Chapter udf

Turing Machine Computations

mac.1 Introduction

tur:mac:int:sec

What does it mean for a function, say, from $\mathbb{N}$ to $\mathbb{N}$ to be *computable*? Among the first answers, and the most well known one, is that a function is computable if it can be computed by a Turing machine. This notion was set out by Alan Turing in 1936. Turing machines are an example of *a model of computation*—they are a mathematically precise way of defining the idea of a "computational procedure." What exactly that means is debated, but it is widely agreed that Turing machines are one way of specifying computational procedures. Even though the term "Turing machine" evokes the image of a physical machine with moving parts, strictly speaking a Turing machine is a purely mathematical construct, and as such it idealizes the idea of a computational procedure. For instance, we place no restriction on either the time or memory requirements of a Turing machine: Turing machines can compute something even if the computation would require more storage space or more steps than there are atoms in the universe.

explanation

It is perhaps best to think of a Turing machine as a program for a special kind of imaginary mechanism. This mechanism consists of a *tape* and a *read-write head*. In our version of Turing machines, the tape is infinite in one direction (to the right), and it is divided into *squares*, each of which may contain a symbol from a finite *alphabet*. Such alphabets can contain any number of different symbols, say, but we will mainly make do with three: $\triangleright$, 0, and 1. When the mechanism is started, the tape is empty (i.e., each square contains the symbol 0) except for the leftmost square, which contains $\triangleright$, and a finite number of squares which contain the *input*. At any time, the mechanism is in one of a finite number of *states*. At the outset, the head scans the leftmost square and in a specified *initial state*. At each step of the mechanism's run, the content of the square currently scanned together with the state the mechanism is in and the Turing machine program determine what happens next. The Turing machine program is given by a partial function which takes as input a state q and a symbol σ and outputs a triple $\langle q', \sigma', D\rangle$. Whenever the mechanism is in

tur:mac:int:
fig:tm

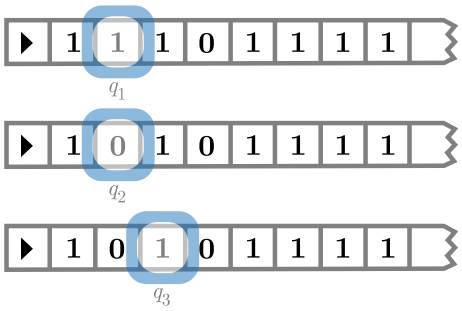


Figure mac.1: A Turing machine executing its program.

state q and reads symbol σ, it replaces the symbol on the current square with σ', the head moves left, right, or stays put according to whether D is L, R, or N, and the mechanism goes into state q'.

For instance, consider the situation in section mac.1. The visible part of the tape of the Turing machine contains the end-of-tape symbol $\triangleright$ on the leftmost square, followed by three 1's, a 0, and four more 1's. The head is reading the third square from the left, which contains a 1, and is in state q_1—we say "the machine is reading a 1 in state q_1." If the program of the Turing machine returns, for input $\langle q_1, 1\rangle$, the triple $\langle q_2, 0, N\rangle$, the machine would now replace the 1 on the third square with a 0, leave the read/write head where it is, and switch to state q_2. If then the program returns $\langle q_3, 0, R\rangle$ for input $\langle q_2, 0\rangle$, the machine would now overwrite the 0 with another 0 (effectively, leaving the content of the tape under the read/write head unchanged), move one square to the right, and enter state q_3. And so on.

We say that the machine *halts* when it encounters some state, q_n, and symbol, σ such that there is no instruction for $\langle q_n, \sigma\rangle$, i.e., the transition function for input $\langle q_n, \sigma\rangle$ is undefined. In other words, the machine has no instruction to carry out, and at that point, it ceases operation. Halting is sometimes represented by a specific halt state h. This will be demonstrated in more detail later on.

digression

The beauty of Turing's paper, "On computable numbers," is that he presents not only a formal definition, but also an argument that the definition captures the intuitive notion of computability. From the definition, it should be clear that any function computable by a Turing machine is computable in the intuitive sense. Turing offers three types of argument that the converse is true, i.e., that any function that we would naturally regard as computable is computable by such a machine. They are (in Turing's words):

1. A direct appeal to intuition.

2. A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

3. Giving examples of large classes of numbers which are computable.

Our goal is to try to define the notion of computability "in principle," i.e., without taking into account practical limitations of time and space. Of course, with the broadest definition of computability in place, one can then go on to consider computation with bounded resources; this forms the heart of the subject known as "computational complexity."

