13. Modeling Computation

Computers can perform many tasks. Given a task, two questions arise:

- Can it be done using a computer?
- If so, how can it be done?

We'll consider three models of computation:

- Finite State Machines
- Turing Machines
- Grammars and Languages

13.1 Languages and Grammars

We'll study *formal* languages and grammars.

- Formal languages can be used to model
  - *natural* languages
  - *programming* languages
- Applications are language translation, compiler design, etc.

Consider the natural English language:

e.g. “the frog writes neatly” is a valid sentence: syntax is correct; semantics, i.e., the meaning of the sentence, is however another issue that we won’t address here.

e.g. “swims quickly mathematics” is not even valid since it is not syntactically correct.

The English grammar is extremely complicated. In fact, a natural language does not usually have a well-defined rules of syntax.

On the other hand, a formal language, such as a programming language, has a well-defined rules of syntax, which can completely be described using a grammar.

Consider the use of grammars in application to programming languages:

1. How can we determine whether a combination of words is a valid sentence?

2. How can we generate the valid sentences of a formal language?

We now give a technical definition of a grammar, but first some terminology…

Definition:

- A vocabulary V is a finite set of *symbols* (e.g., words in English).

- A sentence over V is a finite string of elements from V. The set of all sentences over V is denoted by V^*.

- A language over V is a subset of V^*.

Definition:

A *phrase-structure grammar* $G = (V, T, S, P)$ is a 4-tuple, in which:

- V is a vocabulary (e.g., set of words in English)
- $T \subseteq V$ is a set of symbols called *terminals*
  - Actual words of the language (e.g., ball, house, run, course, mathematics, easy).
- $N = V - T$ is a set of special "words" called *non-terminals* (e.g., representing concepts like noun, verb, adjective, adverb in English).
- $S \in N$ is a special non-terminal, the *start symbol*.
- P is a set of *productions* of the form $b \rightarrow a$.
  - Rules for substituting one sentence fragment to another.
  - Every production must contain at least one nonterminal on its left side.

e.g., $G = (V, T, S, P)$ where $V=\{a, b, A, B, S\}$, $T=\{a, b\}$, S is the start symbol,
$P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}$

Derivation: One possible derivation using this grammar is:

$$S \Rightarrow ABa \Rightarrow Aaba \Rightarrow BBaba \Rightarrow Bababa \Rightarrow abababa.$$

e.g., $G = (V, T, S, P)$ where $V=\{a, b, A, B, S\}$, $T=\{a, b\}$, S is the start symbol,
$P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}$

Derivation: One possible derivation using this grammar is

$$S \Rightarrow ABa \Rightarrow Aaba \Rightarrow BBaba \Rightarrow Bababa \Rightarrow abababa.$$

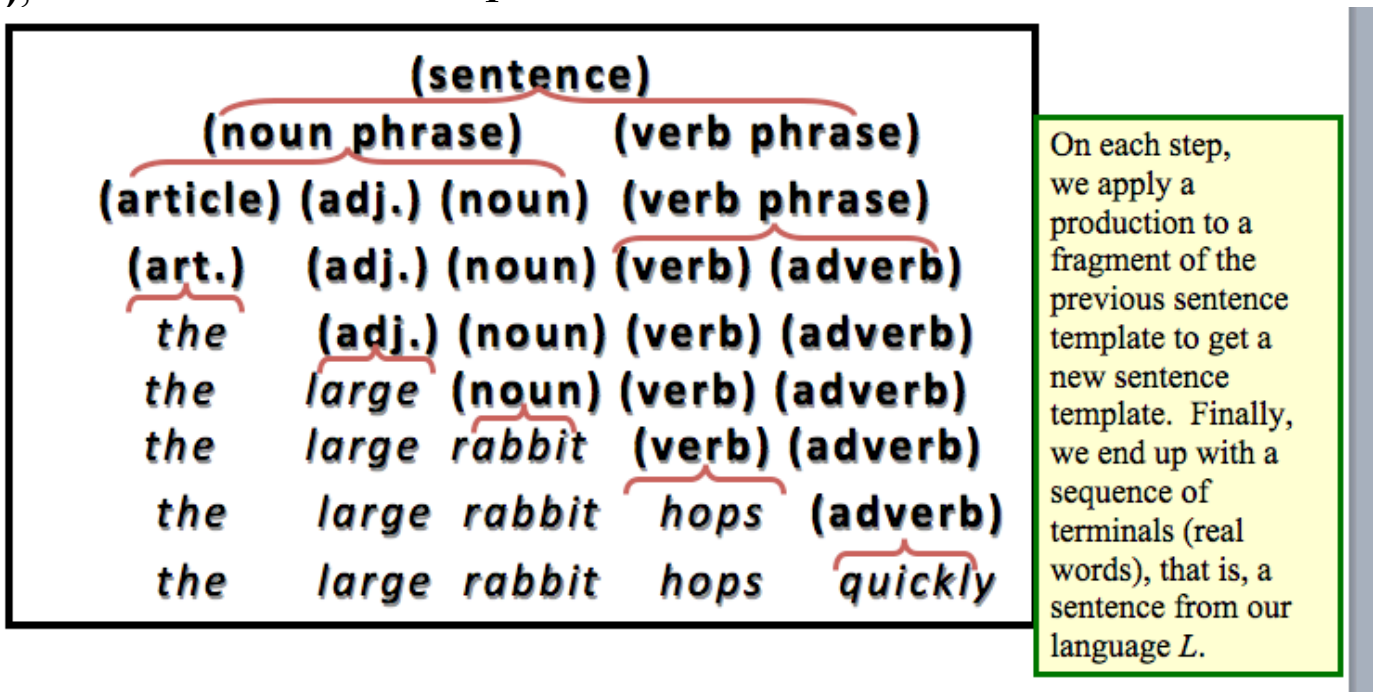
We say "*abababa* is derivable from *S* ", that is, $S \Rightarrow^* abababa$.
Hence *abababa* is a valid sentence in this language.

- $w_0 \Rightarrow^* w_n$ denotes that there's a series of substitutions starting from w_0 and ending at w_n.
- Then, w_n is *derivable from* w_0.
- Note that the relation $\Rightarrow^*$ is the connectivity relation of the relation $\Rightarrow$.

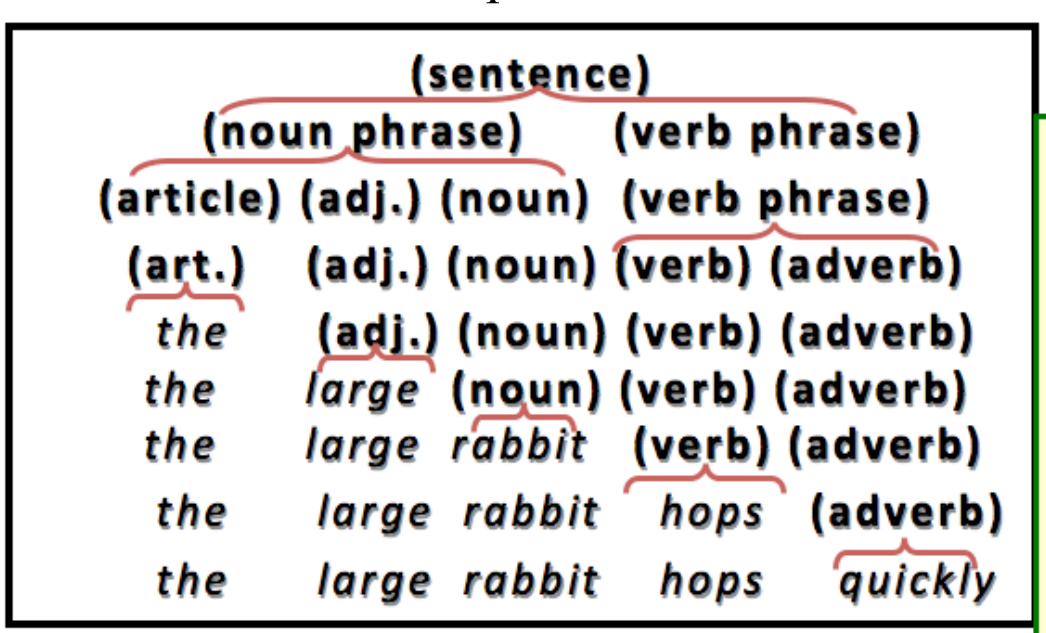
Let us consider an example on natural languages, e.g., English:
Suppose we have $G = (V, T, S, P)$, where:

- V = {**(sentence)**, **(noun phrase)**, **(verb phrase)**, **(article)**, **(adjective)**, **(noun)**, **(verb)**, **(adverb)**, *a*, *the*, *large*, *hungry*, *rabbit*, *mathematician*, *eats*, *hops*, *quickly*, *wildly*}
- T = {*a*, *the*, *large*, *hungry*, *rabbit*, *mathematician*, *eats*, *hops*, *quickly*, *wildly*}
- S = **(sentence)**
- P = { **(sentence)** → **(noun phrase) (verb phrase)**,
  (noun phrase) → **(article) (adjective) (noun)**,
  (noun phrase) → **(article) (noun)**,
  (verb phrase) → **(verb) (adverb)**,
  (verb phrase) → **(verb)**,
  (article) → *a*,
  (article) → *the*,
  (adjective) → *large*,
  (adjective) → *hungry*,
  (noun) → *rabbit*,
  (noun) → *mathematician*,
  (verb) → *eats*,
  (verb) → *hops*,
  (adverb) → *quickly*,
  (adverb) → *wildly* }

A sample derivation:



A sample derivation:



On each step, we apply a production to a fragment of the previous sentence template to get a new sentence template. Finally, we end up with a sequence of terminals (real words), that is, a sentence from our language *L*.

- The language L(G) generated by grammar G is the set of all sentences which are derivable from the start symbol.

- Sentence: Sequence of terminals (words)

Definition: (formal definition of language)

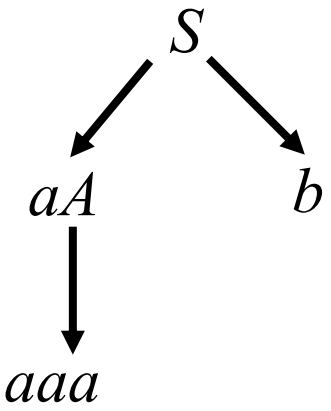
The *language* $L(G)$ generated by a given phrase-structure grammar $G = (V, T, S, P)$ is defined by

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

The set of all sentences over T is denoted by T^*

e.g. Let $G = (\{S,A,a,b\},\{a,b\}, S, \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\})$. What is $L(G)$?

Easy: We just draw a parse tree of all possible derivations.



Then we have

- $S \Rightarrow aA \Rightarrow aaa.$
- $S \Rightarrow b.$

Hence $S \Rightarrow^* aaa$ and $S \Rightarrow^* b$

Answer: $L = \{aaa, b\}$.

General rule: Capital letters denote non-terminals, whereas small letters denote terminals.

A simple PSG (phrase-structure grammar) can easily generate an infinite language:

e.g., $S \rightarrow 11S$, $S \rightarrow 0$ and $T = \{0,1\}$.

The derivations are

- $S \Rightarrow 0$
- $S \Rightarrow 11S \Rightarrow 110$
- $S \Rightarrow 11S \Rightarrow 1111\mathrm{S} \Rightarrow 11110$

and so on…

Hence $L = \{(11)^*0\}$, the set of all strings consisting of some number of concatenations of 11 with itself, followed by 0.

e.g. Construct a PSG that generates the language $L = \{0^n 1^n \mid n \in \mathbf{N}\}$.

