Here we look at some alternative models of computation. All of these have the same computational power as Turing machines—that is, they can compute anything that can be computed (subject to some restrictions about how inputs and outputs are encoded). This implies that the halting problem for each of these models is undecidable, and that there is a ‘universal program’ for each model.

1 Counter machines

Look at the flowchart diagrams in Figure 1.

The left diagram operates on a pair \((a, b)\) of nonnegative integers. The instruction \(\text{dec} a\) means ‘decrease \(a\) by 1’, or, as the CS types prefer to say ‘decrement \(a\)’, and \(\text{inc} a\) means ‘increase \(a\) by 1’ (increment). Which instruction to execute next is determined by the arrows. As you can see, there is a special branching instruction that tests to see if a value is 0, branching to different next instructions depending on the result of the test. By the way, if you decrement a value that is already 0, the result is taken to be 0, not minus 1. These diagrams live in a world in which there are no negative numbers.

Think of the names \(a\) and \(b\) as variables, or counters that hold values. If you start with \(x\) as the value of \(a\) and \(y\) as the value of \(b\), then the value of \(a\) when the halt instruction is reached will be \(x + y\). Thus this flowchart gives an algorithm for adding two natural numbers. You can think of this as a machine whose ‘state’ is the next instruction to execute: At each step, the machine either decrements or increments one of the counters, or tests whether a counter is zero, and changes to
a new state. Alternatively, you can think of the flowchart as the program shown in Figure 2. This programming language has statements to increment and decrement variables, to branch conditionally to another statement if the value of a variable is zero, and to jump unconditionally to another statement. Labels (in this program loop and finish) are used as targets of the branch instructions.

What about our second flowchart? Table 1 traces the values of the counters $a$ and $b$ if the initial value of $a$ is 5. (We will assume that the counter $b$ is initially set to 0.) The variable $a$ is decremented twice in each pass through the main loop, and the variable $b$ is increased once for each such pass. So $b$ is keeping count of

```
loop:
goto finish if b=0
dec b
inc a
 goto loop
finish:
halt
```

Figure 2: A program view of the addition counter machine.
the number of times we can reduce $a$ by 2 before falling to 0. If $a$ falls to zero in the middle of the loop (after 1 rather than 2 decrements) then we set $a$ back to 1. Thus, when we reach the halt instruction, $a$ will hold the remainder of the original value modulo 2—that is, $a$ will be 0 if the original value of $a$ was even, and 1 if it was odd. The counter $b$ will hold the integer part of the quotient of the original value upon division by 2. So this program computes the function

$$x \mapsto (x \mod 2, \lfloor x/2 \rfloor).$$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Trace of the execution of the second counter machine on inputs $a = 5$ and $b = 0$. The result is $5 \mod 2$ in $a$ and $\lfloor 5/2 \rfloor$ in $b$. The analogous result will occur for any initial value of the counter $a$, so this machine computes the quotient and remainder of $a$ upon division by 2.

Every counter machine begins with an initial setting of several counters (in the case of our first example, two counters $a$ and $b$, and in the case of the second, one counter $a$) and finishes with a possibly different setting of several of the counters. (In our first example, $a$ was the only ‘output’ counter, but in our second, both $a$ and $b$ hold results we are interested in). Thus our counter machine computes a partial function

$$f : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N} \times \cdots \times \mathbb{N}.$$  

Why partial? Because like any programming environment, counter machines allow the possibility of infinite loops, so the machine may fail to halt on some inputs.
Theorem 1  Any partial function computed by a Turing machine can be computed by a counter machine.

We will see later why this is true. Thus anything that can be computed, can be computed by a counter machine. This has to be qualified a bit: Counter machines deal with tuples of natural numbers, not with strings, but strings, of course, can be encoded by numbers. A Turing machine that computes a partial function

\[ f : \Sigma_1^* \rightarrow \Sigma_2^* \]

is identified with a function

\[ \hat{f} : \mathbb{N} \rightarrow \mathbb{N}, \]

where \( \hat{f} \) maps the integer encoding a string \( w \) to the integer encoding the string \( f(w) \). It is this function \( \hat{f} \) that the counter machine computes.

