Algorithms & Complexity: Lecture 1

Sam Barrett

February 10, 2021

1 Defining the Turing machine model

In order to talk about the time taken or the space used by an algorithm, we require a precise **model of computation**. There are many proposed models, we will focus on the Turing machine as defined by Arora and Barak in their book.

1.1 Arora-Barak Turing machines

1.1.1 Tapes

A Turing machine is defined as having k tapes where $k \geq 2$

- The first tape is the *input tape* and is **read-only**
- The $2..k-1$ tapes are *work* tapes and are **read-write**
- The k^{th} tape is the *output* tape.

Each tape has a leftmost cell and *potentially* infinitely many cells to the right of it. Potentially infinite meaning that at any given time, there are a finite number of cells but we can infinitely extend the tape over time.

Each tape has a **head** that sits on a cell and can move left and right.

1.1.2 Alphabet

A Turing machine also has an alphabet, denoted Γ. This is a **finite** set and it's elements are called *symbols*. There are 4 primary symbols: $\triangleright, \square, 0, 1$.

Here:

- $\{0,1\}^*$ is the set of bitstrings, the empty string is denoted with ε.
- $\triangleright$ is the left-of-tape symbol and $\square$ is the blank symbol

At any point in time, each cell of each tape contains a symbol. All but a finite number will be blank ($\square$)

1.1.3 Initial configuration

The input tape has $\triangleright$ on the leftmost cell, then a bitstring (the **input**) and the rest of the tape is blank. The work tapes (including the output tape) have $\triangleright$ on the leftmost cell and the rest are blank. Each tape starts with it's head on it's the leftmost cell.

1.1.4 Computation step

In a single step of computation the machine:

- 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**

1.1.5 Formal definition

A **Turing machine** is defined as a (6) tuple, $M = (k, \Gamma, Q, q_{\texttt{start}}, q_{\texttt{halt}}, \delta)$ consists of the following data:

- the number of tapes, k, $k \geq 2$
- the alphabet $\{0, 1, \triangleright, \square\} \subseteq \Gamma$
- a finite set of Q states, including the start state $q_{\texttt{start}}$ and the halt state $q_{\texttt{halt}}$
- a transition function, $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, R, S\}^k$ Where:
  - the initial Q is the state at the start of transition
  - Γ^k is the set of symbols read
  - the final Q is the state at the end of transition
  - Γ^{k-1} is the set of symbols written
  - $\{L, R, S\}^k$ is the set of movement instructions where:
    - L means *move left*
    - R means *move right*
    - S means *stay*

  Note: we read k symbols but only write $k-1$ symbols as we do not write on the input tape, we also have k movement instructions as we are able to move on all k of the tapes.

1.1.6 Example transition

Say we have $k =$, and $\Gamma = \{\triangleright, \square, 0, 1\}$ and $Q = \{4, 5, 6, 7, 8\}$ with $q_{\texttt{start}} = 4$ and $q_{\texttt{halt}} = 8$. We are currently in state 7 and the three tapes respectively say 1 (input), 1 (work) and $\square$ (output).

Say that $\delta(7, \langle 1, 1\square\rangle) = (5, \langle 0, \square\rangle, \langle L, L, S\rangle)$ then we:

- transition to state 5
- overwrite the 1 on the work tape with 0
- overwrite the $\square$ on the output tape with $\square$ (no change)
- move left on the input tape (if possible)
- move left on the work tape (if possible)
- stay put on the output tape

We do not transition from the halt state (regardless of δ)

2 Computing with Turing machines

2.1 Computing a function

Given a function $f : \{0,1\}^* \to \{0,1\}^*$ a Turing machine $M = (k, \Gamma, Q, q_{\texttt{start}}, q_{\texttt{halt}}, \delta)$,

1. 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 $q_{\texttt{start}}$ with the initial configuration showing x (meaning x appears on the input tape and and the work tapes are blank), when we run M, we eventually reach $q_{\texttt{halt}}$ with the output tape showing $\triangleright$ on the leftmost cell and then the bitstring $f(x)$ followed by all blanks.

Our initial configuration can be shown as:

Input tape:

$\triangleright$	1	0	1	1	

Work tapes:

$\square$	$\square$	$\square$	$\square$	$\square$	

Output tape: (also a work tape)

$\square$	$\square$	$\square$	$\square$	$\square$	

Given that $f(x) = 0110111$, our required output tape will then be as follows:

$\triangleright$	0	1	1	0	1	1	1	1	

If the machine, M does *this* for every bitstring x then we say it **computes** f. In the Arora-Barak definition, it does not matter what is on the work tape at the end of execution or the location of the work heads.

