15-251

Some ~~AWESOME~~ Mind-blowing

~~Great Theoretical~~ Ideas

~~in Computer Science~~

about ~~Generating Functions~~

~~Probability~~

~~Infinity~~

Computability

With Alan! (not Turing)

What does this do?

```
_(__,___,____){___/__<=1?
_(__,___+1,____):!(___%__)?
_(__,___+1,0):___%__==___/ __&&!____?
(printf("%d\t",___/__),_(__,___+1,0)):___
%__>1&&___%__<___/__?_(__,1+
___,____+!(___/__%(___
%__))):___<__*__?
_(__,___+1,____):0;}main(){_(100,0,0);}
```

Turing's Legacy:
The Limits Of Computation

This lecture will change the way you think about computer programs…

Many questions which appear easy at first glance are impossible to solve in general

The HELLO assignment

Write a Java program to output the words "HELLO WORLD" on the screen and halt.

Space and time are not an issue.

The program is for an ideal computer.

PASS for any working HELLO program, no partial credit.

Grading Script

The grading script G must be able to take any Java program P and grade it.

$$G(P) = \begin{cases} \text{Pass, if P prints only the words "HELLO WORLD" and halts.} \\ \text{Fail, otherwise.} \end{cases}$$

How exactly might such a script work?

What does this do?

```
_(__,___,____){___/__<=1?
_(__,___+1,____):!(___%__)?
_(__,___+1,0):___%__==___/ __&&!____?
(printf("%d\t",___/__),_(__,___+1,0)):___
%__>1&&___%__<___/__?_(__,1+
___,____+!(___/__%(___
%__))):___<__*__?
_(__,___+1,____):0;}main(){_(100,0,0);}
```

Nasty Program

```
n:=0;
while (n is not a counter-example
   to the Riemann Hypothesis) {
   n++;
}
print “Hello World”;
```

The nasty program is a PASS if and only if the Riemann Hypothesis is false.

A TA nightmare: Despite the simplicity of the HELLO assignment, there is no program to correctly grade it!

And we will prove this.

The theory of what can and can't be computed by an ideal computer is called **Computability Theory** or **Recursion Theory**.

From the last lecture:

Are all reals describable? NO
Are all reals computable? NO

We saw that

computable describable

but do we also have

describable computable?

The "grading function" we just described is not computable! (We'll see a proof soon.)

This lecture will hopefully shed light on what is and isn't possible using a program.

But wait! Why are we reasoning about "programs"? Don't we need to use Turing Machines to be mathematically precise?

Not necessarily. Remember the Church-Turing Thesis: any reasonable (and sufficiently powerful) notion of a “program” is equivalent to a Turing Machine. It's okay to just reason about “algorithms”.

What's Allowed in an "Algorithm"?

Anything that we can create
using Turing Machines!

<u>Some examples:</u>

- Arrays, pointers
- Functions
- Integers, strings
- Arithmetic operations
- Conditionals (if)
- Loops (while, for, do)

As long as we use reasonable primitives like these, we are really reasoning about Turing Machines, so our statements have a formal backing.

Extending the Idea of a Program

Program	Turing Machine
Source code	Description of states and transitions
Print statement	Write to a special "output" area of the tape
Return true/false	Accept/Reject

All of the proofs in this lecture will be about programs.
We are still being rigorous because of this equivalence.

Computable Function

Fix a finite set of symbols, Σ

A function f: Σ^* Σ^* is **computable** if there is a program P that when executed on an ideal computer (one with infinite memory), computes f.

That is, for all strings x in Σ^*, $f(x) = P(x)$.

Hence: **countably many** computable functions!

There are only countably many programs.

Hence, there are only countably many computable functions.

Uncountably Many Functions

The functions f: Σ^* $\{0,1\}$ are in 1-1 onto correspondence with the subsets of Σ^* (the powerset of Σ^*).

Subset S of Σ^*	Function f_S
x in S	$f_S(x) = 1$
x not in S	$f_S(x) = 0$

Hence, the set of all f:Σ^* $\{0,1\}$ has the same size as the power set of Σ^*, which is uncountable.

