Theory of Computing

Notes for COMP2048, The University of Queensland

George Stuart

Semester 1, 2026

About these notes. These are my notes from COMP2048, written to go alongside the course's lecture slides rather than to replace them. Some sections only summarise the main points, a few stop part-way through, and many proofs are left to the slides.

Contents

1 Overview of Computing 2
2 Regular Languages 3
3 Context-Free Languages 8
4 Turing Machines 12
5 Cellular Automata 18
6 Complexity Theory 20

1 Overview of Computing

Computing is the process of taking inputs and applying a well-defined procedure (i.e. an algorithm) to produce outputs.

- A computing problem can be framed as a function from inputs to outputs (Figure 1)

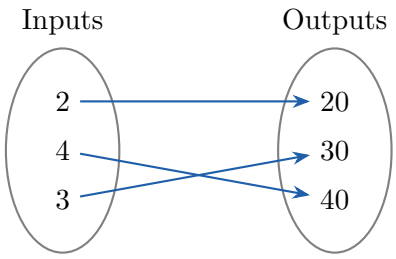


Figure 1: A computing problem as a function: each input is mapped to an output.

1.1 Decision Problem

One special kind of computing problem is a **decision problem**. This is a computing problem where the output set is a binary set

- All inputs map to one of the elements in the binary set (i.e. to yes/no or accept/reject etc)

Example:

- Is this string of even length?
- Is this an image of a cat?

This course will be focussed on decision problems.

1.2 Models of Computing

There are several models of computing that we can use to solve different decision problems, and we can represent them in a hierarchy in terms of how "powerful" the various models are (Figure 2).

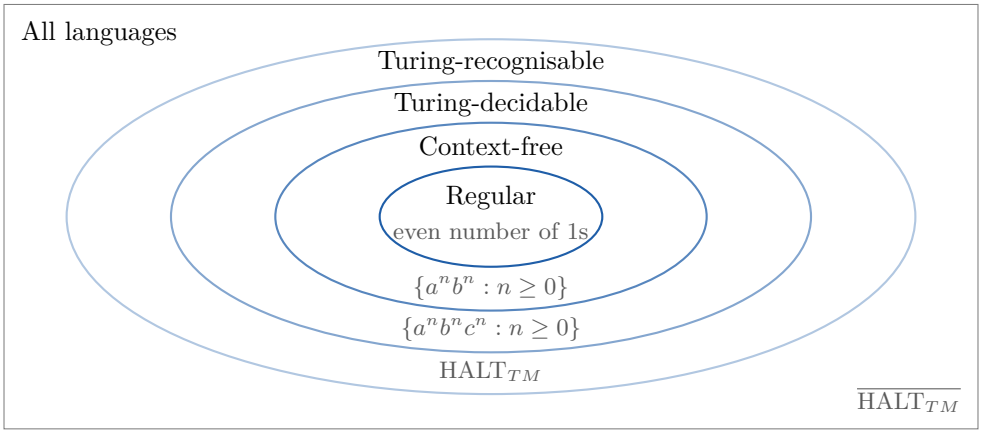


Figure 2: The hierarchy of languages covered in the course, with an example language in each region. Each class is strictly contained in the next.

This course builds up through these models of computation incrementally.

1.3 Important Notation

For any decision problem we have some **alphabet** Σ

- Σ contains all of the characters that make up the inputs to the problem
- Σ must be finite and non-empty

A **string** ω is a finite sequence of symbols where each symbol is in Σ:

- The length of the string is denoted $|\omega|$
- ε denotes the "empty string" and $|\varepsilon| = 0$
- Just about anything can be encoded as a string (text, images, numbers, etc)

Formally we say that any string ω is an element of Σ^* where Σ^* is called the **Kleene star** of the alphabet Σ. The Kleene star is simply the set of all finite strings that can be formed from the characters in Σ.

1.4 Languages

Next we have the concept of a **language**. A language $L \subseteq \Sigma^*$ is any subset of Σ^*. Given some string $\omega \in \Sigma^*$ we say that:

- a machine that recognises L accepts ω iff $\omega \in L$
- a machine that recognises L rejects ω iff $\omega \notin L$

Notice that,

$$\text{solving decision problem} \iff \text{recognising the corresponding language}$$

Here are a few important operations on languages:

- Language complement
$$\overline{L} = \Sigma^* - L$$
- Language concatenation
$$L_1 L_2 = \{\omega_1 \omega_2 : \omega_1 \in L_1 \text{ and } \omega_2 \in L_2\}$$
- Language Kleene star
$$L^* = \{\varepsilon\} \cup \{\omega_1 \cdots \omega_n : n \geq 1,\ \omega_i \in L\ \forall i = 1, \ldots, n\}$$

(i.e. the Kleene star of a language is defined in the same way as the Kleene star of an alphabet, except that the strings in the language are used as the individual terms).

2 Regular Languages

We start our journey into methods of computation for recognising languages with the study of deterministic finite automata (DFAs) which describe what are known as **regular languages**.

2.1 Deterministic Finite Automata (DFAs)

A DFA is a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

Where,

- Q is a set of states
- Σ is the alphabet
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states

A DFA can be fully described by either:

- Defining the 5-tuple explicitly
- Drawing a diagram (such as Figure 3)

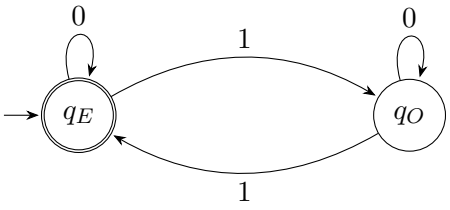


Figure 3: A DFA that accepts the binary strings containing an even number of 1s. The arrow marks the start state, and accepting states are drawn with a double circle.

