Turing Machines

Task: Take a more realistic look at the model of a computer than a finite state acceptor.

Turing machines were first proposed by Alan Turing in 1936 in order to explore the theoretical limits of computation. We shall see that certain problems cannot be solved even by a Turing machine and are thus beyond the limits of computation.

A Turing machine is similar to a finite state acceptor but has **unlimited memory** given by an **infinite tape** (countably infinite). The tape is divided into **cells** each of which contains a character of a **tape alphabet**.

The Turing machine is equipped with a **tape head** that can read and write symbols on the tape and move left (back) or right (forward) on the tape. Initially, the tape contains only the input string and is blank everywhere else. To store information, the Turing machine can write this information on the tape. To read information that it has written, the Turing machine can move its head back over it.

The Turing machine continues computing until it decides to produce an output. The outputs “**accepts**” and “**rejects**” are obtained by entering accepting or rejecting states respectively. It is also possible for the Turing machine to go on forever if it does not enter either an accepting nor rejecting state.

[0][1][0][_][_][...   The **blank symbol** (_) is part of the tape alphabet.

Example: Let A = {0, 1} and L = $\{0^m 1^m \mid m \in N, m \geq 1\}$
We know L is not a regular language, so there is no finite state acceptor that can recognise it, but there is a Turing machine that can.

Initial State of the Tape:
Input string of 0s and 1s, then infinitely many blanks.

Idea of this Turing machine:
Change a 0 to an X, and a 1 to a Y until either:

a) All 0s and 1s have been matched - ACCEPT
b) The 0s and 1s do **not** match or the string does not have the form 0*1* - REJECT

Algorithm:
The tape head is initially positioned over the first cell.

1. If anything other than 0 is in the first cell - REJECT
2. If 0 is in the cell, then change 0 to X.
3. Move right to the first 1. If none - REJECT
4. Change 1 to Y.
5. Move left to the leftmost 0. If none, move right looking for either a 0 or 1.
   If either a 0 or 1 is found before the first blank symbol - REJECT
   Otherwise - ACCEPT
6. Go to STEP 2.

Examples of processing strings: (Tape Head)

0011_	001_	011_	010_
X011_	X01_	X11_	X10_
X011_	X01_	X11_	X10_
X0Y1_	X0Y_	XY1_	XY0_
X0Y1_	X0Y_	XY1_	XY0_
XXY1_	XXY_	XY1_	XY0_
XXY1_	XXY_	REJECT (STEP 5)	REJECT (STEP 5)
XXYY_	REJECT (STEP 3)		
XXYY_			
XXYY_			
ACCEPT (STEP 5)			

Note that we have the following:

- A = {0, 1} is the **input alphabet**.
- _ ∉ A, where _ is the **blank symbol**.
- Ã = {0, 1, X, Y, _) is the **tape alphabet**.
- S is the set of **states**.

Note that the tape head is moving right or left, so we also need to have a set {L, R} with L for left and R for right for specifying where the tape head goes.

Recall that a finite state acceptor is given by (S, A, i, t, F) where:

- S = Set of States
- A = Alphabet
- i = Initial State
- t = Transition Mapping
- F = State of Finishing States

The transition mapping is given by $t : S \times A \rightarrow S$

By contrast, a Turing machine's transition mapping is of the form $t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$

- $\tilde{A}$ = Indicates what the Turing machine can write
- {L, R} = Indicates the Turing machine can move left or right

Definition: A Turing machine is a 7-tuple $(S, A, \tilde{A}, t, i, S_{acc}, S_{rej})$ where S, A, $\tilde{A}$ are finite sets.

- S = Set of States
- A = Input Alphabet **not** containing the blank symbol _
- $\tilde{A}$ = Tape Alphabet where $\_ \in \tilde{A}$ and $A \subseteq \tilde{A}$
- t = Transition Mapping
- i = Initial State
- S_{acc} = Accept State
- S_{rej} = Reject State

Remarks:

1. Since A does not contain the blank symbol _, the first blank on the tape marks the end of the input string.
2. If the Turing machine is instructed to move left and it has reached the first cell of the tape, then it stays at the first cell.
3. The Turing machine continues to compute until it enters either the accept or reject states, at which point it halts. If it does not enter either state, it goes on forever.

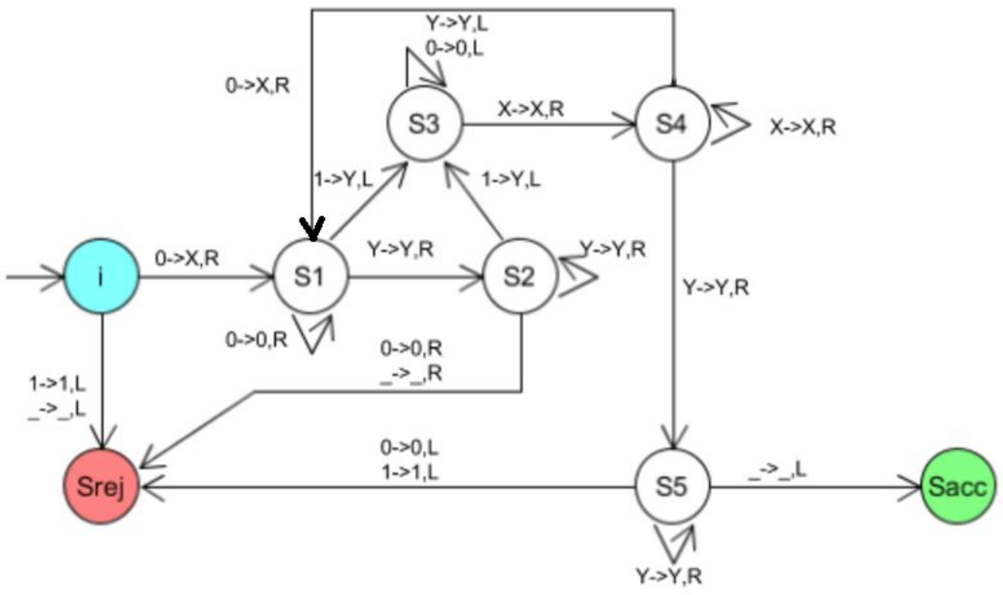
Example: (Same example as previous)
$A = \{0, 1\}$ and $L = \{0^m1^m \mid m \in N, m \geq 1\}$

