Module 8

Turing machines

Simplifying computers down to automata?

CS 360: Introduction to the Theory of Computing
Fall 2023

Daniel G. Brown
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 can't solve? What sorts of problems?
- It turns out that the class of languages computers can't decide membership in 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 to solve.

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

Some programs are easy, some aren't.

```
def easy():
   return 1

def hard():
   i = 0
   while i >= 0:
     for j in range (i+1):
        for k in range (j+1):
           for n in range (3,k+1):
              if i**n== j**n+k**n:
                 return 1
     i = i+1
```

Does this 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's an interesting fact in number theory, or combinatorics, or whatever, and we want to know if it's always true, for all integers i, j, k.

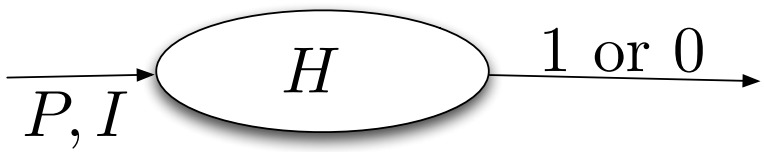
- Write a test for that property.
- Enumerate over all possible choices of i, j, k.
- If the test is false, return 1.
- If the function ever returns, then the property is not always true.

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

Another reason it's not easy to solve

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

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



Here's a new program, H_1, which uses H:

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

What's 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, H_1 doesn't.
- If P doesn't return 1 on input I, 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 used as the input. That's okay.

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

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

Turing machines

- We've been studying simple models of computation.
- We need a model of computation similar to our understanding of how people and computers work.

We need to be able to access memory more robustly than in a pushdown automata.

How do computers compute?

Simpler question: how do people compute?

- Split between "short-term" and "long-term" memory
- Small amount of info in "active" memory
- Potentially enormous amount in long-term memory (paper notebook, ...)

When we compute:

- Pull information from long-term memory into short term
- Analyze it and perform some computations on it.
- Write results to the long-term memory
- Until we've solved our problem.

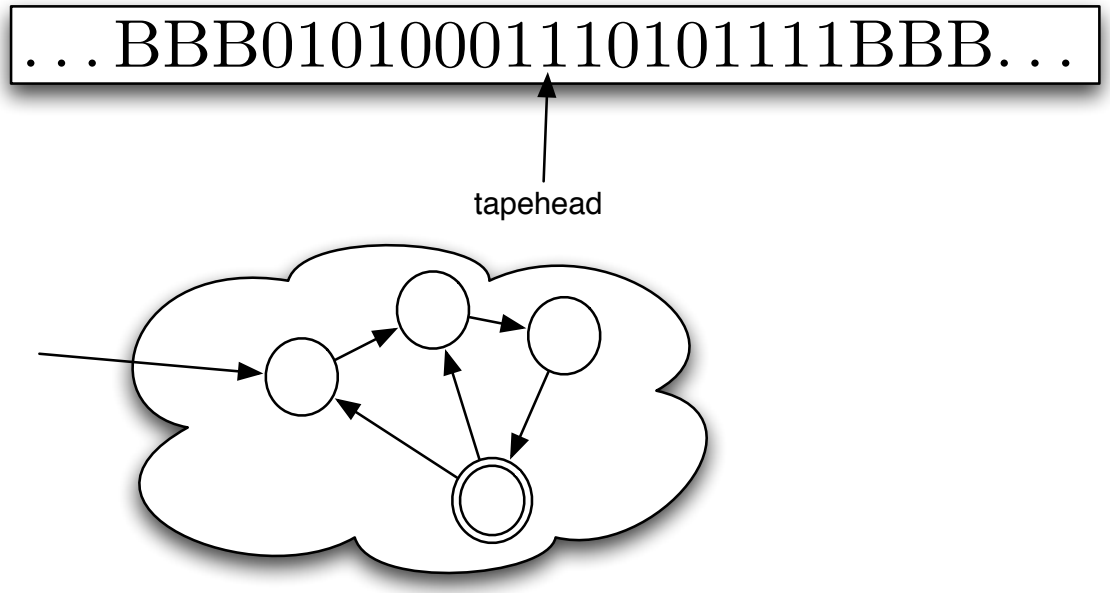
As an automaton model

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

- A finite automaton
  - 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
  - Doesn't access memory like a regular computer, but they're 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



