



# *Algorithmen und Datenstrukturen*

## *Korrektheit von Algorithmen*

---

Matthias Teschner  
Graphische Datenverarbeitung  
Institut für Informatik  
Universität Freiburg

SS 09



# *Lernziele der Vorlesung*

- Algorithmen
  - Sortieren, Suchen, Optimieren
- Datenstrukturen
  - Repräsentation von Daten
  - Listen, Stapel, Schlangen, Bäume
- Techniken zum Entwurf von Algorithmen
  - Algorithmenmuster
  - Greedy, Backtracking, Divide-and-Conquer
- **Analyse von Algorithmen**
  - Korrektheit, Effizienz

# Analyse von Algorithmen



- **Korrektheit**
  - Ein korrekter Algorithmus stoppt (terminiert) für jede Eingabeinstanz mit der durch die Eingabe-Ausgabe-Relation definierten Ausgabe.
  - Ein inkorrekt Algorithmus stoppt nicht oder stoppt mit einer nicht durch die Eingabe-Ausgabe-Relation vorgegebenen Ausgabe.
- **Effizienz**
  - Bedarf an Speicherplatz und Rechenzeit
  - Wachstum (Wachstumsgrad, Wachstumsrate) der Rechenzeit bei steigender Anzahl der Eingabe-Elemente (Laufzeitkomplexität)



# Quelle

- Michael Gellner, "Der Umgang mit dem Hoare-Kalkül zur Programmverifikation", Universität Rostock

<http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/gellner.html>

[http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/michael/lehre/pt\\_2001/Hoare\\_akt\\_008.pdf](http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/michael/lehre/pt_2001/Hoare_akt_008.pdf)



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements



# *Motivation*

- Programmfehler sind teuer.
  - Prototype der Ariane-5-Rakete der ESA mit vier Satelliten wird eine Minute nach dem Start gesprengt (1996).
  - Fehler bei Typumwandlung in Ada
  - 12 Mrd. DM Entwicklung
  - 250 Mio. DM Start
  - 850 Mio. DM Satelliten
  - 600 Mio. DM Nachbesserungen
  - nächster erfolgloser Test nach 1,5 Jahren
  - erster kommerzieller, erfolgreicher Flug 1999

# Motivation



- Programmfehler kosten Menschenleben.
  - 1985 Bestrahlungsgerät Therac-25. Drei Tote.
  - 1991 Patriot-Rakete. 28 Tote.
  - 2007 vollautomatisches 35-mm-Flakgeschütz. 10 Tote.



# *Fehlerarten*

- syntaktisch
  - Kompilierung nicht erfolgreich oder Interpretation bricht ab
- **semantisch**
  - **Abweichung zwischen Spezifikation und Implementierung**
- weitere Fehler
  - Speicherlecks (Programm belegt kontinuierlich mehr Speicher)
  - fehlerhaftes Bedienkonzept (Programm reagiert auf Eingaben nicht, wie vom Benutzer erwartet)



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements

# Algorithmus



- wohldefinierte **Rechenvorschrift**, die eine Menge von Elementen als **Eingabe** verwendet und eine Menge von Elementen als **Ausgabe** erzeugt
- beschreibt eine Rechenvorschrift zum Erhalt einer durch die Formulierung eines Problems gegebenen **Eingabe-Ausgabe-Beziehung**

Algorithmus  $f$  : Eingabe  $\rightarrow$  Ausgabe



# Beispiel

## Rest bei ganzzahliger Division

- **Eingabe:** Dividend, Divisor ( $x \geq 0, y > 0$ )
- **Ausgabe:** ganzzahliger Rest  $r$

```
function rest (x:integer; y:integer) : integer;  
var q, r : integer;  
begin  
    q := 0; r := x;  
    while r >= y do  
        begin  
            r := r-y; q := q+1;  
        end;  
    rest := r;  
end;
```

Pascal !

- Liefert die Funktion immer den korrekten Rest für gültige Eingaben? Stimmt die Spezifikation mit der Implementierung überein?



# Korrektheit

- Ein Programm  $S$  ist bezüglich einer **Vorbedingung**  $P$  und einer **Nachbedingung**  $Q$  **partiell korrekt**, wenn für jede Eingabe, die  $P$  erfüllt, das Ergebnis auch  $Q$  erfüllt, falls das Programm terminiert.
- Ein Programm ist **total korrekt**, wenn es partiell korrekt ist und für jede Eingabe, die  $P$  erfüllt, terminiert.
- **Beispiel:**
  - Vorbedingung: gültige Werte für Divisor und Dividend
  - Nachbedingung: Programm liefert den ganzzahligen Rest
- Wir beschränken uns auf partielle Korrektheit.



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements



# Hoare-Kalkül

- formales System zum Beweisen der Korrektheit imperativer Programme
- von Sir Charles Antony Richard Hoare (1969) entwickelt, mit Beiträgen von Lauer (1971) und Dijkstra (1982)
- besteht aus Axiomen und Ableitungsregeln für alle Konstrukte einer einfachen imperativen Programmiersprache
  - elementare Operationen
  - sequenzielle Ausführung
  - bedingte Ausführung
  - Schleife



# Hoare-Tripel

- zentrales Element des Hoare-Kalküls
- beschreibt die Zustandsveränderung durch eine Berechnung

$$\{P\} S \{Q\}$$

- P und Q
  - sind Zusicherungen (Vor- und Nachbedingung).
  - sind Formeln der Prädikatenlogik.
- S ist ein Programmsegment.
- Wenn Programmzustand P vor der Ausführung von S gilt, dann gilt Q nach der Ausführung von S.
- Beispiel:  $\{x + 1 = 43\} y := x + 1 \{y = 43\}$



