Computing functions with Turing machines

Turing Machines with Outputs

- When we begin the computation the tape contains the input.
- When the TM accepts (halts) return what is written in the tape.
- TM doesn’t reject in any input.

Number representation

- Decimal: 12
- Binary: 1100
- Unary: 111111111111

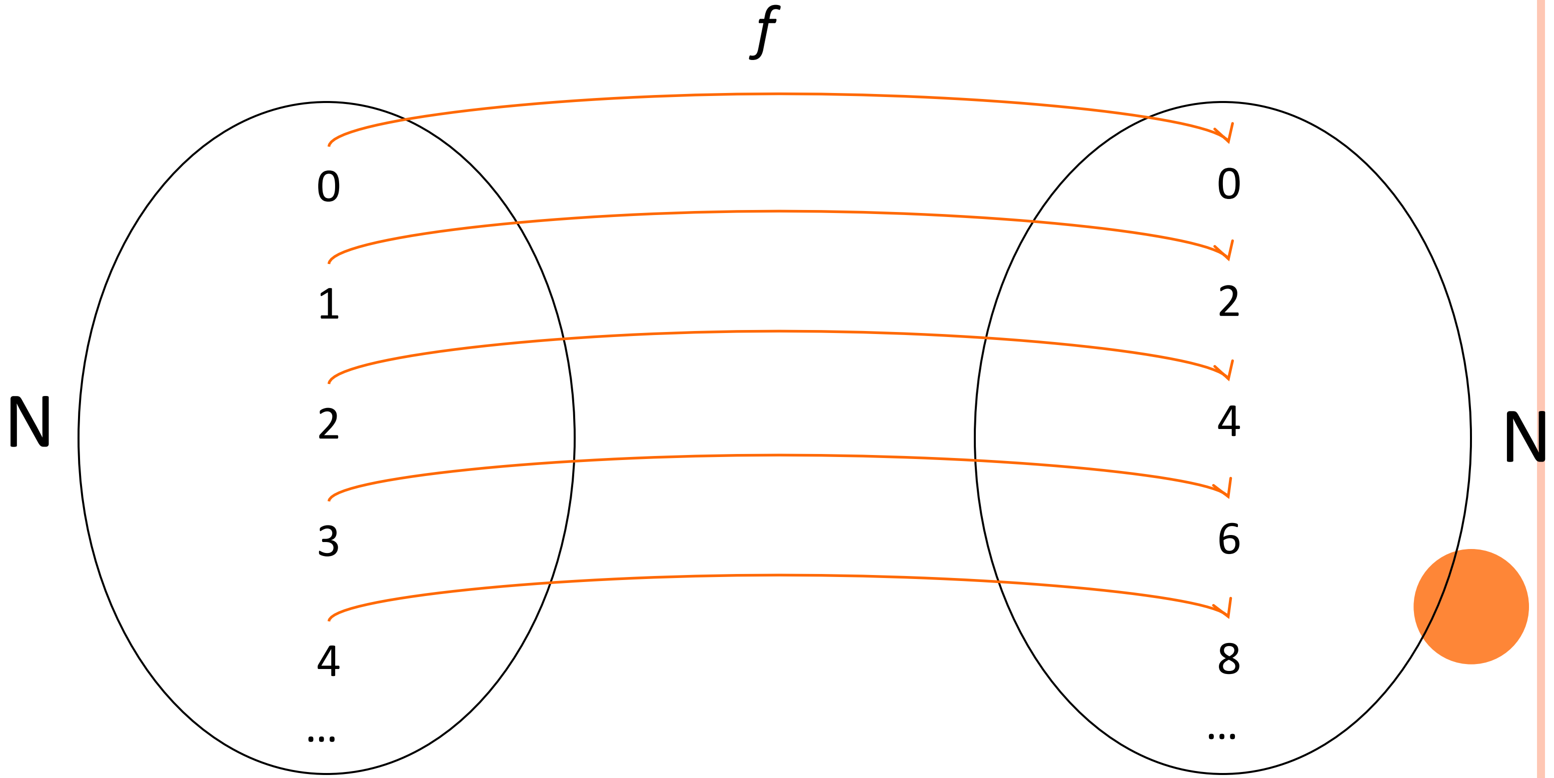
The unary is space consuming so generally we prefer binary.

When we don’t care about resources it is more convenient to use unary (easier to manipulate with TMs).

TOTAL AND PARTIAL FUNCTIONS

- A function *f* : N → N is total (or just function) when *f*(n) is defined for every n

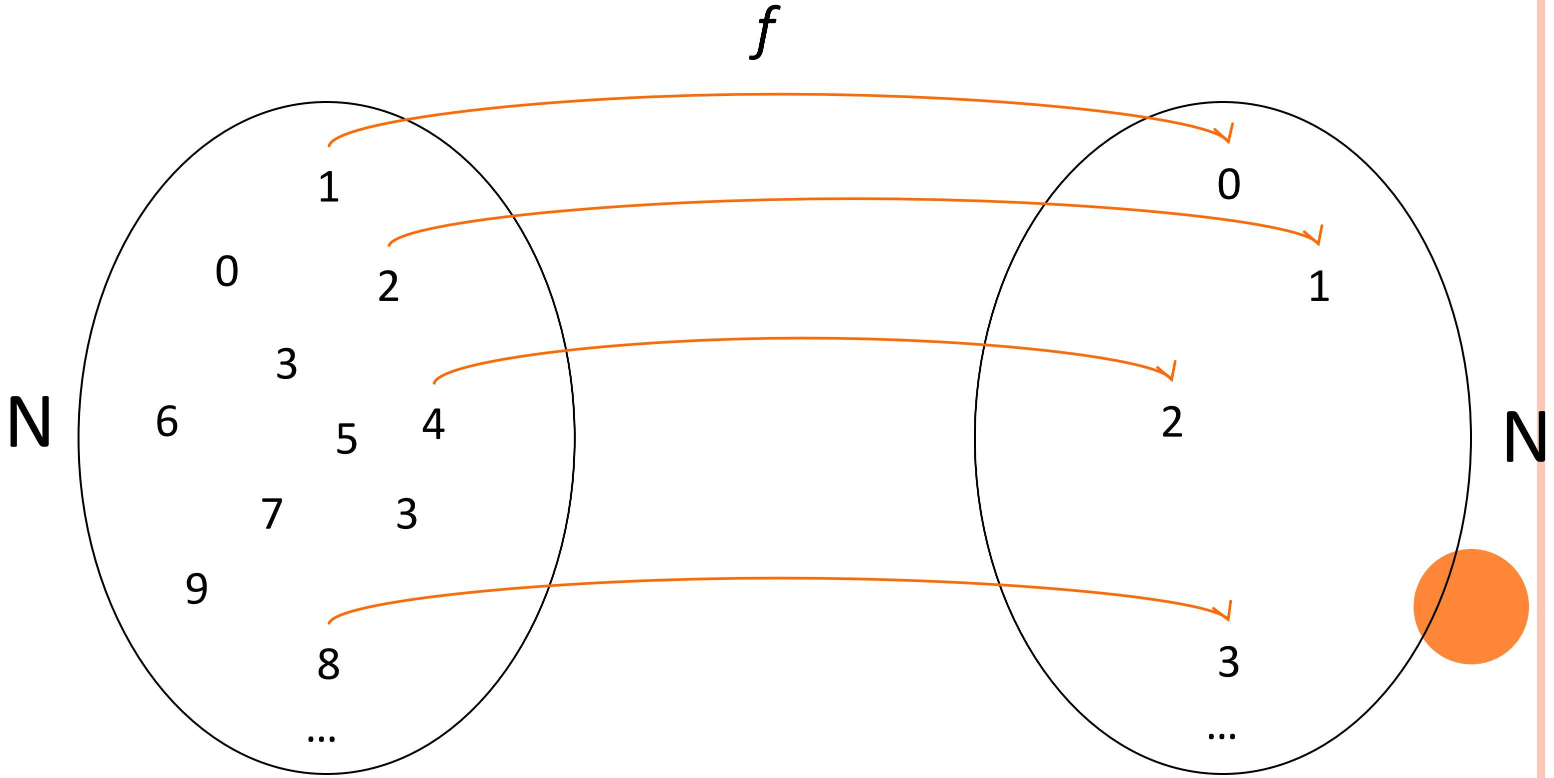
Example: *f*(n) = 2n



TOTAL AND PARTIAL FUNCTIONS

- A function *f* : N → N is called partial when *f*(n) is defined for some n.

Example: *f*(n) = logn



Computable and Partially Computable Functions

- A (total) function *f* : N → N is (total) computable if we can find a Turing Machine that computes it (given any number n in Unary as input in the tape, after completing the computation the tape contains f(n) in Unary).
- A partial function *f* : N → N is said to be partially computable if there is a Turing Machine that partially computes it (if *f* is defined for n then the machine should output *f*(n), else it should loop for ever).

EXAMPLE: *F*(N) = 2N IS COMPUTABLE

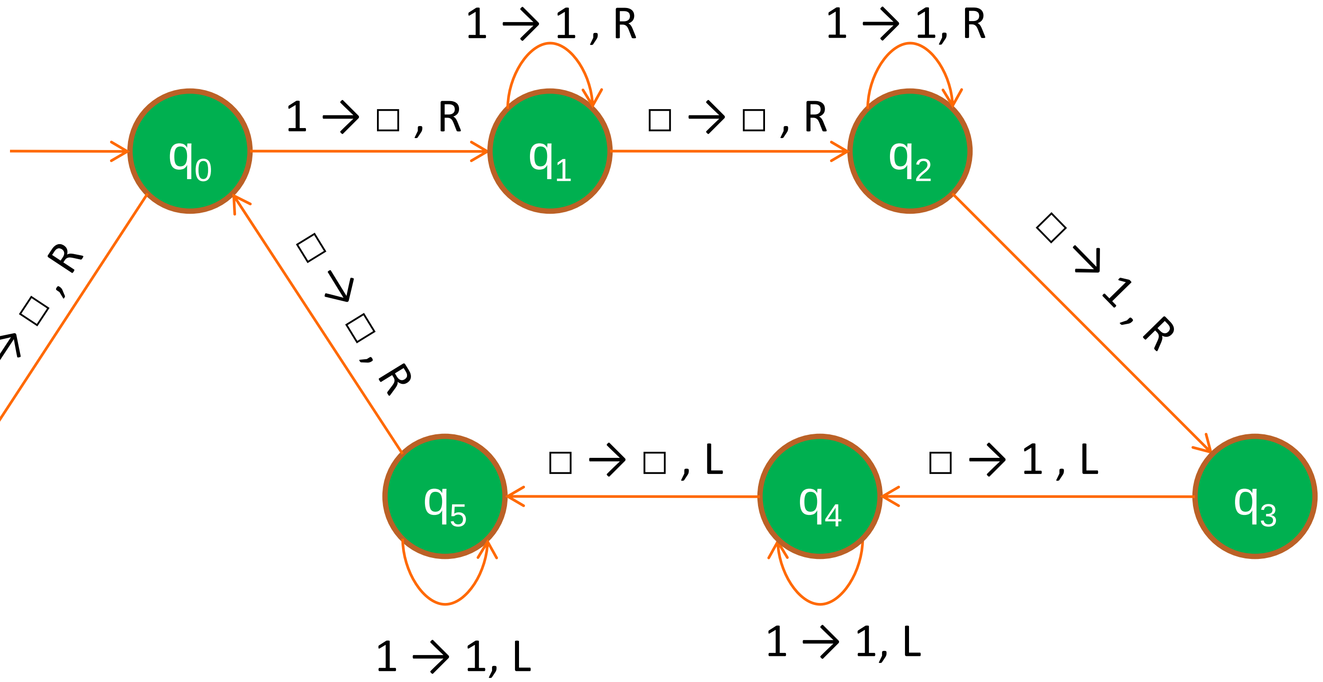
We design a TM that computes *f*(n).

High Level Program:

- The tape is divided into input and output (output is right after the first blank after the input)
- Repeat:
  - Erase one 1 from the input.
  - Pass along the rest of the input
  - Pass the blank that separates the input from the output.
  - Pass along the output until you reach the end (blank).
  - write two 1s.
  - Go to the beginning of the input.
- Until the input is erased (accept).

