FLA 2026 Lecture Notes

Brendan Griffiths

2026-02-13

Table of contents

Preface

This book is a collection of lecture notes for COMS3003A and COMS3021A.

1 An Introduction to Theoretical Computer Science

Maybe one day I will write something for this page...

2 Data, Representations, and Problems

2.1 Data

2.1.1 Primitive Data

Symbols: The most basic unit - may be anything that can be Read & Written

- Numbers: $0, 1$
- Latin Letters: $A, B, C, ..., Y, Z$
- Greek Letters: α, β
- Electricity: $1.5V$, $3.0V$
- Sound

Alphabet: A finite set of symbols

- Common notation Σ, Γ, Δ

Strings / Words:

- A finite sequence of symbols
- ε is the empty word
- Words have a length $|w| = n \iff$ there are n symbols in w
  - $|\varepsilon| = 0$

String Operators:

- Concatenation - connecting two symbols or strings
  - $w_1 \circ w_2 = w_1 w_2$
- Repeat Concatenation - concatenate the same string the specified number of times
  - $w^n = w \circ w \circ w \circ \, ... \, \circ w$

2.1.2 Describing Strings

String Generators: Describe a set of strings

- Union - Either string
  - $w \cup v = \{ w, v \}$
- Kleene Star - Repeat the string any numbers of times
  - $w^* = \{ \varepsilon, w, ww, www, wwww, \dots \}$
- Kleene Plus - repeat the string at least once
  - $w^+ = ww^* = \{ w, ww, www, \dots \}$
- Otherwise called a regular expression

The set of all strings over an alphabet is

$$\Sigma^*$$

Note

Let $\Sigma = \{ a, b \}$

$$\Sigma^* = \left\{ \begin{matrix} \varepsilon, \\ a, aa, aaa, \dots \\ b, bb, bbb, \dots \\ ab, aba, abaa, \dots \\ abba, abbaa, abbaaa, \dots \end{matrix} \right\}$$

2.1.3 Languages

A language L is a set of strings (i.e. a subset of Σ^*)

Notation

As shorthand, we may specify a set L is a language by indicating that it is a subset of all strings.

$$L \subseteq \Sigma^*$$

Language Examples:

- The set of all strings with two symbols

$$\{w \in \Sigma^* \mid |w| = 2\}$$

- The set of all strings that end in 0

$$\{w \in \Sigma^* \mid w = v \circ u \text{ and } u = 0\}$$

2.2 Representations

A representation is an interpretation of objects into strings

If I give you 5 objects, how would you label them? How about 10 objects? What about an infinite number of objects?

Formally, a representation is a mapping from a set of objects X to a string

$$r : X \mapsto \Sigma^*$$

We also want to add the following properties to our representation:

- Every input has some output

$$\forall x \in X, \exists w \in \Sigma^*, r(x) = w$$

- Every output has at least one input

$$\forall w \in \Sigma^*, \exists x \in X, r(x) = w$$

- If two inputs are different, their outputs must be different

$$\forall x, y \in X, x \neq y \implies r(x) \neq r(y)$$

A brief interlude on relations

A relation $\alpha : X \mapsto Y$ is the more general idea of a function. A function $f : X \mapsto Y$ requires that each $x \in X$ maps to only one $y \in Y$. Relations are a relaxation of that to-one property (called functional). Our first property of all $x \in X$ having at least one $y \in Y$ is called serial.
Another way to think of relations is using sets. A binary Relation $R : X \mapsto Y$ is the subset $R \subseteq X \times Y$. Notationally, we say $x \, R \, y$ (x relates to y by R) if $(x, y) \in R$
An example is equality of numbers $=: \mathbb{R} \mapsto \mathbb{R}$

$$\forall x, y \in \mathbb{R}, x = y \iff (x, y) \in \{\ (a, a) \mid a \in \mathbb{R}\ \}$$

Notation

Instead of writing $r(x)$ we will write $\langle x \rangle_r$ to mark it as a string
If it is clear what representation we are using, then we can drop the r

A representation allows us to take random objects and 'insert' them into strings, but we also want to be able to 'extract' an object out of a string.

So we add the additional requirement that our representation must be invertible. We define some default value for X. If the string cannot be identified, we just map it to this default value.