We need to be able to write down the transition mapping hence the set of states S. Recall that what we gave was an algorithm, and using that algorithm we processed strings to convince ourselves that the corresponding Turing machine behaved correctly.

Algorithm:
The tape head is initially positioned over the first cell.

1. If anything other than 0 is in the first cell - REJECT
2. If 0 is in the cell, then change 0 to X.
3. Move right to the first 1. If none - REJECT
4. Change 1 to Y.
5. Move left to the leftmost 0. If none, move right looking for either a 0 or 1.
   If either a 0 or 1 is found before the first blank symbol - REJECT
   Otherwise - ACCEPT
6. Go to STEP 2.

Before we can write down the set of states S or the transition mapping t, let us draw a **transition diagram** which is the Turing machine equivalent of drawing a finite state acceptor.



- $i \to S_{rej}$ represents step 1 of the algorithm.
- $i \to S_1$ and $S_4 \to S_1$ represent step 2 of the algorithm.
  ($i \to S_1$ at the first pass through the string, $S_4 \to S_1$ at subsequent passes)
- $S_1 \to S_1$, $S_1 \to S_2$ and $S_2 \to S_2$ represent the first part of step 3.
- $S_2 \to S_{rej}$ represents the second part of step 3.
- $S_1 \to S_3$ represents step 4.
- $S_3 \to S_3$ and $S_3 \to S_4$ represent the first sentence in step 5.
- $S_4 \to S_4$ and $S_4 \to S_5$, $S_5 \to S_5$ represent the second sentence in step 5.
- $S_5 \to S_{rej}$ represents the first half of the third sentence in step 5.
- $S_5 \to S_{acc}$ represents the second half of the third sentence in step 5.
- $S_4 \to S_1$ represents the step 6.

We have accounted for all pieces of our algorithm, therefore we have written down a Turing machine when A = {0, 1}, Ã = {0, 1, X, Y, _ }

$S = \{i, S_{acc}, S_{rej}, S_1, S_2, S_3, S_4, S_5\}$
- i = Initial State
- $S_{acc} \in S$ = Accept State
- $S_{rej} \in S$ = Reject State

We just have to write the **transition mapping** $t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$

1. $t(i, 0) = (S_1, X, R)$ If in initial state and read 0, write X & move right to state 1
2. $t(i, 1) = (S_{rej}, 1, L)$
3. $t(i, \_) = (S_{rej}, \_, L)$

These are the only 3 transitions possible out of state i, but $t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$ so technically, to write down the full transition mapping we must assign triplets in $S \times \tilde{A} \times \{L, R\}$ even to inputs from $\tilde{A}$ that cannot occur in i.

4. $t(i, X) = (S_{rej}, X, L)$
5. $t(i, Y) = (S_{rej}, Y, L)$

We assign S_{rej} some element of $\tilde{A}$ and one of the allowable tapehead distinctions.

Technically, the Turing machine halts when it enters either an accepting or rejecting state, so in practice we can define

$$\check{S} = \{i, S_1, S_2, S_3, S_4, S_5\} = S \setminus \{S_{acc}, S_{rej}\} \text{ (Set of resulting states)}$$

and $t : \check{S} \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$, so we avoid writing down the transitions from S_{acc} and S_{rej}. We only have states S_1, S_2, S_3, S_4 and S_5 left.

6. $t(S_1, 0) = (S_1, 0, R)$
7. $t(S_1, Y) = (S_2, Y, R)$ On the diagram
8. $t(S_1, Y) = (S_3, Y, L)$

9. $t(S_1, X) = (S_{rej}, X, R)$ Not on the diagram - cannot occur, so added for completeness
10. $t(S_1, \_) = (S_{rej}, \_, R)$

11. $t(S_2, Y) = (S_2, Y, R)$
12. $t(S_2, 1) = (S_3, Y, L)$ On the diagram - can occur
13. $t(S_2, 0) = (S_{rej}, 0, R)$
14. $t(S_2, \_) = (S_{rej}, \_, R)$

15. $t(S_2, X) = (S_{rej}, X, R)$ Not on the diagram - cannot occur, so added for completeness

16. $t(S_3, Y) = (S_3, Y, L)$
17. $t(S_3, 0) = (S_3, 0, L)$ On the diagram - can occur
18. $t(S_3, X) = (S_4, X, R)$

19. $t(S_3, \_) = (S_{rej}, \_, R)$ Not on the diagram - cannot occur, so added for completeness
20. $t(S_3, 1) = (S_{rej}, 1, R)$

