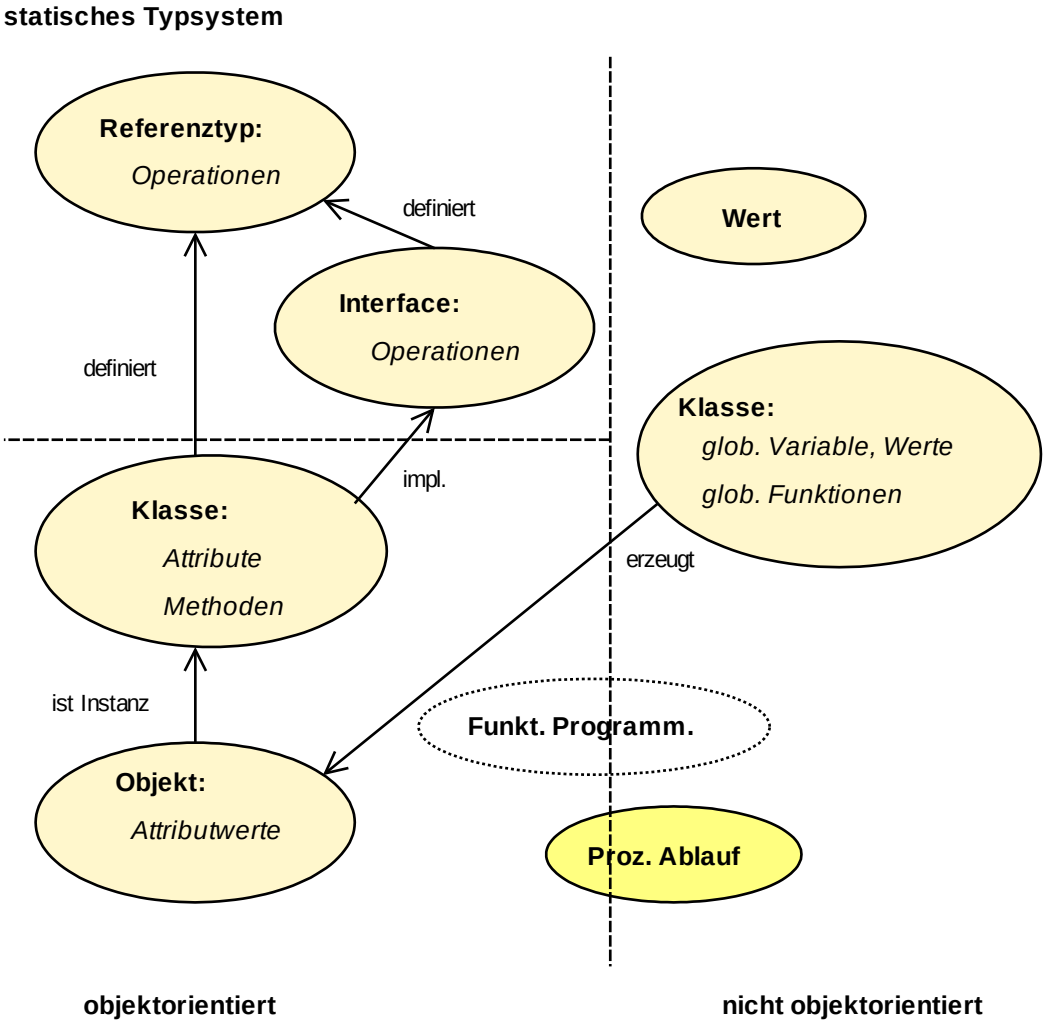
Algorithmen und Datenstrukturen

- Definition und wichtige Eigenschaften
- Korrektheit
- Terminierung und Laufzeiteffizienz
- Systematische Entwicklung der Iteration
- Korrektheit bei Klassen

Fokus: Algorithmen und Datenstrukturen



Geschichte

- In der Antike entwickelte sich systematische mathematische Forschung (Euklid, Alexandria).
- Im frühen Mittelalter war insbesondere in Europa kaum noch etwas davon bekannt.
- Mohamed al-Khwârizmî schrieb im 9. Jhd. ein Buch über Rechenverfahren mit Dezimalzahlen $\Rightarrow$ Algorithmus (*Algoritmi de numero Indorum*).
  Leider endete schon bald danach wieder die Blütezeit der arabischen Mathematik
- In Italien begann im 12. Jhd. die systematische Entwicklung von mathematischen Algorithmen

Beispiele

- Euklidischer Algorithmus (ggt)
- Sieb des Eratosthenes
- Lösung quadratischer und kubischer Gleichungen

Muhammad Ibn Musa Al-Khwarizmi
b. c. 780 Baghdad
d. c. 850

Muslim mathematician and astronomer whose major works introduced Hindu-Arabic numerals and the concepts of algebra into European mathematics. He lived in Baghdad under the caliphates of al-Ma'mun and al-Mu'tasim in the first golden age of Islamic science. His work on elementary mathematics, **Kitab al-jabr wa al-muqabalah** ("The Book of Integration and Equation"), was translated into Latin in the 12th century and originated the term **algebra**.

