Musterlösungen

May 31, 2025

Lösung zu Aufgabe 1: Analyse eines Algorithmus

Gegebener Java-Code

```
public class Algorithmus {
 public static int berechne(int x, int y) {
  if (y == 0) {
   return x;
  }
  return berechne(y, x % y);
 }

 public static void main(String[] args) {
  int wert1 = 56;
  int wert2 = 98;
  System.out.println("Ergebnis: " + berechne(wert1, wert2));
 }
}
```

1. Beschreibung der Funktionsweise

Der Algorithmus berechnet den **größten gemeinsamen Teiler (ggT)** zweier Zahlen mithilfe des **Euklidischen Algorithmus**. Die Rekursion basiert auf der Formel:

$$ggT(a,b) = \begin{cases} a, & \text{wenn } b = 0 \\ ggT(b, a \mod b), & \text{sonst} \end{cases}$$

2. Art der Implementierung und Vorteile

Implementierung: Der Algorithmus verwendet eine **rekursive Methode**. Vorteile sind:

- Elegante und kompakte Umsetzung
- Effizient für große Zahlen
- Direkte Umsetzung der mathematischen Definition

3. Beispielrechnung für berechne(56, 98)

$$\begin{aligned}
&\text{berechne}(56, 98) \rightarrow \text{berechne}(98, 56)\\
&\text{berechne}(98, 56) \rightarrow \text{berechne}(56, 42)\\
&\text{berechne}(56, 42) \rightarrow \text{berechne}(42, 14)\\
&\text{berechne}(42, 14) \rightarrow \text{berechne}(14, 0)\\
&\quad \text{Ergebnis: } 14
\end{aligned}$$

4. Anzahl der Aufrufe

Insgesamt werden **4 rekursive Aufrufe** benötigt, um den ggT zu berechnen.

5. Herleitung der Zeitkomplexität

Die Anzahl der Rekursionsschritte im Euklidischen Algorithmus hängt davon ab, wie schnell sich die Werte x und y durch die Modulo-Operation verringern. Eine grundlegende Eigenschaft dieses Algorithmus ist:

$$a_{i+1} = a_i \mod a_{i-1}$$

Ein wichtiger mathematischer Zusammenhang wurde von Gabriel Lamé (1812) bewiesen: Die Anzahl der Rekursionsschritte ist höchstens proportional zur Anzahl der Fibonacci-Zahlen, die in x und y enthalten sind.

Sei F_n die n-te Fibonacci-Zahl. Falls x und y zwei aufeinanderfolgende Fibonacci-Zahlen sind, dann ist die maximale Anzahl der Rekursionsschritte $O(n)$, wobei gilt:

$$F_n \approx \frac{\varphi^n}{\sqrt{5}}, \quad \text{mit } \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618.$$

Daher folgt:

$$n = O(\log \min(x, y)).$$

Das bedeutet, dass der Euklidische Algorithmus in maximal $O(\log \min(x, y))$ Schritten terminiert.

Beispiel 1: ggT(56, 98)

Die Berechnungsschritte sind:

$$\begin{aligned}
&ggT(56, 98) \rightarrow ggT(98, 56)\\
&ggT(98, 56) \rightarrow ggT(56, 42)\\
&ggT(56, 42) \rightarrow ggT(42, 14)\\
&ggT(42, 14) \rightarrow ggT(14, 0)\\
&\quad \text{Ergebnis: } 14
\end{aligned}$$

Hier sehen wir, dass in jedem Schritt die kleinere Zahl stark reduziert wird, insbesondere durch die Modulo-Operation.

Beispiel 2: ggT(89, 55)

Die Zahlen 89 und 55 gehören zur Fibonacci-Folge. Der Euklidische Algorithmus benötigt bei Fibonacci-Zahlen die meisten Schritte:

$$\begin{aligned} ggT(89,55) &\to ggT(55,34) \\ ggT(55,34) &\to ggT(34,21) \\ ggT(34,21) &\to ggT(21,13) \\ ggT(21,13) &\to ggT(13,8) \\ ggT(13,8) &\to ggT(8,5) \\ ggT(8,5) &\to ggT(5,3) \\ ggT(5,3) &\to ggT(3,2) \\ ggT(3,2) &\to ggT(2,1) \\ ggT(2,1) &\to ggT(1,0) \\ \text{Ergebnis: } &1 \end{aligned}$$

Wir haben hier **9 Schritte** benötigt. Tatsächlich wächst die Anzahl der Schritte proportional zur Anzahl der Stellen in der Binärdarstellung von 89, was der logarithmischen Zeitkomplexität entspricht.

