Musterlösungen

May 31, 2025

Lösung zu Aufgabe 1: Analyse eines Algorithmus

Gegebener Java-Code

```
public class Algorithmus {
 public static int berechne(int n) {
  if (n <= 1) {
   return n;
  }
  return berechne(n - 1) + berechne(n - 2);
 }

 public static void main(String[] args) {
  int n = 5;
  System.out.println("Ergebnis: " + berechne(n));
 }
}
```

1. Funktionsweise und mathematische Bedeutung

Der Algorithmus berechnet die n-te Zahl der Fibonacci-Folge. Diese ist rekursiv definiert durch:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2)$$

Die Methode `berechne(n)` ist eine direkte rekursive Umsetzung dieser Definition.

2. Art der Implementierung

Es handelt sich um eine **rekursive Implementierung**. Vorteilhaft ist, dass sie sehr elegant ist und der mathematischen Definition entspricht. Sie ist jedoch ineffizient für große n, da viele Zwischenwerte mehrfach berechnet werden.

3. Beispiel: Aufruf `berechne(5)`

Die rekursive Struktur lässt sich als Baum darstellen:

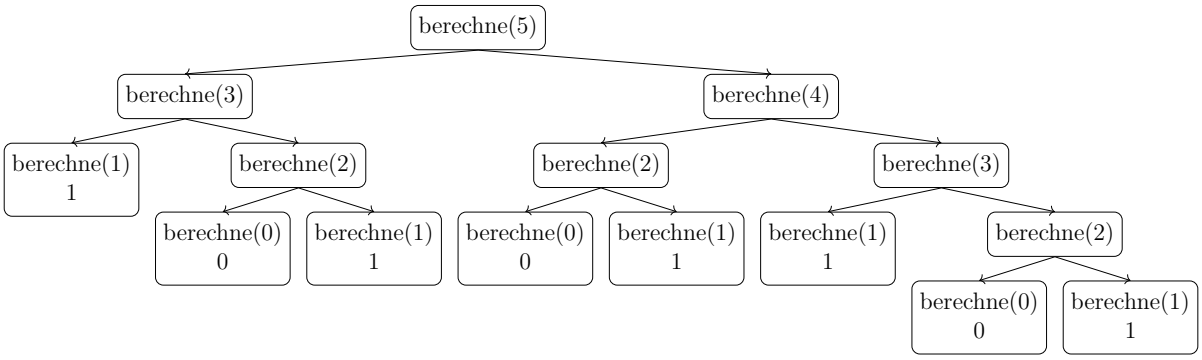

Figure 1: Rekursionsbaum für `berechne(5)`

Am Ende ergibt sich:

$$\texttt{berechne(5)} = \texttt{berechne(4)} + \texttt{berechne(3)} = 3 + 2 = 5$$

4. Anzahl der Aufrufe für `berechne(5)`

Durch Zählen aller Knoten im Baum ergibt sich:

$$\texttt{berechne(5)} \Rightarrow 15 \text{ rekursive Aufrufe}$$

5. Zeitkomplexität

Die Zeitkomplexität dieser rekursiven Lösung ist exponentiell:

$$T(n) = T(n-1) + T(n-2) + \mathcal{O}(1) \Rightarrow \mathcal{O}(2^n)$$

Grund: Jeder Aufruf erzeugt zwei weitere.

6. Vergleich mit iterativer Lösung

Eine iterative Lösung speichert Zwischenwerte und vermeidet doppelte Berechnungen:

Listing 1: Iterative Berechnung

```
public static int fibonacciIterativ(int n) {
        if (n <= 1) return n;
        int a = 0, b = 1;
        for (int i = 2; i <= n; i++) {
                int temp = a + b;
                a = b;
                b = temp;
        }
        return b;
}
```

Vorteil: Nur $\mathcal{O}(n)$ Zeit und konstanter Speicherverbrauch.

7. Optimierungsmöglichkeiten

- **Memoization (Top-Down)**: Zwischenergebnisse werden gespeichert $\rightarrow \mathcal{O}(n)$ Zeit, rekursiv.
- **Bottom-Up (Iteration)**: Noch effizienter und speichersparend $\rightarrow \mathcal{O}(n)$ Zeit.
- **Dynamische Programmierung**: Allgemeiner Ansatz bei rekursiven Problemen mit überlappenden Teilproblemen.

8. Zusammenfassung

Variante	**Zeit**	**Speicher**	**Bemerkung**
Rekursiv (naiv)	$\mathcal{O}(2^n)$	Hoch (Call Stack)	Einfach, aber ineffizient
Iterativ	$\mathcal{O}(n)$	Gering	Schnell, optimal für große n
Rekursiv + Memo	$\mathcal{O}(n)$	Mittel	Elegant und effizient

Aufgabe 2: Datenmodellierung und Normalisierung

Eine Schule speichert die Noten der Schüler in einer relationalen Datenbank. Die ursprüngliche Tabellenstruktur ist wie folgt:

Table 1: Ursprüngliche nicht normalisierte Tabelle

SchülerID	**Name**	**Klasse**	**Fach**	**Lehrer**	**Note**
101	Max Meier	10A	Mathematik	Herr Schmidt	2
102	Lisa Becker	10A	Deutsch	Frau Müller	1
101	Max Meier	10A	Deutsch	Frau Müller	3
103	Anna Keller	10B	Englisch	Herr Weber	2
102	Lisa Becker	10A	Mathematik	Herr Schmidt	2

1. Redundanzen in der Tabelle

- Der Schülername 'Max Meier' taucht mehrfach auf.
- Die Klasse '10A' wird mehrfach gespeichert.
- Lehrer wie 'Herr Schmidt' erscheinen mehrfach mit identischem Fach.
- Fach-Lehrer-Zuordnungen sind redundant.

Dies führt zu höherem Speicherbedarf und potenzieller Fehleranfälligkeit (z.B. Tippfehler).

2. Normalisierung bis zur 3. Normalform (3NF)

1NF: Bereits erfüllt, da alle Werte atomar sind.
2NF: Es gibt partielle Abhängigkeiten (z.B. Name hängt nur von SchülerID ab). Daher Zerlegung in kleinere Tabellen:

Tabelle: `Schueler`

SchülerID	Name	Klasse
101	Max Meier	10A
102	Lisa Becker	10A
103	Anna Keller	10B

Tabelle: `Lehrer`

LehrerID	Name
L01	Herr Schmidt
L02	Frau Müller
L03	Herr Weber

Tabelle: `Fach`

FachID	Fachname	LehrerID
F01	Mathematik	L01
F02	Deutsch	L02
F03	Englisch	L03

Tabelle: `Noten`

SchülerID	FachID	Note
101	F01	2
102	F02	1
101	F02	3
103	F03	2
102	F01	2

3NF: Erfüllt, da keine transitiven Abhängigkeiten bestehen.