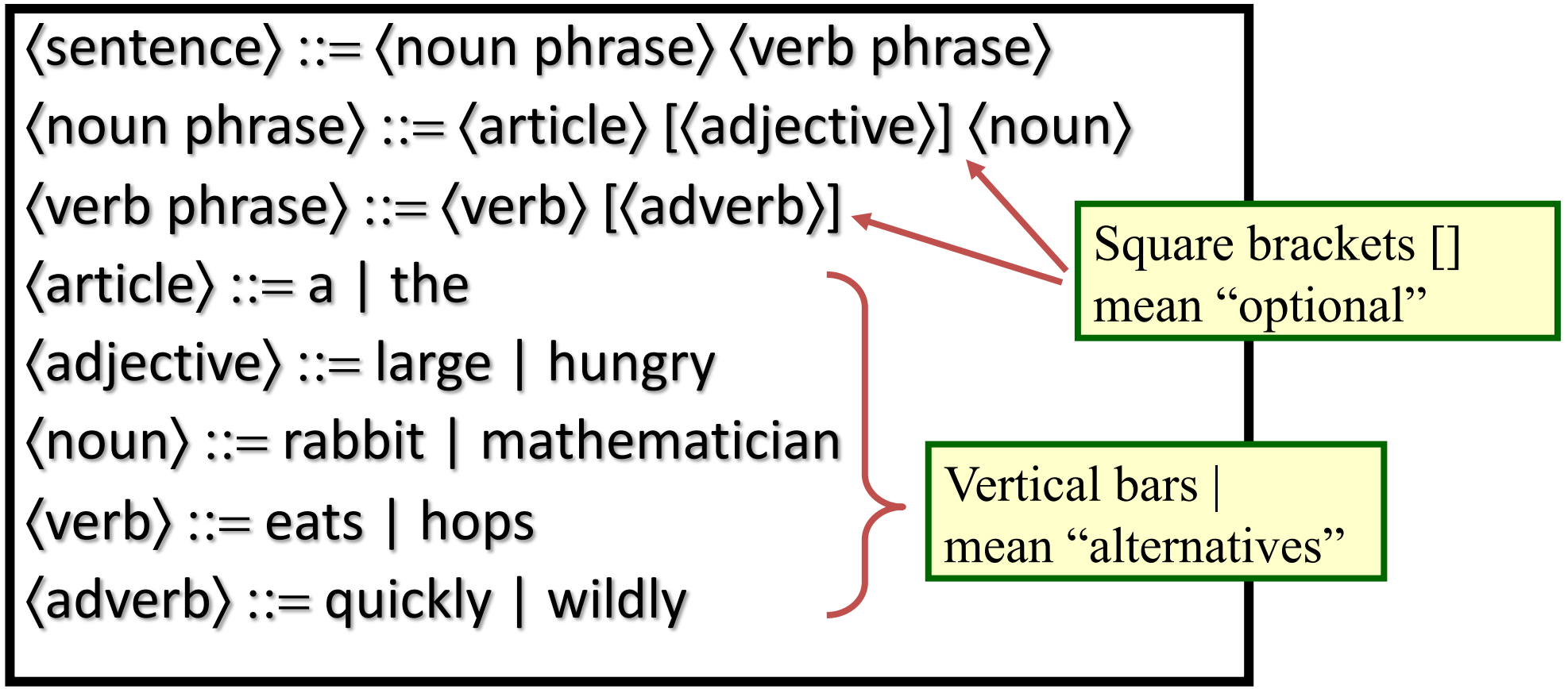
- 0^n and 1^n here represent symbols being concatenated n times (**not** integers raised to the n^{th} power).

- Solution strategy: Each step of the derivation should ensure that,
  number of 0's = number of 1's in the template so far, and all 0's come before all 1's.

$G = (V, T, S, P)$, where $V = \{0, 1, S\}$, $T = \{0, 1\}$, S is the start symbol, and the productions are

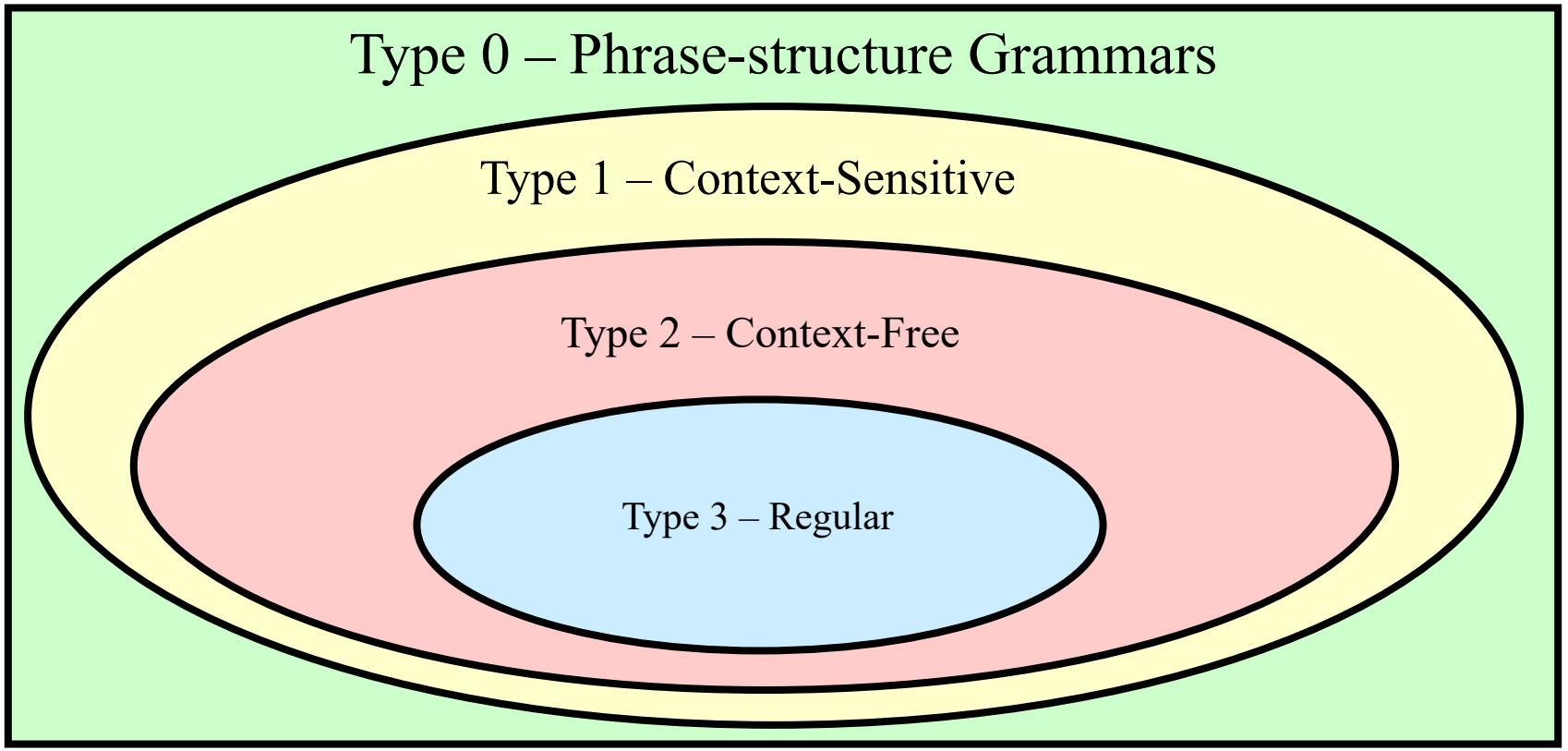
$S \rightarrow 0S1$
$S \rightarrow \lambda$ (λ denotes the empty string).

Backus-Naur Form (BNF):
Provides a much more compact representation.



Notation used to specify syntax of many computer programming languages (e.g., Java)

Types of phrase-structure grammars



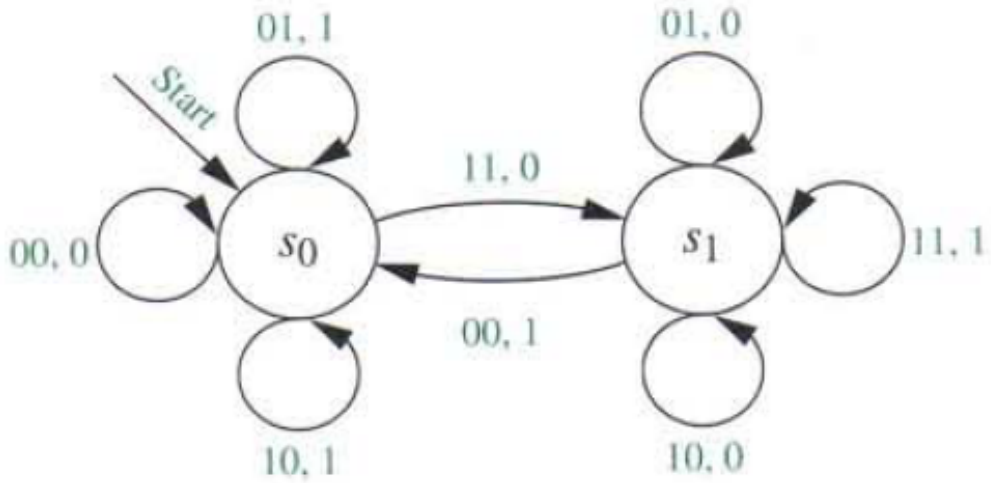
A type-0 grammar has no restrictions on its productions whereas the other types impose certain restrictions.

For example, in *regular* grammars, each production is of the form $A \to \lambda$, $A \to a$, or $A \to aB$.

13.2 Finite State Machines (with output)

Finite state machines are used for various applications in computer science:

- text/document processing: spell check, grammar check, searching, indexing;
- machine learning: speech recognition, video analysis, artificial intelligence;
- computer network protocols;
- modeling of computer components;
- game development;
- etc…



e.g. Vending Machine (a classical example):
Suppose a certain vending machine accepts **nickels** (5¢), **dimes** (10¢), and **quarters** (25¢) and a can of coke costs 30¢.

- If >30¢ is deposited, change is immediately returned.
- If the "coke" **button** is pressed, the machine drops a can of coke. The machine can then accept new payment.

Vending Machine:

- Input symbol set:
  I = {**n**ickel, **d**ime, **q**uarter, **b**utton}
- Output symbol set:
  O = {∅, 5¢, 10¢, 15¢, 20¢, 25¢, coke}
- State set: S = {0, 5, 10, 15, 20, 25, 30}
  - (S represents the money collected)

How does this machine work?

