Halting Problem

CSE 311 Autumn 2025
Lecture 30

Uncountable set of functions $g: \mathbb{N} \to B$

…so we have that the set of binary-valued functions (with $\mathbb{N}$ as their domains) is uncountable.

We showed this in the last lecture using a similar argument to showing that the set of real numbers is uncountable.

Our Second big takeaway: countable number of Java Programs

How many Java methods can we write:

public boolean g(int input) ?

Can you list them?

Yeah!! Put them in **lexicographic** order

i.e. in increasing order of length, with ties broken by alphabetical order.

Wait…that means the number of such Java programs is countable.

And…the number of functions we're supposed to write is uncountable.

Our Second big takeaway: uncomputability

There are more functions $g: \mathbb{N} \to B$ than there are Java programs to compute them.

Some function must be **uncomputable**.

That is there is no piece of code which tells you the output of the function when you give it the appropriate input.

Not just Java

This isn't just about java programs. (all we used about java was that its programs are strings)…that's…well every programming language.

There are functions that simply cannot be computed.

Doesn't matter how clever you are. How fancy your new programming language is. Just doesn't work.*

*there's a difference between `int` and $\mathbb{N}$ here, for the proof to work you really need all integers to be valid inputs, not just integers in a certain range.

Does this matter?

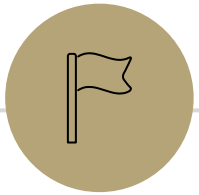
It's even worse than that – almost all functions are not computable.

So…how come this has never happened to you?

This might not be meaningful yet. Almost all functions are also inexpressible in a finite amount of English (English is a language too!)

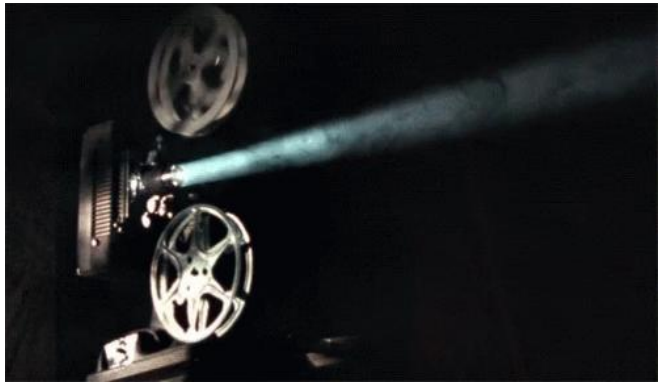
You've probably never decided to write a program that computes a function you couldn't describe in English…

Are there any problems anyone is **interested** in solving that aren't computable?



What’s a computer?

Formal definition of computer



To prove a theorem, we'll need some kind of definition.

The first theoretical description of a ("normal") computer is a **Turing Machine**

Turing Machines have:

1. A finite control
   - Think: a DFA-like object to represent program state, what is the program I'm executing, what line number am I on, etc.
2. As much memory as we need
   - Stored on an infinite "tape"
3. A "focus-of-attention"
   - The bit of memory on the tape we're paying attention to right now. Can read and write at this spot. The finite control can move the focus.

Look familiar?

[Read the story!]

Does that sound like a computer?

Maybe not the first analogy you would have come up with.

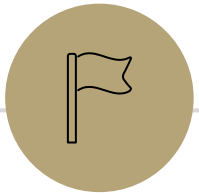
But computer scientists still love talking about Turing Machines---take 431 to learn more

But an idea called “Turing-completeness” says we can think about Java programs instead

A computational device is Turing-complete if it can do everything a Turing-machine can do.

Java is Turing-complete, as are C++, Python, and any other general-purpose programming language you know. And moreover, none can really solve problems the others can’t

See 431 for more on these.



The Halting Problem

A Practical Uncomputable Problem

Ever pressed the run button on your code and have it take a long time?

Like an infinitely long time?

Why didn't your compiler…like, tell you **not** to push the button yet.

It tells you when your code doesn't compile before it runs it…why doesn't it check for infinite loops?

The Halting Problem

The Halting Problem

Given: source code for a program P and x an input we could give to P
Return: True if P will halt on x, False if it runs forever (e.g. goes in an infinite loop or infinitely recurses)

This would be super useful to solve!

We **can't** solve it…let's find out why.

A Proof By Contradiction

Suppose, for the sake of contradiction, there is a program H, which given input `P.java`, x will accurately report

"P would halt when run with input x" or

"P will run forever on input x."

Important: H does not just compile `P.java` and run it. To count, H needs to return "halt" or "doesn't" in a finite amount of time.

And remember, it's not a good idea to say "but H has to run P.java to tell if it'll go into an infinite loop" that's what we're trying to prove!!

A Very Tricky Program

Run `Diagonal.java` given String input x (which could be a Java program)

```
class Diagonal { // taking input x, a Java program
   ...
   result = H(x, x)
   if (result == true) // H says "x halts on x"
      while (true) { // Go into an infinite loop
         int x = 2 + 2
      }
   else // H says "x doesn't halt on x"
      return; // Halt
   }
}
```

So, uhh that's a weird program. (1)

What do we do with it?

USE IT TO BREAK STUFF

Does `Diagonal.java` halt when its input is `Diagonal.java`?

Let's assume it does halt and see what happens…

Let's see what happens…

```
class Diagonal { // taking input x, a Java program
    ...
    result = H(x, x)
    if (result == true) // H says "x halts on x"
        while (true) { // Go into an infinite loop
            int x = 2 + 2
        }
    else // H says "x doesn't halt on x"
        return; // Halt
    }
}
```

Imagine Diagonal.java halts on Diagonal.java.
Then H better say it halts.
So it goes into an infinite loop.