Warning

We are hand-waving away a fairly important problem. Two strings may represent the same object, for example in decimal numbers: $0.99999... = 1$
As an exercise, try write a more rigorous definition of a representation using a tool you were taught in abstract maths (hint: equivalence)

2.2.1 Natural Numbers $\mathbb{N}$ in an alphabet of k symbols

$$\forall n \in \mathbb{N}, \langle n \rangle_{\mathbb{N},k} = \text{The base-}k\text{ encoding of } n$$

By default, we will use $k = 2$. If the k is left off, then assume it is 2

2.2.2 Integers $\mathbb{Z}$ in an alphabet of k symbols:

$$\forall z \in \mathbb{Z}, \langle z \rangle_{\mathbb{Z},k} = \begin{cases} 0 \circ \langle |z| \rangle_{\mathbb{N},k} & \text{if } z \geq 0 \\ 1 \circ \langle |z| \rangle_{\mathbb{N},k} & \text{if } z < 0 \end{cases}$$

Warning

What does the string 00 mean in this representation?
This is where our "default" value for a "garbage" string comes into play. We cannot recogize 00 according to our rules, so we instead just map it to the integer 0.

2.2.3 Rationals $\mathbb{Q}$ in an alphabet of k symbols

$$\forall \frac{x}{y} \in \mathbb{Q}, \left\langle \frac{x}{y} \right\rangle_{\mathbb{Q},k} = \langle x \rangle_{\mathbb{Z},k} \# \langle y \rangle_{\mathbb{Z},k}$$

We've had to add the symbol # to separate our integers in the fraction. We can simplify this down into just our alphabet of k symbols.

To pull a rational number out of a string, we use the reverse of the above, but if we cannot recognize the string, we instead default to 0.

Binary simplifier example:

$$e_2(x) = \begin{cases} 00 & \text{if } x = 0 \\ 01 & \text{if } x = 1 \\ 10 & \text{if } x = \# \\ 11 & \text{otherwise} \end{cases}$$

2.2.4 Booleans

$$\langle b \rangle_{\mathbb{B},k} = \begin{cases} 1 & \text{if } b = \text{True} \\ 0 & \text{if } b = \text{False} \end{cases}$$

2.2.5 Complex Objects

We know have a fairly robust set of primitive representations. We can encode natural numbers, integers, rationals, and booleans, but how can we encode a more complex object? One that is made up of several sub-objects?

If our object has 2 components (imagine a 2D vector), we can employ the same separation as our rationals.

$$\left\langle \begin{pmatrix} x \\ y \end{pmatrix} \right\rangle := \langle x \rangle \# \langle y \rangle$$

For m components?

$$\left\langle \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \right\rangle := \langle a_1 \rangle \# \langle a_2 \rangle \#...\# \langle a_m \rangle$$

⚠ What if we have a vector of rationals?

We use # as a separator for both our vector components and our numerator and denominator. This is a collision of meaning. How can we write something like

$$\begin{pmatrix} \frac{a}{b} \\ \frac{c}{d} \end{pmatrix}$$

Our current encoding of this just becomes

$$\left\langle \begin{pmatrix} \frac{a}{b} \\ \frac{c}{d} \end{pmatrix} \right\rangle = \langle a \rangle \# \langle b \rangle \# \langle c \rangle \# \langle d \rangle$$

which is either a 4D integer vector, a 3D vector in $\mathbb{Z} \times \mathbb{Q} \times \mathbb{Z}$, or a 2D vector of rationals. We can replace the vector separator # with either a new symbol ; or an additional #

$$\left\langle \begin{pmatrix} \frac{a}{b} \\ \frac{c}{d} \end{pmatrix} \right\rangle = \langle a \rangle \# \langle b \rangle \, ; \langle c \rangle \# \langle d \rangle$$

To get back to Binary alphabet we can use the following encoding:

$$e(s) = \begin{cases} 00 \; s = 0 \\ 01 \; s = 1 \\ 10 \; s = \# \\ 11 \; s = ; \end{cases}$$

⚠ What is the smallest alphabet?

What is the smallest possible alphabet we can construct? How might this alphabet represent different objects?

2.3 Encodings

Encodings are the technique we use to write symbols in a different alphabet. We apply them on a semi-regular basis in computer science.

