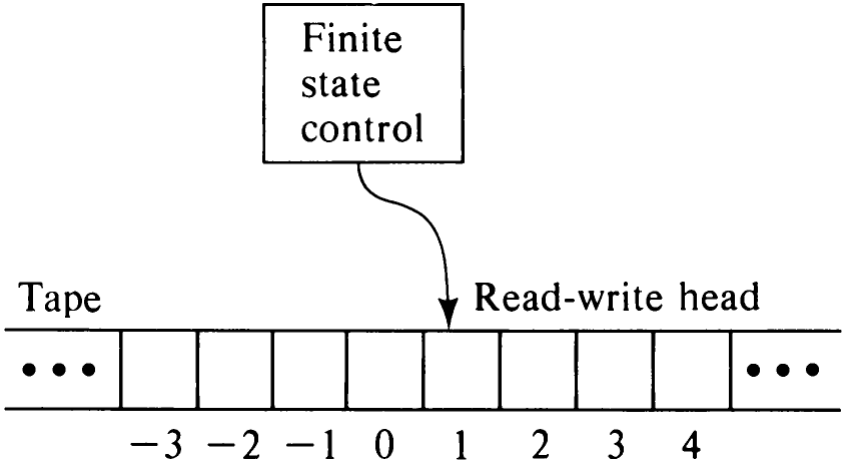
Turing Machines



Recall

- We have investigated the capabilities of simple models of computation – all use “basic” machines with very “little” complexity
  - How to recognize strings of increasing complexity
  - How to match regular expressions
  - How to parse programming languages
- We have also seen these machines – FA, PDA – come with serious limitations that undermine their usefulness as realistic models of computation

Questions

- How much more powerful do our toy systems need to get before they can escape these limitations and do everything that a normal computer can do?
- How much more complexity is required to model the behavior of RAM, or a hard drive, or a proper output mechanism?
- What does it take to design a machine that can actually *run programs* instead of always executing a single *hardcoded* task?

Deterministic Turing Machines

- We were able to increase the computational power of a finite automaton by giving it a stack to use as external memory
- The real advantage of a stack is that it can grow dynamically to accommodate any amount of information, allowing a pushdown automaton to handle problems where an arbitrary amount of data needs to be stored
- What is the limitation?

Deterministic Turing Machines

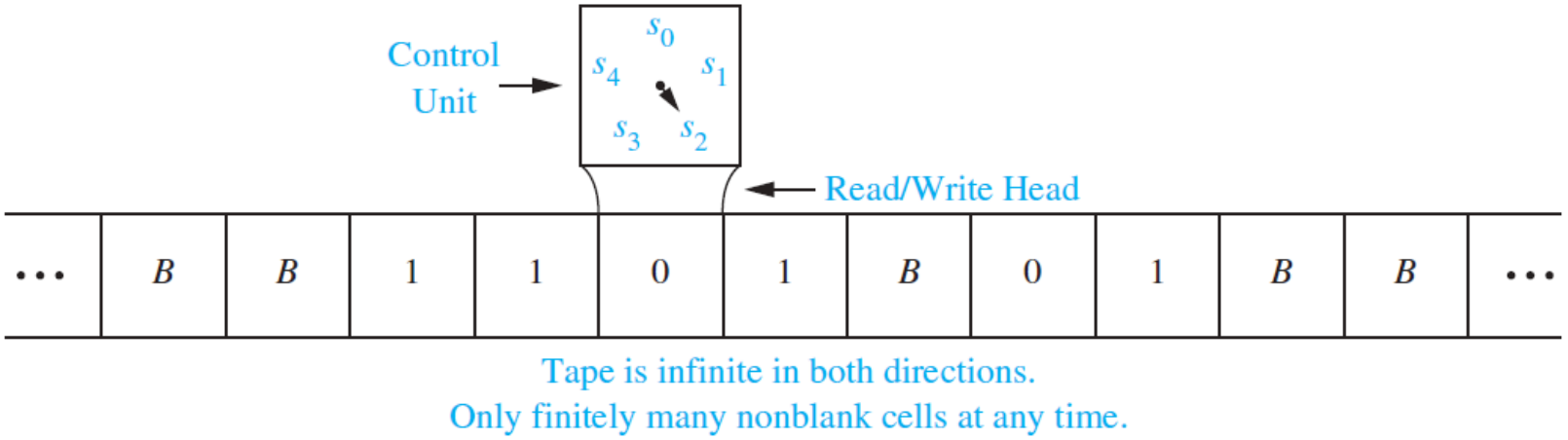
- A pushdown automaton can only access a single *fixed* location in its external storage—the top of the stack—but that seems too restrictive for a Turing machine
- The whole point of providing a tape is to allow arbitrary amounts of data to be stored anywhere on it and read off *again* in any order
- How do we design a machine that can interact with the entire length of its tape?

