Übersicht

Einführung

- Korrektheit oder *Leistet Algorithmus das, was er soll?*
  - Was bedeutet *was er soll*?
  - Welche Möglichkeiten gibt es, Korrektheit zu zeigen?
- Komplexität oder *Welchen Aufwand benötigt ein Algorithmus?*
  - Wir haben Sortieralgorithmen bereits dahingehend analysiert.
  - Wie kann man Aufwand sinnvoll fassen und vergleichen?
  - Wie kann man Aufwand systematisch messen oder abschätzen?
  - Kann man *Probleme* nach Aufwand klassifizieren?
- Literatur
  - [Saake&Sattler] (Kapitel 7.2 und 7.3)
  - ergänzend [Goodrich&Tamassia] , [Sedgewick] (Komplexität)

Korrektheit von Algorithmen

- Korrektheit ist eine nichttriviale semantische Eigenschaft
  - Korrektheit ist nicht entscheidbar
  - *Es gibt keinen Algorithmus, der Korrektheit eines beliebigen Algorithmus überprüfen kann!*
- Welche Möglichkeiten bleiben?
  - Korrektheitsbeweis im Einzelfall
  - Pragmatisches (und systematisches) Testen

- Korrektheit ist immer *relativ* zu einer Spezifikation
  - Spezifikation = eindeutige Festlegung der berechneten Funktion und der Terminierung

- Beispiel: Verhalten eines Sortieralgorithmus
  - Eingabe: unsortierte Liste (a_i)
  - *Nach Ausführung* soll gelten $\forall (i \leqslant j)(a_i \leqslant a_j)$
  - Diese Spezifikation umfasst z.B. *nicht* Stabilität!

Nachweis der Korrektheit

- Durch Verifikation
  - Formaler (mathematisch rigider) Beweis der Korrektheit
  - Verifikation bezüglich einer *formalen* Spezifikation
- Durch Validierung (*validation*)
  - Nichtformaler Nachweis bezüglich formaler oder informeller Spezifikation
  - z.B. durch systematisches Testen für *alle* möglichen Eingaben
- In der *Praxis* z.B. für komplexe Softwaresysteme
  - Verifikation oft zu teuer oder nicht praktikabel
  - Einschränkung der Spezifikation für Validierung auf einige *mögliche* Eingaben (Testfälle)
  - Validierung oft eher Test auf *Plausibilität* als auf Korrektheit!
  - *Algorithmen sollten immer wenigstens validiert werden!*

Übersicht

Verifikation von Algorithmen

- Je nach Programmierparadigma bieten sich bestimmte Beweismethoden an
- Imperative Programmierung
  - Zentral: *Zustandsänderung*
  - Spezifikation jeweils durch Vor- und Nachbedingungen
  - Schleifeninvariante
- Funktionale Programmierung
  - Zentral: *Funktions-/Termauswertung*
  - Bekannte mathematische Beweismethoden, z.B. Vollständige Induktion für rekursive Funktionsauswertung

Verifikation: Imperative Programmierung

- Anweisung (bzw. Sequenz) α bedingt Zustandsänderung
- Spezifiziere Semantik von α über Zustand:
  - Vorbedingung `VOR`
  - Nachbedingung `NACH`
- Wir schreiben

$$\{\ \texttt{VOR}\ \}\ \ \alpha\ \ \{\ \texttt{NACH}\ \}$$

Gilt `VOR` *unmittelbar vor* Ausführung von α und terminiert α, so gilt `NACH` *unmittelbar nach* Ausführung von α.

  - Erfüllt, falls α *nicht* terminiert! – Unabhängig von `VOR`/`NACH`
  - Erfüllt, falls `VOR` *nicht* gilt! – Unabhängig von α und `NACH`
- Vor- und Nachbedingungen als logische Ausdrücke
  - Wir betrachten einfache Beispiele.
  - Zur Prädikatenlogik, siehe auch Vorlesung *Logik*

Partielle und totale Korrektheit

- Sei α
  - eine Anweisung *oder*
  - eine Sequenz von Anweisungen *oder*
  - ein imperatives Programm.
- α heißt partiell korrekt bezüglich `VOR` und `NACH`, *gdw.* $\{\ \texttt{VOR}\ \}\ \alpha\ \{\ \texttt{NACH}\ \}$ erfüllt bzw. wahr ist.
- α heißt total korrekt bezüglich `VOR` und `NACH`, *gdw.* gilt
  - α ist partiell korrekt bezüglich `VOR` und `NACH` **und**
  - α *terminiert* immer dann, wenn vorher `VOR` gilt.

Beispiele für Vor- und Nachbedingungen

- $\{\, X = 0 \,\}$ `X:=X+1` $\{\, X = 1 \,\}$ wahr
- $\{\, \texttt{true} \,\}$ `X:=Y` $\{\, X = Y \,\}$ wahr

Seien $a, b \in \mathbb{Z}$.

- $\{\, Y = a \,\}$ `X:=Y` $\{\, X = a \wedge Y = a \,\}$ wahr
- $\{\, X = a \wedge Y = b \wedge a \neq b \,\}$ `X:=Y; Y:=X` $\{\, X = b \wedge Y = a \,\}$ falsch
- $\{\, X = a \wedge Y = b \,\}$ `Z:=X; X:=Y; Y:=Z` $\{\, X = b \wedge Y = a \,\}$ wahr

- $\{\, \texttt{false} \,\}$ α $\{\, \texttt{NACH} \,\}$ wahr für alle α und $\{\, \texttt{NACH} \,\}$
- $\{\, \texttt{true} \,\}$ α $\{\, \texttt{false} \,\}$ wahr *gdw.* α *nicht* terminiert
- $\{\, X = a \,\}$ `while X≠0 do X:=X-1 od` $\{\, X = 0 \,\}$ wahr,

  ... *denn* $a < 0 \Rightarrow$ Schleife terminiert *nicht*!

Verschiedene Anweisungstypen

- Atomare Anweisungen `X:=X+1`
- Sequenz `X:=Y; X:=X+1`
- Auswahl `if X=0 then Y:=1 fi`
- Iteration `while X>0 do X:=X-1 od`

Das Hoare-Kalkül definiert formale Regeln für die Bestimmung von Vor- bzw. Nachbedingungen zum Nachweis der Korrektheit.

Im folgenden eine kurze, informelle Einführung und Beispiele.

- Anweisung `X:=t` mit Variable `X` vom Typ `int` und Term t
- Transformiere Vorbedingung in Nachbedingung:
  Ersetze jedes Auftreten von t *durch* `X`
- Beispiele
  - $\{\, 3 > 0 \,\}$ `X:=3` $\{\, X > 0 \,\}$
  - $\{\, X+1 > 0 \,\}$ `X:=X+1` Nachbedingung? *Hilfskonstruktion...*
    $\{\, X+1 > 0 \,\}$ `X':=X+1` $\{\, X+1 > 0 \,\}$ $\Leftrightarrow$
    $\{\, X+1 > 0 \,\}$ `X':=X+1` $\{\, X' > 0 \,\}$ $\Leftrightarrow$
    $\{\, X+1 > 0 \,\}$ `X:=X+1` $\{\, X > 0 \,\}$
  - $\{\, X > 0 \,\}$ `X:=X-1` $\{\, X \geqslant 0 \,\}$

    Denn $t = X - 1$ und Vorbedingung
    $X > 0 \;\Leftrightarrow\; X - 1 + 1 > 0 \;\Leftrightarrow\; t + 1 > 0.$

    Ersetze t durch X, also Nachbedingung $X + 1 > 0 \;\Leftrightarrow\; X \geqslant 0$

Bemerkung zur Konstruktion

- Das Hoare-Kalkül wird i.d.R. **rückwärts** angewandt:

  Transformiere Nachbedingung in Vorbedingung.

- Diese Richtung ist i.a. einfacher!
- Zusätzlich kennen wir das gewünschte Ergebnis ja schon, und wir wollen formal zeigen, dass es tatsächlich gilt.

- Für

  $$\{\ \texttt{VOR}\ \}\ \ \texttt{X:=t}\ \ \{\ \texttt{NACH}\ \}$$

  ergibt sich dann $\{\ \texttt{VOR}\ \}$ aus $\{\ \texttt{NACH}\ \}$ durch Ersetzen jedes Vorkommens von X in $\{\ \texttt{NACH}\ \}$ durch t.

