Computability

Are there problems that computers cannot solve?

- This question is older than computers!
- Came out of mathematicians' efforts to formalize the concept of proof in the early 1900s.
- They needed a notion of algorithm to desribe procedures for generating proofs from axioms.

What is an algorithm?

- Informally: it's a well-defined finite sequence of finite steps for solving a problem.

What is an algorithm?

- Informally: it's a well-defined finite sequence of finite steps for solving a problem.
- Alonzo Church (1935): it's a lambda-calculus expression.
- Alan Turing (1936): it's a Turing machine.

What is an algorithm?

- Informally: it's a well-defined finite sequence of finite steps for solving a problem.
- Alonzo Church (1935): it's a lambda-calculus expression.
- Alan Turing (1936): it's a Turing machine.
- Church and Turing soon realised they'd defined exactly the same thing in completely different ways.
- The Church–Turing Thesis states that lambda-calculus and Turing machines correspond to exactly what we mean by "algorithm".

Relation to computer programs

- General-purpose languages (Java, Python, C, etc.) can write compilers for each other.
- They can also evaluate lambda-calculus expressions and simulate Turing machines.

Relation to computer programs

- General-purpose languages (Java, Python, C, etc.) can write compilers for each other.
- They can also evaluate lambda-calculus expressions and simulate Turing machines.
- So computer programs also correspond to algorithms.
- Here, we're thinking of non-interactive programs that receive some input string, do some computation and produce an output string.

The halting problem

Input: Strings `prog` and `data`.
Task: Determine whether `prog` is a syntactically valid program that terminates when given input `data`.

The halting problem

Input: Strings `prog` and `data`.
Task: Determine whether `prog` is a syntactically valid program that terminates when given input `data`.

- Is there an algorithm to solve this?

The halting problem

Input: Strings `prog` and `data`.
Task: Determine whether `prog` is a syntactically valid program that terminates when given input `data`.

- Is there an algorithm to solve this?
- Is there an efficient algorithm?

Does it terminate? (1)

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j *= 2)
        s++;
```

Does it terminate? (1)

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j *= 2)
        s++;
```

- Terminates for $\mathtt{n} \leq 1$.
- Otherwise, the inner loop is infinite.

Does it terminate? (2)

```
int ack(int x, int y) {
    if (x == 0) return y+1;
    if (y == 0) return ack(x-1, 1);
    return ack(x-1, ack(x, y-1));
}
```

Does it terminate? (2)

```
int ack(int x, int y) {
    if (x == 0) return y+1;
    if (y == 0) return ack(x-1, 1);
    return ack(x-1, ack(x, y-1));
}
```

- This is Ackermann's function.
- It terminates for all $x, y \geq 0$, but this isn't obvious.

Does it terminate? (2)

```
int ack(int x, int y) {
    if (x == 0) return y+1;
    if (y == 0) return ack(x-1, 1);
    return ack(x-1, ack(x, y-1));
}
```

- This is Ackermann's function.
- It terminates for all $x, y \geq 0$, but this isn't obvious.
- It takes a *very* long time to terminate: `ack(4,2)` has nearly 20 000 decimal digits.
- Since the answer is computed eventually only by adding 1s to y, running time is at least proportional to the answer.

Does it terminate? (2)

```
int ack(int x, int y) {
    if (x == 0) return y+1;
    if (y == 0) return ack(x-1, 1);
    return ack(x-1, ack(x, y-1));
}
```

- This is Ackermann's function.
- It terminates for all $x, y \geq 0$, but this isn't obvious.
- It takes a *very* long time to terminate: `ack(4,2)` has nearly 20 000 decimal digits.
- Since the answer is computed eventually only by adding 1s to *y*, running time is at least proportional to the answer.
- The age of the universe in nanoseconds has 24 digits...

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.
- $\mathtt{x} = 2 \rightarrow 1$ terminates.

