DFAs Continued

Announcements

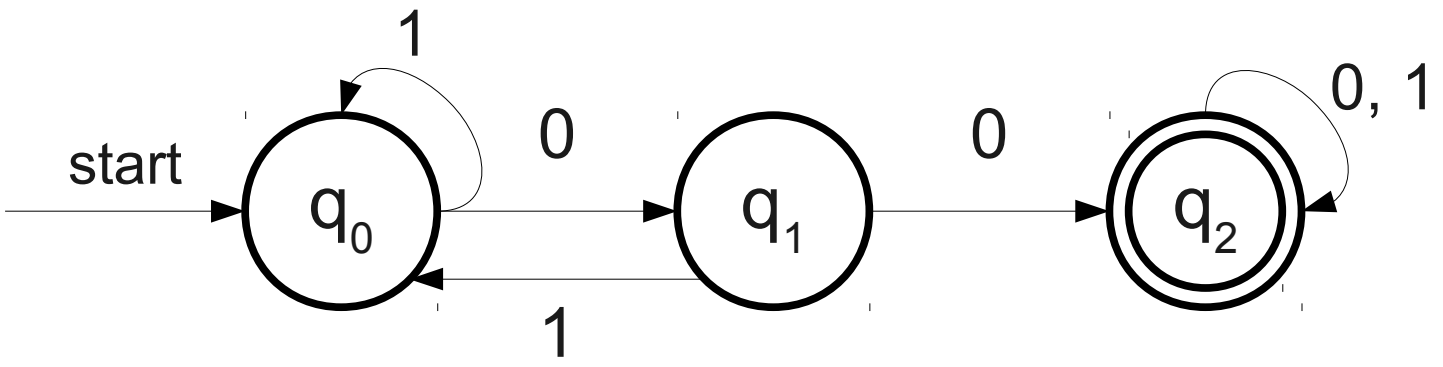
- Problem Set 4 due **Thursday** at **7:00PM** (right before the midterm).
  - Stop by OH with questions!
  - Email **cs103@cs.stanford.edu** with questions!
- Midterm tomorrow, 7 – 10PM in Hewlett 200
  - Alternate midterm **tonight**, 7 – 10PM, Gates 100.
  - Alternate midterm **tomorrow**, 9AM – Noon, Gates 167.
- My office hours today: 3:30 – 5:30PM
- Hrysoula's OH: 6:30 – 8:30PM

Recap: DFAs

- A DFA is defined relative to some alphabet Σ.
- For each state in the DFA, there must be **exactly one** transition defined for each symbol in the alphabet.
  - This is the “deterministic” part of DFA.
- There is a **unique** start state.
- There may be multiple **accepting states**.

Recognizing Languages with DFAs

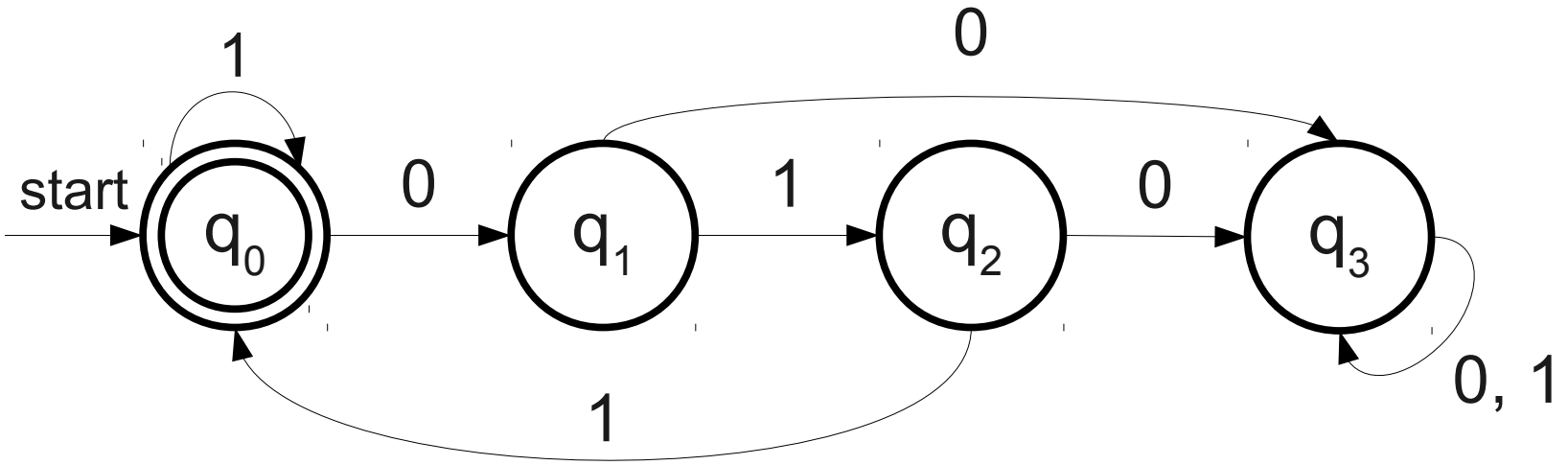
L = { w | w contains 00 as a substring }



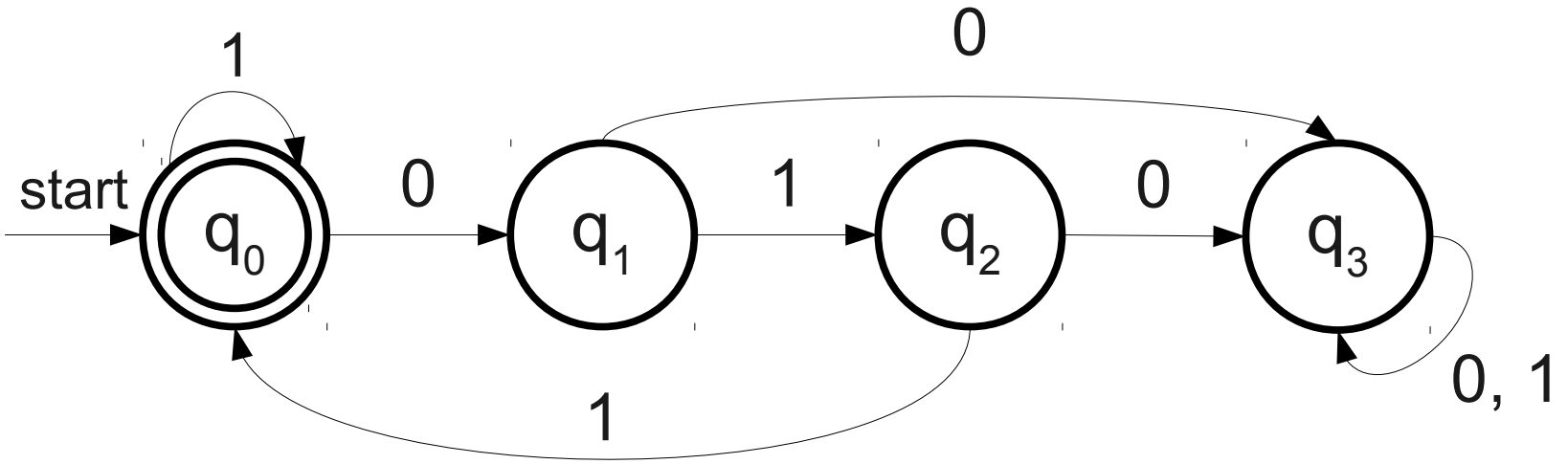
The **language of a DFA** D (denoted $L(D)$) is the set of strings that D accepts.

A language is called a **regular language** if there is some DFA that accepts it.

Tabular DFAs

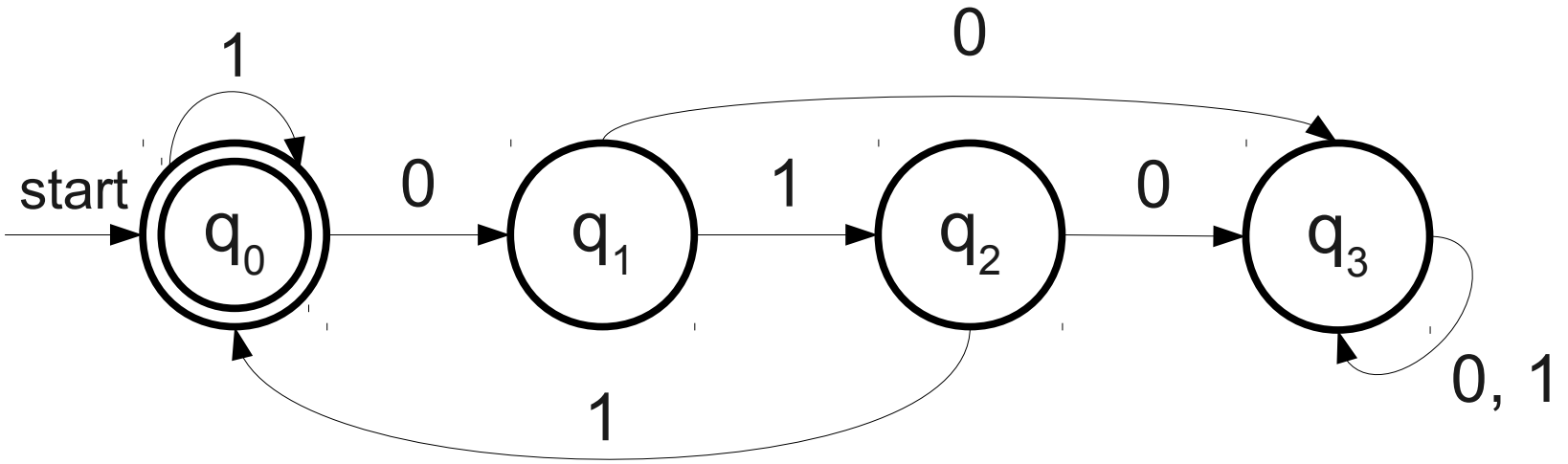


Tabular DFAs



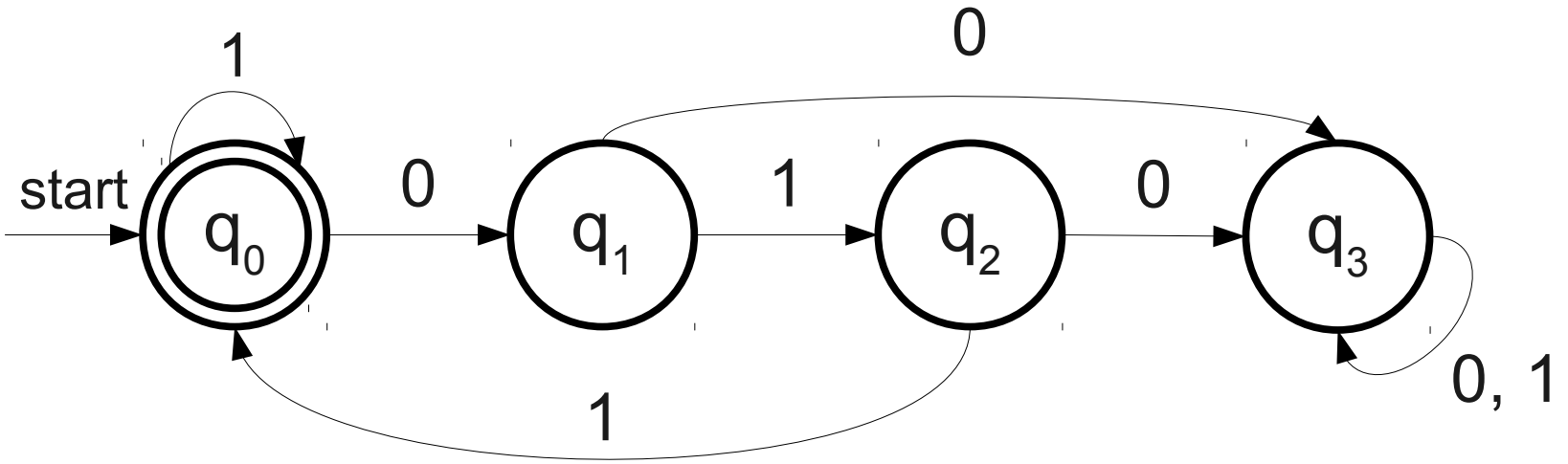
	0	1
q_0		
q_1		
q_2		
q_3		

Tabular DFAs



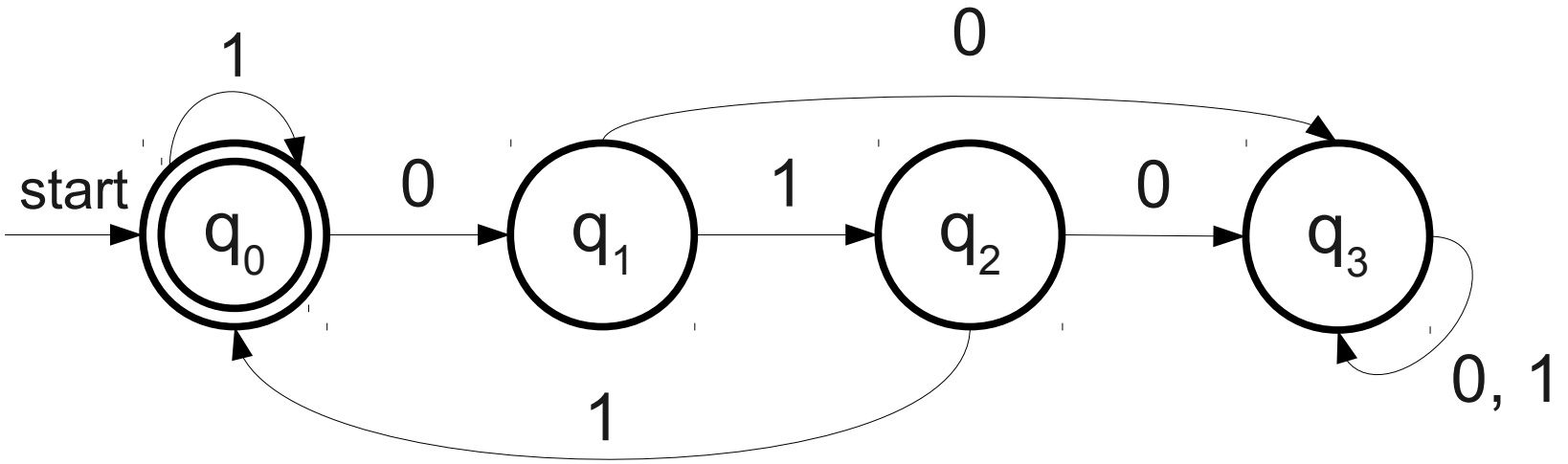
	0	1
q_0	q_1	
q_1		
q_2		
q_3		

Tabular DFAs



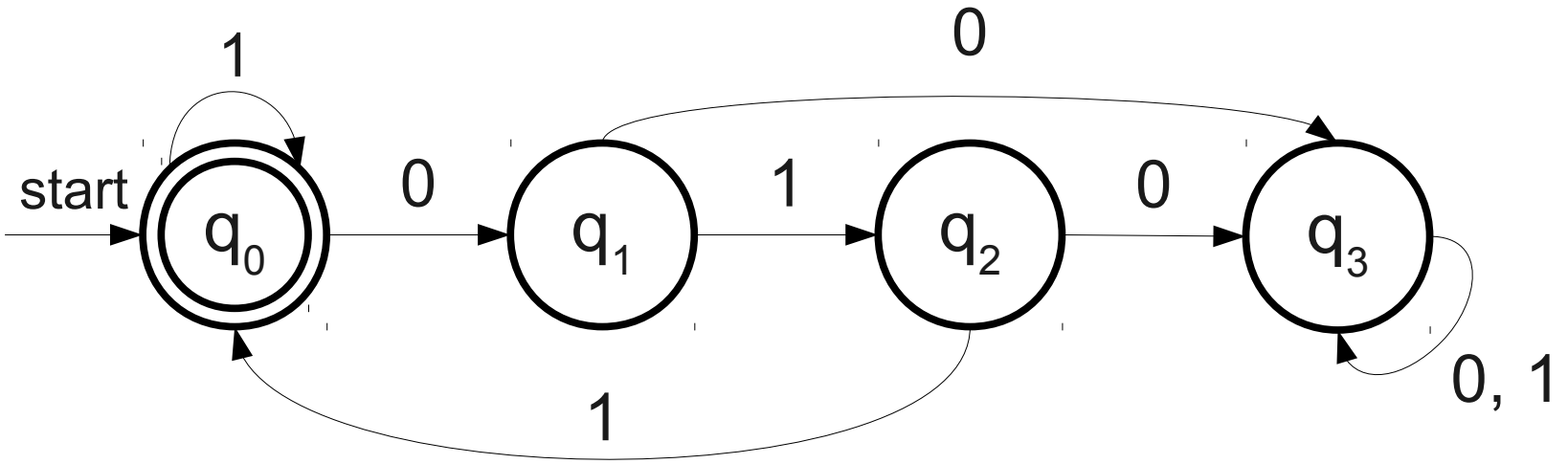
	0	1
q_0	q_1	q_0
q_1		
q_2		
q_3		

Tabular DFAs



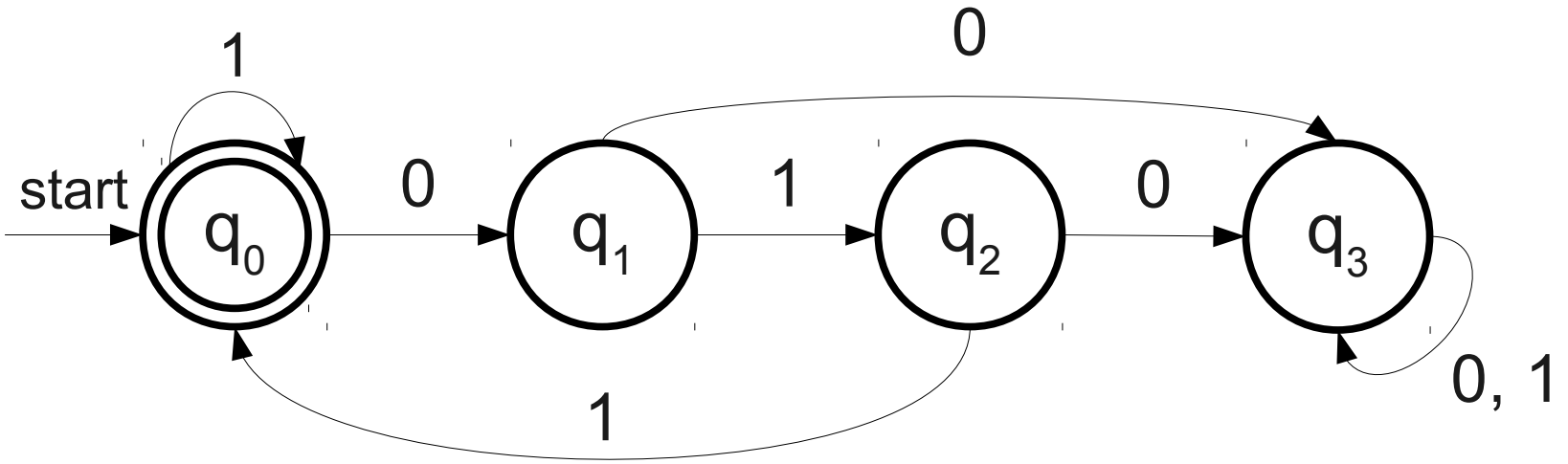
	0	1
q_0	q_1	q_0
q_1	q_3	
q_2		
q_3		

Tabular DFAs



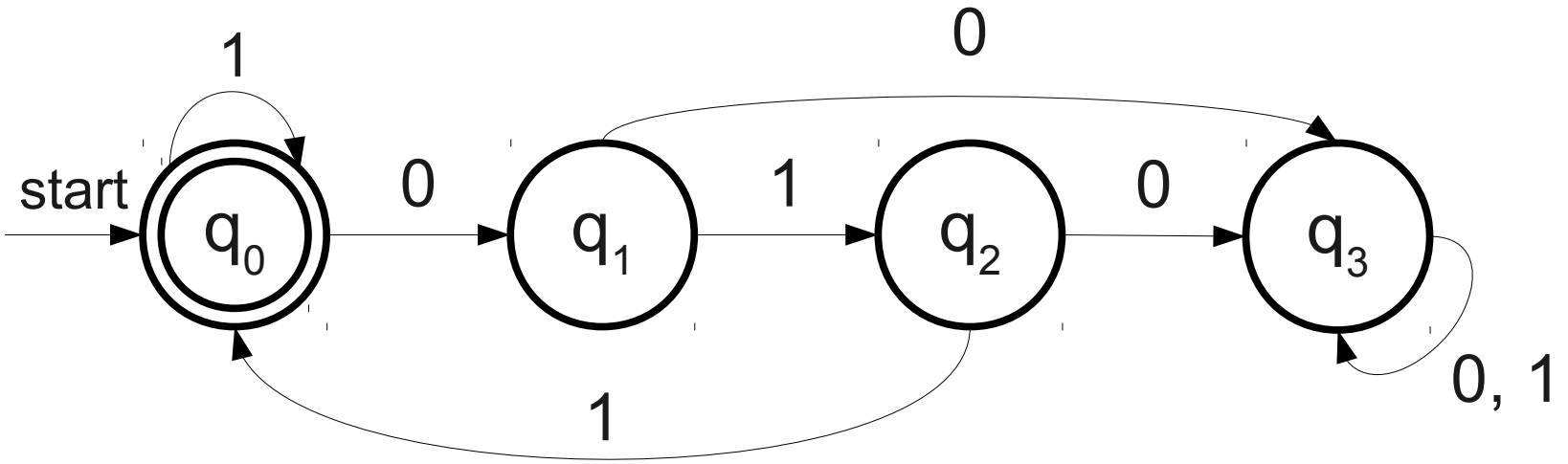
	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2		
q_3		

Tabular DFAs



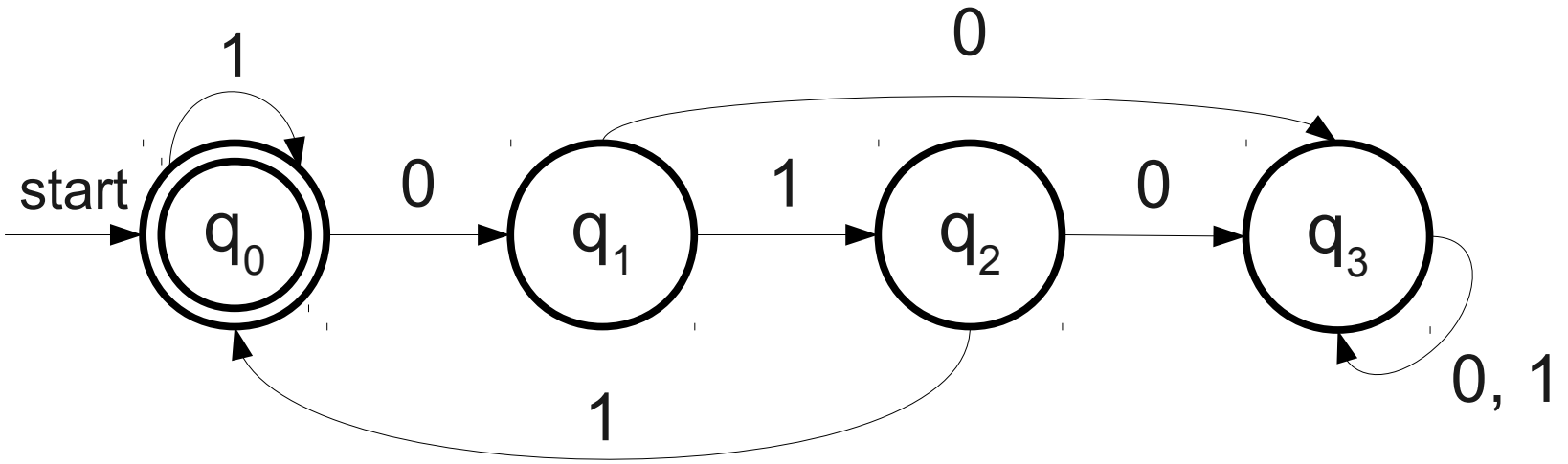
	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	
q_3		

Tabular DFAs



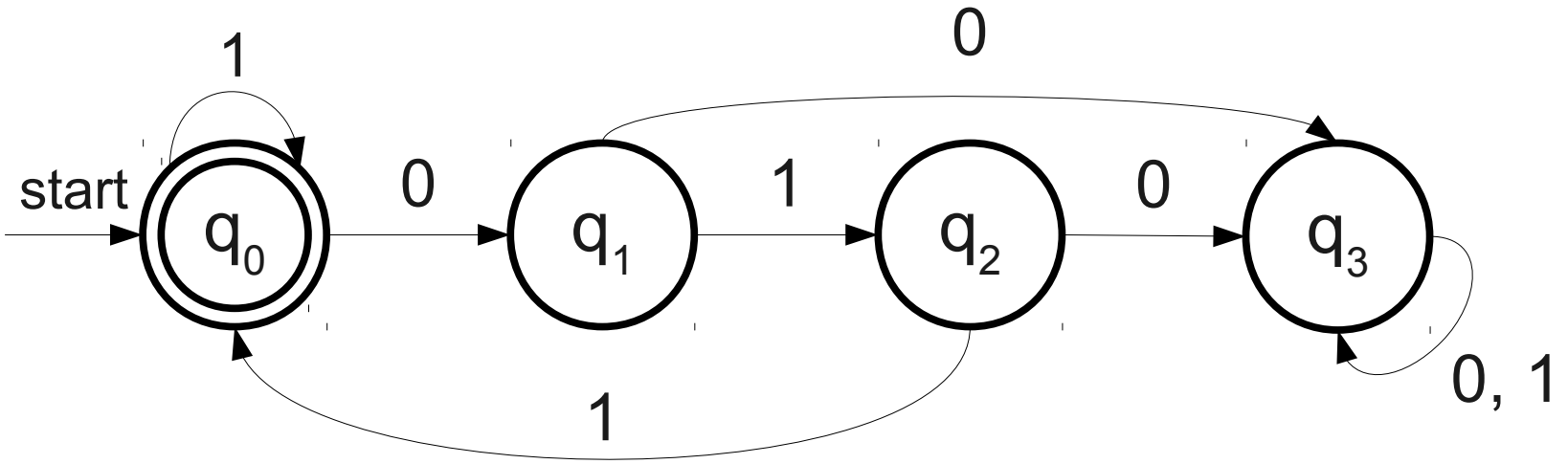
	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3		

Tabular DFAs



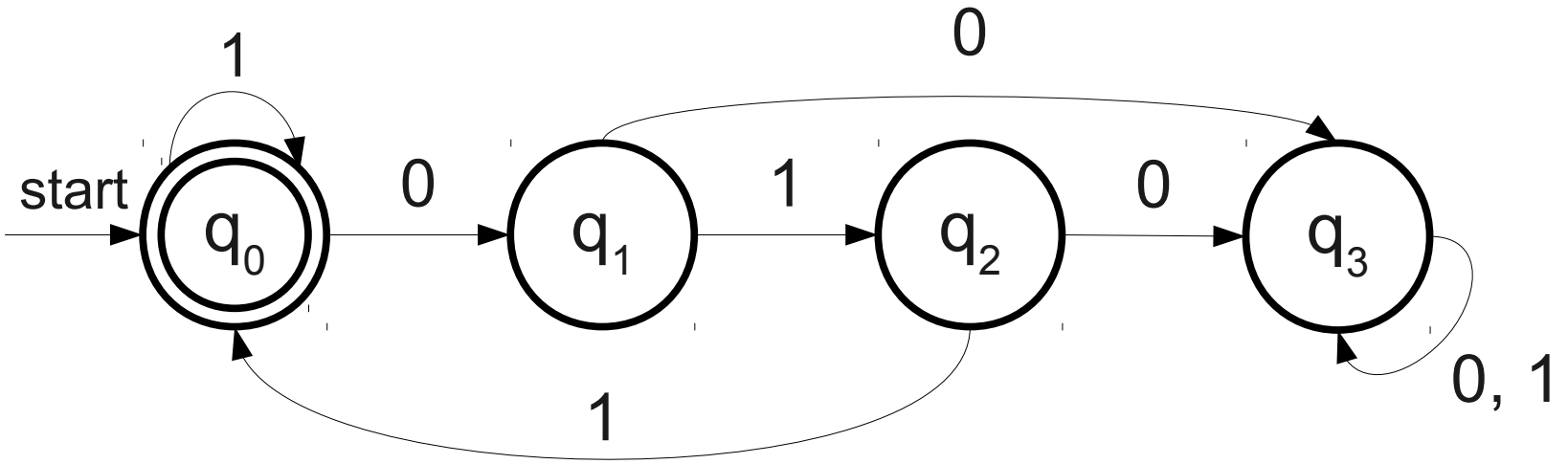
	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3	q_3	

Tabular DFAs



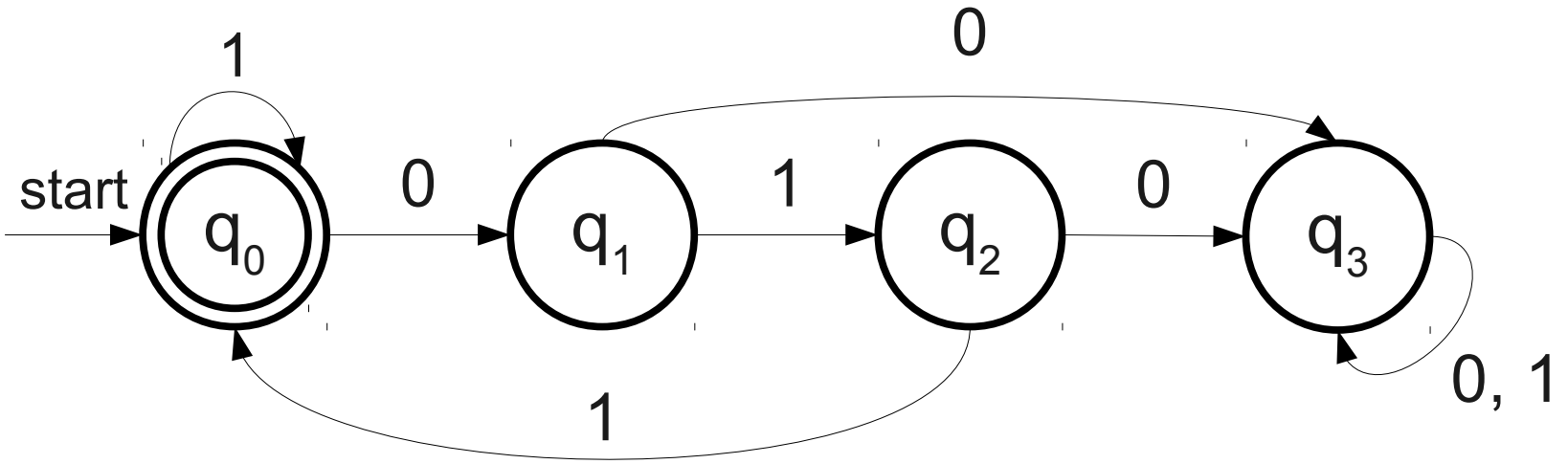
	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3	q_3	q_3

Tabular DFAs



	0	1
$*q_0$	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3	q_3	q_3

Tabular DFAs



	0	1
*q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3	q_3	q_3

The star indicates that this is an accepting state.

Code!? In a Theory Class!?

```
int kTransitionTable[kNumStates][kNumSymbols] = {
     {0, 0, 1, 3, 7, 1, …},
      …
};
bool kAcceptTable[kNumStates] = {
    false,
    true,
    true,
    …
};
bool SimulateDFA(string input) {
    int state = 0;
    for (char ch in input)
        state = kTransitionTable[state][ch];
    return kAcceptTable[state];
}
```

Back to the Math...

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.
  - Σ is an alphabet.

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.
  - Σ is an alphabet.
  - $\delta : Q \times \Sigma \to Q$ is the **transition function**.

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.
  - Σ is an alphabet.
  - $\delta : Q \times \Sigma \to Q$ is the **transition function**.

	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3	q_3	q_3

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.
  - Σ is an alphabet.
  - $\delta : Q \times \Sigma \to Q$ is the **transition function**.

	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_3	q_0
q_3	q_3	q_3

δ is a function, so there must be exactly one transition defined for each state/symbol pair.

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.
  - Σ is an alphabet.
  - $\delta : Q \times \Sigma \to Q$ is the **transition function**.
  - $q_0 \in Q$ is the **start state**.

A Formal Definition of DFAs

- Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a set of states.
  - Σ is an alphabet.
  - $\delta : Q \times \Sigma \to Q$ is the **transition function**.
  - $q_0 \in Q$ is the **start state**.
  - $F \subseteq Q$ is a set of **accepting states**.

A Formal Definition of Acceptance

- Given a DFA $(Q, \Sigma, \delta, q_0, F)$, we want to find some way to formally define what it means for the DFA to accept a string $w \in \Sigma^*$.
- **Idea**: Define a function $\delta^* : \Sigma^* \to Q$ that says what state we end up in if we run the DFA on a given string.
- This function represents the effect of running the computer on a given input.

A Formal Definition of Acceptance

- **Mathematical Notation**: If *w* is a string and *a* is a character, then *w*a is the string formed by appending a to *w*.
- Given a DFA $(Q, \Sigma, \delta, q_0, F)$, δ^* is defined recursively.
- $\delta^* (\varepsilon) = q_0$
  - Running the automaton on ε ends in the start state.
- $\delta^* (wa) = \delta(\delta^* (w), a)$
  - Running on *w*a is equal to running the automaton on *w*, then following the transition for a.

A Formal Definition of Acceptance

- Using our δ^* function, we can define the **language of a DFA**.
- Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA.
- Then $L(D) = \{ w \mid \delta^*(w) \in F \}$
  - The set of strings w that cause the DFA to end up in an accepting state.

So What?

- We now have a mathematically rigorous way of defining whether a DFA accepts a string.
- We can try making changes to DFAs and can formally prove how those changes transform the language of the DFA.

The Complement of a Language

- Given a language $L \subseteq \Sigma^*$, the **complement** of that language (denoted $\overline{L}$) is the language of all strings not in L.
- Formally:

$$\overline{L} = \{ w \mid w \in \Sigma^* \wedge w \notin L \}$$

The Complement of a Language

- Given a language $L \subseteq \Sigma^*$, the **complement** of that language (denoted $\overline{L}$) is the language of all strings not in L.
- Formally:

$$\overline{L} = \{ w \mid w \in \Sigma^* \wedge w \notin L \}$$

The Complement of a Language

- Given a language $L \subseteq \Sigma^*$, the **complement** of that language (denoted $\overline{L}$) is the language of all strings not in L.
- Formally:

$$\overline{L} = \Sigma^* - L$$

The Complement of a Language

- Given a language $L \subseteq \Sigma^*$, the **complement** of that language (denoted $\overline{L}$) is the language of all strings not in L.
- Formally:

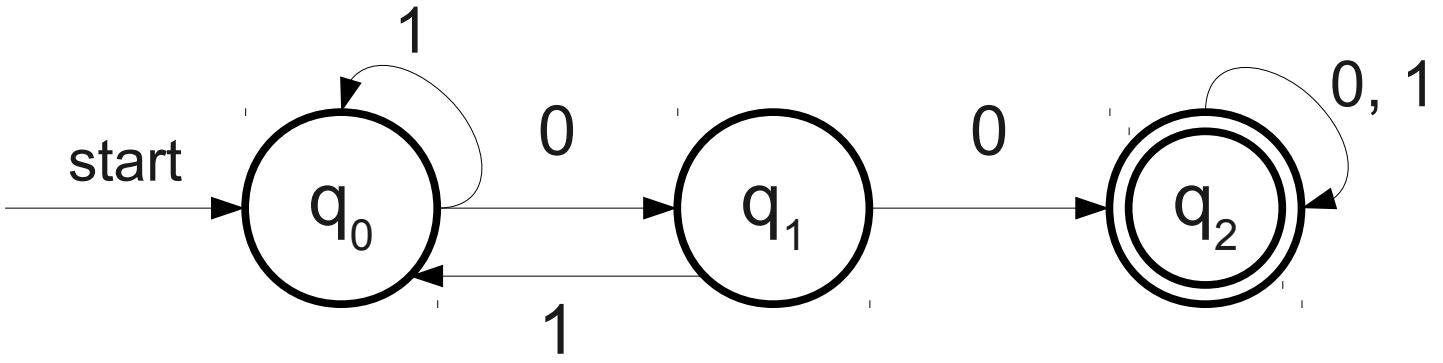
$$\overline{L} = \Sigma^* - L$$

Complementing Regular Languages

- Recall: A **regular language** is a language accepted by some DFA.
- **Question:** If L is a regular language, is $\overline{L}$ a regular language?
- If the answer is “yes,” then there must be some way to construct a DFA for $\overline{L}$.
- If the answer is “no,” then some language L can be accepted by a DFA, but $\overline{L}$ cannot be accepted by any DFA.