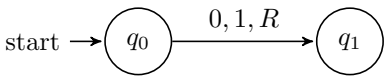
Historical Remarks Alan Turing invented Turing machines in 1936. While his interest at the time was the decidability of first-order logic, the paper has been described as a definitive paper on the foundations of computer design. In the paper, Turing focuses on computable real numbers, i.e., real numbers whose decimal expansions are computable; but he notes that it is not hard to adapt his notions to computable functions on the natural numbers, and so on. Notice that this was a full five years before the first working general purpose computer was built in 1941 (by the German Konrad Zuse in his parent's living room), seven years before Turing and his colleagues at Bletchley Park built the code-breaking Colossus (1943), nine years before the American ENIAC (1945), twelve years before the first British general purpose computer—the Manchester Small-Scale Experimental Machine—was built in Manchester (1948), and thirteen years before the Americans first tested the BINAC (1949). The Manchester SSEM has the distinction of being the first stored-program computer—previous machines had to be rewired by hand for each new task.

tms.2 Representing Turing Machines

tms:tms:rep:sec

explanation

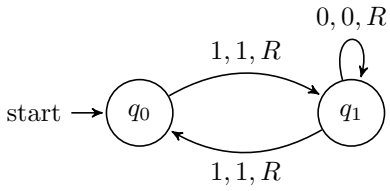
Turing machines can be represented visually by *state diagrams*. The diagrams are composed of state cells connected by arrows. Unsurprisingly, each state cell represents a state of the machine. Each arrow represents an instruction that can be carried out from that state, with the specifics of the instruction written above or below the appropriate arrow. Consider the following machine, which has only two internal states, q_0 and q_1, and one instruction:



Recall that the Turing machine has a read/write head and a tape with the input written on it. The instruction can be read as *if reading a blank in state q_0, write a stroke, move right, and move to state q_1*. This is equivalent to the transition function mapping $\langle q_0, 0\rangle$ to $\langle q_1, 1, R\rangle$.

Example tms.1. *Even Machine*: The following Turing machine halts if, and

only if, there are an even number of strokes on the tape.



The state diagram corresponds to the following transition function:

$$\delta(q_0, 1) = \langle q_1, 1, R\rangle,$$
$$\delta(q_1, 1) = \langle q_0, 1, R\rangle,$$
$$\delta(q_1, 0) = \langle q_1, 0, R\rangle$$

explanation

The above machine halts only when the input is an even number of strokes. Otherwise, the machine (theoretically) continues to operate indefinitely. For any machine and input, it is possible to trace through the *configurations* of the machine in order to determine the output. We will give a formal definition of configurations later. For now, we can intuitively think of configurations as a series of diagrams showing the state of the machine at any point in time during operation. Configurations show the content of the tape, the state of the machine and the location of the read/write head.

Let us trace through the configurations of the even machine if it is started with an input of 4 1s. In this case, we expect that the machine will halt. We will then run the machine on an input of 3 1s, where the machine will run forever.

The machine starts in state q_0, scanning the leftmost 1. We can represent the initial state of the machine as follows:

$$\triangleright 1_0 1110\ldots$$

The above configuration is straightforward. As can be seen, the machine starts in state one, scanning the leftmost 1. This is represented by a subscript of the state name on the first 1. The applicable instruction at this point is $\delta(q_0, 1) = \langle q_1, 1, R\rangle$, and so the machine moves right on the tape and changes to state q_1.

$$\triangleright 11_1 110\ldots$$

Since the machine is now in state q_1 scanning a stroke, we have to "follow" the instruction $\delta(q_1, 1) = \langle q_0, 1, R\rangle$. This results in the configuration

$$\triangleright 111_0 10\ldots$$

As the machine continues, the rules are applied again in the same order, resulting in the following two configurations:

$$\triangleright 1111_1 0\ldots$$

$$\triangleright 11110_0 \ldots$$

The machine is now in state q_0 scanning a blank. Based on the transition diagram, we can easily see that there is no instruction to be carried out, and thus the machine has halted. This means that the input has been accepted.

