

Nondeterministic Finite Automata (NFA)

A useful generalization of a DFA

Learning Objectives

By the end of this lecture, you will be able to:

- **Define** what an NFA is and how it differs from a DFA
- **Identify** nondeterministic transitions in automata
- **Convert** between NFAs and DFAs
- **Implement** NFAs in code
- **Recognize** regular languages

What is an NFA?

Nondeterministic Finite Automata (NFA)

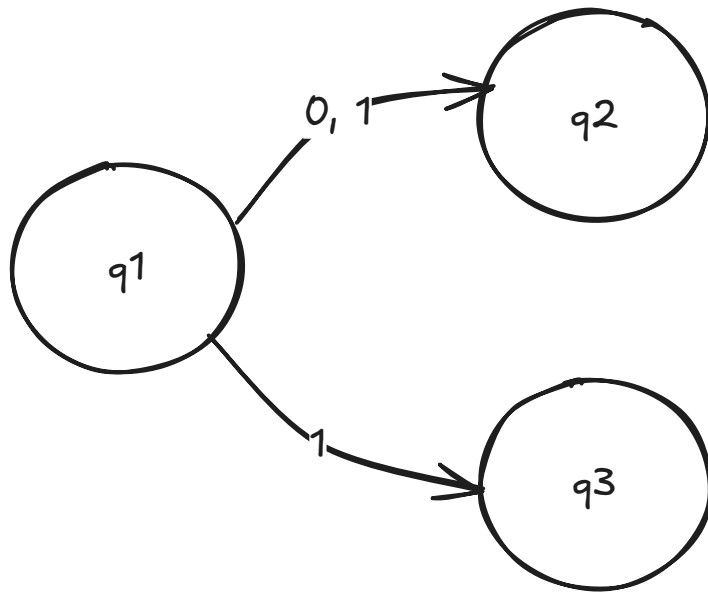
- A useful generalization of a DFA
- Allows more flexible state transitions
- Makes some automata simpler to design

Key Question: Does this added flexibility make NFAs more powerful than DFAs?

NFA Enhancement #1

Multiple Transitions on Same Symbol

A state can have transitions to **multiple states** on the same symbol



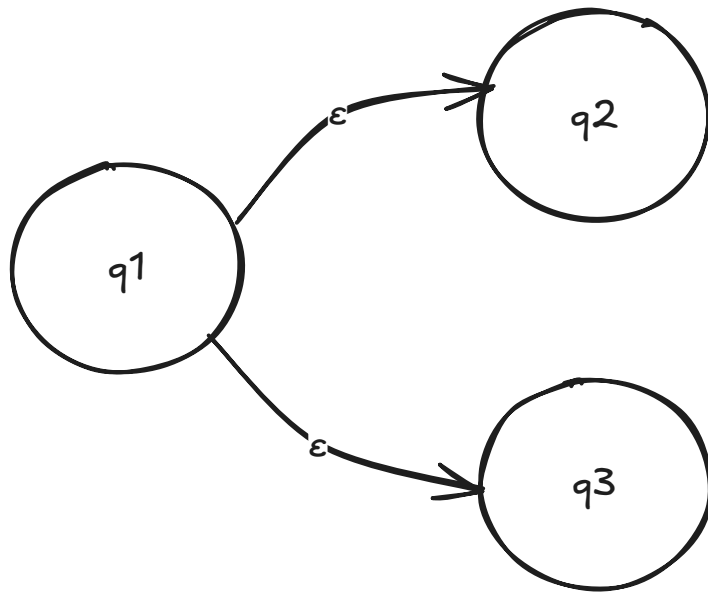
When reading '1' from state q1:

- Can go to q2 OR
- Can go to q3

NFA Enhancement #2

Null Transitions (ϵ -transitions)

