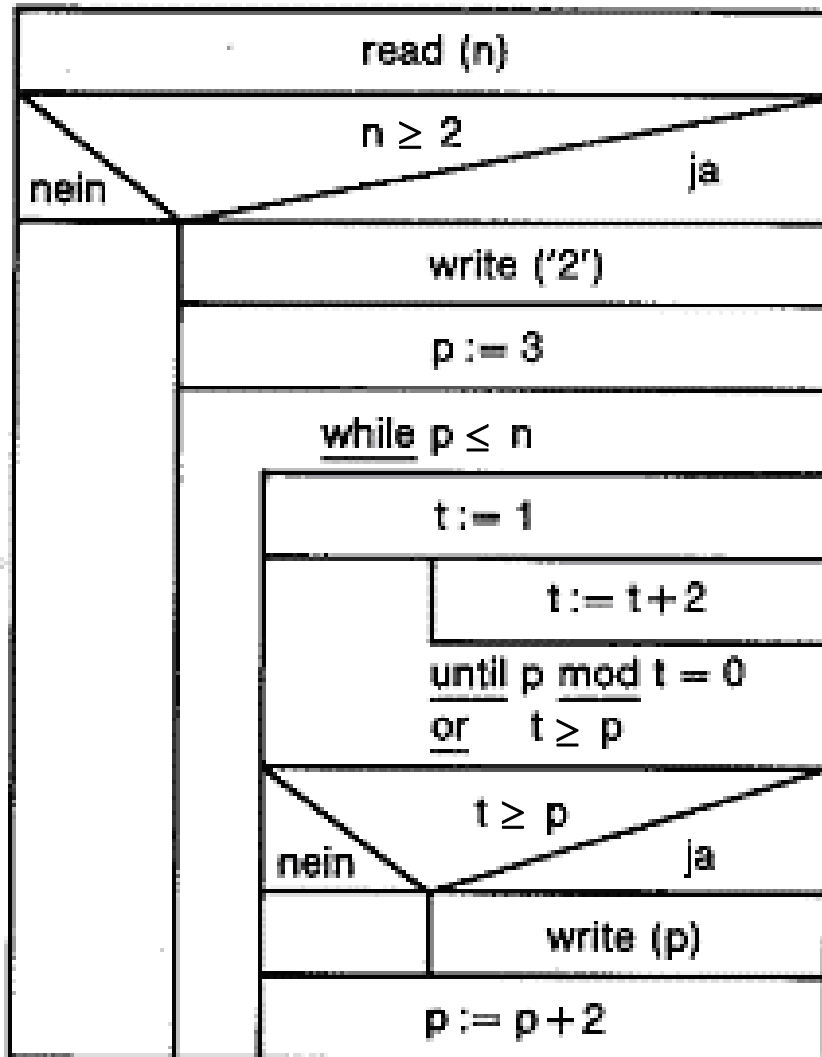


# Beispiel: Primzahlberechnung



```

class Prime {
    static void primesUpTo(int n) {
        if (n >= 2) {
            System.out.println(2);
            int p = 3;
            while (p <= n) {
                int t = 1;
                do {
                    t = t + 2;
                } while (p % t != 0 && t < p);
                if (t >= p) {
                    System.out.println(p);
                }
                p = p + 2;
            }
        }
    }
}

```

# Rekursiver Ansatz der Problemlösung

## Lösungsschema für Problem $P(X)$

- Basis
  - Direkte Lösung, falls Problemstellung (Eingabe)  $X$  einfacher Natur ist (Trivialfall)
- Schritt
  - Führe eine Lösung für das Problem  $P(X)$  für komplexere Problemstellungen  $X$  durch einen Schritt der Problemreduktion auf die Lösung des gleichen Problems für eine einfachere Problemstellung  $P(X')$  zurück
  - Dabei muss  $X > X'$  gelten für eine wohlfundierte Ordnungsrelation  $>$

# Beispiel: Fakultät in Java (rekursiv)

- Fakultät rekursiv

- $factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$

- Beispiel

- $factorial(4) = 4 * ($   
 $factorial(3) = 3 * ($   
 $factorial(2) = 2 * ($   
 $factorial(1) = 1 * ($   
 $factorial(0) = 1)))))$

```
class Recursive {  
    static int factorial(int n) {  
        if (n == 0) {  
            // Trivialfall  
            return 1;  
        } else {  
            // Problemreduktion  
            return n * factorial(n - 1);  
        }  
    }  
}
```

