Introduction to Algorithms **Notes on Turing Machines**
CS 4820, Spring 2009 **Wednesday, April 8**

1 Definition of a Turing machine

Turing machines are an abstract model of computation. They provide a precise, formal definition of what it means for a function to be computable. Many other definitions of computation have been proposed over the years — for example, one could try to formalize precisely what it means to run a program in the language F# on a computer with an infinite amount of memory — but it turns out that all known definitions of computation agree on what is computable and what is not. The Turing Machine definition seems to be the simplest, which is why we present it here.

A Turing machine can be thought of as a finite state machine sitting on an infinitely long tape containing symbols from some finite alphabet Σ. Based on the symbol it's currently reading, and its current state, the Turing machine writes a new symbol in that location (possibly the same as the previous one), moves left or right or stays in place, and enters a new state. It may also decide to halt and, optionally, to output "yes" or "no" upon halting. The machine's *transition function* is the "program" that specifies each of these actions (overwriting the current symbol, moving left or right, entering a new state, optionally halting and outputting an answer) given the current state and the symbol the machine is currently reading.

Definition 1. A Turing machine is specified by a finite alphabet Σ, a finite set of states K with a special element s (the starting state), and a *transition function* $\delta : K \times \Sigma \to (K \cup \{\text{halt, yes, no}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. It is assumed that Σ, K, $\{\text{halt,yes,no}\}$, and $\{\leftarrow, \rightarrow, -\}$ are disjoint sets, and that Σ contains two special elements $\triangleright, \sqcup$ representing the start and end of the tape, respectively. We require that for every $q \in K$, if $\delta(q, \triangleright) = (p, \sigma, d)$ then $\sigma = \triangleright$ and $d \neq \leftarrow$. In other words, the machine never tries to overwrite the leftmost symbol on its tape nor to move to the left of it.

Note that a Turing machine is not prevented from overwriting the rightmost symbol on its tape or moving to the right of it. In fact, this capability is necessary in order for Turing machines to perform computations that require more space than is given in their original input string.

Having defined the *specification* of a Turing machine, we must now pin down a definition of *how they operate*. This has been informally described above, but it's time to make it formal. That begins with formally defining the configuration of the Turing machine at any time (the state of its tape, as well as the machine's own state and its position on the tape) and the rules for how its configuration changes over time.

Definition 2. The set Σ^* is the set of all finite sequences of elements of Σ. When an element of Σ^* is denoted by a letter such as x, then the elements of the sequence x are denoted by $x_0, x_1, x_2, \ldots, x_{n-1}$, where n is the length of x. The length of x is denoted by $|x|$.

A *configuration* of a Turing machine is an ordered triple $(x, q, k) \in \Sigma^* \times K \times \mathbb{N}$, where x denotes the string on the tape, q denotes the machine's current state, and k denotes the position of the machine on the tape. The string x is required to begin with $\triangleright$ and end with $\sqcup$. The position k is required to satisfy $0 \leq k < |x|$.

If M is a Turing machine and (x, q, k) is its configuration at any point in time, then its configuration (x', q', k') at the following point in time is determined as follows. Let $(p, \sigma, d) =$

$\delta(q, x_k)$. The string x' is obtained from x by changing x_k to σ, and also appending $\sqcup$ to the end of x, if $k = |x| - 1$. The new state q' is equal to p, and the new position k' is equal to $k-1, k+1$, or k according to whether d is $\leftarrow, \rightarrow$, or $-$, respectively. We express this relation between (x, q, k) and (x', q', k') by writing $(x, q, k) \xrightarrow{M} (x', q', k')$.

A *computation* of a Turing machine is a sequence of configurations (x_i, q_i, k_i), where i runs from 0 to T (allowing for the case $T = \infty$) that satisfies: that begins with a *starting configuration* in which $q = s$ and $k = 0$, and either

- The machine starts in a valid starting configuration, meaning that $q_0 = s$ and $k_0 = 0$.
- Each pair of consecutive configurations represents a valid transition, i.e. for $0 \le i < T$, it is the case that $(x_i, q_i, k_i) \xrightarrow{M} (x_{i+1}, q_{i+1}, k_{i+1})$.
- If $T = \infty$, we say that the computation *does not halt.*
- If $T < \infty$, we require that $q_T \in \{\text{halt,yes,no}\}$ and we say that the computation *halts.* If $q_T = \text{yes}$ (respectively, $q_T = \text{no}$) we say that the computation outputs "yes" (respectively, outputs "no").

2 Examples of Turing machines

Example 1. As our first example, let's construct a Turing machine that takes a binary string and appends 0 to the left side of the string. The machine has four states: s, r_0, r_1, ℓ. State s is the starting state, in state r_0 and r_1 it is moving right and preparing to write a 0 or 1, respectively, and in state ℓ it is moving left.

The state s will be used only for getting started: thus, we only need to define how the Turing machine behaves when reading $\triangleright$ in state s. The states r_0 and r_1 will be used, respectively, for writing 0 and writing 1 while remembering the overwritten symbol and moving to the right. Finally, state ℓ is used for returning to the left side of the tape without changing its contents. This plain-English description of the Turing machine implies the following transition function. For brevity, we have omitted from the table the lines corresponding to pairs (q, σ) such that the Turing machine can't possibly be reading σ when it is in state q.