Suppose next we start the machine with an input of three strokes. The first few configurations are similar, as the same instructions are carried out, with only a small difference of the tape input:

$$\triangleright 1_0 110 \ldots$$

$$\triangleright 11_1 10 \ldots$$

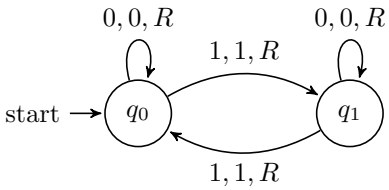
$$\triangleright 111_0 0 \ldots$$

$$\triangleright 1110_1 \ldots$$

The machine has now traversed past all the strokes, and is reading a blank in state q_1. As shown in the diagram, there is an instruction of the form $\delta(q_1, 0) = \langle q_1, 0, R \rangle$. Since the tape is infinitely blank to the right, the machine will continue to execute this instruction *forever*, staying in state q_1 and moving ever further to the right. The machine will never halt, and does not accept the input.

It is important to note that not all machines will halt. If halting means that the machine runs out of instructions to execute, then we can create a machine that never halts simply by ensuring that there is an outgoing arrow for each symbol at each state. The even machine can be modified to run infinitely by adding an instruction for scanning a blank at q_0. explanation

Example tms.2.



Machine tables are another way of representing Turing machines. Machine tables have the tape alphabet displayed on the x-axis, and the set of machine states across the y-axis. Inside the table, at the intersection of each state and symbol, is written the rest of the instruction—the new state, new symbol, and direction of movement. Machine tables make it easy to determine in what state, and for what symbol, the machine halts. Whenever there is a gap in the table is a possible point for the machine to halt. Unlike state diagrams and instruction sets, where the points at which the machine halts are not always immediately obvious, any halting points are quickly identified by finding the gaps in the machine table. explanation

Example tms.3. The machine table for the even machine is:

	0	1
q_0		$1, q_1, R$
q_1	$0, q_1, 0$	$1, q_0, R$

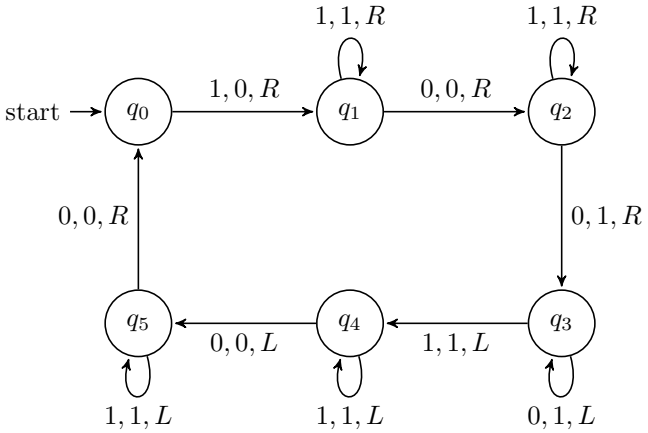
As we can see, the machine halts when scanning a blank in state q_0.

explanation

So far we have only considered machines that read and accept input. However, Turing machines have the capacity to both read and write. An example of such a machine (although there are many, many examples) is a *doubler*. A doubler, when started with a block of n strokes on the tape, outputs a block of $2n$ strokes.

