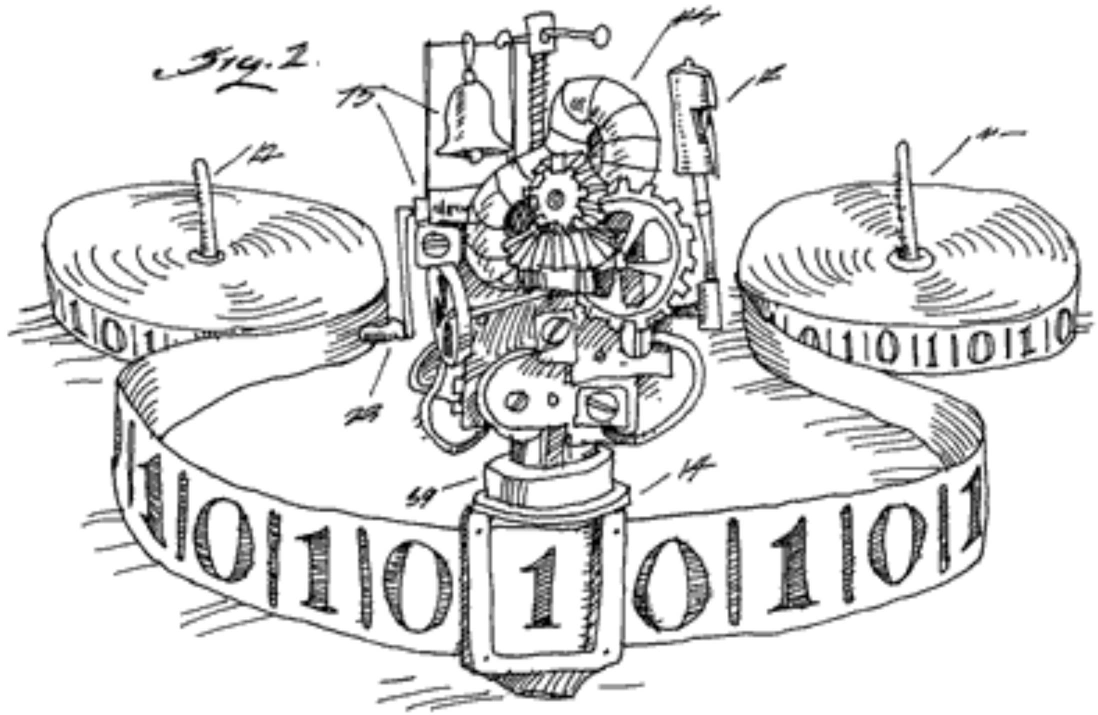
CSE 311: Foundations of Computing

Fall 2014

Lecture 28: Undecidability



Review:

- We have seen that
  - The set of all (Java) programs is countable

Supposed listing of functions: $\mathbb{N}_+ \to \{0,1,\ldots,9\}$

	1	2	3	4	5	6	7	8	9	...
f_1	5	0	0	0	0	0	0	0	...	...
f_2	3	3	3	3	3	3	3	3	...	...
f_3	1	4	2	8	5	7	1	4	...	...
f_4	1	4	1	5	9	2	6	5	...	...
f_5	1	2	1	2	2	1	2	2	...	...
f_6	2	5	0	0	0	0	0	0	...	...
f_7	7	1	8	2	8	1	8	2	...	...
f_8	6	1	8	0	3	3	9	4	...	...
...	...	....	....	...	...	...	...	...	...	

Flipped Diagonal Function D: $\mathbb{N}_+ \rightarrow \{0,1,\ldots,9\}$

Flipping Rule:

If $f_n(n)$ is 5, make $D(n)=1$

If $f_n(n)$ is not 5, make $D(n)=5$

	1	2	3	4						
f_1	5 1	0	0	0						
f_2	3	3 5	3	3					...	...
f_3	1	4	2 5	8	5	7	1	4	...	...
f_4	1	4	1	5 1	9	2	6	5	...	...
f_5	1	2	1	2	2 5	1	2	2	...	...
f_6	2	5	0	0	0	0 5	0	0	...	...
f_7	7	1	8	2	8	1	8 5	2	...	...
f_8	6	1	8	0	3	3	9	4 5	...	...
...	...	....	....	...	...	...	...	...	... ...	

