Module 8

Introduction to Turing machines

Simplifying computers down to automata?

CS 360: Introduction to the Theory of Computing
Winter 2022

Collin Roberts
University of Waterloo

Topics for this module

- Introduction to the limits of programs.
- Turing machines
- How to program a Turing machine
- Variations on Turing machines

Turing machines, and the theorems about them, are the major contributions of computer science to philosophy.

Why might some problems be hard to solve?

Our primary goal in this module of the course:

- Are there problems computers cannot solve? If so, then what sorts of problems are they?
- It turns out that the class of languages for which a computer cannot decide membership is very large.
- In fact, almost every language is not the language of a program written in a normal computer language.

Let's think for a bit about what might be a hard problem for a computer to solve.

- Given a function, does it return 1?
- Why would that be hard for a computer to test?

Some programs are easy, some are not.

```
def easy():
   return 1

def hard():
   i = 1
   while i >= 1:
     for j in range (1,i+1):
        for k in range (1,j+1):
           for n in range (3,k+1):
              if i**n== j**n+k**n:
                 return 1
     i = i+1
```

Does the hard program return 1?

- No, because Fermat's Last Theorem is true.
- Instead, it runs forever.

Simple properties can be arbitrarily hard to test

Suppose there is an interesting fact in number theory, or combinatorics, or whatever, and we want to know if it is always true, for all integers i, j, k. We might proceed as follows, to attempt to find a counterexample:

- Write a test for that property.
- Enumerate over all possible choices of i, j, k.
- If the test is false, return 1.
- If the function ever returns, then the property is not always true.

If we could write a program that can test other functions for a specific result, then it could verify any math result.

Another reason it is not easy to solve

Suppose we have a Python function, H that determines if a computer program, P will return 1, when it is run with the input I.

- If P does return 1, then H does, too.
- If not, then H returns 0.

Here is a new program, H_1, which uses H:

```
def H1 (program, input):
  if H (program, input):
     return 0
  else:
     return 1
```

What is so weird about H_1?

Our program H_1, when run on program P and input I, returns either 0 or 1, depending on whether P returns 1 or not.

- If P does return 1 on input I, then H_1 does not.
- If P does not return 1 on input I, then H_1 does.

Let's have one more program, H_2:

```
def H2 (program):
  if H (program, program):
     return 0
  else:
     return 1
```

Here, H_2 behaves just like H_1, except that the program P is now used as the input. We imposed no constraints on what the input I could look like, so for now let's see what happens if we allow a program to be passed in as the input in this way.

Now, we reach the end

What happens when we call `H2 (H2)`?
Consider the test in the `if` command:

- If H_2, with the input H_2 returns 1, then the `if` test is `True`, and the program H_2 returns 0, when run with the input H_2.
- If H_2, with the input H_2 returns 0, then the `if` test is `False`, and the program H_2 returns 1, when run with the input H_2.

Huh?!
The basic problem:

- We have designed H_2 to be one of the programs we need to be able to study.
- But it is also inspecting its own result.
- There seem to be limits to what we can program.

We will come back to this in Module 9, when we can talk about Turing machines, not Python programs.

Turing machines

- We have been studying simple models of computation.
- We need a model of computation which is more similar to our understanding of how people and computers work.

In particular, we need to be able to access memory more robustly than we can in a pushdown automaton.

How do computers compute?

A simpler question: how do people compute?

- There is a distinction between short-term and long-term memory.
- We can use a small amount of information in short-term memory.
- There is a potentially enormous amount of information in long-term memory (e.g. paper notebook, etc. ...)

When we compute:

- We pull information from long-term memory into short term.
- We work with the contents of short-term memory.
- When finished, we write results to the long-term memory,
- Until we have solved our problem.

As an automaton model

Turing's model of a computing machine has two parts:

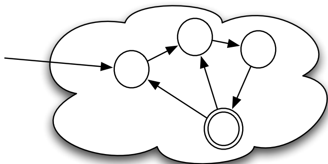
- A finite automaton
  - has short-term memory
  - tells us what to do with the short-term information, and what to write to long-term memory
- A 1-dimensional tape which represents the long-term storage
  - This machine does not access memory like a regular computer, but they are equivalent.

Church/Turing thesis: Anything we can do with a Turing machine we can do with any other reasonable computing model.

Turing machine = Finite automaton plus memory tape

...BBB01010001110101111BBB...

tapehead



- Long-term memory is only accessed at the tape head: we can only see one letter of memory at a time.

1 step in the TM:

- Given the current state in the FA, and the letter at the tape head
  - Move to a different state
  - Maybe change the letter at the tape head (or leave it alone)
  - Move tape head left or right

Formalization

To describe this, we need a 7-tuple:

- Finite automaton control:
  - Q: finite set of states of the automaton
  - q_0: the start state of the automaton
  - F: the set of accepting states of the automaton
  - Σ: the alphabet for words of the machine's language
- Tape features:
  - Γ, the tape alphabet
  - B, the blank character for the tape
  - Note: $B \in \Gamma$, but $B \notin \Sigma$.
- And a transition function:
  - δ: transition function. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
  - If $\delta(q, a) = (p, b, L)$, then the machine switches to state p, places a b where the tape head had pointed to a, and moves the tape head to the left.
  - The δ function need not be a full function: if there is no value of $\delta(q, a)$, the machine crashes in state q upon seeing the symbol a.

