Algorithms

Task:
Use Hilbert's 10^{th} 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 1^{st} of Hilbert's 23 problems in 1900 at the International Congress of Mathematicians.

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

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 $Z^n = \{c_1, c_2, \ldots\}$ where $c_i = (x_1^{(i)}, \ldots, x_n^{(i)})$.
Then M_n = an input p, where p is a polynomial in $x_1, \ldots, x_n$

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

If p has an integer root $(x_1^{(i)}, \ldots, x_n^{(i)}) \in Z^n$ 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 x A → 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 ∈ 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
$EQ_{DFA} = \{< B_1, B_2> \mid B_1 \text{ and } B_2 \text{ are DFAs and } L(B_1) = L(B_2)\}$.

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. $\Gamma \setminus \Sigma \neq \emptyset$) or $\exists\ x \in \Sigma$ such that $x \notin \Gamma$ (i.e. $\Sigma \setminus \Gamma \neq \emptyset$).
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, $\Gamma \neq \Sigma \Leftrightarrow (\Gamma \cap \sim\Sigma) \cup (\Sigma \cap \sim\Gamma) \neq \emptyset$. This expression is called the **symmetric difference** of sets Γ and Σ in set theory.