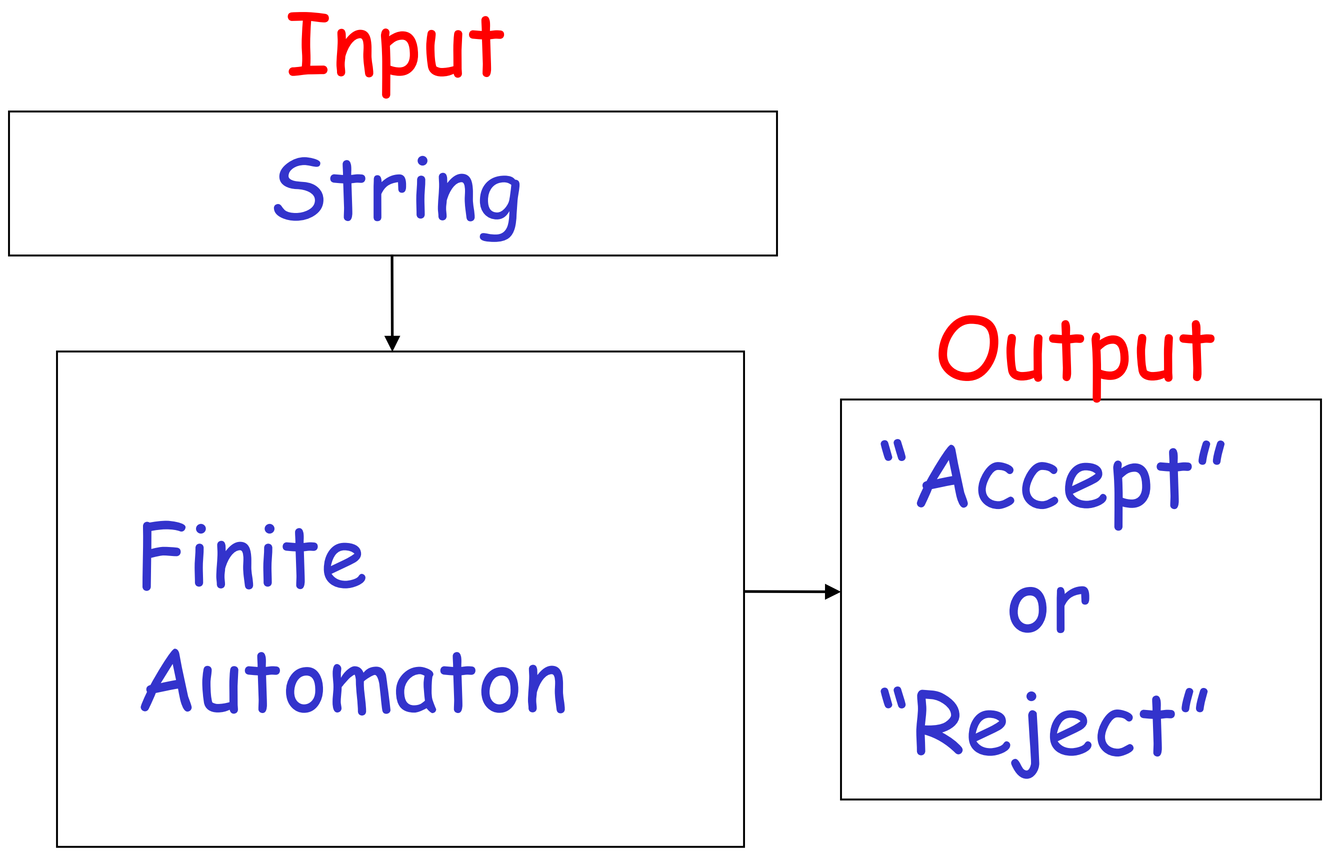
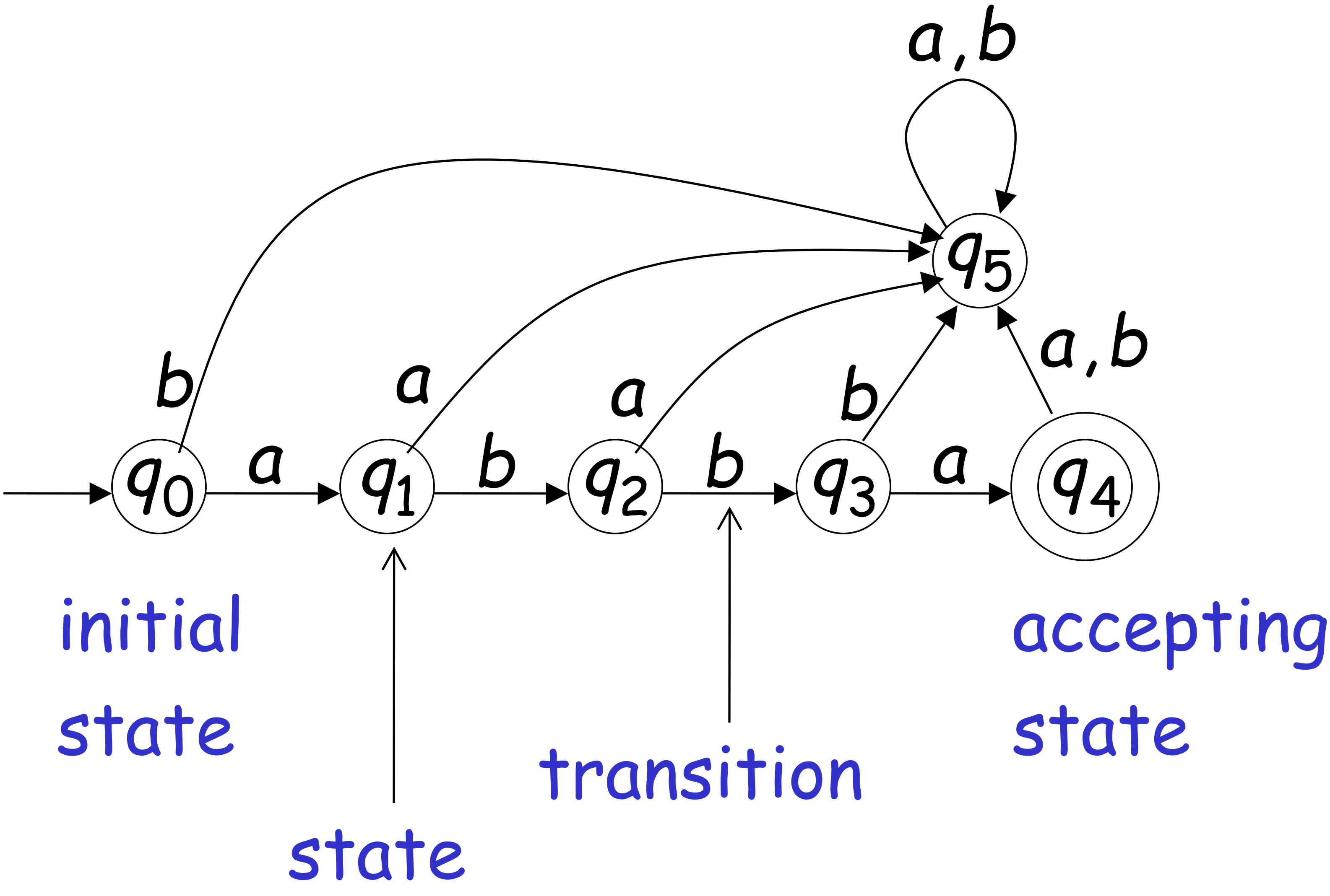
Graph Applications
Pattern Recognition
Automata