The machine halts when we reach an accept state or crash. It can also run forever.

More specifics about TMs

The machine has an infinite tape. When we launch the TM with input $w \in \Sigma^*$:

- $|w|$ consecutive positions of the tape are filled with w,
- the tape head points to w_1, the first character of w, and
- all of the rest of the tape is filled with the blank symbol B.

The machine starts with the input on the tape. Remember: the input could be of arbitrary size, but is always a finite string.

Instantaneous descriptions for Turing machines

Instantaneous description of the TM's state after some number of steps of computation:

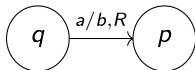
- Current state, q.
- The contents of the tape, $X_1 X_2 \cdots X_k$. Surrounding it is an infinite number of blanks in both directions.
- The current position of the tape head. We will underline the current position; your text does something much grosser.
- If we want the tape head pointing to the first symbol, w_1 of a sequence of symbols, $w = w_1 w_2 \cdots w_n$, we will underline w, referring to the first symbol in it.

So, at the beginning of the execution of the TM, the instantaneous description is written as $(q_0, \underline{w_1} w_2 \cdots w_n)$, or as $(q_0, \underline{w})$.

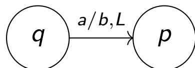
How to transition in the TM

If the current instantaneous description of the machine is $(q, x\underline{a}y)$:

- If $\delta(q, a) = (p, b, R)$, then the new instantaneous description is $(p, xb\underline{y})$. Shorthand this by writing $(q, x\underline{a}y) \vdash (p, xb\underline{y})$.



- If $\delta(q, a) = (p, b, L)$, then the new instantaneous description of the machine is $(p, x_1 \cdots x_{k-1}\underline{x_k}by)$, and we write $(q, x\underline{a}y) \vdash (p, x_1 \cdots x_{k-1}\underline{x_k}by)$.



If there is no value for $\delta(q, a)$, then the machine crashes.

Special transitions in the Turing machine

Special cases:

- Tape head at leftmost nonblank character and we move left: the tape head will move to a new blank character, B: $(q, \underline{a}y) \vdash (p, \underline{B}by)$, if $\delta(q, a) = (p, b, L)$.
- Tape head at rightmost nonblank character: similarly, the tape head will move to a new blank character, B: $(q, x\underline{a}) \vdash (p, xb\underline{B})$, if $\delta(q, a) = (p, b, R)$.
- Erasing the last letter on the tape. For example, if $\delta(q, b) = (p, B, L)$, then $(q, xa\underline{b}) \vdash (p, x\underline{a})$.

Acceptance in the TM

We also have multi-step computations: in the Turing machine M, we write $(q, x\underline{a}y) \overset{*}{\underset{M}{\vdash}} (p, v\underline{b}x)$ if we can move from the first configuration to the second in some finite number of steps.

- To keep the notation uncluttered, we only indicate the machine M when necessary.

The Turing machine M accepts $w = w_1 w_2 \cdots w_n$ exactly if $(q_0, \underline{w_1} w_2 \cdots w_n) \overset{*}{\underset{M}{\vdash}} (p, x\underline{a}y)$ for any accept state $p \in F$ and any string $xay \in \Gamma^*$.

- The tape need not be empty, and
- the input need not have been fully examined.

Now we can define the language of a Turing machine:

- The language of the machine M, $L(M)$, is the set of words which M accepts.
- If L is the language accepted by a Turing machine, then we say that L is recursively enumerable, or r.e. for short.

Rejection in the TM, running forever

How does a TM reject an input?

- If the machine winds up in a state q, pointing at a, and $\delta(q, a)$ is not defined, then it crashes and rejects the word.
- This is called "Halting and rejecting".
- (Later, once we move beyond implementation details and into describing algorithms for Turing machines, we will simply make our algorithms explicitly crash when necessary.)

A Turing machine can also fail to accept an input word w by running forever while processing it.

- This is possible with a PDA, too, or even with an ε-NFA.

If M either accepts w or crashes on it, we say that M halts on input w.

- Halting on every input is a good property for a Turing machine.
- If a language L is the language accepted by a Turing machine which halts on every input, then we say that L is recursive, or decidable.

An example: palindromes

Let's construct our first TM, to accept the language of palindromes, $L = \{x \mid x = x^R\}$.
How would a human do it?

- Start at both ends.
- Match letters until we reach the middle of the word.
- And probably use two fingers to keep track of both positions.

How will our Turing machine do it?

