AAA – Automata, Computability and Complexity

Ian Sanders

Second Semester, 2024

Overarching question

What are the fundamental capabilities and limitations of computers? (Sipser, 2013, p.1)

- This question dates back to the 1930s when mathematical logicians started to explore the meaning of *computation*.
- Recall this was before the first general purpose computer!
- ENIAC (Electronic Numerical Integrator and Computer) was the first programmable, electronic, general-purpose digital computer and was completed in 1945.
- The study of complexity and computability based on an understanding of automata theory allows us to tackle this question.

Complexity

- We want to use computers to solve problems.
- You will have seen some easy problems – *tractable*.
- And you will have seen some hard problems – *intractable*.
- Central question in *complexity theory*:

  What makes some problems computationally hard and other easy?

- One result of considering this question is a scheme for classifying problems so that we can say something about them.
- More on this later.

Computability theory

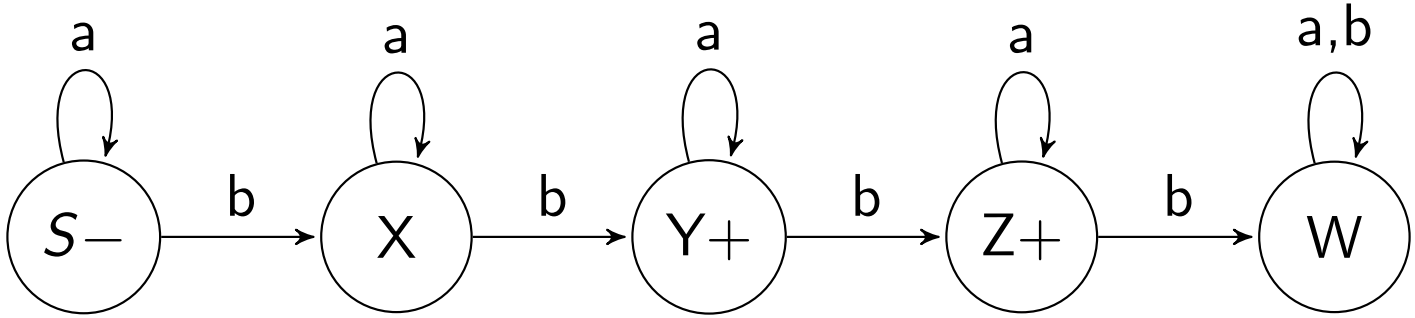
- During the first half of the twentieth century, Kurt Gödel, Alan Turing and Alonzo Church discovered that some basic problems cannot be solved by computers.
- For example, determining whether a mathematical statement is true or false.
- Most of this lecture expands on this – after relooking at automata.

Automata Theory

- Deals with the definitions and properties of mathematical models of computation.
- You have seen
  - Finite Automata (FA) (or Finite State Machines (FSM))
  - Pushdown Automata (PDA)
  - Turing Machines (TM)
- Machines and grammars/languages
  1. FA – regular languages
  2. PDA – context free languages
  3. TM – essentially all