- Beispiel

  $$\begin{aligned} \{\ ???\ \}\ \ &\texttt{X:=X-1}\ \ \{\ X \geqslant 0\ \} \\ \{\ X-1 \geqslant 0\ \}\ \ &\texttt{X:=X-1}\ \ \{\ X \geqslant 0\ \} \\ \{\ X > 0\ \}\ \ &\texttt{X:=X-1}\ \ \{\ X \geqslant 0\ \} \end{aligned}$$

Verifikation von Sequenzen

- Sequenz $\alpha;\beta$ mit

$$\{\ \mathtt{VOR}\ \}\ \ \alpha;\beta\ \ \{\ \mathtt{NACH}\ \}$$

- *Aufspalten in einzelne Schritte*
- Bestimme *Zwischenbedingung* `MITTE`, so dass

$$\{\ \mathtt{VOR}\ \}\ \ \alpha\ \ \{\ \mathtt{MITTE}\ \}\quad \text{und}\quad \{\ \mathtt{MITTE}\ \}\ \ \beta\ \ \{\ \mathtt{NACH}\ \}$$

- Beispiel: $\{\ 3>0\ \}$ `X:=3; Y:=X;` Nachbedingung?
  - $\big[\ \{\ 3>0\ \}\ \mathtt{X:=3;}\ \{\ X>0\ \}\ \big] \wedge \big[\ \{\ X>0\ \}\ \mathtt{Y:=X;}\ \{\ Y>0\ \}\ \big]$
  - $\mathtt{MITTE} \Leftrightarrow \{X>0\}$
  - $\{\ 3>0\ \}\ \mathtt{X:=3;}\ \mathtt{Y:=X;}\ \{\ Y>0\ \}$ bzw.
    $\{\ 3>0\ \}\ \mathtt{X:=3;}\ \mathtt{Y:=X;}\ \{\ X>0 \wedge Y>0\ \}$

Verifikation bei Auswahl

- Auswahl `if` B `then` α `else` β `fi` mit

  $$\{\ \mathtt{VOR}\ \}\ \texttt{if}\ \mathrm{B}\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\ \texttt{fi}\ \{\ \mathtt{NACH}\ \}$$

- *Aufspalten in Fälle* B *und* $\neg\mathrm{B}$
- Zeige

  $$\{\ \mathtt{VOR} \wedge \mathrm{B}\ \}\ \alpha\ \{\ \mathtt{NACH}\ \} \quad \text{und} \quad \{\ \mathtt{VOR} \wedge \neg\mathrm{B}\ \}\ \beta\ \{\ \mathtt{NACH}\ \}$$

  - `NACH` gilt in *beiden* Fällen!

- Beispiel: $\{\ X > 0\ \}$ `if` $Y > 0$ `then X:=X+Y fi` $\{\ X > 0\ \}$
  - $\{\ X > 0 \wedge Y > 0\ \}$ `X:=X+Y` $\{\ X > 0\ \}$ ✓
  - $\{\ X > 0 \wedge Y \leqslant 0\ \}$ `X:=X` $\{\ X > 0\ \}$ ✓

Verifikation von Schleifen

- Schleife `while` B `do` α `od` mit

$$\{\ \mathtt{VOR}\ \}\ \mathtt{while}\ B\ \mathtt{do}\ \alpha\ \mathtt{od}\ \{\ \mathtt{NACH}\ \}$$

- Bestimme Schleifeninvariante P:

  P gilt unmittelbar vor, *während* und unmittelbar nach Iteration

1. Zeige $\mathtt{VOR} \Rightarrow P$ (vor Eintritt in `while`)
2. Zeige $\{\ P \wedge B\ \}\ \alpha\ \{\ P\ \}$ (Iteration über Schleifenrumpf)
3. Zeige $P \wedge \neg B \ \Rightarrow\ \mathtt{NACH}$ (Verlassen der Schleife)

- Beispiel: $\{\ X > 0\ \}$ `while` $\mathtt{X}{\neq}0$ `do` `X:=X-1` `od` $\{\ X = 0\ \}$
  1. Invariante $P\ \Leftrightarrow\ X \geqslant 0$. Es gilt $\mathtt{VOR} \Rightarrow P$.
  2. $\{\ X > 0\ \}$ `X:=X-1` $\{\ X \geqslant 0\ \}$ ✓
  3. $X \geqslant 0 \wedge X = 0\ \Rightarrow\ X = 0$

- Spezifikation

  $\{\, Y \geqslant 0 \,\}$ `MULT` $\{\, Z = X \cdot Y \,\}$

- Schleifeninvariante

  $$P \;\Leftrightarrow\; X \cdot Y = Z + W \cdot X$$

Ansatz: Ergebnis $X \cdot Y$ durch Teilergebnis Z ausdrücken

```
MULT:
var W,X,Y,Z : int;
input X,Y;
Z:=0;
W:=Y;
while W≠0 do
  Z:=Z+X;
  W:=W-1;
od;
output Z;
```

- Schritt 1: Zeige, P gilt bei Eintritt in die Schleife

  $\{\, Y \geqslant 0 \,\}$ `Z':=0; W':=Y` $\{\, X \cdot Y = Z' + W' \cdot X \,\}$, denn

  $X \cdot Y = Z' + W' \cdot X = 0 + Y \cdot X$ (Werte ersetzen)

Korrektheit von `MULT` (2)

- Invariante P $\Leftrightarrow$ $X \cdot Y = Z + W \cdot X$ gilt bei Eintritt in Schleife

  `while` W$\neq$ 0 `do Z:=Z+X; W:=W-1 od`

- Schritt 2: Zeige für Iteration

  $$\{\ P \wedge W \neq 0\ \}\ \texttt{Z:=Z+X; W:=W-1}\ \{\ P\ \}$$

- Setze $\{\ P \wedge W \neq 0\ \}\ \texttt{Z}':=\texttt{Z+X; W}':=\texttt{W-1}\ \{\ P'\ \}$ mit

  $$P' \Leftrightarrow X' \cdot Y' = Z' + W' \cdot X'$$

  und zeige P $\Leftrightarrow$ P$'$

- Einsetzen von $X' = X$, $Y' = Y$, $Z' = Z + X$, $W' = W - 1$ liefert

  $$P' \Leftrightarrow X \cdot Y = (Z + X) + (W - 1) \cdot X = Z + W \cdot X \Leftrightarrow P$$

Korrektheit von MULT (3)

- Invariante $\mathrm{P} \Leftrightarrow \mathrm{X} \cdot \mathrm{Y} = \mathrm{Z} + W \cdot \mathrm{X}$ gilt während Iteration

  `while` $\mathrm{W} \neq 0$ `do Z:=Z+X; W:=W-1 od`

- Schritt 3: Beim Verlassen der Schleife gilt $\mathrm{P} \wedge W = 0$, zeige

$$\mathrm{P} \wedge W = 0 \;\Rightarrow\; \mathrm{Z} = \mathrm{X} \cdot \mathrm{Y}$$

$$\mathrm{P} \wedge W = 0 \quad\Rightarrow\quad \mathrm{X} \cdot \mathrm{Y} = \mathrm{Z} + W \;\wedge\; W = 0 \quad\Rightarrow\quad \mathrm{X} \cdot \mathrm{Y} = \mathrm{Z} \quad \checkmark$$

- Hier liefert *Nachbedingung* $\mathrm{Z} = \mathrm{X} \cdot \mathrm{Y}$ das Ergebnis von `MULT`

- Somit ist *partielle Korrektheit* bewiesen
- Beweis der Terminierung fehlt (in diesem Fall einfach)
  - $\mathrm{Y} \geqslant 0 \Rightarrow W \geqslant 0$ vor `while`
  - $W = 0 \quad \Rightarrow$ keine Iteration (`MULT` terminiert)
  - Für $W > 0$ wird W in *jeder* Iteration um eins erniedrigt.
    `MULT` terminiert nach genau Y Iterationen

Was berechnet das folgende Programm?

- Das folgende Programm beschreibt einen Algorithmus.
- Es ist nicht einfach zu sehen, welchen ...!?

```
XYZ:
var     W,X,Y,Z : int;
input   X;
Z:=0; W:=1; Y:=1;
while W⩽X do
  Z:=Z+1; W:=W+Y+2; Y:=Y+2;
od;
output Z;
```

- Eine Möglichkeit: verschiedene Eingabewerte probieren
- Immer noch schwer!
- *Später:* Wir zeigen $[\mathtt{XYZ}](\mathrm{X}) = \lfloor\sqrt{\mathrm{X}}\rfloor$

- Behauptung: XYZ berechnet $\lfloor\sqrt{X}\rfloor$
- Spezifikation

$$\{\ X \geqslant 0\ \}\ \texttt{XYZ}\ \{\ Z = \lfloor\sqrt{X}\rfloor\ \}$$

wobei

$$Z = \lfloor\sqrt{X}\rfloor \ \Leftrightarrow\ Z^2 \leqslant X < (Z+1)^2$$

```
XYZ:
var W,X,Y,Z : int;
input X;
Z:=0; W:=1; Y:=1;
while W≤X do
   Z:=Z+1;
   W:=W+Y+2;
   Y:=Y+2;
od;
output Z;
```

- Sei

α = `Z:=0; W:=1; Y:=1`

β = `Z:=Z+1; W:=W+Y+2; Y:=Y+2`

- Invariante

$$P \ \Leftrightarrow\ Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$$

Korrektheit von XYZ (1)