- Look at the leftmost character in the word we have not yet matched.
- Match it with the rightmost character.
- Remove both of them.
- Keep doing this until we have only 0 letters or 1 letter remaining in the middle.

More specific

1. Start with the tape head on first input letter.
2. Repeat, until only 0 or 1 letters are left:
   - Erase the current (leftmost) letter, and remember it.
   - Find the rightmost letter not yet erased.
   - If they do not match each other, then reject.
   - Else, erase that matching rightmost letter.
   - Move to the left end of the string, to find the leftmost character not yet erased.
3. If only 0 or 1 letters are left, then accept.

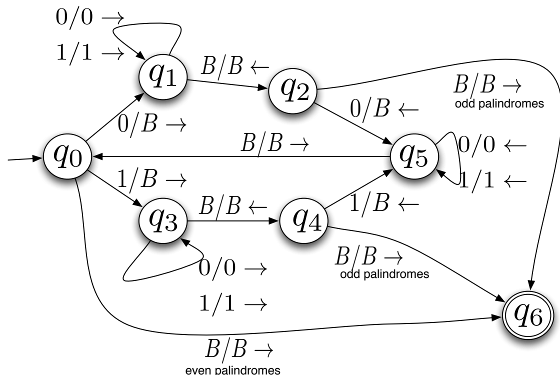
Can we encode this as a TM?

How to encode it as a Turing machine?

- Erase the first letter and remember what it was.
  - Create two parallel tracks in the finite control.
  - These tracks "remember" what letter we are currently attempting to match.
- Find the last letter and compare.
  - Move right as far as possible until we reach a B.
  - Then move back one letter.
  - Compare the two letters. If they differ, then crash!
  - Otherwise, delete the matched letter, and go back to the beginning of the word.
- If all that is left is B or a single character, then we are done.

As a TM

Draw like a DFA, but with a new character for the tape and the arrow for the tape direction after a slash. (Or use *L* and *R* instead)



Seems complicated, but not that bad. The top branch matches 0s at both ends of the word, while the bottom branch matches 1s.

An accepting computation

Consider the palindrome $w = 010$.

$$
\begin{aligned}
(q_0, \underline{0}10) &\vdash (q_1, \underline{1}0) \\
&\vdash (q_1, 1\underline{0}) \\
&\vdash (q_1, 10\underline{B}) \\
&\vdash (q_2, 1\underline{0}) \\
&\vdash (q_5, \underline{1}) \\
&\vdash (q_5, \underline{B}1) \\
&\vdash (q_0, \underline{1}) \\
&\vdash (q_3, \underline{B}) \\
&\vdash (q_4, \underline{B}) \\
&\vdash (q_6, \underline{B})
\end{aligned}
$$

The machine accepts, having deleted the whole word.

A word not in L

Consider the non-palindrome $w = 0100$.

$$
\begin{aligned}
(q_0, \underline{0}100) &\vdash (q_1, \underline{1}00) \\
&\vdash (q_1, 1\underline{0}0) \\
&\vdash (q_1, 10\underline{0}) \\
&\vdash (q_1, 100\underline{B}) \\
&\vdash (q_2, 10\underline{0}) \\
&\vdash (q_5, 1\underline{0}) \\
&\vdash (q_5, \underline{1}0) \\
&\vdash (q_5, \underline{B}10) \\
&\vdash (q_0, \underline{1}0) \\
&\vdash (q_3, \underline{0}) \\
&\vdash (q_3, 0\underline{B}) \\
&\vdash (q_4, \underline{0})
\end{aligned}
$$

...and the machine crashes, rejecting $w = 0100$.

What to learn?

We can encode basic operations in transitions:

- Find the first letter of the string.
- Remember a letter (two sets of state paths)
- Match two letters

And so on...
TMs can also accept non-context-free languages, like

$$L = \{s!s \mid s \in \{a, b\}^*\}.$$

Programming the TM for L

We are not going to draw the machine that accepts this language. (See Section 8.3.2 in the text.) Idea: match letters from one copy of s with those in the other copy.

- First, we ensure that there is exactly one ! character in the word.
- Then, we match the first "remaining" character on each side of the ! character, and remove them.
- Until there is nothing left.

Programming Turing machines

- Store a finite amount of information in the state (e.g. which symbol to match)
- Tag letters of the word with more bits of information, expanding the alphabet.
- Use subroutines where needed.

Important distinctions about languages of TMs

We have seen this once before, but it bears repeating:

- Turing machine M accepts language L if $L(M) = L$: if M accepts w exactly when $w \in L$.
- Turing machine M decides language L if $L(M) = L$ and for all words $w \notin L$, M crashes on w.

Remarks:

- Our machine for palindromes decides the language of palindromes, since it accepts palindromes and crashes on all non-palindromes.
- If our machine instead ran forever on some non-palindromes, it would only accept the language of palindromes.