# Prinzip der Verifikation

$$\{P\} S_1 \{A_1\} S_2 \{A_{n-1}\} S_n \{Q\}$$

- $\{P\}$  und  $\{Q\}$ 
  - sind Vor- und Nachbedingung des Programms  $S_1, S_2, \dots, S_n$
  - stellen die Spezifikation der Ein- / Ausgabe-Relation dar
  - sind definierte, bekannte Zusicherungen
- $A_1, A_2, \dots, A_{n-1}$ 
  - sind unbekannte Zusicherungen
  - werden durch das Hoare-Kalkül bestimmt
  - beispielsweise wird  $A_{n-1}$  über eine Hoare-Regel für das Hoare-Tripel  $\{A_{n-1}\} S_n \{Q\}$  mit bekanntem  $S_n$  und bekanntem  $\{Q\}$  bestimmt
- Tritt dabei kein Widerspruch auf, ist  $S_1, S_2, \dots, S_n$  partiell korrekt bezüglich  $\{P\}$  und  $\{Q\}$ .



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements

# Aussagenlogik



- beschäftigt sich mit Aussagen und deren Verknüpfung durch logische Operatoren
- Motivation:
  - Zusicherungen müssen formal wahr oder falsch sein.
  - Notwendigkeit von Auswertungen oder Umformungen logischer Ausdrücke.

A	$\neg A$
wahr	falsch
falsch	wahr

**Negation**  
Verneinung einer Aussage

A	B	$A \wedge B$
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

**Konjunktion**  
Und

# Aussagenlogik



A	B	$A \vee B$
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

**Disjunktion**  
Nichtausschließendes Oder

A	B	$A \oplus B$
wahr	wahr	falsch
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

**Kontravalenz**  
Ausschließendes Oder  
Exklusives Oder  
Entweder Oder

A	B	$A \rightarrow B$
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	wahr
falsch	falsch	wahr

**Implikation**  
Wenn dann

A	B	$A \leftrightarrow B$
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	wahr

**Äquivalenz**  
Genau dann wenn

# Aussagenlogik



- Umformung logischer Aussagen
- beispielsweise mit Hilfe der De Morganschen Regeln

$$\neg ( A \vee B ) \leftrightarrow \neg A \wedge \neg B$$

$$\neg ( A \wedge B ) \leftrightarrow \neg A \vee \neg B$$



# Prädikatenlogik

- Erweiterung der Aussagenlogik um Quantoren (Operatoren)
- Existenzquantor:  $\exists x$ 
  - für mindestens ein Element  $x$  gilt
  - es existiert ein Element  $x$ , für das gilt
- Allquantor:  $\forall x$ 
  - für alle Elemente  $x$  gilt
- Beispiel  
 $x \in \mathbb{N}$   
 $\exists x: A(x) \leftrightarrow A(0) \vee A(1) \vee A(2) \dots$   
 $\forall x: A(x) \leftrightarrow A(0) \wedge A(1) \wedge A(2) \dots$



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements



# Hoare-Regeln

- Hoare-Kalkül besteht aus Menge von Regeln
- Regeln bestehen aus Prämissen und Schlussfolgerungen

Prämisse 1
Prämisse 2
...
Prämisse n
-----
Konklusion

- Wenn alle Prämissen erfüllt sind, dann folgt die Konklusion (Schlussfolgerung).
- Ähnlich zur Implikation, aber keine Aussage, sondern eine Schlussregel.

# *Beispiel zum Aufbau einer Regel*



Pokerregeln gelten

Ich habe Pik 10

Ich habe Pik Bube

Ich habe Pik Dame

Ich habe Pik König

Ich habe Pik As

Nur ich habe eine Straße

---

Ich gewinne das Spiel



# Beispiele für Schlussregeln

- Modus tollens

$$\begin{array}{c} A \Rightarrow B \\ \neg B \\ \hline \neg A \end{array}$$

Wenn ich ein Pik Ass habe, dann habe ich eine Pik Straße ( $A \Rightarrow B$ )  
Ich habe keine Pik Straße ( $\neg B$ )  

---

Ich habe kein Pik Ass ( $\neg A$ )

- Modus ponens

$$\begin{array}{c} A \Rightarrow B \\ A \\ \hline B \end{array}$$

Wenn ich ein Pik Ass habe, dann habe ich eine Pik Straße ( $A \Rightarrow B$ )  
Ich habe ein Pik Ass ( $A$ )  

---

Ich habe eine Pik Straße ( $B$ )

# *Schlussregeln für Hoare-Tripel*



- Hoare-Tripel  $\{ P \} S \{ Q \}$
- Axiome und Ableitungsregeln für die Zusicherungen  $P$  und  $Q$  für alle Konstrukte  $S$  einer einfachen imperativen Programmiersprache
  - elementare Operationen
  - sequenzielle Ausführung
  - bedingte Ausführung
  - Schleife



# *Axiom der leeren Anweisung*

- Für eine Programmsequenz, die den Zustand von Variablen nicht verändert, bleibt die Vorbedingung auch nach Ausführung der Sequenz gültig.

