Chapter 2: Varieties of Turing Machine

The TM with
  a 2-way infinite tape,
  taking its input from the tape,
  writing its output on the tape,
  moving and writing in separate steps
is not the only TM model that’s been designed.

We’ll look at several variations on TM design, and compare their computing power.

- 2-way tape
- Accepts by writing 1
- One tape
- Deterministic

  Variant: 1-way tape
  Variant: Accepts by final state
  Variant: Multiple tapes
  Variant: Nondeterministic

1. Turing Machines with one-way infinite tape

A TM can be restricted to have its tape be only one-way infinite.

**Input:** Written on tape, starting at the far left end, with tape head scanning leftmost symbol
  If there are several input arguments (as in numeric functions), separate the input strings by a single blank.

**Output:** Written on tape, starting at the far left end, with tape head scanning leftmost symbol

**Extra features:**
  End marker \( \lambda \) (so machine can tell when it’s at the end of the tape)

**Example:**

\[
\begin{array}{cccccccc}
\lambda & a & b & a & a & B & B & B & \ldots \\
\end{array}
\]

Copyright © 2010 Nancy Lynn Tinkham
Theorem:
Let $f$ be a $k$-ary number theoretic function. Then there exists a TM with 2-way infinite tape that computes $f$ iff there exists a TM with 1-way infinite tape that computes $f$.
(Similarly for language recognition/acceptance and transduction.)

Proof:

1. **Transform 1-way $\rightarrow$ 2-way**

   Easy:
   - Write a $\lambda$ on the left end of the input.
   - Run the instructions for the 1-way.
   - Erase the $\lambda$.
   - I.e., simply ignore the half of the tape that's to the left of the input.

2. **Transform 2-way $\rightarrow$ 1-way**

   Simulate the 2-way-infinite tape by a 2-track 1-way-infinite tape.
   (We break the 2-way tape into 2 pieces and fold it in half, so to speak.)
   That is, each tape square will contain a pair of symbols, together with a third symbol (U or D) representing which track we’re currently scanning. (In this 2-track machine, only one track will be "active" at any given time.)
   $U^a_b$ will mean that the upper track contains $a$, the lower track contains $b$, and the upper track is active.
   $L^a_b$ will mean that the upper track contains $a$, the lower track contains $b$, and the lower track is active.

Copyright © 2010 Nancy Lynn Tinkham
A completely blank square that has never been scanned will still be represented by B (no L or U).

Example:

```
B a b b | a b B
↑
```

if broken just before the 2nd a, becomes

```
λ a b B B (lower track active)
 b b a B
```

i.e.,

```
λ L^a_b L^b_b L^B_a L^B_B
```

To simulate rewriting a symbol, rewrite the symbol on the appropriate track.
To simulate a L move, move L if upper is active, R if lower is active.
To simulate a R move, move R if upper is active, L if lower is active.
A left move at the end of the tape requires switching tracks:

- Scan through all symbols on tape replacing $U^a_b$ by $L^a_b$ (upper $\rightarrow$ lower),
- or $L^a_b$ by $U^a_b$ (lower $\rightarrow$ upper),
- then return to left.
To simulate a numeric computation,

1. Rewrite the input from $\lambda 1111$ form to $\lambda U_B^1 U_B^1 U_B^1 U_B^1$ form.
2. Simulate the $\delta$-transitions as described above.
3. Rewrite the output from 2-track form to 1-track form.
   The computation (if it halts) will halt scanning the representation of the leftmost 1 in a sequence of 1s. This 1 may be on either track.
   a. If this 1 is on the upper track, then we change all $U_B^1$s to 1s and $U_B^B$s to Bs.
      (We can stop the scan-and-replace when we reach a B.)
      We then copy the 1s down to the left, if necessary, so that the sequence of 1s starts at the left end of the tape.
   b. If this 1 is on the lower track, then we change all $L_B^1$s to 1s and $L_B^B$s to Bs. Each $L_B^1$ must be changed to a 1, plus an extra 1 must be written at the end of the sequence of 1s (since $L_B^1$ represents 2 1s).
2. Turing Machines that accept by final state (terminal state)

Context: Language acceptance

A TM can indicate acceptance by the state it halts in instead of by writing symbols on a tape.

**Input:** (same as for standard TM)
- Written on tape, with tape head scanning leftmost symbol
- If there are several input arguments (as in numeric functions), separate the input strings by a single blank.