Finite Automata

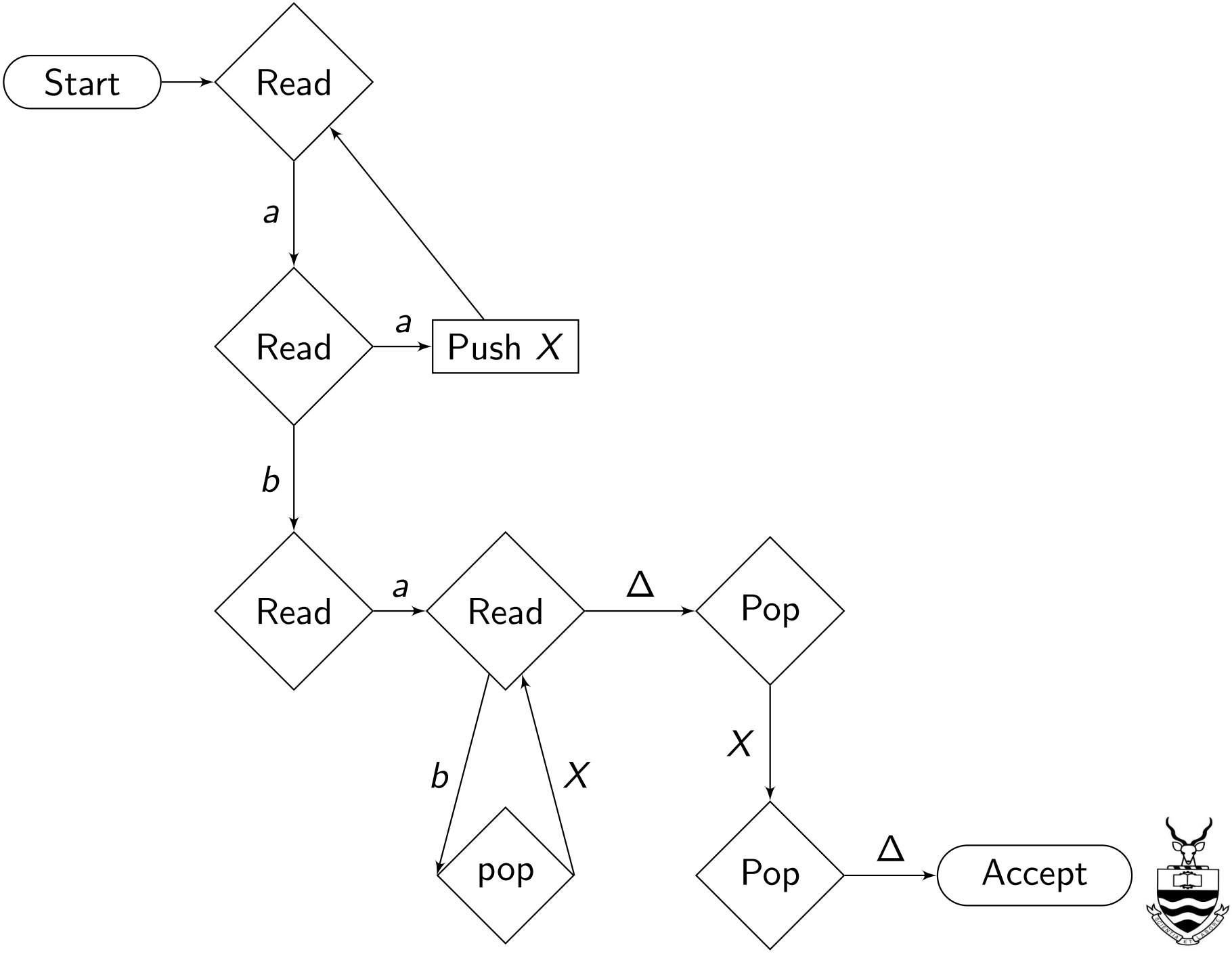
A finite automaton is a machine that *accepts* words in a given language.
For example, suppose the language is all words that have exactly two or three *b*'s.
abb abab, *bbb*, *bbbaaaaaaa* and *aaaaaabaaaabaaaab* are examples of words that should be accepted.



FAs are used in computer science/programming for regular expressions, text processing, compilers, etc.
A FA has *no memory*!

Pushdown Automata

PDAs are more powerful machines.
They have some *memory*!
This is because they have a *stack*.
They can deal with more complicated languages than FAs can and can accept any language that would be accepted by an FA.
For example:
The language that has as many a's as it has b's.
Or
The language $L = \{(aa)^{n+1}(aba)(b)^n \mid n \geq 0\}$ over the alphabet $\Sigma = \{a, b\}$.


