<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>42</td>
<td>18</td>
</tr>
<tr>
<td>47</td>
<td>18</td>
</tr>
<tr>
<td>100</td>
<td>19</td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
</tr>
<tr>
<td>10000</td>
<td>21</td>
</tr>
<tr>
<td>100000</td>
<td>21</td>
</tr>
<tr>
<td>100005</td>
<td>23</td>
</tr>
<tr>
<td>100042</td>
<td>24</td>
</tr>
<tr>
<td>100000000</td>
<td>21</td>
</tr>
<tr>
<td>1000000005</td>
<td>23</td>
</tr>
<tr>
<td>100000...00000</td>
<td>23</td>
</tr>
</tbody>
</table>

This green function seems alien!!!
These are fun fns!

\[ \text{cmp} \]
\[ \text{cmp}(5) == 17 \]

\[ \text{hc} \]
\[ \text{hc}(\text{CS5}) == \text{True} \]

Two uncomputable functions

these would be useful - if only they were possible

Office hours this week: Wednesday 2-4 (NOT Thursday!)

**PLAN!**

**Friday 5/3:** final project due (by 8pm)  
**come to labs for f.p. help!**

**Thursday 5/9:** final exam **review** @ class time (here)

**Tues. 5/14 @ 2 or 7:** final exam
All known and imagined computers can compute only what Turing Machines can... though some are faster than others

Quantum computation

Molecular computation
http://www.arstechnica.com/reviews/2q00/dna/dna-1.html

Parallel computers

Integrated circuits

Electromechanical computation

Water-based computation

Tinkertoy computation

Turing Machines!
TM's can do everything... though not fast

but FMs + TMs can get **unruly** pretty fast!
TM's can do everything...

... but only everything that *can* be computed!

There are – *surprisingly* – many mathematical functions that no program, computer, or Turing Machine™ can compute!

You can certainly *write* programs for them, but those programs – *provably* – have at least one bug! And, fixing the bug *necessarily* creates more!

Now *that's* a frustrating program to debug!
(Many) more **functions** than **programs**!

Uncountably infinite

This is how many int-bool **functions** there are.

.Countably infinite

This is how many **programs** there are.

the Reals from 0 to 1 $\mathbb{R}$

Positive integers $\mathbb{N}$
Cantor and "friends"

No one shall expel us from the paradise that Cantor has created.

"Charlatan"
"Renegade"
"Corrupter of Youth"

"Laughable and wrong"
"Utter nonsense"

And this was before 1900!

Leopold Kronecker
1823-1891

Lugwig Wittgenstein
1889-1951

David Hilbert
1862-1943
HOW MATH WORKS:

STEP 1: INSIGHT

MY GOD.
I WONDER
IF THIS IS
TRUE.

STEP 2: RESISTANCE

IMPOSSIBLE!
INSANE!
IT'S NOT JUST INCORRECT;
IT'S AN ENTIRELY NEW
CATEGORY OF STUPID!

(Many) more functions than programs! 

Uncountably infinite

This is how many int-bool functions there are.

Countably infinite

This is how many programs there are.

the Reals \( R \) from 0 to 1

Positive integers \( \mathbb{N} \)
Let's get **real**

-the reals (from 0 to 1) \{0.0 \leq r \leq 1.0\}

Are these sets equal-sized?

-the integers (positive) \{1, 2, 3, 4, 5, ...\}

Why is this **NOT** a valid matching?

Essence? Finite patterns!
Cantor Diagonalization

From *any* list, there are *always* real #'s missing!

<table>
<thead>
<tr>
<th>Real numbers are always missing</th>
<th>( r_0 = 0.4424 \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 = 0.333333333333 \ldots )</td>
<td>( \leftrightarrow 1 )</td>
</tr>
<tr>
<td>( r_2 = 0.424242424242 \ldots )</td>
<td>( \leftrightarrow 2 )</td>
</tr>
<tr>
<td>( r_3 = 0.314159426535897 \ldots )</td>
<td>( \leftrightarrow 3 )</td>
</tr>
<tr>
<td>( r_4 = 0.4200000000000000 \ldots )</td>
<td>( \leftrightarrow 4 )</td>
</tr>
<tr>
<td>( r_5 = \ldots )</td>
<td>( \leftrightarrow 5 )</td>
</tr>
</tbody>
</table>

\( r_i = \ldots \) if \( r_i[1] \neq 4 \) else 2

\( \mathbb{R} \) the Reals from 0 to 1

Positive integers \( \mathbb{N} \)
more **functions** than **programs**

**uncountably** many functions

Well-defined mathematical **functions** (uncountable)

What's swimming around out here?