"Output":
- **Acceptance:** halt in final state
- **Non-acceptance:** halt in a different state, or don’t halt
Example: TM that accepts $a^+b^+$ by final state
Theorem:

Let $L$ be a language over alphabet $\Sigma$. Then there exists a TM that accepts $L$ by writing 1 on the tape iff there exists a TM that accepts $L$ by final state.

Proof:

1. Transform Final state $\rightarrow$ tape

   Idea: on entering final state of old TM, the new TM erases all its tape contents and writes a 1.

   To do this, we have to know where the tape contents start and stop. Add **endmarker** symbols $\lambda$ and $\rho$ (*left* and *right*) to $\Gamma$. At the beginning of the computation, write $\lambda$ on the left of the input and $\rho$ on the right. Whenever the computation expands or contracts the input, copy $\lambda$ ($\rho$) to the left or right one square as needed.

   E.g., to write an $a$ at the left of $\lambda ba \rho$, the tape becomes $\lambda aba \rho$.

   At the end of an accepting computation, erase $\lambda$ and $\rho$ along with other work symbols, so that only 1 remains.

   (*For more details of machine construction, see text.*)

2. Transform Tape $\rightarrow$ final state

   Idea: Whenever we would halt with 1 on the tape, halt in a final state instead.

   The states that potentially may make the old TM halt scanning a 1 on an otherwise blank tape are those that have no outgoing 1 arc.

   At each of these states, we’ll add a 1-transition to a set of states that checks whether the entire tape is blank except for the 1. If it is, halt in an accepting state; otherwise, halt in a non-accepting state.

   To enable scanning the entire non-blank portion of the tape, add endmarkers $\lambda$ and $\rho$ as in part I.

   ($\lambda$ and $\rho$ do not have to be erased at the end, since only the ending state is important.)

   (*For more details of machine construction, see text.*)
3. Multitape Turing Machines

A TM can be given multiple tapes to work on.

Storage:

- An input tape (the first tape), read-only
- $k$ work tapes
- An output tape (the last tape), write-only
- A read-write head for each tape;
  tape heads can move independently of each other
  (Note, however, that the $\delta$ function takes into account all currently-scanned tape symbols.)

Input:

- Written on input tape, with tape head scanning leftmost symbol
  If there are several input arguments (as in numeric functions), separate the input strings by a single blank.

Output:

- Written on output tape, with tape head scanning leftmost symbol
Example: 3-tape machine to accept palindromes

```
Example: 3-tape machine to accept palindromes

a: R
B: B
B: B

0
B: L
B: B
B: B

b: R
B: B
B: B

1
B: B
B: B
B: B

2
b: b
B: B
B: B

3
B: B
B: B
B: B

4
B: B
B: B
B: B

5
B: B
B: B
B: B

a: L
a: R
B: B

b: L

Copyright © 2010 Nancy Lynn Tinkham
```
Theorem:

A language \( L \) is accepted by some multitape TM iff \( L \) is accepted by some single-tape TM.

A language \( L \) is recognized by some multitape TM iff \( L \) is recognized by some single-tape TM.

A function \( f \) is computed by some multitape TM iff it is a computed by some single-tape TM.

Proof of function portion of theorem:

1. Transform Single tape \( \rightarrow \) multitape

   Use a 3-tape machine.
   Copy the input onto the work tape.
   Simulate the actions of the old machine on the work tape.
   At end, copy the work tape onto the output tape.

2. Transform Multitape \( \rightarrow \) single tape

   Use a multitrack tape, with 2 tracks for each tape of the old machine.
   One track holds the tape contents;
   the other track holds an H at the square scanned on the old machine’s tape, and B elsewhere.

Example:

\[
\begin{array}{cccc}
1 & 1 & 1 & B \\
\uparrow \\
B & 1 & 1 & 1 \\
\uparrow \\
\end{array}
\]

\textit{becomes}

\[
\begin{array}{cccc}
1 & 1 & 1 & B \\
H & B & B & B \\
B & 1 & 1 & 1 \\
B & B & H & B \\
\end{array}
\]
To represent a multitrack tape on a single tape, assign a single symbol to each possible combination of track symbols.

For example, table 2.3.1 in the book shows a correspondence between the possible symbols on a 4-track tape with $\Sigma = \{1\}$ and the digits 0-F.

