



universität
wien

Fakultät für Informatik

Vorlesung

Algorithmen und Datenstrukturen

3 Stunden / 4 ECTS Punkte

Univ.-Prof. Dipl.-Inf. Dr. **Monika Henzinger**
Forschungsgruppe Theorie und Anwendung von Algorithmen

Univ.-Prof. Dipl.-Ing. Dr. **Erich Schikuta**
Dipl.-Ing. **Helmut Wanek**
Forschungsgruppe Workflow Systems and Technology

SS 2016

1. Algorithmen

Paradigmen, Analyse

2. Datenstrukturen

Motivation, Überblick

3. Vektoren

Hashing, Sortieren

4. Listen

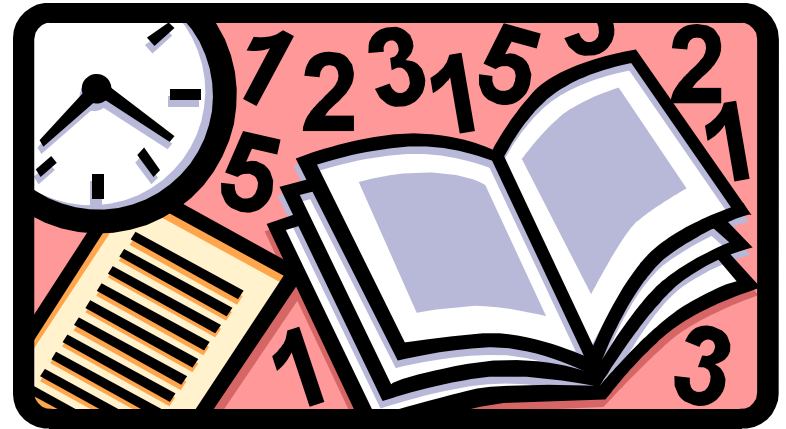
Lineare Speicherstrukturen, Stack, Queue

5. Bäume

Suchstrukturen

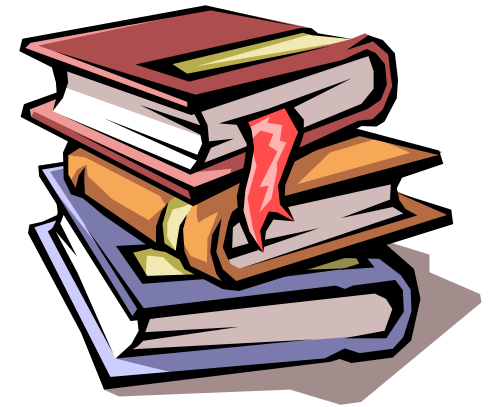
6. Graphen

Traversierungs- und Optimierungsalgorithmen



R. Sedgewick, *Algorithmen in C++* (Teil 1-4), Addison Wesley, 3. überarbeitete Auflage, 2002

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein, *Introduction to Algorithms*, published by MIT Press, 2009



Für Mitarbeit und Durchsicht der Folien geht mein besonderer Dank an Helmut Wanek und Martin Polaschek

Mein weiterer Dank geht an zahlreiche Studierende der letzten Jahre, die im Rahmen ihrer Übungen die Basis für einige der dynamischen Beispiele der VO lieferten.

Kapitel 1

Algorithmen

Algorithmen

Verfahrensvorschriften, Anweisungsfolgen, Vorgangsmodellierungen,
beschriebene Lösungswege

Zwei Ziele

1. Algorithmen zu Problemstellungen finden!

Lösungsansätze finden, „konstruieren“

2. „Gute“ Algorithmen finden!

“bessere” Algorithmen

schneller, vollständiger, korrekter, ...

“leistungsfähigere” Datenstrukturen

kompakter, effizienter, flexibler, ...