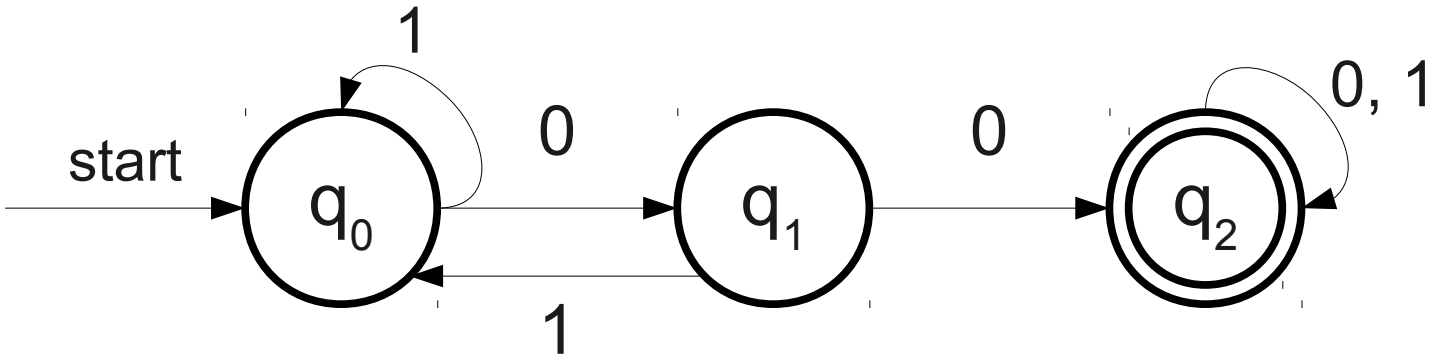
Complementing Regular Languages

L = { w | w contains 00 as a substring }



Complementing Regular Languages

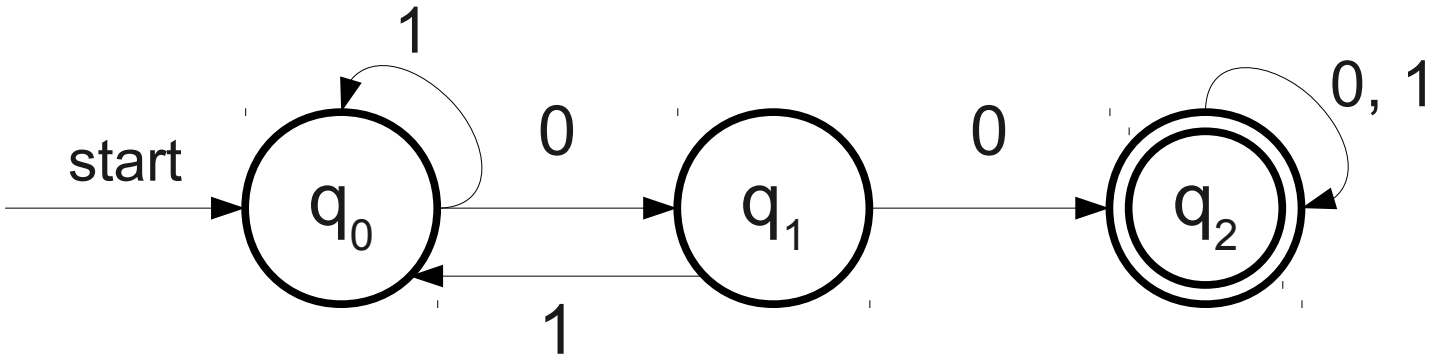
L = { w | w contains 00 as a substring }



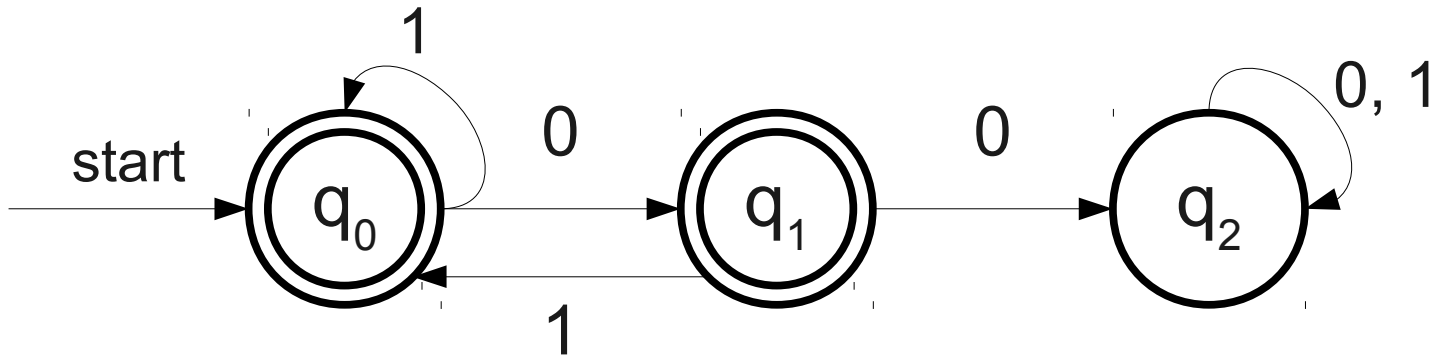
$\overline{L}$ = { w | w **does not contain** 00 as a substring }

Complementing Regular Languages

L = { w | w contains 00 as a substring }

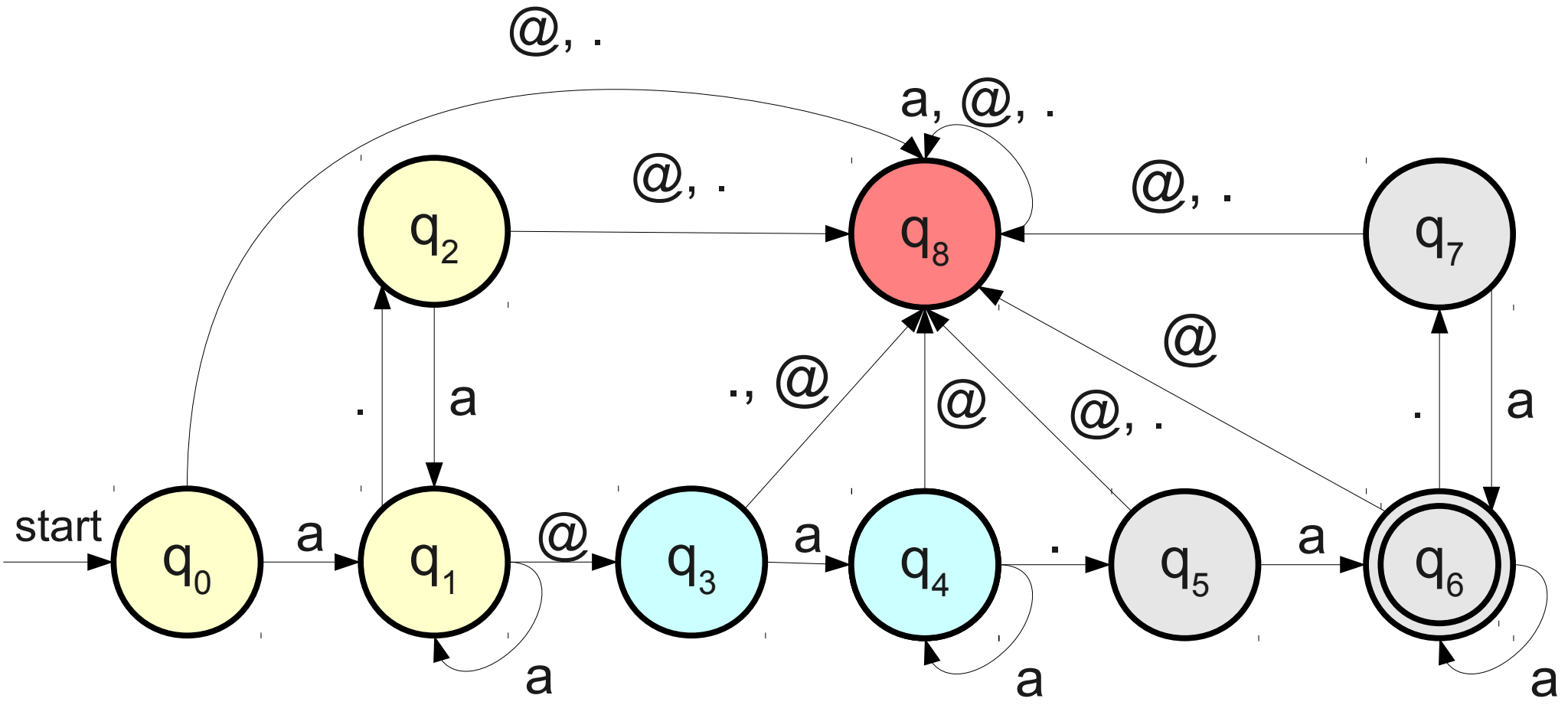


$\overline{L}$ = { w | w **does not contain** 00 as a substring }



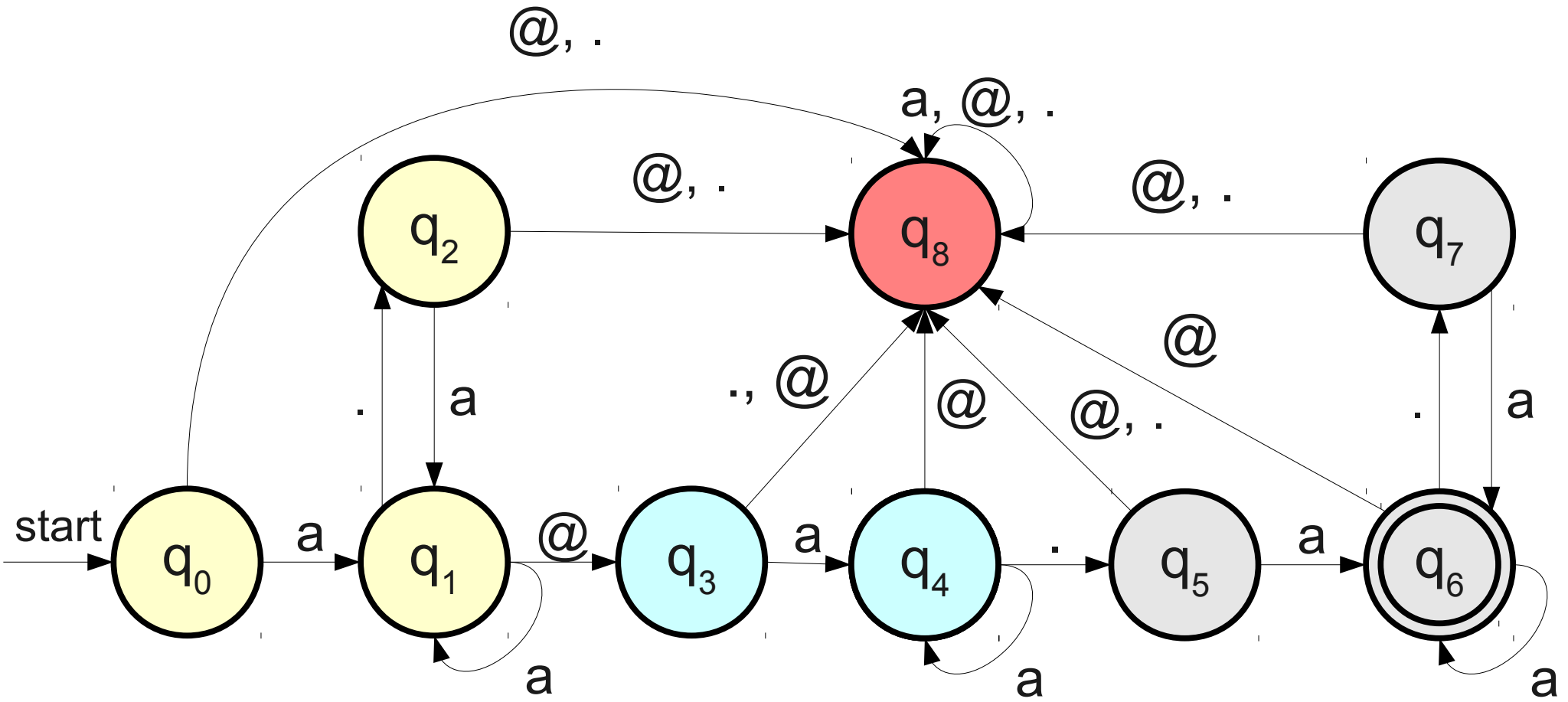
Complementing Regular Languages

L = { w | w is a legal email address }



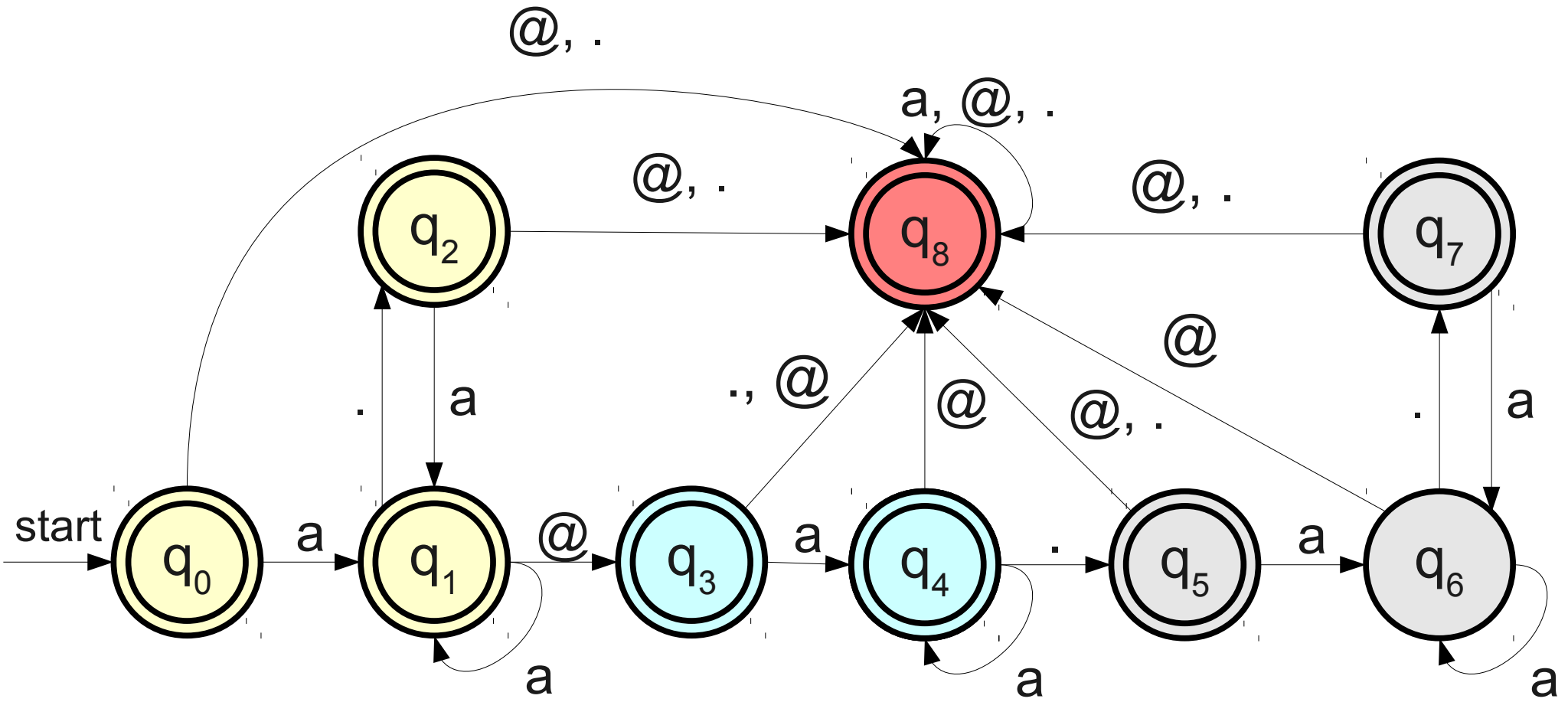
Complementing Regular Languages

$\overline{L}$ = { w | w is **not** a legal email address }



Complementing Regular Languages

$\overline{L}$ = { w | w is **not** a legal email address }



Constructions on Automata

- Much of our discussion of automata will consider **constructions** that transform one automaton into another.
- Exchanging accepting and rejecting states is a simple construction sometimes called the **complement construction**.
- Does this construction always work?
- How would we prove it?

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$.

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$.

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$. Thus $\delta^*(w) \in Q$ means that $w \in \Sigma^*$.

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$. Thus $\delta^*(w) \in Q$ means that $w \in \Sigma^*$. So

$$L(D') = \{ w \mid w \in \Sigma^* \} - \{ w \mid \delta^*(w) \in F \}$$

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$. Thus $\delta^*(w) \in Q$ means that $w \in \Sigma^*$. So

$$L(D') = \{ w \mid w \in \Sigma^* \} - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - \{ w \mid \delta^*(w) \in F \}$$

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$. Thus $\delta^*(w) \in Q$ means that $w \in \Sigma^*$. So

$$L(D') = \{ w \mid w \in \Sigma^* \} - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - L(D)$$

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$. Thus $\delta^*(w) \in Q$ means that $w \in \Sigma^*$. So

$$L(D') = \{ w \mid w \in \Sigma^* \} - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - L(D)$$
$$L(D') = \overline{L(D)}.$$

Theorem: If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA with language $L(D)$, then the DFA $D' = (Q, \Sigma, \delta, q_0, Q - F)$ has language $\overline{L(D)}$.

Proof: By definition, $L(D') = \{ w \mid \delta^*(w) \in Q - F \}$. So

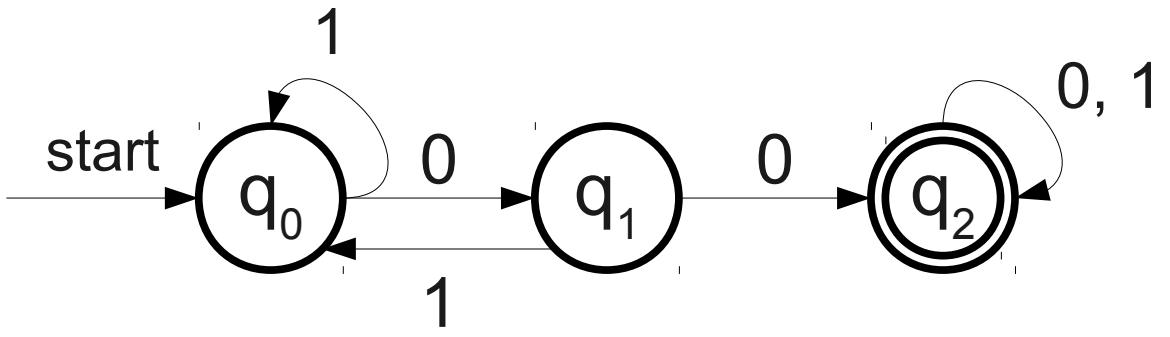
$$L(D') = \{ w \mid \delta^*(w) \in Q \wedge \delta^*(w) \notin F \}$$
$$L(D') = \{ w \mid \delta^*(w) \in Q \} - \{ w \mid \delta^*(w) \in F \}$$

Since $\delta^*(w) : \Sigma^* \rightarrow Q$, any string w satisfies $\delta^*(w) \in Q$. Thus $\delta^*(w) \in Q$ means that $w \in \Sigma^*$. So

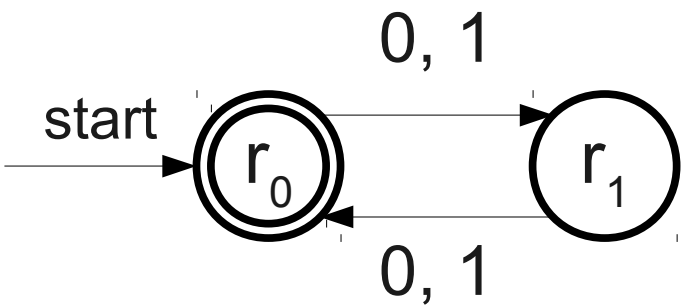
$$L(D') = \{ w \mid w \in \Sigma^* \} - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - \{ w \mid \delta^*(w) \in F \}$$
$$L(D') = \Sigma^* - L(D)$$
$$L(D') = \overline{L(D)}. \blacksquare$$

Intersecting Regular Languages

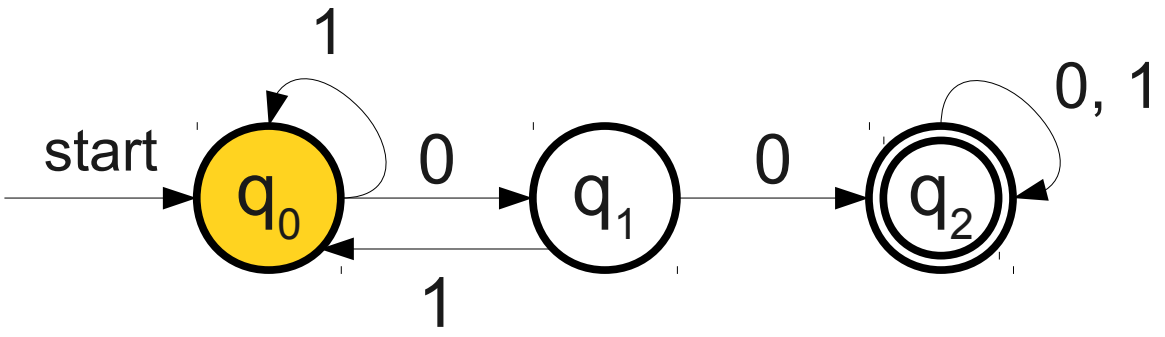
- The **intersection** of two languages L_1 and L_2 is the set $L_1 \cap L_2$.
- Intuitively, the language formed by taking the strings in L_1 and filtering them down by eliminating anything not in L_2.
- **Question:** If L_1 and L_2 are regular languages, is $L_1 \cap L_2$ a regular language?



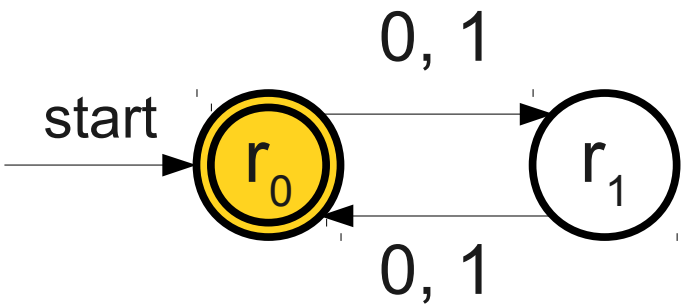
L_1 = { w | w contains 00 as a substring }



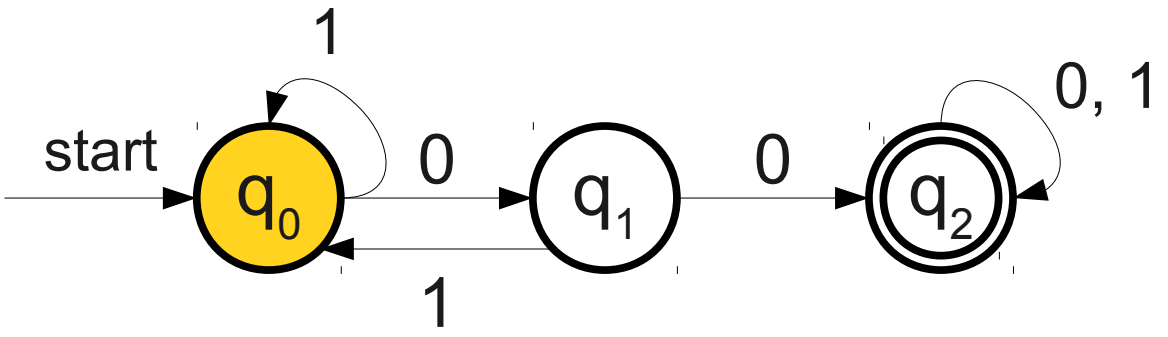
L_2 = { w | w has even length }



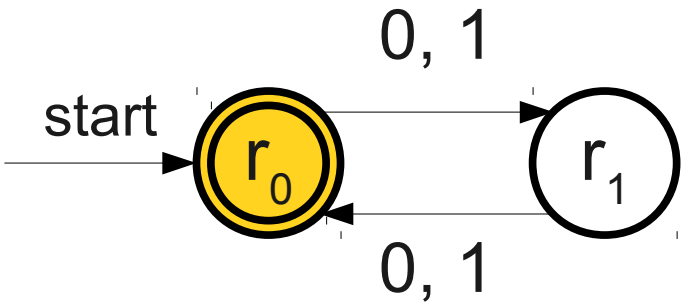
L_1 = { w | w contains 00 as a substring }



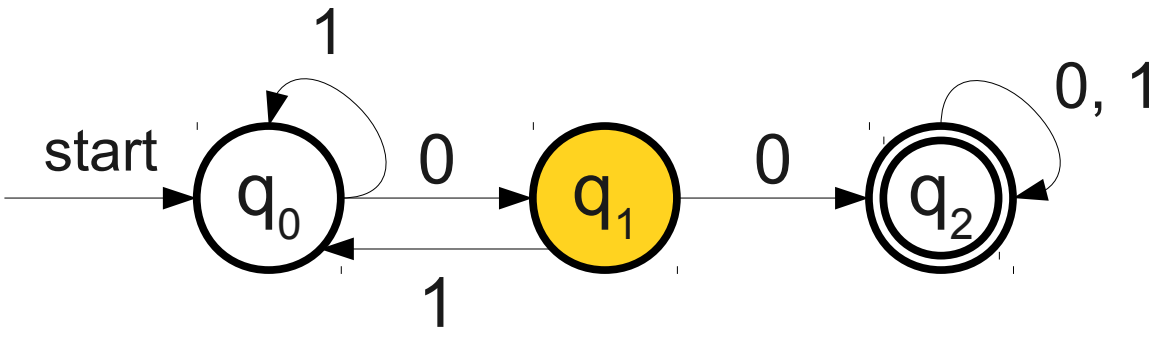
L_2 = { w | w has even length }



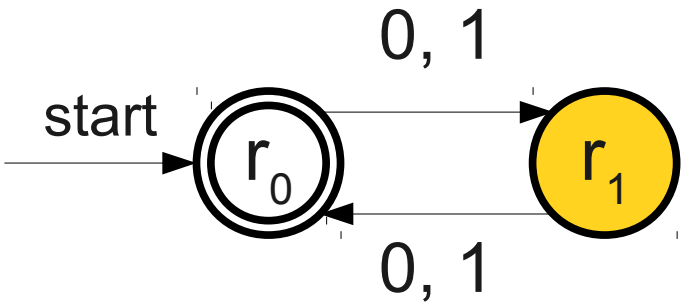
L_1 = { w | w contains 00 as a substring }



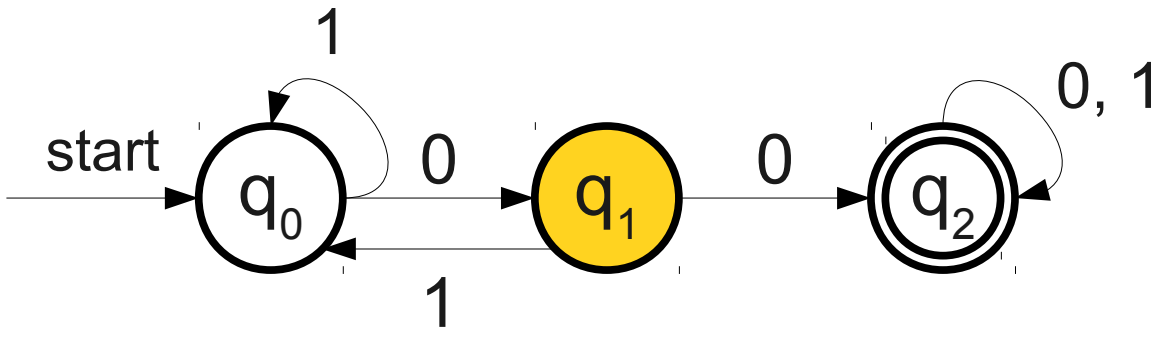
L_2 = { w | w has even length }



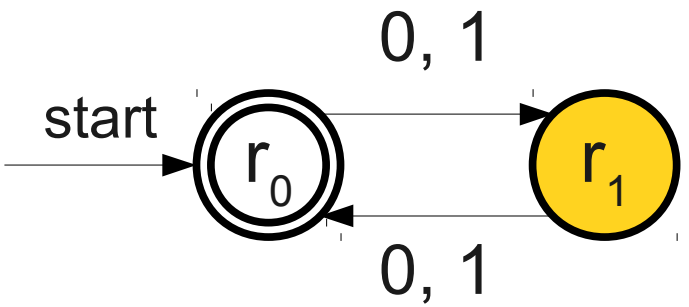
L_1 = { w | w contains 00 as a substring }



