4

Noncomputable Languages

Recall that a language $A \subseteq \Sigma^*$ is Turing-computable (or simply computable) when there is a *total* Turing machine M that accepts A. By the Church-Turing thesis, we can say that a decision problem $d : \Sigma^* \longrightarrow \{1, 0\}$ is computable when its associated language $\{u \in \Sigma^* \mid f(u) = 1\}$ is Turing-computable.

In this lecture, we focus on Turing machines over input alphabet $\{0, 1\}$ for ease of exposition.

Encoding Turing Machines

It will be useful in this lecture to be able to encode a description of a Turing machine as a string over alphabet $\{0, 1\}$. The exact details of the encoding are somewhat irrelevant, but it needs to be such that it is reasonably easy to extract out the states and transitions of the machine.

If $M = (Q, \{0, 1\}, \Gamma, \sqcup, \vdash, \delta, s, acc, rej)$, without loss of generality, we can write the states of Q as $\{Q_1, \ldots, Q_k\}$, with $Q_1 = s$, $Q_2 = acc$, and $Q_3 = rej$. Without loss of generality, we can also write the symbols of the tape alphabet Γ as $\{X_1, \ dots, X_n\}$, where $X_1 = 0$, $X_2 = 1$, $X_3 = \vdash <$ and $X_4 = \sqcup$. Let D_1 be the direction L and D_2 be the direction R. This means, really, that to describe the Turing machine M, we need only list the transitions. Each transition is of the form $\delta(Q_{n_1}, X_{n_2}) = (Q_{n_3}, X_{n_4}, D_{n_5})$ where $n_1, \ldots, n_5$ are natural numbers. Each individual transition therefore can be encoded as a string

$$0^{n_1}10^{n_2}10^{n_3}10^{n_4}10^{n_5} \tag{4.1}$$

and the entire Turing machine be encoded as a string

$$111\ code_1\ 11\ code_2\ 11\ \ldots\ 11\ code_k\ 111 \tag{4.2}$$

where each $code_i$ is a string of the form 4.1, and each transition of M is encoded by one of the $code_i$

Let $\langle M \rangle$ represent the encoding of Turing machine M as a string over $\{0, 1\}$.

Existence of Noncomputable Languages

A language is said to be *noncomputable* when it is not computable.[1] It is pretty easy to show that there must be at least one noncomputable language: a simple counting argument reveals that there are more languages over $\{0, 1\}$ than there are Turing machines. One has to be careful, because there are infinitely many Turing machines and languages, and counting in the presence of infinite sets is tricky. Formally, there are countably many Turing machines, and uncountably many languages over $\{0, 1\}$.

Thankfully, we can show directly that there has to be noncomputable languages. Let $\mathcal{T}$ be the set of Turing machines, and $\wp(\{0, 1\}^*)$ the set of of languages over $\{0, 1\}$. Consider the map $L : \mathcal{T} \longrightarrow \wp(\{0, 1\}^*)$ associating to every Turing machine M the language $L(M)$ that M accepts. There must be a language B with the property that there is no M such that $L(M) = B$. Such a language is, by definition, noncomputable.

We can construct the language by diagonalization, a technique due to Cantor. Define the following language:

$$B = \{\langle M \rangle \mid \langle M \rangle \notin L(M)\}$$

Let's parse this. B is the set of all strings u over $\{0, 1\}$ representing the encoding of a Turing machine such that u is not in the language of the Turing machine.

I claim that there is no Turing machine M_b (total or otherwise) that accepts B, and therefore B is noncomputable.

Suppose there were a M_b with the property that $L(M_b) = B$. Just for kicks, do we have $\langle M_b \rangle \in B$, or not? Turns out, neither! Let's look at both cases:

[1] A noncomputable language is also sometimes called undecidable.

(1) If $\langle M_b \rangle \in B$, then by definition of B, it must be that $\langle M_b \rangle \notin L(M_b)$, and since $L(M_b) = B$, then $\langle M_b \rangle \notin B$!

(2) If $\langle M_b \rangle \notin B$, then by definition of B, it must be that $\langle M_b \rangle \in L(M_b)$, and since $L(M_b) = B$, then $\langle M_b \rangle \in B$!

Either way, we get a contradiction. Since these are the only two possibilities for $\langle M_b \rangle$ and B, we must conclude that M_b cannot exist. Therefore, there is no Turing machine that accepts B, and B is noncomputable.

Granted, B is weird and not particularly interesting. A better question is: can we find a *natural* language that is noncomputable?

Universal Turing Machines

To define a natural noncomputable language. we use one of Turing's main result about his machines: universality. It is possible to develop a *universal* Turing machine U that takes as input an encoding of a Turing machine M and an input string w and simulates running Turing machine M on input string w, accepting when M accepts, rejecting when M rejects, and looping when M loops. This may loop weird at first, but really it is no weirder than writing a Python interpreter in Python.