Example tms.4. Before building a doubler machine, it is important to come up with a *strategy* for solving the problem. Since the machine (as we have formulated it) cannot remember how many strokes it has read, we need to come up with a way to keep track of all the strokes on the tape. One such way is to separate the output from the input with a blank. The machine can then erase the first stroke from the input, traverse over the rest of the input, leave a blank, and write two new strokes. The machine will then go back and find the second stroke in the input, and double that one as well. For each one stroke of input, it will write two strokes of output. By erasing the input as the machine goes, we can guarantee that no stroke is missed or doubled twice. When the entire input is erased, there will be $2n$ strokes left on the tape.

tms:tms:rep:ex:doubler



Problem tms.1. Choose an arbitary input and trace through the configurations of the doubler machine in Example tms.4.

Problem tms.2. The double machine in Example tms.4 writes its output to the right of the input. Come up with a new method for solving the doubler problem which generates its output immediately to the right of the end-of-tape marker. Build a machine that executes your method. Check that your machine works by tracing through the configurations.

Problem tms.3. Design a Turing-machine with alphabet $\{0, A, B\}$ that accepts any string of As and Bs where the number of As is the same as the number of Bs *and* all the As precede all the Bs, and rejects any string where the number of As is not equal to the number of Bs or the As do not precede all the Bs. (E.g., the machine should accept $AABB$, and $AAABBB$, but reject both AAB and $AABBAABB$.)

Problem tms.4. Design a Turing-machine with alphabet $\{0, A, B\}$ that takes as input any string α of As and Bs and duplicates them to produce an output of the form $\alpha\alpha$. (E.g. input $ABBA$ should result in output $ABBAABBA$).

Problem tms.5. *Alphabetical?:* Design a Turing-machine with alphabet $\{0, A, B\}$ that when given as input a finite sequence of As and Bs checks to see if all the As appear left of all the Bs or not. The machine should leave the input string on the tape, and output either halt if the string is "alphabetical", or loop forever if the string is not.

Problem tms.6. *Alphabetizer:* Design a Turing-machine with alphabet $\{0, A, B\}$ that takes as input a finite sequence of As and Bs rearranges them so that all the As are to the left of all the Bs. (e.g., the sequence $BABAA$ should become the sequence $AAABB$, and the sequence $ABBABB$ should become the sequence $AABBBB$).

mac.3 Turing Machines

tur:mac:tur:sec

explanation

The formal definition of what constitutes a Turing machine looks abstract, but is actually simple: it merely packs into one mathematical structure all the information needed to specify the workings of a Turing machine. This includes (1) which states the machine can be in, (2) which symbols are allowed to be on the tape, (3) which state the machine should start in, and (4) what the instruction set of the machine is.

Definition mac.5 (Turing machine)**.** A *Turing machine* $T = \langle Q, \Sigma, q_0, \delta \rangle$ consists of

1. a finite set of *states* Q,
2. a finite *alphabet* Σ which includes $\triangleright$ and 0,
3. an *initial state* $q_0 \in Q$,
4. a finite *instruction set* $\delta \colon Q \times \Sigma \nrightarrow Q \times \Sigma \times \{L, R, N\}$.

The partial function δ is also called the *transition function* of T.

explanation

We assume that the tape is infinite in one direction only. For this reason it is useful to designate a special symbol $\triangleright$ as a marker for the left end of the tape. This makes it easier for Turing machine programs to tell when they're "in danger" of running off the tape.

Example mac.6. *Even Machine:* The even machine is formally the quadruple $\langle Q, \Sigma, q_0, \delta \rangle$ where

$$\begin{aligned} Q &= \{q_0, q_1\} \\ \Sigma &= \{\triangleright, 0, 1\}, \\ \delta(q_0, 1) &= \langle q_1, 1, R \rangle, \\ \delta(q_1, 1) &= \langle q_0, 1, R \rangle, \\ \delta(q_1, 0) &= \langle q_1, 0, R \rangle. \end{aligned}$$

tur.4 Configurations and Computations

explanation

cmp:tur:con:sec

Recall tracing through the configurations of the even machine earlier. The imaginary mechanism consisting of tape, read/write head, and Turing machine program is really just in intuitive way of visualizing what a Turing machine computation is. Formally, we can define the computation of a Turing machine on a given input as a sequence of *configurations*—and a configuration in turn is a sequence of symbols (corresponding to the contents of the tape at a given point in the computation), a number indicating the position of the read/write head, and a state. Using these, we can define what the Turing machine M computes on a given input.