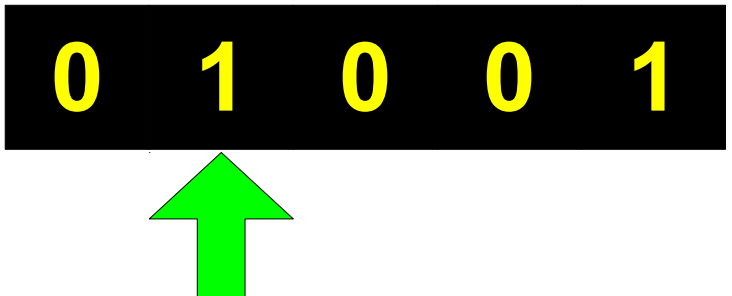
L_2 = { w | w has even length }

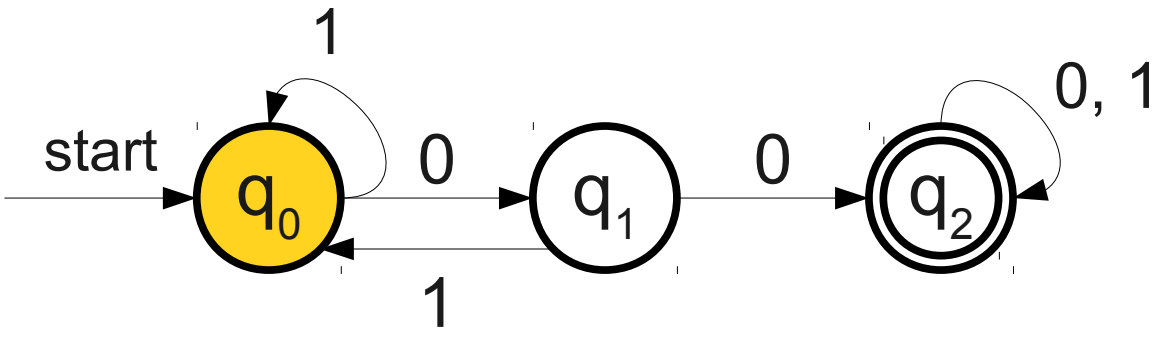


L_1 = { w | w contains 00 as a substring }

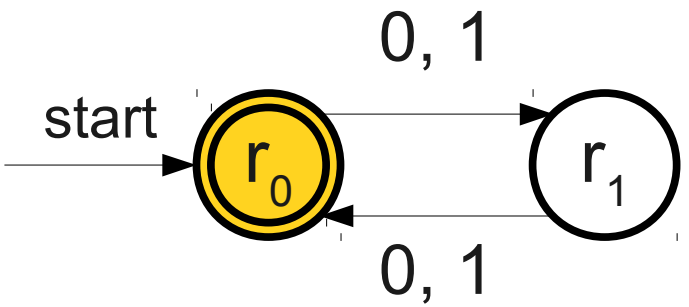


L_2 = { w | w has even length }

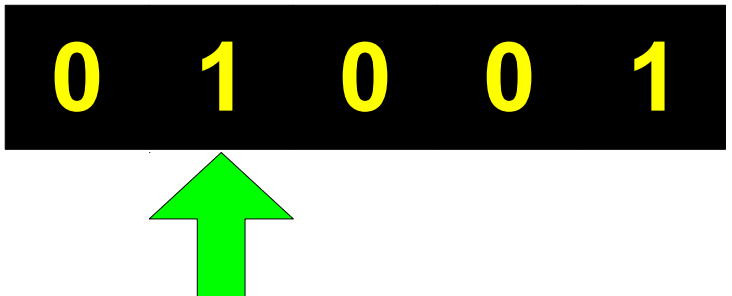


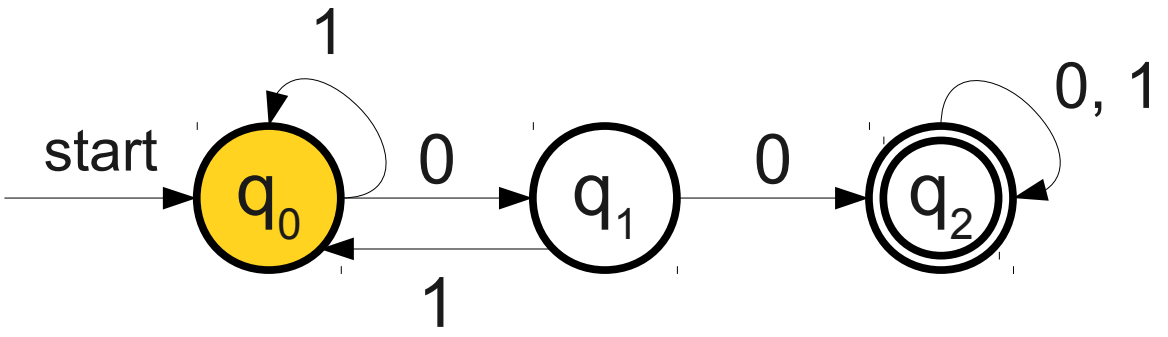


L_1 = { w | w contains 00 as a substring }

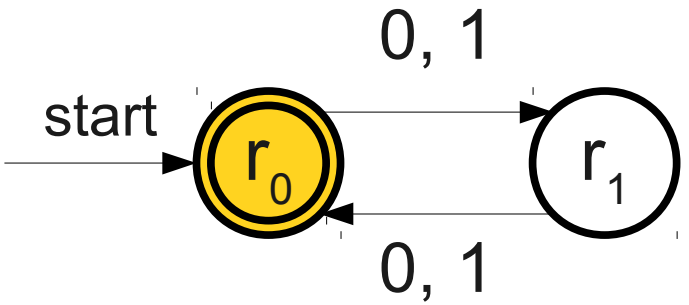


L_2 = { w | w has even length }

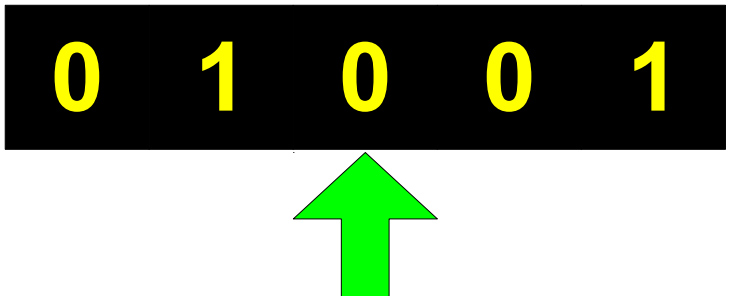


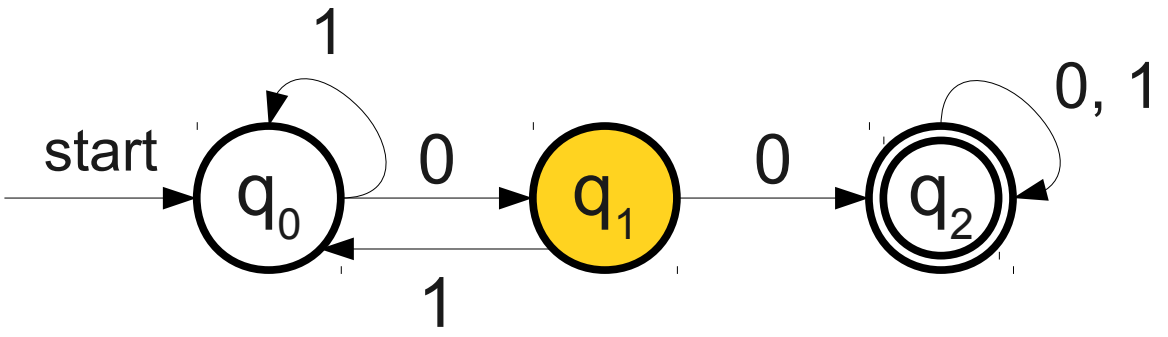


L_1 = { w | w contains 00 as a substring }

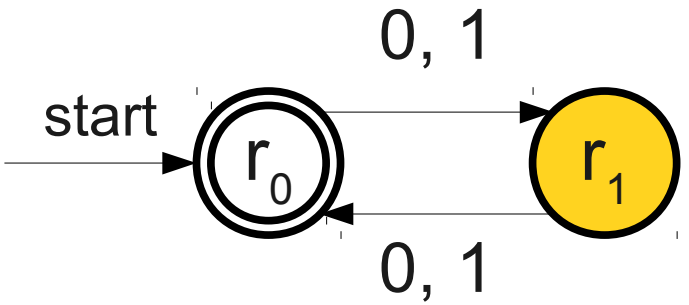


L_2 = { w | w has even length }

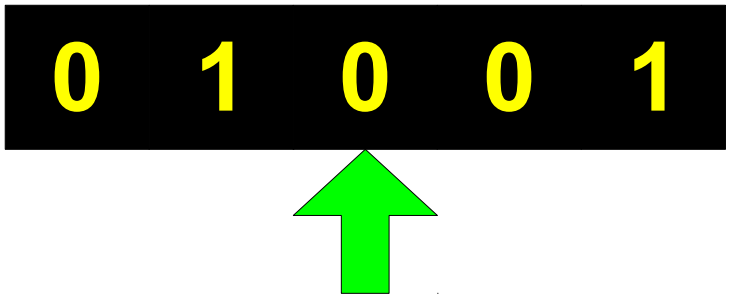


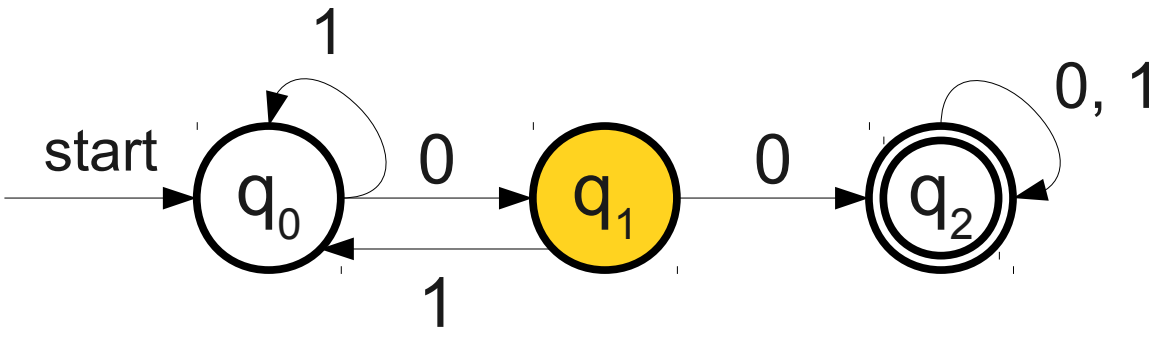


L_1 = { w | w contains 00 as a substring }

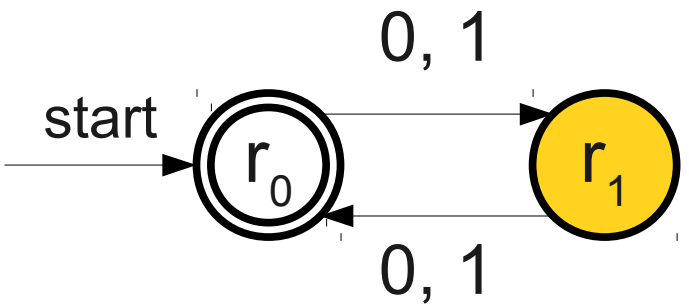


L_2 = { w | w has even length }

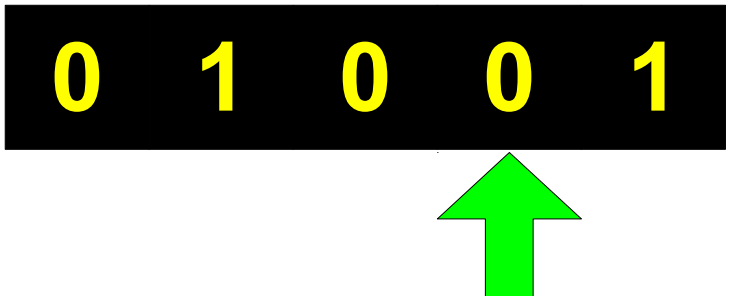


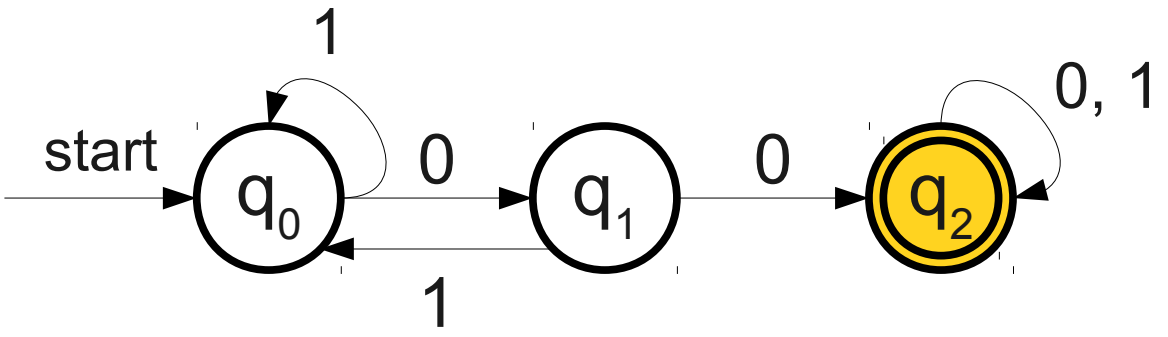


L_1 = { w | w contains 00 as a substring }

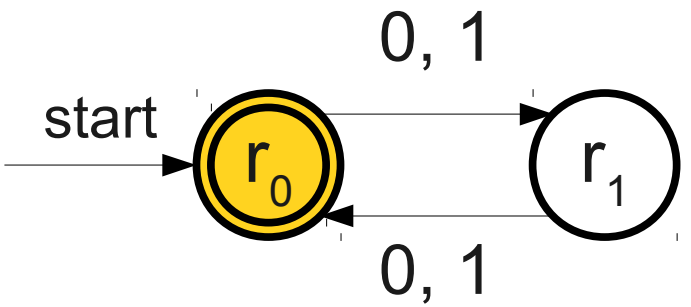


L_2 = { w | w has even length }

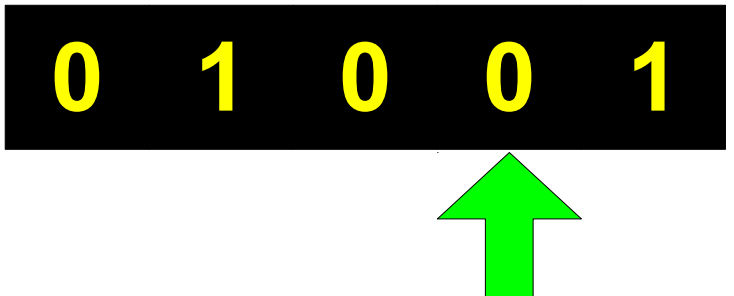


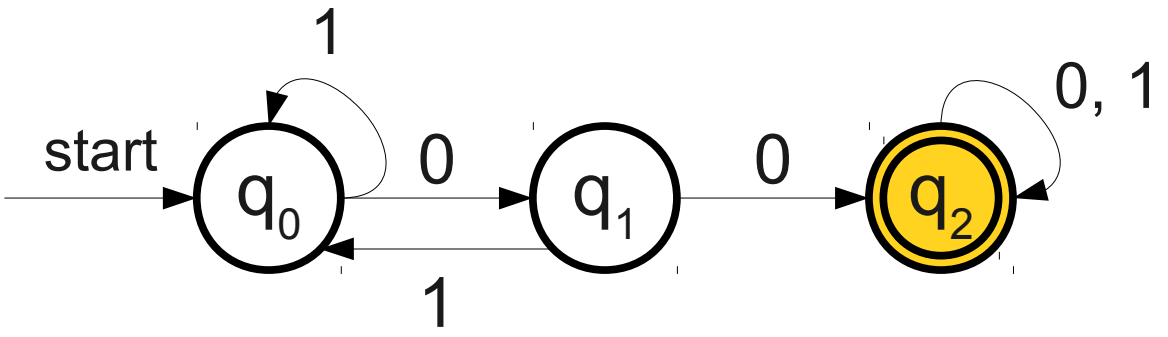


L_1 = { w | w contains 00 as a substring }

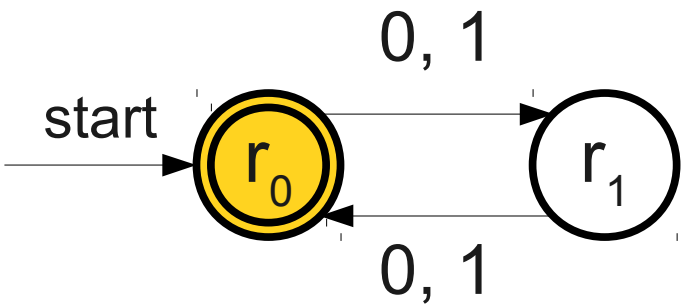


L_2 = { w | w has even length }

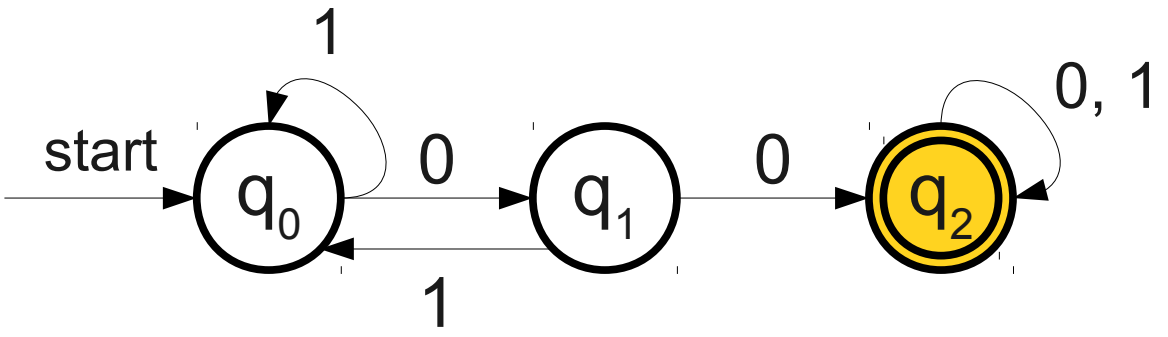




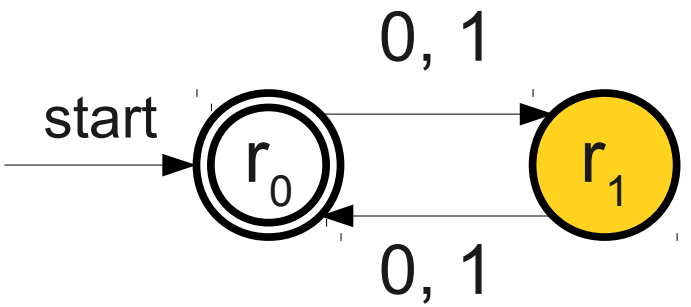
L_1 = { w | w contains 00 as a substring }



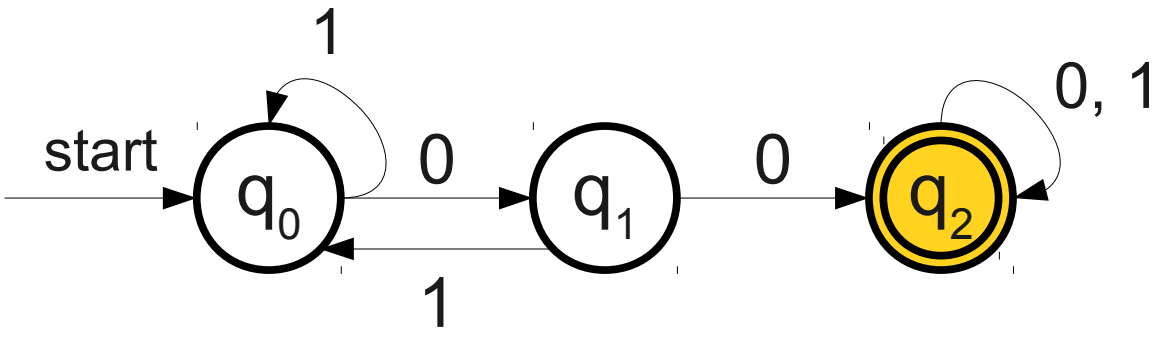
L_2 = { w | w has even length }



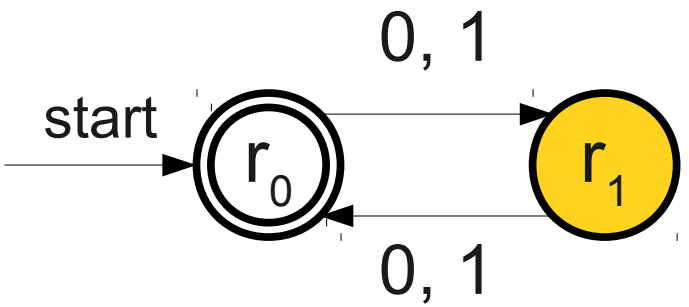
L_1 = { w | w contains 00 as a substring }



L_2 = { w | w has even length }

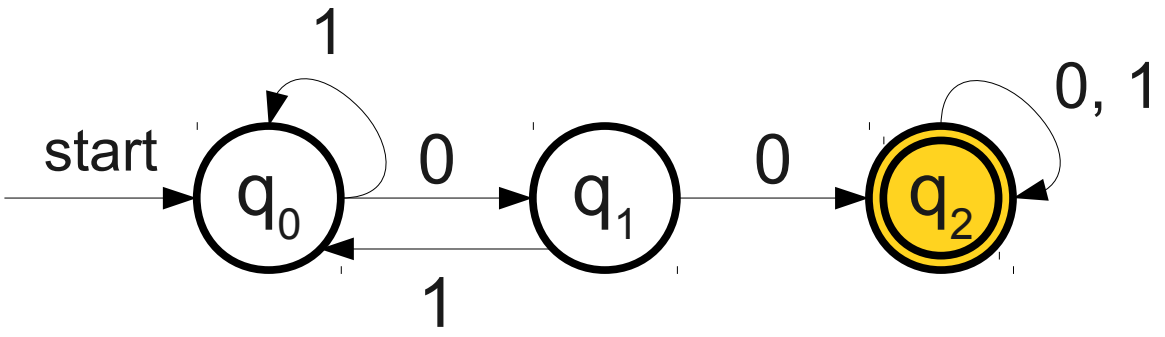


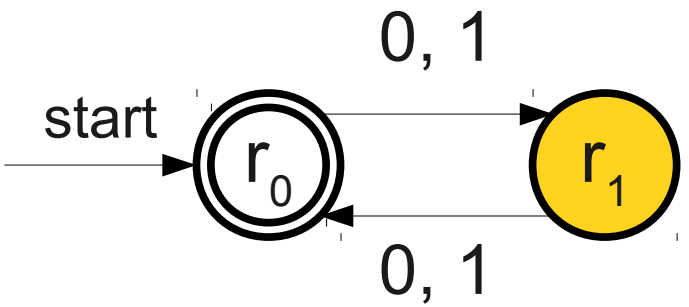
L_1 = { w | w contains 00 as a substring }



L_2 = { w | w has even length }

0 1 0 0 1





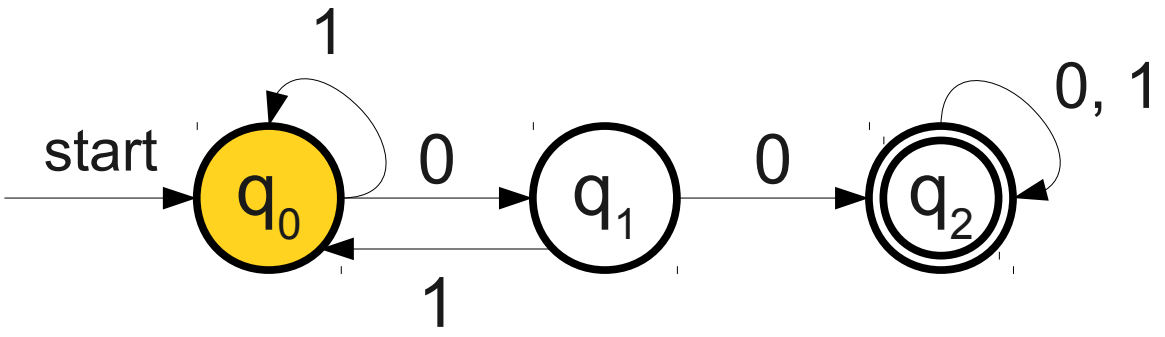
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

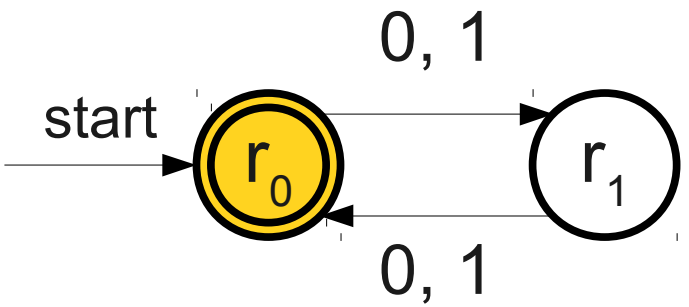
0 1 0 0 1

An Important Concept

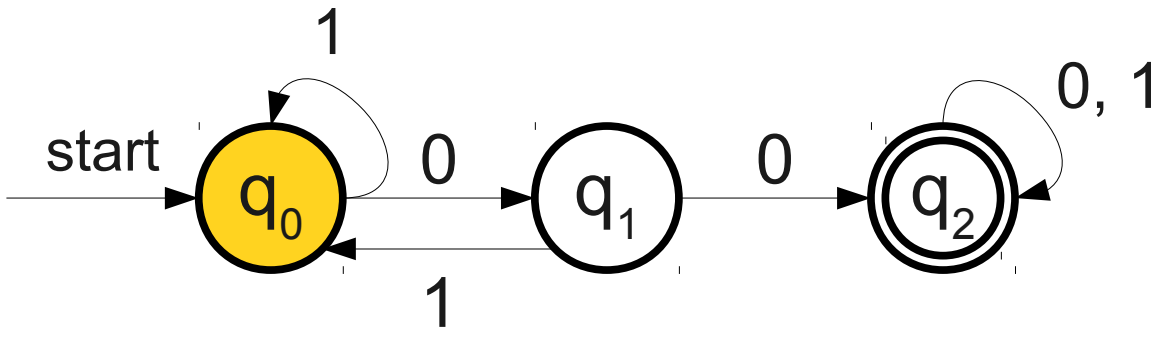
- If we could run both machines at the same time, we could do the following:
  - If both machines accept, accept.
  - If either machine rejects, reject.
- While a human could easily do this, from a computability perspective we'd rather have this done by a machine.
- Is there a way to build a DFA that runs both machines in parallel?
- Key technique: **Simulation**
- Show how to construct a new DFA from the two old DFAs that **simulates** what happens if you run the two DFAs concurrently.



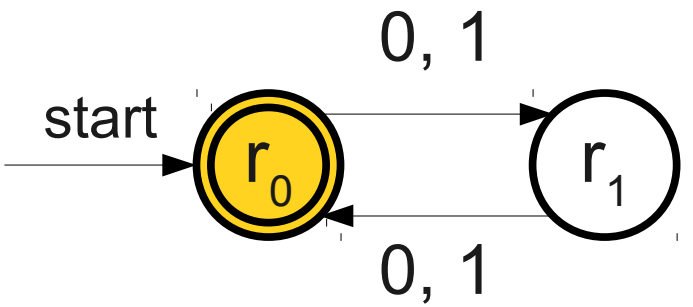
L_1 = { w | w contains 00 as a substring }



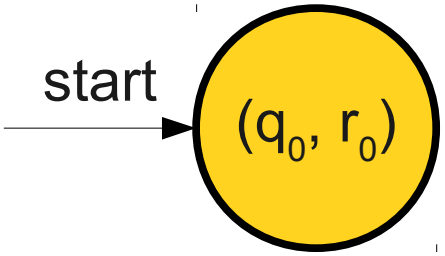
L_2 = { w | w has even length }

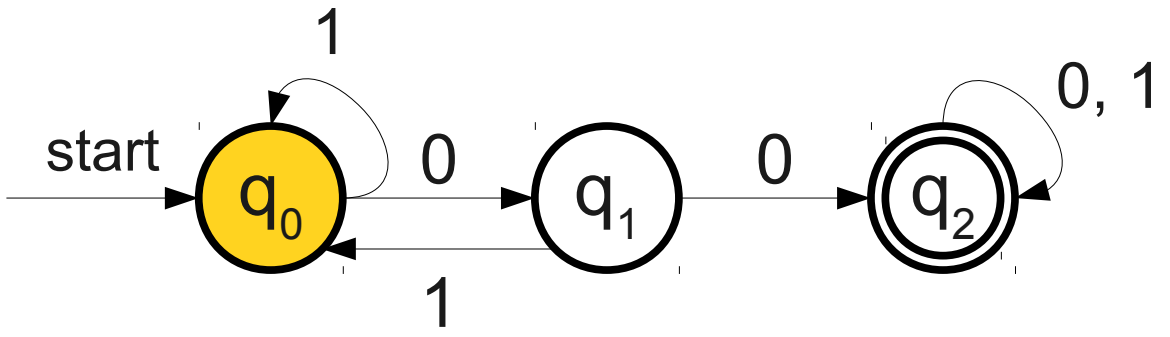


L_1 = { w | w contains 00 as a substring }

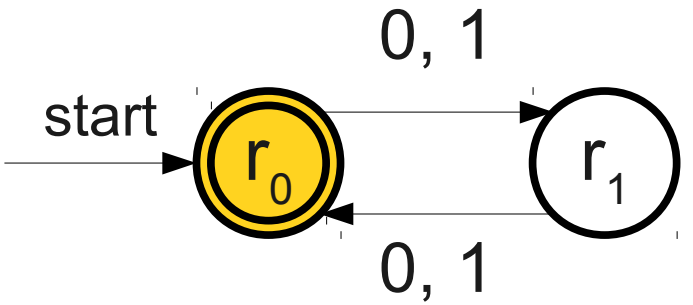


L_2 = { w | w has even length }

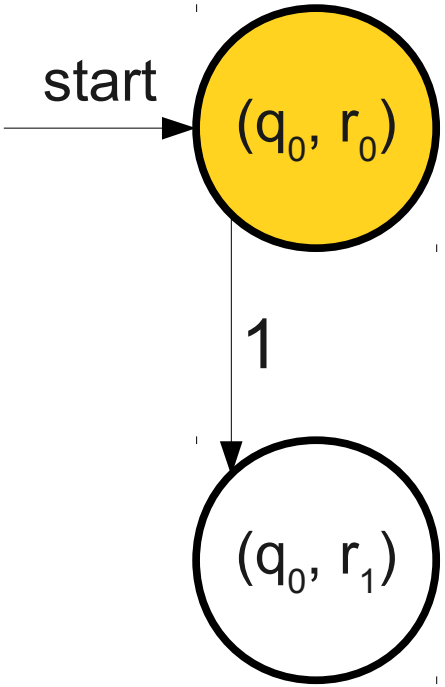


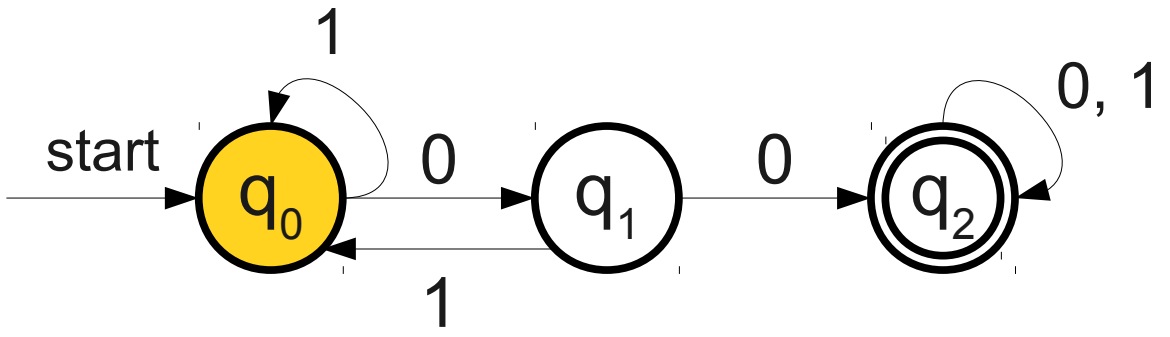


$L_1 = \{ w \mid w \text{ contains } 00 \text{ as a substring} \}$

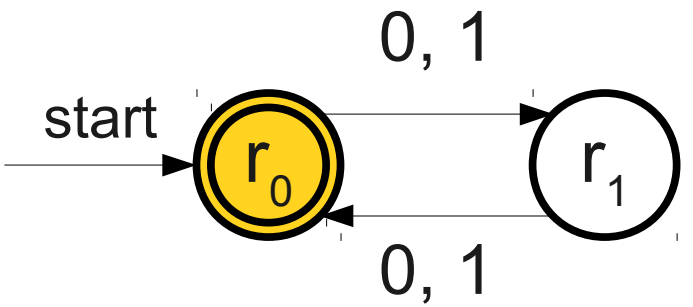


$L_2 = \{ w \mid w \text{ has even length} \}$

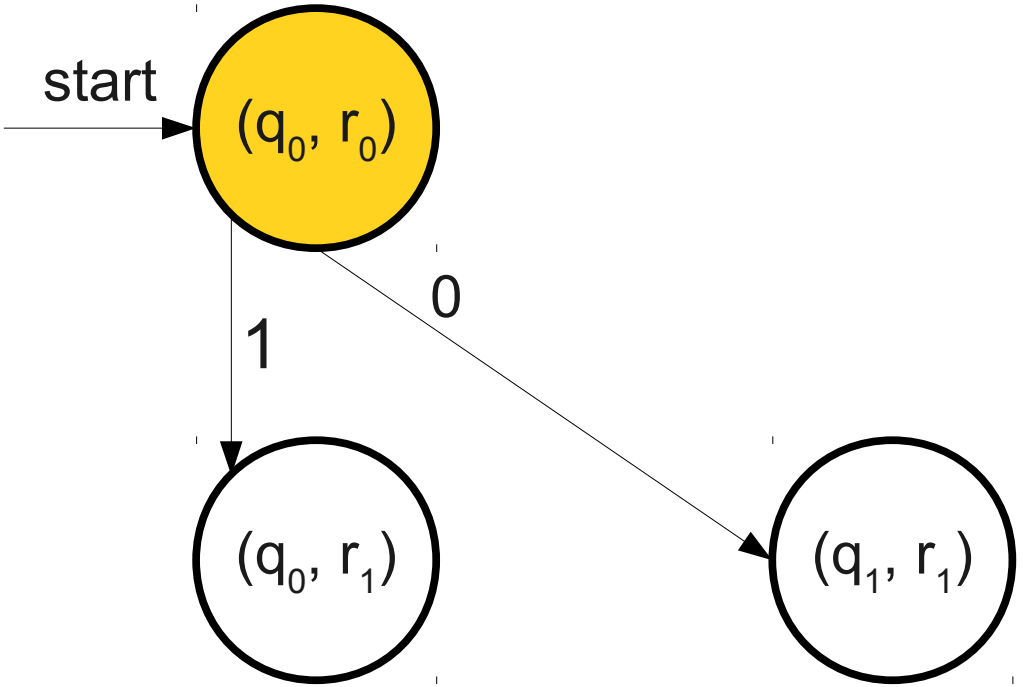


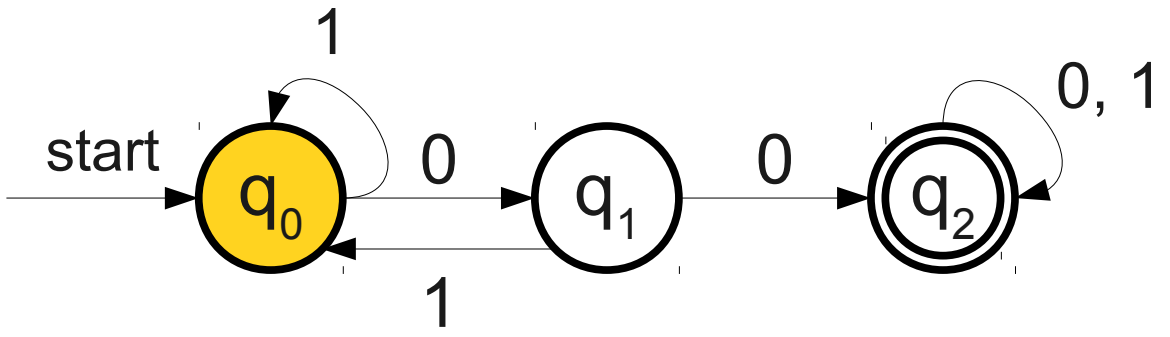


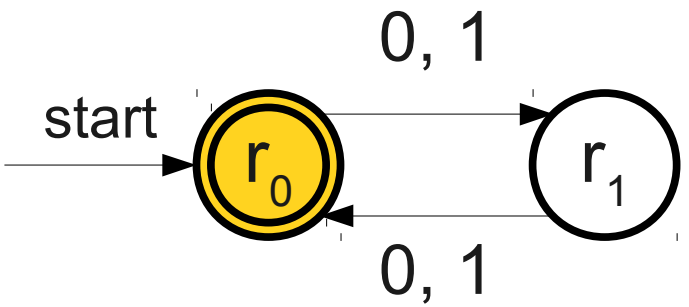
L_1 = { w | w contains 00 as a substring }



L_2 = { w | w has even length }

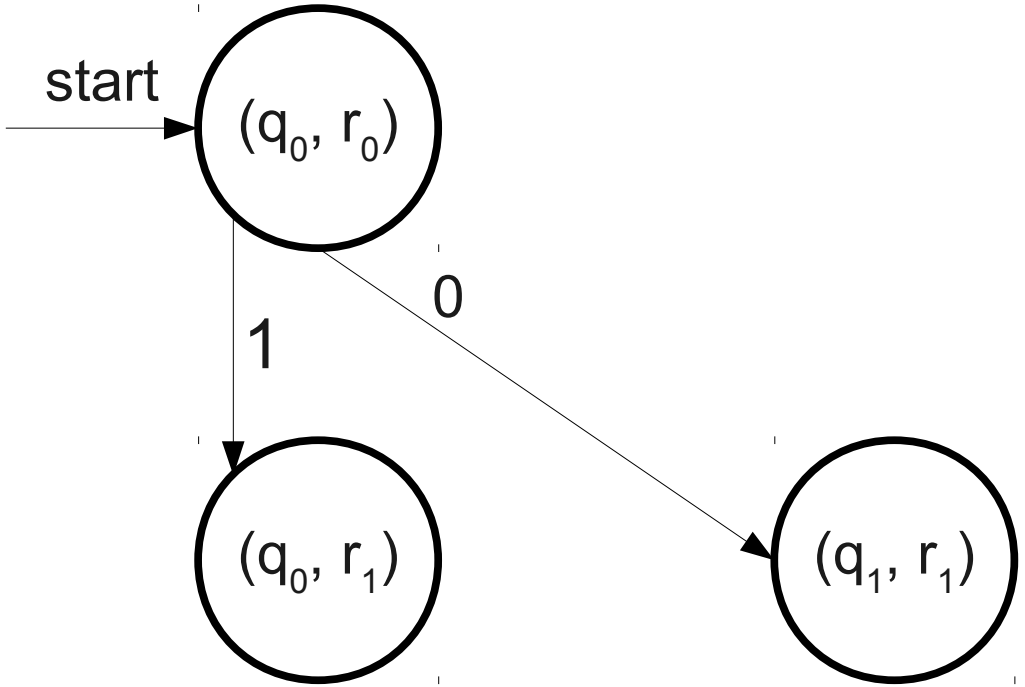


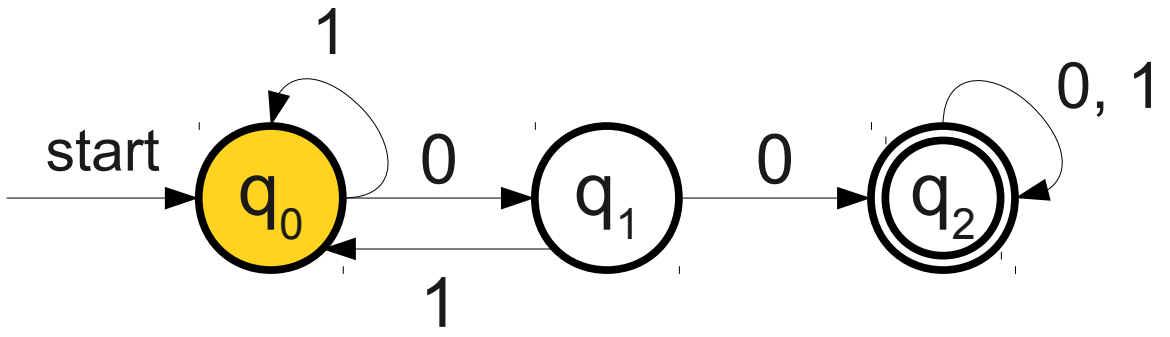


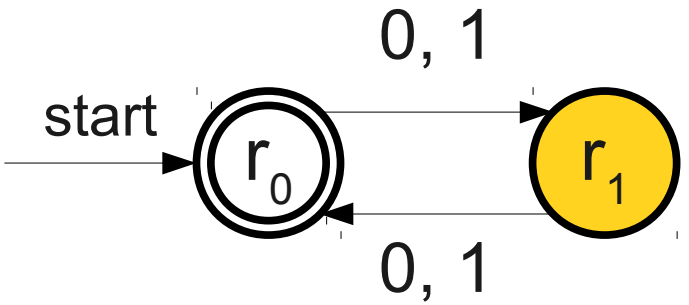


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

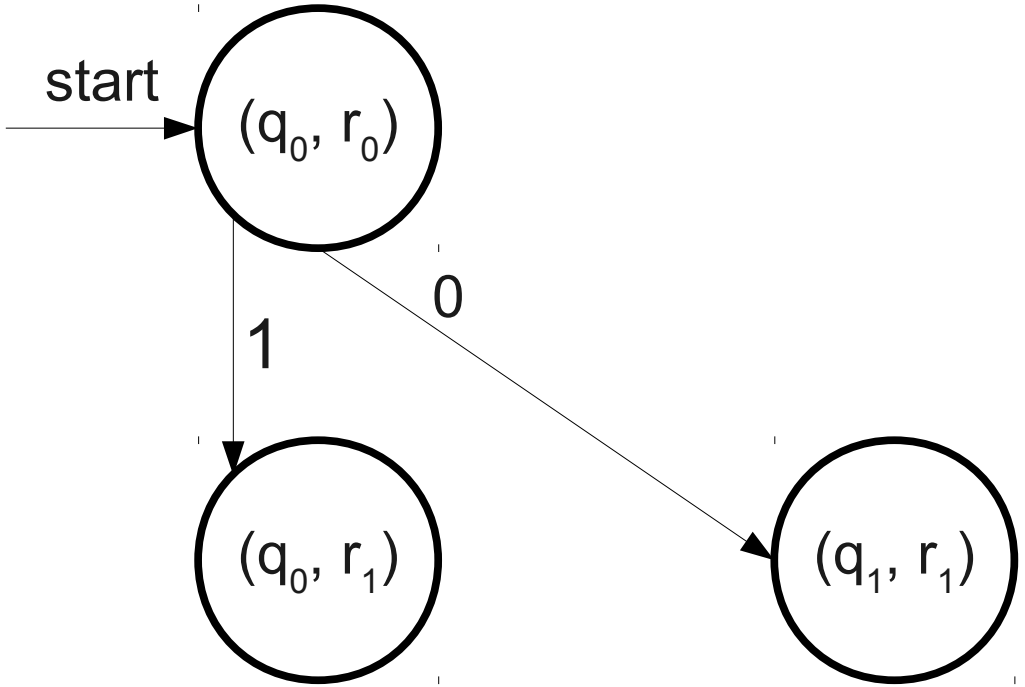


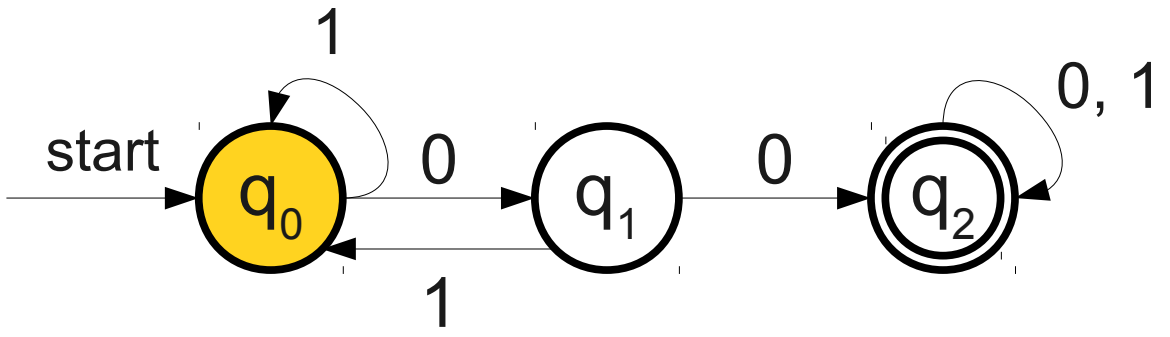




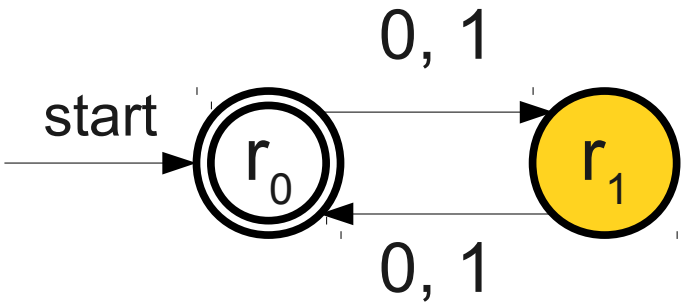
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

