Algorithmics

Lecture 12

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 5 – Computability

Introduction

Models of computation

- finite-state automata
- pushdown automata
- Turing machines
- Counter machines
- Church-Turing thesis

Turing machines

A Turing machine is a universal model of computation

- all computable functions can be defined using a Turing machine
- a generalisation of a push-down automaton where we have unlimited amount of memory (an infinite tape) and no restriction on how to read it (not a stack/LIFO)

Invented by Alan Turing in 1936

- "Any process which could naturally be called an effective procedure can be realized by a Turing machine." (then called automatic machine)
- many variants exist
- a universal Turing machine is able to simulate any other Turing machine
- modern computers as implementations of Turing's universal machine

A DYI Turing machine http://aturingmachine.com/

Turing machines

A **Turing machine T** to recognise a particular language consists of

Turing machines

A Turing Machine T to recognise a particular language consists of

- a finite alphabet Σ, including a blank symbol (denoted by #)

Turing machines

A Turing Machine T to recognise a particular language consists of

- a finite alphabet Σ, including a blank symbol (denoted by #)
- an unbounded tape of squares
  - each can hold a single symbol of Σ
  - tape unbounded in both directions

Turing machines

A Turing Machine T to recognise a particular language consists of

- a finite alphabet Σ, including a blank symbol (denoted by #)
- an unbounded tape of squares
  - each can hold a single symbol of Σ
  - tape unbounded in both directions
- a tape head that scans a single square
  - it can read from it and write to the square
  - then moves one square `left` or `right` along the tape

Turing machines

A Turing Machine T to recognise a particular language consists of

- a finite alphabet Σ, including a blank symbol (denoted by #)
- an unbounded tape of squares
  - each can hold a single symbol of Σ
  - tape unbounded in both directions
- a tape head that scans a single square
  - it can read from it and write to the square
  - then moves one square `left` or `right` along the tape
- a set S of states
  - includes a single start state s_0 and two halt (or terminal) states s_Y and s_N

Turing machines

A Turing Machine T to recognise a particular language consists of

- a finite alphabet Σ, including a blank symbol (denoted by #)
- an unbounded tape of squares
  - each can hold a single symbol of Σ
  - tape unbounded in both directions
- a tape head that scans a single square
  - it can read from it and write to the square
  - then moves one square `left` or `right` along the tape
- a set S of states
  - includes a single start state s_0 and two halt (or terminal) states s_Y and s_N
- a transition function
  - essentially the inbuilt program

Turing machines – Computation

The transition function is of the form

$$f : ((S/\{s_Y, s_N\}) \times \Sigma) \rightarrow (S \times \Sigma \times \{Left, Right\})$$

For each non-terminal state and symbol the function f specifies

- a new state (perhaps unchanged)
- a new symbol (perhaps unchanged)
- a direction to move along the tape

$f(s,\sigma)=(s',\sigma',d)$ means reading symbol σ from the tape in state s

Turing machines – Computation

The transition function is of the form

$$f : ((S/\{s_Y, s_N\}) \times \Sigma) \rightarrow (S \times \Sigma \times \{Left, Right\})$$

For each non-terminal state and symbol the function f specifies

- a new state (perhaps unchanged)
- a new symbol (perhaps unchanged)
- a direction to move along the tape

$f(s,\sigma)=(s',\sigma',d)$ means reading symbol σ from the tape in state s

- move to state $s' \in S$

Turing machines – Computation

The **transition function** is of the form

$$f : ((S/\{s_Y, s_N\}) \times \Sigma) \rightarrow (S \times \Sigma \times \{Left, Right\})$$

For each non-terminal state and symbol the function f specifies

- a new state (perhaps unchanged)
- a new symbol (perhaps unchanged)
- a direction to move along the tape

$f(s,\sigma)=(s',\sigma',d)$ means reading symbol σ from the tape in state s

- move to state $s' \in S$
- overwrite the symbol σ on the tape with the symbol $\sigma' \in \Sigma$
  - if you do not want to overwrite the symbol write the symbol you read

Turing machines – Computation

The transition function is of the form

$$f : ((S/\{s_Y, s_N\}) \times \Sigma) \rightarrow (S \times \Sigma \times \{Left,\ Right\})$$

For each non-terminal state and symbol the function f specifies

- a new state (perhaps unchanged)
- a new symbol (perhaps unchanged)
- a direction to move along the tape

$f(s,\sigma)=(s',\sigma',d)$ means reading symbol σ from the tape in state s

- move to state $s' \in S$
- overwrite the symbol σ on the tape with the symbol $\sigma' \in \Sigma$
- move the tape head one square in direction $d \in \{Left,\ Right\}$

Turing machines – Computation

The (finite) input string is placed on the tape

- assume initially all other squares of the tape contain blanks or #

The tape head is placed on the first symbol of the input

T starts in state s_0 (scanning the first symbol)

- if T halts in state s_Y, the answer is 'yes' (accepts the input)
- if T halts in state s_N, the answer is 'no' (rejects the input)
- no condition on what state the tape is at the end (see PDAs)

The palindrome problem

Instance: a finite string **Y**

Question: is **Y** a palindrome, i.e. is **Y** equal to the reverse of itself

- simple Java method to solve the above:

```java
 public boolean isPalindrome(String s){
   int n = s.length();
   if (n < 2) return true;
   else
     if (s.charAt(0) != s.charAt(n-1)) return false;
     else return isPalindrome(s.substring(1,n-2));
}
```

We will design a Turing Machine that solves this problem

- in fact, as stated previously, a NDPDA can recognise palindromes

For simplicity, we assume that the string is composed of a's and b's

The palindrome problem – Turing machine

Formally defining a Turing Machine for even simple problems is hard

- much easier to design a pseudocode version

Recall: for pushdown automata we needed nondeterminism to solve the palindrome problem

- needed to guess where the middle of the palindrome was

However as we will show using Turing machines we do not need nondeterminism

The palindrome problem – Turing machine

Formally defining a Turing Machine for even simple problems is hard

- much easier to design a pseudocode version

TM Algorithm for the Palindrome problem

```
read the symbol in the current square;
erase this symbol;
enter a state that 'remembers' it;
move tape head to the end of the input;
if (only blank characters remain)
   enter the accepting state and halt;
else if (last character matches the one erased)
   erase it too;
else
   enter rejecting state and halt;
if (no input left)
   enter accepting state and halt;
else
   move to start of remaining input;
   repeat from first step;
```

The palindrome problem – Turing machine

We need the following states (assuming alphabet is Σ={#,a,b}):

- s_0 reading and erasing the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
  - i.e. s_1 when read (and erased) a and s_2 when read (and erased) b
- s_3, s_4 testing for the appropriate rightmost symbol
  - i.e. s_3 testing against a and s_4 testing against b
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

Transitions:

- from s_0, we enter s_Y if a blank is read, or move to s_1 or s_2 depending on whether an a or b is read, erasing it in either case

States:

- s_0 reading, erasing and remembering the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
- s_3, s_4 testing for the appropriate rightmost symbol
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

Transitions:

- from s_0, we enter s_Y if a blank is read, or move to s_1 or s_2 depending on whether an a or b is read, erasing it in either case
- we stay in s_1/s_2 moving right until a blank is read (reached the end), at which point we enter s_3/s_4 and move left

States:

- s_0 reading, erasing and remembering the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
- s_3, s_4 testing for the appropriate rightmost symbol
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

Transitions:

- from s_0, we enter s_Y if a blank is read, or move to s_1 or s_2 depending on whether an a or b is read, erasing it in either case
- we stay in s_1/s_2 moving right until a blank is read, at which point we enter s_3/s_4 and move left
- from s_3/s_4 we enter s_Y if a blank is read (nothing left on the tape), s_N if the 'wrong' symbol is read, otherwise erase it, enter s_5, and move left

States:

- s_0 reading, erasing and remembering the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
- s_3, s_4 testing for the appropriate rightmost symbol
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

Transitions:

- from s_0, we enter s_Y if a blank is read, or move to s_1 or s_2 depending on whether an a or b is read, erasing it in either case
- we stay in s_1/s_2 moving right until a blank is read, at which point we enter s_3/s_4 and move left
- from s_3/s_4 we enter s_Y if a blank is read, s_N if the 'wrong' symbol is read, otherwise erase it, enter s_5, and move left
- in s_5 we move left until a blank is read, then move right and enter s_0

States:

- s_0 reading, erasing and remembering the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
- s_3, s_4 testing for the appropriate rightmost symbol
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

A Turing machine can be described by its state transition diagram which is a directed graph where

- each state is represented by a vertex
- $f(s,\sigma) = (s',\sigma',d)$ is represented by an edge from vertex s to vertex s', labelled $(\sigma \rightarrow \sigma', d)$

The palindrome problem – Turing machine

A Turing machine can be described by its state transition diagram which is a directed graph where

- each state is represented by a vertex
- f(s,σ) = (s′,σ′,d) is represented by an edge from vertex s to vertex s′, labelled (σ→σ′,d)
  - edge from s to s’ represents moving to state s’

The palindrome problem – Turing machine

A Turing machine can be described by its state transition diagram which is a directed graph where

- each state is represented by a vertex
- f(s,σ) = (s′,σ′,d) is represented by an edge from vertex s to vertex s′, labelled ($\sigma \rightarrow \sigma$′,d)
  - edge from s to s' represents moving to state s'
  - $\sigma \rightarrow \sigma$′ represents overwriting the symbol σ on the tape with the symbol σ′

The palindrome problem – Turing machine

A Turing machine can be described by its state transition diagram which is a directed graph where

- each state is represented by a vertex
- $f(s,\sigma) = (s',\sigma',d)$ is represented by an edge from vertex s to vertex s', labelled $(\sigma \rightarrow \sigma', d)$
  - edge from s to s' represents moving to state s'
  - $\sigma \rightarrow \sigma'$ represents overwriting the symbol σ on the tape with the symbol σ'
  - d represents moving the tape head one square in direction d

The palindrome problem - Turing machine

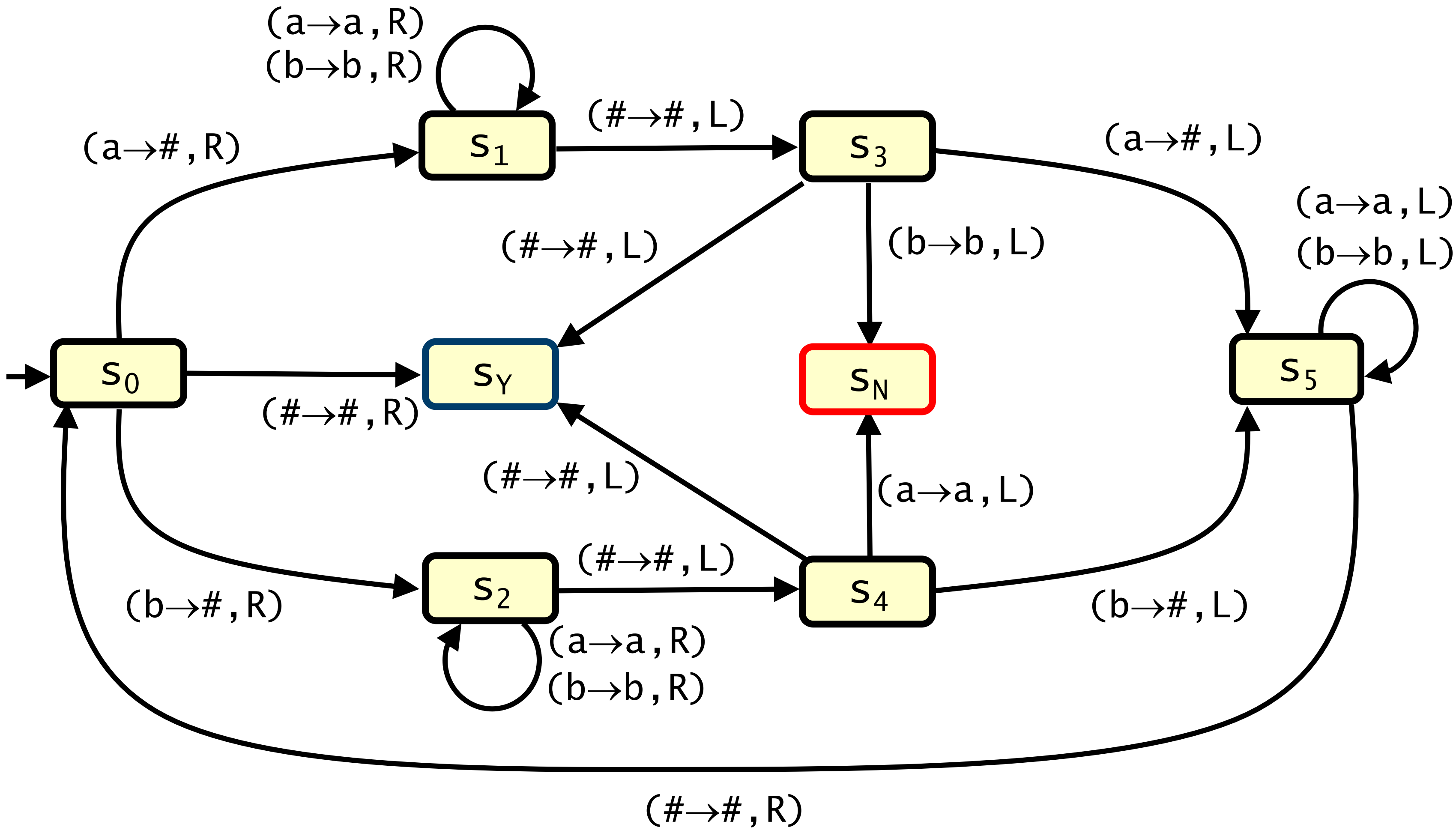
A Turing machine can be described by its state transition diagram which is a directed graph where

- each state is represented by a vertex
- $f(s,\sigma) = (s',\sigma',d)$ is represented by an edge from vertex s to vertex s', labelled $(\sigma \rightarrow \sigma', d)$
  - edge from s to s' represents moving to state s'
  - $\sigma \rightarrow \sigma'$ represents overwriting the symbol σ on the tape with the symbol σ'
  - d represents moving the tape head one square in direction d

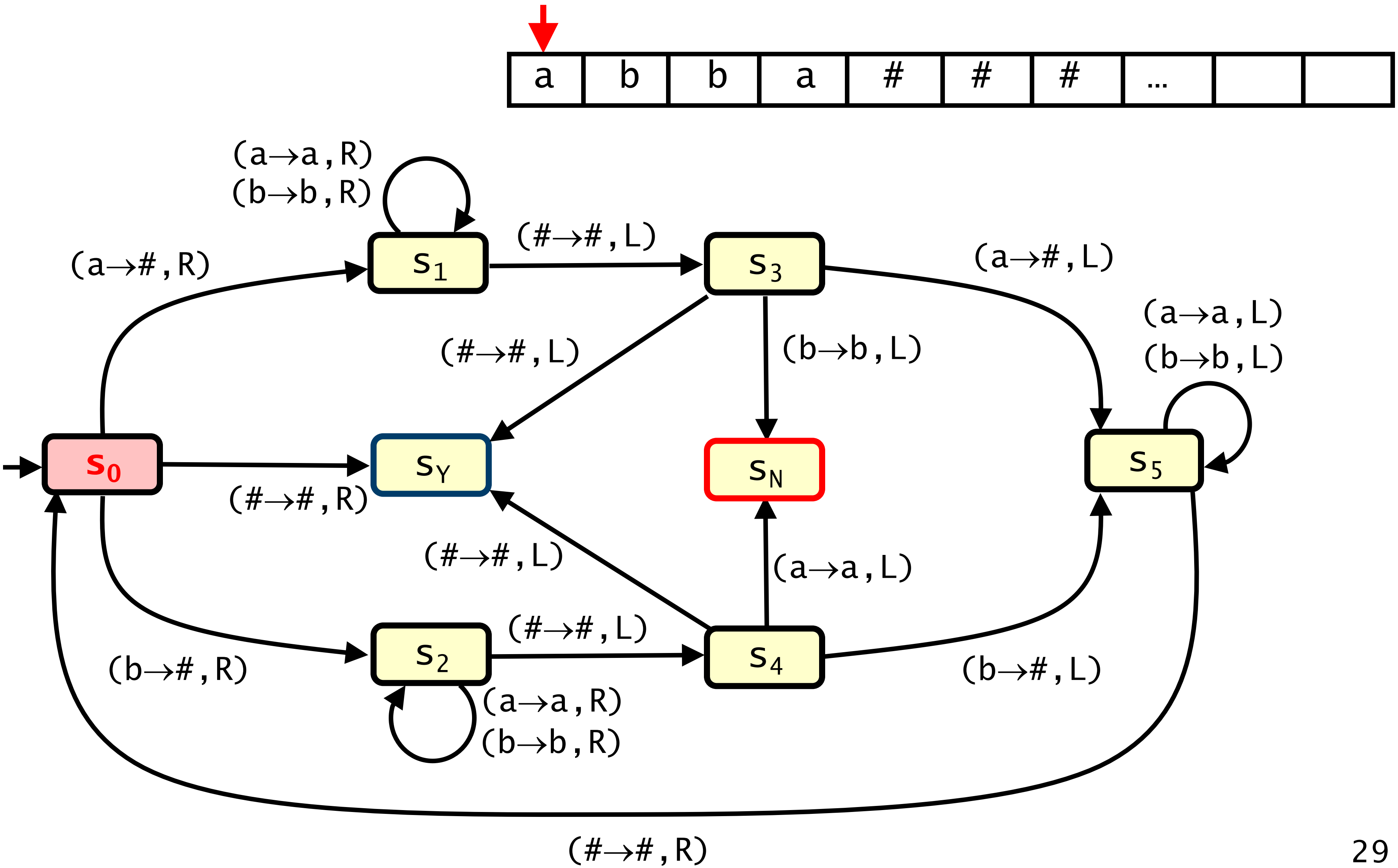
TM for the Palindrome problem (see next slide)

- alphabet is $\Sigma = \{\#, a, b\}$ where # is the blank symbol
- states are $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_Y, s_N\}$