Deterministic Turing Machines

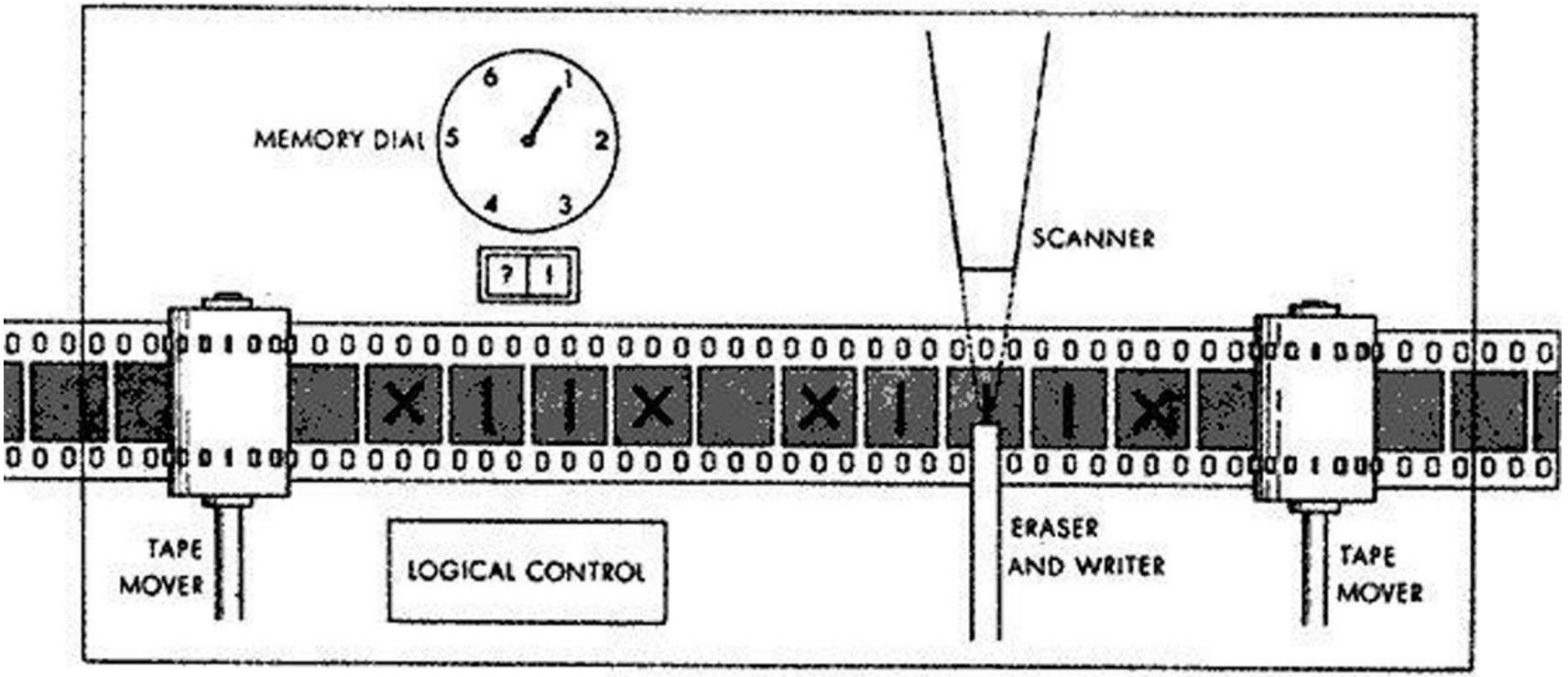
- A finite state machine with access to an infinitely long *tape* is called a Turing Machine (TM)
- That name usually refers to a machine with deterministic rules, but we can also call it a deterministic Turing machine (DTM) to be completely unambiguous

Definition

- A Turing Machine $T = (S, I, f, s_0)$ consists of a finite set S of states, an alphabet I containing the blank symbol B, a partial function f from $S \times I$ to $S \times I \times \{R, L\}$, and a starting state s_0



How Is It Engineered?



Deterministic Turing Machines

- A conventional Turing machine uses a simpler arrangement: a *tape head* that points at a specific position on the tape and can only read or write the character at that position
- The tape head can move left or right by a single square after each step of computation, which means that a Turing machine must move its head laboriously back and forth over the tape in order to reach distant locations

Deterministic Turing Machines

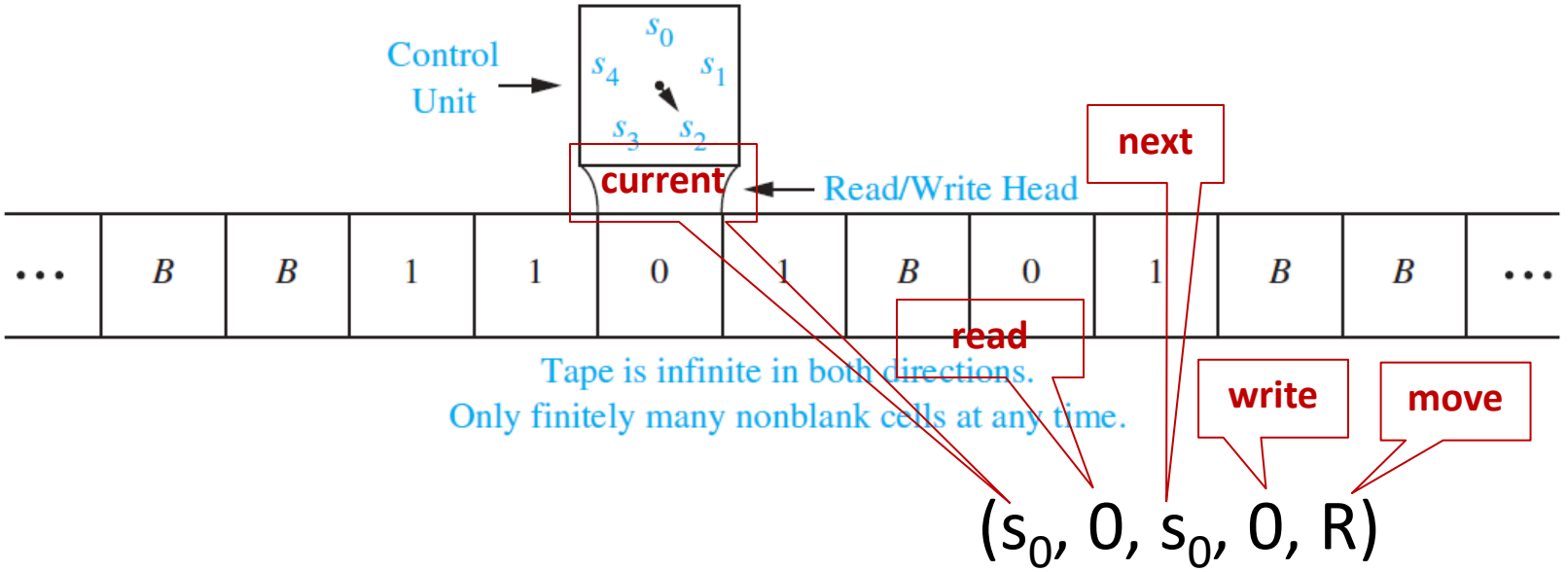
- Having access to a tape allows us to solve new kinds of problems beyond simply accepting or rejecting strings
- There are several “simple operations” we might want a Turing machine to perform in each step of computation:
  - Read the character at the tape head’s current position
  - Write a new character at that position, move the head left or right, or change state