General notes:

- Deterministic (since all of the transitions are deterministic).
- Has no memory other than its current state (similar to a Markov chain).
- A transition must be defined for *every* input symbol in *every* state.
- Tip: sometimes it may be useful to include "trap states" when designing a DFA

We denote the language recognised by the DFA by $L(M)$ where,

$$L(M) = \{\omega \in \Sigma^* \,:\, M \text{ accepts } \omega\}$$

2.2 Regular Languages

A language $L \subseteq \Sigma^*$ is said to be **regular** if there exists a DFA M such that,

$$L = L(M)$$

(or in other words there exists a DFA that recognises that language).

Suppose L and L' are two regular languages. Then the following languages can be shown to be regular,

- $\overline{L}$
- $L \cup L'$
- $L \cap L'$
- LL' (language concatenation)
- L^*

2.2.1 Proof Sketch for Complement

Here we want to prove that if L is regular then so is $\overline{L}$.

Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA that recognises L. Then define the DFA M' by,

$$M' = (Q, \Sigma, \delta, q_0, Q - F)$$

Then $L(M') = \overline{L}$ and so $\overline{L}$ is also regular.

2.2.2 Proof for Union and Intersection (Product Construction)

To prove that the union and the intersection of regular languages is still regular we use what is known as the "product construction". The basic idea is that we "essentially" run the two DFAs together in parallel.

For the union:

- Suppose $M = (Q, \Sigma, \delta, q_0, F)$ recognises L
- Suppose $M' = (Q', \Sigma, \delta', q_0', F')$ recognises L'
- Construct a new DFA M_{union} as follows,
  - states: $Q_\times = Q \times Q'$
  - alphabet: Σ
  - transition function: $\forall a \in \Sigma, \forall q \in Q, \forall q' \in Q'$

$$\delta_\times((q, q'), a) = (\delta(q, a), \delta'(q', a))$$

  - Start state: (q_0, q_0')
  - Accepting states: $\{(q, q') \in Q_\times : q \in F \text{ or } q' \in F'\}$

We can then do a very similar thing with the intersection. The only thing that changes is the set of accepting states (we change the "or" to an "and").

2.3 Regular Expressions

Regular expressions (regex) are another way to describe regular languages.

A regular expression is made up of two parts: base expressions and regular operations.

Base expressions:

- $\emptyset$
- ε
- any symbol from the alphabet (i.e. any $a \in \Sigma$)

Regular operations:

- Union: $R \cup S$
- Concatenation: RS
- Kleene star: R^*

Order of operations:

- parentheses
- star
- concatenation
- union

Example:

- All binary strings: $(0 \cup 1)^*$
- Binary strings starting with a 1: $1(0 \cup 1)^*$
- Binary strings containing 00: $(0 \cup 1)^*00(0 \cup 1)^*$

Since regular expressions are another way of describing *regular* languages, they are equivalent to DFAs in terms of the languages that they can describe. This is shown explicitly below.

2.4 Non-Deterministic Finite Automata (NFAs)

One attempt to try to increase the "power" of our method of computation might be to introduce a certain element of non-determinism. By power here I mean the set of languages that we are able to recognise.

The idea of an NFA is to try to extend the power of a DFA by allowing:

- zero, one or more possible next states in each transition
- transitions known as ε-transitions that don't consume an input

This introduces non-determinism, since after reading some input a you can end up in several states.

Here are the key points:

- Represented in a similar way to a DFA with $(Q, \Sigma, \delta, q_0, F)$
- The transition function however is different now

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$$

- i.e. the transition function maps a state–input pair to a set of possible future states, not just to a single future state. Note often we denote $\Sigma \cup \{\varepsilon\}$ simply by Σ_ε.

Notice that whilst a particular run of the machine on some input is non-deterministic, whether the machine accepts a given string is still well defined. This is because the acceptance rule for an NFA is defined as follows,

$$\text{Accept } \omega \in \Sigma^* \text{ if } \textit{any} \text{ branch of computation that reads the entire string}$$
$$\text{leads to an accept state}$$

Therefore so long as there is some valid branch of computation in the NFA then we say that the NFA "accepts" that string.

Also note that if the machine is in a state where it receives some input but can no longer transition (i.e. that transition is undefined) then the machine effectively stops and this can be treated as "rejecting" the string.

2.5 Uniting DFAs, NFAs and Regex

An important theorem,

$$\text{set of languages recognised by DFA}$$
$$=$$
$$\text{set of languages recognised by NFA}$$
$$=$$
$$\text{set of languages recognised by regex}$$
$$=$$
$$\text{regular languages}$$

The idea of the proof is by construction. I.e. given a DFA you should be able to construct a corresponding NFA and regex, and likewise in all the other directions.

Notice that some directions are trivial:

- DFA $\to$ NFA (trivial, since every DFA is already an NFA)

One important thing to note is that despite these models of computation all being equivalent it does not mean that they are not individually useful. For example NFAs can be very useful since the representation of a problem as an NFA will often be much smaller than its representation as a DFA.

2.6 NFA to DFA (via the Subset Construction)

Probably the hardest link to show is that given an NFA we can construct a corresponding DFA. However this is true and we can do it using what is known as the subset-construction.

Here is the proof assuming that there are no ε transitions.

Suppose we have an NFA N given by,

$$N = (Q, \Sigma, \delta, q_0, F)$$

Our goal is to construct a DFA M given by,

$$M = (Q', \Sigma, \delta', q_0', F')$$

such that $L(N) = L(M)$.