An encoding is a mapping that converts a symbol to a string in another alphabet

$$e : \Sigma \mapsto L \subseteq \Gamma^*$$

$$s \in \Sigma \implies e(s) \in L \subseteq \Gamma^*$$

Notation

We can also "encode" strings

$$\text{Let } w = s_1 s_2 ... s_n \in \Sigma^* \text{ and } e : \Sigma \mapsto L \subseteq \Gamma^*$$

$$e(w) = \langle w \rangle_e = e(s_1)e(s_2)...e(s_n) \in \Gamma^*$$

There are two types of encodings:

- Fixed Width: All encoded strings are the same length
  - A new symbol is indicated by reaching the end of the fixed length
  - Example: ASCII
$$e : \Sigma \mapsto \Gamma^n$$
- Variable Width: All encoded strings have different lengths
$$e : \Sigma \mapsto \{w \in \Gamma^* \mid |w| \leq n\}$$
  - A new symbol is indicated in some fashion
    - All symbols start the same way, but feature different symbols for the rest
    - A space between symbols
  - Example: Morse Code
    - Morse Code was originally designed as an auditory alphabet. It has three symbols, a *short sound* (dot), a *long sound* (dash), and *no sound* (seperator). However, we can simplify it into a binary alphabet of sound, no sound by defining a dash as the repetition of sound over a fixed time interval (the standard is 3 times the length of a dot).

Warning

All encodings are representations, just from an alphabet of symbols Σ to strings Γ^* instead of a arbitrary set X. Can we make an encoding from a set of strings to another set of strings?

$$e : \Sigma^* \mapsto \Gamma^*$$

2.4 Problems

We want to draw a distinction between an algorithm and a problem.

We define a problem, as the class of inputs, and class of outputs for which we are interested in.

$$P : X \mapsto Y$$

An algorithm is the set of instructions by which we manipulate our inputs into our outputs.

A good intuition for the distinction is that algorithms can call other algorithms, but they cannot call a problem.

Note

To rely on some prior programming experience, you can imagine that a problem is a "declaration" in code, while the algorithm is the actual "definition".
For example, in C++, in some header (.h) file we write

```
unsigned int compute_square(unsigned int n);
```

which is our "Problem" COMPUTE_SQUARE : $\mathbb{N} \mapsto \mathbb{N}$.
and then in a .cpp file we have

```
unsigned int compute_square(unsigned int n) {
    unsigned int n_square = n * n;
    return n_square;
}
```

as our algorithm.
This is not a perfect one-to-one relationship. Our header definition lacks the description of what our output should look like as the full definition of COMPUTE_SQUARE is

$$\text{COMPUTE\_SQUARE}(n) = n^2$$

and implementing it in code is itself an *almost* verbatim description of the problem. Use it as a starting off position.

Examples:

- Prime Factorisation: Input any natural, output: list of prime numbers

$$P : \mathbb{N} \mapsto [\mathbb{N}]$$

- Derivative: Input a differentiable function, output the derivative function

$$P : F \mapsto F'$$

- Swap Two Numbers: Input two naturals, output two naturals swapped

$$P : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N} \times \mathbb{N}$$

We want to "compute" the answer, so we turn the domain and co-domain into a string with representations r_X and r_Y that we can manipulate

$$P : r_X(X) \mapsto r_Y(Y)$$

A **Computational Problem** is therefore a function f from one set of strings Σ^* to another set of strings Γ^*

$$f : \Sigma^* \mapsto \Gamma^*$$

The most common form however is

$$f : \{ 0, 1 \}^* \mapsto \{ 0, 1 \}^*$$

Sometimes, we just want to identify if a given object has some property attached to it

- Is this number even?
- Is it a rational?
- Can I take the derivative?

We call these kinds of a problems a **Decision Problem**

$$f : \Sigma^* \mapsto \{ 0, 1 \}$$

In fact, we can restructure any computational problem into a decision problem.

2.4.1 Decision Problem and Computational Problem Equivalence

As an example: Lets take the computation problem of dividing a natural number

$$\text{Let } n, m \in \mathbb{N}, \text{ and } \frac{n}{m} \in \mathbb{Q}$$
$$\text{COMPUTE\_NATURAL\_DIV}(n, m) = \frac{n}{m}$$

Let assume an algorithm for COMPUTE_NATURAL_DIV exists, say A.

If we give A a string natural number pair $\langle n, m \rangle$ it will output the string $\langle \frac{n}{m} \rangle$

We define the Decision problem with $n, m, q \in \mathbb{N}$

$$\text{IS\_NATURAL\_DIV}(n, m, q) = \begin{cases} 1 \text{ if } \frac{n}{m} = q \\ 0 \text{ if } \frac{n}{m} \neq q \end{cases}$$

Let A be an algorithm that solves COMPUTE_NATURAL_DIV.