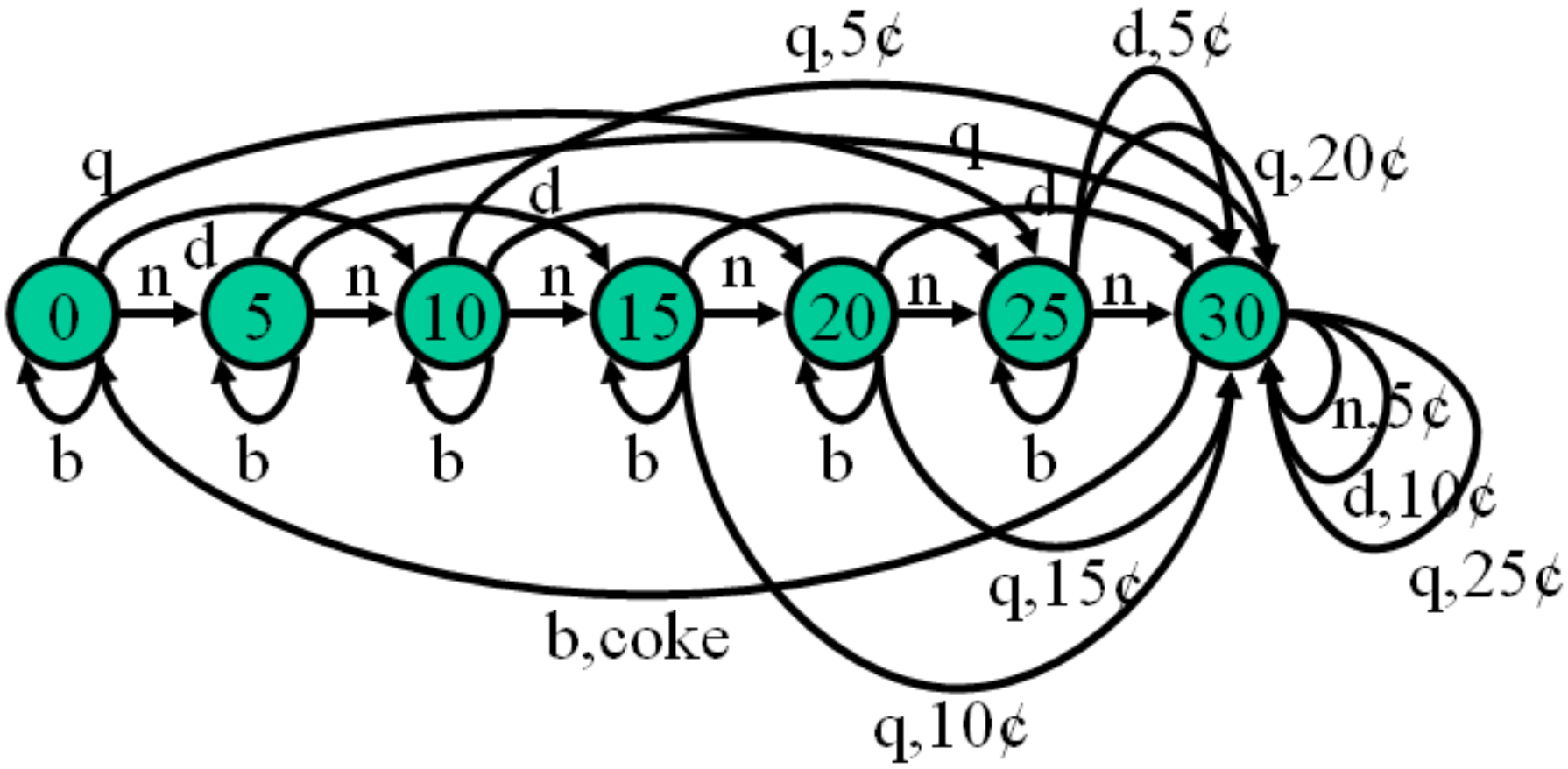
We can use a *state table* to show how it works:

Old state	Input Symbol			
	n	d	q	b
0	5,∅	10,∅	25,∅	0,∅
5	10,∅	15,∅	30,∅	5,∅
10	15,∅	20,∅	30,5¢	10,∅
15	20,∅	25,∅	30,10¢	15,∅
20	25,∅	30,∅	30,15¢	20,∅
25	30,∅	30,5¢	30,20¢	25,∅
30	30,5¢	30,10¢	30,25¢	0,coke

Each entry shows
new state, *output symbol*

State depends on the amount of money that the machine has collected.
Output is either the money that the machine returns or a can of coke.

State diagram: The graph representation of the FSM for the vending machine:



Old state	Input Symbol			
	n	d	q	b
0	5,∅	10,∅	25,∅	0,∅
5	10,∅	15,∅	30,∅	5,∅
10	15,∅	20,∅	30,5¢	10,∅
15	20,∅	25,∅	30,10¢	15,∅
20	25,∅	30,∅	30,15¢	20,∅
25	30,∅	30,5¢	30,20¢	25,∅
30	30,5¢	30,10¢	30,25¢	0,coke

A *finite-state machine* $M = (S, I, O, f, g, s_0)$ consists of

- S : state set
- I : alphabet (vocabulary) of input symbols
- O : alphabet (vocabulary) of output symbols
- $f : S \times I \rightarrow S$: state transition function
- $g : S \times I \rightarrow O$: output function
- s_0 : initial state

Note that the state (transition) table of the vending machine displays (f, g) pairs.

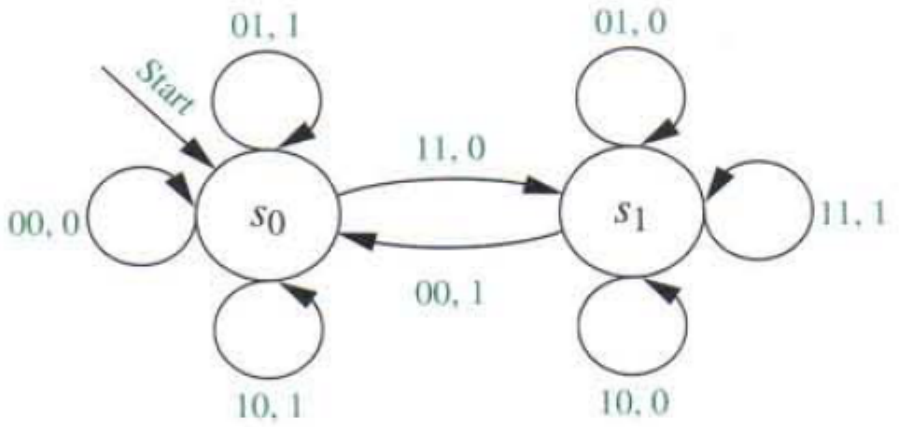
e.g. Find an FSM to add two integers ($\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$), using their binary representations:
$\boldsymbol{x} = (x_n \ldots x_1 x_0)_2$ and $\boldsymbol{y} = (y_n \ldots y_1 y_0)_2$. Hence the input will be a sequence of bit pairs (x_i , y_i) and the output will be another sequence of bits z_i.

Solution:

- Add bit by bit (first x_0 and y_0, then x_1 and y_1, …, then x_i and y_i, and so on).
- The result z_i depends on the carry bit (either 0 or 1, initially 0).

$I = \{00, 01, 10, 11\}$ (the first bit is for x, the second is for y)
$O = \{\mathbf{0},\mathbf{1}\}$ (output bit for $x+y$)

- Therefore we need two states to remember what the carry bit was previously.
- Suppose s_1 denotes the state in which the carry bit is 1, and s_0 denotes the state in which the carry bit is 0.
- The start state is s_0.

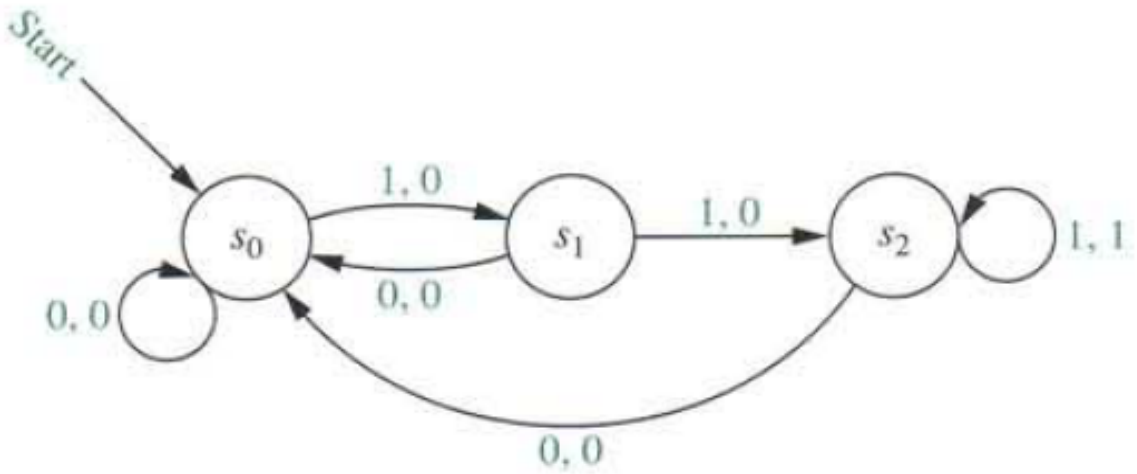


e.g. Suppose that, in a particular coding scheme, if a binary message contains three or more consecutive 1s, then that means there is a transmission error. Construct an FSM which outputs a **1** as its current output bit (hence an error signal) iff the last three bits received so far are all 1s.

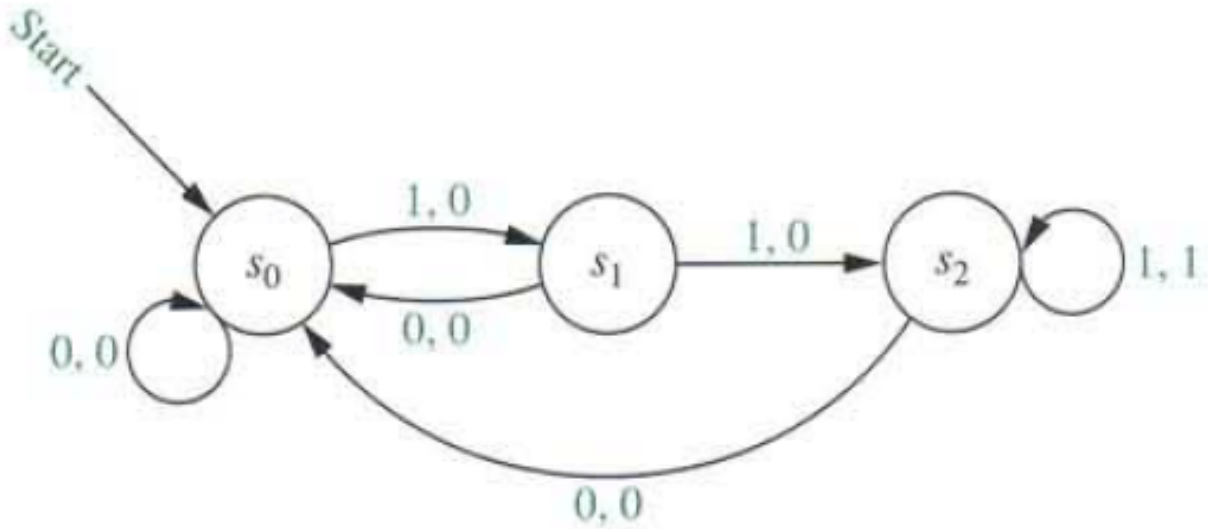
$I = \{0,1\}$ (bits of the message)
$O = \{\mathbf{0},\mathbf{1}\}$ (output is **0** if there is no error, **1** if there is an error)

Solution:

- We need to count the number of consecutive 1s in the message. It can be 0, 1, or 2.
- If it is ever 3 or more, then we need to output **1**.
- Therefore we need states, s_0 indicating zero 1s, s_1 indicating one 1, and s_2 indicating two consecutive 1s. More consecutive 1s will result in an output of **1** but will keep the state at s_2.



$I = \{0,1\}$ (bits of the message)
$O = \{\mathbf{0},\mathbf{1}\}$ (output is **0** if there is no error, **1** if there is an error)



The final output bit of the FSM constructed above is **1** iff the input string ends with 111.

Hence we say that this FSM *recognizes* the set of bit strings ending with 111.

This observation leads us to the following definition:

Definition:

A finite-state machine M *recognizes* (or *accepts*) a language L iff the last output of M is **1** for any given $x \in L$, and is **0** for any $x \notin L$.

