At the Foot of the
Mountains of Undecidability

6 May 2021
Assignment 8
  Corrections due today

Assignment 9
  Out today

Exam 2
  Graded someday, someday
“What problems can we solve with a computer?”
“What problems can we solve with a computer?”
The *Church–Turing thesis* states:

Every effective method of computation is either equivalent to or weaker than a Turing machine.
Problems solvable by any feasible computing machine

Regular languages

Context-free languages

All languages
Problems solvable by Turing machines

- Regular languages
- Context-free languages

All languages
“What problems can we solve with a computer?”

What is a “problem”?
A *decision problem* is a type of problem where the goal is to answer yes or no.
How can we represent inputs?

Think about files on your computer:

Each file represents some data, but each file is encoded purely using 0s and 1s.

If $Obj$ is an object, then $\langle Obj \rangle$ denotes some string representing $Obj$, like how it might be stored on disk.

We can encode multiple objects as a single string, e.g., $\langle x, y \rangle$. 
Now we can ask “what’s 5 + 3?” by asking for each possible $x$ whether $\langle 5, 3, x \rangle$ is in the language of sums.
Our goal is to speak of *computers solving problems*. We model this by looking at *Turing machines recognizing languages*. By turning any problem into an equivalent *decision problem*, this precisely captures what we’re interested in.
“What problems can we solve with a computer?”

What does it mean to "solve" a problem?
Unlike finite automata, which automatically halt after reading the input, Turing machines keep running until they explicitly enter an accept or reject state.

As such, it’s possible for a Turing machine to run forever without accepting or rejecting.
If a Turing machine might run forever, how do we formally define what it means to “build a Turing machine for a language”?

What implications does this have for problem-solving?
Terminology

Let $M$ be a Turing machine.

$M$ accepts a string $w$ if it enters an accept state when run on $w$.

$M$ rejects a string $w$ if it enters a reject state when run on $w$.

$M$ loops infinitely (or just loops) on a string $w$ if, when run on $w$, it never enters an accept or reject state.
M does not accept $w$ if it either rejects $w$ or loops infinitely on $w$.

M does not reject $w$ if it either accepts $w$ or loops on $w$.

M halts on $w$ if it accepts $w$ or rejects $w$. 
The **language of a Turing machine** $M$ is

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

If $w \in L(M)$, $M$ accepts $w$.

If $w \not\in L(M)$, $M$ does not accept $w$.

That is, when $M$ is run on $w$, either it rejects or it loops forever.
Does this correspond to what you think it means to “solve a problem”?
A language $L$ is *Turing-recognizable* (or just *recognizable*) if there is a TM $M$ such that

$$\forall w \in \Sigma^*. (M \text{ accepts } w \iff w \in L).$$

This is a “weak” notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \not\in L$, then $M$ does not accept $w$.
- If might reject $w$ or it might loop on $w$.

The class $\text{RE}$ consists of all Turing-recognizable languages.
If a Turing machine $M$ halts on every possible input – i.e., it never goes into an infinite loop – then we call $M$ a \textit{decider}.

For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting:
A language $L$ is **Turing-decidable** (or just **decidable**) if there is a TM $M$ such that

\[ \forall w \in \Sigma^*. \ (M \text{ accepts } w \iff w \in L) \land (M \text{ halts on all inputs}). \]

This is a “strong” notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

The class $\mathbf{R}$ consists of all Turing-decidable languages.
Decidable problems – the languages in $\mathbb{R}$ – are problems that can truly be “solved” by a computer. (Though that solution isn’t guaranteed to be acceptably fast.)
All regular languages are in $\mathbb{R}$.

We can use a Turing machine to simulate a DFA, and DFAs always halt.

$$\{0^n1^n \mid n \in \mathbb{N}_0\} \in \mathbb{R}.$$ Proof: The Turing machine we built is a decider; it always halts.

In fact, all context-free languages are in $\mathbb{R}$.