Deterministic Turing Machines

- This unified rule format has five parts:
  - The current state of the machine
  - The character that must appear at the tape head's current position
  - The next state of the machine
  - The character to write at the tape head's current position
  - The direction (left or right) in which to move the head after writing to the tape

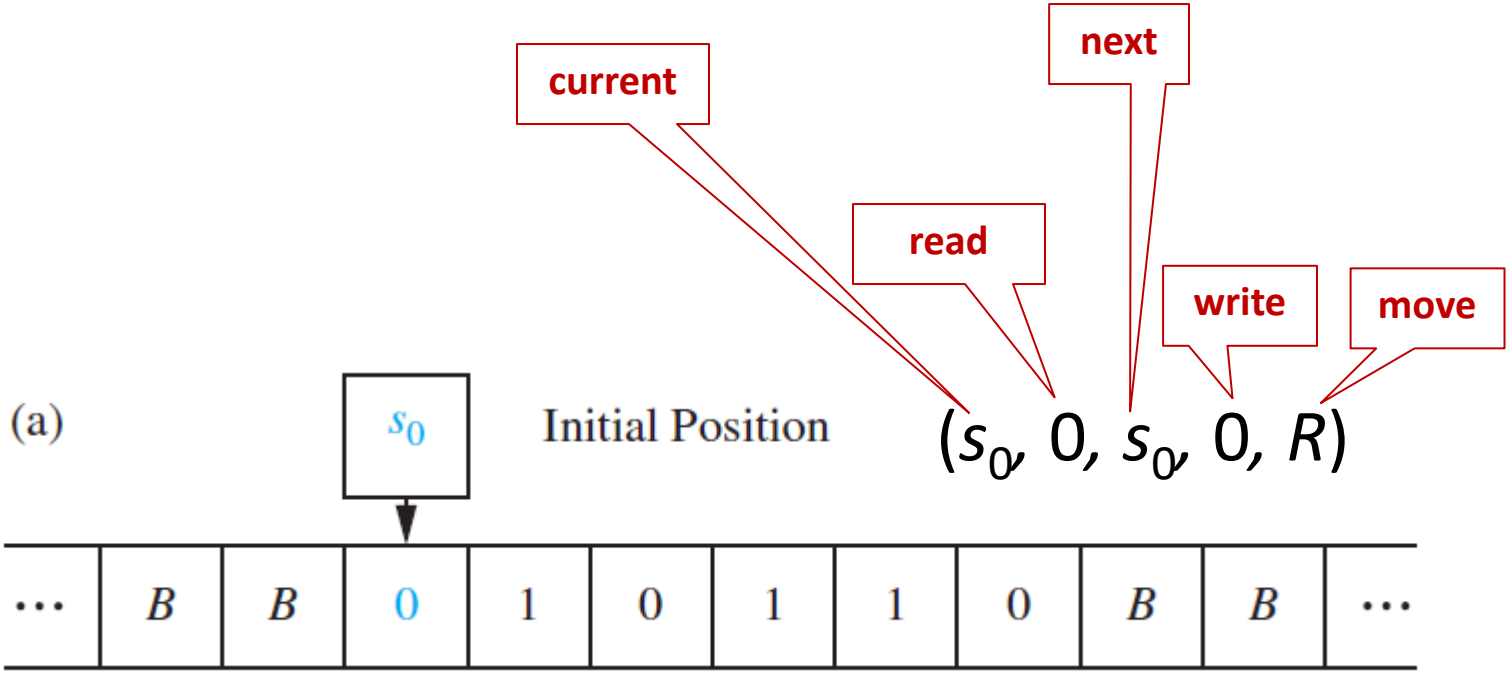
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



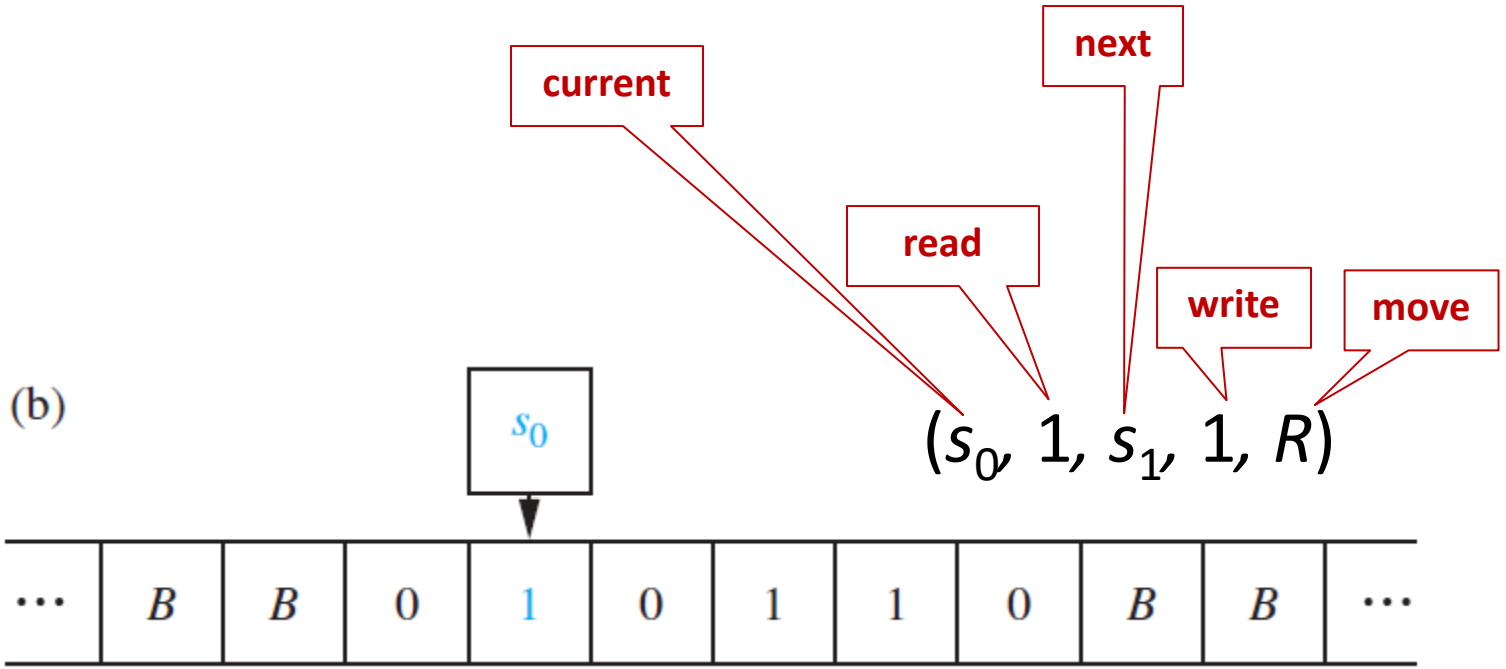
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