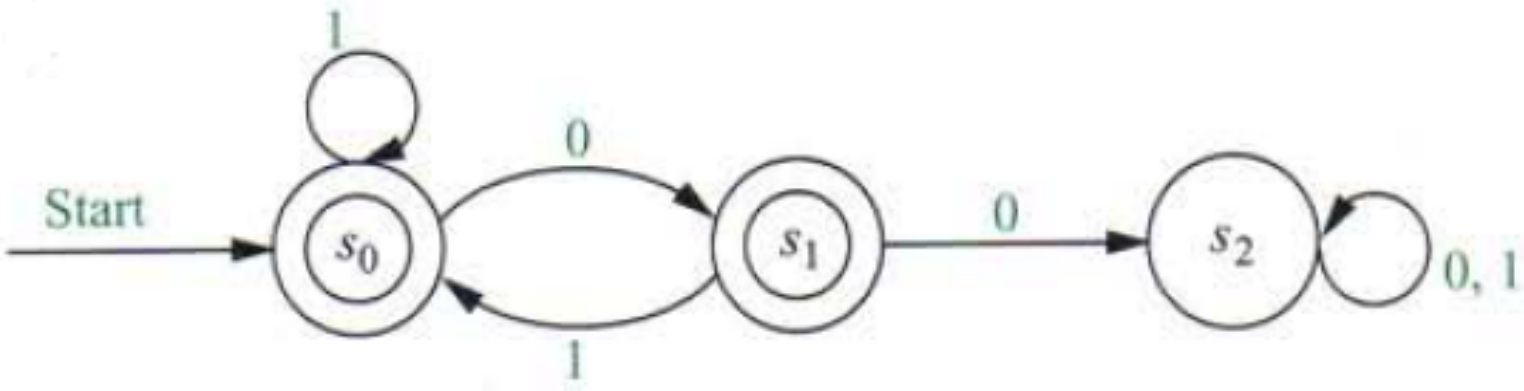
Types of finite-state machines (with output):

The type of machines we have studied so far is called *Mealy machines*, where state transitions produce output.

- A Mealy machine can be used for language recognition as demonstrated by the previous example.
- However, another type of finite-state machine, giving no output, is usually used for this purpose.
- We'll next study finite-state machines with no output, which are also referred to as *finite-state automata*.

13.3 Finite State Machines (with no output)

A finite-state *automaton* is essentially a finite-state machine with no output but with *final states*.



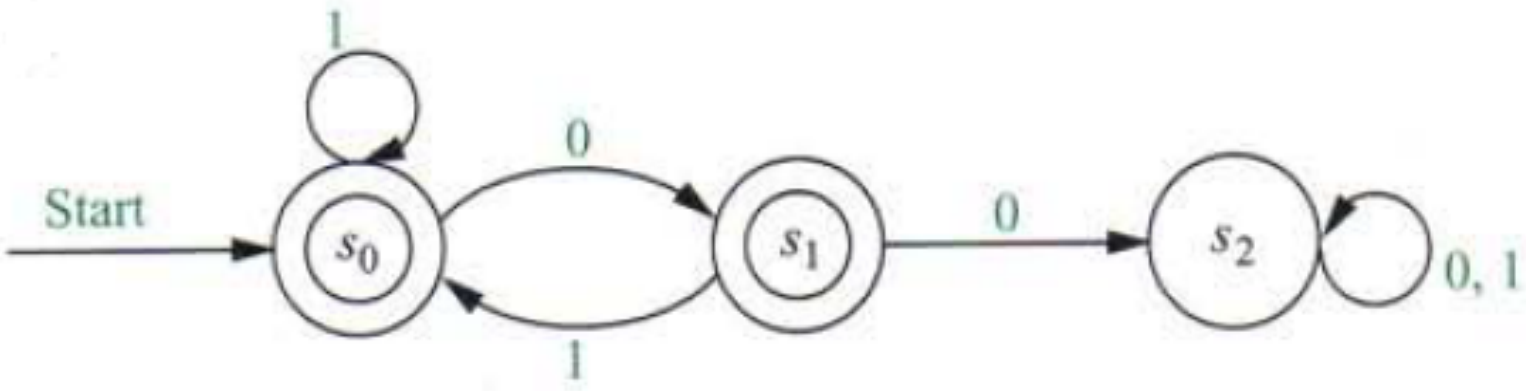
Definition:

A *finite-state automaton* $M = (S, I, f, s_0, F)$ consists of

- S : state set.
- I : alphabet (vocabulary) of input symbols
- $f: S \times I \rightarrow S$, state transition function
- s_0 : initial state
- F : set of final states.

Definition:

A finite-state automaton M *recognizes* (or *accepts)* a string x if x takes the initial state s_0 to a final state.



e.g. The finite-state automaton above recognizes the string "11011"

- The language recognized by M, denoted by $L(M)$, is the set of all strings that are recognized by M.

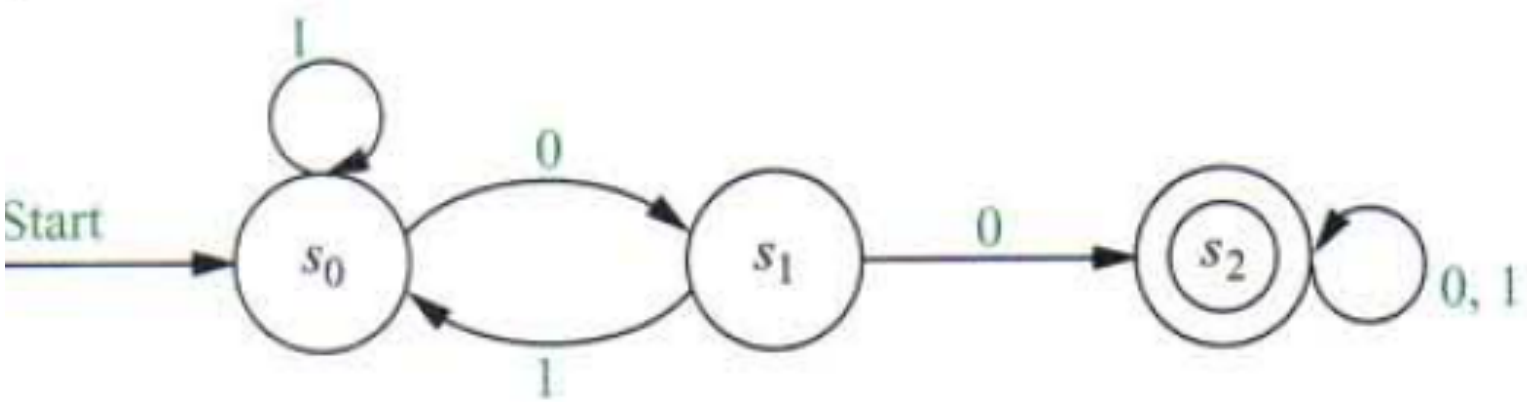
- Two finite-state automata are called *equivalent* if they recognize the same language.

e.g. Construct a finite-state automaton to recognize the set of bit strings that contain two consecutive 0s.

Vocabulary: $I = \{0,1\}$ (bits of the input)

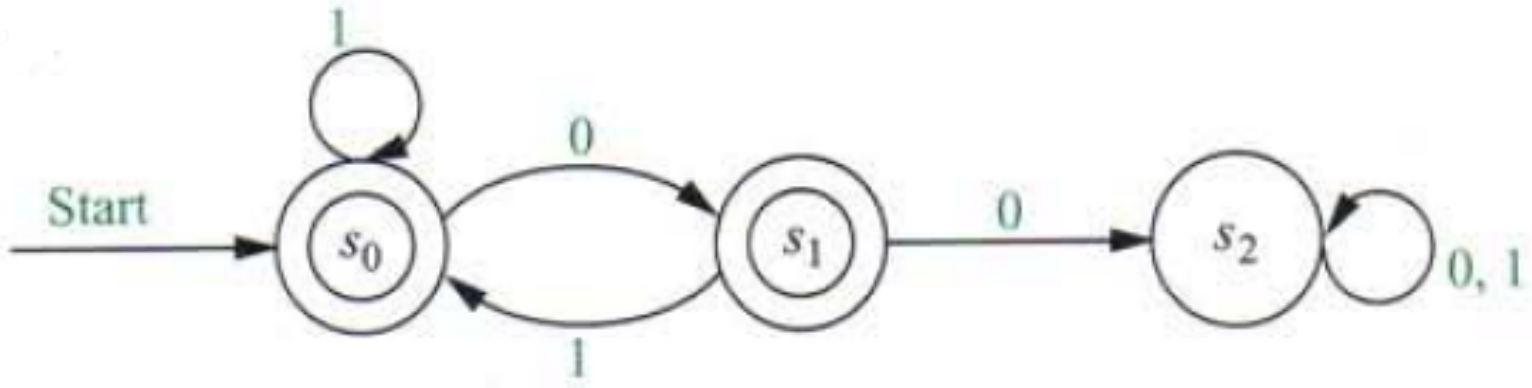
Solution:

- We need to count (remember) the number of consecutive 0s in the input. It is either 0, or 1, or 2 (or more).
- If it is ever 2 or more, then we need to stay at a final state.
- Therefore, we need state s_0 indicating zero 0s, s_1 indicating one 0, and s_2 indicating two or more consecutive 0s.
- The only final state is s_2, i.e., $F = \{s_2\}$



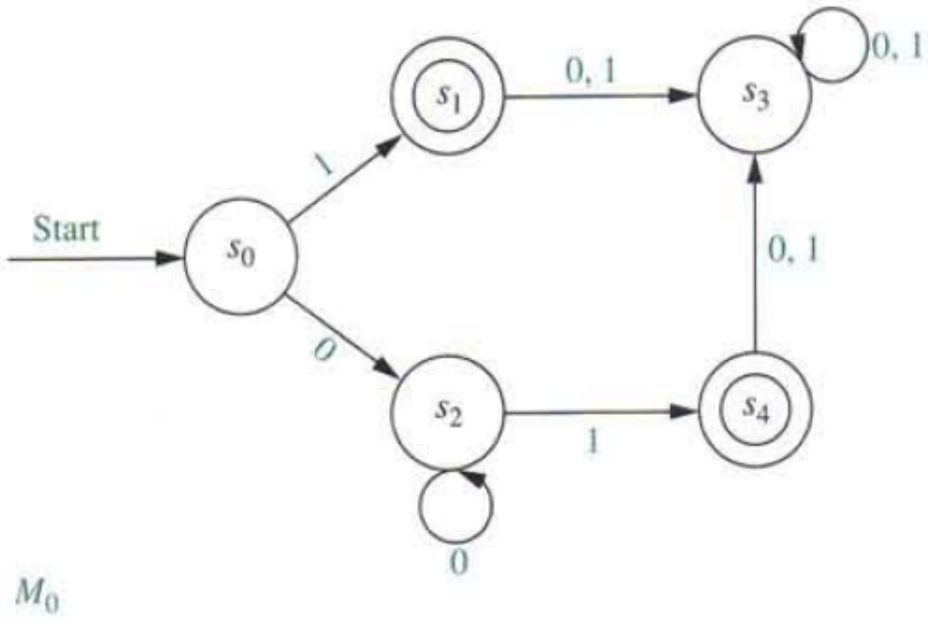
e.g. What about the **inverse** finite-state automaton? The one that recognizes bit strings that do **not** contain two consecutive 0s.

Solution: Make previous non-final states final, and previous final states non-final.