We write a new algorithm B on that solves IS_NATURAL_DIV

define B on input $\langle n, m, q \rangle$

1. If $A(\langle n, m \rangle) = q$
    - True => Return 1
    - False => Return 0

And now for the reverse:

Let B be an algorithm that solves IS_NATURAL_DIV

define A on input $\langle n, m \rangle$:

1. for q in $[1, n]$
    1. if $B(\langle n, m, q \rangle) = 1$
        - True => return q
        - False => do nothing
2. return $\langle 0 \rangle$

Algorithm A solves COMPUTE_NATURAL_DIV as:

1. If $n < m$ then $\frac{n}{m} \notin \mathbb{N}$ and no answer exists or $n = 0$ therefore the loop will never have a q with $B(\langle n, m, q \rangle) = 1$, thus we return $\langle 0 \rangle$
2. If $n = m$ then $\frac{n}{m} = 1$, therefore for $q = 1$, we will return q
3. If $n > m$ then $n = qm$
    - The smallest possible q is 1 and $m = n$
    - The largest possible q value is $q = n$ and occurs when $m = 1$
    - If $q > n$ then $m < 1$, thus $m \notin \mathbb{N}$, therefore B does not apply

⚠ Warning

Try to do the same for a computational problem P. Given $P : \{0,1\}^* \mapsto \{0,1\}^*$ and algorithm A_P, show there exists a decision problem $P' : \{0,1\}^* \mapsto \{0,1\}$

2.4.2 Language and Decision Problem Equivalence

Languages are equivalent to decision problems. We can construct a decision problem out of language membership, and we can construct language membership out of a decision problem.

$$w \in L \iff D(w) = 1$$

Note

Show $D(w) = 1 \implies w \in L$

Let $D : \Sigma^* \mapsto \{0, 1\}$ be a decision problem.

Define the language L as

$$L = \{w \in \Sigma^* \mid D(w) = 1\}$$

Show $w \in L \implies D(w) = 1$

Let $L \subseteq \Sigma^*$ be a language.

Define the decision problem D as

$$D(w) = \begin{cases} 1 & w \in L \\ 0 & w \notin L \end{cases}$$

3 Turing Machines I

3.1 A machine that computes

Your intuition might be that a computer is a small box that sits on a desk, in a server rack, and even in your pocket. Making use of electronics, digital circuits, and millions of bits and bytes to perform what is sufficiently indistinguishable from magic [1].

In 1936, they had a very different picture. The closest equivalent were mechanical devices that used levers, linkages, and gears to produce work - similar to the combustion engine. Instead of work being motion, it would be the evaluation a series of steps, generally finding a solution to some form of arithmetic. Mechanical calculators were probably the most well known instances of a "computer".

https://www.youtube.com/embed/s_hbvRTGcUI

Note

The first general purpose computer was actually proposed about 100 years before Turing in 1837. Charles Babbage designed a machine he called the "Analytical Engine".
This machine was a digital-mechanical hybrid that made use of components that would not be out of place in a contemporary digital computer. Had it been finished, it would have been the first known instance of device that is Turing Complete [2].
Ada Lovelace is often called the first programmer because she wrote the first *published* progam for this device. [3] This program, called Note G, calculated a sequence of Bernoulli Numbers.

When we build our mathematical model of a computer and algorithm, I want you to keep in mind the 'mechanical' nature of it.

For our model to be correct, there are important ideas that need to be captured:

1. It should be as simple as possible
    - What is the smallest amount of 'work' we can get away with for each step in our device?

[1] Arthur C. Clarke's third law
[3] citation needed
[3] One day I will find citations for this...

- We will use a string as the only data structure

2. It should be able to do everything an algorithm can do - For loops, if statements, variables etc. should all be implementable on the device

3.2 Turing Machines

3.2.1 Some intuition

Imagine a physical device that has three components:

1. A space to read and write symbols - a tape of sequential cells
2. A mechanism to perform the read and write - a head that can read or write on a given cell on our tape
3. A control unit to tell the the head what symbol it should be writing - a box that keeps track of our algorithm steps and where along that algorithm where are

- On our tape, we have a set of symbols on every cell - even if that symbol is a "blank" ($\sqcup$) symbol to represent nothing ($\sqcup, 0, 1, 2, ...$).
- We will also assume that our tape has as much space as it needs - i.e. an infinite tape.
- Our head mechanism, can move between the different cells going either left or right along it.
- The control unit is composed of various 'device states' ($q_1, q_2, q_3, ...$) that holds the information about whats happened so far (Gears in different positions, or electronic signals in specific patterns).

This device has only one instruction: A conditional statement based off the current device state q_i, and the symbol currently on the tape s_n that tells us what state to go to next q_j, what symbol should be written on the tape s_m, and whether we should go left or right $D \in \{L, R\}$.

$$q_i, s_n \to q_j, s_m, D$$

Note

If we tried to build this device in the real world, these instructions are just the series of levers and gears that interact to create our desired effect.