The proof of this is trickier. It relies on using CFGs rather than PDAs. See Sipser page 200.
R matters because it is exactly the class of languages for which there is an algorithm to decide if a string is in the language.

By the Church–Turing thesis, this isn’t just about Turing machines.

If there is any algorithm to decide membership in the language, then there is a decider for it.
A feel for $R$ and $RE$

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$. 
A feel for \textbf{R} and \textbf{RE}

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

\textit{Not whether the language is $a^n b^n$; that’s impossible for a DFA!}
A feel for **R** and **RE**

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

**An RE perspective**: Run the DFA on $a^0 b^0$, $a^1 b^1$, $a^2 b^2$, etc. If the DFA ever accepts, return true. But, if not, you may never learn this.

**An R perspective**: Look at the structure of the DFA and, somehow, determine whether it accepts any strings of this form, but without running the DFA on all of them.

*Can we do this?*
A feel for R and RE

Say you’re working on a computer science assignment. You wonder if there’s any input that will make your program crash.

An RE perspective: Try running the program on every possible input. If you see it crash, return true. If it never crashes, you will never learn this.

An R perspective: Look at the source code and somehow determine, with 100% certainty, whether the program will ever crash.
A feel for **R** and **RE**

You have an X. You want to see if there’s a Y where X and Y go well together.

**An RE perspective:** List all the Ys in some order and check if X and Y go well together. If so, return true. If not, you might not learn anything.

**An R perspective:** Look at X and, somehow, determine whether such a Y exists without checking all Ys.

Can we do this?
Intuition 1: Problems in RE are ones that can be approached by doing some sort of exhaustive search over a potentially infinite list of options.

Intuition 2: Problems in R are ones that can be solved without having to exhaustively try infinitely many possibilities.
Every decider is a Turing machine, but not every Turing machine is a decider.

So, $R \subseteq \text{RE}$.

But is $R = \text{RE}$?

That is, if you can confirm “yes” answers to a problem, can you also solve that problem?
Is this right?
Or this?

All languages

CFLs

Regular languages

R

RE
“What problems can we solve with a computer?”
We're getting closer!

However, to understand the answer, we’re going to need to step back for a moment.
Let’s think about *emergent properties*.

An emergent property of a system is a property that arises out of smaller pieces but which doesn’t seem to exist in any of the individual pieces. E.g.,

Individual neurons fire in response to particular combinations of inputs and this gives rise to human consciousness.

Individual atoms obey the laws of quantum mechanics and just interact with other atoms, and this gives rise to literally everything.
All computing systems equal to Turing machines exhibit several surprising emergent problems.

Because of the Church–Turing thesis, these must be *inherent* to computation; computation can’t exist without them.

These emergent properties are what ultimately make computation so interesting and powerful.

But they’re also computation’s Achilles heel – they’re how we find concrete examples of impossible problems.
The key emergent properties of computation that we’ll discuss are:

*Universality:* There is a single computing device capable of performing any computation.

*Self-reference:* Computing devices can ask questions about their own behavior.

The combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.
Emergent property: Universality
A central idea in the theory of computation is that of a *universal computer* – a computer powerful enough to simulate any other computing device.
The idea of a universal computer was described by Turing in 1937.

Like many computing pioneers, Turing was interested in the problem of making a computer that could *think*. Towards this end, he invented a scheme for a general-purpose computing machine.
Turing referred to his imaginary construct as a “universal machine” since at the time “computer” still meant a person – typically a woman – who performed computations.
We’ve been designing Turing machines to solve specific problems.

Do you have a dedicated computer for each task you need to perform?

Your email computer and your word-processing computer and your cute-cat-picture computer?
Most computers we encounter in everyday life are universal computers.

With the right software and enough time and memory, any universal computer can simulate any other type of computer.

To have a real computer perform a particular task, we load a program into it and have the computer execute the program.
Can we make a “reprogrammable Turing machine”?
A Turing machine simulator

It’s possible to program a Turing machine simulator on an unbounded memory computer.