A state can transition without reading any symbol

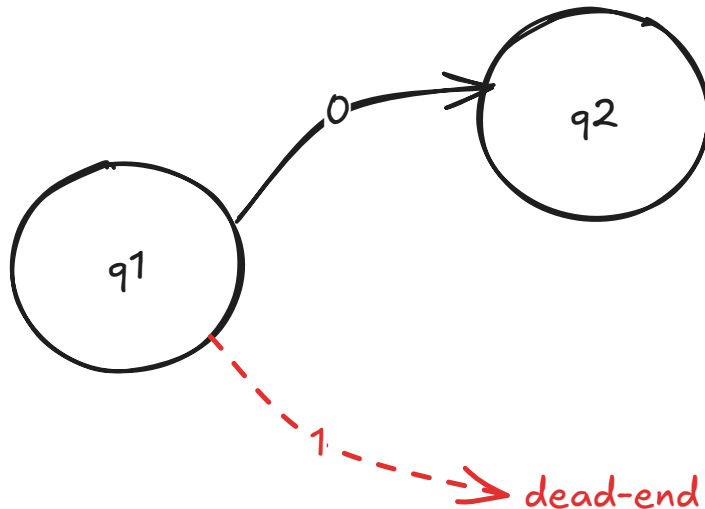


From q_1 , can spontaneously move to q_2 or q_3 without consuming input

NFA Enhancement #3

Missing Transitions

A state can have **no transitions** for some symbols (dead ends)



No transition defined for '1' from q_1

- If '1' is read, this path dies

How to Run an NFA?

The Nondeterministic Approach:

1. **Follow all possible paths** simultaneously
2. If **any path** leads to an accept state → **ACCEPT**
3. If **all paths** fail → **REJECT**

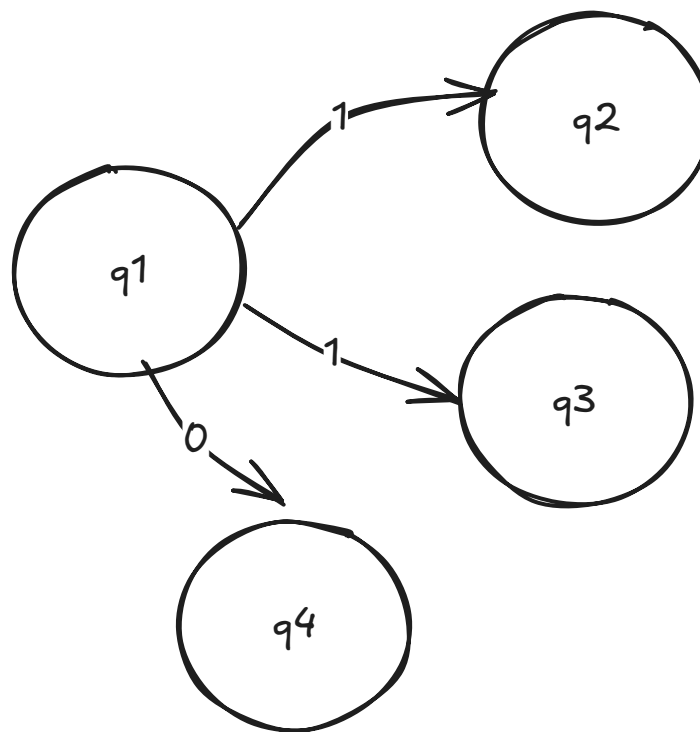
Think of it as:

- Exploring multiple parallel universes
- Success in any universe = overall success



Quick Check #1

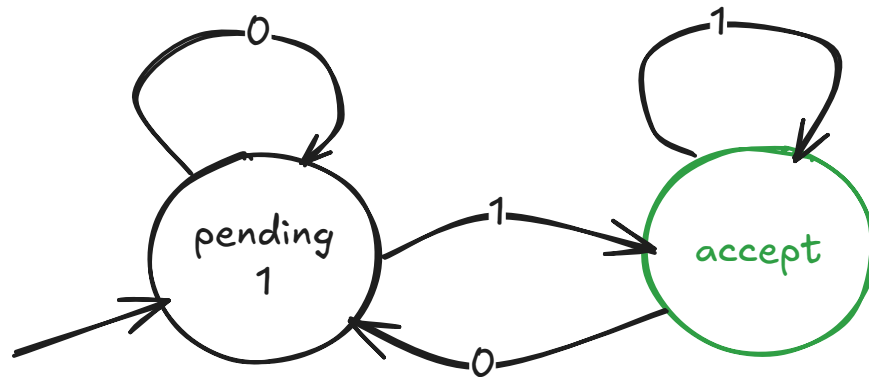
Given this NFA fragment, what happens when we read '1' from q1?