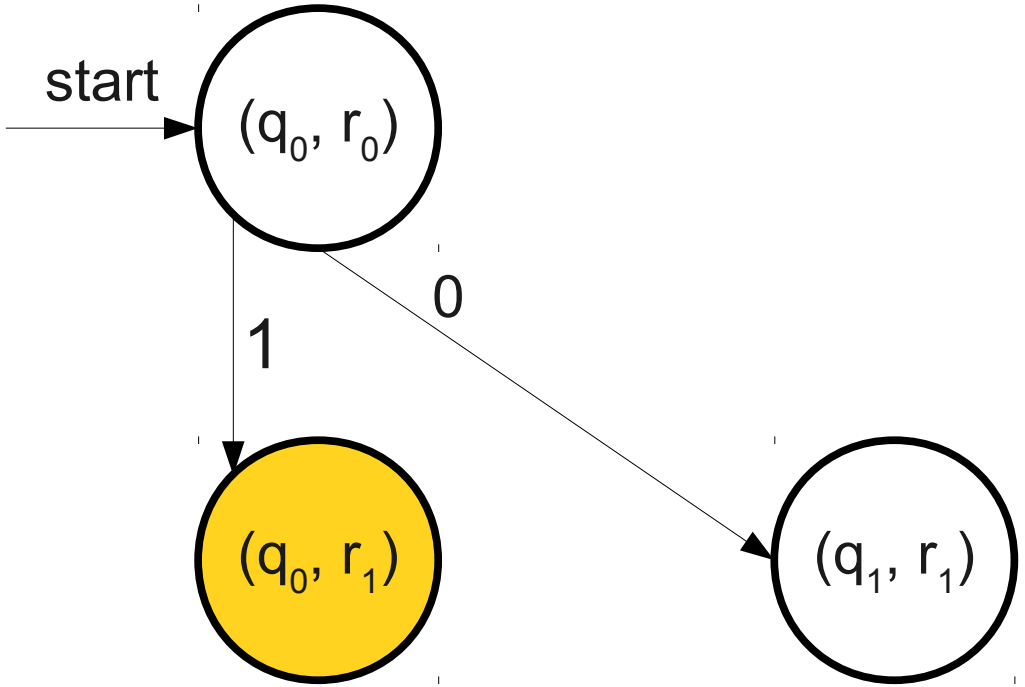


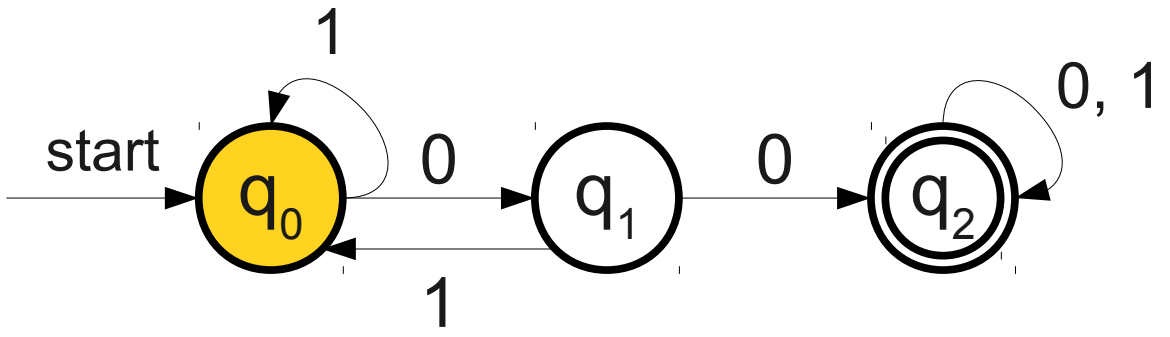


L_1 = { w | w contains 00 as a substring }

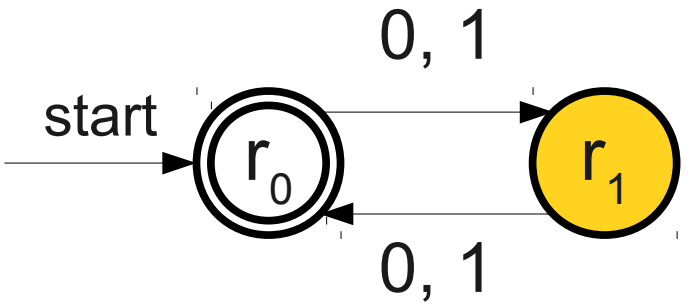


L_2 = { w | w has even length }

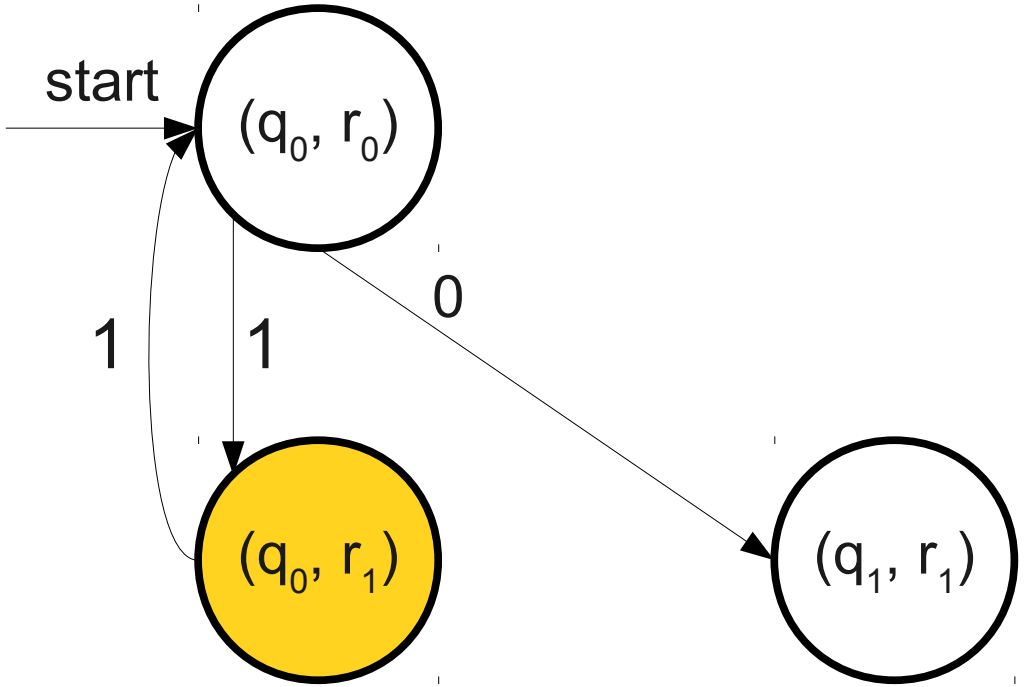


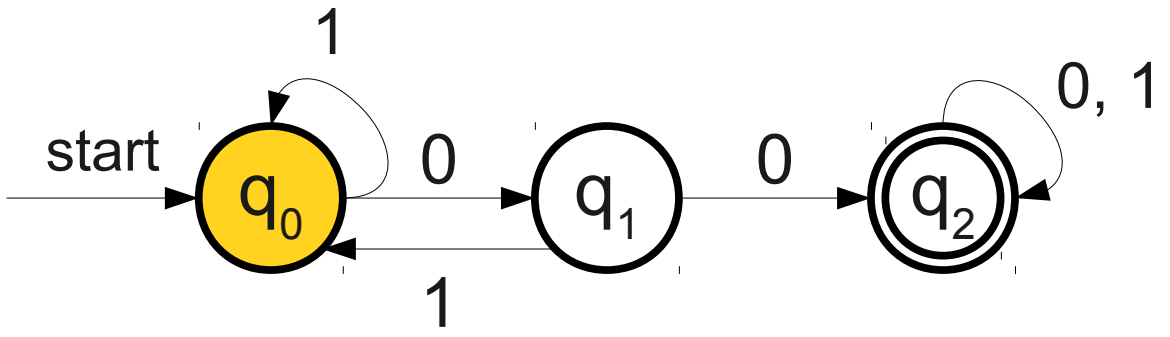


$L_1 = \{ w \mid w \text{ contains 00 as a substring} \}$

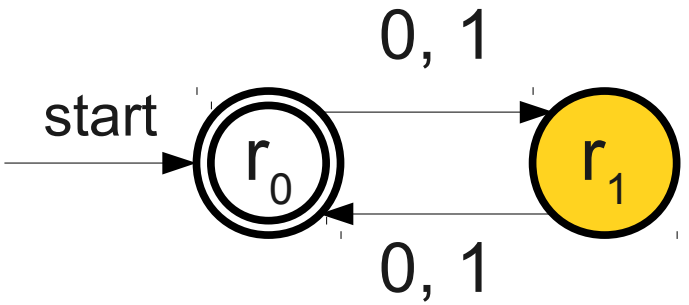


$L_2 = \{ w \mid w \text{ has even length} \}$

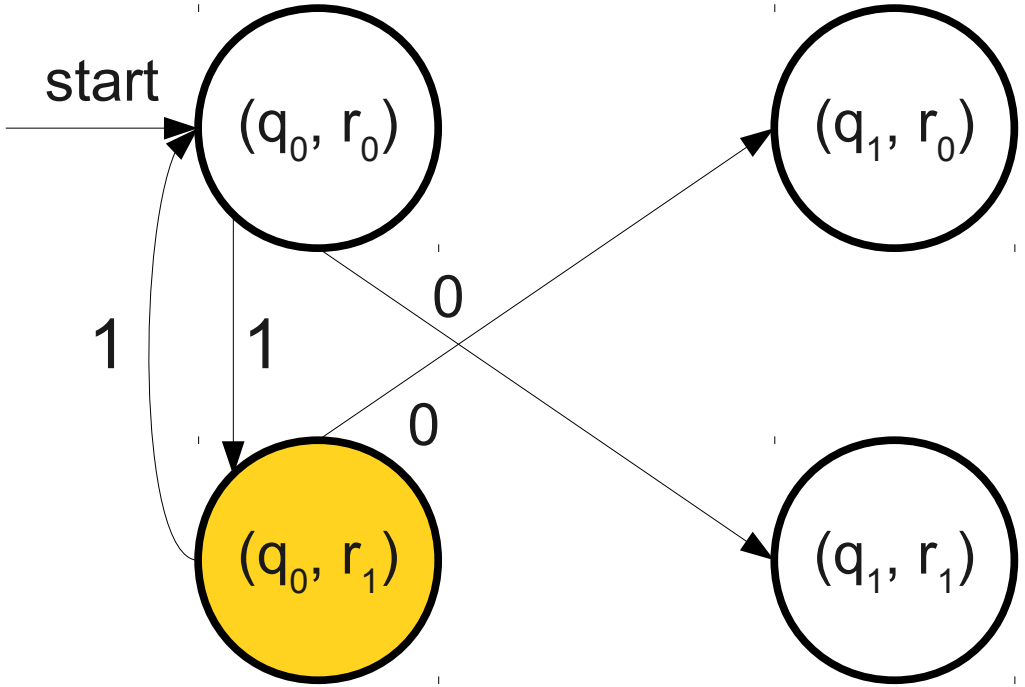


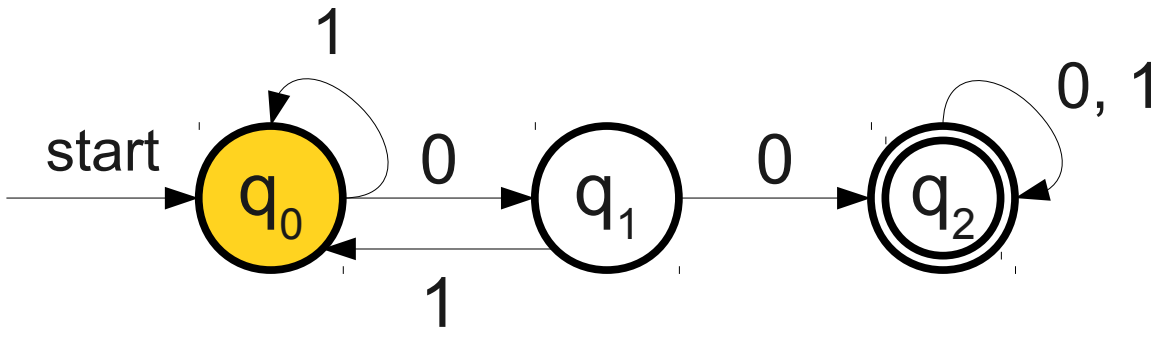


L_1 = { w | w contains 00 as a substring }

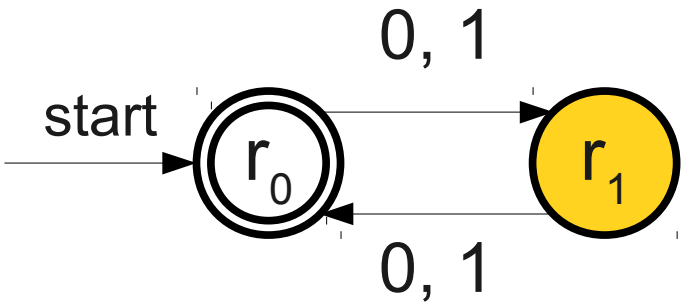


L_2 = { w | w has even length }

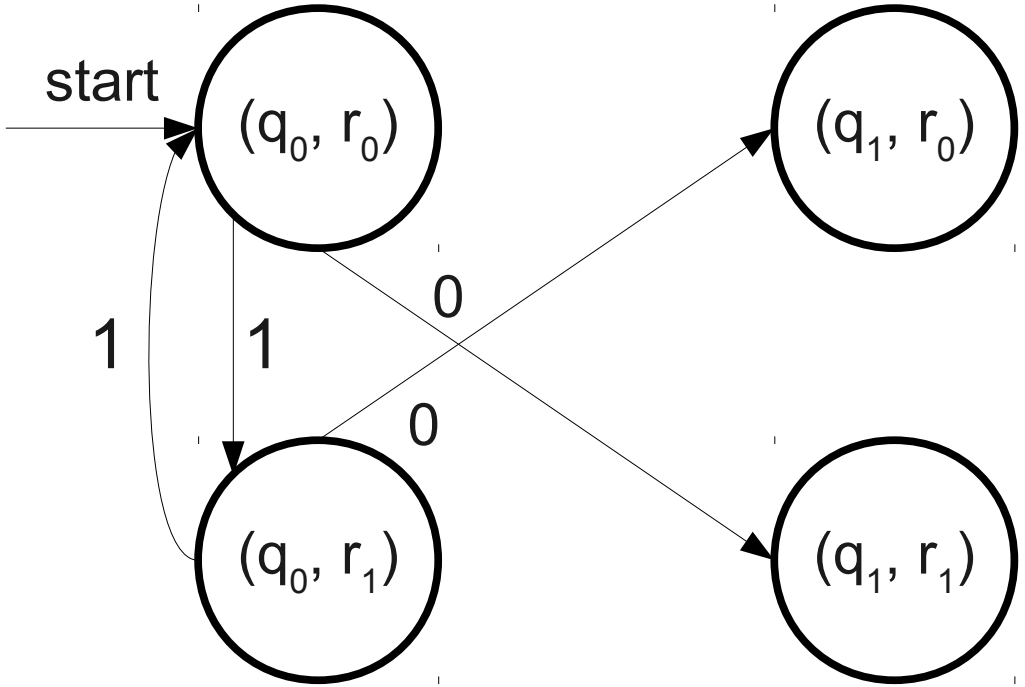


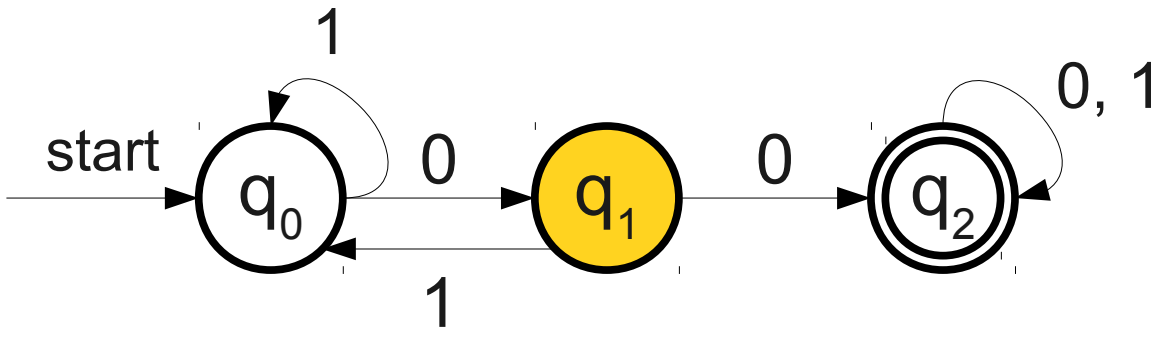


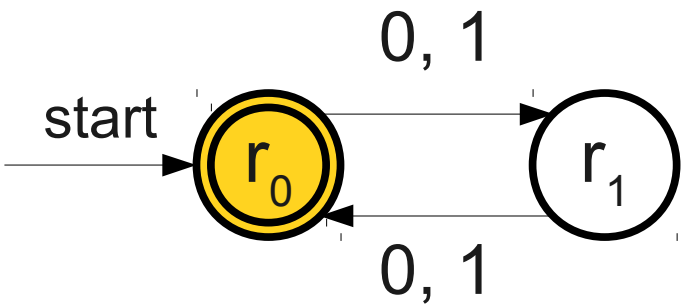
L_1 = { w | w contains 00 as a substring }



L_2 = { w | w has even length }

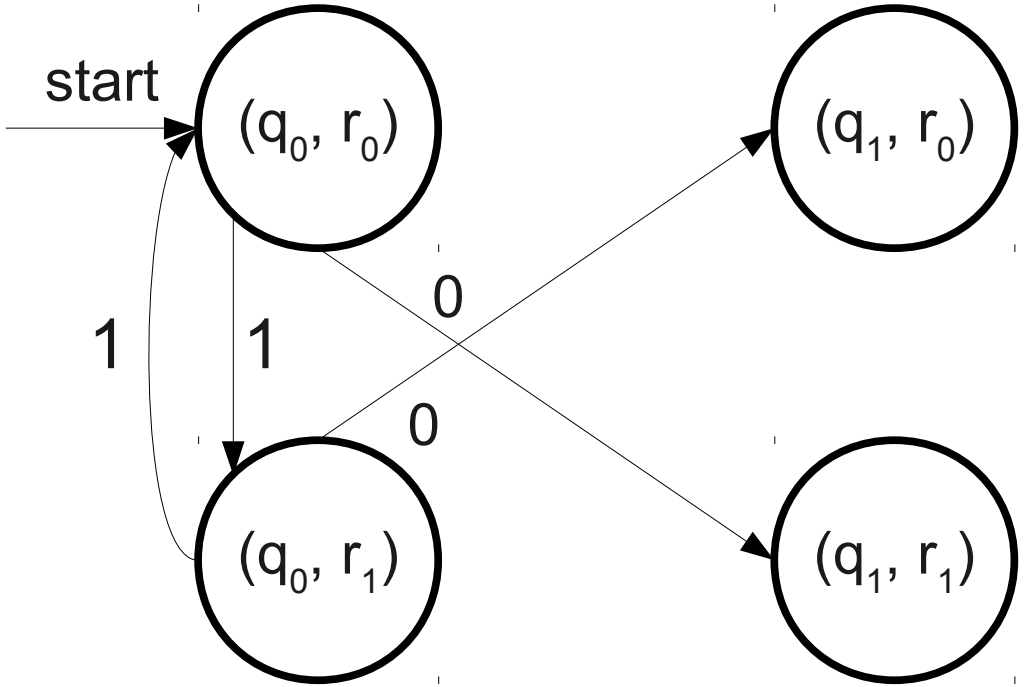


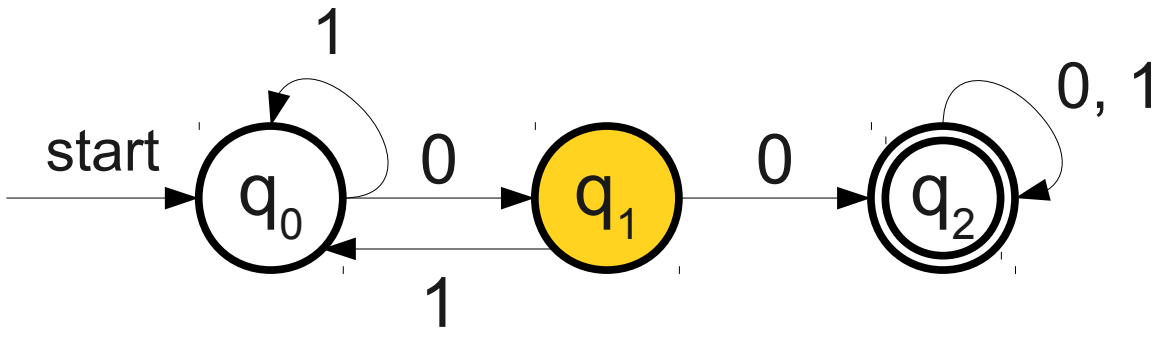




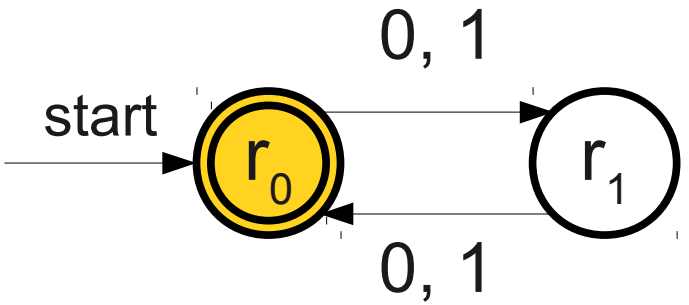
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

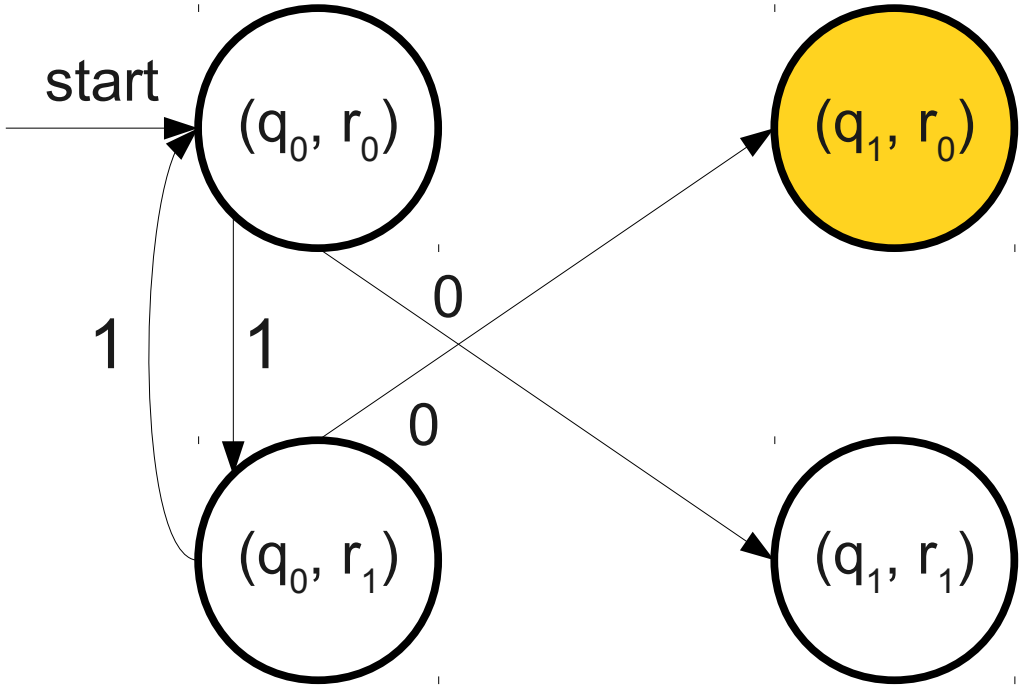


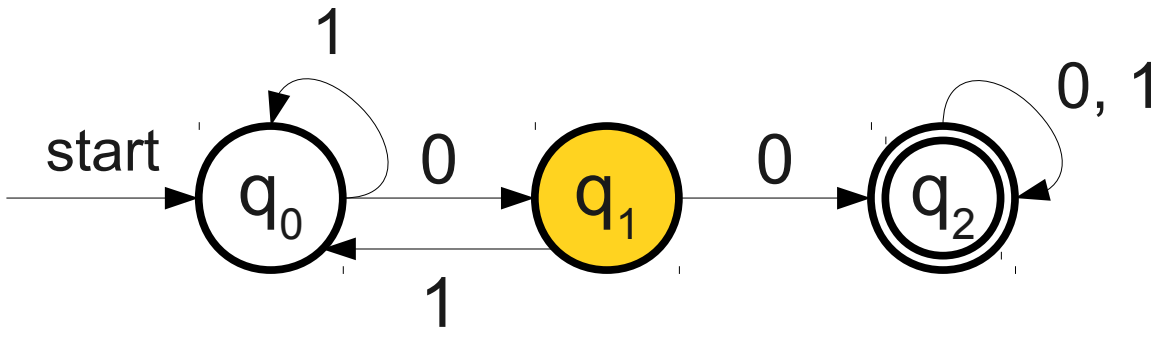


L_1 = { w | w contains 00 as a substring }

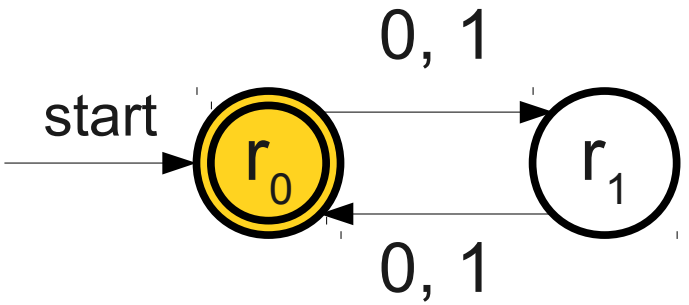


L_2 = { w | w has even length }

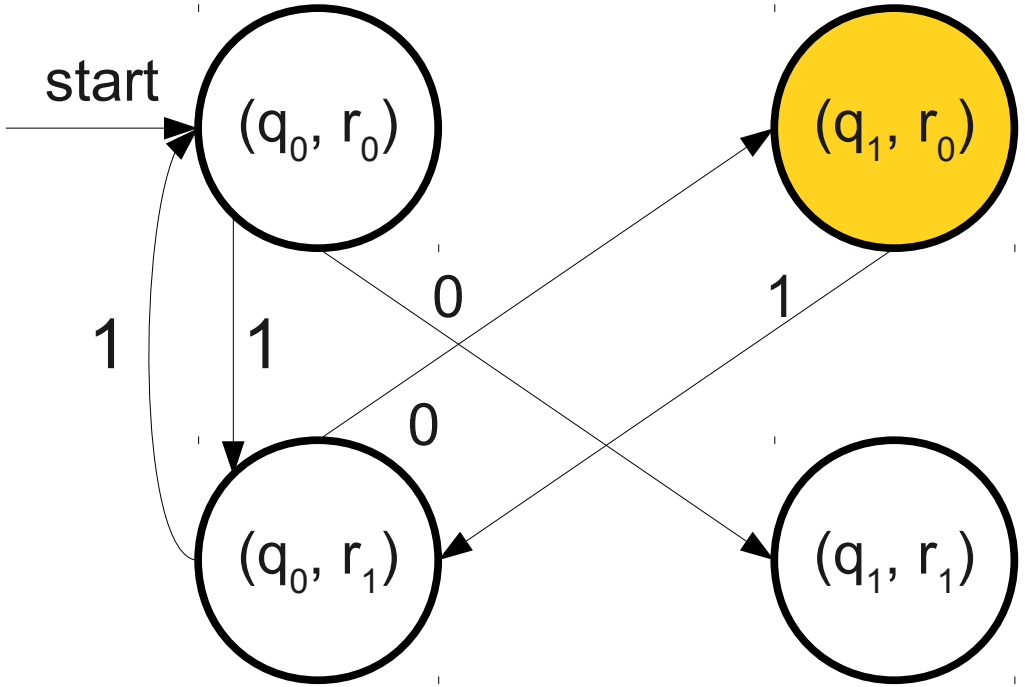


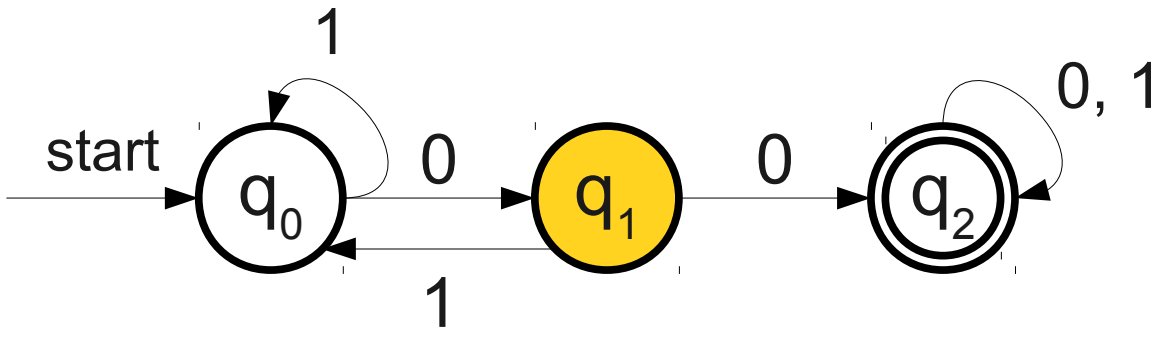


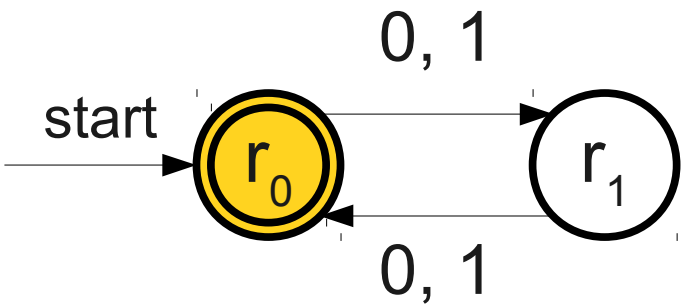
L_1 = { w | w contains 00 as a substring }