Definition tur.7 (Configuration)**.** A *configuration* of Turing machine $M = \langle Q, \Sigma, q_0, \delta \rangle$ is a triple $\langle C, n, q \rangle$ where

1. $C \in \Sigma^*$ is a finite sequence of symbols from Σ,
2. $n \in \mathbb{N}$ is a number $< \text{len}(C)$, and
3. $q \in Q$

Intuitively, the sequence C is the content of the tape (symbols of all squares from the leftmost square to the last non-blank or previously visited square), n is the number of the square the read/write head is scanning (beginning with 0 being the number of the leftmost square), and q is the current state of the machine.

explanation

The potential input for a Turing machine is a sequence of symbols, usually a sequence that encodes a number in some form. The initial configuration of the Turing machine is that configuration in which we start the Turing machine to work on that input: the tape contains the tape end marker immediately followed by the input written on the squares to the right, the read/write head is scanning the leftmost square of the input (i.e., the square to the right of the left end marker), and the mechanism is in the designated start state q_0.

Definition tur.8 (Initial configuration)**.** The *initial configuration* of M for input $I \in \Sigma^*$ is

$$\langle \triangleright \frown I, 1, q_0 \rangle$$

The $\frown$ symbol is for *concatenation*—we want to ensure that there are no blanks between the left end marker and the beginning of the input. explanation

Definition tur.9. We say that a configuration $\langle C, n, q\rangle$ *yields* $\langle C', n', q'\rangle$ *in one step* (according to M), iff

1. the n-th symbol of C is σ,
2. the instruction set of M specifies $\delta(q, \sigma) = \langle q', \sigma', D\rangle$,
3. the n-th symbol of C' is σ', and
4. a) $D = L$ and $n' = n - 1$ if $n > 0$, otherwise $n' = 0$, or
   b) $D = R$ and $n' = n + 1$, or
   c) $D = N$ and $n' = n$,
5. if $n' > \text{len}(C)$, then $\text{len}(C') = \text{len}(C) + 1$ and the n'-th symbol of C' is 0.
6. for all i such that $i < \text{len}(C')$ and $i \neq n$, $C'(i) = C(i)$,

Definition tur.10. A *run of M on input I* is a sequence C_i of configurations of M, where C_0 is the initial configuration of M for input I, and each C_i yields C_{i+1} in one step.

We say that M *halts on input I after k steps* if $C_k = \langle C, n, q\rangle$, the nth symbol of C is σ, and $\delta(q, \sigma)$ is undefined. In that case, the *output* of M for input I is O, where O is a string of symbols not beginning or ending in 0 such that $C = \triangleright \frown 0^i \frown O \frown 0^j$ for some $i, j \in \mathbb{N}$.

According to this definition, the output O of M always begins and ends in a symbol other than 0, or, if at time k the entire tape is filled with 0 (except for the leftmost $\triangleright$), O is the empty string. explanation

mac.5 Unary Representation of Numbers

tur:mac:una:sec

Turing machines work on sequences of symbols written on their tape. Depending on the alphabet a Turing machine uses, these sequences of symbols can represent various inputs and outputs. Of particular interest, of course, are Turing machines which compute *arithmetical* functions, i.e., functions of natural numbers. A simple way to represent positive integers is by coding them as sequences of a single symbol 1. If $n \in \mathbb{N}$, let 1^n be the empty sequence if $n = 0$, and otherwise the sequence consisting of exactly n 1's. explanation

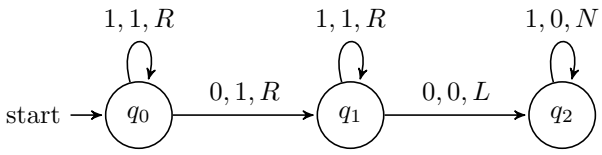
Definition mac.11 (Computation)**.** A Turing machine M *computes* the function $f\colon \mathbb{N}^n \to \mathbb{N}$ iff M halts on input

$$1^{k_1}01^{k_2}0\ldots01^{k_n}$$

with output $1^{f(k_1,\ldots,k_n)}$.