The Kitab al-jabr is a compilation of rules for arithmetical solutions of linear and quadratic equations, for elementary geometry, and for inheritance problems concerning the distribution of money according to proportions. The work was based on a long tradition originating in Babylonian mathematics of the early 2nd millennium BC and traceable through Hellenistic, Hebrew, and Hindu treatises. Its elementary and practical nature contributed to its survival when other works on the same subject were lost.

Another work on Hindu-Arabic numerals is preserved only in a Latin translation, **Algoritmi de numero Indorum** ("Al-Khwarizmi Concerning the Hindu Art of Reckoning"). From the title originated the term **algorithm**. Al-Khwarizmi also compiled a set of astronomical tables, based largely on the Sindhind, an Arabic version of the Sanskrit work Brahma-siddhanta (7th century AD), but also showing Greek influence.

Definitionen

Ein **Algorithmus** ist eine systematische Vorschrift, nach der durch eine Folge von mehreren Zustandsveränderungen aus einem Anfangszustand der gewünschte Endzustand (die Lösung eines Problems) hervorgebracht wird.
Die einzelnen Zustandsveränderungen sind durch den Algorithmus genau definiert, sie verlaufen deterministisch, und der Algorithmus terminiert nach endlich vielen Schritten.

Unter der **Korrektheit** eines Algorithmus verstehen wir die Übereinstimmung seines Verhaltens mit der Aufgabenstellung (Spezifikation).
Partielle Korrektheit verlangt (noch) nicht zwingend die Terminierung.

Mit **Komplexität** bezeichnen wir allgemein den Aufwand, den man für die Problemlösung treiben muss. Die **Laufzeitkomplexität** ist die Abhängigkeit der Anzahl der Arbeitsschritte von der Problemgröße.

Design by Contract

Die Idee, dass Methoden einen Vertrag erfüllen, ist zentral für den Entwurf einer guten Klassenschnittstelle.

Der Vertrag besteht darin, dass bei Vorliegen einer Vorbedingung die Erfüllung einer Nachbedingung garantiert wird. Die Verletzung der Vorbedingung wird in der Regel durch eine Ausnahme signalisiert. Der Vertrag ist (in Java) ein wichtiger Bestandteil der Dokumentation einer Schnittstelle.

Eine Klasse, die ein Interface implementiert, muss in der Regel dieselbe oder eine schwächere Vorbedingung wie die Schnittstelle verlangen; die Nachbedingung muss entweder gleich oder stärker sein. Entsprechendes gilt für die Ableitung von Klassen. Beides ist eine Konsequenz von Lyskov‘s Substitutionsprinzip.

Eigenschaften von Algorithmen und der Implementierung von Algorithmen

Die wichtigste Eigenschaft eines Algorithmus ist seine **Korrektheit**, gefolgt von seiner **Komplexität**.

Für die Implementierung eines Algorithmus in einer Programmiersprache gibt es weitere wichtige Anforderungen, u.a.:

- **Robustheit**, d.h. definiertes Verhalten, wenn eine Methode falsch eingesetzt wird (Exceptions).
- Möglichst **allgemeine Verwendbarkeit** (vgl. Polymorphie).
- Gute **Dokumentation** der Verwendung des Verfahrens.
- **Lesbarkeit** der Implementierung.
- **Testbarkeit** der Implementierung.
- **Laufzeiteffiziente Implementierung**

Wie kann man Korrekheit belegen ?

Zunächst braucht man zwingend eine **hinreichend genaue Spezifikation**.

Es gibt grundsätzlich zwei sich ergänzende Verfahren zur Untersuchung der Korrektheit:

1. **Testen (dynamisch)**
   Ziel: vollständiger Test
   Praxis: Kombination verschiedener Testverfahren

2. **Beweisen (statisch)**
   Praxis: Code-Inspektion, systematische Softwareentwicklung

Test für eine Funktion power(x, y)

```
// Testprogramm:

static final double EPS = 1e-12;

@Test
public static void testPower() {
    assertEquals(2.0, power(2.0, 1), EPS);
    assertEquals(4.0, power(2.0, 2), EPS);
    assertEquals(1.5, power(1.5, 1), EPS);
}
```

Ergebnis: **grün**

Alles ok.?