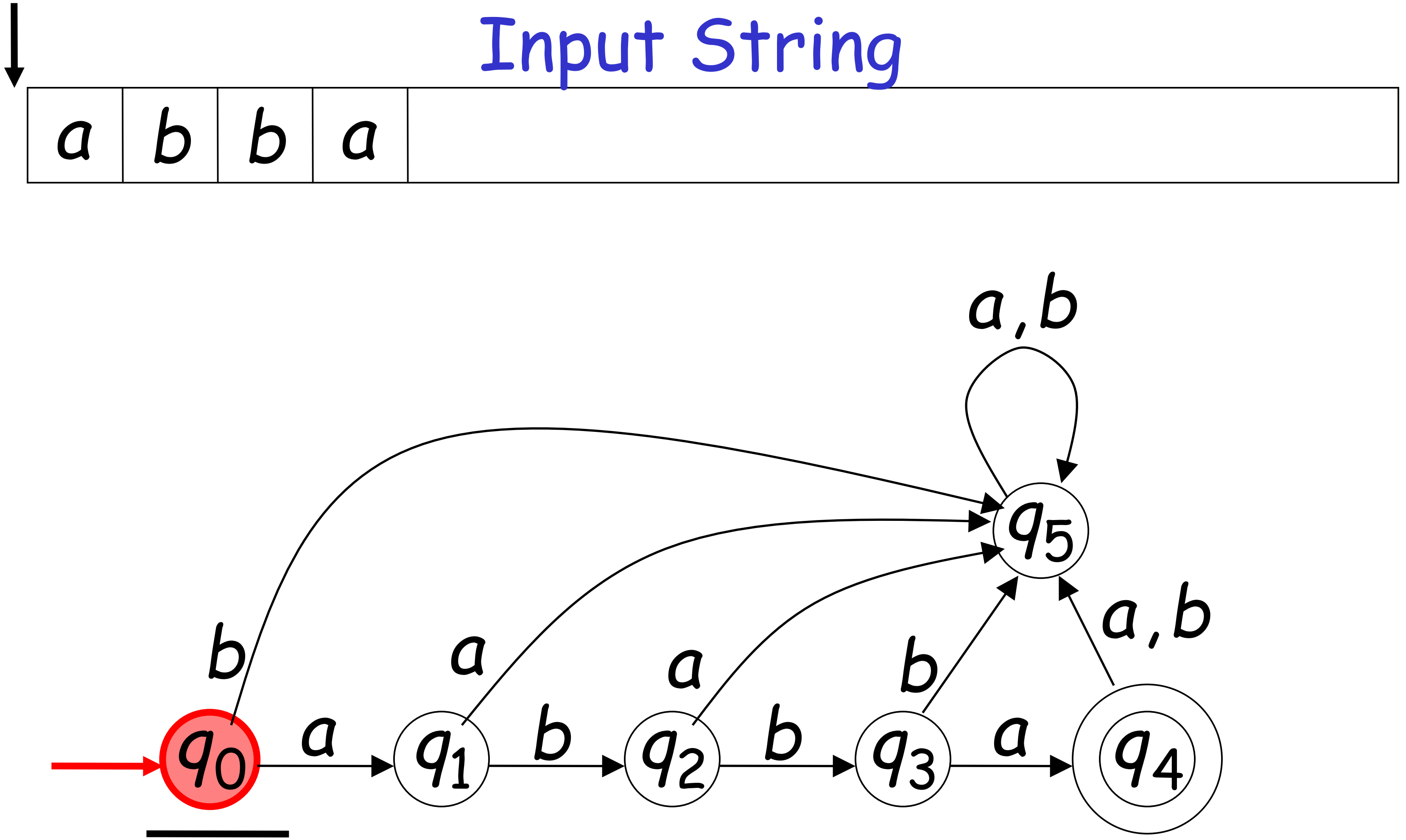
Finite Automaton



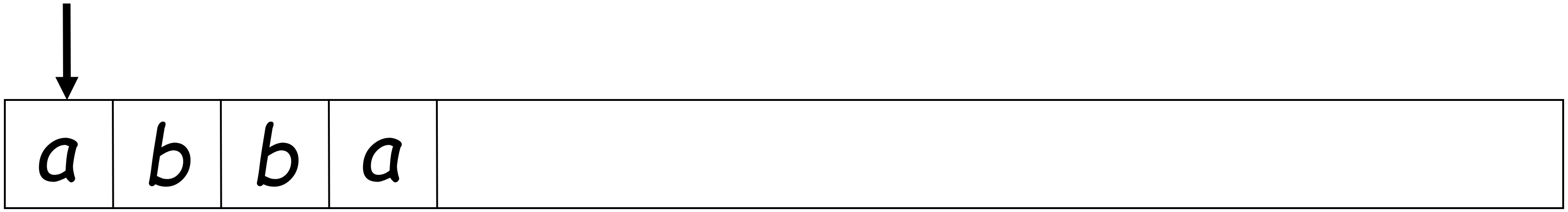
Transition Graph

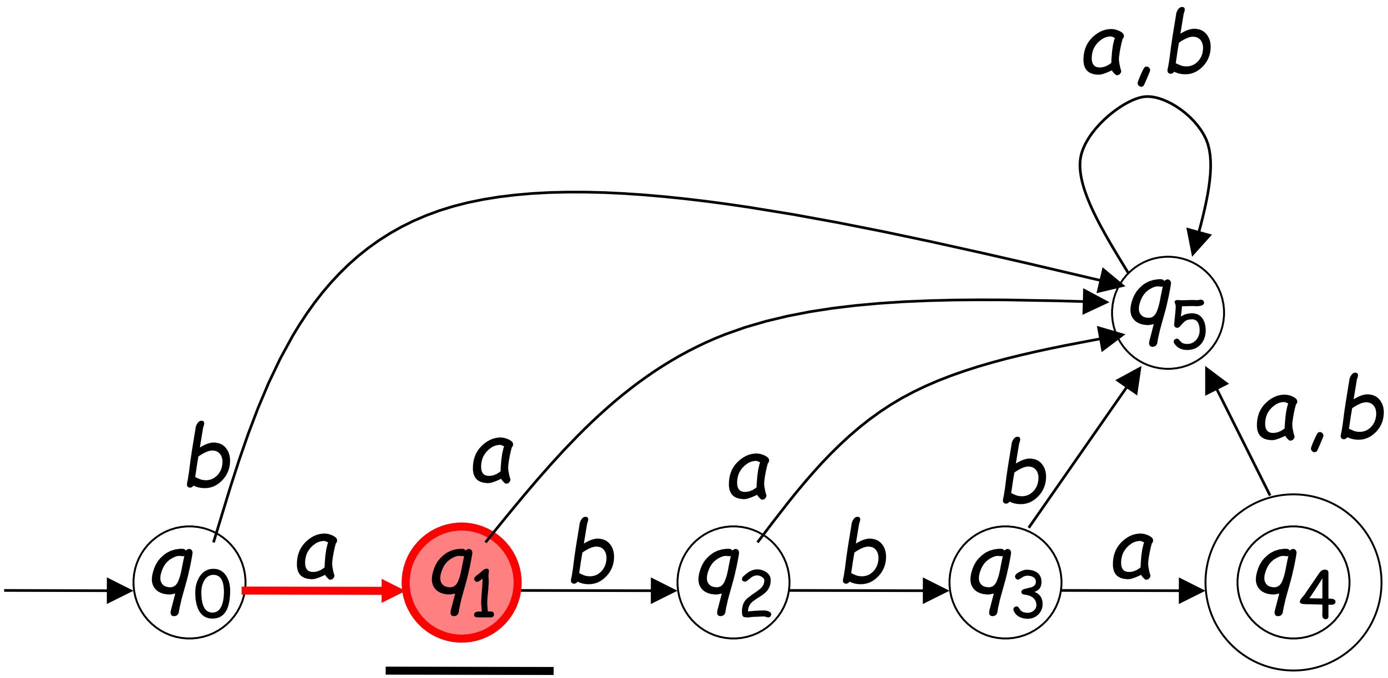


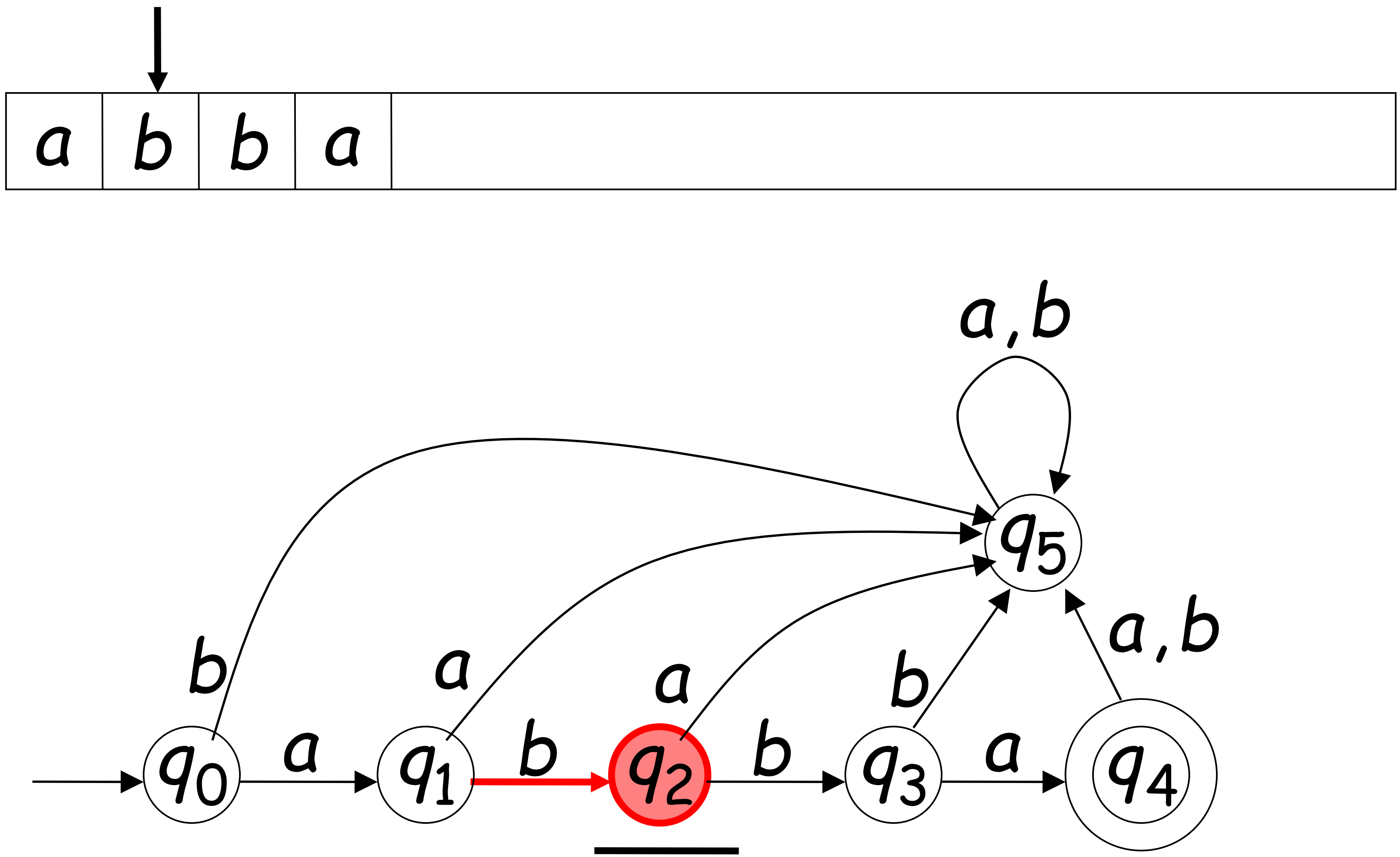
Initial Configuration

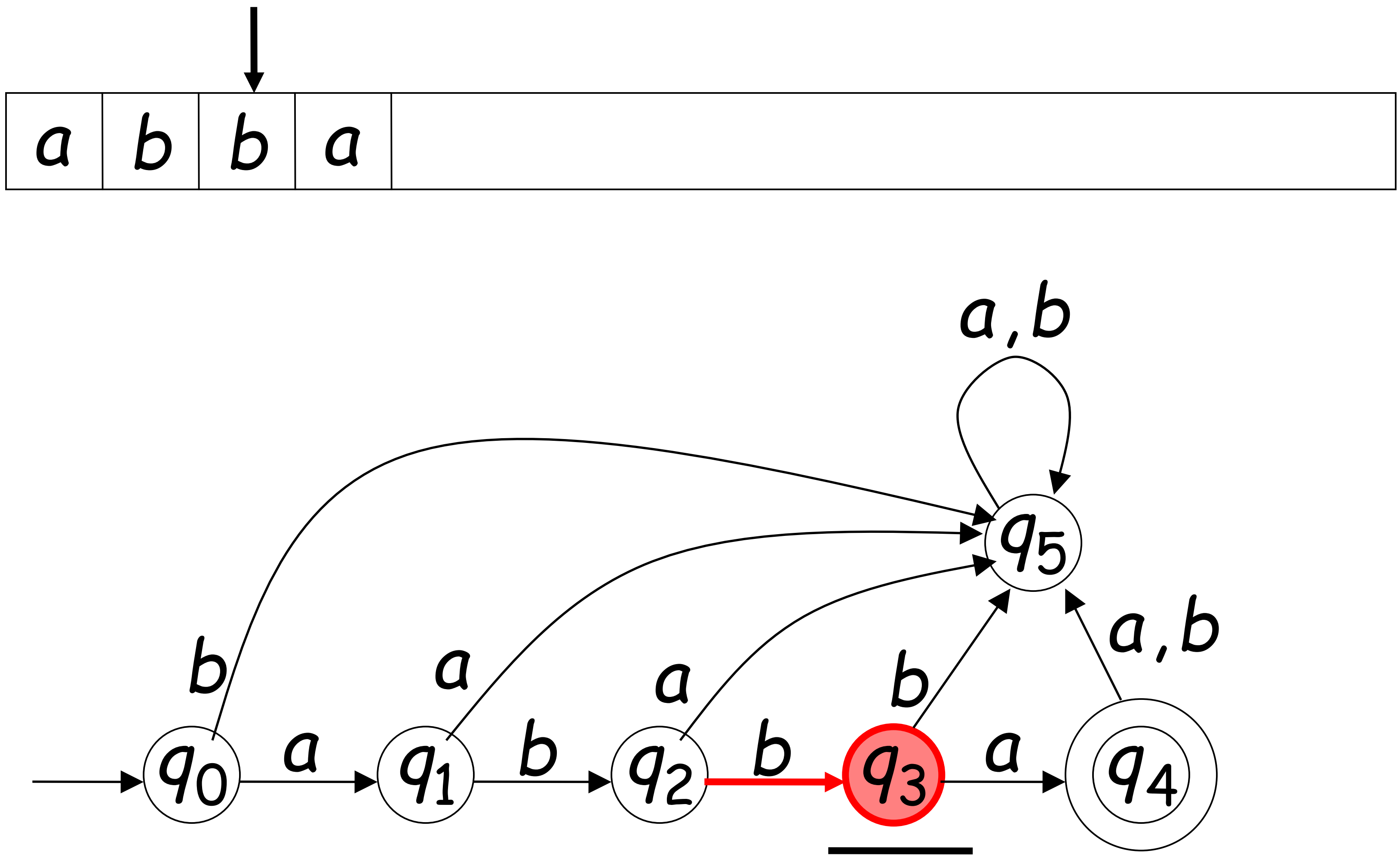


Reading the Input

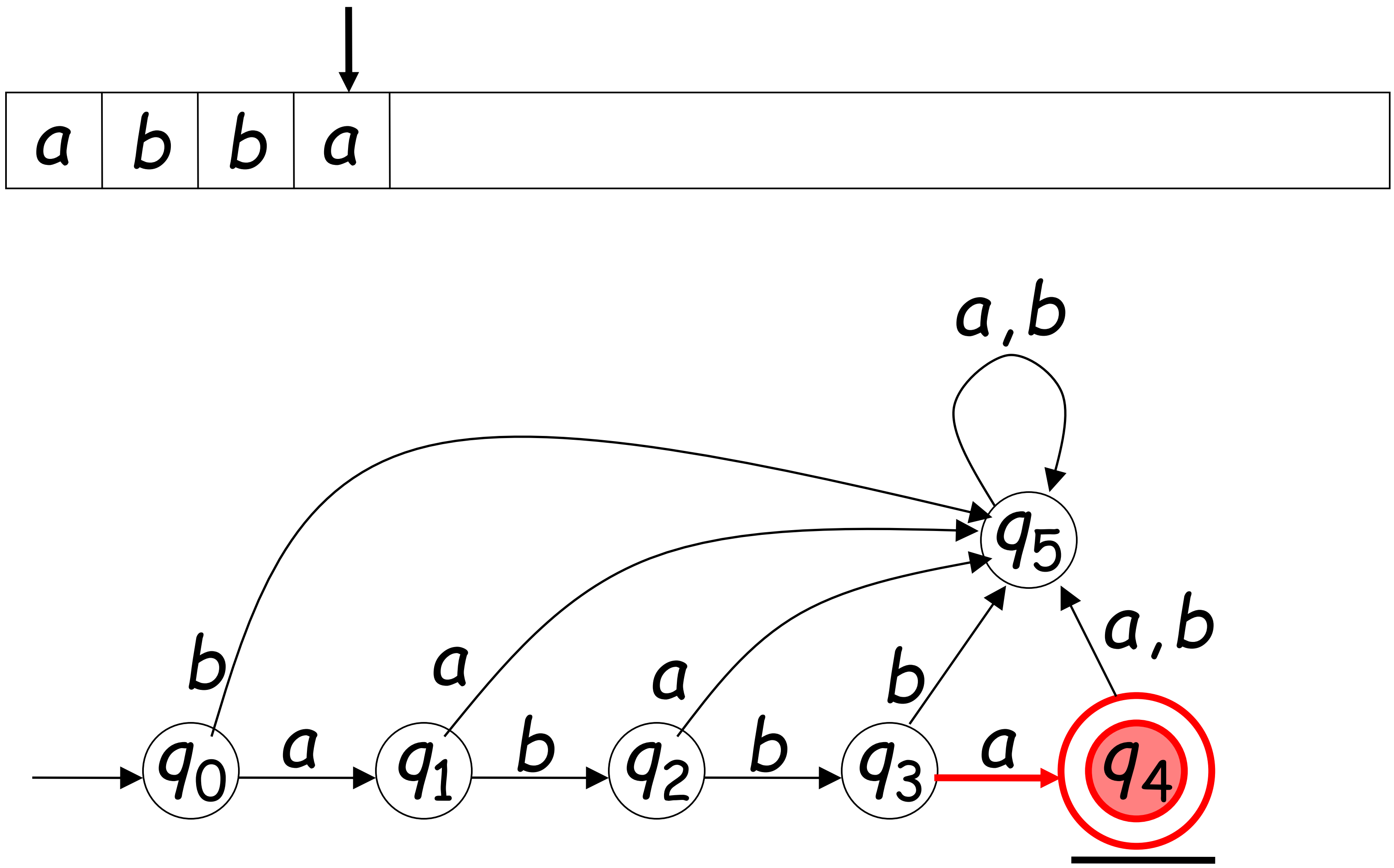




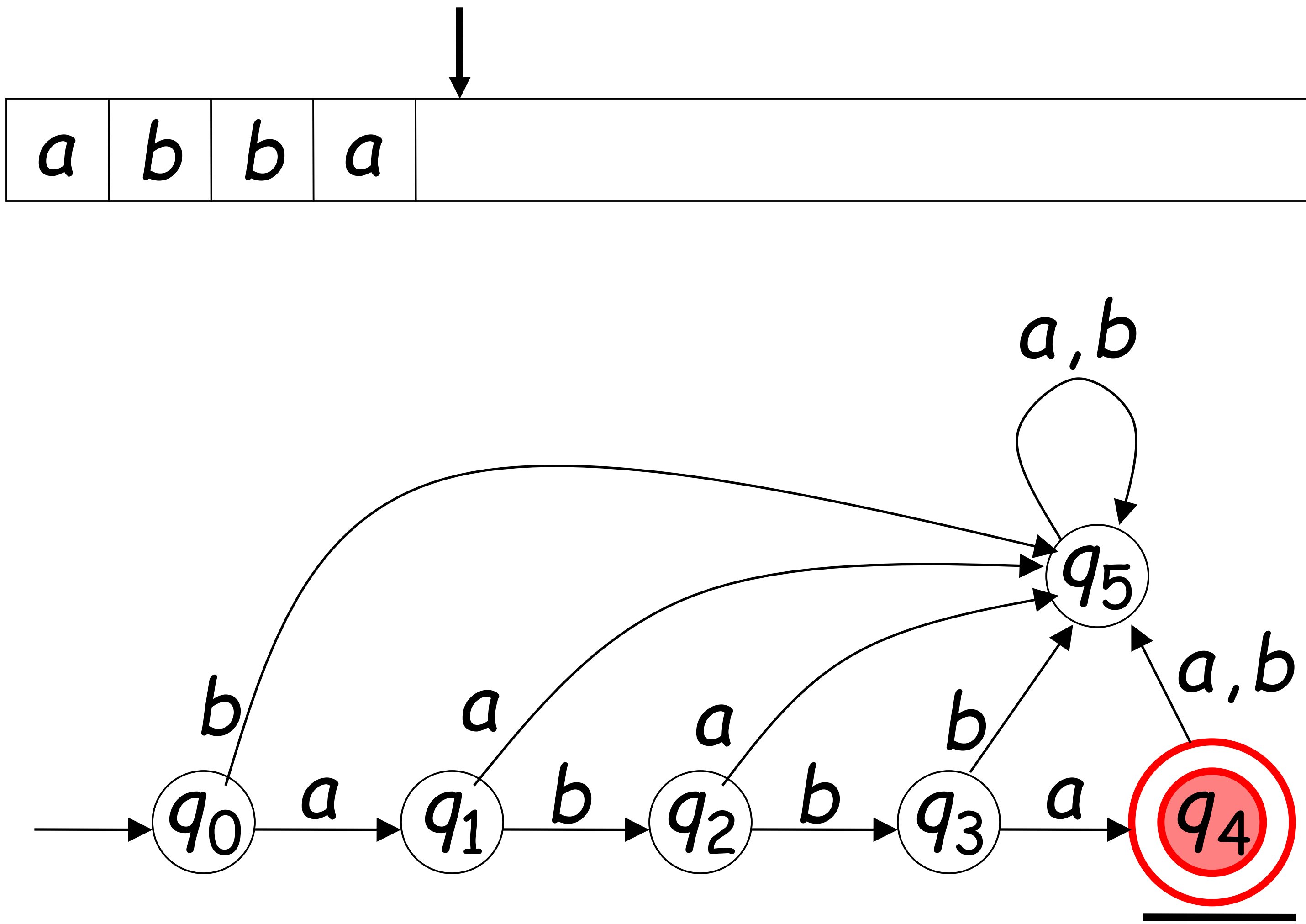