- XYZ: α ; **while** W⩽X **do** β **od**

  $\alpha \quad = \quad$ `Z:=0; W:=1; Y:=1`

  $$P \quad \Leftrightarrow \quad Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$$

- Zeige, P gilt bei Eintritt in die Schleife

  $$\{\, X \geqslant 0 \,\} \;\; \alpha \;\; \{\, P \,\}$$

- Einsetzen der Werte für Z, W, Y nach α in P liefert

  $$0^2 \leqslant X \wedge (0+1)^2 = 1 \wedge 2 \cdot 0 + 1 = 1 \wedge 1 > 0 \quad \Leftrightarrow \quad 0 \leqslant X \quad \checkmark$$

- Betrachte Schleife `while W⩽X do β od`

$$\beta \quad = \quad \texttt{Z:=Z+1; W:=W+Y+2; Y:=Y+2}$$

$$P \quad \Leftrightarrow \quad Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$$

- Zeige für Iteration $\{\ P \wedge W \leqslant X\ \}\ \ \beta\ \ \{\ P\ \}$
- *Spalte Konjunktion in* P *in 4 Teile auf (rechte Seite) und nutze nur Teile der Vorbedingung (linke Seite).*

1 $\{\ (Z+1)^2 = W \wedge W \leqslant X\ \}\ \ \beta\ \ \{\ Z'^2 \leqslant X\ \} \quad \Leftrightarrow$

$\{\ (Z+1)^2 \leqslant X\ \}\ \ \beta\ \ \{\ (Z+1)^2 \leqslant X\ \} \quad \Rightarrow$

$\{\ (Z+1)^2 \leqslant X\ \}\ \ \beta\ \ \{\ Z^2 \leqslant X\ \} \quad \checkmark$

- Betrachte Schleife `while W⩽X do β od`

$$\beta \quad = \quad \texttt{Z:=Z+1; W:=W+Y+2; Y:=Y+2}$$

$$P \quad \Leftrightarrow \quad Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$$

- Zeige für Iteration $\{ P \wedge W \leqslant X \} \; \beta \; \{ P \}$

1 $\{ (Z+1)^2 \leqslant X \} \; \beta \; \{ Z^2 \leqslant X \}$ ✓

2 $\{ (Z+1)^2 = W \wedge 2Z+1 = Y \} \; \beta \; \{ W' = (Z'+1)^2 \}$

$$\begin{array}{lcll} W' & = & (Z'+1)^2 & \Leftrightarrow \\ W+Y+2 & = & (Z+1+1)^2 & \Leftrightarrow \\ W+(2Z+1)+2 & = & Z^2+4Z+4 & \Leftrightarrow \\ W & = & Z^2+2Z+1 & \Leftrightarrow \\ W & = & (Z+1)^2 & \end{array}$$

$\{ (Z+1)^2 = W \wedge 2Z+1 = Y \} \; \beta \; \{ W = (Z+1)^2 \}$ ✓

Korrektheit von XYZ (2.3)

- Betrachte Schleife `while W⩽X do β od`

$$\beta \quad = \quad \texttt{Z:=Z+1; W:=W+Y+2; Y:=Y+2}$$

$$P \quad \Leftrightarrow \quad Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$$

- Zeige für Iteration $\{\ P \wedge W \leqslant X\ \}\ \beta\ \{\ P\ \}$

1. $\{\ P \wedge W \leqslant X\ \}\ \beta\ \{\ Z^2 \leqslant X\ \}$ ✓
2. $\{\ P \wedge W \leqslant X\ \}\ \beta\ \{\ W = (Z+1)^2\ \}$ ✓
3. $\{\ 2Z+1 = Y\ \}\ \beta\ \{\ 2Z'+1 = Y'\ \}$

$$\begin{array}{rcll} 2Z'+1 & = & Y' & \Leftrightarrow \\ 2(Z+1)+1 & = & Y+2 & \Leftrightarrow \\ 2Z+1 & = & Y & \end{array}$$

$\{\ 2Z+1 = Y\ \}\ \beta\ \{\ 2Z+1 = Y\ \}$ ✓

Korrektheit von XYZ (2.4)

- Betrachte Schleife `while W⩽X do β od`

  $\beta \quad = \quad$ `Z:=Z+1; W:=W+Y+2; Y:=Y+2`

  $P \quad \Leftrightarrow \quad Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$

- Zeige für Iteration $\{\, P \wedge W \leqslant X \,\} \;\; \beta \;\; \{\, P \,\}$

1. $\{\, P \wedge W \leqslant X \,\} \;\; \beta \;\; \{\, Z^2 \leqslant X \,\}$ ✓
2. $\{\, P \wedge W \leqslant X \,\} \;\; \beta \;\; \{\, W = (Z+1)^2 \,\}$ ✓
3. $\{\, 2Z+1 = Y \,\} \;\; \beta \;\; \{\, 2Z+1 = Y \,\}$ ✓
4. $\{\, Y > 0 \,\} \;\; \beta \;\; \{\, Y > 0 \,\}$ ✓

- Damit insgesamt $\{\, P \wedge W \leqslant X \,\} \;\; \beta \;\; \{\, P \,\}$ ✓

Korrektheit von XYZ (3)

- Betrachte Schleife `while W⩽X do β od`

  $\beta \;=\;$ `Z:=Z+1; W:=W+Y+2; Y:=Y+2`

  $P \;\Leftrightarrow\; Z^2 \leqslant X \wedge (Z+1)^2 = W \wedge 2Z+1 = Y \wedge Y > 0$

- Noch z.z. $P \wedge W > X \;\Rightarrow\; Z = \lfloor\sqrt{X}\rfloor$ (Nachbedingung)
- $Z = \lfloor\sqrt{X}\rfloor \;\Leftrightarrow\; Z^2 \leqslant X \;\wedge\; X < (Z+1)^2$
  - $P \;\Rightarrow\; Z^2 \leqslant X$
  - $W = (Z+1)^2 \wedge W > X \;\Rightarrow\; X < (Z+1)^2$ ✓
- Damit ist *partielle Korrektheit* gezeigt bezüglich

  $\{\, X \geqslant 0 \,\}$ XYZ $\{\, Z = \lfloor\sqrt{X}\rfloor \,\}$

Ansatz zum Beweis der Terminierung

Konstruiere Folge von Werten $u_0, u_1, u_2, \ldots,$ für die gilt

- Beim *Eintritt* in Schleife ist $u_0 > 0$ definiert.
- *Schleifenrumpf* berechnet u_{i+1} aus u_i.
- Für alle i gilt $u_{i+1} < u_i$.
- Alle $u_i > 0$. –
  Sobald $u_{i+1} < 0$ bedeutet das ein *Verlassen* der Schleife

XYZ terminert!

- Anwenden des Ansatzes für `XYZ`: α ; **while** `W`$\leqslant$`X` **do** β **od**

$$\alpha \;=\; \texttt{Z:=0; W:=1; Y:=1}$$
$$\beta \;=\; \texttt{Z:=Z+1; W:=W+Y+2; Y:=Y+2}$$

- Betrachte $u = X - W$
  - Zum *Eintritt* in Schleife gilt $1 \leqslant W \leqslant X \;\rightarrow\; u \geqslant 0$
  - *Schleifenrumpf* (ein Iterationsschritt)
    - Vorher $u = X - W$
    - Nachher $u' = X' - W'$ mit $X' = X,\; W' = W + Y + 2$

  somit gilt $u' < u$, denn $W > 0$ und $Y > 0$
  - D.h., die Iteration *muss* irgendwann abbrechen mit $u' \leqslant 0$

- Damit ist *totale Korrektheit* gezeigt bezüglich

$$\{\, X \geqslant 0 \,\}\ \texttt{XYZ}\ \{\, Z = \lfloor \sqrt{X} \rfloor \,\}$$

Bemerkungen zur Verifikation

- Verifikation benötigt Beweise
- Es gibt *Beweistechniken* — keine automatische Beweisführung
- Grundmuster: Zeige
  - Vorbedingung
  - Invariante(n)
  - Nachbedingung
  - Terminierung
- *Konstruktion* des Algorithmus nach Spezifikation
- Weitere Beispiele zur Übung
  - Fakultätsfunktion `FAC`
  - Addition von natürlichen Zahlen `ADD`
  - Berechnung der Quadratwurzel (nur Invariante)
- Beachte
  - Eingabe sollte *nicht* überschrieben werden (z.B. `FAC`)
  - Schwieriger für numerischen Algorithmen (reelle Zahlen)

Ausflug in die Praxis: Validierung

- Validiere Spezifikation z.B. durch *unit tests*
  - Auch Modultest oder Komponententest
  - Betrachte einzelne, abgeschlossene Einheiten (*unit*)
  - Definiere Testfälle (auch und v.a. für fehlerhafte Eingabe)
  - Automatisiertes Testen
  - *Unser Einreichungssystem basiert auf solchen Tests.*
- Beachte Entwicklungszyklus: z.B. Regressionstests
  - Automatische Wiederholung für verschiedene Versionen
  - *Welche (negativen) Auswirkungen hat eine Modifikation?*
- Beides lohnt sich schon für kleine Projekte
  - Vor allem im Team: *Wen/was stören meine Änderungen?*
- Testen ersetzt i.a. keine Verifikation!