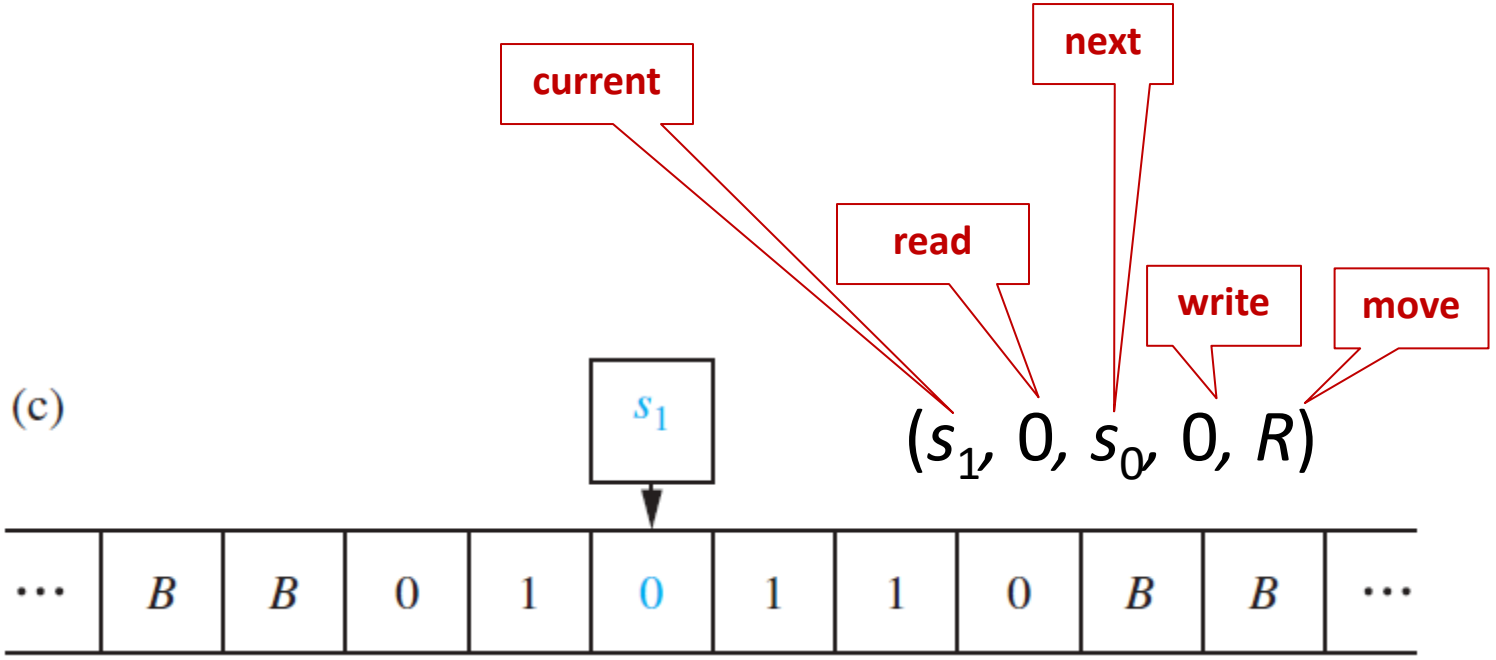
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