To do this we can construct M as follows:

- states: $Q' = \mathcal{P}(Q)$
- For any $R \in Q'$ and $a \in \Sigma$
$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$
- start state: $q_0' = \{q_0\}$
- Accepting states

$$F' = \{R \in Q' : R \text{ contains an accept state of } N\}$$

This construction essentially runs all chains of computation "simultaneously" and then accepts if any of the resulting states are accept states.

2.7 Power of NFA/DFA Equivalence

There are a few reasons why this is powerful:

- some systems are much easier to describe as NFAs
- we can also use NFAs to prove results involving regular languages much more easily

For example using an NFA style argument it is very easy to show that:

- union of two regular languages is regular
- intersection of two regular languages is regular
- concatenation of regular languages is regular
- star of a regular language is regular

2.8 Converting a Regex to an NFA

This is quite straightforward; see slide 21 of Lecture 3 for the details.

2.9 Pumping Lemma

We can prove that certain languages are not regular. For example the language,

$$L = \{0^n 1^n : n \geq 0\}$$

is not regular. The key to proving this is using what is known as the pumping lemma.

Theorem (Pumping Lemma) If L is a regular language then there exists a constant $p \geq 1$ (known as the pumping length) such that for every string $\omega \in L$ with $|\omega| \geq p$ there exist strings x, y, z such that,

- $\omega = xyz$
- $|y| \geq 1$
- $|xy| \leq p$
- For all $i \geq 0$, the string $xy^i z \in L$

2.10 Proving that a Language is NOT Regular

The idea of such a proof is like this.

1. Suppose (for the sake of contradiction) that L was regular
2. This means that the pumping lemma applies and so there exists some pumping length p
3. Choose some string $\omega \in L$ such that
$$|\omega| \geq p$$
4. Consider all possible decompositions of ω into xyz such that
   - $|xy| \leq p$
   - $|y| \geq 1$
5. Show that, for each of these decompositions, for some $i \geq 0$,
$$xy^i z \notin L$$

This would then contradict the pumping lemma and thus imply that L is not regular.

(These proofs mostly just take practice.)

2.11 Recognisers and Descriptors

We say that DFAs / NFAs are **recognisers** of regular languages. We say that regular expressions are **descriptors** of regular languages.

This is relevant to the next section because we have analogous terms there.

3 Context-Free Languages

3.1 Limitations of Regular Languages

Finite automata are a weak model of computation:

- have a fixed finite number of states
- have no memory of which states they came from

We saw as an example previously that $\{0^n 1^n : n \geq 1\}$ is not regular and this is fundamentally because regular languages have no way of "remembering" how many 0's they saw previously to try and match this with the 1's.

Given this limitation we may want to introduce some form of memory to the machine. And this leads into the concept of context-free languages.

3.2 Context-Free Languages

Here we study two equivalent models:

- Context-Free Grammars (CFG)
  - A way to generate context-free languages
- Pushdown Automata (PDAs)
  - A way to recognise context-free languages

3.3 Context-Free Grammars

A context-free grammar is just a set of substitution rules.

Formally it is a 4-tuple,

$$G = (V, \Sigma, R, S)$$

where,

- V is a finite set of variables
- Σ is the alphabet (finite set of terminals)
- R is a finite set of rules where each rule is of the form $A \to w$ for some $A \in V$ and $w \in (V \cup \Sigma)^*$
- $S \in V$ is the start variable

Example: Here is a context-free grammar with start variable S that is defined as follows,

$$S \to 0S1 \mid R$$
$$R \to \varepsilon$$

Notice that "|" here means "or". This is just notational shorthand instead of defining two separate rules for $S \to 0S1$ and $S \to R$.

What language does the context-free grammar above represent? It turns out that it represents the language $\{0^n1^n : n \geq 0\}$, which is the language that we proved previously is not regular. So clearly context-free grammars are able to describe languages that are not regular.

Figure 4 shows a parse tree of the grammar above for the string 0011.

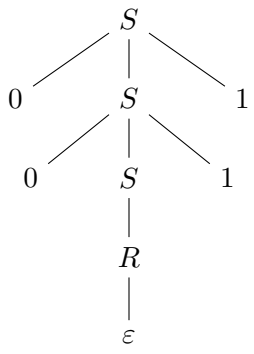


Figure 4: A parse tree for the string 0011 under the grammar $S \to 0S1 \mid R$, $R \to \varepsilon$.

3.3.1 More Terminology

We say that u yields v (denoted $u \Rightarrow v$)

- if u can become v by one substitution step

We say that u derives v (denoted $u \overset{*}{\Rightarrow} v$)

- if u can become v in zero or more steps

Given some context-free grammar G we define the language of G by,

$$L(G) = \{\omega \in \Sigma^* : S \overset{*}{\Rightarrow} \omega\}$$

(i.e. it is all strings of terminals that can be derived from the start variable S)

A language L is then said to be context-free if there is a context-free grammar G such that $L = L(G)$.

Also as a bonus note why do we call it a *context-free* grammar?

- A context-free grammar is one where every rule only has a single variable on the LHS
- This means that you can replace the variable regardless of what surrounds it (i.e. regardless of the context in which it appears)
- This is as opposed to grammars that are not context-free, where you might define a rule such as $B1 \to 1B$ (i.e. a rule where you can replace any B that is followed by a 1 with $1B$)

Convention: when deriving strings from a context-free grammar, always use the leftmost-first convention. That is, always replace the leftmost variable first.

3.4 Ambiguous Grammars

A grammar is **ambiguous** if some string can be generated by two or more different parse trees.

In general, to avoid ambiguity you need to use multiple variables that force a unique grouping. (This was only mentioned briefly in the lecture slides.)