a
b
b
a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4



a
b
b
a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4

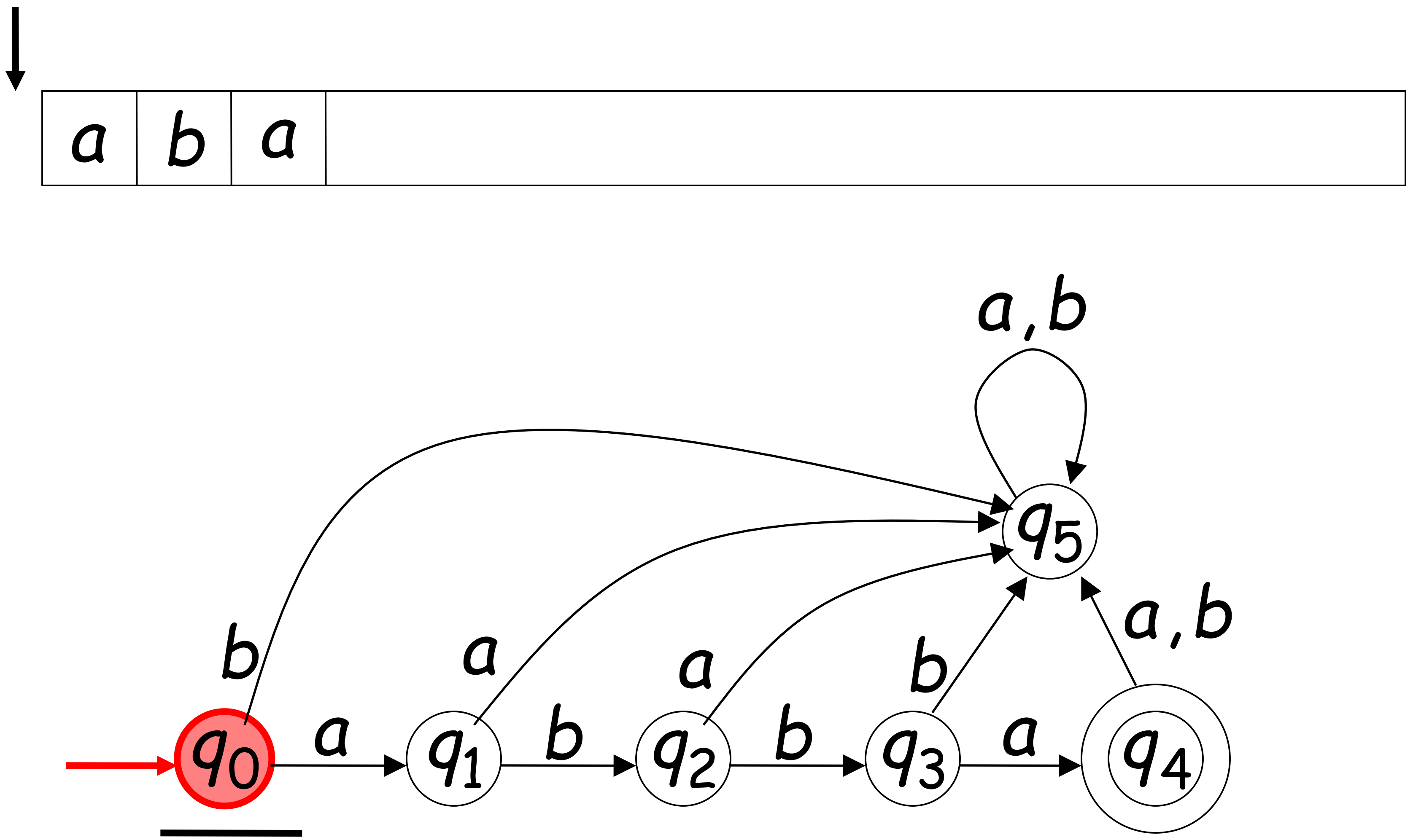

a b b a
a,b
q5
b
a
a
b
a,b
q0 a q1 b q2 b q3 a q4

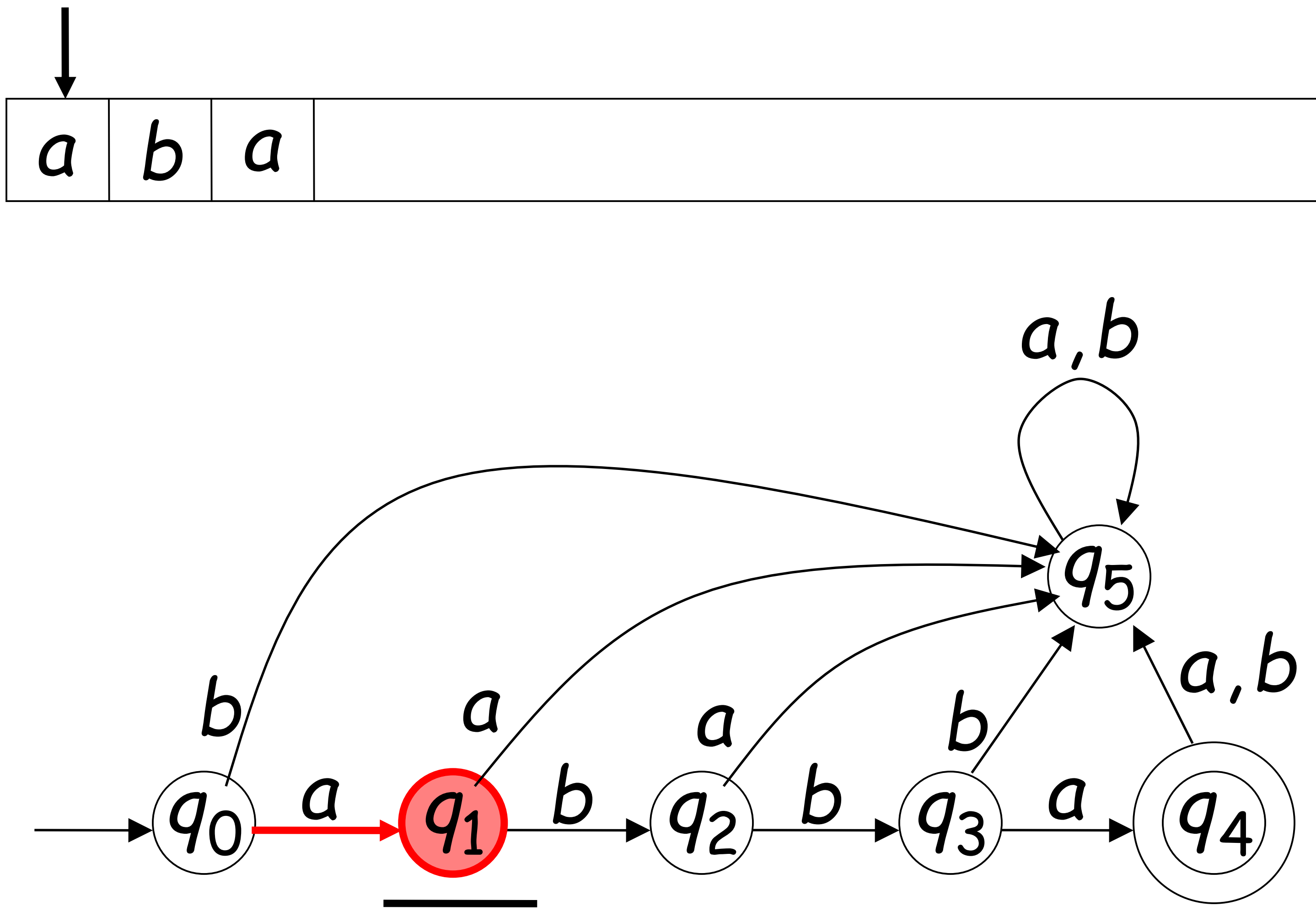
Input finished

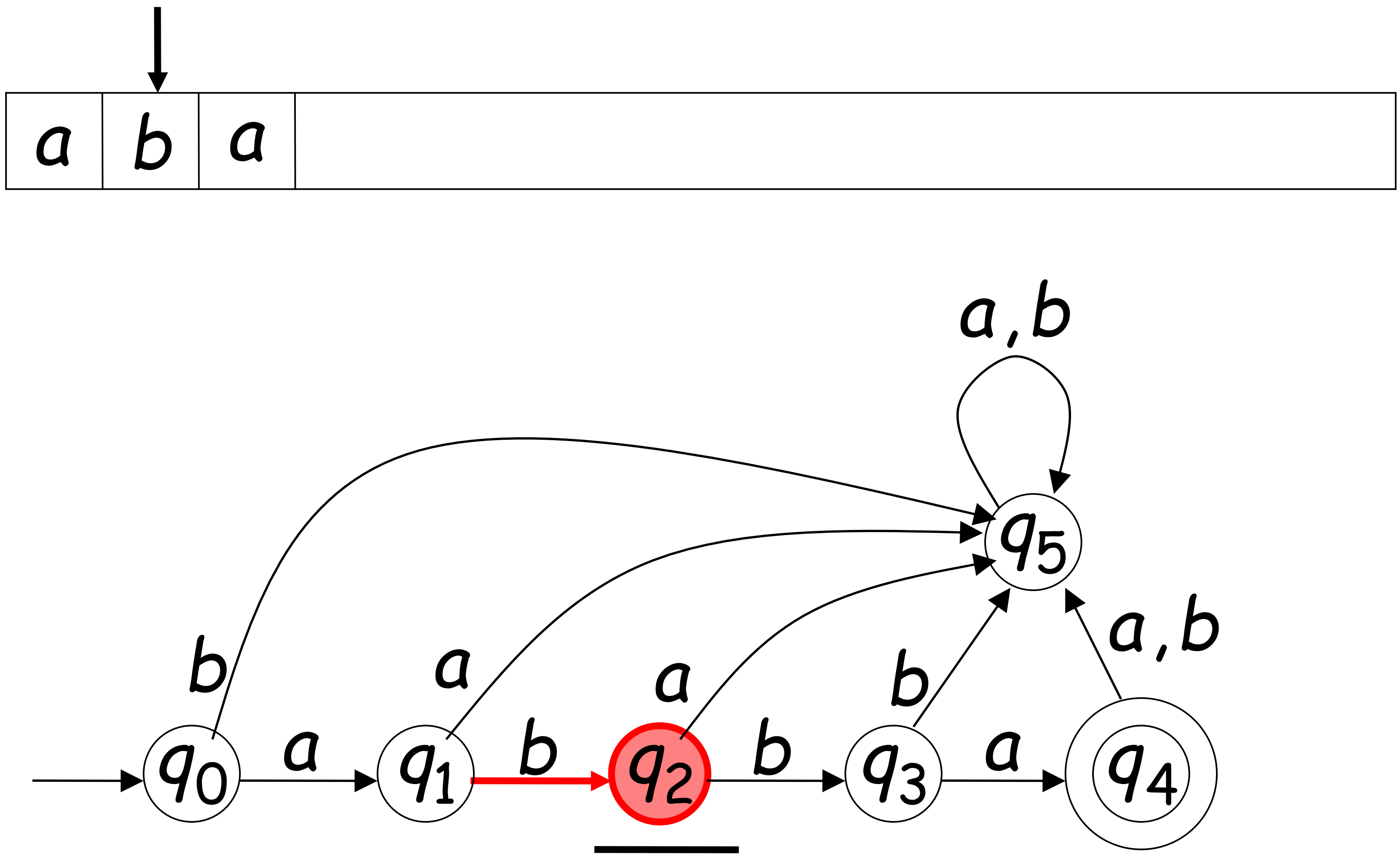


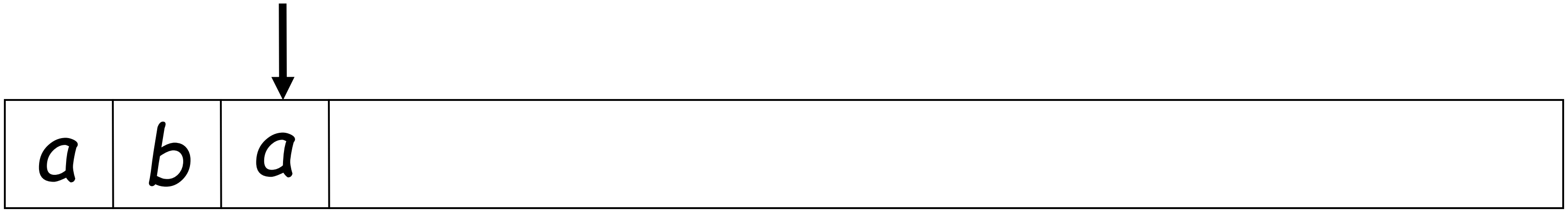
