CS 360
Fall 2021
Enumerators and Recursively Enumerable Languages

Jeffrey Shallit
School of Computer Science
University of Waterloo
shallit@uwaterloo.ca
https://cs.uwaterloo.ca/~shallit
Recall that a language $L$ is *Turing-recognizable* if there exists a Turing machine $T$ such that $L = L(T)$.

- If $x \in L$, then $T$ must eventually halt and accept $x$.
- If $x \not\in L$, then $T$ can either halt and reject $x$, or run forever.

Another term for Turing-recognizable is *recursively enumerable* or *r.e.*.

In this lecture we will see why this other term is sometimes used.
Enumerators

An *enumerator* for a language $L$ is basically a Turing machine that lists the elements of $L$ in some order (not necessarily radix order).

Formally, it is a Turing machine with two tapes. Tape 1 is a work tape. Tape 2 is the output tape. When started on two blank tapes, the Turing machine (eventually) writes the following on its second tape:

$$x_1 \# x_2 \# x_3 \# \cdots$$

where $L = \{x_1, x_2, x_3, \ldots\}$.

The Turing machine never moves left on tape 2, nor does it ever rewrite any symbol other than a blank.

In general, an enumerator is not required to halt.

And, of course, if the language is infinite, the enumerator will definitely not halt.
Enumerators

Every $x \in L$ eventually gets written.

No $x \notin L$ gets written.

There is no order required.

It is allowed to write the same $x$ multiple times.
The main result

**Theorem.** A language $L$ is Turing-recognizable iff there is an enumerator TM for $L$.

**Proof.** $\iff$: Suppose there is an enumerator TM $T$ for $L$. We now create a 3-tape recognizer DTM $T'$ for $L$ from it.

Here is what $T'$ does: on input $x$ on tape 1, it simulates $T$ on $T'$'s tapes 2 and 3.

Tape 2 of $T'$ is like tape 1 of $T$, and tape 3 of $T'$ is like tape 2 of $T$.

Every time $T$ writes a $\#$ symbol, $T'$ compares its input on tape 1 to the string that $T'$ just finished writing on tape 3. If they agree, then $T'$ halts and accepts. Otherwise, $T'$ continues the simulation of $T$. 


The main result

If $x \in L$, then it is guaranteed that $T$ eventually writes $x$ on tape 2, so $T'$ will write it on tape 3 and accept.

If $x \notin L$, then $T$ will never write it on tape 2, so $T'$ never accepts, but rather it runs forever.

This completes one direction of the proof.

The other direction is harder.
The naive solution would be to take a recognizer TM $T$ for $L$ and somehow produce an enumerator TM $T'$ from it, by running $T$ on every possible string $x \in \Sigma^*$, writing $x$ on its output tape if $T$ accepts.

But of course this doesn’t work, because $T$ might fail to halt on some input, at which point $T'$ would never write anything more out.

Instead we use a very clever technique called “dovetailing”.
In dovetailing, we *don’t* run $T$ on $x$ until it halts. Instead, we run $T$ for a certain number of steps, and increase this number of steps throughout the computation.

Here’s the idea: write all the possible strings in $\Sigma^*$ in radix order: $x_0 = \epsilon$, $x_1 = 0$, $x_2 = 1$, etc.

At iteration $n$ we run $T$ on $x_i$ for $n$ steps, for $0 \leq i \leq n$.

This requires putting a “clock” on a Turing machine. When we simulate another Turing machine, we have an extra tape, and at each move of the simulated Turing machine, we increment a counter on that tape.

Dovetailing
Dovetailing

For example for $\Sigma = \{0, 1\}$ we would

$n = 0$:  
run $T$ on $\epsilon$ for 0 steps

$n = 1$:  
run $T$ on $\epsilon$ for 1 step  
run $T$ on 0 for 1 step

$n = 2$:  
run $T$ on $\epsilon$ for 2 steps  
run $T$ on 0 for 2 steps  
run $T$ on 1 for 2 steps

$n = 3$: run $T$ on $\epsilon$ for 3 steps  
run $T$ on 0 for 3 steps  
run $T$ on 1 for 3 step  
run $T$ on 00 for 3 steps ... and so forth.
We now construct an enumerator $T'$ for $L$. $T'$ has 5 tapes.

- Tape 1 holds $n$, initially 0.
- Tape 2 holds the $i$’th string $x_i$ from $\Sigma^*$.
- Tape 3 is a work tape where we copy $x_i$ and then simulate $T$ on $x_i$ for $n$ steps.
- Tape 4 is used to record the number of steps carried out on tape 3. We increment this for every simulated move of $T$.
- Tape 5 is $T'$’s output tape.
Dovetailing

So the algorithm works as follows:

– increment $i$. If $i > n$, then increment $n$ and set $i = 0$.

– erase tape 3 and write $x_i$ on it.

– Simulate $T$ on tape 3, recording the number of steps on tape 4, and continuing until this number is $n$.

– If $T$ accepts at any point during this simulation, then $T'$ writes $x_i\#$ on tape 5

– go back to the top of the loop.
If $T$ accepts $x_i$, then it must do so after $r$ steps for some $r$. Then when $n = r$, the machine $T'$ writes out $x_i$ on its output tape.

If $T$ does not accept $x_i$, then $T'$ will never write it out.

So $T'$ is an enumerator for $L$.

So, now you know why recursively enumerable languages are called that.
Enumerators and Turing-decidable languages

There is also a characterization of the Turing-decidable languages in terms of enumerators.

Call an enumerator ordered if it enumerates the members of \( L \) in radix order, each element of \( L \) printed exactly once, separated by \#. If \( L \) is finite, the ordered enumerator should eventually halt.

**Theorem.** A language \( L \) has an ordered enumerator iff it is Turing-decidable.

Exercise: prove this.