**countably** many programs

Programs (countable)

Wow ~ I can't even **describe** the stuff out here!
Well-defined mathematical functions (uncountable)

What's swimming around out here?

uncountably many functions

programs (countable)

Wow ~ I can't even describe the stuff out here!

countably many programs

Human-describable things...
Well-defined mathematical functions (uncountable)

What's swimming around out here?

Wow ~ I can't even describe the stuff out here!

Programs (countable)

Human-describable things...

Next...

more functions than programs

uncountably many functions

countably many programs
But what's a specific function that can't be computed!?

an example!?

anything expressed w/ math...

any describable pattern...

the complexity of an integer
What is the **complexity** of an integer?

100 zeros total

100000000...0000000000000000000000000

each of these integers has the same # of digits

1568424042...2603635422093849215049

same number of total digits as above

Which *feels like* the more complex - or more compressible - number... *Why*?
What is the *complexity* of an integer?

**Intuition:** The *complexity* or *compressibility* of \( x \) is the length of the shortest description of \( x \).

Which "feels like" the more complex - or more compressible - number... *Why?*
The complexity, $\text{cmp}$, of an integer $x$ is the length of $x$'s shortest description.
The *complexity* of $x$, $\text{cmp}(x)$, is the length of the shortest zero-input function that outputs $x$. 

"description"

```
def f(): return 5
```

```
42 ➔ 18
```

The complexity, $\text{cmp}$, of an integer $x$ is the length of $x$'s shortest description.
The *complexity* of $x$, $\text{cmp}(x)$, is the length of the shortest zero-input function that outputs $x$.

So, $42$ has a complexity of $18$.  

Why $18$?
The **complexity** of \( x, \text{cmp}(x) \), is the length of the shortest zero-input function that outputs \( x \).

"description"

**Kolmogorov Complexity**

\[
\text{cmp} \left( \begin{array}{c} 42 \\ \end{array} \right) = 18
\]

**def** \( f() \): return \( 42 \)

... because, Python has 16 characters of overhead and 2 more are needed
def f(): return 100

... Python has 16 characters of overhead, so \( \text{cmp}(100) \) is 19
... Python has 16 characters of overhead, so \( \text{cmp}(1000) \) is 20.
... Python has 16 characters of overhead, so \( \text{cmp}(10000) \) is 21
Inputs (x)

5 → 17
42 → 18
43 → 18
100 → 19
1000 → 20
10000 → 21
100000 → 21

Outputs cmp(x)

?...

```python
def f():
    return 100000
```

... Python has 16 characters of overhead, but we can do better here!
... Python has 16 characters of overhead, and \(\text{cmp}(100000)\) is also 21.
**Quiz!**

What does `cmp(x)` return for each of these integers, `x`?

- `cmp(42) = 16 + 2 = 18`
- `cmp(9001) = 16 + 4 = 20`
- `cmp(1000000000) = 16 + 5 = 21`
- `cmp(1000000000) = 16 + 5 = 21`
- `cmp(10000042) = 16 + 7 = 23`
- `cmp(10000000000) = 16 + 5 = 21`
- `cmp(100000000000) = 16 + 7 = 23`
- `cmp(10000...000) = 16 + 11 = 27`
- `cmp(31415926...) = 16 + ____`
- `cmp(86753098...) = 16 + 1000 = 2000`

**Extra: What's the largest x with `cmp(x) == 20`?**

**Extra Extra: Are there any integers x with `cmp(x) > 1,000,000`?**

*Extra Extra Extra!* How big is the SMALLEST such integer?

http://www.googolplexwrittenout.com/
\[
\text{\texttt{cmp}(42)} = 16 + 2 = 18 \\
\text{\texttt{cmp}(9001)} = 16 + \underline{4} = 20 \\
\text{\texttt{cmp}(1000000)} = 16 + \underline{5} = 21 \\
\text{\texttt{cmp}(1000042)} = 16 + \underline{7} \text{ whoa!} = 23 \\
\text{\texttt{cmp}(100\ldots000)} = 16 + \underline{11} = 27 \\
\text{\texttt{cmp}(31415926\ldots)} = 16 + \underline{26} \text{ just estimate this one} \\
\text{\texttt{cmp}(86753098\ldots)} = 16 + \underline{1000} \\
\]

**Try this on the back page first...**

**Quiz!**

What does \texttt{cmp}(x) return for each of these integers, x?