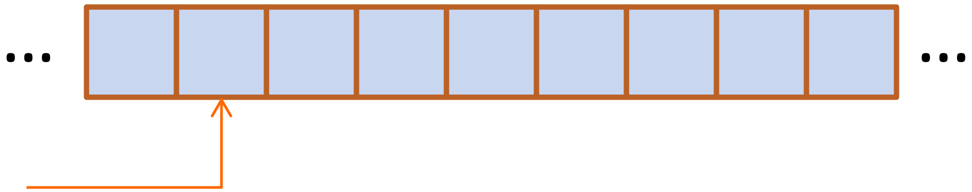
Example: *f*(n) = 2n is computable

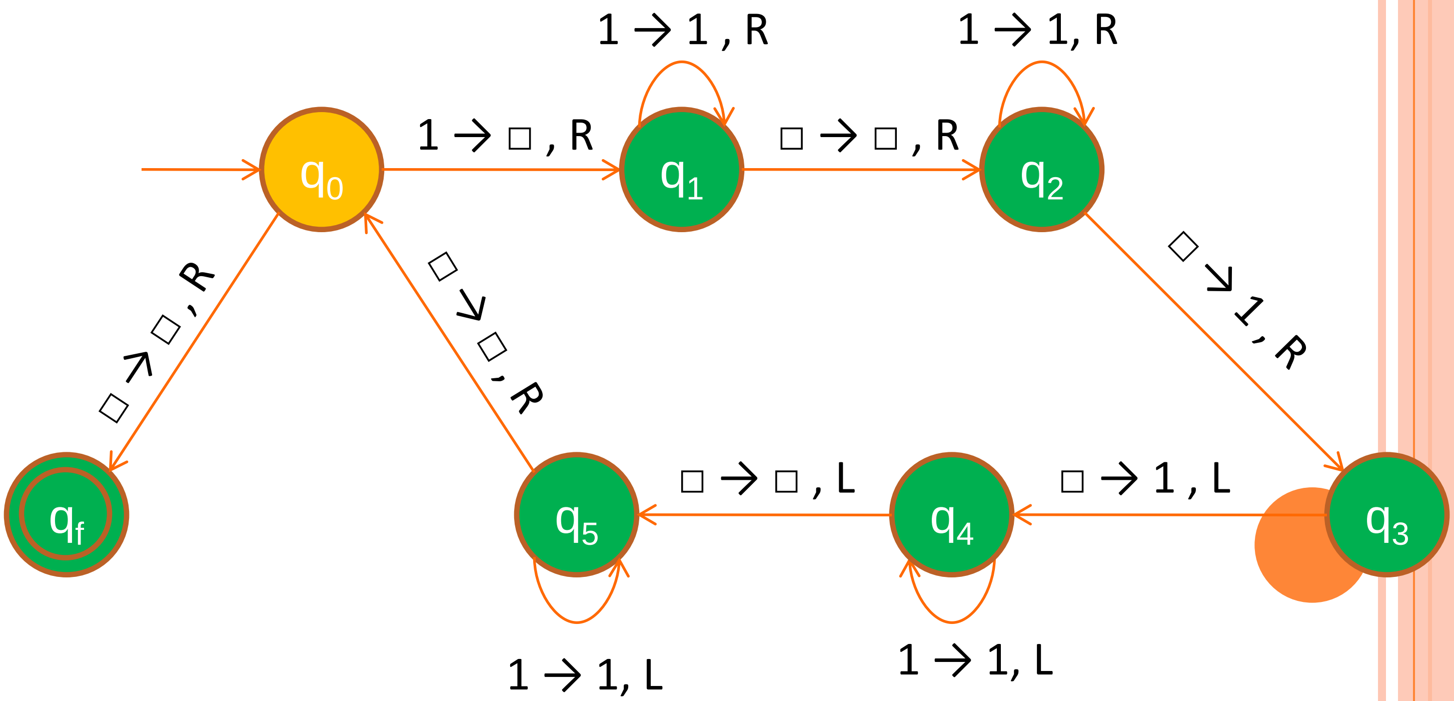
The machine for *f*(n) = 2n



Example: *f*(n) = 2n is computable

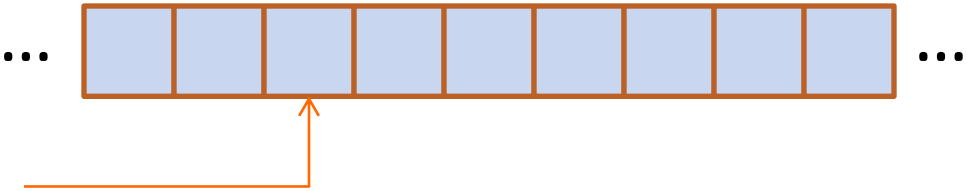
- Test input: ε

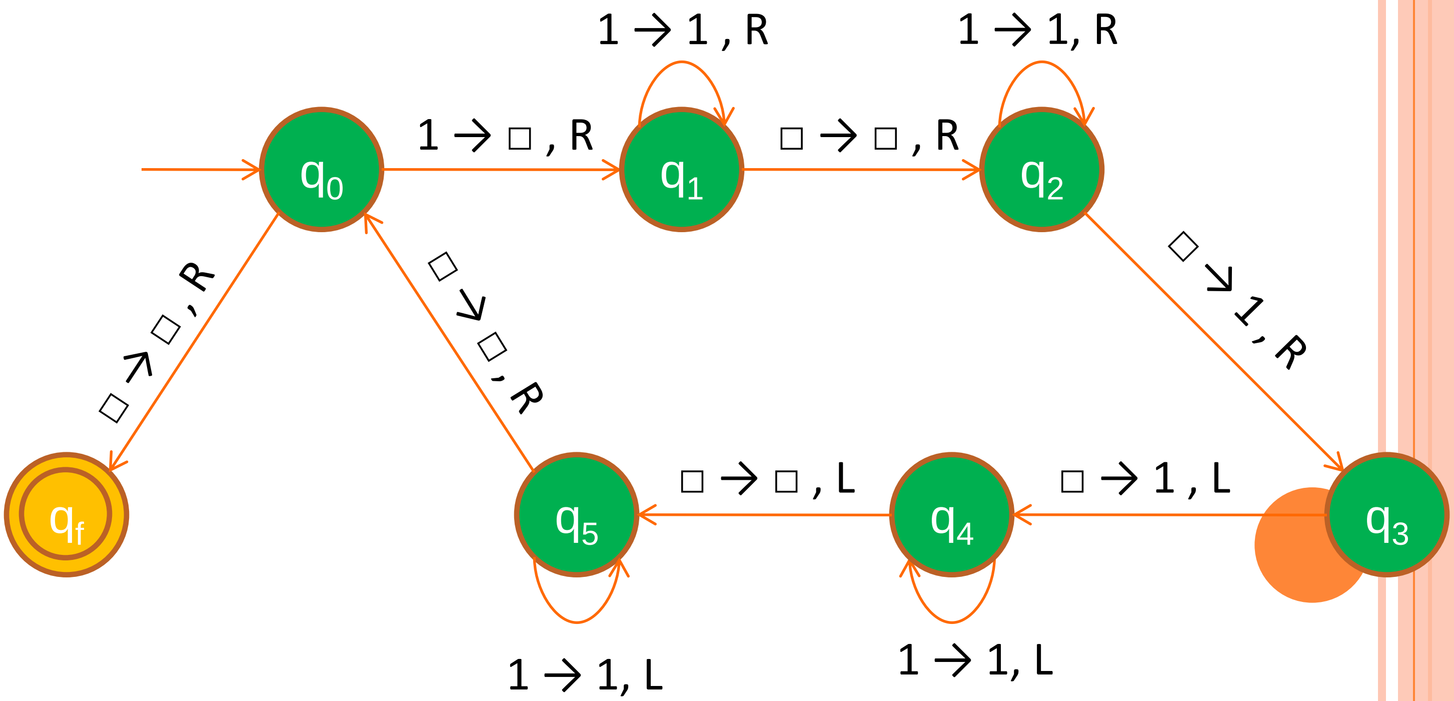




Example: *f*(n) = 2n is computable

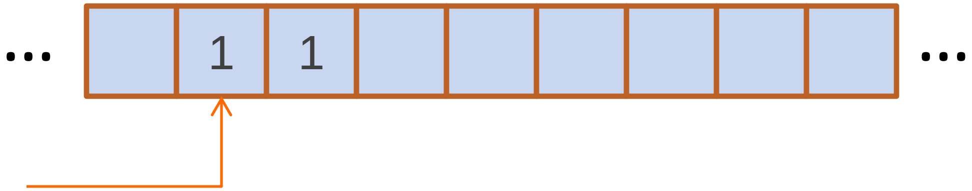
- Test input: ε

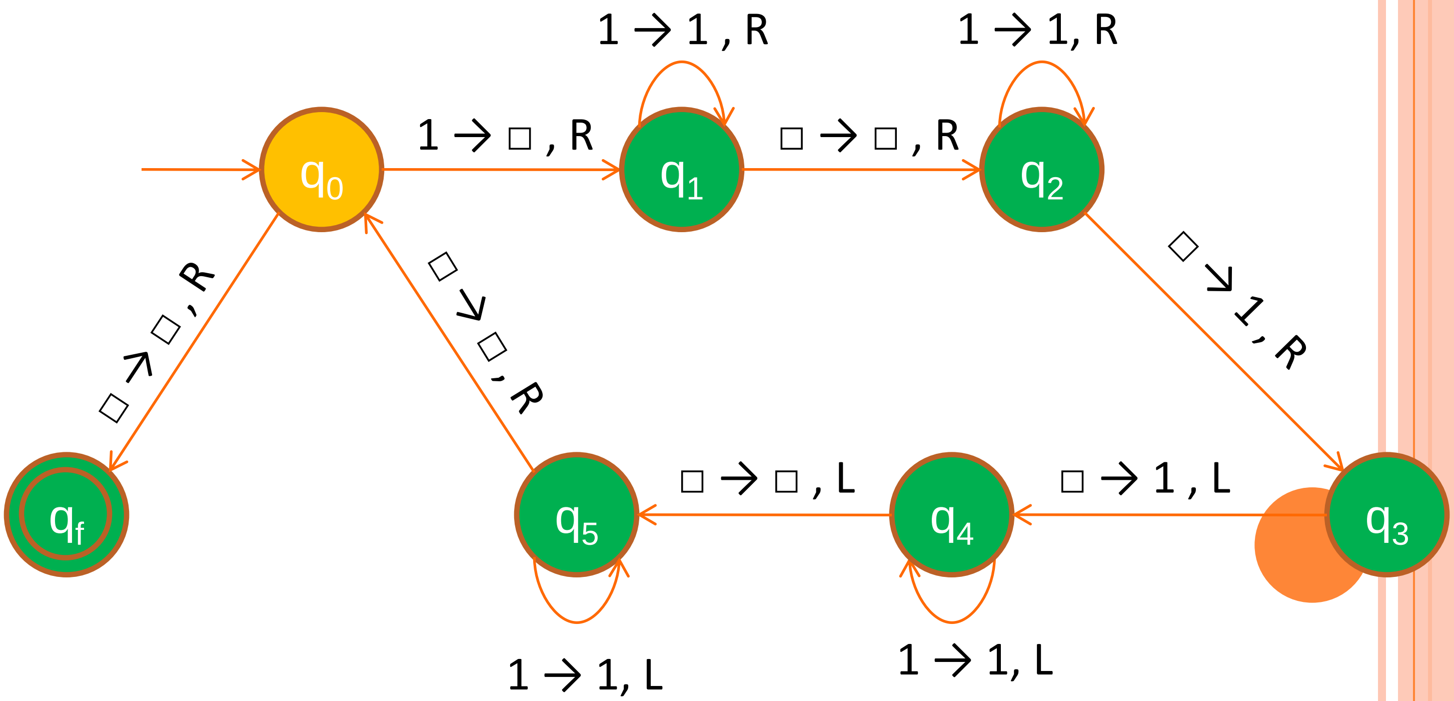




Example: *f*(n) = 2n is computable

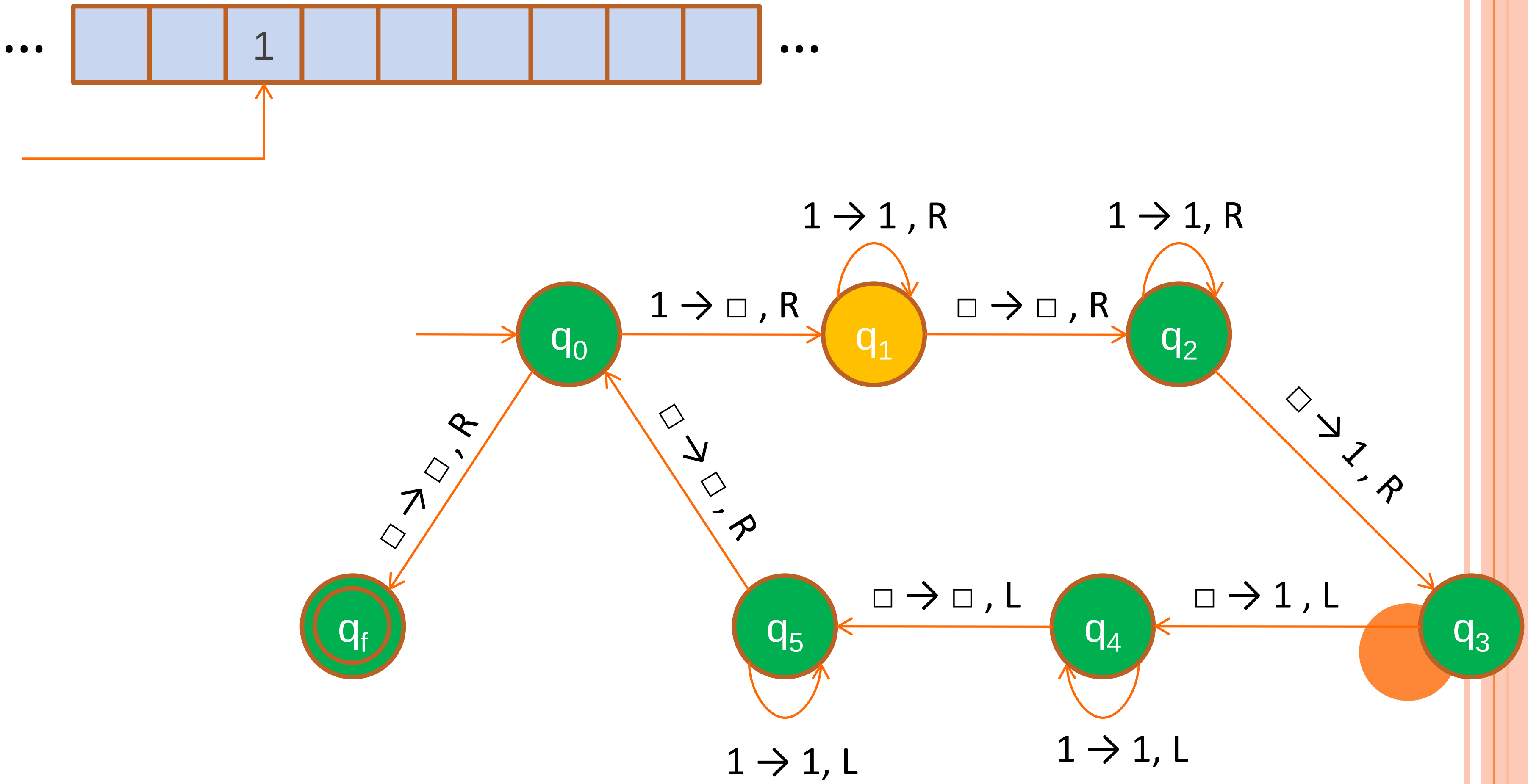
- Test input: 11





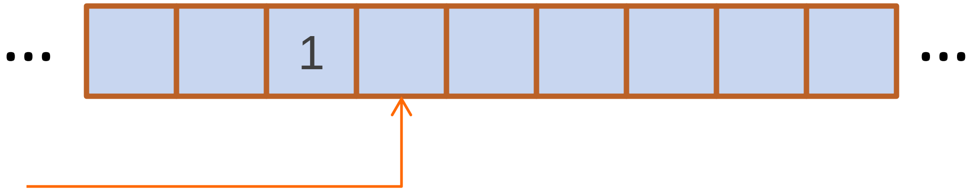
EXAMPLE: *F*(N) = 2N IS COMPUTABLE

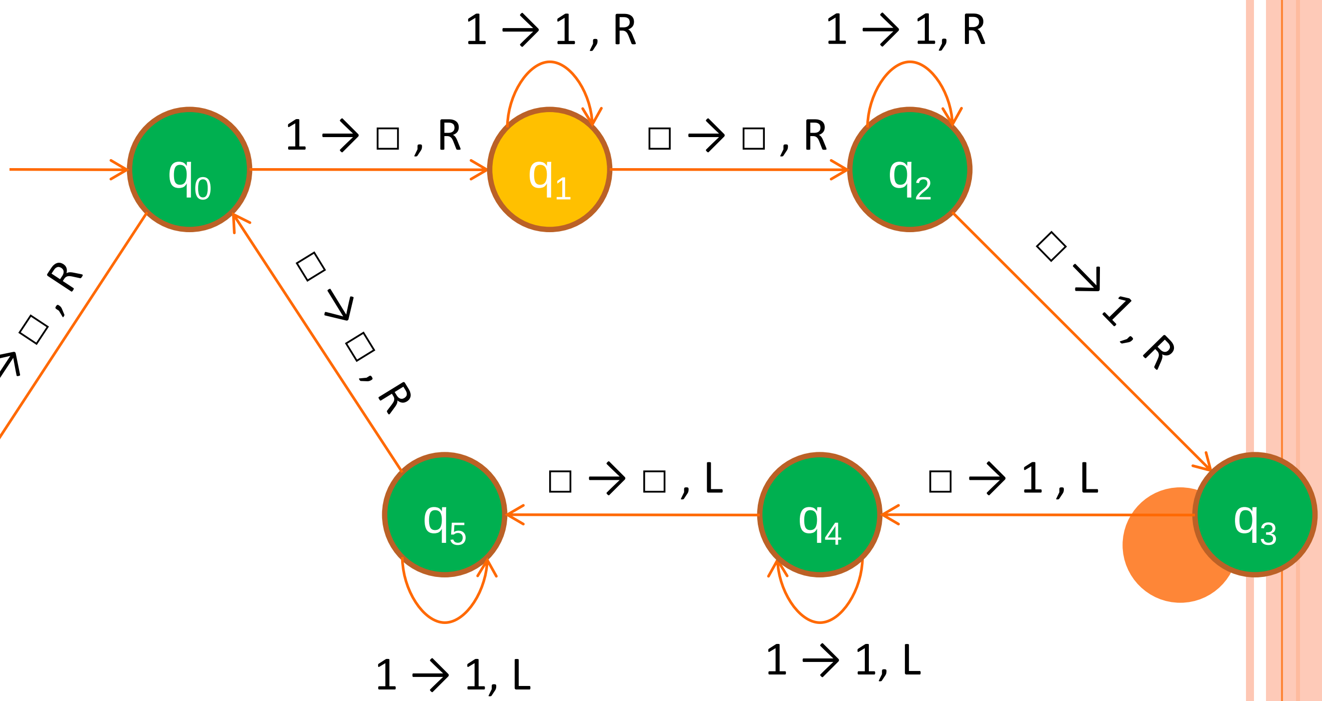
- Test input: 11



EXAMPLE: *F*(N) = 2N IS COMPUTABLE

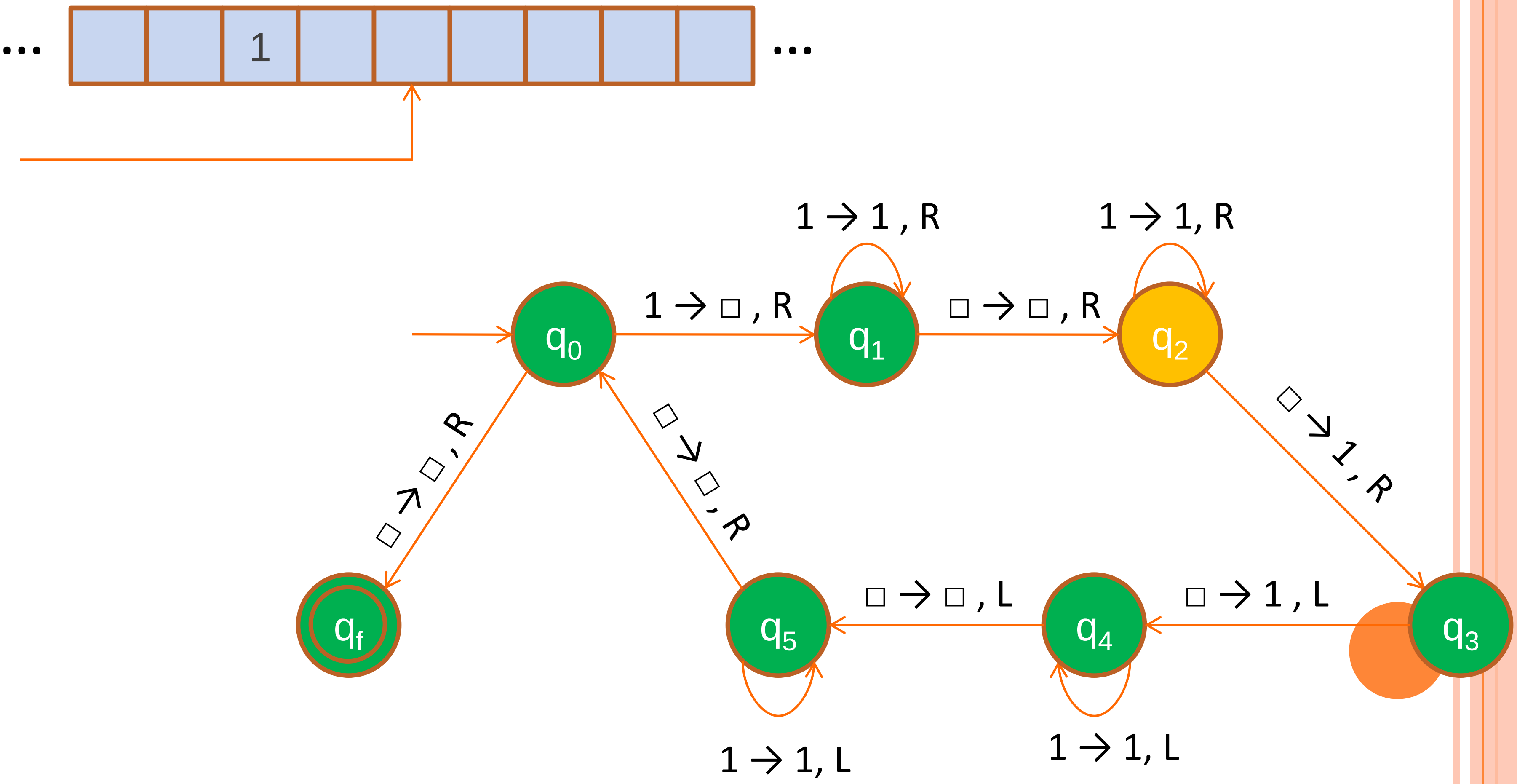
- Test input: 11





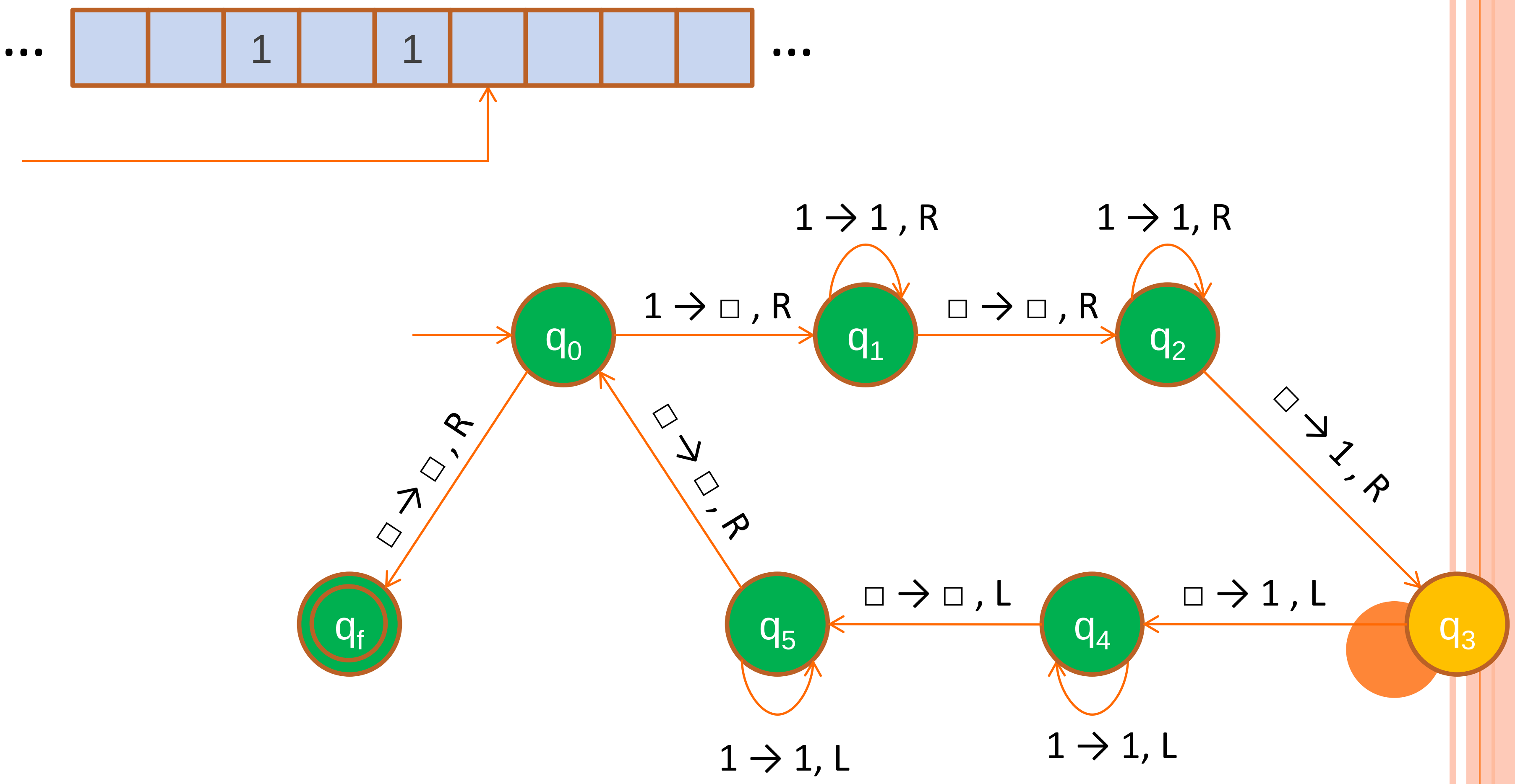
EXAMPLE: *F*(N) = 2N IS COMPUTABLE

- Test input: 11



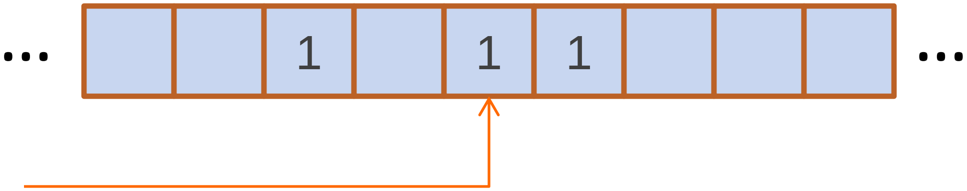
Example: *f*(n) = 2n is computable

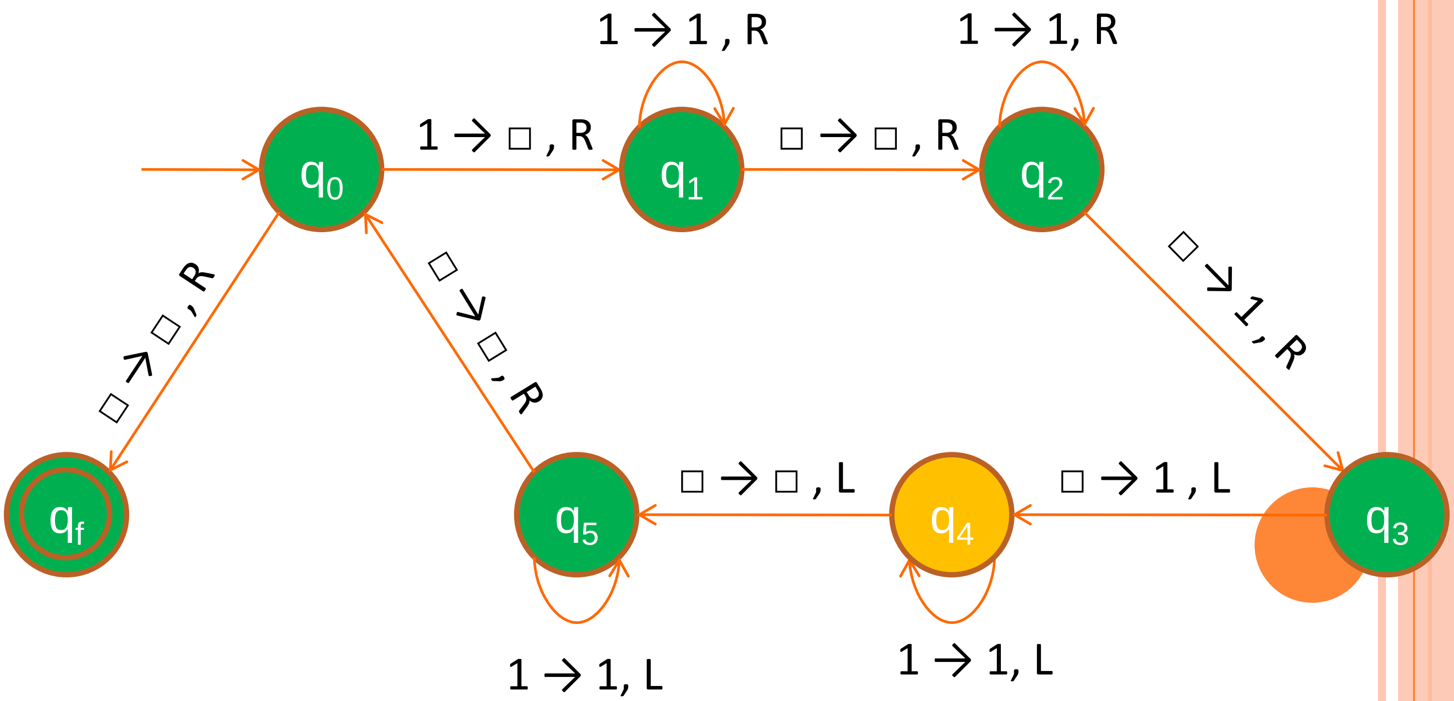
- Test input: 11