Beispielimplementierung von power(x,y)

```
public static double power(double base, int exp) {
    // falsche Fassung
    double result = 0.0;
    for (int i = 1; i <= exp; i++)
    result += base;
    return result;
}
```

Was ist hier falsch?

Was war an dem Test falsch?

Wieso können wir überhaupt richtig und falsch unterscheiden?

Korrektheit, Vor- und Nachbedingung

Ein Programmfragment

{Q}
S
{R}

ist **korrekt**, wenn für alle anfänglichen Variablenbelegungen, die mit der Vorbedingung Q verträglich sind, nach Ausführen des Programmabschnitts S eine Variablenbelegung vorliegt, für die die geforderte Zielbedingung R erfüllt ist.

Anmerkung: Ein Test führt S mit Testdaten, die der Vorbedingung Q genügen aus und prüft ob die Ergebnisse die Nachbedingung R erfüllen.

Logische Bedingungen sind allgemein formuliert, der Test prüft Einzelfälle.

Vorteile Logik: Allgemeingültigkeit
Vorteile Test: Automatische Ausführung, Aussagekraft negativer Ergebnisse

Spezifikation ganzzahliger Potenzierung

```
public static double power(double base, int exp) {
    double result;
    // Q: wenn exp < 0: base != 0 (wird hier nicht geprüft).
    ... Programmstück S
    // R: result = base ^ exp
    return result;
}
```

Formeln:

1) $x^{-y} = 1 / x^{y}$
2) $x^{y} = 1$, falls $y = 0$
3) $x^{y} = x\, x^{y-1}$, falls $y > 0$
4) $x^{y} = (x\, x)^{y/2}$, falls y gerade und $y > 0$

Vorgehensweise

Bei der Entwicklung eines Klassenmodells gehen wir meist bottom-up vor, d.h. wir realisieren Grundbegriffe und Grundstrukturen durch Klassen und damit wieder mächtigere usw.

Bei der Entwicklung eines Algorithmus gehen wir top-down vor, d.h. wir verfeineren nach und nach unseren Lösungsansatz. (schrittweise Verfeinerung).

Entwicklung einer Funktion (1)

```
public static double power(double base int exp) {
    double result;
    // Q: wenn exp < 0: base != 0
    if (exp >= 0) {
        // Q1: exp >= 0
        result = posPower(base, exp);
    }
    else {
        // Q2: exp < 0
        // nach Formel 1:
        result = 1.0 / posPower(base, -exp);
    }
    // R: result = base ^ exponent
    return result;
}

private static double posPower(double base,  int exp) {
    double result;
    // Q: exp >= 0
    ... Programmstück S
    // R: result = base ^ exp
    return result;
}
```

Entwicklung einer Funktion (2)

```
private static double posPower(double base,  int exp) {
    double result;
    // Q: exp >= 0
    if (exp == 0) {
        // Q1: exp == 0
        // nach Formel 2:
        result = 1.0;
    }
    else {
        // Q2: exp > 0
        // nach Formel 3:
        result = base * posPower(base, exp-1);
    }
    // R: result = base ^ exp
    return result;
}
```

Terminierung und Komplexität

Terminierung: Ein rekursiver Algorithmus **terminiert**, wenn bei jedem rekursiven Aufruf die Problemgröße reduziert wird und wenn er über eine Abbruchbedingung verfügt, bei der kein weiterer rekursiver Aufruf mehr erfolgt.

partiell korrekt:	*wenn* der Algorithmus terminiert, ist das Ergebnis korrekt.
total korrekt:	der Algorithmus terminiert *und* das Ergebnis ist korrekt.
Laufzeiteffizienz:	Abschätzung der Anzahl der Lösungsschritte.

Im Folgenden geht es darumfür die Laufzeiteffizienz eines Algorithmus eine einfache Formel zu finden.

Was dürfen wir erwarten?

- **Nicht:** Eine Aussage über die tatsächliche Laufzeit
- **Sehr wohl:** Eine Aussage, wie die Laufzeit mit der Problemgröße skaliert

O-Notation

Mathematik:

Gegeben seien eine Funktion *f(n)* und eine positive Funktion *g(n)*. Man sagt, *f(n)* ist *O(g(n))*, wenn es ein n_0 gibt, so dass für alle $n > n_0$ gilt $c\ g(n) > |f(n)|$.

g(n) kann so gewählt werden, dass es eine sehr einfache Funktion ist.