3.5 Pushdown Automata (PDAs)

The idea is to extend the functionality of a DFA by giving it memory. We do this by giving the DFA access to a LIFO stack.

LIFO (Last-In First-Out stack):

- Push: add symbol to the top of the stack
- Pop: remove and read the top symbol
- Can only access the top (not elements further down)
- Unbounded (can grow as large as is needed)

(Technically in the formal definition of a LIFO stack there is no way to know if you are at the bottom of the stack and so the standard approach is to push a special "bottom-of-stack marker" at the start. Typically we push a \$ character.)

Within PDAs there are two types:

- DPDAs (deterministic PDAs)
- NPDAs (non-deterministic PDAs)

Recall that with regular languages we found that DFA = NFA. It turns out that this is *not* the case with pushdown automata. With pushdown automata DPDA $\subset$ NPDA. For example recognising the language of all palindromes requires NPDAs. This course is going to focus on NPDAs, so from here on, when you see PDA you can assume that it is an NPDA.

Formally a PDA is a 6-tuple,

$$(Q, \Sigma, \Gamma, \delta, q_0, F)$$

where:

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$
- $q_0 \in Q$
- $F \subseteq Q$

Focussing in on the transition function an example of a transition function might look like this,

$$\delta(a, b, c) = \{(r_1, d), (r_2, e)\}$$

This means that in state a upon receiving input b and popping a value of c from the stack we:

- Move to r_1 and push d **or**
- Move to r_2 and push e

PDAs are often represented as diagrams where in a diagram you have a transition arrow coming out of a state with the label,

$$a, b \to c$$

which means that if you read a and pop b then you should push c to the stack and then move to where the arrow points. Figure 5 shows an example.

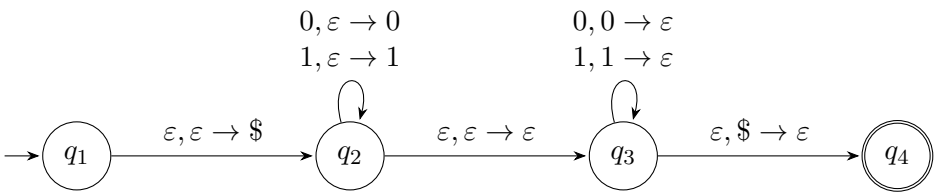


Figure 5: A PDA that accepts the even-length palindromes over $\{0, 1\}$. It pushes \$ to mark the bottom of the stack, pushes the first half of the input, guesses where the middle is, then pops the second half against the first.

Notice that popping ε simply means that we leave the stack unchanged (we don't take anything off the stack). Likewise pushing ε just means that we don't add anything new onto the stack.

A PDA is said to accept a string ω if there exists some path of computation that:

- starts in state q_0 with an empty stack
- reads all of ω
- ends in an accept state (a state in F)

(this is similar to how acceptance was defined in an NFA).

We then say that the language of a PDA P is,

$$L(P) = \{\omega \in \Sigma^* : P \text{ accepts } \omega\}$$

3.6 The Uniting Theorem: CFG = PDA

Theorem: a language is context-free if and only if some pushdown automaton recognises it.

To prove this we need to prove both directions:

- Given a CFG we need to be able to build a PDA (proof is covered)
- Given a PDA we need to be able to build a CFG (this proof is not covered in this course)

3.6.1 Proof: CFG $\implies$ PDA

The key idea is that given a CFG G, we build a PDA that uses its stack to simulate a leftmost-first derivation in G.

(here we take the stack alphabet Γ to be the union of the variables in the CFG and the terminals)

Algorithm:

- Push the start variable onto the stack.
- If the top is a variable replace it with the RHS (this can be done non-deterministically if there are multiple rules in the grammar)
- If the top is a terminal then pop it and match it against the next input symbol (reject at this branch if they disagree)
- If the stack is empty once all of the input has been read, then accept

3.7 Completing the picture

So we have proved that,

$$\text{CFG} \implies \text{PDA}$$

And it turns out that the other direction is also true (but the proof is more involved).

In other words it turns out that,

$$\text{CFG} = \text{PDA}$$

One other result that is quite trivial to prove is that,

$$\text{regular} \subset \text{context-free}$$

(with this being strict since, for example, $\{0^n 1^n : n \geq 0\}$ is context-free but not regular).

Looking ahead though, there are some languages that are not context-free either. For example,

$$L = \{a^n b^n c^n : n \geq 0\}$$

is a language that is **not** context-free (the proof isn't covered here).

This leads us to search for even more powerful models of computation.

4 Turing Machines

4.1 Limitations of Context-Free Languages

With DFAs (that define regular languages):

- no memory beyond the current state

With PDAs (that define context-free languages):

- access to an unbounded stack
- can only access the top of the stack

A logical extension is to consider machines that have a similar structure but with access to:

- unbounded memory
- ability to read & write anywhere in memory

This leads to the concept of a Turing Machine.

4.2 Turing Machines (TMs)

A Turing machine consists of:

- A finite state machine
- An infinite memory tape
- A read/write head that can move along the tape

(see Figure 6).

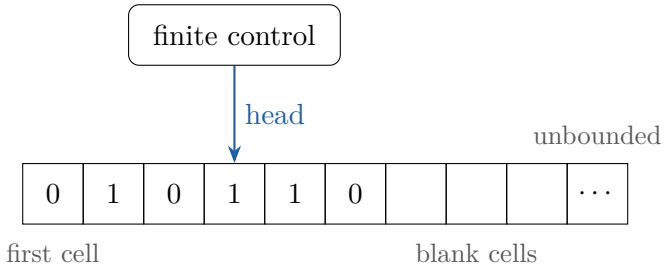


Figure 6: A Turing machine: a finite control, and a head that reads and writes an unbounded tape. The input (here 010110) starts at the left of the tape and the rest of the tape is blank.