21. $t(S_4, X) = (S_4, X, R)$
22. $t(S_4, Y) = (S_5, Y, R)$ On the diagram - can occur
23. $t(S_4, 0) = (S_1, X, R)$

24. $t(S_4, 1) = (S_{rej}, 1, R)$ Not on the diagram - cannot occur, so added for completeness
25. $t(S_4, \_) = (S_{rej}, \_, R)$

26. $t(S_5, Y) = (S_5, Y, R)$
27. $t(S_5, \_) = (S_{acc}, \_, L)$ On the diagram - can occur
28. $t(S_5, 0) = (S_{rej}, 0, L)$
29. $t(S_5, 1) = (S_{rej}, 1, L)$

30. $t(S_5, X) = (S_{rej}, X, L)$ Not on the diagram - cannot occur, so added for completeness

Moral of the Story:
The transition mapping is a very inefficient way of specifying a Turing machine as a lot of transitions cannot occur unlike what we saw for a finite state acceptor, where the input alphabet was exactly the alphabet of the language.
Here $A \subset \tilde{A}$. Therefore we will specify a Turing machine via either an algorithm or the transition diagram **only**.

To figure out which languages are recognised by a Turing machine we need to introduce the notion of a confirmation. As a Turing machine goes through its computations, changes take place in:

1. The state of the machine
2. The tape contents
3. The tape head location

A setting of these three items is called a **configuration**.

Configurations

Representing Configurations:
We represent a configuration as U S_i V, where the U, V are strings in the tape alphabet Ã and S_i is the current state of the machine. The tape contents are then the string UV and the current location of the tape head is on the first symbol of V. The assumption here is that the tape contains only blanks after the last symbol in V.

Example: εi001 is the configuration [0][0][1][_][… Tape Head
as we start examining the string 001 in our previous example of a Turing machine.

Definition:
Let C_1, C_2 be two configurations of a given Turing machine. We say that the configuration C_1 **yields** the configuration C_2 if the Turing machine can go from C_1 to C_2 in one step.

Example: If s_i,s_j are states, u and v are strings in the tape alphabet Ã, and a,b,c ∈ Ã.

A configuration C_1 = u$s_i$bv yields a configuration C_2 = u$s_j$acv if the transition mapping t specifies a transition t(s_i, b) = (s_j,c, L).
In other words, the Turing machine is in the state s_i, it reads character b, writes character c in its place, enters state s_j, and its head moves left.

Types of Configurations:
1. **Initial Configuration** with input u is iu, which indicates that the machine is in the initial state i with its head at the leftmost position on the tape (which is the reason why this configuration has no strings left of the state).
2. **Accepting Configuration** us_{acc}v for u,v ∈ Ã* (u,v string in Ã), namely the machine in the accept state.
3. **Rejecting Configuration** us_{rej}v for u,v ∈ Ã*, namely the machine in the reject state.
4. **Halting Configurations** yield no further configurations - no transitions are defined out of their states. Accepting and rejecting configurations are examples of halting configurations.

Definition:
A Turing machine M accepts input w ∈ A* (string over the input alphabet A) if ∃ a sequence of configurations C_1, C_2, …, C_k such that:
1. C_1 is the start configuration with input w.
2. Each C_i yields C_{i+1} for i = 1, ... , k-1.
3. C_k is an accepting configuration,

Definition:
Let M be a Turing machine. L(M) = {w ∈ A* | M accepts w} is the language recognised by M.

Definition:
A language $L \subset A^*$ is called **Turing-recognisable** if $\exists$ a Turing machine that recognises L, i.e. $L = L(M)$.

NB: Some textbooks use the terminology **recursively enumerable language** (RE language) instead of Turing-recognisable.

Turing recognisable is not necessarily as strong a notion as we might need because a Turing machine can:

1. Accept
2. Reject
3. Loop

Looping is any simple or complex behaviour that does not lead to a halting state. The problem with looping is that the user does not have infinite time. It can be difficult to distinguish between looping or taking a very long time to compute. We thus prefer deciders.

Definition:
A **decider** is a Turing machine that enters either an accept state or a reject state for every input in A^*.

Definition:
A decider that recognises some language $L \subset A^*$ is said to **decide** that language.

Definition:
A language $L \subset A^*$ is called **Turing-decidable** if $\exists$ a Turing machine M that decides L.

NB: Some textbooks use the terminology **recursive language** instead of Turing-decidable.

Example:
$L = \{0^m 1^m \mid m \in N, m \geq 1\}$ is Turing-decidable because the Turing machine we built that recognises it was in fact a decider. (check again to convince yourself that the machine did not loop.)
Turing-decidable $\Rightarrow$ Turing-recognisable, but the converse is not true.
Turing-recognisable $\nRightarrow$ Turing-decidable.
We will hopefully have time to cover an example of a language that is Turing-recognisable, but **not** Turing-decidable before the end of the term.

Variants of Turing Machines

Task: Explore variants of the original setup of a Turing machine and show they do not enlarge the set of Turing-recognisable languages.