Example: *f*(n) = 2n is computable

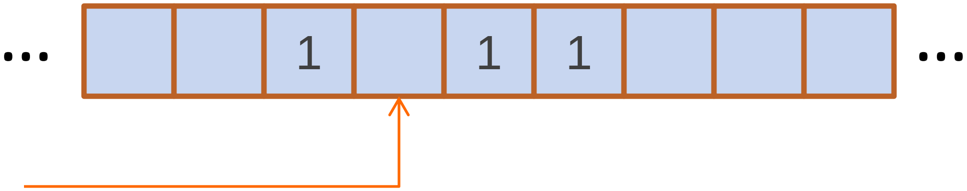
- Test input: 11

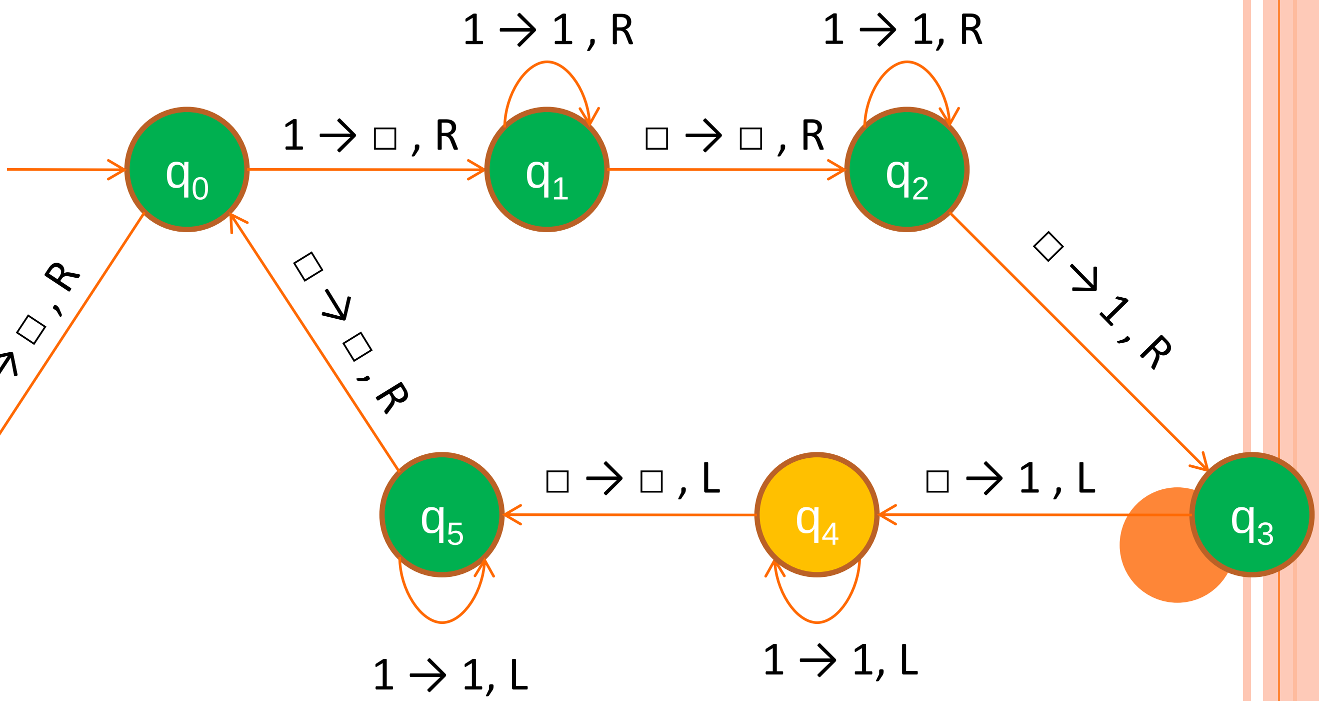




Example: *f*(n) = 2n is computable

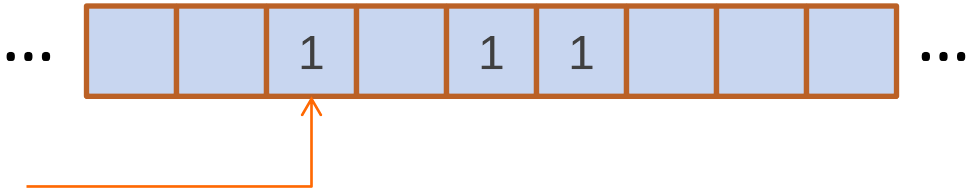
- Test input: 11

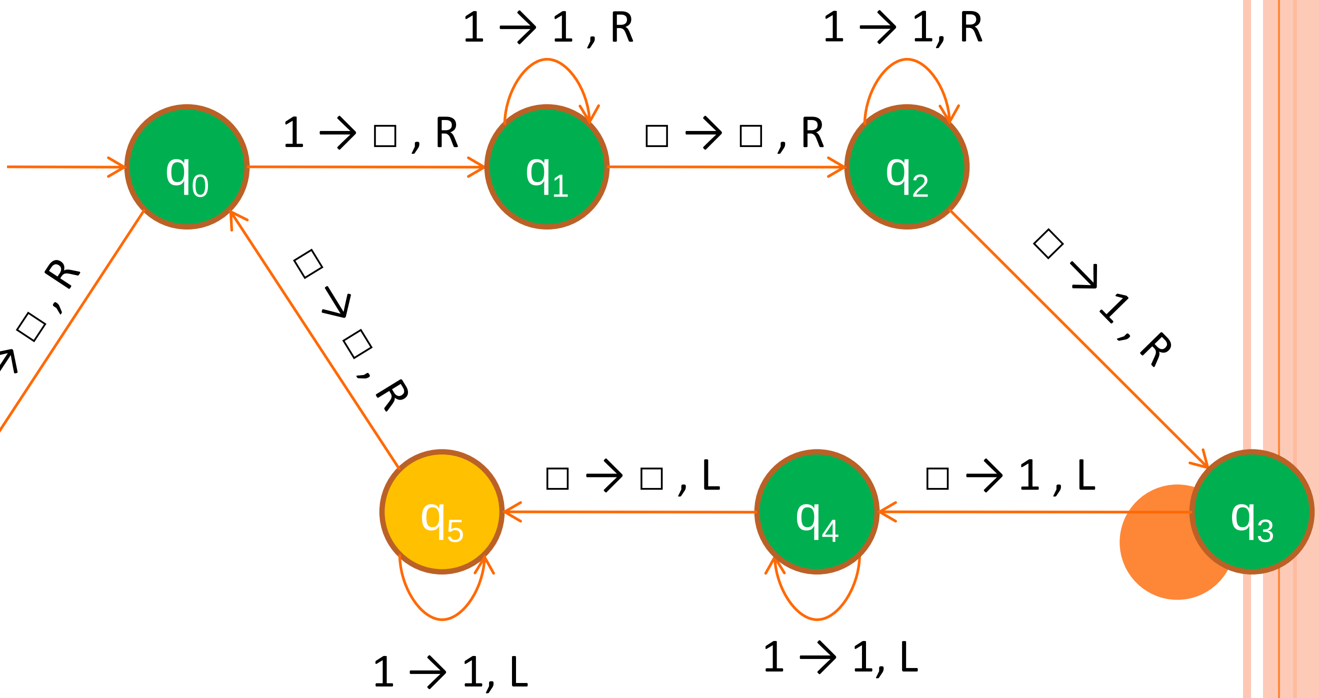




EXAMPLE: *F*(N) = 2N IS COMPUTABLE

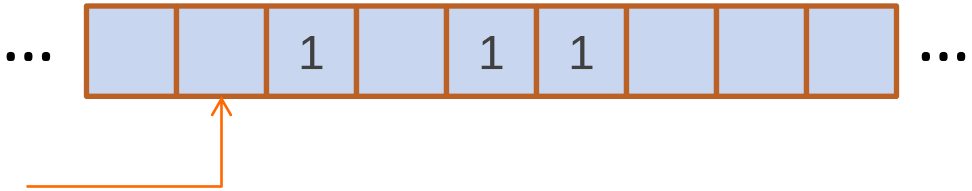
- Test input: 11

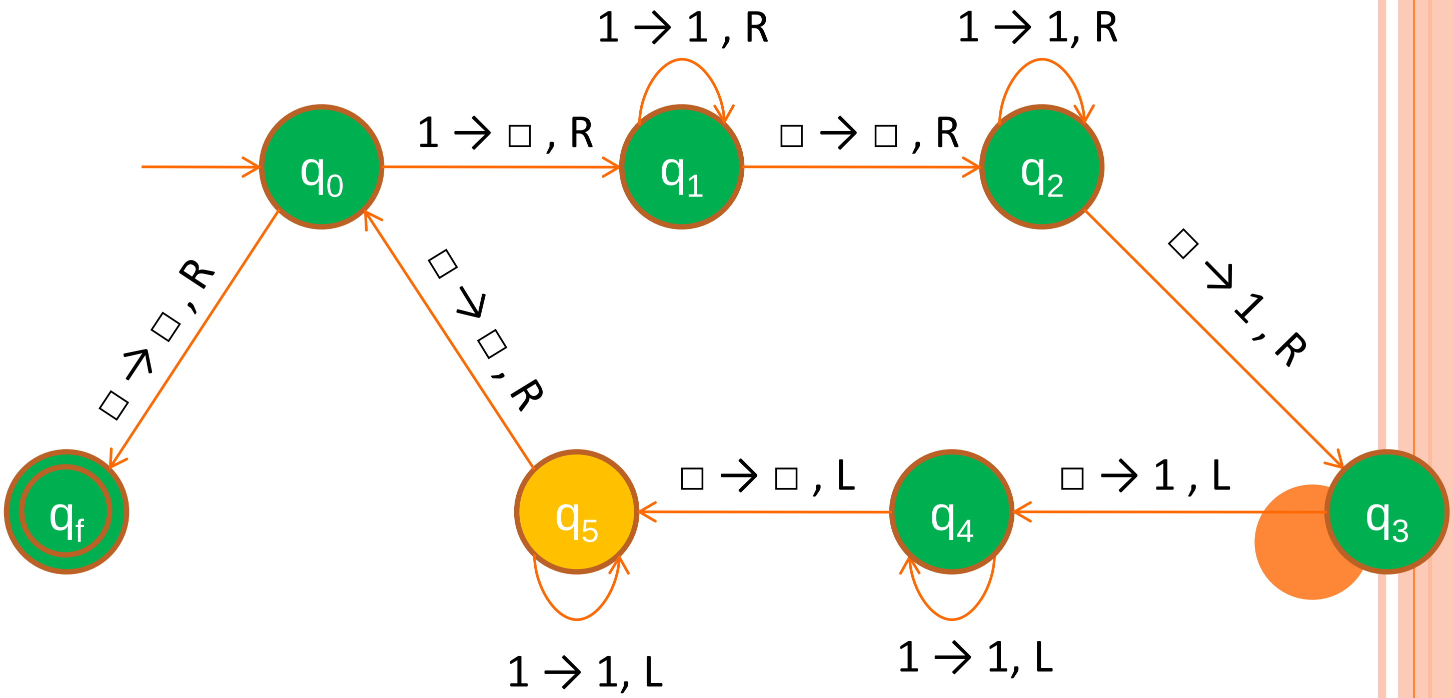




Example: *f*(n) = 2n is computable

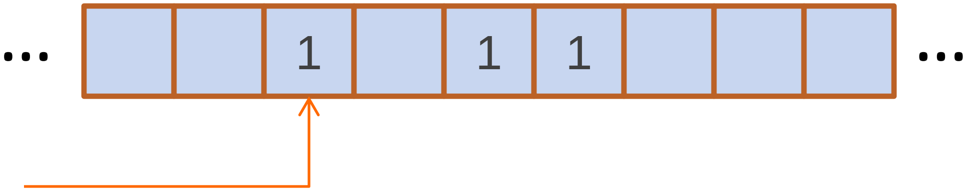
- Test input: 11

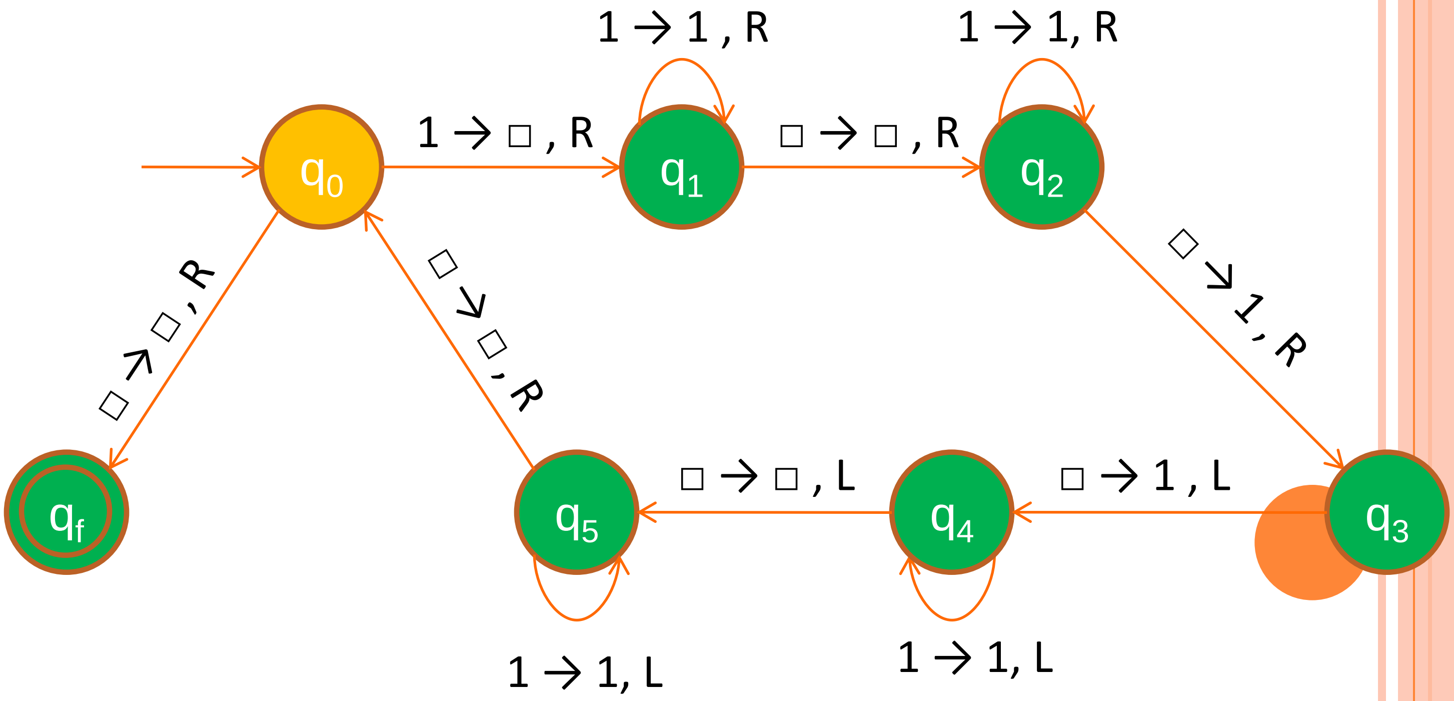




Example: *f*(n) = 2n is computable

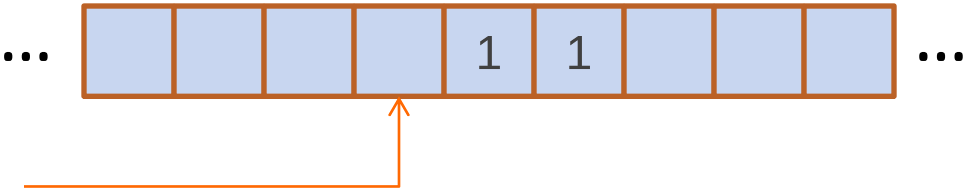
- Test input: 11

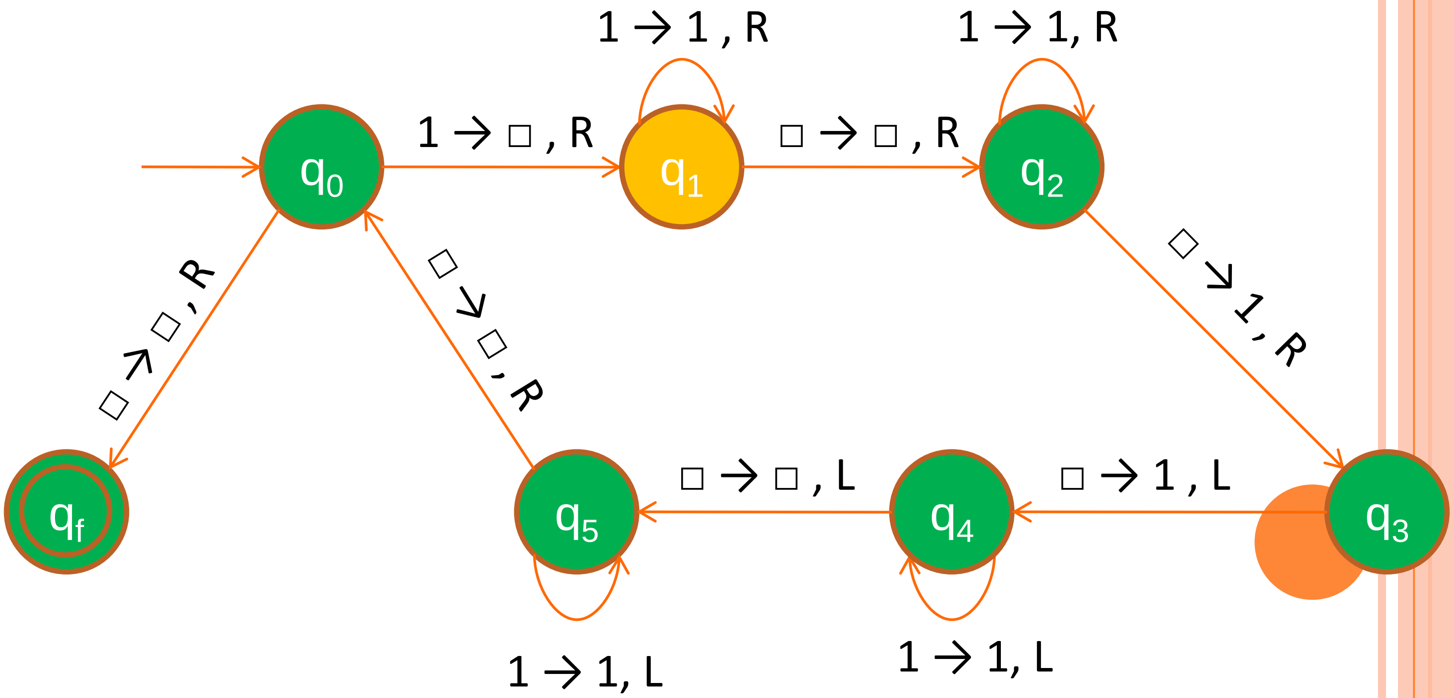




Example: *f*(n) = 2n is computable

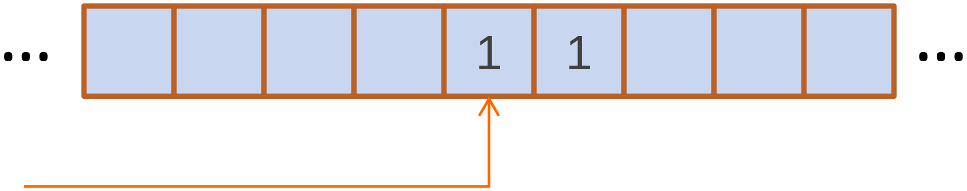
- Test input: 11

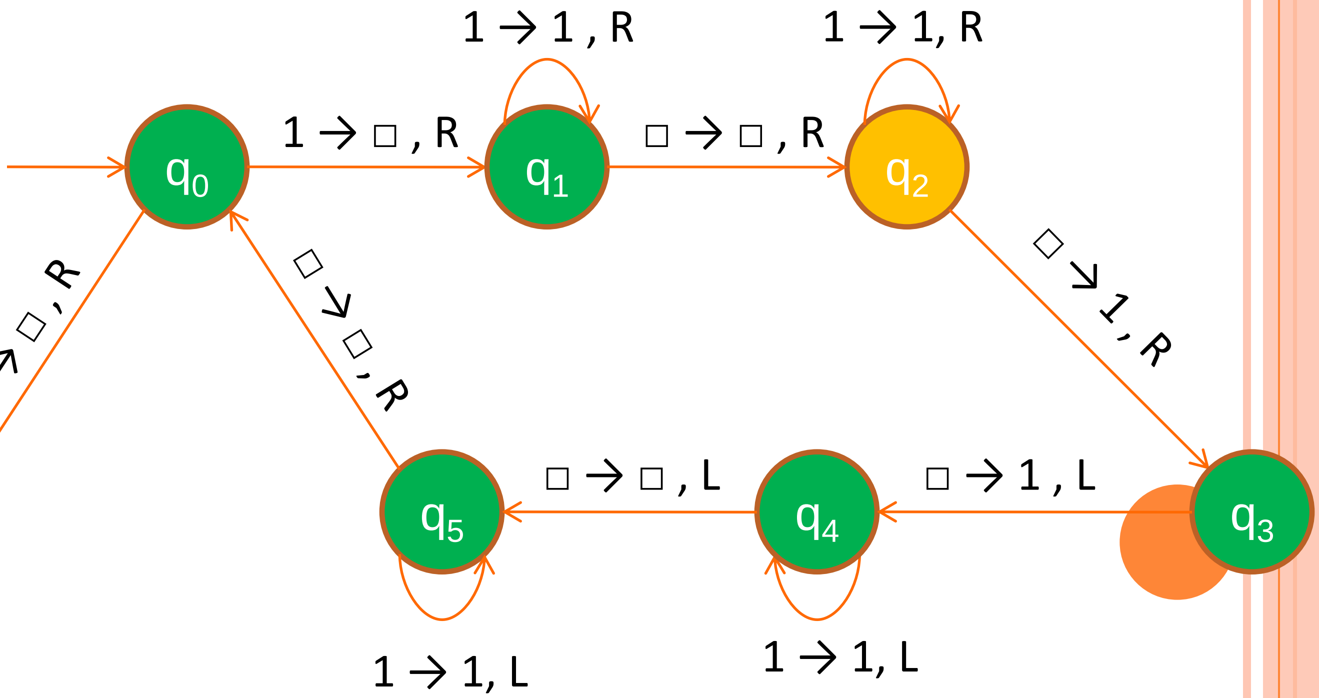




EXAMPLE: *F*(N) = 2N IS COMPUTABLE

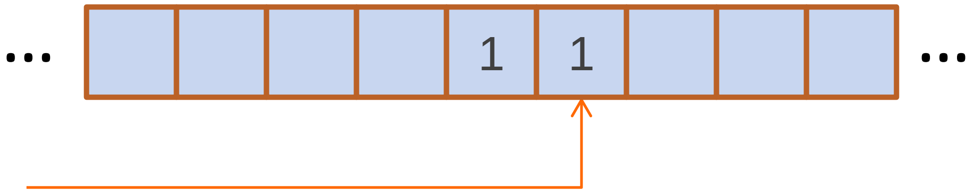
- Test input: 11

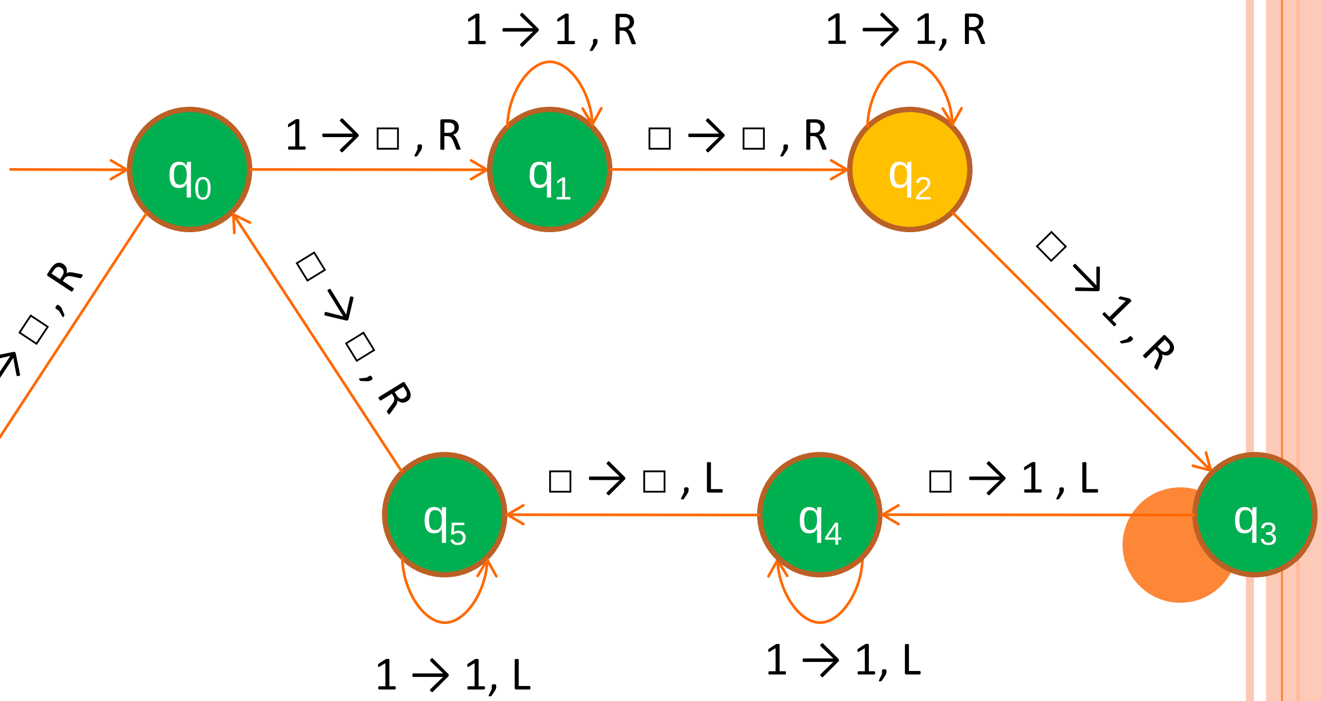