2.2 Computable functions

We say a function $f : \{0,1\}^* \to \{0,1\}^*$ or a function f from bitstrings to bitstrings is **computable** if there exists some Turing machine that computes it and **non-computable** if there isn't.

In the second case where there exists no Turing machine that computes a function, is there some other kind of machine that *does* compute it?

2.2.1 Church's thesis

We have only looked at one definition of Turing machines, there are many different variations that have been studied. 1 tape vs ∞ tapes, large alphabets, tapes infinite in both directions, 2D tapes, etc.

None of these variations affect our definition of computability. The same definition holds for all models that have been investigated, leading to Church's thesis which (informally) states:

> **Thesis 1** *"any algorithm that computes a function $\{0,1\}^* \to \{0,1\}^*$ can be converted into a Turing machine that computes the same function"*

2.3 Boolean functions and language

A language can be defined as any set of *words*, for example *all the words with an even occurrence of 1* is a language.

A **boolean function** is a function of the form: $f : \{0,1\}^* \to \{0,1\}$. Noting that the output is a single bit rather than a bitstring.

A important point about languages and boolean function is that they correspond. There is a one-to-one correspondence in fact 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.

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.

2.3.1 Example: palindromes

A **palindrome** is a bitstring that *reads* the same forwards as backwards.

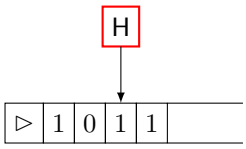
We can define our decision procedure for `PAL`, the set of all palindromes as:

1. Copy the input to the work tape
2. Move the input head to the start of the input
3. move the input head to the right while moving the output head to the left. If at any moment, the machine observes two different values, it writes 0 to the output tape and halts
4. Write 1 to the output tape and halt

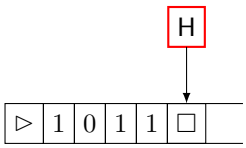
We can represent this as a Turing machine with 3 tapes and 5 states in the following example:

Step 1:

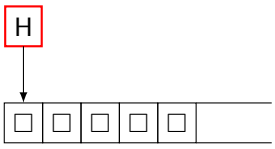
Input tape:



Work tape:

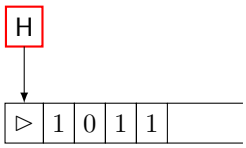


Output tape:

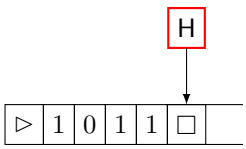


Step 2:

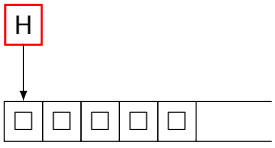
Input tape:



Work tape:

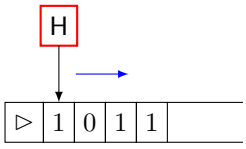


Output tape:

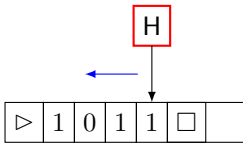


Step 3:

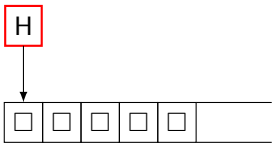
Input tape:



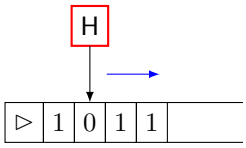
Work tape:



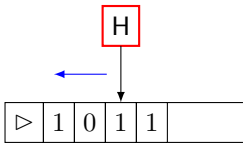
Output tape:



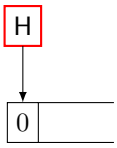
Input tape:



Work tape:



Output tape:



Note: the formal definition may require a $\triangleright$ at the beginning of the output tape, the procedure would be adjusted accordingly

2.4 Data representation

In many real-world problems our input data does not take the innately take the form of a bitstring, when working with Turing machines, it must be encoded as a bitstring. Not all data can be encoded as bitstrings but many can. An example of data that cannot is any member of the set of Real numbers $\mathbb{R}$.

Knowing this, we can answer a common question: "*why don't* we consider Turing machines with more than 1 input? " the answer: We can simply encode a list of bitstrings as a single bitstring!