Countably many computable functions.

Uncountably many functions from Σ^* to $\{0,1\}$.

Thus, most functions from Σ^* to $\{0,1\}$ are not computable.

Decidable/Undecidable Sets

A set (more precisely, a language) L Σ^* is said to be **decidable** (or **recursive**) if there exists a program P such that:

P(x) = yes, if x L

P(x) = no, if x L

Notice that this is the Turing Machine equivalent of a regular language.

The theory becomes nicer if we restrict "computation" to the task of deciding membership in a set.

Again, by giving a counting argument, we can say that there must be some undecidable set.

The set of all languages is uncountable, but there can only be countably many decidable languages because there are only countably many programs.

Can we explicitly **describe** an undecidable set?

The Halting Problem

Notation And Conventions

When we write P by itself, we are talking about the text of the source code for P.

P(x) means the output that arises from running program P on input x, assuming that P eventually halts.

P(x) = means P did not halt on x

The meaning of P(P)

It follows from our conventions that P(P) means the output obtained when we run P on the text of its own source code.

The Halting Set K

Definition:

K is the set of all programs P such that P(P) halts.

K = { Program P | P(P) halts }

The Halting Problem

Is the Halting Set K decidable? In other words, is there a program HALT such that:

HALT(P) = yes, if P(P) halts

HALT(P) = no, if P(P) does not halt

THEOREM: There is no program to solve the halting problem (Alan Turing 1937)

Suppose a program HALT existed that solved the halting problem.

HALT(P) = yes, if P(P) halts

HALT(P) = no, if P(P) does not halt

We will call HALT as a subroutine in a new program called CONFUSE.

CONFUSE

```
CONFUSE(P)
{  if (HALT(P))
      then loop forever;       //i.e., we don't halt
   else exit;                  //i.e., we halt
   // text of HALT goes here
}
```

Does CONFUSE(CONFUSE) halt?

CONFUSE

```
CONFUSE(P)
{  if (HALT(P))
      then loop forever;       //i.e., we don't halt
   else exit;                  //i.e., we halt
   // text of HALT goes here  }
```

Suppose CONFUSE(CONFUSE) halts:
then HALT(CONFUSE) = TRUE,
so CONFUSE will loop forever on input CONFUSE

Suppose CONFUSE(CONFUSE) does not halt
then HALT(CONFUSE) = FALSE,
so CONFUSE will halt on input CONFUSE

CONTRADICTION

Alan Turing (1912-1954)

Theorem: [1937]

There is no program to solve the halting problem

Turing’s argument is essentially the reincarnation of Cantor’s **Diagonalization argument** that we saw in the previous lecture.

All Programs (the input)

All Programs

	P_0	P_1	P_2	…	P_j	…
P_0						
P_1						
…						
P_i						
…						

Programs (computable functions) are countable, so we can put them in a (countably long) list

All Programs (the input)

All Programs

	P_0	P_1	P_2	…	P_j	…
P_0						
P_1						
…						
P_i						
…						

YES, if $P_i(P_j)$ halts
No, otherwise

All Programs (the input)

All Programs

	P_0	P_1	P_2	…	P_j	…
P_0	d_0					
P_1		d_1				
…			…			
P_i				d_i		
…					…	

Let d_i = HALT(P_i)

CONFUSE(P_i) halts iff d_i = no

(The CONFUSE function is the negation of the diagonal.)

Hence CONFUSE cannot be on this list.

Is there a real number that can be described, but not computed?

Consider the real number R whose binary expansion has a 1 in the j^{th} position iff the j^{th} program halts on input itself.