We will see later why this theorem is true. All the usual consequences of universal computation follow from this: for example, there is no algorithm to tell whether a given counter machine halts on a given input.

2  A tiny subset of Python

Here we present a tiny subset of the programming language Python that is still computationally universal—in other words, it can do everything a Turing machine can do. Just as with counter machines, these programs only operate on tuples of natural numbers, so the claim about equivalence to the Turing machine model presupposes that all inputs and outputs are encoded by integers. After deep reflection, I have decided to name this programming language Rubber Boa. \(^1\)

Our programming language will be used to define functions, so we will include Python’s syntax for introducing function definitions:

```
def f(x, y, z):
```

Of course, you can have any number of arguments, and the argument names are arbitrary.

We also include Python’s syntax for returning a function value:

```
def f(x, y, z):
    return x + y + z
```

\(^1\)I wanted to call this language ‘Python with both hands tied behind its back’, but quickly realized that the metaphor was zoologically inaccurate. I consulted Wikipedia and found out that the smallest constrictor snake in the world is the Rubber Boa, cute little guys that don’t get more than two feet long.
return v

where v is a variable.

We also allow statements of the following type (where u and v are variables):

\[
\begin{align*}
v &= 0 \\
u &= v \\
u &= u + 1 \\
v &= f(u, w, g(x, y))
\end{align*}
\]

The third of these statements increases the value of u by 1. The fourth is meant as just an example of what is intended: f and g are assumed to be be previously-defined functions, and these can be composed in any fashion you wish, as long as the ultimate arguments are variables. We do not allow the function being defined to be used in the function definition—that is, we do not allow recursion.

We have to have some way of altering the flow of control, so we allow the following looping constructs from Python:

\[
\begin{align*}
\text{for } u \text{ in range}(v): \\
&\quad \text{SEQUENCE OF STATEMENTS}
\end{align*}
\]

\[
\begin{align*}
\text{while } u \neq 0: \\
&\quad \text{SEQUENCE OF STATEMENTS}
\end{align*}
\]

The first of these repeatedly executes the sequence of statements in the body as u takes on the values 0, 1, ..., v − 1. The second repeatedly executes the statements in the body as long as the value of u is nonzero.² Figure 3 shows several functions written in this language. These demonstrate how to add, multiply, test for zero, and decrement in Rubber Boa.

Just as with counter machines, programs in Rubber Boa compute partial functions

\[f : \underbrace{\mathbb{N} \times \cdots \times \mathbb{N}}_{k \text{ times}} \to \underbrace{\mathbb{N} \times \cdots \times \mathbb{N}}_{\ell \text{ times}}.\]

and the reason these are partial is that the while statement might never terminate. And just as with counter machines, this programming language can do anything any programming language can do:

²What, you might be wondering (unless you are a Python expert) happens if I change the value of v inside the for statement? The answer is that the body of the loop is still executed \(k - 1\) times, where \(k\) is the value of v when execution started.
def add(x, y):
    u = x
    for j in range(y):
        u += 1
    return u

def mult(x, y):
    u = 0
    for j in range(x):
        u = add(u, y)
    return u

#return 1 if x is zero, 0 otherwise
def iszero(x):
    u = 1
    for j in range(x):
        u = 0
    return u

#proper decrement. pred(0) is 0
def pred(x):
    z = 0
    for j in range(x):
        z = j
    return z

Figure 3: How to add, multiply, test for zero, and decrement in Rubber Boa.
Theorem 2 Any partial function computed by a Turing machine can be computed by a Rubber Boa program.

You may have observed that our examples in Figure 3 do not use the while statement. This is no accident—most of what you want to do with computer programs can be done without while.