If we accept some limits on the “infinite” tape, we can even do this on a real computer.
Turing’s World 3.0, 1993
A Turing machine simulator

While a simulator like this is an interactive tool to help us understand the theoretical model, we can also imagine it as a method

```
bool simulateTM(TM M, string w)
```

with the following behavior:

- If $M$ accepts $w$, then `simulateTM(M, w)` returns `true`.
- If $M$ rejects $w$, then `simulateTM(M, w)` returns `false`.
- If $M$ loops on $w$, then `simulateTM(M, w)` loops infinitely.
Sketch of a Turing machine simulator

```
state = start
while True:
    if state.isAccepting():
        return True
    if state.isRejecting():
        return False
    c = tape.readSymbol()
    state, dir, out = state.next(c)
    tape.write(out)
    if dir == "left":
        tape.moveLeft()
    elif dir == "right":
        tape.moveRight()
```
simulateTM

\( M \)

\( w \ldots \text{input...} \)

\{true!, \( (\text{loop}) \), false!\}
Anything that can be done with an unbounded-memory computer can be done with a Turing machine.

So there must be a Turing machine that has the behavior of the simulateTM method.
A TM that runs other TMs. Given an input \( w \), the TM processes it. If it accepts, it outputs "accept!". If it rejects or enters a loop, it outputs "reject!" or "(loop)" respectively.
$M$  

$w$ \text{...input...}  

\text{Universal TM}  

\text{accept!}  

\text{(loop)}  

\text{reject!}
THEOREM (Turing, 1936): There is a Turing machine $U$ called the *universal Turing machine* that, when run on an input of the form $⟨M, w⟩$, where $M$ is a Turing machine and $w$ is a string, simulates $M$ running on $w$ and does whatever $M$ does on $w$.

If $M$ accepts $w$, then $U$ accepts $⟨M, w⟩$.
If $M$ rejects $w$, then $U$ rejects $⟨M, w⟩$.
If $M$ loops on $w$, then $U$ loops on $⟨M, w⟩$. 
On input $\langle M, w \rangle$, $U$ does what $M$ does on input $w$. 
The universal Turing machine $U$, schematically

Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

Machine $M$

Input $w$ $\cdots$ a a a a $\cdots$
Take $M$ and write it down as a string. (Remember how we can encode the finite-state control as table.)
Take \( M \) and write it down as a string. (Remember how we can encode the finite-state control as a table.)
Now take your input \( w \) and write it down too.
Now take your input $w$ and write it down too.

Machine $M$

Start

$q_0$ -> $q_1$, a

$q_1$ -> $q_{\text{acc}}$, R

$q_{\text{acc}}$

$q_{\text{rej}}$

Input $w$ ...

\[ \cdots a a a a a \cdots \]
Feed this into $U$. 

Input $\langle M, w \rangle$: $\cdots q_0 \ a \ \square \ \ R \ \cdots q_1 \ a \ \cdots a \ a \ a \ a \ a \ a \ a \ \cdots$

$M$: $\cdots \ a \ a \ a \ a \ \cdots$

Input $w$: $\cdots \ a \ a \ a \ a \ a \ a \ \cdots$
If $M$ is in an accepting state
Look at next char of $w$

If $M$ is in a rejecting state
Look up what $M$ should do upon reading $w$

Update state and tape
If $M$ is in an accepting state

If $M$ is in a rejecting state

Update state and tape

Input $w \ldots a a a a a \ldots$

Input $\langle M, w \rangle \ldots q_0 a \square R \ldots q_1 a \ldots a a a a a \ldots$
If $M$ is in a
accepting state
Look at next
cchar of $w$

If $M$ is in a
rejecting state
Look up
what $M$
should do
upon
reading $w$

Update state
and tape

Start

Input $w$ \[\cdots a^4 a \cdots\]

Input $\langle M, w \rangle$ \[\cdots q_0 a \; \square \; R \; \cdots q_1 a \; \cdots \; a^4 a \cdots\]
Machine $M$