L_2 = { w | w has even length }

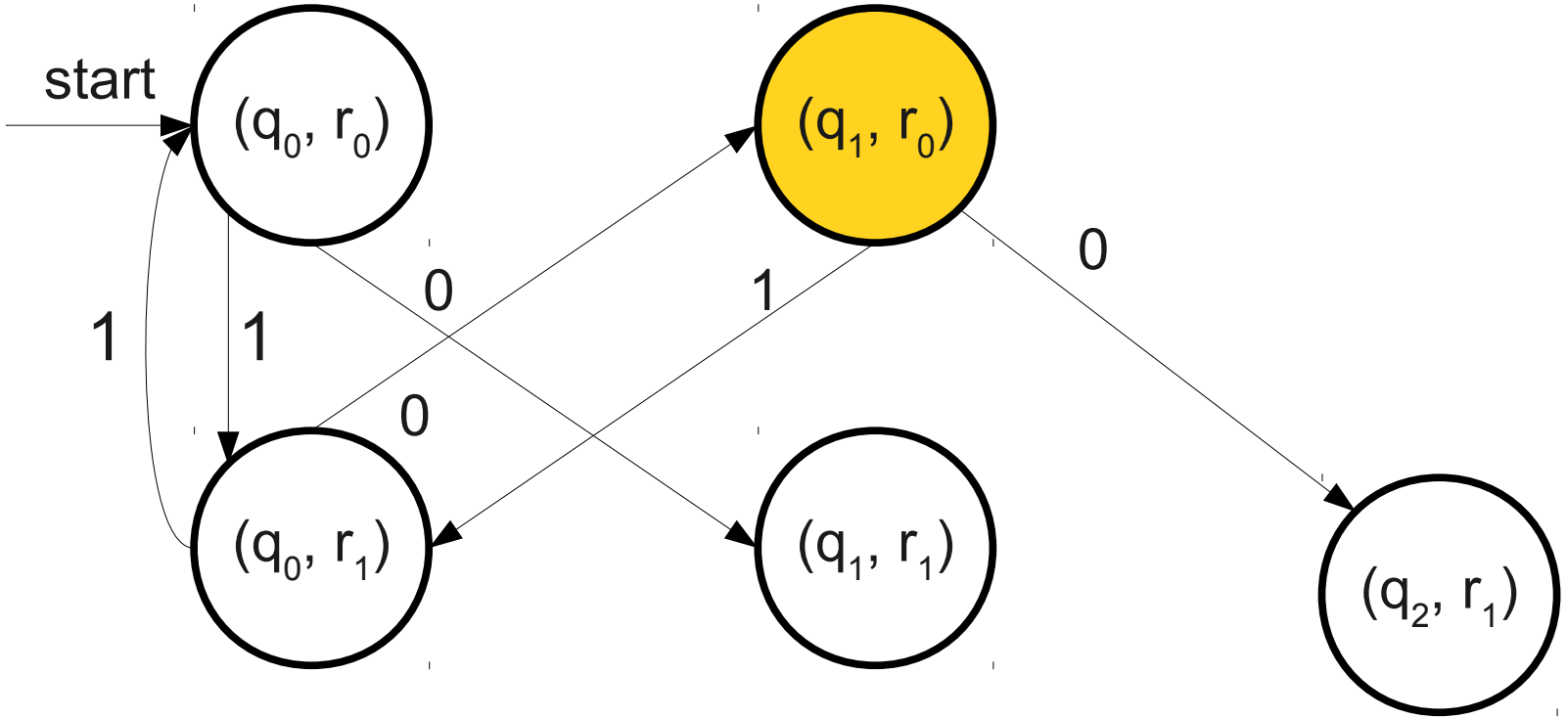


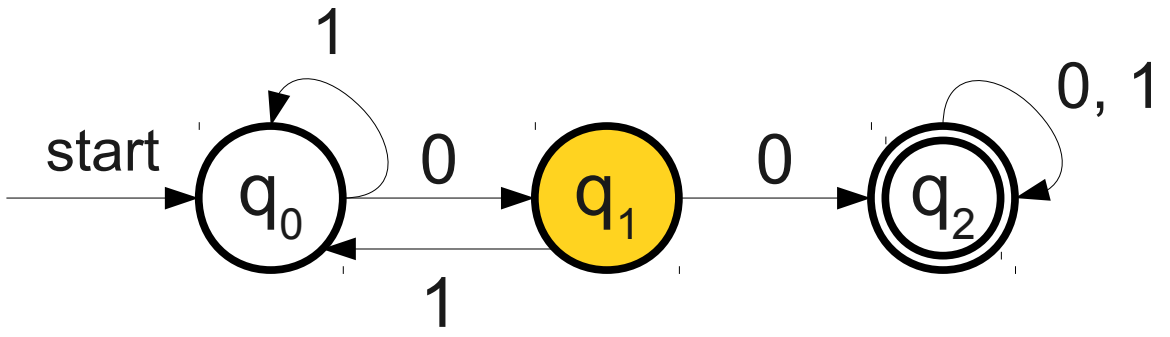


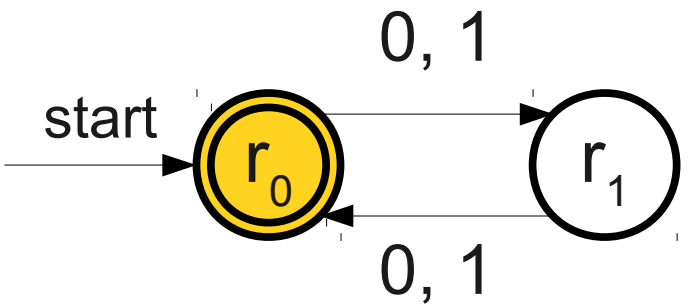


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

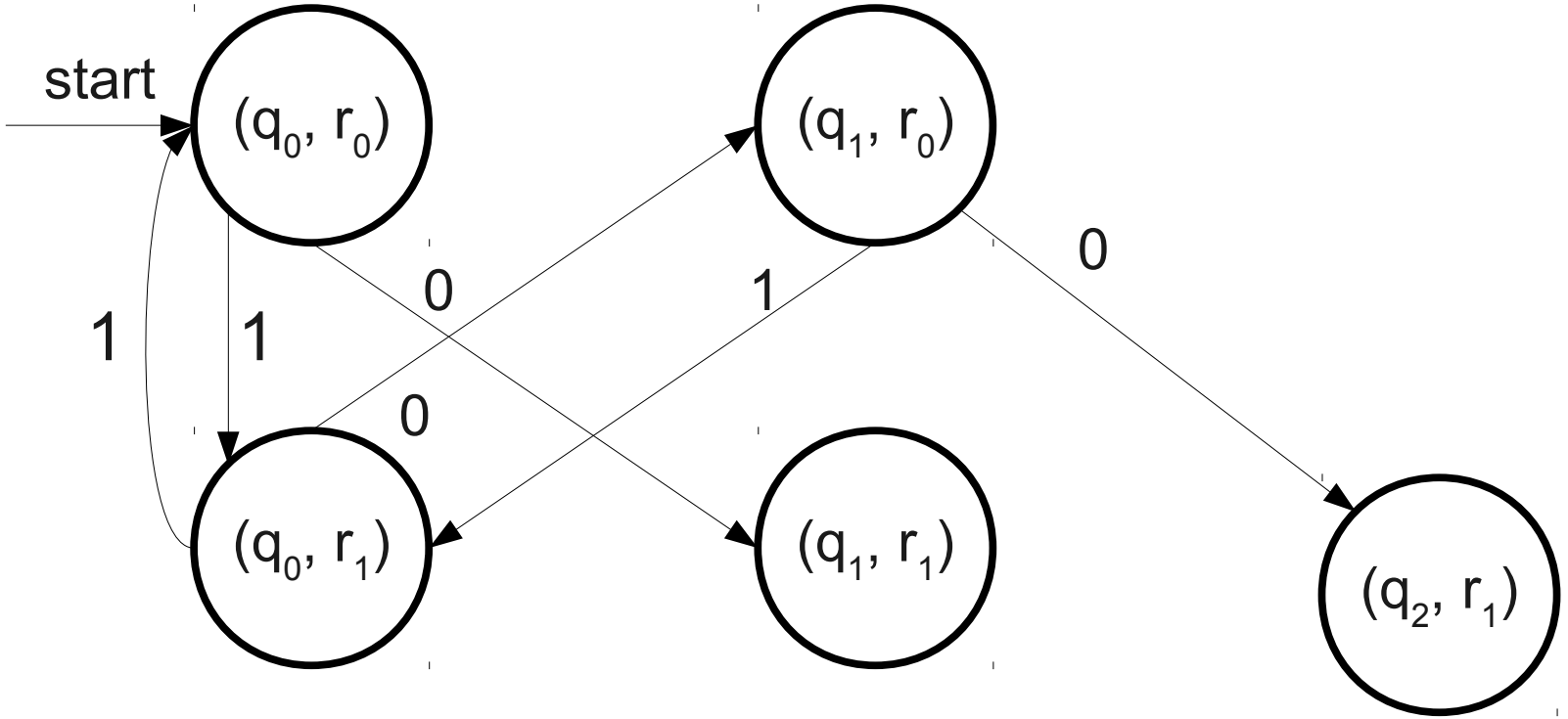


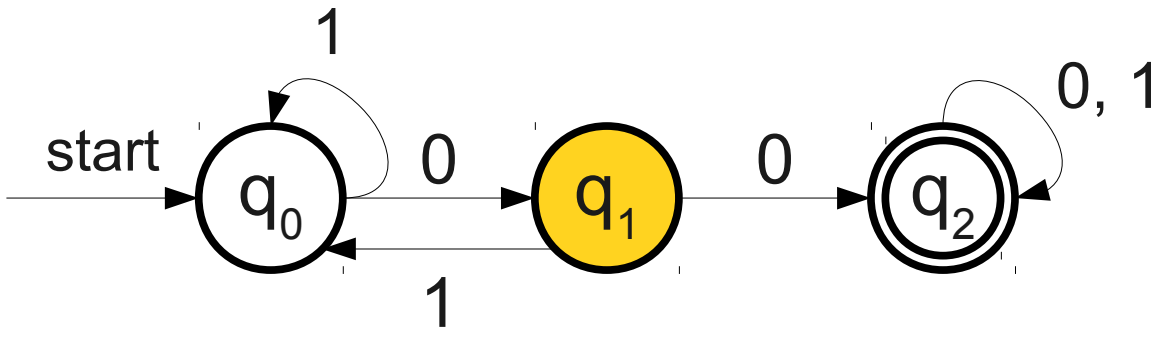


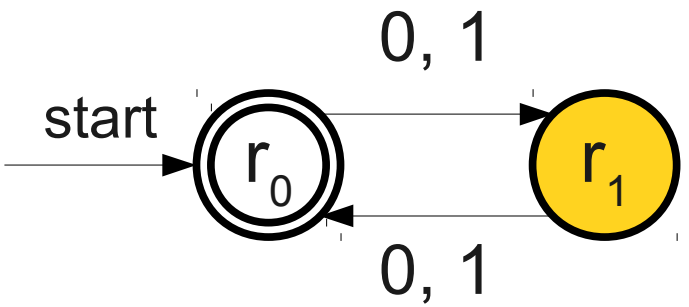


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

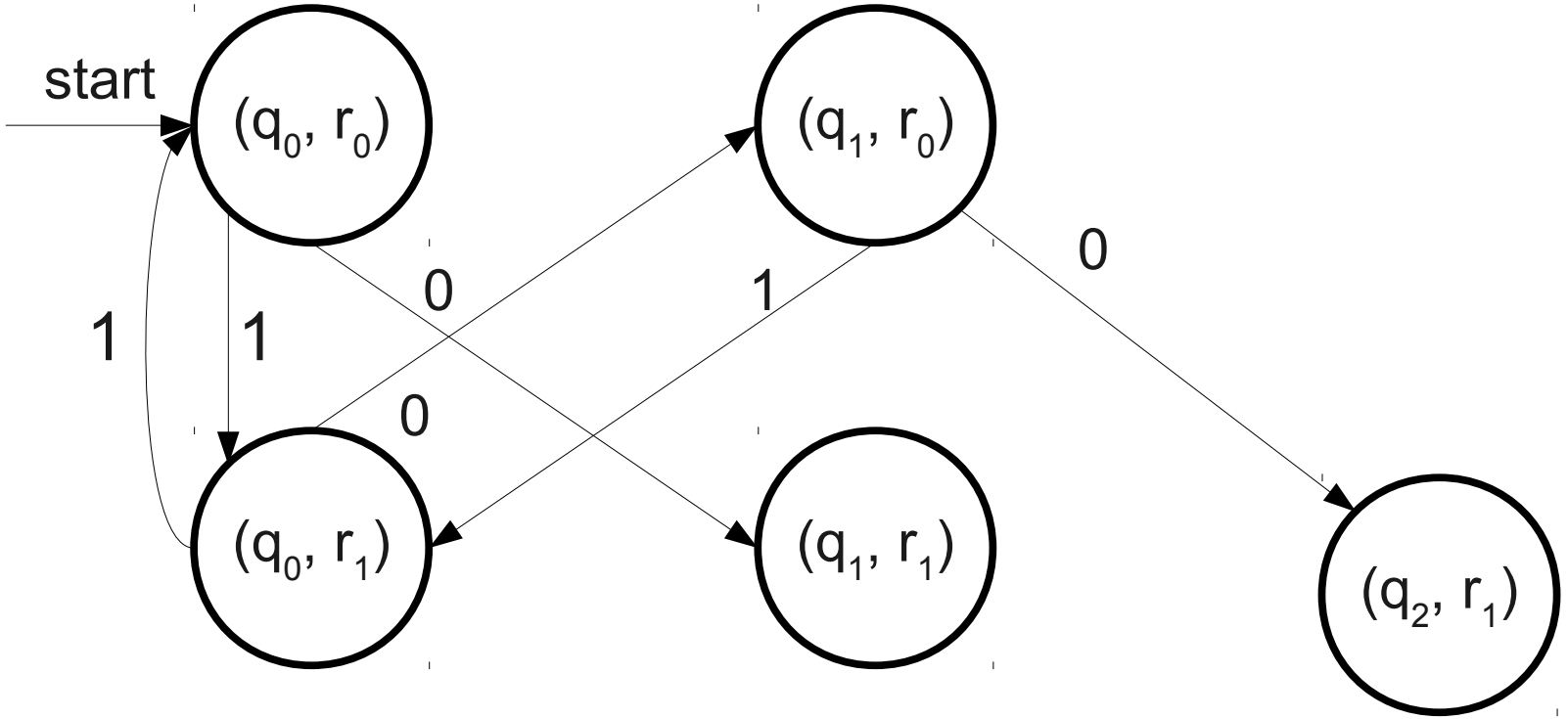


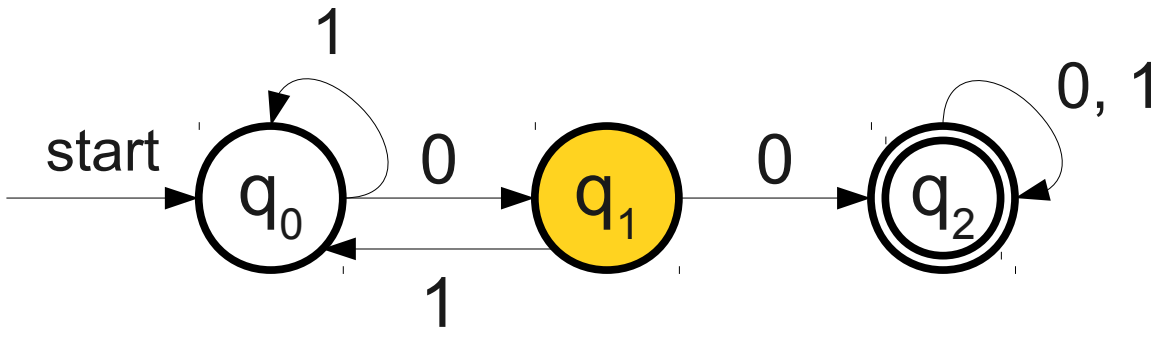


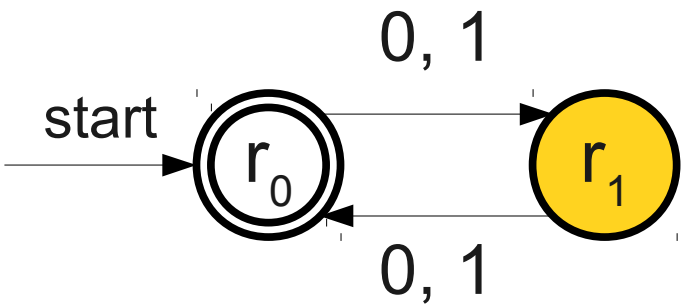


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

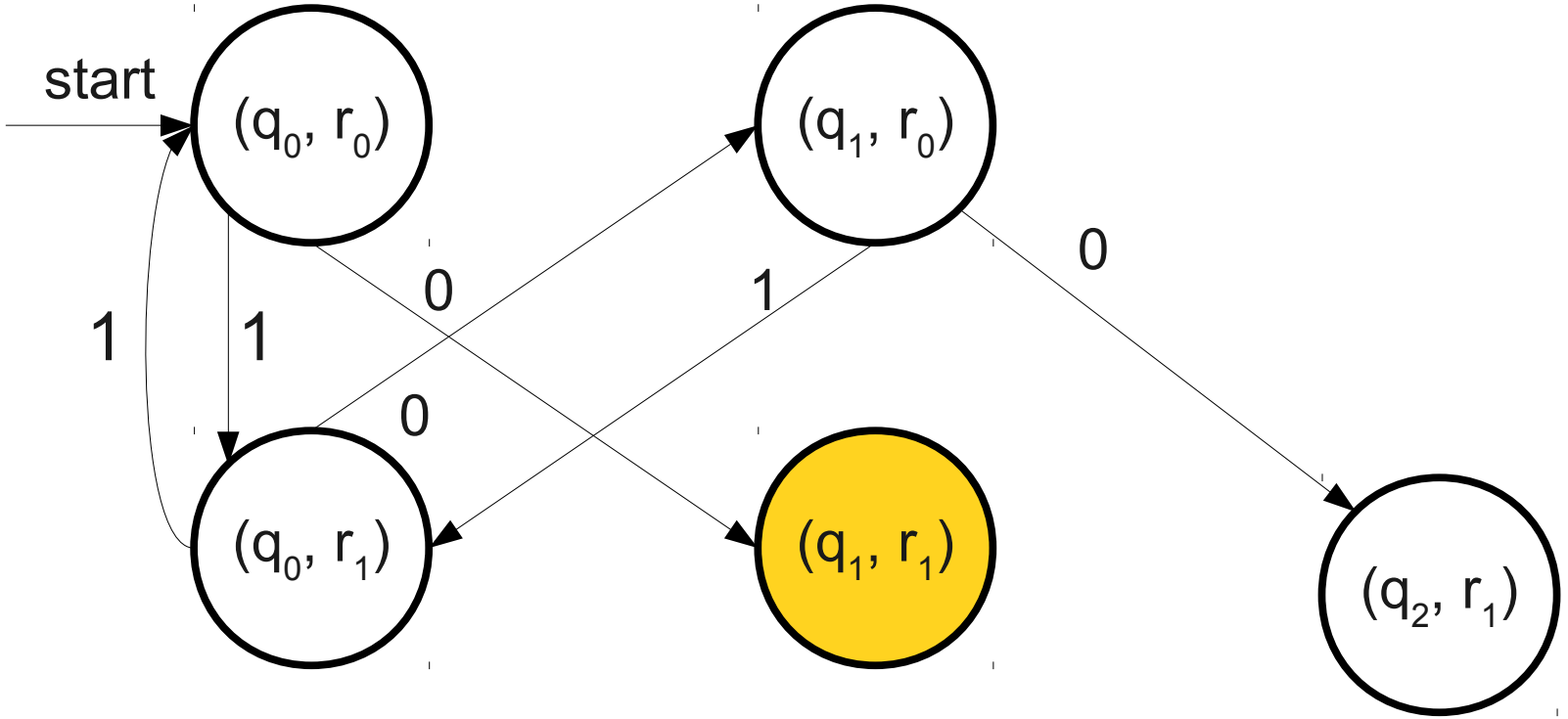


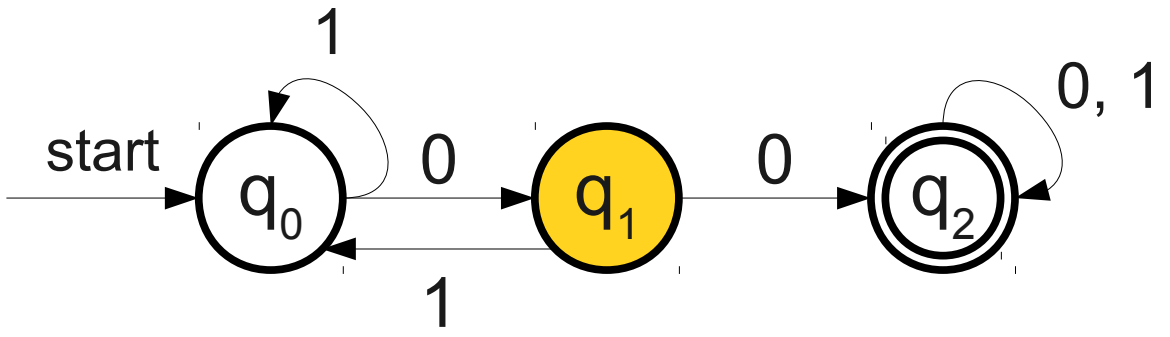


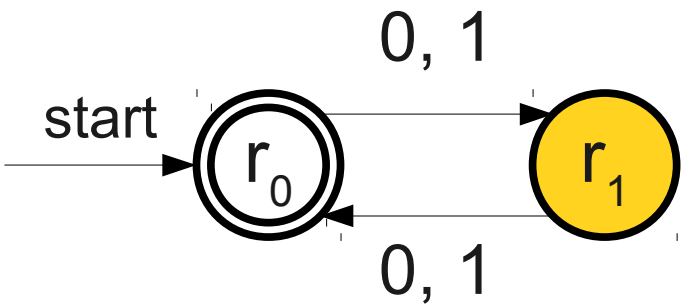


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

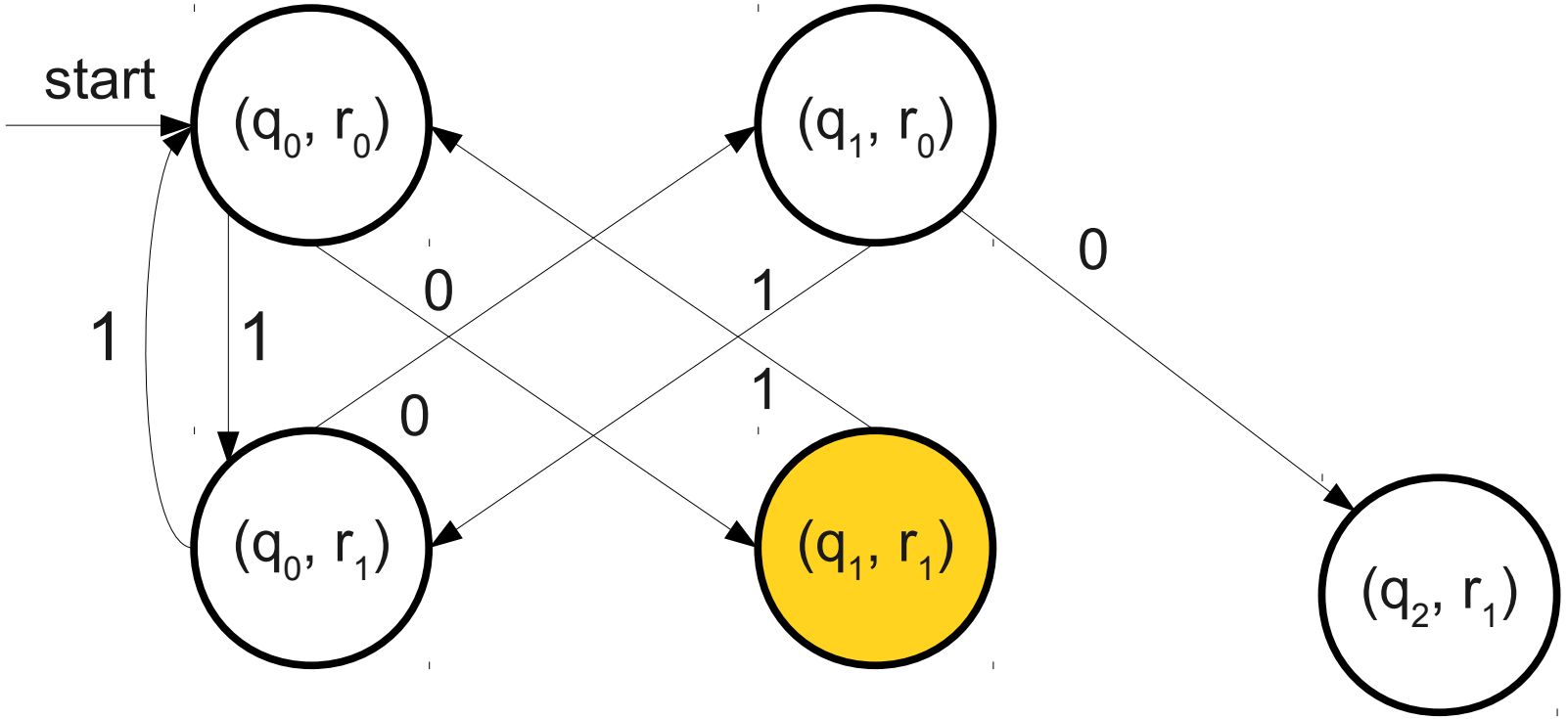


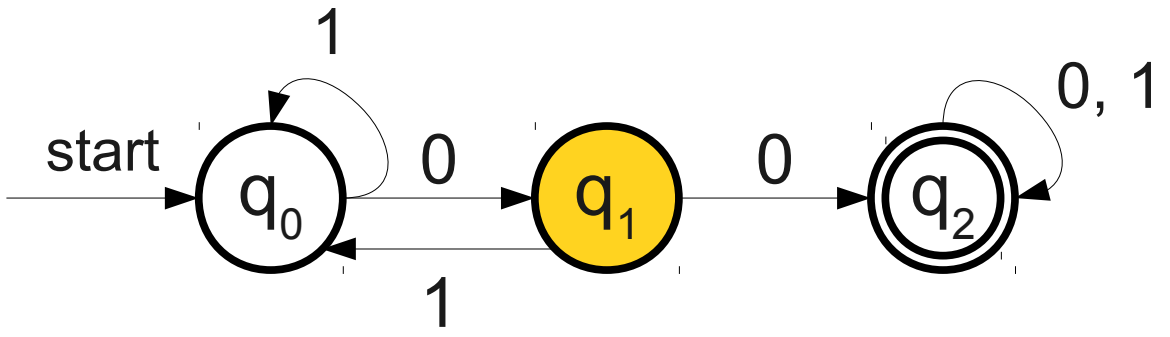


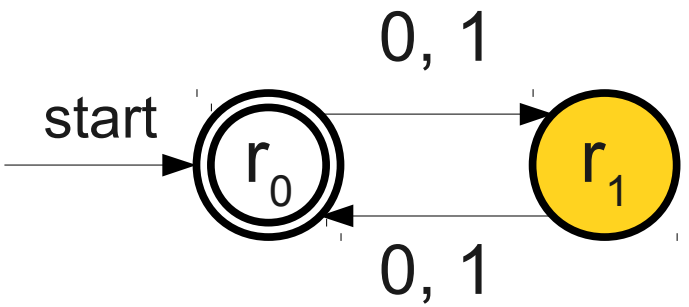


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

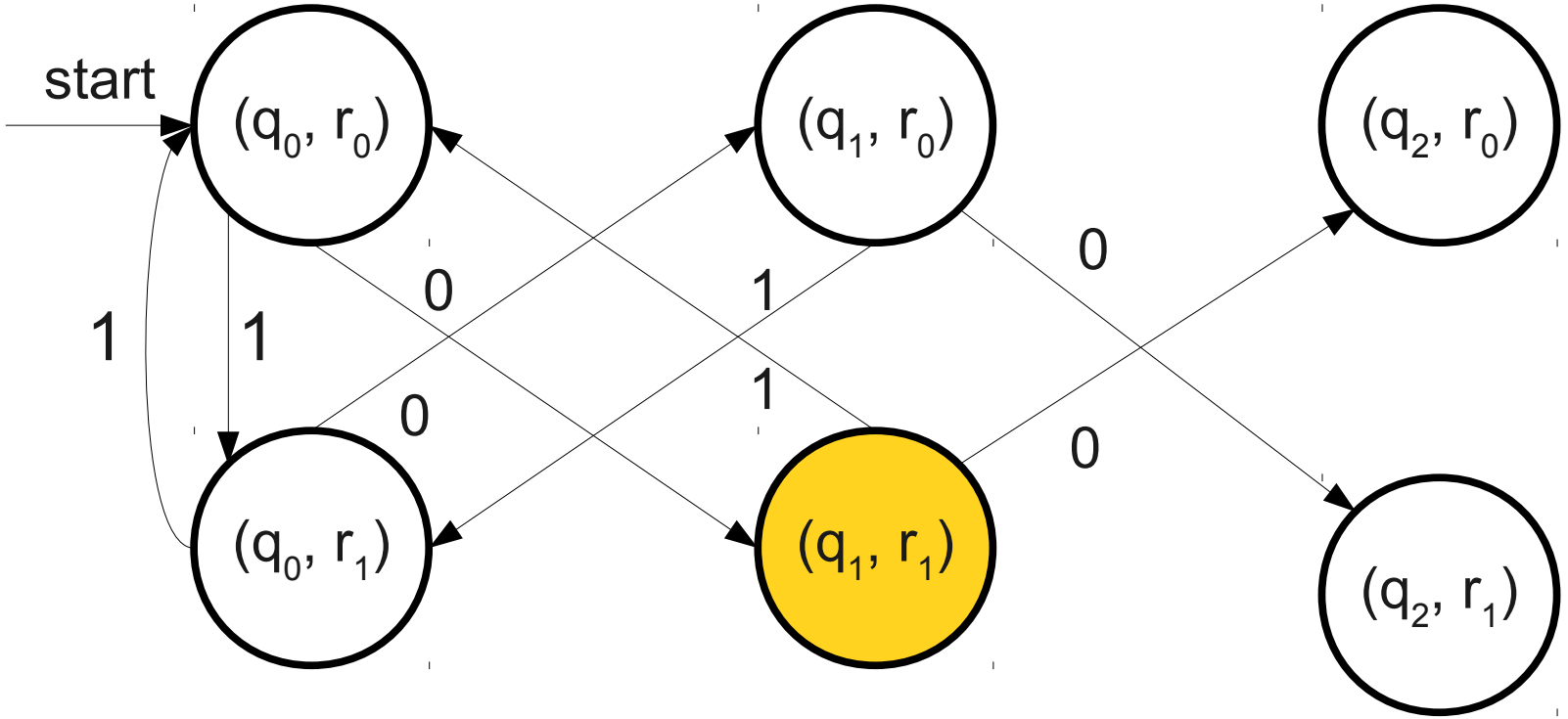


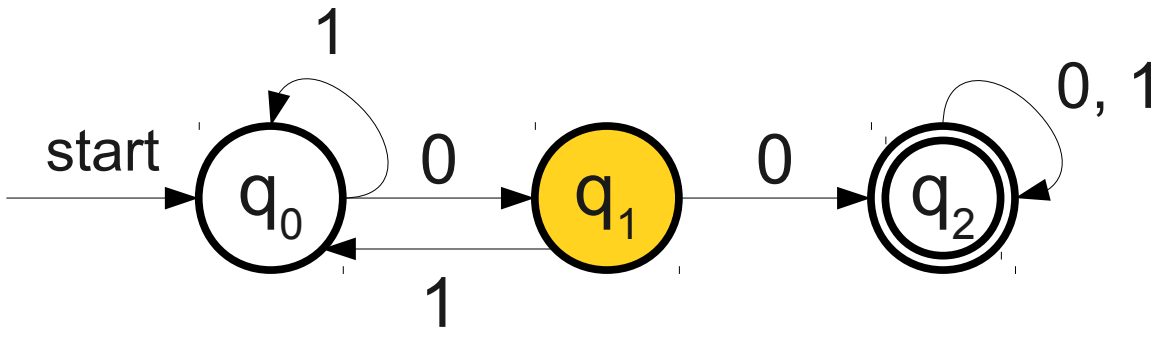


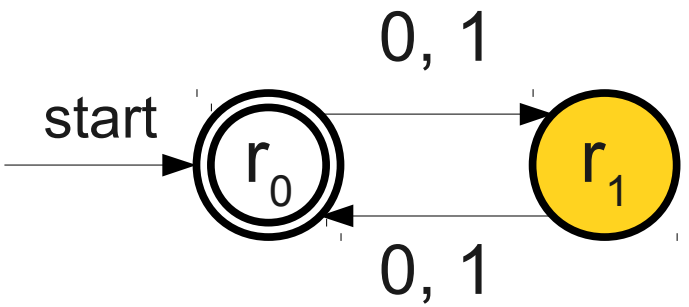


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

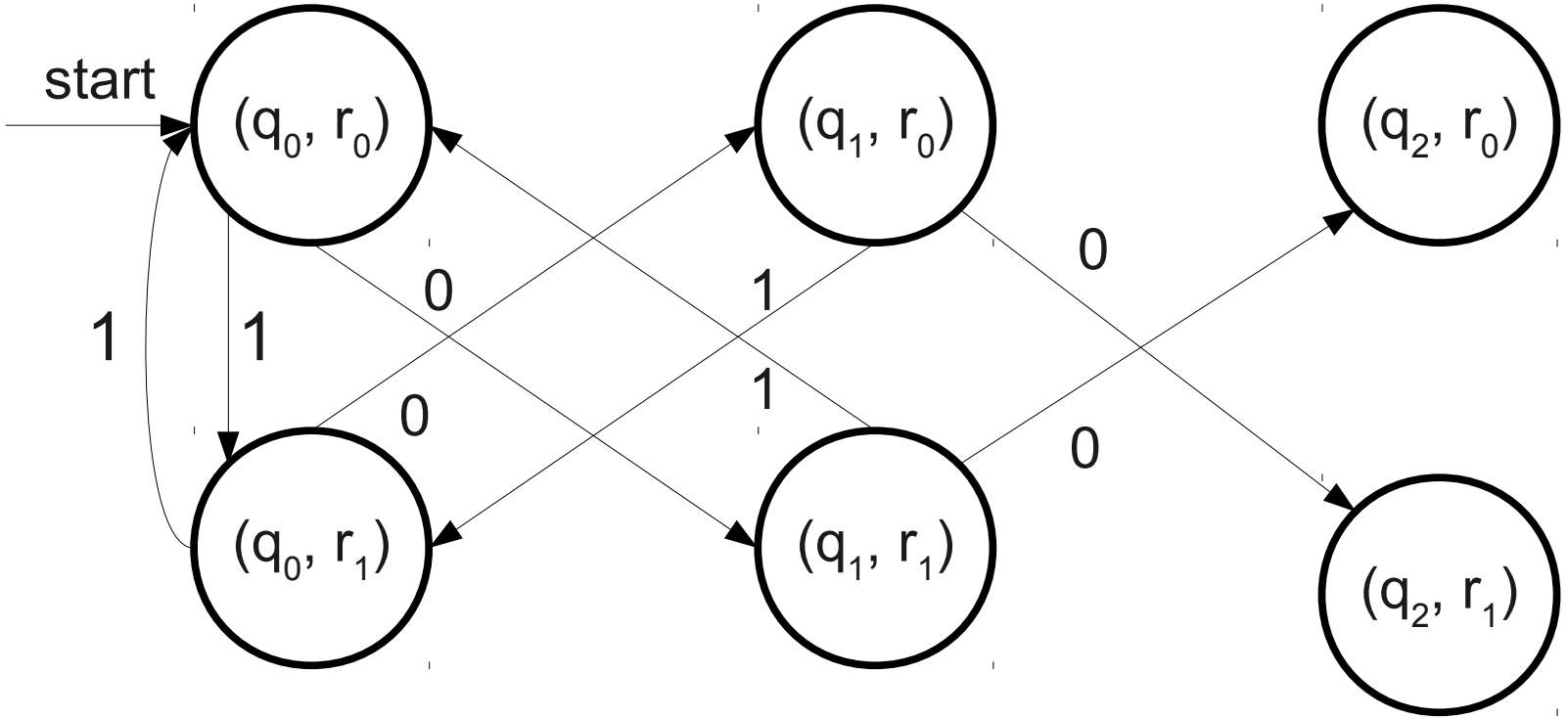


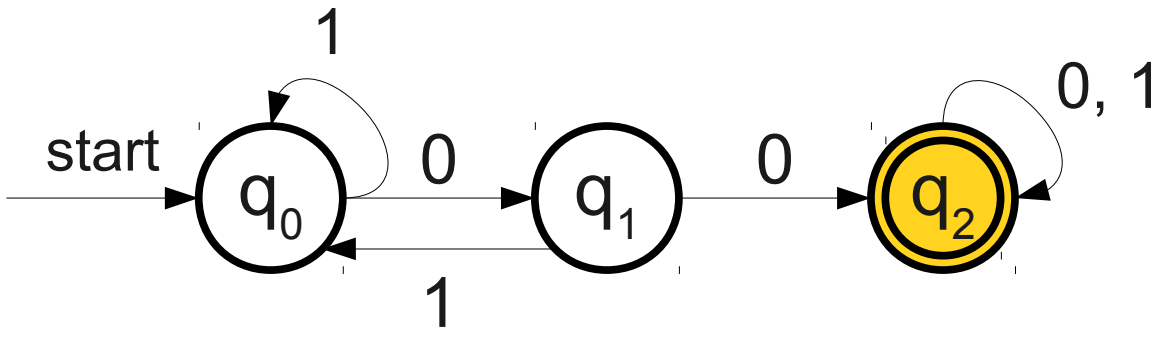


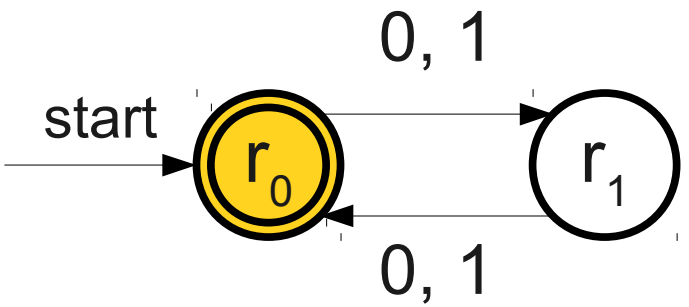


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

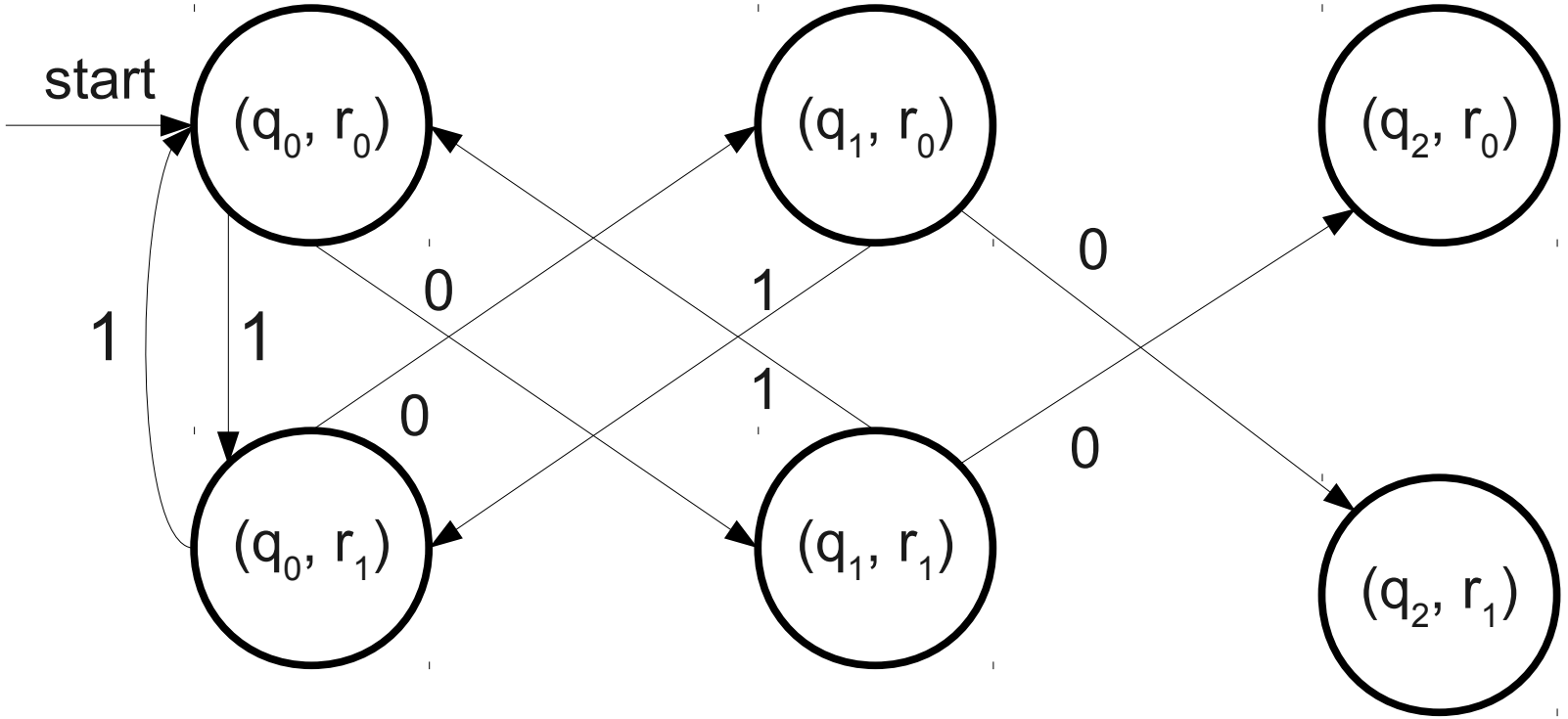


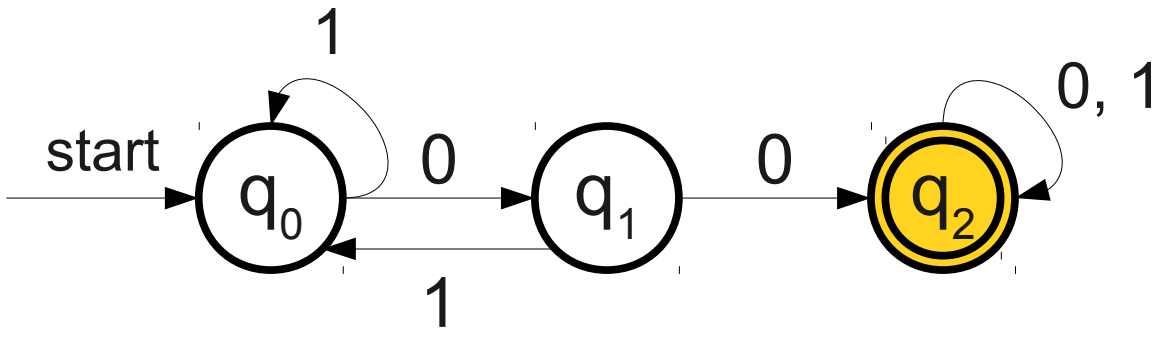


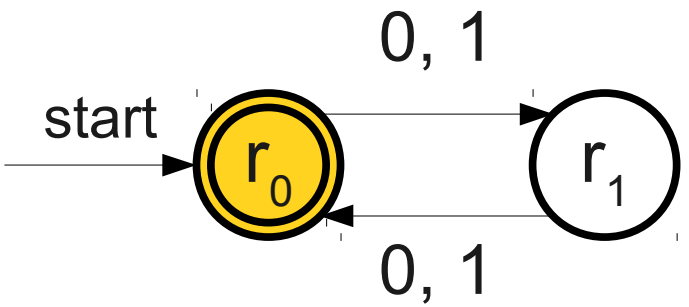


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

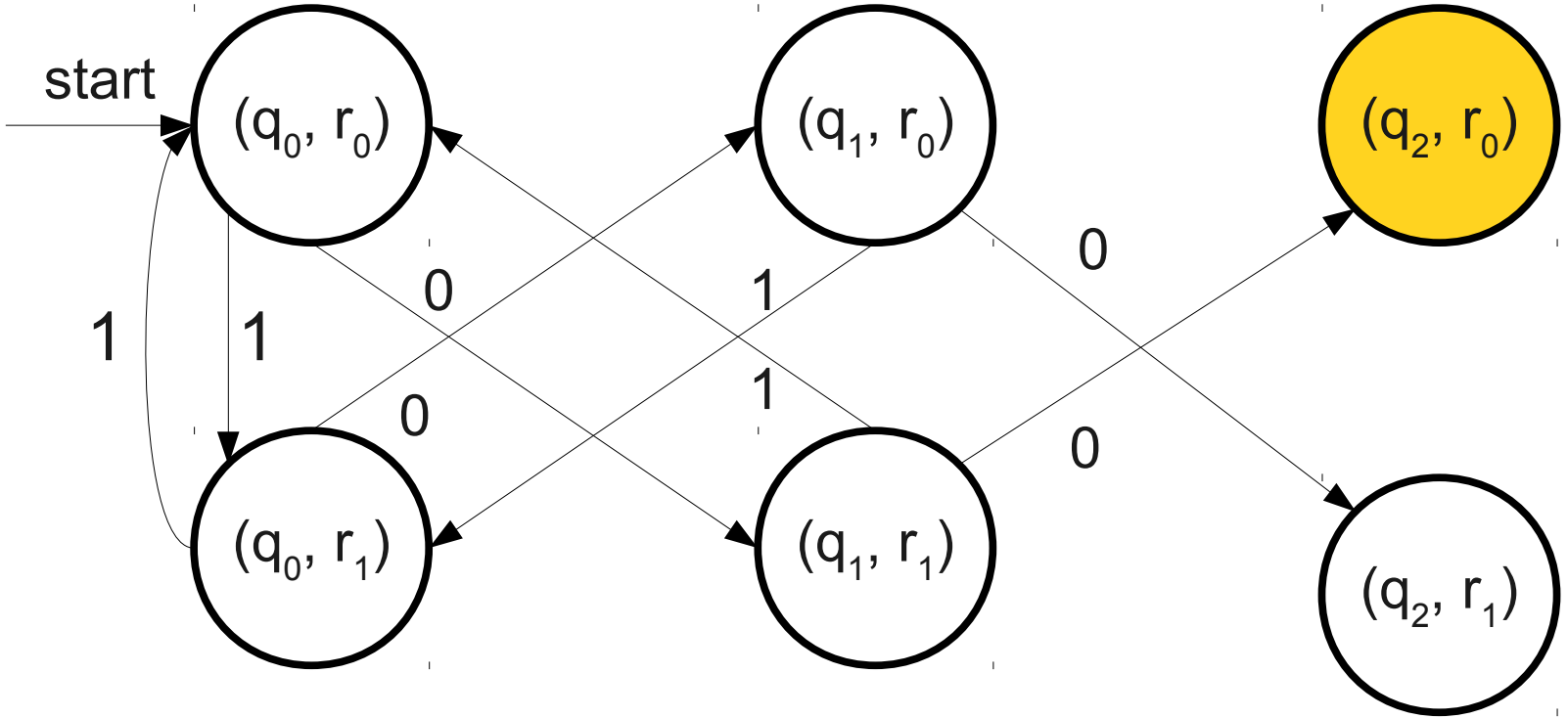


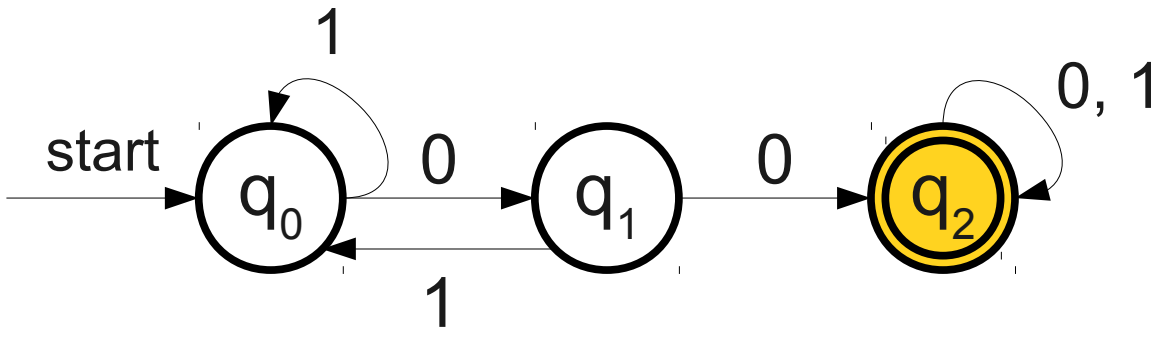


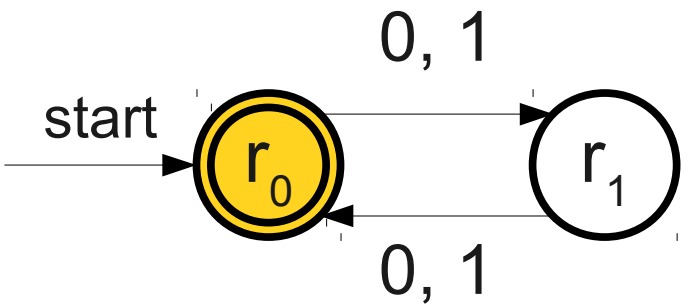


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