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.
- $\mathtt{x} = 2 \to 1$ terminates.
- $\mathtt{x} = 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.
- $\mathtt{x} = 2 \rightarrow 1$ terminates.
- $\mathtt{x} = 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.
- $\mathtt{x} = 4 \rightarrow 2 \rightarrow 1$.

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.
- $\mathtt{x} = 2 \to 1$ terminates.
- $\mathtt{x} = 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.
- $\mathtt{x} = 4 \to 2 \to 1$.
- $\mathtt{x} = 5$ terminates as above.

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.
- $\mathtt{x} = 2 \to 1$ terminates.
- $\mathtt{x} = 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.
- $\mathtt{x} = 4 \to 2 \to 1$.
- $\mathtt{x} = 5$ terminates as above.
- $\mathtt{x} = 6 \to 3$ terminates as above.

Does it terminate? (3)

```
void col(int x) {
    while (x > 1) {
        if (x%2 == 0) x = x/2;
        else x = 3*x + 1;
    }
}
```

- Obviously terminates if $\mathtt{x} \leq 1$.
- $\mathtt{x} = 2 \to 1$ terminates.
- $\mathtt{x} = 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.
- $\mathtt{x} = 4 \to 2 \to 1$.
- $\mathtt{x} = 5$ terminates as above.
- $\mathtt{x} = 6 \to 3$ terminates as above.
- $\mathtt{x} = 7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to \cdots ?$

Does it terminate? (3)

- Nobody knows if `col(x)` terminates for all `x`!
- Collatz conjectured it does (1937).
- "Mathematics may not be ready for such problems." — Paul Erdős.

Is there an algorithm for the halting problem? (1)

Turing proved that there is no algorithm for the halting problem in 1937.

Is there an algorithm for the halting problem? (1)

Turing proved that there is no algorithm for the halting problem in 1937.

Suppose we have a Java function that solves the halting problem:

```
boolean halts (String prog, String data) {
    // Clever stuff here
}
```

Is there an algorithm for the halting problem? (1)

Turing proved that there is no algorithm for the halting problem in 1937.

Suppose we have a Java function that solves the halting problem:

```
boolean halts (String prog, String data) {
    // Clever stuff here
}
```

We could include this in a program.

```
public static void main (String[] args) {
    if (halts (args[0], args[0]))
        while (true) {}
    else return;
}
```

Let `myProg` be the text of this entire program.

Is there an algorithm for the halting problem? (2)

```
public static void main (String[] args) {
    if (halts (args[0], args[0])) while (true) {}
    else return;
}
```

Now, run `myProg` with input `myProg`.

Is there an algorithm for the halting problem? (2)

```
public static void main (String[] args) {
    if (halts (args[0], args[0])) while (true) {}
    else return;
}
```

Now, run `myProg` with input `myProg`.

- If `halts(myProg, myProg)` returns `true`, then running `myProg` with input `myProg` goes into an infinite loop.
- If it returns `false`, `myProg` terminates.

Is there an algorithm for the halting problem? (2)

```
public static void main (String[] args) {
    if (halts (args[0], args[0])) while (true) {}
    else return;
}
```

Now, run `myProg` with input `myProg`.

- If `halts(myProg, myProg)` returns `true`, then running `myProg` with input `myProg` goes into an infinite loop.
- If it returns `false`, `myProg` terminates.

This is the exact opposite of what `halts(myProg,myProg)` is supposed to mean! This is a contradiction.

Undecidability of the halting problem

- We assumed only that we had an algorithm to correctly solve the halting problem.

Undecidability of the halting problem

- We assumed only that we had an algorithm to correctly solve the halting problem.
- We assumed nothing about how that algorithm works: just that it exists.
- We still found an input where this algorithm gives the wrong answer.

Undecidability of the halting problem

- We assumed only that we had an algorithm to correctly solve the halting problem.
- We assumed nothing about how that algorithm works: just that it exists.
- We still found an input where this algorithm gives the wrong answer.
- Conclusion: the algorithm cannot exist.
- The halting problem an example of an **undecidable** problem: a problem for which no algorithm is possible.

