CSC165 fall 2014

Mathematical expression

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/165/F14/
416-978-5899

Course notes, chapter 5

Computer Science
UNIVERSITY OF TORONTO

Outline

problems without algorithms?

contradictory program

notes

an algorithm for everything?

Many of us become interested in computers because we want to solve problems in a systematic, repeatable way. We write programs that implement our systematic solutions, and at the heart of these are *algorithms*: sequences of executable steps.

You already have lots of experience devising algorithms in Python. However, Alonzo Church (lambda calculus) and Alan Turing (Turing machines) showed, before they even had computers we would recognize today, that some problems can't be solved by algorithms.

By anybody. On anything we would conceive of as a computer.

caveats

Turing and Church took pains to show that uncomputable problems existed for any reasonable computational model — anything like a computer running a program. We'll be a bit more modest and show that problems exist for which you cannot write an algorithm to solve them in Python.

It turns out that Python isn't the bottleneck. Modern programming languages running on modern computers are Turing Complete — they can solve (and fail to solve) the same class of problems as the abstract Turing machine considered by Alan Turing.

solve this

Consider the following short function. You should be able to devise an algorithm that predicts whether it will halt, without actually running it

```
def may_halt(s) :
    if len(s) % 2 == 0 :
        while True : pass
    else :
        print s + " has odd length."
```

By the way, why would it not be practical to check whether the function halts by just running it? Also, notice that just two behaviours are possible: either halt (gracefully, or with some exception), or don't halt.

nobody knows the answer (so far)

For over 70 years mathematicians have been stumped in trying to show that the following code halts for every natural number n:

```
def collatz(n) :
    while n > 1 :
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3*n + 1
    return "Reached 1..."
```

The mathematicians have checked (empirically) that the code halts on every n up to more than 2^{58}, but they can't show that it won't loop forever on some big n after that. There are only a few lines of code, and many thousands of eyes have stared at this problem for many years, yet they can't predict whether it will halt on **every** input.

the equalizer

No one in this room (or any other room of programmers) can correctly implement the function `H(f,i)` below:

```
def H(f, i):
    '''Return True if f(i) will halt, False otherwise.'''

    # Once you've figured out how to implement this, delete next line
    return True
```

`H` is passed a reference to function `f` and input `i`, so it can examine all the code in `f`, and all the data in `i`. In spite of this, there are cases of functions `f` with input `i` where the prediction is impossible (at least by an algorithm).

The fact that `H(f,i)` is not computable is a consequence of the fact that `navel_gaze(f)` is not computable for some `f`. We use the contrapositive: if `H(f,i)` is computable, then so is `navel_gaze(f)`

```
def navel_gaze(f) :
    while H(f, f) :
        pass

    return 42
```

But now we have the problem:

```
# what does navel_gaze(navel_gaze) do?
```

It halts if, and only if, it doesn't halt! (a contradiction).

terminology

If f is a well-defined function, that is we can say **what** $f(x)$ is for every x in some domain, but we can't say **how** to compute $f(x)$, then we say f is **noncomputable**. Otherwise, we say f is **computable**

We've just seen our first non-computable function: **halt**. There are many more.

reductions

Suppose I know some function f is non-computable, but that some other well-defined function g could be extended to build f. In the language of implication:

$$g \text{ computable} \Rightarrow f \text{ computable}$$

We say f **reduces** to g, and using the contrapositive:

$$f \text{ non-computable} \Rightarrow g \text{ non-computable}$$

partly-working halt...

Here's a quick-and-dirty "implementation" of `halt`:

```
def halt_0(f, i) :
    """ Does f halt on input i? """
    return (hash(f) + hash(i)) % 2 == 0
```

...and `navel_gaze` for reference:

```
def navel_gaze(g) :
    """ break any halting program! """
    while halt_0(g, g) :
        pass
    return 42
```

If you examine `halt` carefully, you'll conclude that the problem with `navel_gaze` will occur with almost **any** function that returns boolean values for **all** pairs of functions and inputs...

another uncomputable function

```
def initialized(g, v):
    '''g initializes v on every possible input
       (function, str) -> bool
    '''
    # missing implementation!

def halt(f,i):

    def f2(y):
        x = 42 # now x is initialized
        f(i)
        # this line reached iff f(i) halts
        del x
        print(x)

    return not initialized(f2, 'x')
```

No matter how cleverly you program, you can never properly match up every pair (function, input) with an appropriate `True/False` value.

This non-correspondence boils down to counting infinite sets... and coming up with different sizes!

$\infty \neq \infty$ What?!?

how we count finite sets...

When I count some finite set out loud, I put the elements into 1–1 correspondence with the set of names of some numbers, for example {one, two, three}.

I have to be careful that my counting is 1–1 (otherwise I overcount)

I have to be careful that my counting is onto (otherwise I undercount)

Does ∞ always equal ∞?

No. Sets A and B have infinitely many members if they each have more members than any finite number. Having more than any number isn't the same as saying they have the same size as each other!

Two sets (finite or infinite) have the same size (cardinality, to impress friends and family) if we can match them up (count) in a way that is 1–1 and onto.

what are 1–1 and onto anyway?

$f : A \mapsto B$ is 1–1: $\forall x, y \in A, f(x) = f(y) \Rightarrow x = y.$

$f : A \mapsto B$ is onto: $\forall y \in B, \exists x \in A, f(x) = y$

Is $f : \{\text{natural numbers}\} \mapsto \{\text{even natural numbers}\}$ $f(n) = 2n$ 1–1 and onto?

- Prove that $\forall m, n \in \mathbb{N}, 2m = 2n \Rightarrow m = n$
- Prove that for every even natural number m there is a natural number n such that $m = 2n$.

Whoa! — this means the set of natural numbers has the same size as its subset, the even natural numbers. True fact.

When $|A| \leq |\mathbb{N}|$ we say A is **countable**

The rational numbers, $\mathbb{Q}$ are countable. Lots of sets are countable. An informal test: could you come up with a procedure to "list" the elements of a set, each list element appearing beside a natural number?

Surprisingly, not all sets are countable. Cantor showed this using **diagonalization**

Notes