The Limits of Computation

or

"a fascinating tale of infinity and paradox"

George Papamakarios

A fundamental question...

What can computers do?

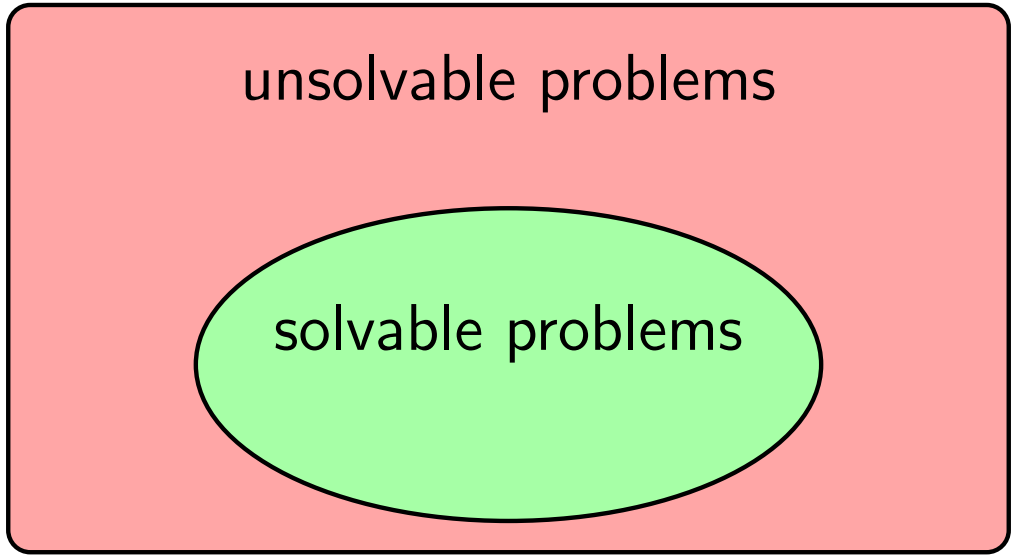
We created computers to help us solve **problems**, and computers have done that astonishingly well. But since the early days of computation, i.e. the first half of the 20th century, people wondered whether there are any problems that computers couldn't solve.

This is a fundamental question in theoretical computer science; can computers solve any computational problem? Or are there problems that no algorithm can ever solve? Similarly to asking what shapes we can construct with a ruler and a compass, we ask what problems we can solve with devices capable of computation. Can we solve them all? Or, like with ruler-and-compass constructions, there is a limit?

We will start by spilling the beans and answering this question straight away...

. . . and the answer!

There exist problems that no algorithm can solve!



The answer to whether computation is all-powerful is a big, cold-hearted **no**. There are problems, and by that I mean **computational** problems, that no algorithm can possibly solve.

Several questions naturally arise. Why is this the case? What do these unsolvable problems look like? Should we care about them? Where does the boundary between solvable and unsolvable problems lie?

Two proofs

- Proof #1: The **infinity** proof
- Proof #2: The **paradox** proof

We're going to discuss two proofs (or rather proof sketches) of the statement that unsolvable problems exist. These will provide some insight into the questions we asked above.

The first proof, I call it "the infinity proof", is **non-constructive**; it will prove the existence of unsolvable problems without naming any of them! The idea behind the proof is embarassingly simple: we will just **count** all possible programs and all possible problems. We will discover that there are more problems than programs, and hence conclude that there simply aren't enough programs to solve them all!

The second proof, I call it "the paradox proof", is **constructive**. We will consider a specific problem, the notorious **halting problem**, and we will show that if a program can solve it, then logical mayhem will follow! Hence, the halting problem will be shown to be unsolvable by contradiction.

The infinity proof

Step 1: count all programs

- A program is a string.
- There are infinitely many strings...
- ...but countably many; strings are **listable**.
- There are as many programs as numbers in $\mathbb{N}$.

The infinity proof

Step 2: count all problems

- A problem is a (desired) transformation from an input to an output.
- Inputs and outputs are strings.
- Strings are isomorphic to $\mathbb{N}$...
- ...hence problems are mappings from $\mathbb{N}$ to $\mathbb{N}$...
- ...which in turn are isomorphic to $\mathbb{R}$.
- There are as many problems as numbers in $\mathbb{R}$.