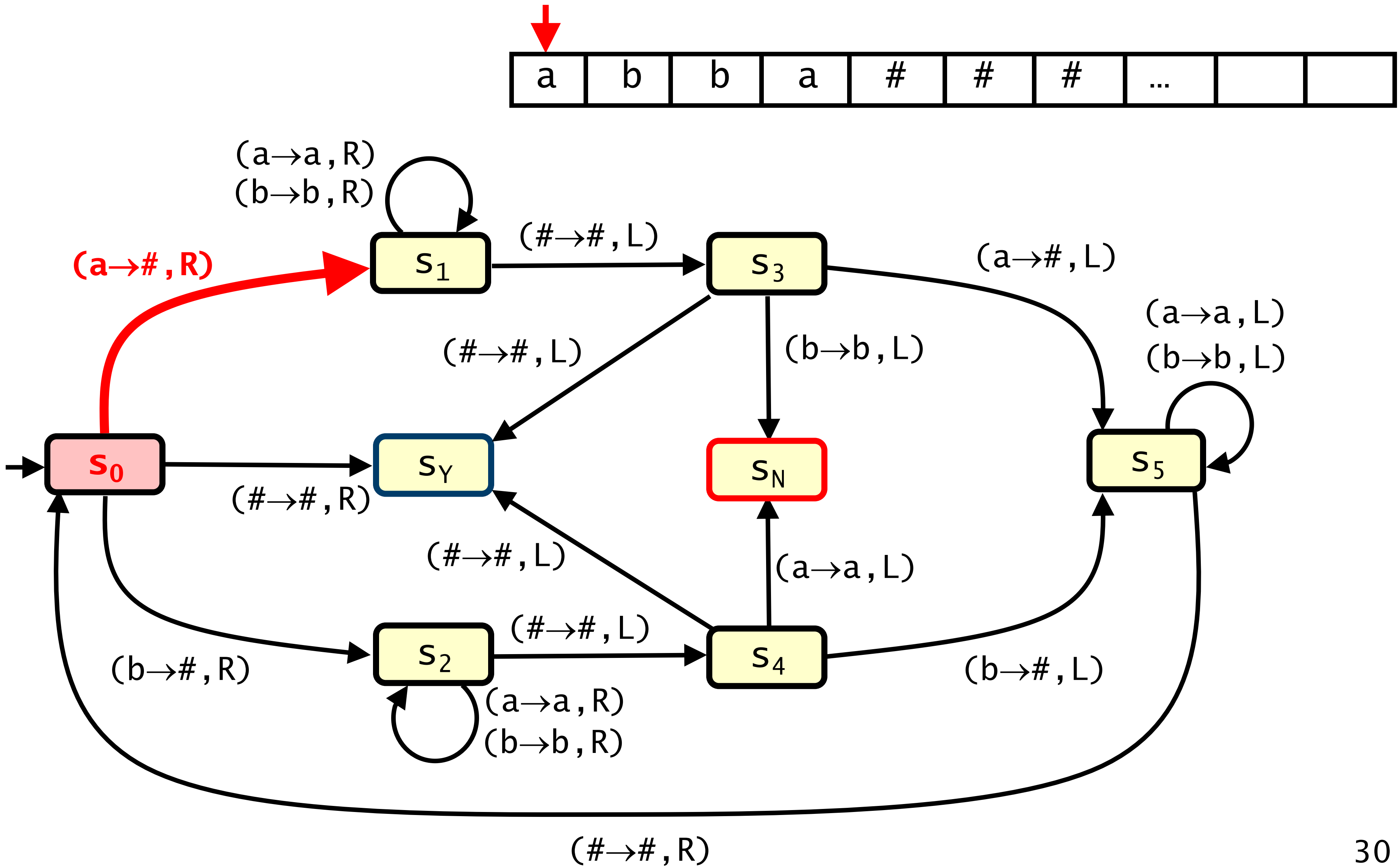
The palindrome problem – Turing machine



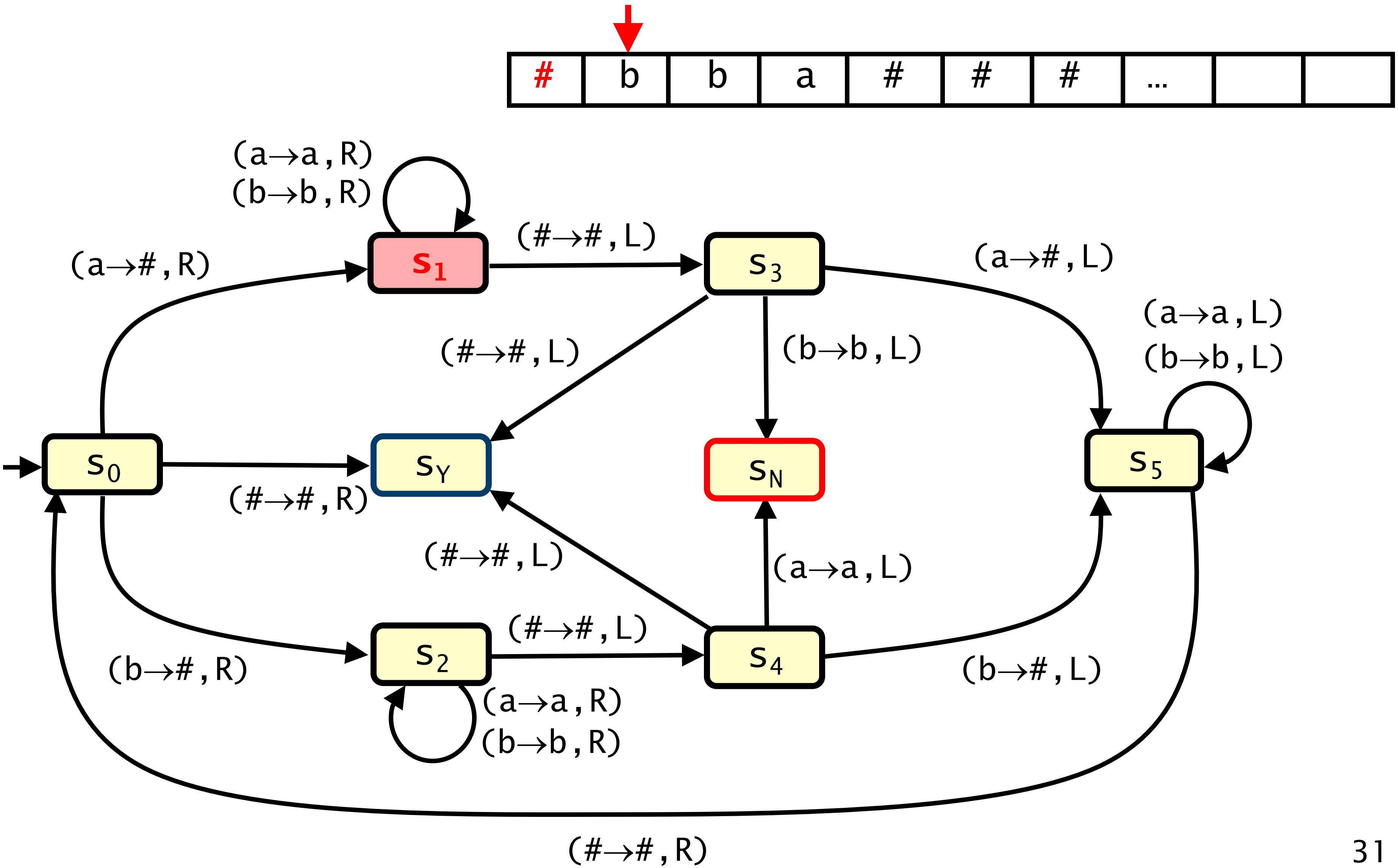
The palindrome problem – Turing machine



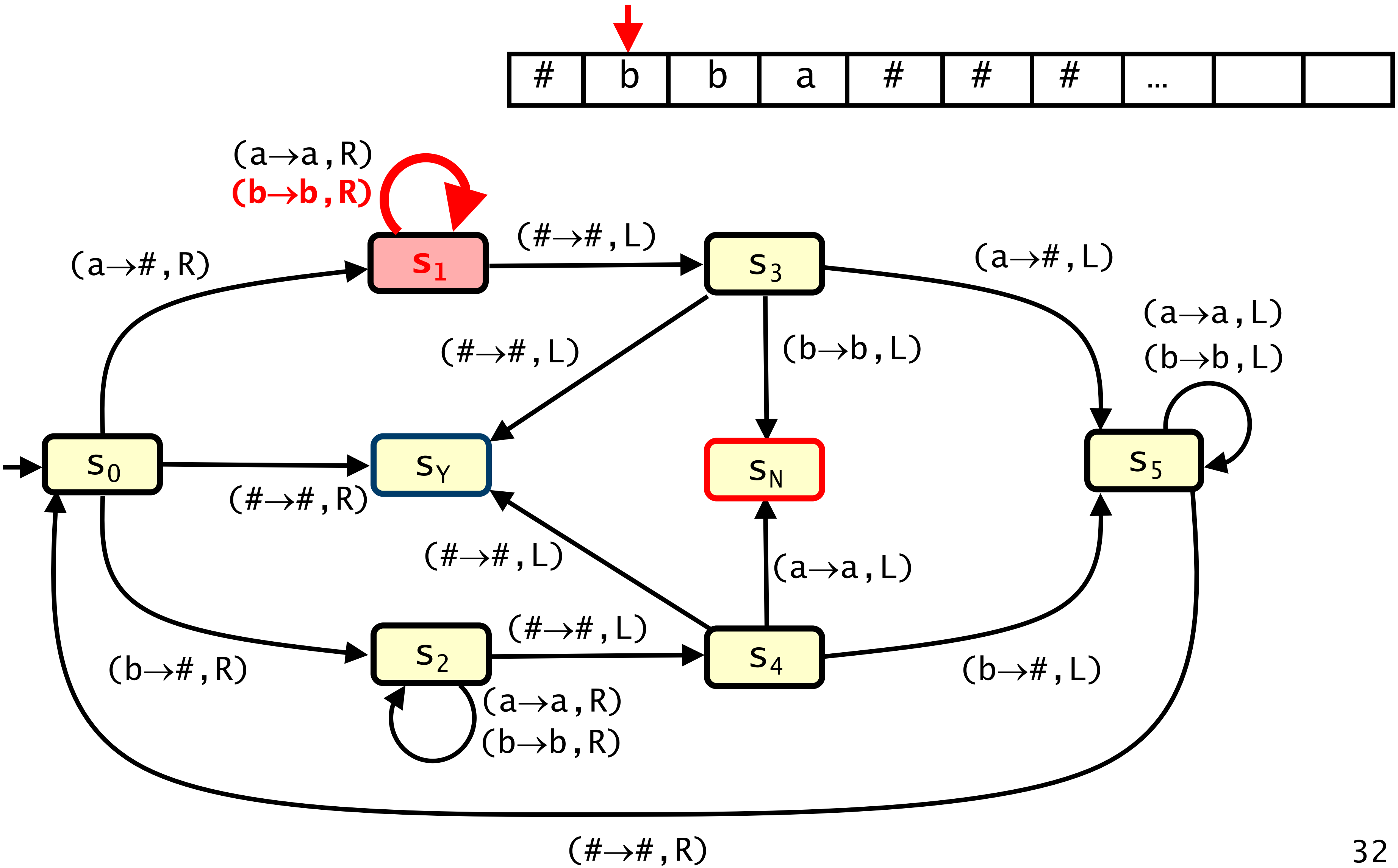
The palindrome problem – Turing machine



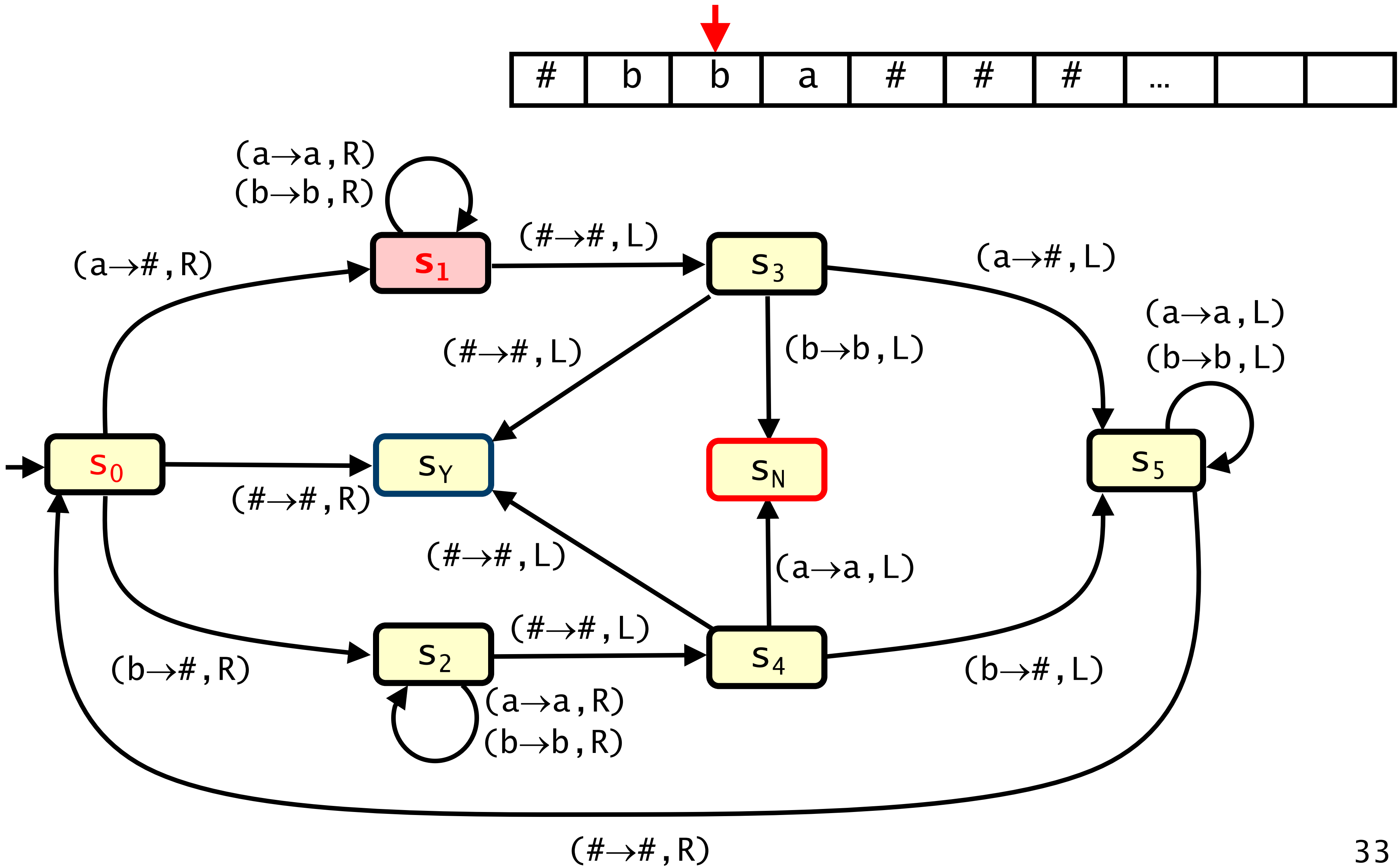
The palindrome problem – Turing machine



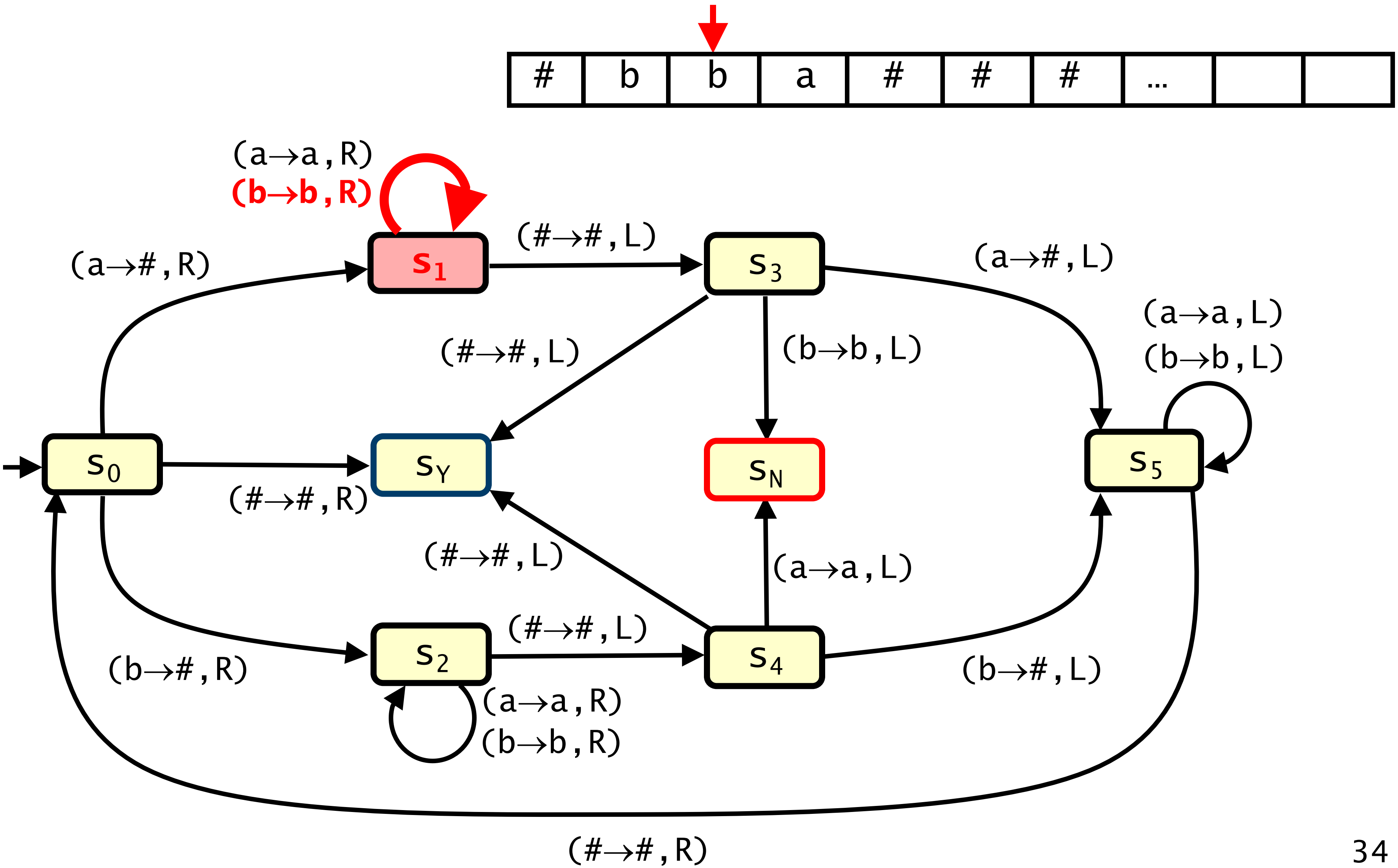
The palindrome problem – Turing machine



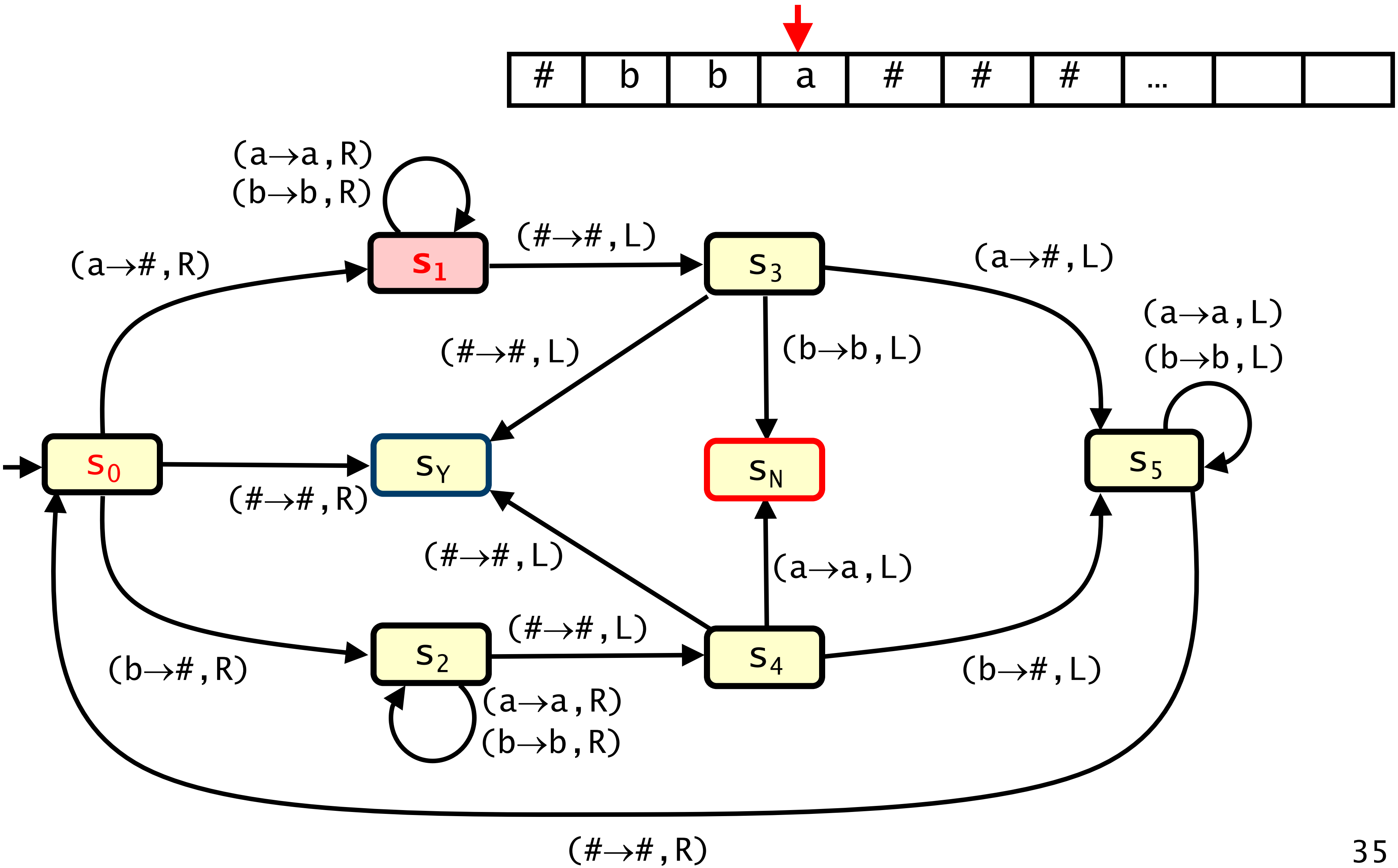
The palindrome problem – Turing machine



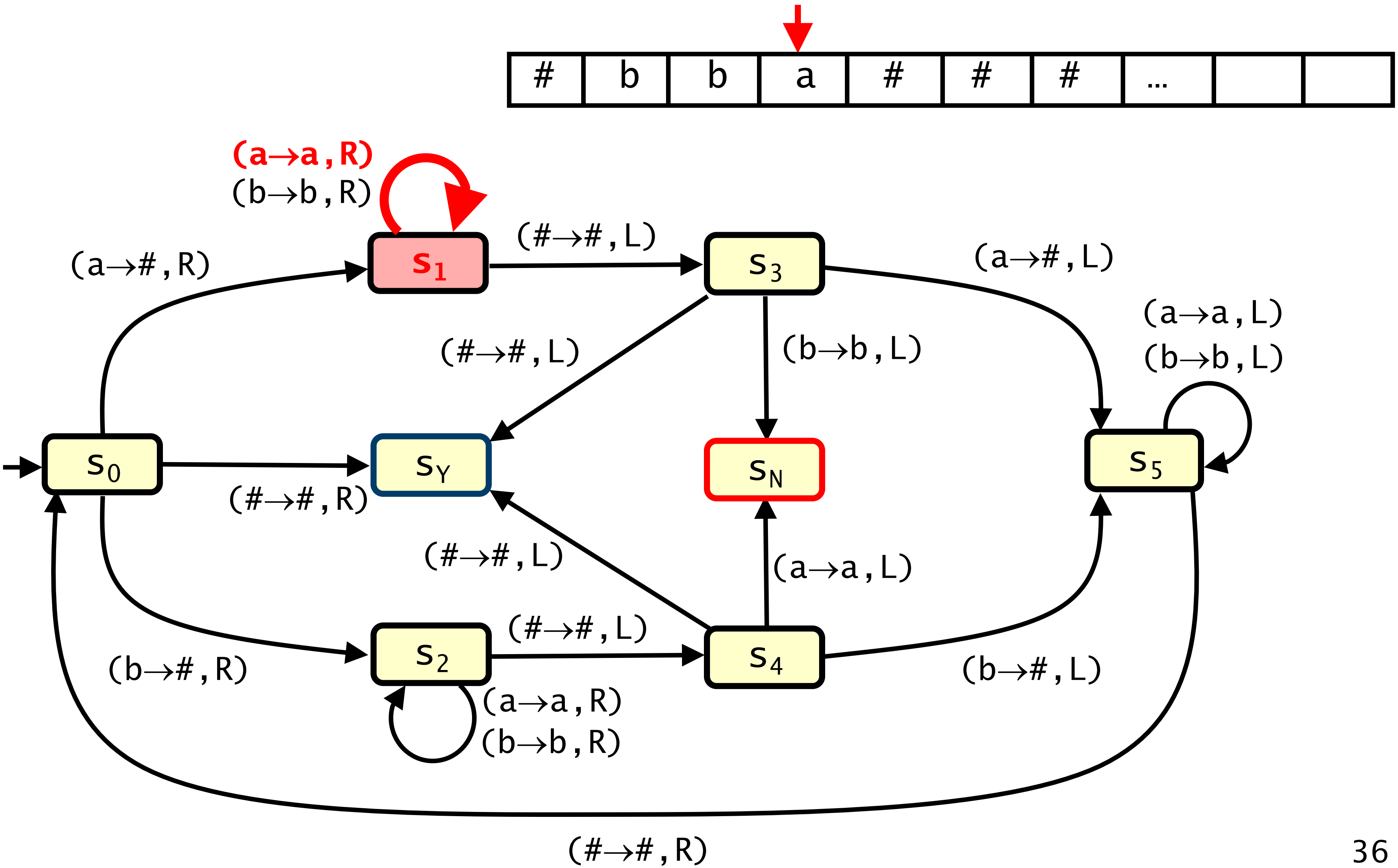
The palindrome problem – Turing machine



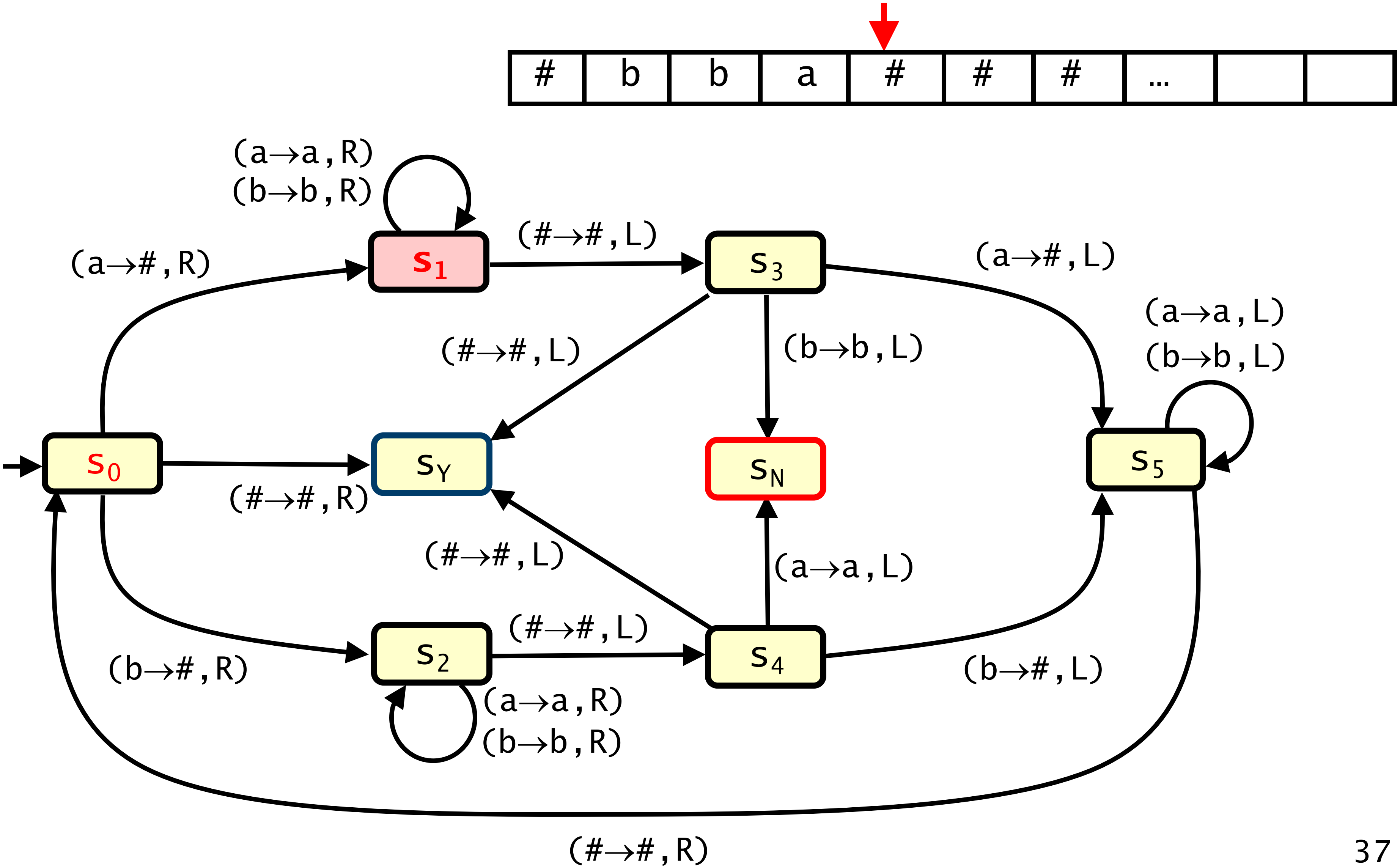
The palindrome problem – Turing machine



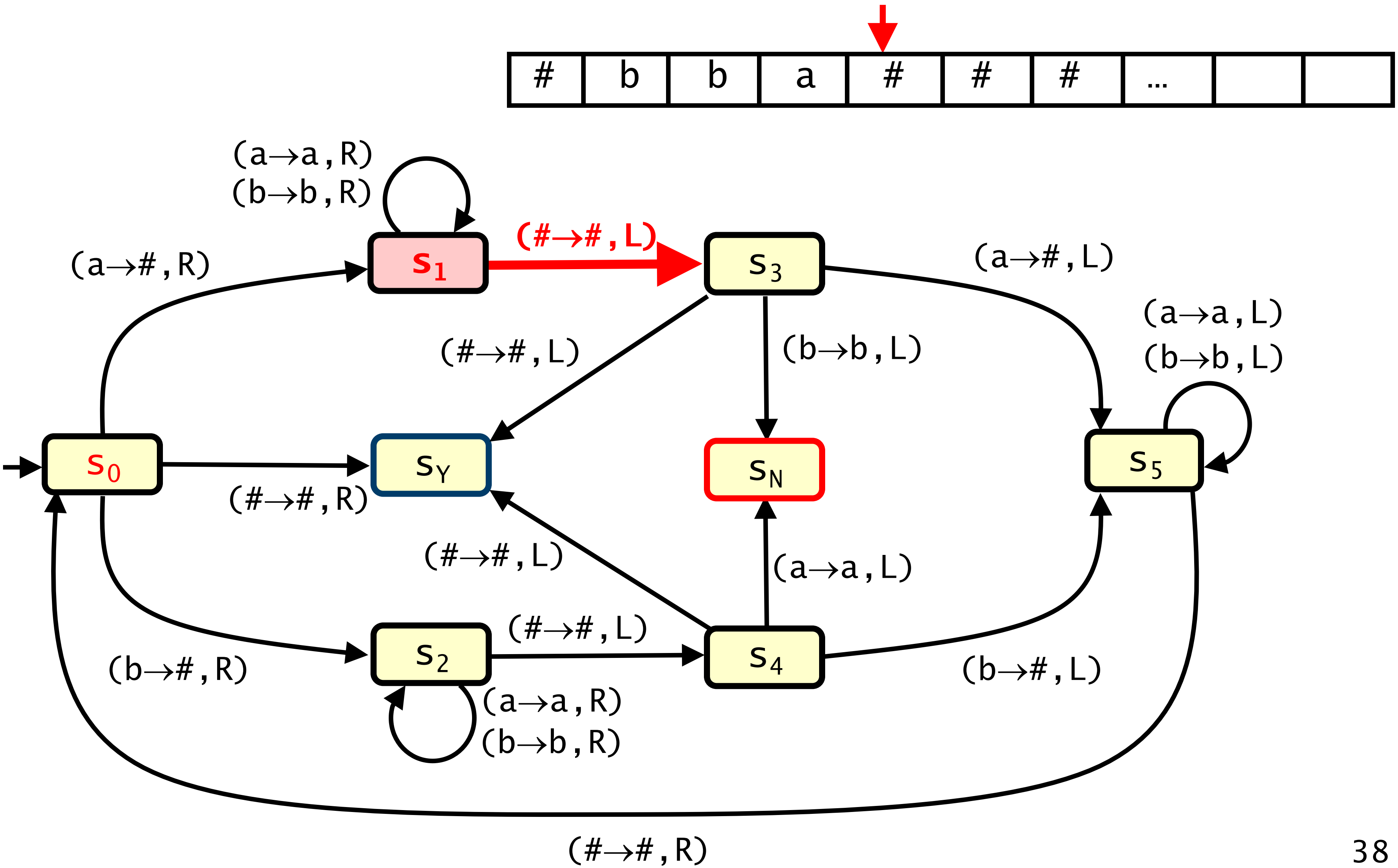
The palindrome problem – Turing machine



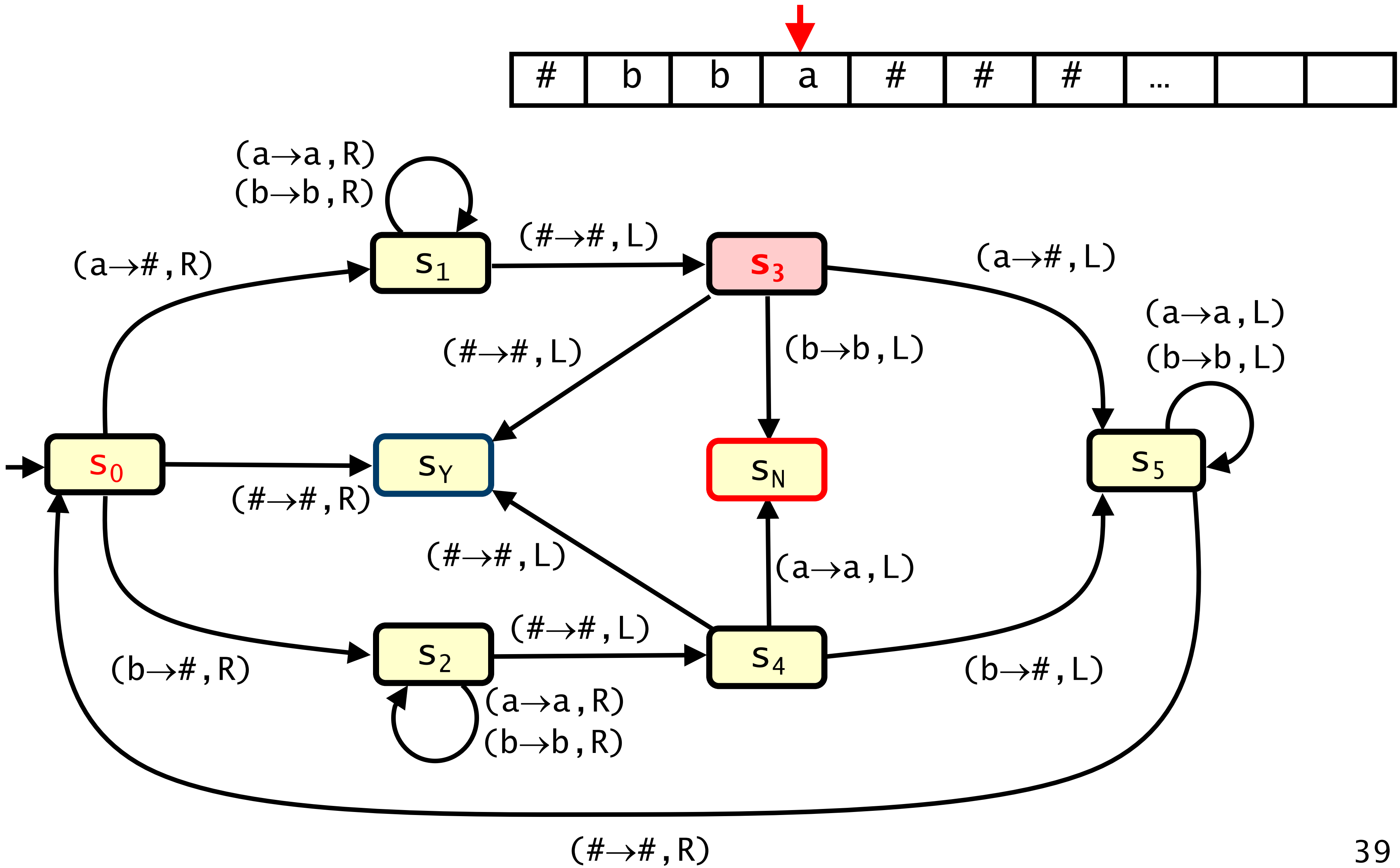
The palindrome problem – Turing machine



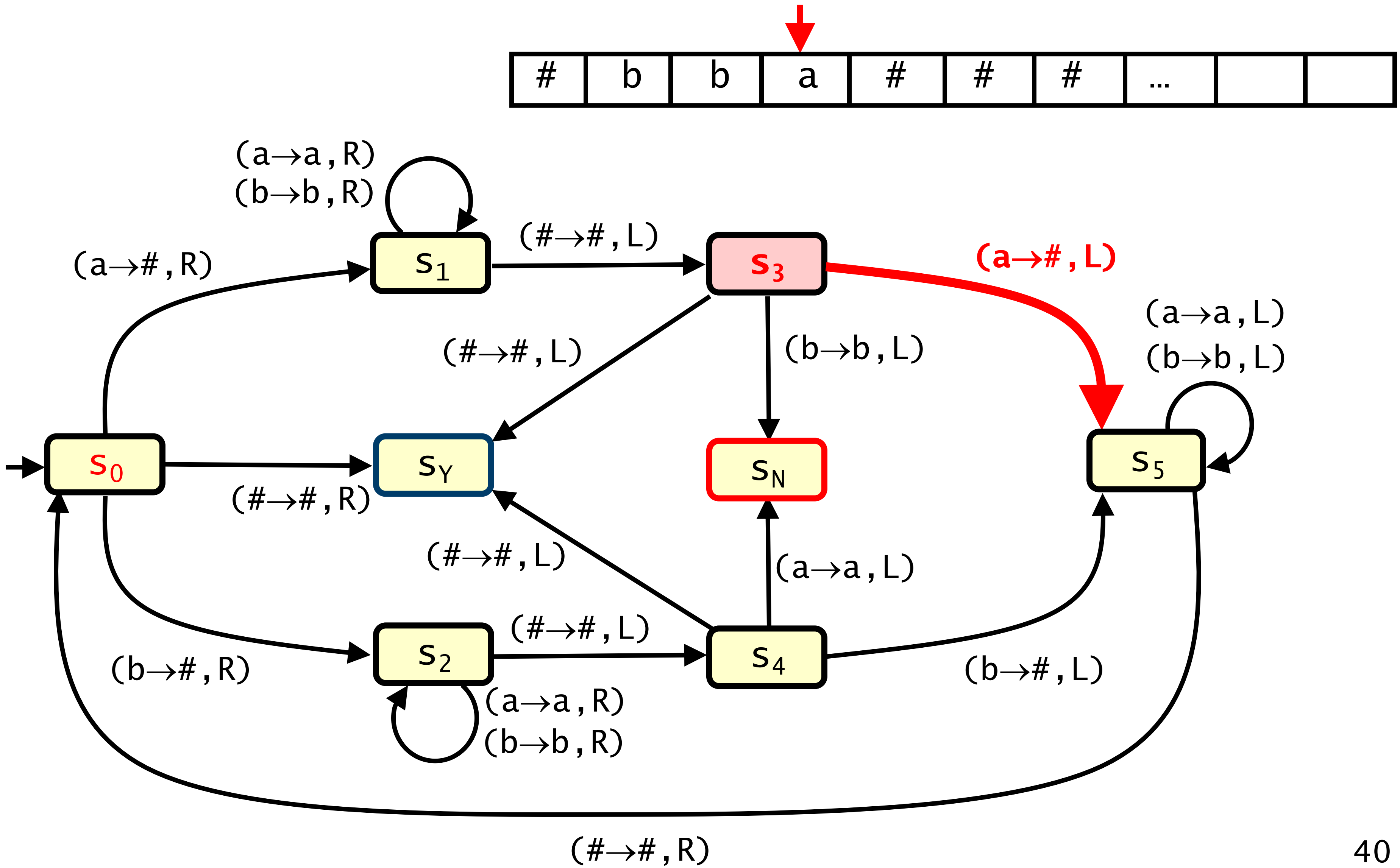
The palindrome problem – Turing machine



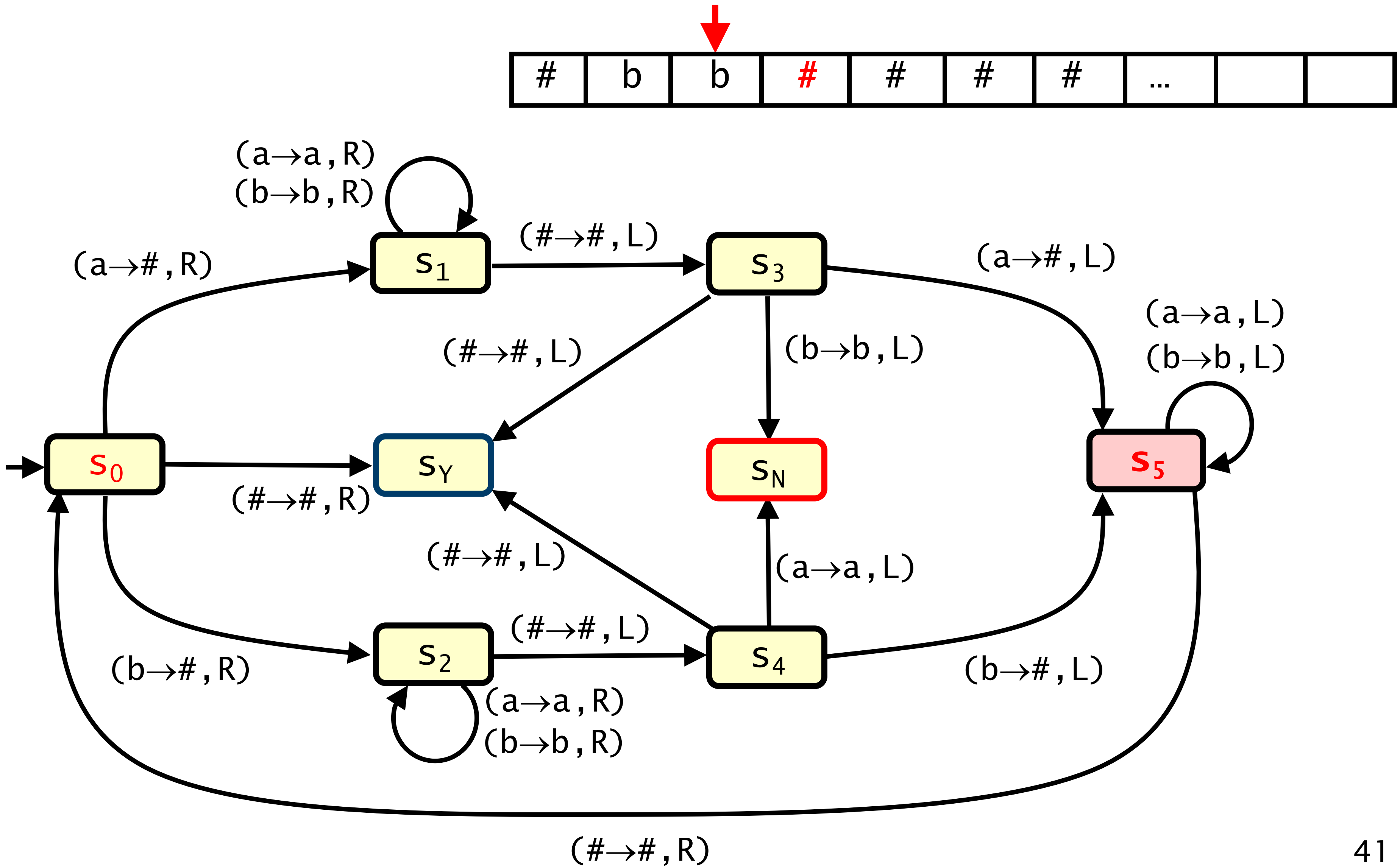
The palindrome problem – Turing machine



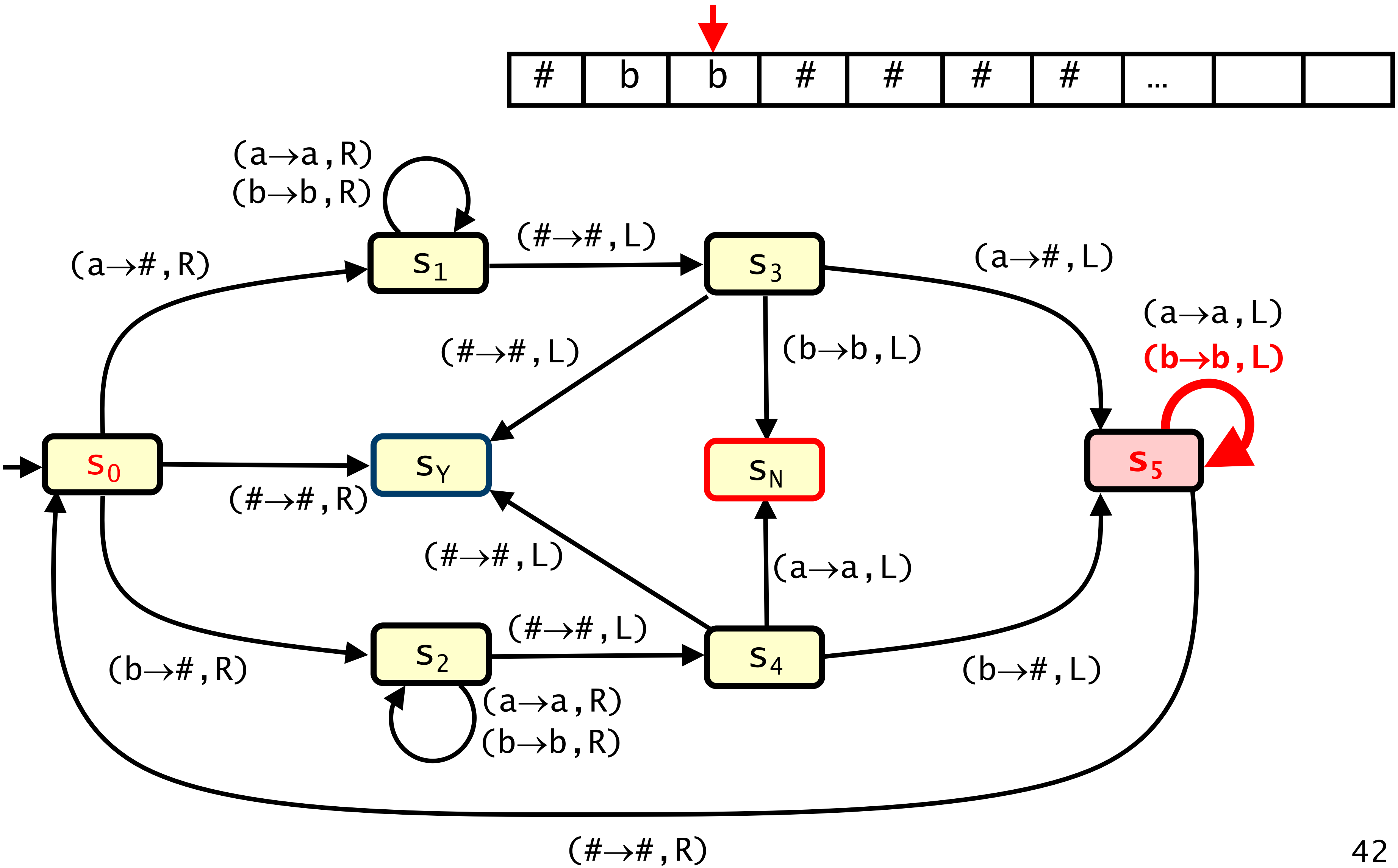
The palindrome problem – Turing machine



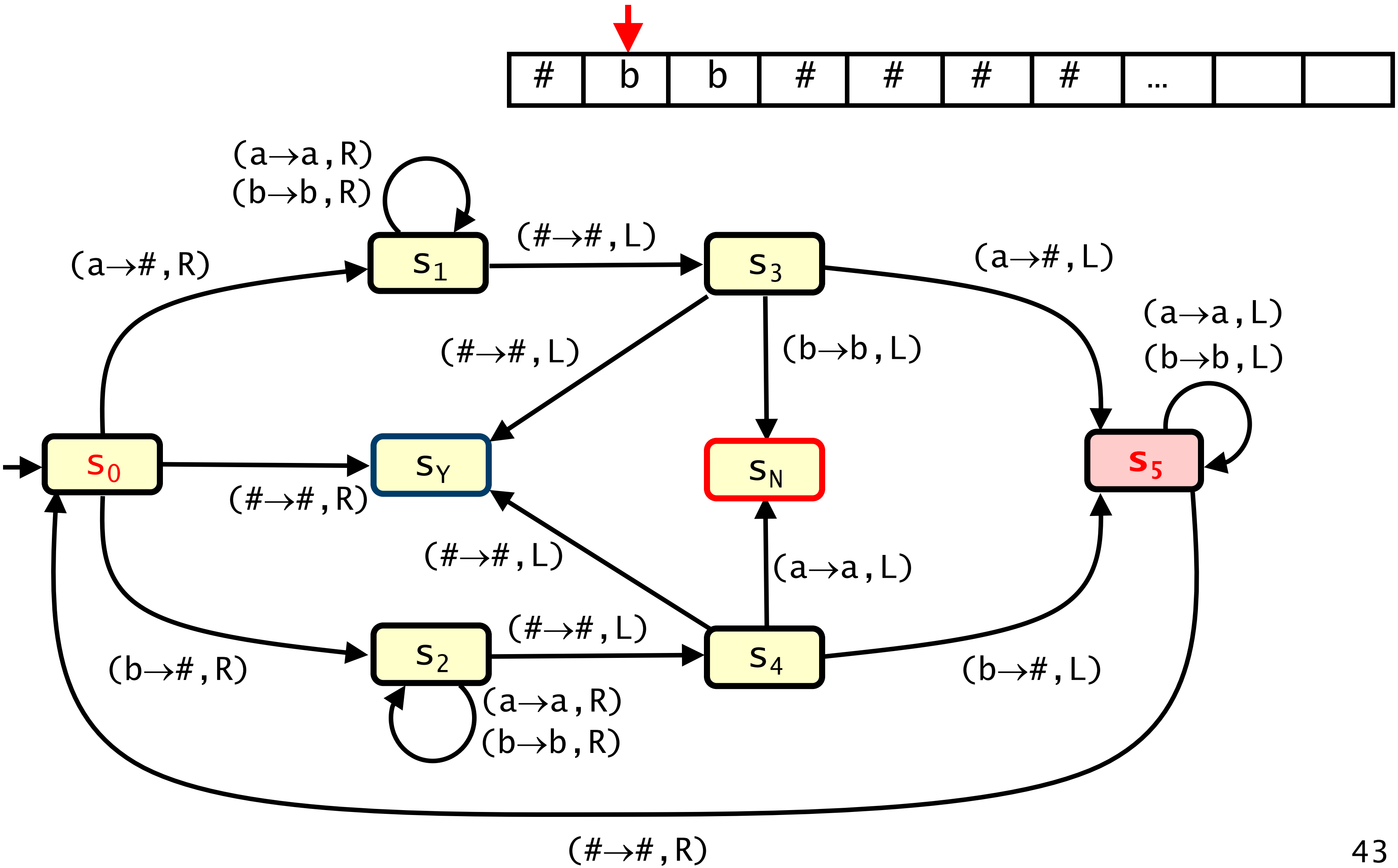
The palindrome problem – Turing machine



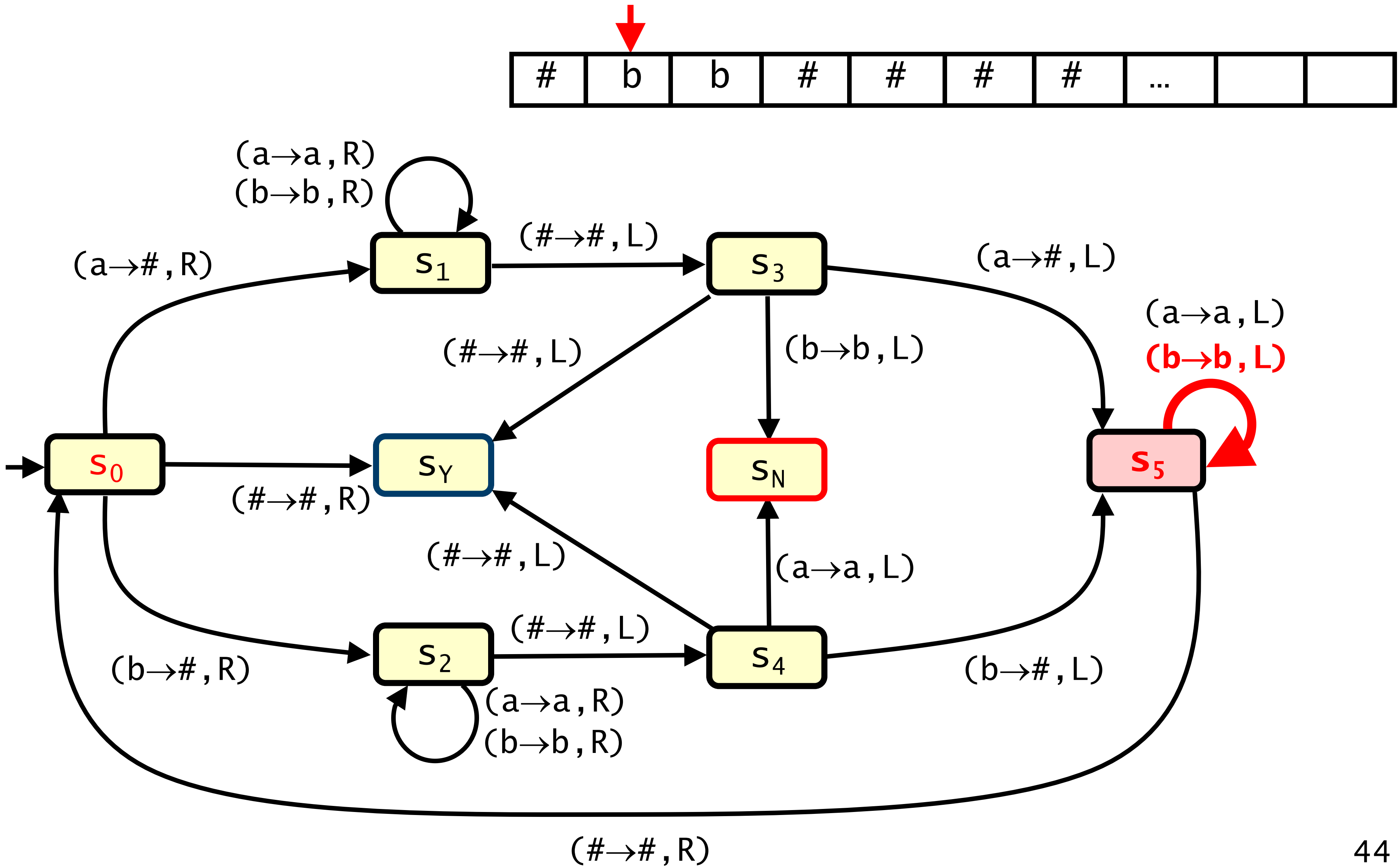
The palindrome problem – Turing machine



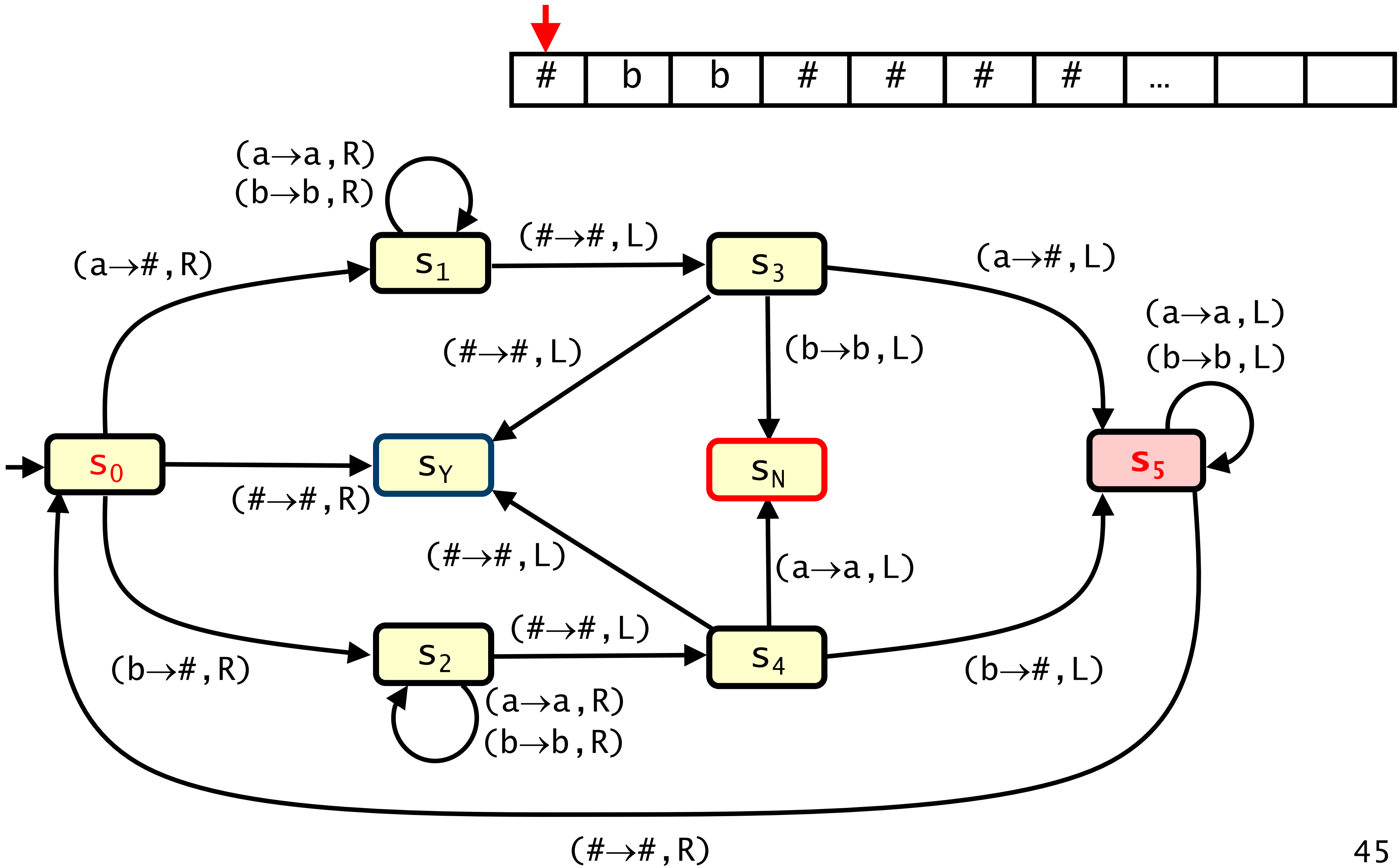
The palindrome problem – Turing machine



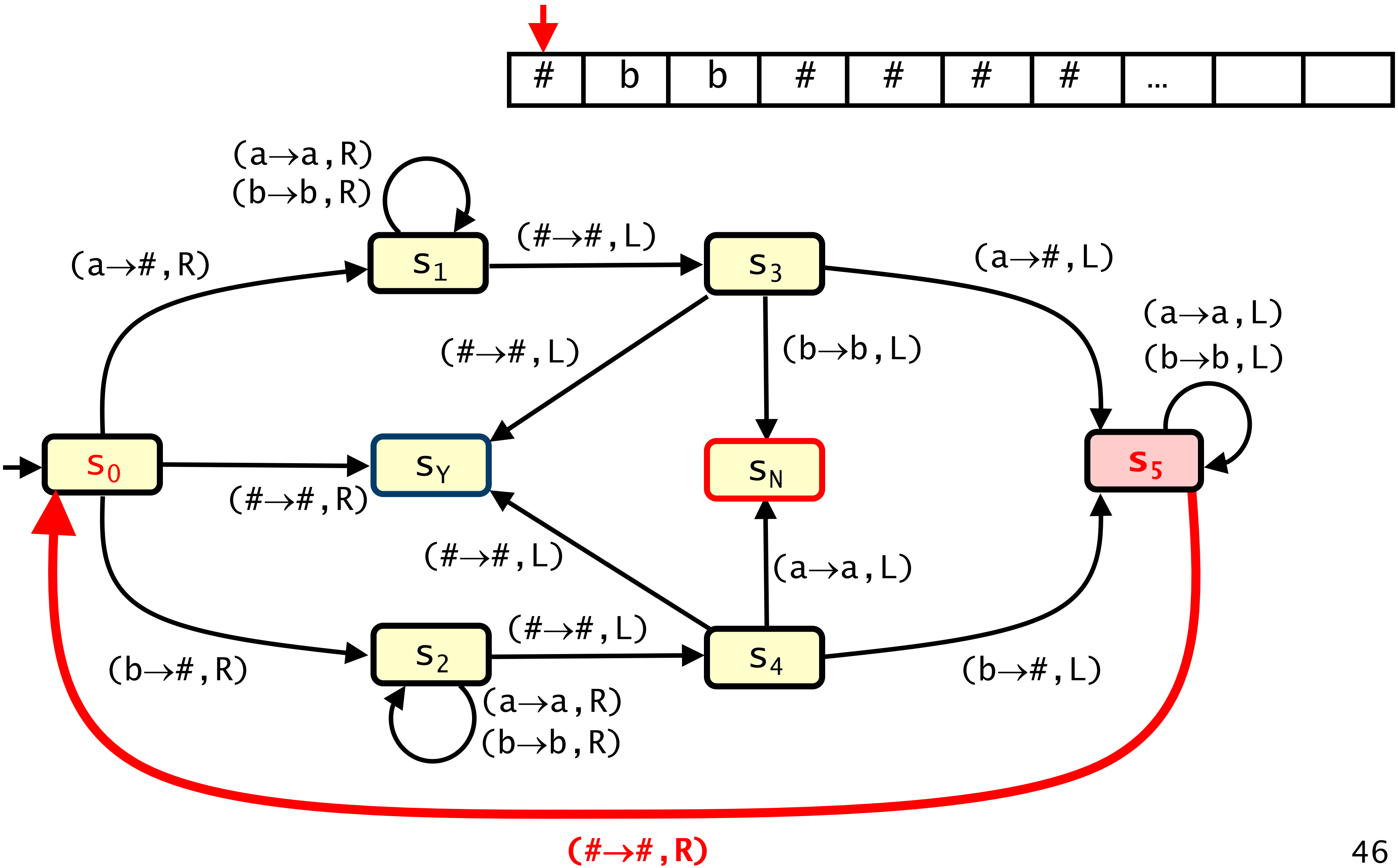
The palindrome problem – Turing machine



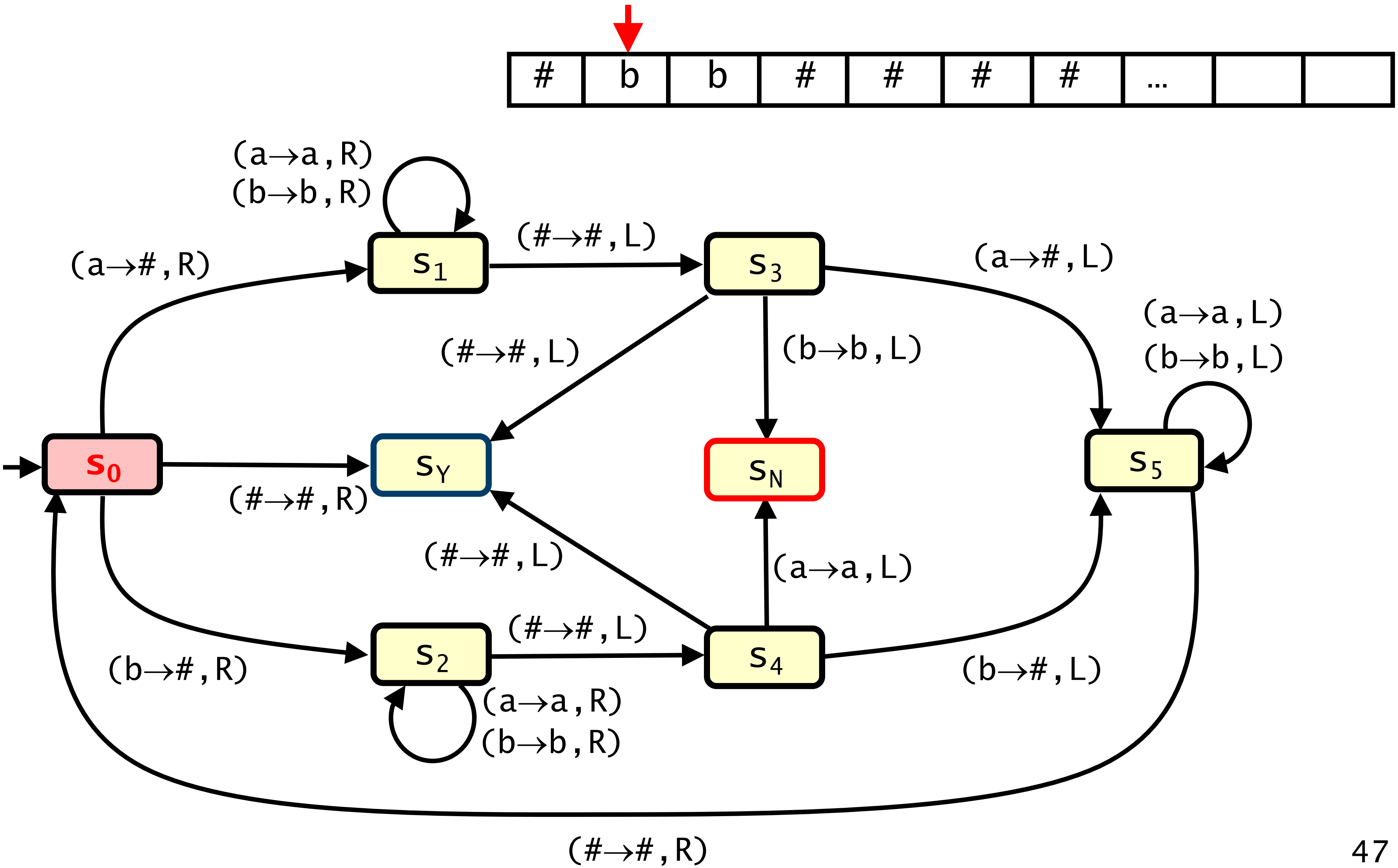
The palindrome problem – Turing machine



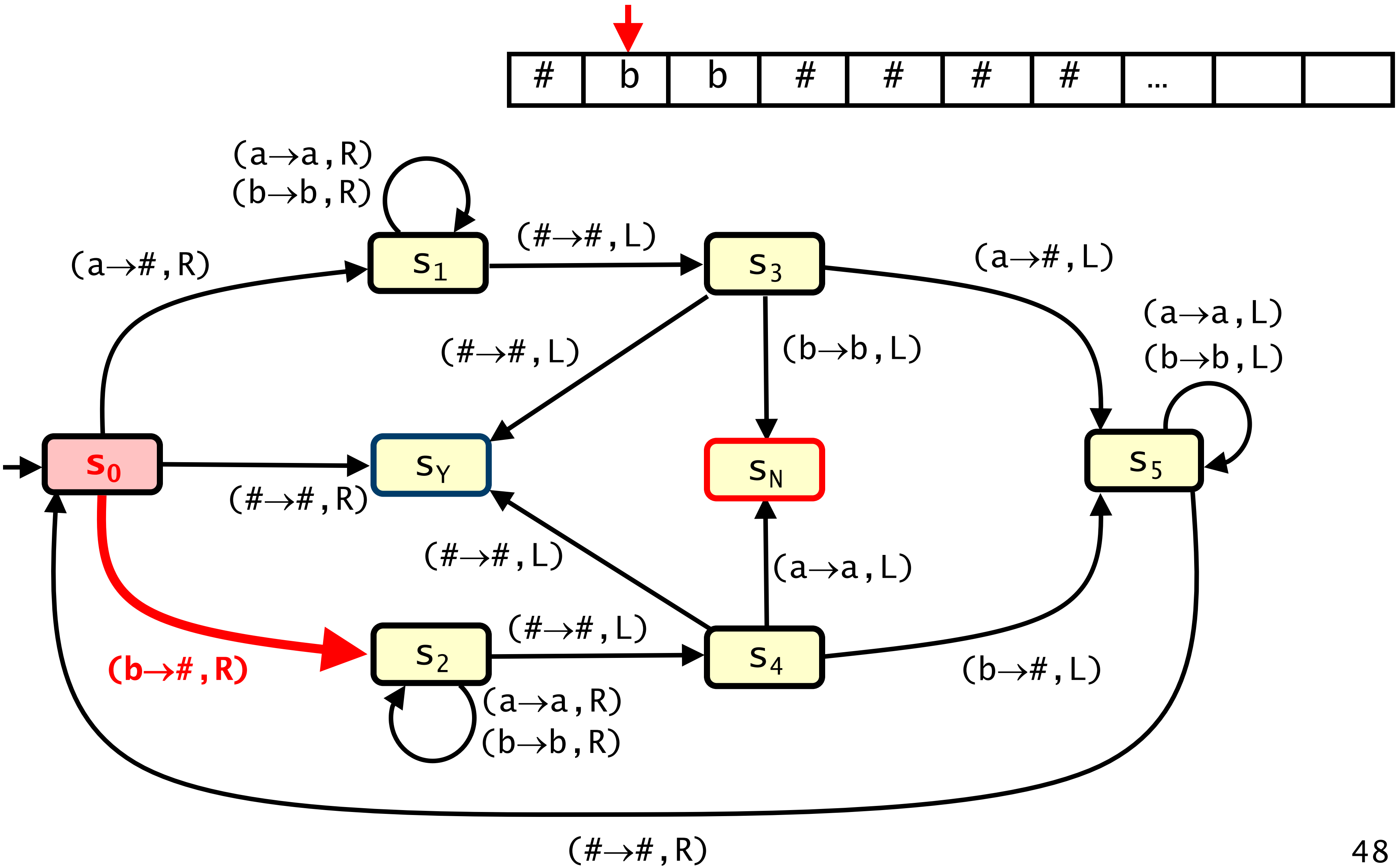
The palindrome problem – Turing machine



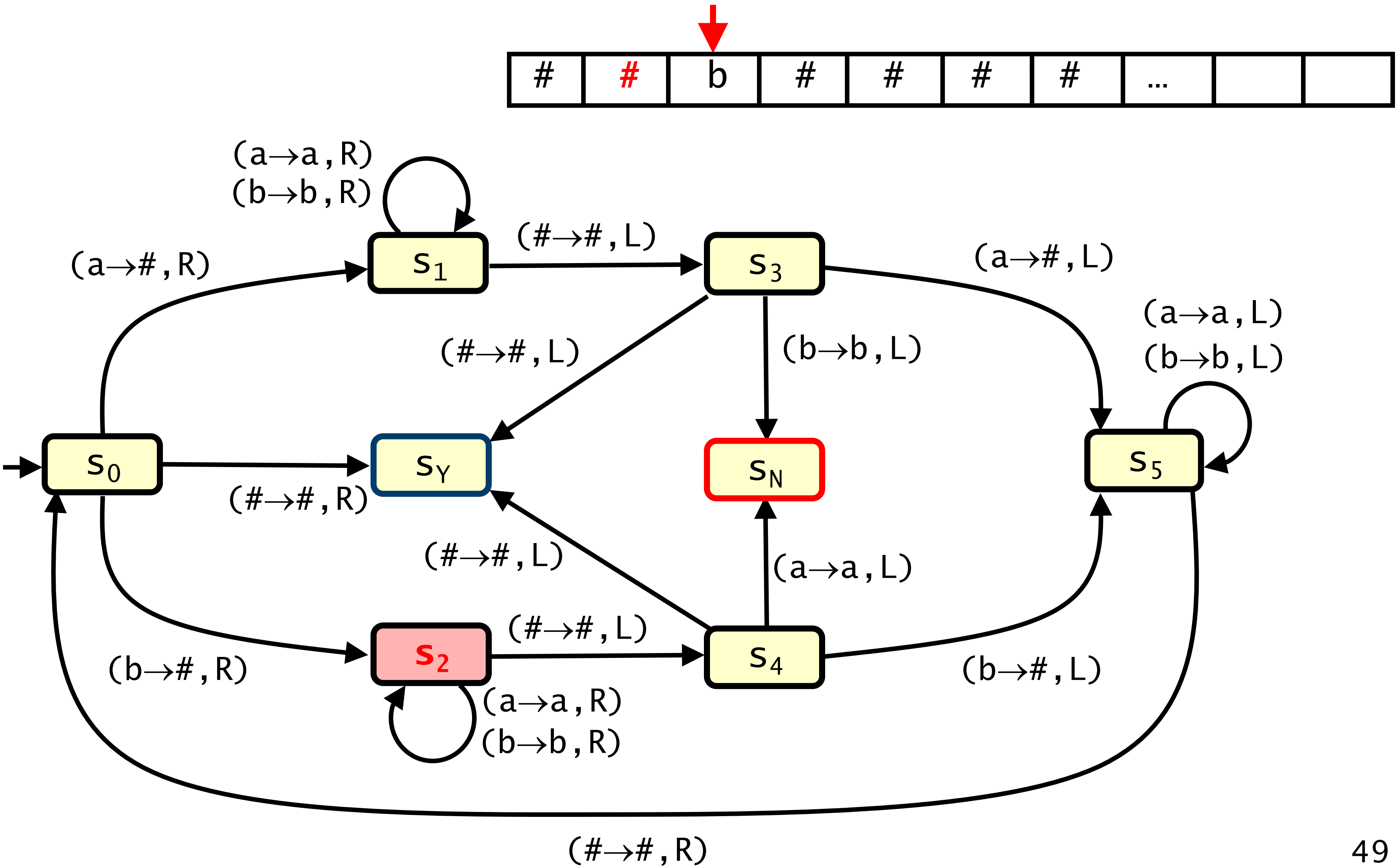
The palindrome problem – Turing machine



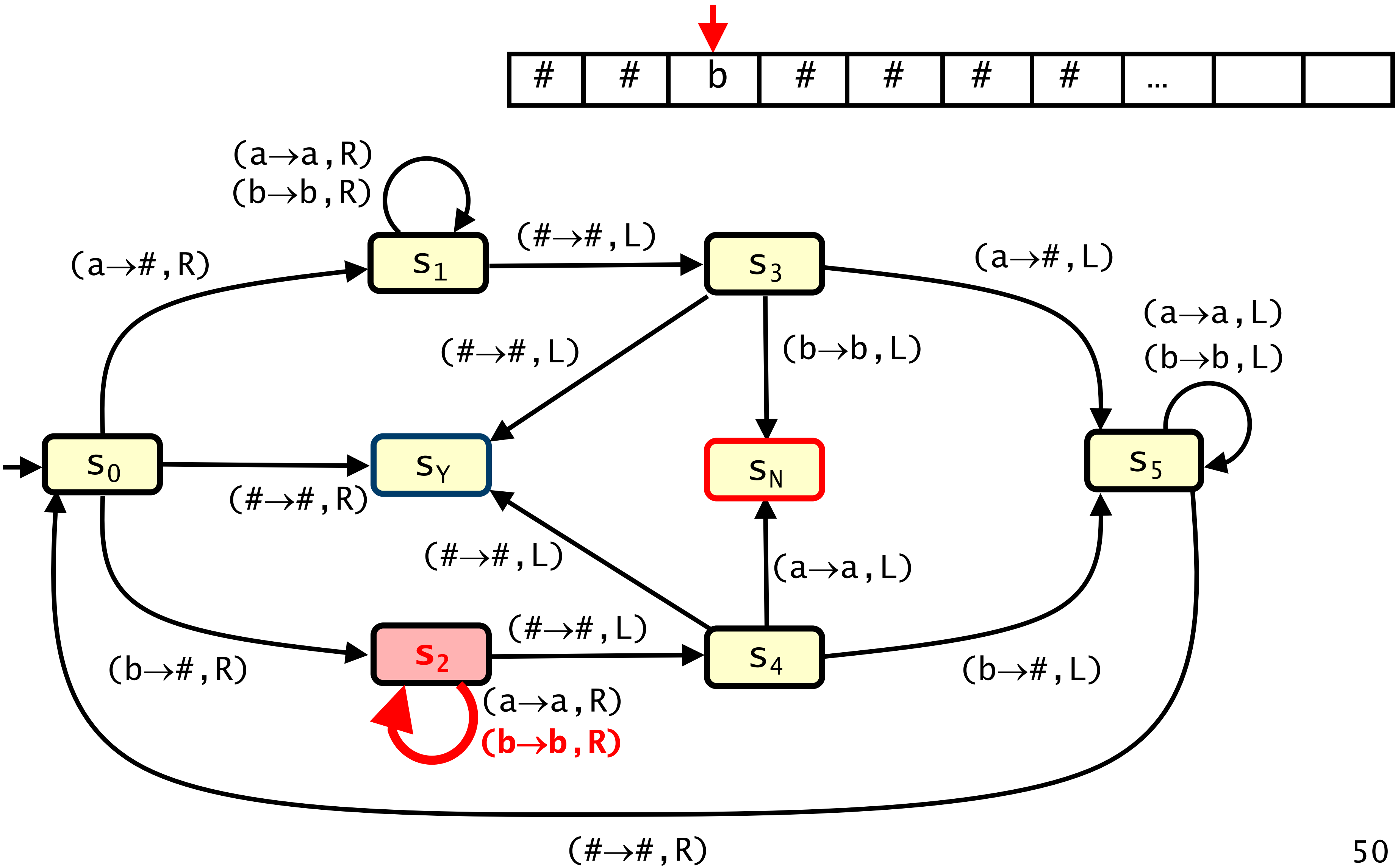
The palindrome problem – Turing machine



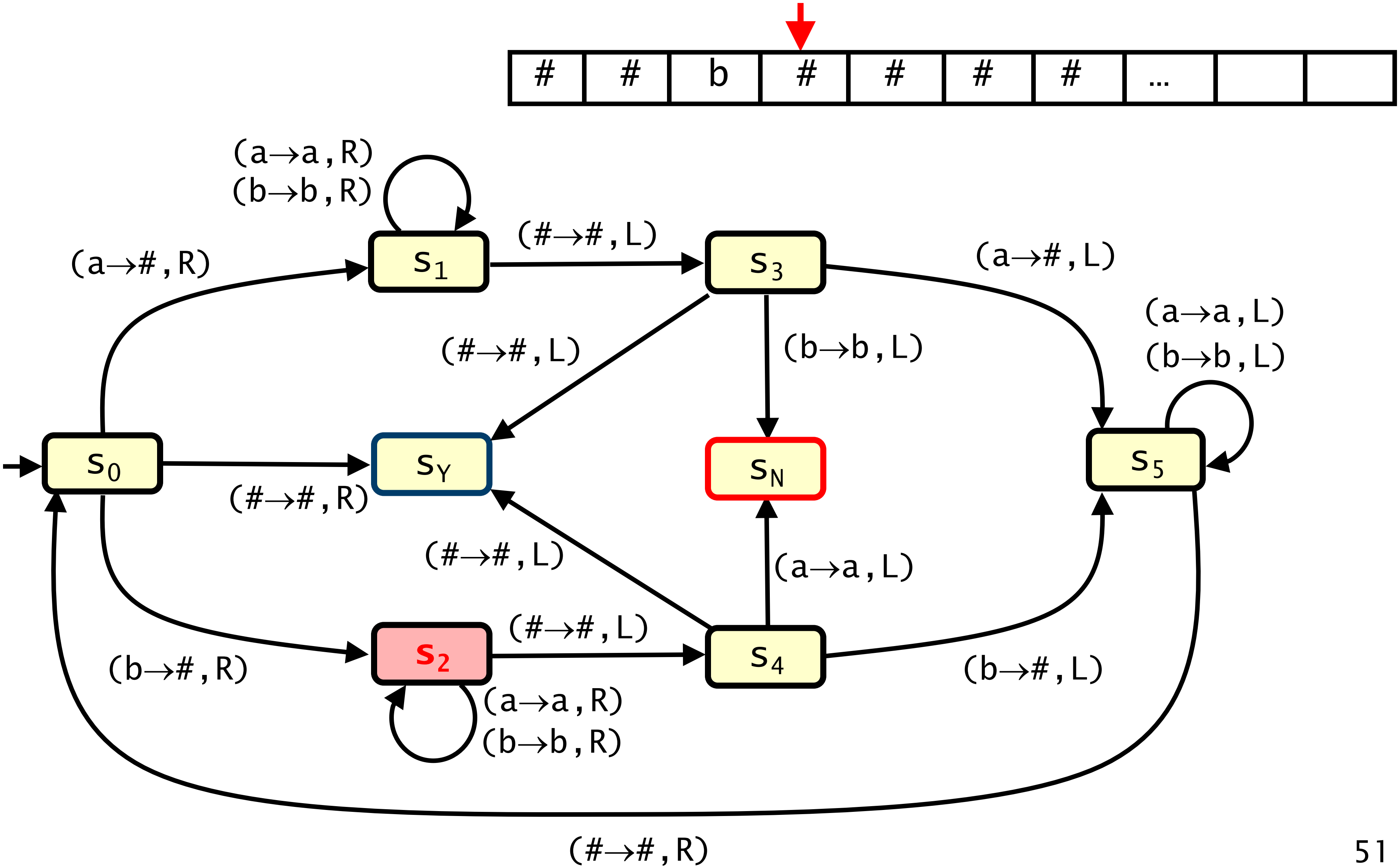
The palindrome problem – Turing machine



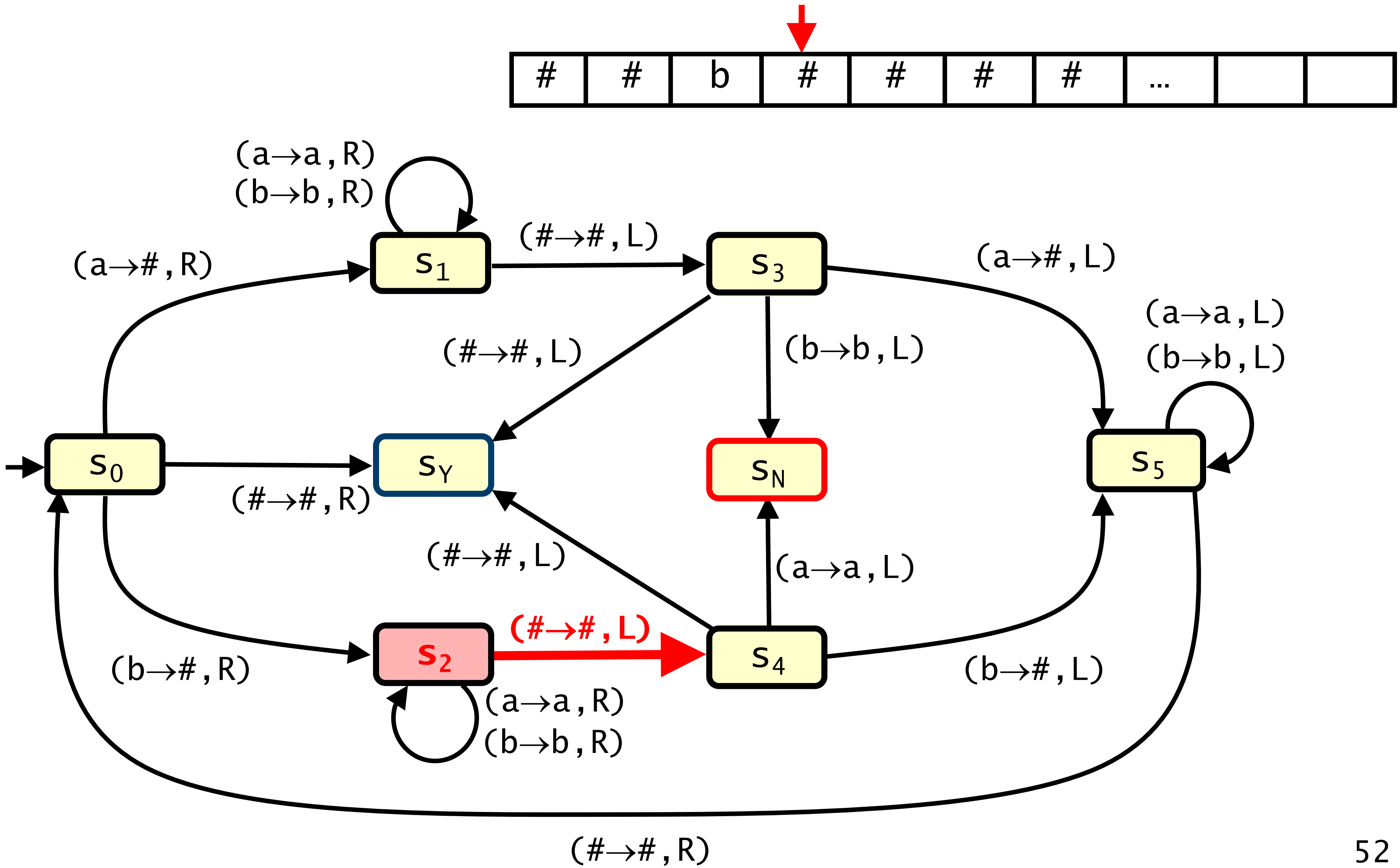
The palindrome problem – Turing machine



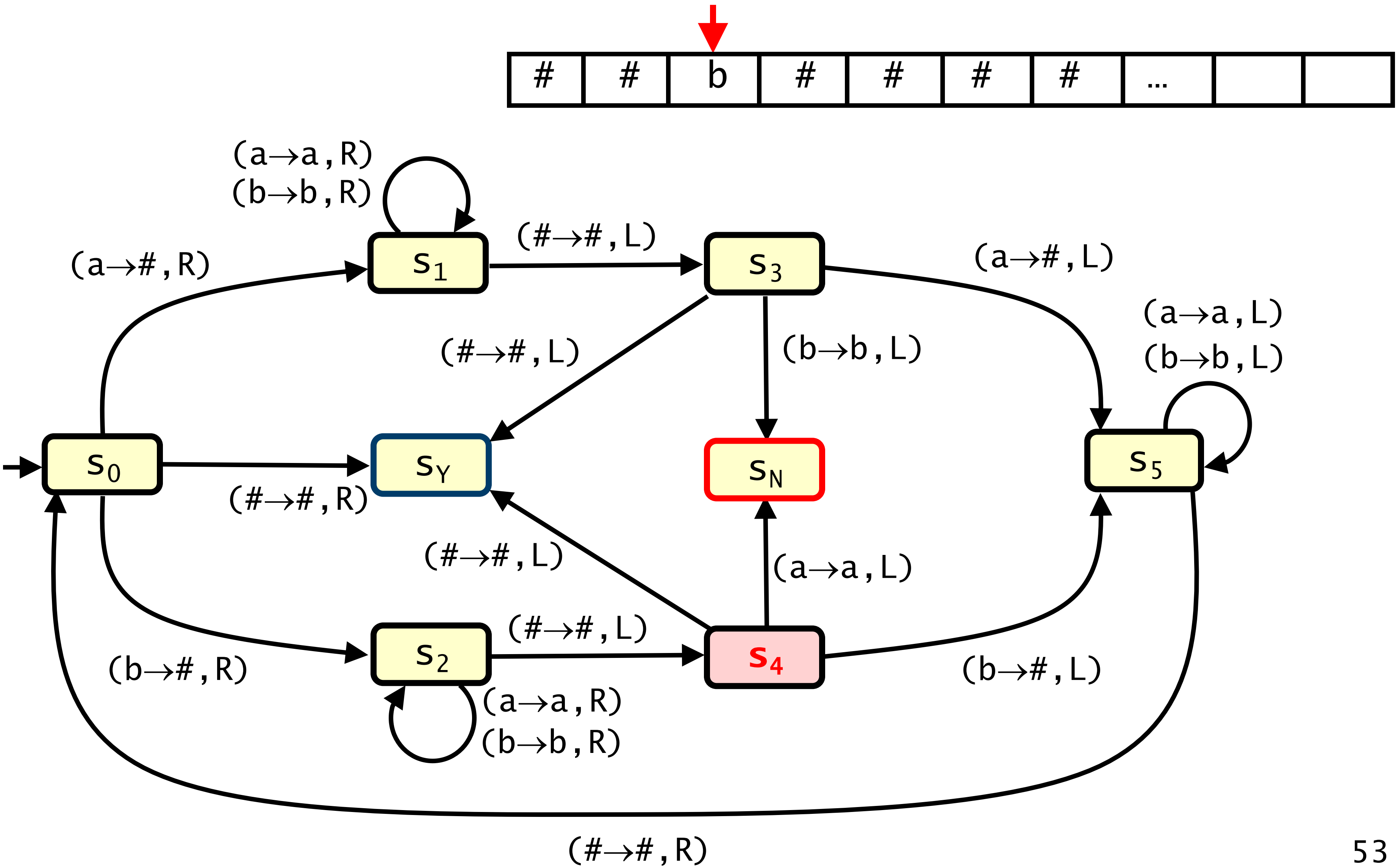
The palindrome problem – Turing machine



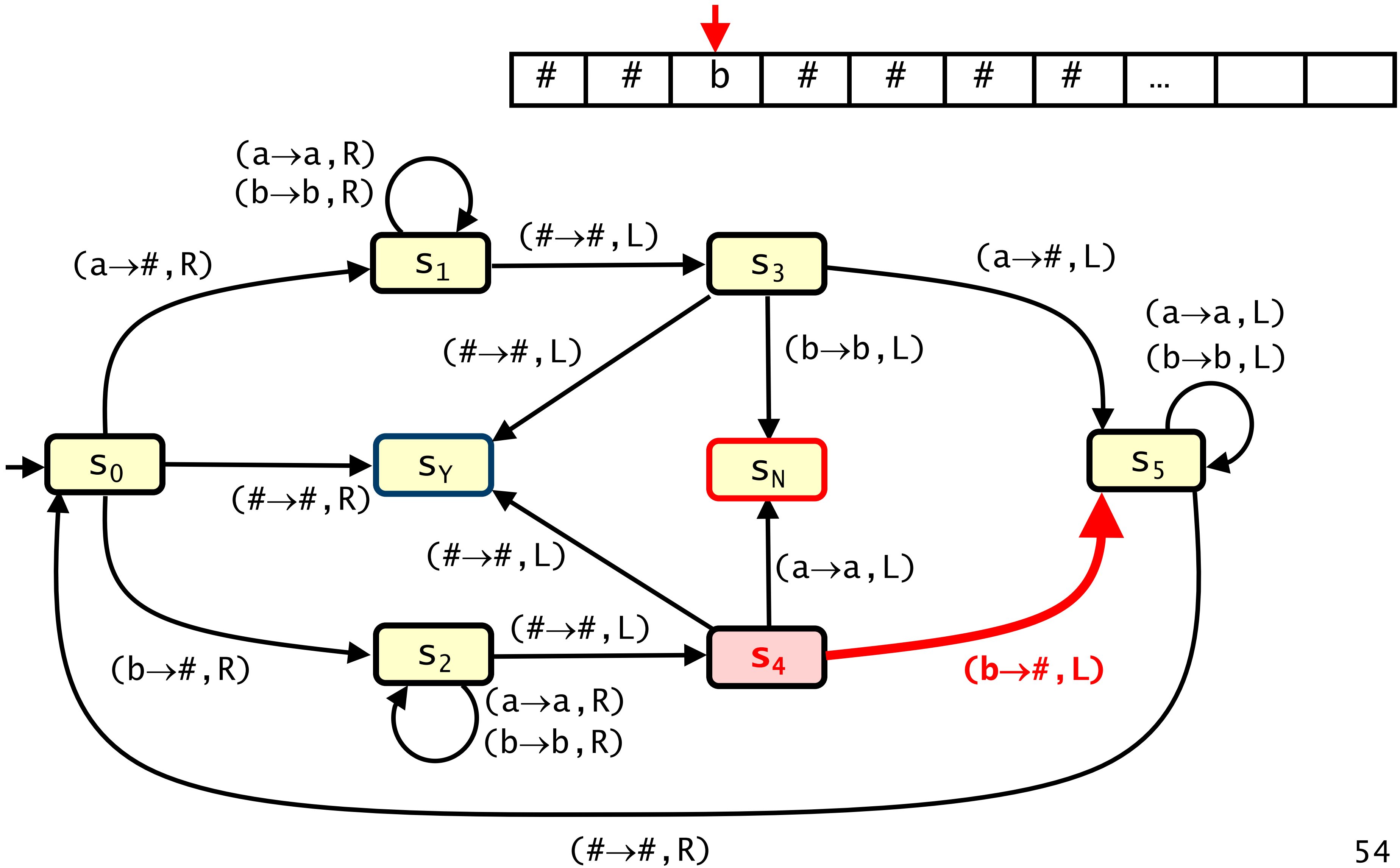
The palindrome problem – Turing machine



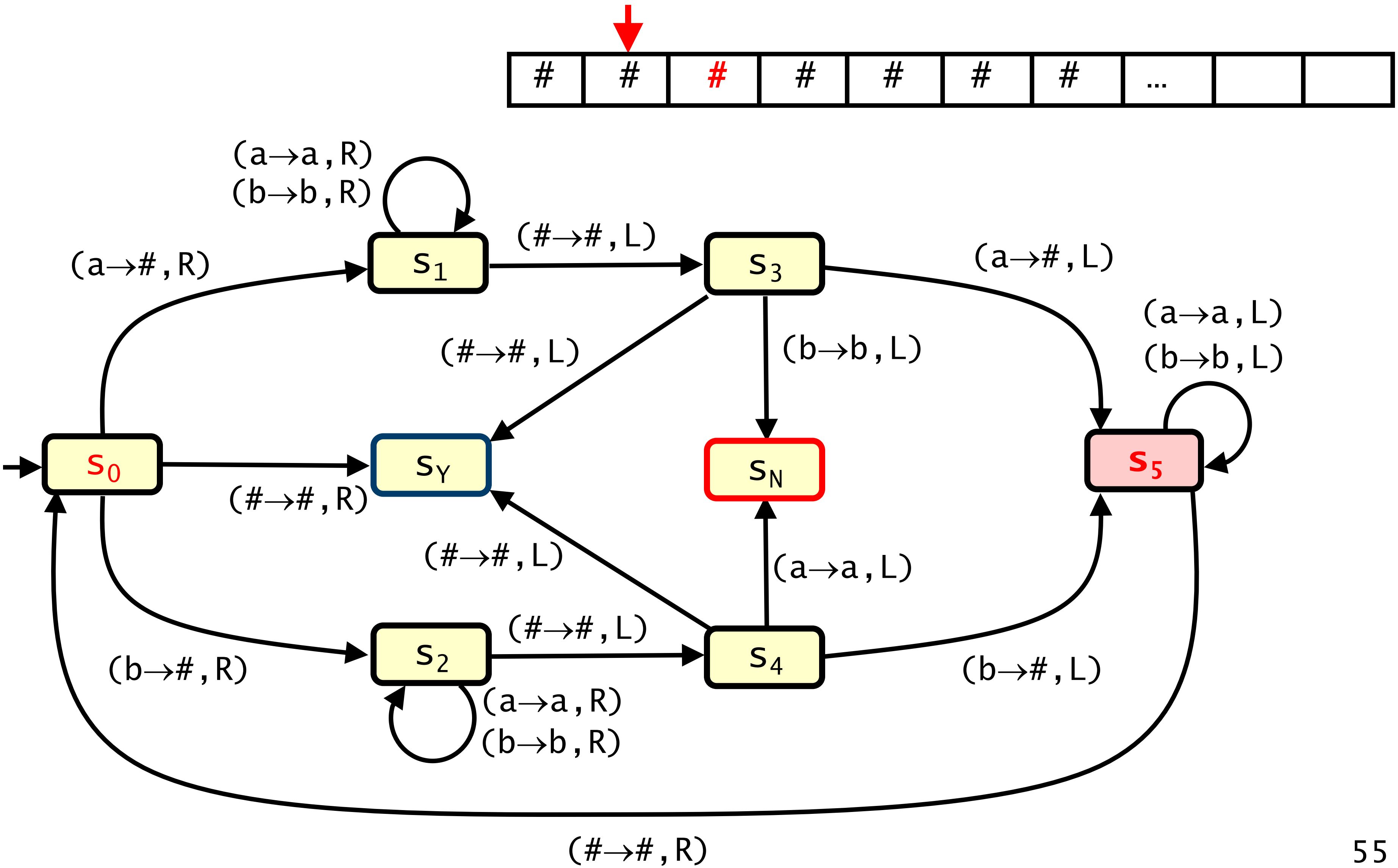
The palindrome problem – Turing machine



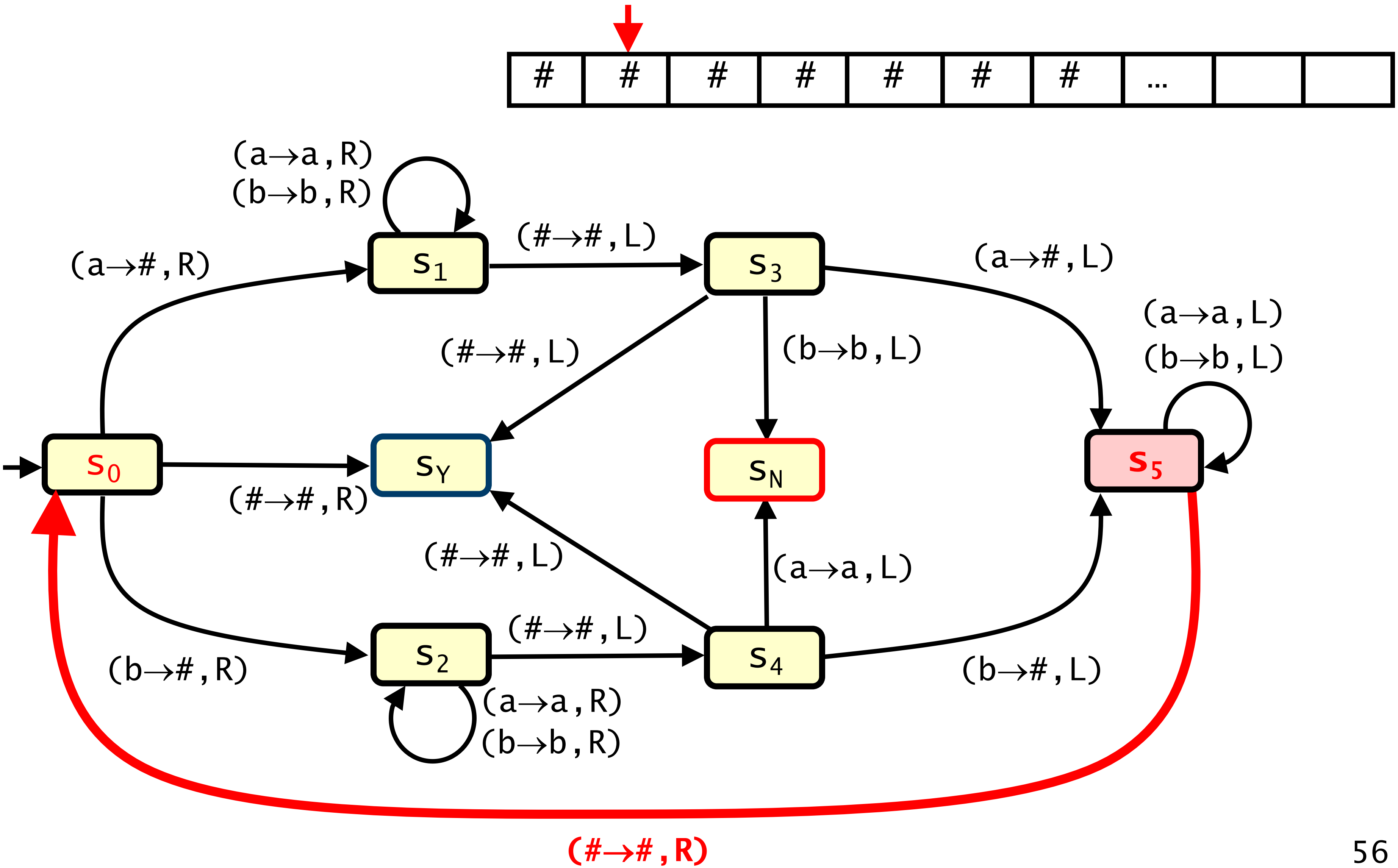
The palindrome problem – Turing machine



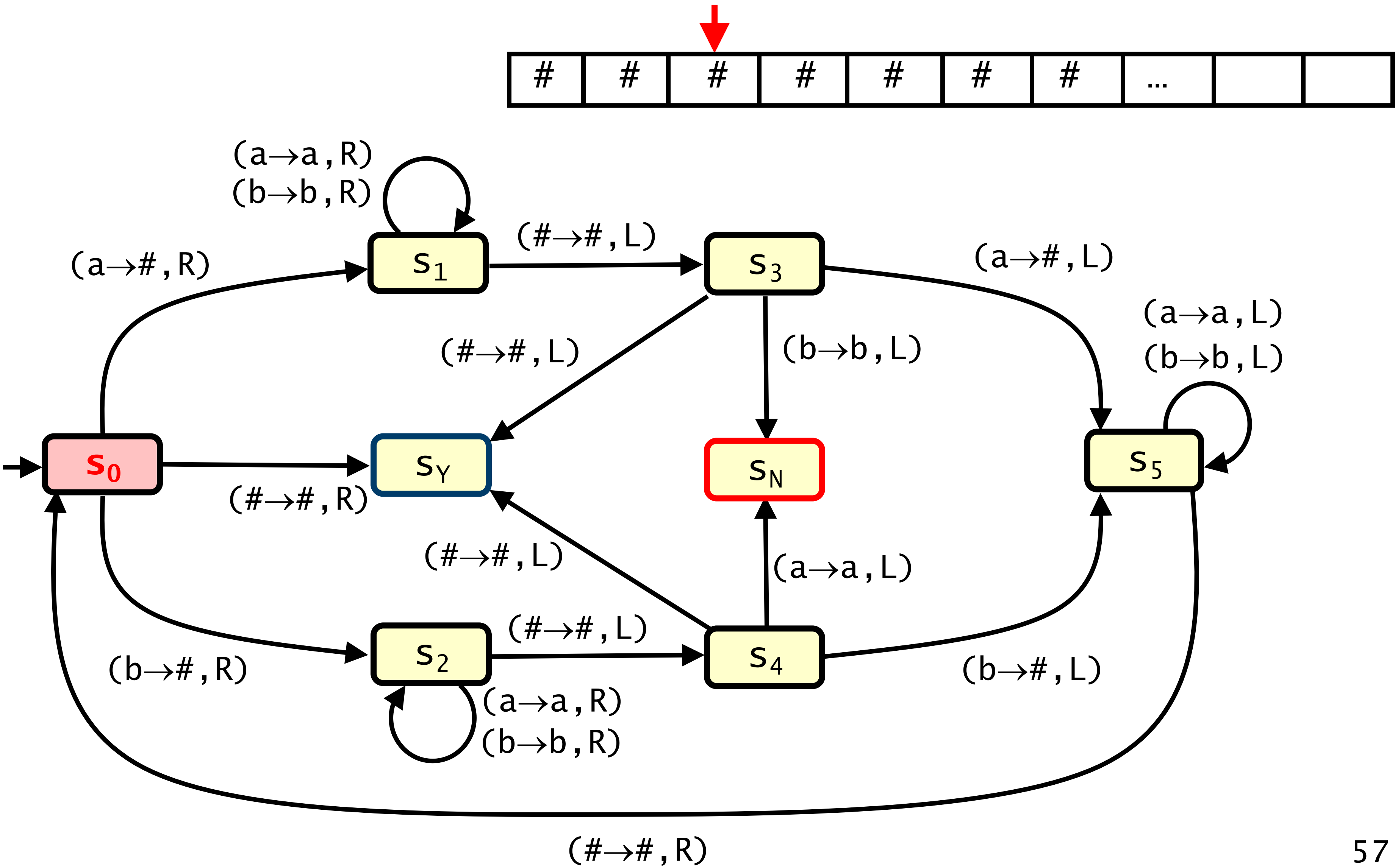
The palindrome problem – Turing machine



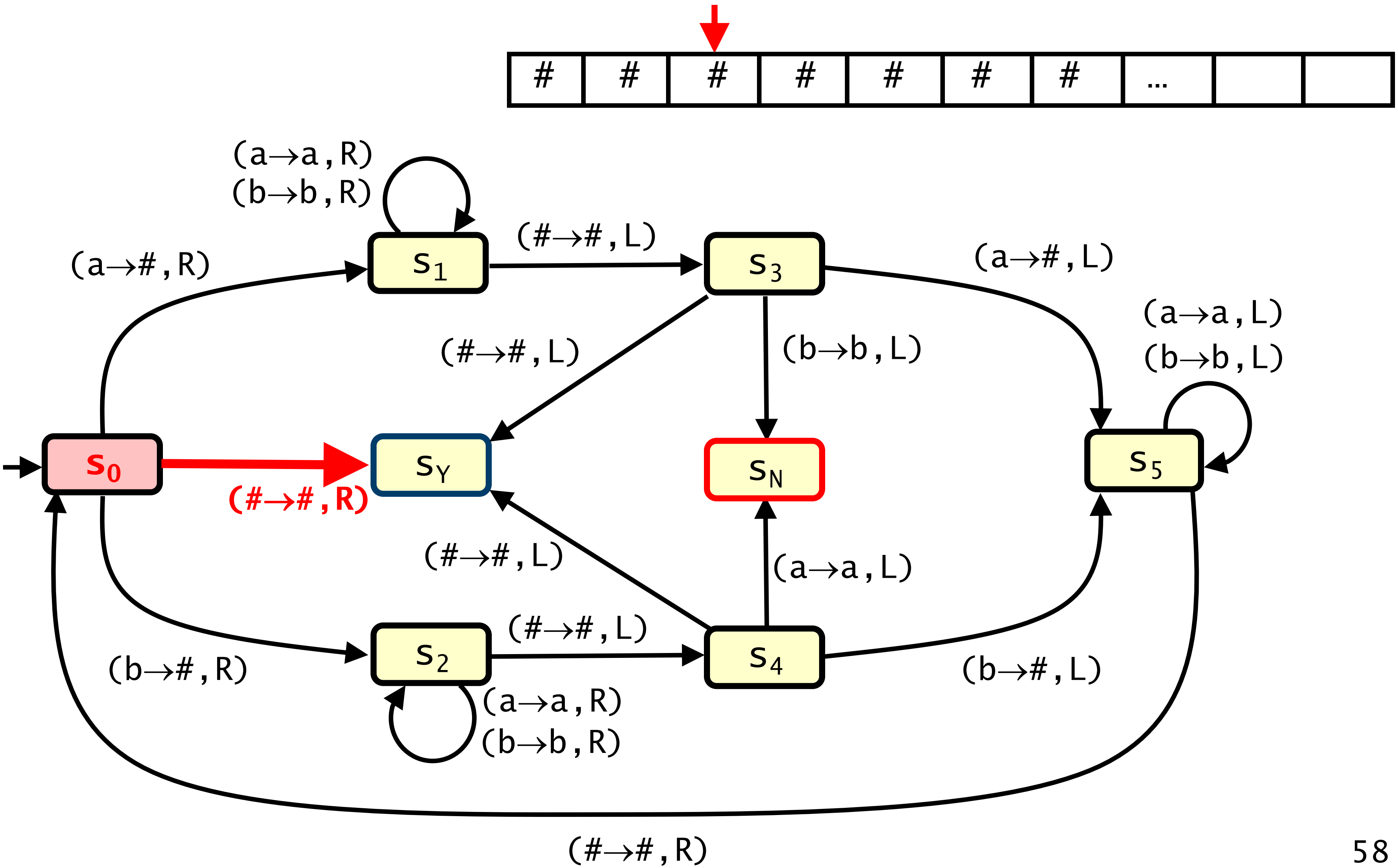
The palindrome problem – Turing machine