3. Vorteile der Normalisierung

- **Weniger Redundanz:** z.B. Lehrername nur einmal gespeichert
- **Bessere Datenintegrität:** Konsistente und korrekte Daten
- **Pflegeleichter:** Änderungen müssen nur an einer Stelle erfolgen
- **Skalierbarkeit:** Neue Fächer, Lehrer oder Schüler lassen sich leicht ergänzen
- **Fehlervermeidung:** Reduzierung von Tippfehlern und Inkonsistenzen

Aufgabe 3: Analyse einer formalen Grammatik

Gegeben sei die Grammatik G mit:

- **Nichtterminale:** $\{S, X, Y\}$
- **Terminale:** $\{0, 1\}$
- **Produktionsregeln:**

$$\begin{aligned} S &\to 0X1 \\ X &\to 0X1 \mid 1Y0 \\ Y &\to 1Y0 \mid \epsilon \end{aligned}$$

- **Startsymbol:** S

Sprachanalyse

Die Grammatik G beschreibt Wörter der Form:

$$w = 0^n\, 1^m\, 1^k\, 0^m\, 1^n \quad \text{mit } n \geq 1,\ m \geq 0,\ k \geq 0$$

Diese Struktur ergibt sich aus:

- jeder Anwendung von $X \to 0X1$ erzeugt symmetrisch äußere 0–1-Paare (Zähler n)
- der abschließenden Regel $X \to 1Y0$, die einmalig vorkommen muss
- beliebiger Anzahl von $Y \to 1Y0$, die zentrale Mitte bildet (Zähler m)
- $Y \to \epsilon$ schließt die Mitte ab

Überprüfung der Wörter

Wort 1: `000111000111`

- Länge: 12
- Struktur: 000 111 000 111
- Es liegt eine gespiegelte Struktur vor:

$$\texttt{000}\,\texttt{1}\underline{\texttt{11}}\texttt{0}\,\texttt{111}$$

- Zerlegung möglich in:

$$S \Rightarrow 0\,X\,1X \Rightarrow 0\,X\,1X \Rightarrow 0\,X\,1X \Rightarrow 1\,Y\,0Y \Rightarrow 1\,Y\,0Y \Rightarrow \epsilon$$

- Daraus ergibt sich:

$$0\,0\,0\,1\,1\,1\,0\,0\,0\,1\,1\,1 = \texttt{000111000111}$$

Fazit: Das Wort `000111000111` gehört zur Sprache $L(G)$.

Wort 2: `000100011000`

- Länge: 12
- Struktur nicht symmetrisch: Mittelteil ist nicht durch Y erklärbar
- Keine passende Ableitung der Form $0^n\,1^m\,1^k\,0^m\,1^n$
- Beispiel: die Mitte `100011` verletzt das symmetrische Muster

Fazit: Das Wort `000100011000` gehört **nicht** zur Sprache $L(G)$.

Teilfragen zur Grammatik

1. **Ableitung von `101`:** Es gilt:

$$S \Rightarrow 0X1 \quad X \Rightarrow 1Y0 \quad Y \Rightarrow \epsilon$$

$$\Rightarrow 0\,1\,0\,1 \neq \texttt{101}$$

   `101` kann **nicht** abgeleitet werden, da jedes Wort mit `0` beginnt und mit `1` endet. **Fazit:** `101` gehört **nicht** zur Sprache $L(G)$.

2. **Kann die Grammatik alle Binärzahlen erzeugen?** Nein, es können nur Wörter mit einer ganz bestimmten symmetrischen Struktur abgeleitet werden. Die Grammatik erzeugt genau die Sprache:

$$L(G) = \left\{0^n\,1^m\,1^k\,0^m\,1^n \mid n \geq 1, m \geq 0, k \geq 0\right\}$$

   Anpassungsidee: Um beliebige Binärzahlen zu erzeugen:

$$S \to 0S \mid 1S \mid \epsilon$$

3. **Erweiterung für nur Wörter mit gerader Länge:** Idee: immer zwei Zeichen pro Ableitung erzeugen:

$$S \to 00S \mid 01S \mid 10S \mid 11S \mid \epsilon$$

   Jede Ableitung erzeugt exakt zwei Zeichen → Nur Wörter mit gerader Länge werden erzeugt.

Kolloquium-Antworten

Algorithmen

1. Welche Eigenschaften muss ein Algorithmus haben?
Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems. Er muss folgende Eigenschaften besitzen:

- Finitheit (Endlichkeit): Der Algorithmus muss nach einer endlichen Anzahl von Schritten terminieren.
- Eindeutigkeit (Determinismus): Jeder Schritt des Algorithmus muss klar definiert und eindeutig sein.
- Ausführbarkeit: Jeder Schritt muss tatsächlich ausführbar sein (z.B. mit einem Computer).
- Eingabe: Ein Algorithmus besitzt null oder mehr Eingabewerte, auf denen er operiert.
- Ausgabe: Ein Algorithmus liefert mindestens eine Ausgabe, also ein Ergebnis.
- Effektivität: Jeder Schritt muss in endlicher Zeit mit den gegebenen Ressourcen ausführbar sein.

2. Warum ist die iterative Lösung oft effizienter als eine rekursive Lösung?

- Speicherverbrauch: Iterative Lösungen benötigen in der Regel weniger Speicher, da sie keine zusätzliche Speicherstruktur wie den Call-Stack (für Funktionsaufrufe) beanspruchen
- Overhead: Jeder rekursive Funktionsaufruf erzeugt Overhead durch das Ablegen von Rücksprungadressen und lokalen Variablen im Stack.

```
def factorial(n):
if n == 0:
return 1
return n * factorial(n - 1)
```

```
factorial(3)
→ 3 * factorial(2)
→ 3 * (2 * factorial(1))
→ 3 * (2 * (1 * factorial(0)))
→ 3 * (2 * (1 * 1))
→ 6
```

Stack-Aufbau (von oben nach unten)

```
factorial(3)   → wartet auf Ergebnis von factorial(2)
factorial(2)   → wartet auf Ergebnis von factorial(1)
factorial(1)   → wartet auf Ergebnis von factorial(0)
factorial(0)   → gibt 1 zurück (Basisfall)
```

Rückweg / Stack wird abgearbeitet

```
factorial(0) → gibt 1 zurück
factorial(1) → 1 * 1 = 1
factorial(2) → 2 * 1 = 2
factorial(3) → 3 * 2 = 6
```

- Ausführungszeit: Iterationen sind oft schneller, da sie keine wiederholten Funktionsaufrufe verursachen.
- Begrenzung: Rekursion kann z.B. aufgrund einer Sematikfehlers zum Stack Overflow führen, wenn zu viele Aufrufe ineinander verschachtelt werden.

```
def factorial(n):
if n == 0:
return 1
return n * factorial(n + 1)
```

3. In welchen Fällen könnte Rekursion einer Iteration vorzuziehen sein?

- Eleganz und Lesbarkeit: Bei Problemen mit natürlicher rekursiver Struktur (z.B. Divide-and-Conquer-Strategien bei Margesort oder Quicksort, Fakultät, Fobonacci) ist Rekursion oft intuitiver.
- Konzeptualisierung und Umsetzung vereinfachen: Auch wenn sie ggf. ineffizienter ist.