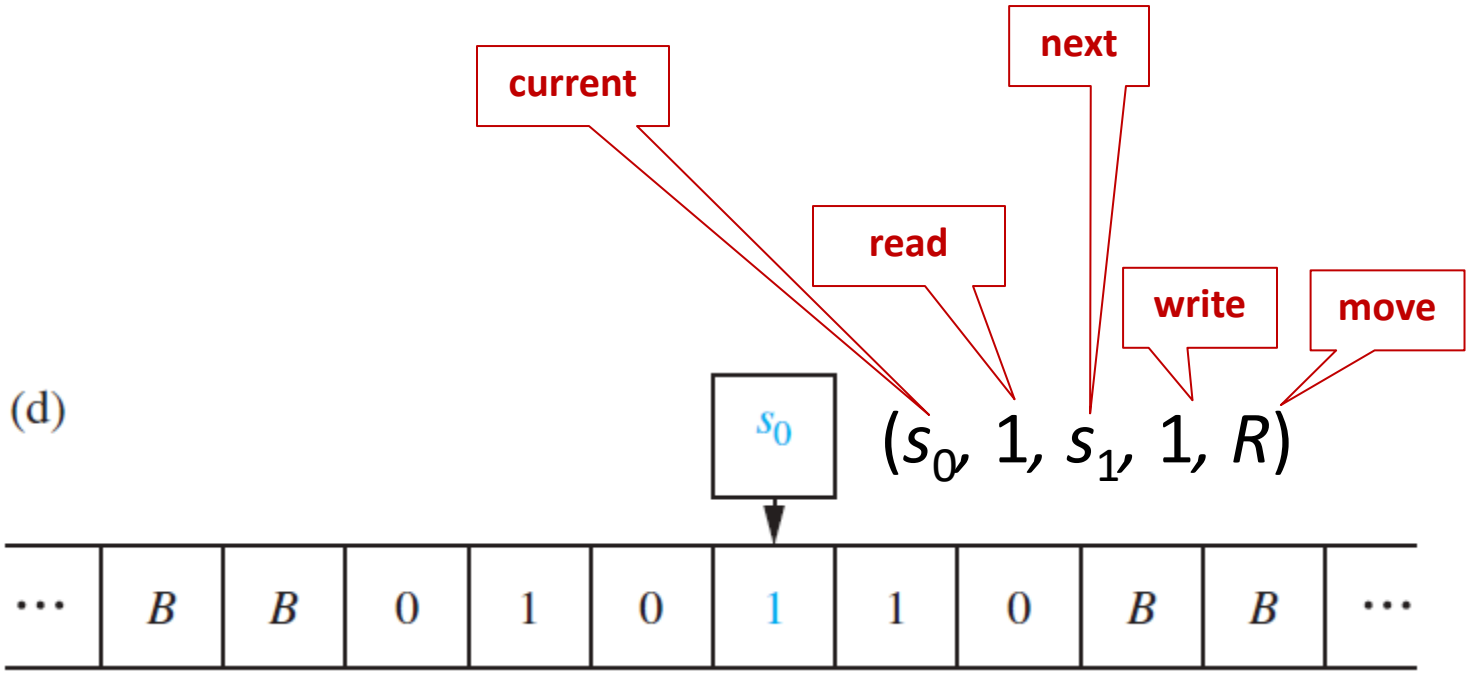
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



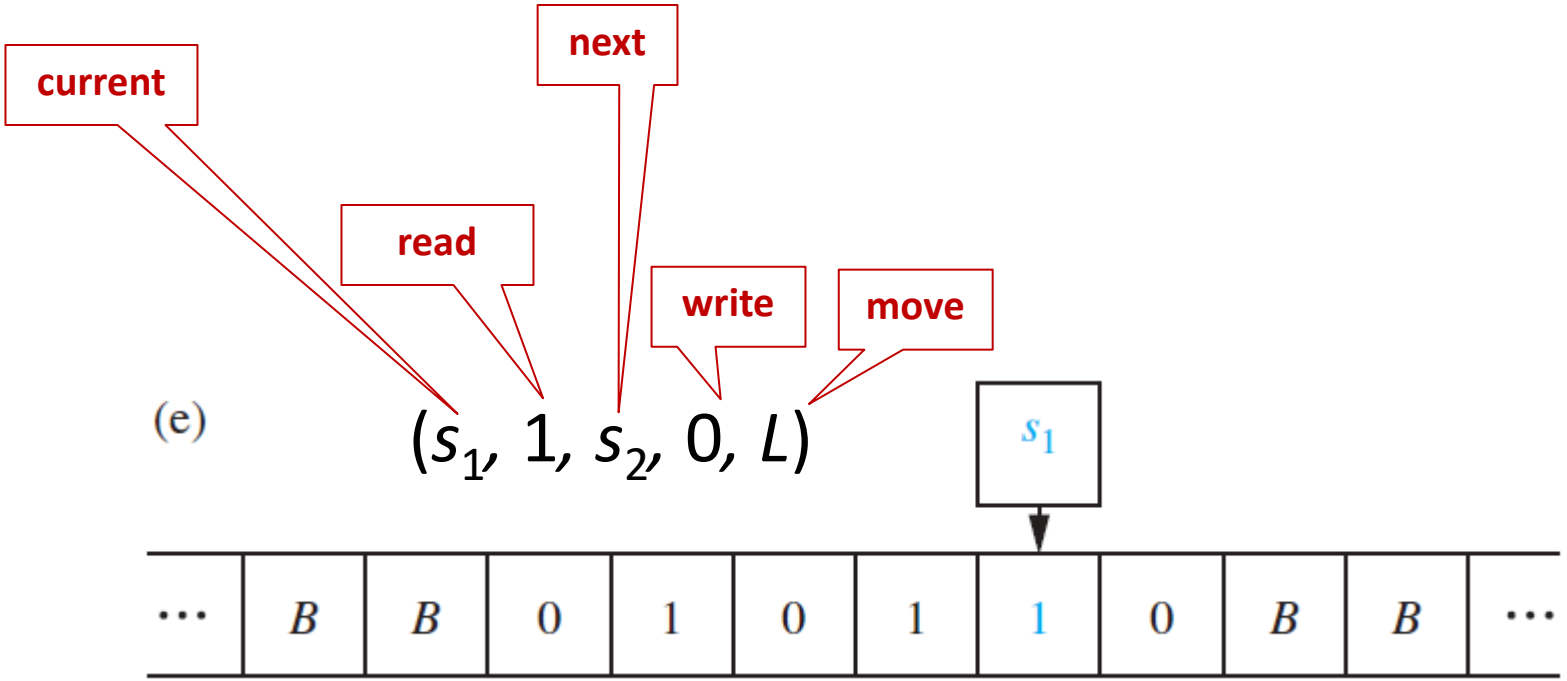
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