The infinity proof

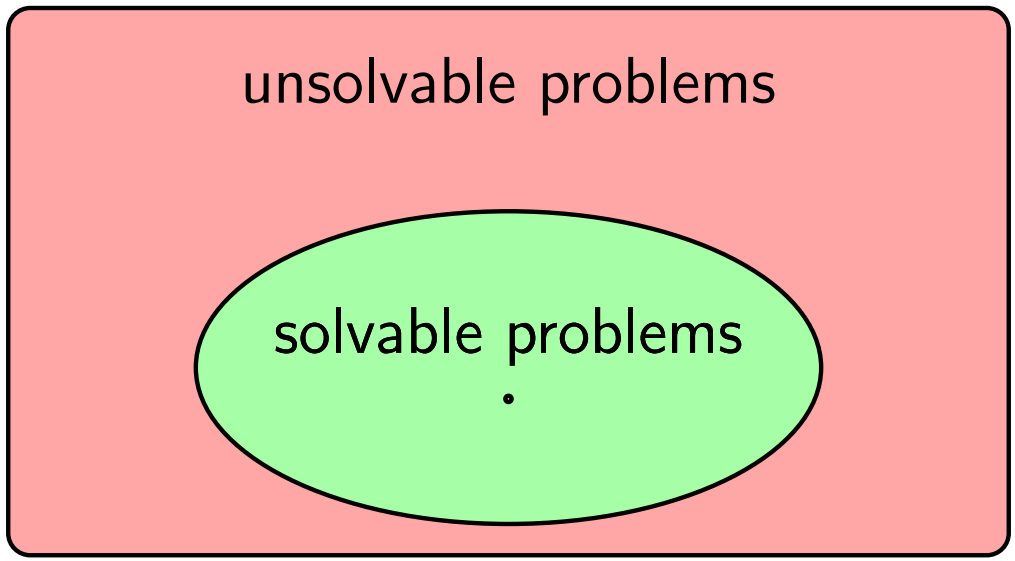
. . . and the punchline

$$\text{elements in } \mathbb{R} > \text{elements in } \mathbb{N}$$

QED

A shocking realization

Almost all problems are unsolvable!



We've shown, simply by counting, that the set of programs is isomorphic to $\mathbb{N}$ and the set of problems is isomorphic to $\mathbb{R}$. We know that there are more elements in $\mathbb{R}$ than in $\mathbb{N}$, in the sense that $\mathbb{N} \subset \mathbb{R}$ and there is no injective mapping from $\mathbb{N}$ to $\mathbb{R}$. Hence there simply aren't enough programs to solve all problems; **unsolvable problems must exist!**

But the result is even more shocking than that. We know that, in a measure-theoretic sense, almost all real numbers are not integers. In other words, the probability that a randomly chosen real number is an integer is 0. Hence, the chances that a randomly chosen problem is solvable are negligible; **almost all problems are unsolvable!**

Wait a minute...

- So are computers totally useless?
- Real-life problems have **structure**.
- We can only **describe** countably many problems!

So are computers totally useless? That seems to be the conclusion of our reasoning, since we've said that computers can solve almost no problems. Yet our experience with them says otherwise; what is going on here?

The reason for this result is that we counted as problems **arbitrary mappings** from $\mathbb{N}$ to $\mathbb{N}$. Real problems we care about are not arbitrary mappings, they have some **structure**. After all, we can only describe countably many problems! It's easy to see why; just observe that the description of a problem, be it in English or in a formal language, is a string, and there are countably many of them. Hence the problems that we can actually talk about, and possibly care about, are as many as numbers in $\mathbb{N}$, as many as programs! Perhaps then we can solve them all!

This is indeed reassuring, but we will still see that it is not the case. There are problems that we can easily describe and are of practical interest that are still unsolvable. The infamous **halting problem** is one of them.

The paradox proof

The halting problem

Given a program `P` and an input `x`, does `P` halt when run with input `x`?

A program that halts

```
1.  P(x):
2.      print x
3.  end
```

A program that doesn't

```
1.  P(x):
2.      while True:
3.          print x
4.  end
```

The paradox proof

Step 1: postulate

Suppose that `halt(P,x)` solves the halting problem.

$$\texttt{halt(P,x)} = \begin{cases} \texttt{yes} & \text{if } \texttt{P(x)} \text{ halts} \\ \texttt{no} & \text{if } \texttt{P(x)} \text{ loops} \end{cases}$$

The paradox proof

Step 2: construct the program of doom

```
1.  doom(P):
2.    if halt(P,P) == yes:
3.        loop forever
4.     else:
5.        return
6.  end
```

The paradox proof

Step 3: pull the trigger

What happens if we run `doom(doom)`?

```
1.  doom(doom):
2.    if halt(doom,doom) == yes:
3.        loop forever
4.     else:
5.        return
6.  end
```

`doom(doom)` halts when it loops, and it loops when it halts!