**Extra:** What's the largest x with \( \text{\texttt{kc}}(x) = 20 \)?

**Extra Extra:** Are there any integers x with \texttt{cmp}(x) > 1,000,000?

**Extra Extra Extra!** How big is the SMALLEST such integer?
\[
\begin{align*}
\text{cmp}(42) &= 16 + \boxed{2} = 18 \\
\text{cmp}(9001) &= 16 + \boxed{4} = 20 \\
\text{cmp}(1000000) &= 16 + \boxed{5} = 21 \\
\text{cmp}(1000042) &= 16 + \boxed{7} = 23 \\
\text{cmp}(1000000000) &= 16 + \boxed{11} = 27 \\
\text{cmp}(31415926...) &= 16 + \boxed{2} = 22 \\
\text{cmp}(86753098...) &= 16 + \boxed{1000} = 23 \\
\text{cmp}(100...000) &= 16 + \boxed{9} = 25 \\
\text{cmp}(10000...000) &= 16 + \boxed{11} = 27 \\
\text{cmp}(1000000000) &= 16 + \boxed{11} = 27 \\
\text{cmp}(31415926...) &= 16 + \boxed{2} = 22 \\
\text{cmp}(86753098...) &= 16 + \boxed{1000} = 23 \\
\end{align*}
\]

**Try this on the back page first...**

**Quiz!**

What does \text{cmp}(x) return for each of these integers, \(x\)?

**Extra: What's the largest \(x\) with \(\text{kc}(x) = 20\)?**

**Yes, a LOT!**

**Extra Extra: Are there any integers \(x\) with \(\text{cmp}(x) > 1,000,000\)?**

**Yes, a LOT!**

**Extra Extra Extra!** How big is the SMALLEST such integer? **Sorry...**
Although \texttt{cmp}(x) is a well-defined mathematical function, with an int output for each int input \texttt{x},

\texttt{cmp}(x) is not a programmable function.

the complexity of \texttt{x}
the compressability of \texttt{x}

This is a function that \texttt{we will prove can't be programmed}...

How!?
Although \texttt{cmp}(x) is a well-defined mathematical function, with an int output for each int input \( x \),

\texttt{cmp}(x) is \textit{not} a programmable function.

the complexity of \( x \)
the compressability of \( x \)

You'll show \textbf{every possible} program for \texttt{cmp} has a bug! you

that we will prove \textit{can't be programmed}...

How!?
Although \texttt{cmp}(x) is a well-defined \textit{mathematical} function, with an int output for each int input \texttt{x},

\texttt{cmp}(x) \textit{is not} a \texttt{computable} function.

the \textit{complexity} of \texttt{x}  
the \textit{compressability} of \texttt{x}  

\texttt{all} of its programs \textit{have} bugs!

You'll show \textbf{every possible} program for \texttt{cmp} \textit{has} a bug! 
\textbf{you} \textit{will} prove \texttt{can't be computed}...

\textit{How!?}
We know \texttt{cmp(5) == 17}, so we propose this:

\begin{verbatim}
def cmp( x ):
    return x+12
\end{verbatim}
We know `cmp(5) == 17`, so we propose this:

```python
def cmp( x )
    return x+12
```

This `cmp` has a bug! (actually, **LOTS** of bugs!)
**cmp version #2**

We know \( \text{cmp}(x) \) is 16+___, so, let's try this:

```python
def cmp(x):
    return 16 + len(str(x))
```

It works for 5, 42, and 47!

**bug(s)?**
def cmp(x):
    return 16 + len(str(x))

It works for 5, 42, and 47!