accept

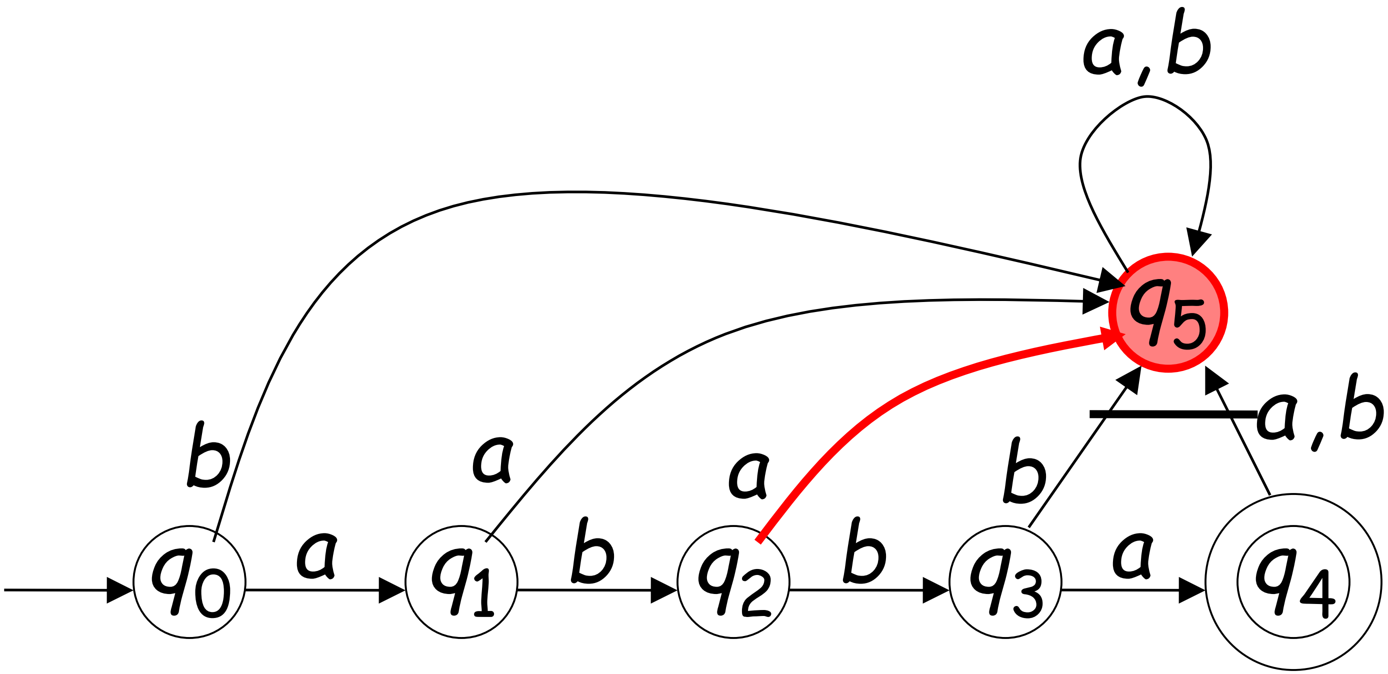
Rejection

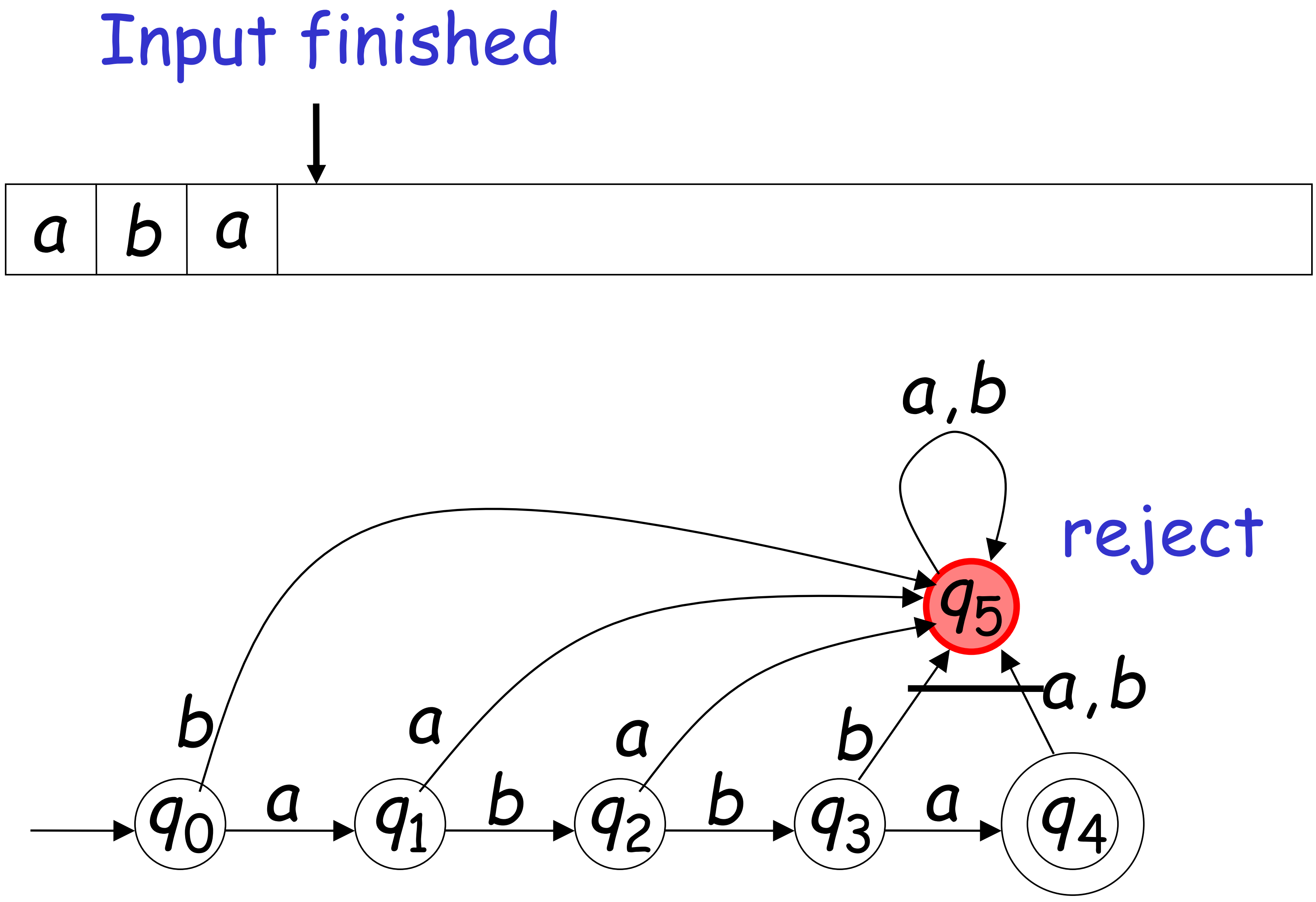



a
b
a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4


a
b
a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4


a
b
a
a,b
b
a
a
b
a,b
q5
q0
a
q1
b
q2
b
q3
a
q4




Input finished
a
b
a
a,b
reject
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4
q5

