

# Konzepte der Informatik

Algorithmik

Komplexität und Korrektheit

Barbara Pampel

Universität Konstanz, SoSe 2022

---

## Inhalt

1 Komplexität I

2 Das  $\mathcal{O}$ -Kalkül

3 Korrektheit

4 Literatur

## Sortieren – schon wieder

- Algorithmus A

```
void sortA(int[] a, int from, int to) {  
    if (to - from == 1) {  
        if (a[from] > a[to]) { swap(a, from, to); }  
    } else if (to - from > 1) {  
        int mid = (to + from) / 2;  
  
        sortA(a, from, mid);  
        sortA(a, mid + 1, to);  
  
        int[] temp = new int[to - from + 1];  
        int l = from, r = mid + 1;  
        for (int i = 0; i < temp.length; i++) {  
            ...  
        }  
        for (int i = 0; i < temp.length; i++) {  
            a[i + from] = temp[i];  
        }  
    }  
}
```

## Sortieren – schon wieder

```
...
if (l > mid) {
    temp[i] = a[r++];
} else if (r > to) {
    temp[i] = a[l++];
} else if (a[l] <= a[r]) {
    temp[i] = a[l++];
} else {
    temp[i] = a[r++];
}
}
for (int i = 0; i < temp.length; i++) {
    a[from + i] = temp[i];
}
```

## Sortieren – schon wieder

- Algorithmus B

```
void sortB(int[] a, int from, int to) {  
    if (to - from > 0) {  
        int index = to;  
        swap(a, index, to);  
  
        index = to;  
        int p = a[index];  
  
        int l = from, r = to;  
        while (l < r) {  
            while ((l < r) && (a[l] <= p)) { l++; }  
            while ((l < r) && (a[r] >= p)) { r--; }  
            ...  
        }  
    }  
}
```

## Sortieren – schon wieder

```
...
if (l != r) {
    swap(a, l, r);
} else if (a[l] > p) {
    swap(a, l, index);
}
sortB(a, from, l - 1);
sortB(a, l + 1, to);
}
```

## Sortieren – schon wieder

- Algorithmus C

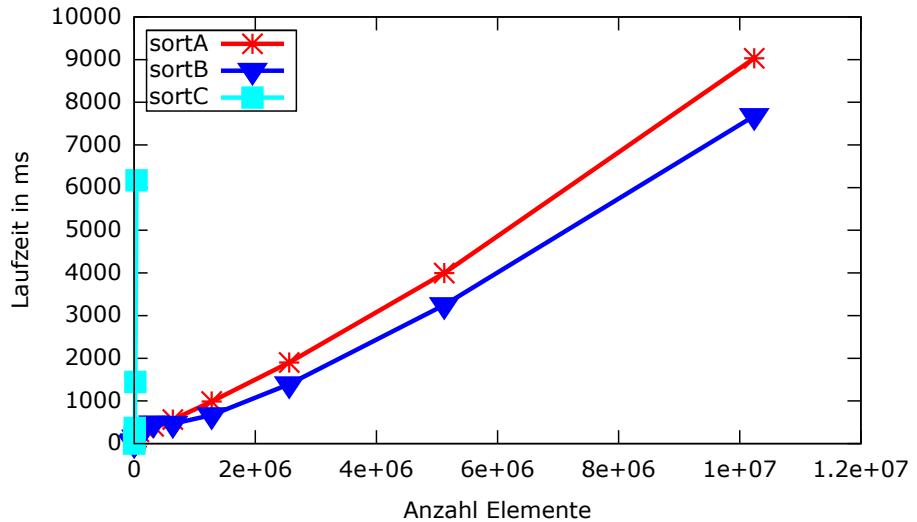
```
void sortC(int[] a) {  
    int n = a.length;  
    do {  
        int newN = 0;  
        for (int i = 0; i < n - 1; i++) {  
            if (a[i] > a[i + 1]) {  
                swap(a, i, i + 1);  
                newN = i + 1;  
            }  
        }  
        n = newN;  
    } while (n > 1);  
}
```

## Laufzeitvergleich

# Elemente	sortA	sortB	sortC
10	0	0	0
1.000	2	2	35
5.000	15	18	140
10.000	48	43	369
20.000	103	57	1447
40.000	147	144	6171
80.000	277	216	-
160.000	370	376	-
320.000	406	475	-
640.000	562	467	-
1.280.000	986	670	-
2.560.000	1889	1394	-
5.120.000	3997	3249	-
10.240.000	9032	7678	-

Laufzeiten in ms

## Laufzeitvergleich



## Beobachtungen

- sortA benötigt für mehr Elemente mehr Zeit
- sortB benötigt für mehr Elemente mehr Zeit, „hüpft“ aber etwas
- sortC ist deutlich langsamer
- sortB scheint das schnellste Verfahren zu sein

## Beobachtungen

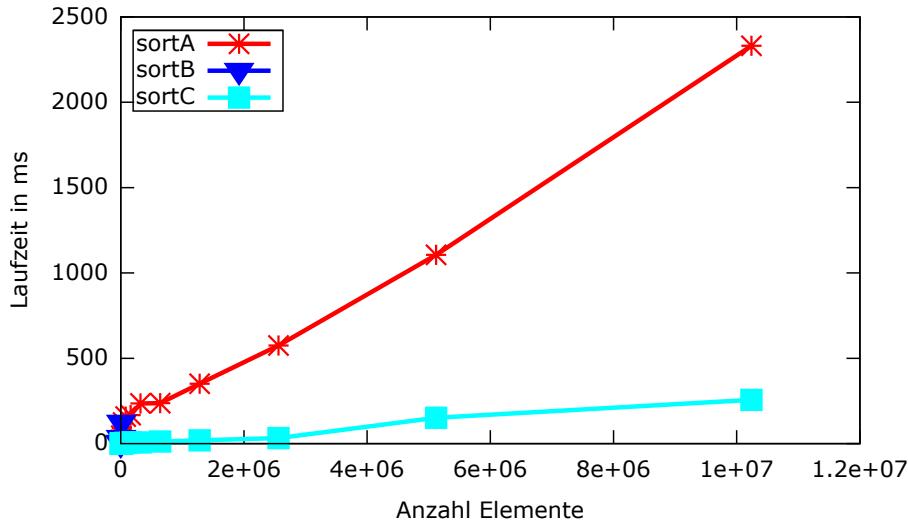
- sortA benötigt für mehr Elemente mehr Zeit
- sortB benötigt für mehr Elemente mehr Zeit, „hüpft“ aber etwas
- sortC ist deutlich langsamer
- sortB scheint das schnellste Verfahren zu sein
  - Wirklich?

