as we know them grew out of a desire to avoid bugs in mathematical reasoning

4

A Brief History of Reasoning

- Ancient Greece
  - Deductive logic
    - Euclid's Elements
  - Infinite things are a problem
    - Zeno's paradox

5

A Brief History of Reasoning

- 1670's-1800's Calculus & infinite series
  - Suddenly infinite stuff really matters
  - Reasoning about infinite still a problem
    - Tendency for buggy or hazy proofs
- Mid-late 1800's
  - Formal mathematical logic
    - Boole Boolean Algebra
  - Theory of infinite sets
    - Cantor
      "There are more real #'s than rational #'s"

6

A Brief History of Reasoning

- 1900
  - Hilbert's famous speech outlines goal: mechanize all of mathematics
    23 problems
- 1930's
  - Gödel, Turing show that Hilbert's program is impossible.
    Gödel's Incompleteness Theorem
    Undecidability of the Halting Problem

Both use ideas from Cantor's proof about reals & rationals

A 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
  - Huge numbers of equivalent models to TM's based on radically different ideas

What is a Turing Machine?

- Formal definition
  - Turing & Post excerpts give descriptions
    - Turing's justification, based on intuition, that this is really the right notion.
  - Sipser handout gives full details
- Key properties
  - Manipulates finite sequences of symbols
  - Use a finite set of possible instructions
  - Each does only a finite amount of work per step
  - No *a priori* bound on resource usage
    - Can always get more resources if needed

Turing Machine = Ideal 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
    - **malloc** never fails
    - constructor methods always succeed
- Equivalent to Turing machines except a lot easier to program !
  - Exact TM definition doesn't matter to us so we'll just think of these programs as TM's.

Turing machines as data

- Original Turing machine definition
  - A different machine **P** for each task
  - Each machine **P** is defined by a finite set of possible operations on finite set of symbols
    - **P** has a finite description as a sequence of symbols, its code
- Notation:
  - We'll write <**P**> for the code of program **P** and <**P**,**x**> for the pair of the program code and an input **x**
  - i.e. <**P**> is the program text as a sequence of ASCII symbols and **P** is what actually executes

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 Neuman 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"

Undecidability of the Halting Problem

- Suppose that there is a program **H** that computes the answer to the Halting Problem
- We'll build a table with
  - all the possible programs down one side
  - all the possible inputs along the other side
- Then we'll use the supposed program **H** to build a new program that can't possibly be in the table!

input **x**

program code <P>	ε	0	1	00	01	10	11	000	001	010	011	....
ε	0	1	1	0	1	1	1	0	0	0	1	....
0	1	1	0	1	0	1	1	0	1	1	1	....
1	1	0	1	0	0	0	0	0	0	0	1	....
00	0	1	1	0	1	0	1	1	0	1	0	....
01	0	1	1	1	1	1	1	0	0	0	1	....
10	1	1	0	0	0	1	1	0	1	1	1	....
11	1	0	1	1	0	0	0	0	0	0	1	....
000	0	1	1	1	1	0	1	1	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(<P>,x) entry is **1** if program **P** halts on input **x**
and **0** if it runs forever

Diagonal construction

- Consider a row corresponding to some program code **<P>**
  - the infinite sequence of **0**'s and **1**'s in that row of the table is like a **fingerprint** of **P**
- Suppose a program for **H** exists
  - Then it could be used to figure out the value of any entry in the table
  - We'll use it to create a new program **D** that has a **different fingerprint** from every row in the table
  - But that's impossible since there is a row for every program ! Contradiction

input **x**

program code <P>	ε	0	1	00	01	10	11	000	001	010	011	....
ε	0	1	1	0	1	1	1	0	0	0	1	....
0	1	1	0	1	0	1	1	0	1	1	1	....
1	1	0	1	0	0	0	0	0	0	0	1	....
00	0	1	1	0	1	0	1	1	0	1	0	....
01	0	1	1	1	1	1	1	0	0	0	1	....
10	1	1	0	0	0	1	1	0	1	1	1	....
11	1	0	1	1	0	0	0	0	0	0	1	....
000	0	1	1	1	1	0	1	1	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(<P>,x) entry is **1** if program **P** halts on input **x**
and **0** if it runs forever

input x

program code <P>	ε	0	1	00	01	10	11	000	001	010	011	....
ε	**0**	1	1	0	1	1	1	0	0	0	1	....
0	1	**1**	0	1	0	1	1	0	1	1	1	....
1	1	0	**1**	0	0	0	0	0	0	0	1	....
00	0	1	1	**0**	1	0	1	1	0	1	0	....
01	0	1	1	1	**1**	1	1	0	0	0	1	....
10	1	1	0	0	0	**1**	1	0	1	1	1	....
11	1	0	1	1	0	0	**0**	0	0	0	1	....
000	0	1	1	1	1	0	1	**1**	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(<P>,x) entry is 1 if program P halts on input x
and 0 if it runs forever

input **Flipped diagonal**

program code <P>	ε	0	1	00	01	10	11	000	001	010	011	....
ε	**1**	1	1	0	1	1	1	0	0	0	1	....
0	1	**0**	0	1	0	1	1	0	1	1	1	....
1	1	0	**0**	0	0	0	0	0	0	0	1	....
00	0	1	1	**1**	1	0	1	1	0	1	0	....
01	0	1	1	1	**0**	1	1	0	0	0	1	....
10	1	1	0	0	0	**0**	1	0	1	1	1	....
11	1	0	1	1	0	0	**1**	0	0	0	1	....
000	0	1	1	1	1	0	1	**0**	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