Example mac.12. *Addition:* Build a machine that, when given an input of two non-empty strings of 1's of length n and m, computes the function $f(n, m) = n + m$.

We want to come up with a machine that starts with two blocks of strokes on the tape and halts with one block of strokes. We first need a method to carry out. The input strokes are separated by a blank, so one method would be to write a stroke on the square containing the blank, and erase the first (or last) stroke. This would result in a block of $n + m$ 1's. Alternatively, we could proceed in a similar way to the doubler machine, by erasing a stroke from the first block, and adding one to the second block of strokes until the first block has been removed completely. We will proceed with the former example.



Problem mac.7. Trace through the configurations of the machine for input $\langle 3, 5 \rangle$.

Problem mac.8. *Subtraction:* Design a Turing machine that when given an input of two non-empty strings of strokes of length n and m, where $n > m$, computes the function $f(n, m) = n - m$.

Problem mac.9. *Equality:* Design a Turing machine to compute the following function:

$$\text{equality}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

where x and y are integers greater than 0.

Problem mac.10. Design a Turing machine to compute the function $\min(x, y)$ where x and y are positive integers represented on the tape by strings of 1's separated by a 0. You may use additional symbols in the alphabet of the machine.

The function min selects the smallest value from its arguments, so $\min(3, 5) = 3$, $\min(20, 16) = 16$, and $\min(4, 4) = 4$, and so on.

tur.6 Halting States

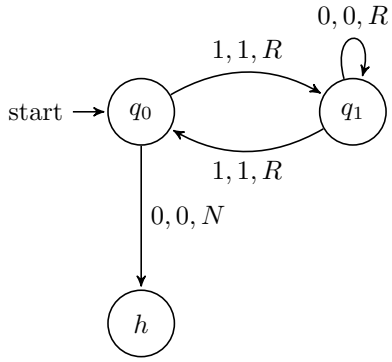
explanation

Although we have defined our machines to halt only when there is no instruction to carry out, common representations of Turing machines have a dedicated *halting state*, h, such that $h \in Q$.

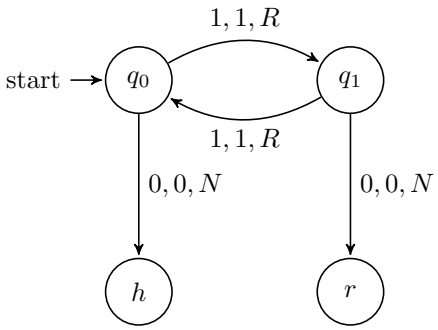
tur:tur:tur:sec

The idea behind a halting state is simple: when the machine has finished operation (it is ready to accept input, or has finished writing the output), it goes into a state h where it halts. Some machines have two halting states, one that accepts input and one that rejects input.

Example tur.13. *Halting States.* To elucidate this concept, let us begin with an alteration of the even machine. Instead of having the machine halt in state q_0 if the input is even, we can add an instruction to send the machine into a halt state.



Let us further expand the example. When the machine determines that the input is odd, it never halts. We can alter the machine to include a *reject* state by replacing the looping instruction with an instruction to go to a reject state r.



Adding a dedicated halting state can be advantageous in cases like this, where it makes explicit when the machine accepts/rejects certain inputs. However, it is important to note that no computing power is gained by adding a dedicated halting state. Similarly, a less formal notion of halting has its own advantages. The definition of halting used so far in this chapter makes the proof of the *Halting Problem* intuitive and easy to demonstrate. For this reason, we continue with our original definition. explanation

mac.7 Combining Turing Machines

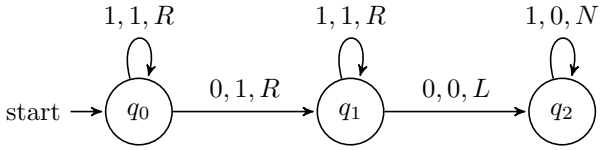
tur:mac:cmb:sec

The examples of Turing machines we have seen so far have been fairly simple in nature. But in fact, any problem that can be solved with any modern programming language can als o be solved with Turing machines. To build more complex Turing machines, it is important to convince ourselves that we can combine them, so we can build machines to solve more complex problems by explanation

