Infinite weirdness and the limits of computation!

Don’t get me started… I could talk about infinity forever!
Some pretty.py Pictures!

Phoebe Chan

Gracyn Buenconsejo

Elissa Hou & Mandy Wu

Lulu Pinczower

Celine Wang

Aech Loar
Prof Bush and Prof Ran turn Greenies into Python

CS5 Green

STARS!

Callie Glanton (2016)
What's a computer?

Useful, but limited ~ can't count

Universal definition of what can be computed

Finite State Machines

Turing Machines

Extra™!
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 \textit{functions}? 

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

\begin{align*}
  f_B(x) &= \begin{cases} 
    1 & \text{if } x \text{ is odd} \\
    0 & \text{if } x \text{ is even}
  \end{cases} \\
  \textbf{def} \ progl1(x): & \textbf{return} \ x \ % \ 2
\end{align*}
functions vs. programs!

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

$\frac{\text{functions}}{?} > \frac{\text{programs}}{\checkmark}$
To infinity - and beyond

the Reals from 0 to 1

functions vs. programs

Strictly larger

Uncountably infinite

> Countably infinite

Positive integers

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

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

Positive integers

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

ALL integers

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

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

With no negative repercussions!

Prove that the set of all rationals \( \mathbb{Q} \) is also countably infinite.

```python
def e2n(e):
    return e // 2

def n2e(n):
    return 2 * n
def n2z(n):
    return (-1) ** (n % 2) * \
    (n - (n % 2)) // 2)
def z2n(z):
    return abs(z) * 2 + (z <= 0)
```
Let’s get real

the reals (from 0 to 1)

\{ 0.0 \leq r < 1.0 \}

Are these sets equal-sized?

the integers (positive)

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

\begin{align*}
&.1 \\
&.2 \\
&.3 \\
&\ldots \\
&.11 \\
&.12 \\
&\ldots \\
\end{align*}

\begin{align*}
&1 \\
&2 \\
&3 \\
&\ldots \\
&11 \\
&12 \\
&\ldots \\
\end{align*}

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

There are always real numbers missing from any list!

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

The Reals from 0 to 1

Positive integers

\[
\begin{align*}
\mathbf{r}_0 &= \langle 3 \rangle \\
\mathbf{r}_1 &= \langle .333333333333333 \rangle \\
\mathbf{r}_2 &= \langle .424242424242424 \rangle \\
\mathbf{r}_3 &= \langle .314159426535897 \rangle \\
\mathbf{r}_4 &= \langle .090909090909090 \rangle \\
\mathbf{r}_5 &= \langle \ldots \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{set } r_0[i] &= (r_1[i] + 1) \mod 10 \\
&= 1 \\
&= 2 \\
&= 3 \\
&= 4 \\
&= 5
\end{align*}
\]
Cantor Diagonalization

There are always real #s missing from any list!

The Reals from 0 to 1

Proving that [0,1) is uncountably infinite!
Real insanity!

How does the cardinality of $\mathbb{R}$ compare to the cardinality of (0, 1)?
Real insanity!

How does the cardinality of $\mathbb{R}$ compare to the cardinality of (0, 1)?

Even though (0,1) “fits in” $\mathbb{R}$, there exists a one-to-one mapping, so they have the same cardinality!
(Many) more **functions** than **programs**!

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

\[ \text{functions} \quad > \quad \text{programs} \]

Uncountably infinite

Countably infinite
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...
(finite)

Next…

more functions than programs

uncountably many functions
countably many programs
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

170117684…20006872822488857785601

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>5</th>
<th>17</th>
</tr>
</thead>
<tbody>
<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>1000005</td>
<td>22</td>
</tr>
<tr>
<td>100042</td>
<td>22</td>
</tr>
<tr>
<td>10000000</td>
<td>21</td>
</tr>
<tr>
<td>10000005</td>
<td>23</td>
</tr>
<tr>
<td>100000...00000</td>
<td>23</td>
</tr>
</tbody>
</table>

**Outputs**

This green function seems alien!!!

