Great Theoretical Ideas in Computer Science

What does this do?

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

Turing's Legacy: The Limits Of Computation
Lecture 21 (November 17, 2007)

Anything I say say is false!

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 word “HELLO” 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.

\[
G(P) = \begin{cases} 
\text{Pass, if P prints only the word “HELLO” and halts.} \\
\text{Fail, otherwise.}
\end{cases}
\]

What exactly might such a script work?

What does this do?

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

\[ n := 0; \]
\[ \text{while (n is not a counter-example to the Riemann Hypothesis) { } } \]
\[ n++; \]
\[ \text{print "Hello";} \]

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 \( \Rightarrow \) describable
but do we also have
describable \( \Rightarrow \) computable?

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

Fix a finite set of symbols, $\Sigma$. Fix a precise programming language, e.g., Java.

A program is any finite string of characters that is syntactically valid.

A function $f : \Sigma^* \to \Sigma^*$ is computable if there is a program $P$ that when executed on an ideal computer, computes $f$. That is, for all strings $x$ in $\Sigma^*$, $f(x) = P(x)$.

Hence: countably many computable functions!

**Uncountably Many Functions**

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

Subset $S$ of $\Sigma^*$ ⇔ Function $f_S$

- $x$ in $S$ ⇔ $f_S(x) = 1$
- $x$ not in $S$ ⇔ $f_S(x) = 0$

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

There are only countably many Java programs.

Hence, there are only countably many computable functions.

Countably many computable functions.

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

Thus, most functions from $\Sigma^*$ to $\{0,1\}$ are not computable.
Can we explicitly describe an uncomputable function?

Can we describe an interesting uncomputable function?

**Notation And Conventions**
Fix a single programming language (Java)

When we write program P 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 = { Java P | P(P) halts }
The Halting Problem

Is there a program HALT such that:

\[
\begin{align*}
\text{HALT}(P) &= \text{yes, if } P(P) \text{ halts} \\
\text{HALT}(P) &= \text{no, if } P(P) \text{ does not halt}
\end{align*}
\]

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

Suppose a program HALT existed that solved the halting problem.

\[
\begin{align*}
\text{HALT}(P) &= \text{yes, if } P(P) \text{ halts} \\
\text{HALT}(P) &= \text{no, if } P(P) \text{ does not halt}
\end{align*}
\]

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 dont 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 dont halt
  else exit;  //i.e., we halt
  // text of HALT goes here
}

Suppose CONFUSE(CONFUSE) halts:
then HALT(CONFUSE) = TRUE
\Rightarrow\text{CONFUSE will loop forever on input CONFUSE}

Suppose CONFUSE(CONFUSE) does not halt
then HALT(CONFUSE) = FALSE
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.

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

<table>
<thead>
<tr>
<th>All Programs</th>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>...</th>
<th>Pᵢ</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pᵢ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

YES, if $Pᵢ(Pᵢ)$ halts
No, otherwise
Let \( d_i = \text{HALT}(P_i) \)

All Programs

<table>
<thead>
<tr>
<th></th>
<th>( P_0 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>...</th>
<th>( P_j )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>( d_0 )</td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_1 )</td>
<td>( d_1 )</td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_i )</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>( d_i )</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


CONFUSE(\( P_i \)) halts iff \( d_i = \text{no} \)
(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.

Proof that \( R \) cannot be computed

Suppose it is, and program FRED computes it. then consider the following program:

```plaintext
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!
Computability Theory: Vocabulary Lesson

We call a set $S \subseteq \Sigma^*$ decidable or recursive if there is a program $P$ such that:

\[ P(x) = \text{yes, if } x \in S \]
\[ P(x) = \text{no, if } x \notin S \]

We already know: the halting set $K$ is undecidable

Decidable and Computable

Subset $S$ of $\Sigma^*$ $\Leftrightarrow$ Function $f_s$

- $x$ in $S$ $\Leftrightarrow$ $f_s(x) = 1$
- $x$ not in $S$ $\Leftrightarrow$ $f_s(x) = 0$

Set $S$ is decidable $\Leftrightarrow$ function $f_s$ is computable

Sets are “decidable” (or undecidable), whereas functions are “computable” (or not)

Oracles and Reductions

Oracle For Set $S$

Is $x \in S$?

YES/NO

Oracle for $S$
Example Oracle
S = Odd Naturals

Oracle for S

K₀ = 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 K₀.

But you can use this oracle for K₀ to build an oracle for K.

P = [input; Q]

Does P(P) halt?

BUILD: Oracle for K

Does [I:=P;Q] halt?

Given: Oracle for K₀

We’ve reduced the problem of deciding membership in K to the problem of deciding membership in K₀.

Hence, deciding membership for K₀ 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.

Hence, the set HELLO is not decidable.
Halting with input, Halting without input, HELLO, and EQUAL are all undecidable.

Diophantine Equations

Does polynomial $4x^2y + xy^2 + 1 = 0$ have an integer root? I.e., does it have a zero at a point where all variables are integers?

$D = \{\text{multivariate integer polynomials } P \mid P \text{ has a root where all variables are integers}\}$

Famous Theorem: $D$ is undecidable! [This is the solution to Hilbert’s 10th problem]

Resolution of Hilbert’s 10th Problem: Dramatis Personae

Martin Davis, Julia Robinson, Yuri Matiyasevich (1982)

Polynomials can Encode Programs

There is a computable function

$F: \text{Java programs that take no input} \rightarrow \text{Polynomials over the integers}$

Such that

program P halts $\iff F(P)$ has an integer root
\[ D = \text{the set of all integer polynomials with integer roots} \]

**GIVEN:** Oracle for \( D \)

**BUILD:**
- **HALTING Oracle**

Does program \( P \) halt?

\( F(P) \) has integer root?

**PHILOSOPHICAL INTERLUDE**

**CHURCH-TURING THESIS**

Any well-defined procedure that can be grasped and performed by the human mind and pencil/paper, can be performed on a conventional digital computer with no bound on memory.

The Church-Turing Thesis is NOT a theorem. It is a statement of belief concerning the universe we live in.

Your opinion will be influenced by your religious, scientific, and philosophical beliefs...

...mileage may vary
### Empirical Intuition

No one has ever given a counter-example to the Church-Turing thesis. I.e., no one has given a concrete example of something humans compute in a consistent and well defined way, but that can't be programmed on a computer. The thesis is true.

### Mechanical Intuition

The brain is a machine. The components of the machine obey fixed physical laws. In principle, an entire brain can be simulated step by step on a digital computer. Thus, any thoughts of such a brain can be computed by a simulating computer. The thesis is true.

### Quantum Intuition

The brain is a machine, but not a classical one. It is inherently quantum mechanical in nature and does not reduce to simple particles in motion. Thus, there are inherent barriers to being simulated on a digital computer. The thesis is false. However, the thesis is true if we allow quantum computers.