Zusammenfassung: Die Anzahl der Schritte entspricht ungefähr der Anzahl der Male, die man eine Zahl halbieren kann, bevor sie 1 erreicht. Das ist genau der logarithmische Zusammenhang, weshalb der Algorithmus in $O(\log \min(x, y))$ Schritten terminiert.

Logarithmus-Basen und Big-$\mathcal{O}$: Warum die Basis egal ist?

1. Unterschied bei konkreten Werten:
Die Basis eines Logarithmus verändert den Zahlenwert:

$$\begin{aligned} \log_{10}(55) &\approx 1{,}74 \\ \ln(55) &\approx 4{,}007 \\ \log_{\varphi}(55) &\approx 8{,}33 \quad \text{mit } \varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618 \end{aligned}$$

Diese Werte unterscheiden sich, weil jede Basis anders "zählt".

2. Umrechnung zwischen Basen:
Für beliebige Basen b und k gilt:

$$\log_b(n) = \frac{\log_k(n)}{\log_k(b)} = \text{Konstante} \cdot \log_k(n)$$

→ Der Unterschied ist nur ein **konstanter Faktor**.

3. Bedeutung für die Big-$\mathcal{O}$-Notation:
In der asymptotischen Laufzeitanalyse (Big-O) ignorieren wir konstante Faktoren:

$$O(\log_{10} n) = O(\ln n) = O(\log_{\varphi} n) = O(\log n)$$

→ Die Basis ist asymptotisch **nicht relevant**.

Fazit: Bei konkreten Rechnungen macht die Basis des Logarithmus einen Unterschied, aber bei der asymptotischen Betrachtung in der Big-$\mathcal{O}$-Notation zählt nur das *Wachstumsverhalten*, nicht die Skala.

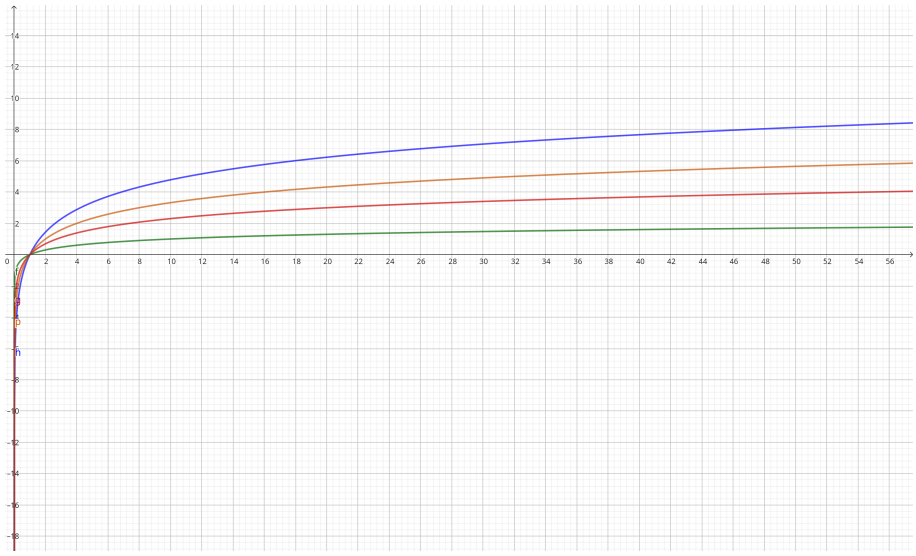

Figure 1: Vergleich verschiedener Logarithmusfunktionen: $\log_{10}(n)$, $\ln(n)$, $\log_2(n)$, $\log_\varphi(n)$

Warum taucht beim ggT-Algorithmus die goldene Zahl φ auf?

Hintergrund: Der euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers (ggT) hat im schlimmsten Fall eine Laufzeit von $O(\log n)$. Der Worst Case tritt auf, wenn die Eingabezahlen zwei aufeinanderfolgende Fibonacci-Zahlen sind.