4. Was versteht man unter einem effizienten Algorithmus? Welche Maßstäbe werden zur Effizienzbewertung verwendet?

Ein effizienter Algorithmus liefert die korrekte Lösung in möglichst kurzer Zeit und mit minimalem Ressourcenverbrauch.
Maßstäbe zur Effizienzbewertung:

- Zeitkomplexität: Wie verändert sich die Rechenzeit in Abhängigkeit von der Eingabegröße nn?
- Platzkomplexität (Speicherverbrauch): Wie viel zusätzlicher Speicher wird benötigt?
- Best-, Average- und Worst-Case-Verhalten

Ziel ist, möglichst geringe Wachstumsraten bei zunehmender Eingabegröße zu erreichen.

5. Wie analysiert man die Zeitkomplexität eines Algorithmus?

- Zähle die Anzahl der elementaren Operationen (z.B. Vergleiche, Zuweisungen) als Funktion von n.

- Identifiziere Schleifen, rekursive Aufrufe, bedingte Anweisungen.

Konstrukt	**Typischer Zeitaufwand**
Einfache Schleife über n	$\mathcal{O}(n)$
Zwei verschachtelte Schleifen	$\mathcal{O}(n^2)$
Drei verschachtelte Schleifen	$\mathcal{O}(n^3)$
Rekursion mit einem Aufruf pro Ebene	$\mathcal{O}(n)$
Rekursion mit zwei rekursiven Aufrufen	$\mathcal{O}(2^n)$
Binäre Suche	$\mathcal{O}(\log n)$
Schleife halbiert sich pro Schritt ($i = i//2$)	$\mathcal{O}(\log n)$
Konstante Operation (z. B. $x = a + b$)	$\mathcal{O}(1)$

Table 2: Faustregeln zur Abschätzung der Zeitkomplexität

Beispiel: Zwei verschachtelte Schleifen $O(n^2)$

```
for i in range(n):
for j in range(n):
print(i + j)
```

Beispiel: Gesucht wird eine Zahl zwischen 1-1000 durch das Raten? $2^k > 1000$

$$\begin{aligned}
\text{Produktregel:} \quad & \log_b(x \cdot y) = \log_b(x) + \log_b(y) \\
\text{Quotientenregel:} \quad & \log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y) \\
\text{Potenzregel:} \quad & \log_b(x^a) = a \cdot \log_b(x) \\
\text{Wurzelregel:} \quad & \log_b\left(\sqrt[n]{x}\right) = \frac{1}{n} \cdot \log_b(x) \\
\text{Basiswechsel:} \quad & \log_b(x) = \frac{\log_k(x)}{\log_k(b)}
\end{aligned}$$

$$\log_2(1000) = \frac{log_{10}(1000)}{log_{10}(2)} = \frac{3}{0,3010} = 9,97$$

- Nutze asymptotische Notation ($Big-O$, $Big-\Omega$, $Big-\Theta$) zur Klassifikation des Wachstums.

6. Warum ist $O(n\ log(n))$ schneller als $O(n^2)$?

Warum ist $\mathcal{O}(n \log n)$ schneller als $\mathcal{O}(n^2)$?

Um das Wachstum zweier Funktionen zu vergleichen, bildet man ihr Verhältnis:

$$\frac{n \log n}{n^2} = \frac{\log n}{n}$$

Für $n \to \infty$ gilt:

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

Das bedeutet:

- $n \log n$ wächst viel langsamer als n^2
- Deshalb ist $\mathcal{O}(n \log n) \subset \mathcal{O}(n^2)$

Beispielhafte Werte:

n	$\log_2(n)$	$n \log_2(n)$	n^2
10	≈ 3.3	≈ 33	100
100	≈ 6.6	≈ 660	10 000
1000	≈ 10	$\approx 10\,000$	1 000 000

Fazit: Ein Algorithmus mit Laufzeit $\mathcal{O}(n \log n)$ ist für große n deutlich effizienter als einer mit $\mathcal{O}(n^2)$.

7. Was ist der Unterschied zwischen exponentiellen $O(2^n)$ und polynomiellen $O(n^k)$ Algorithmen?

Polynomielle Laufzeit: $\mathcal{O}(n^k)$, wobei k eine feste Konstante ist.

- Die Anzahl der Schritte wächst gemäß eines Polynoms in n
- Beispiele: $O(n)$, $O(n^2)$, $O(n^3)$
- Auch für große Eingaben praktikabel

Exponentielle Laufzeit: $\mathcal{O}(2^n)$, $\mathcal{O}(3^n)$, $\mathcal{O}(n!)$

- Die Anzahl der Schritte verdoppelt oder vervielfacht sich mit jedem zusätzlichen Eingabeelement
- Wächst extrem schnell – unpraktisch für große n
- Beispiel: Brute-Force für das Rucksackproblem, naive Fibonacci-Berechnung

Wachstumsvergleich:

n	n^2	n^3	2^n
5	25	125	32
10	100	1000	1024
20	400	8000	1 048 576
50	2 500	125 000	$\sim 10^{15}$
100	10 000	1 000 000	$\sim 10^{30}$

Fazit: Polynomielle Algorithmen sind deutlich effizienter und für große Eingaben geeignet.
Exponentielle Algorithmen explodieren im Aufwand und sind nur für kleine Eingaben sinnvoll.

8. Wie kann man den Speicherverbrauch eines Algorithmus reduzieren?

- **In-Place-Operationen:** Verwende vorhandene Speicherplätze, anstatt neue Datenstrukturen anzulegen.
  - Beispiel: `list.sort()` statt `sorted(list)`
- **Vermeidung unnötiger Datenstrukturen:** Nutze nur das, was wirklich benötigt wird (z. B. kein *array* wenn auch *strig* geht).
- **Iteration statt Rekursion:** Rekursive Aufrufe verbrauchen Stack-Speicher.
  - Beispiel: Iterative Fibonacci-Version spart Speicher gegenüber rekursiver.
- **Nicht mehr benötigte Daten löschen:** Mit `del` oder durch Überschreiben Speicher freigeben. In Java über übernimmt diese Funktion der sog. Garbage Collectors (GC)
- **Effizientere Datenrepräsentation:** Nutze kompakte Datentypen. Z.B. *int* statt *double*.
- **Teile-und-Herrsche optimieren:** Achte bei rekursiven Algorithmen darauf, keine überflüssigen Kopien zu erzeugen.

Typische Speicherkomplexitäten:

Struktur / Technik	**Speicherbedarf**
einfache Schleife	$O(1)$
Liste mit n Elementen	$O(n)$
Rekursion (Call-Stack)	$O(n)$
Memoisierung / DP-Tabellen	$O(n)$ oder $O(n^2)$

Fazit: Durch gezielte Optimierungen wie In-Place-Verarbeitung, Generatoren und iterative Verfahren kann der Speicherverbrauch erheblich reduziert werden.