Flipped Diagonal Function D: $\mathbb{N}_+ \to \{0,1,\ldots,9\}$

	1	2	3	4	5	6	7	8	9	...
D =	1	5	5	1	5	5	5	5	...	

For all n, we have
$D \neq f_n$ since $D(n) \neq f_n(n)$

$\Rightarrow$ list was incomplete

$\Rightarrow \{f \mid f : \mathbb{N}_+ \to \{0,1,\ldots,9\}\}$ is not countable

Non-computable functions

- We have seen that
  - The set of all (Java) programs is countable
  - The set of all functions f : $\mathbb{N} \rightarrow \{0,1,\ldots,9\}$ is not countable

- So…
  - There must be some function f : $\mathbb{N} \rightarrow \{0,1,\ldots,9\}$ that is not computable by any program!

Back to the Halting Problem

- Suppose that there is a program H that computes the answer to the Halting Problem
- We will build a table with a row for each program (just like we did for uncountability of reals)
- If the supposed program H exists then the D program we constructed as before will exist and so have a row in the table
- We will see that D must have entries like the "flipped diagonal"
  - D can't possibly be in the table.
  - Only assumption was that H exists. That must be false.

Some possible inputs x

C(P) is shorthand for CODE(P)

programs P

	$C(P_1)$	$C(P_2)$	$C(P_3)$	$C(P_4)$	....							
P_1	0	1	1	0	1	1	1	0	0	0	1	...
P_2	1	1	0	1	0	1	1	0	1	1	1	...
P_3	1	0	1	0	0	0	0	0	0	0	1	...
P_4	0	1	1	0	1	0	1	1	0	1	0	...
P_5	0	1	1	1	1	1	1	0	0	0	1	...
P_6	1	1	0	0	0	1	1	0	1	1	1	...
P_7	1	0	1	1	0	0	0	0	0	0	1	...
P_8	0	1	1	1	1	0	1	1	0	1	0	...
P_9	. . . .											
.	. . . .											
.												

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

Some possible inputs **x**

programs P

	C(P_1)	C(P_2)	C(P_3)	C(P_4)	....							
P_1	**0**	1	1	0	1	1	1	0	0	0	1	...
P_2	1	**1**	0	1	0	1	1	0	1	1	1	...
P_3	1	0	**1**	0	0	0	0	0	0	0	1	...
P_4	0	1	1	**0**	1	0	1	1	0	1	0	...
P_5	0	1	1	1	**1**	1	1	0	0	0	1	...
P_6	1	1	0	0	0	**1**	1	0	1	1	1	...
P_7	1	0	1	1	0	0	**0**	0	0	0	1	...
P_8	0	1	1	1	1	0	1	**1**	0	1	0	...
P_9	.	.	. .	.	.	.	.	.	.	.	.	.
.	.	.	. .	.	.	.	.	.	.	.	.	.
.												

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

recall: code for D assuming subroutine H that solves the halting problem

- Function **D(x)**:

```
if (H(x,x)==true) {
    while (true); /* loop forever */
}
else {
    return; /* do nothing and halt */
}
```

Some possible inputs **x**

D behaves like **flipped diagonal**

programs **P**

	$\langle P_1 \rangle$	$\langle P_2 \rangle$	$\langle P_3 \rangle$	$\langle P_4 \rangle$	$\langle P_5 \rangle$	$\langle P_6 \rangle$	….					
P_1	0^{1}	1	1	0	1	1	1	0	0	0	1	…
P_2	1	1^{0}	0	1	0	1	1	0	1	1	1	…
P_3	1	0	1^{0}	0	0	0	0	0	0	0	1	…
P_4	0	1	1	0^{1}	1	0	1	1	0	1	0	…
P_5	0	1	1	1	1^{0}	1	1	0	0	0	1	…
P_6	1	1	0	0	0	1^{0}	1	0	1	1	1	…
P_7	1	0	1	1	0	0	0^{1}	0	0	0	1	…
P_8	0	1	1	1	1	0	1	1^{0}	0	1	0	…
P_9	.	.	. .	.	.	.	.	.	.	.	.	.
.	.	.	. .	.	.	.	.	.	.	.	.	.
.												