Generell: Ersparnis an Rechenzeit und/oder Speicherplatz

Aufgabe: “Summe der ganzen Zahlen bis n”

Straight-forward solution: “Aufsummieren der einzelnen Werte zwischen 1 und n”

$$summe \leftarrow \sum_{i=1}^n i$$

Realisierung (C/C++ Programm)

```
int sum(int n) {  
    int i, summe = 0;  
    for(i=1; i<=n; i++)  
        summe += i;  
    return summe;  
}
```

Vergleiche mit anderem Programmieransatz, z.B. while- statt for-Schleife!
⇒ alternativer Programmierstil

Zwei Alternativen

1. Alternativer Programmierstil

Problemlösungsansatz beibehalten, aber programmiertechnische Umsetzung überarbeiten

Beispiel:

Schleifenform (siehe oben)

Rekursion statt Iteration

```
int sum(int n) {  
    if(n <= 0) return 0;  
    if(n == 1) return 1;  
    else  
        return n+sum(n-1);  
}
```


2. Alternativer Lösungsweg

Wahl eines anderen Problemlösungsweges, z.B.

Gauß'sche Summenformel

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

Umsetzung

```
int sum(int n) {  
    return (n*(n+1)) / 2;  
}
```

Ableitung der Gauß'schen Summenformel

1	...	n
n	...	1
n+1	...	n+1

} = 2 * Summe von 1 bis n

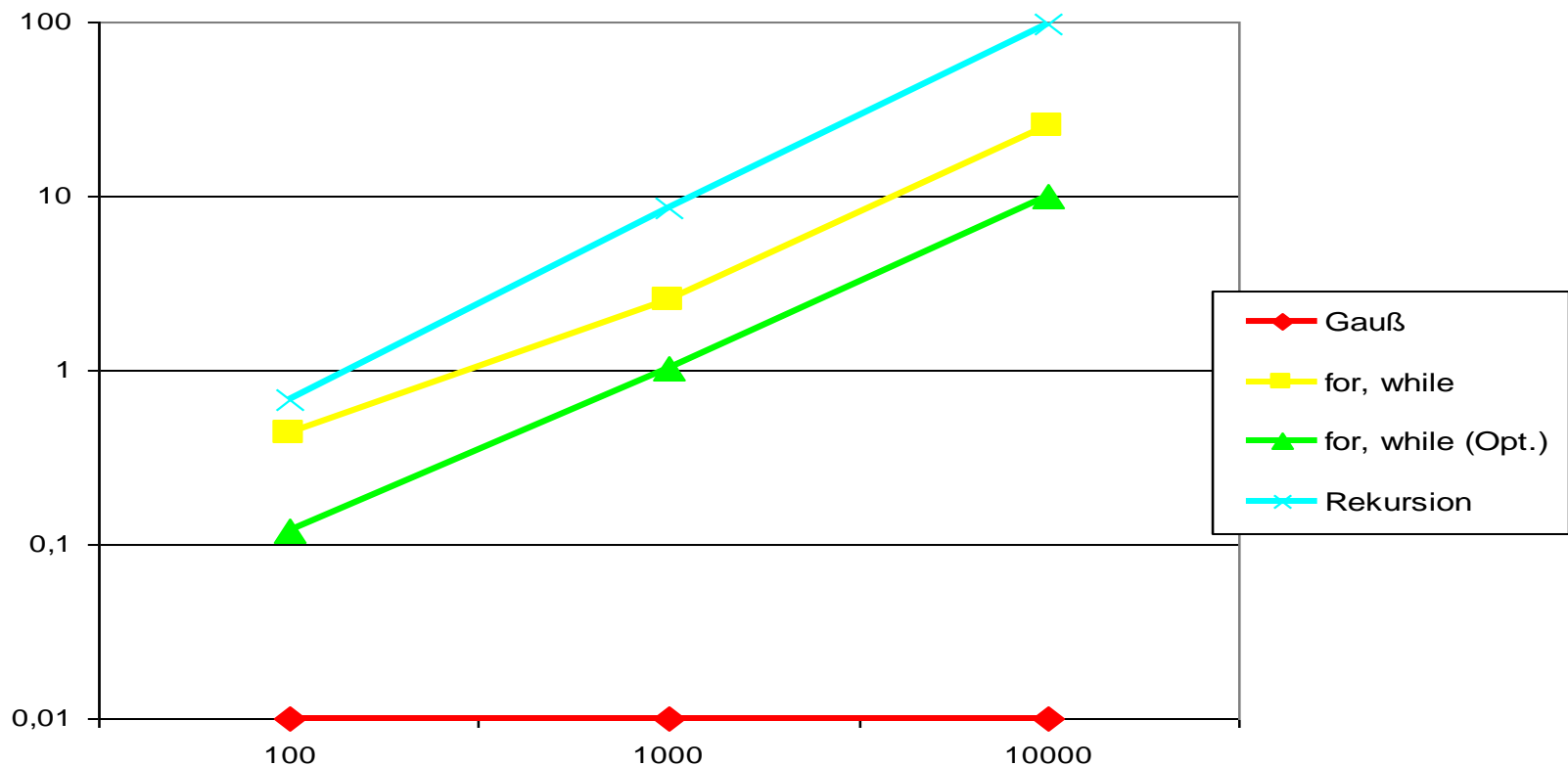
{

n * (n+1)

Daraus folgt die
Summen-Formel
n * (n+1) / 2

Vergleich der Laufzeiten

Summenberechnung, 100000 Wiederholungen, CPU: 200 MHz Pentium



Problem: Laufzeitenvergleich führt nur zu punktueller
Qualitätsbestimmung

Laufzeit, Speicherplatzverbrauch