Start
Read
a
Read
a
Push X
b
Read
a
Read
Δ
Pop
b
X
pop
X
Pop
Δ
Accept

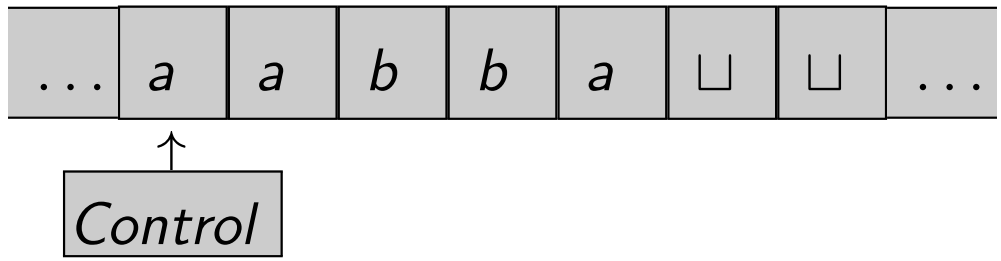
Turing Machines

Proposed by Alan Turing in 1936.
TMs are more powerful than FAs and PDAs.
They can accept/reject words from various languages.
Anything a PDA can do, a TM can as well.
And they can *compute*!
It is an accurate model of a general purpose computer.

A Turing machine is made up of

1. an external storage – unbounded (infinite) tape divided into squares
2. tape head – to read from and write to the tape – at any instant points to one square
3. control unit – to move the tape head along the tape

Each square on the tape is either blank or contains a *symbol* from a finite *alphabet*.


. . .
a
a
b
b
a
␣
␣
. . .
Control

The program for a Turing Machine (TM) consists of a finite set of quintuples.
The control unit acts by selecting and executing the appropriate quintuple from the program.
A quintuple consists of

1. a current state
2. an input symbol
3. an output symbol
4. a next state
5. a head movement

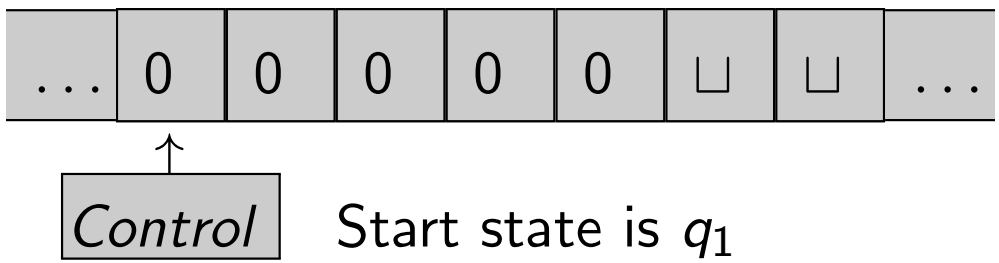
More formally a Turing Machine is specified by

1. A finite external tape alphabet $\Sigma = \{\sqcup\} \bigcup \{s_1, \ldots, s_m\}$ ($\sqcup$ is the blank symbol)
2. A finite set of internal states $Q = \{h\} \bigcup \{q_1, q_2, \ldots, q_k\}$ (h is the halt state).
3. A program consisting of a finite set of quintuples of the form
   $Q_i S_j S_k Q_l M$
   $Q_i, Q_l \in Q$
   $S_j, S_k \in \Sigma$
   $M \in \{L, R\}$
   Every quintuple must be defined *uniquely* by its first two symbols.
4. A tape with a finite number of non-blank squares, each such square containing a symbol from $\{s_1, \ldots, s_m\}$
5. an initial state which is in $\{q_1, q_2, \ldots, q_k\}$
6. the initial tape head position (the first square on the tape to be considered)