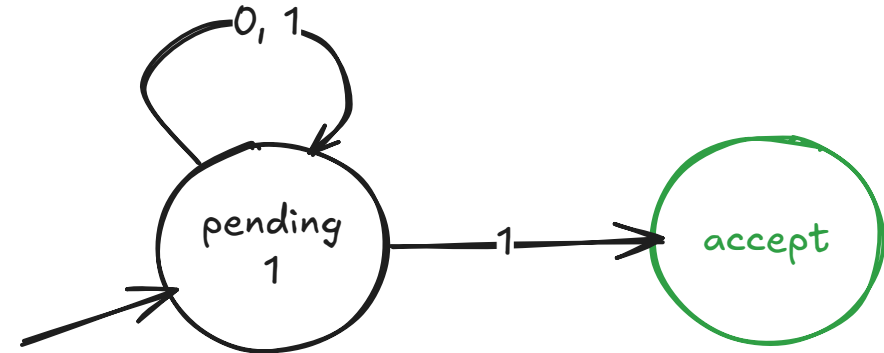
Example 1: Strings Ending in "1"

Language: Binary strings ending in "1"

DFA Version

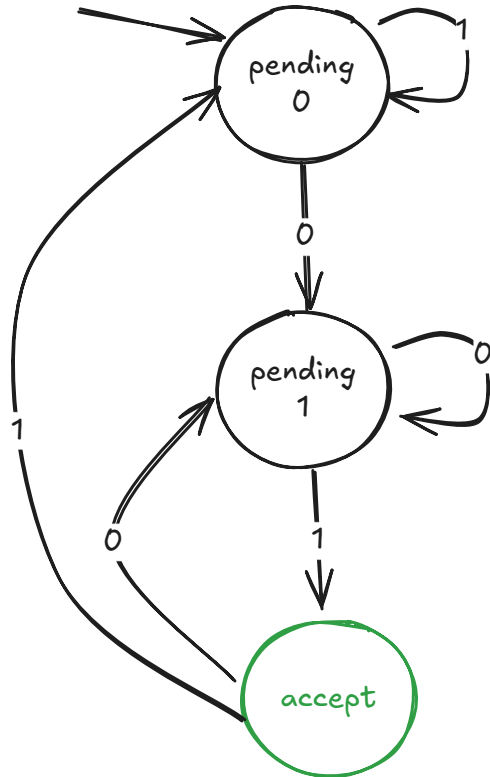


NFA Version (Simpler!)

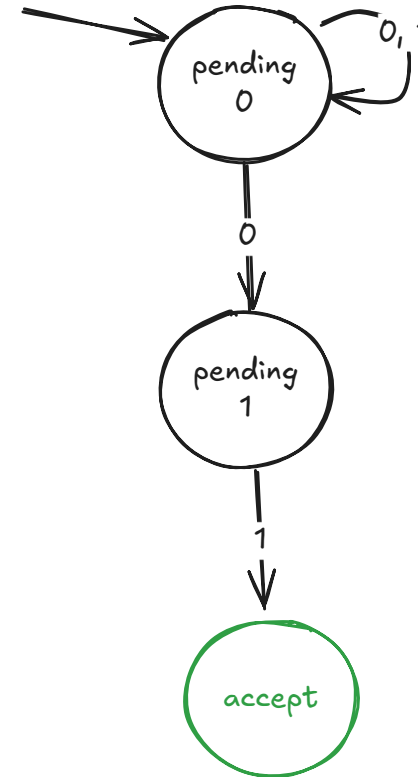


Example 2: Strings Ending in "01"