		$\delta(q,\sigma)$		
q	σ	state	symbol	direction
s	$\triangleright$	r_0	$\triangleright$	$\rightarrow$
r_0	0	r_0	0	$\rightarrow$
r_0	1	r_1	0	$\rightarrow$
r_0	$\sqcup$	ℓ	0	$\leftarrow$
r_1	0	r_0	1	$\rightarrow$
r_1	1	r_1	1	$\rightarrow$
r_1	$\sqcup$	ℓ	1	$\leftarrow$
ℓ	0	ℓ	0	$\leftarrow$
ℓ	1	ℓ	1	$\leftarrow$
ℓ	$\triangleright$	halt	$\triangleright$	$-$

Example 2. Using similar ideas, we can design a Turing machine that takes a binary integer n (with the digits written in order, from the most significant digits on the left to the least significant digits on the right) and outputs the binary representation of its successor, i.e. the number $n+1$.

It is easy to see that the following rule takes the binary representation of n and outputs the binary representation of $n+1$. Find the rightmost occurrence of the digit 0 in the binary representation of n, change this digit to 1, and change every digit to the right of it from 1 to 0. The only exception is if the binary representation of n does not contain the digit 0; in that case, one should change every digit from 1 to 0 and prepend the digit 1.

Thus, the Turing machine for computing the binary successor function works as follows: it uses one state r to initially scan from left to right without modifying any of the digits, until it encounters the symbol $\sqcup$. At that point it changes into a new state ℓ in which it moves to the left, changing any 1's that it encounters to 0's, until the first time that it encounters a symbol other than 1. (This may happen before encountering any 1's.) If that symbol is 0, it changes it to 1 and enters a new state t that moves leftward to $\triangleright$ and then halts. On the other hand, if the symbol is $\triangleright$, then the original input consisted exclusively of 1's. In that case, it prepends a 1 to the input using a subroutine very similar to Example 1. We'll refer to the states in that subroutine as prepend::s, prepend::r, prepend::ℓ.

		$\delta(q,\sigma)$		
q	σ	state	symbol	direction
s	$\triangleright$	r	$\triangleright$	$\rightarrow$
r	0	r	0	$\rightarrow$
r	1	r	1	$\rightarrow$
r	$\sqcup$	ℓ	$\sqcup$	$\leftarrow$
ℓ	0	t	1	$\leftarrow$
ℓ	1	ℓ	0	$\leftarrow$
ℓ	$\triangleright$	prepend::s	$\triangleright$	$-$
t	0	t	0	$\leftarrow$
t	1	t	1	$\leftarrow$
t	$\triangleright$	halt	$\triangleright$	$-$
prepend::s	$\triangleright$	prepend::s	$\triangleright$	$\rightarrow$
prepend::s	0	prepend::r	1	$\rightarrow$
prepend::r	0	prepend::r	0	$\rightarrow$
prepend::r	$\sqcup$	prepend::ℓ	0	$\leftarrow$
prepend::ℓ	0	prepend::ℓ	0	$\leftarrow$
prepend::ℓ	1	prepend::ℓ	1	$\leftarrow$
prepend::ℓ	$\triangleright$	halt	$\triangleright$	$-$

Example 3. We can compute the binary precedessor function in roughly the same way. We must take care to define the value of the predecessor function when its input is the number 0, since we haven't yet specified how negative numbers should be represented on a Turing machine's tape using the alphabet $\{\triangleright, \sqcup, 0, 1\}$. Rather than specify a convention for representing negative numbers, we will simply define the value of the predecessor function to be 0 when its input is 0. Also, for convenience, we will allow the output of the binary predecessor function to have any number of initial copies of the digit 0.

Our program to compute the binary predecessor of n thus begins with a test to see if n is equal to 0. The machine moves to the right, remaining in state z unless it sees the digit 1. If it reaches the end of its input in state z, then it simply rewinds to the beginning of the tape. Otherwise, it decrements the input in much the same way that the preceding example incremented it: in this case we use state ℓ to change the trailing 0's to 1's until we encounter the rightmost 1, and then we enter state t to rewind to the beginning of the tape.

			$\delta(q,\sigma)$	
q	σ	state	symbol	direction
s	$\triangleright$	z	$\triangleright$	$\rightarrow$
z	0	z	0	$\rightarrow$
z	1	r	1	$\rightarrow$
z	$\sqcup$	t	$\sqcup$	$\leftarrow$
r	0	r	0	$\rightarrow$
r	1	r	1	$\rightarrow$
r	$\sqcup$	ℓ	$\sqcup$	$\leftarrow$
ℓ	0	ℓ	1	$\leftarrow$
ℓ	1	t	0	$\leftarrow$
t	0	t	0	$\leftarrow$
t	1	t	1	$\leftarrow$
t	$\triangleright$	halt	$\triangleright$	$-$

3 Universal Turing machines

The key property of Turing machines, and all other equivalent models of computation, is *universality*: there is a single Turing machine U that is capable of simulating any other Turing machine — even those with vastly more states than U. In other words, one can think of U as a "Turing machine interpreter", written in the language of Turing machines. This capability for self-reference (the language of Turing machines is expressive enough to write an interpreter for itself) is the source of the surprising versatility of Turing machines and other models of computation. It is also the Pandora's Box that allows us to come up with undecidable problems, as we shall see in the following section.

3.1 Describing Turing machines using strings

To define a universal Turing machine, we must first explain what it means to give a "description" of one Turing machine as the input to another one. For example, we must explain how a single Turing machine with bounded alphabet size can read the description of a Turing machine with a much larger alphabet.