A simple TM

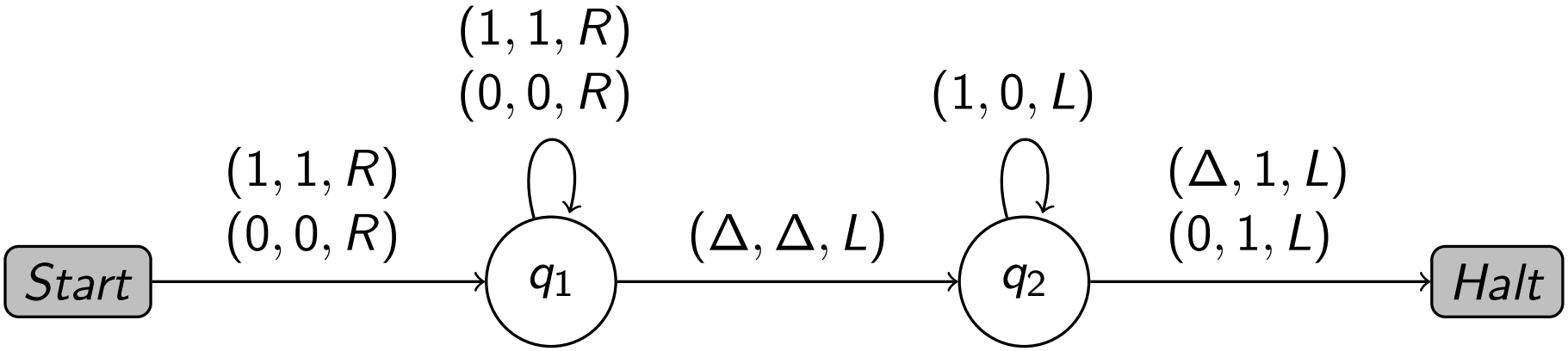
TM_1
$\Sigma = \{\sqcup, 0, 1\}$
$Q = \{h, q_1\}$
$Program := \{q_1 01 q_1 R, q_1 \sqcup \sqcup h L\}$
Start state is q_1
Start position is leftmost 0

A simple TM



$Program := \{q_1 01 q_1 R, q_1 \sqcup \sqcup h L\}$

Consider the Turing Machine (TM) T (over the input alphabet $\Sigma = \{0, 1\}$) given below.



What does it do?

It is an incrementer!

It calculates x++

More on computability

Questions about the nature of computation first arose in the context of pure mathematics

Mathematical rigour and formal notation are relatively recent developments – late nineteenth century

German mathematician Gottlob Frege was instrumental in developing a precise system of notation to formalise mathematical proofs

His work lead to the conclusion that the mathematical system contained a contradiction

Mathematicians believed any theorem could be proved – this seemed not to be the case

Asked questions "What is a proof?" and "What is a mathematical system?"

David Hilbert – each proof should be able to be written in an explicit and unambiguous notation and should be checkable in a finite series of elementary mechanical steps – he wanted all proofs to be checkable by an algorithm

In 1931 Kurt Gödel put an end to Hilbert's hopes by proving the *incompleteness theorem*

> *For every countable set of axioms and inference rules there are true formulas which cannot be proved.*

Basically this means that any complete study of our system of mathematics lies beyond the capabilities of algorithmic activities. In other words *there are problems for which no computations exist*.

Also independent work on models of computation and calculability:

- finding an appropriate formalism
- trying to understand what it was

Done by....

- Gödel and Kleene – partial recursive functions
- Herbrand – primitive recursive functions
- Church and Kleene : λ-calculus which became the inspiration for the programming language Lisp
- Post: production systems based on deductive mechanisms
- Alan Turing – mechanistic model of problem solving – the *Turing Machine*

All of these actually define the same class of functions – whatever one model can compute; all the others can too.
This justifies the claim that they are *universal models of computation*

Turing Machines and Computability

The execution of a set of instructions by an *agent* is called a *computation*.
Specifying the computation for complex tasks can become very complicated and leads to questions about what problems computations can be found for.
That is, which tasks are *computable*.
To consider what is computable (algorithmic) we require a formalism that is simple enough for logical analysis and at the same time is sufficiently general to describe all processes that could be reasonably called algorithms.
One such formalism or model of computation is the *Turing machine*. [Alan Turing (1921-1954)]