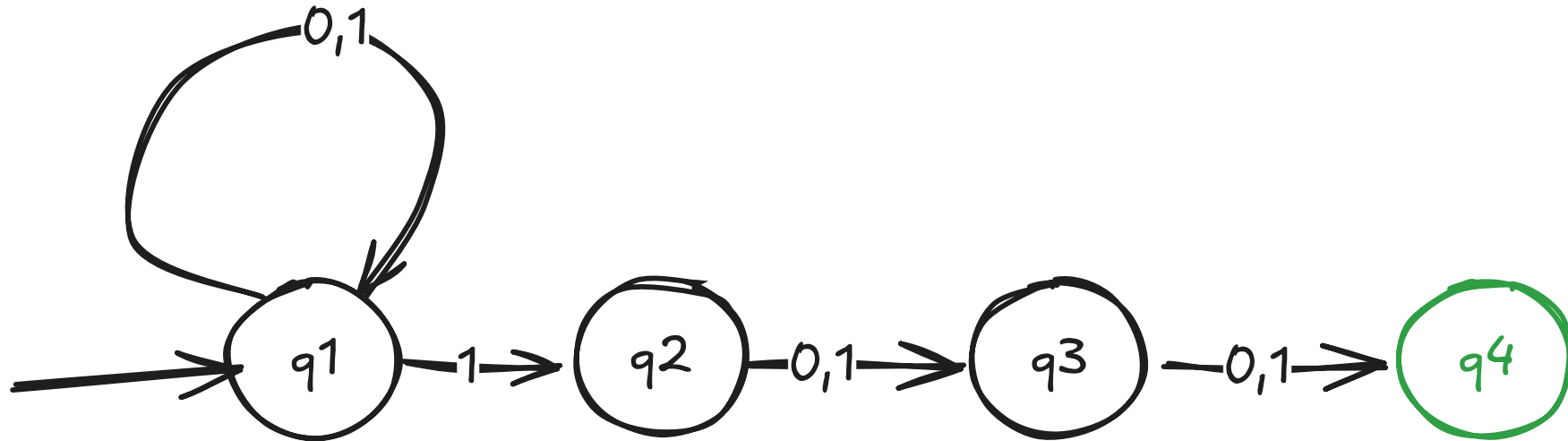
DFA Version



NFA Version

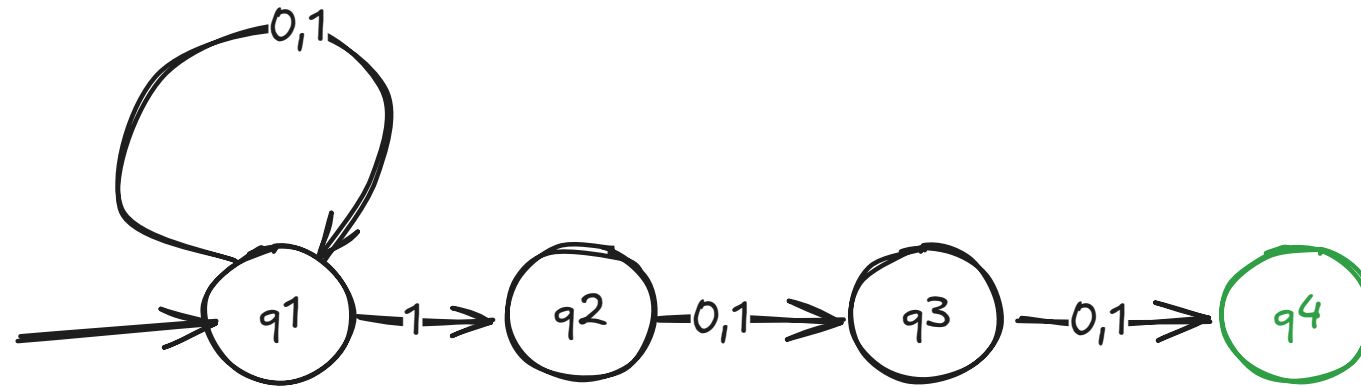


Example 3: Third Position from End



Question: Where's the nondeterminism?

Example 3: Analysis



Nondeterminism: $q1$ has multiple transitions on '1'

- $q1 \rightarrow q1$ (stay)
- $q1 \rightarrow q2$ (guess this is 3rd from end)