Input $w$ \[ \cdots \quad a \quad a \quad a \quad a \quad \cdots \]

Input $\langle M, w \rangle$ \[ \cdots \quad q_0 \quad a \quad \square \quad R \quad \cdots \quad q_1 \quad a \quad \cdots \quad a \quad a \quad a \quad a \quad \cdots \]

$M$
Machine $M$:

- **Input $w$**: $\ldots a \ a \ a \ a \ a \ \ldots$

- **States**:
  - $q_0$ (start state)
  - $q_1$
  - $q_{\text{acc}}$
  - $q_{\text{rej}}$

- **Transitions**:
  - $\square \rightarrow \square$, $R$
  - $a \rightarrow \square$, $R$

- **Initial State**: $q_0$
- **Accepting State**: $q_{\text{acc}}$
- **Rejecting State**: $q_{\text{rej}}$

- **Actions**:
  - If $M$ is in an accepting state, look at the next character of $w$.
  - If $M$ is in a rejecting state, look at the next character of $w$.
  - Update state and tape.

**Input $\langle M, w \rangle$**:

- $\ldots q_0 \ a \ \square \ R \ \ldots q_1 \ a \ \ldots a \ a \ a \ a \ a \ \ldots$

- **States**:
  - $q_0$
  - $q_1$
  - $q_{\text{acc}}$
  - $q_{\text{rej}}$

- **Initial State**: $q_0$
- **Accepting State**: $q_{\text{acc}}$
- **Rejecting State**: $q_{\text{rej}}$

- **Actions**:
  - Look up what $M$ should do upon reading $w$.
  - Update state and tape.

- **Machine $M$**:

- **Input $w$**: $\ldots a \ a \ a \ a \ a \ \ldots$
If $M$ is in an accepting state

If $M$ is in a rejecting state

Look at next char of $w$

Look up what $M$ should do upon reading $w$

Update state and tape
If $M$ is in an accepting state

If $M$ is in a rejecting state

Look at next char of $w$

Look up what $M$ should do upon reading $w$

Update state and tape

Machine $M$

Input $w$: $\cdots a a a a a \cdots$

Input $\langle M, w \rangle$: $\cdots q_0 a \square R \cdots q_1 a \cdots a a a a a \cdots$
If $M$ is in an accepting state

If $M$ is in a rejecting state

Look at next char of $w$

Look up what $M$ should do upon reading $w$

Update state and tape
Machine $M$ 

Input $w$ \[\cdots a \ a \ a \ a \ a \ \cdots\]

Input $\langle M, w \rangle$ \[\cdots q_0 \ a \ \square \ R \ \cdots q_1 \ a \ \cdots a \ a \ a \ a \ \cdots\]

- If $M$ is in an accepting state
  - Look at next char of $w$

- If $M$ is in a rejecting state
  - Look up what $M$ should do upon reading $w$

Update state and tape

$M$

$w$
If $M$ is in an accepting state
Look at next char of $w$
Update state and tape

If $M$ is in a rejecting state
Look up what $M$ should do upon reading $w$
Machine $M$

- **Start state**: $q_0$
- **Accepting state**: $q_{\text{acc}}$
- **Rejecting state**: $q_{\text{rej}}$

Transitions:
- $q_0$: $\square \rightarrow \square$, R
- $q_1$: $a \rightarrow \square$, R
- $q_{\text{rej}}$: $\square \rightarrow \square$, R

Input $w$: $\cdots$ a a a $\cdots$

Input $\langle M, w \rangle$: $\cdots$ $q_0$ a $\square$ R $\cdots$ $q_1$ a $\cdots$ a a a $\cdots$

- **Start**: $\langle M, w \rangle \rightarrow q_0, R$
- **Update state and tape**
- **Look at next char of $w$**
- **If $M$ is in an accepting state**
- **If $M$ is in a rejecting state**

Diagram:
- Graph shows transitions between states $q_0$, $q_1$, $q_{\text{rej}}$, $q_{\text{acc}}$.
- Transitions include reading symbols and moving the tape.