We know \texttt{cmp}(x) \text{ is } 16+\_\_\_ ,
so, let's try this:

\texttt{cmp}(100000) \text{ is actually 21.} \quad \text{because } 10^{**5}

\textbf{Find a bug:}

\texttt{But, this cmp(100000) \text{ isn't 21.}} \quad \text{it's 22}

\texttt{So, this cmp also has a bug!} \quad \text{(lots, in fact...)}
cmp \sim any \text{ version}

Aargh! Let's try this \texttt{cmp(x)}:

\begin{verbatim}
def cmp(x):
    # do stuff and then...
    return answer
\end{verbatim}

I guess you'd call this part of the code a GRAY AREA...
You'd call this part of the code a \textit{GRAY AREA}…

Aargh! Let's try this \texttt{cmp(x)}:

```python
def \texttt{cmp}(x):
    do stuff and then...
    \texttt{return} answer
```

The challenge: we have to find a bug in \textit{any} possible version of \texttt{cmp(x)}!
We need to prove that this `cmp(x)` function contains a bug.

`def cmp( x ):`
  do stuff and then...
  `return answer`

We're going to write a function to find and show that bug!
def BFF():
    def cmp( x ):
        do stuff and then...
        return answer
    x = 0
    while cmp(x) < 1000000:
        x += 1
    return x

bug = BFF()

This version of cmp is 900,000 chars long and claims to return the complexity of x
```python
def BFF():
    def cmp(x):
        do stuff and then...
        return answer
    x = 0
    while cmp(x) < 1000000:
        x += 1
    return x

bug = BFF()
```

This version of `cmp` is 900,000 chars long and claims to return the complexity of `x`.

`x` is an integer positive has a complexity 2,000,000

`BFF?`

Bug-Finding Function!
def BFF():
    def cmp(x):
        do stuff and then...
        return answer
    x = 0
    while cmp(x) < 1000000:
        x += 1
    return x

bug = BFF()

This version of `cmp` is 900,000 chars long and claims to return the complexity of `x`

Note that BFF is a zero-input function that is 900,058 characters long...
... and it returns an integer, `bug`!

yet...

The complexity of `x` is the length of the shortest zero-input function that outputs `x.`
def BFF():
    def cmp(x):
        do stuff and then...
        return answer

    x = 0
    while cmp(x) < 1000000:
        x += 1
    return x

bug = BFF()

This version of \texttt{cmp} is \textbf{900,000} chars long and claims to return the complexity of \texttt{x}

\begin{itemize}
    \item [(1)] When does this \texttt{while} loop stop?
        \begin{enumerate}
            \item [A] when \texttt{cmp(x)} < 1,000,000 \ or
            \item [B] when \texttt{cmp(x)} >= 1,000,000 \ or
            \item [C] it \textit{never} stops
        \end{enumerate}

    \item [(2)] The \texttt{while} ensures the \texttt{x} BFF returns is
        \begin{enumerate}
            \item [T] an int \texttt{x} with complexity < 1,000,000
            \item [U] an int \texttt{x} with complexity >= 1,000,000
            \item [V] \textit{nothing} -- \texttt{x} is never returned at all
        \end{enumerate}

    \item [(3)] Why does this mean \texttt{cmp} has a bug?
        \begin{enumerate}
            \item [F] because \texttt{x} \texttt{!=} \texttt{bug}
            \item [G] \texttt{BFF} shows \texttt{bug}'s complexity is 900,058 \ or less!
            \item [H] there were no three-eyed aliens involved \(\smile\)
        \end{enumerate}
\end{itemize}
```python
def BFF():
    def cmp(x):
        do stuff and then...
        return answer

    x = 0
    while cmp(x) < 1000000:
        x += 1
    return x

bug = BFF()
```

This version of `cmp` is 900,000 chars long and claims to return the complexity of `x`.

Even this implementation of `cmp(x)` contains a bug!

```
number with complexity over a million!
```

```
zero-input f'n of size less than a million
```

```
cmp failed!
```

so, every implementation of `cmp(x)` contains a bug!

```
```

```
but, we allowed it to do anything at all!
```
Although \texttt{cmp (x)} is a \textit{well-defined} mathematical function, with an int output for each int input \texttt{x},

\texttt{cmp (x)} is \textit{not} a computable function.

\texttt{cmp (x)} can't be debugged!