1. Fibonacci-Zahlen und der ggT-Algorithmus

Berechnet man `ggT(F`$_{n+1}$`, F`$_n$`)`, ergibt jeder Modulo-Schritt das nächste Fibonacci-Glied:

$$F_{n+1} \bmod F_n = F_{n-1}, \quad F_n \bmod F_{n-1} = F_{n-2}, \quad \ldots, \quad F_2 \bmod F_1 = F_0 = 0$$

$\Rightarrow$ Der Algorithmus braucht genau n Schritte.

2. Wachstum der Fibonacci-Zahlen

Die Fibonacci-Zahlen wachsen exponentiell nach der Binet-Formel:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}, \quad \text{mit } \varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618, \quad \psi = \frac{1-\sqrt{5}}{2}$$

Für große n ist ψ^n vernachlässigbar, daher:

$$F_n \approx \frac{\varphi^n}{\sqrt{5}} \quad \Rightarrow \quad n \approx \log_\varphi(F_n)$$

3. Laufzeitabschätzung:

Ist $b = \min(a, b)$, dann:

$$T(a,b) \in \Theta(\log_\varphi b) \Rightarrow O(\log b)$$

Die Basis φ ergibt sich direkt aus dem Wachstum der Fibonacci-Folge.

4. Fazit: Die goldene Zahl φ taucht auf, weil Fibonacci-Zahlen den Worst Case im ggT-Algorithmus erzeugen – und diese wachsen mit Basis φ. Deshalb ist die Anzahl der Schritte asymptotisch:

$$\boxed{O(\log n)}$$

wobei die genaue Zahl der Schritte $\approx \log_\varphi(\min(a,b))$ ist.

6. Vergleich mit iterativen Methoden

Eine iterative Implementierung vermeidet die Stack-Tiefe der Rekursion und spart Speicher:

```
public class Algorithmus {
 public static int berechne(int x, int y) {
  while (y != 0) {
   int temp = y;
   y = x % y;
   x = temp;
  }
  return x;
 }
}
```

Beide Varianten haben dieselbe Zeitkomplexität $O(\log \min(x,y))$, doch die iterative Variante benötigt weniger Speicher.

Fazit

- Der Algorithmus berechnet den **größten gemeinsamen Teiler** zweier Zahlen effizient. - Die Rekursion reduziert die Zahlen in jedem Schritt. - Die Zeitkomplexität beträgt $O(\log \min(x,y))$, was sehr effizient ist. - Die mathematische Herleitung basiert auf Fibonacci-Zahlen und der goldenen Zahl φ. - Eine **iterative Variante** vermeidet die Tiefe der Rekursion und spart Speicher.

Lösung zu Aufgabe 2: Datenmodellierung und Normalisierung

1. **Redundanzen in der Tabelle:**

   - **Kundendaten** (Kundenname, Adresse) werden mehrfach gespeichert, wenn ein Kunde mehrere Bestellungen tätigt.

- **Produktdaten** (Produktname, Kategorie, Preis) werden mehrfach gespeichert, wenn dasselbe Produkt mehrfach bestellt wird.
- Diese Wiederholungen führen zu **Redundanz**, **Speicherplatzverschwendung** und möglichen **Inkonsistenzen** (z. B. bei Änderungen).

2. **Normalisierung bis zur 3. Normalform:**

   Ausgangstabelle (nicht normalisiert):

BestellID	KundenID	Kundenname	Adresse	ProduktID	Produktname	Kategorie	Preis	Menge
1	1001	Anna Müller	Hauptstr. 12, Berlin	501	Laptop	Elektronik	1200	1
2	1002	Thomas Becker	Marktstr. 5, Hamburg	502	Smartphone	Elektronik	800	2
3	1001	Anna Müller	Hauptstr. 12, Berlin	503	Drucker	Bürobedarf	150	1
4	1003	Julia Schmitt	Lindenallee 9, München	504	Schreibtisch	Möbel	300	1
5	1002	Thomas Becker	Marktstr. 5, Hamburg	505	Monitor	Elektronik	200	1

1. Normalform (1NF):
Alle Attributwerte sind atomar. Diese Bedingung ist bereits erfüllt.

2. Normalform (2NF):
Es gibt funktionale Abhängigkeiten von Nicht-Schlüsselattributen zu nur einem Teil des (angenommenen) zusammengesetzten Schlüssels. Deshalb wird die Tabelle in drei Tabellen aufgeteilt:

Tabelle: `Kunde`

KundenID	Kundenname	Adresse
1001	Anna Müller	Hauptstr. 12, Berlin
1002	Thomas Becker	Marktstr. 5, Hamburg
1003	Julia Schmitt	Lindenallee 9, München

Tabelle: `Produkt`

ProduktID	Produktname	Kategorie	Preis
501	Laptop	Elektronik	1200
502	Smartphone	Elektronik	800
503	Drucker	Bürobedarf	150
504	Schreibtisch	Möbel	300
505	Monitor	Elektronik	200

Tabelle: `Bestellung`

BestellID	KundenID	ProduktID	Menge
1	1001	501	1
2	1002	502	2
3	1001	503	1
4	1003	504	1
5	1002	505	1

3. Normalform (3NF):
Alle Nicht-Schlüsselattribute hängen nur vom Primärschlüssel ab – keine transitiven Abhängigkeiten. Die Tabellenstruktur entspricht jetzt der 3NF.

3. **Vorteile der Normalisierung:**
   - **Vermeidung von Redundanzen:** Kundendaten und Produktdaten werden nur einmal gespeichert.
   - **Konsistenz:** Änderungen (z. B. Adresse eines Kunden, Preis eines Produkts) müssen nur an einer Stelle vorgenommen werden.
   - **Bessere Datenstrukturierung:** Logische Trennung von Entitäten (Kunde, Produkt, Bestellung).
   - **Weniger Speicherbedarf** und bessere Wartbarkeit.

Lösung zu Aufgabe 3: Analyse einer formalen Grammatik

1. **Ableitung von aab:**
   Gesucht ist eine Ableitung mit Startsymbol S:
   $$S \to aA$$
   $$A \to aA$$
   $$A \to b$$
   Das ergibt: aab.
   Antwort: Das Wort aab gehört zur Sprache $L(G)$.

2. **Ableitung von $abba$:**
   Versuch:
   $$S \to aA$$
   $$A \to bA$$
   $$A \to bA$$
   $$A \to a$$
   Das ergibt: $abba$.
   Antwort: Auch das Wort $abba$ gehört zur Sprache $L(G)$.

3. **Struktur der Sprache:**
   Alle Wörter bestehen aus:
   - einem Anfang a oder b (von S)
   - optional beliebigen a- oder b-Folgen (von A)
   - und enden, falls A verwendet wurde, mit einem a

4. **Grammatikänderung für Wörter mit Endung auf b:**
   Um auch Wörter zuzulassen, die auf b enden, kann man die letzte Regel von A erweitern:
   $$A \to aA \mid bA \mid a \mid b$$

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 1: 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?

Auch nach vollständiger Normalisierung (bis zur 3. Normalform) können bestimmte Redundanzen auftreten – diese sind jedoch **gewollt und notwendig**.

Beispielhafte Struktur nach Normalisierung:

- `Kunde(KundenID, Kundenname, Adresse)`
- `Produkt(ProduktID, Produktname, Kategorie, Preis)`
- `Bestellung(BestellID, KundenID)`
- `Bestellposition(BestellID, ProduktID, Menge)`

1. Gewollte Redundanz durch Fremdschlüsselverweise

- `KundenID` kommt sowohl in der Tabelle `Kunde` als auch in `Bestellung` vor.
- `ProduktID` steht in `Produkt` und in `Bestellposition`.
- Diese Wiederholung ist notwendig, um **Beziehungen** zwischen Entitäten darzustellen.

2. Mögliche kontrollierte Redundanz (Performanceoptimierung)

- Beispiel: In der Tabelle `Bestellung` könnte zusätzlich der Kundenname gespeichert werden.
- Das ist redundant, da er bereits in `Kunde` gespeichert ist – aber:
  - Dadurch können Berichte oder Rechnungen schneller generiert werden.
  - Es ist eine **bewusst eingeführte Redundanz** zur **Verbesserung der Lesbarkeit und Abfragegeschwindigkeit**.

3. Redundanz durch Historisierung (Datenkonsistenz über Zeit)

- Beispiel: Der Preis eines Produkts wird zur Zeit der Bestellung in `Bestellposition` gespeichert.
- Auch wenn sich der Preis im Produktkatalog später ändert, bleibt der ursprüngliche Preis der Bestellung erhalten.
- Diese Redundanz ist **gewollt**, um eine korrekte Rechnungsstellung und Nachvollziehbarkeit zu gewährleisten.