- Long-term memory is only accessed at tape head: 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 tapehead 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 tapehead points to w_1, 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:

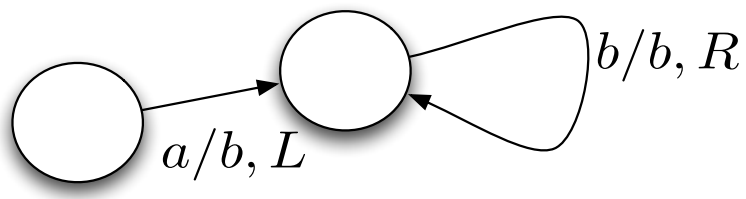
- Current state, q.
- The tape, $X_1 X_2 \ldots X_k$. Surrounding it is an infinite number of blanks in both directions.
- The current position of the tapehead. We'll underline the current position; your book does something much grosser.
- If we want the tapehead pointing to the first symbol, w_1 of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

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

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



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

Special transitions in the Turing machine

Special cases:

- Tapehead 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 $\delta(q, a) = (p, b, L)$.
- Tapehead at rightmost nonblank character: if $\delta(q, a) = (p, b, R)$, then $(q, x\underline{a}) \vdash (p, xb\underline{B})$.
- 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 multistep computations: in the Turing machine M, $(q, x\underline{a}y) \overset{*}{\underset{M}{\vdash}} (p, v\underline{b}x)$ if we can move between the configurations in some finite number of steps.

- Only show machine M when necessary.

The Turing machine M accepts w exactly if $(q_0, \underline{w_1} w_2 \ldots w_n) \overset{*}{\underset{M}{\vdash}} (p, x\underline{a}y)$ for any 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.

- The language of the machine M, $L(M)$, is all words M accepts.
- If L is the language accepted by a Turing machine, then L is recursively enumerable, or r.e.

Rejection in the TM, running forever

How does a TM reject an input?

- If it winds up in a state q, pointing at a, and $\delta(q, a)$ is not defined, it crashes and rejects the word.
- "Halting and rejecting"

A Turing machine can also not accept an input by running forever on it.

- Possible with a PDA, too, or even 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.
- A L is the language accepted by a Turing machine that halts on every input, 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 using two fingers.

How to do it with a TM?

- Look at the leftmost character in the word we haven't yet matched.
- Match it with the rightmost character.
- Remove both of them.
- Keep doing this until only 0 or 1 letter left.

More specific

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

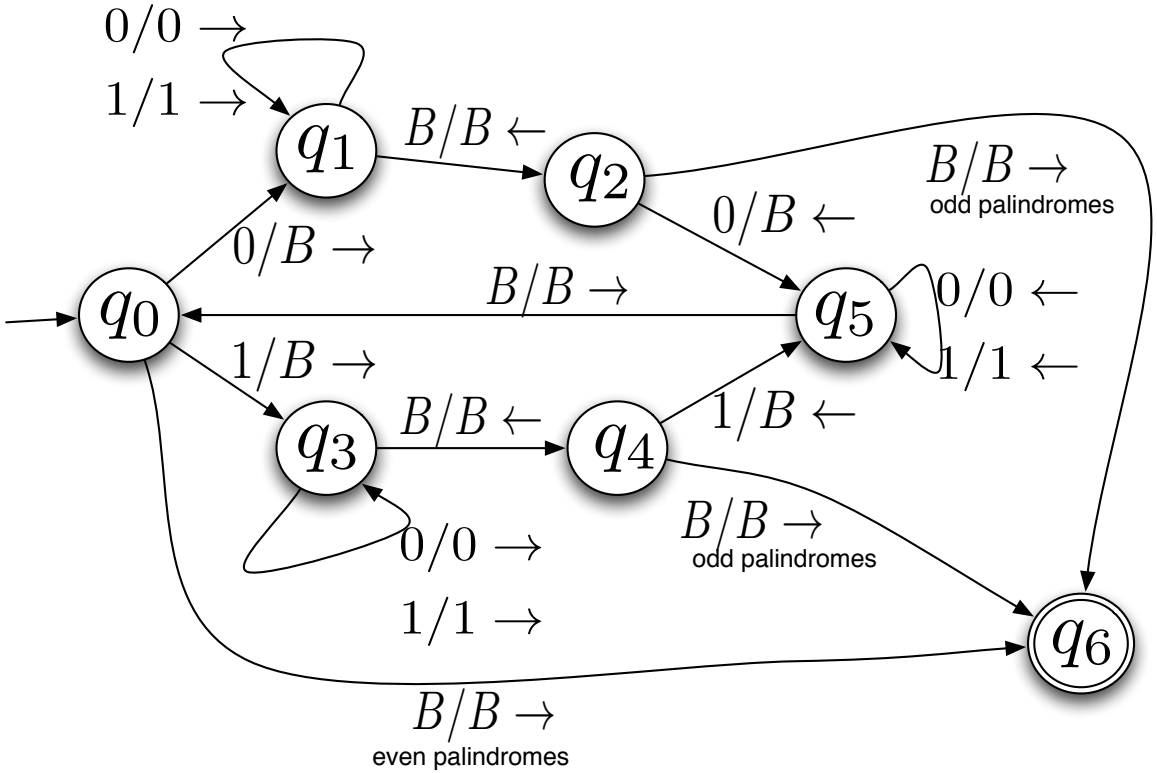
Can we encode this as a TM?