Example: *f*(n) = 2n is computable

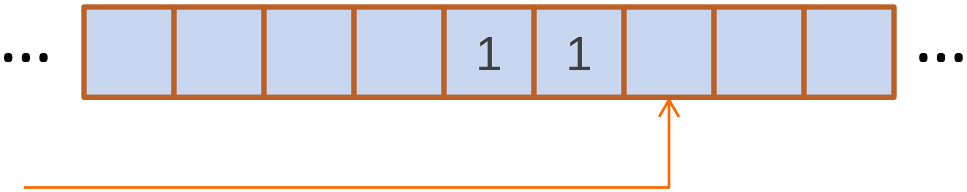
- Test input: 11

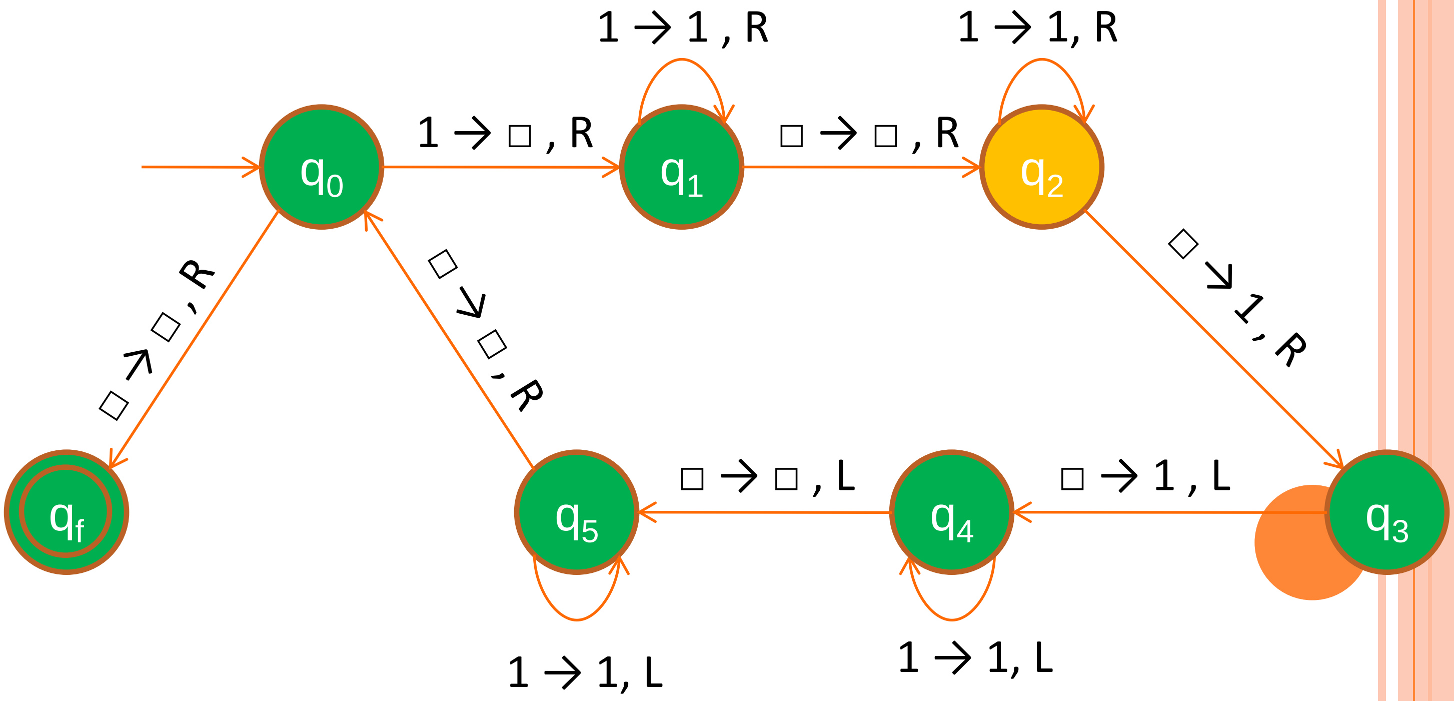




Example: *f*(n) = 2n is computable

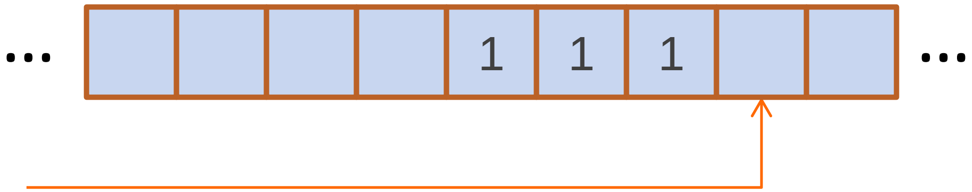
- Test input: 11

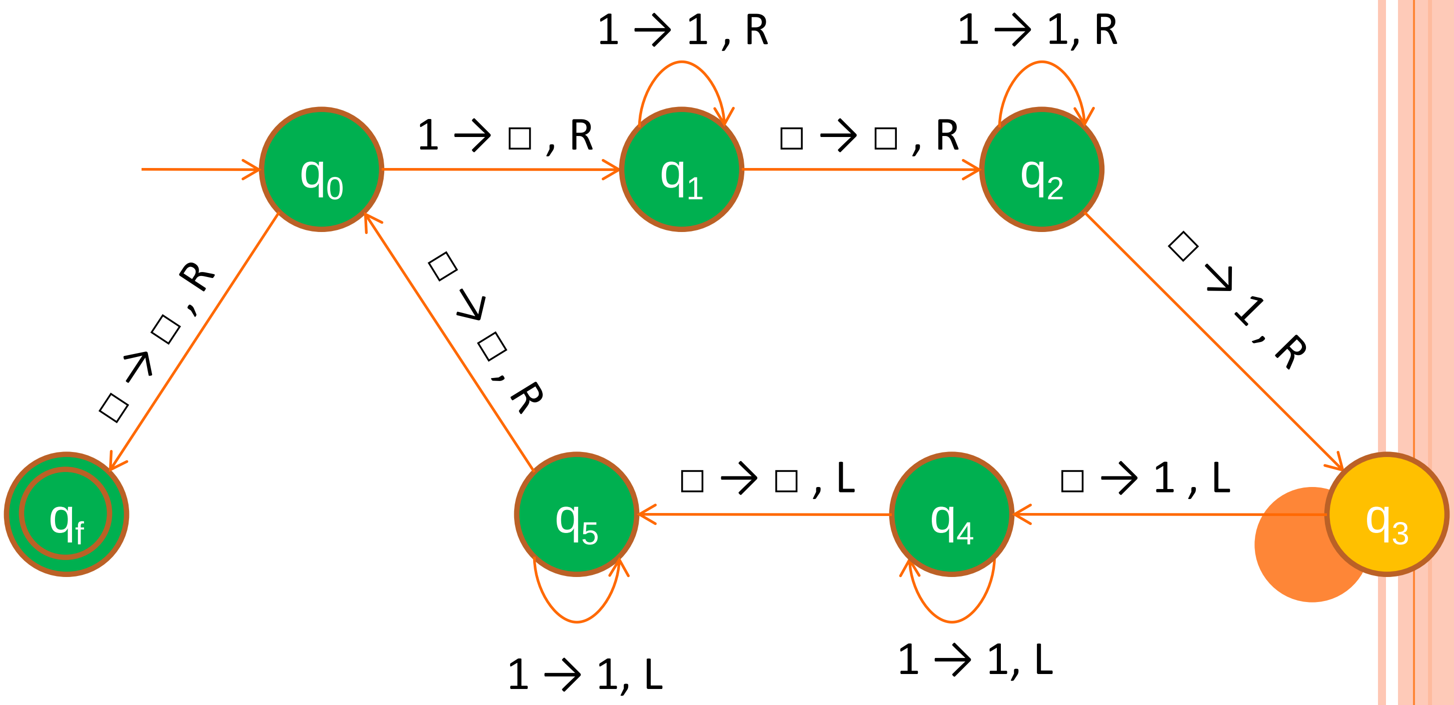




Example: *f*(n) = 2n is computable

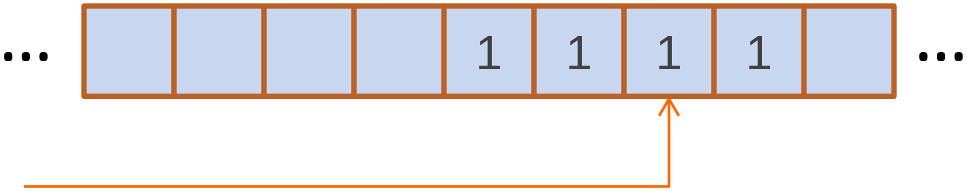
- Test input: 11

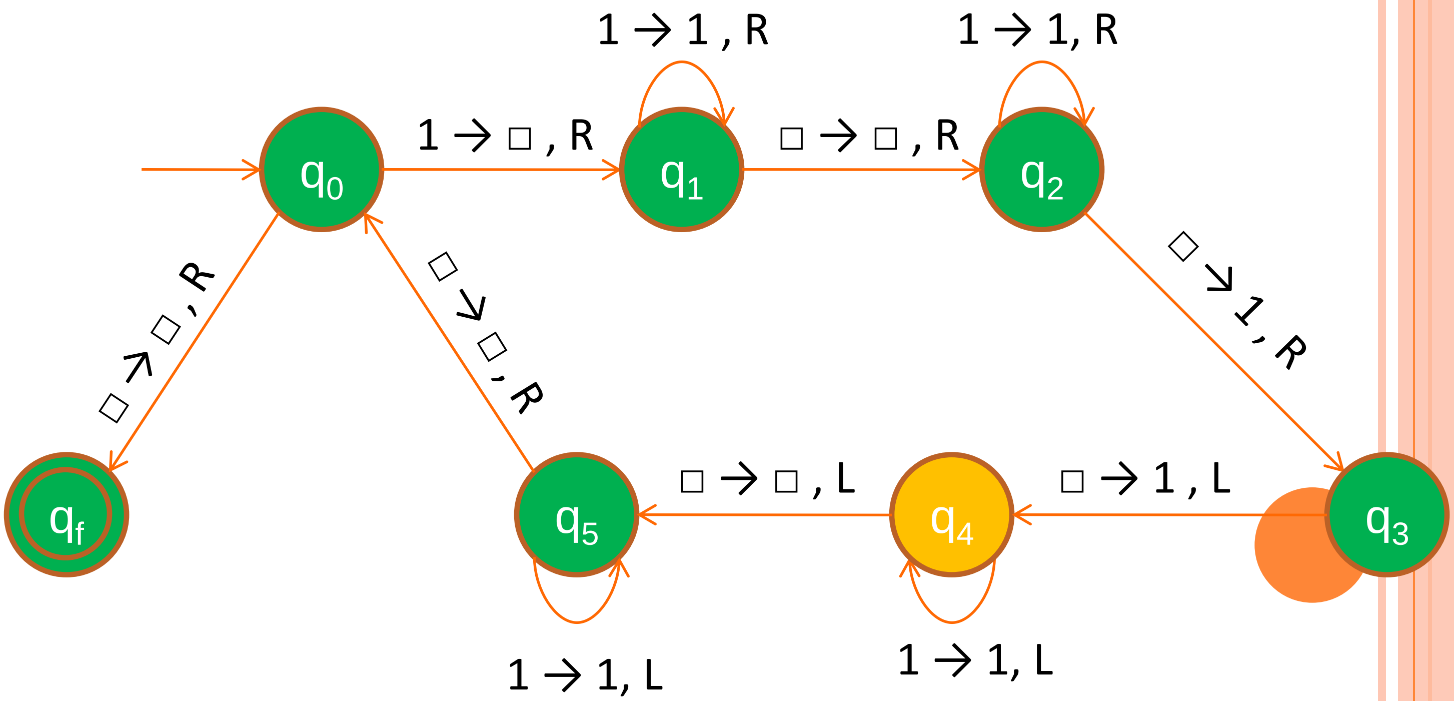




Example: *f*(n) = 2n is computable

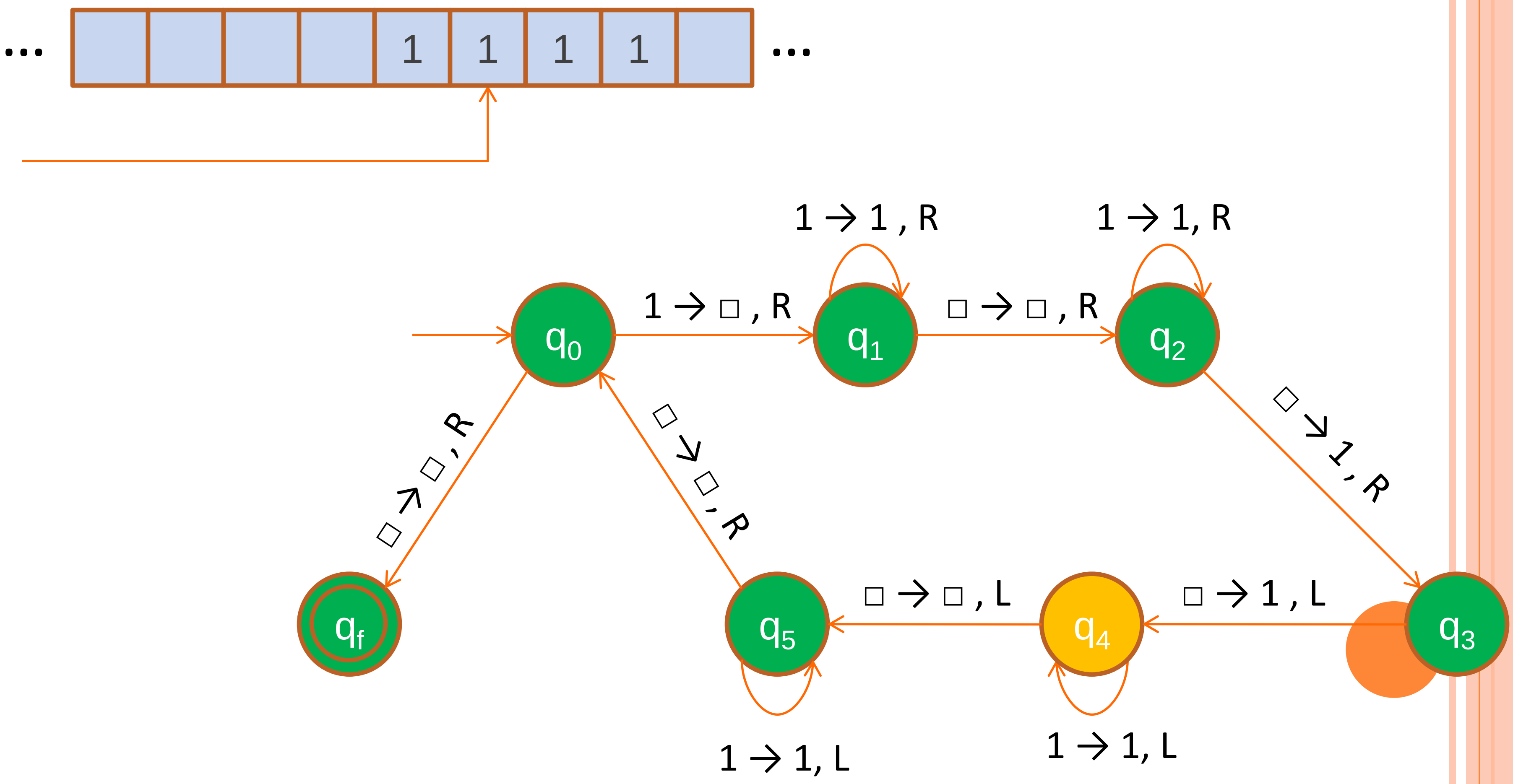
- Test input: 11





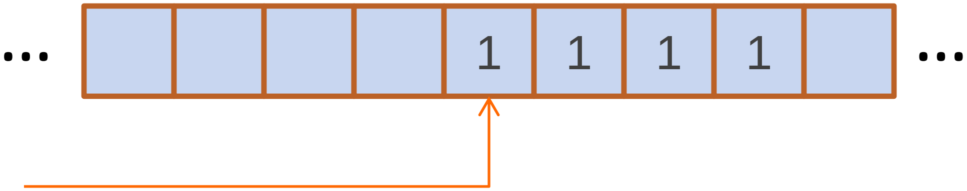
Example: *f*(n) = 2n is computable

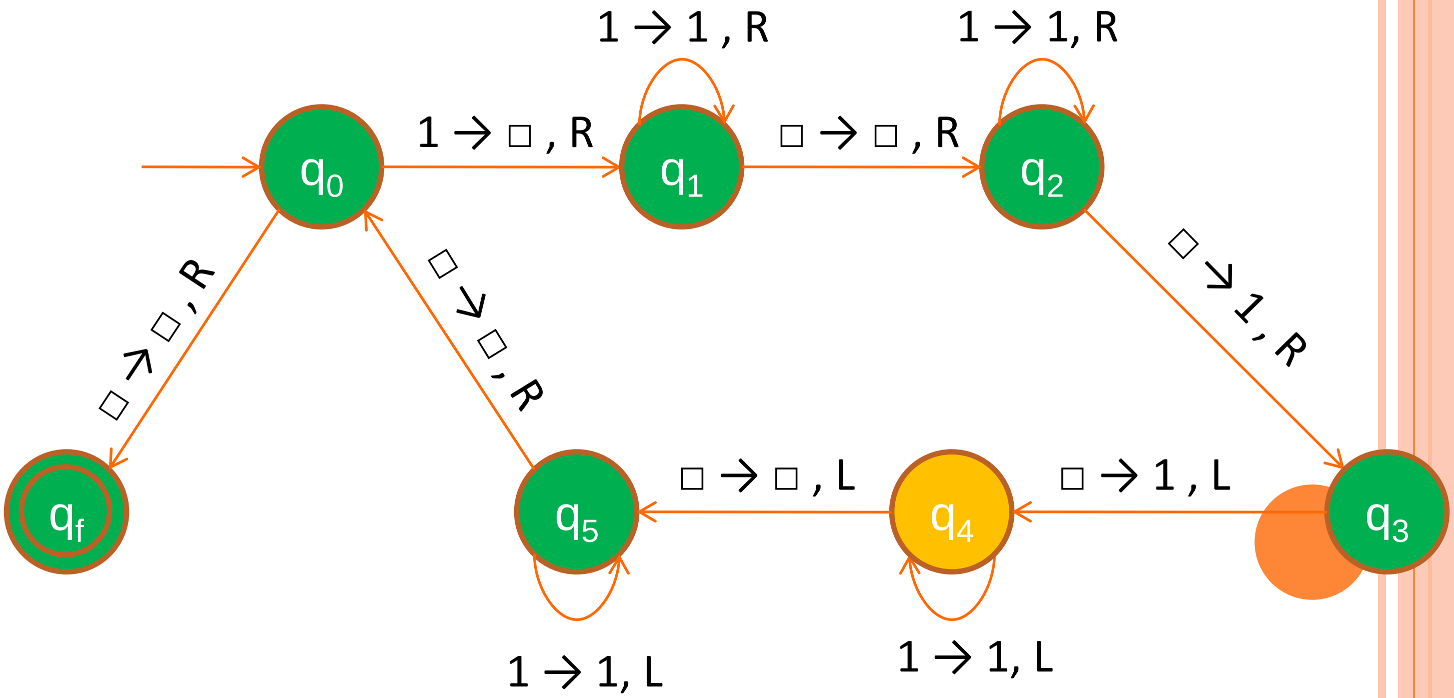
- Test input: 11



Example: *f*(n) = 2n is computable

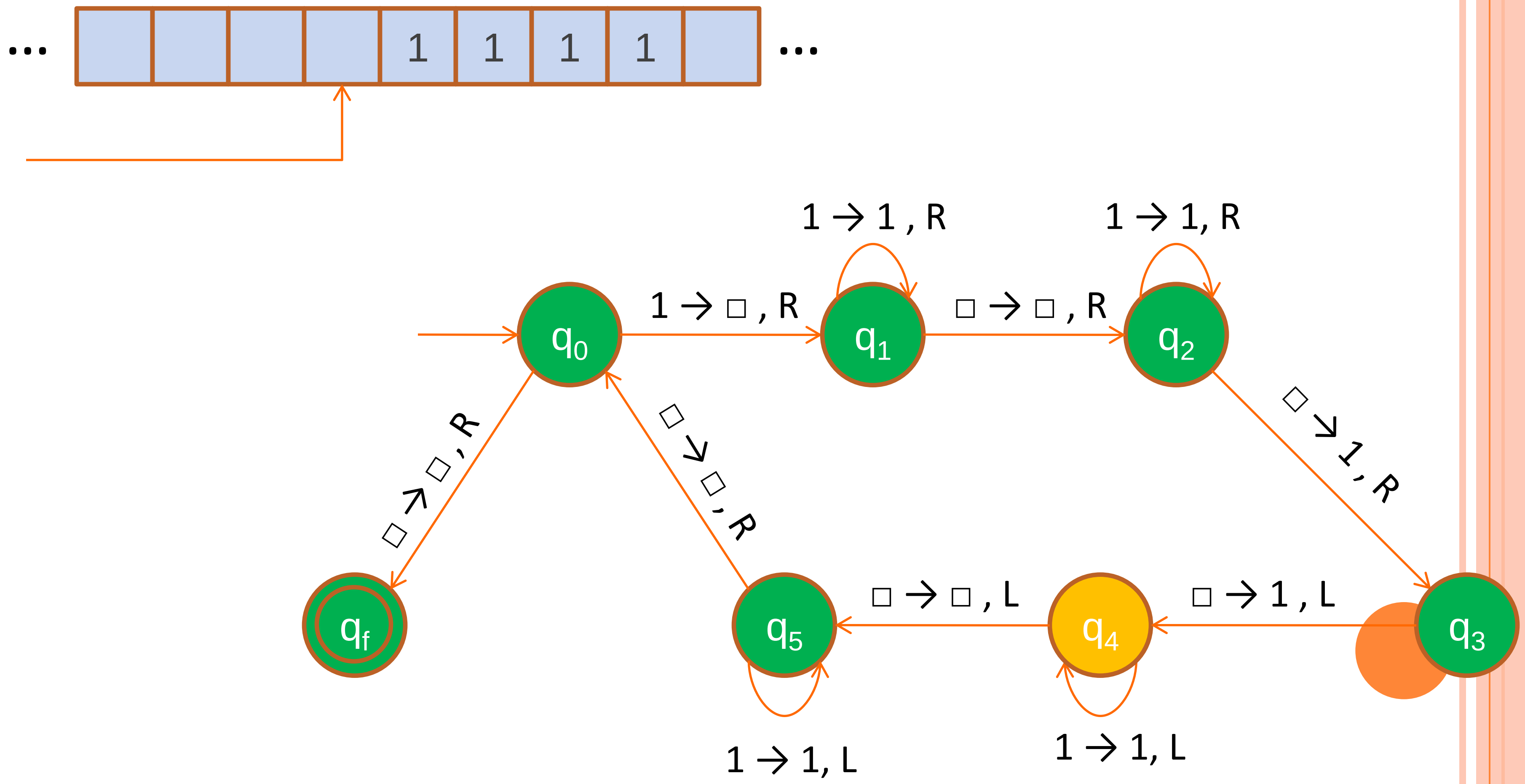
- Test input: 11





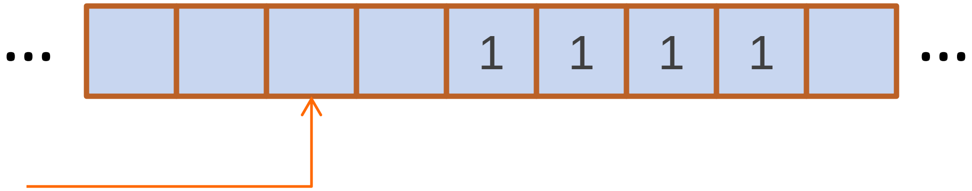
EXAMPLE: *F*(N) = 2N IS COMPUTABLE

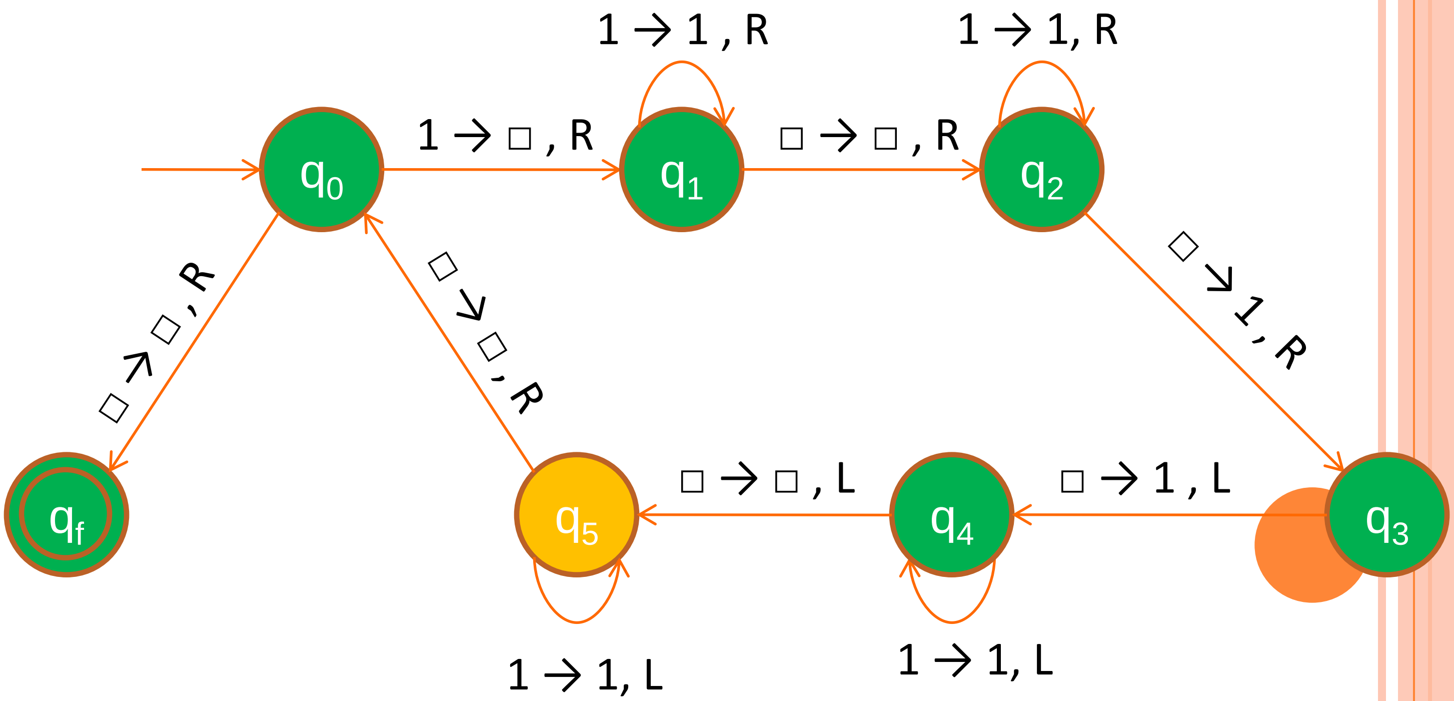
- Test input: 11



EXAMPLE: *F*(N) = 2N IS COMPUTABLE

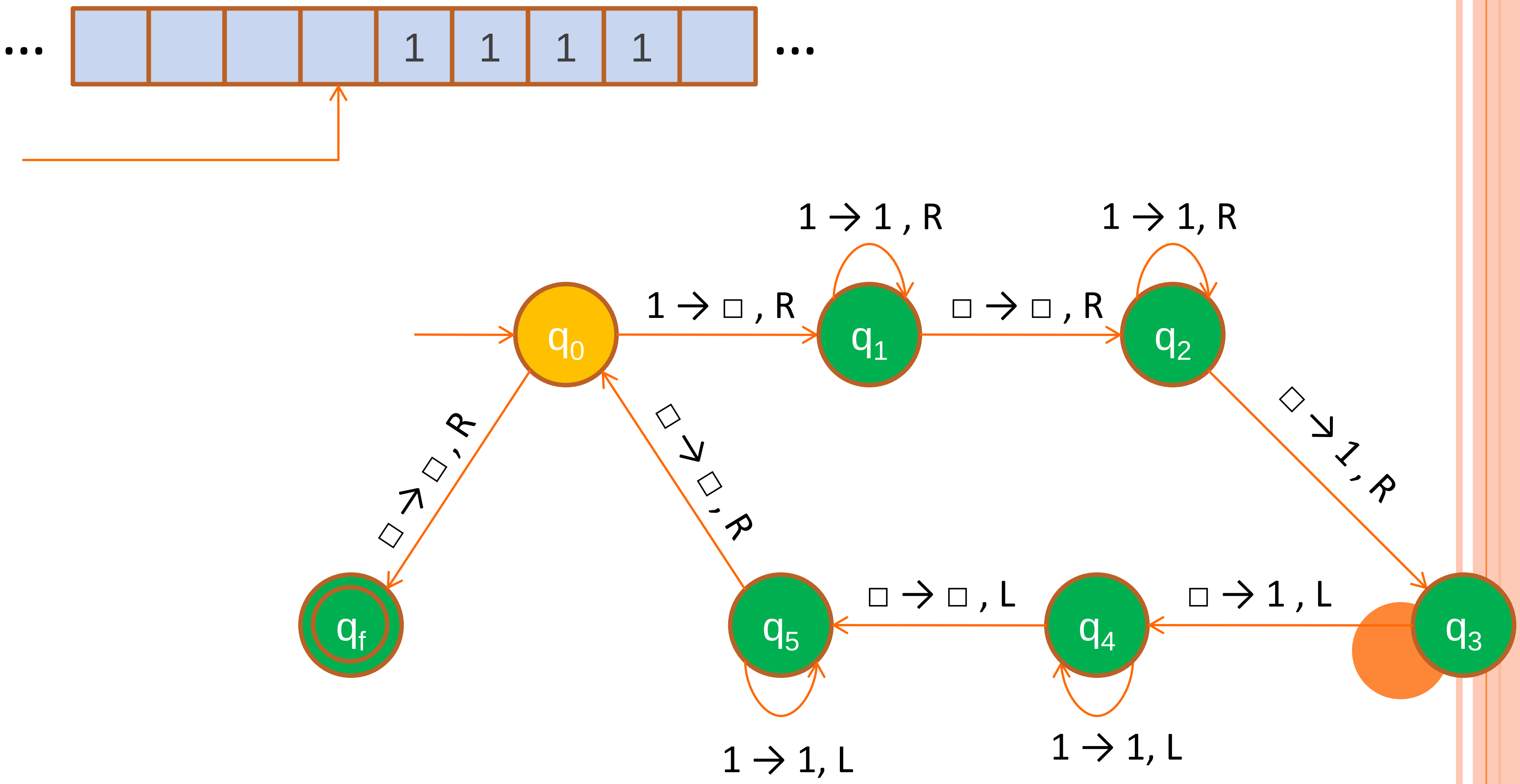
- Test input: 11