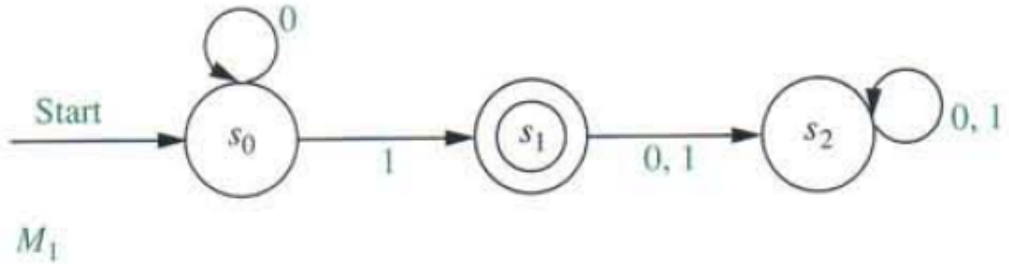
Therefore, now $F = \{s_0, s_1\}$

e.g. Show that M_0 and M_1 are equivalent:



M_0

$L_0 = \{0^n1 \mid n \geq 0\}$



M_1

$L_1 = \{0^n1 \mid n \geq 0\}$

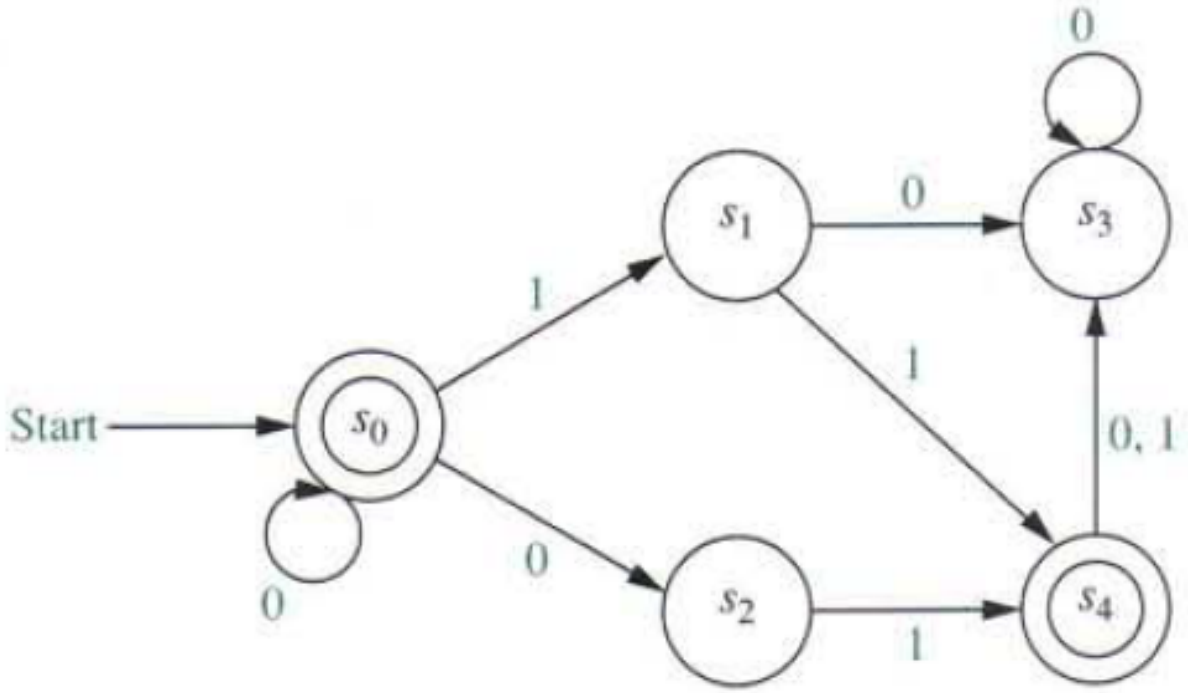
$L_0 = L_1$ $\therefore$ M_0 and M_1 are equivalent; they both recognize the bit strings with all zeros but ending with 1.

Non-deterministic vs. deterministic finite state machines

- The finite-state automata we have seen so far are all *deterministic*. Hence they are referred to as *deterministic* finite-state automata (DFA).

- In *non-deterministic* finite-state automata, the same input received at a given state can transition the machine to possibly multiple states.

e.g.



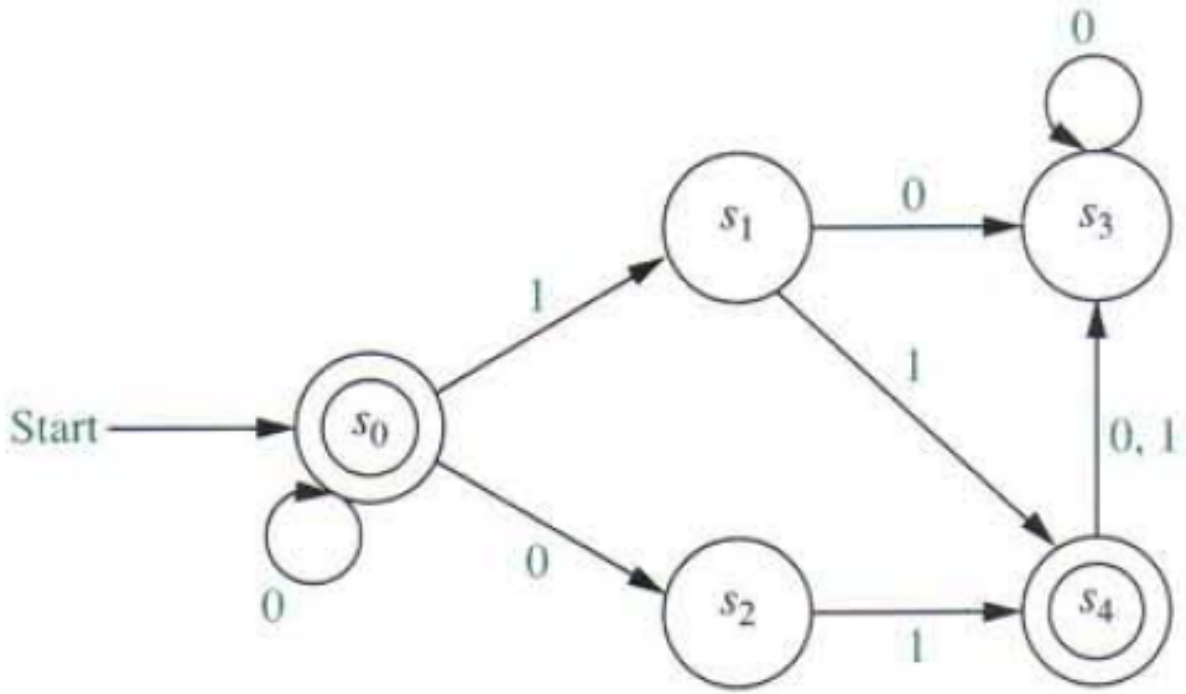
Input 0 can transition the state s_0 to both s_0 (itself) and s_2.

That is,

$$f(s_0,0) = \{s_0, s_2\}$$

Definition:
A *nondeterministic finite-state automaton* (NFA) is a finite-state automaton $M = (S, I, f, s_0, F)$ with one modification to the definition of the transition function:

- $f: S \times I \rightarrow P(S)$ is the state transition function, where $P(S)$ is the power set of S.

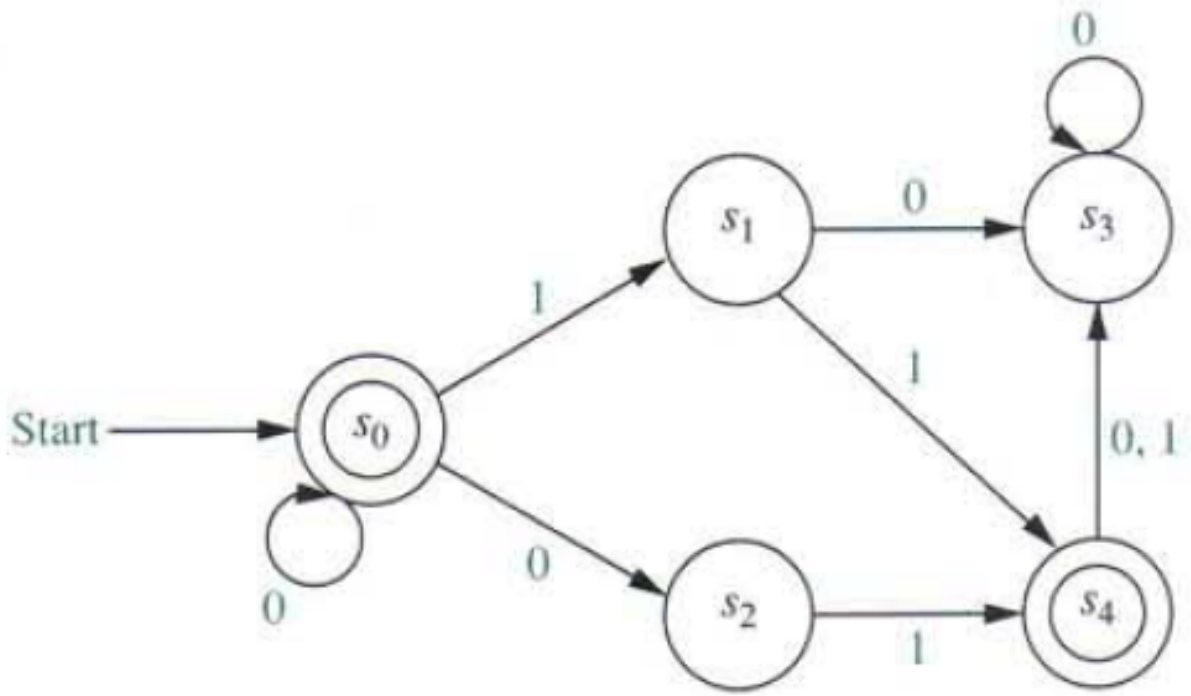


You can think of this as a machine executing multiple paths in parallel:
The machine copies itself and each copy transitions to a different state as directed by the transition function.

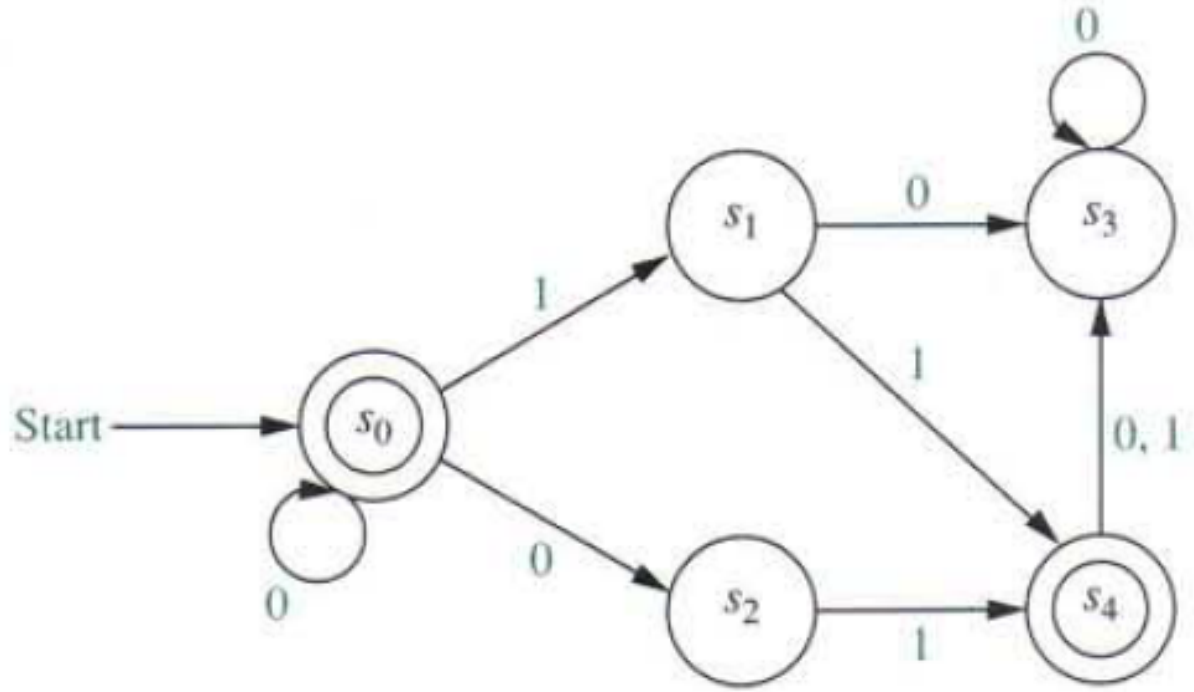
Definition:

- A nondeterministic finite-state automaton M *recognizes* (or *accepts*) a string x if there is a final state that can be obtained from s_0 when x is given as input.
- The language L recognized by M is the set of all strings recognized by this automaton.

e.g. The NFA below recognizes the string "0011".
Because the input 0011 can take the initial state to s_4 which is a final state.



e.g. Find the language recognized by this NFA.