Languages Accepted by FAs

FA M

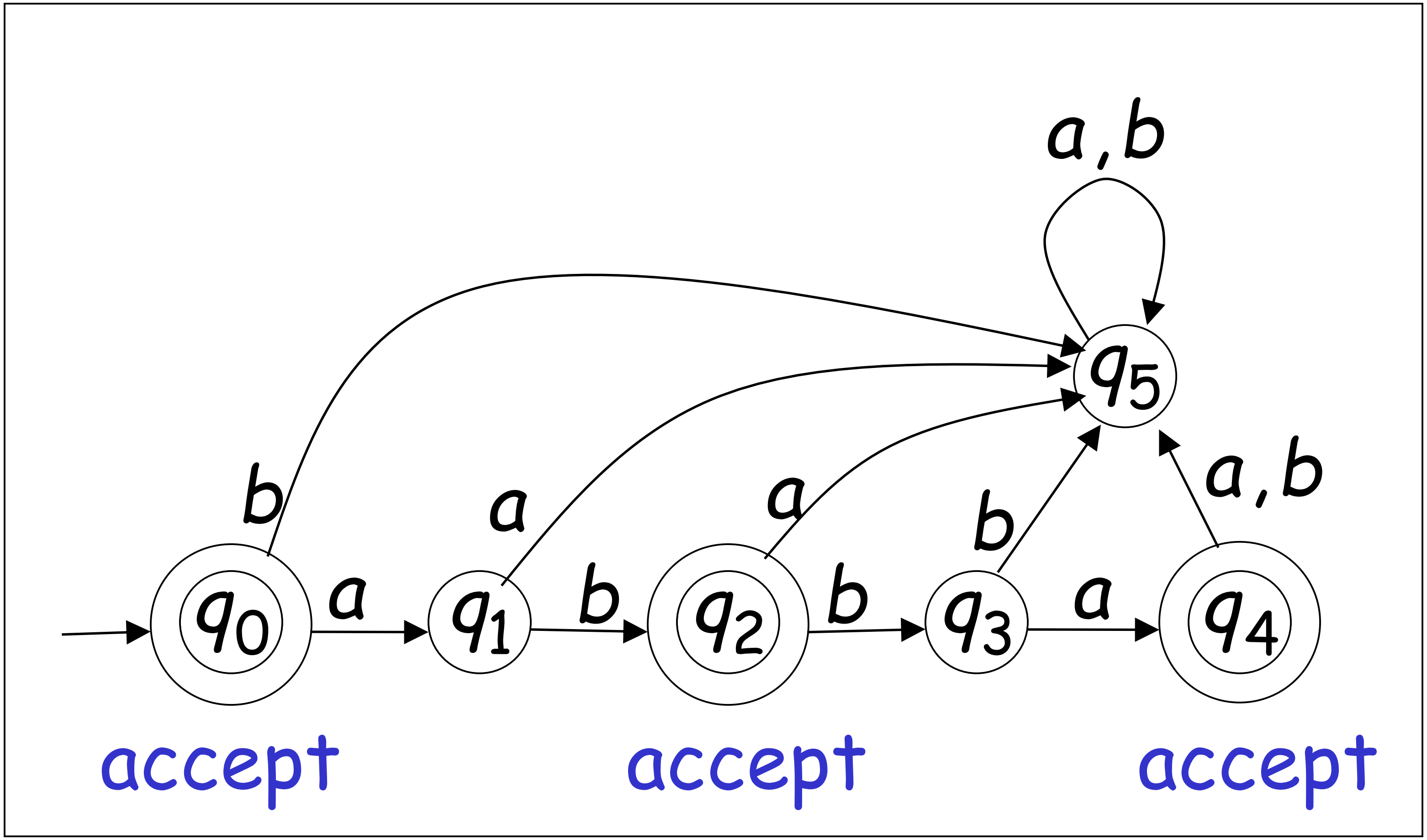
Definition:

The language $L(M)$ contains all input strings accepted by M

$L(M)$ = { strings that bring M to an accepting state}

Example

$$L(M) = \{\lambda, ab, abba\} \qquad M$$



Formal Definition

Finite Automaton (FA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q : set of states

