# Lecture 17

# Enumerators and Dovetailing

In this lecture, we will go over a different characterization of the recognizable languages, which can be useful to get intuition for them. We also introduce the dovetailing technique, and we conclude with some additional closure properties for recognizable and decidable languages.

## 17.1 Enumerators

We define a variant of a Turing machine that is connected to a printer: such a machine can repeatedly "print out" strings during the course of its run. Such a Turing machine is called an *enumerator*.

The precise definition of an enumerator won't matter too much (because, as is usually the case with Turing machines, different variants of the definition will end up being equivalent). One way to define an enumerator is to say that is like a Turing machine with a second, special tape, and also a special "print" state $q_{\text{print}}$. When the Turing machine enters $q_{\text{print}}$, the contents of the second tape get cleared (we imagine that the string written on this tape got printed out), but the first tape remains the same, and the enumerator can keep running. Therefore, during the course of its run, the enumerator prints out a sequence of strings $w_1, w_2, w_3, \ldots$ in some output alphabet of our choice.

Unlike Turing machines, enumerators will always run starting with both tapes being empty; in other words, enumerators do not take in any inputs (this is equivalent to always running with the input being the empty string $\epsilon$).

For an enumerator $M$, we will define the language of the enumerator, denoted $L(M)$, to be the set of all strings $w \in \Sigma^*$ such that $M$ eventually prints out $w$; that is to say, $w \in L(M)$ if and only if there is some positive integer $k$ such that after $k$ steps, $M$ enters $q_{\text{print}}$ with the string $w$ written on the second tape. Note that an enumerator $M$ need not halt; even if it runs forever, the language $L(M)$ consisting of all strings $M$ eventually prints out is still well-defined.

We say that a language $A \subseteq \Sigma^*$ is *recursively enumerable* (abbreviated RE) if there is an enumerator $M$ such that $L(M) = A$. (The word "recursive" is an old term that refers to a computation, so "recursively enumerable" is a bit like saying "enumerable via computations".)

We will next show that a language is RE if and only if it is recognizable; this gives a somewhat different characterization of the recognizable languages (though it still involves some type of computing machines).

## 17.2 RE languages are recognizable

It is not hard to show that every recursively enumerable language is recognizable. To see this, consider a recursively enumerable language $A$, and let $M$ be an enumerator with $L(M) = A$. We construct a Turing machine $M'$ recognizing $A$. On input $w \in \Sigma^*$, the machine $M'$ will store $w$ and simulate $M$ starting with two empty tapes (using its one tape to simulate a two-tape Turing machine, as we've seen is possible). Whenever $M$ enters its print state $q_{\text{print}}$, $M'$ will check whether the string $x$ written on the second tape of $M$ is equal to the string $w$ that the machine $M'$ received as input; that is, $M'$ checks whether $M$ just printed the input string $w$. If so, $M'$ accepts. Otherwise, $M'$ clears the second tape of $M$ and continues in its simulation of $M$.

Note that if $w \in A$, then $M$ eventually prints out $w$, since $L(M) = A$. This means that eventually (after a finite number of steps) $M'$ will accept $w$, as it will see its simulation of $M$ print the string $w$.

Conversely, if $w \notin A$, then $M$ never prints out $w$. This means that $M'$ will never accept $w$, since it will never see its simulation of $M$ print the string $w$. Note that in this case, $M'$ might loop (if the enumerator $M$ never halts), but it will never accept. Hence $L(M')$ which is the set of all strings $M'$ accepts, must be exactly equal to $A$, so $A$ is recognizable.

## 17.3 Recognizable languages are RE

We've seen that every RE language is recognizable. The other direction, showing that every recognizable language is RE, is a little trickier.

Let $A \subseteq \Sigma^*$ be a recognizable language, and let $M$ be a Turing machine recognizing it, so $L(M) = A$. We wish to construct an enumerator $M'$ which prints out all the strings of $A$ and no other strings (in the sense that for any $w \in A$, eventually $M'$ will print out $w$, and also that $M'$ never prints out strings not in $A$).

Intuitively, what we would like to do is to have $M'$ simulate $M$ on every possible input string $w \in \Sigma^*$; that is, we would want to simulate $M(\epsilon)$, and also $M(0)$ (assuming $0 \in \Sigma$), and also $M(00)$, and $M(0111010)$ (assuming $1 \in \Sigma$), and so on for every $w \in \Sigma^*$. However, while $M'$ can simulate the Turing machine $M$ on an input $w$, the problem is that $M(w)$ may not halt. If, say, $M(\epsilon)$ does not halt, when should $M'$ give up and start simulating $M(0)$ instead? Since there is no way to tell if $M(\epsilon)$ will halt, $M'$ always risks giving up too early, no matter how long it waits before switching to $M(0)$.