Wait shoot.

So, uhh that's a weird program. (2)

What do we do with it?

USE IT TO BREAK STUFF

Does `Diagonal.java` halt when its input is `Diagonal.java`?

Let's assume it does halt and see what happens…
That didn't work. ✗

Let's assume it doesn't halt and see what happens…

What about this time? 😟

```
class Diagonal { // taking input x, a Java program
    ...
    result = H(x, x)
    if (result == true) // H says "x halts on x"
        while (true) { // Go into an infinite loop
            int x = 2 + 2
        }
    else // H says "x doesn't halt on x"
        return; // Halt
    }
}
```

Imagine Diagonal.java doesn't halt on Diagonal.java.
Then H better say it doesn't halt.
So we go into the else branch.
And it halts

Wait shoot.

So, uhh that's a weird program. (3)

What do we do with it?

USE IT TO BREAK STUFF

Does `Diagonal.java` halt when its input is `Diagonal.java`?

Let's assume it does halt and see what happens…
That didn't work. ✗

Let's assume it doesn't halt and see what happens…
That didn't work either. ✗

There's no third option. It either halts or it doesn't. And it doesn't do either. That's a contradiction! This program **H** can't exist!

So…

So there is no general-purpose algorithm that decides whether any input program halts (on any input string).

The Halting Problem is undecidable (i.e. uncomputable) there is no algorithm that solves every instance of the problem correctly.

But what about this? (1)

Does this code halt?

```
public static void main(String[] args) {
    int i=0;
    while(i<100){
        System.out.println("*");
        i++;
    }
}
```

It does halt!

But what about this? (2)

Does this code halt?

```
public static void main(String[] args){
    int i=0;
    while(i<100){
        System.out.println("*");
        //i++; //doesn't seem important.
    }
}
```

It doesn't halt!

What that does and doesn't mean

That doesn't mean that there aren't algorithms that often get the answer right

For example, if there's no loops, no recursion, and no method calls, it definitely halts. No problem with a program that checks that there's no loops, recursion, or method calls and tells you it's going to halt if it doesn't find any.

This isn't just a failure of computers – if you think **you** can do this by hand, well…

…you can't either.

Takeaways

Don’t expect that there’s a better IDE/better compiler/better programming language coming that will make it possible to tell if your code is going to hit an infinite loop.

It’s not coming.

More Uncomputable problems

Imagine we gave the following task to 121 students:

Write a program that prints “Hello World”

Can you make an autograder?

Technically…NO!

In practice, we declare the program wrong if it runs for 1 minute or so. That’s not right 100% of the time, but it’s good enough for your programming classes.

How Would we prove that?

With a **reduction**

Suppose, for the sake of contradiction, I can solve the HelloWorld problem. (i.e. on input P.java I can tell whether it eventually prints HelloWorld)

Let **`W`** solve that problem.

Consider this program…

A Reduction

```
// takes a program P input, and x String input
class Trick {
    result = P(x)
    // only simulate printing if P prints things
    print("Hello World")
}
```

This actually prints “hello world” iff P halts on x.

Plug `Trick` into W and….we solved the Halting Problem!

Reductions in General

The big idea for reductions is “reusing code”

Just like calling a library

But doing it in contrapositive form.

Instead of

“If I have a library, then I can solve a new problem”,

Reductions can do the contrapositive:
“If I can solve a problem I know I shouldn’t be able to, then that library function can’t exist”

Fun (Scary?) Fact

Rice's Theorem

Says any "non-trivial" input-output behavior of programs cannot be computed (in finite time).

What Comes next?

CSE 312 (foundations II)

Fewer proofs ☹

Basics of probability theory (super useful in algorithms, ML, and just everyday life).
Fundamental statistics.

CSE 332 (data structures and parallelism)

Data structures, a few fundamental algorithms, parallelism.

Graphs. Graphs everywhere.

Also, induction. [same for 421, 422 the algorithms courses]

CSE 431 (complexity theory)

What can't you do with computers **in a reasonable amount of time.**

Beautiful theorems – more on CFGs, DFAs/NFAs as well.

We've Covered A LOT

Propositional Logic.

Boolean logic and circuits.

Boolean algebra.

You'll use quantifiers in 332 to define big-O

Predicates, quantifiers and predicate logic.

Inference rules and formal proofs for propositional and predicate logic.

English proofs.

Set theory.

431 is basically 10 weeks of fun set proofs.

Modular arithmetic.

Prime numbers.

Interested in crypto? They'll come back.

GCD, Euclid's algorithm and modular inverse

No really. A lot

Induction and Strong Induction.

Lots of induction proof [sketches] in 332

Recursively defined functions and sets.

Structural induction.

Regular expressions.

You'll see these in compilers

Context-free grammars and languages.

One-to-one and Onto

Graphs

You'll use graphs at least once a week for the rest of your CS career.

Like A lot a lot.

DFAs, NFAs and language recognition.
Cross Product construction for DFAs.
Finite state machines with outputs at states.
Conversion of regular expressions to NFAs.
Powerset construction to convert NFAs to DFAs.
Equivalence of DFAs, NFAs, Regular Expressions
Method to prove languages not accepted by DFAs.
Cardinality, countability and diagonalization
Undecidability: Halting problem and evaluating properties of programs.

Promise you won't ever try to solve the Halting Problem? It's tempting to try to sometimes if you don't remember it's undecidable