$\{ P \} \text{ NOP } \{ P \}$

- keine Prämisse, gilt immer

# Zuweisungsaxiom



- Regel für Zuweisungen bzw. Ergibt-Anweisungen

$$\{ P^x_E \} x := E \{ P \}$$

- In  $P$  wird  $x$  durch  $E$  substituiert, um  $P^x_E$  zu erhalten.
- keine Prämisse, gilt immer, muss nicht begründet werden
- wenn Nachbedingung bekannt, dann kann Vorbedingung abgeleitet werden
- Beispiel:

Vorbedingung:  $\{ ? \}$   $\{ x+1 > a \}$

Programmsequenz:  $x := x+1;$   $\Rightarrow$   $x := x+1;$

Nachbedingung:  $\{ x > a \}$   $\{ x > a \}$

# Zuweisungsaxiom



- Beispiel:

$$\begin{array}{ccc} \{?\} & & \{x = (q + 1) \cdot y - (r - y) \wedge r - y > 0\} \\ r := r - y; & \Rightarrow & r := r - y; \\ \{x = (q + 1) \cdot y - r \wedge r > 0\} & & \{x = (q + 1) \cdot y - r \wedge r > 0\} \end{array}$$

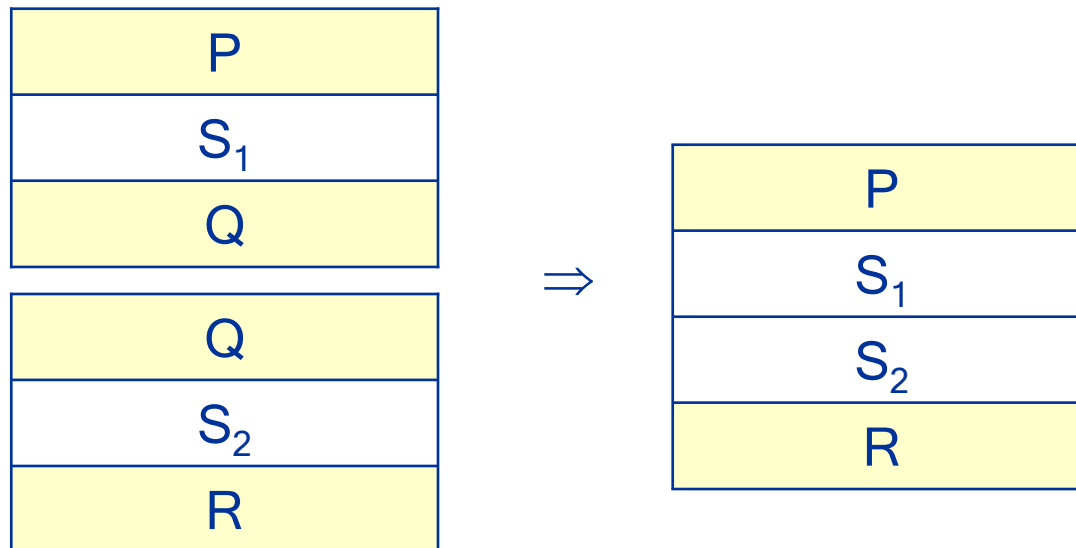
- Im unteren Ausdruck wird  $r$  durch  $r-y$  ersetzt, um den oberen Ausdruck zu erhalten.
- Prinzip der Probe: Beginne mit dem Ergebnis, rechne rückwärts und hoffe, keinen Widerspruch zu finden.



# Regel für Anweisungsfolgen

- Zwei aufeinanderfolgende Anweisungen können zu einem Programmstück zusammengefasst werden, wenn die Nachbedingung der ersten Anweisung äquivalent zur Vorbedingung der zweiten Anweisung ist.

$$\frac{\begin{array}{l} \{P\} S_1 \{Q\} \\ \{Q\} S_2 \{R\} \end{array}}{\{P\} S_1 ; S_2 \{R\}}$$





# Regel für Anweisungsfolgen

- erlaubt das zeilenweise (befehlsweise, anweisungsweise) Auswerten von Anweisungssequenzen

$$\frac{r := r - y;}{q := q + 1; \quad \{x = y \cdot q + r\}} \Rightarrow \frac{\{x = y \cdot (q+1) + r - y\}}{r := r - y; \quad \{x = y \cdot (q+1) + r\}} \Leftrightarrow \frac{\{x = y \cdot (q+1) + r - y\}}{r := r - y; \quad q := q + 1; \quad \{x = y \cdot q + r\}}$$

- ermöglicht das Einbringen von Zusicherungen zwischen Anweisungen

# Regel für bedingte Anweisungen



- Anweisung  $S_1$  führt unter Bedingung  $B$  von Zustand  $P$  zu Zustand  $R$
- Anweisung  $S_2$  führt unter Bedingung  $\neg B$  von Zustand  $P$  zu Zustand  $R$

$\{ P \wedge B \} S_1 \{ R \}$
$\{ P \wedge \neg B \} S_2 \{ R \}$
$\{ P \} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{ R \}$

P	
B ?	
true	false
$P \wedge B$	$P \wedge \neg B$
$S_1$	$S_2$
R	R
R	

# Regel für bedingte Anweisungen



- Beispiel:

P	
B ?	
true	false
$P \wedge B$	$P \wedge \neg B$
$S_1$	$S_2$
R	R
R	

$\{x > 0\}$	
if (x==5)	
then	else
$\{x > 0 \wedge x = 5\}$	$\{x > 0 \wedge x \neq 5\}$
x := 0;	x := 1;
$\{x = 0 \vee x = 1\}$	$\{x = 0 \vee x = 1\}$
$\{x = 0 \vee x = 1\}$	

