Chapter 4: How Universal Are Turing Machines?

CS105: Great Insights in Computer Science
Cantor’s Diagonalisation

• How many fractions are there? *Infinite.*

• How many decimals are there? *Infinite.*

• Are they the same size infinity?

• Well, we can make an infinitely long list that includes every fraction:
List of Fractions

- Start with all the fractions where the numerator and denominator add up to 1, then 2, then 3.
- Every fraction must eventually appear.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0/1</td>
<td>1/5</td>
<td>0/9</td>
</tr>
<tr>
<td>0/2</td>
<td>2/4</td>
<td>1/8</td>
</tr>
<tr>
<td>1/1</td>
<td>3/3</td>
<td>2/7</td>
</tr>
<tr>
<td>0/3</td>
<td>0/7</td>
<td>3/6</td>
</tr>
<tr>
<td>1/2</td>
<td>1/6</td>
<td>4/5</td>
</tr>
<tr>
<td>0/4</td>
<td>2/5</td>
<td>0/10</td>
</tr>
<tr>
<td>1/3</td>
<td>3/4</td>
<td>1/9</td>
</tr>
<tr>
<td>2/2</td>
<td>0/8</td>
<td>2/8</td>
</tr>
<tr>
<td>0/5</td>
<td>1/7</td>
<td>3/7</td>
</tr>
<tr>
<td>1/4</td>
<td>2/6</td>
<td>4/6</td>
</tr>
<tr>
<td>2/3</td>
<td>3/5</td>
<td>5/5</td>
</tr>
<tr>
<td>0/6</td>
<td>4/4</td>
<td>...</td>
</tr>
</tbody>
</table>
And The Decimals?

- Can we list all the decimals?
- The “add to a constant” trick doesn’t work anymore, since we have decimals like 0.3333... where the digit sum is infinite.
- So, let’s say we can list them all.
- Here’s the list, hypothetically:
On The List?

• Read down the diagonal.
  0.5898032467...

• Add 1 to each digit (with wraparound).
  0.6909143578...

• The resulting decimal is not on the list!
  (Differs from the $i$th one in the $i$th digit.)

0.57953916570123654
0.98877675309679680
0.54921087722147810
0.97889400202076116
0.68930952976230064
0.73758318399567813
0.33201823212447767
0.62085164445848273
0.22859580612307950
0.26912510570620329

...
On The List?

- Read down the diagonal.
  0.5898032467...
  0.57953916570123654
  0.98877675309679680
  0.54921087722147810
  0.978894002076116
  0.68930952976230064
  0.952976230064...
  0.6909143578...

- Add 1 to each digit (with wraparound).
  0.6909143578...
  0.68930952976230064
  0.73758318399567813
  0.33201823212447767
  0.62085164445848273
  0.22859580612307950
  0.26912510570620329
  0.26912510570620329...

- The resulting decimal is not on the list! (Differs from the \(i\)th one in the \(i\)th digit.)
Conclusion

• You *can’t* make a list of all the decimals.
• You *can* make a list of all the fractions.
• There are more decimals (real numbers) than fractions (rational numbers)!
Each of these scripts counts $i$ backwards from 10 to 1 and pops 10 times.

For each, is there an initial value we can set $i$ to be that would cause the script to loop forever?
The Halting Problem

- Looping forever is one of the most annoying classes of programming errors.
- Would be great if a tool could automatically detect whether a program always halted.
- We’d like a subroutine haltTester that takes a program as input and returns true if the program halts on all inputs and false if some input makes it loop forever.
haltTester

• Scratch doesn’t let you set a variable to a program, but many languages do.

• Let’s imagine that we can.

• Let’s imagine that *haltTester* is a Scratch script that expects a program in a variable called “prog”.

• After it’s done its test, it sets a variable “answer” to 1 (True) if prog always halts and 0 (False) otherwise.
Contrary

- If a `haltTester` subroutine exists, can use it in other scripts.

- For example, this script takes a program “prog” as input, and, if prog is the program `contrary` and `haltTester` says it always halts, `contrary` loops forever.

- Otherwise it halts.
Contrary Analysis

- What does contrary do if prog is set to contrary?
  - If haltTester answers 1, that means contrary halts on all inputs, so it should halt on itself as input.
  - But, in this case, contrary loops forever!

- So, it must be that haltTester returns 0, meaning that contrary loops forever on some input.
  - But, in this case, note that contrary halts for any input, including contrary!
  - It can’t happen!
Halting Summary

- If `haltTester` says `contrary` always halts, then it purposefully loops forever when `prog = contrary`.
- If `haltTester` says `contrary` sometimes doesn’t halt, then it never loops, no matter what `prog` is.
- As with the Barber paradox, the problem here is our assumption, specifically, that a program like `haltTester` can be written.
- So, `haltTester` is a well-defined problem that no program can solve: It is *incomputable*. 
CS Implications

- There are many problems that turn out to be incomputable.
- All involve computations that might take an infinite number of comparisons to solve and you’re never quite sure when to stop.
- An open problem I posed in my thesis (finding optimal policies for partially observable Markov decision processes) was later shown to be incomputable.
Some have argued that since people can tell if programs halt but programs can’t tell if programs halt, people are fundamentally more powerful / intelligent than computers.

Hogwash.
• Take a number. Half it if it’s even. Otherwise, triple and add 1. Continue until 1 is reached.

• Any power of 2 will be brought to 1 quickly.

• Some take awhile: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

• No one knows if it always halts!

• We can’t (easily) solve the halting problem.
Subber

- Here’s an odd little aside.

- For many formal self-reference-based proofs, programs need to be able to refer to themselves.

- How do you do that?

- Consider a subroutine *subber*.

- It takes a string as input and produces a new string as output.

- The output is essentially a copy, but some special characters are converted (subbed).
def subber(q):
    o = ""
    for i in q:
        if i == str(1+1): o = o + ""
        elif i == str(1+1+1): o = o + q
        elif i == str(1+1+1+1): o = o + "\n"
        elif i == str(1+1+1+1+1): o = o + "\t"
        elif i == str(1+1+1+1+1+1): o = o + "\\n"
        elif i == str(1+1+1+1+1+1+1): o = o + "\\t"
        elif i == str(1+1+1+1+1+1+1+1): o = o + "\\\\"
        else: o = o + i
    return o
Self-Referential Program

- Running this program causes it to print precisely the program itself!

```python
def selfPrint():
    print subber("def selfPrint(): print subber(232)"")
```

- Can even include `subber`, too:

```python
def selfContained():
    print subber("def subber(q): o = 2245 for i in q:
        if i == str(1+1): o = o + '2'
        elif i == str(1+1+1): o = o + q
        elif i == str(1+1+1+1): o = o + 2
        elif i == str(1+1+1+1+1): o = o + 2
    o = o + i
return o")
```
Weird?

• Something weird and “birth”-like here. The program has a string, which is the program, which somehow has the string, which is the program...

• Should be infinitely big, but it’s not via clever use of variables and substitution.
• “Since computers can’t solve the Halting Problem, we are smarter.”: Uh, no.

• “Since the halting problem can’t be solved, one cannot reliably predict the future, even when the future is the future execution of a deterministic program. ‘Free Will’ is consequence of the Halting Problem.”: Hmm!