Algorithmen:

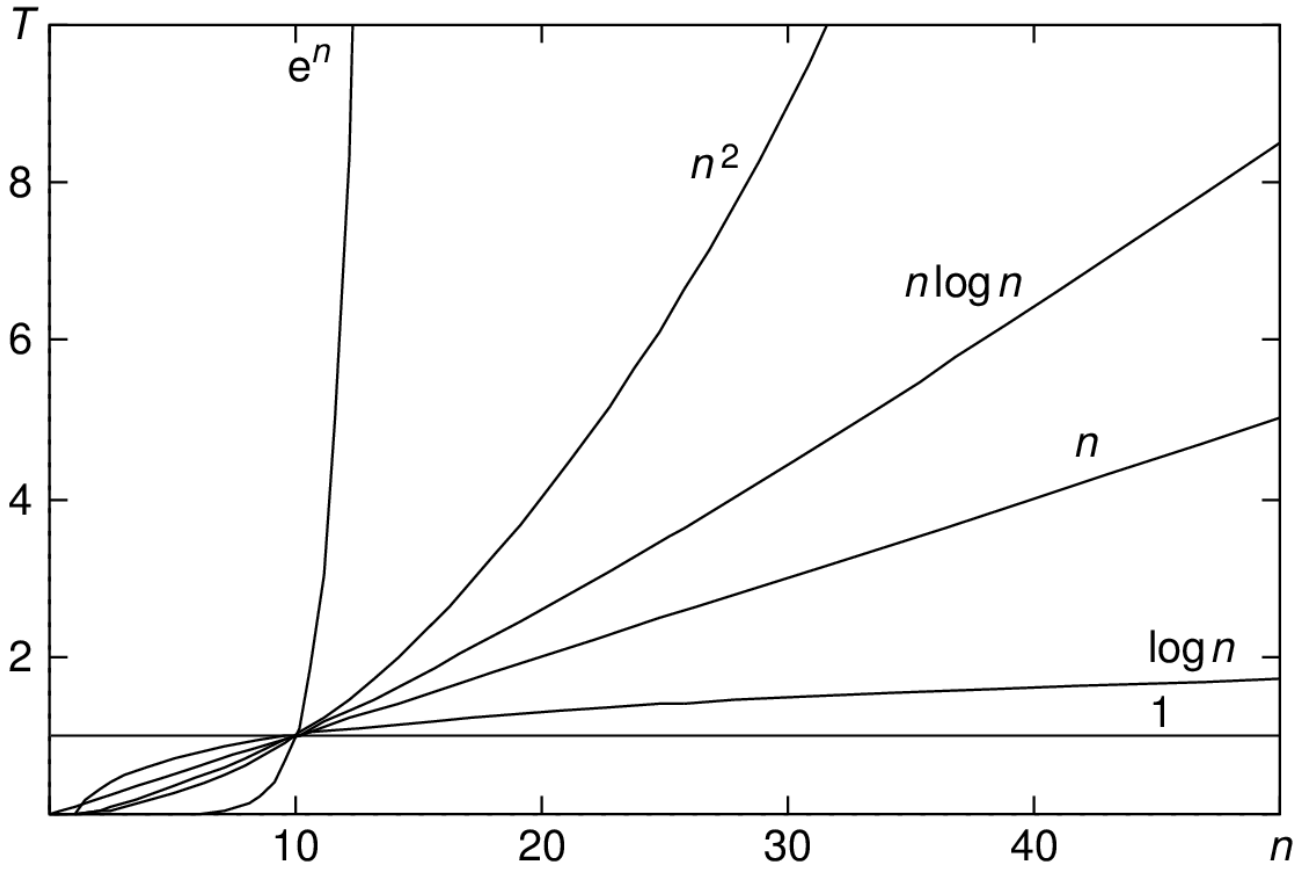
Die **O-Notation** gibt den führenden Term in der Formel für die Anzahl der Schritte eines Algorithmus in Abhängigkeit der Problemgröße *n* an. Konstante Faktoren werden weggelassen. Wenn bei gleichem *n* (durch andere Problemparameter) unterschiedliche Laufzeiten entstehen, beschreibt die O-Notation den ungünstigsten Fall (*worst case*).