Recursive versus recursively enumerable

This distinction gives rise to two different kinds of languages:

- A language L is recursively enumerable if there exists a Turing machine M that accepts L.
- A language L is recursive (or decidable) if there exists a Turing machine M that decides membership in L.

Remarks:

- There are languages that are recursively enumerable, but not recursive.
- And there are languages that are neither.
- If a language is recursive, then it is recursively enumerable.
- If a Turing machine M accepts the language L, but does not halt on all inputs, then this does not imply that L is not recursive.
  - You may just have the wrong TM.
  - It might be possible to choose an improved TM which halts on all inputs and accepts L.

Computing a function with Turing machines

We often want to compute the value of a function, not just test membership in a language.
At first glance, these seem like fundamentally different questions:

- What is $2+2$?
- Is $2+2=5$?

For the second of these:

- $L = \{a^i b^j c^k \mid i+j=k\}$ is a language.
- Membership in L answers whether $i+j=k$.

For the first, change the way a Turing machine works.

Computing a function, continued

To compute the function $y = f(x)$:
Use the Turing machine tape for output, not just for input.

- Suppose $(q_0, \underline{x}) \overset{*}{\underset{M}{\vdash}} (q_a, \underline{y})$, where q_a is an accept state.
- In other words M, when the tape is initialized with x, computes the output $f(x)$ and halts.
- Then M computes $y = f(x)$, as desired.

Now, what if M computes $f(x)$ for all possible values of x?

A computable function

Suppose that for all inputs $x \in \Sigma^*$:

- $(q_0, \underline{x}) \overset{*}{\underset{M}{\vdash}} (q_a, \underline{y})$, where $y = f(x)$ and q_a is any accept state.
- We will need to figure out what this means if x or y is ε.

Then we say that M computes the function f.

- M accepts every input x.
- The important thing is the tape contents when the machine reaches an accept state.
- Our assumption that this happens for every input $x \in \Sigma^*$ was very strong. We can weaken this assumption and still get something useful.
- Note that f need not be total: there could be words in Σ^* for which f is not defined. On those inputs, M may crash or run forever.
- The function can have multiple arguments or be multi-valued.
- If $f : \Sigma^* \times \Sigma^* \to \Gamma^*$, then we will have the two arguments for f next to each other on the tape when M starts running, with a blank character B between them.

Computable functions

A function f which is computed by a Turing machine M is called a computable function.

- f might be total (or not) or have multiple arguments (or not).

Numeric functions can be computable, too. Our first computable functions will be unary (i.e. they will take a single input), and return a single output:

- Suppose $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ is a function, and for all $x \geq 1$, we have $(q_0, \underline{1^x}) \overset{*}{\underset{M}{\vdash}} (q_a, \underline{1^{f(x)}})$, where q_a is any accept state.
- Then M computes f, and f is computable.
- Again, for multi-argument functions, the arguments on the tape are separated by blanks.

A simple example

Consider the addition function: $+ : \mathbb{Z}^+ \times \mathbb{Z}^+ \to \mathbb{Z}^+$.

- We encode the positive integer i on the tape as 1^i.
- Then $x + y$ is the concatenation of the two arguments 1^x and 1^y, namely 1^{x+y}.
- This $+$ is computable: we can certainly concatenate two strings with a Turing machine.
- **Exercise:** Construct a Turing machine that performs this computation.

Characteristic functions

- In set theory, the characteristic function of the subset $X \subseteq S$ is the function:

$$\begin{array}{rcccl} \chi_X & : & S & \to & \{0,1\} \\ & & s & \mapsto & \begin{cases} 1 & \text{if } s \in X \\ 0 & \text{if } s \notin X \end{cases} \end{array}$$

  which indicates whether an arbitrary $s \in S$ lies in X or not.
- Hence the characteristic function of the language $L \subseteq \Sigma^*$ is the function:

$$\begin{array}{rcccl} \chi_L & : & \Sigma^* & \to & \{0,1\} \\ & & x & \mapsto & \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases} \end{array}$$

- This function answers the question, "is the word x in the language L?"
- If the Turing machine M computes χ_L:
  - On any input $x \in \Sigma^*$, M always accepts.
  - When M halts, either 1 or 0 is left on the input, with the tape head pointing at that symbol.

If we can compute χ_L, then L is decidable

Suppose M computes χ_L.
Then we can build a machine M' to decide membership in L:

On input x:

- Since M always accepts, M' always gets through the M module.
- Then M' either
  - accepts (if the tape head is pointing at a 1 when M accepts), so $x \in L$, or
  - crashes (if the tape head is pointing at a 0 when M accepts).

Then M' decides membership in L.
If f is computable (or respectively if $\chi(L)$ is computable), then we say that f (or respectively L) has an algorithm.

Using subroutines in general

In this example, M is a subroutine in M'.

- Set up the input for the subroutine.
- Transition into the first state of the subroutine.
- Wait until the subroutine halts (accepts).
- Note that we must correctly handle the possibility that the subroutine runs forever (every subroutine is a Turing machine after all).