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

recall: code for D assuming subroutine H that solves the halting problem

- Function **D(x)**:

```
if (H(x,x)==true) {
      while (true); /* loop forever */
}
else {
      return; /* do nothing and halt */
}
```

- If **D** existed it would have a row different from every row of the table
  - **D** can’t be a program so **H** cannot exist!

Diagram: Using Hypothetical Program H to build D

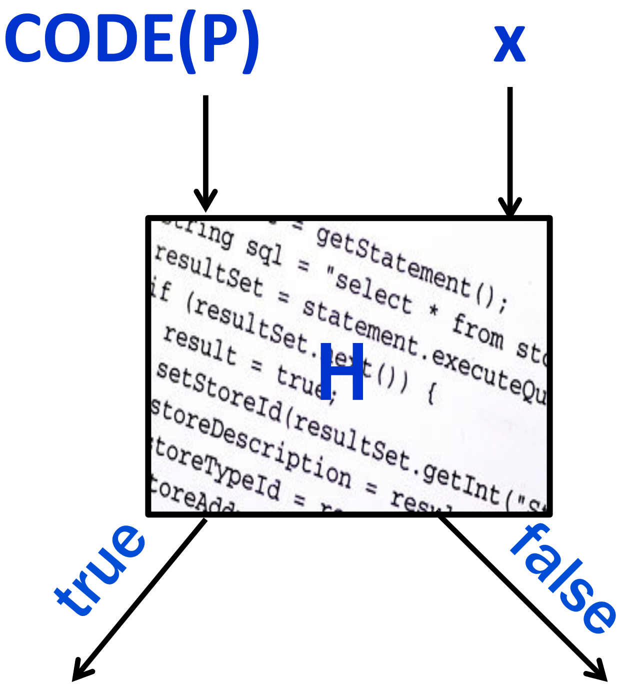
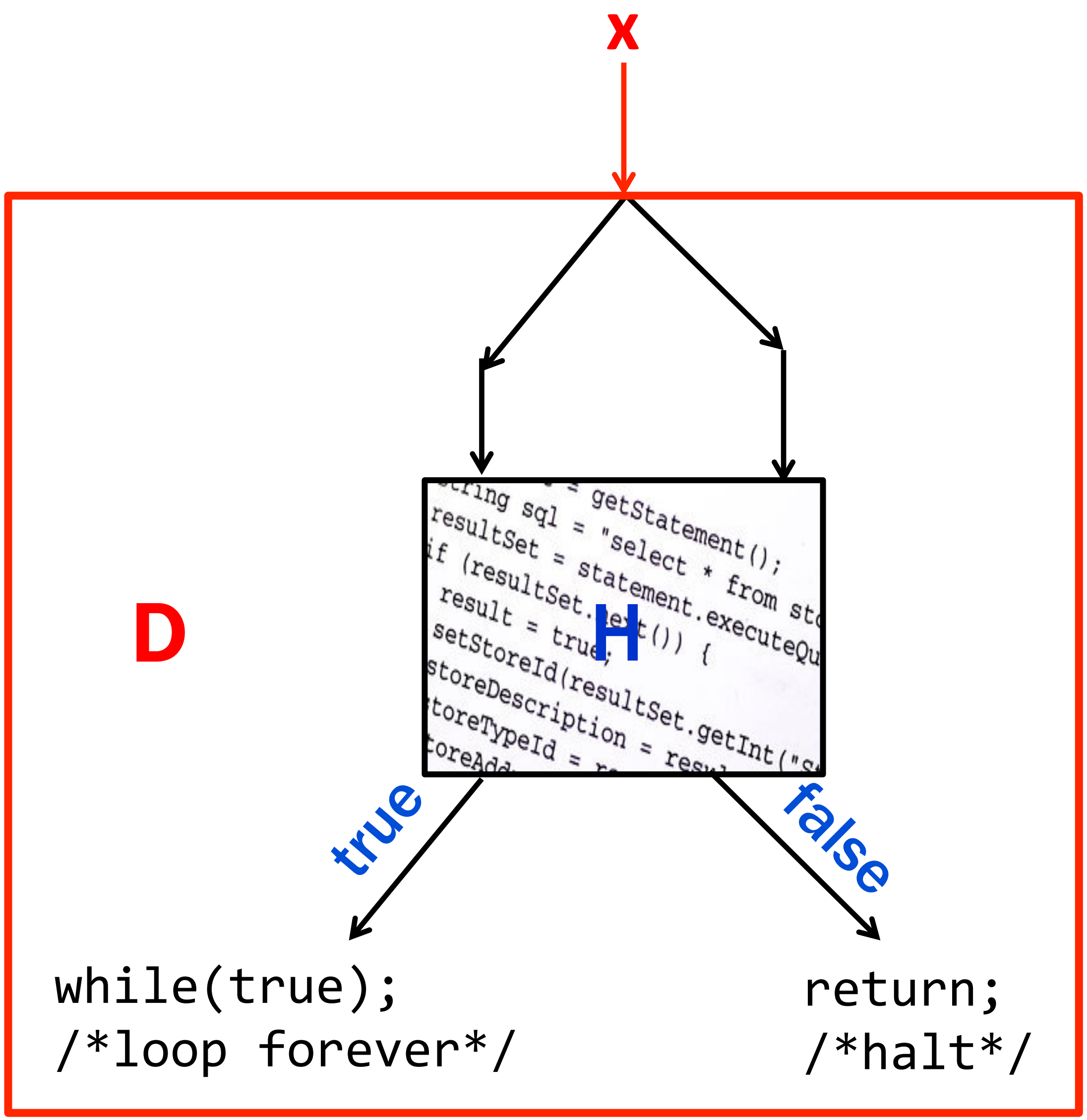


