🟦 Endliche Automaten (Finite Automata) – DFA & NFA

1️⃣ Grundidee

Ein **endlicher Automat** ist ein formales Modell zur Verarbeitung von Eingaben **Symbol für Symbol** mit **endlich vielen Zuständen**.

Typische Anwendungen:

- Mustererkennung / reguläre Ausdrücke (Regex)
- Lexer im Compilerbau
- Protokolle & Steuerlogik
- einfache Verifikation (zulässige Sequenzen)

2️⃣ Formale Definition

Ein endlicher Automat ist ein 5-Tupel:

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

- **Q**: endliche Menge von Zuständen
- **Σ**: Alphabet (Eingabesymbole)
- **δ**: Übergangsfunktion
- **q_0**: Startzustand
- **F**: Menge der akzeptierenden Zustände

Ein Wort w wird akzeptiert, wenn der Automat nach dem Einlesen von w in einem Zustand aus **F** endet.

3️⃣ DFA (Deterministischer endlicher Automat)

3.1 Definition (DFA)

Beim **DFA** gilt:

- Für jeden Zustand und jedes Symbol gibt es **genau einen** Folgezustand.

Formal:

$$\delta: Q \times \Sigma \to Q$$

3.2 Intuition

- Keine Wahlmöglichkeiten
- Der Automat läuft wie ein "Programm" mit eindeutigem nächsten Schritt

---

4️⃣ NFA (Nichtdeterministischer endlicher Automat)

4.1 Definition (NFA)

Beim **NFA** gilt:

- Für einen Zustand und ein Symbol kann es **0, 1 oder mehrere** Folgezustände geben.
- Zusätzlich können **ε-Übergänge** existieren (Zustandswechsel ohne Eingabesymbol).

Formal:

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$$

P(Q) = Potenzmenge von Q (Menge aller Zustandsmengen)

4.2 Intuition

- Der Automat kann "parallel" mehrere Wege ausprobieren
- Akzeptiert, wenn **mindestens ein Weg** zu einem akzeptierenden Zustand führt

---

5️⃣ Unterschiede DFA vs. NFA (prüfungsrelevant)

Merkmal	DFA	NFA
Übergang pro Symbol	genau 1	0..n
ε-Übergänge	nein	möglich
Simulation	einfach	über Zustandsmengen
Ausdrucksstärke	gleich	gleich
Umwandlung	–	NFA → DFA (Subset Construction)

Wichtig: DFA und NFA sind **gleich mächtig**: Beide erkennen exakt die **regulären Sprachen**. Ein NFA kann immer in einen DFA umgewandelt werden (evtl. mit mehr Zuständen).

6️⃣ Beispiel 1: DFA

Sprache

> Alle Binärwörter mit **gerader Anzahl von 1en**

Zustände:

- q0 = gerade Anzahl (Start, akzeptierend)
- q1 = ungerade Anzahl

Übergänge:

```
q0 --1--> q1
q1 --1--> q0
q0 --0--> q0
q1 --0--> q1
```

7️⃣ Beispiel 2: NFA

Sprache

> Alle Wörter über {a, b}, die **"ab" enthalten**

Idee:

- NFA kann an jeder Position entscheiden: “Starte jetzt das Matching für ab”
- Sobald "ab" gefunden wurde, bleibt der Automat im akzeptierenden Zustand

Zustände:

- s (Start)
- a_seen (ein 'a' gesehen, jetzt 'b' erwartet)
- accept (akzeptierend, "ab" gefunden)

Übergänge:

- s liest 'a' → kann bei s bleiben oder nach a_seen gehen (Nichtdeterminismus)
- a_seen liest 'b' → accept
- accept bleibt auf a/b in accept

8️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- **DFA zeichnen** und Wort simulieren ist Standard
- **NFA akzeptiert**, wenn mindestens ein Pfad akzeptiert
- ε-Übergänge: zuerst ε-Abschluss (epsilon-closure) bilden

- Häufige Frage: *Warum ist NFA oft kleiner als DFA?* → weil Nichtdeterminismus Zustände spart, der DFA aber alle Kombinationen abbilden muss

---

9️⃣ Vor- und Nachteile

DFA – Vorteile

- sehr einfach zu simulieren
- effizient in der Ausführung (ein Zustand)

DFA – Nachteile

- kann deutlich mehr Zustände benötigen als ein NFA

NFA – Vorteile

- oft kompakter und leichter zu konstruieren
- nahe an Regex-Logik

NFA – Nachteile

- Simulation benötigt Zustandsmengen (mehr Aufwand)
- ε-Übergänge erhöhen Komplexität

🧠 Merksätze für die Prüfung

- *DFA: pro Symbol genau ein Übergang – eindeutiger Lauf.*
- *NFA: mehrere mögliche Läufe – akzeptiert, wenn mindestens einer akzeptiert.*
- *Beide erkennen reguläre Sprachen; NFA kann in DFA umgewandelt werden.*

🔟 Python-Implementierung: DFA

In [2]:
```python
class DFA:
    def __init__(self, transitions, start_state, accept_states):
        # transitions: dict {(state, symbol): next_state}
        self.transitions = transitions
        self.start_state = start_state
        self.accept_states = set(accept_states)

    def accepts(self, word):
        state = self.start_state
        for symbol in word:
            state = self.transitions[(state, symbol)]
```

```
        return state in self.accept_states


# Beispiel-DFA: gerade Anzahl von 1en
transitions = {
    ("q0", "0"): "q0",
    ("q0", "1"): "q1",
    ("q1", "0"): "q1",
    ("q1", "1"): "q0",
}
dfa = DFA(transitions, start_state="q0", accept_states={"q0"})

print(dfa.accepts("1010"))  # True
print(dfa.accepts("111"))   # False
```

```
True
False
```

1️⃣ 1️⃣ Python-Implementierung: NFA (ohne ε-Übergänge)

In [ ]:

```
class NFA:
    def __init__(self, transitions, start_state, accept_states):
        # transitions: dict {(state, symbol): set(next_states)}
        self.transitions = transitions
        self.start_state = start_state
        self.accept_states = set(accept_states)

    def accepts(self, word):
        current_states = {self.start_state}

        for symbol in word:
            next_states = set()
            for state in current_states:
                next_states |= self.transitions.get((state, symbol), set())
            current_states = next_states

            if not current_states:
                return False

        return any(state in self.accept_states for state in current_states)


# Beispiel-NFA: Wörter, die "ab" enthalten
transitions = {
    ("s", "a"): {"s", "a_seen"},  # nichtdeterministisch: bei s bleiben oder starte
    ("s", "b"): {"s"},
    ("a_seen", "b"): {"accept"},
    ("a_seen", "a"): {"a_seen"},  # optional: bleibt im a_seen bei 'a' (robuster)
    ("accept", "a"): {"accept"},
    ("accept", "b"): {"accept"},
}
```

```
nfa = NFA(transitions, start_state="s", accept_states={"accept"})

print(nfa.accepts("bbbb"))      # False
print(nfa.accepts("aaab"))      # True  (enthält "ab" am Ende)
print(nfa.accepts("baba"))      # True  (enthält "ab")
print(nfa.accepts("aaaa"))      # False
```