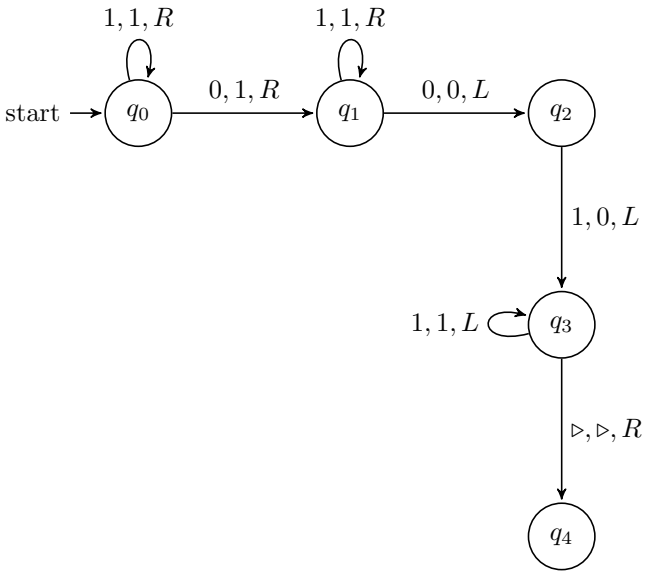
breaking the procedure into simpler parts. If we can find a natural way to break a complex problem down into constituent parts, we can tackle the problem in several stages, creating several simple Turing machines and combining then into one machine that can solve the problem. This point is especially important when tackling the Halting Problem in the next section.

Example mac.14. *Combining Machines:* Design a machine that computes the function $f(m, n) = 2(m + n)$.

In order to build this machine, we can combine two machines we are already familiar with: the addition machine, and the doubler. We begin by drawing a state diagram for the addition machine.

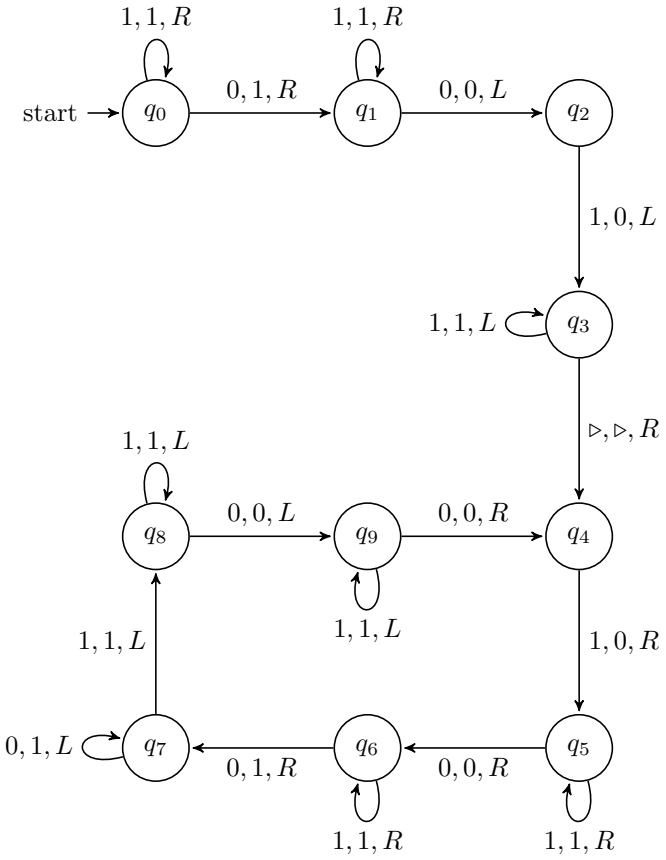


Instead of halting at state q_2, we want to continue operation in order to double the output. Recall that the doubler machine erases the first stroke in the input and writes two strokes in a separate output. Let's add an instruction to make sure the tape head is reading the first stroke of the output of the addition machine.



