Algorithms II

What comes next, 1, 3, 7, 61, ?

Math Circle

March 11, 2018

Today we are going to continue our discussion of algorithms, but today we aren't going to talk about algorithms for solving specific problems, but rather we are going to talk about algorithms themselves. If this doesn't make sense, just hang on a second.

1. First, let's do a little intsy review of what we talked about last week. If these problems start off easy, great. We'll get to more difficult stuff, I promise

   (a) Describe in words what the following algorithm does:

```
function mystery(x,y)
r = x
while (r >= y)
    r = r - y
return r;
```

   (b) Remember when we talked about recursive functions/algorithms when we were talking about the Tower of Hanoi problem? Keep that in mind and figure out what this algorithm does. Ask the instructors if this still doesn't make sense.

```
function mystery2(a)
if (a == 1)
    return 1
if (a == 0)
    return 0
return mystery2(a - 2)
```

(c) What about this program? Assume that isPrime(x) returns true if x is prime, and false if x isn't prime. Also, assume that the input n is a natural number

```
function mystery3(n)
while (false == isPrime(n))
    n = n + 2
return n
```

(d) **What does the following function do? Here the variable list represents a list of numbers, and list[i] represents the i'th element in the list. so maybe if list = [0,-13,$-\frac{\pi}{4}$], then list[1] = 0, list [2] = -13, list[3] = $-\frac{\pi}{4}$, and list[4] doesn't exist. For this problem, assume that the number n no larger than the length of the list.**

```
function mystery4(x,list,n)
if (x == 0)
    return true
if (n == 0)
    return false
if (true == mystery4(x - list[n],list,n-1))
    return true
return mystery4(x,list,n-1)
```

(e) **Super challenging one here. What is mystery5(5,5)?**

```
function mystery5(m,n)
if (m == 0)
    return n + 1
if (n == 0)
    return mystery5(m-1,1)
return mystery5(m-1,mystery5(m,n-1))
```

2. Now let's talk a bit more about algorithms in general, but let's start with a really practical example. Suppose that Alice owns a computer, and Bob wants to use Alice's computer to run some of his code. Bob has been known in the past to write code that would just loop endlessly, without every stopping. In order to keep Bob from hogging Alice's computer, Alice wants to figure out if Bob's code is going to run forever on a given input.

(a) Before we get into particulars, what would you look for if your are Alice? Don't worry about getting it exactly right, just try and get a general idea. What kinds of things are probably Ok? What kinds of things are less Ok?

(b) Let's down to business. Suppose that Bob's code is finitely long, and has no recursion, while loops, or looping of any kind. What can Alice say about Bob's code?

(c) Things get tricky if you allow for recursion, so for now let's imaging that Bob's code had some while loops in it. Suppose that Bob's code looked like below. What could Alice conclude?

```
function function_name()
SOME CODE HERE
x = 0
while (2 > 1)
    CODE HERE
MORE CODE HERE
```

where SOME CODE HERE, CODE HERE and MORE CODE HERE is actually code of some kind.

(d) What should Alice think about this program?

```
function BobsFunction(n)
while (x = x)
    if (isASquare(x-1)  == true and isACube(x+1) == true)
        return x
    x = x + 1
```

For some values of n this is definitely ok. What about all values of n?

(e) **Should Alice let the following code of Bob's run for any input n?**

```
function twinsies(n)
while (2 > 1)
    if (isPrime(n) and isPrime(n + 2))
        return n
    n = n + 1
```

3. Alice's problem is famous enough that it has a special name. It's called the halting problem, because it amounts to figuring out if a function is going to halt for a given input. As it turns out, it is very, very hard to write down an algorithm to determine if another algorithm eventually halts or not.

   (a) Up until now we've been studying algorithms that take as input a number, but an algorithm could just as easily take list of characters as input. There's nothing earth-shatteringly different between numbers and letters as far as a computer is concerned.

   Let's say that you had a function that would solve the halting problem for you. That is, that there was a function Halt(alg,inut) that took two inputs. An algorithm (it could take the algorithm code in one big-long string for example) and an input. It would return true of alg(input) would stop eventually, and false otherwise.

   What would the output of Halt(myFun,)be?

```
function myFun()
x = 1;
while (-2 < 2)
    if (Halt(twinsies,x) == 1)
        x = twinsies(x) + 1
    if (Halt(twinsies,x) == 0)
        return x
```

   (b) There is a famous open problem in math called the twin prime problem. The conjecture says that there are infinitely many pairs of numbers $(n, n+2)$ such that both n and $n+2$ are prime. No one knows if this conjecture is true or not. Can you describe how you could solve this problem if you could write this Halt function?

(c) The above problem shows that writing the Halt function is at least as hard as solving the twin prime conjecture. Do you know of any other famous open math problems? Can you show that solving them is also easier than writing the Halt function?

(d) It turns out that you can prove that writing Halt is not just hard, it's impossible. Let's see why! If solving the halting problem was possible, what would the output of Halt(counterHalt,) be?

```
function counterHalt()
if (Halt(counterHalt,) == 0)
    return 1
if (Halt(counterHalt,) == 1)
    x = 1;
    while (1 == 1)
        x = x + 1
```

(e) How does your answer to the above problem show that Halt can no exist?

(f) **Although halting problem is not solvable, there is an easier problem called the busy beaver problem. The busy beaver function (called BB) is also a function that analyzes other functions, but it does so in a strange way. One version of BB is as follows. BB(n) is the maximum number of times that an algorithm that doesn't go on forever can output the number 1, provided that the algorithm has no more than n characters, so BB(10) is maximum number of times that an algorithm with 10 characters or less, that doesn't go on forever, can output 1's. Prove that there is some n for which BB(n) is impossible to compute.**