Another option might be to run different copies of $M$ on different inputs in parallel. Concretely, let the alphabet be $\Sigma = \{0, 1\}$. $M'$ can go over the strings in $\Sigma^*$ in lexicographic order (that is, sorted by length first and then sorted alphabetically). For each such string $w$, it can try to run $M'(w)$ for just a few steps before giving up and moving on to the next string (with the hope of coming back to $w$ later). The problem with this "try everything in parallel" approach is that there are infinitely many strings; if $M'$ plans to make (say) 5 steps of $M(w)$ for each string $w$ and come back later to do 10 steps, it can never come back because it will never finish running $M(w)$ for 5 steps on every string $w$.

### 17.3.1 Dovetailing

What turns out to work is for $M'$ to do something called *dovetailing*. The machine $M'$ will start by running $M$ on the first string for 1 step. Then it will run $M$ on the first two strings for 2 steps each. Next, it will run $M$ on the first three strings for 3 steps each, and so on. If ever the simulation of $M(w)$ reaches an accept state, $M'$ prints out $w$ and continues.

Note that when we say "the first string" or "the first three strings", we mean in lexicographic order. It is easy to see that $M'$ can generate the first $k$ strings in lexicographic order for any positive integer $k$.

We now claim that this enumerator $M'$ enumerates $A = L(M)$. It is easy to see that if $w \notin L(M)$, then $M'$ never prints out $w$; this is because $M'$ only prints a string $w$ if $M$ accepted $w$. The other direction is more interesting. Let $w \in L(M)$. Then $M(w)$ accepts. This means there is some number $k$ of steps such that $M$ accepts $w$ after $k$ steps. Also, since every string occurs somewhere in the lexicographic order, there is some integer $\ell$ such that $w$ is string number $\ell$ in the lexicographic ordering. Now, take $n = \max\{k, \ell\}$. Eventually, $M'$ will run $M$ on the first $n$ strings for $n$ steps each (this is because all the previous rounds, running $M$ on the first input for one step, the first two inputs for two steps, etc., all take a finite amount of time to simulate; they only require simulating $1^2 + 2^2 + 3^2 + \cdots + (n-1)^2$ steps of the machine $M$ on various inputs). When this happens, $M'$ will see that $M$ accepts $w$, and will print out $w$. Hence $L(M') = L(M)$, so $A$ is enumerable.

Note that the enumerator $M'$ we constructed actually prints each $w \in L(M)$ infinitely many times, not just once. This is allowed by the definition of an enumerator, so it does not bother us. However, if we wanted to make an enumerator which prints each unique string in the language only once, we could do this as well: we would just modify $M'$ so that it stores a copy of all the strings it prints out, and before printing out each new string it will first compare it to all the strings in storage to ensure it is not a duplicate.

## 17.4 Other examples of Dovetailing

This "dovetailing" trick works whenever you are trying to run infinitely many machines in parallel. This has many possible applications. For example, consider the language

$$A = \{\langle M \rangle : M \text{ is a TM which halts on some input}\}.$$

We show that $A$ is recognizable. How can we construct a Turing machine $M'$ recognizing $A$? Such a machine needs to accept the inputs $\langle M \rangle$ which halt on *some* input string. This means that (at least intuitively) such a machine needs to simulate $M$ on every input string $w \in \Sigma^*$ (where $\Sigma$ is the input alphabet of $M$). However, if our TM $M'$ tries to run the input machine $M$ on even a single input $w$, it runs the risk of $M(w)$ never halting.

The solution is to use dovetailing: the machine $M'$ will simulate $M(w_1)$ on the first string $w_1$ (in lexicographic order) for 1 step, then simulate both $M(w_1)$ and $M(w_2)$ for 2 steps each, then $M(w_1)$, $M(w_2)$, $M(w_3)$ for 3 steps each, and so on. If any of these simulations halt, then $M'$ can immediately stop simulating and accept. Otherwise, if none of the simulations halt, this process will continue forever, so $M'(\langle M \rangle)$ will loop in that case. It is not hard to see that this machine $M'$ (if modified to handled badly-formatted inputs) correctly recognizes the language $A$.