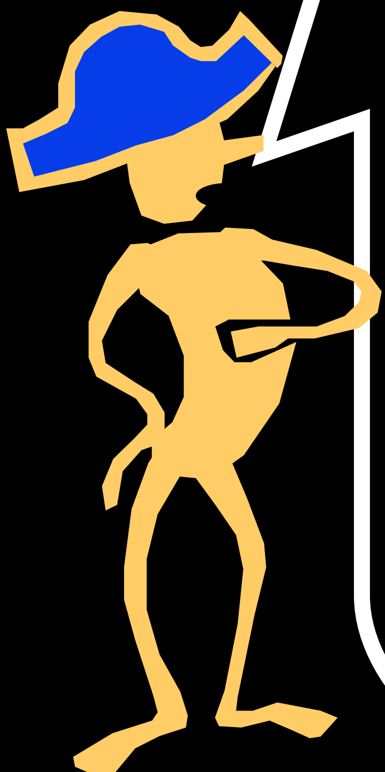
Proof that R cannot be computed

Suppose it is, and program **FRED** computes it.

then consider the following program:

```
MYSTERY(program text P)
  for j = 0 to forever do {
    if (P == P_j)
    then use FRED to compute j^th bit of R
    return YES if (bit == 1), NO if (bit == 0)
  }
```

MYSTERY solves the halting problem!



I'm still not satisfied by the CONFUSE argument that the Halting Problem is undecidable. Isn't there a simpler way to specifically show an undecidable problem?

There actually is a simpler proof, but showing it requires us to extend our understanding of what Turing Machines are capable of.

Reminder: What is a Turing Machine Capable of?

Some examples:

- Arrays, pointers
- Functions
- Integers, strings
- Arithmetic operations
- Conditionals (if)
- Loops (while, for, do)

But what about "obtain a copy of my own source code"? Is this allowed as pseudocode for an "algorithm"?

Have we seen something like this before?

Yes, we have seen this before! The Auto-cannibal Maker assignment demonstrated that given a program (“Eat”), it is possible to construct a program (the “Auto-cannibal”) that behaves like Eat but is aware of its own source code.

The assignment was in C++/Java, but the construction is possible with Turing Machines as well. This result is known as the **recursion theorem**: when writing an algorithm, it is always possible for that algorithm to be aware of its own source code.

Using the recursion theorem, we can come up with a simpler example of an undecidable problem.

A Simpler Variant of the Halting Set

Definition:

K_0 is the set of all programs P that halt when given no input.

K_0 = { Program P | P() halts }

K_0 is much simpler than K!

K_0 is Undecidable

Suppose there was some program HALT2 that decides K_0. Consider the following program:

```
CONTRADICT()
{
   Let s be the source code for CONTRADICT;
   if (HALT2(s))
     then loop forever;
  else exit;
  // text of HALT2 goes here
}
```

CONTRADICT is able to directly turn around and contradict the statement from HALT2, so HALT2 cannot be correct in all cases, so K_0 is undecidable.

Recursively Enumerable Sets

Recursively Enumerable

A set (more precisely, a language) L Σ^* is defined to be **recursively enumerable** (or **semi-decidable**) if there exists a program P such that:

- If x L, then P(x) halts and outputs “yes”.
- If x L, then either P(x) halts and outputs “no”, or P(x) does not halt.

Does decidable imply recursively enumerable? **Yes!**

Does recursively enumerable imply decidable? **No!**

Example: The Halting Set K

Even though K is not decidable, it is easy to show that it is recursively enumerable. Here is a program P that demonstrates that fact:

```
HALF_HALT(P)
{
        run P(P);
        output "yes";
}
```

HALF_HALT will always output “yes” if P(P) halts, and will never output “yes” if P(P) does not halt, which is all that needs to be true.

Why the Name "Enumerable"?

A language L is recursively enumerable if and only if there is some program P that enumerates it. Such a program must output an infinite list of strings, where each string that is output belongs to L and each string in L eventually shows up at least once in the list.

How do we Enumerate the Halting Set?

```
ENUM_HALT(P)
{
    for each natural number i
    {
        For each program P of length    i
        {
            Run P(P) for i steps;
            If P(P) halted in that time, output it
        }
    }
}
```

A Non-Example?

Is there a simple set that is not recursively enumerable?