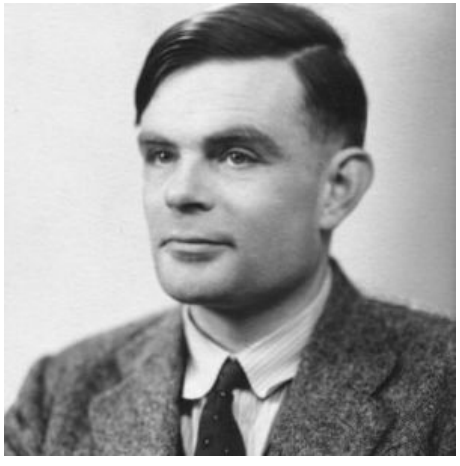
The paradox proof

. . . and the punchline

By contradiction `halt(P,x)` cannot exist!

QED

Alan Turing
1936



We've proved that the halting problem is unsolvable by **contradiction**. That is, we showed that the existence of a program that solves it leads to paradoxical results. This is a sketch of a famous proof by **Alan Turing** published in **1936**. Turing introduced the **Turing machine**, a theoretical computing device which formalizes the notion of a program (in our proof sketch we used the notion of a program informally). Then, he went on to show that no Turing machine can determine whether a given Turing machine halts on a given input. It is remarkable that this proof dates back to 1936; computers barely existed back then!

It is worth mentioning that the reason Turing was interested in computability is David Hilbert's **Entscheidungsproblem**, which asks for an algorithm to determine whether a statement of first order logic can be derived from a set of axioms. The undecidability of the halting problem implies the undecidability of the Entscheidungsproblem, which is exctly what Turing set out to prove. At the same time as Turing, in 1936, Alonzo Church (Turing's PhD supervisor!) published an equivalent proof of the undecidability of the Entscheidungsproblem using lambda calculus.

Other unsolvable problems

- **The verification problem**: Given a program and a specification, does the program follow the specification?
- **The Entscheidungsproblem**: Given a set of axioms and a statement of first order logic, does the statement follow from the axioms?
- **The number theory problem**: Given a statement of number theory, is it true?

Having shown one particular problem to be unsolvable, we can use **reduction** to show other problems to be unsolvable too.

The **verification problem** asks whether a given program follows a given specification. This would be a great way to check the correctness of our programs! However, we can show that if there is a program that solves it, then the halting problem becomes solvable too. Suppose you want to know if program `P` halts with input `x`. Then construct a program `Q` which first simulates `P(x)` and then returns 0. Notice that `Q` will return 0 if and only if `P(x)` halts. If you can solve the verification problem, then you can ask "does `Q` return 0" and immediately solve the halting problem, which is a contradiction.

The **Entscheidungsproblem** was posed by **David Hilbert** in **1928**, hoping (as it turned out, in vain) for a positive answer to it. We can show that a program which solves the Entscheidungsproblem leads to a program which solves the halting problem. If you want to determine whether `P(x)` halts, construct a statement in first-order logic with the meaning "there exists a number of steps after which `P(x)` halts" and use your solution to the Entscheidungsproblem to answer this, which leads to a contradiction.

Finally, a solution to the **number theory problem** would provide an automatic way to prove or disprove any statement of number theory! If a program solved it, then we would simply feed into it Fermat's last theorem, Goldbach's conjecture, the abc conjecture, or any other proposition of number theory we wanted to know whether it's true, wait for some time (maybe long, but still finitely long!) and get a definite answer. However, no such program exists, and this is a direct consequence of **Gödel's first incompleteness theorem** (published in 1931), which says that no automatic procedure can generate all true statements of number theory and only them. If such a program existed that would immediately provide such an automatic procedure, which would be a contradiction.

The above problems give us a flavour of the type of problems that computers cannot solve. They typically contain some element of **infinity** in their computational requirements. For instance, one way to "solve" the halting problem would be to run `P(x)` for long enough and see whether it halts. This might require running it forever, in case it doesn't halt! Similarly, "solving" the verification problem might require running a program for infinitely long or for infinitely many inputs to see if the specification is met. A program "solving" the Entscheidungsproblem could generate all statements that follow from the axioms and check whether the given statement is included in them. Since infinitely many statements can be generated from the axioms, this procedure might take infinitely long. Finally, in "proving" a number-theoretic proposition such as "there exist infinitely many primes", a program would have to go through all numbers in $\mathbb{N}$ which of course would take infinitely long.

Ultimately, the limits of computation are due to the fact that **we require programs to run in finite time**. If we allowed programs to run infinitely long, such problems would become solvable. But clearly, a program running infinitely long is a rather useless one.

Takeaways

- Is computation limited? Yes
- Do unsolvable problems include problems we care about? Yes
- Am I going to stumble across any unsolvable problems in my work? Probably not!