A) Add "stay put" to the list of allowable directions
Say instead of allowing just {L, R} (The tape head moves left or right) we also allow the "stay put" option (no change in the position of the tape head).
Thus, the transition mapping is defined as $t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R, N\}$ where N is for "no movement" or "stay put" instead of $t : S \times \tilde{A} \rightarrow S \times \tilde{A} \times \{L, R\}$.
We realise N is the same as R+L or L+R (move the tape head left then right and vice versa)
⇒ variant A) yields no increase in computational power

B) Multiple Turing machines
We allow the Turing machine to have several tapes, each with its own tape head for reading and writing. Initially, the input is on tape 1, and the others are blank. The transition mapping then must allow for reading, writing and moving the tape head on some or all of the tapes simultaneously. If k is the number of tapes, then the transition mapping is defined as
$t : S \times \tilde{A}^k \rightarrow S \times \tilde{A}^k \times \{L, R, N\}^k$ where:

- $\tilde{A}^k$ = k-fold Cartesian product, i.e. $(\tilde{A} \times \tilde{A} \times \dots \times \tilde{A})$ k times
- $\{L, R, N\}^k$ = k-fold Cartesian product, i.e. $(\{L, R, N\} \times \{L, R, N\} \times \dots \times \{L, R, N\})$ k times

Since one of the tape heads or more might not move for some transitions, we make use of the option N ("no movement") besides left and right.
Multiple Turing machines seem more powerful than ordinary simple-tape ones, but that is **not** the case.

Definition:
We call the two Turing machines M_1 and M_2 **equivalent** if $L(M_1) = L(M_2)$, namely if they recognise the same language.

Theorem: Every multi-tape Turing machine has an equivalent single-tape Turing machine.

Sketch of Proof:
Let M^k be a Turing machine with k tapes. We will simulate it with a simple-tape Turing machine M^1 constructed as follows:
We add # to the tape alphabet $\tilde{A}$ and use it to separate the contents of the different tapes.
M^1 also needs to keep track of the locations of the tape heads of M^k. It does so by adding a dot to the character to which a tape head is pointing. We thus only need to enlarge the tape alphabet $\tilde{A}$ by allowing a version with a dot above for every character in $\tilde{A}$ apart from # and the blank symbol _.

Corollary:
A language L is Turing-recognisable ⇔ some multi-tape Turing machine recognises L.

Proof:
"⇒" A language L is Turing-recognisable if ∃ M a single-tape Turing machine that recognises it. A single-tape Turing machine is a special type of a multi-tape Turing machine, so we are done.
"⇐" follows from the previous theorem. (*q.e.d*)

C) A nondeterministic Turing machine
Just like a nondeterministic finite state acceptor, a nondeterministic Turing machine may proceed according to different possibilities, so its computation is a tree, where each branch corresponds to a different possibility. The transition mapping of such a nondeterministic Turing machine is given by t : S x $\tilde{A}^k \rightarrow$ P(S x $\tilde{A}^k$ x {L, R}) where:

- P shows we have different possibilities on how to processed.

Theorem:
Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Idea of the Proof:
We construct a deterministic Turing machine that simulates the nondeterministic one by trying out all possible branches. If it finds an accept state on one of these computational branches, it accepts the input, otherwise it loops.

Corollary:
A language L is Turing-recognisable ⇔ some nondeterministic Turing machine recognises L.

Proof:
"⇒" A deterministic Turing machine is a nondeterministic one, so this direction is obvious.
"⇐" follows from the previous theorem.

D) Enumerators
As we saw, a Turing-recognisable language is called in some textbooks a recursively enumerable language. The term comes from a variant of a Turing machine called an **enumerator**.
Loosely, an enumerator is a Turing machine with an attached printer. The enumerator prints out the language L it accepts as a sequence of strings. Note that the enumerator can print out the strings of the language in any order and possibly with repetitions.

Theorem:
A language L is Turing-recognisable ⇔ some enumerator enumerates (outputs) L.

Proof:
"⇐" Let E be the enumerator. We construct the following Turing machine M:
M = on input w

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, accept w.

Thus, M accepts exactly those strings that appear on E's list and no others, hence exactly L.

"⇒" Let M be a Turing machine that recognises L. We would like to construct an enumerator E that outputs L. Let A be the alphabet of L, i.e. $L \subset A^*$.
In the unit on countability, we proved A* is countably infinite (note that the alphabet A is always assumed to be finite), so A* has an enumeration as a sequence $A^* = \{w_1, w_2, \dots\}$
E = Ignore the input

1. Repeat the following for i = 1, 2, 3, …
2. Run M for i steps on each input $w_1, w_2, \dots, w_i$
3. If any computations accept, print out the corresponding w_j.

Every string accepted by M will eventually appear on the list of E, and once it does, it will appear infinitely many times because M runs from the beginning on each string for each repetition of step 1.
Note that each string accepted by M is accepted in some finite number of steps, say k steps, so this string will be printed on E's list for every $i \geq k$. (*q.e.d*)

Moral of the Story:
The single-tape Turing machine we first introduced is as powerful as any variants we can think of.

Algorithms

Task:
Use Hilbert's 10th problem to give an example of something that is Turing-recognisable but not Turing-decidable.
We saw that the continuum hypothesis of Cantor was the 1st of Hilbert's 23 problems in 1900 at the International Congress of Mathematicians.

Hilbert's 10th Problem:
Find a procedure that tests whether a polynomial in 2 variables (x and y) with integer coefficients (2, -1, -1) that has integer roots $p(1, 1) = 2.1^2 - 1.1 - 1^2 = 0$ so $x = 1 = y$, $1 \in Z$ is a solution.
Hilbert's problem asked how to find integer roots via a set procedure.
In 1936 independently Alonzo Church invented λ-calculus to define algorithms, while Alan Turing invented Turing machines. Church's definition was shown to be equivalent to Turing's. This equivalence says:

- Intuitive notion of algorithms = Turing machine algorithms

and is known as the **Church-Turing thesis**. It led to the formal definition of an algorithm and eventually to resolving in the negative Hilbert's 10th problem. Using previous work by Martin Davis, Hilary Putnam and Julia Robinson, Yuri Matijasevic proved in 1970 that there is no algorithm which can decided whether a polynomial has integer roots.
As we shall see now, Hilbert's 10th problem is an example of a problem that is Turing-recognisable but **not** Turing-decidable.