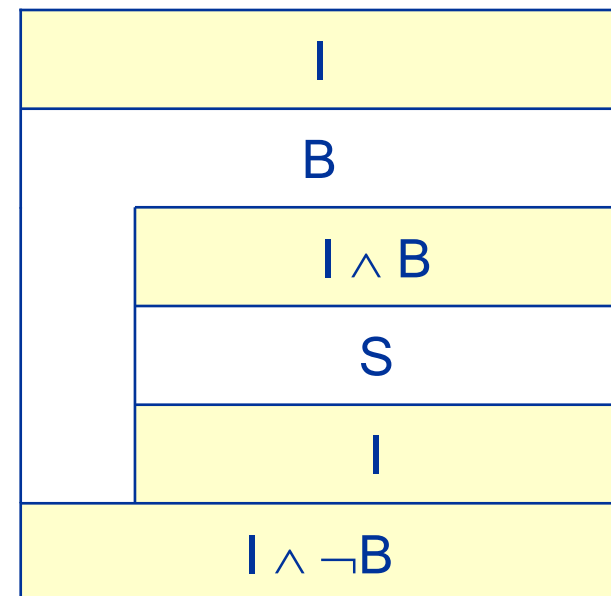
- $\{x > 0\}$   
if (x==5) then x:=1; else x:=0;  
 $\{x = 0 \vee x = 1\}$



# Regel für Schleifen

- Unter Bedingung B wird Anweisung S wiederholt ausgeführt, bis  $\neg B$  gilt
- Zur Verifikation von Schleifen muss eine Invariante I gefunden werden (Zusicherung, die durch die Schleife nicht verändert wird).
- sichert nicht die Terminierung zu

$$\frac{\{I \wedge B\} S \{I\}}{\{I\} \text{while } B \text{ do } S \{I \wedge \neg B\}}$$





# *Schleifeninvariante*

- Schleifenanweisungen verändern Zustände einzelner Variablen.
- Es muss eine Verknüpfung (Funktion) dieser Variablen gefunden werden, die unverändert bleibt.
- Tabellarische Aufstellung aller Variablen für die ersten Schleifendurchläufe kann hilfreich zum Finden dieser Funktion sein.
- Berücksichtigung der Motivation für die Entwicklung der Schleife kann hilfreich sein.

# Regeln für weitere Sprachkonstrukte



- Oft können weitere Sprachkonstrukte auf die zuvor behandelten Konstrukte abgebildet werden.
- `for do`  $\Rightarrow$  `while do`
- `repeat until`  $\Rightarrow$  `while do`
- `case`  $\Rightarrow$  `if then else`



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - **Verifikation**
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements



# *Prinzip der Verifikation*

- Formulierung der Nachbedingung des ganzen Programms (formale Beschreibung, wozu das Programm dient)
- Formulierung der Vorbedingung des Programms (formale Beschreibung der Anforderung an die Eingabe)
- Anwendung der Hoare-Regeln von unten nach oben
- Schleifen werden von außen nach innen ausgewertet.
- Wenn Hoare-Klauseln für alle Anweisungen ermittelt sind, liegt eine Beweisskizze vor.
- Enthalten die Klauseln keinen Widerspruch, ist das Programm partiell korrekt bezüglich der Vor- und Nachbedingung.



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Potenzieren
  - Suche eines Elements



# *Rest bei ganzzahliger Division*

- **Eingabe:** Dividend, Divisor
- **Ausgabe:** ganzzahliger Rest

```
function rest (x:integer; y:integer) : integer;  
var q, r : integer;  
begin  
    q := 0; r := x;  
    while r >= y do  
        begin  
            r := r-y; q := q+1;  
        end;  
    rest := r;  
end
```



# *Relevanter Programmteil*

	<code>q := 0;</code>
	<code>r := x;</code>
<code>while r &gt;= y do begin</code>	
	<code>    r := r - y;</code>
<code>end</code>	<code>    q := q + 1;</code>



# Vor – und Nachbedingung

- sinnvolle Eingaben:  $x \geq 0 \wedge y > 0$  (keine Division durch Null)
- Der Rest ist ermittelt, wenn nach  $q$  Schleifendurchläufen  $q \cdot y$  vom Eingabewert  $x$  subtrahiert wurde:  $r = x - q \cdot y$ .
- Überprüfen, ob diese Formulierung wirklich dem Zweck des Programms entspricht!

$\{P\} \quad \{x \geq 0 \wedge y > 0\}$	
$q := 0;$	
$r := x;$	
while $r \geq y$ do begin	
	$r := r - y;$
end	$q := q + 1;$
$\{Q\} \quad \{x = q \cdot y + r \wedge r \geq 0 \wedge r < y\}$	



# Verifikation der Schleife

$\{ I \}$	
while $r \geq y$ do begin	$\{ I \wedge B \}$
	$r := r - y;$
	$q := q + 1;$
end	$\{ I \}$
$\{ I \wedge \neg B \}$	

- Ermitteln von  $\{ I \}$  und  $\{ B \}$
- $\{ B \}$  ist identisch mit der while-Klausel:  $r \geq y$
- Als  $\{ I \}$  kann in diesem Fall die Nachbedingung des Programms ausprobiert werden:  $x = q \cdot y + r \wedge r \geq 0$