- Write this symbol - rotate a ball with all the symbols on it
- Change to the 5th state - rotate this gear by 5 teeth from this notch
- Go left - rotate this other gear counter-clockwise

https://www.youtube.com/embed/yKpIwi1UUIk?si=_2KDfjjzrn8RyoPD&start=115

We will add the stipulation that if our machine enters specific states, the computation will stop: For a Yes / No (Accept / Reject) machine we will have the special states $q_{\text{accept}}, q_{\text{reject}}$ indicating the corresponding answer.

Additionally, we need to start somewhere, so we designate q_0 as the 'start state'. If we need to run this device, we must first reset it so that we are in q_0.

Our programming language for this device will be all the possible permutations of these instructions that direct the device:

- In q_0,
  - reading $\sqcup$, go to $\{q_0, q_1, q_{\text{halt}}\}$, writing symbol $\{\sqcup, 0, 1\}$ and moving $\{L, R\}$.
  - reading 0, go to $\{q_0, q_1, q_{\text{halt}}\}$, writing symbol $\{\sqcup, 0, 1\}$ and moving $\{L, R\}$.
  - reading 1, go to $\{q_0, q_1, q_{\text{halt}}\}$, writing symbol $\{\sqcup, 0, 1\}$ and moving $\{L, R\}$.
- In q_1,
  - reading $\sqcup$, go to $\{q_0, q_1, q_{\text{halt}}\}$, writing symbol $\{\sqcup, 0, 1\}$ and moving $\{L, R\}$.
  - reading 0, go to $\{q_0, q_1, q_{\text{halt}}\}$, writing symbol $\{\sqcup, 0, 1\}$ and moving $\{L, R\}$.
  - reading 1, go to $\{q_0, q_1, q_{\text{halt}}\}$, writing symbol $\{\sqcup, 0, 1\}$ and moving $\{L, R\}$.

Our program δ is the subset of of these permutations that describe the particular behaviour we want.

- In q_0
  - reading $\sqcup$, go to q_1, writing $\sqcup$ and moving L
  - reading 0, go to q_0, writing 0 and moving R
  - reading 1, go to q_0, writing 1 and moving R
- In q_1
  - reading $\sqcup$, go to q_{reject}, writing $\sqcup$ and moving R
  - reading 0, go to q_{reject}, writing 0 and moving R
  - reading 1, go to q_{accept}, writing 1 and moving R

⚠ Question

What do these instructions actually do? Try describe the above algorithm in english.
If we use the binary repesentation of integer's, $\langle z \rangle_{\mathbb{Z}}$, what is happening to the number $z \in \mathbb{Z}$?

And a computation is the series of changes that has occured in our device and tape.

1. Set the tape to be the input string 0111,
2. Set the device to be in q_0 and the head to be at the first symbol

- We always start at the first symbol of our input

3. Follow each instruction until q_{accept} or q_{reject}
   - Step 0: In q_0, scanning cell 0, reading symbol 0, go to q_0, write 0, move right to cell 1
     - $q_0 0111$
   - Step 1: In q_0, scanning cell 1, reading symbol 1, go to q_0, write 1, move right to cell 2
     - $0q_0 111$
   - Step 2: In q_0, scanning cell 2, reading symbol 1, go to q_0, write 1, move right to cell 3
     - $01q_0 11$
   - Step 3: In q_0, scanning cell 3, reading symbol 1, go to q_0, write 1, move right to cell 4
     - $011q_0 1$
   - Step 4: In q_0, scanning cell 4, reading symbol $\sqcup$, go to q_1, write $\sqcup$, move left to cell 3
     - We have moved off the end of our string into a "blank" cell
     - $0111q_0 \sqcup$
   - Step 5: In q_1, scanning cell 3, reading symbol 1, go to q_{accept}, write 1, move right to cell 4
     - $011q_1 1$
   - Step 6: In q_{accept} scanning cell 4 $\rightarrow$ halt
     - $0111q_{\text{accept}} \sqcup$
4. Read the result as 1 as we are in q_{accept}

Congratulations, we have just built a physical device that evaluates some algorithm. If we use $\langle z \rangle_{\mathbb{Z}}$, then this algorithm determines if the given integer is even or odd.

3.2.2 Formal Definition

The previous device is essentially an implementation of a Turing Machine as a mechanical machine.

