1 Summary

- Two kinds of computational problems: decision problems (yes-no answer), and search or function-evaluation problems (longer answer).

- Inputs to problems encoded as strings of symbols over a finite alphabet of symbols. (Notation: $\Sigma^*$ denotes the set of all strings over the alphabet $\Sigma$.)

- A decision problem is identified with the set of input strings that give a ‘yes’ answer (a language $L \subseteq \Sigma^*$). A search problem is identified with a function $f : \Sigma^* \rightarrow \Gamma^*$, where $\Gamma$ is the output alphabet. Solving an instance of a decision problem means determining if $w \in L$ for the given input $w$. Solving an instance of a search problem means evaluating $f(w)$.

- A general solution is an algorithm for determining $w \in L$, or calculating $f(w)$ for any given input $w$.

- Turing machine: mathematical model of a machine executing an algorithm. It is claimed that any algorithm can be implemented with a Turing machine, and thus this provides a rigorous mathematical definition of what a computation is.

2 What is a computational problem?

2.1 Some computational problems

These are like the example problems in Lecture 0.
1. Given two non-negative integers $m, n$, find the sum $m + n$.

$$77 + 231 \mapsto 308$$

2. Given an integer $n > 1$, determine if $n$ is prime.

$$91 \mapsto \text{no}$$
$$97 \mapsto \text{yes}$$

3. Given a graph $G$, determine if there is a path that visits every vertex exactly once. (Hamiltonian path.) The pictured graph has a Hamiltonian path (find it).

4. Given a graph $G$, find a Hamiltonian path, if one exists.

Problems 2 and 3 are decision problems: the answer is ‘yes’ or ‘no’. Problems 1 and 4 are search problems, or function-evaluation problems.

2.2 Encoding problem instances as strings

If we want to prove something about computational problems, we will need a formal definition of such problems. Problem instances (that is, the input—the
pair of summands to add, the graph to test, the polynomial in several variables, etc.) are always encoded as strings, or words over a finite alphabet of letters, or symbols.

If the alphabet is $\Sigma = \{a, b\}$, then $aaba$, $bbaabab$, $a$ are all words. So is $\epsilon$, which we use to denote the empty word.

Some notation: We write $|aaba| = 4$, $|\epsilon| = 0$, etc. for the length of a word. We write $\Sigma^*$ to denote the set of all words over $\Sigma$. Also we abbreviate using exponential notation, for example:

$$aaba = a^2 ba, bbaabab = b^2 a(ab)^2.$$

What do the encodings themselves look like? For primality testing, the problem instance is a single integer, which we can encode by its decimal representation:

ninety-seven $\mapsto 97$,

so that a problem instance is a word over the alphabet $\{0, 1, 2, 3, 4, 5, 7, 8, 9\}$. Of course we have some leeway here—we can encode a problem instance in binary, or some other base, or some different number system.

For the addition problem, a problem instance is a pair of integers, which we might encode as follows:

$$77 + 231;$$

that is, as a word over the alphabet $\{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

What about graphs? There are a number of ways we might encode a graph. We could simply list all the edges Each edge is a pair of vertices, and if we number the vertices in decimal, we get a word over $\{\#, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, where $\#$ is used to separate vertices in our list.

For example, the graph depicted in the illustration in these notes would be encoded by

$$1\#3\#2\#1\#5\#2\#5\#4\#3\#4\#6\#5\#6,$$

representing the sequence

$$(1, 3), (2, 1), (5, 2), (5, 4), (3, 4), (4, 6), (5, 6).$$

As we will discuss, there is a subtle drawback in this encoding scheme.

If we wanted, we could do everything over an alphabet of two letters 0 and 1, and encode every problem instance as a sequence of bits.
Now that we know how to encode problem instances as strings, how do we define the problem itself? We can think of the decision problem as the set of all problem instances for which the answer is ‘yes’. Thus the problem is a set of strings, or a language. For instance, Problem 3 above is treated as identical with the set of all strings that are encodings of graphs with a Hamiltonian path.

We identify a search problem not with a set of strings, but with a function from the set of strings over the input alphabet to the set of strings over some output alphabet. For example, the addition problem is a function

$$f : \{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*.$$ 

If we let $\text{enc}(m)$ denote the decimal encoding of the positive integer $m$, then the function $f$ is defined by

$$f((\text{enc}(m)) + (\text{enc}(n))) = \text{enc}(m + n).$$

We stress that the argument of $f$ on the left-hand side of the equation is a string, and the right-hand side of the equation is another string. The ‘+’ on the left-hand side is a letter in a string. However the expressions $m, n, m + n$ represent integers.