Turing's goal was to provide a model by which the limits of "computational processes" could be studied.
Model – a simplified (often mathematical) description of a system.
Turing machine – a mathematical description of the process of doing computation.
The claim is that
Any problem for which there exists an algorithmic solution can be solved by a Turing machine.
or, stated differently,
Every effectively calculable function is computable by some Turing machine.
This claim is called the *Church-Turing thesis*

A procedure is effective if

- it is set out in a finite number of instructions
- it will always produce the desired result in a finite number of steps
- it can be carried out by a human being
- it demands no insight or ingenuity.

PCs, Macs, Crays, parallel computers, etc. are all much more powerful and generally much easier and more convenient to program but a problem can be solved by these *if and only if* it can be solved by a program on a Turing machine.
The simplicity of the Turing machine makes it a good model to use to explore computability.

The Universal Turing Machine

To show a function is computable, we had to construct a TM to compute that function.
Turing designed a *Universal Turing Machine* (UTM) which is capable of imitating the operation of *any* other TM.
The idea of a UTM will help us to show that some well-defined functions are not computable.

UTM

1. A uniform method of describing any TM in a finite symbol set.
2. A set of TM programs that perform the basic TM functions.

The UTM takes as input

- an encoding of a TM, M
- x which is some input for M

There can be many different encodings of M.

So what makes a UTM?

1. an alphabet
2. a set of states
3. an infinite tape divided into squares
4. tape head
5. control unit
6. a program

How does a UTM simulate a TM?

Let U be the UTM and A be the TM to be simulated.
U must be able to

1. take the A's input as part of its input
2. take the A's states as part of its input
3. take the A's program as part of its input
4. read the current input symbol of A
5. "remember" current state of A
6. write a new symbol on A
7. change state of A
8. "remember" direction of A

How does it do this?

Points **??** to **??** rely on a suitable encoding – writing A in the alphabet of U.
Points **??** to **??** are functions that U's program must be able to execute.
With this can simulate A on U

Why is this important?

From *Church-Turing thesis – Every effectively calculable function is computable by some Turing machine.*
Next we show that there are some functions that cannot be computed.

Noncomputability or Undecidability

We have seen

- tractable problems
- intractable problems

Some problems cannot be solved using any machine; these are called *noncomputable* or *undecidable*.
Before proving rigorously that there are some problems which are noncomputable, we first look at some examples and then some paradoxes . . .

The tiling problem

Given:

- a finite set T of card descriptions
- the restriction that only card edges of the same colour may be placed adjacent

Can any finite area, of any size, be covered using only cards from T?
An algorithm for the tiling problem should answer *yes* if T does admit such a tiling and *no* if it doesn't.
A possible algorithm is:

1. if the types in T can tile any area, output "yes" and stop
2. otherwise output "no" and stop

No algorithm exists to do **??**
(We cannot in general answer the question if some set of tiles can be used to cover some area – we may be able to answer the question in specific instances but not for *all* instances.)
Therefore the problem is undecideable.

Does a given program halt?

```
Algorithm A1:
  While x != 1
    x =  x - 2
  stop
```

Legal inputs $I = < 1, 2, 3, \ldots >$
Deciding whether an arbitrary legal input will cause A1 to terminate is trivial – we have an algorithm to do so.

```
Algorithm A2:
   While x != 1
      if x even
         x =  x / 2
      otherwise
         x = 3 * x + 1
   stop
```

Legal inputs $I = < 1, 2, 3, \ldots >$
Given an arbitrary input can we tell whether A2 will terminate?
Haven't been able to prove this always terminates.

The paradox of the Barber

In some town there is only one barber. The barber is male. All men in the town shave or are shaved. The barber shaves only the men who do not shave themselves.
Who shaves the barber?
Either
- he shaves himself
- he doesn't shave himself
Each possibility leads to an impossibility.
This implies no such town exists. . . .