We do not use subroutines in Turing machines to shorten the program code.

- We do not care about program length.
- We care very much about ease of understanding.

Using subroutines

Characterize what the subroutine does:

- Identify a common task, and separate it from the flow of the program.
- Give a full specification.
- Create a program that just does the subtask, and plug a call to that subroutine everywhere you used to have all of the code.

Example: inserting a character

The specification of our subroutine for inserting the character a at the current position of the tape head:

- $(q_0, y\underline{z}) \overset{*}{\vdash} (q_a, ya\underline{z})$, where q_a is an accept state in the subroutine machine.
- Constraint: z does not have the blank character in it, so that we know when we have read the entire string z and moved it one character to the right.

We might use this subroutine if we want to insert a different character into a string, by first inserting the character a, and then replacing the a with that character.

Another example: deleting a character

Delete the character the tape head is pointing at:

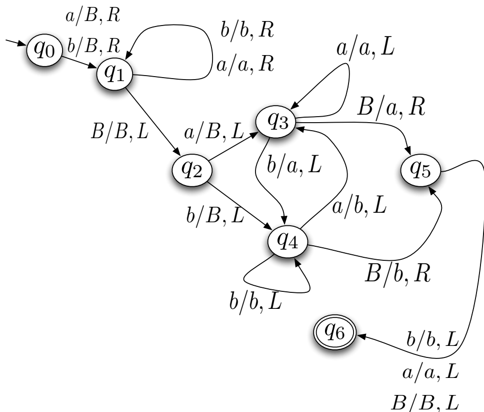
- $(q_0, y\underline{a}z) \overset{*}{\vdash} (q_a, y\underline{z})$, where q_a is again an accept state in the subroutine machine.
- Again, constrain z to not contain any blank character.

How to implement basic string processing in Turing machines?
Very carefully...

Deletion machine

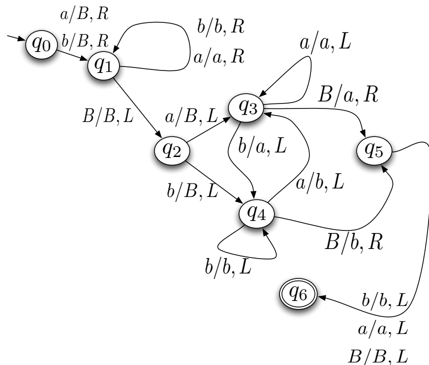
One simple approach:

- Mark the current tape position.
- Go the whole way to the end of the characters on the tape.
- Push left, keeping track of the character we are deleting, until we get to the marked position.
- Then stop.



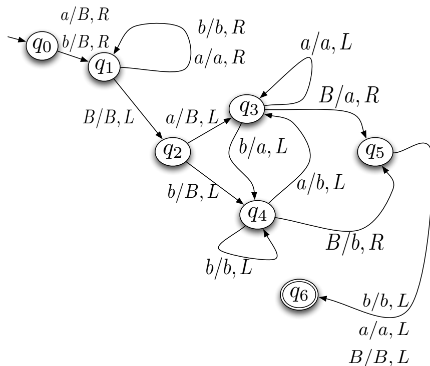
How does that work?

- In state q_0, we delete the current tape head character and move right.
- In state q_1, we move to the right until we have read the entire word on the input.
- Then, we remember the last letter, and put it into the position where the second-to-last letter was, remembering that.
- In state q_3, the previous symbol was *a*.
- In state q_4, the previous symbol was *b*.



How does that work? (cont'd)

- Push the whole way to the beginning of the string, and copy in the last character as we move to state q_5.
- From state q_5, move the tape head back to the correct position.
- Accept in q_6.



An Example

- This is the computation for processing $a\underline{b}ba$.

$$
\begin{aligned}
(q_0, a\underline{b}ba) &\vdash (q_1, aB\underline{b}a) \\
&\vdash (q_1, aBb\underline{a}) \\
&\vdash (q_1, aBba\underline{B}) \\
&\vdash (q_2, aBb\underline{a}) \\
&\vdash (q_3, aB\underline{b}) \\
&\vdash (q_4, a\underline{B}a) \\
&\vdash (q_5, ab\underline{a}) \\
&\vdash (q_6, a\underline{b}a)
\end{aligned}
$$

Storage in the state

Another trick, implicitly just used:

- Imagine finite "working memory".
- Augment the state by storing k bits of information: add 2^k states, and store memory via which state we are currently in.
- If we denote the memory by M, and its values come from a finite set of choices, $\mathcal{M}$, then δ, the transition function becomes $\delta : Q \times \Gamma \times \mathcal{M} \to Q \times \Gamma \times \mathcal{M} \times \{L, R\}$.
- This modified transition function keeps track of both the old and new types of memory.

In our previous example, we could combine q_2, q_3 and q_4 into a single state by storing the previous symbol seen in the memory of M.

Variations on Turing machines

Lots of modifications we can make to augment Turing machines turn out to have no effect on their power.