Let D = {p | p is a polynomial with an integer root}. Hilbert's 10th problem is asking whether D is decidable.
Let us simplify the problem to the one variable case:

D_1 = {p | p is a polynomial in variable X with an integer root}.

We can easily write down a Turing machine M_1 that recognises D_1:
M_1 = on input p, where p is a polynomial in X.

1. Evaluate p with X not successively to R values 0, 1, -1, 2, -1, …
   If at any value the polynomial evaluates to 0, accept.

If p does indeed have an integer root, M_1 will eventually find it and accept p. If p does not have an integer root, then M_1 will run forever.

Principle behind M_1:
Z ~ N; i.e Z is countably infinite, so we can write Z as a sequence (enumerate it) .
$Z = \{S_1, S_2, \dots\} = \{S_i\}$ i = 1, 2, 3… = {0, 1, -1, 2, -2, ...}
Now, consider polynomials of n variables $p(x_1, \dots, x_n)$. We want to find $(x_1, \dots, x_n) \in Z^n$ such that $p(x_1, \dots, x_n) = 0$, so in general Hilbert's 10th problem is asking us to build a decider for $D_n = \{p(x_1, \dots, x_n) \mid \exists\ (x_1, \dots, x_n) \in Z^n \text{ such that } p(x_1, \dots, x_n) = 0\}$.

We can easily build a Turing machine M_n that recognises D_n using the principles behind M_1 : Z^n is countably infinite because it is the Cartesian product of a countably infinite set with itself n times. Since Z^n is countably infinite, we can enumerate it, namely write it as a sequence $Z^n = \{c_1, c_2, \dots\}$ where $c_i = (x_1^{(i)}, \dots, x_n^{(i)})$.
Then M_n = an input p, where p is a polynomial in $x_1, \dots, x_n$

1. Evaluate p with $(x_1, \dots, x_n)$ set successively to the values $c_1, c_2 \dots$
   If at any value $c_i = (x_1^{(i)}, \dots, x_n^{(i)})$, $p(x_1^{(i)}, \dots, x_n^{(i)}) = 0$, accept p.

If p has an integer root $(x_1^{(i)}, \dots, x_n^{(i)}) \in Z^n$ then the Turing machine accepts, otherwise it loops forever just like M_1. It turns out M_1 can be converted into a decider because if p(x) of one variable has a root, then that root must fall between certain bounds, so the checking of possible values can be made to terminate when those bounds are reached.
By contrast, no such bounds exist when the polynomial is of two variables or more $\Rightarrow M_n$ for $n \geq 2$ **cannot** be converted into a decider. This is what Matijasevic proved.

Decidable Languages

Task:
Explore whether certain languages are decidable that come from our study of formal languages and grammars.

1) The acceptance problem for deterministic finite state acceptors (DFAs)
Test whether a given deterministic finite state acceptor B accepts a given string w.
We can write the acceptance problem as a language:
L_{DFA} = {<B, w> | B is a DFA that accepts input string w}

Theorem: L_{DFA} is a Turing-decidable language.

Proof: We construct a Turing machine M that decides L_{DFA} as follows:
M = on input <B, w>, where B is a DFA and w is a string

1. Simulate B on input w.
2. If the simulation ends in an accept state of B, accept <B, w>.
   If it ends in a non-accepting state of B, reject <B, w>

We need to provide more details on the input <B, w>. B is a finite state acceptor, which we defined as a 5-tuple (S, A, i, t, F) where:

- S = Set of States
- A = Alphabet
- i = Initial State
- t = Transition Mapping $t : S \times A \rightarrow S$
- F = State of Finishing States

The string w is over the alphabet A, so the pair <B, w> as input for our own Turing machine is in fact (S, A, i, t, F, w). The Turing machine M starts in the configuration εiw. (Remind yourself of what a configuration is.)

If w = uv, where $u \in A$ is the first character in the word w and if t(i, u) = s, then the next configuration of the Turing machine M is usv, i.e. the new state corresponds to the state S in which B enters from the initial state i upon receiving input character u and the tape head has moved right past u ready to examine the second character of w.
Once the string w has been completely processed, then the configuration of the Turing machine is $ws_w\varepsilon$. If the final state s_w where we ended up is an accepting state, i.e. $s_w \in F$, then we accept <B, w>, otherwise we reject <B, w>. (*q.e.d*)

2) The acceptance problem for nondeterministic finite state acceptors (NFAs)
Test whether a given nondeterministic finite state acceptor B accepts a given string w.
Rewrite this acceptance problem as a language:
L_{NFA} = {<B, w> | B is a NFA that accepts input string w}

Theorem: L_{NFA} is a Turing-decidable language.