# Beweis eines Algorithmus

- Algorithmus

$$factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$$

- Spezifikation

- Eingabe

- $n \in \mathbb{N}_0$

- Ausgabe

- $factorial(n) = n!$

# Beweis eines Algorithmus

- Durchführbarkeit  $factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$ 
  - Endliche Beschreibung
    - offensichtlich ja
  - Effektivität
    - Fallunterscheidung, Multiplikation, Subtraktion und rekursiver Aufruf des Verfahrens sind mechanisch ausführbar
  - Determiniertheit
    - „ $n = 0$ “ und „sonst“ schließen sich gegenseitig aus, also ist die Reihenfolge der Ausführung immer eindeutig

# Beweis eines Algorithmus

$$factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$$

---

- Korrektheit

- Partielle Korrektheit

- Ab nächster Folie

- Terminierung

- *factorial*(0) terminiert sofort
    - Für  $n > 0$  wird  $n$  bei jedem rekursiven Aufruf um 1 kleiner, muss also 0 erreichen
    - $n < 0$  ist laut Eingabespezifikation nicht erlaubt!

# Partielle Korrektheit

## Beweis rekursiver Algorithmen

- Rekursive Algorithmen können durch vollständige Induktion bewiesen werden
  - Induktionsanfang
    - Trivialfall
    - Beweis, dass der Algorithmus für Trivialfall richtig ist
  - Induktionsschritt
    - Problemreduktion
    - Beweis, dass der Algorithmus für Problemstellung  $X$  richtig ist, wenn er bereits für einfachere Problemstellung  $X'$  gilt

# Partielle Korrektheit

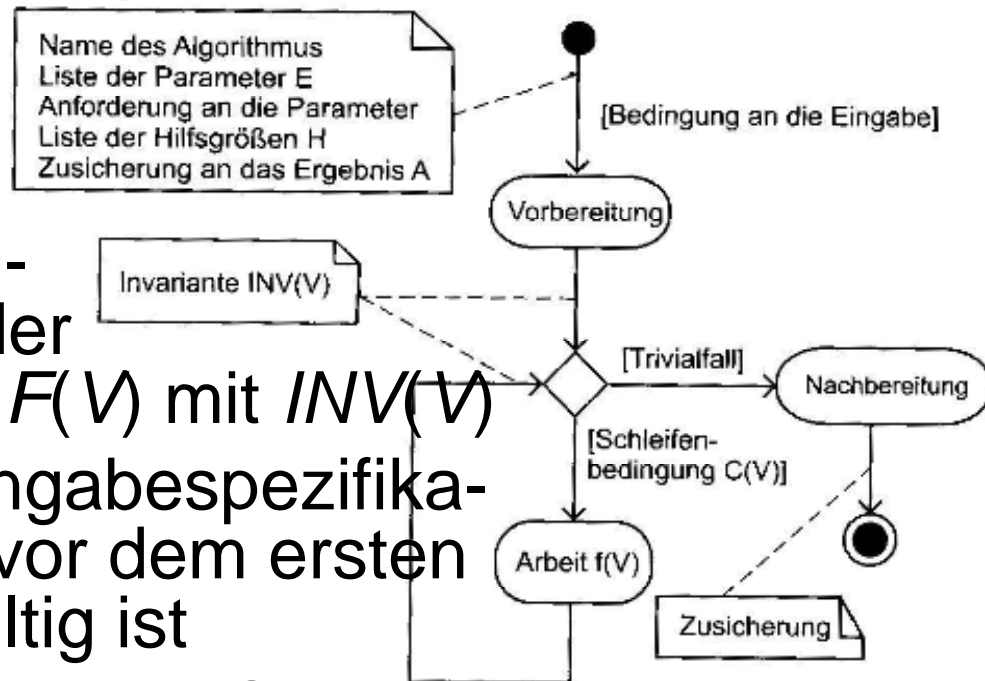
- Behauptung
  - Die Funktion  $factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$  berechnet die Fakultät von  $n$ , d.h.  $factorial(n) = n!$
- Induktionsanfang
  - $0! = 1 = factorial(0)$
- Induktionsschritt
  - $(n+1)! = factorial(n+1)$  || Def. von  $factorial()$  eins.
  - $= (n+1) \cdot factorial(n+1-1)$
  - $= (n+1) \cdot factorial(n)$  || Induktionsannahme eins.
  - $= (n+1) \cdot n!$
  - $= (n+1)!$  q.e.d.