Mathematically, a Turing Machine for Decision Problems are defined as follows

A **Decision Turing Machine** M is the tuple $(Q, \Gamma, \Delta, \delta, q_{\text{init}}, q_{\text{accept}}, q_{\text{reject}})$ where

- Q is a finite non-empty set of *states*
- Γ is an **input alphabet** such that $\sqcup \notin \Gamma$
- Δ is a **tape alphabet**, satisfying the following conditions:

    1. $\Gamma \subseteq \Delta$
    2. $\sqcup \in \Delta$
    3. $\Delta \cap \Gamma = \emptyset$

- $q_{\text{init}}, q_{\text{accept}}, q_{\text{reject}} \in Q$
- δ (called the **transition function** or **program**) is a finite set of instructions in the form $q_i, s_n \to q_j, s_m, D$
    1. $q_i, q_j \in Q$
    2. $s_n, s_m \in \Gamma$
    3. $D \in \{L, R\}$
    4. and for all $q \in Q \backslash \{q_{\text{accept}}, q_{\text{reject}}\}$ and all $s \in \Delta$ there exists only one instruction in δ that has the form $q, s \to q', s', D$

Question

We have seen previously that Decision Problems and Computational Problems are equivalent if an algorithm exists for one, then an algorithm must exist for the other. Try to rewrite this proof but using a Turing Machine (See Computation Problems for a definition of Computational Turing Machines) instead of a generic algorithm.

Notation

Decision and Computation equivalence means that the exact specifics of our Turing Machine are not that important. We stick to a Decision Turing Machine because it is a stronger continuation from Regular Languages and Context-Free Grammers. However, there is no reason you cannot tackle this entire branch using Computational problems instead.

3.2.3 Computing

A **configuration** is the current state of our Turing Machine, where the head is positioned, and what symbols are on the tape.

Let M be a Turing Machine,

- A **configuration** of M and tape is a word w in the alphabet $Q \cup \Delta$ satisfying the following:
    1. w contains exactly one symbol from Q
    2. w does not begin with $\sqcup$
    3. If the last symbol of w is a $\sqcup$, the it is preceded by a symbol from Q
- A **configuration** is *halting* if it contains $q_{\text{accept}}, q_{\text{reject}}$
- A **configuration** is *accepting* if it contains q_{accept}

- A **configuration** is *rejecting* if it contains q_{reject}

Our machine starts in q_{init} and has the input string written on its tape

Let M be a Turing Machine, and $x \in \Gamma^*$,

The **initial configuration** of M on x is

$$q_{\text{init}} x$$

Computation is the process of moving between different configurations of our machine and tape. In fact, a computation is the sequence of configurations start from the initial configuration until it is in a halting configuration.

Let M be a Turing Machine, and w and v are configurations of M,

We say that w yields to v ($w \vdash v$) if one of the following conditions hold:

1. $q, s \rightarrow, q', s', L \in \delta$ and
   - either $w = xtqsy$ and $v = xq'ts'y$ for $t \in \Delta$ and $x, y \in \Delta^*$
   - or $w = qsx$ and $v = q' \sqcup s'x$ for $x \in \Delta *$
2. $q, s \rightarrow, q', s', R \in \delta$ and
   - either $w = xqsy$ and $v = xs'q'y$ for $x, y \in \Delta^*$
   - or $w = xqs$ and $v = xs'q' \sqcup$ for $x \in \Delta *$

Note

Our example in 3.2.1 produces the sequence of configurations

$$q_0 0111 \vdash 0q_1 111 \vdash 01q_1 11 \vdash 011q_1 1 \vdash 0111q_1 \sqcup \vdash 0111 \sqcup q_{\text{halt}} \sqcup$$

Let M be a Turing Machine and $x \in \Gamma^*$,

The **computation** of M on x is the sequence of configurations $c_0, c_1, c_2, \ldots$ such that

1. c_0 is the initial configuration of M on x
2. $c_i \vdash c_{i+1}$ for all $i \geq 0$
3. The sequence has at most one halting configuration.
4. If the sequence has a halting configuration c_n, then c_n is the last configuration in the sequence.

Notation

We will use the symbol $C(M,x)$ to refer to the computation of M on x.
The length of a computation $|C(M,x)|$ is the number of configurations in a sequence.

We want to identify the **status** of our computation

Let M be a Turing Machine M and $x \in \Gamma^*$

- M **halts** on x if the computation $C(M,x)$ (sequence $c_0, c_1, c_2, \ldots$) contains a halting configuration
- M **loops** on x if the sequence $c_0, c_1, c_2, \ldots$ does not contain a halting configuration
- M **accepts** x if the sequence $c_0, c_1, c_2, \ldots$ contains an accepting configuration
- M **rejects** x if the sequence $c_0, c_1, c_2, \ldots$ contains a rejecting configuration

Notation

$$M(x) = \begin{cases} 1 & \text{if } M \text{ accepts } x \\ 0 & \text{if } M \text{ rejects } x \\ \infty & \text{otherwise} \end{cases}$$

Lastly, we want a way to classify TM's based on whether they accept, reject or loop on all inputs.