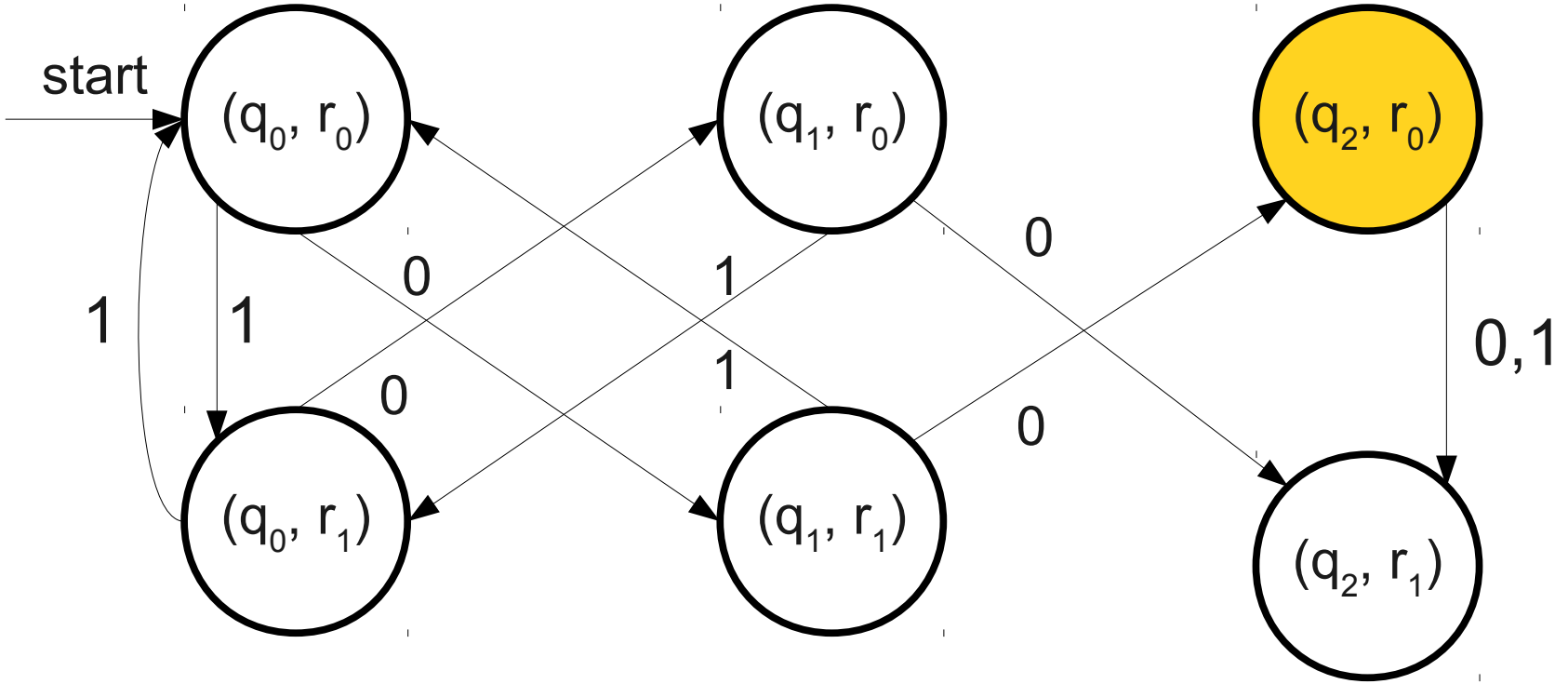


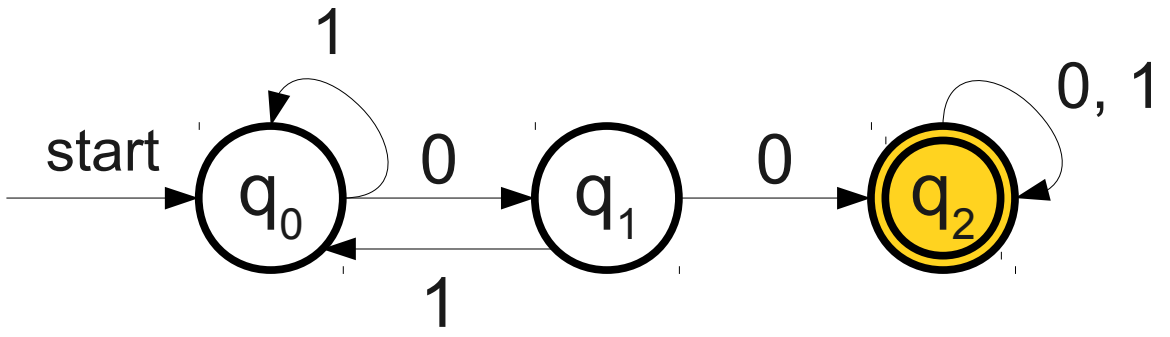


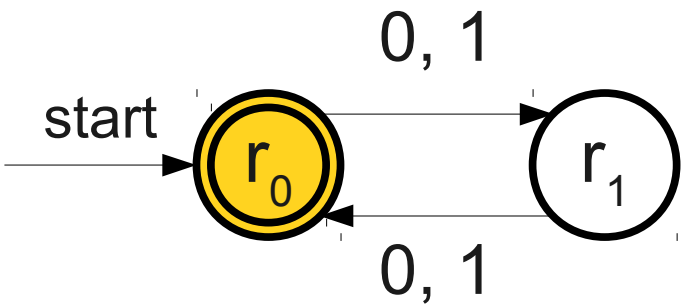


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

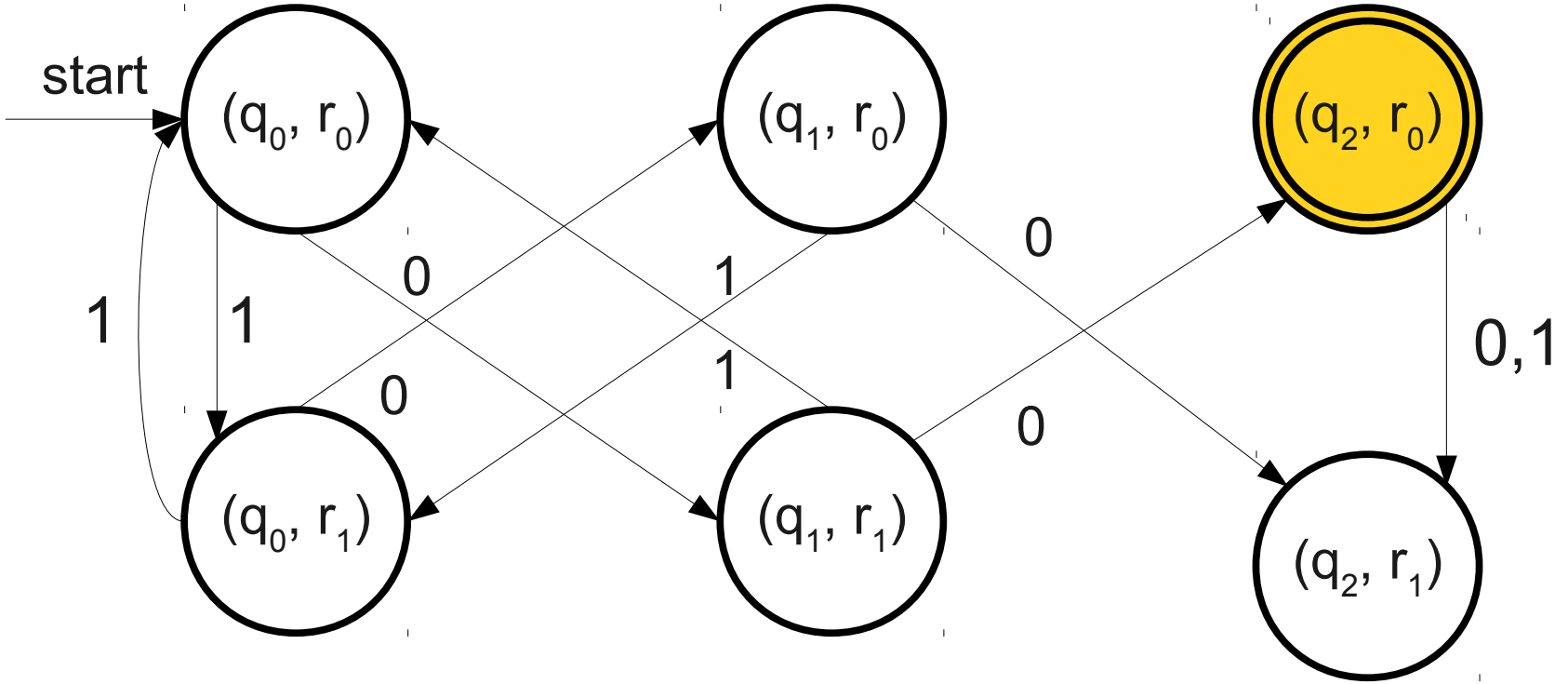


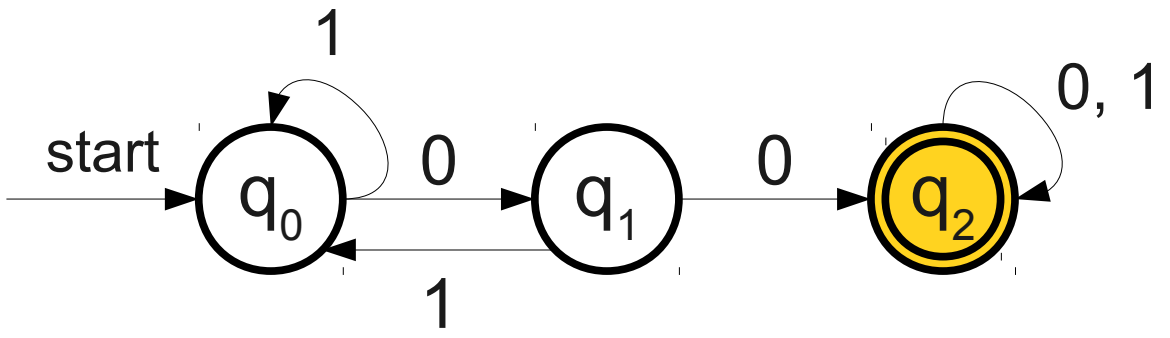


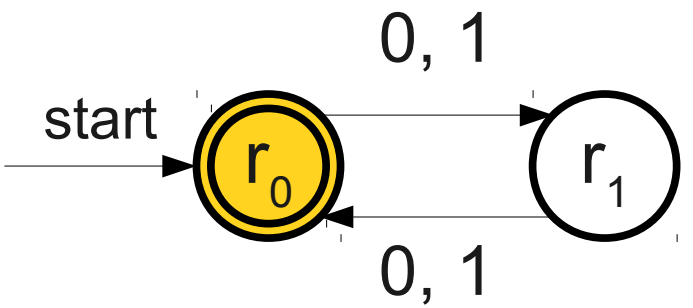


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

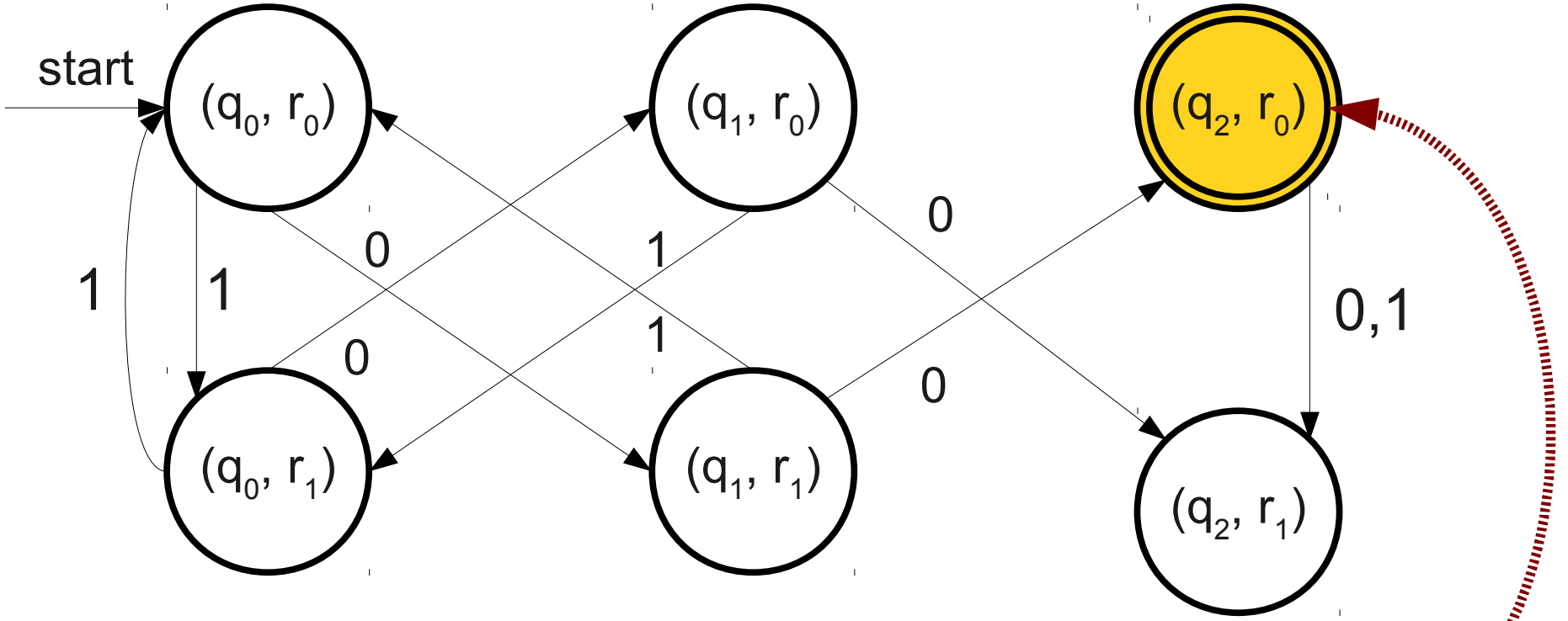




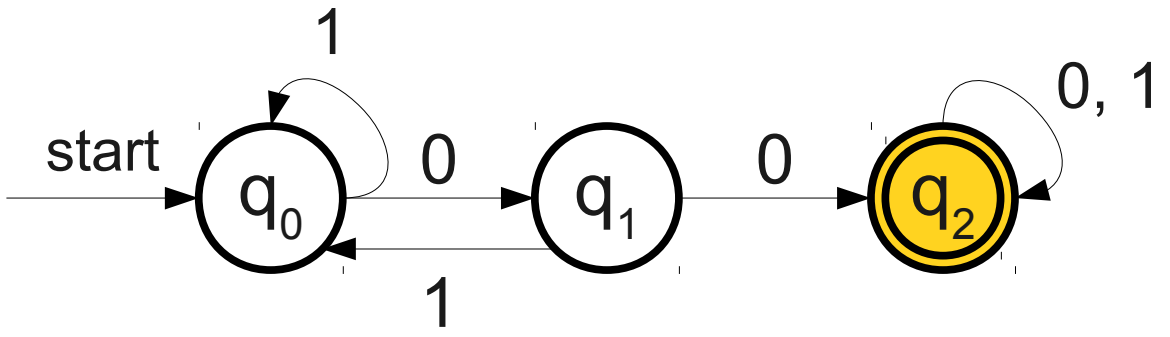


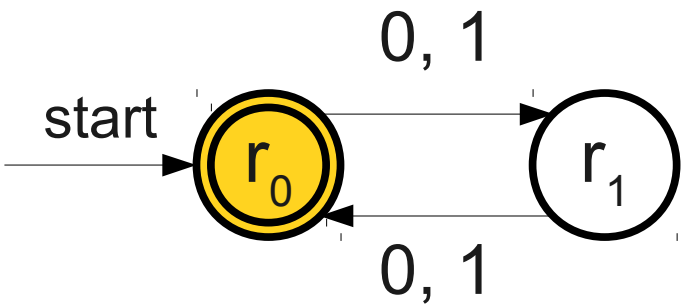
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



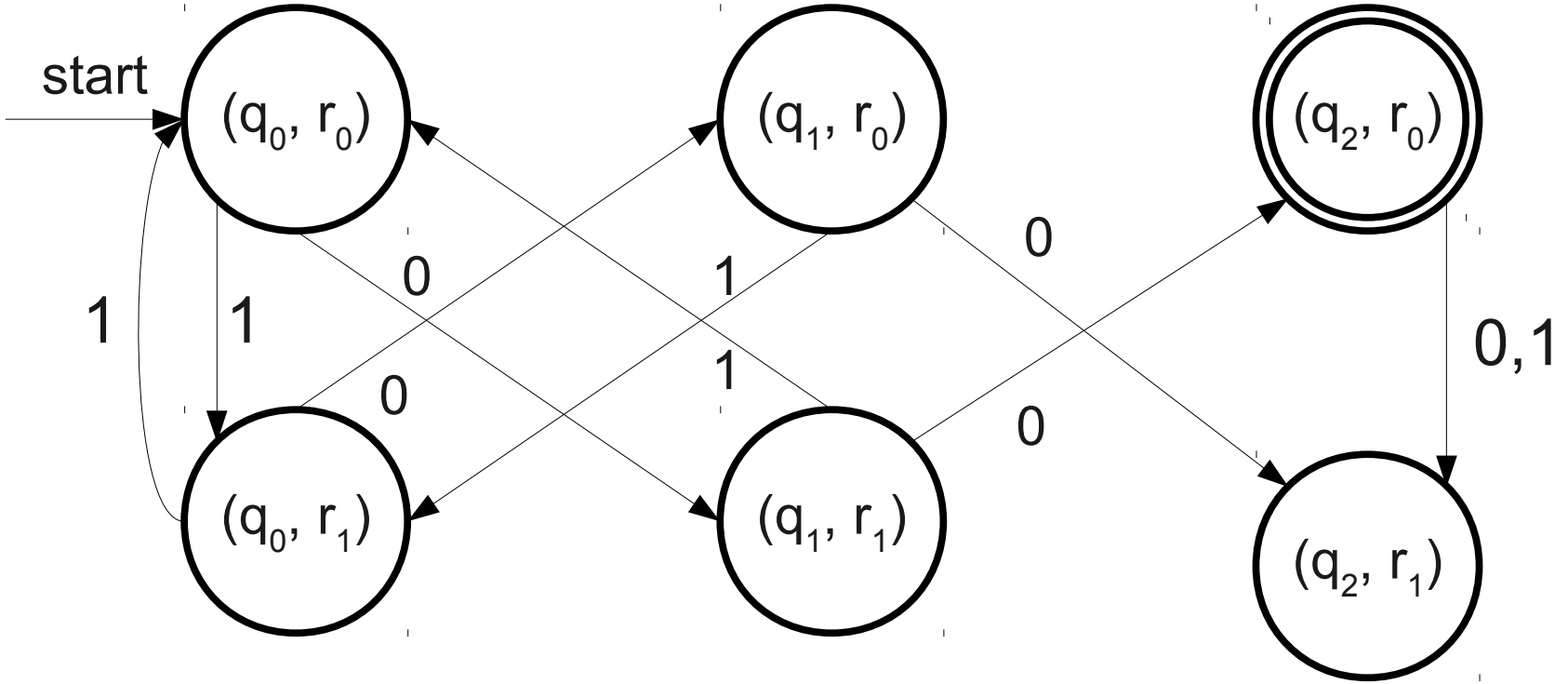
This state is accepting because both q_2 and r_0 are.

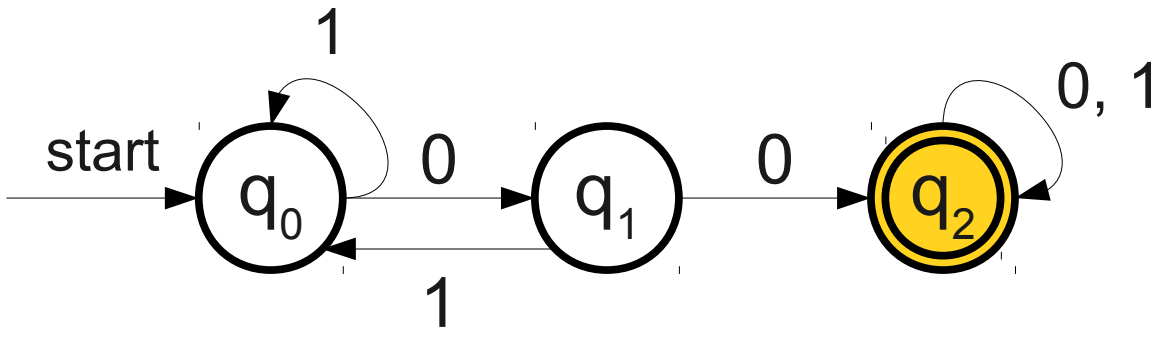


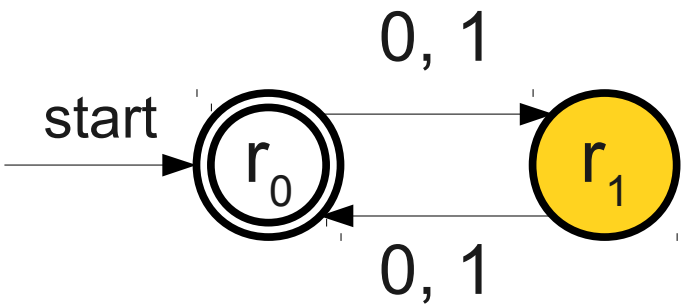


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

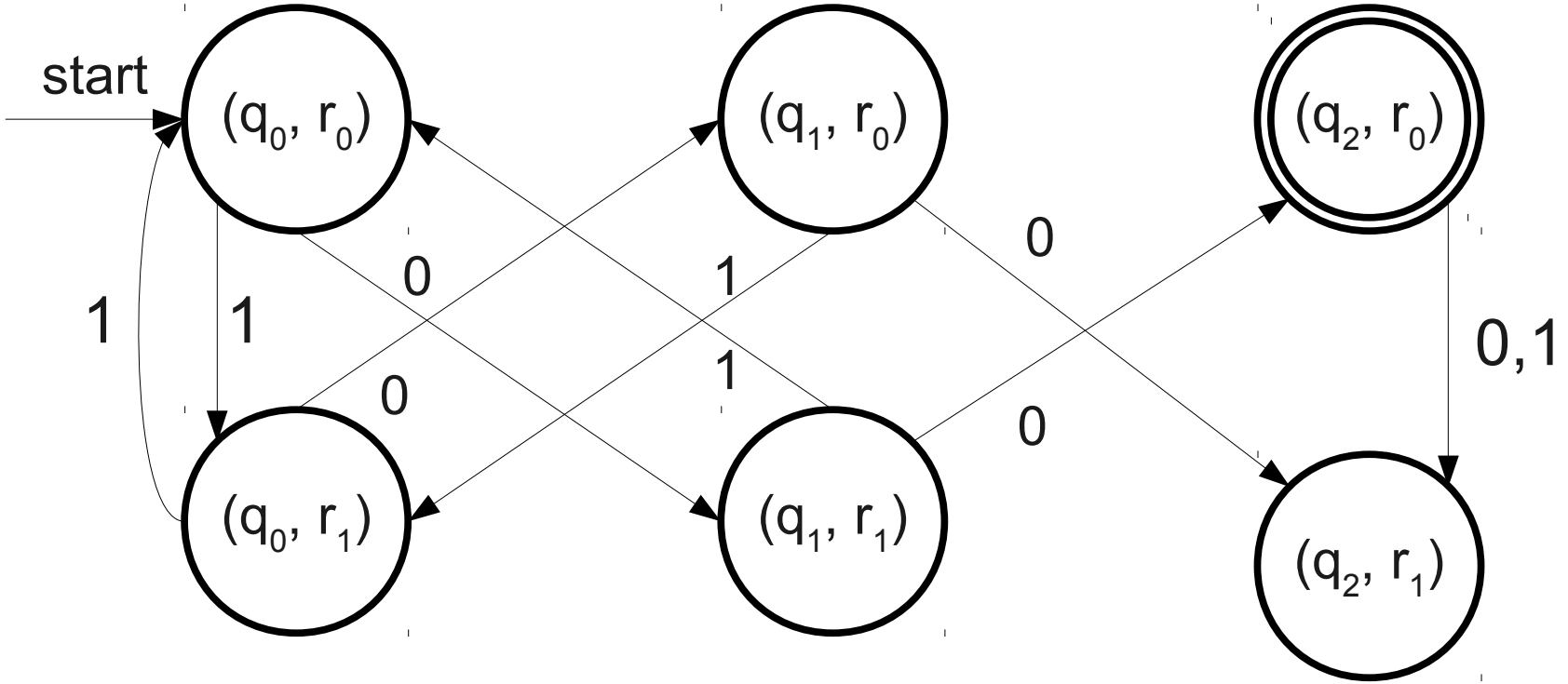


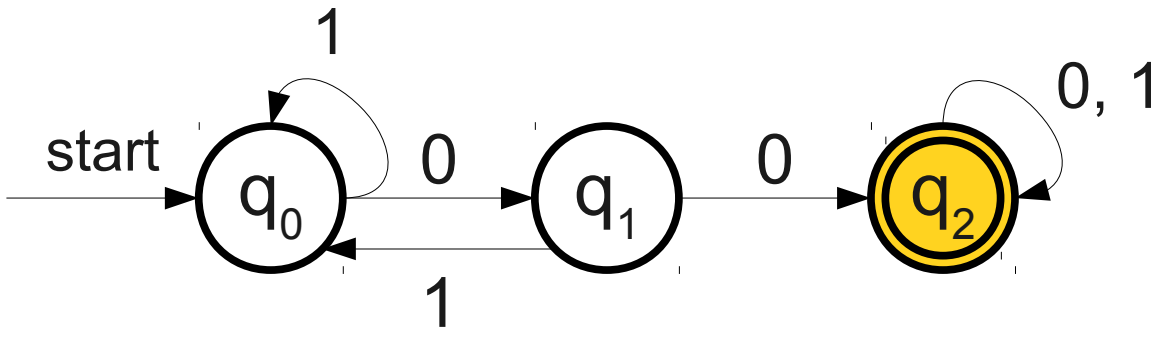


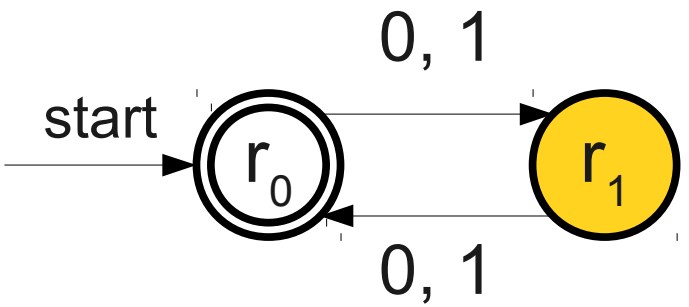


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

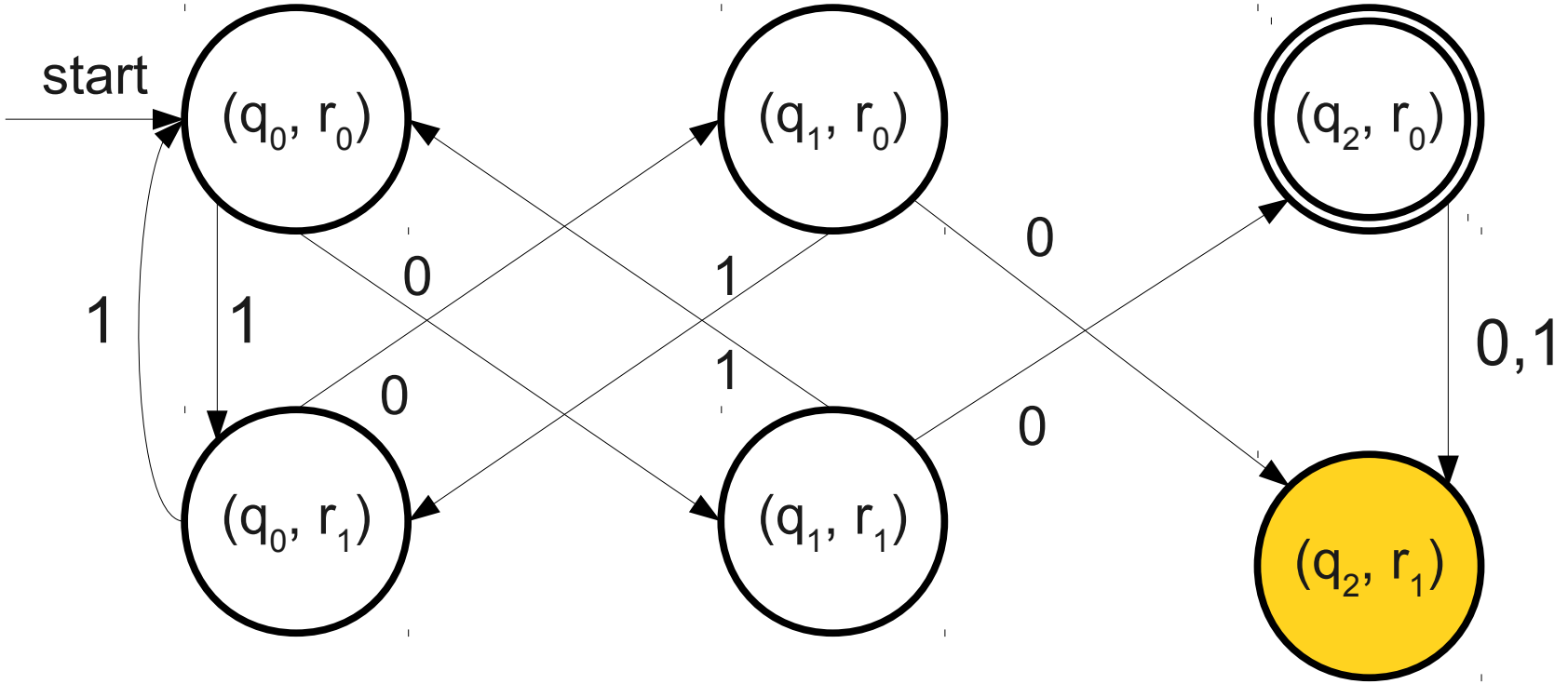


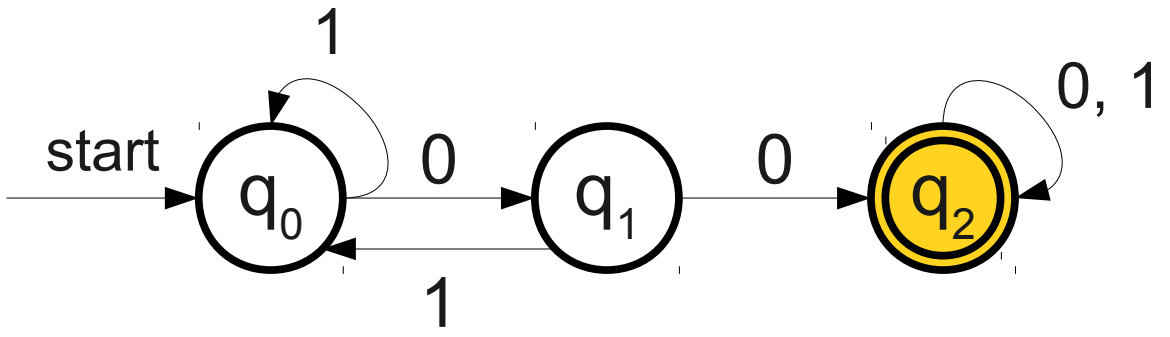


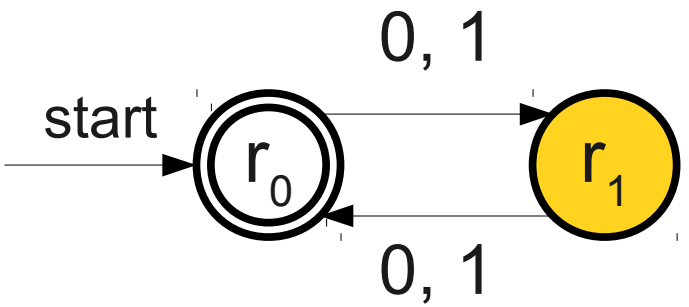


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

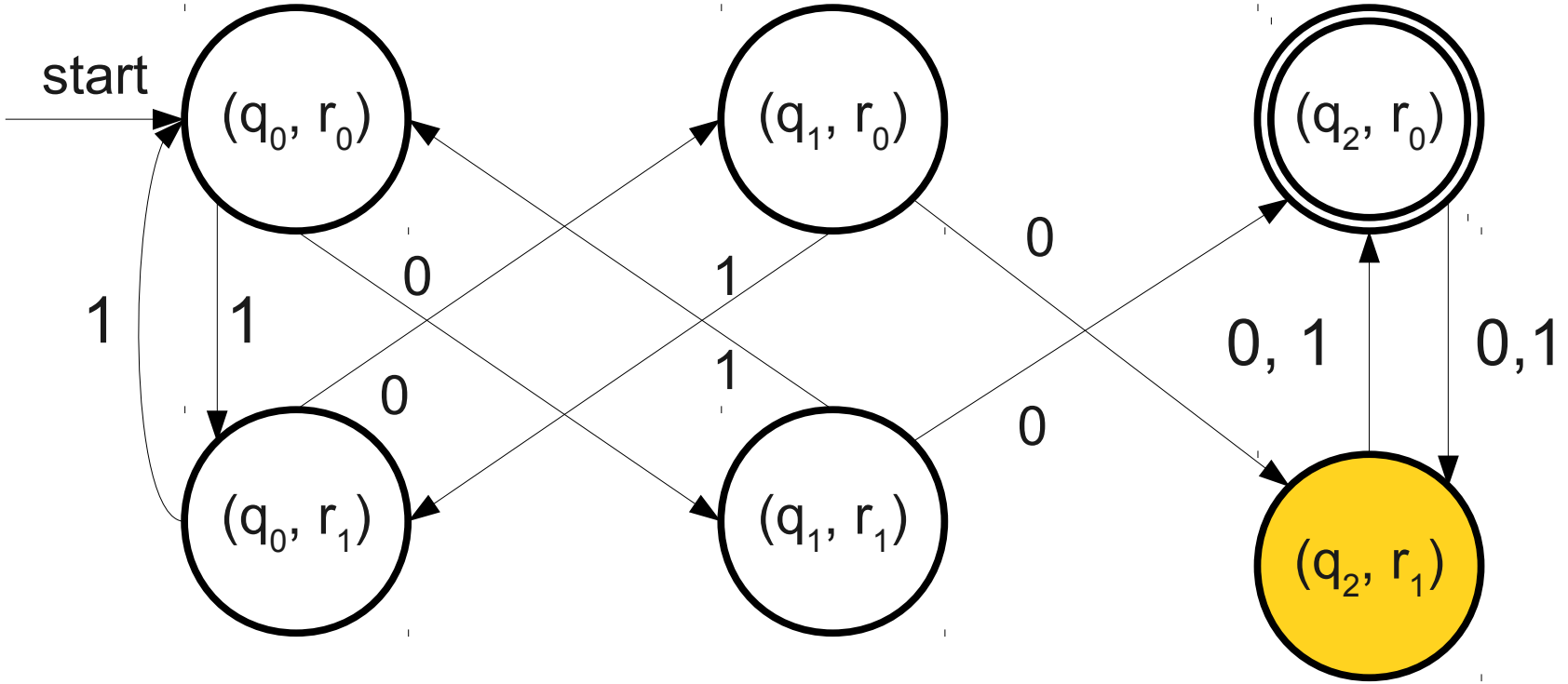


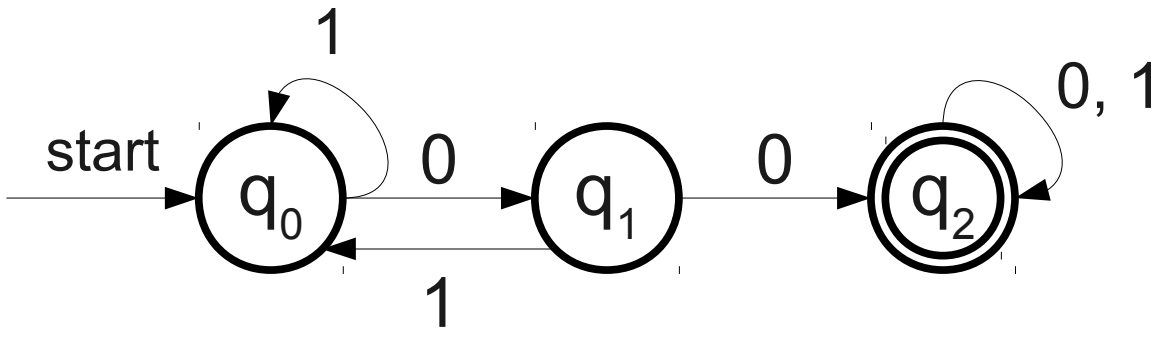


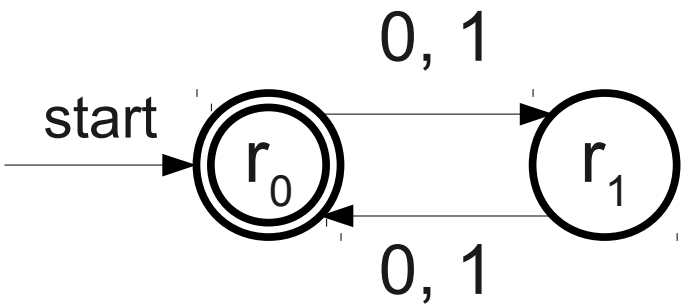


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

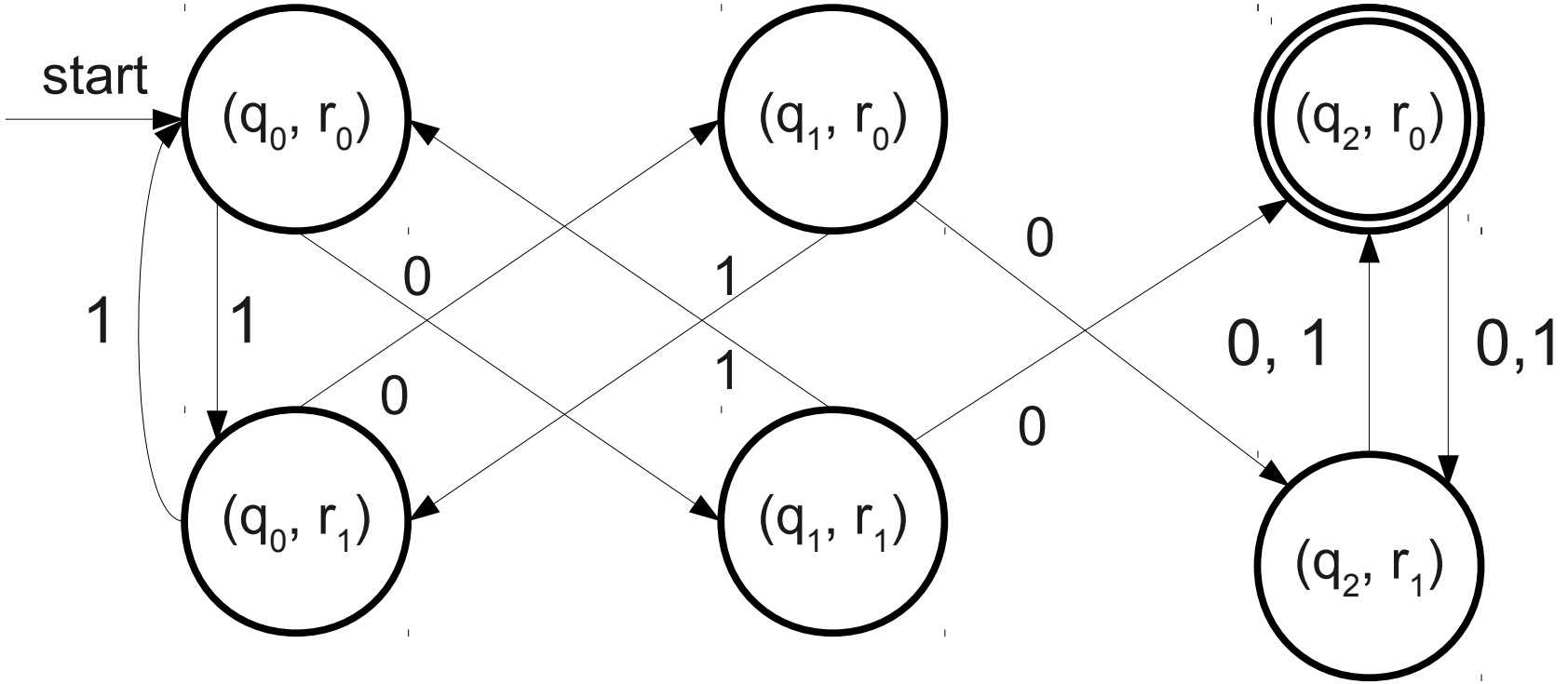


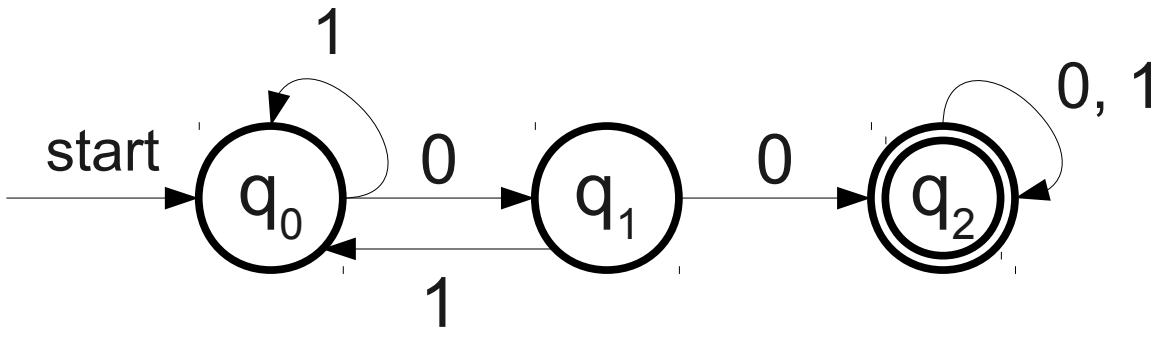


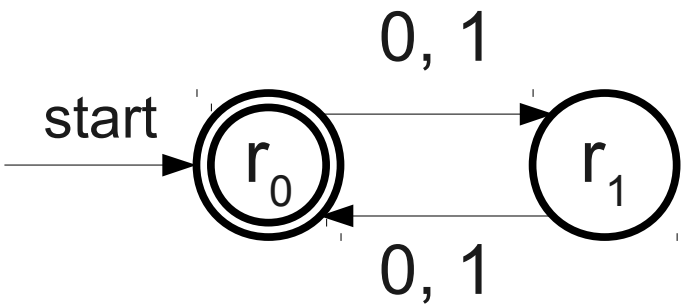


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

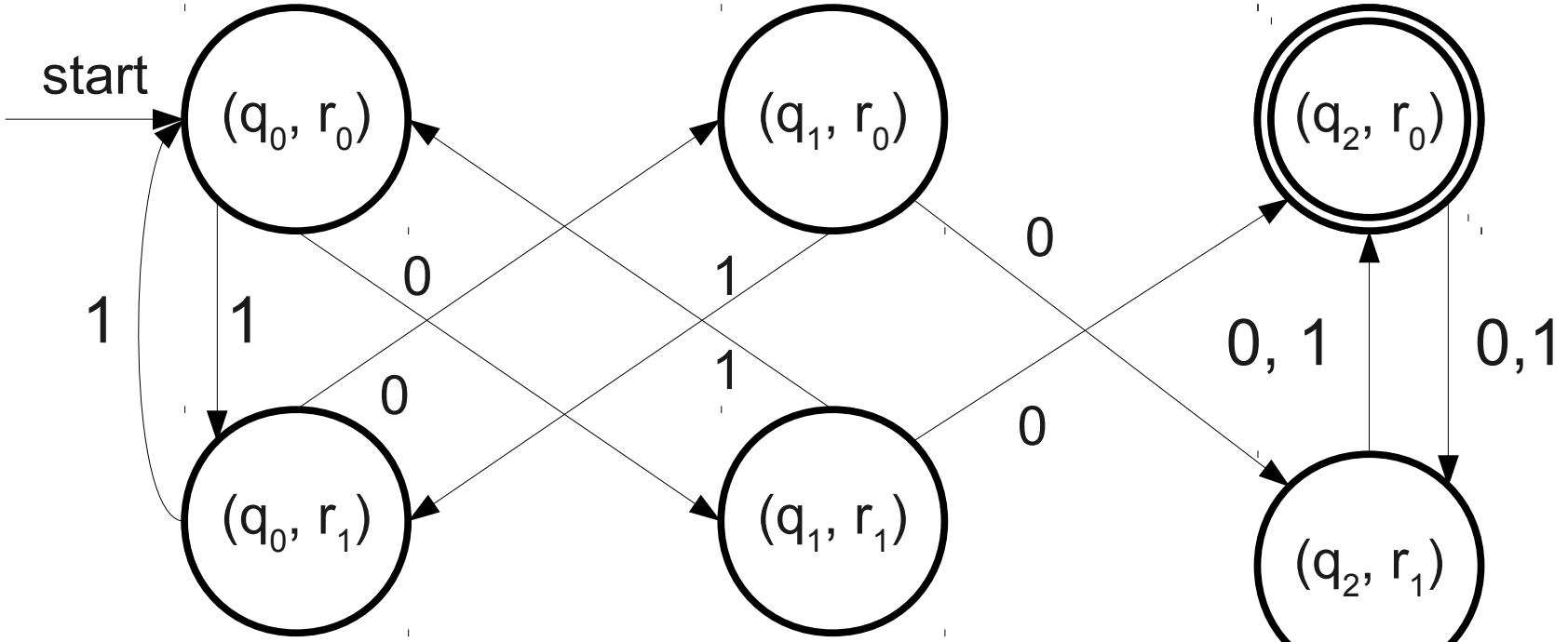




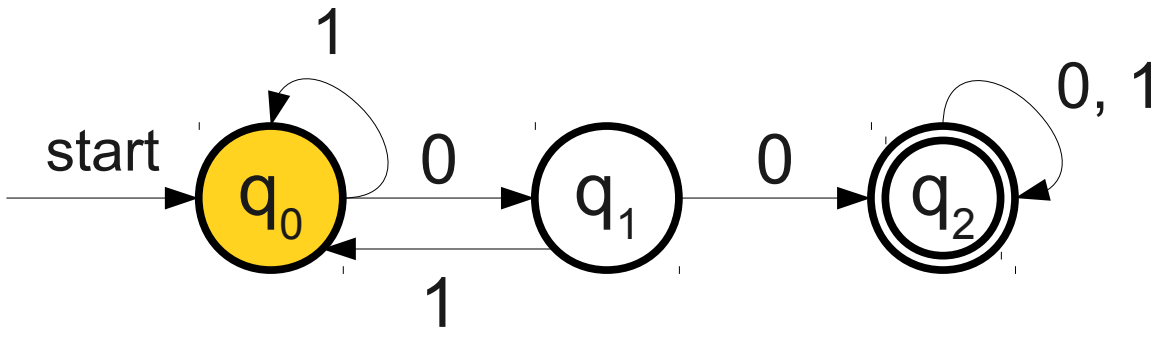


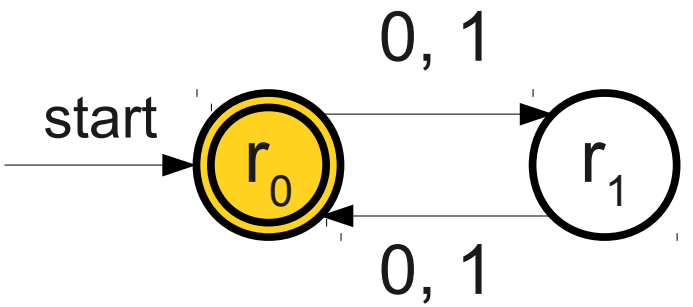
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