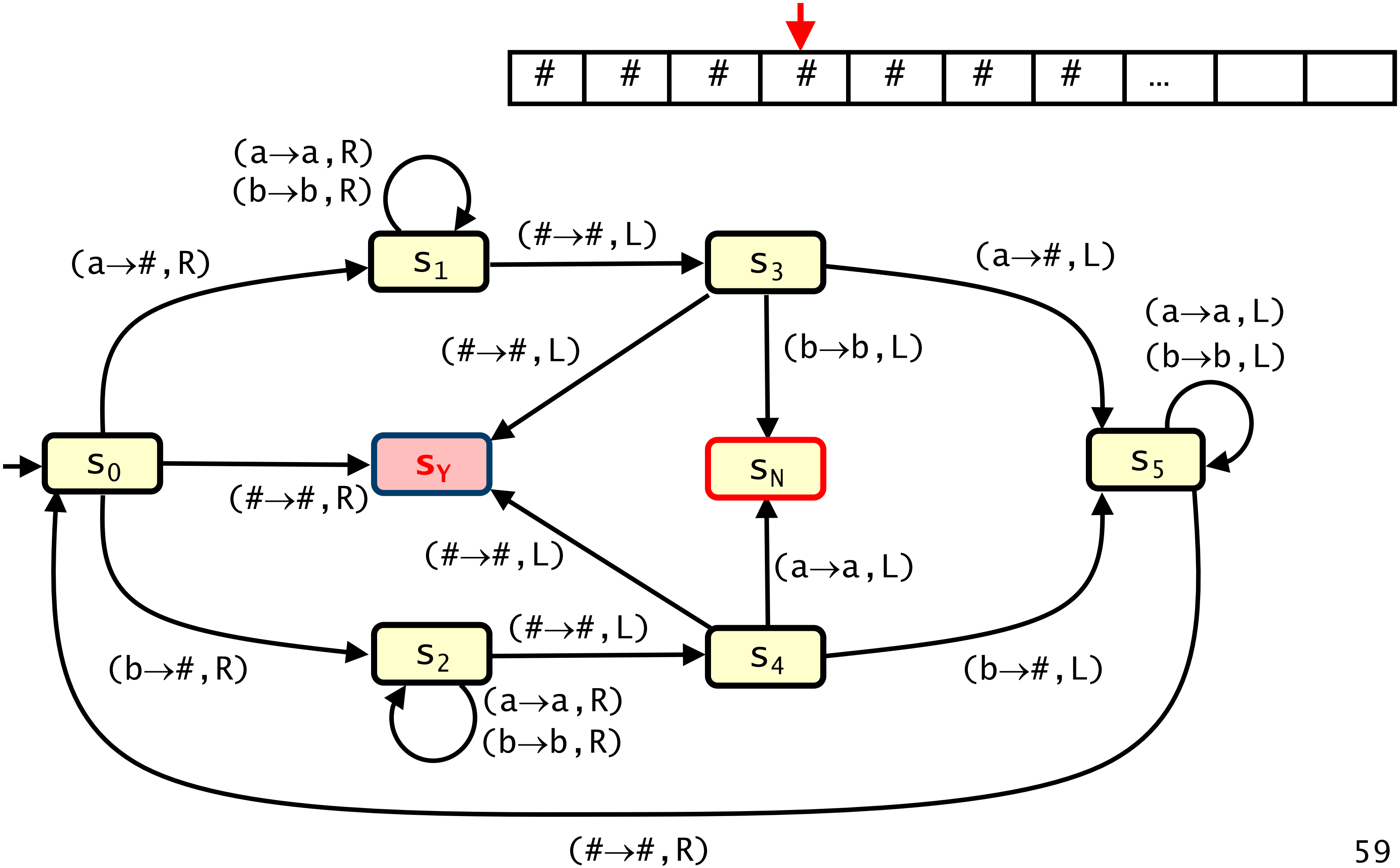
The palindrome problem – Turing machine



The palindrome problem – Turing machine



The palindrome problem – Turing machine



Turing machines – Functions

The Turing machine that accepts language L actually computes the function f where f(x) equals 1 if $x \in L$ and 0 otherwise

The definition of a TM can be amended as follows:

- to have a set H of halt states marking the end of the computation
- the function it computes is defined by $f(x)=y$ where
  - x is the initial string on the tape
  - y is the string on the tape when the machine halts

For example, the palindrome TM could be redefined such that it deletes the tape contents and

- instead of entering s_Y it writes 1 on the tape and enters a halt state
- instead of entering s_N it writes 0 on the tape and enters a halt state

Turing machines – Functions – Example

Design a Turing machine to compute the function `f(k) = k+1`

– where the input is in binary

Example 1

– input: 1 0 0 0 1 **0**

– output: 1 0 0 0 1 **1**

Example 2

– input: 1 0 **0** 1 1 1

– output: 1 0 **1** 0 0 0

Example 3 (special case)

– input 1 1 1 1 1

– output: 1 0 0 0 0 0

pattern: replace right-most 0 with 1
then moving **right**:
  if 1 replace with 0 and continue **right**
  if `blank` **halt**

special case: no right-most 0, i.e. only 1's in the input pattern:
replace first `blank` before input with 1
then moving right:
  if 1 replace with 0 and continue **right**
  if `blank` **halt**

Turing machines – Functions – Example

Design a Turing machine to compute the function `f(k) = k+1`

– where the input is in binary

TM Algorithm for the function `f(k) = k+1`

```
move right seeking first blank square;
move left looking for first 0 or blank;
when 0 or blank found
   change it to 1;
   move right changing each 1 to 0;
   halt when blank square reached;
```

Now to translate this pseudocode into a TM description

– identify the states and specify the transition function

Turing machines – Functions – Example

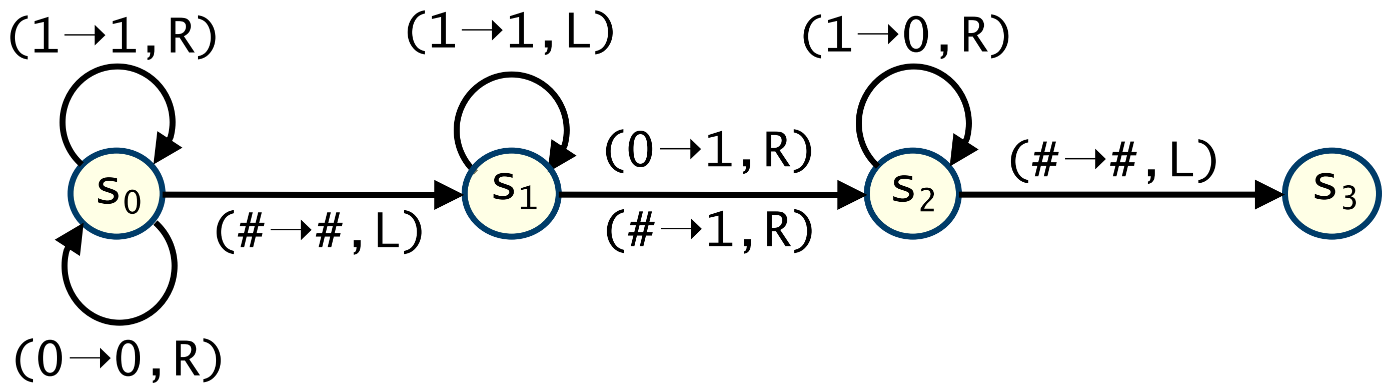
We need the following states

- s_0: (start state) moving right seeking start of the input (first blank)
- s_1: moving left to right-most 0 or blank
- s_2: find first 0 or blank, changed it to 1 and moving right changing 1s to 0s
- s_3: the halt state

and the following transitions

- from s_0 we enter s_1 at the first blank
- from s_1 we enter s_2 if a 0 (found right-most 0) or blank is read