The complexity, $kc$, of an integer $x$ is the length of $x$’s shortest description.
The complexity of $x$ is the length of the shortest zero-input function that outputs $x$. 

“description”

The complexity, $kc$, of an integer $x$ is the length of $x$’s shortest description.
The **complexity** of \( x \) is the length of the **shortest zero-input function** that outputs \( x \).

\[
\text{Kolmogorov Complexity} \\
\text{\( kc(42) = 18 \)}
\]

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

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

... Python has 16 characters of overhead so \( \text{kc}(100) \text{ is } 19 \)
\textbf{Inputs}

\begin{align*}
5 & \rightarrow 17 \\
42 & \rightarrow 18 \checkmark \\
47 & \rightarrow 18 \\
100 & \rightarrow 19 \checkmark \\
1000 & \rightarrow 20 \checkmark \\
10000 & \rightarrow 21 \\
100000 & \rightarrow 21
\end{align*}

\textbf{Outputs}

```python
def f(): return 1000  
```

... Python has 16 characters of overhead so $\text{kc}(1000)$ is 20
... Python has 16 characters of overhead so $\text{kc}(10000)$ is 21

```
def f(): return 10000
```
... Python has 16 characters of overhead but we can do better here!

so $k(10000)$ is also 21
What does $kc(x)$ return for each of these integers, $x$?

- $kc(42) = 16 + \underline{2} = 18$
- $kc(9001) = 16 + \underline{\phantom{0}} = 20$
- $kc(1000000) = 16 + \underline{\phantom{0}}$
  - 6 zeros, a million
- $kc(1000042) = 16 + \underline{\phantom{0}}$
  - a million and 42
- $kc(1000000000) = 16 + \underline{\phantom{0}}$
  - 9 zeros, a billion
- $kc(10000...0000) = 16 + \underline{\phantom{0}}$
  - 100 zeros, a googol
- $kc(100...000) = 16 + \underline{\phantom{0}}$
  - 1 googol zeros, a googolplex
- $kc(31415926...) = 16 + \underline{\phantom{0}}$
  - 1 billion digits of pi, as an integer
  - just estimate this one
- $kc(86753098...) = 16 + \underline{\phantom{0}}$
  - 1 thousand patternless digits

**Extra**: What’s the largest $x$ with $kc(x) = 20$?

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

**Extra Extra Extra**: How big is the SMALLEST such integer?
\[ kc(42) = 16 + 2 = 18 \]
\[ kc(9001) = 16 + 4 = 20 \]
\[ kc(1000000) = 16 + 5 \]
\[ kc(1000042) = 16 + 7 \]
\[ kc(1000000000) = 16 + 5 \]
\[ kc(10000000000) = 16 + 7 \]

\[ kc(100\ldots000) = 16 + 11 = 27 \]
\[ kc(31415926\ldots) \] just estimate this one
\[ kc(86753098\ldots) = 16 + 1000 \]

\[ Extra: \text{ What’s the largest } x \text{ with } kc(x) = 20? \]
\[ 9**9 \]

\[ Extra Extra: \text{ Are there any integers } x \text{ with } kc(x) > 1,000,000? \]
\[ Yes, a LOT! \]

\[ Extra Extra Extra: \text{ How big is the SMALLEST such integer?} \]
\[ Sorry… \]
Although $\text{kc}(x)$ is a well-defined mathematical function, with an int output for each int input $x$,

$k_c(x)$ is not a programmable function.

the complexity of $x$
the compressability of $x$

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

You’ll show every possible program for $k_c$ has a bug!
How!? 
Although $kc(x)$ is a well-defined mathematical function, with an int output for each int input $x$,

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

- the complexity of $x$
- the compressability of $x$

This is a useful function that we will prove can’t be computed ... 

You’ll show every possible program for $kc$ has a bug!