Proof: This results is in fact a corollary to the previous theorem. As we showed in our unit on formal language and grammars, given any NFA B, ∃ a deterministic finite state acceptor B that corresponds to it (with potentially many more states). Therefore, to any pair <B, w> ∈ L_{NFA}, there corresponds a pair <B', w> ∈ L_{DFA}.
Since L_{DFA} is a Turing-decidable language, L_{NFA} is Turing-decidable as well. (*q.e.d*)

3) The acceptance problem for regular expressions
We rewrite this acceptance problem as the language L_{REX} = {<R, W> | R is a regular expression that generates string w}.

Theorem: L_{REX} is a Turing-decidable language.

Proof:
Recall that a language L is regular ⇔ L is accepted by a deterministic or nondeterministic finite state acceptor ⇔ L is given by a regular expressions.
There exists an algorithm to construct a nondeterministic finite state acceptor from any given regular expression ⇒ ∀<R, w> ∈ L_{REX}, ∃<B, w> ∈ L_{NFA} that corresponds to it.
Since L_{NFA} is Turing-decidable, L_{REX} is Turing decidable. (*q.e.d*)

4) Emptiness testing for the language of an automaton
Given a DFA B, figure out whether the language recognised by B, L(B) is empty or not, i.e. whether L(B) ≠ Ø or L(b) = Ø.
Rewrite the emptiness testing problem as a language:
E_{DFA} = {<B> | B is a DFA and L<B> = Ø}.

Theorem: E_{DFA} is a Turing-decidable language.

Proof:
A DFA B accepts a certain string w if we are in an accepting state when the last character of w has been processed. We design a Turing machine M to test this condition as follows:
M = on input <B>, where B is a DFA:
1. Mark the initial state of B.
2. Repeat until no new states of N get marked.
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, then accept, otherwise reject.

We have thus marked all states of B where we can end up given an input string. If no such state is an accepting state, then B will not accept any string, i.e. L(B) = Ø as needed. (*q.e.d*)

5) Checking whether two given DFAs accept the same language
Given B_1, B_2 DFAs, test whether $L(B_1) = L(B_2)$.
We rewrite this problem as the language
$EQ_{DFA} = \{< B_1, B_2> \mid B_1$ and B_2 are DFAs and $L(B_1) = L(B_2)\}$.

Theorem: EQ_{DFA} is a Turing-decidable language.

Proof:
Given two sets Γ and Σ, $\Gamma \neq \Sigma$ if $\exists\ x \in M$ such that $x \notin \Sigma$ (i.e. $\Gamma \setminus \Sigma \neq \varnothing$) or $\exists\ x \in \Sigma$ such that $x \notin \Gamma$ (i.e. $\Sigma \setminus \Gamma \neq \varnothing$).
Recall from our unit on set theory that $\Gamma \setminus \Sigma = \Gamma \cap {\sim}\Sigma$, Γ intersect the complement of Σ.
Similarly, $\Sigma \setminus \Gamma = \Sigma \cap {\sim}\Gamma$.
Therefore, $\Gamma \neq \Sigma \Leftrightarrow (\Gamma \cap {\sim}\Sigma) \cup (\Sigma \cap {\sim}\Gamma) \neq \varnothing$. This expression is called the **symmetric difference** of sets Γ and Σ in set theory.

Now, returning to our problem, note that B_1 and B_2 are DFAs $\Rightarrow L(B_1)$ and $L(B_2)$ are regular languages. Furthermore, we showed that the set of regular languages is closed under union, intersection and the taking of complements $\Rightarrow (L(B_1) \cap {\sim}(L(B_2)) \cup (L(B_2) \cap {\sim}(L(B_1))$ is a regular language $\Rightarrow$ C a DFA that recognises the symmetric difference of $L(B_1)$ and $L(B_2)$ $(L(B_1) \cap {\sim}(L(B_2)) \cup (L(B_2) \cap {\sim}(L(B_1))$.
$L(B_1) = L(B_2)$ if this symmetric different is empty $\Rightarrow \forall <B_1, B_2> \in EQ_{DFA}\ \exists <C> \in E_{DFA}$, the language corresponding to the emptiness testing problem.
Since E_{DFA} is Turing-decidable, EQ_{DFA} is Turing-decidable. (*q.e.d*)

Next, we look at context-free grammars (CFGs) that we studied last term.

6) L_{CFG} = {<G, w> | G is a CFG and w is a string}
Theorem: L_{CFG} is a Turing-decidable language.

Sketch of Proof:
We could try to go through all possible applications of production rules allowable under G to see whether we can generate w, but infinitely many derivations may need to be tried. Therefore, if G does not generate w, our algorithm would not halt. We would thus have a Turing machine that is a recogniser but **not** a decider.
To get a decider we have to put G into a special form called a Chomsky normal form that takes 2n-1 steps to generate a string w of length n. We do not need to know what a Chomsky normal form is, just that one exists in order to write down our decider M.

M = on input <G, w>, where G is a context-free grammar and w is a string.
1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with 2n-1 steps, where n is the length of w if n > 0. If n = 0 list all derivations with one step.
3. If any of these derivations generate w then accept, otherwise reject.

7) Emptiness testing for context-free grammars
Given a context-free grammar G, figure out whether the language it generates L(G) is empty or not.
Rewrite as a language E_{CFG} = {<G> | G is a CFG and L(G) = Ø}

Theorem: E_{CFG} is a Turing-decidable language.

Proof:
We use a similar marking argument as we did to show E_{DFA} was Turing-decidable. We define the Turing machine as:
M = on input <G>, where G is a CFG:
1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked.
3. Mark any nonterminal <T> if G contains a production rule <T> ⇒ u_1, …, u_k has already been marked.
4. If the start symbol <S> is not marked then accept, otherwise reject.