## Nochmal Laufzeitvergleich

# Elemente	sortA	sortB	sortC
10	0	0	0
1.000	2	5	0
5.000	21	34	0
10.000	46	123	1
20.000	90	FEHLER	1
40.000	123	-	3
80.000	158	-	4
160.000	167	-	8
320.000	235	-	7
640.000	237	-	12
1.280.000	351	-	19
2.560.000	575	-	32
5.120.000	1106	-	151
10.240.000	2331	-	256

Laufzeiten in ms

## Nochmal Laufzeitvergleich



## Beobachtungen

- Unterschiedliche Klassen des Laufzeitverhaltens
  - sanftes Ansteigen
  - sehr schnelles Ansteigen
- Abhängigkeit von den Daten
  - Laufzeit im schlechtesten Fall (worst case complexity)
  - Laufzeit im Durchschnitt (average case complexity)
  - Laufzeit im besten Fall (best case complexity)

## Laufzeitanalyse

- „Primitive“ Operationen
  - Zuweisungen, Vergleiche, Rechenoperationen, ...
  - 1 Zeiteinheit
- Schleifen
  - $(\text{Anzahl Schleifendurchläufe}) \cdot (\text{Zeiteinheiten im Schleifenrumpf})$
- Methodenaufrufe
  - Zeiteinheiten im Methodenrumpf
- Rekursive Aufrufe
  - Summe der Aufrufe

## Laufzeitanalyse – sortC

```
void sortC(int[] a) {  
    int n = a.length; ①  
    do {  
        int newN = 0; ②  
        for (int i = 0; i < n - 1; i++) { ③  
            if (a[i] > a[i + 1]) { ④  
                swap(a, i, i + 1); ⑤  
                newN = i + 1; ⑥  
            }  
        }  
        n = newN; ⑦  
    } while (n > 1); ⑧  
}
```

- ① Zuweisung = 1 ZE
- ② Zuweisung = 1 ZE
- ③ Zuweisung = 1 ZE  
Vergleich, Subtraktion&  
Addition = 3 ZE

- ④ Addition & Vergleich = 2 ZE
- ⑤ Addition & Zuweisungen = 4 ZE
- ⑥ Addition & Zuweisung = 2 ZE
- ⑦ Zuweisung = 1 ZE
- ⑧ Vergleich = 1 ZE

## Laufzeitanalyse – sortC

```
void sortC(int[] a) {
    int n = a.length; ①
    do {
        int newN = 0; ②
        for (int i = 0; i < n - 1; i++) { ③
            if (a[i] > a[i + 1]) { ④
                swap(a, i, i + 1); ⑤
                newN = i + 1; ⑥
            }
        }
        n = newN; ⑦
    } while (n > 0); ⑧
}
```

## Laufzeitanalyse – sortC

- ① Zuweisung = 1 ZE
- ② Zuweisung = 1 ZE
- ③ Zuweisung = 1 ZE  
Vergleich, Subtraktion&  
Addition = 3 ZE
- ④ Addition & Vergleich = 2 ZE

- ⑤ Addition & Zuweisungen = 4 ZE
- ⑥ Addition & Zuweisung = 2 ZE
- ⑦ Zuweisung = 1 ZE
- ⑧ Vergleich = 1 ZE

⑧ äußere Schleife mit maximal  $n - 1$  Iterationen

③ innere Schleife mit jeweils  $n$  bis 1 Iterationen (Anzahl sinkt!)

$$1 + \left( 1 + 1 + (3 + 2 + 4 + 2) \cdot (n - 1) + 1 + 1 + (3 + 2 + 4 + 2) \cdot (n - 2) + 1 \right. \\ \left. + 1 + (3 + 2 + 4 + 2) \cdot (n - 3) + 1 + \dots + 1 + (3 + 2 + 4 + 2) \cdot (1) \right) + 1 = c + c' \cdot \frac{n^2 - n}{2}$$

## Laufzeitanalyse – sortC

- ① Zuweisung = 1 ZE
- ② Zuweisung = 1 ZE
- ③ Zuweisung = 1 ZE  
Vergleich, Subtraktion&  
Addition = 3 ZE
- ④ Addition & Vergleich = 2 ZE

- ⑤ Addition & Zuweisungen = 4 ZE
- ⑥ Addition & Zuweisung = 2 ZE
- ⑦ Zuweisung = 1 ZE
- ⑧ Vergleich = 1 ZE

⑧ äußere Schleife mit maximal  $n - 1$  Iterationen

③ innere Schleife mit jeweils  $n$  bis 1 Iterationen (Anzahl sinkt!)

$$1 + \left( 1 + 1 + (3 + 2 + 4 + 2) \cdot (n - 1) + 1 + 1 + (3 + 2 + 4 + 2) \cdot (n - 2) + 1 \right.$$

$$\left. + 1 + (3 + 2 + 4 + 2) \cdot (n - 3) + 1 + \dots + 1 + (3 + 2 + 4 + 2) \cdot (1) \right) + 1 = c + c' \cdot \frac{n^2 - n}{2}$$

- Konstanten beeinflussen die Laufzeit bei großem  $n$  nur marginal und werden deshalb ignoriert
- Bestimmender Term ist  $n^2$   
 $\Rightarrow \mathcal{O}\text{-Kalkül}$

## Inhalt

1 Komplexität I

2 Das  $\mathcal{O}$ -Kalkül

3 Korrektheit

4 Literatur

## Das $\mathcal{O}$ -Kalkül

- Erstmals verwendet von Paul Bachmann (1894)
- Oft auch Landau-Symbole genannt

## Das $\mathcal{O}$ -Kalkül

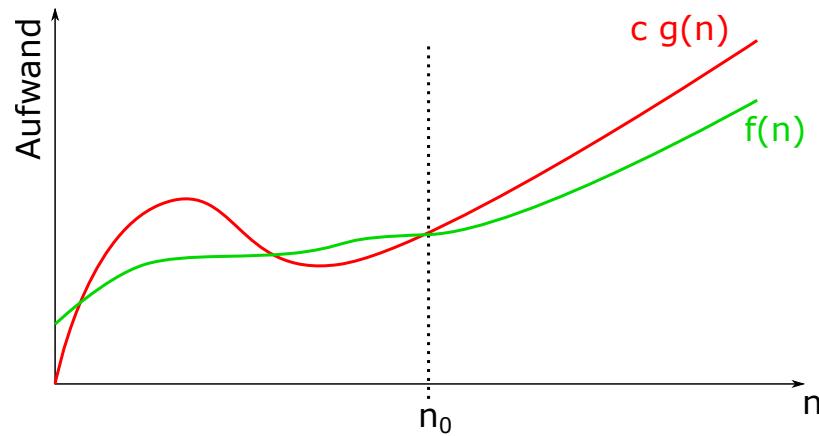
- Erstmals verwendet von Paul Bachmann (1894)
- Oft auch Landau-Symbole genannt
- Beschreibt asymptotisches Verhalten von Funktionen
- Funktionswerte sind ab einem bestimmten Punkt  $n_0$ 
  - immer größer oder gleich, oder
  - immer kleiner oder gleich als eine andere Funktion

## Das $\mathcal{O}$ -Kalkül

- Erstmals verwendet von Paul Bachmann (1894)
- Oft auch Landau-Symbole genannt
- Beschreibt asymptotisches Verhalten von Funktionen
- Funktionswerte sind ab einem bestimmten Punkt  $n_0$ 
  - immer größer oder gleich, oder
  - immer kleiner oder gleich als eine andere Funktion
- Konstanten werden nicht berücksichtigt bzw. durch einen beliebig wählbaren Faktor ausgeglichen
- Das  $\mathcal{O}$ -Kalkül beschreibt Klassen von Funktionen mit bestimmten Eigenschaften

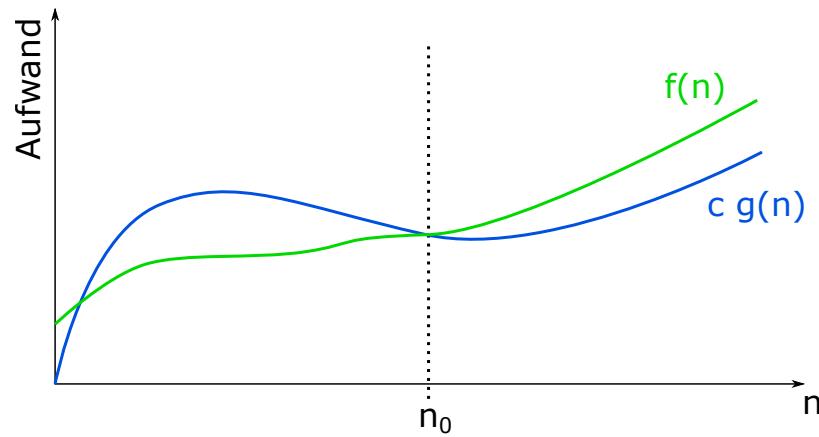
## Groß- $\mathcal{O}$

- $f \in \mathcal{O}(g) \Leftrightarrow \exists c > 0 \ \exists n_0 \ \forall n > n_0 : |f(n)| \leq c \cdot |g(n)|$



