Infinite weirdness and the limits of computation!

Don’t get me started… I could talk about infinity forever!
Last time…

What’s a computer?

Useful, but limited ~ *can't count*

Universal definition of *what can be computed*

---

**Finite State Machines**

![Finite State Machine Diagram]

**Turing Machines**

![Turing Machine Diagram]
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, of course, write programs for them, but those programs – *provably* – have at least one bug! *And* fixing a bug *necessarily* introduces more…

Now *that’s* a frustrating program to debug! 🐛
Uncomputable functions?

There are well-defined mathematical functions that no computer program can compute even with any amount of memory!

\[ f_B(x) = \begin{cases} 
1 & \text{if } x \text{ is odd} \\
0 & \text{if } x \text{ is even} 
\end{cases} \]

\[ \text{def } \text{progl}(x): \]
\[ \text{return } x \% 2 \]
functions vs. programs!

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

Positive integers \( \mathbb{N} \)

functions \( \checkmark \) vs. \( ? \) programs \( \checkmark \)
To infinity - and beyond

the Reals \( R \) from 0 to 1

Positive integers \( N \)

functions vs. programs

Uncountably infinite

\( > \)

Countably infinite

Pixar owes Cantor for this one!
Different infinities !?

There are infinitely many functions + programs…

… but not all infinities are created equal!

Two sets have equal size if their elements have a one-to-one matching.

This matching is called a bijection.

These sets have the same cardinality.
One-to-one mappings also define equal-sized infinite sets, but the results can be surprising!

Positive evens

Positive integers

ALL integers

$E = \{2, 4, 6, 8, 10, \ldots\}$

$N = \{1, 2, 3, 4, 5, \ldots\}$

$Z = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

$E$ and $N$ and $Z$ all have the same size! These sets are said to be countably infinite.

Prove that the set of all rationals $Q$ is also countably infinite.
This seems *irrational*

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>2/1</td>
<td>2/1</td>
</tr>
<tr>
<td>3/1</td>
<td>3/1</td>
</tr>
<tr>
<td>4/1</td>
<td>4/1</td>
</tr>
</tbody>
</table>

They *are* the same size!

\[ \frac{1}{1}, \frac{84}{7}, \frac{22}{15}, \ldots \]

\[ \{1, 2, 3, 4, 5, \ldots \} \]

\[ Q \quad \text{Rational #s} \quad \text{Positive ints} \quad N \]

Just make a list…
Let's get real

the reals (from 0 to 1)

\{ 0.0 \leq r < 1.0 \}

.1
.2
.3
...
.11
.12
...

Are these sets equal-sized?

the integers (positive)

\{ 1, 2, 3, 4, 5, \ldots \}

1
2
3
...
11
12
...

Why is this NOT a valid matching? (multiple flaws)
Cantor Diagonalization

There are always real #s missing from any list!

\[ r_1 = 0.333333333333333... \]
\[ r_2 = 0.424242424242424... \]
\[ r_3 = 0.314159426535897... \]
\[ r_4 = 0.090909090909090... \]
\[ r_5 = ... \]

\[ \text{Real #s are always missing} \]

\[ r_0 = r_0[i] = (r_1[i] + 1) \mod 10 \]

Proving that \([0,1]\) is uncountably infinite!
Cantor Diagonalization

There are always real numbers missing from any list!

positive integers

r0 = .4350...

\[ r_0[i] = (r[i] + 1) \mod 10 \]

r1 = .333333333333333...

r2 = .424242424242424...

r3 = .314159426535897...

r4 = .090909090909090...

r5 = ...

\[ r_0[i] = (r[i] + 1) \mod 10 \]

\[ r_1[i] = (r[i] + 1) \mod 10 \]

\[ r_2[i] = (r[i] + 1) \mod 10 \]

\[ r_3[i] = (r[i] + 1) \mod 10 \]

\[ r_4[i] = (r[i] + 1) \mod 10 \]

\[ r_5[i] = (r[i] + 1) \mod 10 \]
HOW MATH WORKS:

STEP 1: INSIGHT

MY GOD. 
I WONDER 
IF THIS IS 
TRUE.

STEP 2: RESISTANCE

IMPOSSIBLE! INSANE!

IT'S NO
IT'S CARR

STEP 6: TRANSMISSION TO STUDENTS.

HOW DO YOU NOT GET THIS CONCEPT?
WE SPENT AN HOUR ON IT YESTERDAY.
(Many) more functions than programs!

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

functions $\gg$ programs

Uncountably infinite

Countably infinite

Positive integers $\mathbb{N}$
Well-defined mathematical functions (uncountable)

What's swimming around out here?

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

Programs (countable)

Uncountably many functions

Countably many programs

Human-describable things… (finite)

Next…
But what’s a specific function that can’t be computed!? 

an example!? 

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

100000000...00000000000000000000000

each of these integers has the same # of digits

170117684...20006872822488857785601

same number of total digits as above

Which “feels like” the more complex – or more compressible - number... Why?