Beispiel:

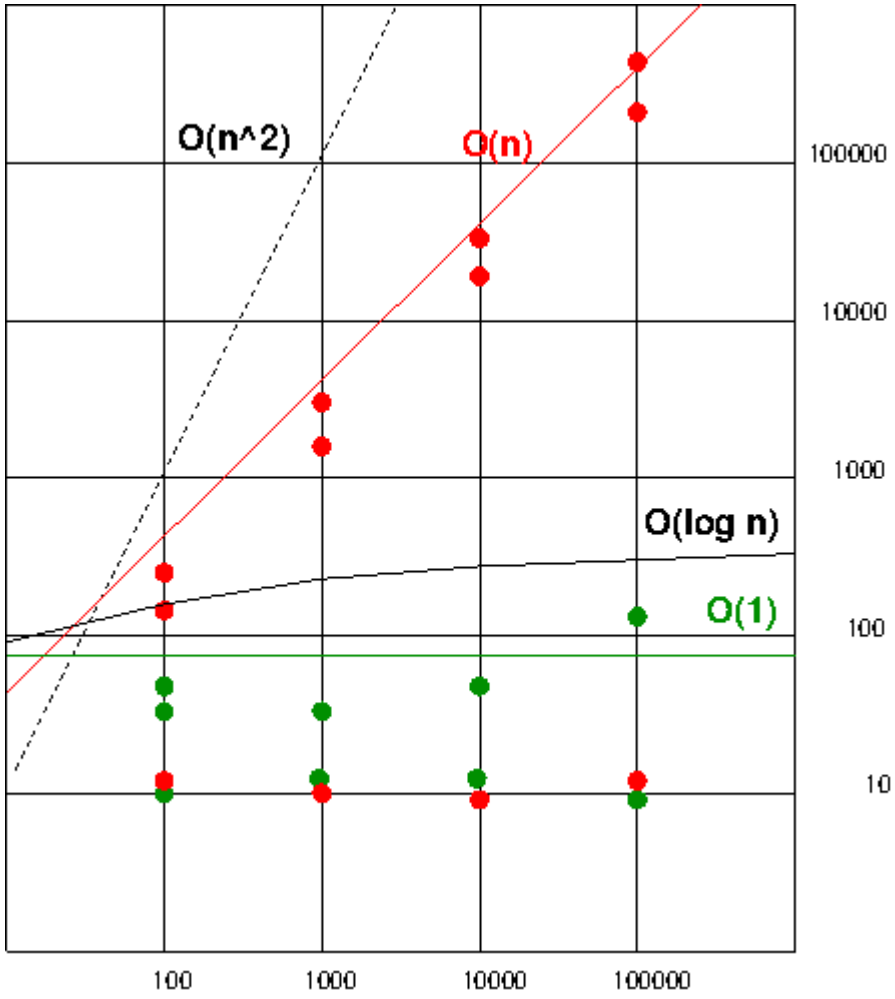
$T(n) = a\ n^2 + b\ n + c$	$O(n^2)$
$T(n) = a\ e^{b\ n} + c\ n^3$	$O(e^n)$ oder genauer $O(e^{b\ n})$
$T(n) = 2^n$	$O(e^n)$ oder genauer $O(2^n)$
$T(n) = 10 \log_2(n) + 20$	$O(\log n)$

Die Basis des Logarithmus spielt keine Rolle da $\log_a(x) = \log_b(x) / \log_b(a)$

Unterschiedliche Komplexitätsklassen zeigen große Unterschiede



Die O-Notation erlaubt die Abschätzung der Laufzeit



Lineare Suche

Hashing

Die gemessenen Werte weisen zufällige Schwankungen auf. Außerdem werden sie durch Details der JVM beeinflusst.

x-Achse: Anzahl der Daten
y-Achse: Zeit für Suche in ns

Hashing: der (sehr sehr seltene) worst case ergibt O(n)!

Die untersten Punkte geben den (unrealistischen) best case an.

Skalierung der Laufzeit lässt sich leicht berechnen

O(1): ***Die Laufzeit ist unabhängig von der Problemgröße***

Beispiel: n= 1000: **1** ms, n: 10000: **1** ms, n: 100000: **1** ms

O(*n*): n-fache Problemgröße benötigt n-fache Zeit:

Beispiel: n= 1000: **1** ms, n: 10000: **10** ms, n: 100000: **0,1** s

O(n^2) ***n-fache Problemgröße benötigt n^2-fache Zeit***

Beispiel: n = 1000: **1** ms, n: 10000: **0,1** s, n: 100000: **10** s

O(log *n*) ***Vergrößerung um konst. Faktor ergibt konst. zusätzlichen Aufwand***