To do so, we will make the following assumptions. For a Turing machine M with alphabet Σ and state set K, let

$$\ell = \lceil \log_2(|\Sigma| + |K| + 6) \rceil.$$

We will assume that each element of $\Sigma \cup K \cup \{\text{halt,yes,no}\} \cup \{\leftarrow, \rightarrow, -\}$ is identified with a distinct binary string in $\{0,1\}^\ell$. For example, we could always assume that Σ consists of the numbers $0, 1, 2, \ldots, |\Sigma| - 1$ represented as ℓ-bit binary numbers (with initial 0's prepended, if necessary, to make the binary representation exactly ℓ digits long), that K consists of the numbers $|\Sigma|, |\Sigma| + 1, \ldots, |\Sigma| + |K| - 1$ (with $|\Sigma|$ representing the starting state s) and that $\{\text{halt,yes,no}\} \cup \{\leftarrow, \rightarrow, -\}$ consists of the numbers $|\Sigma| + |K|, \ldots, |\Sigma| + |K| + 5$.

Now, the description of Turing machine M is defined to be a finite string in the alphabet $\{0,\ 1,\ \text{`('},\ \text{`)'},\ \text{`,'}\}$, determined as follows.

$$M = m, n, \delta_1 \delta_2 \ldots \delta_N$$

where m is the number $|\Sigma|$ in binary, n is the number $|K|$ in binary, and each of $\delta_1, \ldots, \delta_N$ is a string encoding one of the transition rules that make up the machine's transition function δ. Each such rule is encoded using a string

$$\delta_i = (q, \sigma, p, \tau, d)$$

where each of the five parts q, σ, p, τ, d is a string in $\{0,1\}^\ell$ encoding an element of $\Sigma \cup K \cup \{\text{halt,yes,no}\} \cup \{\leftarrow, \rightarrow, -\}$ as described above. The presence of δ_i in the description of M indicates that when M is in state q and reading symbol σ, then it transitions into state p, writes symbol τ, and moves in direction d. In other words, the string $\delta_i = (q, \sigma, p, \tau, d)$ in the description of M indicates that $\delta(q, \sigma) = (p, \tau, d)$.

3.2 Definition of a universal Turing machine

A universal Turing machine is a Turing machine U with alphabet {0, 1, '(', ')', ',', ';'}. It takes an input of the form $\triangleright M; x \sqcup$, where M is a valid description of a Turing machine and x is a string in the alphabet of M, encoded using ℓ-bit blocks as described earlier. (If its input fails to match this specification, the universal Turing machine U is allowed to behave arbitrarily.)

The computation of U, given input $\triangleright M; x \sqcup$, has the same termination status — halting in state "halt", "yes", or "no", or never halting — as the computation of M on input x. Furthermore, if M halts on input x (and, consequently, U halts on input $\triangleright M; x \sqcup$) then the string on U's tape at the time it halts is equal to the string on M's tape at the time it halts, again translated into binary using ℓ-bit blocks as specified above.

It is far from obvious that a universal Turing machine exists. In particular, such a machine must have a finite number of states, yet it must be able to simulate a computation performed by a Turing machine with a much greater number of states. In Section 3.4 we will describe how to construct a universal Turing machine. First, it is helpful to extend the definition of Turing machines to allow multiple tapes. After describing this extension we will indicate how a multi-tape Turing machine can be simulated by a single-tape machine (at the cost of a slower running time).

3.3 Multi-tape Turing machines

A multi-tape Turing machine with k tapes is defined in nearly the same way as a conventional single-tape Turing machine, except that it stores k strings simultaneously, maintains a position in each of the k strings, updates these k positions independently (i.e. it can move left in one string while moving right in another), and its state changes are based on the entire k-tuple of symbols that it reads at any point in time. More formally, a k-tape Turing machine has a finite alphabet Σ and state set K as before, but its transition function is

$$\delta : K \times \Sigma^k \to (K \cup \{\text{halt, yes, no}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k,$$

the difference being that the transition is determined by the entire k-tuple of symbols that it is reading, and that the instruction for the next action taken by the machine must include the symbol to be written, and the direction of motion, for *each* of the k strings.

The purpose of this section is to show that any multi-tape Turing machine M can be simulated with a single-tape Turing machine M°. To store the contents of the k strings simultaneously, as well as the position of M within each of these strings, our single-tape machine M° will have alphabet $\Sigma^\circ = \Sigma^k \times \{0,1\}^k$. The k-tuple of symbols $(\sigma_1, \ldots, \sigma_k) \in \Sigma^k$ at the i^{th} location on the tape is interpreted to signify the symbols at the i^{th} location on each of the k tapes used by machine M. The k-tuple of binary values ("flags") $(p_1, \ldots, p_k) \in \{0,1\}^k$ at the i^{th} location on the tape represents whether the position of machine M on each of its k tapes is currently at location i. ($1 =$ present, $0 =$ absent.) Our definition of Turing machine requires that the alphabet must contain two special symbols $\triangleright, \sqcup$. In the case of the alphabet Σ°, the special symbol $\triangleright$ is interpreted to be identical with $(\triangleright)^k \times (0)^k$, and $\sqcup$ is interpreted to be identical with $(\sqcup)^k \times (0)^k$.

Machine M° runs in a sequence of *simulation rounds*, each divided into k phases. The purpose of each round is to simulate one step of M, and the purpose of phase i in a round is to simulate what happens in the i^{th} string of M during that step. At the start of a phase, M° is always at the left edge of its tape, on the symbol $\triangleright$. It makes a forward-and-back pass through its string, locating the flag that indicates the position of M on its i^{th} tape and updating that symbol (and possibly the adjacent one) to reflect the way that M updates its i^{th} string during the corresponding step of its execution.