Details:

- The tape extends infinitely to the right
  - in some models it extends infinitely in both directions; this doesn't really matter, but this course generally takes it to only extend to the right
- Each cell contains a symbol from the tape alphabet Γ
  - Often one takes $\Gamma = \Sigma \cup \{\sqcup\}$
  - Here $\sqcup$ denotes the blank symbol (a blank is an actual tape symbol, so it can't be ε, which is the empty string)
- Head will move left (L) or right (R) after reading a symbol
- Initially
  - Input string is placed on the tape (that is why $\Gamma \supset \Sigma$)
  - The rest of the tape is blank (i.e. $\sqcup$)
  - Head starts on the first symbol of the input (i.e. at the leftmost position)

The execution process is then as follows:

1. Read the current symbol under the head (i.e. read an input value)
2. Apply the transition function δ:
   - Write a new symbol onto the tape
   - Move to a new state in the state machine
   - Move the head left or right

3. Repeat 1 and 2 until
   - The machine enters q_{accept} (immediately halt and accept)
   - The machine enters q_{reject} (immediately halt and reject)
   - The machine never halts (infinite loop, sometimes called "reject by looping")

Notice that unlike the models that we have studied previously, TMs can accept/reject the input at **any time** (not needing to wait until the whole string is processed). In fact the other models of computation that we saw did not even have reject states: they would only reject if the computation ended on a state that was not accepting.

4.3 Formal Definition of a TM

Formally a Turing Machine is a 7-tuple,

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

Where,

- Q is a finite set of states
- Σ is the input alphabet (not including the blank symbol)
- Γ is the tape alphabet. Includes Σ, $\sqcup$ and possibly more
- $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$ (start state)
- $q_{\text{accept}} \in Q$ (accept state)
- $q_{\text{reject}} \in Q$ (reject state where $q_{\text{reject}} \neq q_{\text{accept}}$)

A few notes:

- The transition function essentially takes in a state + the value that it reads on the tape and then moves to a new state, writes a new value onto the tape and then moves the head left or right.
  - Notice that the transition function doesn't take in Σ: the input is given to the machine on the tape, so it is read from there.
- The accept and reject states are single states (not subsets of Q). This is because unlike DFAs a TM doesn't need a set of accepting states since whenever an accept state is reached the execution stops.

A TM can also be drawn as a diagram, as in Figure 7 (notice the convention used for the transition labels).

4.4 Common TM Design Patterns

There are some common patterns in designing Turing machines:

- Sometimes you may want to "cross" symbols off. You can do this by replacing them with some character X (that is not in Σ)
- Sometimes you may want to remember what the thing that you crossed off was. Say you had a 1 and you wanted to cross it off you could add the character $\not{1}$ to Γ to represent 1 when it is crossed off.
- Usually it's a good idea to initially verify if the input is in a valid form

4.5 TM Variations

There are many variations of the basic Turing machine:

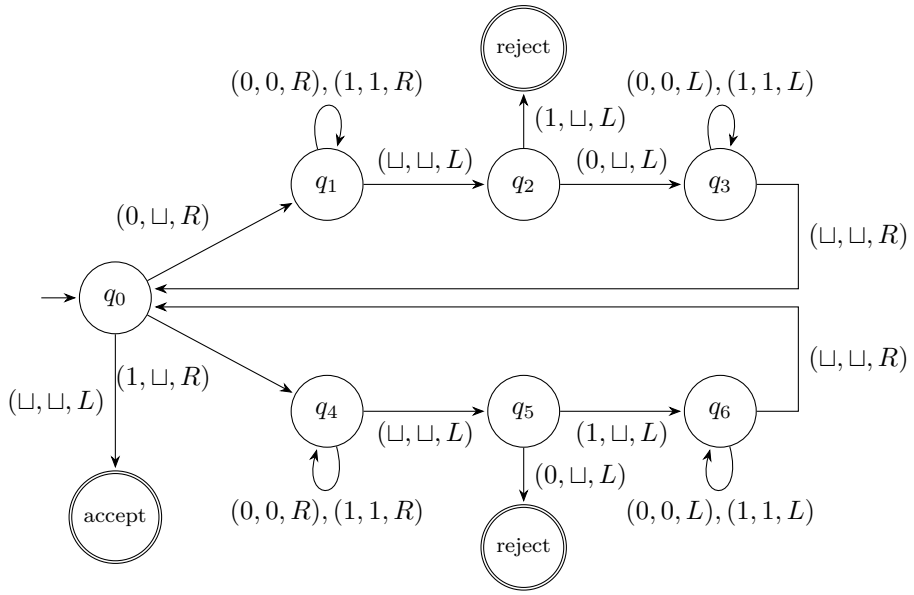


Figure 7: A Turing machine that accepts the even-length palindromes over $\{0,1\}$. It repeatedly blanks out the first symbol, runs to the end of the input and checks that the last symbol matches, then returns to the start. A label (a,b,D) means: read a, write b, then move the head in direction D. Missing transitions go to the reject state.

- Two-way infinite tape
- Multi-tape TM
- Non-deterministic TM

But it turns out that these are all equivalent in power to the standard TM. In fact the Church–Turing thesis states that any calculation that can be performed by an "effective model of computation" can be performed by a TM. It is generally taken to define a fundamental limit of computer science.