- Adding finite memory (we have just seen this one)
- Adding multiple tapes
- Adding multiple tape heads
- Nondeterminism

In some sense, the universality is not surprising:

- Church-Turing thesis: Every reasonable model of computation is equivalent in power to Turing machines.

Multi-tape Turing machines

In a multi-tape Turing machine, we have k infinite tapes (for some positive integer k), each with its own tape head.

- The machine's instantaneous description is characterized by what is stored on each of the k tapes, the position of all k tape heads, and the state in the finite automaton.
- Transition function is of form: $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$.
- We might want k tapes so as to store important information in one memory tape without overwriting that input as we do our processing.
- We adopt the conventions that
  - the (finite) sequence of input symbols is placed onto the first tape (with the first tape head pointing to the leftmost character), and
  - all other tapes start off filled with blanks (with their tape heads pointing to arbitrary positions).

Can we do more with a multi-tape machine than with a single-tape machine?

No - Theorem:

- If a language is decidable by a multi-tape Turing machine, then it is decidable by a one-tape Turing machine.
- If a language is accepted by a multi-tape Turing machine, then it is accepted by a one-tape Turing machine.

Proof of equivalence

We will focus on a language accepted by a 2-tape Turing machine, M, and show it is also accepted by a 1-tape Turing machine, M'.

- This argument generalizes to languages decided by M', or to machines with more than two tapes, with few changes.

How to simulate M with M'? First, some housekeeping:

- Suppose that M uses the tape alphabet Γ. (Recall that $B \in \Gamma$.)
- Then let the tape alphabet for M' be $\Gamma \times \{B, *\} \times \Gamma \times \{B, *\}$.
- That is, each position of the single tape for M' is two symbols from Γ, where each position is either tagged with a $*$ or not.
- The $*$ tags indicate the current tape head positions in M, as we simulate running it.
- We may think of the single tape for M' as having four "tracks", to remember these four pieces of information.
- For example, part of the tape might look like

B	0	1	0	B
B	*	B	B	B
B	1	1	1	B
B	B	B	*	B

First, set things up.

- Suppose that the first non-blank character on the first tape of M is $a \in \Gamma$.
- Recall that, at the start of processing, the second tape of M is filled with all blank characters.
- We start by writing the first character on the input tape for M' with $(a, *, B, *)$ (to indicate that the two tape heads start out at the left ends of their respective inputs).
- Then, for every symbol $a \in \Gamma$ on the first input tape for M, we write (a, B, B, B) to the corresponding position on the single tape for M'.

And we move the tape head for M' back to the beginning of the tape.

Using the one-tape TM to simulate one transition in the multi-tape TM

Now, we must make one transition in the Turing machine M'.

- Store M's current state in M''s finite memory.
- Locate the two tape heads in M, by searching from the left of M''s tape, for the $*$ characters in the second and fourth positions of the four-tuple that is a letter in the tape alphabet of M'.
- Store the two corresponding characters in M''s finite memory.
- Now, M' knows the state in M's finite automaton, and the two symbols pointed to by the two tape heads of M.
- Therefore M' can determine which transition M will make.
- Then, M' updates the two values at the sites of the tape heads, (locating them by searching from left to right, again).
- And M' marks the proper positions on the tape to indicate the new positions of the simulated tape heads of M.
- Store M's new state in M''s finite memory.
- Rewind the tape and repeat.
- Declare the accepting states of M' to coincide with those for M, so that M' will accept exactly when M does.

Why use multi-tape TMs?

They are a modeling discipline:
Example: Consider $L = \{w!w \mid w \in \{0,1\}^*\}$.

- We can create a one-tape TM to test membership in this language.
- But it might be easier to create a multi-tape TM:
  - Scan left-to-right to ensure exactly one ! character is present (otherwise, crash).
  - Copy everything after the ! to the second tape (and delete from the first tape, including the ! character).
  - Move to the left of both tapes.
  - Move left-to-right, matching characters in both tapes to each other.
  - If there is a mismatch, or if the strings are of different lengths, then crash.
  - Otherwise, if we get to the end of both words having matched all characters in both, then accept.

A simpler extension: multiple tape heads

We could have multiple tape heads on one tape.

- Start the multi-head machine with its input word on the single tape, and both heads pointing at the start of the word.
- The description of the machine consists of the state, the string on the tape, and the positions of all of the heads:
- $(q, x, p_1, p_2, \ldots, p_k)$ for a k-head machine.
- Transition function: as a function of current state, and what is pointed to by all k tape heads:
  - Go to a new state.
  - Put new values at each of the k heads.
  - Move each of the tape heads left or right.
- One obvious problem: concurrency. What happens when multiple tape heads point to same position?
- Lots of possible ways to determine which tape head gets writing precedence: only the smaller index wins, never allowed, *etc.*

Not hard: same power

Theorem: Turing machines with 1 tape, 2 heads are no more powerful than ones with 1 tape, 1 head.