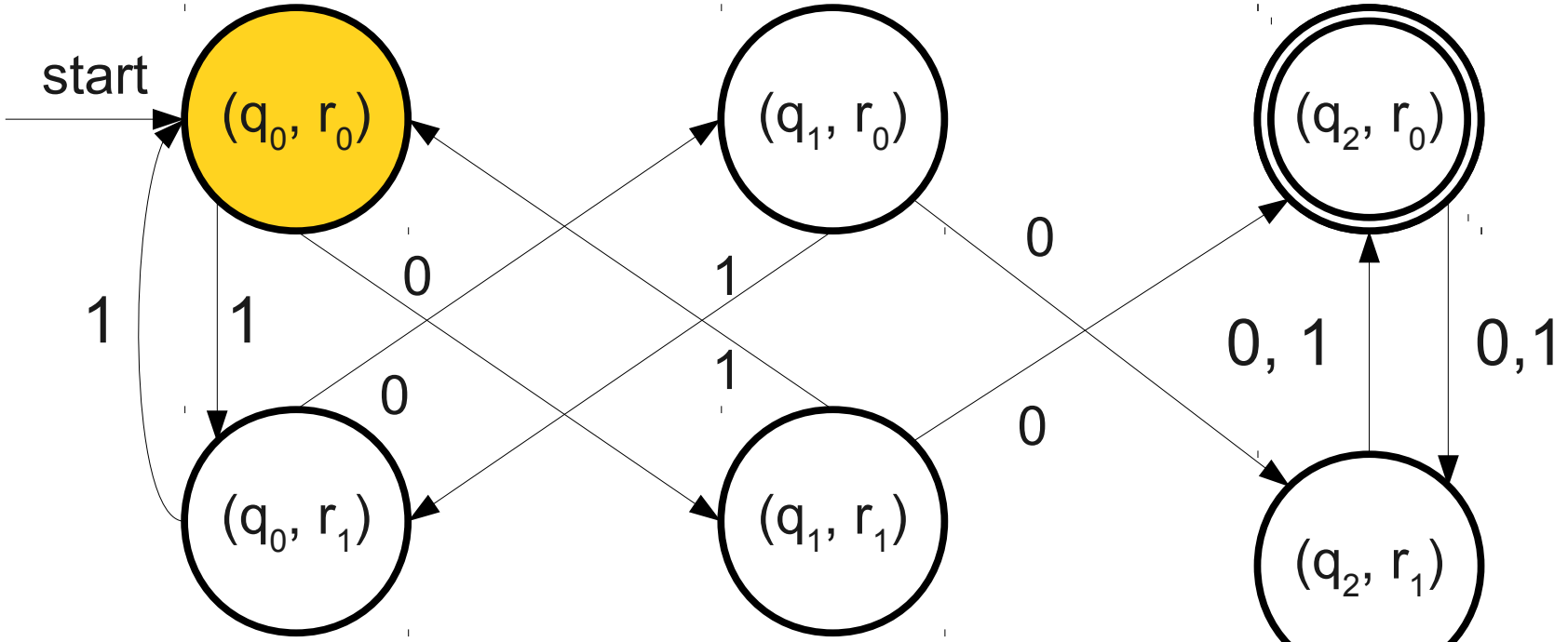
0 1 0 0 1



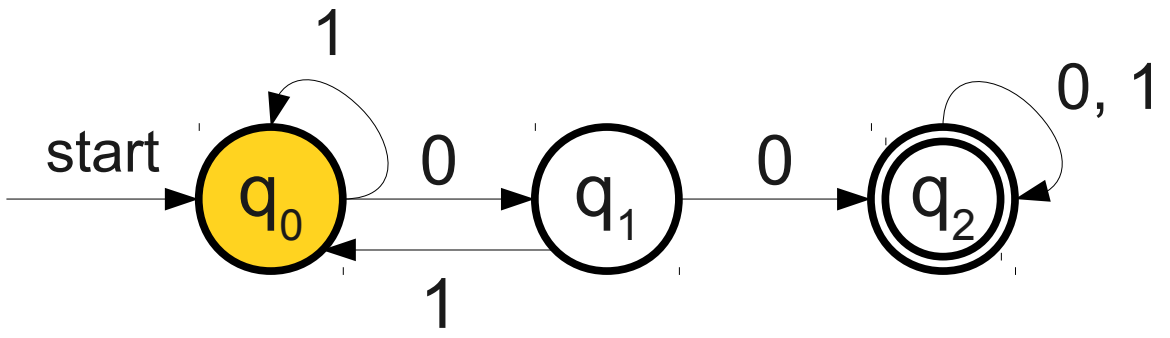


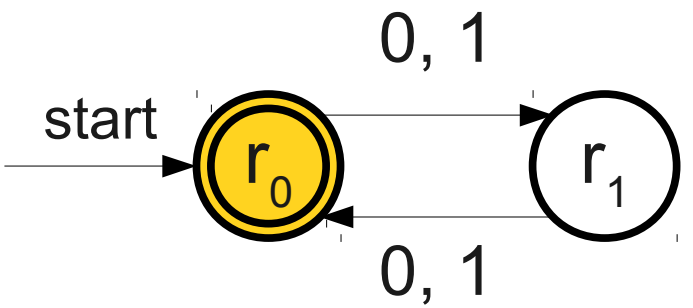
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



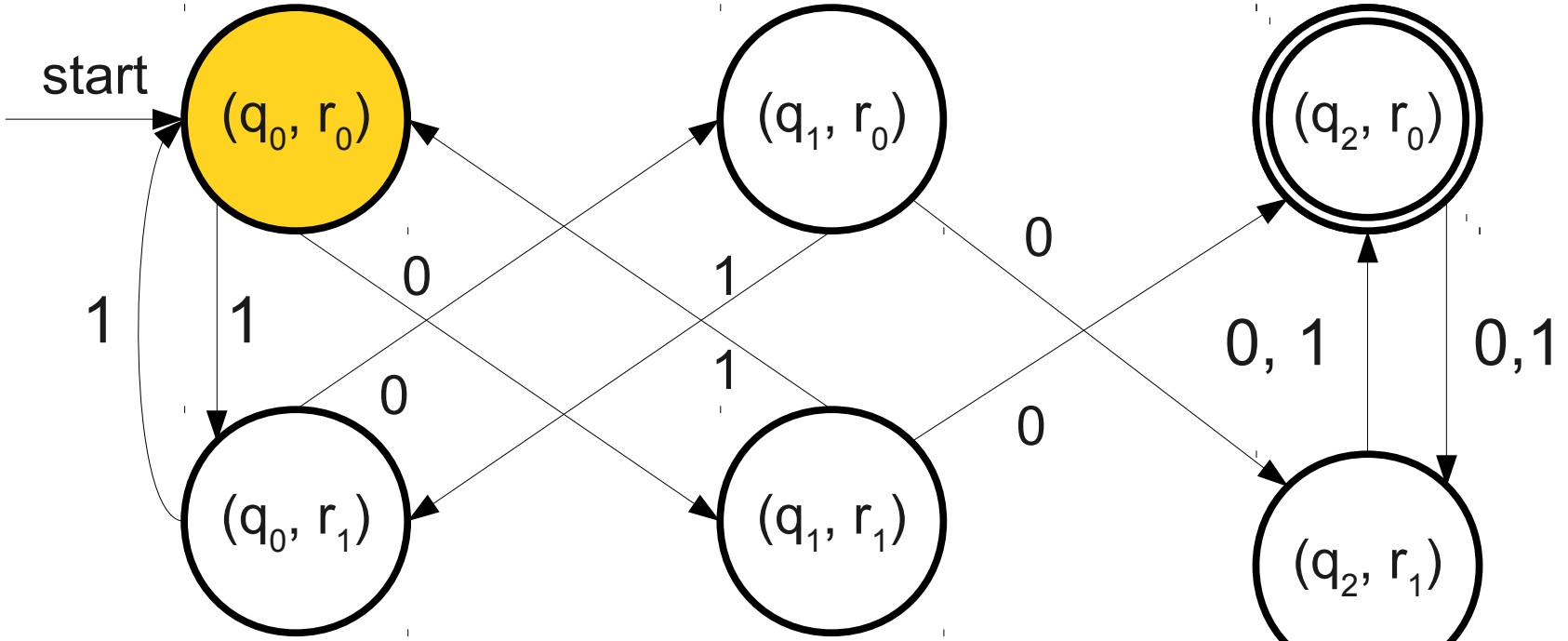
0 1 0 0 1

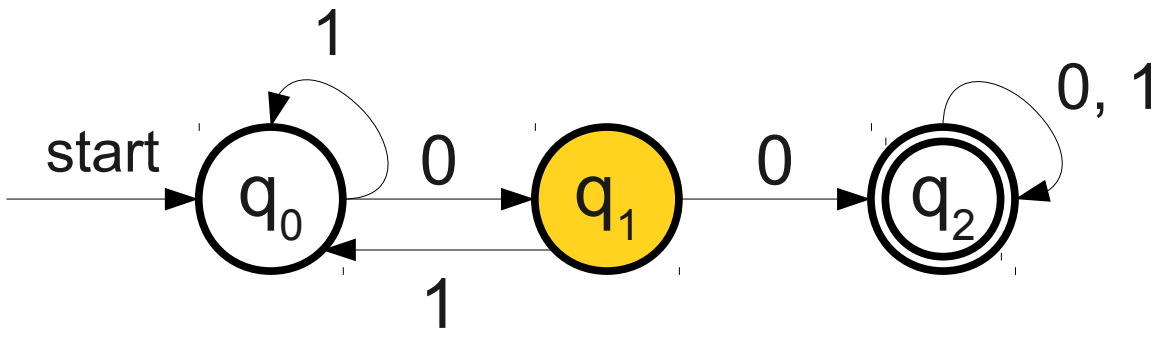


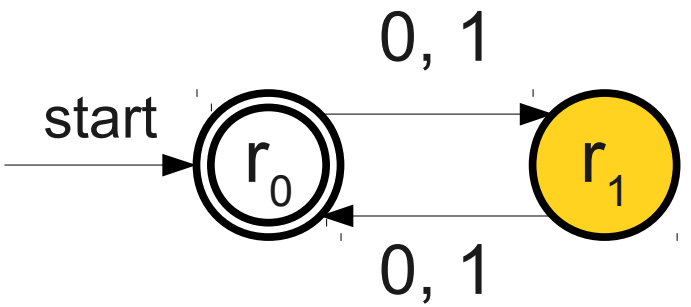


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

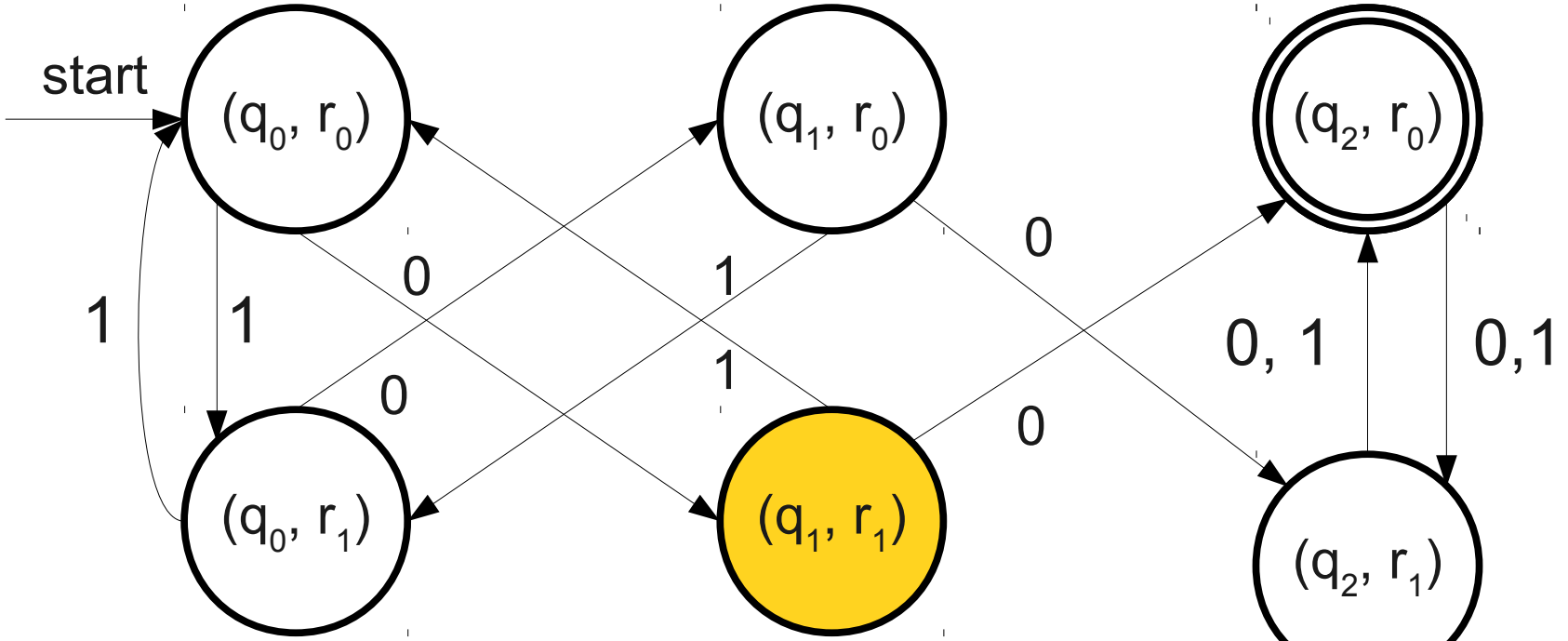


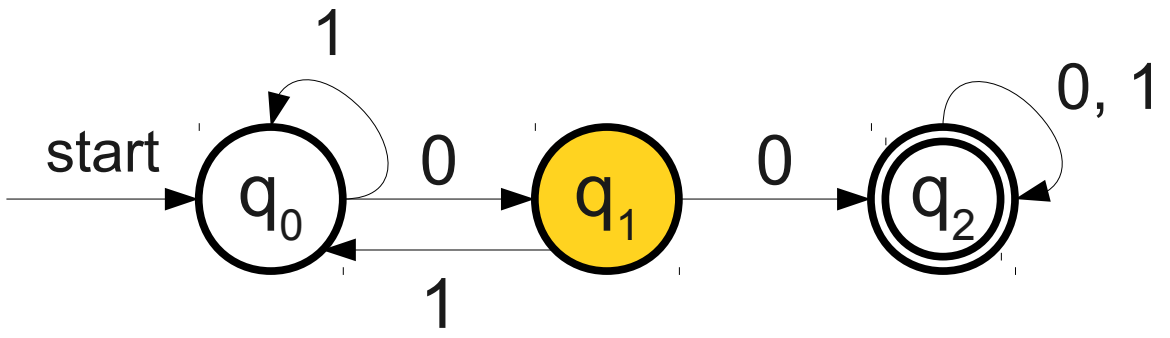


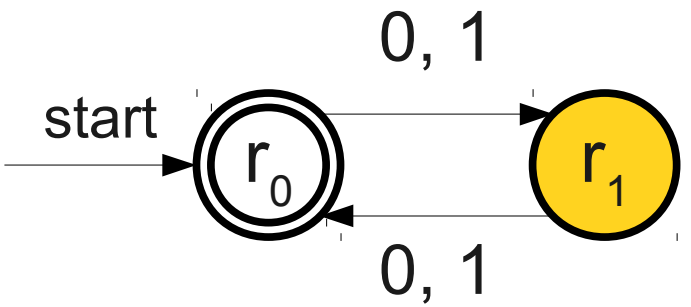


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

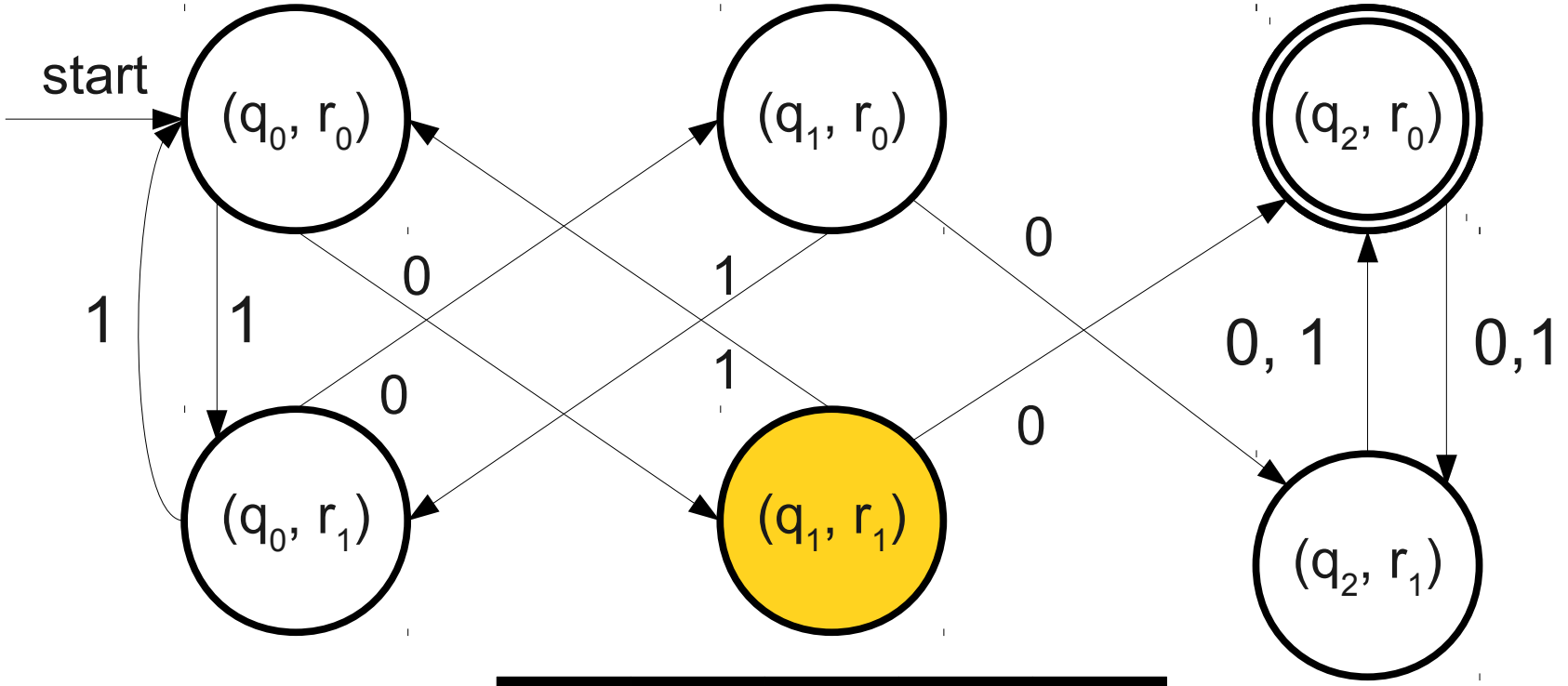


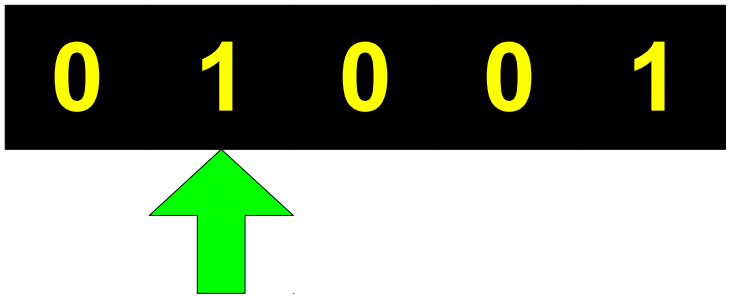


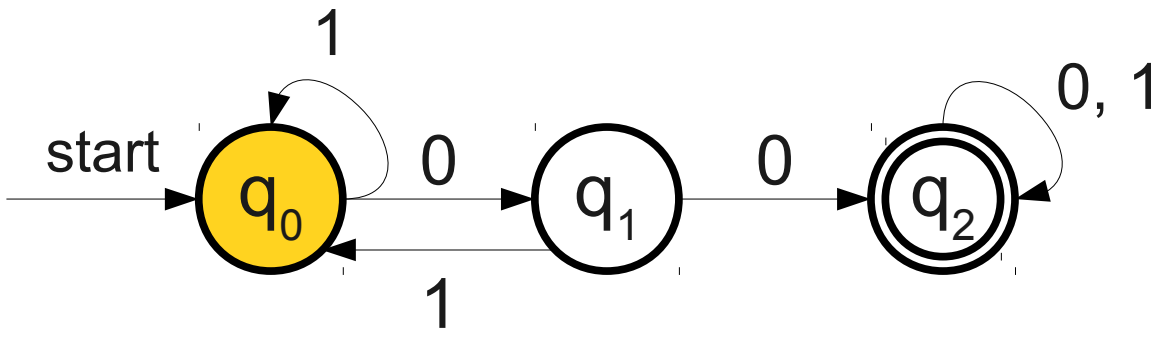


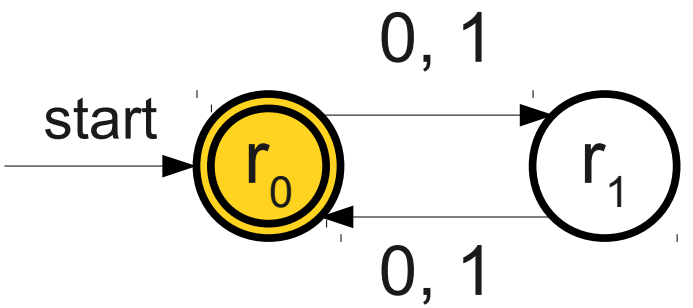
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



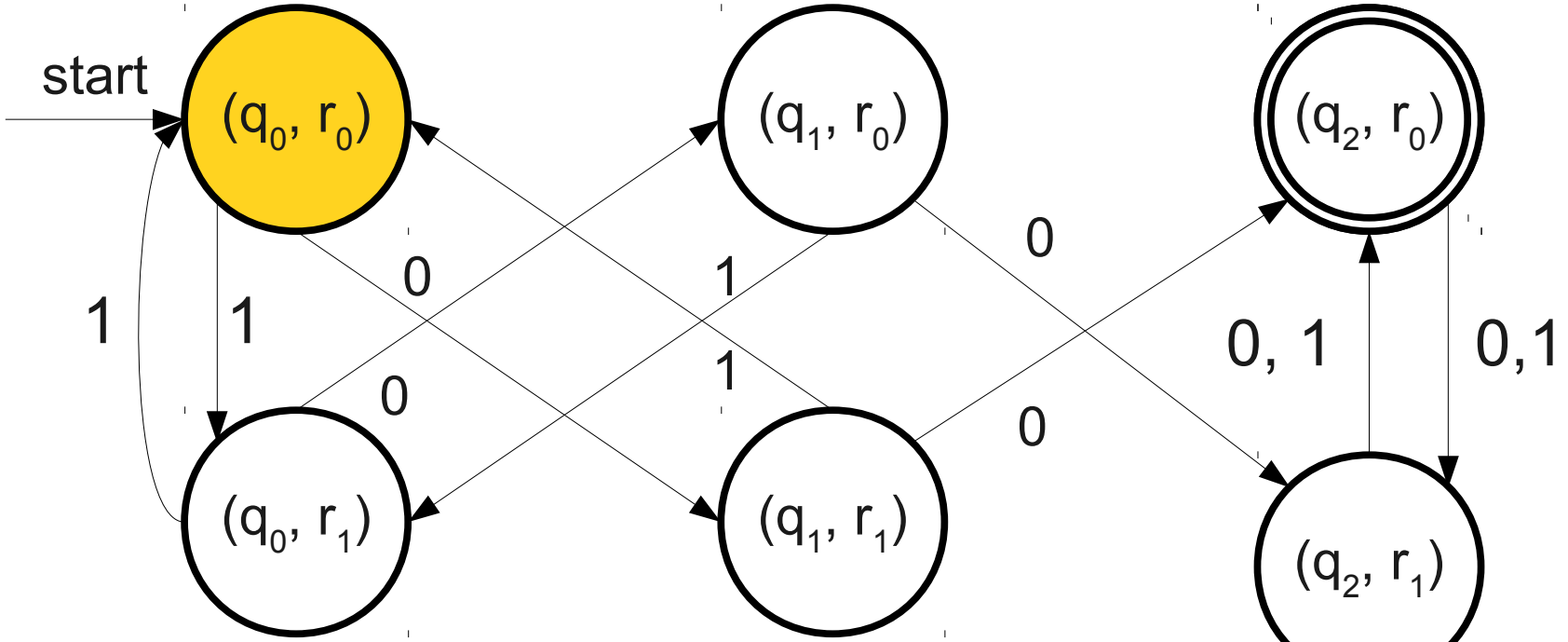


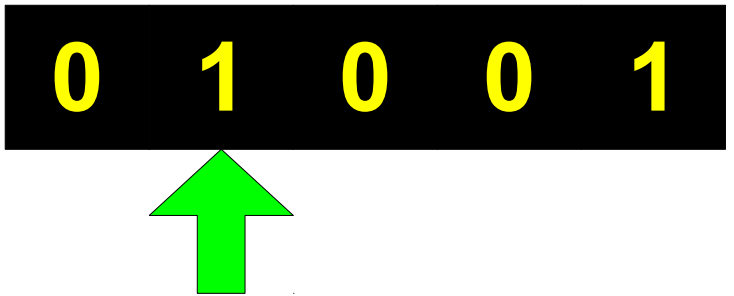


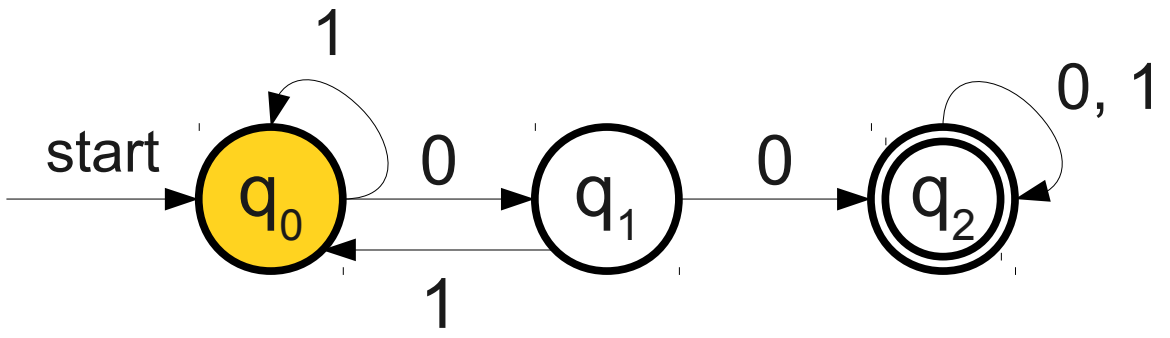


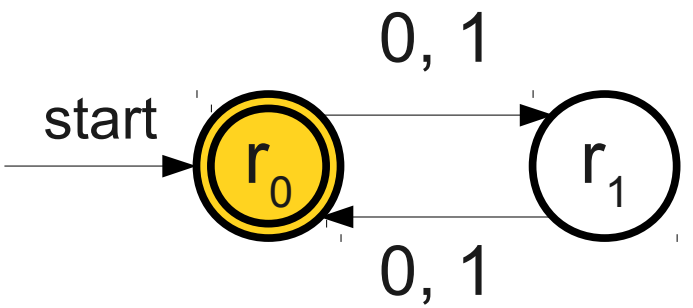
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



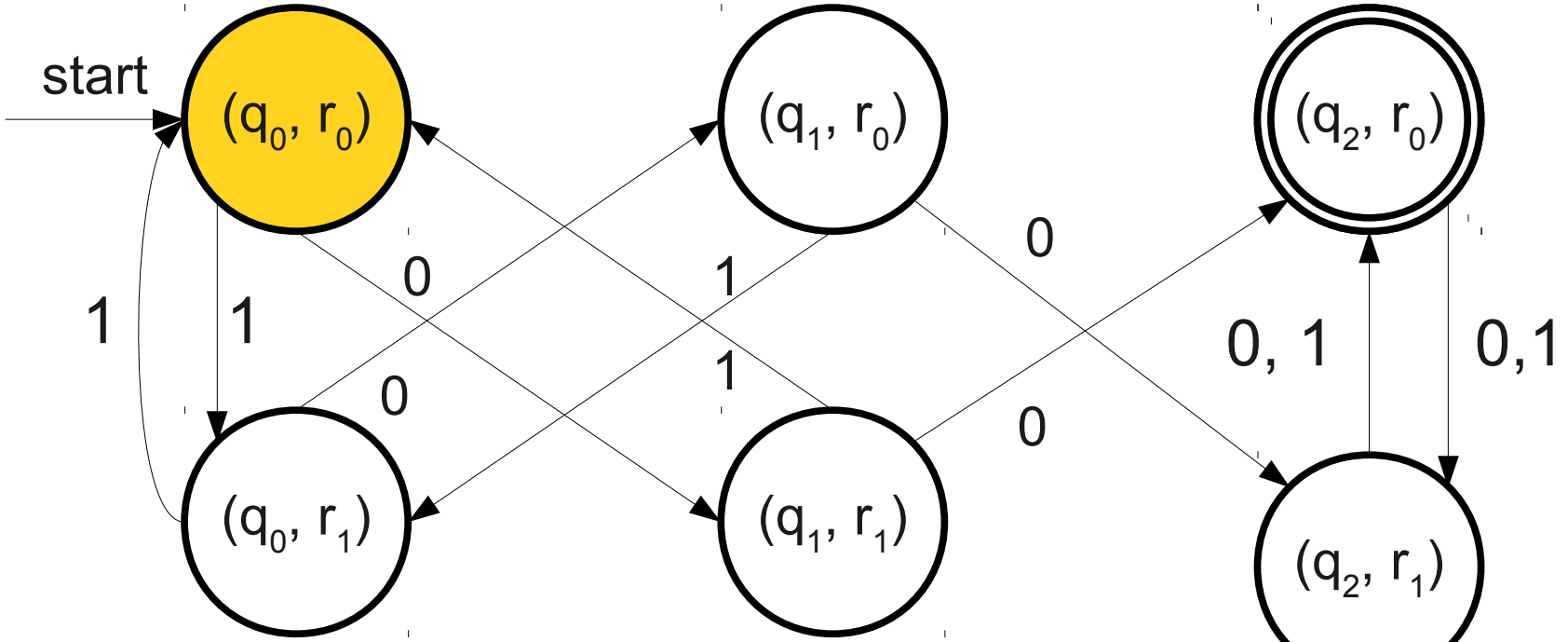


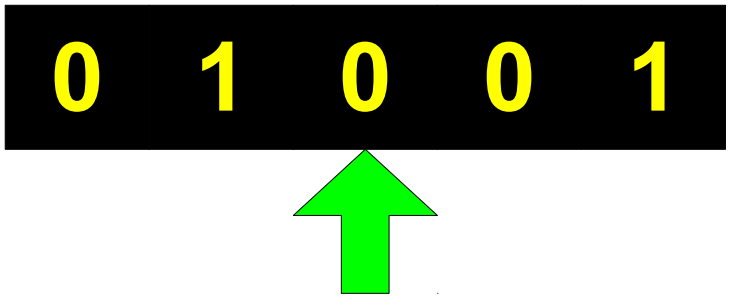


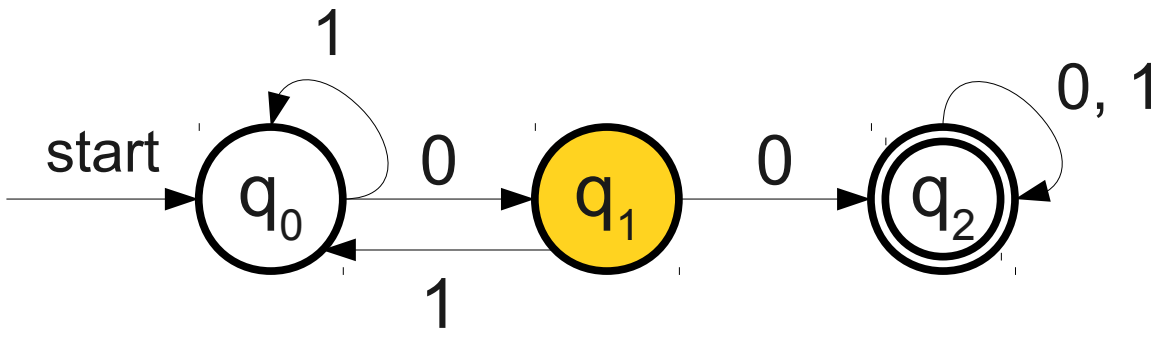


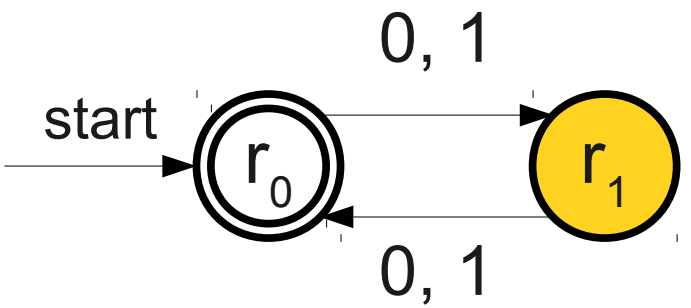
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