4.6 Definitions

A language $L \subseteq \Sigma^*$ is:

- **Turing-recognisable** if:
  - There exists a TM that halts and accepts for any $\omega \in L$ and either rejects or loops indefinitely for any $\omega \notin L$.
- **Turing-decidable** if:
  - There exists a TM that halts on *all* inputs and accepts if $\omega \in L$ and rejects if $\omega \notin L$.

It follows directly from the definitions above that the Turing-decidable languages form a subset of the Turing-recognisable languages (and we will see later that it is a proper subset).

In practice one often simply says "recognisable" or "decidable" instead of "Turing-recognisable" or "Turing-decidable".

4.7 Universal Turing Machine (UTM)

Recall that virtually anything can be encoded as a string (e.g. images, audio, text, etc). Turing machines themselves are no exception: you just need some way to represent the specific details of the TM as a string. In practice the specific encoding that you use is not really important.

Let $\langle M \rangle$ denote a string that is the encoding of some Turing Machine M. Intuitively you can think of this as the "source code" of the Turing Machine.

We also define $\langle M, \omega \rangle$ as the encoding of the TM M with some input ω (this is generally something really simple such as $\langle M, \omega \rangle = \langle M \rangle \, \#\omega$ where # is a special delimiter character used to identify when the encoding of the TM ends and when the input string starts)

It can be shown that there exists a TM $\mathcal{U}$ that takes as input:

1. An encoding of any TM M
2. any input string $\omega \in \Sigma^*$

And then simulates M on the string ω

- If M accepts ω then $\mathcal{U}$ will accept $\langle M, \omega \rangle$
- If M rejects ω then $\mathcal{U}$ will reject $\langle M, \omega \rangle$
- If M loops on ω then $\mathcal{U}$ will loop on $\langle M, \omega \rangle$

This TM $\mathcal{U}$ is called the **Universal TM (UTM)** and underpins the idea of programmable computers. This led to the invention of general-purpose computers: computers whose hardware is designed to do general computation, as opposed to performing a specific computational goal.

4.8 Church–Turing Thesis

As mentioned briefly above, the Church–Turing thesis states that:

> Any function that can be computed by an "algorithm" can be computed by a Turing Machine.

Notes:

- It is not a formal theorem but a widely held belief
- There are no known counter examples

There are several other models of computation that can be shown to be equivalent to TMs:

- λ-calculus
- all major programming languages

4.9 Limits of Computation

One question that follows naturally from this is to ask whether there are certain limits to what can be computed by an "algorithm". Or in other words, are there decision problems that a TM cannot recognise?

It turns out that there are problems that are not computable, and in fact the majority of problems are not computable (the set of computable problems has measure zero over the full set of decision problems).

Here is a counting argument to at least partially justify this claim:

- There are infinitely many TMs but there are also infinitely many decision problems
- It turns out that the set of all possible TMs is countable
- However the set of all decision problems is uncountable
- Thus,

$$\text{Number of TMs} \ll \text{Number of Decision Problems}$$

- And so we simply don't have enough TMs to solve all decision problems

4.10 Acceptance & Halting Problems

Acceptance problem:

- Language: $A_{TM} = \{\langle M, \omega\rangle : M \text{ is a TM and } M \text{ accepts } \omega\}$
- I.e. the problem is to determine if the provided TM accepts the provided string

Halting problem:

- Language $\text{HALT}_{TM} = \{\langle M, \omega\rangle : M \text{ is a TM and } M \text{ halts on } \omega\}$
- (halt here simply means it either accepts or rejects but doesn't loop)

For both of these problems we can show that:

- both are recognisable
- neither are decidable (i.e. both are undecidable)

4.10.1 Acceptance Problem

Proof of Recognisability Given the input $\langle M, \omega\rangle$ we build another TM, R that simulates M on ω:

- If M accepts then R accepts
- If M rejects then R rejects, and if M loops then R loops too

So R accepts exactly the strings in A_{TM}, which is all a recogniser needs to do (i.e. we could use a universal TM to recognise A_{TM}).

Proof of Undecidability Here we use a proof by contradiction. Suppose A_{TM} was decidable and let H be a TM that decides it. Now construct another TM D that takes in the source code of a TM and uses H to determine if the inputted machine accepts itself as input. If it does then D rejects or otherwise D accepts.

Formally,

$$H(\langle M, \omega\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } \omega \\ \text{reject} & \text{otherwise (if } M \text{ rejects or loops on } \omega) \end{cases}$$

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } H \text{ rejects } \langle M, \langle M\rangle\rangle \\ \text{reject} & \text{if } H \text{ accepts } \langle M, \langle M\rangle\rangle \end{cases}$$

When trying to run D on its own source code $\langle D\rangle$ you get a contradiction:

- If H accepts $\langle D, \langle D\rangle\rangle$ (i.e. D accepts $\langle D\rangle$) then D does the opposite and rejects $\langle D\rangle$ (contradiction)
- If H rejects $\langle D, \langle D\rangle\rangle$ (i.e. D doesn't accept $\langle D\rangle$) then D does the opposite and accepts $\langle D\rangle$ (also a contradiction)

4.10.2 Halting Problem

Proof of Recognisability Very similar to the acceptance problem, we can use a UTM to recognise the halting problem (simulate M on ω and accept if it ever halts).

Proof of Undecidability This is also similar to the acceptance problem. Suppose for contradiction that the halting problem were decidable, and let H denote the TM that decides it. Now

construct a TM D that takes as input a TM and uses H to determine whether the provided machine halts on its own source code. If it does then D loops, and otherwise D halts. Formally,

$$H(\langle M, \omega \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ halts on } \omega \\ \text{reject} & \text{otherwise (if } M \text{ loops on } \omega) \end{cases}$$

$$D(\langle M \rangle) = \begin{cases} \text{halt} & \text{if } H \text{ rejects } \langle M, \langle M \rangle \rangle \\ \text{loop} & \text{if } H \text{ accepts } \langle M, \langle M \rangle \rangle \end{cases}$$

Now if you consider running D on its own source code you again get a contradiction: D halts on $\langle D \rangle$ exactly when H says that it doesn't.