As a side note, is the language $A$ decidable? It should be intuitively clear that $A$ is not decidable, since deciding it requires predicting the halting behavior of Turing machines (and essentially all such tasks are undecidable). To formally prove this, one approach would be to show that $H_{TM} \leq_m A$ by giving a reduction from the halting problem $H_{TM}$ to $A$. We can define such a reduction $f$ by describing a Turing machine $M_f$ computing it. On input $s$, the machine $M_f$ will first check if the input is correctly formatted (formatted like $\langle \langle M \rangle, w \rangle$ for some Turing machine $M$ and string $w$). If not, $M_f$ can output anything not in $A$ (for example, the description of a machine which always loops). If the input is correctly formatted, $M_f$ can build a machine $\langle M_w \rangle$ which hard-codes $w$ into $M$. Given any input $x$, the behavior of $M_w(x)$ will be to first delete the input string $x$ from the tape, then write $w$ on the tape, and then run $M$. Hence $M_w(x)$ behaves like $M(w)$ for every input

$x$. Now, if $M(w)$ halts, then $M_w(x)$ halts on all inputs $x$, and if $M(w)$ loops, then $M_w(x)$ loops on all inputs $x$. Hence $\langle M_w \rangle \in A$ if and only if $\langle \langle M \rangle, w \rangle \in H_{TM}$. As is often the case with reductions, our machine $M_f$ does not actually run $M_w$; it merely constructs the string $\langle M_w \rangle$ out of the string $\langle \langle M \rangle, w \rangle$, modifying the code of the input machine $M$ without actually simulating it. (This was just a sketch of the proof, but hopefully you can see why it holds that $H_{TM} \leq_m A$.)

## 17.5 The range of computable functions

Next, we claim that a language is nonempty and recognizable if and only if it is the range of some computable function. Recall that the range of a function is the set of all things it outputs when given some input; that is, if $f \colon \Sigma^* \to \Gamma^*$, then the range of $f$ is the set $\{f(w) : w \in \Sigma^*\}$.

To prove the claim, we must show two directions: that every nonempty recognizable language is the range of a computable function, and that the range of every computable function is a nonempty recognizable language. We will use the RE characterization of recognizability for both directions.

For the former direction, let $A$ be a nonempty recognizable language. If $A$ is finite, then we can give the strings in $A$ the names $x_1, x_2, x_3, \ldots, x_k$ where $k = |A|$, and then define the function $f$ so that $f(w) = x_{|w|}$ if $|w| \leq k$, and $f(w) = x_1$ otherwise. It is easy to see that the range of $f$ is $A$. To see that $f$ is computable, we construct a TM $M_f$ computing it. This TM will, on input $w$, calculate $|w|$, and then output $x_{|w|}$ if $|w| \leq k$ and $x_1$ otherwise. The machine $M_f$ can do this by hard-coding all the strings $x_1, x_2, \ldots, x_k$ into its code.

The more interesting case is where $A$ is infinite. In that case, since $A$ is recognizable, it is recursively enumerable, so let $M$ be an enumerator with $L(M) = A$. Let $x_i$ be the $i$-th string printed out by $M$. We now define $f$ so that $f(w)$ outputs $x_{|w|}$. Since every string $x \in A$ is eventually printed out by $M$, it must also be the case that $f(w)$ outputs $x$ for some long enough input string $w$. Also, since $M$ never prints out strings not in $A$, the range of $f$ cannot contain strings not in $A$; hence the range of $f$ is exactly $A$. It remains to show that $f$ is computable. To compute it, we construct a Turing machine $M_f$. On input $w$, $M_f$ will compute $|w|$, and then it will simulate the enumerator $M$ until $M$ prints out $|w|$ strings. At that point, $M_f$ will output the last string printed out by $M$ and accept. Note that since the enumerator $M$ eventually prints out every string in $A$, it must eventually print out more than $|w|$ strings, which means that $M_f(w)$ halts (since it only simulates $M$ until the latter prints out $|w|$ strings). Moreover, it is easy to see that $M_f(w)$ outputs $x_{|w|}$, which is equal to $f(w)$. Hence $M_f$ computes $f$, so $f$ is computable.

Finally, we prove the reverse direction: if $f$ is computable, then its range is a nonempty recognizable language. It is clear that since $f$ is a function, it must output something (that is, $f(\epsilon)$ must be a string in $\Gamma^*$). Hence the range of $f$ is nonempty. To show that it is recognizable, we construct an enumerator $M$ such that $L(M)$ is the range of $f$. To do so, let $M_f$ be a Turing machine computing $f$. The enumerator $M$ will go over all the strings $w_1, w_2, w_3, \ldots$ in $\Sigma^*$ in lexicographic order, and for each such string $w_i$, it will simulate $M_f(w_i)$, and then print out the output of $M_f(w_i)$. Since $M_f$ computes $f$, this means that $M$ will print out $f(w_1), f(w_2), f(w_3), \ldots$ where $w_i$ is the $i$-th string in $\Sigma^*$ in lexicographic order. Note that since $M_f$ always halts (since it computes a function), the enumerator $M$ will never be stuck on any one simulation $M_f(w_i)$, and hence it will print out infinitely many strings (though there may be lots of duplicates; they don't have to be infinitely many unique strings). This means that the language $L(M)$ of the enumerator $M$ is the set $\{f(w_1), f(w_2), \ldots\}$. Since the lexicographic order covers all strings in $\Sigma^*$, this set is equal to $\{f(w) : w \in \Sigma^*\}$, which is precisely the range of $f$. Hence the range of $f$ is RE, which means it is recognizable.