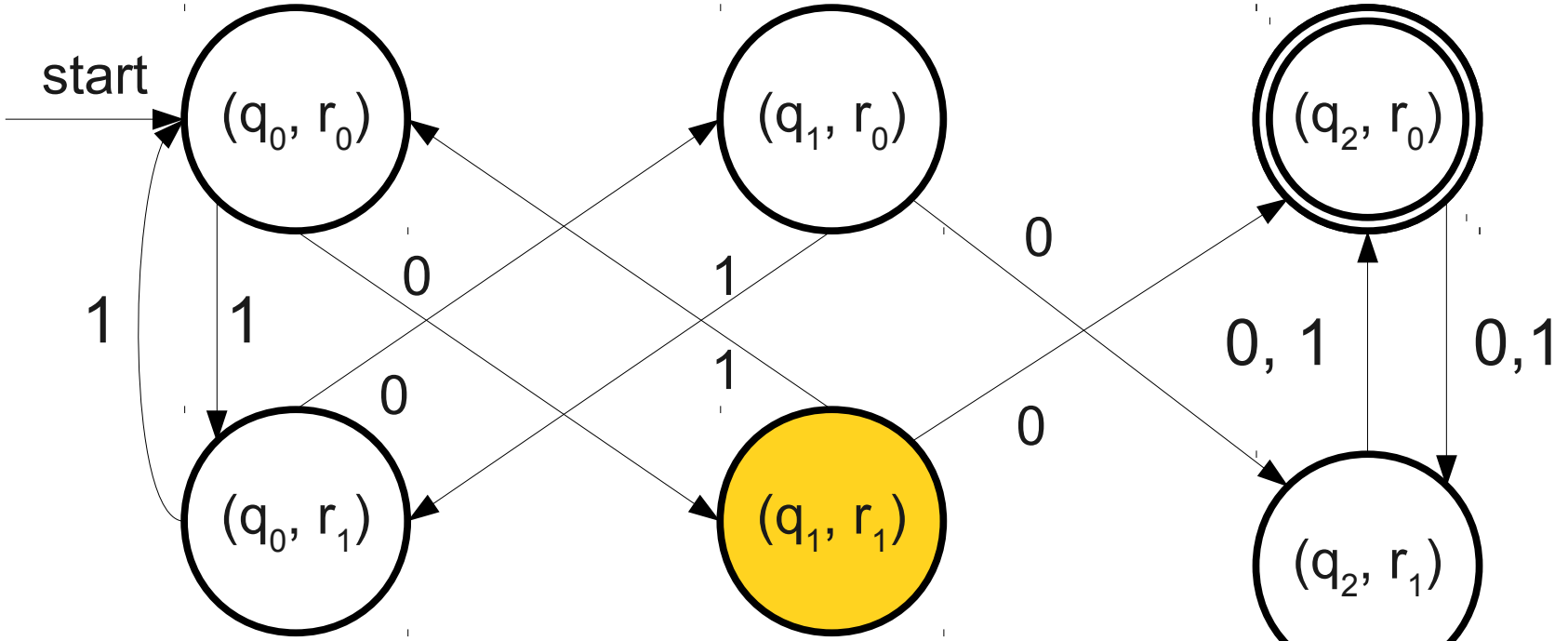


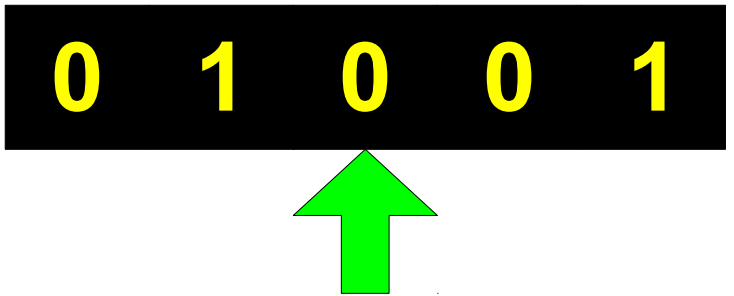


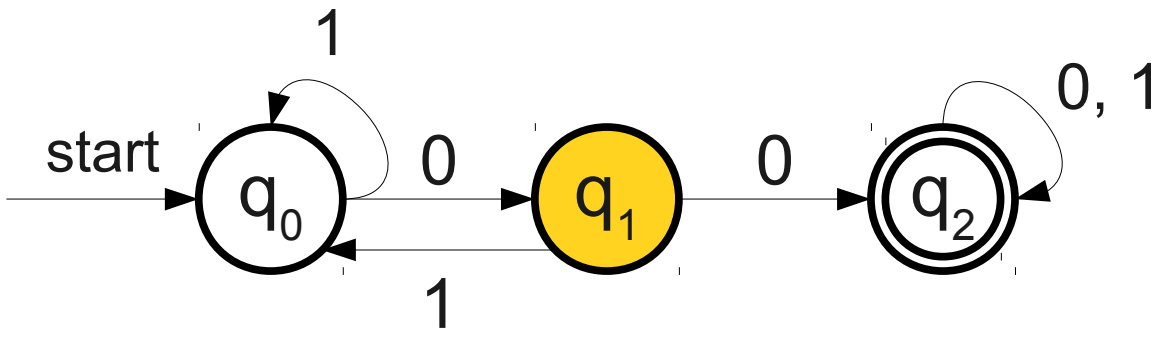


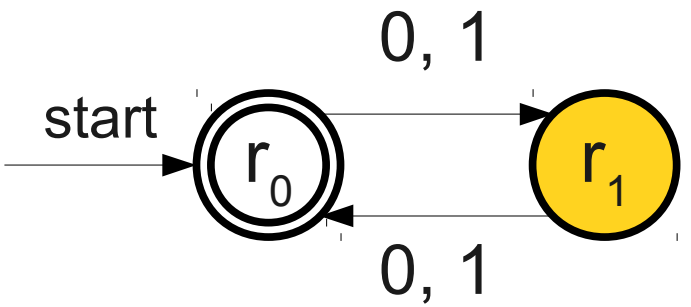
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



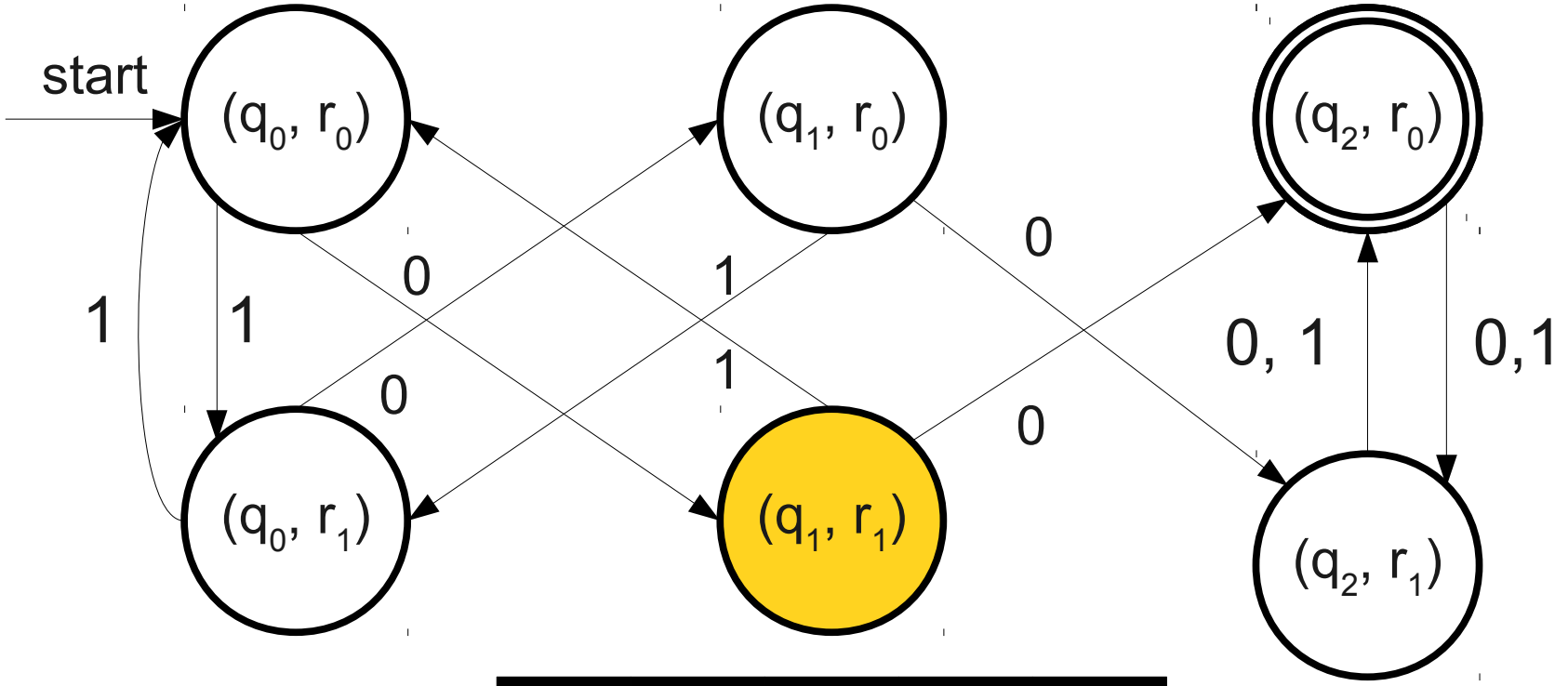


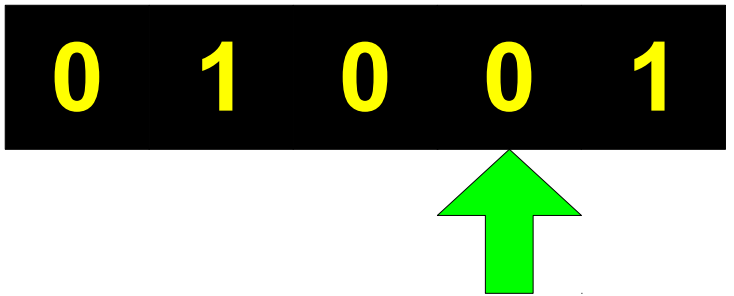


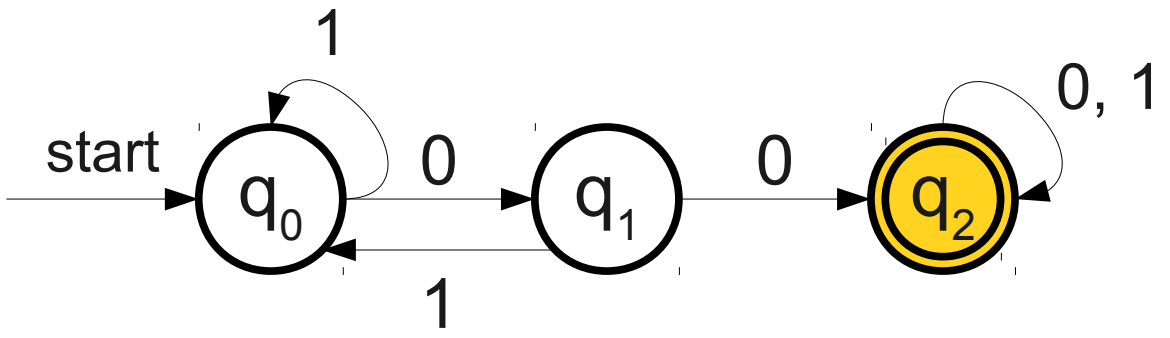


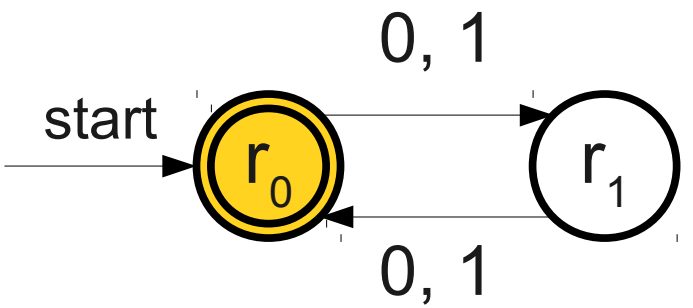
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



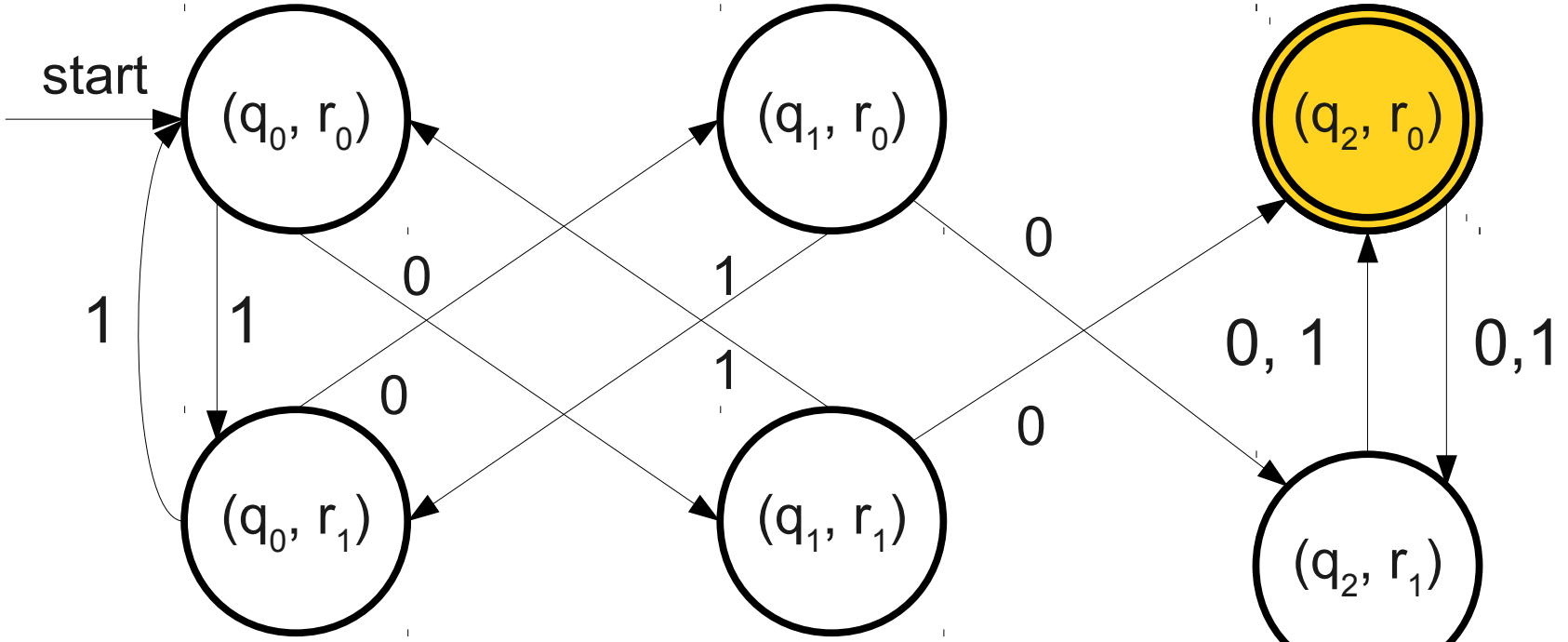


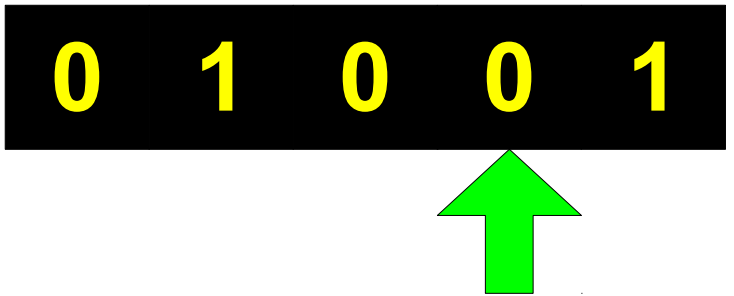


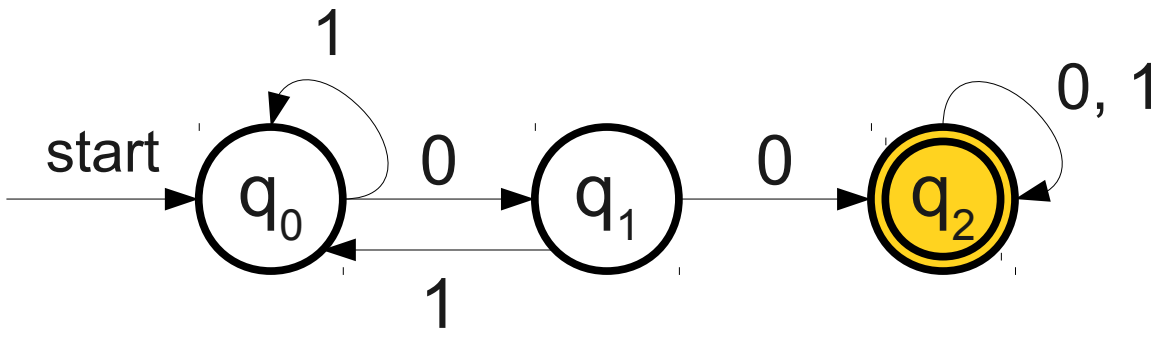


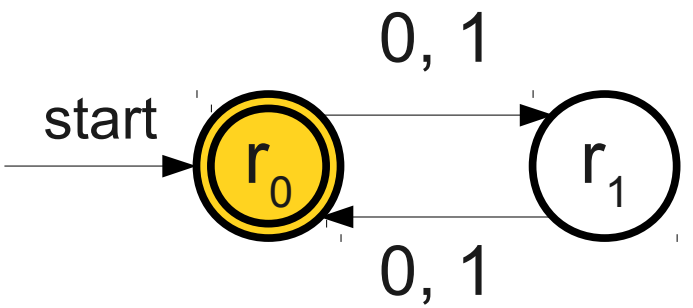
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



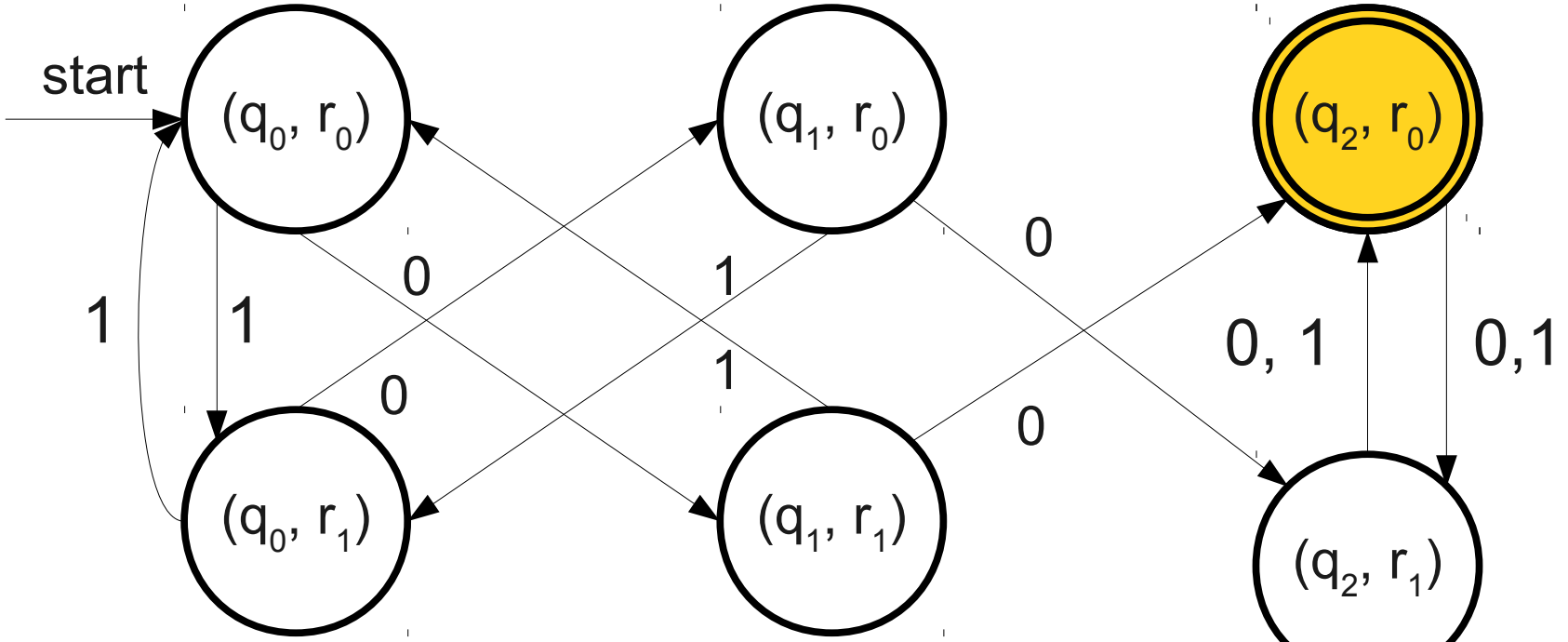


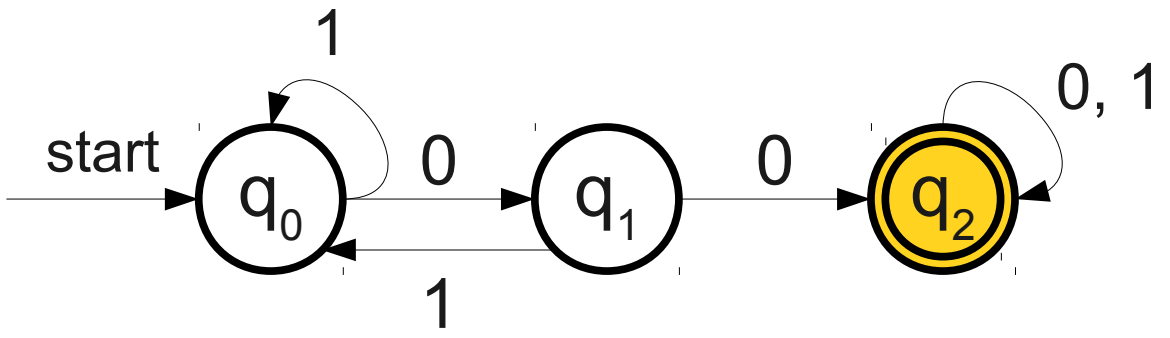


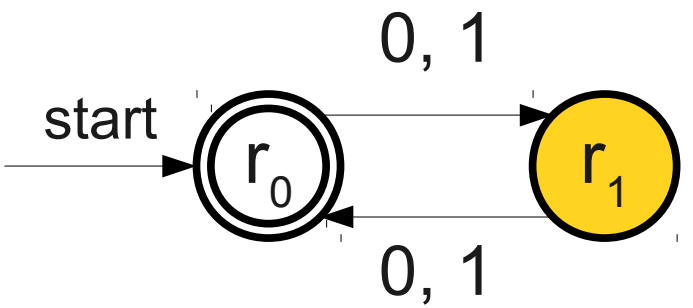


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

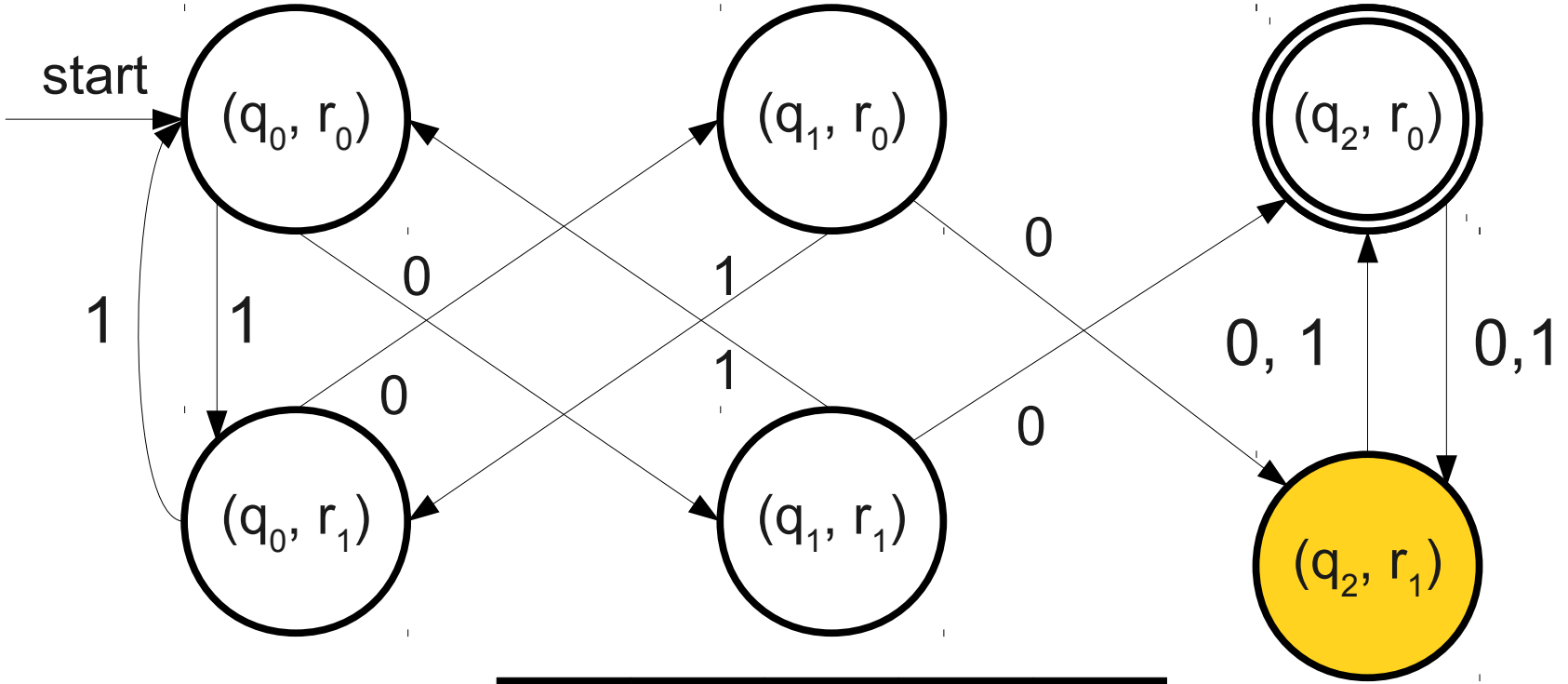


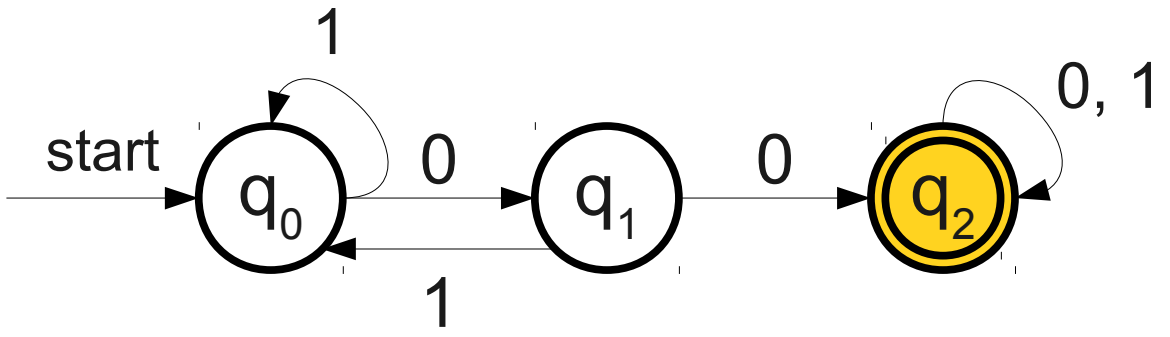


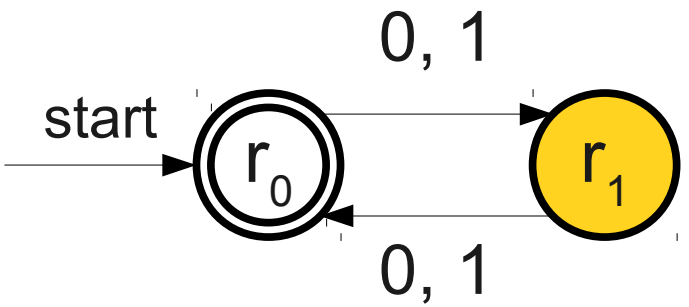


L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }

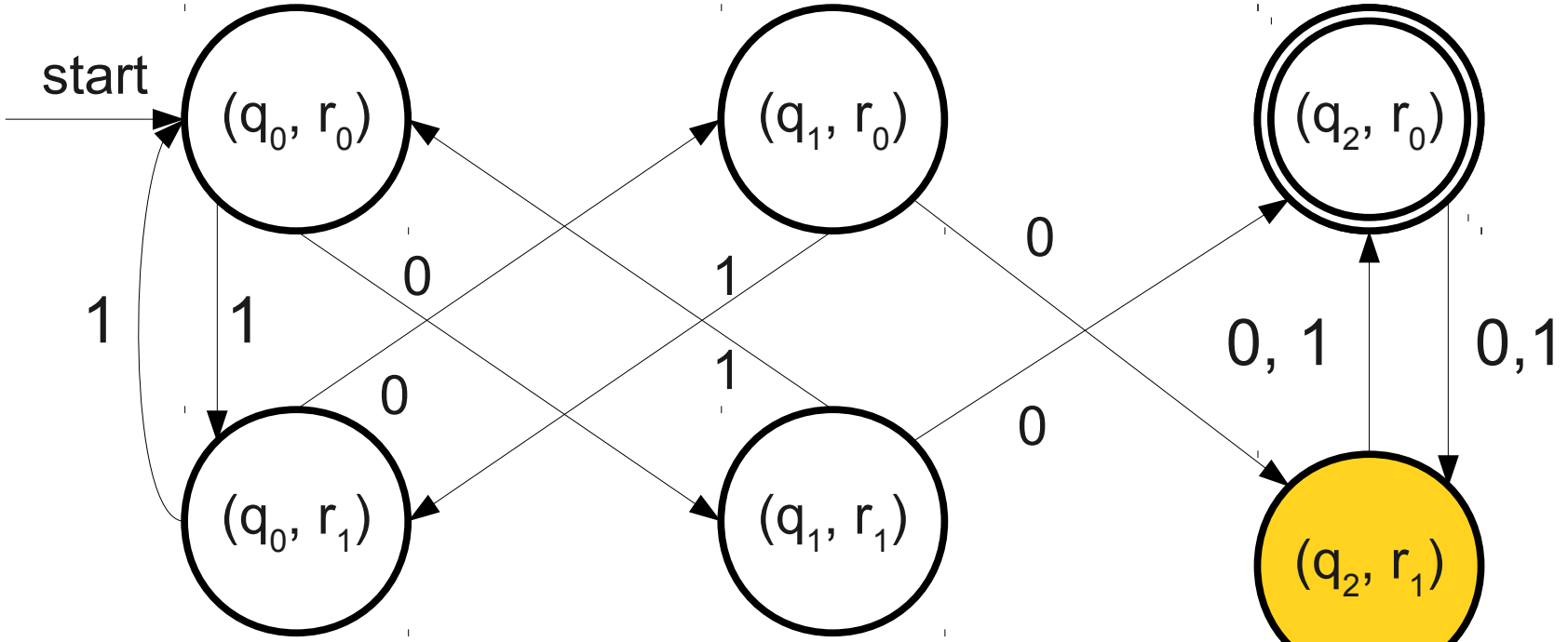




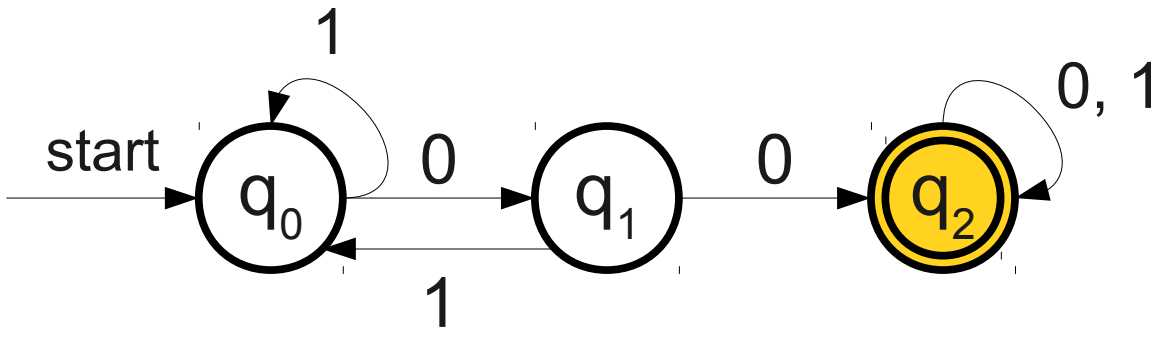


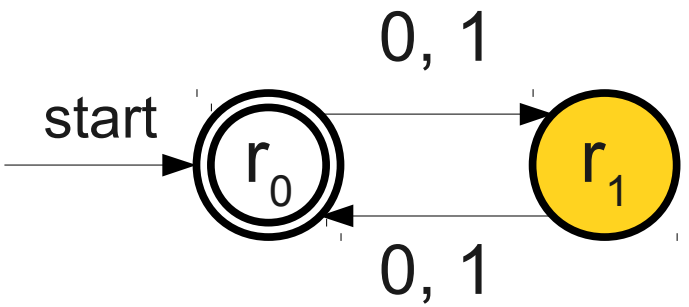
L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



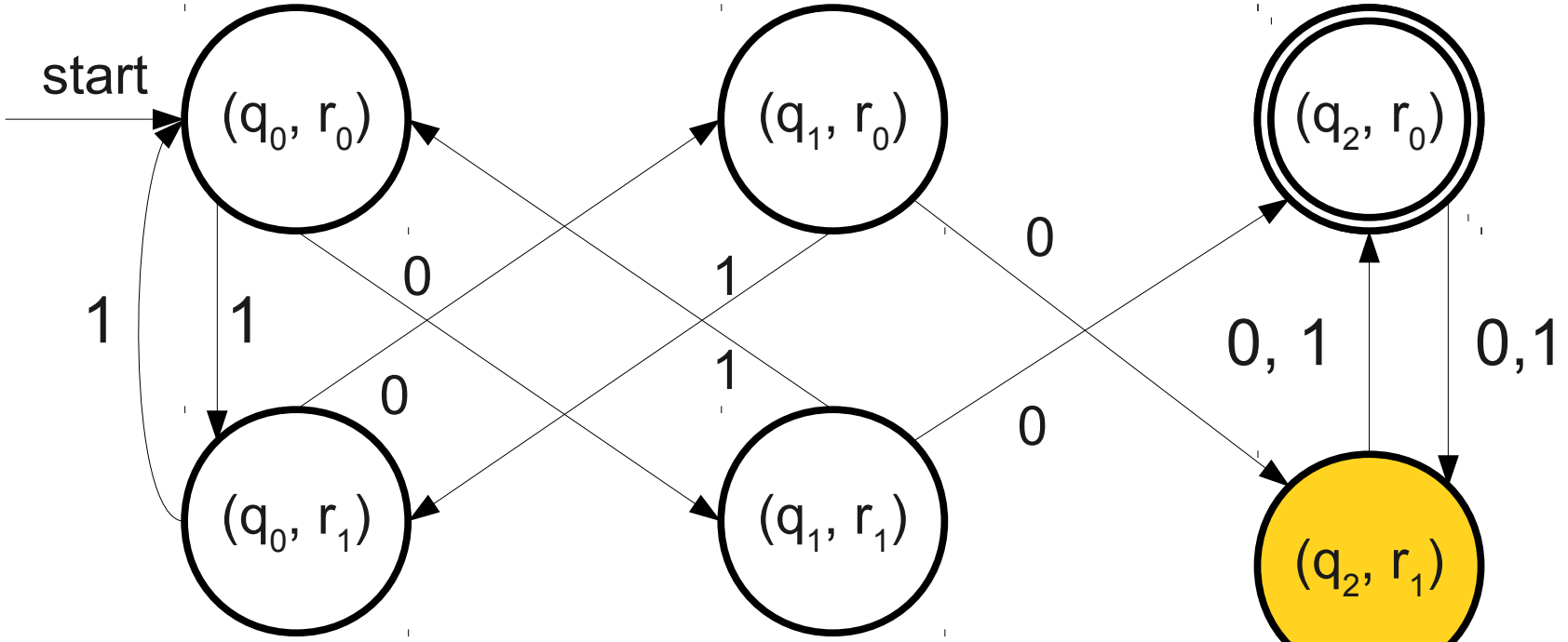
0 1 0 0 1





L_1 = { w | w contains 00 as a substring }

L_2 = { w | w has even length }



0 1 0 0 1

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where

The Product Construction

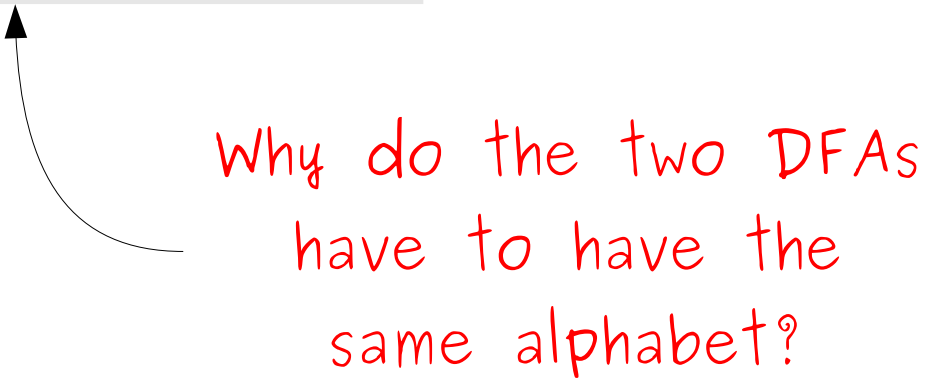
- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.

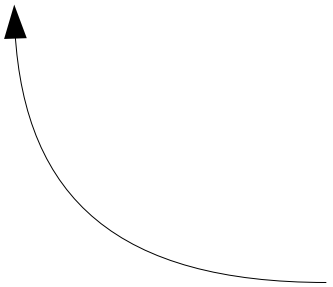


The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.
  - $\delta'((q_i, r_j), a) = (\delta_A(q_i, a), \delta_B(r_j, a))$

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.
  - $\delta'((q_i, r_j), a) = (\delta_A(q_i, a), \delta_B(r_j, a))$



The new transition function applies the old transition functions to both states.

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.
  - $\delta'((q_i, r_j), a) = (\delta_A(q_i, a), \delta_B(r_j, a))$

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.
  - $\delta'((q_i, r_j), a) = (\delta_A(q_i, a), \delta_B(r_j, a))$
  - (q_0, r_0) is the pair of the start states of A and B.

The Product Construction

- Given DFAs $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (R, \Sigma, \delta_B, r_0, F_B)$, the **product construction** constructs a new DFA $A \times B$ that accepts $L(A) \cap L(B)$.
- Formally, $A \times B = (Q \times R, \Sigma, \delta', (q_0, r_0), F_A \times F_B)$, where
  - $Q \times R$ is the Cartesian product of the two sets of states.
  - Σ is the original alphabet.
  - $\delta'((q_i, r_j), a) = (\delta_A(q_i, a), \delta_B(r_j, a))$
  - (q_0, r_0) is the pair of the start states of A and B.
  - $F_A \times F_B$ is the set of pairs of accepting states of A and B.