The pseudocode presented in Algorithm 2 describes the simulation. Of course, the whole point of defining Turing machines is to get away from the *informal* notion of "algorithm" as exemplified by pseudocode. Thus, in addition to presenting the pseudocode, we must say a few words about how to actually implement the pseudocode using a Turing machine M°. This primarily involves specifying how a finite state set K° can be used to encode every piece of information that's used in pseudocode other than the data stored on the tape.

Algorithm 2 uses variables $q, i, \sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_k, d_1, \ldots, d_k$. Each of these can assume only finitely many values. Letting K, Σ denote the state set and alphabet of M, letting $[k] = \{1, \ldots, k\}$ and letting L denote the set of line numbers of the pseudocode in Algorithm 2, we define the state set K° to be

$$K^\circ = K \times [k] \times \Sigma^k \times \Sigma^k \times \{\leftarrow, \rightarrow, -\}^k \times L.$$

Thus, a state of M° determines the values of the variables $q, i, \sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_k, d_1, \ldots, d_k$ as well as the line number of the line of pseudocode that M° is currently working on.

Having defined K° in this way, it should be clear how to transform the pseudocode presented in Algorithm 2 into a state transition function for M°. It is worth pointing out a few subtleties about the mechanics of transforming pseudocode into a Turing machine, as we did here. If the pseudocode had contained a variable that can take infinitely many different values — for example, an integer-valued variable, or a pointer to a location on the tape — then the state set K° would not have been finite (since it needs to encode every possible value that the variables can take) and this would have violated the constraint that a Turing machine can have only finitely many states. Even more subtly, if the pseudocode had been recursive, then K° would have had to allow for a stack of program counters rather than a single program counter; if this stack has unbounded depth (i.e., if the pseudocode does not have bounded recursion depth) then K° again fails to be a finite set and M° fails to be a Turing machine. This is *not* to say that program with integer variables or with unbounded recursion depth cannot be simulated by Turing machines, only that the process of translating their pseudocode into a Turing machine is not straightforward and requires using the tape, rather than the machine's internal state, to store the values of the integer variables or the execution stack.

Algorithm 1 Simulation of multi-tape Turing machine M by single-tape Turing machine M°.

```
$q \leftarrow s$ // Initialize state.
repeat
    // Simulation round
    // First, find out what symbols $M$ is seeing.
    repeat
        Move right without changing contents of string.
        for $i = 1, \dots, k$ do
            if $i^{\text{th}}$ flag at current location equals 1 then
                $\sigma_i \leftarrow i^{\text{th}}$ symbol at current location.
            end if
        end for
    until $M^\circ$ reaches $\sqcup$
    repeat
        Move left without changing contents of string.
    until $M^\circ$ reaches $\triangleright$
    // Now, $\sigma_1, \dots, \sigma_k$ store the $k$ symbols that $M$ is seeing.
    // Evaluate state transition function for machine $M$.
    $q', (\sigma'_1, d_1), (\sigma'_2, d_2), \dots, (\sigma'_k, d_k) \leftarrow \delta(q, \sigma_1, \dots, \sigma_k)$
    for $i = 1, \dots, k$ do
        // Phase $i$
        repeat
            Move right without changing the contents of the string.
        until a location whose $i^{\text{th}}$ flag is 1 is reached.
        Change the $i^{\text{th}}$ symbol at this location to $\sigma'_i$.
        if $d_i =\leftarrow$ then
            Change $i^{\text{th}}$ flag to 0 at this location.
            Move one step left
            Change $i^{\text{th}}$ flag to 1.
        else if $d_i =\rightarrow$ then
            Change $i^{\text{th}}$ flag to 0 at this location.
            Move one step right.
            Change $i^{\text{th}}$ flag to 1.
        end if
        repeat
            Move left without changing the contents of the string.
        until the symbol $\triangleright$ is reached.
    end for
    $q \leftarrow q'$
until $q \in \{\text{halt}, \text{yes}, \text{no}\}$
// If the simulation reaches this line, $q \in \{\text{halt}, \text{yes}, \text{no}\}$.
if $q = \text{yes}$ then
    Transition to “yes” state.
else if $q = \text{no}$ then
    Transition to “no” state.
else
    Transition to “halt” state.
end if
```

3.4 Construction of a universal Turing machine

We now proceed to describe the construction of a universal Turing machine U. Taking advantage of Section 3.3, we can describe U as a 5-tape Turing machine; the existence of a single-tape universal Turing machine then follows from the general simulation presented in that section.

Our universal Turing machine has four tapes:

- the input tape: a read-only tape containing the input, which is never overwritten;
- the description tape: a tape containing the description of M, which is written once at initialization time and never overwritten afterwards;
- the working tape: a tape whose contents correspond to the contents of M's tape, translated into ℓ-bit blocks of binary symbols separated by commas, as the computation proceeds.
- the state tape: a tape describing the current state of M, encoded as an ℓ-bit block of binary symbols.
- the special tape: a tape containing the binary encodings of the special states {halt,yes,no} and the "directional symbols" $\{\leftarrow, \rightarrow, -\}$.

The state tape solves the mystery of how a machine with a finite number of states can simulate a machine with many more states: it encodes these states using a tape that has the capacity to hold an unbounded number of symbols.

