Computability

The Halting Problem

The Halting Problem is an important problem in computer science theory that helps answer questions about which problems are computable by a machine.

Theorem 1. *Suppose there was a program* TestHalt(P, x)*, that returns true if program P halts on input x, and false otherwise. Such a program that determines if any arbitrary program halts cannot exist.*
Proof. *Consider the program*

```
Turing(P)
    if TestHalt(P, P): loop forever
    else: return
```

What would happen if we called Turing(Turing)*? There are two possible cases*

1. *If it halts, that means* TestHalt(Turing, Turing) *returned false and we go to the else case. However, by definition,* TestHalt(Turing, Turing) *should have returned true since* Turing(Turing) *halted.*

2. *If it loops forever, that means* TestHalt(Turing, Turing) *returned true so we pass the pass the if condition. However, by definition* TestHalt(Turing, Turing) *should have returned false since* Turing(Turing) *loops forever.*

In both cases we have a contradiction, therefore TestHalt *cannot exist.* ■

There's a couple takeaways from this. One of them is the idea that source code is data. Since source code is just a binary string, we can use it as input to another program. Also note that the argument in the proof is basically Cantor's diagonalization argument, where we force a contradiction.

Reductions

A reduction is an algorithm for transforming one problem to another problem. We write $A \to B$ (A reduces to B) if B can be used to solve A. Specifically, the transformation from A to B should take polynomial time so solving A can be done efficiently if solving B can be done efficiently. There are two important properties for reductions.

1. Difficulty flows in the direction of the arrow. So this means that solving B is at least as hard as A. If B was easier than A, we could use B to solve A, so A would become just as easy as solving B.

2. Efficient algorithms flow in the opposite direction of the arrow. By the same logic as the previous part, any efficient algorithm to solve B will allow us to solve A efficiently.

Thus, if we show that HALT $\to P$ for some problem P, P must be uncomputable since HALT is uncomputable.

Example 1. Suppose we had a program TESTHELLO(P, x) which returns true if $P(x)$ outputs "Hello World" before executing any other statement. Can such a program exist?
Yes, it is possible to construct such a program. We can run one instruction of $P(x)$, and return true if "Hello World" was outputted. We return false otherwise.

Example 2. Suppose we had a program TESTHELLO(P, x) which returns true if $P(x)$ outputs "Hello World" any time during its execution. Can such a program exist?
No, we will show a reduction from the Halting Problem.
Proof. Suppose we had a program TESTHELLO(P, x) with the specified behaivor. We could then construct TESTHALT as follows.

```
TestHalt(P, x):
        P' = A new program where we remove all print statements from P
             and add print('Hello World') before every return or exit statement
        return TestHello(P', x)
```

If $P(x)$ halts, then by our construction of P', $P'(x)$ will output "Hello World", and therefore TESTHELLO(P', x) will return true. If $P(x)$ never halts, then $P'(x)$ will never output "Hello World" because we removed all print statements, and therefore TESTHELLO(P', x) will return false. Thus, we have shown a reduction HALT $\to$ HELLO, and and therefore TESTHELLO cannot exist. ■

The key difference between these two examples was that the first one had a constraint of "before any other statement". In general, computers can easily test something under some constraint on the execution time or number of instructions. However, if we want to test the unconstrained behaivor of a program, usually this will be uncomputable because we cannot even know for sure whether a program halts or not.

Example 3. A function $f : \mathbb{N} \to \mathbb{N}$ is computable if there exists a program that takes in x, and returns $f(x)$. An increasing function is one such that $x \geq y \implies f(x) \geq f(y)$. Is every increasing function comptuable?
No. We can denote an increasing function as $f(0), f(1), \ldots$. Since f is increasing, there is no upper bound on $f(x)$, and therefore we require an infinite description for f since the sequence can increase as $x \to \infty$. By Cantor's diagonalization argument, this means the number of such functions is uncountable. A computer program is a finite length bitstring, and we know this set is countable. The set of increasing functions is uncountable, while the set of computer programs is countable, and this implies there must exist an increasing function which is not computable.