Let M be a Turing Machine

- M is a **decider** if it accepts or rejects x for all $x \in \Gamma^*$
- M **decides** / **solves** a decision problem $P : \Gamma^* \mapsto \{0,1\}$ if $\forall x \in \Gamma^*, M(x) = P(x)$

Question

Let M be a Turing Machine

1. If M is a decider, does it solve some decision problem?
2. If M solves a decision problem, is it a decider?

Notation

We say that a Turing Machine M **decides** a language $L \subseteq \Gamma^*$ if

$$M(x) = 1 \iff x \in L$$

Two Turing Machines M and N on the same input alphabet Γ are equivalent if they have the same input and output behaviour

$$\forall x \in \Gamma^*, M(x) = N(x)$$

⚠ Question

How many equivalent Turing Machines are there?
Proof Sketch: If I take some Turing Machine M, we can add a new state q and make the new machine M'. We can make q fall between any state q_i that has some transition to q_{accept}.

1. If q_i has a transition to q_{accept}, make it instead transition to q (leaving the rest of the transition unmodified)
2. q transitions to q_{accept} on every symbol, leaving the symbol unmodified.

M' has not changed anything about any computation besides add a new configuration c_{n+1} to the sequence. It has not changed the outcome, nor has it changed the tape in anyway. Thus M' is equivalent to M.
We can then create M'' in the exact same fashion from M'. Repeat this ad-infinitum, and we have an infinite number of equivalent Turing Machines.

3.2.4 Computational Problems

A Turing Machine for computational problems is very similar to the standard definition. The only difference is that we have a single notable state q_{halt} instead of q_{accept} and q_{accept}

A **Computational Turing Machine** M is the tuple $(Q, \Gamma, \Delta, \delta, q_{\text{init}}, q_{\text{halt}})$ where

- Q is a finite non-empty set of *states*
- Γ is an **input alphabet** such that $\sqcup \notin \Gamma$
- Δ is a **tape alphabet**, satisfying the following conditions:

  1. $\Gamma \subseteq \Delta$ - Our input alphabet can be written on the tape
  2. $\sqcup \in \Delta$ - Our tape has a blank symbol
  3. $\Delta \cap \Gamma = \emptyset$ - The symbols and states are different

- $q_{\text{init}}, q_{\text{halt}} \in Q$
- δ (called the **transition function** or **program**) is a finite set of instructions in the form $q_i, s_n \to q_j, s_m, D$

  1. $q_i, q_j \in Q$
  2. $s_n, s_m \in \Gamma$
  3. $D \in \{L, R\}$

4. and for all $q \in Q\backslash\{q_{\text{halt}}\}$ and all $s \in \Delta$ there exists only one instruction in δ that has the form $q, s \to q', s', D$

The definitions of computation and status of a Computational TM remain the same as a Decision TM. However, we will use a slightly different configuration to define these objects:

- A **configuration** is halting if contains q_{halt}

Computational Turing Machines need to provide some form of output string that we can read off the tape. To be consistent, we say that our resultant string is the string from the tapes head until the next blank symbol that is composed of only input symbols Γ.

Let M be a Computational Turing Machine.

We define the **result** of a computation of M on x as the string w satisfying the following properties

- $w \in \Gamma^*$
- w is written on the tap
- w is immediately followed by $\sqcup$
- M is scanning the first symbol of w

Notation

$$M(x) = \begin{cases} w & \text{if } C(M, x) \text{ halts} \\ \infty & \text{otherwise} \end{cases}$$

Computational TM's can also **solve** some Computational Problem $P : \Gamma^* \mapsto \Gamma^*$. However we use the term M **computes** P to specify it is a computational problem.

Let M be a Computational Turing Machine and $P : \Gamma^* \mapsto \Gamma^*$ a computational problem

- M **computes** P if
$$\forall x \in \Gamma^*, M(x) = P(x)$$

and this gives us a fairly strong definition for when something is computable or not.

Let $f : \Sigma^* \mapsto \Sigma^*$ be a function.

- f is **computable** if there exists some Turing Machine that computes f

3.3 Turing Machines as Algorithms

3.3.1 Analysis of Turing Machines

When we perform an analysis of an algorithm, we typically focus on two resources available to our algorithm:

- Time Complexity - How long will it take to run our algorithm
- Space Complexity - How much memory do we need for our algorithm

If we want to use Turing Machines as a mathematical model for our algorithms, we need to define these same tools.