$1H1H = 0, 1H1B = 1, ..., BBBB = F$.

Alternatively, you can think of the tape alphabet containing symbols like $1H1H$.

Completely blank (on all tapes) symbols that have never been scanned are simply represented by $B$.

To simulate a $\delta$-transition of the old tape,

Start at the leftmost H.

Scan to the right to determine the symbol scanned by the leftmost tape head, the next-to-leftmost tape head, etc.

Use this information to determine what the next configuration of the machine should be.

Then scan back to the left, writing the appropriate symbols (or making appropriate $B \leftrightarrow H$ changes).

Stop at the location of the new leftmost H.

At the end of the computation, to produce output in the appropriate form, replace all symbols encoding "1 on the output tape" by 1, and all other symbols by B. Halt scanning leftmost 1.
4. Universal Turing Machines

All the TMs so far have been designed to compute only one function. This is different from our notion of the modern stored-program computer, where a single all-purpose computer is designed to take a program as data, take input as further data, and run the program on the input data — and then go on to run a different program next.

A universal Turing Machine is like our stored-program computer: it takes a description of a TM as part of its input, and it simulates the action of the specified TM on the input data. It is thus able to simulate any other TM.

Think of what you’d need in a Java program to simulate a TM:

- the TM’s program
- the current state
- the current tape contents (initially, the TM’s input)
4.1. Encoding Turing Machines

Several encodings of TMs are possible. It doesn’t matter which one we use, as long as it’s possible:

- To go from a TM to its encoding
  It’s acceptable for a TM to have more than one encoding; the $\delta$-transitions could be listed in a different order, for example.

- To go from an encoding to a TM
  There must be only one TM for a given encoding.

In both of the coding schemes we look at, we’re going to make the TM-description alphabet as small as possible.

- States will be designated $q, q', q''$, etc.
- Tape alphabet symbols will be designated $s, s', s''$, etc.
  I.e., list the tape alphabet in some order, and make $s'$ stand for the first symbol in the alphabet, $s''$ for the second, etc.
  $s$ will represent $B$.
- Moves are still L and R.

A transition function will be represented by a list of quadruples

$\text{Old state, Old symbol, New symbol or move, New state}$

separated by a delimiter (say, ;).

We can now represent any Turing machine using the 7 symbols

$,$ $q$ $s$ $R$ $L$ $;$ $'$
Example: TM to accept $b^n a$ ($\Sigma = \{a, b\}$)

State representations:

$q_0$ $q$
$q_1$ $q'$
$q_2$ $q''$

Alphabet symbol representations:

B $s$
a $s'$
b $s''$
1 $s'''$

Resulting string:

$q, s'', s, q'; q', s, R, q; q, s', s''', q''$

It can be useful to represent a Turing machine as a number instead of as a string of symbols. (Some proofs, for example, depend on there being a natural number corresponding to each TM.)

Two methods of converting TM-strings into numbers are discussed in chapter 2: ASCII-style encoding, and Gödel encoding.

(This is not an exhaustive list of possibilities.)
4.1.1. ASCII encoding

Represent each symbol by a 3-digit binary number from 0 to 6 (000 to 110).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>000</td>
</tr>
<tr>
<td>q</td>
<td>001</td>
</tr>
<tr>
<td>s</td>
<td>010</td>
</tr>
<tr>
<td>R</td>
<td>011</td>
</tr>
<tr>
<td>L</td>
<td>100</td>
</tr>
<tr>
<td>;</td>
<td>101</td>
</tr>
<tr>
<td>'</td>
<td>110</td>
</tr>
</tbody>
</table>

To represent the entire TM, concatenate the codes for each of the symbols in the string:

```
001 000 010 110 000 010 000 001 110 110 000 001 110 101
```

```
001 110 000 010 000 011 000 001 101
```

```
001 000 010 110 000 010 110 110 000 001 110 110
```

Now interpret the string as a binary number.

To decode the number, take the first three digits, look up the symbol they represent; take the next three digits, etc.
4.1.2. Euler-Gödel encoding

For Gödel encoding, we will use prime numbers as placeholders and exponents to represent symbols.