How!? 
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
```

(2) Find a bug:
Proof idea

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

\begin{verbatim}
def kc(x):
    return x - 24
\end{verbatim}

\(\texttt{bc(43)}\) should also be 18.

(2) Find a bug:

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

\textbf{This \texttt{kc} has a bug!} (actually, LOTS of bugs!)
Proof idea

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

```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
```

It works for 42 and 43.

(2) Find a bug:

\( \text{kc}(100 \ldots 00) \) should be 23. Because \( 10^{100} \) is 117.

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 \( \text{kc}(42) == 18 \) and \( \text{kc}(10) == 18 \), we consider this program for \( \text{kc}(x) \):

```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 $\text{kc}(42) == 18$ and $\text{kc}(10) == 18$, we consider this program for $\text{kc}(x)$:

```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 $\text{kc}(x)$!
def kc(x):
    do stuff and then...
    return answer

We need to prove that this function contains a bug.

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

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()
def BFF():
    x = 0
    while kc(x) < 1000000:
        x += 1
    return x

bug = BFF()

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

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

The 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()

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

(1) When does this while loop stop?
[A] when \(\text{kc}(x) < 1,000,000\) or
[B] when \(\text{kc}(x) \geq 1,000,000\) or
[C] it \textit{never} stops

(2) The while ensures the \(x\) BFF returns is
[D] an int \(x\) with complexity \(< 1,000,000\)
[E] an int \(x\) with complexity \(\geq 1,000,000\)
[F] \textit{nothing} – \(x\) is never returned at all

(3) Why does this mean \(\text{kc}\) has a bug?
[G] because \(x \neq \text{bug}\)
[H] \textit{BFF} shows \textit{bug}'s complexity is 900,058 or less!
[I] there were no three-eyed aliens involved : -)
```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!

*so, every* implementation of `kc(x)` contains a bug!
Although $k_c(x)$ is a *well-defined* mathematical function, with an int output for each int input $x$, $k_c(x)$ *is not* a computable function.

$k_c(x)$ can't be debugged!
Compression connection?

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

<table>
<thead>
<tr>
<th>Compressed compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>8=1 2!</td>
</tr>
<tr>
<td>9=3 4</td>
</tr>
<tr>
<td>*=like</td>
</tr>
<tr>
<td>/=them</td>
</tr>
<tr>
<td>+==hero</td>
</tr>
<tr>
<td>1=hat</td>
</tr>
<tr>
<td>2=Sam</td>
</tr>
<tr>
<td>3=I do not</td>
</tr>
<tr>
<td>4=green</td>
</tr>
<tr>
<td>5=yes</td>
</tr>
<tr>
<td>6= or t+</td>
</tr>
<tr>
<td>7=I would not = /</td>
</tr>
<tr>
<td>T8 T8</td>
</tr>
<tr>
<td>3 t8</td>
</tr>
<tr>
<td>Do 5 4?</td>
</tr>
<tr>
<td>3 /,. 2.</td>
</tr>
<tr>
<td>9!</td>
</tr>
<tr>
<td>Would 5 / 6?</td>
</tr>
<tr>
<td>7 6.</td>
</tr>
<tr>
<td>7 anyw+.</td>
</tr>
<tr>
<td>9.</td>
</tr>
<tr>
<td>7 /,. 2.</td>
</tr>
</tbody>
</table>

Total characters: 187 (95% compression ratio)
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.
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*...

**Kolmogorov Complexity**

\[ \text{k}_c(x) \]

~ the *complexity* of an integer \( x \)

*(compressibility)*

**Halt Checking**

\[ \text{h}_c(f) \]

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

\[ hc(f) \]

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

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

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

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

Create this \textbf{BFF}:

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

Is \( hc(BFF) == True \)?

Is \( hc(BFF) == False \)?

\textbf{hc} \textit{always} has a bug \hspace{1cm} \textbf{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

Rice's Theorem: CS81
the iPhone's icon for Google Maps …
Let's celebrate!
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 immediately.

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

Description

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.
Grutoring hours and labs through the week…

They will halt all too soon!

Xkcd’s take is…

… all too True, in fact.

Good luck with final projects