Fazit

Obwohl die Datenbank bis zur 3. Normalform normalisiert wurde, treten weiterhin bestimmte Redundanzen auf. Diese sind jedoch **notwendig** (Fremdschlüssel), **bewusst eingeführt** (zur Performanceverbesserung) oder **funktional gewollt** (z. B. Historisierung). Unnötige Redundanzen wie doppelt gespeicherte Kundendaten in jeder Bestellzeile wurden erfolgreich eliminiert.

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 joinunabhängige 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 normalisierte Datenbank mit folgenden Tabellen:

   - `Kunde(KundenID, Kundenname, Adresse)`
   - `Produkt(ProduktID, Produktname, Kategorie, Preis)`
   - `Bestellung(BestellID, KundenID)`
   - `Bestellposition(BestellID, ProduktID, Menge)`

   Formulieren Sie jeweils eine passende SQL-Abfrage:

   (a) Geben Sie die Namen aller Kunden aus, die mindestens eine Bestellung aufgegeben haben.

   (b) Geben Sie für jede Bestellposition den Kundenname, Produktname, Preis und die bestellte Menge aus.

   (c) Ermitteln Sie die Gesamtanzahl bestellter Produkte je Kategorie.

Lösungen: SQL-Abfragen auf das Bestellsystem

Hinweis: Für die Abfragen nutzen wir Joins zwischen den Tabellen, um die Daten logisch zu verknüpfen.

1. Alle Kunden, die mindestens eine Bestellung aufgegeben haben

```
SELECT DISTINCT k.Kundenname
FROM Kunde k
JOIN Bestellung b ON k.KundenID = b.KundenID;
```

Erklärung: Verknüpft die Tabellen `Kunde` und `Bestellung`, um nur Kunden mit Bestellungen auszugeben.

2. Kundenname, Produktname, Preis und Menge je Bestellposition

```
SELECT k.Kundenname, p.Produktname, p.Preis, bp.Menge
FROM Bestellposition bp
JOIN Bestellung b ON bp.BestellID = b.BestellID
JOIN Kunde k ON b.KundenID = k.KundenID
JOIN Produkt p ON bp.ProduktID = p.ProduktID;
```

Erklärung: Mehrfacher JOIN über die Tabellen `Bestellposition`, `Bestellung`, `Kunde` und `Produkt`, um vollständige Infos zu erhalten.

3. Anzahl bestellter Produkte je Kategorie

```
SELECT p.Kategorie, SUM(bp.Menge) AS Gesamtmenge
FROM Bestellposition bp
JOIN Produkt p ON bp.ProduktID = p.ProduktID
GROUP BY p.Kategorie;
```

Erklärung: Zählt alle bestellten Produkte, gruppiert nach Produktkategorie.

10. Gegeben ist das folgende relationale Datenbankschema eines einfachen Bestellsystems:

    - `Kunde(KundenID, Kundenname, Adresse)`
    - `Produkt(ProduktID, Produktname, Kategorie, Preis)`

    Formulieren Sie jeweils eine einfache SQL-Abfrage:

    (a) Geben Sie alle Kunden mit ihrer Adresse aus.

    (b) Geben Sie alle Produkte der Kategorie `'Elektronik'` aus.

    (c) Geben Sie alle Produkte mit einem Preis unter 500 Euro aus.

Lösungen: Einfache SELECT-Abfragen

1. Alle Kunden mit ihrer Adresse

```
SELECT Kundenname, Adresse
FROM Kunde;
```

Erklärung: Zeigt alle Kundennamen und deren Adressen an.

2. Alle Produkte der Kategorie 'Elektronik'

```
SELECT *
FROM Produkt
WHERE Kategorie = 'Elektronik';
```

Erklärung: Filtert alle Produkte, bei denen die Kategorie `Elektronik` ist.

3. Alle Produkte mit einem Preis unter 500 Euro

```
SELECT Produktname, Preis
FROM Produkt
WHERE Preis < 500;
```

Erklärung: Zeigt den Produktnamen und Preis für alle Produkte mit Preis unter 500 Euro.