Summary:
- Machine $M$ processes input $w$.
- If $M$ reaches an accepting state $q_{\text{acc}}$, it accepts.
- If it reaches a rejecting state $q_{\text{rej}}$, it rejects.
- The process involves looking at the next character, updating the state, and reading the input.
**Java program**: Solves one specific problem

**Turing machine**: Solves one specific problem

**Java simulator in Java**: Java program to simulate any Java program

**U**: Turing machine that can simulate any Turing machine
Since $U$ is a Turing machine, it has a language, $L(U)$.

What is the language of the universal Turing machine?
The language of $U$

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

$U$, when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, will

- accept $\langle M, w \rangle$ if $M$ accepts $w$,
- reject $\langle M, w \rangle$ if $M$ rejects $w$, and
- loop on $\langle M, w \rangle$ if $M$ loops on $w$. 

The language of $U$

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

$U$, when run on a string $⟨M, w⟩$, where $M$ is a TM and $w$ is a string, will

- accept $⟨M, w⟩$ if $M$ accepts $w$,
- reject $⟨M, w⟩$ if $M$ rejects $w$, and
- loop on $⟨M, w⟩$ if $M$ loops on $w$.

$L(U) = \{⟨M, w⟩ | M \text{ is a TM and } M \text{ accepts } w\}$

$= \{⟨M, w⟩ | M \text{ is a TM and } w \in L(M)\}$
The language of \( U \) is called \( A_{\text{TM}} \):

\[
A_{\text{TM}} = L(U) = \{\langle M, w \rangle \mid M \text{ is a TM and } w \in L(M)\}
\]

\( A_{\text{TM}} \) is the *acceptance language for Turing machines*.

Because there is a Turing machine, \( U \), that recognizes \( A_{\text{TM}} \), we know \( A_{\text{TM}} \in \text{RE} \).
Teaser: This language, $A_{TM}$, has some interesting properties beyond what we’ve seen.
Why do we care about universality?
Reason 1: Practical significance
What happens if we replace the Turing machine input with a normal computer program?
What happens if we replace the Turing machine input with a normal computer program?
Programs simulating programs

The fact that there’s a universal Turing machine combined with the fact that computers can simulate TMs and *vice versa* means that it’s possible to write a program that simulates other programs.

These programs go by many names:

An *interpreter* like the Java Virtual Machine or Python.

A *virtual machine* like VMWare or VirtualBox that simulates an entire computer.
Party like it’s 1999 1990

What are the first names of the four other members in your party?

1. Alan
2. Turing
3. Alonzo
4. Church
5. Post_

(Enter names or press Enter)

The key idea behind the universal TM is that TMs can be fed as inputs into other TMs.

Similarly, an interpreter is a program that takes other programs as inputs.

Similarly, an emulator is a program that takes entire computers as inputs.

This hits at the core idea that computing devices can perform computations on other computing devices.
Reason 2: Philosophical interest
Can computers think?

On 15 May 1951, Alan Turing delivered a radio lecture on the BBC, where he argued that “it is not altogether unreasonable to describe digital computers as brains”.

Why would he think this, given the very limited abilities of computers of the time?
“I should also say that ‘If any machine can be appropriately described as a brain, then any digital computer can be so described.’

“This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

“It can be shown to follow from a characteristic property of digital computers, which I will call their universality…”
“A digital computer is a *universal machine* in the sense that it can be made to replace any machine of a certain very wide class.

“It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results.
“In order to arrange for our computer to imitate a given machine it is only necessary to programme the computer to calculate what the machine in question would do under given circumstances, and in particular what answers it would print out. The computer can then be made to print out the same answers.”
“If now some machine can be described as a brain we have only to programme our digital computer to imitate it and it will also be a brain. If it is accepted that real brains, as found in animals, and in particular in men, are a sort of machine it will follow that our digital computer suitably programmed, will behave like a brain.”
“This argument involves several assumptions which can quite reasonably be challenged.”

