The Barber of Seville is a classic puzzle. The barber of Seville is said to shave all men who do not shave themselves. So who shaves the barber of Seville? To make this into a puzzle the words have to be treated unduly logically. In particular, one has to interpret it to mean that anyone shaved by the barber of Seville does not shave himself. Then if the barber of Seville shaves himself it is because he does not shave himself and in turn this is because he does shave himself.

One way out of this conundrum occurs if the Barber of Seville is a woman. But our purpose here is to look at how to set up this type of conundrum, or contradiction.

Let us form a table, on one side listing people in their role as people who are shaved, on the other as potential shavers (or barbers). For simplicity, we just name the people 1, 2, … . So the entries in row \( i \) show who is shaved by person \( i \), with entry \((i, j)\) being Y (“yes”) if person \( i \) shaves person \( j \) and N (“no”) otherwise. Let row \( b \) be the row for the barber of Seville. Then, for every \( j \), entries \((b, j)\) and \((j, j)\) are opposite (one Y and one N). This leads to a contradiction for entry \((b, b)\) cannot be opposite to itself. See Figure 1.

![Figure 1: Who Shaves the Barber of Seville?](image)

Now we are ready to show that the halting problem is undecidable by means of a similar argument. We define the halting problem function \( H \) as follows.
\[ H(P, w) = \begin{cases} 
    \text{"Recognize"} & \text{if program } P \text{ halts on input } w \\
    \text{"Reject"} & \text{if program } P \text{ does not halt on input } w 
\end{cases} \]

By halt, we mean that a program completes its computation and stops.

Recall that the program \( P \) and its input are encoded as binary strings. Let \( i \) denote the \( i \)th such string in lexicographic order, namely the order 0, 1, 00, 01, 10, 11, 000, \( \cdots \). Let \( P_i \) denote the \( i \)th string when it is representing a program, and \( w_i \) denote it when it is representing an input. When considering the pair \((P_i, w_j)\), if \( P_i \) is not an encoding of a legal program, or if \( w_j \) is not an encoding of a legal input to \( P_i \), then the output of program \( P_i \) on input \( w_j \) is deemed to be “Fail”.

We begin by showing that there is no program computing the following partial function \( \text{Diag} \), or \( D \) for short. We define \( D(w_i) \) to have the opposite behavior to \( P_i(w_i) \). In other words, \( D \) on the input with encoding \( i \) does the opposite of the program with encoding \( i \) on the input with encoding \( i \). Now suppose that \( D \) were the \( d \)th program in the lexicographic listing. Then \( D \) on the input with encoding \( d \) would do the opposite of the program with encoding \( d \) (which is \( D \)) on the input with encoding \( d \), and this is not possible. See Figure 2.

![Figure 2: There is no Program to Compute D.](image)

More precisely, we define \( D \) as follows.

\[ D(P_i) = \begin{cases} 
    \text{output "Done"} & \text{if } P_i \text{ runs forever on input } w_i \\
    \text{loop forever} & \text{if } P_i \text{ on input } w_i \text{ eventually halts} 
\end{cases} \]

Now suppose for a contradiction that \( D \) were computed by \( P_d \), and consider \( P_d(w_d) \).

\[ P_d(w_d) = \begin{cases} 
    \text{output "Done"} & \text{if } P_d \text{ runs forever on input } w_d \\
    \text{loop forever} & \text{if } P_d \text{ on input } w_d \text{ halts} 
\end{cases} \]
But this is a contradiction. Hence there is no program to compute the function \( D \); that is, there is no program that can determine whether an arbitrary (one-input) program halts when fed its own encoding as an input.

**Theorem 1** There is no (2-input) program that computes \( H \), the halting function.

**Proof:** Suppose that \( P_H \) were a program that computed \( H \). Then the Program \( P_D \) defined by \( P_D(w_i) = P_H(P_i, w_i) \) would compute the function \( D \). But we have shown there is no such program \( P_D \). Consequently, there is no Program \( P_H \) either. \( \blacksquare \)

The technique we have just used is called diagonalization. It was first developed to show that the real numbers are not countable. We will show this result next.

**Definition 2** A set \( A \) is countable if it is finite or if there is a function \( f \) mapping \( N \), the positive integers, onto \( A \), where \( f \) is a bijection (one-to-one and onto).

In other words, \( A \) can be listed by the function \( f \) \((f(1) = a_1 \text{ is the first item in } A, f(2) = a_2 \text{ is the second item, and so on})\).

If this listing/mapping can be carried out by a program, \( A \) is said to be recursively enumerable. As we will see later, the halting set \( H = \{ \langle P, w \rangle \mid P \text{ halts on input } w \} \) is recursively enumerable.

**Lemma 3** The real numbers are not countable.

**Proof:** We will show that the real numbers in the range \([0, 1]\) are not countable. This suffices to show the result, for given a listing of all the real numbers, one could go through the list, forming a new list of the reals in the range \([0, 1]\).

For a contradiction, suppose that there were a listing of the real numbers in the range \([0, 1]\), \( r_1, r_2, \cdots \), say. Imagine that each real number is written as an infinite decimal: \( r_i = 0.r_{i1}r_{i2} \cdots \), when each \( r_{ij} \) is a digit \((r_{ij} \in \{0, \cdots, 9\})\). Note that \( 1 = 0.999 \cdots \).

We create a new decimal, \( d = d_1d_2 \cdots \), such that \( d \in [0, 1] \) yet \( d \neq r_i \) for all \( i \); so the listing of reals in the range \([0, 1]\) would not include \( d \); but it must do so as it is a listing of all the reals in this range. This is a contradiction, which shows that the reals are not countable.