9. Schreiben Sie einen Algorithmus in Pseudo-Code, der zwei Variablen ohne eine zusätzliche Variable tauscht.

```
Eingabe: a, b (zwei Zahlen)
a := a + b
b := a - b
a := a - b
Ausgabe: a, b (getauscht)
```

Beispiel: $a = 5$ und $b = 9$

```
a = 5, b = 9

1. a = a + b = 5 + 9 = 14
2. b = a - b = 14 - 9 = 5
3. a = a - b = 14 - 5 = 9
```

10. Können Sie eine weitere Möglichkeit, denselben Algorithmus mit Hilfe von boolescher Algebra zu implementieren?

```
Eingabe: a, b (zwei Ganzzahlen)
a := a XOR b
b := a XOR b
a := a XOR b
Ausgabe: a, b (getauscht)
```

Beispiel: $5_2 = 0101$ und $9_2 = 1001$

```
a = 5 (0101)
b = 9 (1001)

1. a = 0101 XOR 1001 = 1100 → 12
2. b = 1100 XOR 1001 = 0101 → 5
3. a = 1100 XOR 0101 = 1001 → 9
```

Datenbanken

1. Warum ist die Normalisierung wichtig für eine relationale Datenbank?
2. Wie lauten die Definitionen von drei Normalformen?

   1. Normalform (1NF):

   Eine Relation befindet sich in der **1. Normalform**, wenn:

   - alle Attributwerte atomar (unteilbar) sind
   - keine Wiederholungsgruppen (z. B. mehrere Noten in einer Spalte) vorhanden sind

Beispiel: Eine Spalte `Noten` mit dem Wert `2,3,1` wäre **nicht 1NF-konform**. Korrekt wäre eine Zeile pro Note.

2. Normalform (2NF):

Eine Relation befindet sich in der **2. Normalform**, wenn:

- sie in 1NF ist
- jedes Nichtschlüsselattribut voll funktional abhängig vom gesamten Primärschlüssel ist

Hinweis: Gilt nur bei zusammengesetzten Primärschlüsseln.

Beispiel: Ist der Primärschlüssel `(SchuelerID, FachID)` und das Attribut `Name` hängt nur von `SchuelerID` ab, liegt eine **partielle Abhängigkeit** vor → Verstoß gegen 2NF.

3. Normalform (3NF):

Eine Relation befindet sich in der **3. Normalform**, wenn:

- sie in 2NF ist
- kein Nichtschlüsselattribut transitiv von einem Schlüssel abhängt

Transitive Abhängigkeit: Wenn $A \rightarrow B$ und $B \rightarrow C$, dann ist C transitiv abhängig von A

Beispiel: Wenn `SchuelerID` → `Klasse` und `Klasse` → `Raum`, dann ist `Raum` transitiv abhängig von `SchuelerID` → Verstoß gegen 3NF.

Fazit: Die Normalformen dienen dazu, Redundanzen zu vermeiden, die Datenstruktur zu vereinfachen und Anomalien zu verhindern.

3. Warum musste die ursprüngliche Tabelle normalisiert werden? Welche Probleme hätte es gegeben, wenn man sie in nicht normalisierter Form belassen hätte?

4. Gibt es in der endgültigen normalisierten Form noch Redundanzen? Falls ja, sind diese gewollt?

   Ja, auch in der vollständig normalisierten Form (z. B. 3. Normalform oder Boyce-Codd-Normalform) können noch Redundanzen vorkommen. Diese sind jedoch in der Regel **gewollt und funktional notwendig**.

   - In relationalen Datenbanken werden Beziehungen durch **Fremdschlüssel** dargestellt.
   - Ein Fremdschlüssel (z. B. `SchuelerID`) kann in einer anderen Tabelle mehrfach vorkommen (z. B. in der Notentabelle).
   - Diese Wiederholungen sind keine ungewollte Redundanz, sondern **essentiell**, um Relationen auszudrücken.

2. Kontrollierte Redundanz zur Performanceoptimierung

- In manchen Fällen werden gezielt zusätzliche Redundanzen eingeführt, um die Performance zu verbessern.
- Dies geschieht bewusst im Rahmen einer **Denormalisierung**.
- Beispiel: Ein häufig verwendeter Wert wird zusätzlich in einer Tabelle gespeichert, um teure Joins zu vermeiden.

Auch wenn eine Datenbank korrekt bis zur 3. Normalform (3NF) normalisiert ist, kann es in bestimmten Situationen sinnvoll sein, gezielt **kontrollierte Redundanz** einzuführen – zum Beispiel zur Performanceverbesserung oder zur leichteren Berichtserstellung.

Beispiel: Schulsystem

Normalisierte Struktur:

- **Schueler:**

SchuelerID	Name	Klasse
101	Max Meier	10A

- **Lehrer:**

LehrerID	Name
L01	Herr Müller

- **Fach:**

FachID	Bezeichnung	LehrerID
F01	Mathematik	L01

- **Noten:**

SchuelerID	FachID	Note
101	F01	2

Diese Struktur ist vollständig normalisiert und vermeidet Redundanz.

Kontrollierte Redundanz – gezielte Erweiterung

Fall 1: Lehrername zusätzlich in der Noten-Tabelle speichern

SchuelerID	FachID	Note	Lehrername
101	F01	2	Herr Müller

Vorteil: Lehrername ist direkt verfügbar, ohne dass ein JOIN mit der Lehrer- und Fach-Tabelle erforderlich ist.

Fall 2: Klasse direkt in der Noten-Tabelle speichern

SchuelerID	FachID	Note	Klasse
101	F01	2	10A

Vorteil: Schnellere Filterung nach Klassen, z. B. zur Anzeige aller Noten der Klasse 10A, ohne JOIN mit der Schueler-Tabelle.

Fazit:

Kontrollierte Redundanz bedeutet, dass Daten bewusst mehrfach gespeichert werden – z. B. zur Steigerung der Abfragegeschwindigkeit oder zur einfacheren Berichterstellung. Diese Redundanz ist zulässig, wenn sie gezielt eingesetzt und technisch (z. B. durch Trigger) gepflegt wird.
In einer korrekt normalisierten Datenbankstruktur sind alle verbliebenen Redundanzen **gewollt und notwendig**. Sie unterstützen die Darstellung von Beziehungen (Fremdschlüssel) oder dienen der Effizienzsteigerung. **Unnötige Redundanzen**, die zu Anomalien führen können, werden durch die Normalisierung entfernt.

5. Welche konkreten Redundanzen wurden durch die Normalisierung beseitigt?
6. Welche Probleme können in einer nicht normalisierten Datenbank auftreten?