It would be too complicated to write down a diagram of the entire state transition function of a universal Turing machine, but we can describe it in plain English and pseudocode. The machine begins with an initialization phase in which it performs the following tasks:

- copy the description of M onto the description tape,
- copy the input string x onto the working tape, inserting commas between each ℓ-bit block;
- copy the starting state of M onto the state tape;
- write the identifiers of the special states and directional symbols onto the special tape;
- move each cursor to the leftmost position on its respective tape.

After this initialization, the universal Turing machine executes its main loop. Each iteration of the main loop corresponds to one step in the computation executed by M on input x. At the start of any iteration of U's main loop, the working tape and state tape contain the binary encodings of M's tape contents and its state at the start of the corresponding step in M's computation. Also, when U begins an iteration of its main loop, the cursor on each of its tapes except the working tape is at the leftmost position, and the cursor on the working tape is at the location corresponding to the position of M's cursor at the start of the corresponding step in its computation. (In other words, U's working tape cursor is pointing to the comma preceding the binary encoding of the symbol that M's cursor is pointing to.)

The first step in an iteration of the main loop is to check whether it is time to halt. This is done by reading the contents of M's state tape and comparing it to the binary encoding of the states "halt", "yes", and "no", which are stored on the special tape. Assuming that M is not in one of the states "halt", "yes", "no", it is time to simulate one step in the execution of M. This is done by working through the segment of the description tape that contains the strings

$\delta_1, \delta_2, \ldots, \delta_N$ describing M's transition function. The universal Turing machine moves through these strings in order from left to right. As it encounters each pair (q, σ), it checks whether q is identical to the ℓ-bit string on its state tape and σ is identical to the ℓ-bit string on its working tape. These comparisons are performed one bit at a time, and if either of the comparisons fails, then U rewinds its state tape cursor back to the $\triangleright$ and it rewinds its working tape cursor back to the comma that marked its location at the start of this iteration of the main loop. It then moves its description tape cursor forward to the description of the next rule in M's transition function. When it finally encounters a pair (q, σ) that matches its current state tape and working tape, then it moves forward to read the corresponding (p, τ, d), and it copies p onto the state tape, τ onto the working tape, and then moves its working tape cursor in the direction specified by d. Finally, to end this iteration of the main loop, it rewinds the cursors on its description tape and state tape back to the leftmost position.

It is worth mentioning that the use of five tapes in the universal Turing machine is overkill. In particular, there is no need to save the input $\triangleright M; x \sqcup$ on a separate read-only input tape. As we have seen, the input tape is never used after the end of the initialization stage. Thus, for example, we can skip the initialization step of copying the description of M from the input tape to the description tape; instead, after the initialization finishes, we can treat the input tape henceforward as if it were the description tape.

Algorithm 2 Universal Turing machine, initialization.

```
// Copy the transition function of $M$ from the input tape to the description tape.
Move right past the first and second commas on the input tape.
while not reading ’;’ on input tape do
    Read input tape symbol, copy to description tape, move right on both tapes.
end while
// Now, write the identifiers of the halting states and the “direction of motion symbols” on the special tape.
Move to start of input tape.
Using binary addition subroutine, write $m + n$ in binary on working tape.
for $i = 0, 1, 2, 3, 4, 5$ do
    On special tape, write binary representation of $m + n + i$, followed by ’;’.
    // This doesn’t require storing $m + n$ in memory, because it’s stored on the working tape.
end for
// Copy the input string $x$ onto the working tape, inserting commas between each $\ell$-bit block.
Write $m + n + 6$ in binary on the state tape. // In order to store the value of $\ell$.
Starting from left edge of input tape, move right until ’;’ is reached.
Move to left edge of working tape and state tape.
Move one step right on state tape.
repeat
    while state tape symbol is not $\sqcup$ do
        Move right on input tape, working tape, and state tape.
        Copy symbol from input tape to working tape.
    end while
    Write ’,’ on working tape.
    Rewind to left edge of state tape, then move one step right.
until input tape symbol is $\sqcup$
Copy $m$ from input tape to state tape.
// Done with initialization!
```

Algorithm 3 Universal Turing machine, main loop.

```
// Infinite loop. Each loop iteration simulates one step of $M$.
Move to the left edge of all tapes.
// First check if we need to halt.
$match \leftarrow$ True
repeat
    Move right on special and state tapes.
    if symbols don't match then
        $match \leftarrow$ False
    end if
until reaching ';' on special tape
if $match =$ True then
    Enter "halt" state.
end if
Repeat the above steps two more times, with "yes", "no" in place of "halt".
Move back to left edge of description tape and state tape.
repeat
    Move right on description tape to next occurrence of '(' or $\sqcup$.
    $match \leftarrow$ True
    repeat
        Move right on description and state tapes.
        if symbols don't match then
            $match \leftarrow$ False
        end if
    until reaching ',' on description tape
    repeat
        Move right on description and working tapes.
        if symbols don't match then
            $match \leftarrow$ False
        end if
    until reaching ',' on description tape
    if $match =$ True then
        Move to left edge of state tape.
        Move left on working tape until ',' is reached.
    end if
    repeat
        Move right on description and state tapes.
        Copy symbol from description to state tape.
    until reaching ',' on description tape
    repeat
        Move right on description and working tapes.
        Copy symbol from description to working tape.
    until reaching ',' on description tape
    Move right on special tape past three occurrences of ';', stop at the fourth.
    $match \leftarrow$ True
    repeat
        Move right on description and special tapes.
        if symbols don't match then
            $match \leftarrow$ False
        end if
    until reaching ';' on special tape
    if $match =$ True then
        repeat
            Move left on working tape.
        until reaching ',' or $\triangleright$
    end if
    $match \leftarrow$ True
    repeat
        Move right on description and special tapes.
        if symbols don't match then
            $match \leftarrow$ False
        end if
    until reaching ';' on special tape
    if $match =$ True then
        repeat
            Move right on working tape.
        until reaching ',' or $\triangleright$
    end if
until Reading $\sqcup$ on description tape.
```

4 Undecidable problems

In this section we will see that there exist computational problems that are too difficult to be solved by any Turing machine. Since Turing machines are universal enough to represent any algorithm running on a deterministic computer, this means there are problems too difficult to be solved by any algorithm.