# Beweis iterativer Algorithmen

## Verifikation nach Floyd

- Finde eine geeignete Formel  $F(V)$  und zeige, dass sie eine Schleifeninvariante am Anfang der Schleife ist; bezeichne  $F(V)$  mit  $INV(V)$
- Zeige, dass aus der Eingabespezifikation folgt, dass  $INV(V)$  vor dem ersten Schleifendurchgang gültig ist
- Zeige, dass nach dem letzten Schleifendurchgang aus  $INV(V)$  und der unerfüllten Fortsetzungsbedingung der Schleife die Ausgabespezifikation folgt



# **Beweis iterativer Algorithmen**

## **Finden einer Schleifeninvariante**

- Schleife baut Ergebnis auf
- In jedem Schleifendurchlauf
  - kommt man dem Ergebnis näher
  - wird das noch zu bearbeitende Problem kleiner
- Schleifeninvariante
  - Ausgabe = „was schon getan“ verknüpft mit „was noch zu tun“

# Beweis iterativer Algorithmen

- Schleifeninvariante

- $n! = r \cdot i!$

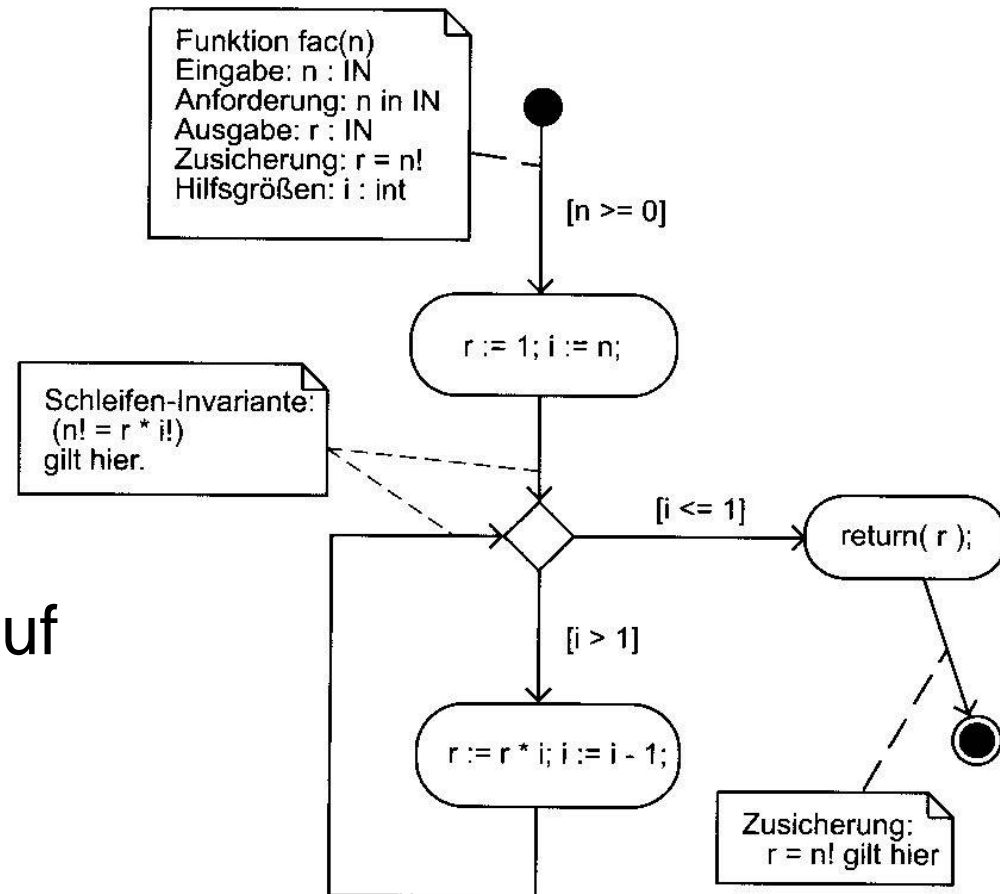
- Invarianz in der Schleife

- Zu Beginn

- $n! = r \cdot i!$
- $r' = r \cdot i, i' = i - 1$

- Nach einem Durchlauf

- $n! = r' \cdot i'!$
- $= (r \cdot i) \cdot (i - 1)!$
- $= r \cdot i \cdot (i - 1)!$
- $= r \cdot i!$



# Beweis iterativer Algorithmen

- Schleifeninvariante

- $n! = r \cdot i!$

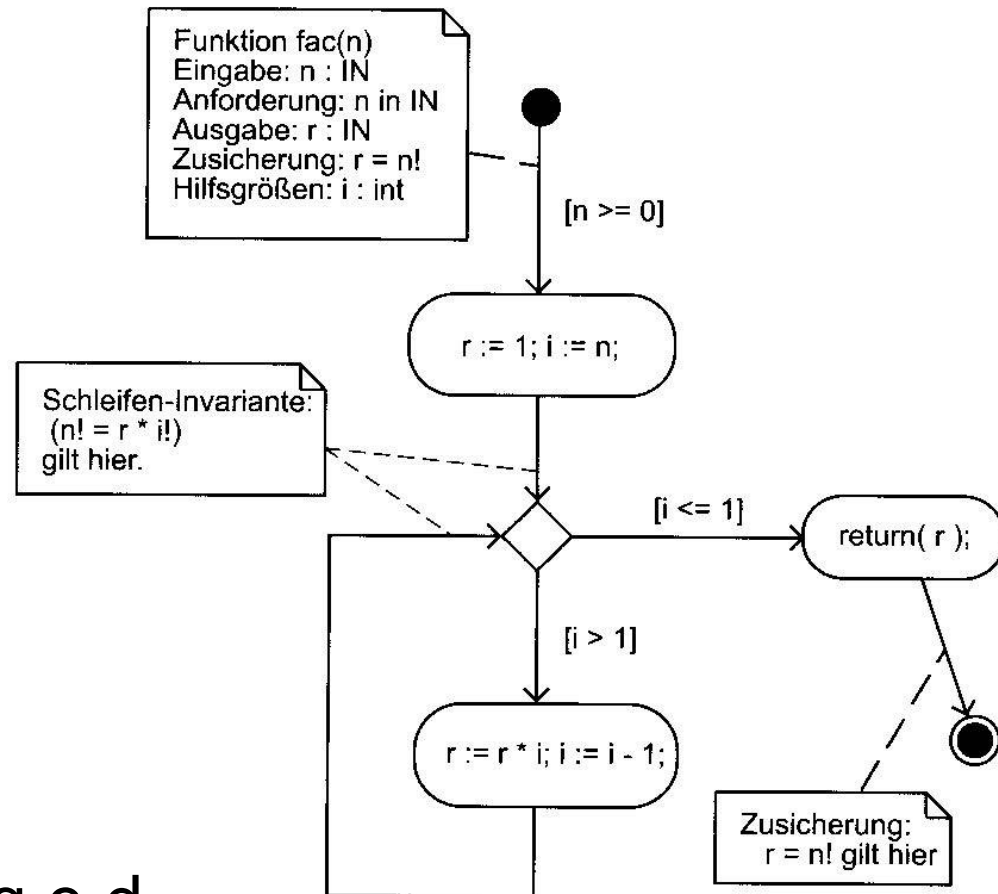
- Vor der Schleife

- $n! = r \cdot i!$
  - $= 1 \cdot n!$
  - $= n!$

- Nach der Schleife

- $n! = r \cdot i!$
  - $= r \cdot 1$
  - $= r$

q.e.d.



# Testen von Programmen

- Was testen?
  - Trivialfälle
  - Normalfälle
  - Fehlerfälle
- Testen ist Pflicht!
  - Ohne dokumentierte Tests gibt's Punktabzug
  - Bewertung
    - Programm 40%
    - Dokumentation 40%
    - Tests 20%