In einer nicht normalisierten Datenbankstruktur treten verschiedene Probleme auf, die die Datenintegrität und Wartbarkeit beeinträchtigen. Im Folgenden sind die typischen Anomalien beschrieben:

1. Redundanz (Datenwiederholung)

- Gleiche Informationen werden mehrfach gespeichert, z. B. der Name eines Lehrers in jeder Zeile der Notentabelle.
- **Folgen:** unnötiger Speicherverbrauch, potenzielle Dateninkonsistenzen.

2. Änderungsanomalie (Update-Anomalie)

- Eine Änderung muss an mehreren Stellen gleichzeitig durchgeführt werden.
- Beispiel: Der Name eines Lehrers ändert sich. Wird er nur an einer Stelle aktualisiert, entsteht Inkonsistenz.

3. Einfügeanomalie (Insert-Anomalie)

- Neue Daten können nicht gespeichert werden, weil andere, abhängige Daten fehlen.
- Beispiel: Ein neues Fach soll erfasst werden, aber es gibt noch keinen Schüler mit einer Note darin → Einfügen nicht möglich.

4. Löschanomalie (Delete-Anomalie)

- Beim Löschen eines Datensatzes gehen ungewollt auch andere, wichtige Informationen verloren.
- Beispiel: Wenn der letzte Schüler mit dem Fach Informatik gelöscht wird, verschwindet auch die Information, dass dieses Fach überhaupt existiert.

5. Fehlende Datenintegrität

- Es gibt keine Zwangsregeln für gültige Datenbeziehungen.
- Beispiel: Ein Lehrername wird eingegeben, obwohl dieser Lehrer in keiner Lehrerliste erfasst ist.

Fazit

In nicht normalisierten Datenbanken treten Redundanzen, Inkonsistenzen sowie Einfüge-, Änderungs- und Löschanomalien auf. Diese führen zu schlechter Wartbarkeit und einer hohen Fehleranfälligkeit. Die Normalisierung schafft klare Strukturen und verhindert solche Probleme.

7. Gibt es Fälle, in denen man bewusst auf eine vollständige Normalisierung verzichtet? Warum?

Ja, es gibt in der Praxis Fälle, in denen man bewusst auf eine vollständige Normalisierung verzichtet. Dieses Vorgehen wird als **Denormalisierung** bezeichnet.

Gründe für gezielte Denormalisierung

(a) **Performance-Optimierung bei Abfragen**
- JOINs zwischen vielen Tabellen können bei großen Datenmengen die Abfrage verlangsamen.
- Durch kontrollierte Redundanz können Daten direkt in einer Tabelle bereitgestellt werden.
- Beispiel: Lehrername zusätzlich in der Noten-Tabelle speichern, um JOINs zu vermeiden.

(b) **Einfachere Datenanalyse und Reporting**
- Analyse-Tools oder Excel arbeiten effizienter mit flachen Tabellen.
- Redundante Informationen vereinfachen Export und Auswertung.

(c) **Bessere Lesbarkeit für Entwickler oder Benutzer**
- Eine leicht redundante Tabellenstruktur ist für bestimmte Zielgruppen einfacher zu verstehen.
- Besonders bei kleinen Projekten oder Einzeltabellen ist das akzeptabel.

(d) **Weniger komplexe SQL-Abfragen**
- Statt komplexer Abfragen mit mehreren JOINs sind direkte Abfragen möglich.
- Dies reduziert Entwicklungsaufwand und Fehleranfälligkeit.

(e) **Caching und Historisierung**
- In historischen Daten (z. B. Notenarchiv) soll der damalige Lehrername erhalten bleiben, auch wenn sich der Lehrer ändert.
- Deshalb wird der Name zusätzlich gespeichert.

Fazit

In bestimmten Szenarien wird bewusst auf eine vollständige Normalisierung verzichtet, z. B. zur Performanceverbesserung, einfacheren Datenanalyse oder Datenarchivierung. Diese **Denormalisierung** ist zulässig, wenn sie gezielt geplant und technisch kontrolliert erfolgt.

8. Welche Normalform ist für den praktischen Einsatz am besten geeignet?

In der Praxis hat sich die **3. Normalform (3NF)** als am besten geeignet erwiesen. Sie bietet einen guten Kompromiss zwischen **Redundanzfreiheit**, **Datenintegrität** und **praktischer Anwendbarkeit**.

Vergleich der Normalformen

Normalform	**Eigenschaften**	**Praxis-Eignung**
1NF	Atomare Werte, keine Wiederholungsgruppen	Grundvoraussetzung jeder relationalen Datenbank
2NF	Keine partielle Abhängigkeit von zusammengesetzten Schlüsseln	Nur relevant bei zusammengesetzten Primärschlüsseln
3NF	Keine transitive Abhängigkeit von Nichtschlüsselattributen	**Sehr gut geeignet: Datenkonsistenz + überschaubare Struktur**
BCNF	Strengere Form der 3NF	In Spezialfällen notwendig, teilweise zu streng
4NF / 5NF	Behandeln mehrwertige und join-unabhängige Abhängigkeiten	Selten in Praxis, eher theoretisch relevant

Vorteile der 3NF in der Praxis

- Verhindert Redundanz und Anomalien (z. B. durch transitive Abhängigkeiten)
- Erzeugt übersichtliche Tabellenstrukturen
- Ist effizient genug für typische Anwendungen
- Wird von ORMs, Frameworks und SQL-Werkzeugen direkt unterstützt

Fazit

Die **3. Normalform (3NF)** ist in der Praxis die am häufigsten verwendete Normalform. Sie bietet eine ausgewogene Balance zwischen theoretischer Datenbankstruktur und praktischer Effizienz. Höhere Normalformen (z. B. BCNF) werden nur bei speziellen Anforderungen benötigt.

9. Gegeben sei eine relationale Datenbank mit folgenden Tabellen:

- `Schueler(SchuelerID, Name, Klasse)`
- `Lehrer(LehrerID, Name)`
- `Fach(FachID, Fachname, LehrerID)`
- `Noten(SchuelerID, FachID, Note)`

Formulieren Sie SQL-Abfragen für folgende Aufgaben:

(a) Geben Sie den Namen aller Schüler aus, die in einem beliebigen Fach die Note `1` erhalten haben.

(b) Geben Sie zu jedem Noteneintrag den Namen des Schülers, den Fachnamen und den Namen des zugehörigen Lehrers aus.

(c) Ermitteln Sie für jede Klasse den Durchschnitt der Noten im Fach `Mathematik`.

Lösungen: SQL-Abfragen zur Schüler–Lehrer–Noten–Datenbank

Gegeben: Die folgenden Tabellen stehen zur Verfügung:

- `Schueler(SchuelerID, Name, Klasse)`
- `Lehrer(LehrerID, Name)`
- `Fach(FachID, Fachname, LehrerID)`
- `Noten(SchuelerID, FachID, Note)`

1. Schüler mit der Note 1

```
SELECT s.Name
FROM Schueler s
JOIN Noten n ON s.SchuelerID = n.SchuelerID
WHERE n.Note = 1;
```

Erläuterung: Verknüpfung von `Schueler` und `Noten`, Filterung nach Note = 1.