Σ : input alphabet

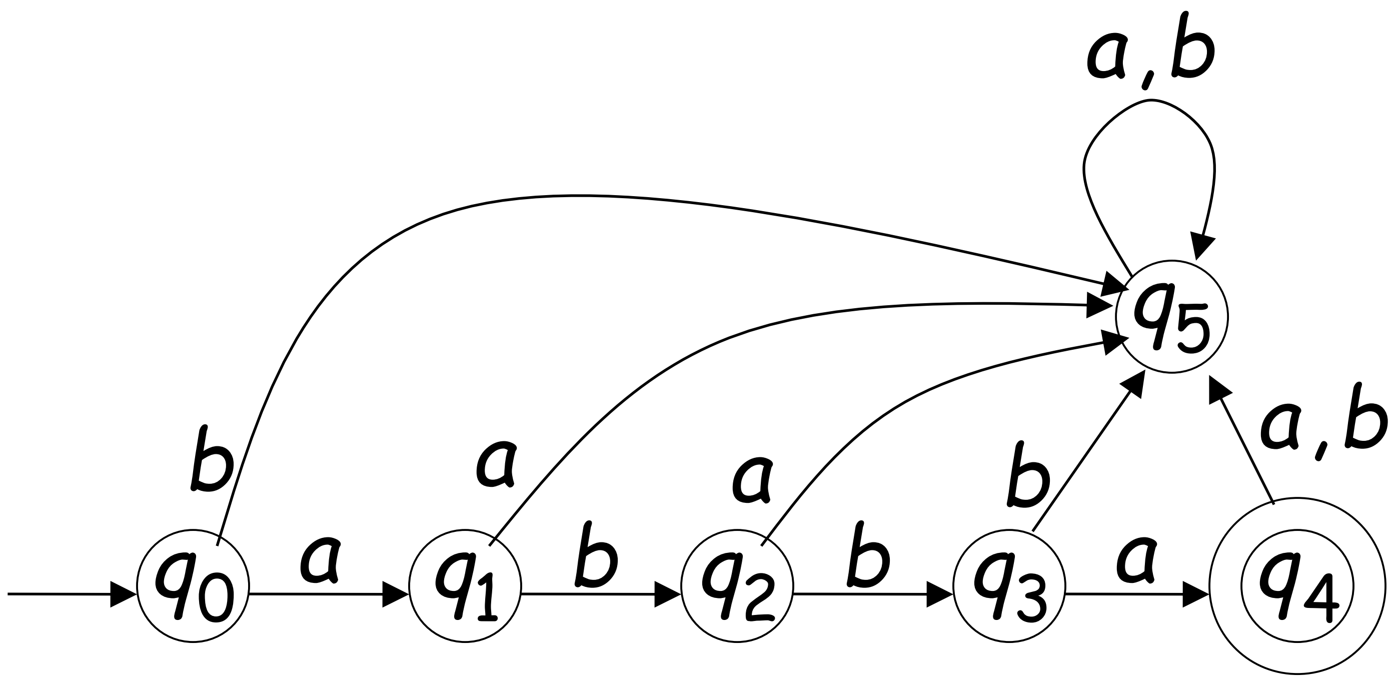
δ : transition function

q_0 : initial state

F : set of accepting states

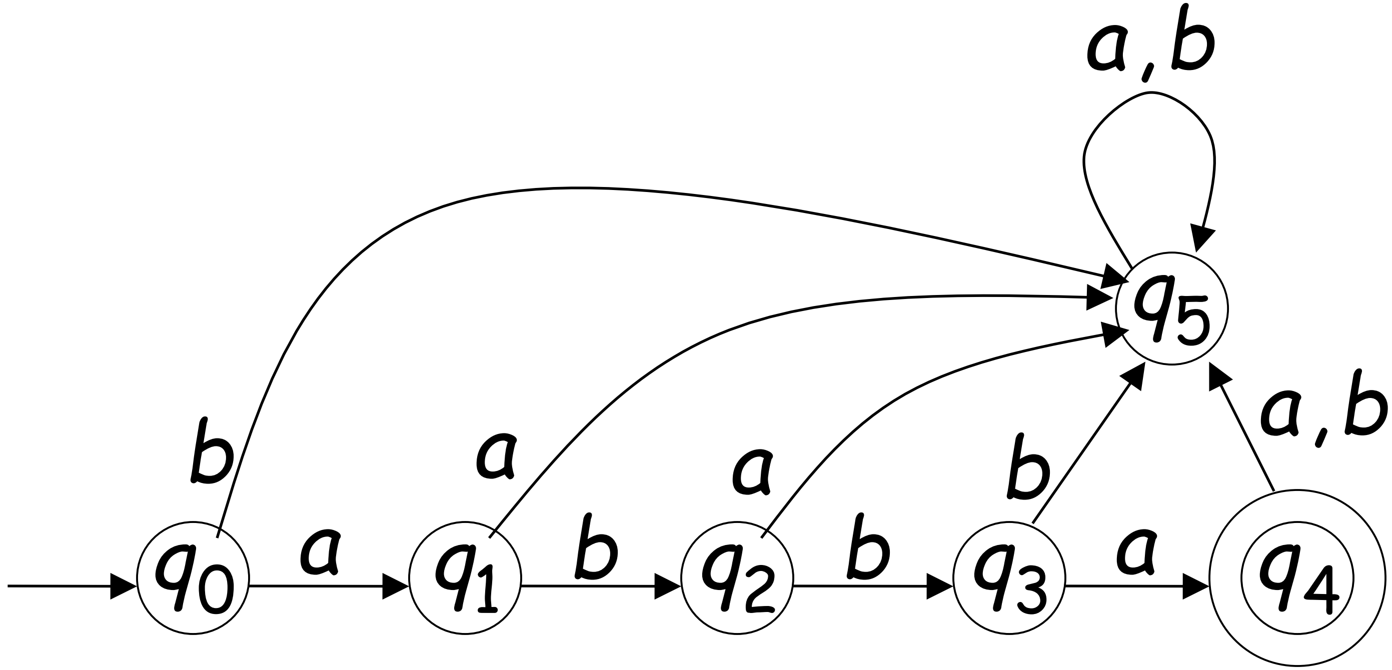
Input Alphabet Σ

$$\Sigma = \{a, b\}$$

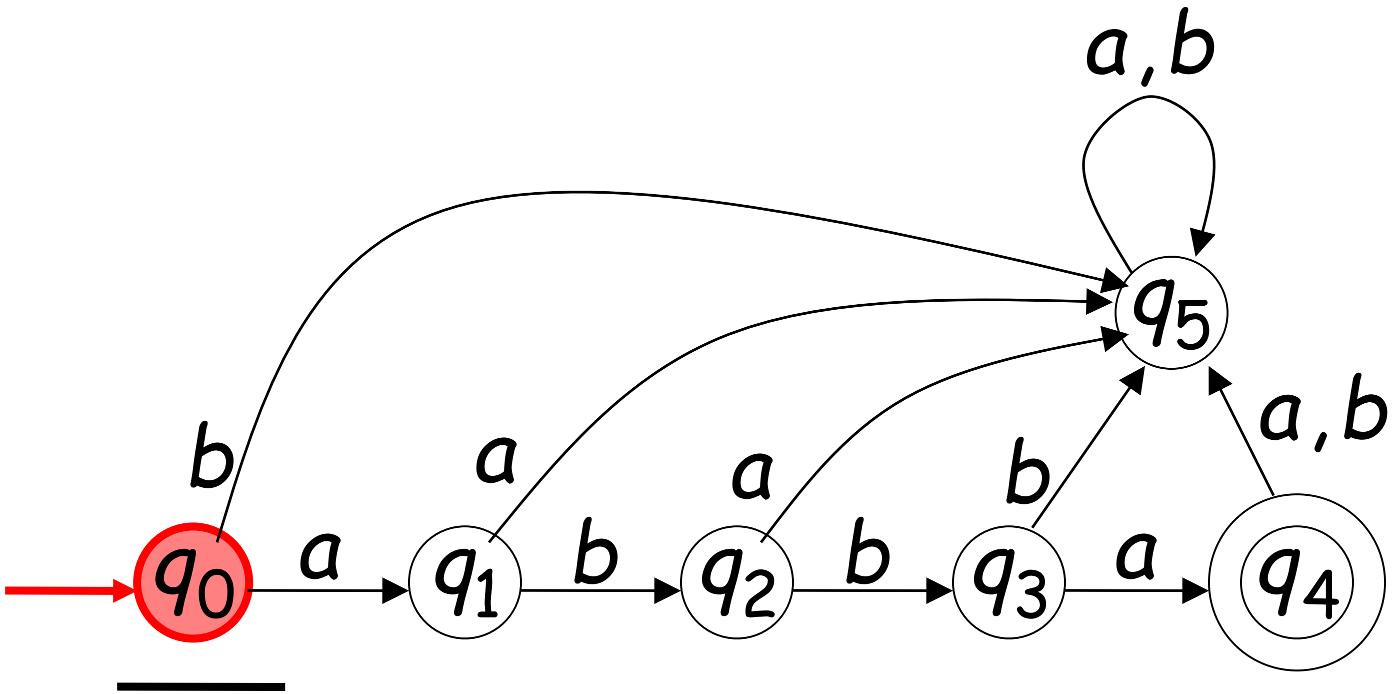


Set of States Q

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

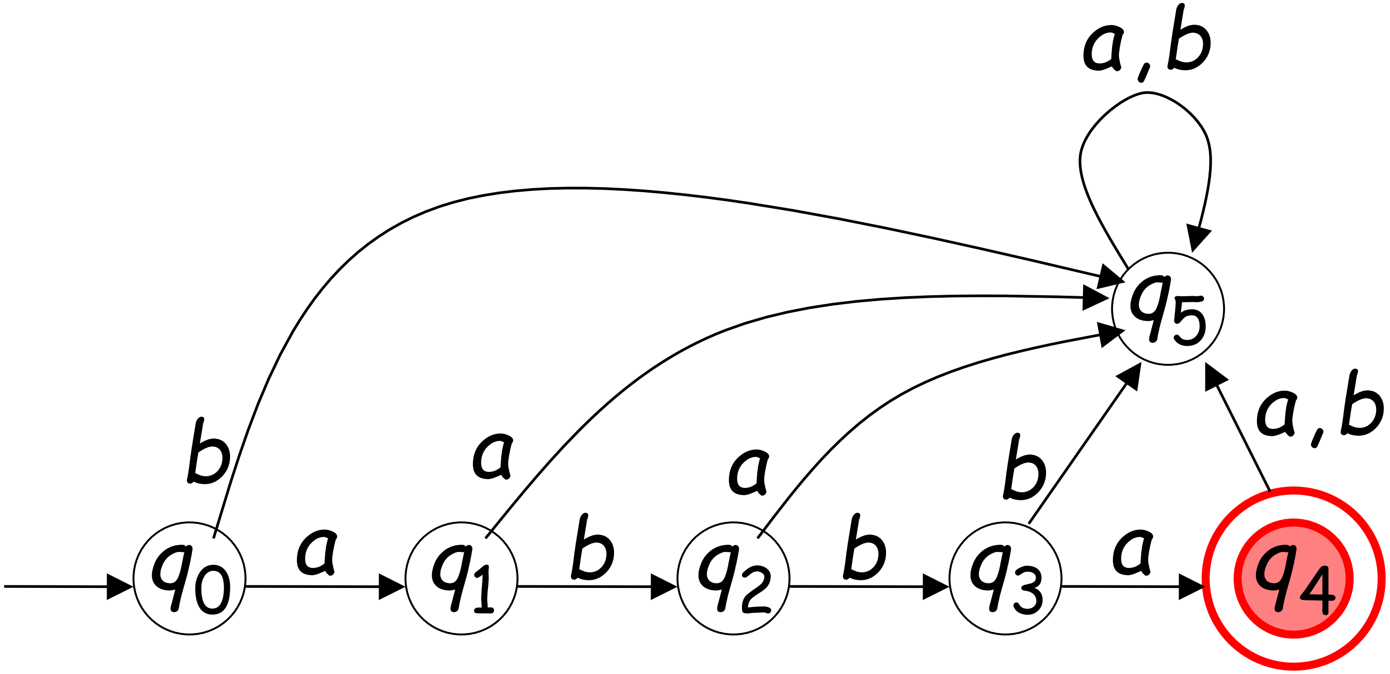


Initial State q_0



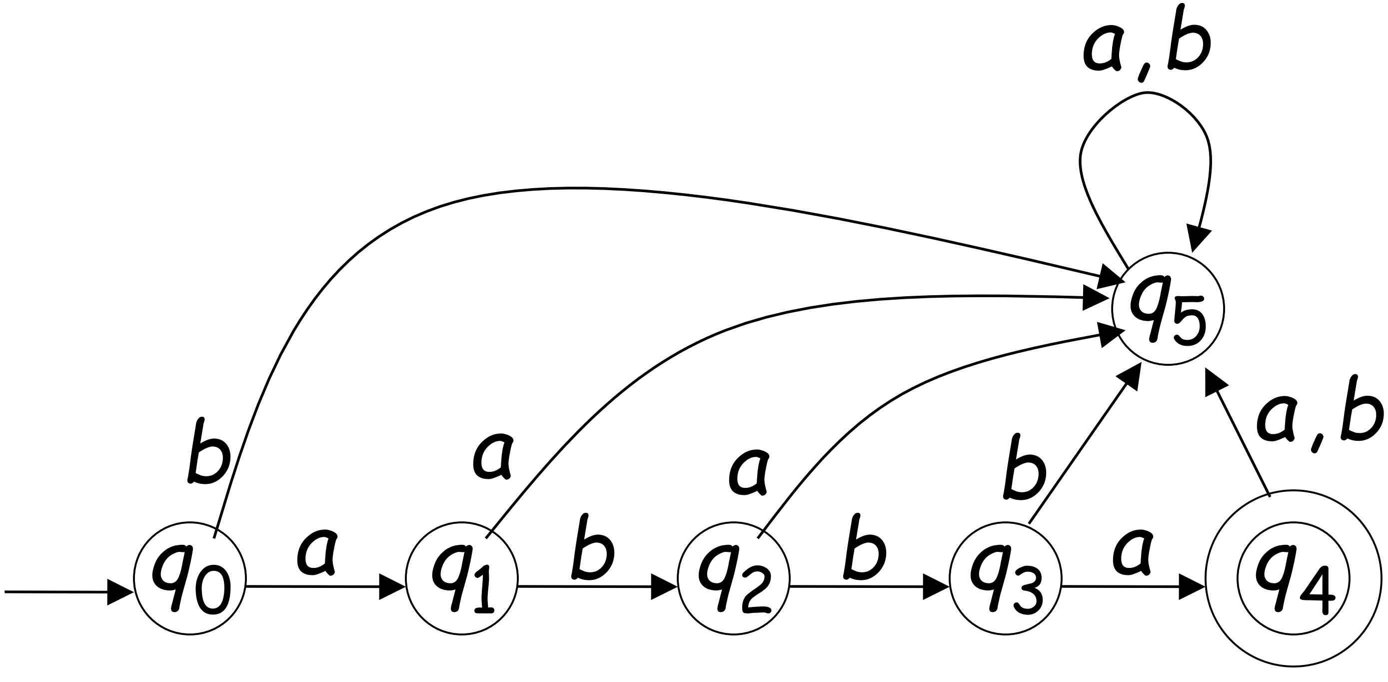
Set of Accepting States F

$$F = \{q_4\}$$

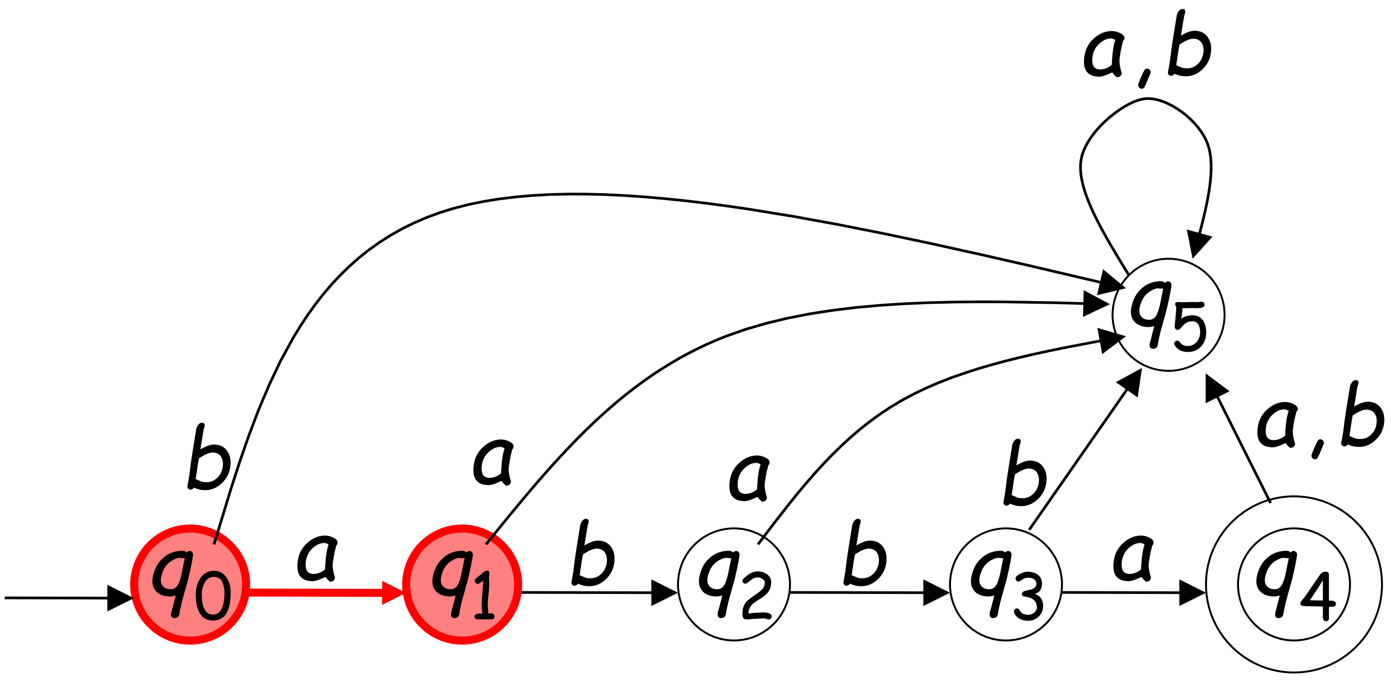


Transition Function δ

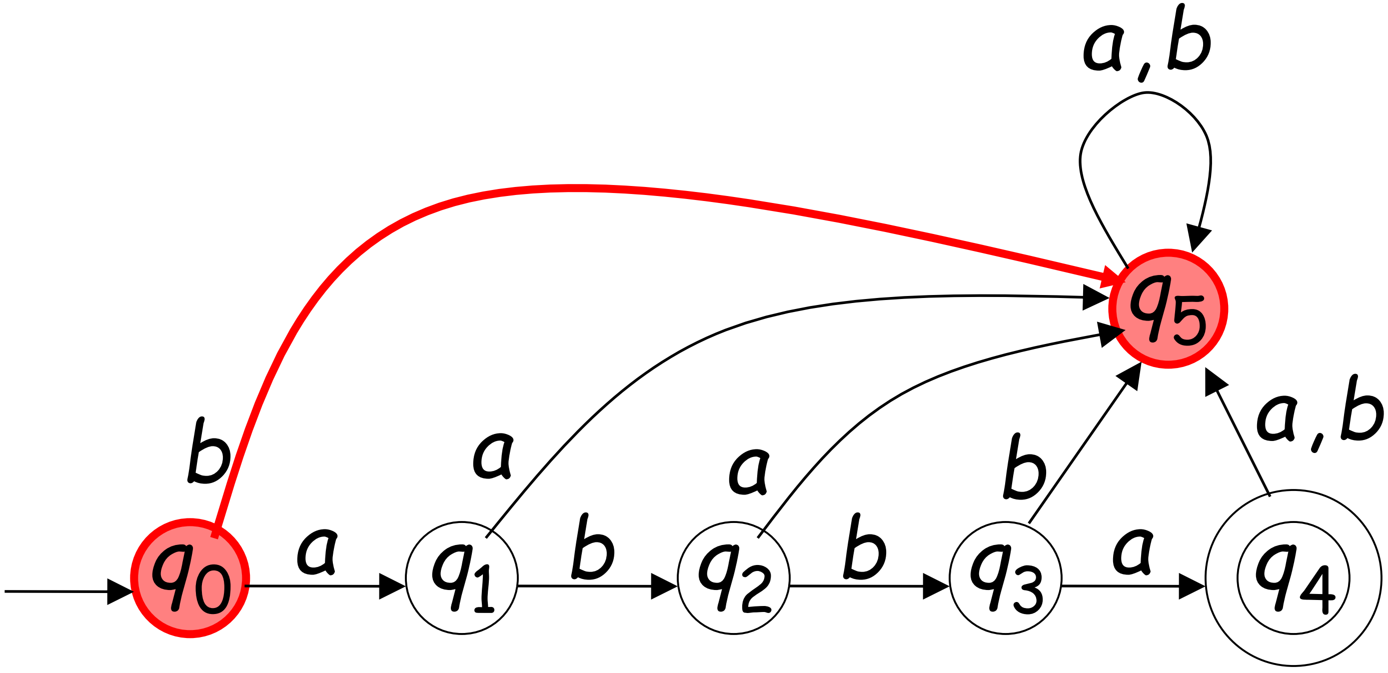
$$\delta : Q \times \Sigma \rightarrow Q$$



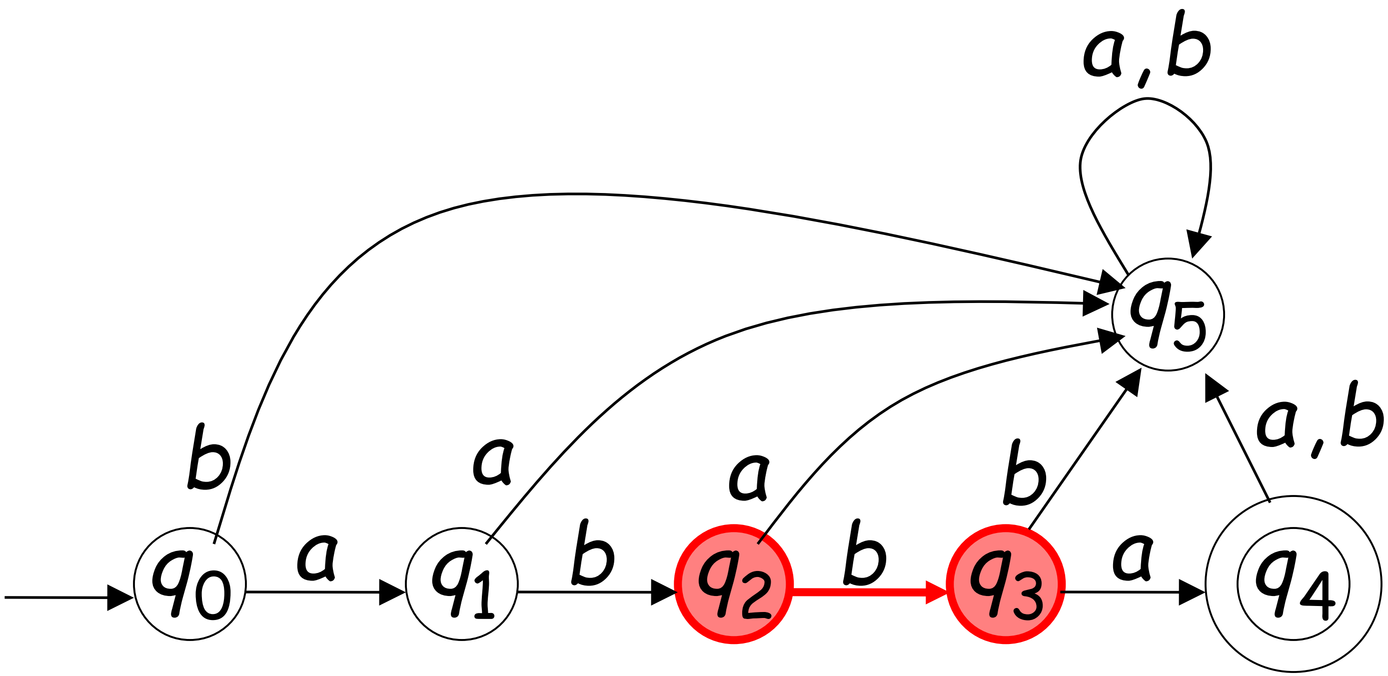
$$\delta(q_0, a) = q_1$$



$$\delta(q_0, b) = q_5$$

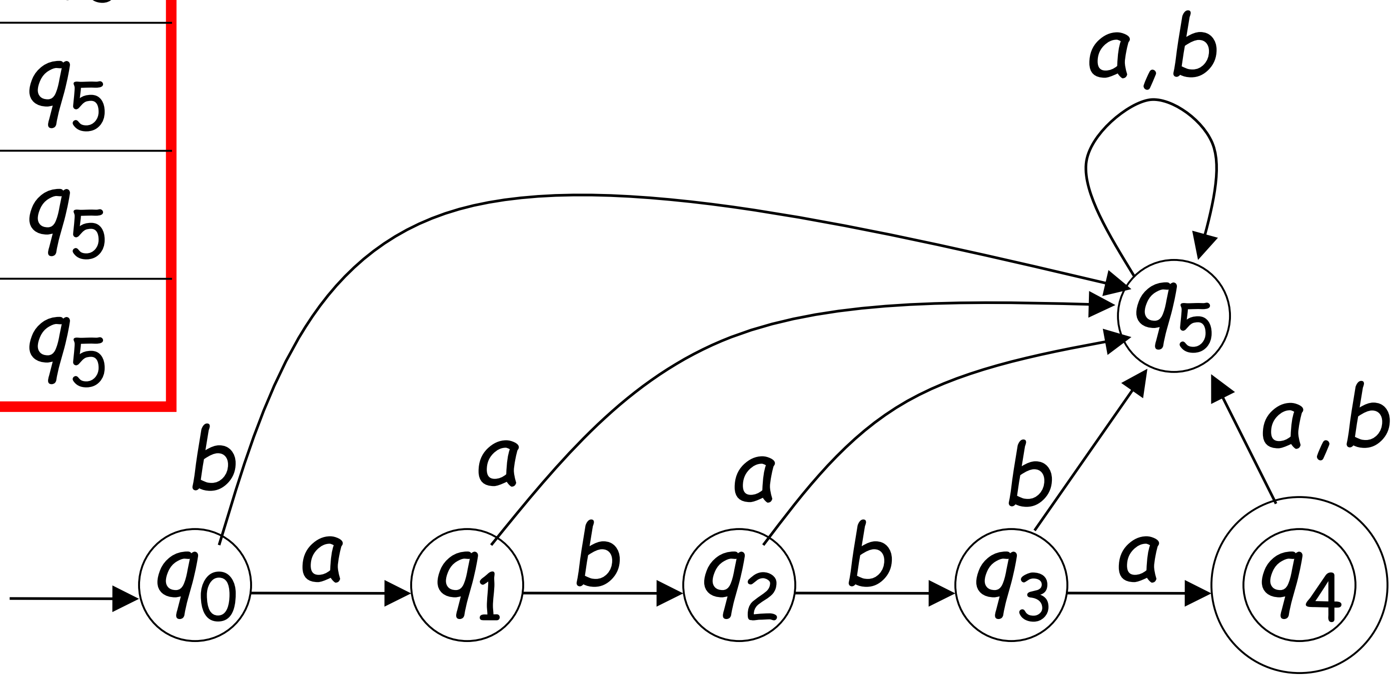


$$\delta(q_2, b) = q_3$$



Transition Function δ

δ	a	b
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_5	q_3
q_3	q_4	q_5
q_4	q_5	q_5
q_5	q_5	q_5



Regular Languages

Definition:

A language L is regular if there is FA M such that $L = L(M)$

Observation:

All languages accepted by FAs form the family of regular languages

There exist languages which are not Regular:

Example: $L = \{a^n b^n : n \geq 0\}$

There is no FA that accepts such a language

Regular Expressions

Regular expressions
describe regular languages

Example: $(a+b\cdot c)^*$

describes the language

$$\{a,bc\}^* = \{\lambda, a, bc, aa, abc, bca, ...\}$$

Recursive Definition

Primitive regular expressions: $\emptyset, \quad \lambda, \quad \alpha$

Given regular expressions r_1 and r_2

$$\left.\begin{array}{l} r_1 + r_2 \\ r_1 \cdot r_2 \\ r_1 * \\ (r_1) \end{array}\right\} \text{Are regular expressions}$$

