CS50 Unsolvable Problems

Overview

Today, it seems like computers have endless capabilities. But it turns out, there are some things computers will never be able to do. Problems that computers cannot definitively arrive at a solution for are called **unsolvable problems**.

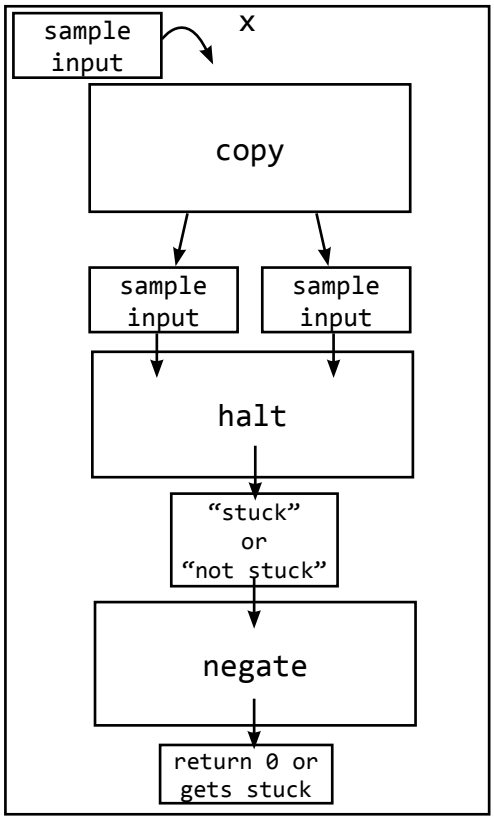
Key Terms

- unsolvable problems
- the halting problem
- heuristic

The Halting Problem

Generally speaking, computers operate by taking input and producing some output. But like any function, 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.

The halting problem is the computability theory problem of determining if a program will finish running, or "halt", given a program and specific input. To examine this problem, let's imagine that we have a program, called halt, that takes in a program and some inputs to that program and then outputs whether or not the given program is going to get stuck. When we pass in calc – a calculator program – and "3 + 5" – inputs to calc – to halt, halt will print "not stuck," but if we pass in calc and "sort 1, 5, 2, 4, 9," halt will print "stuck." Although this theoretical program sounds simple enough, it is proven that this program cannot exist.



Consider a program, x, that has three main functions: copy, halt, and negate. Copy takes input and outputs two of whatever is inputted, halt takes a program and inputs to the given program and outputs whether the given program will get stuck or not. Negate takes halt's output as its input. If negate receives "stuck" it will return 0, and if negate receives "not stuck," it will get stuck.

Let's use a simple case to understand how x works. If the program calc is passed into x, copy will output 2 calcs, and halt will determine that calc would get stuck with an input of calc because all it can do is arithmetic. "Stuck" would then be passed into negate, and x would ultimately output 0. Now, let's pass in the program x as the input to x. Copy takes x and outputs two of them, then halt gets x with the input of x. From here there are two cases: halt can output "stuck" or "not stuck." If halt outputs "stuck," then negate would return 0. This implies that x given the input x does not get stuck, so halt is wrong. If halt outputs "not stuck," then negate would get stuck. This implies that x given the input x does get stuck, so halt is wrong again. Halt is wrong in both possible cases, therefore by contradiction, 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.

Heuristics

We know that there cannot exist an algorithm that can, in a finite amount of time, tell us if a program will halt or not. However, that doesn't mean there aren't ways to work around this logical impossibility. To develop good-enough solutions to impossible or very difficult problems, computer scientists often use heuristics. **Heuristics** are approaches to solving a problem that are not guaranteed to be completely correct, but that work well enough for the matter at hand. For example, using Google Maps to estimate how long it will take to travel home is a heuristic. While Google Map's prediction may not ultimately be correct, it adequately fulfills its purpose – it's good enough. For the halting problem, a possible heuristic could be testing a program for up to 1,000,000,000 seconds and then outputting whether or not the program gets stuck. Although it is plausible that the program could have finished its calculation on the 1,000,000,001th second, chances are the program would not have terminated in a practical amount of time, so for our case, we can deem it "stuck."