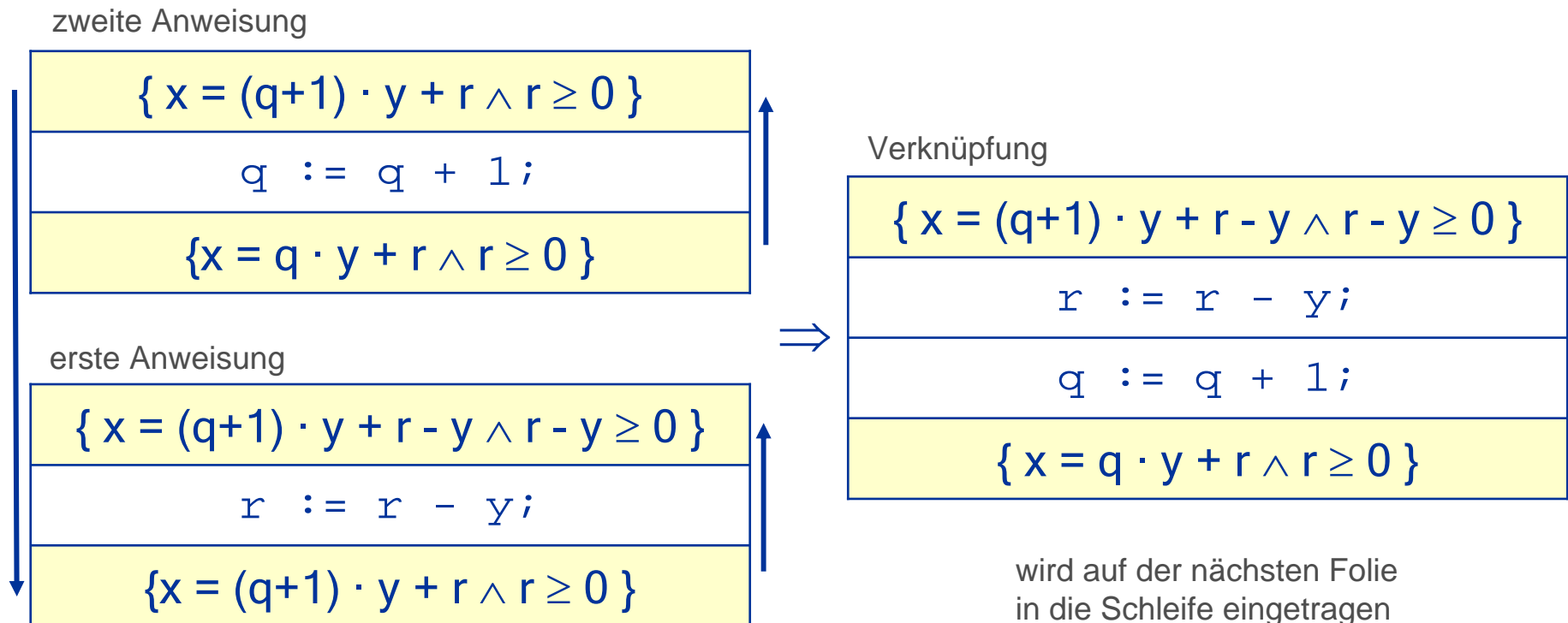


# Verifikation der Schleife

	$\{I\} \quad \{x = q \cdot y + r \wedge r \geq 0\}$
while $r \geq y$ do begin	
	$\{I \wedge B\} \quad \{x = q \cdot y + r \wedge r \geq 0 \wedge r \geq y\}$
	$r := r - y;$
	$q := q + 1;$
end	$\{I\} \quad \{x = q \cdot y + r \wedge r \geq 0\}$
	$\{I \wedge \neg B\} \quad \{x = q \cdot y + r \wedge r \geq 0 \wedge r < y\}$

- Als Nachbedingung der letzten Schleifenanweisung wird  $\{I\}$  eingesetzt.  $\{B\}$  muss nicht gelten!
- Als Vorbedingung der ersten Schleifenanweisung wird  $\{I \wedge B\}$  eingesetzt.

# Verifikation der inneren Anweisungen





# Verifikation der Schleife

- Prüfung, ob die ermittelte Vorbedingung für die Zuweisungen mit der Schleifeninvariante und der Schleifenbedingung äquivalent ist

	$\{I\} \quad \{x = q \cdot y + r \wedge r \geq 0\}$
while $r \geq y$ do begin	
	$\{I \wedge B\} \quad \{x = q \cdot y + r \wedge r \geq 0 \wedge r \geq y\}$
	$\{x = (q+1) \cdot y + r - y \wedge r - y \geq 0\}$
	$r := r - y;$
	$q := q + 1;$
end	$\{I\} \quad \{x = q \cdot y + r \wedge r \geq 0\}$
	$\{I \wedge \neg B\} \quad \{x = q \cdot y + r \wedge r \geq 0 \wedge r < y\}$



# Verifikation der Schleife

- auf Widerspruch testen

$$\frac{\{I \wedge B\} \quad \{x = q \cdot y + r \wedge r \geq 0 \wedge r \geq y\}}{\{x = (q+1) \cdot y + r - y \wedge r - y \geq 0\}}$$

- $x = (q+1) \cdot y + r - y = q \cdot y + r$
- $r - y \geq 0 \Leftrightarrow r \geq y \Rightarrow r \geq 0$
- kein Widerspruch

# Zwischenstand



$\{x \geq 0 \wedge y > 0\}$	
$q := 0;$	
$r := x;$	
$\{x = q \cdot y + r \wedge r \geq 0\}$	
while $r \geq y$ do begin	
	$r := r - y;$
end	$q := q + 1;$
$\{x = q \cdot y + r \wedge r \geq 0 \wedge r < y\}$	

Diese beiden Anweisungen  
müssen noch behandelt werden.

# Verifikation der Initialisierung



zweite Anweisung

$$\{x = q \cdot y + x \wedge x \geq 0\}$$

$r := x;$

$$\{x = q \cdot y + r \wedge r \geq 0\}$$

erste Anweisung

$$\{x = 0 \cdot y + x \wedge x \geq 0\}$$

$q := 0;$

$$\{x = q \cdot y + x \wedge x \geq 0\}$$

$\Rightarrow$

Verknüpfung

$$\{x = 0 \cdot y + x \wedge x \geq 0\}$$

$q := 0;$

$r := x;$

$$\{x = q \cdot y + r \wedge r \geq 0\}$$

# Fertig



$\{x \geq 0 \wedge y > 0\}$	
$\{x = 0 \cdot y + x \wedge x \geq 0\}$	
$q := 0;$	
$r := x;$	
$\{x = q \cdot y + r \wedge r \geq 0\}$	
while $r \geq y$ do begin	
	$r := r - y;$
end	$q := q + 1;$
$\{x = q \cdot y + r \wedge r \geq 0 \wedge r < y\}$	

Kein Widerspruch.



# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - **Quadrat einer Zahl**
  - Potenzieren
  - Suche eines Elements



# Quadrat einer Zahl

- **Eingabe:** natürliche Zahl  $n$
- **Ausgabe:** Quadrat der Eingabe  $quad = n^2$

```
function quad (n:integer) : integer;  
var i, k, y : integer;  
begin  
    i := 0; k := -1; y := 0;  
    while i < n do  
        begin  
            i := i + 1; k := k + 2; y := y + k;  
        end;  
    quad := y;  
end;
```

Wir beweisen lediglich, dass die Implementierung mit der Spezifikation übereinstimmt unter der Bedingung, dass die unten aufgeführte Idee / Motivation korrekt ist. Ob die dort angegebene Summe mit dem Quadrat von  $n$  übereinstimmt, wird nicht überprüft.

- **Motivation**  $n^2 = \sum_{i=1}^n (2i - 1) = 1 + 3 + 5 + \dots + (2n - 1)$

# Relevanter Programmteil

## Vor- und Nachbedingung



$\{n \geq 0\}$	
$i := 0;$	
$k := -1;$	
$y := 0;$	
while $i < n$ do begin	
	$i := i + 1;$
	$k := k + 2;$
end	$y := y + k;$
$\{y = n^2\}$	



# Verifikation der Schleife

$\{ I \}$	
while $i < n$ do begin	
	$\{ I \wedge B \}$
	$i := i + 1;$
	$k := k + 2;$
	$y := y + k;$
end	$\{ I \}$
$\{ I \wedge \neg B \}$	

- Ermitteln von  $\{ I \}$  und  $\{ B \}$
- $\{ B \}$  ist identisch mit der while-Klausel:  $i < n$
- $\{ I \}$  wird in diesem Fall durch Betrachten einiger beispielhafter Schleifendurchläufe ermittelt.



# Schleifeninvariante

- Status der Variablen während des Schleifendurchlaufs (n=5)

Variablen / Schleifendurchlauf	i	k	y	n
Schleifen-Eintritt	0	-1	0	5
Erster Durchlauf	1	1	1	5
Zweiter Durchlauf	2	3	4	5
Dritter Durchlauf	3	5	9	5
Vierter Durchlauf	4	7	16	5
Fünfter Durchlauf	5	9	25	5

- Folgende Invariante kann abgeleitet werden:  
 $\{ k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \}$



# Verifikation der Schleife

	$\{I\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
while $i < n$ do begin	
	$\{I \wedge B\} \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge i < n\}$
	$i := i + 1;$
	$k := k + 2;$
	$y := y + k;$
end	$\{I\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
	$\{I \wedge \neg B\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge \neg(i < n)\}$

Falls bezüglich der Invariante  $I$  kein Widerspruch auftaucht, muss immer noch die Bedingung  $I \wedge \neg B$  gegen die Nachbedingung  $Q$  getestet werden.

# Verifikation der Schleife

## Innere Anweisungen



	$\{I\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
while $i < n$ do begin	
	$\{I \wedge B\} \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge i < n\}$
	$\{k + 2 = 2 \cdot (i + 1) - 1 \wedge y + k + 2 = (i + 1)^2 \wedge i + 1 \leq n\}$
	$i := i + 1;$
	$\{k + 2 = 2 \cdot i - 1 \wedge y + k + 2 = i^2 \wedge i \leq n\}$
	$k := k + 2;$
	$\{k = 2 \cdot i - 1 \wedge y + k = i^2 \wedge i \leq n\}$
	$y := y + k;$
end	$\{I\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
	$\{I \wedge \neg B\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge \neg(i < n)\}$



# Verifikation der Schleife

- Vorbedingung der ersten inneren Anweisung und Schleifeninvariante auf Widerspruch testen

$\{ I \wedge B \} \quad \{ k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge i < n \}$
$\{ k + 2 = 2 \cdot (i+1) - 1 \wedge y + k + 2 = (i+1)^2 \wedge i + 1 \leq n \}$

- $k = 2 \cdot (i+1) - 1 - 2 = 2 \cdot i - 1$
- $y = (i+1)^2 - k - 2 = i^2 + 2 \cdot i + 1 - k - 2 = i^2 + k - k = i^2$
- $i + 1 \leq n \Rightarrow i < n$
- kein Widerspruch

# Verifikation der Schleife mit einem Implementierungsfehler



	$\{I\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
while $i < n$ do begin	
	$\{I \wedge B\} \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge i < n\}$
	$\{k + 1 = 2 \cdot (i + 1) - 1 \wedge y + k + 1 = (i + 1)^2 \wedge i + 1 \leq n\}$
	$i := i + 1;$
	$\{k + 1 = 2 \cdot i - 1 \wedge y + k + 1 = i^2 \wedge i \leq n\}$
	$k := k + 1;$
	$\{k = 2 \cdot i - 1 \wedge y + k = i^2 \wedge i \leq n\}$
	$y := y + k;$
end	$\{I\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
	$\{I \wedge \neg B\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge \neg(i < n)\}$

# Verifikation der Schleife mit einem Implementierungsfehler



- Vorbedingung der ersten inneren Anweisung und Schleifeninvariante auf Widerspruch testen