4.1 Definitions

To be precise about the notion of what we mean by "computational problems" and what it means for a Turing machine to "solve" a problem, we define problems in terms of *languages*, which correspond to decision problems with a yes/no answer. We define two notions of "solving" a problem specified by a language L. The first of these definitions ("deciding L") corresponds to what we usually mean when we speak of solving a computational problem, i.e. terminating and outputting a correct yes/no answer. The second definition ("accepting L") is a "one-sided" definition: if the answer is "yes", the machine must halt and provide this answer after a finite amount of time; if the answer is "no", the machine need not ever halt and provide this answer. Finally, we give some definitions that apply to computational problems where the goal is to output a string rather than just a simple yes/no answer.

Definition 3. Let $\Sigma_0 = \Sigma \setminus \{\triangleright, \sqcup\}$. A *language* is any set of strings $L \subseteq \Sigma_0^*$. Suppose M is a Turing machine and L is a language.

1. *M decides L* if every computation of M halts in the "yes" or "no" state, and L is the set of strings occurring in starting configurations that lead to the "yes" state.

2. L is *recursive* if there is a machine M that decides L.

3. *M accepts L* if L is the set of strings occurring in starting configurations that lead M to halt in the "yes" state.

4. L is *recursively enumerable* if there is a machine M that accepts L.

5. *M computes* a given function $f : \Sigma_0^* \to \Sigma_0^*$ if every computation of M halts, and for all $x \in \Sigma_0^*$, the computation with starting configuration $(\triangleright x \sqcup, s, 0)$ ends in configuration $(\triangleright f(x) \sqcup \cdots \sqcup, \mathrm{halt}, 0)$, where $\sqcup \cdots \sqcup$ denotes any sequence of one or more repetitions of the symbol $\sqcup$.

6. f is a *recursive function* if there is a Turing machine M that computes f.

4.2 Undecidability via counting

One simple explanation for the existence of undecidable languages is via a counting argument: there are simply too many languages, and not enough Turing machines to decide them all! This can be formalized using the distinction between countable and uncountable sets.

Definition 4. An infinite set is *countable* if and only if there is a one-to-one correspondence between its elements and the natural numbers. Otherwise it is said to be *uncountable.*

Lemma 1. *If Σ is a finite set then Σ^* is countable.*

Proof. If $|\Sigma| = 1$ then a string in Σ^* is uniquely determined by its length and this defines a one-to-one correspondence between Σ^* and the natural numbers. Otherwise, without loss of generality, Σ is equal to the set $\{0, 1, \ldots, b-1\}$ for some positive integer $b > 1$. Every natural number has an expansion in base b which is a finite-length string of elements of Σ. This gives a one-to-one correspondence between natural numbers and elements of Σ^* beginning with a non-zero element of Σ. To get a full one-to-one correspondence, we need one more trick: every positive integer can be uniquely represented in the form $2^s \cdot (2t-1)$ where s and t are natural numbers and $t > 0$. We can map the positive integer $2^s \cdot (2t+1)$ to the string consisting of s occurrences of 0 followed by the base-b representation of t. This gives a one-to-one correspondence between positive natural numbers and nonempty strings in Σ^*. If we map 0 to the empty string, we have defined a full one-to-one correspondence. □

Lemma 2. *Let X be a countable set and let 2^X denote the set of all subsets of X. The set 2^X is uncountable.*

Proof. Consider any function $f : X \to 2^X$. We will construct an element $T \in 2^X$ such that T is not equal to $f(x)$ for any $x \in X$. This proves that there is no one-to-one correspondence between X and 2^X; hence 2^X is uncountable.

Let $x_0, x_1, x_2, \ldots$ be a list of all the elements of X, indexed by the natural numbers. (Such a list exists because of our assumption that X is countable.) The construction of the set T is best explained via a diagram, in which the set $f(x)$, for every $x \in X$ is represented by a row of 0's and 1's in an infinite table. This table has columns indexed by $x_0, x_1, \ldots$, and the row labeled $f(x_i)$ has a 0 in the column labeled x_j if and only if x_j belongs to the set $f(x_i)$.

	x_0	x_1	x_2	x_3	$\ldots$
$f(x_0)$	0	1	0	0	$\ldots$
$f(x_1)$	1	1	0	1	$\ldots$
$f(x_2)$	0	0	1	0	$\ldots$
$f(x_3)$	1	0	1	1	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

To construct the set T, we look at the main diagonal of this table, whose i-th entry specifies whether $x_i \in f(x_i)$, and we flip each bit. This produces a new sequence of 0's and 1's encoding a subset of X. In fact, the subset can be succinctly defined by

$$T = \{x \in X \mid x \notin f(x)\}. \tag{1}$$

We see that T cannot be equal to $f(x)$ for any $x \in X$. Indeed, if $T = f(x)$ and $x \in T$ then this contradicts the fact that $x \notin f(x)$ for all $x \in T$; similarly, if $T = f(x)$ and $x \notin T$ then this contradicts the fact that $x \in f(x)$ for all $x \notin T$. □

Remark 1. Actually, the same proof technique shows that there is never a one-to-one correspondence between X and 2^X for any set X. We can always define the "diagonal set" T via (1) and argue that the assumption $T = f(x)$ leads to a contradiction. The idea of visualizing the function f using a two-dimensional table becomes more strained when the number of rows and columns of the table is uncountable, but this doesn't interfere with the validity of the argument based directly on defining T via (1).

