CS50 Unsolvable Problems

Overview

Computers are amazing; they are the **artificial intelligence** that underpins some of our favorite video games, they can beat human beings at checkers, chess, and Jeopardy, and they can even automatically and safely drive cars on the road, even with unpredictable human drivers alongside them. But it turns out, computers can't do everything, and never will be able to. Such tasks, are called **unsolvable problems** because the design of a computer prevents it from arriving definitively at a solution.

This is the most theoretical topic we cover in CS50. Don't worry if it's a bit scary!

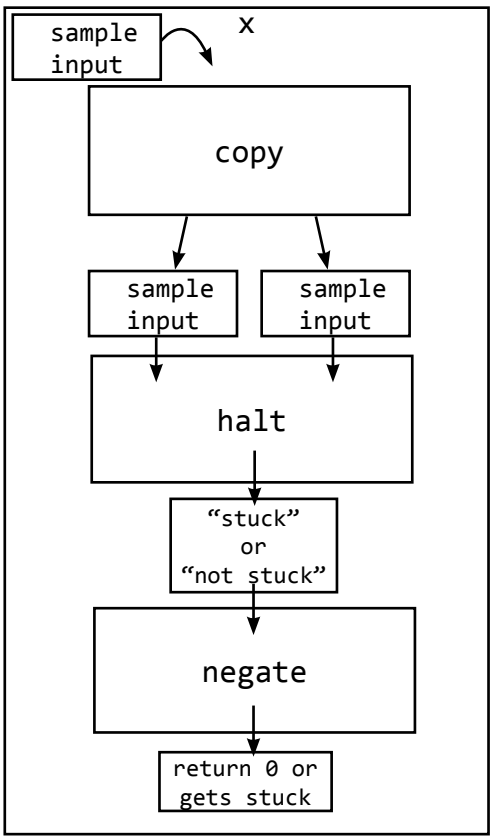
Key Terms

- unsolvable problems
- artificial intelligence
- input
- output

The Halting Problem

At this point, it is pretty clear to see the endless capabilities a computer has. It is difficult to even think of a problem that computers cannot solve, but let's try. Taking a closer look at computers, it is understood that they require an **input** and produce some **output**. Computers can only handle the inputs they were designed to handle. Take for example a calculator. It can handle inputs such as 3 + 8 and output 11, but it cannot sort an array of integers. Similarly, if you have a sorting program, it will not be able to handle arithmetic inputs. Let's imagine that we have a third program, called halt, one that takes a given program and a problem to solve as its inputs, and outputs whether or not it is going to get stuck. When we pass in calculator and "3 + 5" to halt, it will print "not stuck", but if we pass in calculator and "sort 1, 5, 2, 4, 9" it will print "stuck". Sounds simple enough, but this program cannot exist.

Consider a program, x, that has three main functions, copy, halt, and negate. Copy takes its input and outputs two of its inputs, halt takes a program and a problem as its inputs and outputs whether it will get stuck or not. Negate will take halt's output as its input. If it receives "stuck" it will return 0, if it receives "not stuck", it will get stuck. Sounds pretty complicated, but let's try with a simple case. If calc is passed in, halt determines that calc will get stuck if it tries to process "calc" since all it can do is arithmetic. Negate then gets stuck, which is fine. Let's pass in the program x, itself as the input. Copy takes x, and outputs two of them, halt gets x with the input of x. Say it spits out "stuck", then negate would return 0, which means that x, given the input x does not get stuck, so halt is wrong. That must mean that x with the input of x would be "not stuck". Pass x to copy again, halt gets x as the program and x as the problem. Since outputting "stuck" was a contradiction, let's try outputting "not stuck". Negate gets "not stuck" as its input. Negate then gets stuck. This means that x did get stuck when x was the input, but halt said it would not. This means that a program like halt cannot exist. It is impossible for a program like halt to be right 100% of the time using the same algorithm.



Alan Turing

Alan Turing proved that an algortihm that could test whether a program will run forever or not can not possibly exist. Reflect on the problems that we've tackled in this course. They were all finite. Even if we are checking the length of word, there exists a longest word, so that problem too, is finite. With all the tools available to us, there is none that can run infinetely and tell us if the program and its input would get stuck. In theory, we could test a program that is running for 1,000,000,000 seconds and output that the program got stuck, but it is plausible that the program could have processed the information on the 1,000,000,001th second. It is generally true that an infinite process can not be analyzed by a computer in a finite amount of steps.