2. Schülername, Fachname und Lehrername zu jeder Note

```
SELECT s.Name AS Schuelername, f.Fachname, l.Name AS Lehrername
FROM Noten n
JOIN Schueler s ON n.SchuelerID = s.SchuelerID
JOIN Fach f ON n.FachID = f.FachID
JOIN Lehrer l ON f.LehrerID = l.LehrerID;
```

Erläuterung: Mehrfach-JOIN über alle vier Tabellen, um vollständige Informationen zu erhalten.

3. Durchschnittsnote je Klasse im Fach Mathematik

```
SELECT s.Klasse, AVG(n.Note) AS Durchschnitt
FROM Noten n
JOIN Schueler s ON n.SchuelerID = s.SchuelerID
JOIN Fach f ON n.FachID = f.FachID
WHERE f.Fachname = 'Mathematik'
GROUP BY s.Klasse;
```

Erläuterung: Filterung nach `Mathematik`, Gruppierung nach Klasse, Durchschnittsberechnung mit `AVG()`.

Einfache SQL-Abfragen (ohne JOINs)

(a) **Geben Sie alle Schüler aus der Klasse `10A` aus.**

```
SELECT *
FROM Schueler
WHERE Klasse = '10A';
```

(b) **Geben Sie alle Fächer mit der FachID `F01` aus.**

```
SELECT *
FROM Fach
WHERE FachID = 'F01';
```

(c) **Geben Sie alle Noten aus, die besser als `3` sind.**

```
SELECT *
FROM Noten
WHERE Note < 3;
```

Sprachen und Grammatiken

1. Was versteht man unter einer formalen Sprache?
   Eine formale Sprache ist eine Menge von Wörtern über einem bestimmten Alphabet, die nach festen Regeln definiert sind.

2. Was ist ein Alphabet?

   In der formalen Sprachtheorie bezeichnet ein **Alphabet** eine endliche, nichtleere Menge von Symbolen.

   - Die Elemente eines Alphabets werden auch **Zeichen** oder **Symbole** genannt.

- Ein Alphabet wird üblicherweise mit dem griechischen Buchstaben Σ bezeichnet.

Definition: Ein Alphabet ist eine endliche Menge:

$$\Sigma = \{a_1, a_2, \ldots, a_n\}$$

Beispiele:

- $\Sigma_1 = \{0, 1\}$ — binäres Alphabet
- $\Sigma_2 = \{a, b, c\}$
- $\Sigma_3 = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ — DNA-Zeichen

Hinweis: Ein Alphabet enthält keine weiteren Informationen über die Bedeutung der Zeichen — es geht rein um die Menge der verwendbaren Symbole.

3. Können Sie ein Beispiel für eine formale Sprache nennen?

Eine **formale Sprache** ist eine Menge von Wörtern, die aus einem gegebenen Alphabet gebildet wurden und bestimmten Regeln folgen.

Beispiel: Gegeben sei das Alphabet $\Sigma = \{a, b\}$. Die folgende Menge ist eine formale Sprache über Σ:

$$L = \{w \in \{a, b\}^* \mid w \text{ enthält gleich viele } a\text{s wie } b\text{s}\}$$

Beispiele für Wörter in L:

$$\epsilon,\ ab,\ ba,\ aabb,\ abab,\ bbaa,\ baba, \ldots$$

Nicht in L:

$$a,\ b,\ aa,\ abb,\ abbb, \ldots$$

Erklärung: Alle Wörter in L bestehen aus einer beliebigen Anzahl von a und b, aber die Anzahl von a- und b-Symbolen muss gleich sein.

Hinweis: Diese Sprache ist nicht regulär, aber kontextfrei. Sie kann z. B. durch einen Kellerautomaten erkannt oder mit einer kontextfreien Grammatik beschrieben werden.

4. Wie unterscheidet sich eine formale Sprache von einer natürlichen Sprache?

Formale Sprachen und **natürliche Sprachen** unterscheiden sich grundlegend in ihrer Struktur, Entstehung und Verwendung.

Formale Sprachen

- Künstlich definierte Symbolmengen mit exakten Regeln.
- Entstehen durch eine explizite Definition (z. B. Grammatik oder Automaten).
- Bestehen aus einem Alphabet Σ und einer Menge von Wörtern $L \subseteq \Sigma^*$.
- Keine Mehrdeutigkeit: Jedes Wort gehört eindeutig zur Sprache oder nicht.
- Beispiel: Programmiersprachen, mathematische Ausdrucksformen, reguläre oder kontextfreie Sprachen.

Natürliche Sprachen

- Entwickeln sich historisch und gesellschaftlich (z. B. Deutsch, Englisch).
- Enthalten oft Mehrdeutigkeiten, Kontextabhängigkeit und Ausnahmen.
- Grammatik ist nicht vollständig formal beschreibbar.
- Bedeutung hängt vom kulturellen und situativen Kontext ab.

Gegenüberstellung

Aspekt	**Formale Sprache**	**Natürliche Sprache**
Ursprung	Künstlich definiert	Natürlich gewachsen
Regeln	Streng formal, eindeutig	Teilweise flexibel, oft mehrdeutig
Ziel	Mathematische Genauigkeit	Kommunikation im Alltag
Verständlichkeit	Maschinenorientiert	Menschenorientiert

Fazit: Formale Sprachen sind exakt, maschinenlesbar und dienen technischen oder mathematischen Zwecken.

5. Was ist der Unterschied zwischen einer regulären und einer kontextfreien Grammatik?

In der formalen Sprachtheorie unterscheidet man verschiedene Typen von Grammatiken nach der sogenannten **Chomsky-Hierarchie**. Zwei der wichtigsten Typen sind:

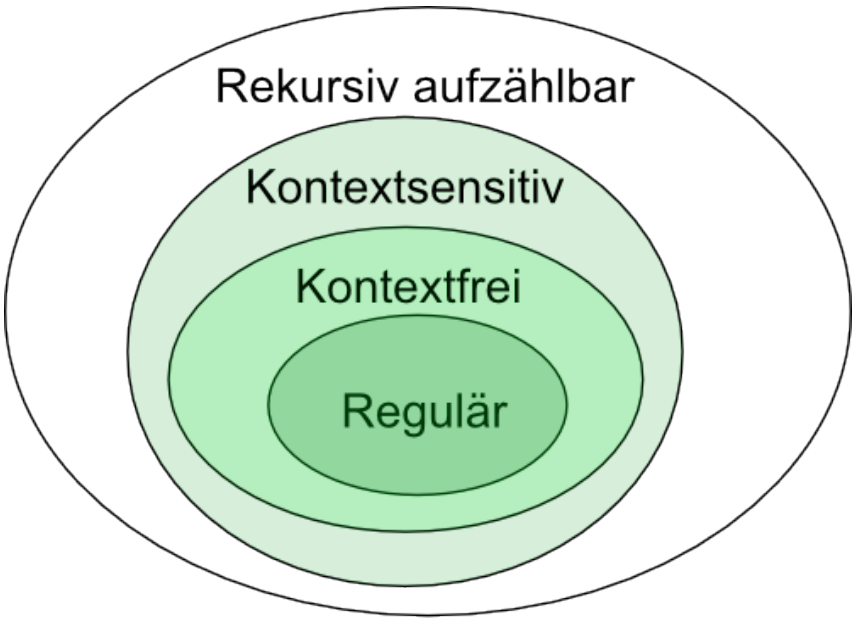

Figure 2: Chomsky-Hierarchie der formalen Sprachen

1. Reguläre Grammatik (Typ-3-Grammatik)

Eigenschaften:

- Produktionsregeln haben die Form: $A \to aB$ oder $A \to a$ (A, B sind Nichtterminale, a ist ein Terminalsymbol)
- Es ist nur eine sehr eingeschränkte Form von Rekursion erlaubt.
- Sie erzeugt genau die **regulären Sprachen**.

Beispiel:

$$S \to aS \mid bS \mid \epsilon$$

Erzeugt alle Wörter über dem Alphabet $\{a, b\}$: $L = \{a, b\}^*$

2. Kontextfreie Grammatik (Typ-2-Grammatik)

Eigenschaften:

- Produktionsregeln haben die Form: $A \to \gamma$, wobei A ein Nichtterminal und $\gamma \in (V \cup \Sigma)^*$ ist
- Das linke Seiten der Regel besteht aus genau **einem** Nichtterminal
- Es sind auch verschachtelte Strukturen und Rekursion möglich
- Sie erzeugt die **kontextfreien Sprachen**

Beispiel:

$$S \to aSb \mid \epsilon$$

Erzeugt die Sprache:

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

Hauptunterschiede

- Reguläre Grammatiken können nur sehr einfache Strukturen beschreiben (keine verschachtelten Abhängigkeiten).
- Kontextfreie Grammatiken erlauben rekursive, symmetrische Strukturen wie Klammerausdrücke oder geschachtelte Blöcke.
- Reguläre Sprachen sind eine Teilmenge der kontextfreien Sprachen:

$$\text{regulär} \subset \text{kontextfrei}$$

Fazit: Reguläre Grammatiken sind einfacher, aber auch eingeschränkter. Kontextfreie Grammatiken sind flexibler und können komplexere Strukturen modellieren.

6. Was ist eine Grammatik, und aus welchen Komponenten besteht sie?

In der formalen Sprachtheorie beschreibt eine **Grammatik** die Regeln, nach denen gültige Wörter (bzw. Sätze) einer formalen Sprache gebildet werden können.

Definition einer Grammatik

Eine Grammatik ist ein Tupel:

$$G = (V, \Sigma, P, S)$$

Dabei gilt:

- V: endliche Menge der **Nichtterminalsymbole** (Variablen), z. B. $\{S, A, B\}$
- Σ: endliche Menge der **Terminalsymbole** (Alphabet), z. B. $\{a, b\}$ $(V \cap \Sigma = \emptyset)$
- P: endliche Menge von **Produktionsregeln** der Form $\alpha \to \beta$, wobei $\alpha, \beta \in (V \cup \Sigma)^*$ und $\alpha \neq \epsilon$
- $S \in V$: **Startsymbol**, von dem die Ableitungen ausgehen

Beispiel einer einfachen Grammatik

- Nichtterminale: $V = \{S\}$
- Terminale: $\Sigma = \{a, b\}$
- Produktionsregeln: $P = \{S \to aSb,\ S \to \epsilon\}$
- Startsymbol: S

Diese Grammatik erzeugt die Sprache:

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

$\to$ also Wörter mit gleich vielen a- und b-Symbolen, wobei alle a's vor den b's stehen.

Fazit

Eine formale Grammatik besteht aus vier Komponenten und definiert eine Sprache durch regelgeleitete Ableitungen. Sie ist ein zentrales Konzept in der formalen Sprachtheorie, Compilerbau und theoretischen Informatik.

7. Erklären Sie den Unterschied zwischen Terminal- und Nichtterminalsymbolen.

In einer formalen Grammatik unterscheidet man zwei Arten von Symbolen:

1. Terminalsymbole (Σ)

- Die **Terminalsymbole** sind die Zeichen des Alphabets der Sprache.
- Sie sind die Endprodukte der Ableitung, aus denen die Wörter der Sprache bestehen.
- Terminalsymbole können nicht weiter ersetzt werden.
- Beispiel: $\Sigma = \{a, b\}$ – die Buchstaben, die später in den erzeugten Wörtern vorkommen.

2. Nichtterminalsymbole (V)

- Die **Nichtterminalsymbole** dienen als Platzhalter oder Variablen im Ableitungsprozess.
- Sie werden im Laufe der Ableitung durch andere Nichtterminale oder Terminale ersetzt.
- Die Ableitung beginnt beim **Startsymbol**, das immer ein Nichtterminal ist.
- Beispiel: $V = \{S, A, B\}$

Gegenüberstellung

Aspekt	**Nichtterminalsymbole**	**Terminalsymbole**
Rolle	Platzhalter im Ableitungsprozess	Zeichen des endgültigen Wortes
Ableitung	Werden durch Regeln ersetzt	Bleiben unverändert
Beispiel	$S \to aSb$	a, b

Fazit: Nichtterminalsymbole steuern den Aufbau eines Wortes und werden im Ableitungsprozess ersetzt, während Terminalsymbole die tatsächlichen Buchstaben der Sprache darstellen und das endgültige Ergebnis bilden.

8. Was bedeutet eine kontextfreie Grammatik (CFG)? Geben Sie ein Beispiel.

Eine **kontextfreie Grammatik (engl. context-free grammar, CFG)** ist ein formales System zur Beschreibung von Sprachen, bei dem jede Produktionsregel nur ein einzelnes Nichtterminal auf der linken Seite hat.

Formale Definition

Eine kontextfreie Grammatik ist ein Tupel:

$$G = (V, \Sigma, P, S)$$

mit:

- V: Menge der **Nichtterminalsymbole**
- Σ: Menge der **Terminalsymbole**, $V \cap \Sigma = \emptyset$
- P: Menge der **Produktionsregeln** der Form $A \to \gamma$, wobei $A \in V$, $\gamma \in (V \cup \Sigma)^*$
- $S \in V$: **Startsymbol**

Merkmal: Jede Regel hat links genau ein Nichtterminalsymbol. Die Ersetzung ist unabhängig vom Kontext, in dem das Symbol steht – daher "kontextfrei".