Beispiel: n=1000: **1** ms, n=10000: **1,33** ms, n = 100000: **1,67** ms

O(*n* log *n*): *beinahe linear*

Beispiel: n = 1000: **1** ms, n=10000: **13** ms, n=100000: **0,167** s

O(exp *n*): *explodiert! (es gibt unlösbare Probleme !!!)*

Beispiel: n = 10: **1** µs, n=100: $\mathbf{2^{90}}$ µs, n=1000: $\mathbf{2^{990}}$ µs

= **1** µs = $\mathbf{10^{21}}$ s =$\mathbf{10^{292}}$ s

Vergleich. Alter unseres Universums = 13,8 10^9 y = **4,35 10^{17}** s

Beispiel Falltürfunktion

Falltürfunktion: eindeutig umkehrbare Funktion f, wobei f effizient berechenbar und f^{-1} (*praktisch)* nicht berechenbar ist.

Anwendung: asymmetrische Verschlüsselung (z.B. Schlüsselaustausch bei ssh)

Effizienz (*sehr* stark vereinfacht, aber grob richtig):

a) Produkt zweier Primzahlen mit *d* Ziffern: $\boldsymbol{O(d^2)}$
b) Faktorisierung einer Zahl mit *d* Ziffern: $\boldsymbol{O(10^{\alpha d})}$ (*es ist kein besserer Alg. bekannt!*)

Das ermöglicht die Berechnung von privaten und öffentlichen Schlüsseln.
Aus dem öffentlichen Schlüssel kann man (großes d) nicht den privaten Schlüssel berechnen !

Quantencomputer machen exponentiell harte Probleme (evtl.) lösbar:

Faktorisierung ist nur $O(d^3)$
– (technisch extrem schwierig: gelungen ist (bisher) die Zerlegung von 15)
Quantencomputer würden die momentanen Verschlüsselungsverfahren obsolet machen!

Effizientere Lösung für Potenzierung

20% des Codes braucht 80% der Zeit.

Rechenzeit entsteht da, wo Wiederholungen mit großem *n* vorkommen.

Ziel: bei zeitkritischen Algorithmen Laufzeitverhalten verbessern!

Beispiel:

$2^{16} = 2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2$ 15 Multiplikationen O(*n*)

$2*2 = 4 = 2^2$
$4*4 = 16 = 2^4$
$16*16 = 256 = 2^8$
$256*256 = 65536 = 2^{16}$ 4 Multiplikationen O(log *n*)

Erkenntnis:

Zur Entwicklung effizienter Verfahren braucht man Problemwissen und Kreativität.

Vorgehensweise:

Erst Rekursion, dann (evtl.) Iteration, weil Rekursion einfacher zu verstehen ist!

Entwicklung einer Funktion (3)

```
private static double posPower(double base,  int exp) {
    double result;
    // Q: exp >= 0
    if (exp == 0) {
        // Q1: exp == 0
        // nach Formel 2:
        result = 1.0;
    }
    else if (exp % 2 == 0) {  // neu eingefügt!
        // Q2:  exp > 0 && exp ist gerade
        // nach Formel 4:
        result = posPower(base*base, exp/2);
    }
    else {
        // Q3: exp > 0 ( && exp ist ungerade )
        // nach Formel 3:
        result = base * posPower(base, exp-1);
    }
    // R: result = base ^ exp
    return result;
}
```

Iteration?

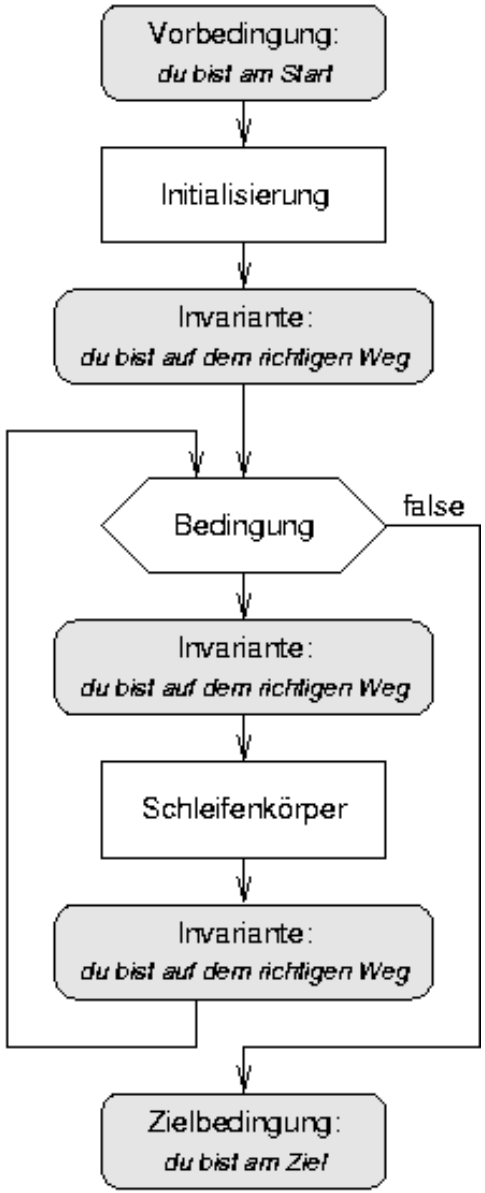
```
private static double posPower(double base, int exp) {
    // Q: exp >= 0
    double result;
    Initialisierung
    // Invariante
    while (Bedingung) {
        // Invariante && Bedingung
    Schleifenkörper
        // Invariante
    }
    // Invariante && ¬ Bedingung ⇒
    // R: result = base ^ exp
    return result;
}
```