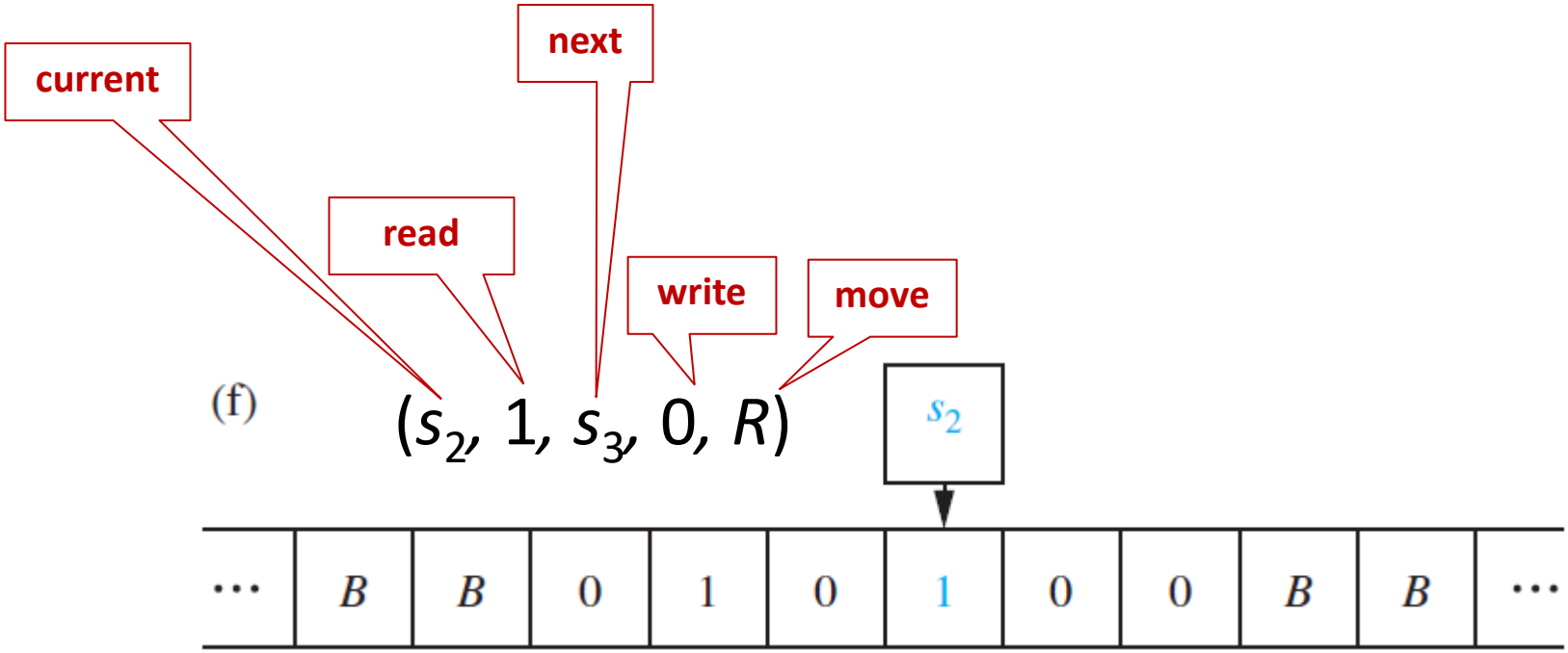
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



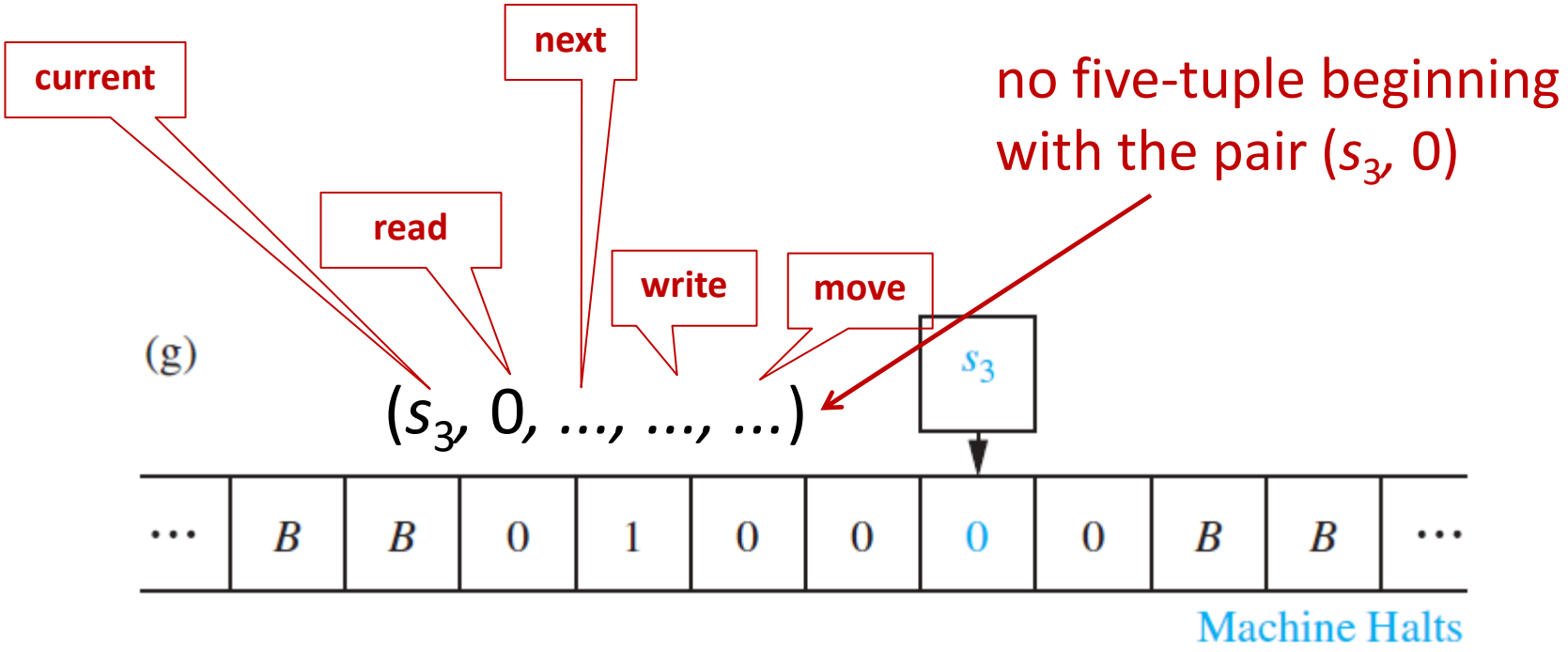
How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



How TMs Work?