Beispiel einer CFG

Gegeben sei folgende Grammatik:

- $V = \{S\}$
- $\Sigma = \{a, b\}$
- $P = \{S \to aSb,\ S \to \epsilon\}$
- S: Startsymbol

Erzeugte Sprache:

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

Beispielhafte Ableitungen:

$$S \Rightarrow \epsilon$$
$$S \Rightarrow aSb \Rightarrow ab$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Fazit

Eine kontextfreie Grammatik erlaubt die Beschreibung von strukturierten, verschachtelten Sprachen wie Klammerausdrücken, Programmiersprachen oder $a^n b^n$. Sie ist mächtiger als eine reguläre Grammatik und spielt eine zentrale Rolle in der Informatik.

9. Stellen Sie einen Vergleich zwischen Regulären und Kontextfreien Grammatik.

Vergleich: Reguläre vs. Kontextfreie Grammatiken

Reguläre Grammatiken (Typ-3) und **Kontextfreie Grammatiken (Typ-2)** gehören beide zur Chomsky-Hierarchie, unterscheiden sich jedoch in Ausdrucksstärke und Struktur.

Kriterium	**Reguläre Grammatik (Typ-3)**	**Kontextfreie Grammatik (Typ-2)**
Produktionsregeln	Nur in der Form $A \to aB$ oder $A \to a$	In der Form $A \to \gamma$, wobei $\gamma \in (V \cup \Sigma)^*$
Linke Seite der Regel	Ein Nichtterminalsymbol	Ein Nichtterminalsymbol
Rechte Seite der Regel	Maximal ein Terminal gefolgt von einem Nichtterminal oder nur ein Terminal	Beliebige Kombination aus Terminalen und Nichtterminalen
Ausdrucksstärke	Einfach, keine verschachtelten Strukturen	Unterstützt rekursive und verschachtelte Strukturen
Erkennbarkeit	Mit endlichen Automaten (DFA/NFA)	Mit Kellerautomaten (PDA)
Typische Anwendung	Reguläre Ausdrücke, einfache Mustererkennung	Programmiersprachen, mathematische Ausdrücke
Beispiel-Sprache	$L = \{a^n \mid n \geq 0\}$	$L = \{a^n b^n \mid n \geq 0\}$

Zusammenfassung

- **Reguläre Grammatiken** sind einfach und effizient, aber in ihrer Ausdrucksstärke eingeschränkt.
- **Kontextfreie Grammatiken** sind flexibler und können auch komplexe, verschachtelte Strukturen modellieren.
- Jede reguläre Sprache ist auch kontextfrei, aber nicht jede kontextfreie Sprache ist regulär:

$$\text{regulär} \subset \text{kontextfrei}$$

10. Sind Programmiersprachen wie beispielsweise Java oder C++ kontext- frei oder regulär.

Programmiersprachen wie **Java**, **C++**, **Python** usw. sind **nicht regulär**, aber in weiten Teilen **kontextfrei**.

Begründung:

- Die Syntax von Programmiersprachen (z. B. Schleifen, Anweisungen, Funktionsdefinitionen) lässt sich meist durch eine **kontextfreie Grammatik (CFG)** beschreiben.
- Die lexikalische Struktur (z. B. Schlüsselwörter, Operatoren, Literale) kann hingegen durch **reguläre Ausdrücke** erkannt werden.

Kombiniertes Parsing-Verfahren

Typische Compiler nutzen zwei Phasen:

- **Scanner (Lexer):** erkennt Tokens mit Hilfe regulärer Ausdrücke ($\to$ regulär)
- **Parser:** analysiert den Satzbau auf Basis einer kontextfreien Grammatik ($\to$ kontextfrei)

Grenzen der Kontextfreiheit

Einige Sprachaspekte wie z. B.:

- Namensauflösung (Gültigkeit von Bezeichnern)
- Typkompatibilität
- Sichtbarkeitsbereiche (Scopes)

... lassen sich **nicht mit einer kontextfreien Grammatik beschreiben**, sondern erfordern eine **semantische Analyse** — also zusätzliche Regeln außerhalb der Grammatik.

Fazit

> Die **Syntax moderner Programmiersprachen** ist im Allgemeinen **kontextfrei**, aber nicht vollständig durch eine kontextfreie Grammatik erfassbar. Ihre lexikalische Struktur ist regulär, ihre semantischen Aspekte erfordern zusätzliche Verarbeitung.

11. Erstellen Sie eine kontextfreie Grammatik der Binärzahlen.

    Eine kontextfreie Grammatik G erzeugt Binärzahlen:

    - Terminale: $\{0, 1\}$
    - Nichtterminale: $\{S\}$
    - Startsymbol: S
    - Produktionsregeln:

$$S \to 0S \mid 1S \mid 0 \mid 1$$

Vergleichsaufgabe: Reguläre vs. Kontextfreie Grammatik

Aufgabenstellung: Gegeben sind zwei Grammatiken G_1 und G_2. Bestimmen Sie für beide:

(a) Die Sprache $L(G)$, die sie erzeugen.
(b) Ob die Grammatik regulär oder kontextfrei ist.
(c) Begründen Sie Ihre Entscheidung anhand der Regeln.

Grammatik G_1

- $V = \{S\}$
- $\Sigma = \{a, b\}$
- $P = \{S \to aS \mid bS \mid \epsilon\}$
- Startsymbol: S

Lösung zu G_1

- Diese Grammatik erzeugt alle Wörter über $\{a, b\}$, also:

$$L(G_1) = \{a, b\}^*$$

- Alle Regeln haben die Form: $S \to aS$, $S \to bS$, $S \to \epsilon$
- Das entspricht genau einer **rechtslinearen regulären Grammatik**.
- **Fazit:** G_1 ist **regulär**.

Grammatik G_2

- $V = \{S\}$
- $\Sigma = \{a, b\}$
- $P = \{S \to aSb \mid \epsilon\}$
- Startsymbol: S

Lösung zu G_2

- Die Regel $S \to aSb$ erlaubt es, ein `a` am Anfang und ein `b` am Ende hinzuzufügen.
- Damit entsteht z. B.:

$$\epsilon,\ ab,\ aabb,\ aaabbb, \ldots$$

- Die Sprache ist:

$$L(G_2) = \{a^n b^n \mid n \geq 0\}$$

- Diese Sprache ist **nicht regulär** (Pumping-Lemma!), aber sie ist **kontextfrei**.
- Die Regel $S \to aSb$ ist **nicht rechtslinear**, da das Nichtterminal S in der Mitte steht.
- **Fazit:** G_2 ist **kontextfrei**, aber nicht regulär.

Zusammenfassung:

Grammatik	Sprache	Typ
G_1	$\{a, b\}^*$	Regulär
G_2	$\{a^n b^n \mid n \geq 0\}$	Kontextfrei, nicht regulär

Erkenntnis: Reguläre Grammatiken können nur lineare Strukturen erzeugen. Kontextfreie Grammatiken sind mächtiger und können auch verschachtelte Strukturen wie $a^n b^n$ erzeugen.