Proof sketch:

- Have a tape alphabet of the form $\Gamma \times \{B, *\} \times \{B, *\}$, as in the proof for the equivalence of multi-tape machines and single tape machines. The second and third parts of the alphabet symbols indicate the positions of the two tape heads.
- Scan to find the tape characters pointed to by both tape heads.
- Keep track of what is pointed to by the first head when searching for second head
- Keep track of what is to be put at the second head's position.
- Then copy in the new values and "move" the tape heads.

Moral: Two (or more) tape heads are no better than 1.

- **Remark:** We could easily re-do the last example, to decide membership in the language of words of the form $w!w$, with a 1-tape, 2-head Turing machine.

Other memory alterations

- 2-dimensional memory (tape head moves up, down, left or right),
- or even higher dimensions,
- or a multi-tape machine where we store an address in the first tape that we can use to access the second tape: we can then model something like random-access memory, instead of sequential.

Turing machines can be as powerful as normal computers, after all.

Non-determinism

- At a given configuration, there are (possibly) multiple choices for the next transition.
- Follow (implicitly) all possible choices.
- This essentially causes us to have an exponential number of simultaneously running Turing machines.

Question: Is this more powerful than one deterministic Turing machine?

- No.

Details for nondeterminism

A nondeterministic Turing machine can have multiple values for $\delta(q, a)$.

- There can only be a finite number of entries in $\delta(q, a)$.
- We write $(q, y\underline{x}) \vdash (p, w\underline{z})$ if one transition in $\delta(q, x)$ gets us to the second configuration.
- The non-deterministic machine accepts input x if a valid computation exists that brings us from the starting configuration to an accepting configuration.

Note:

- It is not true that all paths must lead to an accepting configurations, any more than in NFAs or PDAs.
- These nondeterministic TMs can only accept languages, rather than deciding them.
  - What should it mean if one thread accepts while another crashes or runs forever?
- They also can not compute functions.
  - What should it mean if two valid computations disagree about the value of $f(x)$?

Example: a nondeterministic Turing machine

Let $L = \{ww \mid w \in \{a, b\}^*\}$.
Here is a sketch of a two-tape head nondeterministic TM which accepts L:

- Start with the possible word in L on the tape, and both tape heads at the start of the word.
- Guess the point in the input where the end of the first copy of w lies.
- Place the second tape head at that point.
- Match symbols at both tape heads, until they either mismatch or both tape heads reach the end of their copy of w.

Nondeterminism: guessing the end of the first copy of w.

Notational change

Change our notation a bit:

- In the 2-head TM, if the first tape head points to a and the second tape head points to b, and we change the first tape head's symbol to c, and the second's to d, while moving the first tape head right and the second tape head left, we will notate that by $(a, b)/(c, d), RL$.
- We add the possibility that a transition does not require a tape head to move, so instead of $\{L, R\}$, our transitions will be from $\{L, S, R\}$ (with S for "stationary").
- We will justify on the next slide why an enhanced Turing machine with stationary tape head moves is no more powerful than a standard Turing machine.
- If both tape heads are at the same position, then the new symbol assigned by the second tape head takes precedence.
- In some states, we might not care about what is pointed to by both tape heads: to ignore what is pointed at by one tape head, put a $*$ in the pair before the slash, as in $(a, *)/(c, d), LS$.
- If we do not change the symbol pointed to by a tape head, we can indicate that with a $*$ after the slash, as in $(a, *)/(*, *), LS$.

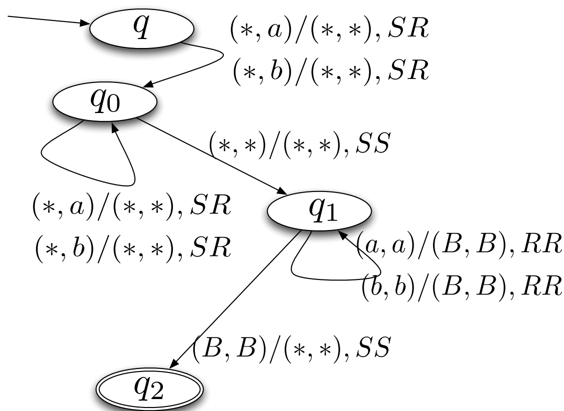
Simulating an enhanced TM with stationary tape head moves, using a standard TM

How would we simulate the transition $\delta(q, x) = (p, y, S)$?

Tape Before: | a | b | $\underline{x}$ | c | d | Tape After: | a | b | $\underline{y}$ | c | d |

- In the standard TM, include two states for each state in the enhanced TM (if the enhanced TM has a state q, then the standard TM has states q and q_S).
- Execute L and R tape head moves as in the enhanced TM.
- To mimic an S tape head move,
  - Write y at the tape head.
  - Move the tape head R.
  - Change to state q_S.
  - Add transitions in state q_S to:
    - Re-write the same character at the tape head.
    - Move the tape head L.
    - Change to state p.
- It is clear from the construction that this will mimic the required S tape head move correctly, and so we are finished.