Language: Binary strings with '1' in 3rd position from end



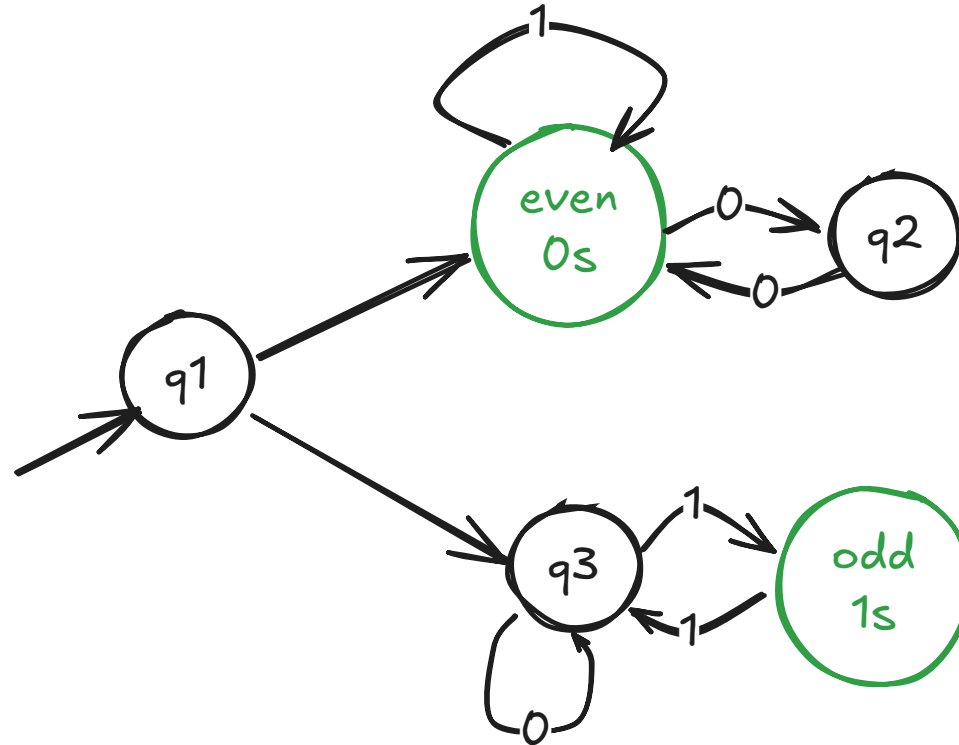
Active Learning: Trace Example

Trace the string "10110" through Example 3 NFA

Work with a partner to:

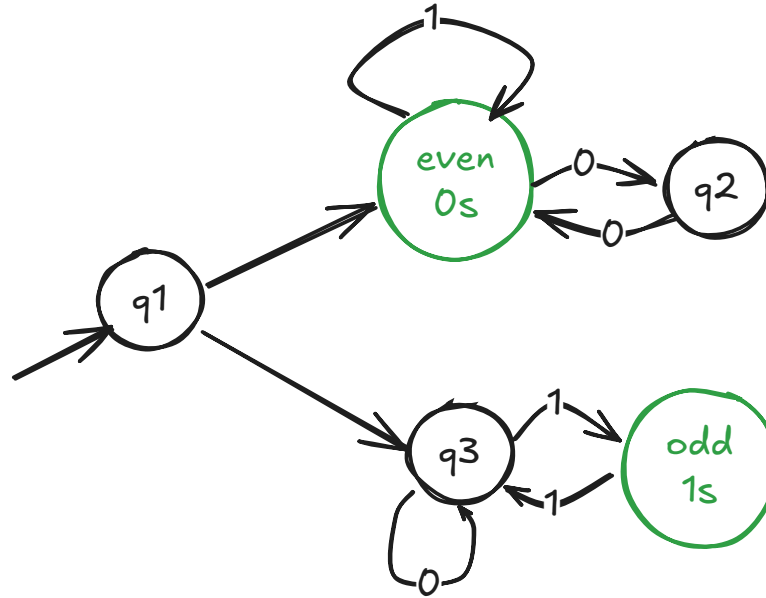
1. List all possible paths
2. Identify which paths accept
3. Determine if string is accepted

Example 4: "OR" with null-transitions



Where's the nondeterminism?

Example 4: Analysis



Nondeterminism: $q1$ has ϵ -transitions to two branches

Language: Strings with even number of 0s OR odd number of 1s

The Big Question

Does nondeterminism make NFAs more powerful than DFAs?



Think about it...

Can NFAs recognize languages that DFAs cannot?

The Surprising Answer

NO!

| An NFA is equivalent to a DFA

Every NFA can be converted to an equivalent DFA!

But how?