$\{I \wedge B\} \quad \{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n \wedge i < n\}$
$\{k + 1 = 2 \cdot (i+1) - 1 \wedge y + k + 1 = (i+1)^2 \wedge i + 1 \leq n\}$

- $k = 2 \cdot (i+1) - 1 - 1 = 2 \cdot i \neq 2 \cdot i - 1 \Rightarrow$  Widerspruch



# Behandlung der Initialisierung

$\{n \geq 0\}$
$\{-1 = 2 \cdot 0 - 1 \wedge 0 = 0^2 \wedge 0 \leq n\}$
$i := 0;$
$\{-1 = 2 \cdot i - 1 \wedge 0 = i^2 \wedge i \leq n\}$
$k := -1;$
$\{k = 2 \cdot i - 1 \wedge 0 = i^2 \wedge i \leq n\}$
$y := 0;$
$\{k = 2 \cdot i - 1 \wedge y = i^2 \wedge i \leq n\}$
while $i < n$ do begin
$i := i + 1;$
$k := k + 2;$
end $y := y + k;$
$\{y = n^2\}$

In Ordnung.





# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - **Potenzieren**
  - Suche eines Elements



# Berechnung der Potenz $x^n$

- **Eingabe:** ganzzahlige Werte  $x$  und  $n$  mit  $n \geq 0$
- **Ausgabe:**  $x^n$

```
function exp (x:integer; n:integer):integer;  
var k, p, y : integer;  
begin  
    k := n; p := x; y := 1;  
    while k>0 do  
        begin  
            if (k mod 2 == 0)  
                then begin  
                    p := p * p; k:= k / 2;  
                end  
            else begin  
                y := y * p; k := k - 1;  
            end;  
        end;  
    exp := y;  
end;
```

wenn  $k$  gerade, dann Basis quadrieren, Exponent halbieren:  $x^k = (x^2)^{k/2}$

wenn  $k$  ungerade, dann Basis mit Zwischenergebnis für den Exponent multiplizieren, dafür  $k$  um 1 reduzieren :  $x^k = x x^{k-1}$



# Relevanter Programmteil

## Vor- und Nachbedingung

$\{n \geq 0\}$	
$k := n;$	
$p := x;$	
$y := 1;$	
while $k > 0$ do begin	
if ( $k \bmod 2 == 0$ )	
then	else
$p := p * p;$	$y := y * p;$
$k := k / 2;$	$k := k - 1;$
end	
$\{y = x^n\}$	

# Verifikation der Schleife / Schleifeninvariante



- Status der Variablen während des Schleifendurchlaufs (n=10)

Variablen / Schleifendurchlauf	y	p	k	x	n
Schleifen-Eintritt	1	x	10	x	10
Erster Durchlauf	1	x <sup>2</sup>	5	x	10
Zweiter Durchlauf	x <sup>2</sup>	x <sup>2</sup>	4	x	10
Dritter Durchlauf	x <sup>2</sup>	x <sup>4</sup>	2	x	10
Vierter Durchlauf	x <sup>2</sup>	x <sup>8</sup>	1	x	10
Fünfter Durchlauf	x <sup>10</sup>	x <sup>8</sup>	0	x	10

- Folgende Invariante kann abgeleitet werden:  
 $\{ x^n = y \cdot p^k \wedge k \geq 0 \}$



# Verifikation der Schleife ...

$\{I\} \{x^n = y \cdot p^k \wedge k \geq 0\}$	
while k > 0 do begin	
$\{I \wedge B\} \{x^n = y \cdot p^k \wedge k \geq 0 \wedge k > 0\}$	
if (k mod 2 == 0)	
then	else
p := p * p;	y := y * p;
k := k / 2;	k := k - 1;
end	$\{I\} \{x^n = y \cdot p^k \wedge k \geq 0\}$
$\{I \wedge \neg B\} \{x^n = y \cdot p^k \wedge k \geq 0 \wedge k \leq 0\}$	
$\{y = x^n\}$	

kein Widerspruch  
zwischen  $\{I \wedge \neg B\}$  und  
der Nachbedingung  
des Programms  $\{Q\}$

# ... benötigt die Verifikation der bedingten Anweisung



$\{x^n = y \cdot p^k \wedge k \geq 0 \wedge k > 0\}$	
if (k mod 2 == 0)	
then	else
$\{x^n = y \cdot p^k \wedge k \geq 0 \wedge k > 0 \wedge k \bmod 2 = 0\}$	$\{x^n = y \cdot p^k \wedge k \geq 0 \wedge k > 0 \wedge k \bmod 2 \neq 0\}$
p := p * p;	y := y * p;
k := k / 2;	k := k - 1;
$\{x^n = y \cdot p^k \wedge k \geq 0\}$	$\{x^n = y \cdot p^k \wedge k \geq 0\}$
$\{x^n = y \cdot p^k \wedge k \geq 0\}$	



## Fall 1: $k \bmod 2 = 0$

$\{ x^n = y \cdot p^k \wedge k \geq 0 \wedge k > 0 \wedge k \bmod 2 = 0 \}$
$\{ x^n = y \cdot (p \cdot p)^{k/2} \wedge k/2 \geq 0 \wedge k \bmod 2 = 0 \}$
$p := p * p;$
$\{ x^n = y \cdot p^{k/2} \wedge k/2 \geq 0 \wedge k \bmod 2 = 0 \}$
$k := k / 2;$
$\{ x^n = y \cdot p^k \wedge k \geq 0 \}$

Die von unten hergeleitete Anforderung widerspricht nicht der von oben kommenden Zusicherung.

$k \bmod 2 = 0$  muss hier aufgrund des Datentyps integer gefordert werden.