As we can see from step 4, if <S> is marked then the context-free grammar will end up generating at least one string as all terminals have already been marked in step 1. Therefore, L(G) ≠ Ø and we reject G. (*q.e.d*)

8) Equivalence problem for context-free grammars
Given two context-free grammars G_1 and G_2, determine whether they generate the same language, i.e. $L(G_1) = L(G_2)$.
Rewrite this problem as a language:

EQ_{CFG} = {< G_1, G_2> | G_1 and G_2 are CFGs and $L(G_1) = L(G_2)$}'.

To solve the equivalence problem for DFAs, we used the symmetric difference and the fact that the emptiness problem for DFAs is Turing-decidable. In this case, the emptiness problem for CFGs is Turing-decidable as we just proved, but the symmetric difference argument does **not** work as the set of languages produced by context-free grammar is **not** closed under complements or intersection so the following result is true instead:

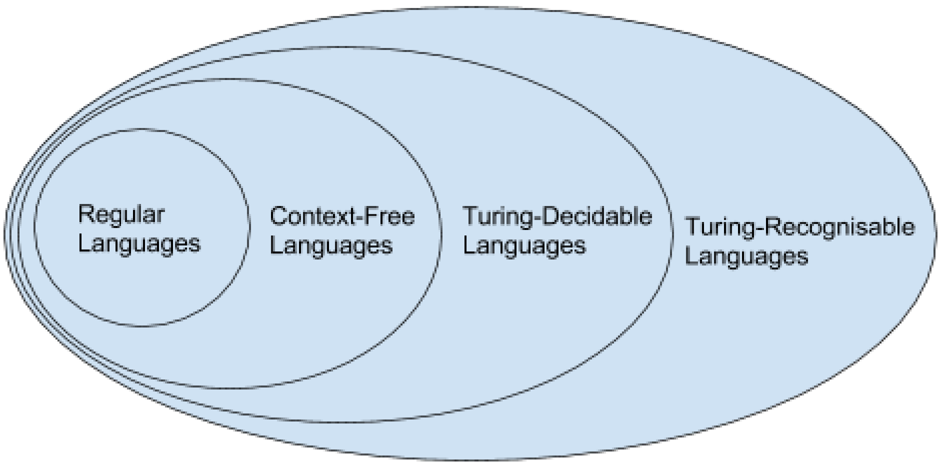
Proposition: EQ_{CFG} is **not** a Turing-decidable language.
This proposition is proven using a technique called reducibility. An even more general result is true, the equivalence problem for Turing machines is undecidable:

EQ_{TM} = {<M_1, M_2> | M_1 and M_2 are Turing machines and $L(M_1) = L(M_2)$}.

Proposition: E_{TM} is **not** a Turing-decidable language.
Returning to context-free grammars, we now know that L_{CFG} and E_{CFG} are Turing-decidable, but EQ_{CFG} is not.
Recall that a language is context-free if it can be generated by a context-free grammar.

Moral of the Story:
We know how the main types of languages relate to each other:
{Regular Languages} ⊂ {Context-Free Languages} ⊂ {Turing-Decidable Languages} ⊂ {Turing-Recognisable Languages}

Visually we represent the relationship with a Venn diagram:



So Turing machines provide a very powerful computational model. What is surprising is that once we have build a Turing machine to recognise a language, we do not know whether there is a simpler computational model such as a DFA that recognises the same language.
Define:

Regular_{TM} = {<M> | M is a Turing machine and L(M) is a regular language}.

Theorem: Regular_{TM} is **not** a Turing-decidable language.
This theorem is proven using reducibility. In fact, even more is true:
Rice's Theorem: Any property of the languages recognised by Turing machines is not Turing-decidable.

Undecidability

Task: Understand why certain problems are algorithmically unsolvable.
Recall that a Turing machine is defined as a 7-tuple (S, A, Ã, t, i, S_{acc}, S_{rej}) where:

- S = Set of States
- A = Input Alphabet **not** containing the blank symbol _
- Ã = Tape Alphabet where _ ∈ Ã and A ⊆ Ã
- t = Transition Mapping t : S x Ã $\rightarrow$ S x Ã x {L, R}
- i = Initial State
- S_{acc} = Accept State
- S_{rej} = Reject State

Definition: An **encoding** <M> of a Turing machine M refers to the 7-tuple (S, A, Ã, t, i, S_{acc}, S_{rej}) that defines M and is therefore a finite string.
Recall that earlier in the module we proved the following results:

Theorem:
If A is a finite alphabet, then the set of all words over A ($A^* = A^0 \cup A^1 \cup \dots \cup A^\infty$) is countably infinite.

Corollary 1:
If A is a finite alphabet, then the set of all languages over A is uncountably infinite.

Corollary 2:
The set of all programs in any programming language is countably infinite.
Recall that we proved *corollary 2* by realising that for any programming language, a program is a finite string over the finite alphabet of all allowable character in that programming language.

Corollary 3:
Given a finite alphabet A, the set of all Turing-recognisable languages over A is countably infinite.

Proof:
An encoding <M> of a Turing machine M is the 7-tuple (S, A, Ã, t, i, S_{acc}, S_{rej}), which is a finite string over a language B that contains A and is finite.
By the theorem, $B^* = B^0 \cup B^1 \cup \dots \cup B^\infty$ is countably infinite.
Since <M> ∈ B*, there are at most countably infinitely many Turing machines M that recognise languages over A
⇒ there are at most countably infinitely many Turing-recognisable languages over A.
We know we can build Turing machines with as large a set of states S as we want
⇒ the set of Turing machines that recognises languages over A cannot be finite
⇒ it is countably infinite. *(q.e.d)*

Proposition: Let A be a finite alphabet. Not all languages over A are Turing-recognisable.