Diagram: Using Hypothetical Program H to build D



More than just halting is hard

- We showed
  - if the hypothetical program **H** deciding the Halting Problem existed, then we could use it to build a program **D** that cannot possibly exist
  - Since **D** doesn't exist, program **H** cannot exist
- We will use similar approach to show that other important problems are hard
  - if there is a hypothetical program **A** solving one of these problems, then we could use it to build a program **H** solving the Halting Problem
  - Since **H** doesn't exist, **A** cannot exist

But first another hard halting-related problem

Halting Problem:

Given: **-** CODE(**P**) for any program **P**

- input **x**

Output: **true** if **P** halts on input **x**

false if **P** does not halt on input **x**

HaltsNoInput Problem:

Given: **-** CODE(**Q**) for any program **Q**

Output: **true** if **Q** halts without reading any input

false if **Q** reads input or runs forever.

Key idea: Hardcoding an Input

```
INPUT is “potato”

public String P(String y) {
     return new String(
          Arrays.sort(y.toCharArray()
     );
}
```

Q: Version of P with “hardcoded” input:

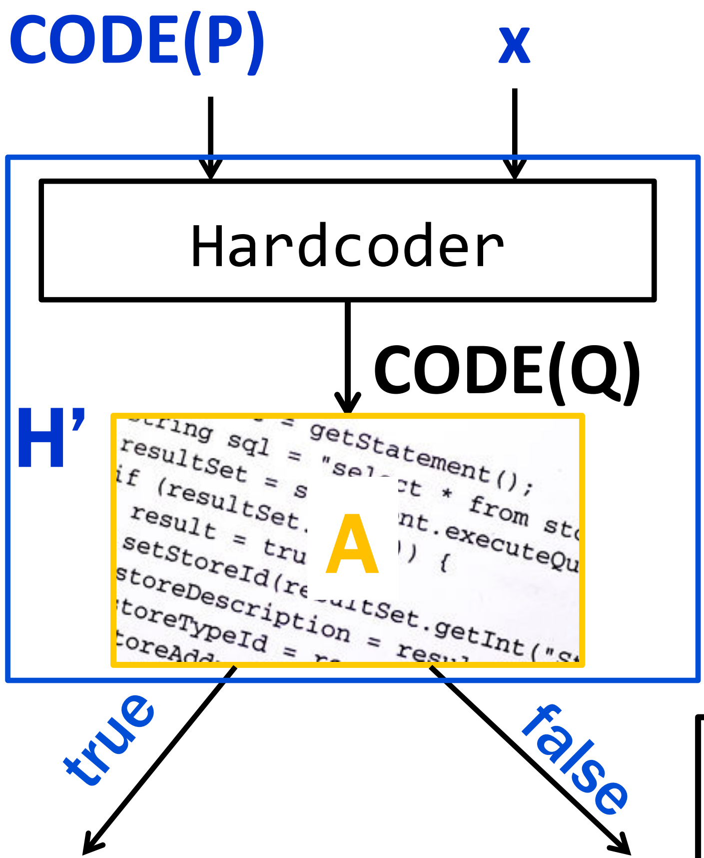
```
public String Q() {
     return new String(
          Arrays.sort(“potato”.toCharArray()
     );
}
```