4.11 Reduction & Reducibility

Problem A is reducible to problem B if:

- an algorithm for solving problem B can be used as a subroutine in an algorithm for solving problem A

Informally this implies that problem A cannot be "harder" than problem B

$$\text{Problem A} \leq \text{Problem B}$$

That is because if we can solve problem B then we automatically have an easy way to solve problem A.

So if we can show that problem A is "very hard" (i.e. maybe it is undecidable) then so is problem B.

4.11.1 Application to the Undecidability of the Halting Problem

Suppose the halting problem was decidable and let H be its decider. We can then use H as a subroutine to decide the acceptance problem.

Construct a TM D that takes in a TM and an input ω.

- D calls H on that provided TM and input
- If H rejects then D rejects
- If H accepts then D simulates the TM on the input and accepts/rejects accordingly

Such a machine D is a decider for the acceptance problem, which is a contradiction. (In the notation above, this shows that $A_{TM} \leq \text{HALT}_{TM}$.)

4.12 Decidability and Recognisability

Theorem: L is decidable if and only if both L and its complement are recognisable.

(The proof was an assignment question.)

5 Cellular Automata

A cellular automaton is an alternative model of computation:

- A grid of cells each holding a value from a (typically) small alphabet
- no central controller; each cell only sees its "neighbours"
- every cell updates in parallel by the same local rule

Formally, a cellular automaton is a tuple (L, Σ, N, δ):

- L is the cellular space. Typically something like $\mathbb{Z}$ or $\mathbb{Z}^2$
- Σ a finite set of all the values that a cell can take
- N the neighbourhood determining which cells influence a given cell's future value
- $\delta : \Sigma^{|N|} \to \Sigma$ the local rule given all of the cells in the neighbourhood

A configuration $c : L \to \Sigma$ assigns a state to every cell. The global rule Δ maps configurations to configurations by applying δ at every cell in parallel.

Notes:

- on a finite lattice Δ is ambiguous at the edges, so a boundary convention is needed (such as zero-padding, cyclic or reflecting boundaries)
- common neighbourhood choices in 1D include all cells within a radius r of the given cell
- in 2D there are two standard neighbourhoods: Moore (the 3×3 block of 9 cells around a cell) and von Neumann (the same but excluding the diagonals)

5.1 Wolfram Number

Consider the simplest non-trivial setting:

- $L = \mathbb{Z}$
- $\Sigma = \{0, 1\}$
- $N = \{-1, 0, +1\}$

For the local rule $\delta : \{0, 1\}^3 \to \{0, 1\}$ there are $2^3 = 8$ possible inputs. For each of those inputs there are two possible outputs and so the total number of possible rules is,

$$2^8 = 256$$

(which is a small enough number that you can enumerate all of the rules)

Each of the 256 rules can be associated with an 8-bit binary integer. This is done by listing the rule's output for each of the 8 neighbourhoods in a fixed order (from 111 down to 000) and reading the outputs as a binary number, as in Figure 8. By convention black cells are 1 ("on") and white cells are 0 ("off").

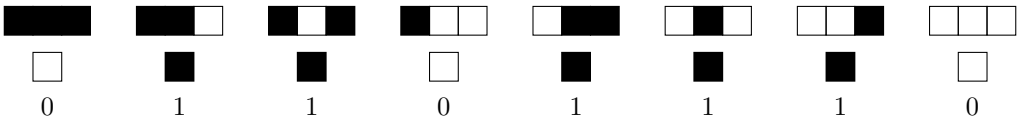


Figure 8: The Wolfram number of a rule. The top row shows the 8 possible neighbourhoods and below each is the new value of the centre cell. For the rule shown, reading the new values as a binary number gives $01101110_2 = 110$, so this is Rule 110.

This number, from 0 to 255, is known as the Wolfram number.

5.2 Further Topics

The rest of the lecture covers a few topics, which aren't written up here:

- The four classes of "behaviour" that you get for the various Wolfram numbers (homogeneous, periodic, chaotic, complex)
- Cook's theorem that Rule 110 is Turing complete

- Using this Turing completeness to justify why certain problems involving Rule 110 are undecidable
- Conway's Game of Life (and that it is Turing complete)
- Langton's λ (a number that determines how "chaotic" a binary cellular automaton is)
- Connections to biology and research in neural networks

6 Complexity Theory

Here we classify decidable problems based on the resources they require (time, space, etc).

6.1 Measuring Time Complexity

We typically use a "worst-case analysis".

For example on a TM let $t(\omega)$ be the number of steps that the given TM takes on the input ω. Then the worst-case running time for inputs of length n is,

$$f(n) = \max\{t(\omega) : |\omega| = n\}$$

(the max number of steps that the TM will take on an input of size n)

6.2 Asymptotic Time Complexity

The exact step counts are usually hard to calculate and they also depend on several factors:

- the model of computation
- implementation details

In practice we are more interested in how the time-complexity grows as the input size grows. This is encapsulated by the idea of Big-$\mathcal{O}$ notation.

