Theory of Computation Notes Notes: Recursively Enumerable Languages

Arjun Chandrasekhar

1 Enumerators and RE Languages

Definition 1.1. A **enumerator** is a Turing machine with an attached printer. At any point in the computation, the machine can send a string to the printer to have it printed.

Definition 1.2. Let L be a language and let E be an enumerator. We say E **enumerates** L if E prints out all of the elements of L. If $w \in L$, and if we give the enumerator infinite time, then E will *eventually* print out w. And if $w \notin L$ it will never get printed out.

If L is enumeratd by some language, we say L is **recursively enumerable (RE)**.

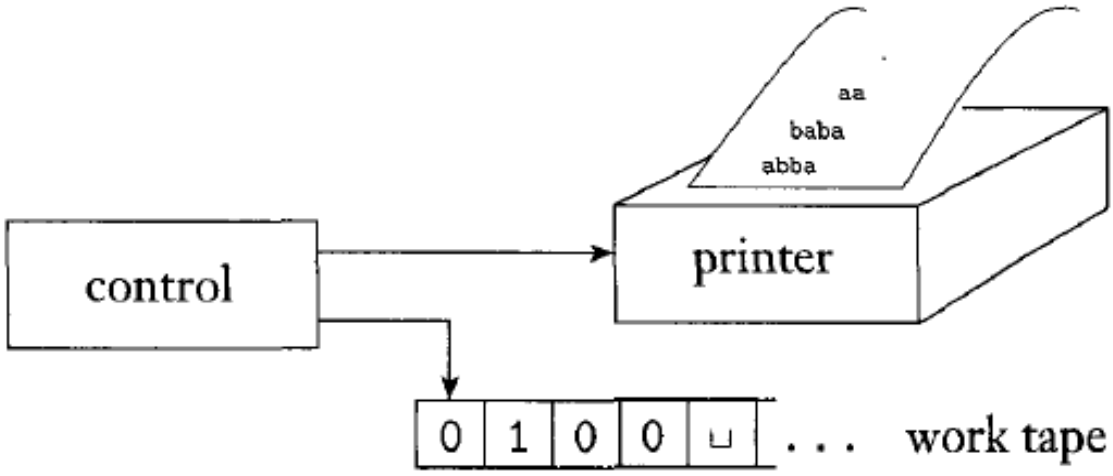

Figure 1: **Schematic illustration of an enumerator.**

2 Equivalence of RE and Recognizable Languages

Theorem 2.1. A language L is Turing-recognizable if and only if L is recursively enumerable.

We will prove this theorem in two parts.

Lemma 2.1. ($\Rightarrow$) If L is recursively enumerable, then L is Turing-recognizable.

Proof. Suppose some machine E enumerates L. We design a machine M to recognize L. On input w, M does the following:

1. Run E to enumerate L
2. If at any point E wants to print out w, M will immediately accept.
3. Otherwise M will run forever

If $w \in L$ then by definition, E must print it out at some point, so M will eventually accept it. If $w \notin L$ then E will never print out w, so M will run forever without accepting. Note that M does not decide L because it runs forever if $w \notin L$. □

Lemma 2.2. ($\Leftarrow$) If L is recognizable, then L is recursively enumerable.

Proof. Suppose some machine M recognizes L. We will design a machine E to enumerate L.

A naive solution would be as follows:

1. Go through each $w_1, w_2, \cdots \in \Sigma^*$
   (a) For each w_i, run M on w
   (b) If M accepts w, print it out
   (c) Move on to the next string

The problem with this method is that M is merely a recognizer, not a decider, for L. So if we encounter a string $w_i \notin L$, it is possible that M will loop on w_i and we never get to test out the rest of the strings.

To get around this problem, we introduce a technique called **dovetailing**. First, let $w_1, w_2, \cdots \in \Sigma^*$ be all the strings on the alphabet. Then, let $S(i, j)$ represent step i of the computation when we run M on string w_j. Then we dovetail M as follows:

1. Run $S(1, 1)$
2. Run $S(1, 2), S(2, 1)$
3. Run $S(1, 3), S(2, 2), S(3, 1)$
4. Run $S(1, 4), S(2, 3), S(3, 2), S(4, 1)$
5. ...

Essentially, we keep a queue of strings that we have started working on. For each string w_i in the queue, we keep track of how far M has progressed on w_i. At each time point, we add the "next" string to the queue; for every string in the queue, we run M for "one more step". We keep doing this as long as we want. In this way, M gets to process every possible string, and it will eventually reach every point in its computation on every string. Figure 2 illustrates this process.

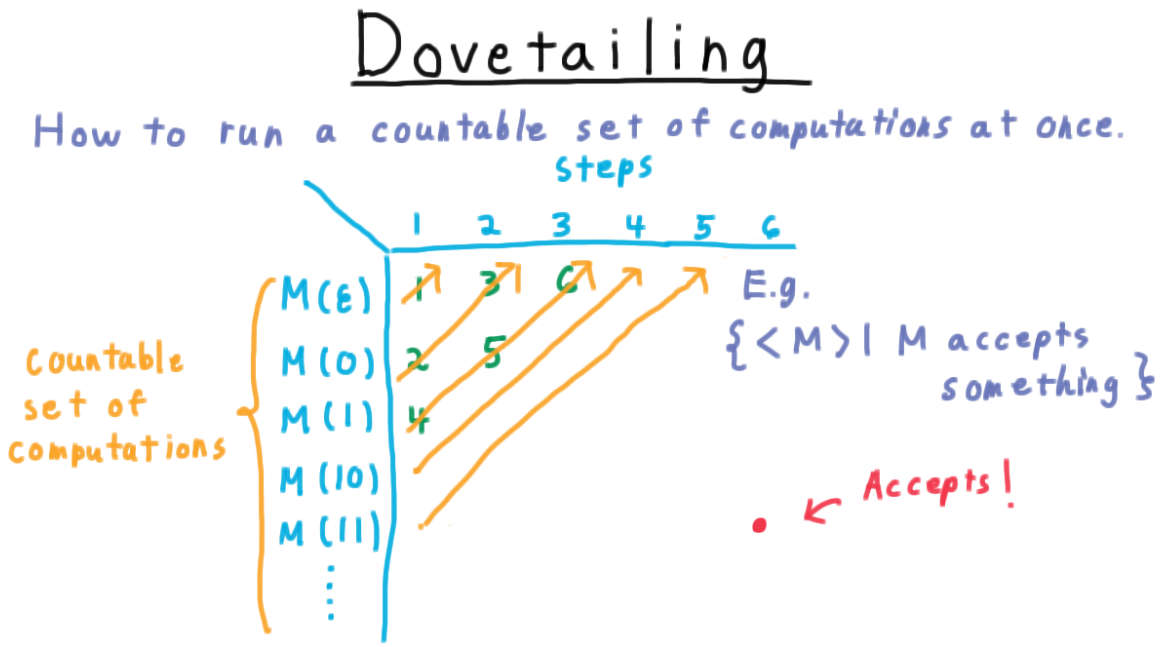

Figure 2: **Illustration of dovetailing technique**

In order to enumerate L, we design an enumerator E as follows:

1. Run M in parallel on every string $w_1, w_2, \ldots \Sigma^*$
2. Whenever M accepts any string w_i, we print it out

Dovetailing ensures that M will process every string eventually. Because M recognizes L, it will only accept w_i if $w_i \in L$. Thus, E will print out all strings in L. □