## Fall 2: $k \bmod 2 \neq 0$

$\{ x^n = y \cdot p^k \wedge k \geq 0 \wedge k > 0 \wedge k \bmod 2 \neq 0 \}$
$\{ x^n = y \cdot p \cdot p^{k-1} \wedge k-1 \geq 0 \}$
$y := y * p;$
$\{ x^n = y \cdot p^{k-1} \wedge k-1 \geq 0 \}$
$k := k - 1;$
$\{ x^n = y \cdot p^k \wedge k \geq 0 \}$

Die von unten hergeleitete Anforderung widerspricht nicht der von oben kommenden Zusicherung. Für  $k \bmod 2$  gibt es keine Anforderung.  $k > 0$  wird von unten gefordert und von oben durch  $k \geq 0 \wedge k > 0$  auch zugesichert.



# Behandlung der Initialisierung

$\{ n \geq 0 \}$
$\{ x^n = x^n \wedge n \geq 0 \}$
$k := n;$
$\{ x^n = x^k \wedge k \geq 0 \}$
$p := x;$
$\{ x^n = p^k \wedge k \geq 0 \}$
$y := 1;$
$\{ x^n = y \cdot p^k \wedge k \geq 0 \}$
while $k > 0$ do begin
if ( $k \bmod 2 == 0$ )
then
$p := p * p;$
else
$y := y * p;$
$k := k / 2;$
$k := k - 1;$
end
$\{ y = x^n \}$

Sieht gut aus.





# Überblick

- Motivation
- Korrektheit
- Hoare-Kalkül
  - Einführung
  - Grundlagen
  - Regeln
  - Verifikation
- Beispiel
  - ganzzahliger Rest
  - Quadrat einer Zahl
  - Suche eines Elements



# Suche eines Elements

- **Eingabe:** Menge elem mit n Elementen, zu suchendes Element x
- **Ausgabe:** Index des gesuchten Elements

```
function search (x:value; elem:vector):integer;  
var i : integer;  
begin  
    i := 1;  
    while elem [i] <> x do  
        begin  
            i := i + 1;  
        end;  
    search := i;  
end;
```

# Relevanter Programmteil Vor- und Nachbedingung



$\{x \in \text{elem}[1..n]\}$	
$i := 1;$	
while elem[i] <> x do begin	
end	$i := i + 1;$
$\{\text{elem}[i] = x\}$	



# Verifikation der Schleife

$\{x \in \text{elem}[1..n]\}$	
$i := 1;$	
$\{I\} \quad \{x \in \text{elem}[i..n]\}$	
while elem[i] <> x do begin	
	$\{I \wedge B\} \quad \{x \in \text{elem}[i..n] \wedge \text{elem}[i] \neq x\}$
	$i := i + 1;$
end	$\{I\} \quad \{x \in \text{elem}[i..n]\}$
$\{I \wedge \neg B\} \quad \{x \in \text{elem}[i..n] \wedge \neg(\text{elem}[i] \neq x)\}$	
$\{\text{elem}[i] = x\}$	

# Verifikation der inneren Anweisung



	$\{x \in \text{elem} [ 1..n ]\}$
	$i := 1;$
	$\{I\} \quad \{x \in \text{elem} [ i..n ]\}$
while elem[i] <> x do begin	
	$\{x \in \text{elem} [ i..n ] \wedge \text{elem} [ i ] \neq x\}$
	$\{x \in \text{elem} [ i+1..n ]\}$
	$i := i + 1;$
end	$\{x \in \text{elem} [ i..n ]\}$
	$\{I \wedge \neg B\} \quad \{x \in \text{elem} [ i..n ] \wedge \neg (\text{elem} [ i ] \neq x)\}$
	$\{\text{elem} [ i ] = x\}$



# Verifikation der Schleife

- auf Widerspruch testen

$$\{ x \in \text{elem} [ i..n ] \wedge \text{elem} [ i ] \neq x \}$$

$$\{ x \in \text{elem} [ i+1..n ] \}$$

- $x \in \text{elem} [ i+1..n ] \Rightarrow x \notin \text{elem} [ 1.. i ] \Rightarrow x \neq \text{elem} [ i ]$
- $x \in \text{elem} [ i+1..n ] \Rightarrow x \in \text{elem} [ i..n ]$
- kein Widerspruch
  
- ... Behandlung der Initialisierung ... fertig

# *Zusammenfassung*



- Motivation
  - Korrektheit
  - Hoare-Kalkül
  - Beispiele
- 
- "Beware of bugs in the above code;  
I have only proved it correct, not tried it."  
Donald Ervin Knuth



# *Weitere Beispiele*

- Michael Gellner, "Der Umgang mit dem Hoare-Kalkül zur Programmverifikation", Universität Rostock

[http://wwwswt.informatik.uni-rostock.de/  
deutsch/Mitarbeiter/michael/lehre/  
pt\\_2001/Hoare\\_akt\\_008.pdf](http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/michael/lehre/pt_2001/Hoare_akt_008.pdf)

- Vorlesungsaufzeichnung SS 2008



# Nächstes Thema

- **Korrektheit**
  - Ein korrekter Algorithmus stoppt (terminiert) für jede Eingabeinstanz mit der durch die Eingabe-Ausgabe-Relation definierten Ausgabe.
  - Ein inkorrekt Algorithmus stoppt nicht oder stoppt mit einer nicht durch die Eingabe-Ausgabe-Relation vorgegebenen Ausgabe.
- **Effizienz**
  - Bedarf an Speicherplatz und Rechenzeit
  - Wachstum (Wachstumsgrad, Wachstumsrate) der Rechenzeit bei steigender Anzahl der Eingabe-Elemente (Laufzeitkomplexität)