CSC165 winter 2013

Mathematical expression

divinotes - later next week.
A3 - office hours next week.
Danny Heap

heap@cs.toronto.edu BA4270 (behind elevators)

http://www.cdf.toronto.edu/~heap/165/W13/ 416-978-5899

Course notes, chapter 5





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.

LISP, Scheme, JavaScript, (Part of Python Algorithms), C, Java, (part of Python

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

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."

certainly with

py thon that

predicts, arthou

sunning, whethe

this halt.
```

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

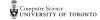
No one in this room (or any other room of programmers) can correctly implement the function H(f,i) below:

```
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

emperical approach, doesn't

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) assume A(f,i) is computable, & fequentions S, Vie Sinpuls ?

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

return 42

But now we have the problem:

what does navel_gaze(navel_gaze) do? Suppose nevel-gage (navel-gaze) halls => doesn't

It halts if, and only if, it doesn't halt! (a contradiction). n-g(n-g) Joesn't half soon half assumption half to computable is take.

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

I know f con be implemented using

g. Then f reduces to g.

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 non-computable \Rightarrow g non-computable any function that H reduces to is also non computable.

another uncomputable function

def g(n):

```
def halt(f,i):
   def initialized(g,v):
        ...code for initialized goes here...
    # Put some code here to scan the code for f and figure out
    # a variable name 'v' that does not appear anywhere in f.
     Stad V = 19 199 -> 1999
   def f_prime(x):
       # Ignore the argument x, call f with the fixed argument
       # (the one passed in to halt).
       f(i)
       print v
   return not initialized(f_prime,v)
```

Notes