- What is the final tape when the Turing machine defined by the seven five-tuples (s_0, 0, s_0, 0, R), (s_0, 1, s_1, 1, R), (s_0, B, s_3, B, R), (s_1, 0, s_0, 0, R), (s_1, 1, s_2, 0, L), (s_1, B, s_3, B, R), and (s_2, 1, s_3, 0, R) is run on the tape shown in below figure?



Determinism

- A Turing machine's next action is chosen according to its current state and the character currently underneath its tape head, so a deterministic machine can only have one rule for each combination of state and character—the "no contradictions" rule—in order to prevent any ambiguity over what its next action will be
- For simplicity, we'll relax the "no omissions" rule, just as we did for DPDAs, and assume there's an implicit stuck state that the machine can go into when no rule applies, instead of insisting that it must have a rule for every possible situation

Nondeterministic Turing Machines

- Nondeterminism makes no difference to what a finite automaton is capable of
- A nondeterministic pushdown automaton can do more than a deterministic one
- How about Turing machines? Does adding nondeterminism make a Turing machine more powerful?
- No

Nondeterministic Turing Machines

- A nondeterministic Turing machine can’t do any more than a deterministic one
- Pushdown automata are *exception* here

Nondeterministic Turing Machines

- Just as with finite automata, a deterministic Turing machine can simulate a nondeterministic one—the simulation works by using the tape to store a queue of suitably encoded Turing machine configurations, each one containing a possible current state and tape of the simulated machine
- We can’t make a Turing machine any more powerful just by adding nondeterminism

Maximum Power

- Deterministic Turing machines represent a dramatic tipping point from limited computing machines to full-powered ones
- Any attempt to upgrade the specification of Turing machines to make them more powerful is doomed to failure, because they're already capable of *simulating* any potential enhancement
- We've already seen why this is true for nondeterminism (i.e., adding nondeterminism does not make it more powerful)

Internal Storage

- Designing a rulebook for a Turing machine can be frustrating because of the lack of arbitrary internal storage
- It would be more convenient if a Turing machine had some temporary internal storage
  - Call it “RAM”, “registers”, “local variables”, or whatever
- It could save the character from the current tape square and refer back to it later, even after the head has moved to a different part of the tape entirely
- This extra flexibility feels like it could give a Turing machine the ability to perform new kinds of tasks?

Internal Storage

- As with nondeterminism, adding extra internal storage to a Turing machine certainly would make certain tasks easier to perform, but it wouldn’t enable the machine to do anything it can’t already do
- The desire to store intermediate results inside the machine instead of on the tape is relatively easy to dismiss, because the tape works just fine for storing that kind of information, even if it takes a while for the head to moveback and forth to access it

Internal Storage

- A Turing machine already has perfectly good internal storage: its current state
- There is no upper limit to the number of states available to a Turing machine – for any particular set of rules, that number must be finite and decided in advance
- If necessary, we can design a machine with a hundred states, or a thousand, or a billion, and use its current state to retain arbitrary amounts of information from one step to the next

Internal Storage

- This inevitably means duplicating rules to accommodate multiple states whose meanings are identical except for the information they are “remembering”
- Instead of having a single state that means scan right looking for a blank square, a machine can have one state for scan right looking for a blank square (remembering that I read an a earlier), another for scan right looking for a blank square (remembering that I read a b earlier), and so on for all possible characters—although the number of characters is finite too, so this duplication always has a limit

Subroutines

- Designing a rulebook would be easier if there was a way of calling a *subroutine*: if some part of the machine could store all the rules for, say, incrementing a number, then our rulebook could just say “now increment a number” instead of having to manually string together the instructions to make that happen
- And again, perhaps that extra flexibility would allow us to design machines with new capabilities?

Subroutines

- This is another feature that is really just about convenience, not overall power
- Several small Turing machines can be connected together to make a larger one, with each small machine effectively acting as a subroutine
- When the smaller machine only needs to be “called” from a single state of the larger one
  - This is easy to arrange

Subroutines

- What if we want to call a particular subroutine from more than one place in the overall machine
- A Turing machine has no way to store a “return address” to let the subroutine know which state to move back into once it has finished, so, it seems we can’t support this more general kind of code reuse
- Duplication: rather than incorporating a single copy of the smaller machine’s states and rules, we can build in many copies, one for each place where it needs to be used in the larger machine

Multiple Tapes

- The power of a machine can sometimes be increased by expanding its external storage
  - Let PDA have access to a second stack
- Any finite state machine with access to an infinite tape is effectively a Turing machine, so just adding an extra stack makes a pushdown automaton significantly more powerful

Multiple Tapes

- A Turing machine might be made more powerful by adding one or more extra tapes, each with its own independent tape head, but again – that's not the case
- A single Turing machine tape has enough space to store the contents of any number of tapes by interleaving them: three tapes containing *abc*, *def*, and *ghi* can be stored together as *adgbehcfi*

Multiple Tapes

- If we leave a *blank* square alongside each interleaved character, the machine has space to write markers indicating where all of the simulated tape heads are located: by using *X* characters to mark the current position of each head, we can represent both the *contents* and the *head positions* of the tapes *ab(c)*, *(d)ef*, and *g(h)i* with the single tape *a_dXg_b_e_hXcXf_i_*

Multiple Tapes

- Programming a Turing machine to use multiple simulated tapes is complicated
- Tedious details of reading, writing, and moving the heads of the tapes can be wrapped up in dedicated states and rules (“subroutines”) – the main logic of the machine does not become too complicated
- A single-tape Turing machine is ultimately capable of performing any task that a multitape machine can, so adding extra tapes to a Turing machine doesn’t give it any new abilities

