

# Algorithms and Complexity, weeks 1-5 key points

Sam Barrett

March 9, 2021

# Turing machine basics I

- ▶ Turing machines are **precise models of computation**
- ▶ First tape of a Turing machine is **always** a read-only **input tape**.
- ▶ The  $k^{th}$  tape is always the output tape
- ▶ The output tape is also considered a work tape
- ▶ The alphabet of a TM is denoted as  $\Gamma$ , it is **finite**.
- ▶ Each tape must always start with the *left-of-tape* symbol,  $\triangleright$
- ▶  $\{0,1\}^*$  is the set of bitstrings, with  $\varepsilon$  denoting the empty string.

# Turing machine basics II

In a single stage of computation a TM may:

- ▶ reads the character at each tape head
- ▶ writes a character at each work tape head
- ▶ may move each tape head to the left or to the right. **note:**  
**our tapes are not recursive, if a head on the leftmost cell moves left, it stays put**

## what does it mean to say that $M$ computes $f$ ?

It means that for every bitstring  $x \in \{0, 1\}^*$ , if we start in state  $q_{\text{start}}$  with the initial configuration showing  $x$  (meaning  $x$  appears on the input tape and the work tapes are blank), when we run  $M$ , we eventually reach  $q_{\text{halt}}$  with the output tape showing  $\triangleright$  on the leftmost cell and then the bitstring  $f(x)$  followed by all blanks.

# Computable functions

## Basics

### Definition

(Computable functions) We say a function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  is **computable** if there exists some Turing machine that computes it and **non-computable** if there isn't.

No variation of Turing machine affects this fact, giving us:

### Thesis

*(Church's Thesis) any algorithm that computes a function from bitstrings to bitstrings can be converted into a Turing machine that computes the same function*

# Boolean functions, languages and decidability I

## Definition

A *language* can be defined as any set of words

## Definition

A **boolean function** is a function of the form:

$f : \{0, 1\}^* \rightarrow \{0, 1\}$ . Noting that the output is a single bit rather than a bitstring.

There is a one-to-one correspondence between languages and boolean functions.

- ▶ For a given boolean function  $f$  the corresponding language is the set of bitstrings  $x$  s.t.  $f(x) = 1$
- ▶ For a language  $L$ , the corresponding boolean function sends  $x$  to 1 if  $x \in L$  and to 0 otherwise.

# Boolean functions, languages and decidability II

This allows us to treat boolean functions, languages and decision problems as essentially the same thing.

A decision problem is said to be **decidable** when the corresponding boolean function is **computable**. I.e. given a language  $L$ , for  $L$  to be decidable there must exist some Turing machine that will start with a bitstring  $x$  and will run continuously until it halts and upon halting there will be a 1 on the output tape if  $x \in L$  or 0 if it is not in the language.

# Data Representation

- ▶ We can encode many real-life data types as bitstrings, but not all. (e.g. Real numbers cannot)
- ▶ We can encode multiple inputs as a single bitstring.



# Code as Data

- ▶ We can not only encode many data types as bitstrings, we can encode program code or even other Turing machines as bitstring inputs to a TM as our TMs are essentially 6-tuples (see full notes for formal definition).
- ▶ We can therefore say that for **any** bitstring  $\alpha$  we can construct a corresponding TM:  $M_\alpha$

# The Universal Turing Machine, $\mathcal{U}$

$\mathcal{U}$  is a Turing machine interpreter written as a Turing machine. It takes 2 inputs (encoded as a single input):  $\alpha$  and  $x$ , where  $\alpha$  is the bitstring describing the machine to be interpreted and  $x$  is the bistring input.

We define  $\mathcal{U}$  as having 4 tapes and the basic alphabet of  $\{\triangleright, \square, 0, 1\}$ . Intuitively,  $\mathcal{U}$  works by simulating the  $M_\alpha$  by *providing* it with the 3 non-input tapes to  $M_\alpha$  as its input, work and output tape respectively.

# Diagonalisation & the Halting problem I

## Problem

*(The halting problem) the set of pairs  $\langle \alpha, x \rangle$  (encoded as a single bitstring) such that the machine  $M_\alpha$  executed on input  $x$  halts.*

# Diagonalisation & the Halting problem II

Turing's proof is as follows:

**Proof.**

Suppose that  $N$  is a machine that solves the halting problem.

We can convert it into a machine  $N'$  that, given  $x$ , runs forever if  $\langle x, x \rangle \in \text{HALT}$  (i.e. the machine  $M_x$  executed on  $x$  halts), and halts otherwise.