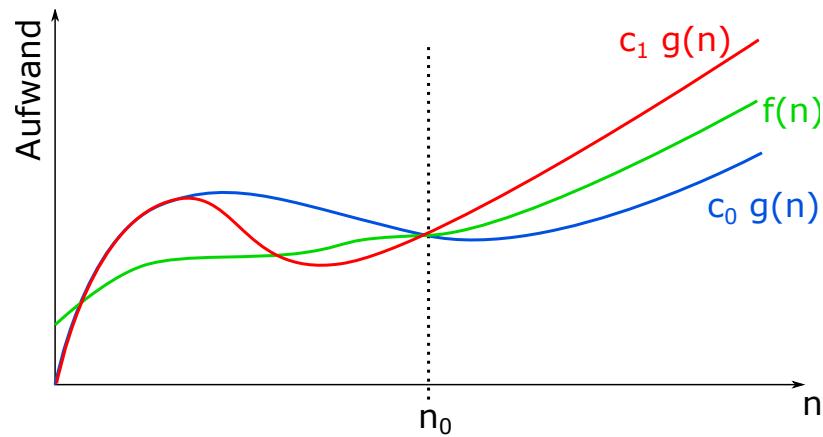
## Groß-Omega

- $f \in \Omega(g) \Leftrightarrow \exists c > 0 \ \exists n_0 \ \forall n > n_0 : |f(n)| \geq c \cdot |g(n)|$

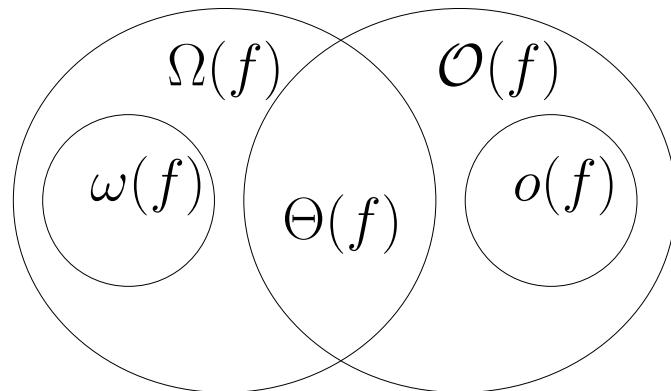


## Groß-Theta

- $f \in \Theta(g) \Leftrightarrow \exists c_0 > 0 \ \exists c_1 > 0 \ \exists n_0 \ \forall n > n_0 :$   
 $c_0 \cdot |g(n)| \leq |f(n)| \leq c_1 \cdot |g(n)|$



## Mengen-Darstellung



## Anwendung

- Notation
  - Oft sieht man  $f = \mathcal{O}(g)$  statt  $f \in \mathcal{O}(g)$   
(streng genommen) mathematisch falsch, aber allgemein toleriert
- Einfache Regeln
  - $f \in \mathcal{O}(f)$
  - $k \cdot f \in \mathcal{O}(f)$ , für konstantes  $k$
  - $\mathcal{O}(\mathcal{O}(f)) \in \mathcal{O}(f)$

## Anwendung

- Notation
  - Oft sieht man  $f = \mathcal{O}(g)$  statt  $f \in \mathcal{O}(g)$   
(streng genommen) mathematisch falsch, aber allgemein toleriert
- Einfache Regeln
  - $f \in \mathcal{O}(f)$
  - $k \cdot f \in \mathcal{O}(f)$ , für konstantes  $k$
  - $\mathcal{O}(\mathcal{O}(f)) \in \mathcal{O}(f)$   
 $\Rightarrow$  für  $f \in \mathcal{O}(g)$  ist  $k \cdot f \in \mathcal{O}(g)$   
für konstantes  $k$
- Additionsregel
  - $f + g \in \mathcal{O}(\max(f, g))$

## Anwendung

- Notation
  - Oft sieht man  $f = \mathcal{O}(g)$  statt  $f \in \mathcal{O}(g)$   
(streng genommen) mathematisch falsch, aber allgemein toleriert
- Einfache Regeln
  - $f \in \mathcal{O}(f)$
  - $k \cdot f \in \mathcal{O}(f)$ , für konstantes  $k$
  - $\mathcal{O}(\mathcal{O}(f)) \in \mathcal{O}(f)$   
 $\Rightarrow$  für  $f \in \mathcal{O}(g)$  ist  $k \cdot f \in \mathcal{O}(g)$   
für konstantes  $k$
- Additionsregel
  - $f + g \in \mathcal{O}(\max(f, g))$   
 $\Rightarrow$  für  $f \in \mathcal{O}(g)$  ist  $f + g \in \mathcal{O}(g)$

## Anwendung

- Notation
  - Oft sieht man  $f = \mathcal{O}(g)$  statt  $f \in \mathcal{O}(g)$   
(streng genommen) mathematisch falsch, aber allgemein toleriert
- Einfache Regeln
  - $f \in \mathcal{O}(f)$
  - $k \cdot f \in \mathcal{O}(f)$ , für konstantes  $k$
  - $\mathcal{O}(\mathcal{O}(f)) \in \mathcal{O}(f)$   
 $\Rightarrow$  für  $f \in \mathcal{O}(g)$  ist  $k \cdot f \in \mathcal{O}(g)$   
für konstantes  $k$
- Additionsregel
  - $f + g \in \mathcal{O}(\max(f, g))$   
 $\Rightarrow$  für  $f \in \mathcal{O}(g)$  ist  $f + g \in \mathcal{O}(g)$ $\Rightarrow$  für  $m$  Grad eines Polynoms  $p$  ist  $p \in \mathcal{O}(n^m)$

## Funktionsklassen

- Mit  $\mathcal{O}$ -,  $\Omega$ - und  $\Theta$ -Notation werden obere, untere und genaue Schranken für das Wachstum von Funktionen beschrieben
- Interesse liegt auf asymptotischen Verhalten für große Eingaben
  - genaue Analyse aufwändig
  - konstante Einflüsse oft uninteressant (aber manchmal für die reale Anwendung doch dominant!)
  - lineare Beschleunigungen einfach zu erreichen (neue Hardware, ...)
- Ziel: Beschreibung des Aufwands durch Funktionsklassen
  - „es kann nicht schlimmer werden als  $f$ “ wenn Aufwand in  $\mathcal{O}(f)$

## Laufzeitanalyse – sortC

- ③ Schleife mit maximal  $n$  Iterationen
- ⑧ Schleife mit maximal  $n$  Iterationen

$$c + c' \cdot \frac{n^2 - n}{2} = \frac{c'}{2}n^2 - \frac{c'}{2}n + c$$

## Laufzeitanalyse – sortC

- ③ Schleife mit maximal  $n$  Iterationen
- ⑧ Schleife mit maximal  $n$  Iterationen

$$c + c' \cdot \frac{n^2 - n}{2} = \frac{c'}{2}n^2 - \frac{c'}{2}n + c \in \mathcal{O}\left(\frac{c'}{2}n^2 - \frac{c'}{2}n + c\right)$$

## Laufzeitanalyse – sortC

- ③ Schleife mit maximal  $n$  Iterationen
- ⑧ Schleife mit maximal  $n$  Iterationen

$$c + c' \cdot \frac{n^2 - n}{2} = \frac{c'}{2}n^2 - \frac{c'}{2}n + c \in \mathcal{O}\left(\frac{c'}{2}n^2 - \frac{c'}{2}n + c\right) \in \mathcal{O}\left(\max\left(\frac{c'}{2}n^2, \frac{c'}{2}n, c\right)\right)$$