How to encode it as a Turing machine?

- Erase the first letter and remember it.
  - Two parallel tracks in the finite control: maintain whatever that letter is.
  - Don't forget to delete it.
- Find last letter and compare.
  - Jump forward as far as possible until we reach a B.
  - Then jump back one letter.
  - Compare the two letters. If they differ, crash!
  - Otherwise, delete the new letter, and go back to the beginning of the word.
- If all that's left is B, we're 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; I prefer that.)



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

An accepting computation

Consider the palindrome $w = 010$.

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

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?

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

Not going to draw the machine (in Section 8.3.2). 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 nothing left.

Programming Turing machines

- Store a finite amount of information in the state (example: symbol to match)
- Tag letters of the word with more bits of information, expanding the alphabet.
- Produce subroutines and other programming ideas.

Important distinctions about languages of TMs

Seen once before, but important:

- 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.

The machine for palindromes decides that language, since it crashes on all non-palindromes.
If it ran forever on some non-palindromes, it would only accept its language.

Recursive versus recursively enumerable

This distinction gives rise to two different kinds of languages:

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

There are languages that are not recursive, but are recursively enumerable.

- (And there are languages that are neither...)
- If a language is recursive, it's recursively enumerable.

If machine M accepts L, but doesn't halt on all inputs, that doesn't mean L is not recursive.

- You may just have the wrong TM; a different one might accept L and halt on all inputs.

Computing a function with Turing machines

We often want to compute the value of a function, not just test membership in a language.
These seem like qualitatively 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 $a + b = c$.

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

Computing a function, continued

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

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

What if M computes $f(x)$ for all possible values of x?

A computable function

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

- $(q_0, \underline{x}) \overset{*}{\underset{M}{\vdash}} (q_a, \underline{y})$, where $y = f(x)$ and q_a is any accept state.
- (make allowances when x or y is ε).

M computes function f.

- M accepts every word x: the important thing is the tape contents.
- f need not be total: there could be elements of Σ^* for which f is not defined. On those, M can either crash or run forever.
- The function can have multiple arguments or be multiple-valued
  If $f : \Sigma^* \times \Sigma^* \to \Gamma^*$, then we'll have the two arguments to f be next to each other on the tape when M starts running, with a blank character B between them.

Computable functions

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

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

Numeric functions can be computable, too. We'll use unary:

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

A simple example

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

- We encode the integer i as 1^i.
- $x + y$ is the concatenation of the two arguments 1^x and $1^y, 1^{x+y}$.
- $+$ is computable: we can certainly concatenate two strings with a Turing machine.

Characteristic functions

- The characteristic function of language L is the function $\chi_L : \Sigma^* \to \{0, 1\}$:
  - $\chi_L(x) = 1$ if $x \in L$.
  - $\chi_L(x) = 0$ if $x \notin L$.
- Answers question, "is x in language L?"
- If the Turing machine M computes χ_L:
  - On any input $x \in \Sigma^*$, it always accepts.
  - When it halts, either 1 or 0 is left on the input, with the tapehead 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 M:

On input x:

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

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

Using subroutines in general

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).

We don't use subroutines in Turing machines to shorten the program code.

- We don't care about program length.
- We care much more 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 tapehead:

- $(q_0, y\underline{z}) \overset{*}{\vdash} (q_a, ya\underline{z})$, where q_a is an accept state in the subroutine machine.
- Restriction: z doesn't have the blank character in it, so that we know when we've 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 with that character.