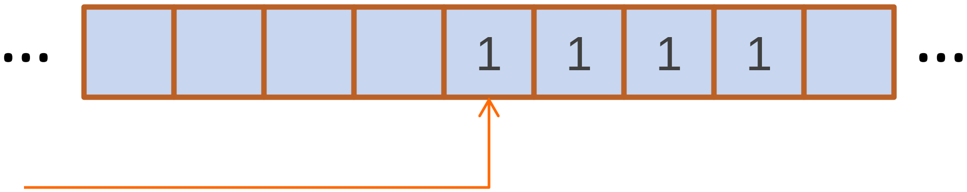
Example: *f*(n) = 2n is computable

- Test input: 11

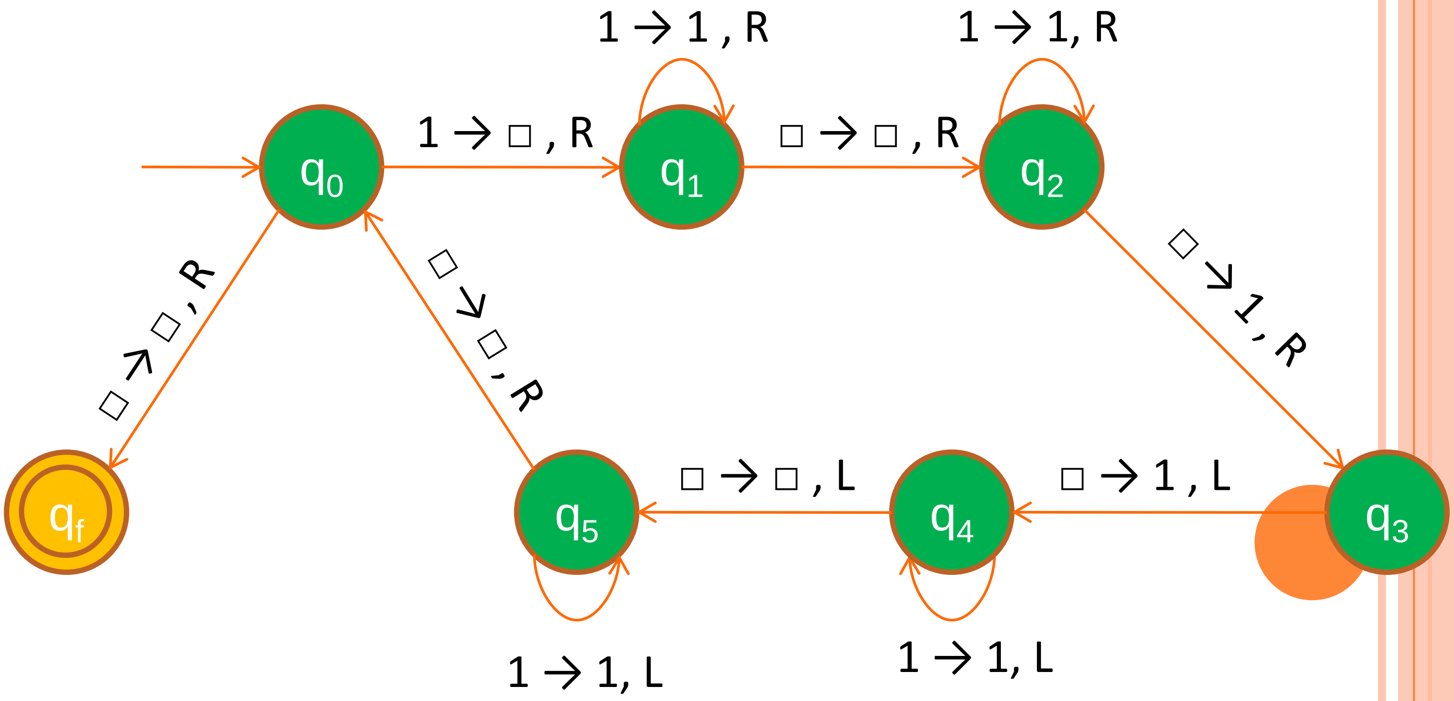


Example: *f*(n) = 2n is computable

- Test input: 11



…					1	1	1	1		…



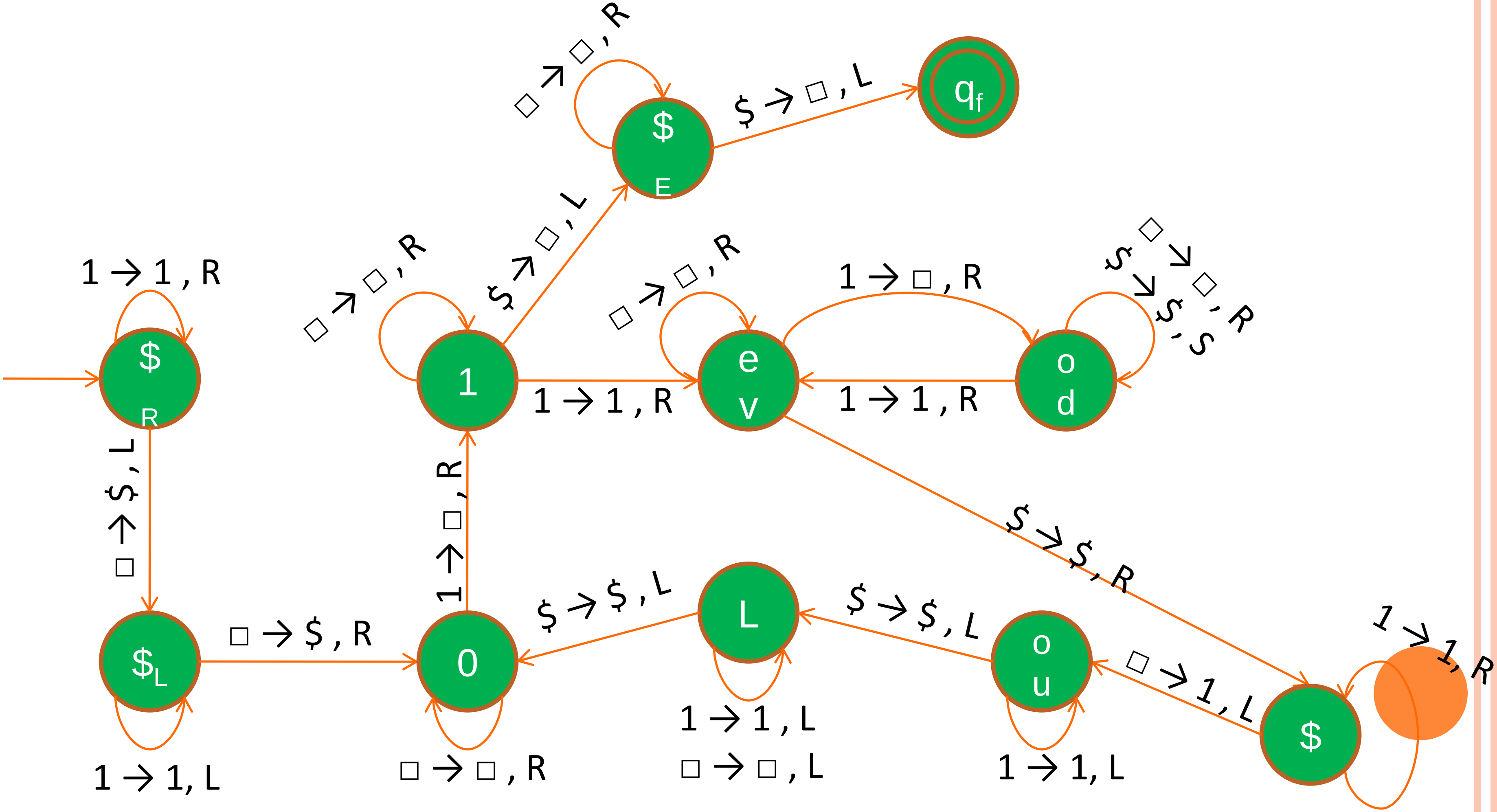
EXAMPLE: *F*(N) = LOGN IS PARTIALLY COMPUTABLE.

We design a TM that partially computes *f*(n).

High Level Program:

- Add a $ in the beginning and the end of the input (this will make the task of going from one state to another easier)
- Repeat:
  - For every two ones in the input erase the first and leave the second there.
  - If there was an odd > 1 number of 1s then loop for ever (the function is undefined)
  - If the number of 1 is even then pass the $ sign
  - Pass along the output until you reach the end (blank).
  - Write one 1 in the output (after the $).
  - Go to the beginning of the input.
- Until there is only one 1 in the input.
- Erase the two $ signs and accept.

Example: *f*(n) = logn is partially computable.