## Laufzeitanalyse – sortC

- ③ Schleife mit maximal  $n$  Iterationen
- ⑧ Schleife mit maximal  $n$  Iterationen

$$c + c' \cdot \frac{n^2 - n}{2} = \frac{c'}{2}n^2 - \frac{c'}{2}n + c \in \mathcal{O}\left(\frac{c'}{2}n^2 - \frac{c'}{2}n + c\right) \in \mathcal{O}\left(\max\left(\frac{c'}{2}n^2, \frac{c'}{2}n, c\right)\right) \in \mathcal{O}\left(\frac{c'}{2}n^2\right)$$

## Laufzeitanalyse – sortC

- ③ Schleife mit maximal  $n$  Iterationen
- ⑧ Schleife mit maximal  $n$  Iterationen

$$c + c' \cdot \frac{n^2 - n}{2} = \frac{c'}{2}n^2 - \frac{c'}{2}n + c \in \mathcal{O}\left(\frac{c'}{2}n^2 - \frac{c'}{2}n + c\right) \in \mathcal{O}\left(\max\left(\frac{c'}{2}n^2, \frac{c'}{2}n, c\right)\right) \in \mathcal{O}\left(\frac{c'}{2}n^2\right) \in \mathcal{O}(n^2)$$

## Laufzeitanalyse – sortA

```
void sortA(int[] a, int from, int to) {  
    ...  
    int mid = (to + from) / 2;  
  
    sortA(a, from, mid); ①  
    sortA(a, mid + 1, to); ②  
  
    int[] temp = new int[to - from + 1];  
    int l = from, r = mid + 1;  
    for (int i = 0; i < temp.length; i++) { ③  
        ...  
        }  
        for (int i = 0; i < temp.length; i++) { ④  
            a[from + i] = temp[i];  
        }  
    }  
}
```

- ① rekursiver Aufruf mit halber Problemgröße
- ② rekursiver Aufruf mit halber Problemgröße
- ③  $n$  Schleifendurchläufe
- ④  $n$  Schleifendurchläufe

## Laufzeitanalyse – sortA

- Problemgröße halbiert sich bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene gibt es für jedes Element einen Vergleich und eine Kopieraktion
  - $2n$  Operationen

$$2n + 2(2\frac{n}{2} + 2(2\frac{n}{4} + 2(2\frac{n}{8} + \dots))) \approx$$

$$2n \cdot \log_2(n)$$

## Laufzeitanalyse – sortA

- Problemgröße halbiert sich bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene gibt es für jedes Element einen Vergleich und eine Kopieraktion
  - $2n$  Operationen

$$2n + 2(2\frac{n}{2} + 2(2\frac{n}{4} + 2(2\frac{n}{8} + \dots))) \approx$$

$$2n \cdot \log_2(n) \in \mathcal{O}(2n \cdot \log_2(n))$$

## Laufzeitanalyse – sortA

- Problemgröße halbiert sich bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene gibt es für jedes Element einen Vergleich und eine Kopieraktion
  - $2n$  Operationen

$$2n + 2(2\frac{n}{2} + 2(2\frac{n}{4} + 2(2\frac{n}{8} + \dots))) \approx$$

$$2n \cdot \log_2(n) \in \mathcal{O}(2n \cdot \log_2(n)) \in \mathcal{O}(n \cdot \log_2(n))$$

## Laufzeitanalyse – sortB

```
void sortB(int[] a, int from, int to) {  
    if (to - from > 0) {  
        ...  
        int l = from, r = to;  
        while (l < r) {  
            while ((l < r) && (a[l] <= p)) { l++; } ①  
            while ((l < r) && (a[r] >= p)) { r--; } ②  
            ...  
        }  
        sortB(a, from, l - 1); ③  
        sortB(a, l + 1, to); ④  
    }  
}
```

- ① zusammen  $n$  Operationen
- ② rekursiver Aufruf mit – im Idealfall – halber Problemgröße
- ③ rekursiver Aufruf mit – im Idealfall – halber Problemgröße

## Laufzeitanalyse – sortB

- Problemgröße halbiert sich – im Idealfall – bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene wird jedes Element einmal verglichen
  - $n$  Operationen

$$n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + 2\left(\frac{n}{8} + \dots\right)\right)\right) \approx n \cdot \log_2(n)$$

## Laufzeitanalyse – sortB

- Problemgröße halbiert sich – im Idealfall – bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene wird jedes Element einmal verglichen
  - $n$  Operationen

$$n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + 2\left(\frac{n}{8} + \dots\right)\right)\right) \approx n \cdot \log_2(n) \in \mathcal{O}(n \cdot \log_2(n))$$

## Laufzeitanalyse – sortB

- Problemgröße halbiert sich – im Idealfall – bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene wird jedes Element einmal verglichen
  - $n$  Operationen

$$n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + 2\left(\frac{n}{8} + \dots\right)\right)\right) \approx n \cdot \log_2(n) \in \mathcal{O}(n \cdot \log_2(n))$$

- Im schlimmsten Fall trennt das Pivotelement allerdings nur eine Zahl ab

$$n + (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^n i = \frac{n \cdot (n+1)}{2}$$

## Laufzeitanalyse – sortB

- Problemgröße halbiert sich – im Idealfall – bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene wird jedes Element einmal verglichen
  - $n$  Operationen

$$n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + 2\left(\frac{n}{8} + \dots\right)\right)\right) \approx n \cdot \log_2(n) \in \mathcal{O}(n \cdot \log_2(n))$$

- Im schlimmsten Fall trennt das Pivotelement allerdings nur eine Zahl ab

$$n + (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^n i = \frac{n \cdot (n+1)}{2} \in \mathcal{O}\left(\frac{1}{2}(n^2 + n)\right)$$

## Laufzeitanalyse – sortB

- Problemgröße halbiert sich – im Idealfall – bei jedem rekursiven Aufruf
  - $\lceil \log_2(n) \rceil$  rekursive Aufrufe bis zu Problemen der Größe 1
  - erzeugt Baum der Tiefe  $\lceil \log_2(n) \rceil$
- In jeder Baumebene wird jedes Element einmal verglichen
  - $n$  Operationen

$$n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + 2\left(\frac{n}{8} + \dots\right)\right)\right) \approx n \cdot \log_2(n) \in \mathcal{O}(n \cdot \log_2(n))$$

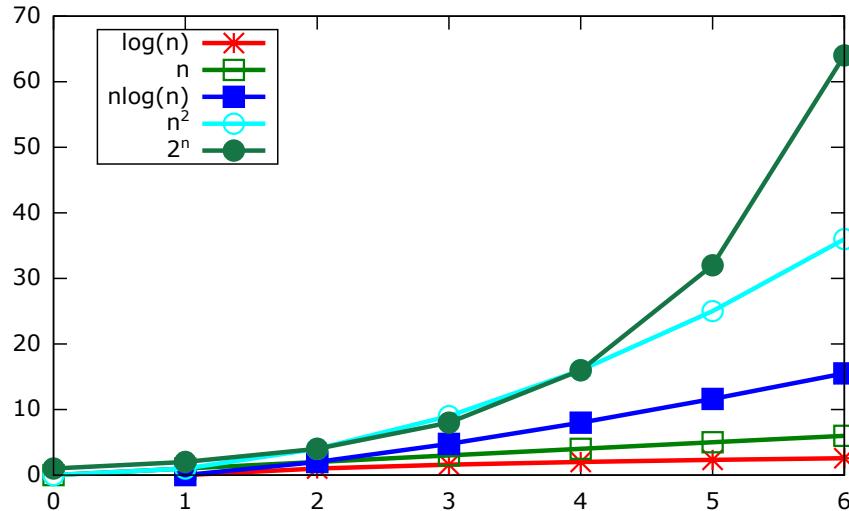
- Im schlimmsten Fall trennt das Pivotelement allerdings nur eine Zahl ab

$$n + (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^n i = \frac{n \cdot (n+1)}{2} \in \mathcal{O}\left(\frac{1}{2}(n^2 + n)\right) \in \mathcal{O}(n^2)$$