Yes! The complement of the Halting Set cannot be recursively enumerable:

K' = {Program P | P(P) does not halt}

K' is not Recursively Enumerable

Proof:

Suppose that there was a program HALF_NON_HALT that demonstrated that K' was recursively enumerable.

```
HALT(P)
{
    for each natural number i:
    {
        run HALF_HALT(P) for i steps;
        run HALF_NON_HALT(P) for i steps;
        if either one halts, we know the correct answer, so output it;
    }
}
```

One of the two calls must eventually finish, so HALT decides K, even though K is undecidable! Contradiction!

In general, if a program is recursively enumerable and its complement is recursively enumerable, then that program must be decidable.

Computability Vocabulary Overview

- A set S is decidable if there is a program that returns "yes" on input x whenever x   S and returns "no" on input x whenever x   S.

- A set S is recursively enumerable if there is a program that returns "yes" on input x whenever x   S. If x   S, then the program may either return "no" or it may run forever.

- A function f is computable if there is a program P such that for all strings x, P(x) = f(x).

- A real number x is computable if there is a program P that outputs the digits of x in order, in such a way that every digit is eventually output.

Oracles and Reductions

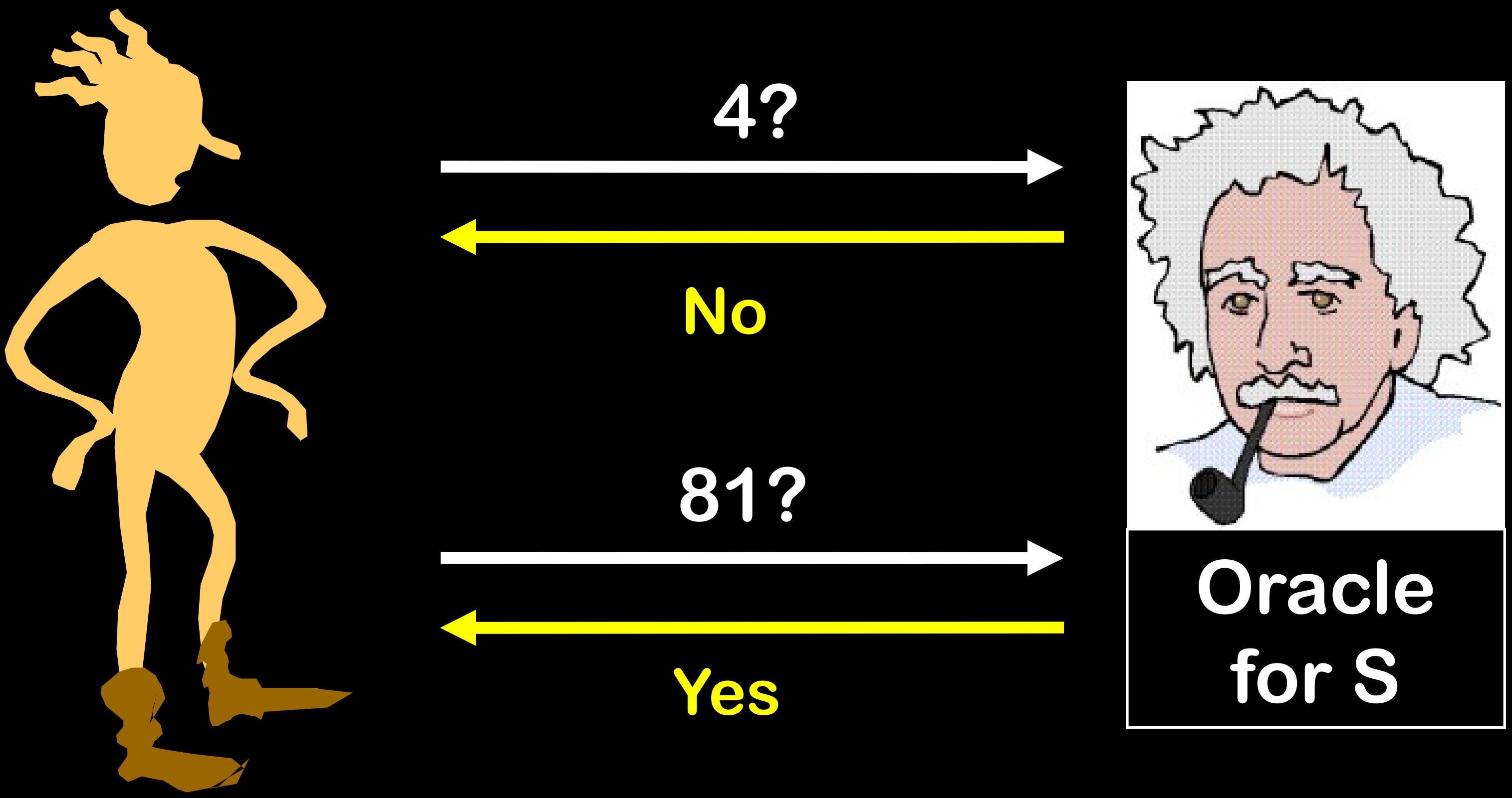
Oracle For Set S

Is x S?

YES/NO

Oracle for S

Example Oracle
S = Odd Naturals



K_0= the set of programs that take no input and halt