$L = \{0^n, 0^n01, 0^n11 \mid n \geq 0\}$

Note that $f(s_3,1) = \emptyset$ (empty set is an element of the power set of S).

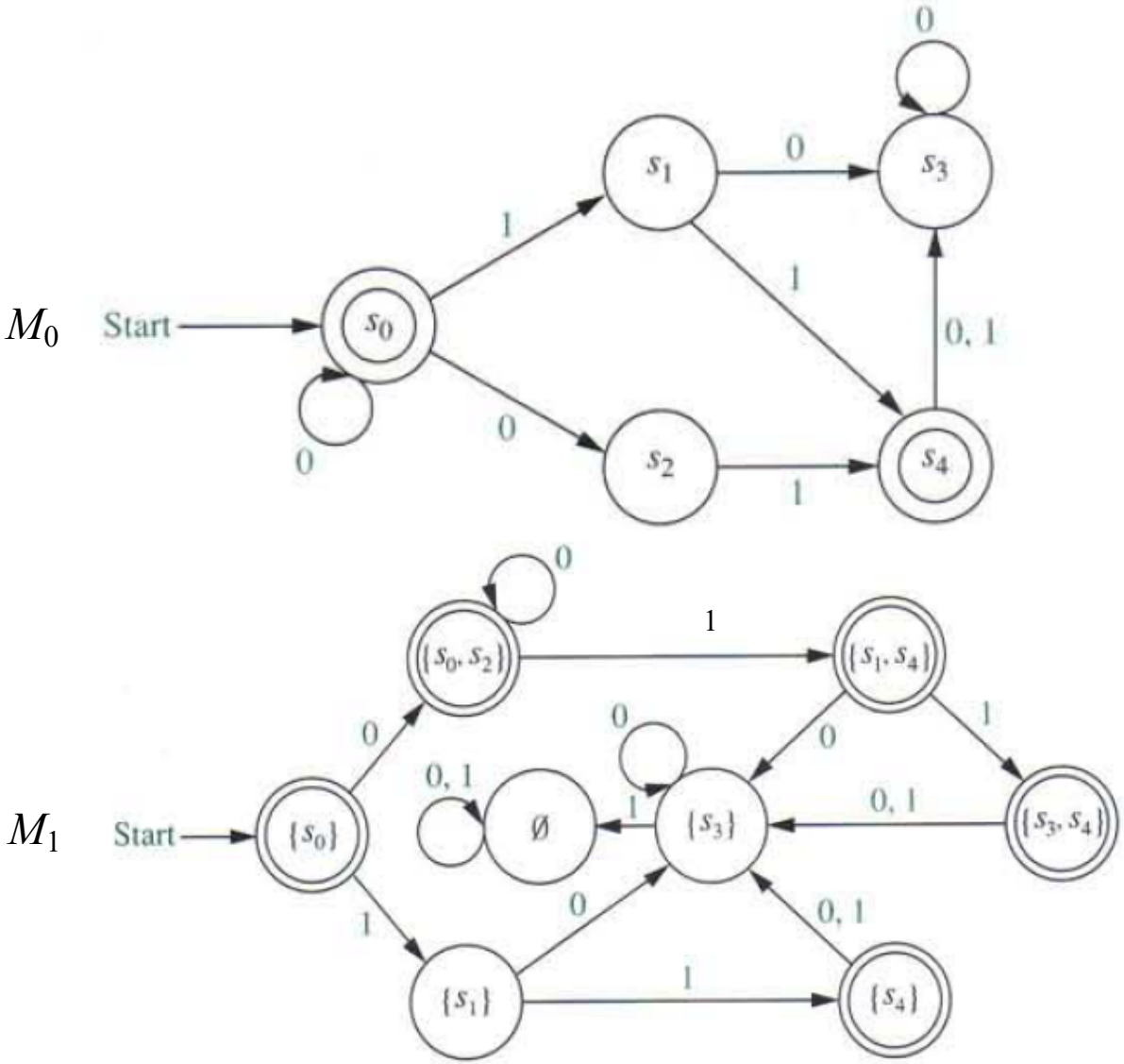
Theorem: NFA ≡ DFA

If language L is recognized by a non-deterministic finite-state automaton M_0, then one can also find a deterministic finite-state automaton M_1 that recognizes L.

Proof: This needs a constructive proof that shows how to construct M_1 from M_0. See the textbook for details of the proof, but the idea is as follows.

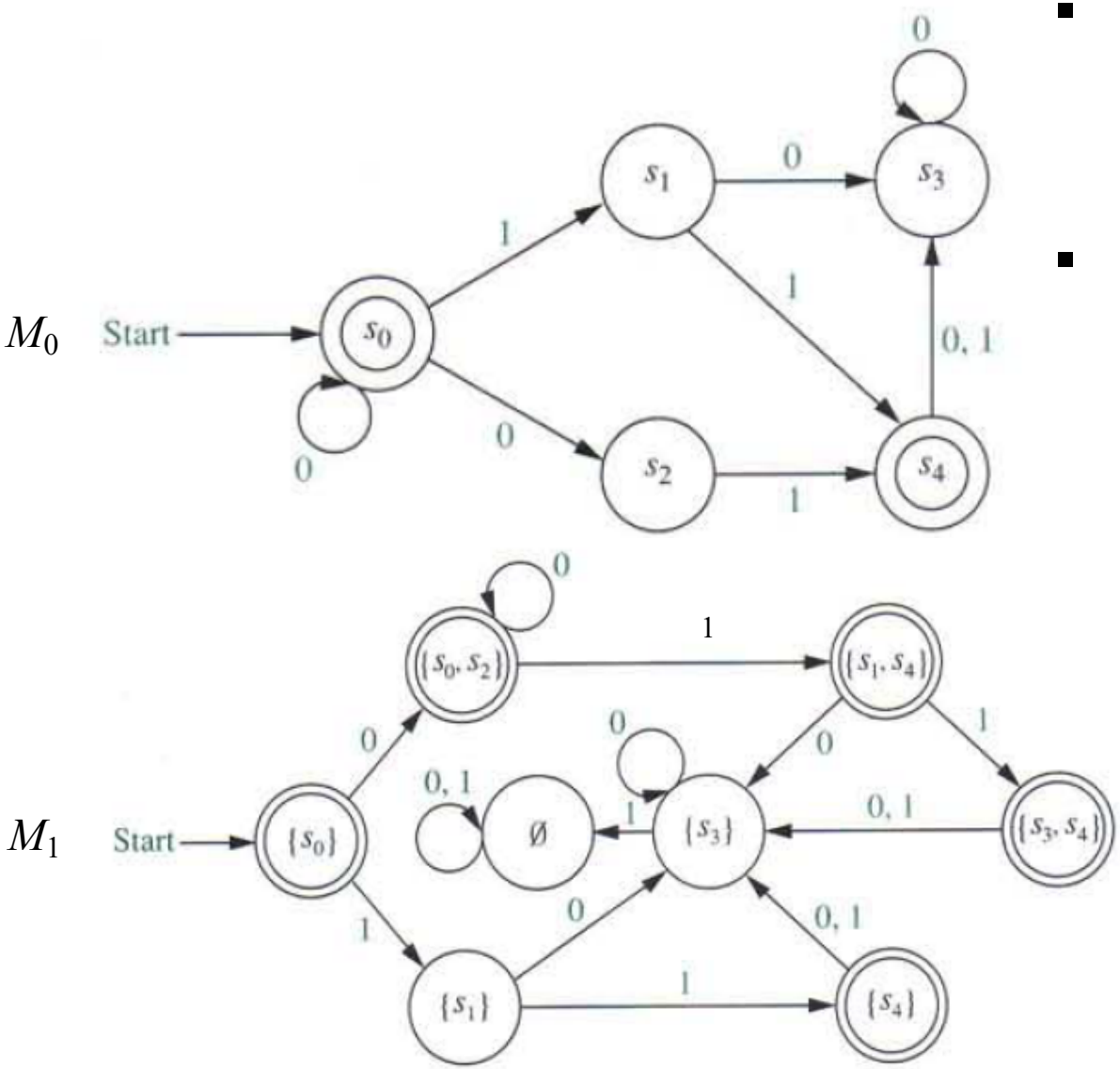
- Each state in M_1 will correspond to a **set of states** in M_0.
- For example, the starting state of M_1 will be $\{s_0\}$, where s_0 is the starting state of M_0.
- Suppose M_1 is at state $\{a_1, \ldots, a_n\}$, where each a_i is some state s_j from M_0.
- Then an input symbol x transitions to state $\{f(a_i, x)\}$, where f is the state transition function of M_0.
- $\{f(a_i, x)\}$ contains all the states from M_0, which can obtained by $f(a_i, x)$ for some i.
- The final states of M_1 are those **containing at least one final state** of M_0.

e.g. DFA equivalent of an NFA



- Each state in M_1 will correspond to a **set of states** in M_0.
- For example, the starting state of M_1 will be $\{s_0\}$, where s_0 is the starting state of M_0.
- Then state $\{s_0\}$ in M_1 will transition to $\{s_0, s_2\}$ with input 0 since in M_0 the state s_0 transitions to both s_0 and s_2 when given input 0.
- The final states of M_1 are those **containing at least one final state** of M_0.
- Note that we may need **as many as 2^n states** in M_1 if M_0 has n states.
- Yet here, we are not concerned about efficiency, we only care about that the existence of a DFA that recognizes L.

e.g. DFA equivalent of an NFA



- Here the empty set is one of the states of the non-deterministic version, because the empty set is the subset containing all the next states of $\{s_3\}$ on input of 1 (which actually leads nowhere).
- Hence you can think of the empty set as the state where the machine "halts".
  - A non-deterministic automaton can have undefined (empty) transitions due to its definition, whereas a deterministic automaton usually has a well-defined transition for any possible input.
  - Non-deterministic finite-state automata are useful since they are usually easier to construct. Once you construct an NFA that solves a problem, then you can easily construct its DFA equivalence.