Another example: deleting a character

Delete the character the tapehead is pointing at:

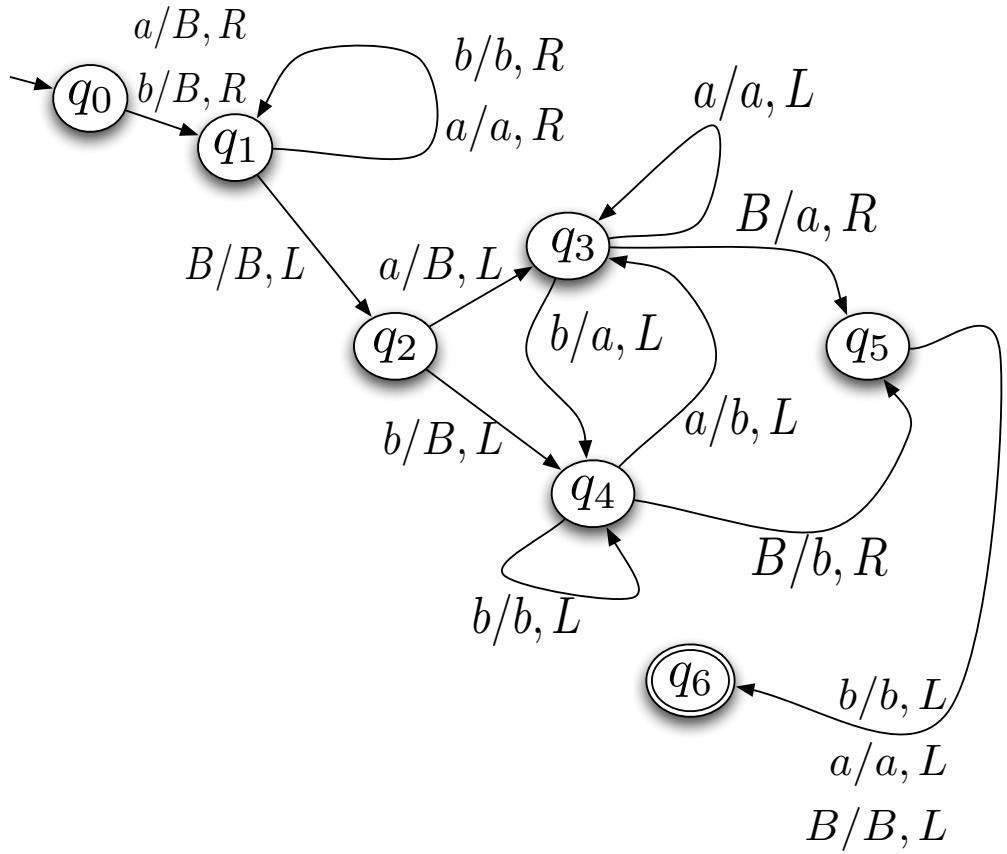
- $(q_0, y\underline{a}z) \overset{*}{\vdash} (q_a, y\underline{z})$, where q_a is again an accept state in the subroutine machine.
- Restrict z to not have the blank character.

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

Deletion machine

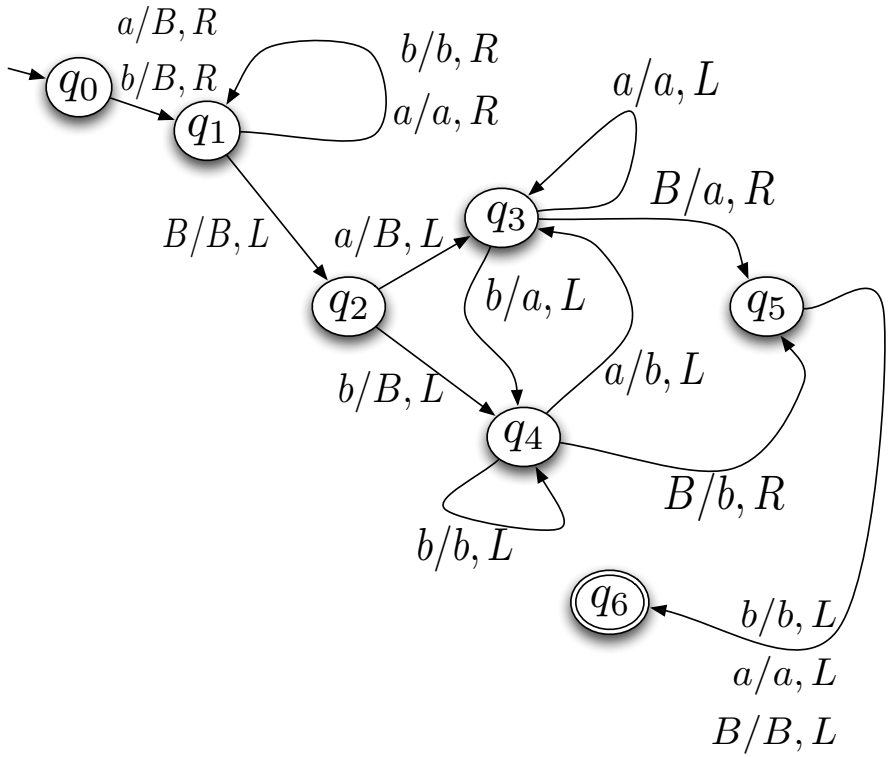
One simple approach:

- 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.



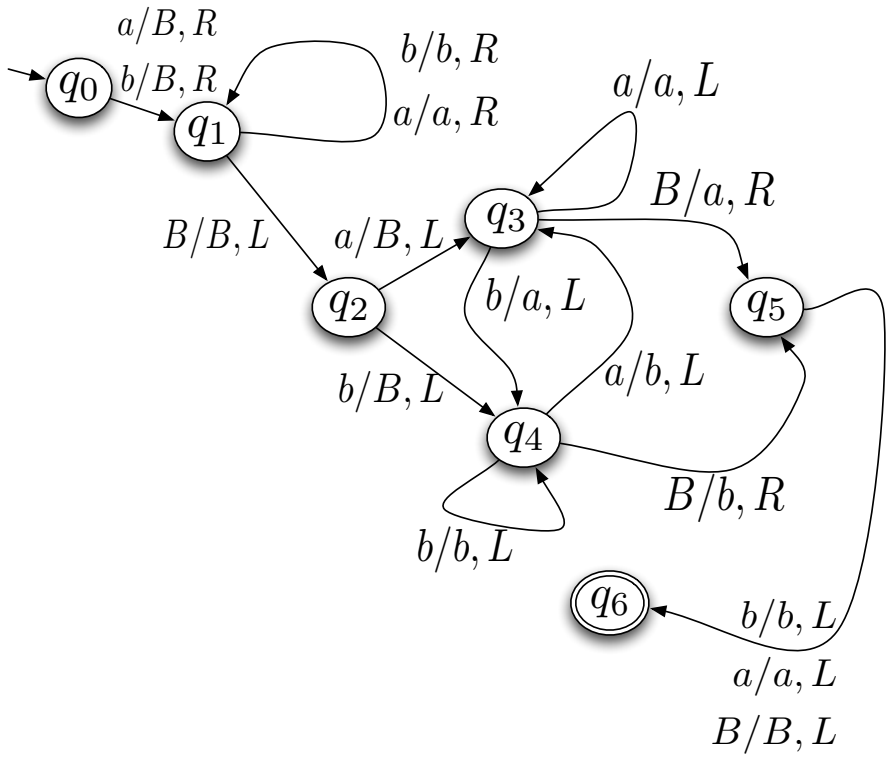
How's that work?

- In state q_0, we delete the current tapehead character and move right.
- In state q_1, we move to the right until we've 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's that work? (cont'd)

- Push the whole way to the beginning of the string, and don't copy in the last character.
- In state q_5, move the tapehead back to the right position.
- Accept in q_6.



Storage in the state

Another trick, implicitly just used:

- Imagine finite "working memory".
- Augment the state by storing k bits of information: multiply state space by 2^k, and store memory as information in the state.
- If the memory is M, and its value is over a finite set of choices, $\mathcal{M}$, then δ, the transition function becomes $\delta : Q \times \Gamma \times \mathcal{M} \to Q \times \Gamma \times \mathcal{M} \times \{L, R\}$.
- Old and new value of the memory join transition function.

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

Variations on Turing machines

Lots of ways to augment Turing machines have no effect on their power.

- Finite memory (seen)
- Multiple tapes
- Multiple tapeheads
- 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, k infinite tapes, each with its own tapehead.

- The machine's instantaneous description is characterized by what is stored on each of the k tapes, the position of all k tapeheads, 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 the input.