## Sortierverfahren im Vergleich

- sortA: Mergesort
  - garantierter Aufwand  $\mathcal{O}(n \cdot \log_2(n))$
  - deswegen sogar  $\Theta(n \cdot \log_2(n))$
  - „doppelter“ Aufwand wegen Kopieren der Elemente am Ende jedes rekursiven Aufrufs
- sortB: Quicksort
  - meistens Aufwand  $\mathcal{O}(n \cdot \log_2(n))$
  - bei schlechter Wahl des Pivotelements  $\mathcal{O}(n^2)$
  - im Normalfall trotzdem schneller als Mergesort
- sortC : Bubblesort
  - im Idealfall  $\mathcal{O}(n)$
  - im Realfall  $\mathcal{O}(n^2)$

# Wachstum von Funktionen



## Wachstum von Funktionen

$n$	1	10	100	1.000	10.000
$\log_{10} n$	0	1	2	3	4
$\log_2 n$	0	$\approx 3$	$\approx 7$	$\approx 10$	$\approx 13$
$\sqrt{n}$	1	$\approx 3$	10	$\approx 32$	100
$n^2$	1	100	10.000	1.000.000	100.000.000
$n^3$	1	1.000	1.000.000	1.000.000.000	1 Billion
$2^n$	2	1.024	$\approx 10^{30}$	$\approx 10^{301}$	$> 10^{3000}$
$1, 1^n$	$\approx 1$	$\approx 3$	$\approx 13.781$	$\approx 2 \cdot 10^{41}$	$> 10^{414}$
$n!$	1	3.628.800	$\approx 9 \cdot 10^{157}$	...	...
$n^n$	1	10.000.000.000	$10^{200}$	$10^{3000}$	$10^{40000}$

$10^{50}$  Operationen? Keine Chance, selbst mit dem Computer...

## Wachstum von Funktionen

- Annahme:  $1\mu s$  pro Operation
- Tabelle mit
  - Anzahl der Operationen in Abhängigkeit von  $n$
  - daraus folgend die maximale Problemgröße, die in einer bestimmten Zeit gelöst werden kann

Programm	Operationen	max. Problemgröße		
		1 s	1 min	1 h
P1	$400n$	2.500	150.000	9.000.000
P2	$20n \lceil \log_2 n \rceil$	4.096	166.666	7.826.087
P3	$2n^2$	707	5.477	42.426
P4	$n^4$	31	88	244
P5	$2^n$	19	25	31

- Unterschiedliches asymptotische Wachstum dominiert irgendwann die Konstanten

## Wachstum von Funktionen

- Annahme: 1000-fach schnellerer Rechner
- Tabelle mit den jetzt zu lösenden Problemgrößen, vorher  $m$

Programm	Operationen	neue Maximalgröße
P1	$400n$	$1.000m$
P2	$20n \lceil \log_2 n \rceil$	$\approx \frac{\log_2 n}{9+\log_2 m} 1.000m$
P3	$2n^2$	$\approx 32m$
P4	$n^4$	$\approx 3m$
P5	$2^n$	$m + 9$

## Größenordnungen

Name	Funktionsklasse
konstant	$\mathcal{O}(1)$
logarithmisch	$\mathcal{O}(\log n)$
quadratisch logarithmisch	$\mathcal{O}(\log^2 n)$
linear	$\mathcal{O}(n)$
$n \cdot \log n$ wachsend	$\mathcal{O}(n \log n)$
quadratisch	$\mathcal{O}(n^2)$
kubisch	$\mathcal{O}(n^3)$
polynomiell	$\mathcal{O}(n^k)$
exponentiell	$\mathcal{O}(k^n)$

- $f$  ist polynomiell beschränkt, wenn es ein Polynom  $p$  gibt mit  $f \in \mathcal{O}(p)$
- $f$  wächst exponentiell, wenn es ein  $\epsilon > 0$  gibt mit  $f \in \Theta(2^{n^\epsilon})$

## Komplexitäten

- Algorithmenkomplexität
  - Aufwand, den der konkrete Algorithmus braucht
  - abhängig von der Implementierung
- Speicherkomplexität
  - benötigter Speicherplatz während der Berechnung
  - Quicksort: Stack für rekursive Aufrufe,  $\mathcal{O}(\log_2(n))$  bzw.  $\mathcal{O}(n)$
  - Mergesort: Stack und temporäre zweite Liste,  $\mathcal{O}(\log_2(n) + n) = \mathcal{O}(n)$
  - Bubblesort: kein zusätzlicher Speicherplatz,  $\mathcal{O}(1)$
- Problemkomplexität
  - Aufwand, den das Problem mindestens zur Lösung benötigt
  - in typischen Fällen
  - unabhängig von konkreten Algorithmen

## Zusammenfassung Laufzeitanalyse

- Ziel:** Einen Aspekt der Qualität eines Algorithms, seine theoretische maschinen-unabhängige Laufzeit, zu klassifizieren.
- ⇒ Vergleich mit anderen Funktionen, typischerweise aus der Folie Größenordnungen, denn sind auf die für große Eingaben relevanten Teile eines Termes reduziert.
- Vergleich wird durch  $\mathcal{O}$ -Notation formal gefasst.

## Laufzeitvergleich Beispiel

$2n^2 + 10n \in \mathcal{O}(n^3)$  z.B. für  $c = 12$  und  $n_0 = 1$  gilt für alle  $n > n_0$ :

$$12n^3 \geq 2n^2 + 10n, \text{ denn ab } n = 1 \text{ ist } n^3 \geq n^2 \geq n.$$

**es gilt sogar:**

$2n^2 + 10n \in \mathcal{O}(n^2)$  z.B. für  $c = 12$  und  $n_0 = 1$  gilt für alle  $n > n_0$ :

$$12n^2 \geq 2n^2 + 10n, \text{ denn ab } n = 1 \text{ ist } n^2 \geq n.$$

**andererseits:**

$2n^2 + 10n \in \Omega(n^2)$  z.B. für  $c = 1$  und  $n_0 = 0$  gilt für alle  $n > n_0$ :  $n^2 \leq 2n^2 + 10n$

**somit gilt sogar:**

$2n^2 + 10n \in \Theta(n^2)$  z.B. für  $c_1 = 12$ ,  $c_0 = 1$  und  $n_0 = 1$  gilt für alle  $n > n_0$ :

$$n^2 \leq 2n^2 + 10n \leq 12n^2$$

## Korrektheit

- Partielle Korrektheit
  - Korrektheit der Einzelschritte
  - Programm  $P$  gibt für alle Eingaben, die  $\mathcal{V}$  erfüllen, die „richtige“ Ausgabe aus, es wird also  $\mathcal{N}$  erfüllt  
Schreibweise  $\mathcal{V}\{P\}\mathcal{N}$
- Totale Korrektheit
  - Programm ist partiell korrekt und terminiert
  - totale Korrektheit ist im Allgemeinen nicht beweisbar, da für Terminierung das Halteproblem gelöst werden müsste
  - für viele spezielle Probleme dennoch beweisbar

