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!

CSE 417: Algorithms and Computational Complexity
Computability: Turing Machines & The Halting Problem

A Brief History of Reasoning
Ancient Greece
  - Deductive logic
    - Euclid’s Elements
  - Infinite things are a problem
    - Zeno’s paradox

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

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”
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

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 \(<P,x>\) 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

\[ \text{input: } P \Rightarrow P(x) \quad \text{output: } U \Rightarrow P(x) \]

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!

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

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.

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
    - 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

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

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

\(<P,x>\) \(<M>\)

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 \(\Rightarrow\) 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 \(<M>\)

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.

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.

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 \(C\)

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.

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!