Abhängig von

Computer

Betriebssystem

Compiler, ...

Ziel: Methodik für generellen Qualitätsvergleich zwischen
Algorithmen

Unabhängig von äußeren Einflüssen

Generelle Techniken zur Lösung großer Klassen von Problemstellungen

Greedy algorithms

“gefräßiger, gieriger” Ansatz, Wahl des lokalen Optimums

Divide-and-conquer algorithms

schrittweises Zerlegen des Problems der Größe n in kleinere Teilprobleme

Dynamic programming

dynamischer sukzessiver Aufbau der Lösung aus schon berechneten Teillösungen

In jedem Schritt eines *Greedy* (*gierigen*) Algorithmus wird die Möglichkeit gewählt, die *unmittelbar* (lokal) den *optimalen* (kleinsten bzw. größten) *Wert* bezüglich der *Zielfunktion* liefert. Dabei wird die globale Sicht auf das Endziel vernachlässigt.

Vorteil:

Effizienter Problemlösungsweg, oft sehr schnell, kann in vielen Fällen relativ gute Lösung finden

Nachteil:

Findet oft keine optimale Lösung

Beispiel: Münzwechselfmaschine

“Wechsle den Betrag von 18.- in eine möglichst kleine Anzahl von Münzen der Größe 10.-, 5.- und 1.-”



Greedy Ansatz:

wähle größte Münze
kleiner als Betrag
gib die Münze aus
subtrahiere ihren
Wert vom Betrag
wiederhole solange
bis Differenz gleich 0

d.h.:

18.-	-	10.-	=	8.-	10
8.-	-	5.-	=	3.-	5
3.-	-	1.-	=	2.-	1
2.-	-	1.-	=	1.-	1
1.-	-	1.-	=	0	1

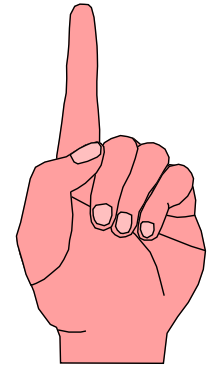
Lösung für diese Problemstellung nicht nur “gut” sondern sogar optimal!