Eine **Schleifeninvariante** ist eine Aussage, die beim Eintritt, bei jeder Wiederholung und nach dem Verlassen der Schleife gilt und aus der zusammen mit der negierten Schleifenbedingung die Zielaussage folgt.

Die **Schleifeninvariante** gilt vor, in und nach der Schleife.

Aus der negierten Bedingung und aus der Invarianten folgt das Ziel.

Die Anzahl der Schleifendurchläufe kann durch eine Formel begrenzt werden (Variante), deren Wert sich bei jedem Durchlauf um wenigstens eins vermindert und dabei größer als 0 ist.



Iterativer Algorithmus(1)

```
private static double posPower(double base, int exp) {
    // Q: exp >= 0
    double result = 1.0;
    int i = 0;
    // Invariante: result = base ^ i
    // Variante: exp - i
    while (i != exp) {
        // Invariante and i != exp (d.h. i < exp)
        Schleifenkörper // verkleinere das Restproblem !
        // Invariante
    }
    // result = base ^ i && i = exp ⇒
    // R: result = base ^ exp
    return result;
}
```

verkleinere das Restproblem:

```
i = i + 1;
```

erhalte die Invariante:

```
result = result * base;
```

Iterativer Algorithmus(2)

```
private static double posPower(double base, int exp) {
    // Q: exp >= 0
    double result = 1.0;
    int i = 0;
    // Invariante: result = base ^ i
    // Variante: exp - i
    while (i != exp) {
        // Invariante && i != exp
        result = result * base;
        i = i + 1;
        // Invariante
    }
    // result = base ^ i && i == exp ⇒
    // R: result = base ^ exp
    return result;
}
```

Laufzeit:

Problemgröße: `exp`
Anzahl der Durchläufe: `exp` also O(*n*)

Inhalt der Resultatvariablen wird nach und nach zum Ergebnis

	Invariante					Variante
`i`	`result`	$= base^i$	`base`	`exp`	$base^{exp}$	$exp-i$
0	1	*1*	3	4	*81*	*4*
1	3	*3*				*3*
2	9	*9*				*2*
3	27	*27*				*1*
4	**81**	*81*		4	*81*	***0***

Zusammenfassung

- Am Anfang steht die Spezifikation, d.h. die Angabe von Vorbedingung und Zielbedingung.
- Bei der Algorithmenentwicklung geht man meist nach der Methode der schrittweisen Verfeinerung vor.
- Fast alle Algorithmen erfordern Wiederholung, d.h. Iteration oder Rekursion.
- Iteration bedeutet schrittweise Annäherung an das Ziel.
- Schleifeninvarianten drücken die Idee hinter einer Schleife aus.
- Wenn ein Algorithmus sein Ziel nicht erreichen kann, muss er dies deutlich machen (Ausnahme).

Einfacher zu verstehen? – endrekursive Formulierung

```
private static
double posPower(double soFar, double base,  int exp) {
    // Q: exp >= 0
    // R: RESULTAT = soFar * base ^ exp
    if (exp == 0) {
        // Q1: exp == 0
        return soFar;
    }
    else if (exp % 2 == 0) {
        // Q2: exp ist gerade und > 0
        return posPower(soFar, base * base, exp / 2);
    }
    else {
        // Q3: exp ist ungerade und > 0
        return posPower(soFar * base, base, exp - 1);
    }
}
```

Mit dem Aufruf: `posPower(1.0, BASE, EXP)` gilt stets

```
// exp >= 0 & soFar * base ^ exp = BASE ^ EXP
```
Invariante

Diese Formulierung führt zur Iteration !