It is now easy to double the input—all we have to do is connect the doubler machine onto state q_4. This requires renaming the states of the doubler machine so that they start at q_4 instead of q_0—this way we don't end up with two

starting states. The final diagram should look like:



mac.8 Variants of Turing Machines

tur:mac:var:sec

There are in fact many possible ways to define Turing machines, of which ours is only one. In some ways, our definition is more liberal than others. We allow arbitrary finite alphabets, a more restricted definition might allow only two tape symbols, 1 and 0. We allow the machine to write a symbol to the tape and move at the same time, other definitions allow either writing or moving. We allow the possibility of writing without moving the tape head, other definitions leave out the N "instruction." In other ways, our definition is more restrictive. We assumed that the tape is infinite in one direction only, other definitions allow the tape to be infinite both to the left and the right. In fact, one can even even allow any number of separate tapes, or even an infinite grid of squares. We represent the instruction set of the Turing machine by a transition function; other definitions use a transition relation where the machine has more than one possible instruction in any given situation.

This last relaxation of the definition is particularly interesting. In our definition, when the machine is in state q reading symbol σ, $\delta(q, \sigma)$ determines what the new symbol, state, and tape head position is. But if we allow the instruction set to be a relation between current state-symbol pairs $\langle q, \sigma \rangle$ and new state-symbol-direction triples $\langle q', \sigma', D \rangle$, the action of the Turing machine may not be uniquely determined—the instruction relation may contain both $\langle q, \sigma, q', \sigma', D \rangle$ and $\langle q, \sigma, q'', \sigma'', D' \rangle$. In this case we have a *non-deterministic* Turing machine. These play an important role in computational complexity theory.

There are also different conventions for when a Turing machine halts: we say it halts when the transition function is undefined, other definitions require the machine to be in a special designated halting state. Since the tapes of our turing machines are infinite in one direction only, there ae cases where a Turing machine can't properly carry out an instruction: if it reads the leftmost square and is supposed to move left. According to our definition, it just stays put instead, but we could have defined it so that it halts when that happens. There are also different ways of representing numbers (and hence the input-output function computed by a Turing machine): we use unary representation, but you can also use binary representation (this requires two symbols in addition to 0).

Now here is an interesting fact: none of these variations matters as to which functions are Turing computable. *If a function is Turing computable according to one definition, it is Turing computable according to all of them.*

mac.9 The Church-Turing Thesis

tur:mac:ctt:sec

Turing machines are supposed to be a precise replacement for the concept of an effective procedure. Turing took it that anyone who grasped the concept of an effective procedure and the concept of a Turing machine would have the intuition that anything that could be done via an effective procedure could be done by Turing machine. This claim is given support by the fact that all the other proposed precise replacements for the concept of an effective procedure turn out to be extensionally equivalent to the concept of a Turing machine—that is, they can compute exactly the same set of functions. This claim is called the *Church-Turing thesis*.

Definition mac.15 (Church-Turing thesis)**.** The *Church-Turing Thesis* states that anything computable via an effective procedure is Turing computable.

The Church-Turing thesis is appealed to in two ways. The first kind of use of the Church-Turing thesis is an excuse for laziness. Suppose we have a description of an effective procedure to compute something, say, in "pseudo-code." Then we can invoke the Church-Turing thesis to justify the claim that the same function is computed by some Turing machine, eve if we have not in fact constructed it.

The other use of the Church-Turing thesis is more philosophically interesting. It can be shown that there are functions whch cannot be computed by a Turing machines. From this, using the Church-Turing thesis, one can conclude that it cannot be effectively computed, using any procedure whatsoever. For if there were such a procedure, by the Church-Turing thesis, it would follow that there would be a Turing machine. So if we can prove that there is no Turing machine that computes it, there also can't be an effective procedure. In particular, the Church-Turing thesis is invoked to claim that the so-called halting problem not only cannot be solved by Turing machines, it cannot be effectively solved at all.

Photo Credits

Bibliography