13.5 Turing Machines

- The finite-state automata are limited in what they can do, simply because they lack memory which computers normally have.
    - We can use them to compute relatively simple functions such as sum of two numbers, but for example we cannot use them in practice to compute product of two numbers.
    - Finite-state automata are capable of recognizing only languages generated by *regular* grammars. Many easy-to-describe languages such as $L = \{0^n 1^n | \ n \geq 0\}$ cannot be recognized by finite-state automata.
        - *Regular* grammars are type-3 phrase-structure grammars, where each production is of the form $S \rightarrow \lambda$, $A \rightarrow a$, or $A \rightarrow aB$. (a is a terminal symbol, A and B are non-terminals)

.

- **Turing machines** have infinite memory, hence they are even more powerful than real computers which have only limited amount of memory.

Some history:

- David Hilbert (1900): “Can all questions of mathematics be answered algorithmically?”
- Kurt Gödel (1931): “This is not possible.”
- Alan Turing (1936): The first formal model of computation, the Turing Machine.

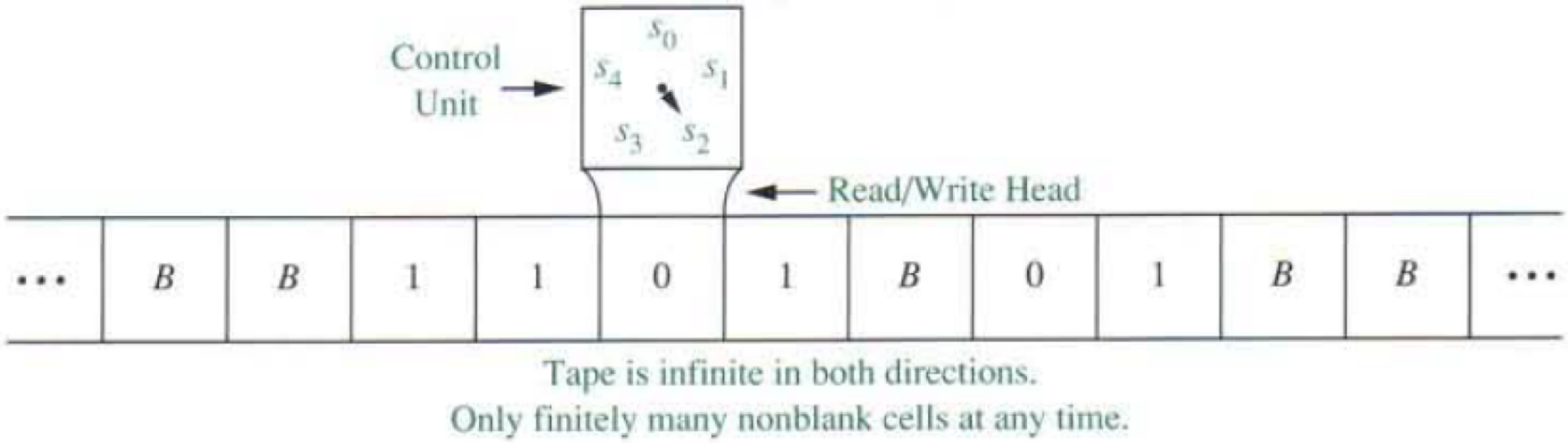
Church-Turing thesis: Any problem that can be solved with an effective **algorithm** (hence with a computer) can be solved using a **Turing Machine**.

- Note that we are not looking at efficiency issues but rather solvability issues.

- There also exist other "models of computation" which are equivalent to Turing machines:
  - Lambda calculus
  - μ-recursive functions
  - Any general-purpose programming language (e.g., Java)

- Any problem that can be solved using one such model can be solved using other models of computation.

- A Turing Machine (TM) is a finite-state machine (called the control unit) along with an ***infinite*** input tape.
- This input tape can be used as a **memory** during computation,
  - as well as to provide **output** at the end of computation.
- At the beginning, the TM is in state s_0, and the tape is at the ***leftmost non-blank position***.

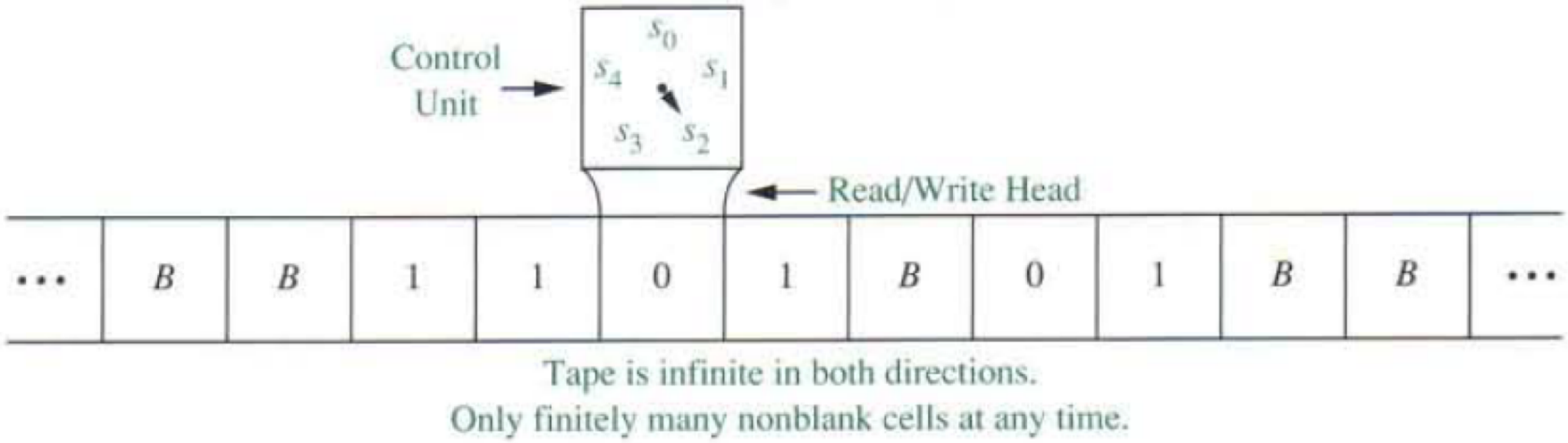


A *Turing Machine* $T = (S, I, f, s_0)$ consists of

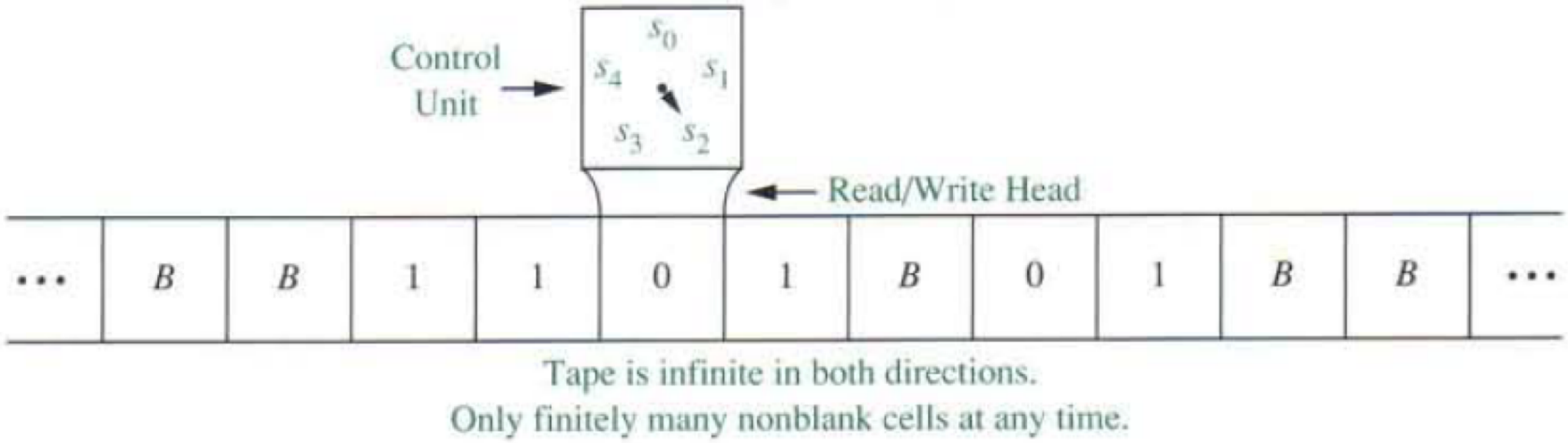
- S : state set.
- I : alphabet (vocabulary) of input symbols
- s_0 : initial state
- $f: S \times I \rightarrow S \times I \times \{L,R\}$, transition function

where

- $\{L,R\}$ symbolize "Left" and "Right"
- f can be a partial function (not necessarily defined for all state and input pairs)
- B is a special input symbol meaning "blank"

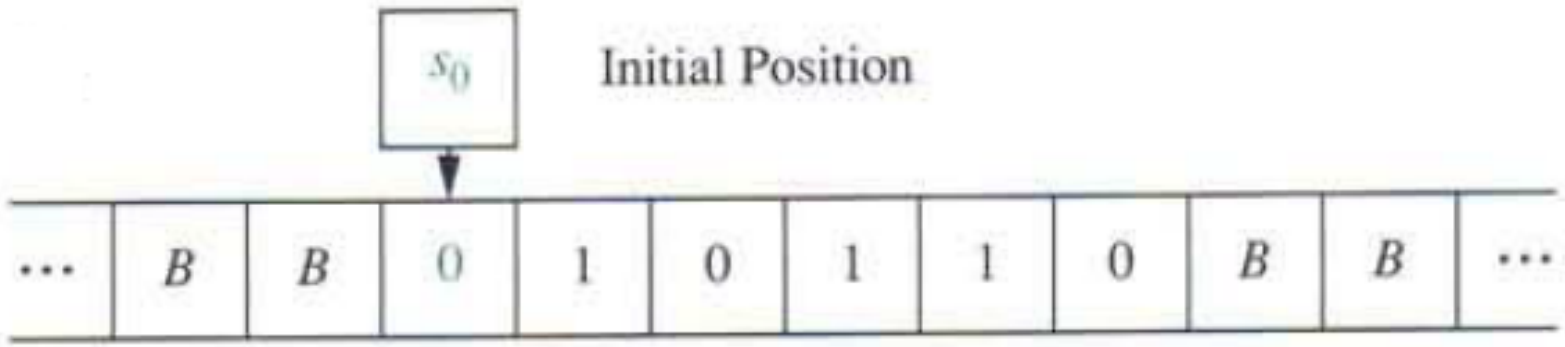


- At each step, the control unit reads the current tape symbol x.
- If the control unit is in state s and if the partial function f is defined for the pair (s, x) with $f(s, x) = (s', x', d)$, the control unit
  1. enters the state s',
  2. writes the symbol x' in the current cell, erasing x, and
  3. moves right one cell if $d = R$ or moves left one cell if $d = L$.
- These operations (conducted at each step) can be represented by a five-tuple (s, x, s', x', d).
- If the partial function f is undefined for pair (s, x), then the Turing machine T will halt.



e.g.