- from s_2 we enter s_3 (halt) at the first blank

Transition state diagram



Exercise: execute this TM for inputs:

- 1 0 0 1 1 1
- 1 0 0 0 1 0
- 1 1 1 1 1

Turing recognizable and decidable

A language L is Turing-recognizable if some Turing Machine recognizes it, that is given an input string x:

- if $x \in L$, then the TM halts in state s_Y
- if $x \notin L$, then the TM halts in state s_N or **fails to halt (infinite loop)**

A language L is Turing-decidable if some Turing Machine decides it, that is given an input string x:

- if $x \in L$, then the TM halts in state s_Y
- if $x \notin L$, then the TM halts in state s_N

Every decidable language is recognizable, but not every recognizable language is decidable

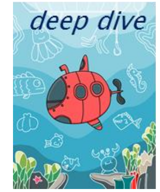
- e.g., the language corresponding to the Halting Problem
  (if a program terminates we will enter s_Y, but not s_N if it does not)

Turing computable

A function $f\colon \Sigma^* \to \Sigma^*$ is Turing-computable if there is a Turing machine M such that

- for any input x, the machine M halts with output $f(x)$

Enhanced Turing machines



A Turing machines may be enhanced in various ways:

- two or more tapes, rather than just one, may be available
- a 2-dimensional 'tape' may be available
- the TM may operate non-deterministically
  - i.e. the transition 'function' may be a relation rather than a function
- and many more ...

None of these enhancements change the computing power

- every language/function that is recognizable/decidable/computable with an enhanced TM is recognizable/decidable/computable with a basic TM (probably more difficult to construct and longer executions)
  - so nondeterminism adds power to pushdown automata but neither to finite-state automata or Turing machines...
- proved by showing that a basic TM can simulate any of these enhanced Turing machines

Turing machines – P and NP

The class P is often introduced as the class of decision problems solvable by a Turing machine in polynomial time

and the class NP is introduced as the class of decision problems solvable by a non-deterministic Turing machine in polynomial time

- in a non-deterministic TM the transition function is replaced by a relation

  $f \subseteq ((S \times \Sigma) \times (S \times \Sigma \times \{Left, Right\}))$

  i.e. can make a number of different transitions based on the current state and the symbol at the tape head