Example: *f*(n) = logn is partially computable.

Try to run the machine by hand

Test inputs:

- ε (shouldn’t accept)
- 1 (should accept)
- 11 (should accept)
- 111 (shouldn’t accept)
- 111111 (shouldn’t accept)
- 11111111 (should accept)

K-ARY FUNCTIONS

- A function *f* might have more than one parameters.
- *f*: N x N x … x N → N is called k-ary function

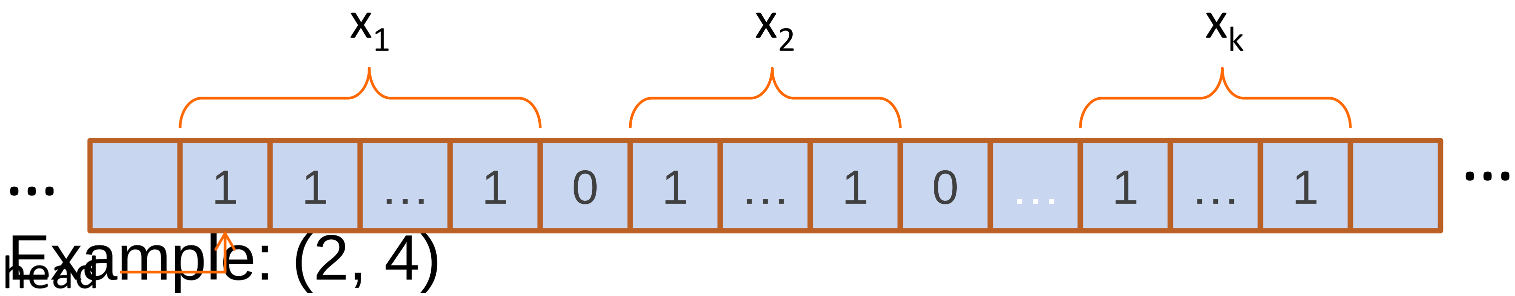
Examples:

$\underbrace{\hspace{10em}}_{\text{k times}}$

- + : N x N → N (addition of integers) is a binary function.

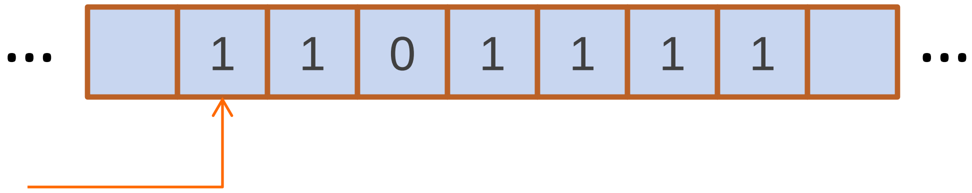
Representation of a k-tuple in a DTM

A k-tuple (x_1 , x_2 , …, x_k) can be represented in a Turing Machine by the following way:



…		1	1	…	1	0	1	…	1	0	…	1	…	1		…

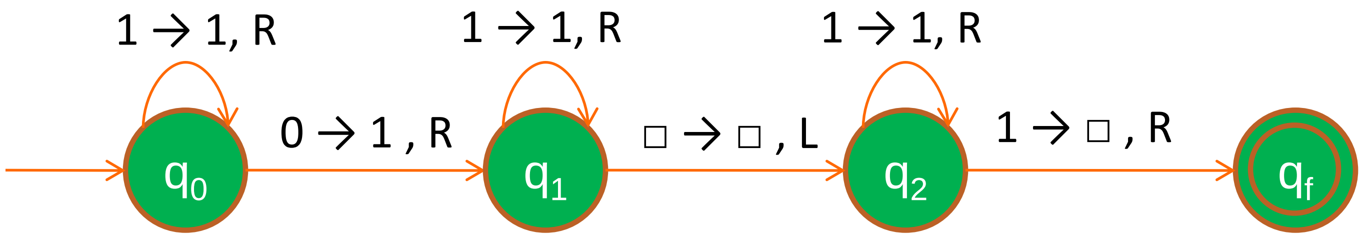
- Example: (2, 4)



…		1	1	0	1	1	1	1		…

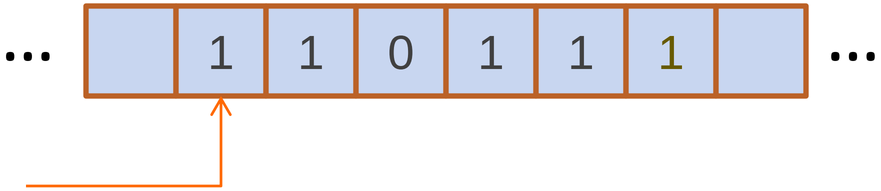
BINARY ADDITION IS COMPUTABLE

- High level program:
  - Remove the 0 from the middle and make the 1s in the input consecutive.
- TM for this function:

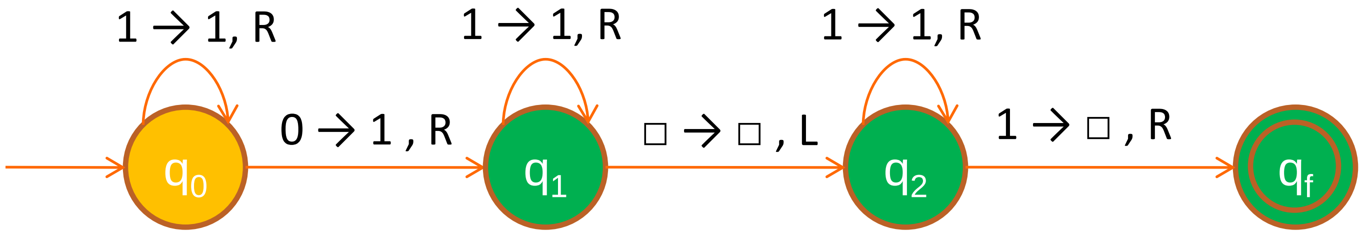


BINARY ADDITION IS COMPUTABLE

- Test input: (2, 3) = 110111

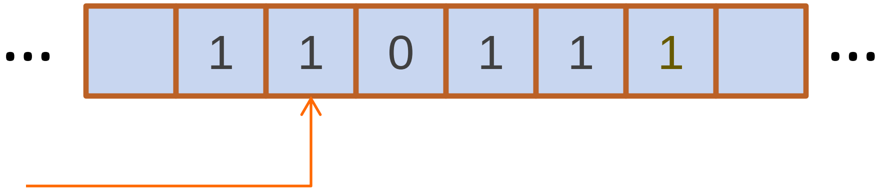


…		1	1	0	1	1	1		…

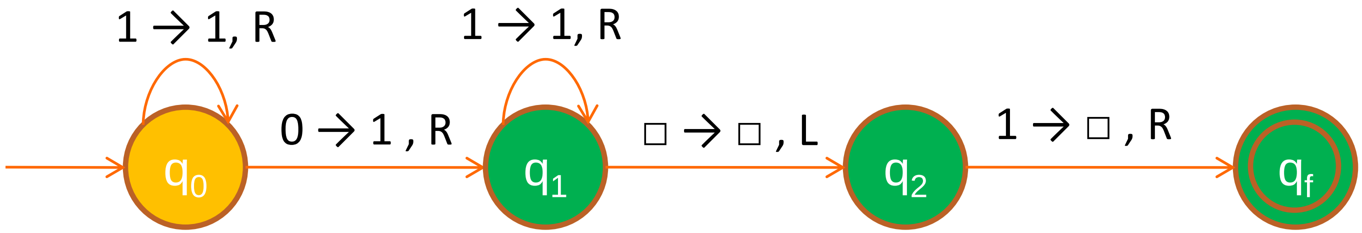


Binary addition is computable

- Test input: (2, 3) = 110111

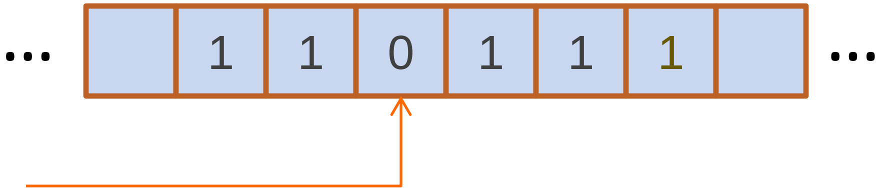


…		1	1	0	1	1	1		…

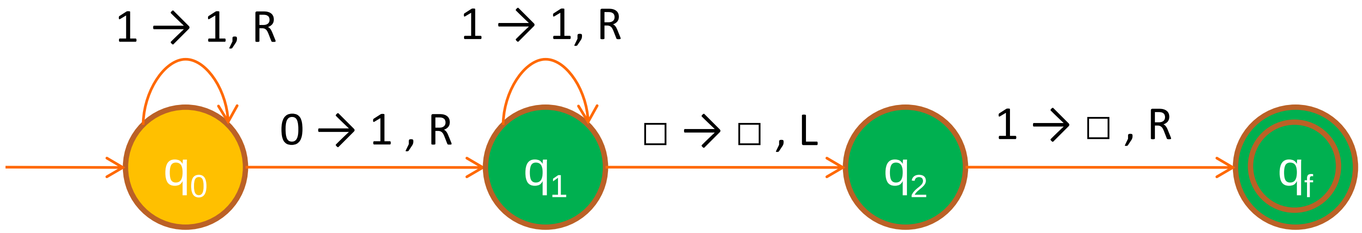


Binary addition is computable

- Test input: (2, 3) = 110111

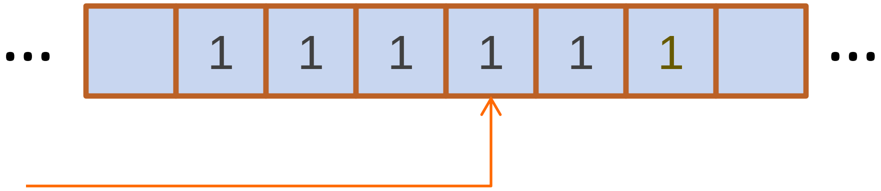


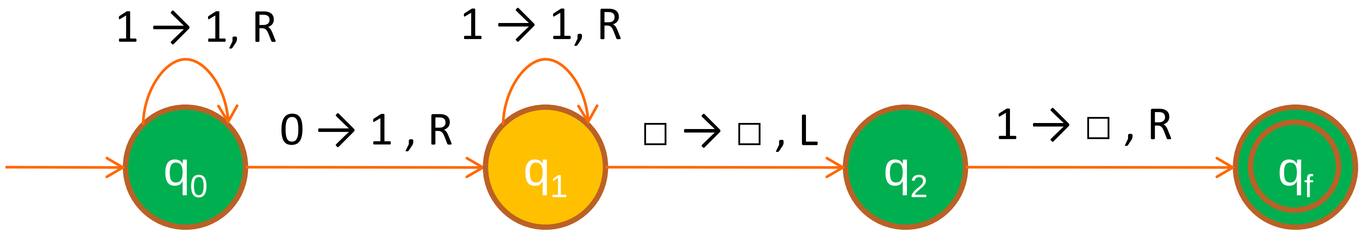
…		1	1	0	1	1	1		…



Binary addition is computable

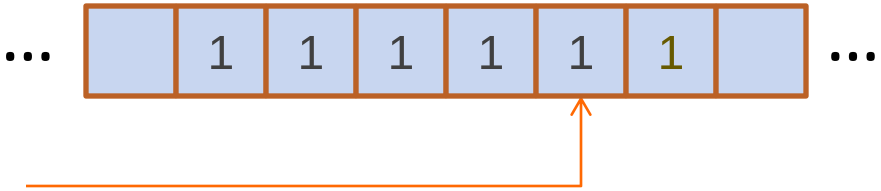
- Test input: (2, 3) = 110111

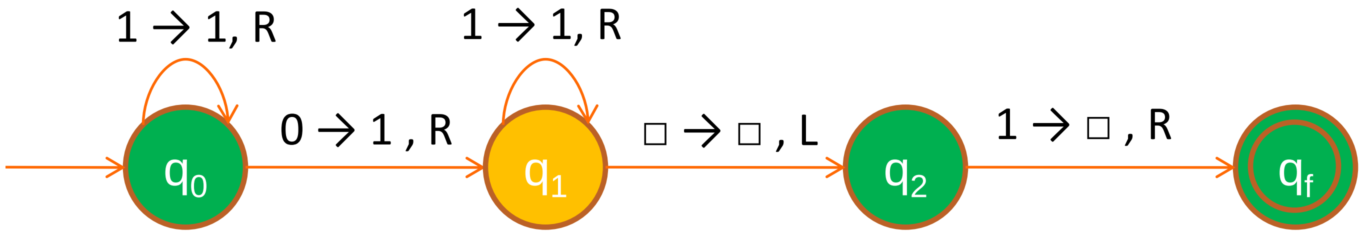




Binary addition is computable

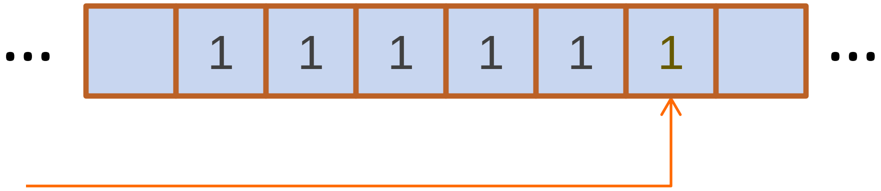
- Test input: (2, 3) = 110111

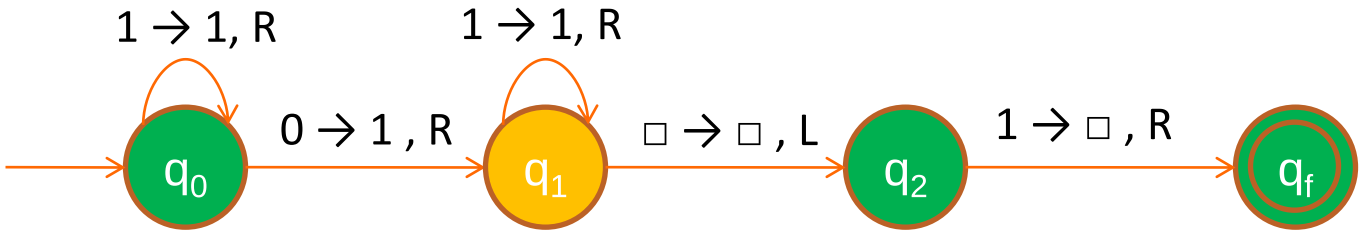




BINARY ADDITION IS COMPUTABLE

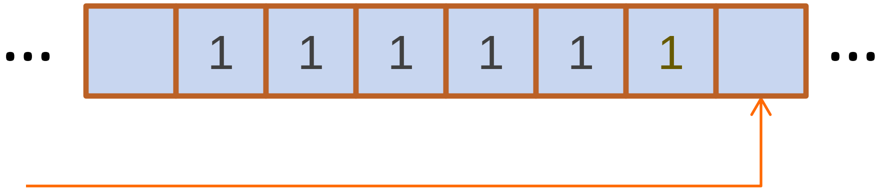
- Test input: (2, 3) = 110111

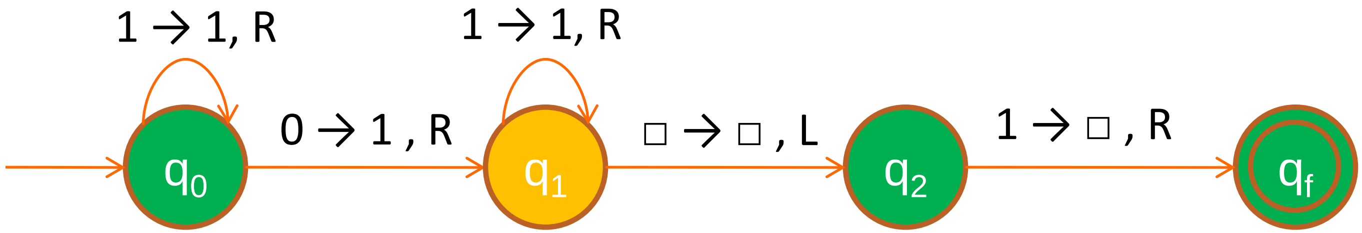




BINARY ADDITION IS COMPUTABLE

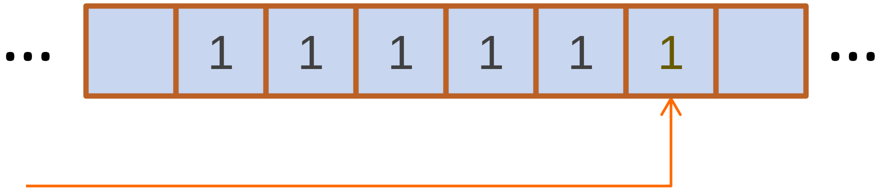
- Test input: (2, 3) = 110111

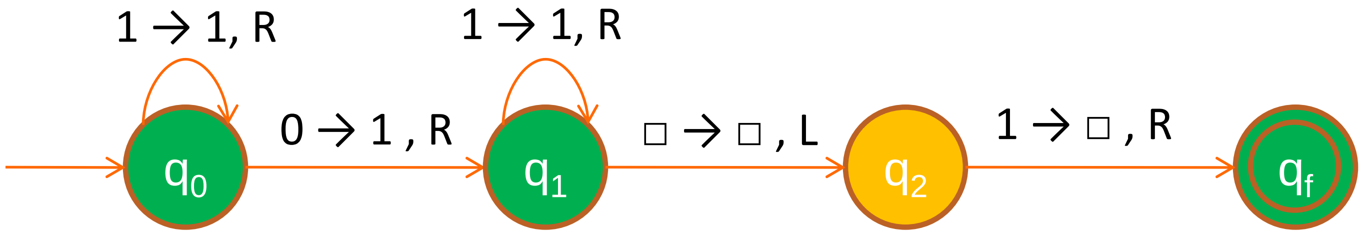




BINARY ADDITION IS COMPUTABLE

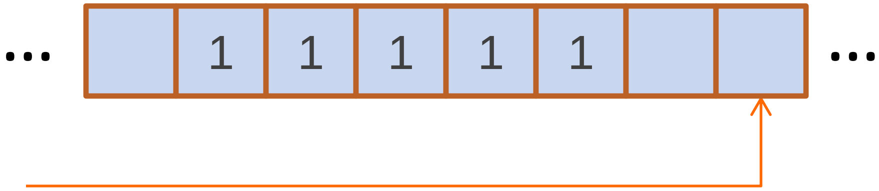
- Test input: (2, 3) = 110111

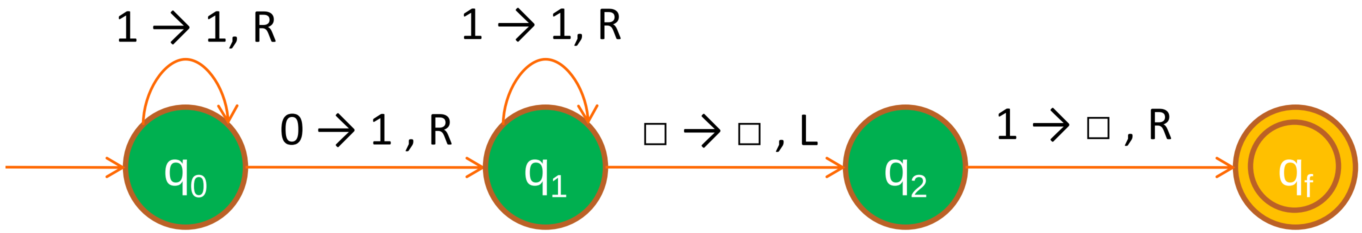




Binary addition is computable

- Test input: (2, 3) = 110111





PREDICATES

- Predicate: A Boolean (yes-no) function.
- $P : \mathrm{N} \longrightarrow \{0,1\}$
- Examples

  Unary predicate

  k-ary predicate

$$P : N \to \{0,1\}, P(n) = \begin{cases} 1, n = 0 \\ 0, else \end{cases}$$

- Partial Predicates: Predicates with unique value that are not defined for some input.

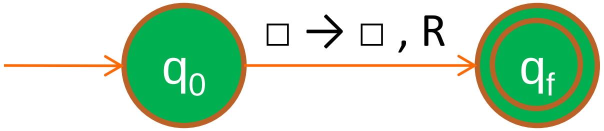
$$Q : N \times N \to \{0,1\}, Q(x, y) = \begin{cases} 1, x < y \\ 0, else \end{cases}$$

Computable Predicates

- A predicate *P* is computable if it can be computed by a Turing Machine
  - a) There is a Turing Machine that, given its parameters n_1, …, n_k as input it outputs *P*(n_1, …, n_k)
  - b) There is a Turing Machine that decides *P* (in other words accepts if the output is 1 and rejects if the output is 0.
- The 2 definitions are equivalent. We use the second one.

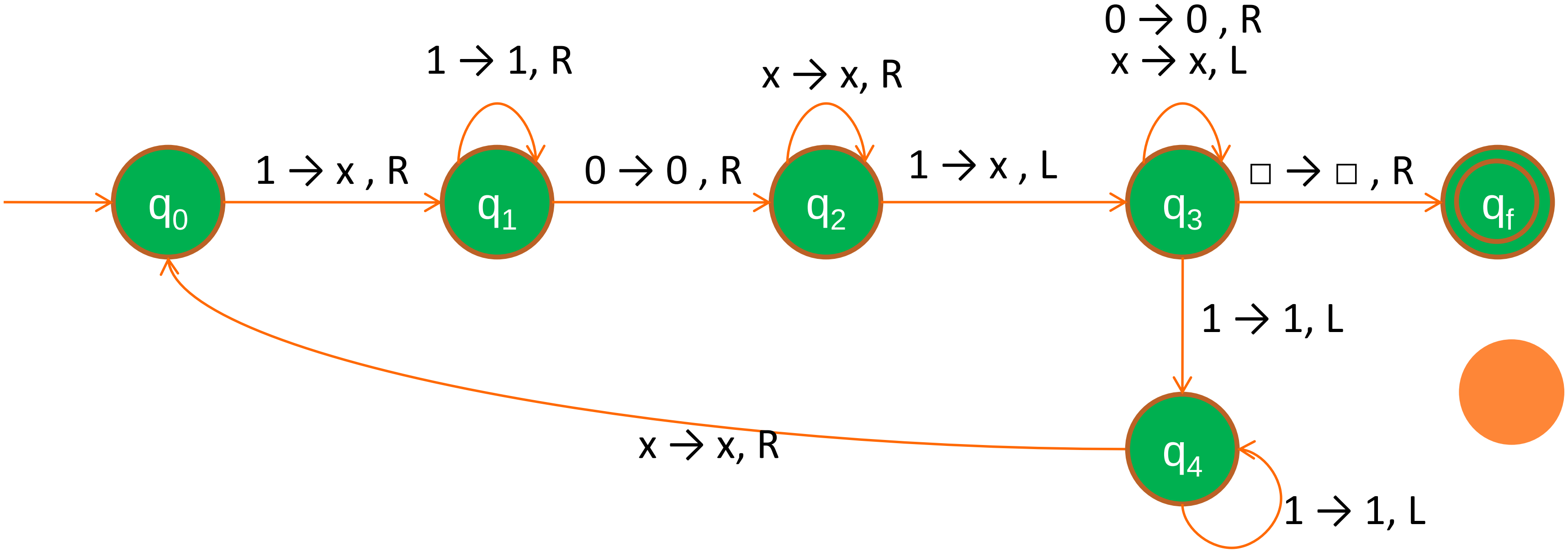
P(*N*) IS COMPUTABLE

- High level program:
  - If the tape is empty accept else reject
- Turing Machine:



Q(*x*, *y*) IS COMPUTABLE

- High level program:
  - Remove one 1 from both *x* and *y*. If you can do that for all the 1s in *x* then accept, else reject.
- TM for this predicate:



Computable Sets, Characteristic function

- The characteristic function χ_A of a set *A* is defined as follows:

$$\chi_A(x) = \begin{cases} 1, x \in A \\ 0, x \notin A \end{cases}$$

- A set is computable if its characteristic function is computable.

COMPUTABLE LANGUAGES

- A language is computable if its characteristic function is computable (we can use a Turing Machine to decide membership)
- For example $\{a^n b^n, n \geq 0\}$ is a computable language because there is a Turing Machine that, given any string in Σ^* it decides whether the string belongs in the language or not.

Partially Computable Languages

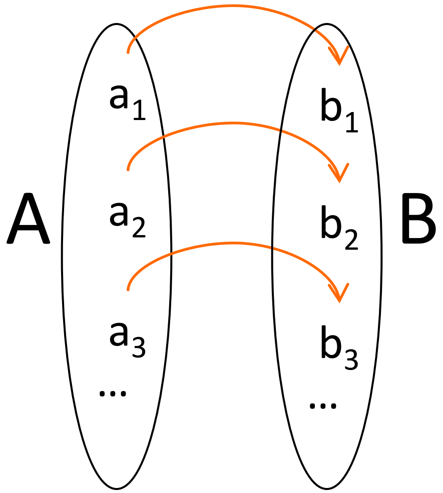
- Partially Computable Predicates: There is a Turing Machine that for output 1 it accepts (halts) and for ↑ it loops for ever.
- Partially Computable Languages: The characteristic function is partially computable (languages that are accepted by Turing Machines).