Represent each symbol by a number from 1 to 7.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>1</td>
</tr>
<tr>
<td>q</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>R</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>5</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
</tr>
<tr>
<td>′</td>
<td>7</td>
</tr>
</tbody>
</table>

To represent the entire TM, use the prime numbers (2, 3, 5, 7, 11, ...) in ascending order; raise the \( n^{th} \) prime to the power corresponding to the \( n^{th} \) symbol in the TM’s string:

\[
q, s, \prime, s, q, 37^2, 41^7, 43^1, 47^3, 53^1, 59^4, 61^1, 67^2, 71^6
\]

(etc.)

Multiply this together to get the resulting number.

To decode the number,

- Find the number of 2s in the prime factorization by repeatedly dividing by 2; that number (exponent) is the encoding of the first symbol.
- Find the number of 3s in the prime factorization by repeatedly dividing the result of the previous step by 3; that number (exponent) is the encoding of the second symbol.
- (etc., for each prime number, until you have completely factored the number, leaving 1)

Note that any positive natural number has a unique prime factorization, so only one TM can be produced from one Gödel number.
4.2. A Universal Turing Machine

(Note: I’m being a little more specific here than the textbook author is, for clarity. The choices I’ve made about which tape hold which scratch work are arbitrary — tape 2 could have held program instead of tape 1, etc. Indeed, all the work could have been done on a single tape, with enough patience and delimiters, or by using an encoding of a multi-track tape — see the earlier proofs.)

A universal Turing Machine (for function computation) is one which starts with a tape containing

\[ m + 1 \text{ 1s, followed by a blank, followed by} \]
\[ n_1 + 1 \text{ 1s, followed by a blank, followed by} \]
\[ n_2 + 1 \text{ 1s,...followed by a blank, followed by} \]
\[ n_k + 1 \text{ 1s} \]

(and blank otherwise, scanning the first 1)

and computes \( f(n_1, n_2, ..., n_k) \),

where \( f \) is the \( k \)-ary function computed by the TM with Gödel number \( m \).

(Note: Any of the string \( \rightarrow \) number encodings can be used — ASCII, Gödel, etc.)

How the machine works

Setup:

An input tape, an output tape, and some (initially blank) work tapes.

Work tape 1 will hold the program of the TM to be simulated.

Work tape 2 will hold a representation of the current state of the TM being simulated.

Work tape 3 will hold the tape contents of the machine being simulated. (The input will be copied onto work tape 3 initially, and the tape will be updated as the computation progresses. At the end, work tape 3 will hold the original machine’s "output".)
**Steps:**

1. Decode \( m \), and write the program it represents onto work tape 1.

2. Write \( q \) (the start state) on work tape 2.

3. Copy the inputs \( n_1 + 1, \ldots, n_k + 1 \) (separated by blanks) onto work tape 3. Move work tape 3’s tape head to the beginning of the first input value.

4. Repeatedly:
   
   a. Look through to program on tape 1 to find a \( \delta \)-transition for the current state (on work tape 2) and the current symbol (on work tape 3).

   b. If no transition exists,

      If work tape 3 contains the representation of a number (i.e., an unbroken string of 1s, scanning the leftmost 1), then copy this number onto the output tape.

      (Work tape 3 can use endmarkers \( \lambda \) and \( \rho \) to make this possible.)

      (Else do not copy anything onto the output tape.)

      Halt.

   c. If a transition does exist, update the state on tape 2 and the symbol on tape 3 appropriately.
5. Nondeterministic Turing Machines

A Nondeterministic TM can have 0, 1, 2, or more outgoing transitions for a given <state, symbol> combination.

If 0 or 1 transitions, act as in a deterministic TM.

If 2 or more transitions, any one of the transitions may be followed.

A NTM accepts a string if any one of the possible computations accepts the string (i.e., ends with a 1 on an otherwise blank tape).

A NTM computes a function if any one of the possible computations ends with the output value of the function on the (otherwise blank) tape.
Example: Nondeterministic TM to accept \( \{ w \mid aab \text{ is a substring of } w \} \)

\( \Sigma = \{a, b\} \)

Sample string: \textbf{aaaabab}

Algorithm:

Nondeterministically guess the correct position to look for the substring.
Verify that the next 3 letters are \textit{aab}.

