Sheet 2 Question 1 part (d)

The reader is asked to supply a partition of $\mathbb{N}$ into two pieces, neither of them r.e. aka semidecidable.

The way to do this is to use Church's thesis: think of a partition of $\mathbb{N}$ into two pieces neither of which is (informally) finitely recognisable. One fairly obvious candidate is

$$\mathbb{N} = \{n : |W_n| < \aleph_0\} \cup \{n : |W_n| = \aleph_0\}$$

Here $W_n = \{k : \{n\}(k) \downarrow\}$, the set of arguments on which the nth partial function `halt`s.

How do you prove that a set is not semidecidable? It's a bit like showing that a set is not recursive aka decidable. In both cases you say "if this thing were [semi]decidable then there would be a finite engine that did something, and i could bend that finite engine to the task of solving the `halt`ing problem." This means doing some coding tricks (coding up an instance of the `halt`ing problem into a problem of the kind the putative finite engine can do) which is actually doing a many-one reduction. I'd never thought of coding tricks as many-one reductions (that is Chiodo's idea) but one learns something new every day.

So let's do some coding tricks.

$\{n : |W_n| < \aleph_0\}$ is not semidecidable

Suppose *per impossibile* that this set were semidecidable.

Given $\{n\}(n)$ an instance of the `halt`ing problem, we can say the following. If it `halt`s, we learn this in finite time. While we are waiting for it to `halt`, create the function: $k \mapsto \{n\}(n)$. By the assumption we can detect in finite time if it halts on only finitely many inputs. But if it `halt`s on only finitely many inputs, that is because $\{n\}(n)$ does not `halt`. So we have a decision procedure for the `halt`ing problem.

$\{n : |W_n| = \aleph_0\}$ is not semidecidable

Suppose *per impossibile* that this set were semidecidable.

Given $\{n\}(n)$ an instance of the `halt`ing problem, do the following. If it `halt`s, we learn this in finite time. Meanwhile, do the following. Create the function:

$$k \mapsto \text{if } \{n\}(n) \text{ is still running after } k \text{ steps, output } k \text{ else do nothing.}$$

By the assumption we can detect in finite time if this function `halt`s on infinitely many inputs. But it `halt`s on infinitely many inputs iff $\{n\}(n)$ does *not* `halt`.

So we have a decision procedure for the `halt`ing problem.

We took the `halt`ing set to be $\{n : \{n\}(n) \downarrow\}$. If we'd wanted it to be instead $\{\langle p, i\rangle : \{p\}(i) \downarrow\}$ we do a global search-and-replace in the above text to rewrite the first string as the second.

We found an answer to this question by using Church's thesis to guess a set that would do the job, and then using a coding trick (a many-one reduction) to show that the set could be used to solve the `halt`ing problem. The use of the word 'trick' here is a reminder that this is something that cannot be reliably taught. It's not an Imre switch-brain-off-do-this process: it requires ingenuity. Not a great deal of ingenuity, but it does require something *ad hoc*. You have to think a bit and tinker a bit. With practice grows confidence.