Q() behaves the same as P(“potato”), except that it doesn’t read any input.

Can write a program `Hardcoder` that, given CODE(P) and an input string x, produces CODE(Q)

Showing there is no program solving **HaltsNoInput**

Suppose that hypothetical program A solves HaltsNoInput problem. ***Combine with*** `Hardcoder`:



H' outputs true on inputs

CODE(P) ***and*** **x**

iff **A outputs true**

iff Q() reads no input

and (always) halts

iff **P(x)** ***halts***

If A existed then H' would solve the Halting Problem: Impossible

Some notation: Decision problems as sets

Every decision problem can be written as asking about membership in a set.

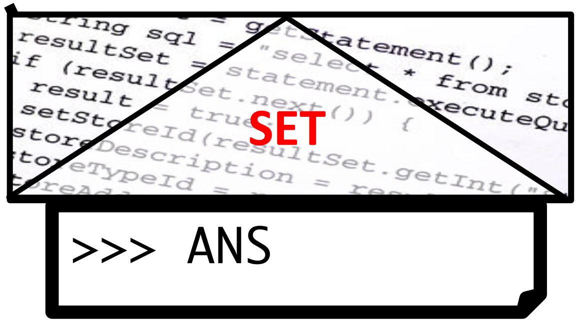
If a program "decides" a set, then it must output **true** on all inputs in the set and **false** on all inputs not in the set.

Halt ={(CODE(P),x) : **P** is a program that halts on input **x**}

HaltsNoInput={CODE(Q): **Q** is a program that halts without reading any input}

Convenient pictures

- Rather than continue to come up with more names like H and A for our hypothetical programs…
- Given a decision problem SET we use the following picture to denote any hypothetical program that solves decision problem SET with ANS denoting its output.



Showing HELLO is Undecidable

Consider the set:

HELLO = {CODE(R) : R is a program that reads no input, prints "Hello", and always halts}

CODE(Q)

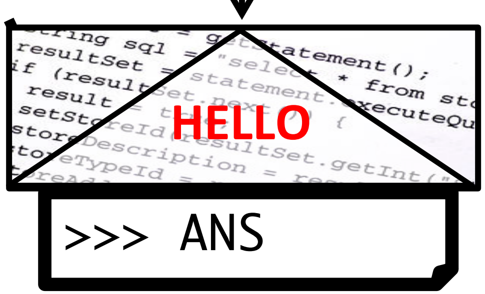
Question: Does Q() halt?

Step 1: Remove all `System.out.print/println` **statements from CODE(Q).**

Step 2: Append `System.out.println("Hello")` **at the end of the program code.**

Call the new program R

CODE(R)



Question:

Does **R**() print "**Hello**" and halt?

Answering with ANS **would solve HaltsNoInput!**

Showing HELLO is Undecidable (Full Proof)

Suppose for contradiction that HELLO is decidable. Then, there is a program HLO(CODE(R)) that returns true when CODE(R) $\in$ HELLO and false otherwise.