Nondeterministic: State 0 has 2 \textit{a}-transitions.
Example: Nondeterministic TM to accept \( \{ w \mid w \text{ has even length or } w \text{ starts and ends with } b \text{ (or both)} \} \)

\[ \Sigma = \{a, b\} \]

Sample strings: \( abaaab \) (even length), \( baaab \) (starts & ends with \( b \)), \( babb \) (both)

Nondeterministic:
1) Guess which sublanguage at state 0
2) Guess end of string at state 9
Theorem:

A language $L$ is accepted by some nondeterministic TM iff $L$ is accepted by some deterministic TM.

Proof:

1. **Transform Deterministic $\rightarrow$ Nondeterministic**

   Trivial: A deterministic TM is a special case of a nondeterministic TM, in which all the <state, symbol> pairs happen to have 0 or 1 transitions.

2. **Transform Nondeterministic $\rightarrow$ Deterministic**

   Idea: Simulate each possible computation of the NTM, one at a time. If any one of the computations accepts, stop and write a 1 on the output tape; otherwise, go on to the next possible computation.

   **How do you enumerate computations?**

   1. Order all the symbols of $\Gamma \cup \{B\} \cup \{L, R\}$; e.g.:
      
      $a < b < 1 < B < L < R$

   2. Order all the states:

      $q_0 < q_1 < q_2 < q_3 \ldots$

   3. Represent a computation sequence as a sequence of quadruples; e.g.:

      $[<q_0, a, b, q_1>, <q_1, b, R, q_2>]$

      is an example of a computation of length 2 that starts in $q_0$, writes a $b$ and goes to $q_1$; then moves right and halts in $q_2$.

   4. List the possible computations of the machine in lexicographical order (i.e., "alphabetical" order, using the character order above), listing computations of length 1 first, then length 2, then length 3,....
Simulate the NTM using a multitape DTM.

Input tape
Work tape 1: Holds the current instruction sequence of the NTM.
Work tape 2: Holds the simulated computation of the NTM
Output tape

Simulation:
Repeatedly:
  Copy the input tape to work tape 2.
  Generate the next NTM instruction sequence, and store it on work tape 1.
  Simulate that instruction sequence on work tape 2.
  If the computation accepts, write a 1 on the output tape and halt.

Note: Increasing-length computation sequence is important here; it keeps us from getting "stuck" on an infinite computation if a finite one exists.
5.1. Time analysis of Nondeterministic Turing Machines

New definition of cost:

For $w \in L$:

The cost of a NTM computation for input word $w$ will be the number of steps in the shortest computation that accepts $w$.

For $w \notin L$:

The cost of a NTM computation for input word $w$ will be the number of steps in the shortest computation for $w$.

Definition:

Let $M$ be a nondeterministic, single-tape Turing machine and let $n$ be an arbitrary natural number. Then

$$time_M = \text{the maximum cost of } M\text{'s computation for any input word } w \text{ with } |w| = n \text{ and such that } M \text{ has some computation sequence for } w \text{ that does terminate.}$$

(As before, we ignore non-terminating computations. Note that, for strings with terminating computations, we are taking the maximum (over strings of length $n$) of minimum-computations.)

Definition:

A language $L$ is said to be polynomial-time nondeterministically Turing acceptable if there exist both a nondeterministic Turing machine $M$ and a polynomial $p(n)$ such that $M$ accepts $L$ and, for any $w \in L$, $M$ accepts $w$ in $O(p(|w|))$ steps.

Definition:

$NP$ is defined to be the class of polynomial-time nondeterministically Turing-acceptable languages.

That is, $L$ is in $NP$ if $L$ can be accepted by a NTM in polynomial time.

It is unknown whether $P = NP$. 
We were able to simulate a NTM with a DTM, but this simulation requires exponential time, in general:

We simulate as many computations as there are paths through the NTM.
Assume the NTM has $s$ states, and that it accepts string $w$ in time $t(n)$, where $n = |w|$.

If every state is connected to every other state (including itself) — the worst case — then the number of paths of length $k$ is $s^k$. For each path, we do $t(n)$ work, and the number of paths of length $t(n)$ is $s^{t(n)}$. Thus, the DTM must do (at least) $O(s^{t(n)})$ work as it simulates the NTM.

There may be a more efficient way for DTMs to simulate NTMs; we don’t know. If there is, then P = NP. If, on the other hand, there are some problems in NP that require exponential time on a DTM, then P $\neq$ NP.

We’ll return to this topic in chapter 8, "The Bounds of Computability".