Example

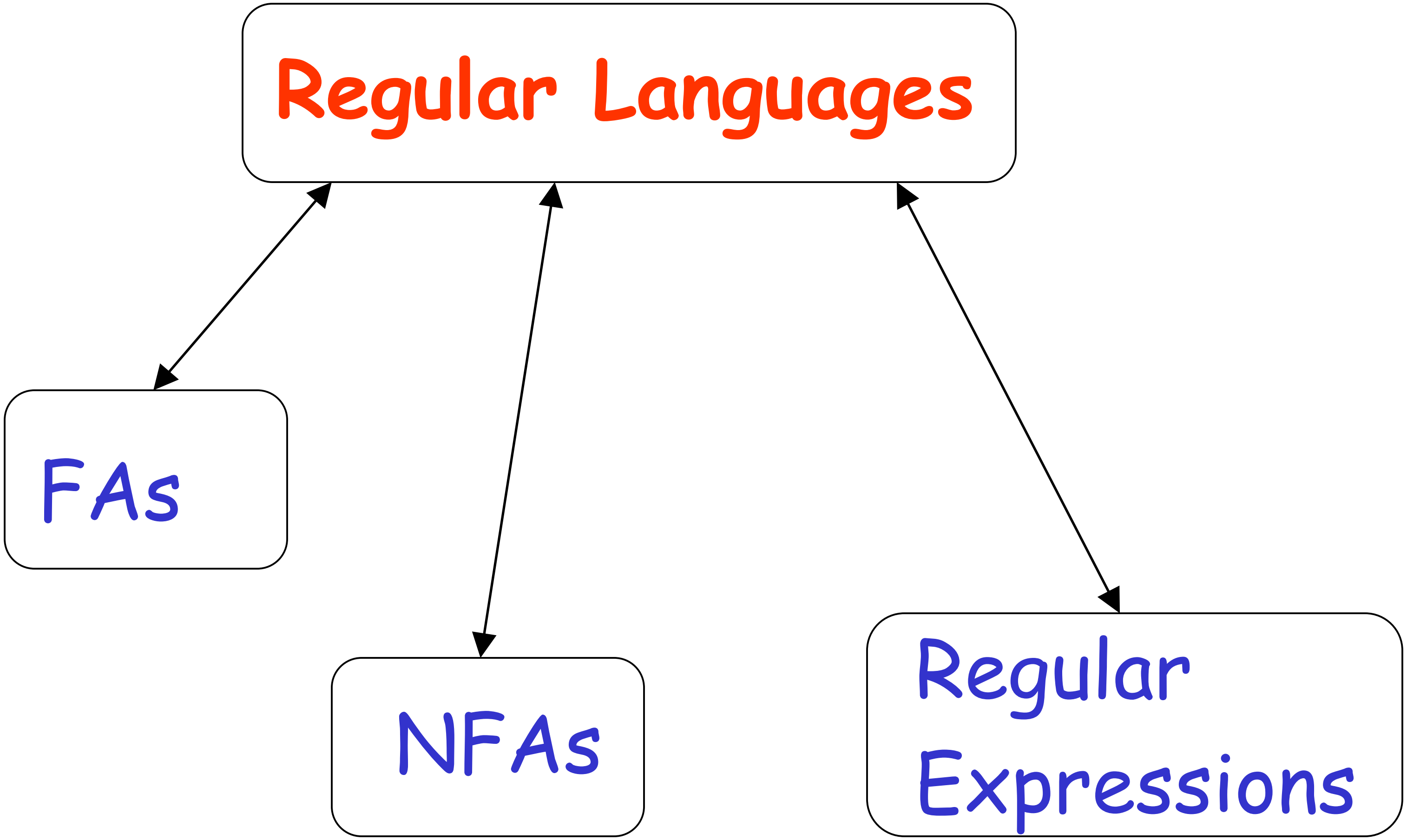
Regular expression: $(a+b)\cdot a^*$

$$
\begin{aligned}
L((a+b)\cdot a^*) &= L((a+b))\,L(a^*) \\
&= L(a+b)\,L(a^*) \\
&= (L(a)\cup L(b))\,(L(a))^* \\
&= (\{a\}\cup\{b\})\,(\{a\})^* \\
&= \{a,b\}\,\{\lambda,a,aa,aaa,\ldots\} \\
&= \{a,aa,aaa,\ldots,b,ba,baa,\ldots\}
\end{aligned}
$$

Theorem

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} = \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Standard Representations of Regular Languages



- String Matching Problem
    - Concept
    - Regular expressions
    - brute force algorithm
    - complexity
- Finite State Machines
- Knuth-Morris-Pratt(KMP) Algorithm
    - Pre-processing
    - complexity

Pattern Matching Algorithms

The Problem

- Given a text T and a pattern P, check whether P occurs in T
  - eg: T = {aabbcbbcabbbcbccccabbabbccc}
  - Find all occurrences of pattern P = bbc
- There are ***variations*** of pattern matching
  - Finding "approximate" matchings
  - Finding multiple patterns etc..

Why String Matching?

Applications in Computational Biology

- DNA sequence is a long word (or text) over a 4-letter alphabet
- GTTTGAGTGGTCAGTCTTTTCGTTTCGACGGAGCCCCCAATTAA TAAACTCATAAGCAGACCTCAGTTCGCTTAGAGCAGCCGAAA …..
- Find a Specific pattern W

Finding patterns in documents formed using a large alphabet

- Word processing
- Web searching
- Desktop search (Google, MSN)

Matching strings of bytes containing

- Graphical data
- Machine code

grep in unix

- grep searches for lines matching a pattern.

String Matching

Text string T[0..N-1]

T = "**abacaabaccabacabaabb**"

Pattern string P[0..M-1]

P = "**abacab**"

Where is the *first* instance of P in T?

T[**10..15**] = P[0..5]

Typically, N >>> M

Java Pattern Matching Utilities

- Java provides an *API* for pattern matching with regular expressions
  - java.util.regex
- *Regular expressions* describe a **set of strings** based on some common characteristics shared by each string in the set. eg: a* ={ ,a, aa, aaa, ...}
- Regular expressions can be used as a tool to search, edit or manipulate text or data
  - perl, java, *C#*

Java Pattern Matching Utilities

java.util.regex

- Pattern
  - Is a compiled representation of a regular expression.
  - Eg: Pattern p = Pattern.compile("a*b");
- Matcher
  - A machine that performs match operations on a character sequence by interpreting a pattern.
  - Eg: _ Matcher m = p.matcher("aabbb");

Example:

```
public static void main( String args[] ) {
    Pattern p = Pattern.compile("(aa|bb)*");
    Matcher m = p.matcher("aabbb");
    boolean b = m.matches(); //match the entire input sequence against the pattern
   // or boolean b = m.find(); // match the entire input sequence against the pattern
    System.out.println("The value is " + b);
}
```

String Matching

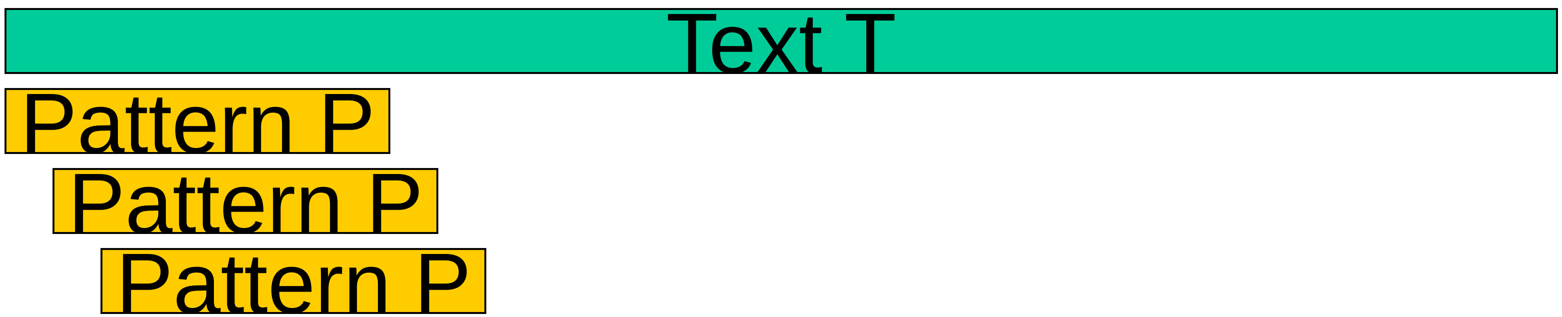
```
abacaabaccabacabaabb
abacab
 abacab
  abacab
   abacab
    abacab
     abacab
      abacab
       abacab
        abacab
         abacab
          abacab
```

The brute force algorithm

22+6=**28** comparisons.

Naïve Algorithm (or Brute Force)

Assume |T| = n and |P| = m



Compare until a match is found. If so, return the index where match occurs

else return -1

Brute Force

```
static int match(char[] T, char[] P){
      for (int i = 0; i < T.length; i++){
        boolean flag = true;
        if (P[0] == T[i])
          for (int j = 1;j < P.length; j++)
            if (T[i+j] != P[j])
              {flag = false; break;}
        if (flag) return i;
      }
}
```

A bad case

```
00000000000000001

0000-
 0000-
  0000-
   0000-
    0000-
     0000-
      0000-
       0000-
        0000-
         0000-
          0000-
           0000-
            00001
```

60+5 = **65** comparisons are needed

A bad case

```
00000000000000001

0000-
 0000-
  0000-
   0000-
    0000-
     0000-
      0000-
       0000-
        0000-
         0000-
          0000-
           0000-
            00001
```

60+5 = **65** comparisons are needed

How many of them could be avoided?

Typical text matching

```
This is a sample sentence

-
 -
  -
   s-
    -
     -
      s-
       -
        -
         -
          s-
           -
            -
             -
              -
               -
                -
                 sente
```

20+5=25 comparisons are needed

(The match is near the same point in the target string as the previous example.)

String Matching

- Brute force worst case
  - **$O(MN)$**
  - Expensive for long patterns in repetitive text
- How to improve on this?
- Intuition:
  - *Remember what is learned from previous matches*

Finite Automaton (FA)

- FA is a computing machine that takes
  - A string as an input
  - Outputs YES/NO answer
    - That is, the machine "accepts" or "rejects" the string

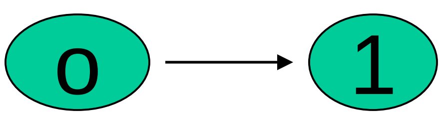
FA Model

Input to a FA

- Strings built from a fixed alphabet {a,b,c}
- Possible inputs: aa, aabbcc, a etc..

The Machine

- A directed graph
  - Nodes = States of the machine
  - Edges = Transition from one state to another



Why Study FA's

Useful Algorithm Design Technique

- Lexical Analysis ("tokenization")
- Control Systems
  - Elevators, Soda Machines….

Modeling a problem with FSM is

- Simple
- Elegant

Knuth Morris Pratt (KMP) Algorithm

KMP - The Big Idea

- Retain information from prior attempts.
- **Compute in advance how far to jump in P when a match fails.**
  - Suppose the match fails at **$P[j] \neq T[i+j]$**.
  - Then we know **$P[0 .. j-1] = T[i .. i+j-1]$**.
- We must next try **$P[0]$? $T[i+1]$**.
  - But we know **$T[i+1]=P[1]$**
  - What if we compare: **$P[1]$? $P[0]$**
    - If so, increment j by 1. No need to look at T.
  - What if **$P[1] = P[0]$** and **$P[2] = P[1]$**?
    - Then increment j by 2. Again, no need to look at T.
- In general, we can determine how far to jump without any knowledge of T!

Implementing KMP

- Never decrement **i**, ever.
  - Comparing
    T[i] with **P[j]**.
- Compute a table **f** of how far to jump **j** forward when a match fails.
  - The next match will compare
    T[i] with **P[f[j-1]]**
- Do this by matching **P** against itself in all positions.

Building the Table for f

P = 1010011

Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
10	.	2	0
10**1**	**1**	3	1
10**10**	**10**	4	2
10100	.	5	0
10100**1**	**1**	6	1
101001**1**	**1**	7	1

What f means

Prefix	Overlap	j	f
1	.	1	0
10	.	2	0
101	1	3	1
1010	10	4	2
10100	.	5	0
101001	1	6	1
1010011	1	7	1

If f is zero, there is no self-match.

Set **j=0**

Do not change **i**.

The next match is

T[i] ? P[0]

f non-zero implies there is a self-match.

E.g., **f=2** means **P**[0..1] = **P**[j-2..j-1]

Hence must start new comparison at **j-2**, since we know **T**[i-2..i-1] = **P**[0..1]

In general:

Set **j=f[j-1]**

Do not change **i**.

The next match is

T[i] ? P[f[j-1]]

Favorable conditions

P = 1234567

Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
12	.	2	0
123	.	3	0
1234	.	4	0
12345	.	5	0
123456	.	6	0
1234567	.	7	0

Mixed conditions

P = 1231234

Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
12	.	2	0
123	.	3	0
123**1**	**1**	4	1
123**12**	**12**	5	2
123**123**	**123**	6	3
1231234	.	7	0

Poor conditions

P = 1111110

Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
11	1	2	1
111	11	3	2
1111	111	4	3
11111	1111	5	4
111111	11111	6	5
1111110	.	7	0

KMP pre-process Algorithm

```
m = |P|;
Define a table F of size m
F[0] = 0;
i = 1; j = 0;
while(i < m) {
   compare P[i] and P[j];
   if(P[j] == P[i])
      { F[i] = j+1;
        i++; j++; }
   else if (j > 0) j = F[j-1];
   else {F[i] = 0; i++;}
}
```

Use previous values of **f**

KMP Algorithm

```
input: Text T and Pattern P
|T| = n
|P| = m
Compute Table F for Pattern P
i=j=0

while(i < n) {
   if(P[j]==T[i])
      { if (j == m-1) return i-m+1;
        i++; j++; }
   else if (j>0) j=F[j-1];
   else i++;
}
output: first occurrence of P in T
```

Use **F** to determine next value for **j.**

KMP Performance

Pre-processing needs $O(M)$ operations.

At each iteration, one of three cases:

- T[i] = P[j]
  - i increases
- T[i] <> P[j] and j>0
  - i-j increases
- T[i] <> P[j] and j=0
  - i increases *and* i-j increases

Hence, maximum of 2N iterations.

Thus, worst case performance is $O(N+M)$.