The TM



- In q_0, identify boundary: by scanning across the entire string, we have the possibility of starting the boundary anywhere.
- In q_1, we compare the two possible copies.
- If we reach the end of both copies simultaneously, then we accept.

The crucial property: computation branches

At each step in state q_0, the machine makes a choice:

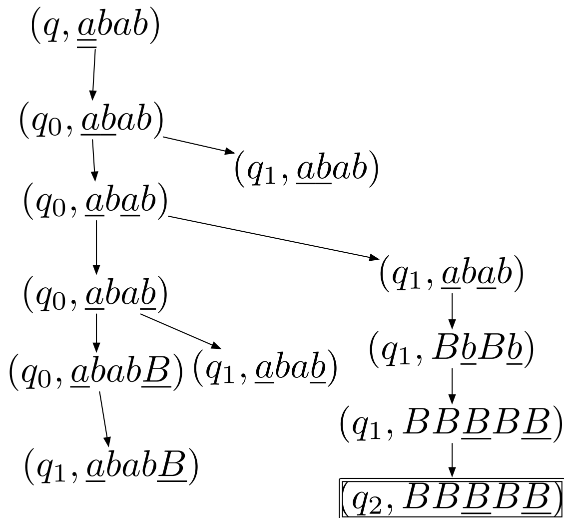
- Either remain in that state and push the second tape head to the right, or
- Move to state q_1.

Different choices lead to different branches on the computation tree.

- Order possible valid choices at a state: if staying in the state is choice 0, and moving states is choice 1, then there is the 001 branch of the tree, the 00001 branch, and so on.
- After k steps, there might be an exponential number of operational branches.
- Include the start state q: this ensures that we move the second tape head at least one position to the right before nondeterministically choosing to move to state q_1 from state q_0.

An example computation tree

Here is the computation tree for *abab*:



Only the boxed path accepts. The others all crash.

How powerful is nondeterminism?

If we have n steps in the computation, and 2 choices at each level, there could be 2^n branches! Are nondeterministic TMs more powerful?

- No.
- There is the possibility that they are faster (see the *P vs. NP* problem in 341).
- But we only care about what they can do, not how quickly they do it.

Theorem: If L is accepted by a nondeterministic Turing machine M, then there exists a deterministic Turing machine M' that also accepts L.

Proving the theorem

What we will do is simulate the computation tree.

- Assume that the computation tree has only two choices at each step. (This is easily made possible.)
- If the nondeterministic machine M accepts w, then it accepts w after a particular sequence of choices, say 010010111001.
- We can simulate all sequences of choices, in the order $0, 1, 00, 01, 10, 11, 000, \ldots$.

We will do this with a multi-tape machine.

Construction for the proof

- On the first tape, we keep the string w.
- On the second, we keep the binary strings corresponding to the possible choices (the choice currently being simulated is marked).
- On the third tape, we simulate the action of the nondeterministic machine's tape as it processes through the string of choices specified on the second tape.
- If we make it to an accepting state in M, then we accept in M'!
- Otherwise, we construct and move to the next string in the ordering of all possible binary sequences on the second tape, and start over again.

Your text (Theorem 8.11) gives a very interesting proof that uses a queue, instead.

Does this work?

The new machine is deterministic.
Does it accept the same language?
Yes.

- If $w \in L$, then there is a binary string s of choices in the nondeterministic machine that leads M to an accept state.
- So when the second tape (eventually) has the string s on it, the new machine M' will accept.
- If the word w is not in L, then no set of choices s will lead to acceptance in M.
- But then no set of choices s will lead to acceptance in M', either.
- Note that this is an example where M' will run forever when processing a word not in L.
- So $L = L(M) = L(M')$, as we had hoped.

About this new machine

Note: This is a very slow simulation of M by M'!

- Prefixes of the computation are repeated for each binary choice string s.
  - If we are using the choice string 0001100, we will start in the configuration at the end of the choice string 000110.
- Also, the non-deterministic TM is running 2^n computations at a time when there have been n choices.
- The deterministic TM is only running 1!
- So the deterministic TM is exponentially slower than the nondeterministic TM.
- Is it possible to avoid this slowdown? The answer is not known, for languages in NP. If it is possible, then $P = NP$.

Many modifications do not make Turing machines more powerful

Being recursively enumerable (accepted by a Turing machine) is a very stable property.

Turing machines do not become more powerful with:

- Finite memory
- Multiple tapes
- Multiple tape heads
- Nondetermism

Turing machines represent our best understanding of what it means to do digital computation.

End of module 8

- There seem to be paradoxes that might make writing computer programs that read computer programs have limits to their power.
- Turing machines are a good representation of the power of computers.
- We can develop a programming discipline for them.
- Turing machines still have the same power, even when augmented with a variety of additions.

Next module: what is the limit of a computer?