Minimalziel

Fehlerhaftes Verhalten *frühestmöglich* erkennen und beheben!

Assertions — Validierung im Kleinen

- Manche Programmiersprachen erzwingen Prüfung von Vor-/Nachbedingungen/Invarianten
- z.B. Eiffel: *design by contract*
  - `require` Vorbedingung
  - `ensure` Nachbedingung
- Java bietet *assertions* (*to assert* = *versichern [, dass gilt]*)
- **Syntax**: `assert B;` oder `assert B: msg;`
  - Bedingung B ist ein Ausdruck vom Typ `boolean`
  - optionale Fehlerbeschreibung als `msg.toString()`, i.d.R. ist `msg` bereits Typ `String`
- **Semantik**
  - aktivieren durch `java -ea` (*enable assertions*)
  - Programmabbruch (genauer `AssertionError`), wenn $\neg B$
- Beispiel: `public static void sort3(int[] a)`

Java *assertions*

- Explizite Aktivierung zur *Laufzeit* durch `java -ea`
  - Tests kosten Rechenzeit
  - Aktivieren während Entwicklung
  - Deaktiviert in „Release Version“ z.B. beim Kunden
- Deshalb
  - *assertions* dürfen keine Seiteneffekte haben!
  - Keine Zustandsänderung (z.B. Zuweisung) in *assertion*!
  - *assertions* ersetzen keine Fehlerbehandlung!
- Warum (möglichst viele) *assertions*?
  - Fehler früh erkennen!
  - *assertions* kosten keine Rechenzeit, wenn deaktiviert
  - *assertions* können Kommentare ersetzen! *(Kommentar=Test)*
- Siehe auch Java Dokumentation
- *assertions* ersetzen keine Tests!
- Ähnliches Konzept in C/C++ als *Makro* `assert()`
  Deaktivierung zur *Compile*zeit durch `-DNDEBUG`

Beispiele: Java *assertions*

```
public static void sort(int[] a) {
  ...                         // sorting algorithm
  assert isSorted(a) : "isSorted failed";
}
```

```
 switch (i) {
   case ...
   default:
     assert false : "invalid value";
 }
```

```
  ... assert (--i>0); // ϟ NO!!! -- side effect!
```

ϟ Verboten: Zustandsänderung (*wird von* `javac` *nicht überprüft*)

Beispiel: Java *assertions*

```
public static int xyz(int x) {
  assert x>=0 : "require nonnegative input";

  int z=0, w=1, y=1;

  while (w>=0) {
    assert ( (z*z<x)              &&
             ((z+1)*(z+1)==w)  &&
             (2*z+1==y)           &&
             (y>0) ) : "invariant";

    ++z; w+=y+2; y+=2;
  }
  assert z*z<=x && x<(z+1)*(z+1);

  return z;
}
```

Verifikation in Funktionaler Programmierung

- Im Gegensatz zu Imperativer Programmierung
  - Auswertung von Termen
  - Keine Zustandsänderung
  - *Rekursion* ersetzt Schleifen
- Typische Beweistechnik: *Vollständige Induktion*, z.B. zeige
  - Geschlossener Ausdruck für Ergebnis
  - Invarianten (über Rekursionsschritte)
  - Terminierung der Rekursion
- *Ähnlich für jede rekursive Beschreibungen von Algorithmen*
- Einfache Beispiele
  - $\textbf{fac}(x)$ berechnet *nach Konstruktion* Fakultätsfunktion $x!$
  - Ebenso berechnet $\textbf{fib}(x)$ Fibonacci-Zahlen
  - Etwas schwieriger:
    $\textbf{add}(x,y) \;=\; \texttt{if}\ y=0\ \texttt{then}\ x\ \texttt{else}\ \textbf{add}(x+1,y-1)\ \texttt{fi}$

Beispiel: Vollständige Induktion für $\mathbf{add}(x, y)$

- Geg.: $\mathbf{add}(x,y) \;=\; \texttt{if}\ y=0\ \texttt{then}\ x\ \texttt{else}\ \mathbf{add}(x+1,y-1)\ \texttt{fi}$
- Beh.: $\mathbf{add}(x,y) = x+y$ für $x \in \mathbb{Z}, y \in \mathbb{N}_0$
- Beweis durch vollständige Induktion

1. Induktionsanfang: Behauptung gilt für $y=0$
   $\mathbf{add}(x,0) \;=\; x \;=\; x+0$ ✓
2. Induktionsvoraussetzung: Behauptung gilt für $0 \leqslant z \leqslant y$:
   $\mathbf{add}(x,z) = x+z$ für alle $0 \leqslant z \leqslant y$
3. Induktionsschritt: z.z. Behauptung gilt für $y+1$

$$\mathbf{add}(x,y+1) \overset{y+1>0}{=} \mathbf{add}(x+1,y)$$

$$\overset{\text{Voraussetzung}}{=} (x+1)+y \;=\; x+(y+1) \quad ✓$$

Was berechnet die folgende Funktion?

- Zum Abschluss eine etwas kuriose Funktion

$$f(x) \;=\; \texttt{if}\ x > 100\ \texttt{then}\ x - 10\ \texttt{else}\ f(f(x+11))\ \texttt{fi}$$

- Nicht einfach zu sehen. Wir probieren ...

$$\begin{aligned}
f(100) &= f(f(111)) = f(101) = 91 \\
f(99) &= f(f(110)) = f(100) = \cdots = 91 \\
f(98) &= f(f(109)) = f(99) = \cdots = 91 \\
\cdots &= \cdots = 91 \ \ \textbf{?}
\end{aligned}$$

- Vermutung f ist äquivalent zu g mit

$$g(x) \;=\; \texttt{if}\ x > 100\ \texttt{then}\ x - 10\ \texttt{else}\ 91\ \texttt{fi}$$

- Äquivalenz besteht tatsächlich: *McCarthys 91-Funktion*, *Beweis folgt später* (Korrektheit von Algorithmen)

- Frage: Kann man Äquivalenz von Algorithmen zeigen? Wie?

Beispiel: McCarthys 91-Funktion (1)

- Geg.: $f(x) = \texttt{if } x > 100 \texttt{ then } x - 10 \texttt{ else } f(f(x+11)) \texttt{ fi}$
- Beh.: $f(x) = g(x)$ mit
  $g(x) = \texttt{if } x > 100 \texttt{ then } x - 10 \texttt{ else } 91 \texttt{ fi}$

1. Induktionsanfang: Behauptung gilt für $x > 100$
   $x > 100 \;\Rightarrow\; f(x) = x - 10 = g(x)$ ✓
2. Induktionsvoraussetzung: Behauptung gilt für $y \geqslant x$:
   $f(y) = g(y)$ für alle $y \geqslant x$
3. Induktionsschritt: z.z. Behauptung gilt für $x - 1$

1. Fall: $101 > x \geqslant 91$ $\Rightarrow\; x + 10 > 100$

$$\begin{aligned} f(x-1) &= f(f(x-1+11)) = f(f(x+10)) \\ &= f(x+10-10) = f(x) \overset{(IV)}{=} g(x) = 91 \\ &= g(x-1) \quad \checkmark \end{aligned}$$

- Geg.: $f(x) \;=\; \texttt{if } x > 100 \texttt{ then } x - 10 \texttt{ else } f(f(x+11)) \texttt{ fi}$
- Beh.: $f(x) \;=\; g(x)$ mit
  $g(x) \;=\; \texttt{if } x > 100 \texttt{ then } x - 10 \texttt{ else } 91 \texttt{ fi}$

1. Induktionsanfang: Behauptung gilt für $x > 100$
   $x > 100 \;\Rightarrow\; f(x) = x - 10 = g(x)$ ✓
2. Induktionsvoraussetzung: Behauptung gilt für $y \geqslant x$:
   $f(y) = g(y)$ für alle $y \geqslant x$
3. Induktionsschritt: z.z. Behauptung gilt für $x - 1$

1. Fall: $101 > x \geqslant 91$ ✓

2. Fall: $90 \geqslant x$: $\Rightarrow\; x + 10 \leqslant 100$

$$
\begin{aligned}
f(x-1) &= f(f(x+10)) \overset{\text{IV}}{=} f(g(x+10)) \;=\; f(91) \\
&= g(91) \;=\; 91 \\
&= g(x-1) \quad \checkmark
\end{aligned}
$$

Kurze Zusammenfassung

- Korrektheit *relativ* zu Spezifikation
- Nachweis durch
  - Verifikation
  - Validierung
- Verifikation
  - Zustandsänderung: Vorbedingung und Nachbedingung
  - Invariante
  - Terminierung — partielle und totale Korrektheit
  - Vollständige Induktion
- Validierung in der Praxis
  - Systematisches und konsequentes Testen
  - Impliziert i.d.R. *nicht* Korrektheit!
  - Nutze Sprachkonstrukte wie z.B. *assertions*