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</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>1000000</td>
<td>21</td>
</tr>
<tr>
<td>10000000</td>
<td>23</td>
</tr>
<tr>
<td>100000000</td>
<td>23</td>
</tr>
<tr>
<td>1000000000</td>
<td>23</td>
</tr>
<tr>
<td>1000000000000000000</td>
<td>27</td>
</tr>
</tbody>
</table>

### Outputs

The complexity, $kc$, of an integer $x$ is the length of $x$'s shortest description.

This green function seems alien!!!
The complexity of \( x \) is the length of the shortest zero-input function that outputs \( x \).
The *complexity* of $x$ is the length of the shortest zero-input function that outputs $x$.

Kolmogorov Complexity

$$\text{kc}(42) = 18$$

The int 42 has a complexity of 18. *Why 18?*

```python
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 $\text{kc}(100) = 19$
def f(): return 1000

... Python has 16 characters of overhead so $\text{kc}(1000)$ is 20
def f(): return 10000

... Python has 16 characters of overhead so \( \text{kc}(10000) \) is 21
Python has 16 characters of overhead but we can do better here!

so \( k(c(10000)) \) is also 21
What does \( k_c(x) \) return for each of these integers, \( x \)?

\[
\begin{align*}
k_c(42) &= 16 + 2 = 18 \\
k_c(9001) &= 16 + \_ = 20 \\
k_c(1000000) &= 16 + \_ \\
k_c(1000042) &= 16 + \_ \\
k_c(1000000000) &= 16 + \_ \\
k_c(42) &= 16 + 2 = 18 \\
k_c(1000000) &= 16 + \_ \\
k_c(9001) &= 16 + \_ = 20 \\
k_c(1000000000) &= 16 + \_ \\
k_c(10000...0000) &= 16 + \_ \\
k_c(31415926...) &= 16 + \_ \\
k_c(86753098...) &= 16 + \_
\end{align*}
\]

1 googol zeros
1 billion digits of pi, as an integer
1 googolplex
9 zeros
a billion
100 zeros
a googol
1 googol zeros
a googolplex
just estimate this one
1 thousand patternless digits

Extra: What’s the largest \( x \) with \( k_c(x) = 20 \)?

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

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

What does $kc(x)$ return for each of these integers, $x$?

- $kc(42) = 16 + 2 = 18$
- $kc(9001) = 16 + 4 = 20$
- $kc(1000000) = 16 + 5 + 10^{**6}$ (a million)
- $kc(1000042) = 16 + 7 + 10^{**6+42}$ vs. 1000042 (a million and 42)
- $kc(1000000000) = 16 + 5 + 10^{**9}$ (9 zeros)
- $kc(1000000...0000) = 16 + 7 = 23$

**Extra:** What’s the largest $x$ with $kc(x) = 20$?
- $9^{**9}$

**Extra Extra:** Are there any integers $x$ with $cmp(x) > 1,000,000$?
- Yes, a LOT!

**Extra Extra Extra:** How big is the SMALLEST such integer? Sorry...
Although $\text{kc}(x)$ is a well-defined mathematical function, with an int output for each int input $x$, the complexity of $x$ is not a programmable function.

This is a useful function that we will prove can’t be programmed…

You’ll show every possible program for $\text{kc}$ has a bug! How!? 
Although $\text{k}_c(x)$ is a well-defined *mathematical* function, with an int output for each int input $x$, $\text{k}_c(x)$ is not a *computable* function.

This is a useful function that we will prove *can’t be* computed ... You’ll show every possible program for $\text{k}_c$ has a bug! How!? 
Proof idea

(1) We know $kc(42) == 18$, so consider one possible program for $kc(x)$:

```python
def kc(x):
    return x - 24
```

(2) Find a bug:
Proof idea

(1) We know $\text{kc}(42) == 18$, so consider one possible program for $\text{kc}(x)$:

```python
def kc(x):
    return x - 24
```

$k\text{c}(43)$ should also be 18.

(2) Find a bug:

But this $\text{kc}(43)$ outputs 19, not 18.

*This $\text{kc}$ has a bug!* (actually, LOTS of bugs!)
Proof idea

(1) Fixed! Since $\textbf{kc}(42) == 18$ and $\textbf{kc}(43) == 18$, we consider this program for $\textbf{kc}(x)$:

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

(2) Find a bug:
Proof idea

(1) Fixed! Since \( \text{kc}(42) == 18 \) and \( \text{kc}(43) == 18 \), we consider this program for \( \text{kc}(x) \): 

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

(2) Find a bug: 

\( \text{kc}(100 \ldots 00) \) should be 23.

But, this \( \text{kc}(100 \ldots 00) \) isn’t 23. It’s 117.

So, this \( \text{kc} \) also has a bug! (lots, in fact…)
Proof idea

(1) Fixed! Since $\textbf{kc}(42)==18$ and $\textbf{kc}(10)==18$, we consider this program for $\textbf{kc}(x)$:

```python
def kc(x):
    do stuff and return
```

(2) Find a bug:

I guess you'd call this part of the code a GRAY AREA...
Proof idea

(1) Fixed! Since $\text{k}\text{c}(42) == 18$ and $\text{k}\text{c}(10) == 18$, we consider this program for $\text{k}\text{c}(x)$:

```python
def kc(x):
    do stuff and return
```

(2) Find a bug: ???

The challenge: we have to find a bug in any possible version of $\text{k}\text{c}(x)$!
We need to prove that this \texttt{kc}(x) function contains a bug.

\begin{lstlisting}[language=Python]
def kc( x ):
    do stuff and then...
    \texttt{return} answer
\end{lstlisting}

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

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

bug = BFF()

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

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

bug = BFF()

This version of \( kc \) 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, \( \textbf{bug} \)!

yet...

The \textit{complexity} of \( x \) is the length of the shortest zero-input function that outputs \( x \).
def BFF():
    def kc(x):
        do stuff and then...
        return answer
    x = 0
    while kc(x) < 1000000:
        x += 1
    return x

bug = BFF()

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

Note: this loop checks \texttt{kc(x)}, not \texttt{x} itself.

(1) When does this \texttt{while} loop stop?

- [A] when \texttt{kc(x)} < 1,000,000 or
- [B] when \texttt{kc(x)} >= 1,000,000 or
- [C] it \textit{never} stops

(2) The \texttt{while} ensures the \texttt{x} BFF returns is

- [D] an int \texttt{x} with complexity < 1,000,000
- [E] an int \texttt{x} with complexity >= 1,000,000
- [F] \textit{nothing} – \texttt{x} is never returned at all

(3) Why does this mean \texttt{kc} has a bug?

- [G] because \texttt{x} \neq \texttt{bug}
- [H] \texttt{BFF} shows \texttt{bug}'s complexity is 900,058 or less!
- [I] there were no three-eyed aliens involved : -)

The \textit{complexity} of \texttt{x} is the length of the shortest zero-input function that outputs \texttt{x}. 

Proof
```python
def BFF():
    def kc(x):
        do stuff and then...
        return answer
    x = 0
    while kc(x) < 1000000:
        x += 1
    return x

bug = BFF()
```

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

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

but, we allowed it to do anything at all!

so, **every** implementation of `kc(x)` contains a bug!
Although $\text{kc}(x)$ is a \textit{well-defined} mathematical function, with an int output for each int input $x$,

\[ \text{kc}(x) \text{ is not a computable function.} \]

\[ \text{kc}(x) \text{ can't be debugged!} \]
Compression connection?

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

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.

People get really grumpy when they realize you're giving them directions for how to go to the store and buy a GPS.
Some infinite patterns (functions) have finite descriptions.

- *they're all computable/programmable*

More infinite patterns (functions) do *not* have finite descriptions.

- *they're not computable/programmable*
Two *useful* functions that *provably can't be computed*...

\[
\text{kc}( x )
\]

\[\sim \text{ the } \textit{complexity} \text{ of an integer } x\]

\(\text{(compressibility)}\)

----

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

\[\sim \text{ does } f() \text{ halt or loop forever?}\]

\(\text{Halt Checking}\)
Haltchecking is uncomputable.

\(hc(f)\)

\(\sim\) returns whether \(f()\) halts or not

\(hc\) always has a bug!
Haltchecking is uncomputable.

```
def hc(f):
```

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

1. `hc(f)` returns `True` if `f()` halts and
2. `hc(f)` returns `False` if `f()` loops infinitely
Suppose $\text{hc}(f)$ worked for all $f$ Create this \textbf{BFF}:

```python
def BFF():
    if \text{hc}(BFF) == True:
        while 1+1==2: print 'Ha!'
    else:
        return  # halt!
```

Is $\text{hc}(\text{BFF}) == \text{True}$ ?

Is $\text{hc}(\text{BFF}) == \text{False}$ ?

$\text{hc}$ \textit{always} has a bug

Proven!
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

programming is not automatable...

not perfect programming, at least
the iPhone's icon for Google Maps …
Let's celebrate!

https://www.youtube.com/watch?v=vmINGWsyWX0
Good luck with final projects

They will halt all too soon!

Grutoring hours and labs through the week...

Xkcd's take is...

... all too True, in fact.
Hello, we have an Expert team of 70 Webdesigners and developers, who will do very impressive designing for you. They will start working on your Project Bug Finder.

Herr T, Despite the fact that I died in 1918 you may find the following advice important. I would be happy to work on your project but you will quickly find that it is impossible, nay undecidable. It is a simple consequence of my diagonalization argument for uncountable numbers that were such a program to exist it would be possible to create a program on which it could not work and hence you would obtain a contradiction and your original program would 'disappear in a puff of logic'. Gruß Gott, Georg

Bid Time: 11-25-2008 08:08

The purpose of this project is to create a debugger program. This program will take as input the source code another program, and will analyze that other program and determine if it will run to completion, or have an error, or go into an infinite loop.

To state that another way, given a function f and input x, determine if f(x) will halt.