So, this is just to jot down the key idea behind proofs of the two challenge problems we’ll discuss in class on March 20 (regarding the end of the Chapter 3, Part 2 slides).

Let us for this document cut over from the Sipser notation to the standard one, regarding machines and encodings of them. So let $M_1, M_2, M_3, \ldots$ be a standard enumeration of Turing machines. And $i$ is called the code or index of $M_i$.

We’ll use recursive presentation here to mean what often is called an effective enumeration. That is, a recursive presentation of a class $C$ is a (total) recursive function $\sigma$ such that $C = \{L(M_{\sigma(1)}), L(M_{\sigma(2)}), L(M_{\sigma(3)}), \ldots\}$.

## 1 A Recursive Presentation of the Recursive Sets

The first challenge problem was, in effect, to show that there is a (total) recursive function $\sigma$ such that the class $C = \{L(M_{\sigma(1)}), L(M_{\sigma(2)}), L(M_{\sigma(3)}), \ldots\}$ is exactly the Turing-decidable (i.e., recursive) sets.

And the proof idea is $\sigma(i)$ will output (will be) the index of a Turing machine that does the following. When given an arbitrary input $y$, it will, in sequence, for each string $z$ in $\Sigma^*$ in short-lex order starting with $\epsilon$ and continuing up to the string immediately before $y$, simulate $M_i$ on input $z$, except if $M_i$ in that simulation tries to enter an accept or a reject state, we instead just move on to the next simulation. And then it does a true simulation of $M_i(y)$.

Note that if $M_i$ halts on all inputs, then $L(M_{\sigma(i)}) = L(M_i)$. So $C$ clearly contains all the recursive sets.

Note again that if $M_i$ halts on all inputs, then $L(M_{\sigma(i)}) = L(M_i)$, and so $L(M_{\sigma(i)})$ is recursive. And also note that if $M_i$ runs forever on at least one input then $L(M_{\sigma(i)})$ is finite, but note too that all finite sets are recursive. And so each set in $C$ is recursive.
So $C$ is exactly the recursive sets. Done!

However, sometimes it can be helpful to see things as actual (pseudo)code. So... the following is a write-up I did on this problem some time ago, and I include it verbatim in case having it might be helpful to you:

Hi! Here are some notes on the very hard problem that is the main problem give on Chapter 3, Part 2, slide 46.

This is very advanced, so you don’t have to read this. It is just provided for those who are interested. We won’t test you on it.

Anyway, here is, as pseudocode, a description of a function sigma that satisfies the problem’s requirements. Let $M_1, M_2, \ldots$ be the std enumeration of TMs. the type of $x$ below is a string, and we assume that $++$ on strings increments by 1 in string order. we now describe the program that is what sigma(i) does, but note that it is easy to write a program on input $i$ prints a TM that does what the pseudocode below does.

**PROGRAM NAME SIGMA(i)**

**INPUT TO OUR PROGRAM:** A string $y$

**BEGIN**

$x = \text{theemptystring}$

Do until $(x == y)$

\{ 

simulate $M_i$ running on input $x$ except if in the simulation $M_i$ is going to accept or reject, instead just end its simulation (and so we move on to step FOO in the present program)

FOO: $x++$

\}

simulate $M_i$ running on input $y$ (if it accepts we accept, if it rejects we reject, and if it runs forever we will end up also running forever).

**END**

If $A$ is decidable, then for $A$ is the language accepted by some $M_j$ that halts on each input, and note that $L(\text{sigma}(j)) = A$, so $A$ is handled by our list.
For each $k$, note that $L(\sigma(k))$ is decidable. If $M_k$ halts on all inputs, then $L(\sigma(k)) = L(M_k)$ and so is decidable. If $M_k$ runs forever on some input, then $L(\sigma(k))$ is finite, and so is decidable.

That completes the proof sketch.

Cheers,
Lane

2 There Is No Recursive Presentation of the Recursive Sets Using Total Machines

Can one improve the above to even ensure that each $M_{\sigma(i)}$ is a machine that halts on all inputs? Impossible! We can directly diagonalize against that. How? Assume it is true. Consider the set that on the $i$th string in short-lex order contains that string if and only if that string is not part of $L(M_{\sigma(i)})$. Our set is (under our assumption) recursive, yet it is not in $\mathcal{C}$—a contradiction.

This is an example of a diagonalization proof.

3 There Is A Recursive Presentation of the Recursively Enumer-able Sets Such that No Three Machines in a Row Accept the Same Language

So, we are here going to do the “no 3 in a row” case (see the next section for the “no 2 in a row” case).

For each $j \geq 1$, set the following. Let $\sigma(3j - 2)$ be the code of a machine that simply does what $M_j$ does. Let $\sigma(3j - 1)$ be the code of a machine that accepts the empty set. Let $\sigma(3j)$ be the code of a machine that accepts $\Sigma^*$. This assures no 3 in a row (in this presentation) accept the same language, as any 3 in a row contain one machine accepting the empty set and one accepting $\Sigma^*$, and those differ.
4 There Is Even A Recursive Presentation of the Recursively Enumerable Sets Such that No Two Machines in a Row Accept the Same Language

To do no 2 in a row, for each $j$ set the following. Let $\sigma(2j-)$ be the code of a machine that on input $\epsilon$ accepts and on all other inputs simulates $M_j$ on that input. Let $\sigma(2j)$ be the code of a machine that on input $\epsilon$ rejects and on all other inputs simulates $M_j$ on that input.

No 2 in a row accept the same set, as adjacent machines in this presentation differ on the empty string and so don’t accept the same set.

Special note: The 2 case on its surface seems to violate the so-called Recursion Theorem, and so to be impossible. However, the 2 case is using an enumeration that is “effective” but not “acceptable,” and the Recursion Theorem requires acceptable enumerations, so this claim and proof evade the grasp of the Recursion Theorem. In contrast, our no 3 in a row proof gives an enumeration that is not just effective but is even acceptable. (Note: Acceptable basically means there is a recursive function that maps from it to the standard enumeration and there is a recursive function that maps from the standard enumeration to it. For the no-2-in-a-row case, one can easily recursively map from it to the standard enumeration, but one can prove that there is no recursive mapping from the standard enumeration to it.)