NFA to DFA Conversion: The Idea

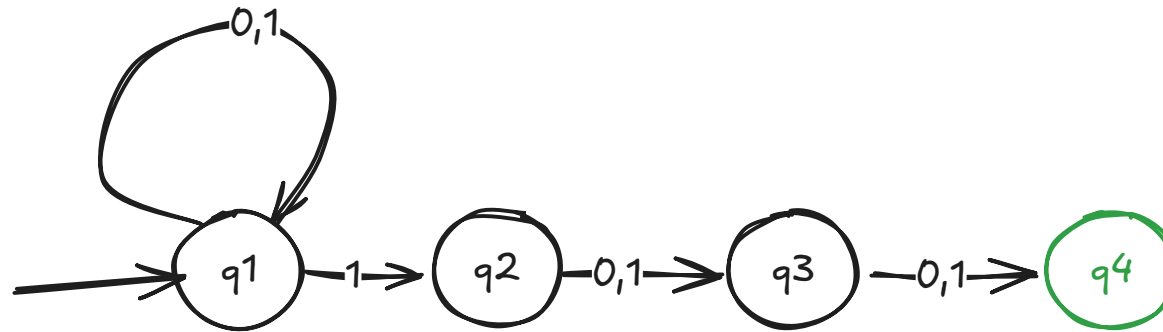
Subset Construction:

- If NFA has N states
 - There are 2^N possible subsets of states
- Each subset becomes a DFA state

Example: NFA with 3 states $\{q_1, q_2, q_3\}$

- DFA states: $\{\}, \{q_1\}, \{q_2\}, \{q_3\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\}, \{q_1, q_2, q_3\}$

Conversion Example: Building the Table



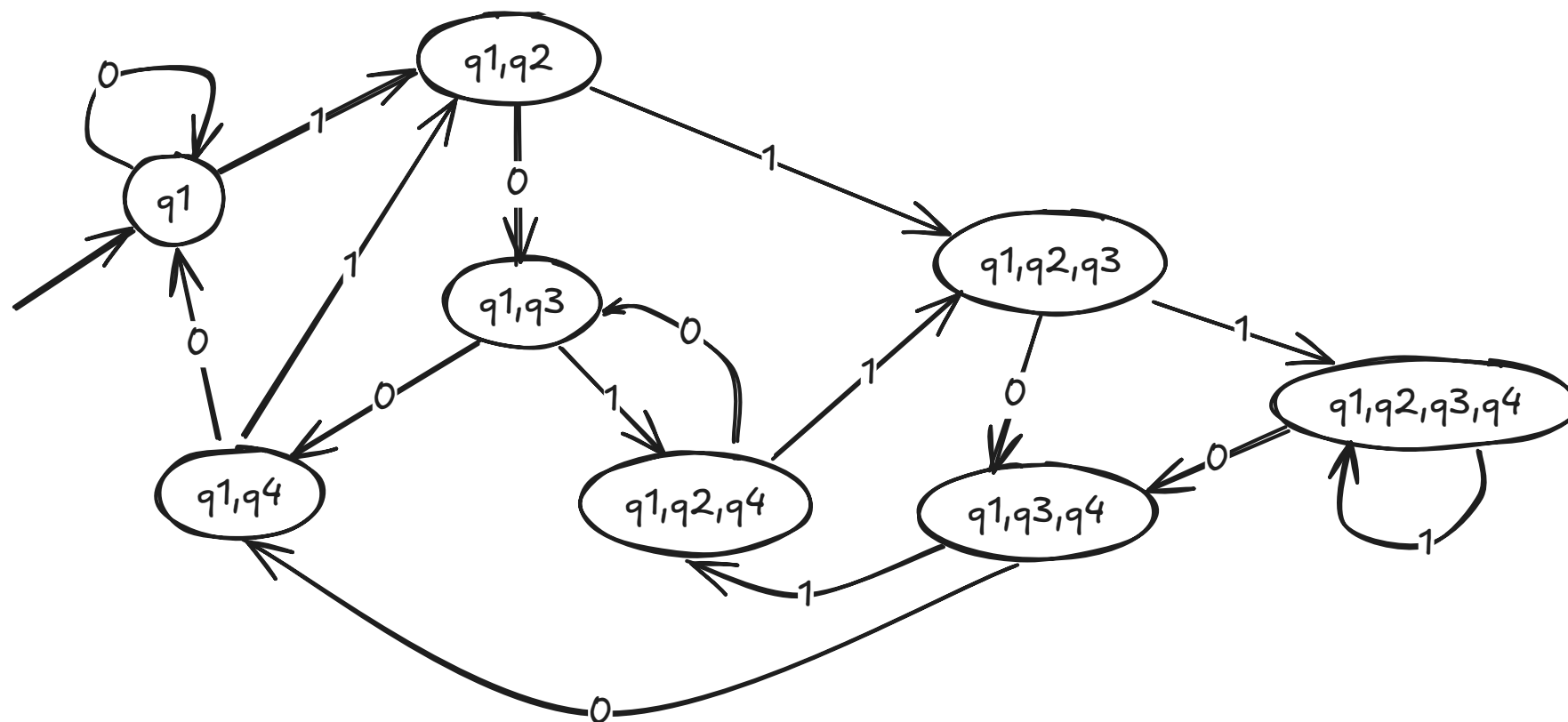
Let's convert this NFA to a DFA...