- nondeterminism does not change what can be computed, but can speed up the computation

Hence to show $P \neq NP$ sufficient to show a (standard) Turing machine cannot solve an NP-complete problem in polynomial time

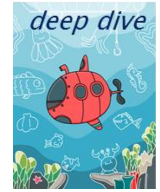
Section 5 – Computability

Introduction

Models of computation

- finite-state automata
- pushdown automata
- Turing machines
- Counter programs
- Church-Turing thesis

Counter programs (register machines)



A completely different model of computation

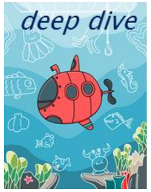
- no states, no transitions and no accepting states
- some variants equivalent to the computational power of Turing machines

All general-purpose programming languages have essentially the same computational power

- a program written in one language (e.g., C) could be translated (or compiled) into a functionally equivalent program in any other (Python)

So how simple can a programming language be and still have the same computational power as C, Java, Python, etc.?

Counter programs



Counter programs have

- variables of type `int`
- labelled statements are of the form:
  - `L : unlabelled_statement`
- unlabelled statements are of the form:
  - `x = 0;` (set a variable to zero)
  - `x = y+1;` (set a variable to be the value of another variable plus 1)
  - `x = y-1;` (set a variable to be the value of another variable minus 1)
  - `if x==0 goto L;` (conditional goto where L is a label of a statement)
  - `halt;` (finished)

Counter programs – Example

A counter program to evaluate the product **x·y**

(A, B and C are labels and have variables x, y, u, v and z)

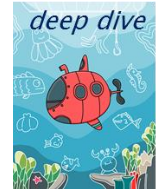
```
// initialise some variables
u = 0; // dummy variable (always equals 0)
z = 0; // this will be the product of x and y when we finish

A: if x==0 goto C; // end of outer for loop
  x = x-1; // perform this loop x times
  v = y+1; // each time around the loop we set v to equal y
  v = v-1; // in a slightly contrived way

B: if v==0 goto A; // end of inner for loop (return to outer loop)
  v = v-1; // perform this loop v times (i.e. y times)
  z = z+1; // each time incrementing z
          // so really added y to z by the end of the inner loop
  if u==0 goto B; // really just goto B (return to start of inner loop)

C: halt;
```

Counter programs – Example



A counter program to evaluate the product x·y

(A, B and C are labels and have variables x, y, u, v and z)