Consider an arbitrary program Q. We will now construct a program that decides if Q $\in$ HaltsNoInput. Define a new program R by applying the following transformations to Q's code:

Step 1: Remove all System.out.print/println statements from CODE(Q).

Step 2: Append System.out.println("Hello") at the end of the code (and before all return statements).

Then, we can create the following program:

HNI(Q) { return HLO(R);}

We claim HNI solves HaltsNoInput. Suppose HNI(Q) = true. Then, it should be the case that Q halts. Note that HLO(R) must be true by definition of HNI. So, R $\in$ HELLO; it follows that R reads no input, prints "Hello", and halts. Note that Q also reads no input. Furthermore, Step 2 ensures that Q prints "Hello", and Step 1 ensures it prints nothing else. It follows that when R $\in$ HELLO, R must halt. But R halts exactly when Q halts, because we didn't change anything that effects whether it halts or not.

We make a very similar argument for when HNI(Q) = false. Then, since we've solved HaltsNoInput, which is undecidable, we've reached a contradiction. So, it follows that HELLO is undecidable as well.

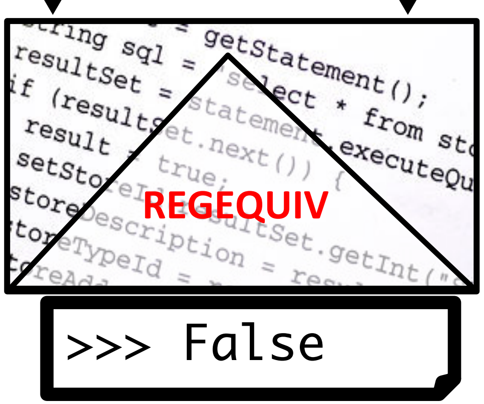
A Decision Problem We Can Solve

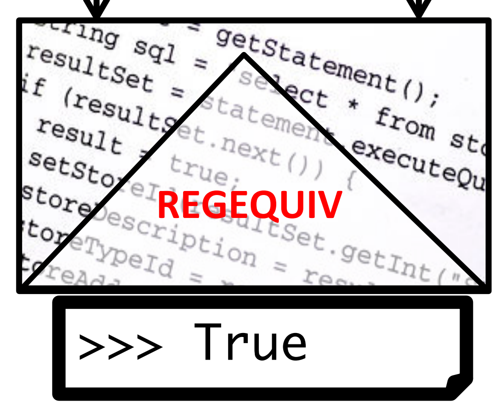
REGEQUIV = {(R_1, R_2) : R_1 and R_2 are equivalent regexps}

In this case the hypothetical program does exist:

Convert both to NFAs then DFAs, minimize and compare

00*(10)*11*   0*(01)*1*   00*(10)*11*   0*(01)*01*1





Showing **EQUIV** is Undecidable

Consider the set:

EQUIV = {(CODE(P), CODE(R)): P, R are programs, P(x) = R(x) for all inputs x}

CODE(Q)

Question: Does Q() halt?

Step 1: Construct P:

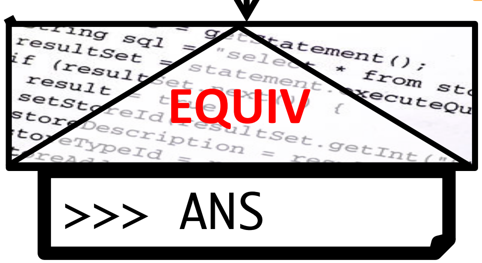
```
public static boolean P() {return true;}
```

Step 2: Construct R:

- **Step a: Replace return type of Q with boolean**
- **Step b: Replace all return values with true**
- **Step c: Add "return true;" to the end of the program**
- **Call this program R**

(CODE(P),CODE(R))

Question: Are P and R Equivalent?



Answering with ANS
would solve HaltsNoInput!

Pitfalls

- Not *every* problem on programs is undecidable! Which of these is decidable?
- Input `CODE(P)` and x
  Output: true if P prints “ERROR” on input x
  after less than 100 steps
  false otherwise

- Input `CODE(P)` and x
  Output: true if P prints “ERROR” on input x
  after more than 100 steps
  false otherwise