## Verifikationsmethode von Floyd

- Methode von Robert Floyd (1967) zur Verifikation von iterativen Programmen
  1. Finde eine geeignete Formel  $\mathcal{INV}$  und zeige, dass sie eine Schleifeninvariante ist
    - $\mathcal{INV}$  muss zunächst innerhalb der Schleife gelten
  2. Zeige, dass aus der Eingabespezifikation folgt, dass  $\mathcal{INV}$  auch vor dem ersten Schleifendurchgang gültig ist
  3. Zeige, dass nach dem letzten Schleifendurchgang aus  $\mathcal{INV}$  und aus der Negation der Schleifenbedingung  $b$ , die Gültigkeit der Ausgabespezifikation folgt
  4. Zeige, dass die Schleife terminiert

## Vollständige Induktion

z.B.: Beweis einer Schleifeninvariante

## Vollständige Induktion

z.B.: Beweis einer Schleifeninvariante

Induktionsvoraussetzung  
IV:



Induktionsanfang:  $i_0$ :



Induktionsschritt:

$$i - 1 \rightarrow i:$$



## Beispiel - Einsortieren

---

### Algorithm 1: Einsortieren

---

```
e ← L.head()
L'.head() ← e
while L.next(e) ≠ null do
    e ← L.next(e)
    uk ← L'.head()
    while uk ≠ nil and e ≥ uk do
        uk ← L'.next(uk)
    L'.insert(e, uk)
return L'
```

---

### Schleifeninvariante:

## Beispiel - Einsortieren

---

### Algorithm 2: Einsortieren

---

```
e ← L.head()
L'.head() ← e
while L.next(e) ≠ null do
    e ← L.next(e)
    uk ← L'.head()
    while uk ≠ nil and e ≥ uk do
        uk ← L'.next(uk)
    L'.insert(e, uk)
return L'
```

---

### Schleifeninvariante:

Nach jedem Durchlauf (DL)  $i$  der äußeren Schleife gilt:

$L'$  ist nicht-absteigend sortiert.

Induktionsbeweis:  $\mathcal{I} \wedge \mathcal{V}$  gilt in der Schleife

Schleifeninvariante (Induktionsvoraussetzung IV):

Nach jedem Schleifendurchlauf (DL)  $i$  gilt:  $L'$  ist nicht-absteigend sortiert.

Induktionsbeweis:  $\mathcal{I} \wedge \mathcal{V}$  gilt in der Schleife

Schleifeninvariante (Induktionsvoraussetzung IV):

Nach jedem Schleifendurchlauf (DL)  $i$  gilt:  $L'$  ist nicht-absteigend sortiert.

Induktionsanfang:  $i = 1$ :

Bei Begin des ersten Schleifendurchlaufs enthält  $L'$  nur ein Element.  $e$  wird davor einsortiert, falls es echt kleiner ist, danach, falls es grösser oder gleich ist. Damit bleibt  $L'$  nicht-absteigend sortiert.

Induktionsbeweis:  $\mathcal{I} \wedge \mathcal{V}$  gilt in der Schleife

#### Schleifeninvariante (Induktionsvoraussetzung IV):

Nach jedem Schleifendurchlauf (DL)  $i$  gilt:  $L'$  ist nicht-absteigend sortiert.

#### Induktionsanfang: $i = 1$ :

Bei Begin des ersten Schleifendurchlaufs enthält  $L'$  nur ein Element.  $e$  wird davor einsortiert, falls es echt kleiner ist, danach, falls es grösser oder gleich ist. Damit bleibt  $L'$  nicht-absteigend sortiert.

#### Induktionsschritt: $i - 1 \rightarrow i$ :

$e$  wird vor dem ersten echt grösseren Element der nicht-absteigend sortierten (IV) Liste  $L'$  eingefügt. Damit bleibt  $L'$  auch nach dem Einfügen und damit nach dem Schleifendurchlauf nicht-absteigend sortiert.

## Kompletter Beweis nach Floyd

1. nach dem Beweis oben, ist  $\mathcal{INV}$  eine Schleifeninvariante
2. zu Beginn des Algorithmus wird  $L'$  mit einem Element angelegt, damit gilt  $\mathcal{INV}$  auch vor dem ersten Schleifendurchlauf.
3. die Schleife bricht ab, wenn alle Elemente aus der Eingabeliste  $L$  durchlaufen wurden. Alle wurden innerhalb der Schleife in  $L'$  eingesortiert.  $L'$  ist nicht-absteigend sortiert und entspricht damit der Ausgabespezifikation.
4.  $L$  wird in der Schleife Schritt für Schritt durchlaufen und da  $L$  endlich lang ist, terminiert die Schleife.

## Beispiel - Selection Sort

---

### Algorithm 3: SelectionSort

---

```
for i = 1, . . . , n − 1 do
    m ← i
    for j = i + 1, . . . , n do
        if A[j] < A[m] then m ← j
    vertausche A[i] und A[m]
```

---

Schleifeninvariante  $\mathcal{INV}$ :

## Beispiel - Selection Sort

---

### Algorithm 4: SelectionSort

---

```
for i = 1, . . . , n − 1 do
    m ← i
    for j = i + 1, . . . , n do
        if A[j] < A[m] then m ← j
    vertausche A[i] und A[m]
```

---

### Schleifeninvariante $\mathcal{INV}$ :

Nach jedem Schleifendurchlauf (DL)  $i$  gilt:

- $A[1] \dots A[i]$  ist nicht-absteigend sortiert.
- Für jedes Element  $x \in \{A[1], \dots, A[i]\}$  und für jedes Element  $y \in \{A[i+1], \dots, A[n]\}$  gilt  $x \leq y$ .

Schleifeninvariante  $\mathcal{INV}$  (Induktionsvoraussetzung IV):

Nach Schleifendurchlauf (DL)  $i$  gilt:

- a.  $A[1] \dots A[i]$  ist nicht-absteigend sortiert.
- b. Für jedes Element  $x \in \{A[1], \dots, A[i]\}$  und für jedes Element  $y \in \{A[i+1], \dots, A[n]\}$  gilt  $x \leq y$ .

Schleifeninvariante  $\mathcal{INV}$  (Induktionsvoraussetzung IV):

Nach Schleifendurchlauf (DL)  $i$  gilt:

- $A[1] \dots A[i]$  ist nicht-absteigend sortiert.
- Für jedes Element  $x \in \{A[1], \dots, A[i]\}$  und für jedes Element  $y \in \{A[i+1], \dots, A[n]\}$  gilt  $x \leq y$ .

Induktionsanfang:  $i = 1$ :

- Nach DL 1 ist  $A[1] \dots A[i] = A[1]$  korrekt sortiert, da nur aus einem Element bestehend.
- Da  $A[1] = \min\{A[1], \dots, A[n]\}$  gilt  $A[1] \leq A[k], k = 2, \dots, n$ .

## Schleifeninvariante $\mathcal{INV}$ (Induktionsvoraussetzung IV):

Nach Schleifendurchlauf (DL)  $i$  gilt:

- $A[1] \dots A[i]$  ist nicht-absteigend sortiert.
- Für jedes Element  $x \in \{A[1], \dots, A[i]\}$  und für jedes Element  $y \in \{A[i+1], \dots, A[n]\}$  gilt  $x \leq y$ .

Induktionsanfang:  $i = 1$ :

- Nach DL 1 ist  $A[1] \dots A[i] = A[1]$  korrekt sortiert, da nur aus einem Element bestehend.
- Da  $A[1] = \min\{A[1], \dots, A[n]\}$  gilt  $A[1] \leq A[k], k = 2, \dots, n$ .

Induktionsschritt:  $i - 1 \rightarrow i$ :

$A[i]$  wird in DL  $i$  der Wert  $z = \min\{A[i], \dots, A[n]\}$  zugewiesen.