Want to create a new program whose halting properties are given by the **flipped** diagonal

Code for D given subroutine for H

- Function **D**(**x**):
  - if **H**(**x**,**x**)=**1** then
    - **while** (true); /* loop forever */
  - else
    - **no-op**; /* do nothing and halt */
  - endif
- **D**'s fingerprint is different from every row of the table

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
  - The full story is even worse

Using the undecidability of the halting problem

- We have one problem that we know is impossible to solve
  - Halting problem
- Showing this took serious effort
- We'd like to use this fact to derive that other problems are impossible to solve
  - don't want to go back to square one to do it

Another undecidable problem

- **The "always halts" problem**
  - **Given:** <**M**>, the code of a program **M**
  - **Output:** **1** if **M** halts on every input
    0 if not.
- Claim: the "always halts" problem is undecidable
- Proof idea:
  - Show we could solve the Halting Problem **if** we had a solution for the "always halts" problem.
  - No program solving for Halting Problem exists
    Þ no program solving the "always halts" problem exists

What we would like

- To solve the Halting Problem need to handle inputs of the form <P,x>
- Our program will create a new program code <M> so that
  - If **P** halts on input **x**
    - then **M** **always** halts
  - If **P** runs forever on input **x**
    - then **M** runs forever on at least one input
- In fact, the <M> we create will act the same on all inputs

Creating <M> from <P,x>

- Given **<P,x>** modify code of **P** to:
  - Replace all input statements of **P** that read input **x**, by assignment statements that 'hard-code' **x** in **P**
- This creates a new program text **<M>**
- It would be easy to write a program **T** that changes **<P,x>** to **<M>**

The transformation

```
int main(){
  …
  scanf("%d",&u);
  …
  scanf("%d",&v);
  …
  }
123 712
```

<P,x>

```
int main(){
  …
  u = 123;
  …
  v = 712;
  …
  }
```

<M>

Program to solve Halting Problem if "always halts" were decidable

- Suppose "always halts" were solvable by program **A**
- On input **<P,x>**
  - execute the program **T** to transform **<P,x>** into **<M>** as on last slide
  - call **A** with **<M>** (the output of **T**) as its input and use **A**'s output as the answer.

Another undecidable problem

- **The "yes" problem**
  - **Given:** <M>, the code of a program **M**
  - **Output:** **1** if **M** outputs **"yes"** on every input
    0 if not.
- Claim: the "yes" problem is undecidable
- Proof idea:
  - Show we could solve the Halting Problem **if** we had a solution for the "yes" problem.
  - No program solving for Halting Problem exists
    Þ no program solving the 'yes" problem exists

What we would like

- To solve the Halting Problem need to be able to handle inputs of the form **<P,x>**
- We'll create a new program code **<M>** so that
  - If **P** halts on input **x**
    - then **M** **always** outputs **"yes"**
  - If **P** runs forever on input **x**
    - then **M** does something else on at least one input.

Creating <M> from <P,x>

- Given **<P,x>** modify code of **P** to:
  - Remove all output statements from **P**
  - Replace all input statements of **P** that read input **x**, by assignment statements that hard-code **x** in **P**
  - Add a new last statement that prints **"yes"**
- This creates a new program text **<M>**
- It would be easy to write a program **T** that changes **<P,x>** to **<M>**

31

Program to solve Halting Problem if the "yes" problem were decidable

- Suppose the "yes" problem were solvable by program **Y**
- On input **<P,x>**
  - execute the code to transform **<P,x>** into **<M>** as on last slide
  - call **Y** with **<M>** (the output of **T**) as its input and use **Y**'s output as the answer.

32

Equivalent program problem

- Given: the codes of two programs, **<P>** and **<Q>**
- Output: **1** if **P** produces the same output as **Q** does on every input

  0 otherwise

Exercise: Show that the equivalent program problem is undecidable.

33

A general phenomenon: Can't tell a book by its cover

- Suppose you have a problem **C** that asks, given program code **<P>**, to determine some property of the input-output behavior of **P**, answering **1** if **P** has the property and **0** if **P** doesn't have the property.
- Rice's Theorem: If **C**'s answer isn't always the same then there is no program deciding **C**

34

Even harder problems

- Recall that with the halting problem, we could always get at least one of the two answers correct
  - if it halted we could always answer 1 (and this would cover precisely all 1's we need to do) but we can't be sure about answering 0
- There are natural problems where you can't even do that!
  - The equivalent program problem is an example of this kind of even harder problem.

35

Quick lessons

- Don't rely on the idea of improved compilers and programming languages to eliminate major programming errors
  - truly safe languages can't possibly do general computation
- Document your code!!!!
  - there is no way you can expect someone else to figure out what your program does with just your code ....since....in general it is provably impossible to do this!

36