Multidimensional Tape

- Instead of using a linear tape, we could provide an infinite two-dimensional grid of squares and allow the tape head to move up and down as well as left and right
- A grid can be simulated with one-dimensional tape
  - Use *two* one-dimensional tapes: a primary tape for actually storing data, and a secondary tape to use as scratch space
  - Each row of the simulated grid is stored on the primary tape, top row first, with a special character marking the end of each row

Multidimensional Tape

- The primary tape head is positioned at the current character
- To move left and right on the simulated grid
  - Simply moves the head left and right: if the head hits an end-of-row *marker*, a subroutine is used to shuffle everything along the tape to make the grid one space wider
- To move up or down on the simulated grid, the tape head must move *a complete row* to the left or right respectively
  - First move the tape head to the beginning or end of the current row, using the secondary tape to record the distance travelled, and then moving the head to the same offset in the previous or next row
  - If the head moves off the top or bottom of the simulated grid, a subroutine can be used to allocate a new empty row for the head to move into

Multidimensional Tape

- Wait a minute – this simulation does require a machine with two tapes?
- We already know how to simulate that too
  - Recalled multiple tapes
- We end up with a simulated grid stored on two simulated tapes that are themselves stored on a single native tape

General-Purpose Machines

- The rules of the machines we have seen so far are hardcoded – this isn't how most real-world computers work
- Rather than being specialized for a particular job, modern digital computers are designed to be *general purpose* and can be programmed to perform many different tasks
- Can any of our simple machines do that?

General-Purpose Machines

- A Turing machine is powerful enough to read the description of a simple machine from its tape—a deterministic finite automaton, say—and then run a simulation of that machine to find out what it does
- There's an important difference between a Turing machine that simulates a *particular* DFA and one that can simulate *any* DFA

General-Purpose Machines

- A Turing machine that performs a *general* DFA simulation
- This machine can read a DFA design from the tape—rules, start state, and accept states—and walk through each step of that DFA's execution, using another part of the tape to keep track of the simulated machine's current state and remaining input
- The same applies to NFAs, DPDAs, NPDAs, each of which can be turned into a Turing machine capable of simulating any automaton of that type

General-Purpose Machines

- More crucially, it also works for our simulation of Turing machines themselves
- We are able to design a machine that can simulate any other DTM by reading its rules, accept states, and initial configuration from the tape and stepping through its execution, essentially acting as a Turing machine rulebook interpreter
- A machine that does this is called a *universal Turing machine* (UTM)

Encoding

- How to represent an entire Turing machine as a sequence of characters on a tape? – we need a practical way of storing all this information in a way that the UTM can work with
  - A UTM has to read in the rules, accept states, and starting configuration of an arbitrary Turing machine, then repeatedly update the simulated machine's current configuration as the simulation progresses

Encoding

- Every Turing machine has a finite number of states and a finite number of different characters it can store on its tape, with both of these numbers being fixed in advance by its rulebook, and a UTM is no exception
- There's also the risk of unintentional character collisions between the simulated machine and the UTM
  - To store Turing machine rules and configurations on a tape, we need to be able to mark their boundaries with characters that will have special meaning to the UTM, so that it can tell where one rule ends and another begins

Encoding

- We need to do some kind of escaping to prevent ordinary characters from the simulated machine getting incorrectly interpreted as special characters by the UTM
- Design a scheme that uses a fixed repertoire of characters to encode the tape contents of a simulated machine
  - If the encoding scheme only uses certain characters, then we can be sure it's safe for the UTM to use other characters for special purposes
  - if the scheme can accommodate any number of simulated states and characters, then we don't need to worry about the size and complexity of the machine being simulated

Encoding

- The specific details of the encoding scheme aren't important as long as it meets these goals
- For example: one possible scheme uses a *unary* representation to encode different values as different-sized strings of a single repeated character (e.g., 1):
  - If the simulated machine uses the characters *a*, *b*, and *c*, these could be encoded in unary as 1, 11, and 111
  - Another character, say 0, can be used as a marker to delimit unary values: the string *acbc* might be represented as 101110110111

Encoding

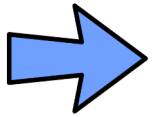
- We need a way to represent the rules of the simulated Turing machine
- We can do that by encoding the separate parts of the rule (state, character, next state, character to write, direction to move) and concatenating them together on the tape, using special separator characters where necessary
- In our example encoding scheme, we could represent states in unary too—state 1 is 1, state 2 is 11, and so on
- Concatenate individual rules together to represent an entire rulebook

Encoding

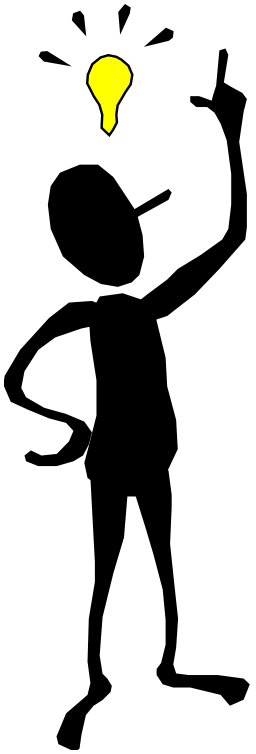
- Encode the current configuration of the simulated machine by concatenating the representation of its current state with the representation of its current tape contents
- This gives us what we want: a complete Turing machine written as a sequence of characters on another Turing machine's tape, ready to be brought to life by a simulation

Simulation

- In sum, the description of the simulated machine—its rulebook, accept states, and starting configuration—is stored in encoded form on the UTM's tape
- To perform a single step of the simulation, the UTM moves its head back and forth between the rules, current state, and tape of the simulated machine in search of a rule that applies to the current configuration
- When it finds one, it updates the simulated tape according to the character and direction specified by that rule, and puts the simulated machine into its new state
- That process is repeated until the simulated machine enters an accept state, or gets stuck by reaching a configuration to which no rule applies

Next 

- Computability and Complexity



Thanks ! ☺

Questions ?