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

Proof of equivalence

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

- 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.
- The tape alphabet for M' is $\Gamma \cup \Gamma \times \{B, *\} \times \Gamma \times \{B, *\}$.
- That is, each position of the one tape for M' is either a letter from Γ, or two letters from Γ, where each is either tagged with a $*$ or not.
- Tags identify current tapehead positions in M, as we simulate it.

First, set things up.

- We start by replacing the first character a from Γ on the input tape for M' with $(a, *, B, *)$ (to indicate that the tapeheads start out at the left end of the input).
- Then, we replace every other character a on the input tape with (a, B, B, B).

And we move the tapehead back to the beginning of the tape.

Using the one-tape TM to simulate a step in the multitape TM

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

- We first find the two tape heads, searching from the left of the tape, searching for the $*$ character in the second and fourth positions of the four-tuple that is a letter in the alphabet.
- Then, we store the two characters in a finite memory.
- Now, we know the state in the finite automaton, and the two symbols pointed to on the tapeheads.
- Then, we identify the two new values we're supposed to place at the site of the tapeheads, and we search left to right to find them, again.
- We replace the characters with the new values.
- And we mark the proper positions on the tape to be the new simulated tapeheads.
- Lastly, rewind the tape.

Why use multi-tape TMs?

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

- We can write a one-tape TM to test membership in this.
- But easier might be a multi-tape TM:
  - Scan left-to-right to ensure only one ! character.
  - Copy everything after the ! to the second tape.
  - Move to the left of both tapes.
  - Move left-to-right, matching characters in both tapes to each other.
  - If there's a mismatch, crash. Otherwise, if we get to the end of both copies, accept.

A simpler extension: multiple tapeheads

We could have multiple tape heads on one tape.

- The description of the machine consists of the state, the string on the memory, 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's pointed to by all k heads:
  - Go to a new state.
  - Put new values at each of the k heads.
  - Move each of them left or right.
- One problem: concurrency. What happens when multiple heads point to same point?
- Lots of possible solutions: 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 form $\Gamma \times \{B, *\} \times \{B, *\}$, as in the proof for the equivalence of multitape machines and single tape machines. The second and third parts of the alphabet symbols indicate the position of the two tape heads.
- Scan to find the values at both tapeheads.
- Keep track of what's pointed to by the first head when searching for 2nd head
- Keep track of what's to be put at the 2nd head's position.
- Then copy in the new values and "move" the tapeheads.

Two (or more) tapeheads are no better than 1.

- Can easily decide membership in the same language of copied words with 2-head machines with one tape.

Other memory alterations

- 2-dimensional memory (tapehead moves up, down, left or right).
- Or higher dimensions.
- Or a tape where we keep an address in the main tape that we are interested in: we can then do something like random-access memory.

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
- This essentially causes us to have an exponential number of simultaneously running Turing machines.

More powerful than a deterministic Turing machine?

- No.

Details for nondeterminism

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

- Must be a finite number of enries in $\delta(q, a)$.
- We say $(q, y\underline{x}) \vdash (p, w\underline{z})$ if one transition in $\delta(q, x_1)$ 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 only accept languages, rather than deciding them. They don't compute functions. What if the valid computations disagreed about the value of $f(x)$?

Example nondeterministic Turing machine

$L = \{ww \mid w \in \{a, b\}^*\}$.
Sketch of a two-tapehead nondeterministic TM for L:

- Start with the possible word in L on the tape, and both tapeheads 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 tapehead at that point.
- Match symbols at both tapeheads, until they either mismatch or both tapeheads reach the end of their copy of w.