- Bisher Algorithmen,
  später folgen *Spezifikationen für Datenstrukturen*

Übersicht

8 Analyse von Algorithmen: Korrektheit und Komplexität

- Einführung
- Verifikation von Algorithmen
- Komplexität von Algorithmen

- *Bisher betrachtet:* Korrektheit — *jetzt:* Aufwand
  - Sowohl *Insertion Sort* als auch *Bubble Sort* sind korrekt
  - *Insertion Sort* benötigt i.d.R. weniger Vergleiche
  - Effektivität vs Effizienz
- Fragestellungen
  - Wie kann man Aufwand messen bzw. abschätzen?
  - Wie kann man Aufwand vergleichen?
  - Wie kann man unterschiedliche (*alle* möglichen) Eingaben berücksichtigen?
  - Gibt es einen Mindestaufwand für bestimmte Klassen von Problemen?
- Erste Überlegungen bereits zu Sortieralgorithmen
  - Zähle Anzahl Vergleiche bzw. Vertauschungen
  - Betrachte schlechtesten/mittleren/besten Fall

Aufwand von Algorithmen

- Messung in Abhängigkeit von verschiedenen Eingabedaten
- Empirische Messung
  - Bestimme Laufzeit
  - z.B. mit Stoppuhr, `time` (`bash`) bzw. `/usr/bin/time`,...
  - *Problem:* Ergebnis abhängig von Rechnerumgebung
- Zähle Anzahl ausgeführter Instruktionen
  - z.B. für eine Registermaschine oder Java VM
- Abstraktion: zähle *primitive Operationen* z.B.
  - Arithmetische Operation
  - Vergleich zweier Werte
  - Ändern einer Variablen
  - Auslesen des Werts einer Variablen
  - ...

Annahme

Die Zeit zur Ausführung *einer* primitiven Operation ist *konstant*, d.h. unabhängig von der Eingabe!

Beispiel: Sequentielle Suche

- Eingabe: Zahl b und Folge von Zahlen $(a_1, a_2, \ldots, a_n)$
  - Vorgabe: a_i sind *paarweise verschieden*
- Gesucht: Index i
  - so dass $a_i = b$ (erfolgreiche Suche) oder
  - $i = n + 1$ (erfolglose Suche)

```
i:=1; while i ⩽ n ∧ b ≠ a_i do i:=i+1 od
```

- Bestimme Aufwand als Anzahl s der Zustandsänderungen
  - `i:=1` und `i:=i+1` als primitive Operation
- *Erfolglose* Suche: $s = n + 1$
- *Erfolgreiche* Suche abhängig von Eingabe b, $(a_i), 1 \leqslant i \leqslant n$
  - Relevant ist die Länge n der Folge
  - *Bester Fall*: $a_1 = b \;\Rightarrow\; s = 1$
  - *Schlechtester Fall*: $a_n = b \;\Rightarrow\; s = n$
  - Allgemein: $a_i = b \;\Rightarrow\; s = i$
  - *Wie groß ist s für gegebenes n im Mittel?*

Beispiel: Sequentielle Suche (mittlerer Aufwand)

- Erfolgreiche sequentielle Suche

  $\texttt{i:=1; while } i \leqslant n \wedge b \neq a_i \texttt{ do i:=i+1 od}$

- Modell: Wiederholte Anwendung für verschiedene Eingaben
- Benötige Modell für Verteilung von Werten in Folge (a_i)
  - Vorgabe: a_i sind paarweise verschieden, d.h. $\forall (i \neq j)(a_i \neq a_j)$
  - Annahme: *Gleichverteilung*, d.h.
    Wahrscheinlichkeit $P(a_i = b) = \frac{1}{n}$
- Betrachte erfolgreiche Suche $(a_i = b)$ für alle $1 \leqslant i \leqslant n$
  - $i = 1$: mit $P(a_1 = b)$, d.h. Aufwand $1 \cdot \frac{1}{n}$
  - $i = 2$: mit $P(a_2 = b)$, d.h. Aufwand $2 \cdot \frac{1}{n}$ $\ldots$
  - $i = j$: allgemein Aufwand $\frac{j}{n}$
  - Wir berechnen den *Durchschnitt* $\bar{s}$ über alle $1 \leqslant i \leqslant n$

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

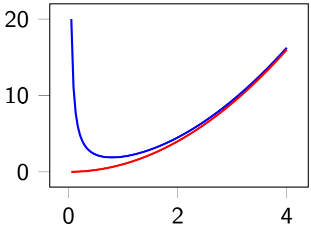
Asymptotische Notation

- Weitere Abstraktion zur Beschreibung des Aufwands
- Beschreibe Aufwand (oder *Komplexität*) durch Funktion

$$f : \mathbb{N}_0 \to \mathbb{R}_0^+$$

  - Aufwand $f(n)$
  - in Abhängigkeit von *Problemgröße* n
- Problemgröße = Größe der Eingabedaten
  - z.B. Anzahl zu durchsuchender/zu sortierender Einträge
- Aufwand
  - Anzahl primitiver Operationen zur Abschätzung der *Rechenzeit*
  - Alternativ: benötigter *Speicher*(!)
- f kann meist nicht exakt bestimmt werden, daher
  - Abschätzung im Mittel bzw. für schlechtesten Fall
  - „*Ungefähres Rechnen in Größenordnungen*"

- Beschreibe Aufwand durch Funktion $f : \mathbb{N}_0 \to \mathbb{R}_0^+$
- Angabe der asymptotischen oberen Schranke von f
  - Beschreibung des *Wachstums* von f
  - Abstrakte Beschreibung in „Größenordnungen“
  - Asymptote als einfache Vergleichsfunktion
- Asymptote = Kurve, die sich einer vorgegebenen Kurve in einem Grenzprozess beliebig annähert
  - Beispiel: $g(x) = x^2$ ist Asymptote zu $f(x) = x^2 + \frac{1}{x}$ für $x \to \infty$



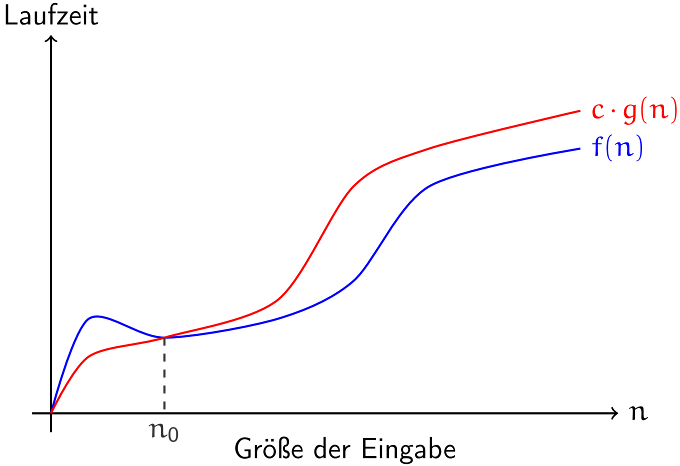
- Seien f und g Funktionen $\mathbb{N}_0 \to \mathbb{R}_0^+$.

Definition ($f \in O(g)$)

f ist $O(g)$, genau dann wenn es Konstanten $c > 0$ und $n_0 \geqslant 1$ gibt, so dass $f(n) \leqslant c \cdot g(n)$ für alle $n \geqslant n_0$.

- $O(g)$ bezeichnet *Klasse von Funktionen*, deren Wachstum asymptotisch durch g begrenzt ist.
- Man sagt gleichbedeutend $f \in O(g)$, f *ist* $O(g)$ oder $f = O(g)$.
- Intuition
  - $\frac{f(n)}{g(n)}$ ist für genügend große n durch Konstante c beschränkt.
  - f wächst nicht schneller als g.

O-Notation



Beispiele zur O-Notation

- Jede positive Funktion $f(n)$ ist $O(f)$ $n_0 = 1, c = 1$
- $f(n) = 12345$ ist $O(1)$ $n_0 = 1, c = 12345$
- $f(n) = 3n - 10$ ist $O(n)$ $n_0 = 1, c = 3$
- $f(n) = 3n + 10$ ist $O(n)$ $n_0 = 10, c = 4$
- $f(n) = 2n^2$ ist $O(n^2)$ $n_0 = 1, c = 2$
- $f(n) = 3n^2 + 2n + 1$ ist $O(n^2)$ $n_0 = 3, c = 4$
- $f(n) = \sqrt{n+2}$ ist $O(\sqrt{n})$ $n_0 = 1, c = 2$
- $f(n) = \log_b n$ ist $O(\log n)$ — Basis b irrelevant, denn
$$\log_b n = \frac{1}{\log_e b} \cdot \log_e n = a \cdot \log n \in O(\log n)$$
- Es genügt zu zeigen, dass Zahlen $c > 0$ und $n_0 \geqslant 1$ existieren!
  - Wahl von c und n_0 i.d.R. *nicht eindeutig*
  - Muss nicht die kleinstmöglichen Konstanten finden
- Offensichtlich gilt aber auch z.B. $f \in O(n) \Rightarrow f \in O(n^2)$
  - Beschreibe i.d.R. die „kleinste“ Funktionsklasse !