```
// initialise some variables
u = 0; // dummy variable (always equals 0)
z = 0; // this will be the product of x and y when we finish

A: if x==0 goto C; // end of outer for loop
  x = x-1; // perform this loop x times
  v = y+1; // each time around the loop we set v to equal y
  v = v-1; // in a slightly contrived way

B: if v==0 goto A; // end of inner for loop (return to outer loop)
  v = v-1; // perform this loop v times (i.e. y times)
  z = z+1; // each time incrementing z
          // so really added y to z by the end of the inner loop
  if u==0 goto B; // really just goto B (return to start of inner loop)

C: halt;
```

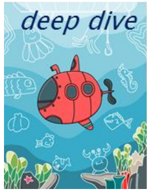
Try this out with an example, say 3 times 4!

Section 5 – Computability

Introduction

Models of computation

- finite-state automata
- pushdown automata
- Turing machines
- Counter machines
- Church-Turing thesis



David Hilbert's 10th problem

Devising an "algorithm" that tests whether a polynomial has an integral root

- did not say "algorithm" but rather devising a process according to which it can be determined by a finite number of operations
- https://en.wikipedia.org/wiki/Hilbert%27s_problems
- later proved to be unsolvable (1944–1970)
- proving that an algorithm does not exist requires having a clear definition of algorithm
- algorithms are defined in 1936
  - Church: lambda calculus
  - Turing: machines
- the Church–Turing thesis

  provides the definition of algorithm

Hilbert's problems are 23 problems in mathematics published by German mathematician David Hilbert in 1900. They were all unsolved at the time, and several proved to be very influential for 20th-century mathematics. Hilbert presented ten of the problems (1, 2, 6, 7, 8, 13, 16, 19, 21, and 22) at the Paris conference of the International Congress of Mathematicians, speaking on August 8 at the Sorbonne. The complete list of 23 problems was published later, in English translation in 1902 by Mary Frances Winston Newson in the *Bulletin of the American Mathematical Society*.[1]

Alonzo Church and Alan Turing

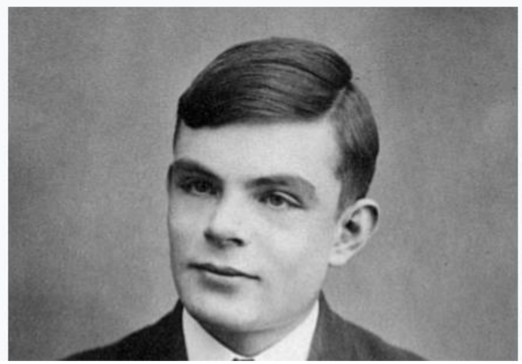
Alonzo Church (June 14, 1903 – August 11, 1995) was an American mathematician, computer scientist, logician, and philosopher who made major contributions to mathematical logic and the foundations of theoretical computer science.[2] He is best known for the lambda calculus, the Church–Turing thesis, proving the unsolvability of the Entscheidungsproblem, the Frege–Church ontology, and the Church–Rosser theorem. He also worked on philosophy of language (see e.g. Church 1970). Alongside his doctoral student Alan Turing, Church is considered one of the founders of computer science.[3][4]