Hey, I ordered an oracle for the famous halting set K, but when I opened the package it was an oracle for the different set K_0.

GIVEN: Oracle for K_0

But you can use this oracle for K_0 to build an oracle for K.

K_0= the set of programs that take no input and halt

P = [input I; Q]

Does P(P) halt?

Does [I:=P;Q] halt?

BUILD:
Oracle for K

GIVEN:
Oracle for K_0

We've **reduced** the problem of deciding membership in K to the problem of deciding membership in K_0.

Hence, deciding membership for K_0 must be **at least as hard** as deciding membership for K.

Thus if K_0 were decidable then K would be as well.

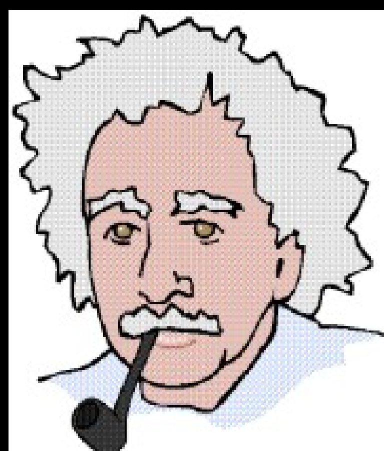
We already know K is not decidable, hence K_0 is not decidable.

HELLO = the set of programs that print hello and halt

Does P halt?

Let P’ be P with all print statements removed.

(assume there are no side effects)



BUILD: Oracle for K_0

Is [P’; print HELLO] a hello program?

GIVEN: HELLO Oracle

Hence, the set HELLO is not decidable.

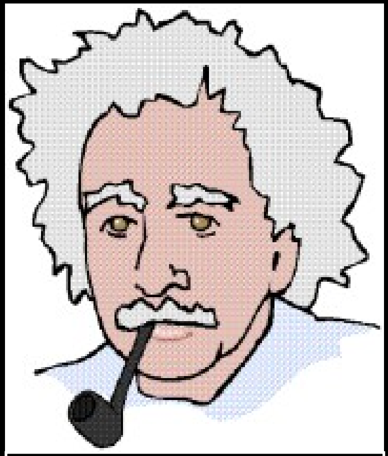
EQUAL = All <P,Q> such that P and Q have identical output behavior on all inputs

Is P in set HELLO?

Let HI = [print HELLO]

BUILD:
HELLO
Oracle

Are P and HI equal?



GIVEN:
EQUAL
Oracle

Halting with input, Halting without input, HELLO, and EQUAL are all undecidable.

Here's What You Need to Know…

Turing Machine/Program equivalence

The Halting Problem: Definition, Proof of Undecidability

The recursion theorem

Recursively enumerable sets

Oracles and Computability reductions