3 Code as data

We have just seen that many types of data can be encoded as bitstrings and processed by Turing machines. We will now focus on the case where we encode **Program code** as input to a Turing machine.

We can encode Java programs as bitstrings, we can even encode other Turing machines as bitstring input to a Turing machine as our Turing machine is essentially a 6-tuple as we mentioned earlier.

For any bitstring α we will denote the corresponding Turing machine M_α.

We will examine two consequences of this encoding:

3.1 Universal Java program

A **universal Java program**, this simply refers to a Java *interpreter* written entirely in Java. It is able to execute **any** Java program.

This is the same principal as a **universal Turing machine**, $\mathcal{U}$ being a Turing machine interpreter written in (on?) a Turing machine.

The universal Java program takes 2 parameters: α and x, encoded as a single bitstring and returns the same result as the machine M_α when applied to x

How can this be achieved?

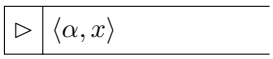
3.1.1 Sketch of the universal Turing machine, $\mathcal{U}$

$\mathcal{U}$ can be implemented using 4 (work) tapes. It first *unpacks* the input $\langle \alpha, x \rangle$ into its two constituent parts and places them onto the first two work tapes.

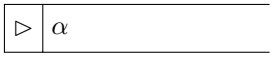
The machine M_α may use an alphabet of 100 symbols and 700 work tapes, we can always simulate it using just the alphabet $\{\triangleright, \square, 0, 1\}$ and two work tapes (inc. output tape).

We essentially simulate M_α by providing the machine defined on 1^{st} work tape (α) with our final 3 work tapes as it's input, work and output tapes respectively. An example will clarify this:

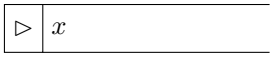
Input tape:



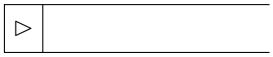
Work tape 1:



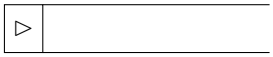
Work tape 2:



Work tape 3:



Work tape 4:



Where work tape 2,3,4 become input, work tape and output tape for the machine defined on work tape 1 (M_α)

3.2 Diagonalisation

This is our second consequence of treating code as data.

Diagonalisation is a proof method that can be used to show problems to be hard or even impossible.

Let us examine how to use it to prove the undecidability of the **halting problem**

The halting problem (HALT) can be defined as follows:

Problem 1 (The halting problem) *the set of pairs $\langle \alpha, x \rangle$ (encoded as a single bitstring) such that machine M_α, executed on input x, halts, i.e. it does not run forever.*

Turing's proof to this problem is as follows:

Proof 1 *Suppose that N is a machine that solves the halting problem.*

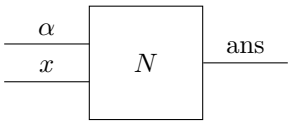
We can convert it into a machine N' that, given x, runs forever if $\langle x, x \rangle \in$ `HALT` (i.e. the machine M_x executed on x halts), and halts otherwise.

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

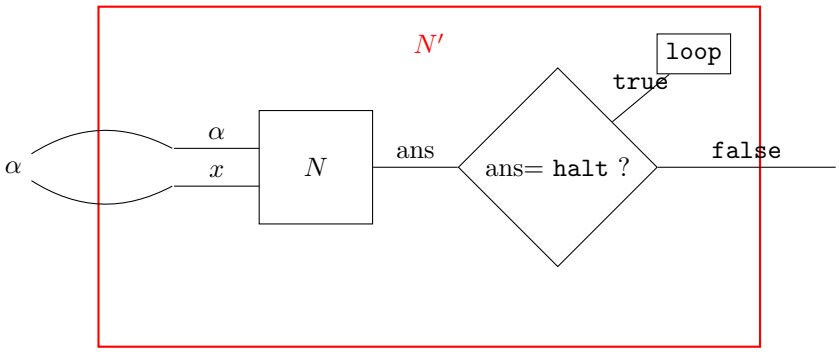
Running N' on α halts if it runs forever and runs forever if it halts.

We have derived a contradiction.

This can be shown clearly diagrammatically:
Given a machine N that can solve the halting problem:



We can construct a machine N' that given a single input, x, runs forever if M_x executed on x halts and halts otherwise:



Where the outermost α is the bitstring of N'

Above you can clearly see that if N' is run on the bitstring representation of itself (α), it will halt only if it does not halt and it will hang if it halts. This cannot occur, therefore we cannot construct N and the problem is undecidable.