We use the code defined by 4.2, and write $\langle M, w \rangle$ for the encoding of machine M followed by input string w. Note that we can always recognize w in $\langle M, w \rangle$ as the string following the second block of 111.

The easiest way to define Turing machine U is to start with a 3-tapes Turing machine U' with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, \vdash, \sqcup\}$.[2] The first tape is the input tape, and the tape head on that tape is used to look up the transitions to make when given input $\langle M, w \rangle$. (Recall that the encoding $\langle M \rangle$ records all the transitions of the Turing machine using codes of the form 4.1.) The second tape of U' will simulate the tape of M. Multiple cells of the tape will simulate single cell of U', since we need to encode the tape alphabet of M using esentially $0, 1$. We can use 0^n to represent symbol n, and separate the cells using 1. The third tape of U' holds the state M, with state n represented as 0^n. The behavior of U' is as follows:

1. Check the format of tape 1 to make sure its prefix describes the encoding of a Turing machine (checking it starts with 111, that is has a

[2]This description is taken from Hopcroft and Ullman *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley.

sequence of codes of the form 4.1 separated by 11 and ended by 111, etc). Reject if not.

2. Initialize tape 2 to contain w, the portion of the input beyond the second block of 111, suitably encoded to represent the symbols of the tape alphabet of M. Initialize tape 3 to hold 0, the initial state of M.

3. if tape 3 holds 00, accept; if tape 3 holds 000, reject.

4. Let j be the encoded symbol currently scanned by tape head 2, and let 0^i be the current content of tape 3. Scan tape 1 from left to the second block 111, looking for a substring beginning 110^i10^j1. If no such string is found, reject. If such a code is found, let be $0^i10^j10^k10^l10^m$. Put 0^k on tape 3, put l on tape 2 in place of j (possibly shifting the tape left and right if needed), and move the head in direction m. Go back to step 3.

It is straightforward, if tedious, to check that U' accepts $\langle M, w\rangle$ when M accepts w and U' rejects $\langle M, w\rangle$ when M rejects w.

We know from previous lectures that a three-tapes Turing machine can be simulated by a one-tape Turing machine, and we can take U to be the one-tape Turing machine simulating U'.

The Halting Problem

We say that a Turing machine M halts on input w if M either accepts or rejects w, we don't care which. Basically, M does not spin forever on input w.

Consider the following language, called the *Halting Problem*:

$$HP = \{\langle M, w\rangle \mid M \text{ halts on input } w\}$$

I claim that HP is noncomputable, that is, there is no total Turing machine M that accepts HP. We argue by contradiction: assume HP is computable, and derive an absurdity.

Assume HP is computable. That means we have a total Turing machine K that accepts HP. That is, K accepts $\langle M\rangle w$ when M halts on w, and rejects $\langle M\rangle w$ when M does not halt on w. Let $\langle K\rangle$ be the encoding of K.

Using K, construct another Turing machine I as follows:

On input x:

1. Run U with input $\langle K \rangle xx$
2. If U rejects, accept
3. If U accepts, go into an infinite loop

Clearly, we can implement I as a Turing machine since we know $\langle K \rangle$. We can just modify U to rewrite $\langle K \rangle xx$ on its input tape and modify the reject and accept states of U. Since I is a Turing machine, it has an encoding $\langle I \rangle$, which is just a string over $\{0, 1\}$. We can also ask whether I halts on any given input. Let's be devious and ask whether I halts on input $\langle I \rangle$!

There are only two possibilities. Either I halts on input $\langle I \rangle$, or it does not. Neither makes sense.

- Say I halts on input $\langle I \rangle$. By definition of I, this happens only when U rejects $\langle K \rangle \langle I \rangle \langle I \rangle$. Since U is a universal Turing machine, U rejects when K rejects $\langle I \rangle \langle I \rangle$. But K is the Turing machine accepting HP, and it rejects exactly when I *does not* halt on $\langle I \rangle$. But we said I halts on $\langle I \rangle$. That's absurd: it can't do both.

- Say I does not halt on input $\langle I \rangle$. By definition of I, this happens only when U accepts $\langle K \rangle \langle I \rangle \langle I \rangle$. But U is a universal Turing machine, so U accepts when K accepts $\langle I \rangle \langle I \rangle$. But K is the Turing machine accepting HP, and it accepts when I *does* halt on input $\langle I \rangle$. But we said I does not halt on $\langle I \rangle$. That's absurd: it can't do both.

Either way, we get an absurdity. So our assumption that HP is computable must be wrong. There is no total Turing machine K that accepts HP, and HP is noncomputable.