DFA State Transition Table

DFA State	0	1
{q1}	{q1}	{q1,q2}
{q1,q2}	{q1,q3}	{q1,q2,q3}
{q1,q3}	{q1,q4}	{q1,q2,q4}
{q1,q4}	{q1}	{q1,q2}

(Partial table shown - 8 more rows needed for complete DFA)

The Resulting DFA



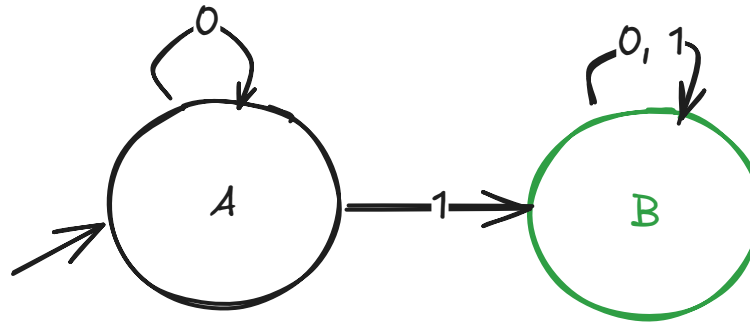
Which looks simpler to you?





Active Learning: Conversion Practice

Given this simple NFA:



1. Identify all subset states needed
2. Build first 3 rows of transition table
3. Which subsets are accept states?

Implementing NFAs in Java

Key API Difference from DFA

```
public class NFA {  
    public static class State {  
        void addTransition(Character symbol, State to) {...}  
  
        // Returns SET of states, not single state!  
        Set<State> getTransition(Character symbol) {...}  
    }  
  
    public boolean accepts(String input) {...}  
}
```

Implementation Strategy

Processing the i -th symbol:

1. **Before reading:** Check ϵ -reachable states

- Follow all null transitions
- Beware of cycles!

2. **After reading:** Follow all symbol transitions

- Explore all possible next states

Recommendation: Use recursion!

Supporting Null Transitions

Practical approach:

```
// Use special character for epsilon
char EPSILON = '\0';

// Add epsilon transitions
state1.addTransition(EPSILON, state2);

// Check epsilon transitions
Set<State> epsilonStates =
    currentState.getTransition(EPSILON);
```

Regular Languages

Definition

A formal language is called a **regular language** if some DFA or NFA recognizes it.

Key insight: Since $\text{DFA} \equiv \text{NFA}$

- Language regular if DFA recognizes it
- Language regular if NFA recognizes it
- Same expressive power!

Why Use NFAs Then?

If NFAs = DFAs in power, why bother?

Advantages of NFAs:

- **Simpler** to design
- **Fewer states** needed
- **More intuitive** for certain patterns
- **Easier** to combine (union, concatenation)

Trade-off:

- Harder to implement/simulate



Design Challenge

Your Turn: Design an NFA for:

"Binary strings containing the substring '101'"

1. Sketch your NFA
2. Compare with a partner
3. How many states did you use?

Would the DFA be simpler or more complex?

Key Takeaways

1. NFAs add two types of nondeterminism:
 - i. Multiple transitions per symbol
 - ii. Epsilon transitions
2. NFAs are equivalent to DFAs in power
3. NFAs often simpler to design but harder to simulate
4. Regular languages = Languages recognized by DFA or NFA