Rechnen in Größenordnungen

- Einige Rechenregeln

  (1) Multiplikation mit einer Konstanten $a > 0$

  $$f \in O(g) \;\Rightarrow\; a \cdot f \in O(g)$$

  (2) Produkt von Funktionen $f_1 \cdot f_2$

  $$f_1 \in O(g), f_2 \in O(h) \;\Rightarrow\; f_1 \cdot f_2 \in O(g \cdot h)$$

  (3) Summe von Funktionen $f_1 + f_2$

  $$f_1 \in O(g) \wedge f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$$

- Das heißt insbesondere für Polynome

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_q n^q \in O(n^q)$$

denn $a_0 \in O(n^q) \wedge a_1 \in O(n^q) \wedge \ldots \wedge a_q n^q \in O(n^q)$

Komplexitätsklassen

$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(\sqrt{n})$	
$O(n)$	linear
$O(n \log n)$	
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^q)$ für $q \geqslant 0$	*polynomiell*
$O(a^n)$ für $a > 0$	*exponentiell*

- Ungefähre Werte für verschiedene Funktionen von n

$\log_2 n$	$\sqrt{n}$	n	$n \log_2 n$	n^2
3	3	10	30	100
6	10	100	600	10.000
9	31	1.000	9.000	1.000.000
13	100	10.000	130.000	100.000.000
16	316	100.000	1.600.000	$\geqslant 10^{10}$
19	1.000	1.000.000	19.000.000	$\geqslant 10^{12}$

Vergleich Komplexitätsklassen (2)

- Ungefähre Werte für verschiedene Funktionen von n

n	$n^{\frac{3}{2}}$	n^2	n^3	$(\frac{11}{10})^n$
10	31	100	1.000	2
100	1.000	10.000	1.000.000	13.780
1.000	31.622	1.000.000	$\geqslant 10^9$	$\geqslant 10^{41}$
10.000	1.000.000	100.000.000	$\geqslant 10^{12}$	$\geqslant 10^{413}$
100.000	31.622.776	$\geqslant 10^{10}$	$\geqslant 10^{15}$	$\geqslant 10^{4.139}$
1.000.000	$\geqslant 10^9$	$\geqslant 10^{12}$	$\geqslant 10^{18}$	$\geqslant 10^{41.393}$

Zeitaufwand anschaulich

- Zeitaufwand für 1 Schritt: $1\mu s \;=\; 10^{-6}s$ (1 Mikrosekunde)
- Gesucht: Maximale Größe n,
  für die das Problem in gegebener Zeit lösbar ist

Aufwand	1 Minute	1 Stunde	1 Tag	1 Woche	1 Jahr
n	$6 \cdot 10^7$	$3,6 \cdot 10^9$	$8,6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3,1 \cdot 10^{13}$
$n \log_2 n$	2.801.417	$1,3 \cdot 10^8$	$2,7 \cdot 10^9$	$1,7 \cdot 10^{10}$	$7,9 \cdot 10^{11}$
$n^{\frac{3}{2}}$	153.261	2.348.920	$1,9 \cdot 10^7$	$7,1 \cdot 10^7$	$9,9 \cdot 10^8$
n^2	7745	60.000	293.938	777.688	5.615.692
n^3	391	1.532	4.420	8.456	31.593
$(\frac{11}{10})^n$	187	230	264	284	326
2^n	25	31	36	39	44

Mehr zur asymptotischen Notation

- Wir betrachten hier nur die O-Notation („*Big* O *notation*")
- Es gibt weitere, verwandte Notationen
  - siehe z.B. *"relatives"* and *"distant cousins" of the Big O* [Goodrich&Tamassia]
  - *allgemein*: Landau-Notation
- Informelle Beschreibung

Notation	anschauliche Bedeutung
$f \in O(g)$	f wächst nicht wesentlich schneller als g
$f \in o(g)$	f wächst langsamer als g
$f \in \Omega(g)$	f wächst nicht wesentlich langsamer als g
$f \in \omega(g)$	f wächst schneller als g
$f \in \Theta(g)$	f wächst genauso schnell wie g

- Nicht betrachtet: Abhängigkeit von *mehreren* Parametern

Aufwandsabschätzung

- Wie bestimmt man den Aufwand eines Algorithmus?
  - Zählen von primitiven Operationen (bzw. Speicherverbrauch)
  - Bestimmen der Komplexitätsklasse
  - Ggf. für einzelne Teile, dann rechnen in O-Notation

- Im Folgenden allgemeine Regeln für
  - Schleifen und insbesondere geschachtelte Schleifen
  - Sequenzen (Hintereinanderausführung von Teilen)
  - Fallunterscheidungen
  - Rekursion

- Wir betrachten dabei i.d.R. den *schlechtesten* Fall (*worst case*)
  - Bester Fall i.d.R. ähnlich oder einfacher zu bestimmen
  - Aufwand im Mittel (*average case*) ist schwieriger: benötigt i.d.R. zusätzliche Annahmen (z.B. Verteilung der Daten)

Aufwand für Schleife

- Annahme `while ... do` α `od`
  - Aufwand für Schleifenrumpf α ist $O(f)$
  - n Iterationen
- Aufwand: $n \cdot O(f)$
- Einfachste Form: `for` Schleife

```
for (int i=0;i<n;++i)
  a[i]=0;
```

  - Aufwand für Schleifenrumpf ist konstant, also $O(1)$
  - Aufwand für Schleife ist $n \cdot O(1) \; = \; O(n)$
- Das gleiche gilt auch für die folgenden Beispiele

```
for (int i=0;i<n;++i) { a[i]=b[i]; b[i]=0; }
```

```
for (int i=100;i<n;++i) a[i]=0;
```

- (Einfach) geschachtelte Schleife (*nested loop*): Schleifenrumpf ist selbst Schleife mit $O(n)$ Iterationen
- Es gilt die gleiche Regel!
- Beispiele

```
for (int i=0;i<n;++i)
  for (int j=0;j<n;++j) { α; /* ∈O(1) */ }
```

Aufwand: $n \cdot n \cdot O(1) \; = \; O(n^2)$

```
for (int i=0;i<n;++i)
  for (int j=0;j<n;++j)
    for (int k=0;k<n;++k) { α; /* ∈O(1) */ }
```

Aufwand: $O(n^3)$

Nachgezählt: geschachtelte Schleifen

- Welchen Aufwand benötigt folgendes Code-Fragment?

```
for (int i=0;i<=n;++i)
  for (int j=0;j<i;++j) {   // j=0,...,i−1
    α; /* O(1) */
  }
```

- `for i`: n Iterationen
- `for j`: 0 (für $i=0$), 1 (für $i=1$), $\ldots$, n (für $i=n$) Iterationen
- α wird $0+1+2+\cdots+n$ mal ausgewertet
- Aufwand:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2)$$

Sequenzen

- Annahme $\alpha;\beta$
  - Aufwand für α ist $O(f)$
  - Aufwand für β ist $O(g)$
- Aufwand: $O(f+g)$
- Beispiel

```
for (int i=0;i<n;++i)    { α; /* ∈O(1) */ }

for (int i=0;i<n;++i)
  for (int j=0;j<n;++j) { β; /* ∈O(1) */ }
```

  - `for`-$\alpha \in O(n)$ und `for`-$\beta \in O(n^2)$
  - Aufwand: $O(n+n^2) \;=\; O(n^2)$
  - Denn $O(n) \subset O(n^2)$, d.h. `for`-$\alpha \in O(n^2) \wedge$ `for`-$\beta \in O(n^2)$
  - *Teuerster Teilausdruck dominiert Aufwand!*

Fallunterscheidungen

- Annahme `if ... then` α `else` β `fi`
  - Aufwand für α ist $O(f)$
  - Aufwand für β ist $O(g)$
- Aufwand: $O(f+g)$
  - Abschätzung des *maximalen* Aufwands (*worst case*)
  - $O(f+g)$ ist auch obere Schranke für „Maximum" aus f und g
- Beispiel

```
for (int i=0;i<n;++i) {
  if (a[i]>0) x+=a[i]; // α
  else {}              // β
}
```

- $\alpha \in O(1)$ und Aufwand für β ist 0
- Aufwand: $n \cdot O(1+0) \; = \; O(n)$