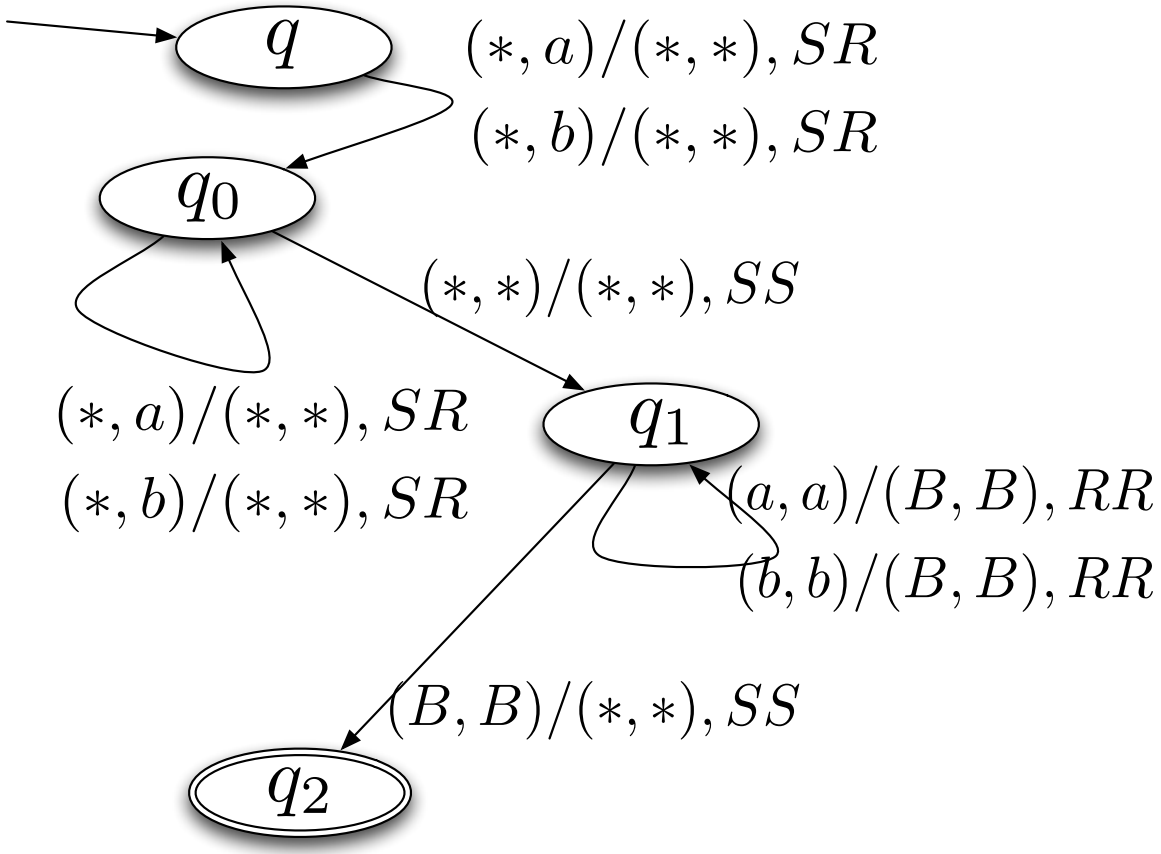
Nondeterminism: guessing the end of the first copy of w.

Notational change

Change our notation a bit:

- In the 2-head TM, if the first tapehead points to a and the second tapehead points to b, and we change the first tapehead's symbol to c, and the second's to d, while moving the first tapehead right and the second tapehead left, we'll notate that by $(a, b)/(c, d), RL$.
- We add the possibility that a transition doesn't require a tapehead to move, so instead of $\{L, R\}$, our transitions will be from $\{L, R, S\}$ (with S for "stationary").
- If both tapeheads are at the same site, the new symbol assigned by the second tapehead takes precedence.
- We don't have to care about what's at both tapeheads: to ignore what's at a tapehead, put a $*$ in the pair before the slash, as in $(a, *)/(c, d), LS$.
- If we don't change the symbol pointed to by a tapehead, we can indicate that with a $*$ after the slash, as in $(a, *)/(*, *), LS$.

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, we accept.