- Wegen  $z \stackrel{\text{IVb}}{\geq} x \in \{A[1], \dots, A[i-1]\}$  ist nach DL  $i$   $A[1] \dots A[i]$  nicht-absteigend sortiert.  
 $\rightarrow$  IVa gilt.
- Nach IVb gilt  $x' \leq y$  für alle  $x' \in \{A[1], \dots, A[i-1]\}$  und für alle  $y \in \{A[i+1], \dots, A[n]\}$ . Da auch  $z = \min\{A[i], \dots, A[n]\} \leq y$  gilt IVb nach DL  $i$ .

## Kompletter Beweis nach Floyd

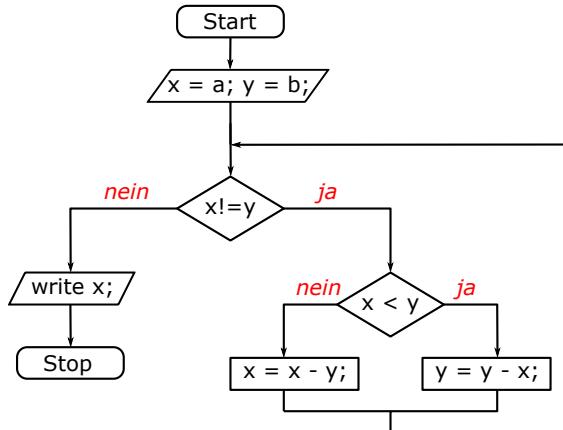
1. nach dem Beweis oben, ist  $\mathcal{INV}$  eine Schleifeninvariante
2. zu Beginn des Algorithmus  $A$  unsortiert, der sortierte Bereich ist aber leer und damit gilt  $\mathcal{INV}$
3. nach Durchlauf  $n - 1$  ist  $A[1], \dots, A[n - 1]$  nach  $\mathcal{INV}$  nicht-absteigend sortiert und  $A[n] \geq x$  für alle  $x \in \{A[1], \dots, A[n - 1]\}$ . Damit ist ganz  $A$  nicht-absteigend sortiert.
4. die Schleife terminiert nach  $n - 1$  Durchläufen.

## Korrektheit über Zusicherungen

- Größter gemeinsamer Teiler
  - $ggT(x, 0) = |x|$
  - $ggT(x, x) = |x|$
  - $ggT(x, y) = ggT(x, y - x)$
  - $ggT(x, y) = ggT(x - y, y)$
  - ...

## Korrektheit über Zusicherungen

- Kontrollflussgraph für ggT



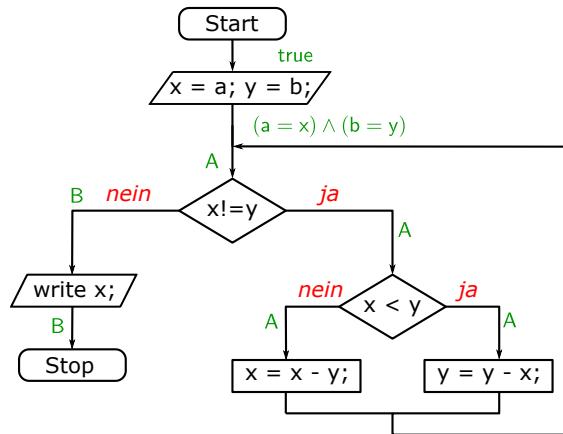
- Eine Zusicherung an jeder Kante benötigt

## Korrektheit über Zusicherungen

- Am Anfang gibt es keine Zusicherungen
  - außer, dass die Eingabe ganze Zahlen sein müssen
- Nach  $x = a$ ; gilt  $x = a$ , nach  $y = b$ ; gilt  $y = b$ 
  - trivial...
- Vor dem Betreten der Schleife soll gelten  $A \equiv (ggT(a, b) = ggT(x, y))$
- Während der Schleife soll ebenfalls A gelten
- Nach der Schleife/am Programmende soll gelten  $B \equiv A \wedge (x = y)$

## Korrektheit über Zusicherungen

- Kontrollflussgraph für ggT mit Zusicherungen



## Beweisvorgang

- Wie überprüft man, ob die Zusicherungen lokal zusammen passen?
- Teilproblem 1: Zuweisungen
  - Beispiel: Zuweisung  $x = y + z$ ;
  - damit nach der Zuweisung  $x > 0$  gilt, muss
  - vor der Zuweisung  $y + z > 0$  gelten
  - $\mathcal{V} \equiv (y + z > 0)$ ,  $\mathcal{N} \equiv (x > 0)$
- Allgemeines Prinzip
  - jede Anweisung transformiert eine Nachbedingung  $\mathcal{N}$  in eine minimale Anforderung, die vor der Ausführung erfüllt sein muss, damit  $\mathcal{N}$  nach der Ausführung gilt

## Weakest Precondition

- Schwächste Vorbedingung, die bei einer Anweisung erfüllt sein muss
- Einfache Zuweisung  $x = e$ ;
  - $\mathbf{WP}[\![x = e;]\!](\mathcal{N}) \equiv \mathcal{N}[x \rightarrow e]$
  - in der Nachbedingung  $\mathcal{N}$  werden alle Vorkommen von  $x$  durch  $e$  ersetzt
- Eine beliebige Vorbedingung  $\mathcal{V}$  für eine Anweisung  $S$  ist gültig, sofern  $\mathcal{V} \Rightarrow \mathbf{WP}[\![S]\!](\mathcal{N})$ 
  - $\mathcal{V}$  impliziert die schwächste Vorbedingung für  $\mathcal{N}$
- Beispiel:  $x = x - y$ ;
  - Nachbedingung  $\mathcal{N}$ :  $x > 0$
  - schwächste Vorbedingung:  $x - y > 0$ 
    - Ersetzen von  $x$  durch  $e := (x - y)$  in  $\mathcal{N}$
  - stärkere Vorbedingung:  $x - y > 2$
  - noch stärkere Vorbedingung:  $x - y = 3$

## Beispiel ggT V

- Zuweisung  $x = x - y;$ 
  - Nachbedingung  $\mathcal{N} = A \equiv (ggT(a, b) = ggT(x, y))$
  - schwächste Vorbedingung

$$\begin{aligned}A[x \rightarrow x - y] &\equiv (ggT(a, b) = ggT(x - y, y)) \\&\equiv (ggT(a, b) = ggT(x, y)) \\&\equiv A\end{aligned}$$

## Beispiel ggT V

- Zuweisung  $x = x - y$ ;
  - Nachbedingung  $\mathcal{N} = A \equiv (ggT(a, b) = ggT(x, y))$
  - schwächste Vorbedingung

$$\begin{aligned}A[x \rightarrow x - y] &\equiv (ggT(a, b) = ggT(x - y, y)) \\&\equiv (ggT(a, b) = ggT(x, y)) \\&\equiv A\end{aligned}$$

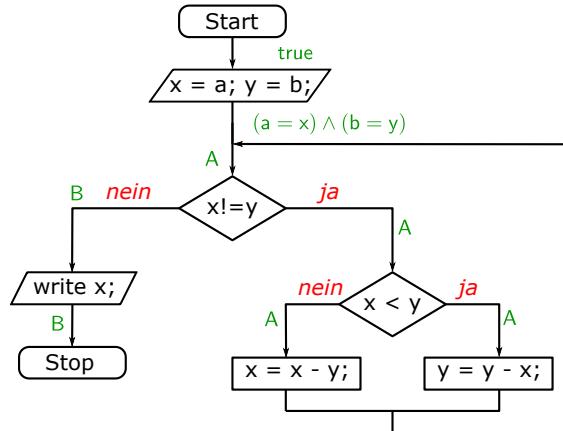
- Zuweisung  $y = y - x$ ;
  - Nachbedingung  $\mathcal{N} = A \equiv (ggT(a, b) = ggT(x, y))$
  - schwächste Vorbedingung

$$\begin{aligned}A[y \rightarrow y - x] &\equiv (ggT(a, b) = ggT(x, y - x)) \\&\equiv (ggT(a, b) = ggT(x, y)) \\&\equiv A\end{aligned}$$