Consequences

- Your IDE/compiler can't warn you about all infinite loops.

Consequences

- Your IDE/compiler can't warn you about all infinite loops.
- It can't warn you about all uninitialized variables.
- ... or unreachable code
- ... or memory leaks, or buffer overruns
- ... or do perfect garbage collection, or tell you if two programs do the same thing.

Proving that problems are undecidable

- We proved the undecidability of the halting problem using "diagonalization".
- Proving undecidability of other problems that way is usually difficult.

Proving that problems are undecidable

- We proved the undecidability of the halting problem using "diagonalization".
- Proving undecidability of other problems that way is usually difficult.
- Instead, we show that, if there was an algorithm for problem X, we could adapt it to give an algorithm for the halting problem.
- This is called "reduction from the halting problem."
- (In this context, "reduction" means translation between problems.)

A reduction from the halting problem (1)

- We'll show that it is undecidable whether a program halts when run with no input.
- In principle, this problem could be easier than the halting problem, which requires us to handle any possible input.

A reduction from the halting problem (1)

- We'll show that it is undecidable whether a program halts when run with no input.
- In principle, this problem could be easier than the halting problem, which requires us to handle any possible input.
- Suppose we have a method `haltsWithoutInput(String prog)`.
- To solve the halting problem, we must give an algorithm that determines whether program `prog` halts when given the string `data` as input.

A reduction from the halting problem (2)

Here is the algorithm to determine if program `prog` halts on input `data`.

- Produce a new program `newProg` by renaming `main` to `oldMain` (or some other unusued name).

A reduction from the halting problem (2)

Here is the algorithm to determine if program `prog` halts on input `data`.

- Produce a new program `newProg` by renaming `main` to `oldMain` (or some other unused name).
- Add a new method

```
public static void main (String[] args) {
    String data = "[contents of the string data]";
    String[] newArgs = {data};
    oldMain (newArgs);
}
```

A reduction from the halting problem (2)

Here is the algorithm to determine if program `prog` halts on input `data`.

- Produce a new program `newProg` by renaming `main` to `oldMain` (or some other unused name).
- Add a new method

```
    public static void main (String[] args) {
        String data = "[contents of the string data]";
        String[] newArgs = {data};
        oldMain (newArgs);
    }
```

- Finally, call `haltsWithoutInput(newProg)`.

A reduction from the halting problem (2)

Here is the algorithm to determine if program `prog` halts on input `data`.

- Produce a new program `newProg` by renaming `main` to `oldMain` (or some other unused name).
- Add a new method

```
public static void main (String[] args) {
    String data = "[contents of the string data]";
    String[] newArgs = {data};
    oldMain (newArgs);
}
```

- Finally, call `haltsWithoutInput(newProg)`.
- `newProg` with no input does exactly the same thing as `prog` does with input `data`, so `haltsWithoutInput()` lets us decide the halting problem.
- This is impossible, so `haltsWithoutInput()` cannot exist.

Another reduction from the halting problem

- A similar reduction tells us that it's undecidable whether a program uses uninitialized variables.

Another reduction from the halting problem

- A similar reduction tells us that it's undecidable whether a program uses uninitialized variables.
- Modify the program text so every variable is initialized in its declaration.
- Modify `main` so it says

```
public static void main (String[] args) {
    int variableNameThatsNotUsedInOriginalProgram;
    /* Old text of main here */
    variableNameThatsNotUsedInOriginalProgram++;
}
```

Another reduction from the halting problem

- A similar reduction tells us that it's undecidable whether a program uses uninitialized variables.
- Modify the program text so every variable is initialized in its declaration.
- Modify `main` so it says

```
public static void main (String[] args) {
    int variableNameThatsNotUsedInOriginalProgram;
    /* Old text of main here */
    variableNameThatsNotUsedInOriginalProgram++;
}
```

- The new program uses an uninitialized variable if, and only if, the original program halts.