The crucial property: computation branches

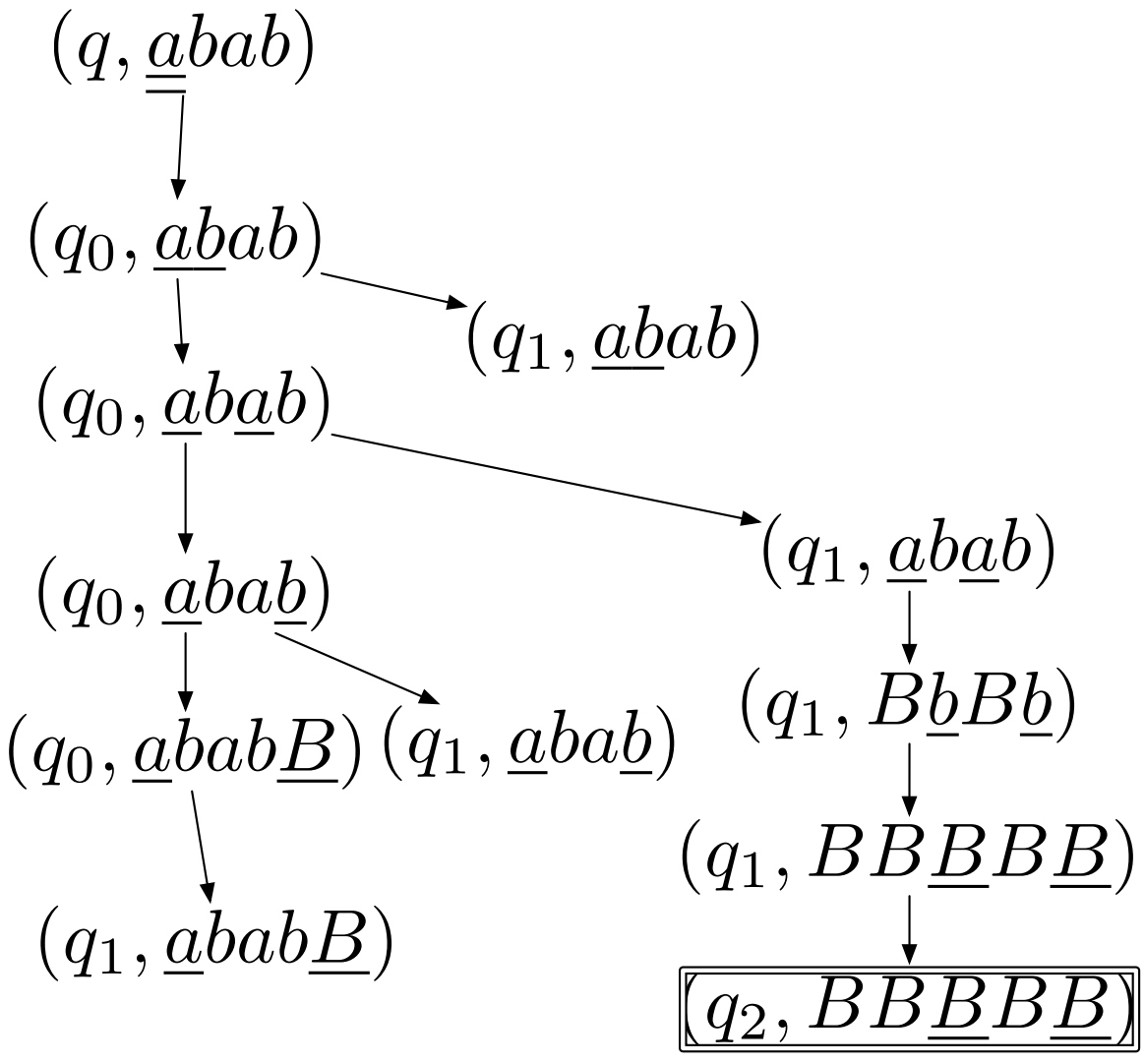
At each step in state q_0, the machine makes a choice:

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

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's the 001 branch of the tree, the 00001 branch, and so on. After k steps, might be an exponential number of operational branches.
Include start state q: ensures we move the second tapehead 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's 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).
- We care only about what they can do, not how fast.

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'll do it 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 string corresponding to the choices.
- On the third tape, we simulate the nondeterministic machine through the string of choices on the second tape.
- If we make it to an accepting state, we accept!
- Otherwise, we move to the next string in the ordering of all binary sequences, and start over again.

Your book gives (Theorem 8.11) 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's a binary string s of choices in the nondeterministic machine that leads M to an accept state.
- But when the second tape has the string s on it, the new machine M' will accept.
- If the word w isn't in L, no set of choices will lead to acceptance in M. But they won't in M', either.
- So $L = L(M) = L(M')$, as we hoped.

About this new machine

Very slow simulation!

- Prefixes of the computation are repeated for each binary choice string s.
  - If we're using the choice string 0001100, we'll 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 it's exponentially slower.
- Is it possible to avoid this slowdown? Not known, for languages in NP. If so, then $P = NP$.

Many things don't make Turing machines more powerful

Being recursively enumerable (accepted by a Turing machine) is a very stable property.
Turing machines don't become more powerful with:

- Nondetermism
- Multiple tapes
- Multiple tapeheads
- Finite memory

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?