3.3.1.1 Time

Our Turing Machine has only the single instruction. If we build this machine in reality, then it will take a roughly fixed amount of time to execute this instruction, no matter how many times we do it.

- If we built this device using mechanical components, this time is how long it takes to rotate the gears, manipulate the linkages and so on.
- If we built it use electronic signals, it would be the time to change the voltage of our circuit.

This means we can discard the actual unit and instead count the number of times an instruction is executed. We can recover the actual execution time by taking an average time to execute a single instruction and multiply it by the number of instructions.

A single instruction allows us to move between the different configurations of our machine, thus the number of instructions is the number of configurations in a computation.

Let M be a Turing Machine and $x \in \Gamma^*$,

The **computation time** of M on x is the length of the computation.

$$T(M, x) = |C(M, x)|$$

$T(M, x)$ allows us to analyse the execution of a single input, but we want to understand how it runs on all inputs. However, if two strings are of equal length, then the machine will do the roughly the same *amount* of work, even if the result of that work is different. Instead of the individual strings we analyse the length of the strings.

Let M be a Turing Machine that will halt on every input, The **time function** $t : \mathbb{N} \mapsto \mathbb{N}$ of M is

$$t(n) = \max\{T(M,x) = |C(M,x)| : x \in \Gamma^n\}$$

In english, our time function for a given input n is the maximum computation time across all strings of length n.

Note

We use the maximum time because some inputs of the same length may occasionally do less work.
Consider the following python script

```
x = input()
for i in range(0, len(x))
    if x[i] == "0":
        break
```

If we input "111", the for loop will be executed 3 times. If we instead input "101" it will only be executed twice, even though the strings are the same length.
When analysing this algorithm, we want to know the worst it will do at a given length, not the best. So we take $t(3) = \max\{1,2,3,3\} = 3$ (Inputs: $\{0xx, 10x, 110, 111\}$ where $x \in \Gamma$)

3.3.1.2 Space

In a digital computer, space is measured in the number of bits used. We can do the same here, but instead of bits we will use our tape instead of transistors.

Let M be a Turing Machine and $x \in \Gamma^*$,

The **computation space** $S(M,x)$ of M on x is the number of unique tape cells accessed by our machine.

This is more difficult to tie down an exact value, but we can define some bounds.

The machine can only move left or right on an instruction, it cannot stay in place. Even if we use the empty string, the machine scans a cell in the initial configuration, and scans the next cell after the instruction that may let it halt.

$$q_{\text{init}}, \sqcup \to q_{\text{halt / accept / reject}}, s, D$$

For any other string, we can do the same, or just move back and forth between these two cells infinitely. So the smallest number of unique cells we can use is 2. And if we never re-use a cell i.e. every instruction accesses a new cell, then we can only access as many cells as there are instructions.

$$2 \leq S(M, x) \leq T(M, x)$$

For the same reasons as time, we want to study the lengths of a string and how the machine behaves, so we define our **space function** in a similar manner.

Let M be a Turing Machine that will halt on every input,

The **space function** $s : \mathbb{N} \mapsto \mathbb{N}$ of M is

$$s(n) = \max\{S(M, x) : x \in \Gamma^n\}$$

3.3.2 The Church-Turing Thesis

At this point, you should have an understanding that Turing Machines are a fairly strong model for algorithms. We can analyse them the same way as algorithms, we can program them using instructions, we can build them in the real world (this is a surprisingly important idea), and there is more yet to come.

However, we have not really discussed if a Turing Machine is an exact model of an algorithm. To jump the gun, yes. Turing Machines are an incredibly robust model, and every definition of computer that has been developed in the last 100 years has been shown to be equivalent to at least some form of Turing Machine. Even digital computers with Random-Access Memory are just slightly faster versions of a Turing Machine. In Turing Equivalency, we will study this idea more concretely.

To formulate this, we have developed a simple thesis.

> Every computational problem solvable by an algorithm, can also be solved by a Turing Machine.

This is the Church-Turing Thesis. It allows us to assert that an algorithm *is* a Turing Machine. If we can't write a Turing Machine, then we can't write an algorithm and vice versa.

This is not a mathematical statement however. It converts an intuitive concept (an algorithm - sequence of steps) and converts it into a mathematical object (a TM - sequence of instructions). We cannot really prove this assertion, we only take it as true so long as it appears to be true.

4 Turing Machines II

4.1 Resource Usage

4.2 Machine Variants

4.2.1 One-Way Tapes

4.2.2 Multi-step instructions

4.3 Machines as Data

5 Turing Equivalency