Alonzo Church

https://en.wikipedia.org/wiki/Alonzo_Church

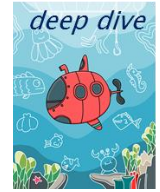
Alan Mathison Turing OBE FRS (/ˈtjʊərɪŋ/; 23 June 1912 – 7 June 1954) was an English mathematician, computer scientist, logician, cryptanalyst, philosopher, and theoretical biologist.[6] Turing was highly influential in the development of theoretical computer science, providing a formalisation of the concepts of algorithm and computation with the Turing machine, which can be considered a model of a general-purpose computer.[7][8][9] He is widely considered to be the father of theoretical computer science and artificial intelligence.[10]

Alan Turing
OBE FRS



https://en.wikipedia.org/wiki/Alan_Turing

The Church–Turing Thesis



Is the Turing machine an appropriate model for the ‘black box’?

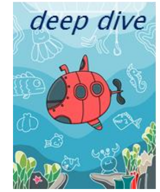
The answer is ‘yes’ this is known as the Church–Turing thesis

- it is based on the fact that a whole range of different computational models turn out to be equivalent in terms of what they can compute
- so it is reasonable to infer that any one of these models encapsulates what is effectively computable

Put simply it states that everything “effectively computable” is computable by a Turing machine

- a thesis not a theorem as uses the informal term “effectively computable”
  - future technologies might change what that means
- means there is an effective procedure for computing the value of the function including all computers/programming languages that we know about at present and even those that we do not

The Church–Turing Thesis



So is the Turing machine an appropriate model for the ‘black box’?

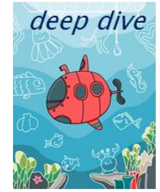
The answer is ‘yes’ this is known as the Church–Turing thesis

– it is based on the fact that a whole range of different computational models turn out to be equivalent in terms of what they can compute
– so it is reasonable to infer that any one of these models encapsulates what is effectively computable

Equivalent computational models (each can 'simulate' all others)

– Lambda calculus (Alonzo Church)
– Turing machines (Alan Turing)