\textit{proven!}
**Compression connection?**

The best-compressed version of any data $D$ is the *shortest program that generates $D$.*

---

**Minimum description length**

From Wikipedia, the free encyclopedia

The **minimum description length (MDL) principle** is a formalization of Occam's razor in which the best hypothesis for a given set of data is the one that leads to the best compression of the data.

[The MDL Principle] is based on the following insight: any regularity in a given set of data can be used to compress the data, i.e. to describe it using fewer symbols than needed to describe the data literally."

- Kolmogorov complexity is **uncomputable**: there exists no algorithm that, when input an arbitrary sequence of data, outputs the shortest program that produces the data.
Compression connection?

The best-compressed version of any data $D$ is the **shortest program that generates** $D$.

```
Compressed compressed
8=1 2!
9=3 4
*=like
/=them
+=here
1=hat
2=Sam-I-Am
3=I do not *
4=green eggs and ham
5=you *
6=+ or t+
7=I would not *
/
```

```
T8
T8
3 t8
Do 5 4?
3 /, 2.
9!
Would 5 / 6?
7 6.
7 anyw+. 9.
7 /, 2.
```

Total characters: 187 (95% compression ratio)

An impressive compression result!
Compression connection?

The **best-compressed** version of any data $D$ is the *shortest program that generates* $D$. 

Uncomputable!
Kolmogorov Directions

How do I get to your place from Lexington?

HMM...

OK, starting from your driveway, take every left that doesn't put you on a prime-numbered highway or street named for a president.

People get really grumpy when they realize you're giving them directions for how to go to the store and buy a GPS.

When people ask for step-by-step directions, I worry that there will be too many steps to remember, so I try to put them in minimal form.

Xkcd

Kolmogorov Complexity?
Some infinite patterns/functions have finite descriptions.

- they're all programmable/computable

More infinite patterns/functions do not have finite descriptions.

- they're not programmable/computable
Some infinite patterns/functions have finite descriptions.
- *they're all programmable/computable*

**More** infinite patterns/functions do **not** have finite descriptions.
- *they're **not** programmable/computable*
Two *useful* functions that *provably* can't be computed...

**Kolmogorov Complexity**

$\text{cmp}(x)$

$\sim$ the complexity of an integer $x$

$(compressibility)$

**Halt Checking**

$\text{hc}(f)$

$\sim$ does $f()$ halt or loop forever?
Haltchecking is uncomputable.

\[ \text{hc}( f ) \]

\[ \sim \text{ returns whether } f() \text{ halts or not} \]

\text{hc always has a bug!}
Haltchecking is uncomputable.

It is impossible to write a (bug-free) function \texttt{hc}(f) that determines if a function \texttt{f} halts when run:

- \texttt{hc}(f) returns \texttt{True} if \texttt{f}() halts and
- \texttt{hc}(f) returns \texttt{False} if \texttt{f}() loops infinitely
Suppose $hc(f)$ worked for all $f$  

Create this $\textbf{BFF}$:

```python
def BFF():
    if $hc$(BFF) == True:
        # never halt
        while 1+1==2: print 'Ha!'
    else:
        # halt
        return
```

Is $hc(BFF) == True$ ?

Is $hc(BFF) == False$ ?

$hc$ always has a bug  

Proven!
And this is important because ...
And this is important because ...

∞ loops are undetectable

some are detectable, but some are not – and there's no way to know!
loops are undetectable

some are detectable, but some are not
– and there's no way to know!

bugs are inevitable

infinite loops are just one type of bug...
In general, they're all undetectable
~ all behavioral, not syntactic, bugs

And this is important because ...

Rice's Theorem: CS81
And this is important because ...

∞ loops are undetectable

some are detectable, but some are not
– and there's no way to know!

bugs are inevitable

infinite loops are just one type of bug...
In general, they're all undetectable
~ all behavioral, not syntactic, bugs

programming is not automatable...

at least, not bug-free programming
Let's celebrate!

https://www.youtube.com/watch?v=vmiNGWsyWX0
Tutoring hours and labs through Friday...!

Good luck with final projects

They will halt all too soon!

xkcd's take is... ... all too True, in fact.

THE BIG PICTURE SOLUTION TO THE HALTING PROBLEM