Russell's Paradox

Let the set of all sets that are not members of themselves be S.
If S is a member of this set S then by definition it must not be a member of itself.
Similary, if S is not a member of itself, then by definition it must be a member of S.
S is a member of itself *iff* it is not a member of itself
This is a contradiction.

Proving undecideability

For decidability need

- a set of legal inputs
- a proposed solution which applies to all inputs in that set

To show how to prove a problem is undecideable we begin with the Halting Problem.

Proving the Halting problem is undecideable

Imagine a UTM, H, which given an arbitrary TM, M, will decide whether or not M halts when given T as its input tape.
Let

$$H(M,T) = \begin{cases} 0 & \text{if } M(T) \text{ does not halt} \\ 1 & \text{if } M(T) \text{ does halt} \end{cases}$$

If H exists we could encode M and T on its tape as per UTMs.
H would then tell us if M terminates on T
That is H would solve the Halting Problem – given an arbitrary Turing Machine operating on arbitrary input can we determine whether or not the Turing Machine will halt.
Actually H cannot be constructed.
So we cannot solve the Halting Problem.
We will now look at this a bit more formally.

Theorem

There can be no UTM that solves the Halting problem.

Proof

Assume $H(M, T)$ exists (there is a TM to decide halting of M on T, that is $M(T)$)

Let

$$H_s(M) = \begin{cases} 0 & \text{if } M(M) \text{ does not halt} \\ 1 & \text{if } M(M) \text{ does halt} \end{cases}$$

$H_s(M) = H(M, M)$

H_s is a simple construction given H

Now let

$$P(M) = \begin{cases} \text{if } H_s(M) = 1 \text{ then } \mathit{loop} \\ \text{if } H_s(M) = 0 \text{ then halt with } 1 \end{cases}$$

P is a simple composition given H_s

What is $H_s(P)$?
We have

$$H_s(P) = \begin{cases} 0 & \text{if } P(P) \text{ does not halt} \\ 1 & \text{if } P(P) \text{ does halt} \end{cases}$$

and

$$P(P) = \begin{cases} \text{if } H_s(P) = 1 \text{ then } \textit{loop} \\ \text{if } H_s(P) = 0 \text{ then halt with 1} \end{cases}$$

Case 1: $H_s(P) = 1$ implies $H(P, P) = 1$ which implies $P(P)$ halts.
However, if $H_s(P) = 1$ then $P(P)$ loops.
Therefore *contradiction*.
Case 2: $H_s(P) = 0$ implies $H(P, P) = 0$ which implies $P(P)$ does not halt.
However, if $H_s(P) = 0$ then $P(P)$ halts with output 1.
Therefore *contradiction*.

We assumed that H existed (or could be constructed).
This leads to a contradiction.
This tells us that our assumption cannot be true.
That is H cannot be constructed.
In other words, since P and H_s are computable based on the assumption that H can be constructed, then H cannot be constructed. H is not computable.

Now that we have one noncomputable function – The Halting Problem – we can show that there are other noncomputable functions as well by showing that they are "the same" as the Halting Problem.

Reducibility is the fundamental method for proving that other functions are noncomputable.

How?

Show that if P is undecidable and P could be reduced to Q then P's yes/no answer is Q's answer to the transformed input.

If P is undecidable then Q must be too.

Otherwise if Q was not undecidable then could reduce P to Q and use the algorithm which decides Q to essentially decide P.

This would be a contradiction.

Some consequences:

There is no general way of determining whether

- a program halts on a given input
- a program halts on all of its inputs
- a program enters a certain state
- two programs are semantically equivalent

Summary

- The Turing Machine is a general mathematical model of a computer.
- We can solve some problems using a computer *but* there are problems we cannot solve.
- Of the problems we can solve, some are easy and some are hard! More about this next.