– Recursive functions (Stephen Kleene)
– Production systems (Emil Post)
– Counter programs and all general purpose programming languages

Turing Machines for SE undergrads



Foundational understanding of computation

- Turing machines provide a fundamental model for understanding what is computationally possible
- define the limits of computation and decision-making processes

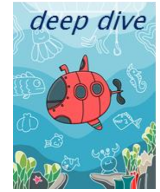
Algorithmic thinking and efficiency

- insights from Turing machines enhance algorithmic thinking - rigorous logical thinking and problem decomposition
- critical for algorithm design, optimization, and understanding computational complexity in software development

Basis for Formal Methods

- knowledge of Turing machines underpins formal methods in software engineering, crucial for formal verification, model checking, and ensuring the reliability and correctness of systems
- see my Honours course Modelling Reactive Systems 😁

Turing Machines for SE undergrads



Conceptual framework for AI and ML

- understanding Turing machines offers a theoretical basis for the development and analysis of AI and Machine Learning algorithms
- provides insights into neural networks' computational capabilities

Ethical and philosophical insights

- the study of Turing machines and their implications for AI prompts reflection on the ethical, philosophical, and societal impacts of software and technology development
- including considerations of the Turing Test as a measure of a machine's ability to exhibit intelligent behavior indistinguishable from that of a human, raising questions about the nature of intelligence, consciousness, and the relationship between humans and machines

Preparation for advanced computational topics:

- familiarity with Turing machines paves the way for advanced topics in computing, including quantum computing and the exploration of new computational models

Outline of course

Section 0: Quick recap on algorithm analysis – individual study

Section 1: Sorting algorithms

Section 2: Strings and text algorithms

Section 3: Graphs and graph algorithms

Section 4: An introduction to NP completeness

Section 5: A (very) brief introduction to computability

Revision plan

May 2025

- 2x30 mins revision sessions at lunch time -TBC
- available during the week: ask questions by email/Teams/Padlet, ask for a meeting
- exam ?? May 2024 at ?? (or ?? for those entitled to extra time), duration 60 minutes