## Zusammenstellung Zuweisungen

- $\text{WP}[\![;]\!](\mathcal{N}) \equiv \mathcal{N}$ 
  - bei einer „Null-Anweisung“ sind Nach- und schwächste Vorbedingung identisch, da keine Variablen geändert werden
- $\text{WP}[\![x = e;]\!](\mathcal{N}) \equiv \mathcal{N}[x \rightarrow e]$ 
  - bei einfachen Zuweisungen wird in der schwächsten Vorbedingung die linke Seite durch die rechte Seite ersetzt
- $\text{WP}[\![\text{write } x;]\!](\mathcal{N}) \equiv \mathcal{N}$ 
  - bei Ausgabeanweisungen sind Nach- und schwächste Vorbedingung identisch, da keine Variablen geändert werden

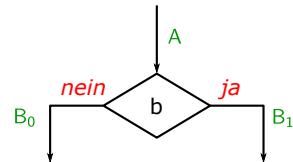
## Orientierung



- Für die Anweisungen  $x = a;$  und  $y = b;$  ergibt sich  
 $\text{WP}[\![x = a; y = b;]\!]A \equiv (ggT(a, b) = ggT(a, b)) \equiv \text{true}$

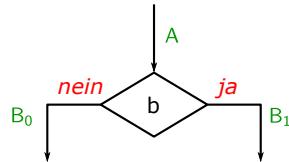
## Beweisvorgang

- Wie überprüft man, ob die Zusicherungen lokal zusammen passen?
- Teilproblem 2: Verzweigungen



- Es soll gelten
  - $A \wedge \neg b \Rightarrow B_0$  und
  - $A \wedge b \Rightarrow B_1$

## Verzweigungen



- Schwächste Vorbedingung für Verzweigungen

- $\mathbf{WP}[\![b]\!](B_0, B_1) \equiv ((\neg b) \Rightarrow B_0) \wedge (b \Rightarrow B_1)$
- umschreibbar in

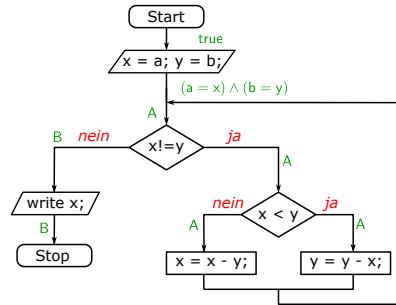
$$\begin{aligned}
 \mathbf{WP}[\![b]\!](B_0, B_1) &\equiv (b \vee B_0) \wedge (\neg b \vee B_1) \\
 &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \vee (B_0 \wedge B_1) \\
 &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1)
 \end{aligned}$$

## Beispiel für Verzweigungen

- **WP**  $\llbracket b \rrbracket (B_0, B_1) \equiv (\neg b \wedge B_0) \vee (b \wedge B_1)$
- $B_0 \equiv (x > y) \wedge (y > 0)$
- $B_1 \equiv (y > x) \wedge (x > 0)$
- Sei  $b$  die Bedingung  $y > x$ , dann ist die schwächste Vorbedingung

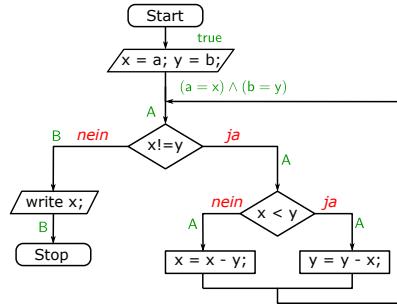
$$\begin{aligned} & (\neg(y > x) \wedge (x > y) \wedge (y > 0)) \vee ((y > x) \wedge (y > x) \wedge (x > 0)) \equiv \\ & \equiv ((x \geq y) \wedge (x > y) \wedge (y > 0)) \vee ((y > x) \wedge (x > 0)) \equiv \\ & \equiv ((x > y) \wedge (y > 0)) \vee ((y > x) \wedge (x > 0)) \equiv \\ & \equiv (x > 0) \wedge (y > 0) \wedge (x \neq y) \end{aligned}$$

## Beispiel ggT VI



- Verzweigungsbedingung und Nachbedingungen
  - Bedingung  $b \equiv (x < y)$
  - nein-Zweig:  $\neg b \wedge A \equiv (x \geq y) \wedge (ggT(a, b) = ggT(x, y))$
  - ja-Zweig:  $b \wedge A \equiv (x < y) \wedge (ggT(a, b) = ggT(x, y))$
- Schwächste Vorbedingung
  - $\mathbf{WP}[\![b]\!] \equiv (\neg b \wedge A) \vee (b \wedge A) \equiv A$

## Beispiel ggT VII



- Zweite Verzweigung analog
  - Bedingung  $b \equiv (x \neq y)$
  - nein-Zweig:  $\neg b \wedge B \equiv \neg(x \neq y) \wedge A \wedge (x = y) \equiv A \wedge (x = y)$
  - ja-Zweig:  $b \wedge A \equiv A \wedge (x \neq y)$
- Schwächste Vorbedingung
  - $\mathbf{WP}[\![b]\!] \equiv (A \wedge (x = y)) \vee (A \wedge (x \neq y)) \equiv A$

## Zusammenstellung der Methode

- Annotiere jeden Programmfpunkt mit einer Zusicherung
- Überprüfe für jede Anweisung  $S$ 
  - zwischen zwei Zusicherungen  $\mathcal{V}$  und  $\mathcal{N}$ , dass
  - $\mathcal{V}$  die schwächste Vorbedingung von  $S$  für  $\mathcal{N}$  impliziert, d.h.
  - $\mathcal{V} \Rightarrow \mathbf{WP}[\![S]\!](\mathcal{N})$
- Überprüfe für jede Verzweigung mit Bedingung  $b$ , ob
  - die Zusicherung  $\mathcal{V}$  vor der Verzweigung
  - die schwächste Vorbedingung für die Nachbedingungen  $\mathcal{N}_0$  und  $\mathcal{N}_1$  der Verzweigung impliziert, d.h.
  - $\mathcal{V} \Rightarrow \mathbf{WP}[\![b]\!](\mathcal{N}_0, \mathcal{N}_1)$

## Verifikation in der Praxis

- Floyds Methode ist für den Menschen gedacht
- Strukturiertere Methode von C.A.R. Hoare (1969)
  - nicht beschränkt auf iterative Verfahren
  - lässt sich automatisieren
  - Hoare-Logik
- Leider nur bei einfachen Programmen anwendbar
- Programme mit globalem Zustand praktisch nicht testbar
  - keine objektorientierten Programme
  - keine nebenläufigen Programme
  - sehr gut testbar sind funktionale Programme

# Literatur

-  H. P. Gumm und M. Sommer.  
Einführung in die Informatik — Kapitel 2.8.  
Oldenbourg Verlag, 7. Ausgabe, 2006, ISBN 978-3-486-58115-7.
-  Helmut Seidl  
Informatik 2 — Kapitel 1.  
Vorlesung an der TU München im WS 2008/09.
-  Wolfgang Küchlin und Andreas Weber  
Einführung in die Informatik — Kapitel 17.2.  
Springer-Verlag, 2005, 3. Auflage, ISBN 3-540-20958-1.