There is a little problem here. As described, the domain of $f$ should be the set of all strings over the input alphabet, but the formula defining $f$ does not tell us what to do on ‘bad’ inputs like $++23++9+$. There are simple ways to patch this up: We could have the value of $f$ on such inputs be the empty string, or we could add a special error symbol to the output alphabet.

Solving an instance of a decision problem $L$ amounts to determining if a given string $w$ over the input alphabet belongs to $L$.

Solving an instance of a search problem $f$ amounts to finding the value of $f(w)$ for a given input string $w$.

A general solution of the decision problem is an algorithm for determining, given any $w$, if $w \in L$. A general solution of the search problem is an algorithm for computing $f(w)$, given any $w \in L$.

So what is an algorithm?

### 3 What is a computation?

#### 3.1 Warmup: binary addition

Here is an algorithm you all know: adding two integers. I’ve illustrated this in binary, because it is easier to describe.
Table 1: Addition in binary. The inputs are written in rows 1 and 2, the outputs and scratch work (for carrying) in rows 0 and 3.

The algorithm is described in detail in the framed text. Observe that the algorithm is independent of the size of the summands; it works for any number of bits.

Turing's great insight: All algorithms work this way. For example, we could describe multiplication in a similar fashion. It would just need more elaborate instructions, and perhaps the use of additional marks to help you keep track of where you are.

3.2 The Turing machine (abstract definition of an algorithm)–informal description

- The workspace is a tape that extends infinitely in both directions, each cell of which holds one symbol.

- At each step, there is a current position. Think of this as a ‘read-write head’ that can both read the symbol at the current position and write a symbol in that square.

- At each step, the machine is in one of a finite number of states–these correspond to the instruction labels 1-7 in our example.

- Tape symbols include both the input symbols and a finite collection of auxiliary symbols, including a special blank symbol □. At each step, all but finitely many of the tape cells hold the blank symbol.

- At each step, the machine updates as follows: Depending on the current state and the currently scanned input symbol, the machine erases the symbol and writes a new symbol into that cell. (The new symbol could be the same as the one that was erased.) It moves the current position one cell to the left or right, and changes to a new state.
0. Move to the rightmost column. Go to 1.

1. If row 1 is 0, go to 3.
   Else if row 1 is 1, go to 4.
   Else if row 1 is blank, go to 6.

2. If row 1 is 0, go to 4.
   Else if row 1 is 1, go to 5.
   Else if row 1 is blank, go to 4.

3. If row 2 is 0 or blank, write 0 in row 3.
   Else if row 2 is 1, write 1 in row 3.
   Move one column to the left.
   Go to 7.

4. If row 2 is 0 or blank, write 1 in row 3
   and move one column to the left.
   Else if row 2 is 1, write 0 in row 3,
   move 1 column to the left,
   and write 1 in row 0.
   Go to 7.

5. If row 2 is 0 or blank, write 1 in row 3.
   Else if row 2 is 1, write 1 in row 3.
   Move 1 column to the left.
   Write 1 in row 0.
   Go to 7.

6. If row 2 is blank, STOP.
   Else if row 2 is 0 write 0 in row 3.
   Else if row 2 is 1, write 1 in row 3.
   Move one column to the left.
   Go to 7.

7. If row 0 is 1, go to 2.
   Else if row 0 is blank, go to 2.

Figure 2: Binary addition in gory detail.
There is a special *halt* state. When the machine enters this state, it stops updating.

### 3.3 Turing machine–formal description–version 1

\[ \mathcal{M} = (Q, \Sigma, \Gamma, q_0, h, \delta). \]

- \( Q \) is a finite set (set of states).
- \( \Sigma \) is a finite alphabet (the input alphabet)
- \( \Gamma \) is a finite alphabet, with \( \Sigma \subseteq \Gamma, \square \in \Gamma - \Sigma \). That is, the tape alphabet contains the input alphabet, and at least one additional symbol, denoting a blank cell.
- \( q_0 \in Q \) (initial state)
- \( h \in Q \) (halt state)
- \( \delta \) is a function
  
  \[ \delta : (Q - \{h\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}. \]

Interpretation:

\[ \delta(q, \gamma) = (q', \gamma', R) \]

means: if current state is \( q \) and currently-scanned symbol is \( \gamma \), the machine writes \( \gamma' \) in this cell (erasing original \( \gamma \)), changes to state \( q' \), and changes the currently-scanned symbol to the next tape square to the right. We could have \( \gamma = \gamma' \), which we typically interpret to mean ‘don’t write anything’, since the symbol in the cell doesn’t change.