Alan Turing, 1951
Emergent property: 
Self-referentiality
PINOCCHIO

MY NAME ISN'T PINOCCHIO.

BRANSON REESE

WOW!
This is your brain on self-reference.
We can write programs that act on themselves.

As a fun case, we can write quines – programs that print out their own source code.

The way to do this varies by language, but it tends to be confusing! E.g., a Scheme/Racket quine:

```scheme
((lambda (x)
    (list x (list 'quote x)))
   '(lambda (x) (list x (list 'quote x)))
```
The fact that we can write quines isn’t a coincidence.

**THEOREM** (Kleene’s second recursion theorem). It is possible to construct TMs that perform arbitrary computations on their own “source code” (the string encoding of that TM).

In other words, any computing system that’s equal to a Turing machine possess some mechanism for self-reference.
If this is already making your brain hurt, it’s ok.

When we think about “real” computer programs (rather than TMs), we can instead think of the program reading its own source code from disk:

```python
selfie.py:
def main(my_input):
    my_source = open("selfie.py").read()
    print(my_source)
```

A program that reads from disk doesn’t count as a true quine, but it lets us use the same idea.
Teaser: Self-reference lets machines compute on themselves. That will let them do *cruel and unusual things.*
A note on TM/program equivalence
Every Turing machine
receives some input,
does some work, then
(optionally) accepts or rejects.

We can model a Turing machine as a computer
program where
the program’s logic is written in a normal programming language, and
the program (optionally) calls the special function accept() to
immediately accept the input and reject() to immediately reject the
input.
Here’s a sample program we might use to model a Turing machine for \( \{w \in \{a, b\}^* \mid w \text{ has the same number of } a \text{ and } b\} \):

```python
def main(input):
    difference = 0
    for c in input:
        if c == "a":
            difference += 1
        elif c == "b":
            difference -= 1
        else:
            reject()
    if difference == 0:
        accept()
    reject()
```
We can always build a function my_source into a program which returns the source code of the program (as in a quine or reading from disk).

For example, here's a narcissistic program:

```python
def main(input):
    me = my_source()
    if input == my_source:
        accept()
    reject()
```
Self-defeating objects
A *self-defeating object* is an object whose essential properties ensure it doesn’t exist.

“I don’t know. I may be man’s best friend, but I’m my own worst enemy.”

Al Ross
The New Yorker
1991
**Question**: Why is there no largest integer?

**Answer**: Because if $n$ is the largest integer, what happens when we look at $n + 1$?
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then $n$ isn’t the largest integer.

Contradiction!
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then $n$ isn’t the largest integer.

Contradiction!

*We’re using $n$ to construct something that undermines $n$, hence the term “self-defeating”.*
The general template for proving that $x$ is a self-defeating object is as follows:

Assume that $x$ exists.

Construct some object $f(x)$ from $x$.

Show that $f(x)$ has some impossible property.

Conclude that $x$ doesn’t exist.

The particulars of what $x$ and $f(x)$ are and why $f(x)$ has an impossible property depend on the specifics of the proof.
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction.
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction.

Careful – we're assuming what we're trying to prove!
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there’s no contradiction.

Careful – we’re assuming what we’re trying to prove!

How do we know there’s no contradiction? We only checked one case!
You *cannot* show that a self-defeating object \( x \) exists by using this line of reasoning:

Suppose that \( x \) exists.
Construct some object \( g(x) \) from \( x \).
Show that \( g(x) \) has no undesirable properties.
Conclude that \( x \) exists.

The fact that \( g(x) \) has no bad properties doesn’t mean that \( x \) exists. It just means you didn’t look hard enough for a counterexample.
Teaser: Certain Turing machines can’t exist, as they’d be self-defeating objects.
We stand at the foot of the mountains of undecidability.

Next time we’ll see how the emergent properties of universality and self-reference bring us to the limits of computation.
Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis, *The Pattern on the Stone*
Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*