2.1 Aside: primitive recursion and min-recursion

The Rubber Boa programming language actually predates the Turing machine as a general model of computation. Of course, there being no computers back then, much less computer programming languages, they were described somewhat differently. Let’s construct a family $F$ of functions

$$f : \mathbb{N}^k \to \mathbb{N},$$

where we use $\mathbb{N}^k$ to mean the cartesian product of $k$ copies of $\mathbb{N}$. The rules are, that $F$ contains the zero function and the successor function, defined by:

$$\text{zero}(x) = 0, \text{succ}(x) = x + 1$$

for all $x \in \mathbb{N}$. $F$ contains the projection functions

$$\pi_i^k(x_1, \ldots, x_k) = x_i$$

for all $n \geq 1$ and all $1 \leq i \leq n$, and is closed under composition. Closure under composition means, for example, that if

$$f : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

is in $F$, and the functions

$$g_1, g_2, g_3 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

are in $F$, then $F$ contains the function

$$h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

defined by

$$h(x, y) = f(g_1(x, y), g_2(x, y), g_3(x, y)).$$
\( F \) is also closed under an operation called *primitive recursion*. This means that if \( F \) contains functions
\[
f : \mathbb{N}^k \to \mathbb{N},
\]
and
\[
g : \mathbb{N}^{k+2} \to \mathbb{N},
\]
then it contains the function
\[
h : \mathbb{N}^{k+1} \to \mathbb{N}
\]
defined by
\[
h(x_1, \ldots, x_k, 0) = f(x_1, \ldots, x_k),
\]
and
\[
h(x_1, \ldots, x_k, y + 1) = g(x_1, \ldots, x_k, y, h(x_1, \ldots, x_k, y)).
\]
If this looks odd, think of it in Rubber Boa:

```python
def h(x, y, z):
    u = f(x, y)
    for v in range(z):
        u = g(x, y, v, u)
    return u
```

The functions in \( F \) constructed by repeated application of these rules are called *primitive recursive functions*. They are exactly the functions that you can define in Rubber Boa without using the `while` statement. You obtain the *recursive* functions by closing up under one more operation, called min-recursion or \( \mu \)-recursion, which adds back the power of `while`: If
\[
f : \mathbb{N}^{k+1} \to \mathbb{N}
\]
is in \( F \), then so is the partial function
\[
g : \mathbb{N}^k \to \mathbb{N}
\]
defined by
\[
g(x_1, \ldots, x_k) = \min \{ y : f(x_1, \ldots, x_k, y) = 0 \}.
\]
This corresponds to
def g(x, y, z):
    u = f(x, y, z, 0)
    v = 0
    while u != 0:
        v += 1
        u = f(x, y, z, v)
    return v

3 FRACTRAN–the strangest programming language

FRACTRAN was invented by the mathematician John H. Conway. The idea seems to date from the 1960’s, although Conway’s paper on FRACTRAN programming did not appear until much later.\(^3\)

As Conway writes, the entire FRACTRAN syntax can be learned in a period of ten seconds. Actually two seconds. A FRACTRAN program is a sequence of fractions, such as

\[
\begin{pmatrix}
2 \\
3
\end{pmatrix}
\] or

\[
\begin{pmatrix}
11 & 5 & 3 & 7 \\
14 & 7 & 11 & 2
\end{pmatrix}.
\]

You’ll need another five seconds or so to learn the semantics. A FRACTRAN program is given as input a positive integer \(N\). At each step, we find the first fraction \(f\) in the sequence such that \(Nf\) is an integer, and replace \(N\) by \(Nf\). We continue like this until no such fraction \(f\) can be found.