A *configuration*, or *instantaneous description* of TM is given by the tape contents, the current position, and the state. Initially the configuration is with the input string \( w \in \Sigma^* \) on the tape, the current position the first letter of \( w \), and state \( q_0 \). We write the configuration this way: if, say, the input is \( aaba \), we write it as

\[ q_0aaba. \]

Suppose \( \delta(q_0, a) = (q_1, X, R) \), then the next configuration is

\[ Xq_1aba. \]

We would then write

\[ q_0aaba \Rightarrow Xq_1aba. \]
We write

\[ c \xrightarrow{*} c' \]

if configuration \( c' \) results from configuration \( c \) after 0 or more steps.

If \( c \) is the initial configuration \( q_0w \), and

\[ c \xrightarrow{*} c' , \]

where \( c' \) is in the halt state, then we say

\[ f_M(w) = w' , \]

where \( w' \in \Gamma^* \) is the tape contents in \( c' \).

The function \( f_M \) is what \( M \) computes. Observe though (and this is important) \( M \) might not halt on every possible input, so \( f_M \) is really only a partial function from \( \Sigma^* \) to \( \Gamma^* \).

### 3.4 Example.

We will describe a Turing machine \( M \) that reads an input bit string, and halts with the reverse of that string on the input tape. That is, for example,

\[ f_M(01001) = 10010 . \]

The idea is this: the machine moves right until it finds the end of the input, and marks this with the symbol \#:

\[ 01001# . \]

A. It then moves left to the next 0 or 1 it finds, crosses it out, and ‘remembers’ the crossed-out symbol in its state.

\[ 0100X# . \]

B. It moves right to the next blank cell, and writes the remembered symbol, then moves left until it finds the #.

\[ 0100X#1 . \]

The machine now repeats steps A and B.

\[ 010XX#1 \]
010XX#10
01XX#10
01XX#10
0XX#10
XXX#10
XXX#10
XXX#10
XXX#10
XXX#10

If, in step A, a blank cell is found, the machine moves to the right, erasing everything up to and including the #.

Let’s implement this by figuring out what states we need and what the state-transition function $\delta$ should do. In the first phase the machine moves to the right until it finds a blank, upon which it writes the mark #, then moves left for the next phase:

$$\delta(q_0, 0) = (q_0, 0, R), \delta(q_0, 1) = (q_0, 1, R), \delta(q_0, \Box) = (q_1, #, L).$$

In the next phase, it moves left past the crossed-out symbols until it finds 0,1, or a blank, and gets ready to move right.

$$\delta(q_1, X) = (q_1, X, L), \delta(q_1, 0) = (q_2, X, R), \delta(q_1, 1) = (q_3, 1, R), \delta(q_1, \Box) = (q_4, \Box, R).$$

If it’s in state $q_2$ or $q_3$, the machine moves right past all the other symbols until it finds a blank, and writes 0 or 1, according to the state. So if $\gamma \neq \Box$:

$$\delta(q_2, \gamma) = (q_2, \gamma, R),$$
$$\delta(q_3, \gamma) = (q_3, \gamma, R),$$

But if $\gamma = \Box$,

$$\delta(q_2, \Box) = (q_5, 0, L), \delta(q_3, \Box) = (q_5, 1, L).$$

Then it moves back to the #, and repeats the computation beginning in $q_1$.

$$\delta(q_5, \gamma) = (q_5, \gamma, L),$$

if $\gamma = 0$ or 1.
\[\delta(q_5, \#) = (q_1, \#, L).\]

If the machine is in state \(q_4\), it cleans up, erasing symbols up to and including the \(\#\).

A complete run of the machine on the length 3 input 100 takes about 40 steps to reach the halt state. The first few configurations are

\[q_0001 \Rightarrow 0q_001 \Rightarrow 00q_0\square \Rightarrow 001q_0\# \Rightarrow 00Xq_3\# \Rightarrow 00x\#q_3\square \Rightarrow 00xq_5\#1\]

Note that \(\delta\) has not been completely specified: for instance, we have not defined \(\delta(q_5, X)\). We can set its value arbitrarily, since we can never be looking at the symbol \(X\) in state \(q_5\). (Better yet, we might have the machine halt if it ever has an unexpected symbol, for example a \(\#\) in state 0. This could mean that the input was prepared incorrectly. The resulting tape contents won’t make any sense, of course—a case of ‘garbage-in, garbage-out’.)

We can depict the behavior of the TM compactly in a state-transition diagram, depicted below. If \(\delta(q_i, \gamma) = \delta(q_j, \beta, R)\), then we write

\[\gamma \rightarrow \beta, R\]

as the label of the arrow from \(q_i\) to \(q_j\). Some conventions: We can leave off the \(\beta\) if \(\gamma = \beta\) (in other words if the new symbol on the scanned square is the same as the old one) and list several different \(\gamma\) before the arrow if the behavior of the machine is the same for each of these symbols.
Figure 3: State-transition diagram of the Turing machine example in this section.