## 17.6 Additional closure properties

We have seen that the recognizable languages are closed under union and intersection, but not complement. We have also seen that the decidable languages are closed under union, intersection, and complement, and also that a language is decidable if and only if it is both recognizable and co-recognizable (equivalently, both RE and co-RE).

We now consider closure under concatenation and star. We claim that the recognizable and decidable classes of languages are both closed under concatenation and under star.

Let's start with concatenation. Let $A$ and $B$ be two recognizable languages over alphabet $\Sigma^*$. We claim that $AB$ is recognizable. To show this, we must construct a Turing machine $M$ recognizing $AB$. Let $M_A$ be a TM recognizing $A$ and let $M_B$ be a TM recognizing $B$. Then the machine $M$, when given input $w$, will list out all ways of splitting $w$ into two strings $x$ and $y$ such that $xy = w$. Note that there are finitely many such splits; let's name them $(x_1, y_1), (x_2, y_2), \ldots (x_k, y_k)$, where $k = |w| + 1$. For each of them, $M$ will simulate $M_A(x_i)$ and $M_B(y_i)$. However, it will do this in parallel, running on step of each of the $2k$ machines, then another step of each machine, and so on, keeping track of which ones halt. If ever $M_A(x_i)$ and $M_B(y_i)$ both accept for the same $i$, then $M$ will stop its simulations and accept. Otherwise, it keeps going (if everything rejects, $M$ will reject).

We claim that $M$ recognizes $AB$. To see this, we show two directions. First, if $w \in AB$, then by definition $w = xy$ for some $x \in A$ and $y \in B$. When we run $M$ on $w$, it will simulate $M_A$ and $M_B$ on several pairs of strings, but one such pair will be $(x, y)$. Since $x \in A$ and $y \in B$, we know that $M_A(x)$ and $M_B(y)$ will both accept. This means that $M(w)$ will eventually accept, as desired. In the other direction, suppose $M(w)$ accepts. Then we must have $w = x_i y_i$ for some $x_i$ and $y_i$ where $M_A(x_i)$ and $M_B(y_i)$ both accept. But this means that $x_i \in A$ and $y_i \in B$, so $w = x_i y_i \in AB$, as desired.

This shows that the recognizable languages are closed under concatenation. A similar argument also shows that the decidable languages are closed under concatenation. Indeed, the exact same construction above works; all we need to note is that if $M_A$ and $M_B$ both always halt, it must be the case that $M$ also always halts, and therefore $M$ decides $AB$ rather than merely recognizing it. Hence the decidable languages are also closed under concatenation.

Finally, let's deal with the star operation. Let $A$ be a recognizable language; we wish to show that $A^*$ is recognizable. Let $M_A$ be a TM recognizing $A$; we construct a TM $M$ recognizing $A^*$. On input $w$, the machine $M$ will list out all ways of partitioning $w$ into any number of substrings $x_1, x_2, \ldots, x_\ell$ such that $x_1 x_2 \ldots x_\ell = w$. Since $w$ has finitely many symbols, there are finitely many ways to partition it into such a list of substrings. For each such partition, we will run $M_A$ on each string in the partition. We do all of this in parallel across all partitions. If ever $M_A$ accepts for all strings $x_i$ in a single partition, then $M$ accepts. Otherwise, $M$ keeps simulating (though if all machines reject, $M$ will reject).

Note that if $w \in A^*$, then it must be the case that $w = x_1 x_2 \ldots x_\ell$ for some $x_1, x_2, \ldots, x_\ell \in A$. In this case, $M(w)$ will eventually accept, because $M_A(x_i)$ will accept for all $x_i$ in this partition. Conversely, if $M(w)$ accepts for some string $w$, then for some partition $x_1, x_2, \ldots, x_\ell$ of $w$, it must be the case that $M_A(x_i)$ accepted for each $i$. This means that $x_1, x_2, \ldots, x_\ell \in A$, so $w = x_1 x_2 \ldots x_\ell \in A^*$.

This shows that the recognizable languages are closed under star. Note that if $M_A$ always halts, then $M$ must also always halt; hence the same proof also shows that the decidable languages are closed under star.