As an example, let’s run the first program with input \(N = 144\):

\[
\begin{align*}
144 & \quad \rightarrow \times \frac{2}{3} \quad 96 \\
    & \quad \rightarrow \times \frac{2}{3} \quad 64.
\end{align*}
\]

\(^3\)Conway was the first speaker in the Mathematics Department’s Distinguished Lecturer Series. One of his lectures was on FRACTRAN.
And the second with input $N = 32$:

$$
32 \rightarrow x^7_2 112 \\
\rightarrow x^{11}_7 88 \\
\rightarrow x^{11}_2 24 \\
\rightarrow x^7_2 84 \\
\rightarrow x^{11}_7 66 \\
\rightarrow x^{11}_2 18 \\
\rightarrow x^7_2 63 \\
\rightarrow x^7_2 45
$$

So what? What is being ‘computed’ here? What I didn’t tell you is how the inputs and outputs are encoded. The first program is supposed to take two inputs $x, y$ and encode them as $2^x 3^y$. The output is supposed to be $z$, where $2^z$ is the value on which the program halts. In our example, $144 = 2^4 \cdot 3^2$, and the final result is $64 = 2^6$. You can see that if I repeatedly multiply $2^x \cdot 3^y$ by $\frac{7}{11}$, I will wind up with $2^{x+y}$. So our first FRACTRAN program adds two integers.

What about the second? The representation of inputs and outputs here is to encode a single input $x$ as $2^x$, and to decode the result, which will have the form $3^y \cdot 5^z$, as $(y, z)$. In our example, $x = 5, y = 2$ and $z = 1$. In this case, $y$ and $z$ are respectively the quotient and remainder that $x$ leaves upon division by 2, and in fact that is what the second program will give you for every initial value $x$.

Subject to these rules about the encoding of inputs and outputs, every Turing-computable function can be computed by a FRACTRAN program, so here is another universal model of computation.

How were these FRACTRAN programs arrived at? There is relatively simple trick for converting counter machine programs into FRACTRAN programs. (This is one of the project topics, and I will leave the details to any students who choose this topic.)

4 Why are these models universal?

Let’s briefly justify our claim that any Turing machine can be simulated by a program in one of these models. We’ll focus on the counter machine model. The trick is to encode the configurations of a Turing machine by pairs of integers,
much as we did in the discussion of Gödel numbering in the previous lecture. To be more concrete about it, let’s suppose we have a Turing machine with tape alphabet \( \{a, b, X, □\} \). We encode these symbols by pairs of bits, reserving the bit sequence 00 to encode the blank symbol. So, for example, we might have

\[
\begin{align*}
00 & \leftrightarrow □ \\
01 & \leftrightarrow a \\
10 & \leftrightarrow b \\
11 & \leftrightarrow X \\
\end{align*}
\]

We can encode the tape contents together with the position of the read-write head by a pair of integers. For instance, if the tape looks like

\[
\cdots □□abXaaX□□\cdots,
\]

where the position of the head is indicated by the bold-faced a, then we get the two bit sequences

\[
\cdots 000001101101, 01110000\cdots.
\]

The first is the binary encoding of 109, and the second the reversal of the binary encoding of 14. So this combination of tape contents and head position is encoded by the pair of integers \((109, 14)\).

Turing machine instructions all look something like this:

In state \( p \), reading \( a \), write \( X \), move left, and go to state \( q \).

Saying that the head is positioned over \( a \) is the same as saying that the last two bits of the first component of the encoding are 01. So if we represent the encoding by \((x, y)\), this is \( x \mod 4 = 1 \). Lopping off the last symbol of the left portion of the tape (which is one of the things that happens when we shift the head left) corresponds to dividing by 4 and throwing away the remainder. Adjoining \( X \) to the right-hand portion of the tape corresponds to multiplication by 4 and adding 3. So the Turing machine instruction can be expressed as:

\[
p: \text{if } x \mod 4 \text{ is } 1,
\]

\[
(x, y) \mapsto \left(\left\lfloor \frac{x}{4} \right\rfloor, 4y + 3\right)
\]

goto q
Now this is very close to our language for counter machines. We have already seen how to use these machines to add and find quotients and remainders mod 2. The same ideas can be used to find quotients and remainders mod 4, and there is a simple trick for testing if a counter is equal to 1. So we can translate each Turing machine instruction into a fragment of counter machine code.