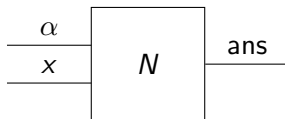
We know  $N' = M_\alpha$  for some  $\alpha$ , i.e. there exists some bitstring  $\alpha$  that represents our new machine, as we know every machine can be represented as a bitstring.

Running  $N'$  on  $\alpha$  halts if it runs forever and runs forever if it halts. We have derived a contradiction.



# Diagonalisation & the Halting problem III

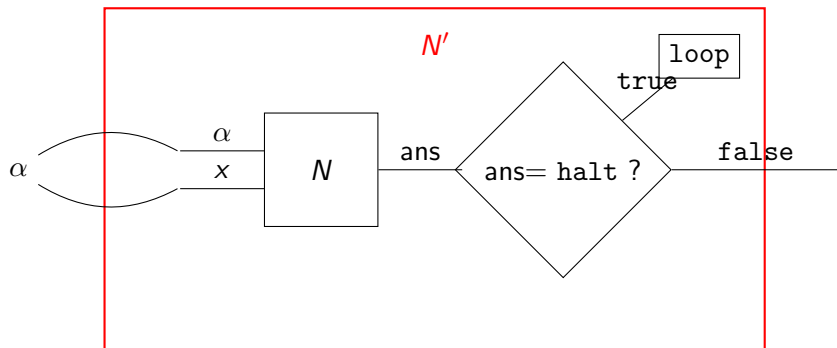
More intuitively, we can consider the described machine  $N$  as:



Where  $\text{ans}$  is whether  $N$  halts.

We can then construct the wrapper  $N'$  as:

# Diagonalisation & the Halting problem IV



**Where the outermost  $\alpha$  is the bitstring of  $N'$**

You can clearly see that if  $N$  were to halt then  $N'$  would hang. We can therefore derive a contradiction as if we pass  $N'$  into  $N'$  it would have to halt if it hangs and vice versa!

# Upper bound notation

- ▶ If we have two function  $f : \mathbb{N} \rightarrow \mathbb{N}$  and  $g : \mathbb{N} \rightarrow \mathbb{N}$
- ▶ We say that  $f(n)$  is  $O(g(n))$  if  $f$  is **no bigger** than  $g$  up to a constant factor, i.e. given a constant  $c$  and a value  $n_0$  s.t.  $\forall n, n \geq n_0$  we have:

$$f(n) \leq c \cdot g(n)$$

- ▶ We say that  $f(n)$  is  $o(g(n))$  if  $f$  is **not as big as**  $g$ , even up to any constant factor. Or, if, **for any**  $\varepsilon > 0$ , there is a  $n_0$  s.t.  $\forall n, n \geq n_0$  we have:

$$f(n) \leq \varepsilon \cdot g(n)$$

Whenever  $f(n)$  is  $o(g(n))$  is it also  $O(g(n))$ , this is the case if you take  $c$  to be 1

# Lower bound notation

- ▶ If we have two function  $f : \mathbb{N} \rightarrow \mathbb{N}$  and  $g : \mathbb{N} \rightarrow \mathbb{N}$
- ▶ We say that  $f(n)$  is  $\Omega(g(n))$  when  $g(n)$  is  $O(f(n))$
- ▶ We say that  $f(n)$  is  $\omega(g(n))$  when  $g(n)$  is  $o(f(n))$
- ▶ We say that  $f(n)$  is  $\Theta(g(n))$  when it is **both**  $O(g(n))$  and  $\Omega(g(n))$

This informally means  *$f(n)$  and  $g(n)$  are the same, up to a constant factor*



# Time complexity

## Definition

Worst case running time:

The running time of a machine  $M$  is the time taken from the input state, where  $x$  sits on the input tape and the other tapes are blank, to reach the halt state ( $q_{\text{halt}}$ ).

For any number  $n$ , we define  $WT_M(n)$  to be the **worst case** running time for an input of length  $n$ .

## Definition

DTIME classes

DTIME is a set of complexity classes.

A complexity class can be thought of as a set of decision problems (or languages or boolean functions...)

$\text{DTIME}(n^2)$  is an example of a complexity class. A function  $f : \{0, 1\}^* \rightarrow \{0, 1\}$  is in  $\text{DTIME}(n^2)$  when there is some machine that decides it and has worst case running time in  $O(n^2)$

# Polynomial Time

We can define the complexity (super) class of **polynomial time decision problems** as:

$$P \stackrel{\text{def}}{=} \bigcup_{k \geq 1} \text{DTIME}(n^k)$$

From this definition, you can see that **any** decision problem in  $\{\text{DTIME}(n^k)\}_{k=0}^{\infty}$  is also in  $P$