It remains to define \( d \), which we do as follows:

\[ d_i = r_i + 2 \text{ mod } 10 \quad \text{for all } i. \]

Suppose that \( d = r_j \) for some \( j \). As \( d_j \neq r_{jj} \), the only way \( d \) and \( r_j \) could be equal is if one of them had the form \( 0.sx99 \cdots \) and the other had the form \( 0.s(x + 1)00 \cdots \), where \( x \) is a single digit and \( s \) is a string of digits. But as the shift on the \( j \)th digit is by 2 this is not possible.

Again, this is a construction by diagonalization, with the fact that \( 0.99 \cdots = 1.00 \cdots \) creating a small complication. The construction of \( d \) is illustrated in Figure 3. \( \blacksquare \)
By contrast the rationals are countable. By a rational we mean a ratio $a/b$ where $a$ and $b$ are positive integers, but not necessarily coprime. So for the purposes of listing the rationals we will view $a/b$ and $2a/2b$ as distinct rationals (it is a simple exercise to modify the listing function to eliminate such duplicates).

Lemma 4 *The rationals are countable.*

**Proof:** The rationals can be displayed in a 2-D table, as shown in Figure 4. The rows and columns are indexed by the positive integers, and entry $(a, b)$ represents rational $a/b$. So the task reduces to listing the table entries, which is done by going through the forward diagonals, one by one, in the order of increasing $a + b$. That is, first the entry, $(1, 1)$, with $a + b$ value 1 is listed, then the entries $(2, 1)$, $(1, 2)$, entries with $a + b$ value 2 are listed, then the entries $(3, 1)$, $(2, 2)$, $(1, 3)$, entries with $a + b$ value 3 are...
listed, and so forth. Within a diagonal, the entries are listed in the order given by increasing the second coordinate.

Clearly every rational is listed eventually. \((a, b)\) will be the \((a + b − 1)(a + b − 2)/2 + b\)th item listed, in fact.) Also every rational is listed exactly once. Thus the rationals are countable. ■

The same idea can be used to list the items in a \(d\)-dimensional table where each coordinate is indexed by the positive integers. Let the coordinate names be \(x_1, x_2, \cdots, x_d\), respectively. Then, in turn, the entries with \(x_1 + x_2 + \cdots + x_d = k\), for \(k = d, d + 1, d + 2, \cdots\) are listed. For a given value of \(k\), in turn, the entries with \(x_d = 1, 2, \cdots, k - d + 1\) are listed recursively.

So for \(d = 3\) the listing begins \((1,1,1), (2,1,1), (1,2,1), (1,1,2), (3,1,1), (2,2,1), (1,3,1), (2,1,2), \ldots\). We call this the \textit{diagonal listing}, and we use it to show that \(H\) is recursively enumerable.

\textbf{Lemma 5} \(H\) is recursively enumerable.

\textbf{Proof:} The listing program explores a 3-dimensional table in the diagonal listing order. At the \((i, j, k)\)th table entry, it simulates program \(P_i\) on input \(w_j\) for \(k\) steps. If \(P_i(w_j)\) runs to completion in exactly \(k\) steps then the listing program outputs the encoding \(⟨i, j⟩\) of the pair \((i, j)\).

Clearly, if \(P_i\) halts on input \(w_j\), then \(⟨i, j⟩\) occurs in this listing, and further it occurs exactly once. As this listing is produced by a program it follows that \(H\), the set listed, is recursively enumerable. ■

Next, we relate recursive enumerability and decidability.

\textbf{Lemma 6} If \(L\) and \(\overline{L}\) are both recursively enumerable then \(L\) is decidable.

\textbf{Proof:} It suffices to give an algorithm \(A_L\) to decide \(L\). \(A_L\) will use the listing procedures for \(L\) and \(\overline{L}\). Let \(\text{List}_L(x)\) be the program that on input \(x\) returns the \(x\)th item in a listing of \(L\), and let \(\text{List}_{\overline{L}}(x)\) be the analogous program with respect to \(\overline{L}\). Then, on input \(w\), \(A_L\) simply runs \(\text{List}_L(x)\) and \(\text{List}_{\overline{L}}(x)\) for increasing values of \(x\) \((x = 1, 2, \cdots\) in turn\) until one of them returns the value \(w\), thereby showing in which set \(w\) occurs. At this point, \(A_L\) outputs “Recognize” or “Reject”, as appropriate. In more detail, \(A_L\) is the following program.

\[A_L(w)\]

\[\text{found} ← \text{FALSE}; \ x ← 1\]
\[\text{while} \ (\text{not found}) \ \text{do}\]
\[w_L ← \text{List}_L(x)\]
\[w_{\overline{L}} ← \text{List}_{\overline{L}}(x)\]
\[\text{if} \ w = w_L \ \text{or} \ w = w_{\overline{L}}\]
\[\text{then} \ \text{found} ← \text{TRUE}\]
\[\text{else} \ x ← x + 1\]
\[\text{end while}\]
if \( w = w_L \) then return(“Recognize”) 
else return(“Reject”) 

We can now show that \( \overline{H} \) is not recursively enumerable.

**Lemma 7** \( \overline{H} \) is not recursively enumerable.

**Proof:** Recall that \( H \) is recursively enumerable (by Lemma 5). Were \( \overline{H} \) also recursively enumerable, then, by Lemma 6, \( H \) would be decidable, which is not the case (by Theorem 1). This shows that \( \overline{H} \) cannot be recursively enumerable. (Strictly, this was a proof by contradiction.) ■

Notice that membership and non-membership are not symmetric in a recursively enumerable but non-decidable set such as \( H \). While membership can be demonstrated simply by listing the set and encountering the item, there is no test for non-membership.