Iterativer Algorithmus(3)

```
private static double posPower(double base, int exp) {
    // Q: exp >= 0, BASE = base, EXP = exp
    double soFar = 1.0;
    // Invariante: BASE ^ EXP = soFar * base ^ exp
    while (exp != 0) {
        // Invariante && exp != 0
        if (exp % 2 == 0) {
            exp = exp / 2;
            base = base * base;
        }
        else {
            exp = exp - 1;
            soFar = soFar * base;
        }
        // Invariante
    }
    // BASE^EXP = soFar * base ^ exp && exp = 0 ⇒
    // R: soFar = BASE ^ EXP
    return soFar;
}
```

Laufzeit:

Problemgröße:	`exp`	(hier = Variante)
Anzahl der Durchläufe:	`2*ld(exp)`	also O(log *n*)

Entwicklung der Invarianten

base	exp	soFar	soFar*$base^{exp}$	BASE	EXP	$BASE^{EXP}$
2	10	1	*1024*	2	10	*1024*
4	5	1	*1024*			
4	4	4	*1024*	== *BASE ^ EXP*		
16	2	4	*1024*			
256	1	4	*1024*			
256	**0**	**1024**	*1024*			
	Variante		Invariante			

Korrektheit bei Klassen

Schleife: Der Schleifenkörper wird wiederholt durchlaufen, dabei werden Variable verändert: Schleifeninvariante

Objekt: Die Methoden eines Objekts werden wiederholt aufgerufen, dabei werden Variable des Objekts verändert: Repräsentationsinvariante, Klasseninvariante.

Die Klasseninvariante gilt nach Beendigung des Konstruktors, vor dem Eintritt in eine Methode und nach dem Verlassen einer Methode.

Beispiel (Klasse Bruch):

Klasseninvariante: `zaehler` und `nenner` bezeichnen die gekürzte Form eines Bruches. `nenner` ist immer positiv und größer als 0, `zaehler` kann auch negativ sein.

Bei Beachtung der Klasseninvariante lassen sich einzelne Methoden unabhängig voneinander entwickeln und testen!

Die Klasseninvariante beschreibt nur die Implementierung einer Klasse.

Klasseninvariante der Klasse Bruch (nur Beispiel - veränderbar !?)

```
public class Bruch {
    // INV: Bruch ist gekuerzt und nenner > 0
    private int zaehler;
    private int nenner;

    public Bruch(int z, int n) {
        zaehler = z;
        nenner = n;
        kuerzen();
        // INV
    }
    public boolean equals(Object other) {
        // INV
        if (! (other instanceof Bruch)) return false;
        Bruch b = (Bruch)other;
        return zaehler == b.zaehler && nenner == b.nenner;
        // INV (automatisch)
    }
    public void erhoeheUm(Bruch b) { //nur zum (schlechten) Beispiel
        // INV
        zaehler = zaehler*b.nenner + nenner*b.zaehler;
        nenner = nenner * b.nenner;
        kuerzen();
        // INV
    }
```

Automatischer Test von Invarianten mit assert-Anweisung

```
public class Bruch {
    // INV: Bruch ist gekuerzt und nenner > 0
    private int zaehler, nenner;

    public Bruch(int z, int n) {
        ...
        assert objectIsValid();
    }

    public void erhoeheUm(Bruch b) { //nur zum (schlechten) Beispiel
        assert objectIsValid();
        zaehler = zaehler*b.nenner + nenner*b.zaehler;
        nenner = nenner * b.nenner;
        kuerzen();
        assert objectIsValid();
    }

    private boolean objectIsValid() {
        return nenner > 0 && gcd(Math.abs(zaehler, nenner) == 1;
    }
```

Assert wird nur ausgeführt bei Aufruf von `java -ea !`
Wenn dann der Test fehlschlägt, wird eine Exception geworfen.

Nachlässiges Programmieren ist sehr teuer

1. Nichtbeachtung von Invarianten macht ein paar Methoden etwas kürzer:

```
public Bruch() {
    zaehler = nenner = 0;
    // INV gilt nicht !!
}

public void erhoeheUm(Bruch b) {
    zaehler = zaehler*b.nenner+nenner*b.zaehler;
    nenner = nenner * b.nenner;
    // INV gilt nicht !!
}
```

2. Die Verwendung der Klasse wird viel fehleranfälliger

```
/**
 * Vorsicht: vor dem Aufruf müssen die
 * beiden Brüche gekürzt werden!
 */
public boolean equals(Object other) { ... /* wie bisher */ }
```

3. Erschwert den Test und macht viele Methoden komplizierter.