Rice's theorem

Rice's theorem gives another way to prove undecidability.

Rice's theorem

Rice's theorem gives another way to prove undecidability.

Two programs `p` and `q` are *behaviourally equivalent* if, for every input `x`, either

- `p` and `q` both halt and give the same output, or
- `p` and `q` both go into infinite loops.

Rice's theorem

Rice's theorem gives another way to prove undecidability.

Two programs `p` and `q` are *behaviourally equivalent* if, for every input `x`, either

- `p` and `q` both halt and give the same output, or
- `p` and `q` both go into infinite loops.

Theorem (Rice's theorem)

Let S be a set of programs that is not empty and is not the set of all programs. The question "Is `p` behaviourally equivalent to a program in S?" is undecidable.

Rice's theorem example

We'll show it's undecidable whether or not a program prints the word "computability" for some input.

Rice's theorem example

We'll show it's undecidable whether or not a program prints the word "computability" for some input.

- Let S be the set of all programs that *do* print "computability" for at least one input.

Rice's theorem example

We'll show it's undecidable whether or not a program prints the word "computability" for some input.

- Let S be the set of all programs that *do* print "computability" for at least one input.
- $S \neq \emptyset$ and S is not the set of all programs.

Rice's theorem example

We'll show it's undecidable whether or not a program prints the word "computability" for some input.

- Let S be the set of all programs that *do* print "computability" for at least one input.
- $S \neq \emptyset$ and S is not the set of all programs.
- A program is functionally equivalent to one in S if, and only if, it prints "computability".

Rice's theorem example

We'll show it's undecidable whether or not a program prints the word "computability" for some input.

- Let S be the set of all programs that *do* print "computability" for at least one input.
- $S \neq \emptyset$ and S is not the set of all programs.
- A program is functionally equivalent to one in S if, and only if, it prints "computability".
- So the property of printing "computability" is undecidable by Rice's theorem.

We can still reason about programs (1)

- The undecidability of the halting problem doesn't stop us writing algorithms that do *some* things with programs.

We can still reason about programs (1)

- The undecidability of the halting problem doesn't stop us writing algorithms that do *some* things with programs.
- It's still possible to, e.g., detect *some* infinite loops, which is still useful.
- You can write an algorithm that detects some infinite loops and some code that always terminates, but it will have to say "I don't know" for infinitely many inputs.

We can still reason about programs (1)

- The undecidability of the halting problem doesn't stop us writing algorithms that do *some* things with programs.
- It's still possible to, e.g., detect *some* infinite loops, which is still useful.
- You can write an algorithm that detects some infinite loops and some code that always terminates, but it will have to say "I don't know" for infinitely many inputs.
- Java dodges undecidability by requiring variables to be initialized before any code that potentially uses them – even if the code is unreachable.

We can still reason about programs (2)

- We can still reason about the syntax of programs: the halting problem applies to runtime behaviour.

We can still reason about programs (2)

- We can still reason about the syntax of programs: the halting problem applies to runtime behaviour.
- We can also decide properties of programs in restricted environments, e.g., programs that are only allowed to run for a certain number of steps (just simulate for that many steps).
- In principle, we can decide termination of programs that can only use a fixed-size memory but the running time of the algorithm is impossibly long.

Computability: summary

- Computability is the study of whether *any* algorithm can exist for a problem.
- The most significant problem for which no algorithm exists is the halting problem: determining whether a given program terminates with a given input.

Computability: summary

- Computability is the study of whether *any* algorithm can exist for a problem.
- The most significant problem for which no algorithm exists is the halting problem: determining whether a given program terminates with a given input.
- As a consequence, many practical problems in program analysis are also undecidable.
- Even when a problem is undecidable, we may still be able to solve some cases – but there will always be infinitely many that we cannot solve.