Beispiele aus der Linearen Algebra

- Typisches Anwendungsbeispiel für verschachtelte Schleifen
- Wir betrachten
  - Vektoren $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ mit Einträgen x_i
  - Quadratische Matrizen $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n \times n}$ mit Einträgen a_{ij} und Zeilenvektoren $\mathbf{a}_{i*}$ bzw. Spaltenvektoren $\mathbf{a}_{*j}$
- Berechne
  - Skalarprodukt (Vektor-Vektor) $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x} \bullet \mathbf{y} = \mathbf{x}^\top \mathbf{y}$
  - Matrix-Vektor-Multiplikation $\mathbf{Ax}$
  - Matrix-Matrix-Multiplikation $\mathbf{AB}$
- Konventionen für Umsetzung in Java
  - Indices starten (wie üblich) mit 0, also $(0 \leqslant i \leqslant n-1)$
  - Vektoren sind Felder der Länge n
  - Matrizen sind Felder a der Länge n^2 mit a_{ij} =`a[i+j*m]`

Beispiel: Skalarprodukt

- Berechne Skalarprodukt $s = \mathbf{x} \bullet \mathbf{y} = \sum_{i=1}^{n} x_i y_i$
- In Java

```java
double s=0.0;
for (int i=0;i<n;++i)
  s+=x[i]*y[i];
```

- Aufwand: $O(n)$

Beispiel: Matrix-Vektor Multiplikation

- Berechne $\mathbf{y} = \mathbf{A}\mathbf{x}$ mit $y_i = \mathbf{a}_{i*} \bullet \mathbf{x} = \sum_{j=1}^{n} a_{ij}\, x_j$
- In Java

```java
for (int i=0;i<n;++i) {
  double s=0.0;
  for (int j=0;j<n;++j)
    s+=a[i+j*n]*x[j];

  y[i]=s;
}
```

- Aufwand: $O(n^2)$

Beispiel: Matrix-Matrix Multiplikation

- Berechne $\mathbf{C} = \mathbf{AB}$ mit $c_{*j} = \mathbf{Ab}_{*j}$ also $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- In Java

```
for (int j=0;j<n;++j) {
  for (int i=0;i<n;++i) {
    double s=0.0;
    for (int k=0;k<n;++k)
      s+=a[i+k*n]*b[k+j*n];

    c[i+j*n]=s;
  }
}
```

- Aufwand: $O(n^3)$

Beispiel: Sortierverfahren

- Betrachte *schlechtesten* Fall!
- Selection Sort ist $O(n^2)$
- Insertion Sort ist $O(n^2)$ im *schlechtesten* Fall
- Insertion Sort ist $O(n)$ im besten Fall
- Mergesort ist $O(n \log n)$

Kurze Zusammenfassung und Übersicht

- Wir kennen jetzt die asymptotische Notation
  - Rechnen mit O-Notation
  - Regeln für Algorithmen-Bausteine
- *Im Folgenden*:
- Aufwandsabschätzung für rekursive Algorithmen
- Beispiel: Wie bestimmt man den *mittleren Aufwand*?
- Bemerkungen zu Theorie und Praxis
- Problemklassen für Algorithmen

Rekursion

- Anzahl Operationen C_n als Rekursionsformeln (*recurrences*)
- Gesucht ist jeweils eine *geschlossene* Formel
- Das erste ist i.d.R. einfach, das zweite schwer(er)
- Beispiele
  - Fakultätsfunktion $C_n = C_{n-1} + 1$
  - Fibonacci Zahlen $C_n = C_{n-1} + C_{n-2}$
  - Binäre Suche $C_n = C_{n/2} + 1$
  - Mergesort (Quicksort) $C_n = C_{n/2} + n$
  - Türme von Hanoi $C_n = C_{n-1} + 1 + C_{n-1}$

Beispiel: Fakultätsfunktion $n!$

- Gegeben: $\textbf{fac}(n) = \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n \cdot \textbf{fac}(n-1)\ \texttt{fi}$
- Aufwand: Anzahl Multiplikationen $n \cdot \textbf{fac}(n-1)$
- Sei C_n die Anzahl der Multiplikationen für $\textbf{fac}(n)$, dann gilt

$$\begin{aligned} C_0 &= 0 \qquad && \textbf{fac}(0) = 1 \\ C_n &= 1 + C_{n-1} \qquad && \textbf{fac}(n) = n \cdot \textbf{fac}(n-1) \end{aligned}$$

- Geschlossene Form ist offensichtlich $C_n = n$
- Aufwand: $O(n)$

Beispiel: Fibonacci Zahlen (1)

- Sei C_n Anzahl Additionen $\mathbf{fib}(n-2)+\mathbf{fib}(n-1)$ in

$$\begin{aligned}\mathbf{fib}(n) \quad &= \quad \texttt{if } n=0 \texttt{ then } 0 \texttt{ else}\\ &\quad\;\; \texttt{if } n=1 \texttt{ then } 1 \texttt{ else } \mathbf{fib}(n-2)+\mathbf{fib}(n-1) \texttt{ fi fi}\end{aligned}$$

- Aufwand folgt offensichtlich gleicher Gesetzmäßigkeit

$$C_n = C_{n-2} + C_{n-1} \;=\; \mathbf{fib}(n) \quad \text{mit } C_0 = 0 \text{ und } C_1 = 1$$

- Geschlossene Form: $C_n = \frac{1}{\sqrt{5}}\left(\phi^n - (1-\phi)^n\right) \approx \frac{1}{\sqrt{5}}\phi^n$
mit $\phi = \frac{1+\sqrt{5}}{2}$ (goldener Schnitt)

- exponentieller Aufwand: $O(\phi^n)$

Beispiel: Fibonacci Zahlen (2)

- Sei C_n Anzahl Additionen $y+z$ in **ifib3** mit

$$
\begin{aligned}
\textbf{ifib}(n) &= \texttt{if } n=0 \texttt{ then } 0 \texttt{ else} \\
& \texttt{if } n=1 \texttt{ then } 1 \texttt{ else } \textbf{ifib3}(n,0,1) \texttt{ fi fi} \\
\textbf{ifib3}(n,y,z) &= \texttt{if } n>2 \texttt{ then } \textbf{ifib3}(n-1,z,y+z) \\
& \texttt{else } y+z \texttt{ fi}
\end{aligned}
$$

- Es gilt offensichtlich $C_n = C_{n-1} + 1$ mit $C_0 = C_1 = 0$.
- Geschlossene Form: $C_n = n - 1$
- Aufwand: $O(n)$
- **ifib3** ist endrekursiv !

ifib3 ist *endrekursiv*

```
public static int ifib3(int n,int y,int z) {
  if (n>2)
    return ifib3(n-1,z,y+z);
  else
    return y+z;
}
```

- Aufruf `ifib(n,z,y+z)` „springt“ an Funktionsanfang
- Iterative Version: Zwischenergebnis in Variablen `y, z`

```
public static int ifib(int n) {
  if (n==0 || n==1) return n;
  int y=0, z=1;
  while (n>2) { int x=y; y=z; z+=x; --n; }
  return y+z;
}
```

Endrekursion = Iteration

Definition (Endrekursion)

Eine rekursive Funktion f ist *endrekursiv*, wenn der rekursive Funktionsaufruf die letzte Aktion zur Berechnung von f ist.

- Ergebnis des letzten Aufrufs wird *unmittelbar* zurückgegeben
- Entsprechender Aufruf $\approx$ „Sprung zum Funktionsanfang"

Endrekursion = Iteration

Eine endrekursive Beschreibung eines Algorithmus kann immer durch eine iterative Beschreibung mit einer `while` Schleife ersetzt werden, – ohne dass zusätzliche Datenstrukturen benötigt werden.

- engl. *tail recursion*, *tail call*
- *Später*: Verallgemeinerung; benötigt zusätzlich *stack*

Beispiel: Fakultätsfunktion ist endrekursiv

- Ursprüngliche Version: Multiplikation $*$ als letzte Aktion

```
public static int factorial(int n) {
  if (n==0)
    return 1;
  else
    return n*factorial(n-1);
}
```

- Transformation mit *endrekursiver* Hilfsfunktion `factorial2`

```
public static int factorial(int n) {
  return factorial2(n,1);
}
public static int factorial2(int n,int x) {
  return n==0 ? x : factorial2(n-1,x*n);
}
```

Beispiel: Binäre Suche

- Jeder rekursive Aufruf von `_find` „halbiert“ Folge
- Sei C_N Anzahl der benötigten Vergleiche für Länge $n = 2^N$
- Es gilt

$$C_N \;=\; C_{N-1} + 1 \quad \text{für } N \geqslant 1 \text{ mit } C_0 = 0$$

- Betrachte

$$\begin{aligned} C_N &= C_{N-1} + 1 \\ &= C_{N-2} + 1 + 1 \\ &= C_{N-3} + 3 \;=\; \ldots \;=\; C_0 + N \\ &= N \end{aligned}$$