Proof:
By *corollary 1*, the set of all languages over A is uncountably infinite.
By *corollary 3*, the set of all Turing-recognisable languages over A is countably infinite
⇒ there are many more languages over A than can be recognised by a Turing machine.
(q.e.d)

Remark:
This result makes a lot of sense because while we normally look at simpler, well-structured problems where there is a pattern, most languages over A have no pattern to them.
To understand more on the set of all Turing machines, we define the language
L_{TM} = {<M, w> | M is a Turing machine and M accepts w}
Here w is a string over the input alphabet A.
We will prove that L_{TM} is a Turing-recognisable language, but L_{TM} is **not** Turing-decidable.

Proposition: L_{TM} is a Turing-recognisable language.

Proof: We define a Turing machine U that recognises L_{TM}:
U = on input <M, w>, where M is a Turing machine and w is a string.

1. Simulate M on string w.
2. If M ever enters its accept state then accept.
   If M ever enters its reject state then reject.

U loops on input <M, w> if M loops on w ⇒ U is a recogniser but not a decider. (*q.e.d*)

Remark:
The Turing machine U is an example of the **universal Turing machine** first proposed by Turing in 1936. This idea of a universal Turing machine led to the development of stored-program computers.

NB: Philosophically, the universal Turing machine we just constructed runs into the following big issues:

1. U itself is a Turing machine. What happens when U is given an input <U, w>?
2. The encoding of a Turing machine is a string. What happens when we input <M, <M>> or even worse <U, <U>>?

We are getting very close to Russell's paradox, the set $\Gamma = \{D \mid D \notin D\}$ which showed the axioms of naive set theory were inconsistent and led to more complicated axioms.
In one case, these issues lead to showing the language L_{TM} cannot possibly be Turing-decidable.

Proposition: L_{TM} is not Turing-decidable.

Proof: Assume L_{TM} is Turing-decidable and obtain a contradiction.
If L_{TM} is Turing-decidable, then ∃ decider H for L_{TM}.
Given input <M, w>, H:

1. Accepts if M accept w.
2. Rejects if M does not accept w.

We now construct another Turing machine D with H as a subroutine, which belongs like the set Γ defined by Russell
D = on input <M>, where M is a Turing machine:

1. Run H on input <M, <M>>
2. Output the opposite of what H outputs.
   If H accepts, then reject.
   If H rejects, then accept.

Now, let us run D on its own encoding <D>:
D on input <D>:

1. Accepts if D does not accept <D>
2. Rejects if D accepts <D>

⇒⇐ D cannot exist, hence H cannot exist. The language L_{TM} has no decider. (*q.e.d*)

Example of a language that is not Turing-recognisable:
Task: Use what we know about L_{TM} to build an example of a language that is not Turing-recognisable.

Definition:
GIven an alphabet A that is finite, ($A^* = A^0 \cup A^1 \cup \ldots \cup A^\infty$), and then a language $L \subset A^*$, we define the complement ~L of L as ~L = A* \ L, i.e. all words over A that are not in L.

Definition:
A language L is called co-Turing-recognisable if its complement ~L is Turing-recognisable.

Theorem: A language L is decidable ⇔ L is Turing-recognisable and co-Turing-recognisable

Proof:
"⇒" If L is decidable ⇒ L is Turing-recognisable. Note that if L is decidable ⇒ ∃ a Turing machine M that decides L.
Build a Turing machine ꝏ that reverses the output of M, i.e. if M accepts a string w, then ꝏ rejects the same string w. If M rejects w then ꝏ accepts w.
M is therefore a decider for ~L ⇒ ~L is Turing-decidable ⇒ ~L is Turing-recognisable, so L is Turing-recognisable and co-Turing-recognisable.

"$\Leftarrow$" If both L and ~L are Turing-recognisable $\Rightarrow$ $\exists$ M_1 that recognises L and $\exists$ M_2 that recognises ~L. We use Turing machines M_1 and M_2 to build a decider M for L as follows:
M = on input w, where w is a string:

1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, then accept.
   If M_2 accepts, then reject.

Running M_1 and M_2 in parallel simply means the M has two tapes: one for simulating M_1 and one for M_2.
Note that for any string w, either $w \in L$ or $w \in$ ~L, which means either M_1 or M_2 accepts w $\Rightarrow$ M either accepts or rejects any string.
In fact, M accepts w $\Leftrightarrow$ $w \in L$ by construction $\Rightarrow$ M is a decider for L.
$\Rightarrow$ L is Turing-decidable. (*q.e.d*)

Corollary: ~(L_{TM}) Is **not** Turing-recognisable.

Proof:
We proved L_{TM} is Turing-recognisable. If ~(L_{TM}) were Turing-recognisable, then L_{TM} would be both Turing-recognisable and co-Turing-recognisable.
$\Rightarrow$ By the previous theorem, L_{TM} would be Turing-decidable $\Rightarrow\Leftarrow$ as we proved the contrary
$\Rightarrow$ ~(L_{TM}) is not Turing-recognisable, and we have constructed our example of a non Turing-recognisable language. (*q.e.d*)