What is the final tape when the Turing machine T 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



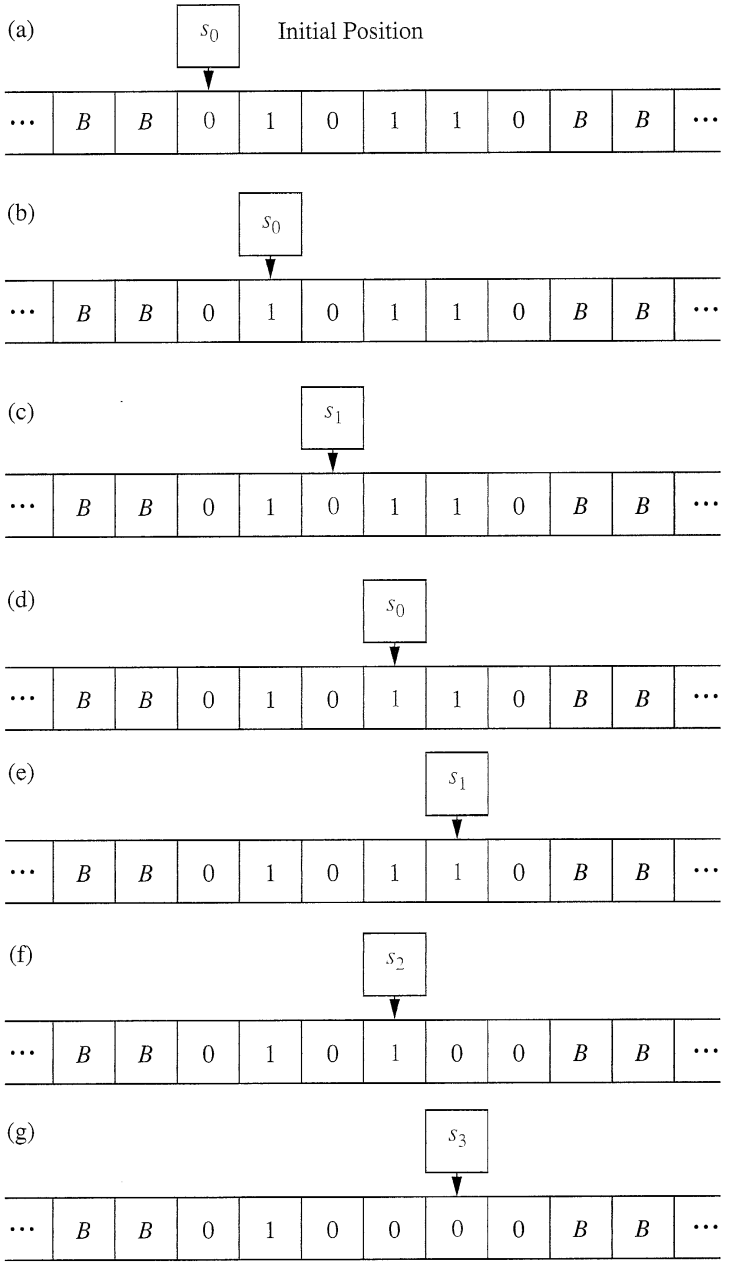
What does this TM do?

$(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, (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 $(s_2, 1, s_3, 0, R)$

The output of this Turing Machine is "010000" when given "010110".

This TM recognizes the string "010110" since it halts at s_3 which is a final state.

s_3 is a final state since it is not the first state in any five-tuple that describes the TM.



Machine halts

Definition:

A Turing Machine T *recognizes* a string x iff it halts at a final state, given x as input (i.e., starting from the initial position when x is written on the tape).

- A *final state* of T is a state that is not the first state in any five-tuple that describes T.
- For example, in the previous example, s_3 is a final state.
- Hence we say, the TM recognizes the string "010110".

$(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, (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 $(s_2, 1, s_3, 0, R)$

Definition:

The language L recognized by a Turing Machine T is the set of all strings recognized by T.

Definition:

The language L recognized by a Turing Machine T is the set of all strings recognized by T.

- It can be shown that a language can be recognized by a Turing Machine iff it can be generated by a *phrase-structure grammar*. The proof will not be presented here.

- *e.g.* It is possible to construct a TM that recognizes the language $\{0^n 1^n \mid n \geq 0\}$.

  - The same language however cannot be recognized by finite-state machines (with or without output).
  - See example Example 3 in Section 13.5.3 (8^{th} edition) for construction of this TM.

Remarks

- Given an input, a Turing Machine usually halts after a finite number of execution steps
- It is also possible that it enters an infinite loop for some inputs.
- When it halts (if ever), the string on the tape is the output of the Turing machine.
- If the Turing machine has halted at a final state, we also say that "the machine recognizes the given input string", i.e., the string which was initially on the tape.

Computing functions with Turing Machines

- A Turing Machine T can also be thought of as a computer that computes functions on integers.
- Suppose that T, when given string x, halts with string y on the tape.

  Then we say, y is the output when given x: $T(x) = y$.
- Note that the domain of T is the set of strings for which T halts.
- In the previous example: T(“010110”) = “010000”.

e.g. Show that the TM described below adds two non-negative integers: $f(n_1, n_2) = n_1 + n_2$.
Assume that the integers are represented in *unary* and separated by * on the tape.

$(s_0, 1, s_1, B, R)$, $(s_1, 1, s_2, B, R)$, $(s_2, 1, s_2, 1, R)$, $(s_2, *, s_3, 1, R)$, $(s_1, *, s_3, B, R)$

We first need to define what unary representation is:
A nonnegative integer n can be ***represented in unary*** by using $(n+1)$ 1s, i.e., by the string 1^{n+1}.

e.g.
The number 0 can be represented in unary as "1".
The number 5 can be represented in unary as "111111".
The number 2 can be represented in unary as "111".
The pair of the numbers (5,2) is represented in unary on the tape by "111111*111".

e.g. Show that the TM described below adds two non-negative integers: $f(n_1, n_2) = n_1 + n_2$.
Assume that the integers are represented in *unary* and separated by * on the tape.

$(s_0, 1, s_1, B, R)$, $(s_1, 1, s_2, B, R)$, $(s_2, 1, s_2, 1, R)$, $(s_2, *, s_3, 1, R)$, $(s_1, *, s_3, B, R)$

We have to see that

when the input is $1^{n_1+1} * 1^{n_2+1}$ the TM halts with $1^{n_1+n_2+1}$ on the tape:

- The TM simply starts by erasing the leftmost two 1s.
- Then proceeds right without any change,
- until it finds the asterisk, replaces it with 1 and halts at state s_3, which is a final state.
- If there is only one 1 followed by an asterisk at the leftmost part (i.e., 0+n), then the TM first changes its state to s_1 by erasing this 1, and then to s_3 by erasing the asterisk, and halts.
- Hence the output is simply (n+1) 1s, which represents the number n.

e.g. Construct a Turing machine with tape symbols 0, 1, and B that, when given a bit string as input, replaces the first 0 that it encounters with a 1 and does not change any of the other symbols on the tape.

$(s_0, 0, s_1, 1, R), (s_0, 1, s_0, 1, R)$

e.g. What is the output of the Turing machine below, when given the string “aab” as input.

$(s_0, \text{a}, s_1, B, R)$, $(s_1, \text{a}, s_0, B, L)$, (s_0, B, s_1, B, R), (s_1, B, s_0, B, L), $(s_1, \text{b}, s_2, \text{a}, R)$

The TM enters into an infinite loop; hence generates no output.

Can you find an input for which this TM generates an output?

Generality and Equality of Turing Machines:

- There exist different types of Turing Machines:
  - A TM with multiple tapes
  - A TM with a multi-dimensional tape (e.g., can also move up and down)
  - A TM that can read multiple tape locations simultaneously
  - A TM that is non-deterministic (where a state and input pair can transition in multiple ways)

- All types of Turing Machines compute the same set of functions.

- Hence they are all equivalent in terms of computability, but not necessarily speed or ease-of-use.

- Note that a TM (of any type above) can only perform a specific task, when specified in terms of some tuples. Hence for each given task, we need to define a different TM.

- To address this problem, there is also so called Universal Turing Machine (proposed by Alan Turing) that can simulate any TM when given the description of that TM and its input.

Complexity, Computability, and Solvability

- All these notions can be studied more *formally* using Turing machines.

- The kind of problems that are most easily studied by Turing machines are those problems that can be answered either by "yes" or "no".

Definition: A ***decision problem*** (*yes-or-no problem*) asks whether statements of a particular type are true.

e.g., is $x \in L$? or is n prime? etc.

- If there exists an effective algorithm which can decide whether every instance of a problem is true or not, then we say this problem is *decidable* (*solvable*). Otherwise, that is, if it can be shown that no such algorithm exists, we say the problem is *undecidable* (*unsolvable*).

- Note that most problems can be recast as decision problems.

    - *e.g.* Finding the maximum can be recast as: Is 25 the maximum of this sequence? Is 26 maximum of this sequence? Is 27 …

Definition: Computability

- A function that can be computed using a Turing Machine is called ***computable***.
- Other functions are called *uncomputable*.
- Note that every decision problem can be formulated as a function returning 1 for true instances and 0 for false instances.

- Some *unsolvable* problems (all decision)
  - The problem of determining whether two context-free grammars generate the same language
  - The tiling problem (whether a plane can be covered using a given set of tiles, with repetition)
  - Hilbert's 10^{th} problem (whether there are integer solutions to a given polynomial equation with integer coefficients)
  - Halting problem: Given a program *P* and its input *I*, decide if the program halts or loops forever. No algorithm can do this for all programs! (Proved by Alan Turing; see your textbook for the proof.)

- Some *uncomputable* functions

  e.g. Busy beaver function (see Exercise 31 in Chapter 12.5, 6^{th} edition)

- In *Algorithms* chapter, we had informally defined the notion of complexity classes P and NP as well as tractable and intractable problems:

- We said that

  "If a problem is solvable with polynomial worst-case complexity, it is called tractable, and other problems are intractable such as those with solutions of complexity $O(2^n)$, $O(n!)$, $O(n^n)$, etc."

- We will now be a bit more formal and precise about these concepts…

Complexity class P (Polynomial-time):

- A (decision) problem is in P, if and only if there is a **deterministic** Turing Machine that solves the problem in polynomial time in the size of its input at the worst case.
- In other words, given any input of size n, the TM halts in at most $O(n^c)$ steps (with c constant).
- Problems of this type are called *tractable*.
- Problems not in P are called *intractable*.

Complexity class NP (Non-deterministic polynomial-time):

- A (decision) problem is in NP iff there is a **non-deterministic** Turing Machine that solves the problem in polynomial time in the size of its input at the worst case.
  - In other words, given any input of size n, the non-deterministic TM halts in at most $O(n^c)$ steps (with c constant).
- Note that, for a non-deterministic Turing Machine, a (state, input) pair can transition in multiple ways.
- An important property of NP: Any given solution can be verified in polynomial time with a *deterministic* Turing machine.
- Example: *Subset-sum* is an NP problem:
  - Consider the set of integers {2,5,-3,4,-1,-2}. Is there a subset of integers whose sum is equal to 0?
  - You can easily check any given combination; for instance {5,-3,-2} works. But finding out whether such a combination exists is a more difficult problem, which is NP.

Complexity class NP (Non-deterministic polynomial-time):

- You can think of non-deterministic TM as if all states transitioned using the same state-input pair are continued to be executed in parallel.
  - Or as if the TM makes the right "guess" and follows only the path that leads to the correct output.
- Since every deterministic TM is also a non-deterministic TM, $P \subseteq NP$.
- One of the most important and hardest problems in computer science is to figure out whether or not $P = NP$. Many researchers believe $P \neq NP$.
- Note also that there are intractable problems which are neither in P nor in NP.

Further terminology (informal)

- NP-complete problems are the hardest NP problems.
  - An NP-complete problem is such that any NP problem can be “poly-time” reduced to it.
  - So any of these problems can be solved in polynomial time then all NP problems can be solved in polynomial time!
  - It is accepted, though not proven yet, that no NP–complete problem can be solved in in polynomial time.
  - Example: Graph isomorphism is an NP problem whereas subgraph isomorphism is NP-complete.
  - Example: Subset-sum problem is NP-complete.
- NP-hard problems are those which are not necessarily NP but at least as hard as any NP problem.
  - An NP-hard problem is such that all NP problems can be “poly-time” reduced to it.