- $N = \log_2 n$ Vergleiche für Länge $2^N = n$
- Aufwand: $O(\log n)$

- Es gilt

$$C_N = 2\,C_{N-1} + 2^N \quad \text{für } N \geqslant 1 \text{ mit } C_0 = 0$$

- Betrachte

$$\begin{aligned}
C_N &= 2\,C_{N-1} + 2^N \\
\frac{C_N}{\color{red}{2^N}} &= 2\,\frac{C_{N-1}}{\color{red}{2^N}} + 1 \qquad = \frac{C_{N-1}}{2^{N-1}} + 1 \\
&= \frac{2\,C_{N-2} + 2^{N-1}}{2^{N-1}} + 1 = \frac{C_{N-2}}{2^{N-2}} \color{red}{+1} + 1 \\
&= \frac{C_{N-3}}{2^{N-3}} + 3 = \ldots \\
&= N
\end{aligned}$$

- D.h. es gilt $C_N = N \cdot 2^N$
- Wir setzen ein $n = 2^N \Leftrightarrow N = \log_2 n$ und erhalten

$$\boxed{C(n) = n \log_2 n}$$

- Sei C_n die Anzahl der Aufrufe von `move_disk()`.
- Es gilt
$$C_n = 2\,C_{n-1} + 1 \text{ für } n \geqslant 1 \text{ mit } C_0 = 0$$
- Ansatz: Wir betrachten $D_n := C_n + 1$

$$\begin{aligned} D_0 &= C_0 + 1 = 1 \\ D_n &= C_n + 1 = 2C_{n-1} + 2 = 2\,(C_{n-1} + 1) = 2\,D_{n-1} \end{aligned}$$

- Offensichtlich gilt $D_n = 2^n$ und damit $C_n = D_n - 1 = 2^n - 1$
- Aufwand: $O(2^n)$

Aufwandsabschätzung bei Rekursion

1. Finde Rekursionsformel für Anzahl Operationen C_n
2. Finde geschlossene Form von C_n
3. Finde Funktionsklasse f mit $C_n \in O(f)$

- Oft bekannte Rekursionsformeln (Formelsammlung!)
- Weiterführende Literatur s. z.B. [Graham, Knuth & Patashnik]
  - Sammlung von Ansätzen/Techniken oder Intuition
  - Einstieg per Faustregel: *Look at small cases first.*
  - Probe: vollständige Induktion
- *Master-Theorem* – s. z.B. [Cormen, Leiserson & Rivest]
  - Bestimmt asymptotisches Verhalten
  - Fertiger Lösungsansatz „zum Einsetzen“

Mittlerer statt *worst case* Aufwand

- Mitteln des Aufwands $f_i(n)$ über alle möglichen Fälle i
- *Gewichtetes* Mittel:

$$\frac{1}{\sum_{i=1}^{N} P_i} \cdot \sum_{i=1}^{N} P_i \cdot f_i(n)$$

  Mit welcher Wahrscheinlichkeit P_i tritt Fall i auf?

- Oft vereinfachende Annahme: *Gleichverteilung*

$$P_i = \frac{1}{N} = \text{const}$$

Mittlerer statt *worst case* Aufwand: Beispiel

- Gegeben: Folge von ganzen Zahlen `int[] a`
- Gesucht: Index `imax` der (ersten) größten Zahl

```
int imax=0;
for (int i=1;i<a.length;++i)
  if (a[imax]<a[i]) imax=i;
```

- Aufwand: Anzahl Zuweisungen `imax=i;`

Beispiel: Mittlerer Aufwand

```
int imax=0;                    // i=0
for (int i=1;i<a.length;++i)
  if (a[imax]<a[i]) imax=i;
```

- `imax=i;` wird ausgeführt gdw. $a_i = \max\{a_0, \ldots, a_i\}$
- Annahme: *Gleichverteilung*

$$P_i \;=\; P(\,a_i = \max\{a_0, \ldots, a_i\}\,) \;=\; \frac{1}{i+1}$$

- Aufwand für `if` Anweisung im i-ten Durchgang

$$\frac{1}{i+1} \cdot 1$$

- Betrachte `for` Schleife: addiere Aufwand für $i = 0, \ldots, n-1$

$$f(n) \;=\; 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i+1} + \cdots + \frac{1}{n-1} + \frac{1}{n}$$

Beispiel: Mittlerer Aufwand

- Für den Aufwand gilt demnach

$$f(n) \;=\; 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \;=\; \sum_{i=1}^{n} \frac{1}{i}$$

- Harmonische Reihe

$$f(n) \;=\; H_n \quad n\text{-te } \textit{harmonische Zahl}$$

- Asymptotische Entwicklung von H_n ist bekannt

$$H_n \;=\; \gamma + \ln n + O\left(\frac{1}{n}\right) \;\approx\; \gamma + \ln n \quad \text{für } n \to \infty$$

$\gamma = 0,5772156649015\ldots$ ist die *Eulersche Konstante*

- Damit gilt $f \in O(\log n)$.

Asymptotisches Verhalten und Praxis

- O-Notation gibt asymptotisches Laufzeitverhalten an!
  - In der Praxis ist Problemgröße n endlich und oft klein!
- Konstanten (z.B. n_0, c) nie völlig aus den Augen verlieren!
- Für kleine Datensätze in der *Praxis*
  - z.B. Sequentielle Suche ggf. schneller als binäre Suche
  - z.B. Insertion Sort schneller als Quicksort
- Solche Aussagen sind auch abhängig von
  - Programmierumgebung (Sprache, virtuelle Maschine,...)
  - Verwendeter Hardware (v.a. Speicherzugriffe über *caches*,...)
- *Im Zweifelsfall testen! – D.h. nachmessen!*
- Beispiel: Matrix-Matrix-Multiplikation
  - Matrix Multiplikation benötigt $O(n^3)$ reelle Multiplikationen
  - Strassen-Algorithmus benötigt $O(n^{2,807})$
  - Coppersmith-Winograd Algorithmus benötigt $O(n^{2,376})$
  - Lohnt nur für (sehr) große Matrizen.

Problemklassen

- „Mindestaufwand“ für Probleme, z.B.
  - $O(\log n)$ für Suche in sortierter Folge
  - $O(n)$ für sequentielle Suche
  - $O(n \log n)$ für das Sortieren einer Folge
  - $O(n^k)$ Matrix-Vektor, Matrix-Matrix Multiplikation
  - *Weitere Algorithmen im nächsten Semester*
- Alle diese Beispiele sind effizient lösbare Probleme, d.h. sie benötigen höchstens polynomialen Aufwand.
  - Effizient lösbar = praktisch lösbar
- *Nicht effizient* lösbar z.B. Problem der Türme von Hanoi, da *exponentieller* Aufwand nötig
- Komplexitätsklasse **P** enthält alle Probleme, die mit Hilfe von *deterministischen Algorithmen* mit **polynomialen** Aufwand gelöst werden können.
  - Klasse der „praktisch lösbaren“ Probleme

Problemklassen **P** und **NP**

- **P** = praktisch lösbare Probleme
- Komplexitätsklasse **NP** enthält alle Probleme, die nur mit Hilfe von nichtdeterministischen Algorithmen mit polynomialen Aufwand gelöst werden können.
- Nichtdeterministisch = „Erraten“ der richtigen Lösung
  - „Simulation“ mit deterministischen Algorithmen führt zu exponentiellen Aufwand
  - *Überprüfen* der „erratenen“ Lösung mit polynomialen Aufwand: „praktisch überprüfbar“
- Es gilt offensichtlich **P** $\subseteq$ **NP**
- Offene Frage: Gilt **P** = **NP** ?
- Mehr dazu:
  Vorlesung *Grundlagen der Theoretischen Informatik*

Kurze Zusammenfassung

- Abstraktion für Aufwand von Algorithmen
  - Asymptotische Betrachtung O-Notation
  - „Rechenregeln“ und Regeln für Abschätzung
  - Grundsätzliche Aussage über Effizienz von Algorithmen
  - Abstraktion kann auch tückisch sein!
- Rekursion
  - Geschlossene Darstellung von Rekursionsformeln
  - Endrekursion = Iteration
- Klassifikation von Problemen und Problemlösungen
  - Grenzen der praktisch entscheidbaren Probleme
  - Theoretische Fragestellungen
- Literatur
  - [Saake&Sattler] (Kapitel 7.3), [Goodrich&Tamassia] (Kap. 2)
  - Weiterführend [Sedgewick] (einige Rekursionsformeln), [Cormen, Leiserson & Rivest] , [Graham, Knuth & Patashnik]

Korrektheit und Komplexität – Ausblick

- Korrektheit als essentielle Eigenschaft von Algorithmen
  - Korrektheit ist *relativ*
  - Verifikation – Vor/Nachbedingungen, Invarianten, Induktion
  - Validierung
  - Partielle und totale Korrektheit
- Aufwand als qualitative Eigenschaft
  - Asymptotische Betrachtung

- *Als nächstes*: Abstrakte Datentypen
  - Spezifikation von Schnittstellen
  - Spezifikation von Semantik
  - ggf. Spezifikation des Aufwands