Theorem 3. *For every alphabet Σ there is a language $L \subseteq \Sigma^*$ that is not recursively enumerable.*

Proof. The set of languages $L \subseteq \Sigma^*$ is uncountable by Lemmas 1 and 2. The set of Turing machines with alphabet Σ is countable because each such Turing machine has a description which is a finite-length string of symbols in the alphabet $\{0,\ 1,\ \text{`('},\ \text{`)'},\ \text{`,'}\}$. Therefore there are strictly more languages then there are Turing machines, so there are languages that are not accepted by any Turing machine. □

4.3 Undecidability via diagonalization

The proof of Theorem 3 is quite unsatisfying because it does not provide any example of an interesting language that is not recursively enumerable. In this section we will repeat the "diagonal argument" from the proof of Theorem 2, this time in the context of Turing machines and languages, to obtain a more interesting example of a set that is not recursively enumerable.

Definition 5. For a Turing machine M, we define $L(M)$ to be the set of all strings accepted by M:

$$L(M) = \{x \mid M \text{ halts and outputs "yes" on input } x\}.$$

Suppose Σ is a finite alphabet, and suppose we have specified a mapping ϕ from $\{0,\ 1,\ \text{`('},\ \text{`)'},\ \text{`,'}\}$ to strings of some fixed length in Σ^*, so that each Turing machine has a description in Σ^* obtained by taking its standard description, using ϕ to map each symbol to Σ^*, and concatenating the resulting sequence of strings. For every $x \in \Sigma^*$ we will now define a language $L(x) \subseteq \Sigma_0^*$ as follows. If x is the description of a Turing machine M then $L(x) = L(M)$; otherwise, $L(x) = \emptyset$.

We are now in a position to repeat the diagonal construction from the proof of Theorem 2. Consider an infinite two-dimensional table whose rows and columns are indexed by elements of Σ_0^*.

	x_0	x_1	x_2	x_3	$\dots$
$L(x_0)$	0	1	0	0	$\dots$
$L(x_1)$	1	1	0	1	$\dots$
$L(x_2)$	0	0	1	0	$\dots$
$L(x_3)$	1	0	1	1	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Definition 6. The *diagonal language* $D \subseteq \Sigma_0^*$ is defined by

$$D = \{x \in \Sigma_0^* \mid x \notin L(x)\}.$$

Theorem 4. *The diagonal language D is not recursively enumerable.*

Proof. The proof is exactly the same as the proof of Theorem 2. Assume, by way of contradiction, that $D = L(x)$ for some x. Either $x \in D$ or $x \notin D$ and we obtain a contradiction in both cases. If $x \in D$ then this violates the fact that $x \notin L(x)$ for all $x \in D$. If $x \notin D$ then this violates the fact that $x \in L(x)$ for all $x \notin D$. □

Corollary 5. *There exists a language L that is recursively enumerable but its complement is not.*

Proof. Let L be the complement of the diagonal language D. A Turing machine M that accepts L can be described as follows: given input x, construct the string $x;x$ and run a universal Turing machine on this input. It is clear from the definition of L that $L = L(M)$, so L is recursively enumerable. We have already seen that its complement, D, is not recursively enumerable. □

Remark 2. Unlike Theorem 3, there is no way to obtain Corollary 5 using a simple counting argument.

Corollary 6. *There exists a language L that is recursively enumerable but not recursive.*

Proof. From the definition of a recursive language, it is clear that the complement of a recursive language is recursive. For the recursively enumerable language L in Corollary 5, we know that the complement of L is not even recursively enumerable (hence, *a fortiori*, also not recursive) and this implies that L is not recursive. □

4.4 The halting problem

Theorem 4 gave an explicit example of a language that is not recursively enumerable — hence not decidable — but it is still a rather unnatural example, so perhaps one might hope that every *interesting* computational problem is decidable. Unfortunately, this is not the case!

Definition 7. The *halting problem* is the problem of deciding whether a given Turing machine halts when presented with a given input. In other words, it is the language H defined by

$$H = \{M; x \mid M \text{ is a valid Turing machine description, and } M \text{ halts on input } x\}.$$

Theorem 7. *The halting problem is not decidable.*

Proof. We give a proof by contradiction. Given a Turing machine M_H that decides H, we will construct a Turing machine M_D that accepts D, contradicting Theorem 4. The machine M_D operates as follows. Given input x, it constructs the string $x; x$ and runs M_H on this input until the step when M_H is just about to output "yes" or "no". At that point in the computation, instead of outputting "yes" or "no", M_D does the following. If M_H is about to output "no" then M_D instead outputs "yes". If M_H is about to output "yes" then M_D instead runs a universal Turing machine U on input $x; x$. Just before U is about to halt, if it is about to output "yes", then M_D instead outputs "no". Otherwise M_D outputs "yes". (Note that it is not possible for U to run forever without halting, because of our assumption that M_H decides the halting problem and that it outputs "yes" on input $x; x$.)