We write $f(n) = \mathcal{O}(g(n))$ if there exists a constant $c > 0$ and $n_0 > 0$ such that,

$$f(n) \leq c \cdot g(n) \qquad \forall n \geq n_0$$

(i.e. the asymptotic growth of $f(n)$ is bounded by growth proportional to $g(n)$)

6.3 Efficient $\coloneqq$ Polynomial Time

There are a few reasons why we generally "like" computation that runs in polynomial time:

- grow much slower than other functions such as exponentials
- polynomial time + polynomial time = polynomial time
- repeating polynomial time = polynomial time

In complexity theory a program that runs in polynomial time is said to be efficient.

Note: a time complexity of $\mathcal{O}(\log n)$ for example is considered polynomial time since it is bounded by the polynomial time $\mathcal{O}(n)$. So just because we say "polynomial time" doesn't mean that the actual function needs to be a polynomial.

6.4 Time Complexity Class $\mathrm{TIME}(f(n))$

For a function $f : \mathbb{N} \to \mathbb{N}$,

$$\mathrm{TIME}(f(n)) = \{L : \text{deterministic TM decides } L \text{ in time } \mathcal{O}(f(n))\}$$

(i.e. it is the set of all languages that can be decided in time of the order of $f(n)$).

For example it was shown in the lectures that,

$$\{0^k 1^k : k \geq 0\} \in \text{TIME}(n^2) \qquad \text{(where } n \text{ is the input length)}$$

Notice that these classes form a nested hierarchy,

$$\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \cdots \subseteq \text{TIME}(n^{100}) \subseteq \ldots$$

One other thing to note is that the specific complexity class depends on your model of computation. For example in the problem above that came from the lectures you can show that if you use a two-tape TM then the language is in TIME(n). But the key point is that this dependence is at most polynomial.

- we may only get polynomial slowdowns/speedups by switching between reasonable deterministic models of computation.

This is why studying problems that run "efficiently" (i.e. in polynomial time) is so important.

6.5 The Class P

P is the class of languages decidable by a deterministic TM in polynomial time,

$$\text{P} = \bigcup_{k \geq 0} \text{TIME}(n^k)$$

(unlike TIME($f(n)$) the class P does not depend on a particular model of computation).

We think of P as the class of "efficiently" solvable decision problems.

Fact:

- all regular languages are in P (a DFA reads its input once, so it runs in linear time)

6.6 Polynomial-Time Verifiers

A verifier for a language L is a deterministic TM V such that,

$$\omega \in L \iff \text{there exists a string } c \text{ such that } V \text{ accepts } \langle \omega, c \rangle$$

c here is known as a certificate of proof of membership in L and a verifier V simply checks that the provided certificate of proof is valid.

For example consider the SAT problem (Boolean satisfiability). For a given Boolean expression ω, a certificate of proof c could contain a specific configuration of TRUE / FALSE values that satisfies the expression. A verifier could then be used to check whether the provided configuration does satisfy the Boolean expression.

A poly-time verifier for L is a verifier such that:

- $|c| \leq p(|\omega|)$ for some polynomial p
- V runs in polynomial time in $|\omega|$

6.7 Non-deterministic Turing Machines (NTM)

Similar to the other non-deterministic models of computation that we have seen:

- computation can branch into many possible paths (this includes the tape branching not just the state)
- the machine accepts if some branch accepts
- the machine rejects if all branches reject
- an NTM is a decider if *all* branches halt

We define,

$$\text{NTIME}(f(n)) = \{L : \text{some NTM decides } L \text{ in time } \mathcal{O}(f(n))\}$$

(i.e. the non-deterministic analogue of TIME).

6.8 The Class NP

NP is the class of all languages decidable in polynomial time by a non-deterministic Turing machine.

$$\text{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k)$$

It can be shown that this class is equivalent to the class of languages that have a polynomial-time verifier. (Intuitively, an NTM can guess a certificate and then run the verifier on it.)

6.9 P vs NP

We know that,

$$\text{P} \subseteq \text{NP}$$

(since a deterministic TM is just an NTM that never branches).

However we don't know whether P = NP or P $\neq$ NP. No polynomial-time algorithm is known for many problems in NP, and so it is suspected that P $\neq$ NP, although this has not been proved.

6.10 Polynomial Time Reducibility

We say language A is poly-time reducible to language B,

$$A \leq_p B$$

if there exists a poly-time computable function f such that,

$$x \in A \iff f(x) \in B$$

for every input x.

This implies that solving A cannot be harder than B, since if you can solve B in poly-time then you automatically have a way to solve A in poly-time.

6.11 NP-Hard and NP-Complete

The Cook–Levin theorem states that,

$$L \leq_p \text{SAT} \quad \forall L \in \text{NP}$$

In other words any problem in NP can be reduced to SAT. Or in other words if one can find an algorithm for solving SAT in polynomial time then we can solve every problem in NP in polynomial time.

SAT was the first problem to be found that satisfies this, but it is not the only one. We define the class of NP-hard problems as all problems such that,

$$B \text{ is NP-hard if } \quad \forall A \in \text{NP} \quad A \leq_p B$$

Furthermore, a language B is NP-complete if:

- $B \in \text{NP}$
- B is NP-hard

6.12 How to Prove that a Language is NP-Complete

From the Cook–Levin theorem we know that SAT is NP-complete, and thousands of NP-complete problems are now known (SAT, HAMPATH, CLIQUE, ...). To prove that a language B is NP-complete:

1. show that $B \in \text{NP}$
2. choose a known NP-complete problem A (e.g. SAT, HAMPATH)
3. prove that $A \leq_p B$

This works because $\leq_p$ is transitive: every language in NP reduces to A, and A reduces to B. The direction of $\leq_p$ matters (the known problem is reduced to B, not the other way round), and it is an easy mistake to make.

6.13 EXPTIME

$$\text{EXPTIME} = \bigcup_{k \geq 1} \text{TIME}(2^{n^k})$$