COUNTING INFINITE SETS

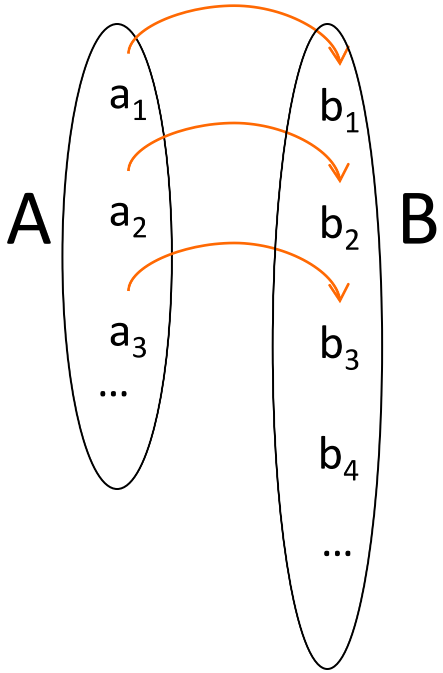
- We say that two infinite sets A, B are of the same size if there is an one to one and onto function from A to B (or from B to A)
- We say that an infinite set A is at most as large as another infinite set B if we can find an one to one function from A to B.

Examples

$|A| = |B|$

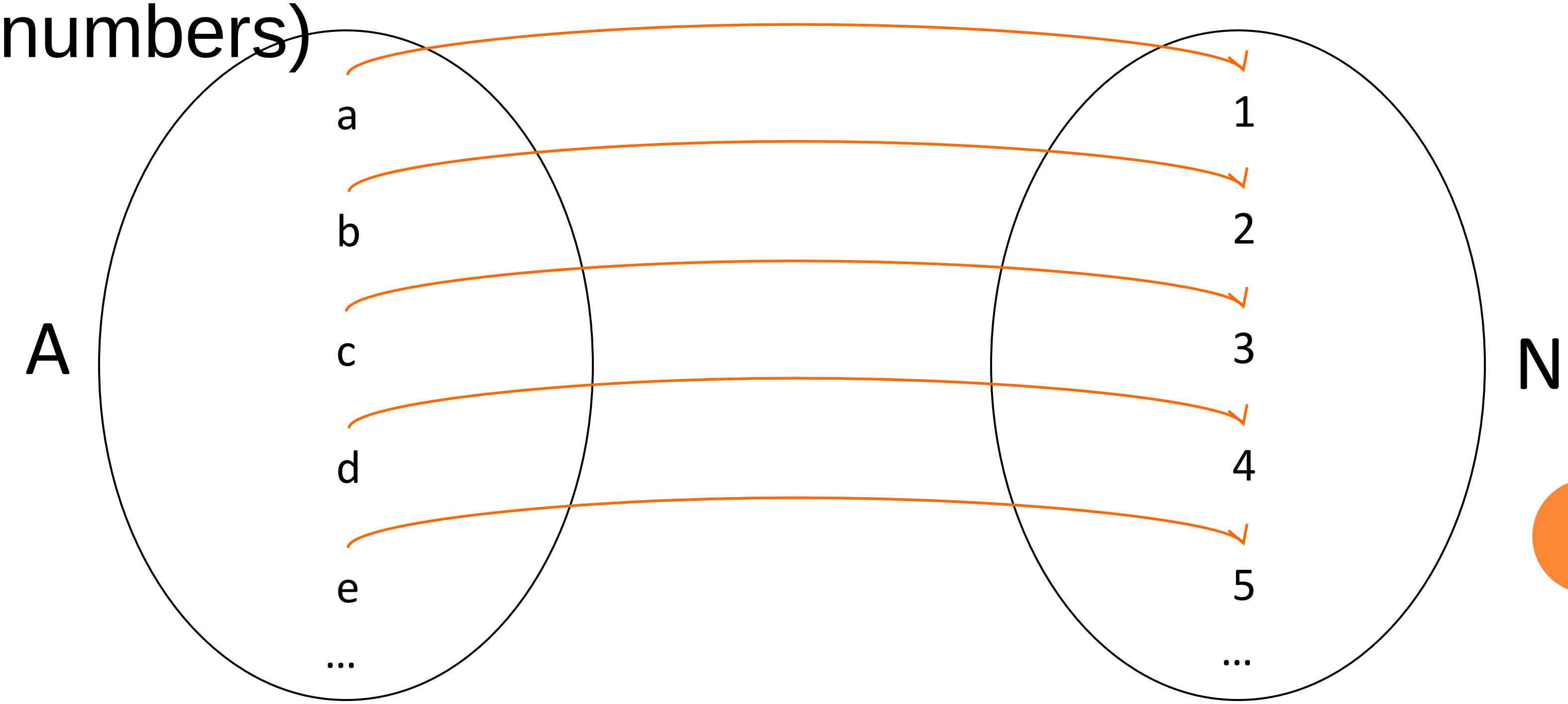


$|A| \leq |B|$



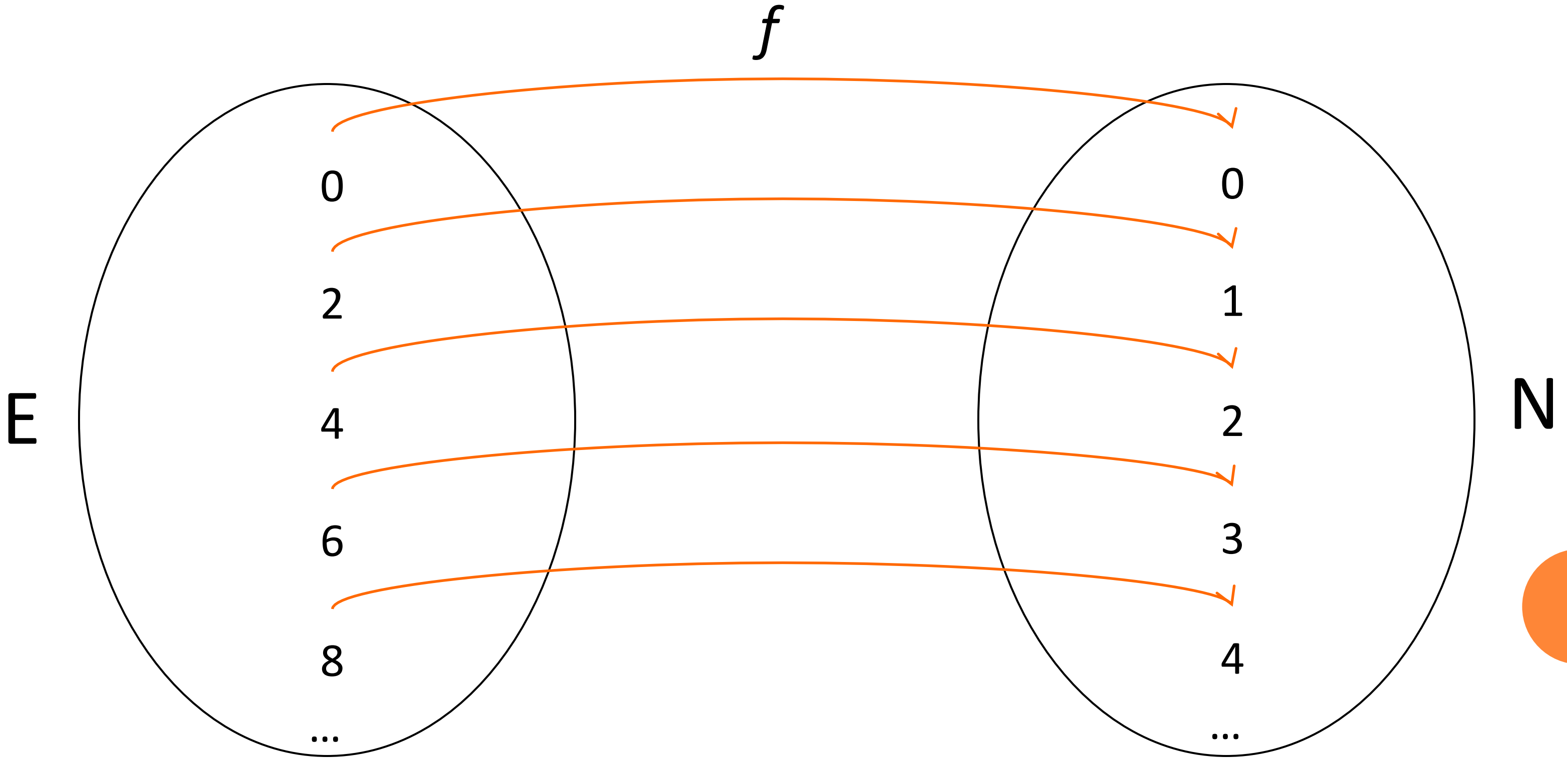
COUNTABLE SETS

- A set is countable if you can find an one to one and onto correspondence with the natural numbers (intuitively this means that it has the same number of elements as the natural numbers)

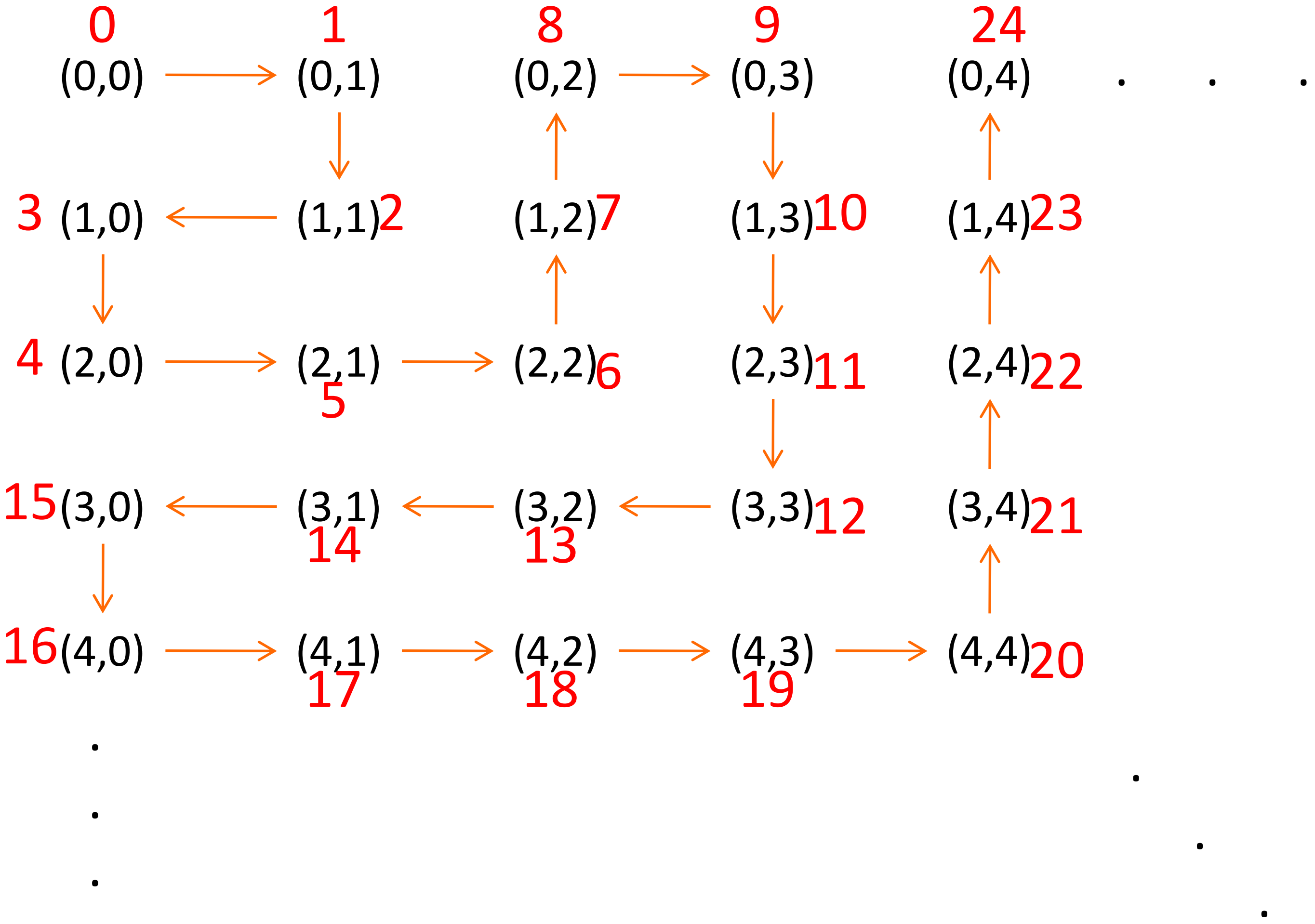


The set of the even numbers is countable

- There is an one to one an onto function from E (the even numbers) to N: f(n) = n/2



N x N is countable



0 (0,0)	1 (0,1)	8 (0,2)	9 (0,3)	24 (0,4)	. . .
3 (1,0)	(1,1)2	(1,2)7	(1,3)10	(1,4)23	
4 (2,0)	(2,1) 5	(2,2)6	(2,3)11	(2,4)22	
15(3,0)	(3,1) 14	(3,2) 13	(3,3)12	(3,4)21	
16(4,0)	(4,1) 17	(4,2) 18	(4,3) 19	(4,4)20	
. . .					. . .

The set of Turing Machines is countable

- Every Turing Machine can be given a unique number in binary as follows:
  - Give to the states a number: q_0 is 1, q_1 is 11 etc…
  - Give to the symbols of the stack a unique number: a is 1, b is 11, c is 111 etc…
  - Assign 1 to L, 11 to R and 111 to S
  - For each δ(q, a) = (q’, b, H) give the binary number 11…1011...1011…1011…1011…1

    q a q’ b H

  where H is L, R or S.

The set of Turing Machines is countable

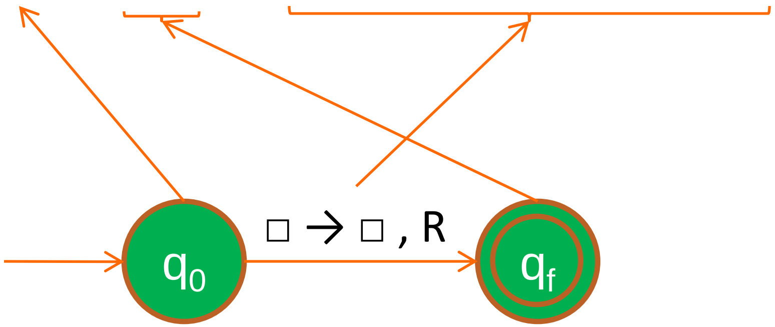
- Every Turing Machine can be given a unique number in binary as follows:
  - To obtain the number of the machine combine each number for the arrows together (separated with 00).
  - The number starts with q_0 00 q_f 00
- The number associated with the Turing Machine M is denoted as <M>

Example

- The Turing Machine M_P that computes the Predicate *P*(n) := n=0 has the following number

?

$<M_P>$ = 100110010101101011



UNCOUNTABLE SETS

- Countable sets are infinite.
- However there are some sets that are considered “even larger”.
- There is no way to enumerate them.
- Diagonalization method: Suppose that there is an enumeration of all the elements of the set.
- Obtain a new element by taking different parts of each element and changing them.
- The new element is not in the enumeration.

Contradiction!!!

The set of N →{0,1} Predicates is not countable

Suppose that it was. An enumeration of all the predicate would be the following:

	1	2	3	4	5	6		
1	0	1	0	1	1	1		f(1) = 1
2	1	0	1	1	0	0	. . .	f(2) = 1
3	0	0	1	0	1	1		f(3) = 0
4	1	1	1	0	1	1		f(4) = 1

.

.

.

The predicate $f(n) = 1 - f_n(n)$ is not in the enumeration.

NOT ALL PREDICATES ARE COMPUTABLE

1. Turing Machines are countable
2. Predicates are uncountable

Thus there is a predicate for which there is no Turing Machine that decides it.

A PREDICATE THAT IS NOT COMPUTABLE

- The Halting Problem: Given a Turing Machine M and an input x does M halt with input x?
- Important problem: If a machine doesn’t stop then we don’t know for sure if it is going to accept the input or not.
- It turns out that this problem is not solvable!
- In other words we can prove that the predicate Halt is not computable (there is no Turing Machine that takes as input the pair (<M>, x) and decides if M is going to accept on x.

PREDICATE HALT IS NOT COMPUTABLE

- We will prove that the predicate $H(<M>) = \begin{cases} 1, M(<M>) & halts \\ 0, M(<M>) & loops \end{cases}$ is not computable.
- Suppose that there is a TM H_{TM} that decides H. In other words, $H_{TM}(<M>)$ halts if $H(<M>) = 1$ else loops.
- Take the machine H'_{TM} which does the opposite: halts on input M if $H(<M>) = 0$ else loops.
- Run H'_{TM} with input the machine H'_{TM} *itself*.
  - H'_{TM} $(<H'_{TM}>)$ is going to halt if $H(<H'_{TM}>) = 0$ which means that H'_{TM} $(<H'_{TM} >)$ loops.
  - H'_{TM} $(<H'_{TM}>)$ is going to loop if $H(<H'_{TM}>) = 1$ which means that H'_{TM} $(<H'_{TM} >)$ halts.

Important Facts

- This method is called *self-reference*
- Thesis Turing-Church implies that any other machine or program can do exactly whatever a Turing Machine can do.
- There is no machine or program that can decide whether or not a machine or program is going to halt.
- Computers CANNOT decide the halting problem.

HALTING PROBLEM FOR PASCAL PROGRAMS

- Suppose that there is a Pascal program Halt that takes as input another Pascal program p and decides whether this program is going to halt or not (outputs true if p halts, else false).
- Create a new Pascal program Halt' by adding to Halt the following code:
  - while Halt(p)=true do a:=1;
- This makes Halt'(p) to loop if p halts and vice versa.

Halting problem for Pascal programs

- Of course Halt' is a proper Pascal program so why not giving it as input to Halt' (self-reference).
- Now:
  - If Halt'(Halt') halts then Halt(Halt') = true so Halt'(Halt') is going to loop .
  - If Halt'(Halt') loops then Halt(Halt') = false so Halt'(Halt') is going to halt

Contradiction!!!

Universal Turing Machine

- We can create a Turing machine U that takes a pair of numbers (<M>, x) as input and then simulates M running on input x:
  - If M accepts on x then U accepts on (<M>, x)
  - If M rejects on x then U rejects on (<M>, x)
  - If M loops on x then U loops on (<M>, x)
- This machine is called a *Universal Turing Machine.*

PREDICATE H IS PARTIALLY COMPUTABLE

- The partial predicate is partially computable.

$$H(<M>) = \begin{cases} 1, M(<M>) & halts \\ \uparrow, M(<M>) & loops \end{cases}$$

- Run the Universal Turing machine U with input (<M>,<M>).

A PREDICATE THAT IS NOT PARTIALLY COMPUTABLE

- Consider the predicate $\overline{H}(<M>) = \begin{cases} \uparrow, M(<M>) & halts \\ 0, M(<M>) & loops \end{cases}$
- $\overline{H}$ is not partially computable
- Suppose that there was a TM U' that could partially compute $\overline{H}$.
- Idea: Run both machines U and U' on input (<M>,<M>). At some point one of them will halt.
  - If U halts then accept
  - If U' halts then reject
- But this decides the Halt predicate. Contradiction!
- Difficult part: Simulate in one machine the concurrent running of U and U'.