By construction, we see that M_D always halts and outputs "yes" or "no", and that its output is "no" if and only if $M_H(x; x) = \text{yes}$ and $U(x; x) = \text{yes}$. In other words, $M_D(x) = \text{no}$ if and only if x is the description of a Turing machine that halts and outputs "yes" on input x. Recalling our definition of $L(x)$, we see that another way of saying this is as follows: $M_D(x) = \text{no}$ if and only if $x \in L(x)$. We now have the following chain of "if and only if" statements:

$$x \in L(M_D) \iff M_D(x) = \text{yes} \iff M_D(x) \neq \text{no} \iff x \notin L(x) \iff x \in D.$$

We have proved that $L(M_D) = D$, i.e. M_D is a Turing machine that accepts D, as claimed. □

Remark 3. It is easy to see that H is recursively enumerable. In fact, if U is any universal Turing machine then $H = L(U)$.

4.5 Rice's Theorem

Theorem 7 shows that, unfortunately, there exist interesting languages that are not decidable. In particular, the question of whether a given Turing machine halts on a given input is not decidable. Unfortunately, the situation is much worse than this! Our next theorem shows that essentially any non-trivial property of Turing machines is undecidable. (Actually, this is an overstatement; the theorem will show that any non-trivial property of *the languages accepted by Turing machines* is undecidable. Non-trivial properties of the Turing machine itself — e.g., does it run for more than 100 steps when presented with the input string $\triangleright\sqcup$ — may be decidable.)

Theorem 8 (Rice's Theorem)**.** *Let $\mathcal{C}$ be a set of languages, and suppose that there is at least one Turing machine M_0 such that $L(M_0) \in \mathcal{C}$ and at least one Turing machine M_1 such that $L(M_1) \notin \mathcal{C}$. Then the following problem is undecidable: given x, is it the case that $L(x) \in \mathcal{C}$? In other words, the language*

$$L_{\mathcal{C}} = \{x \mid L(x) \in \mathcal{C}\}$$

is not recursive.

Proof. Note that $L_{\overline{\mathcal{C}}} = \overline{L_{\mathcal{C}}}$, and that this language is recursive if and only if $L_{\mathcal{C}}$ is, because the complement of a recursive language is recursive. Therefore we may assume without loss of generality that $\emptyset \notin \mathcal{C}$. Starting from this assumption, we will do a proof by contradiction: given a Turing machine $M_{\mathcal{C}}$ that decides $L_{\mathcal{C}}$ we will construct a Turing machine M_H that decides the halting problem, in contradiction to Theorem 7.

The construction of M_H is as follows. On input $M; x$, it transforms M into the description of another Turing machine M^*, and then it feeds this description into $M_{\mathcal{C}}$. The definition of M^* is a little tricky. On input y, machine M^* does the following. First it runs M on input x, without overwriting the string y. If M ever halts, then instead of halting M^* enters the second phase of its execution, which consists of running M_0 on y. (Recall that M_0 is a Turing machine such that $L(M_0) \in \mathcal{C}$.) If M never halts, then M^* also never halts.

This completes the construction of M_H. To recap, when M_H is given an input $M; x$ it first transforms the description of M into the description of a related Turing machine M^*, then it runs $M_{\mathcal{C}}$ on the input consisting of the description of M^* and it outputs the same answer that $M_{\mathcal{C}}$ outputs. There are two things we still have to prove.

1. The function that takes the string $M; x$ and outputs the description of M^* is a recursive function, i.e. there is a Turing machine that can transform $M; x$ into the description of M^*.

2. Assuming $M_{\mathcal{C}}$ decides $L_{\mathcal{C}}$, then M_H decides the halting problem.

The first of these facts is elementary but tedious. M^* needs to have a bunch of extra states that append a special symbol (say, $\sharp$) to the end of its input and then write out the string x after the special symbol, $\sharp$. It also has the same states as M with the same transition function, with two modifications: first, this modified version of M treats the symbol $\sharp$ exactly as if it were $\triangleright$. (This ensures that M will remain on the right side of the tape and will not modify the copy of y that sits to the left of the $\sharp$ symbol.) Finally, whenever M would halt, the modified version of M enters a special set of states that move left to the $\sharp$ symbol, overwrite this symbol with $\sqcup$, continue moving left until the symbol $\triangleright$ is reached, and then run the machine M_0.

Now let's prove the second fact — that M_H decides the halting problem, assuming $M_{\mathcal{C}}$ decides $L_{\mathcal{C}}$. Suppose we run M_H on input $M; x$. If M does not halt on x, then the machine M^* constructed by M_H never halts on any input y. (This is because the first thing M^* does on input y is to simulate the computation of M on input x.) Thus, if M does not halt on x then $L(M^*) = \emptyset \notin \mathcal{C}$. Recalling that $M_{\mathcal{C}}$ decides $L_{\mathcal{C}}$, this means that $M_{\mathcal{C}}$ outputs “no” on input M^*, which means M_H outputs “no” on input $M; x$ as desired. On the other hand, if M halts on input x, then the machine M^* constructed by M_H behaves as follows on any input y: it first spends a finite amount of time running M on input x, then ignores the answer and runs M_0 on y. This means that M^* accepts input y if and only if $y \in L(M_0)$, i.e. $L(M^*) = L(M_0) \in \mathcal{C}$. Once again using our assumption that $M_{\mathcal{C}}$ decides $L_{\mathcal{C}}$, this means that $M_{\mathcal{C}}$ outputs “yes” on input M^*, which means that M_H outputs “yes” on input $M; x$, as it should. □