DOCH

Problem bei kleiner Änderung der Problemstellung:

5.- → 6.-

Münzwerte 10.-, **6.-**, 1.-



greedy Ansatz liefert

$$18.- - 10.- = 8.- \quad \textcircled{10}$$

$$8.- - 6.- = 2.- \quad \textcircled{6}$$

$$2.- - 1.- = 1.- \quad \textcircled{1}$$

$$1.- - 1.- = 0 \quad \textcircled{1}$$

⇒ 4 Münzen

optimal wäre aber 3 x 6.-

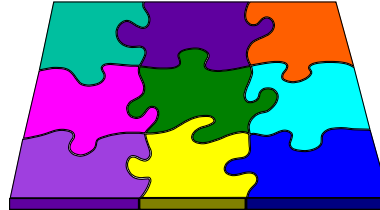
$$18.- - 6.- = 12.- \quad \textcircled{6}$$

$$12.- - 6.- = 6.- \quad \textcircled{6}$$

$$6.- - 6.- = 0 \quad \textcircled{6}$$

⇒ 3 Münzen

Bei *Divide-and-Conquer* (*Teilen-und-Herrschen*) Algorithmen wird ausgehend von einer generellen Abstraktion das Problem iterativ verfeinert, bis Lösungen für vereinfachte Teilprobleme gefunden wurden, aus welchen eine Gesamtlösung konstruiert werden kann.



Diese Vorgangsweise wird auch oft mit “stepwise refinement” oder “top-down approach” bezeichnet

Verschiedene Ansätze

Problem size division

Zerlegung eines Problems der Größe n in eine endliche Anzahl von Teilproblemen kleiner n

Step division

Aufteilen einer Aufgabe in eine Sequenz (Folge) von individuellen Teilaufgaben

Case division

Identifikation von Spezialfällen zu einem generellen Problem ab einer gewissen Abstraktionsstufe

...

Durch Zerlegung Verringerung der Problemgröße, d.h.

$$P(n) \Rightarrow k \cdot P(m),$$

wobei $k, m < n$

Binäre Suche

Suche eine Zahl x in der (aufsteigend) sortierten Folge z_1, z_2, \dots, z_n (allgemein: z_l, z_{l+1}, \dots, z_r mit $l=1, r=n$) und ermittle ihre Position i

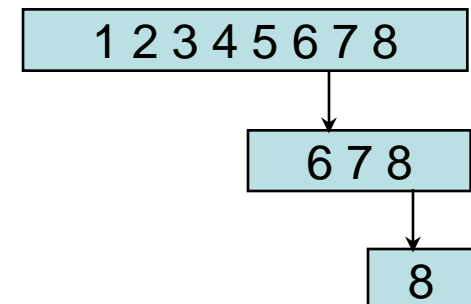
Zerlegung: $k = 1$ und $m \approx n/2$

Trivial: finde mittleren Index $m = (l+r) \div 2$

Falls $z_m = x$ Ergebnis m , sonst

Falls $x < z_m$ suche x im Bereich z_l, z_2, \dots, z_{m-1}
sonst suche x im Bereich $z_{m+1}, z_{m+2}, \dots, z_r$

Suche Zahl 8



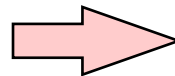
Idee: Straßenkehrer-Philisophie:

“Atemzug - Besenstrich - Schritt”

Beispiel

Gehaltserhöhung

Bestimme Mitarbeiter
Finde eindeutige Identifikation
Suche im Datenbestand
Stelle aktuelles Gehalt fest
Ändere auf neues Gehalt
Speichere Information
Vermerke Änderungsvorgang



Speichere Information

Lösche alten Datensatz
Füge neuen Datensatz ein

Ansatz: Identifikation von Fallunterscheidungen im Problemdatenbereich

Beispiel

Berechnung der Lösungen zu einer quadratischen Gleichung

$$az^2 + bz + c = 0$$

$$z_{1,2} = \frac{-b \pm \sqrt{q}}{2a}, \quad q = b^2 - 4ac$$

if $q > 0$, 2 reelle Wurzeln

$q = 0$, reelle Doppellösung

$q < 0$, Paar komplexer Wurzeln

Problem

Oft ist eine Teilung des Originalproblems in eine 'kleine' Anzahl von Teilproblemen nicht möglich, sondern führt zu einem exponentiellen Algorithmus.

Man weiß aber, es gibt aber nur eine polynomiale Zahl von Teilproblemen.

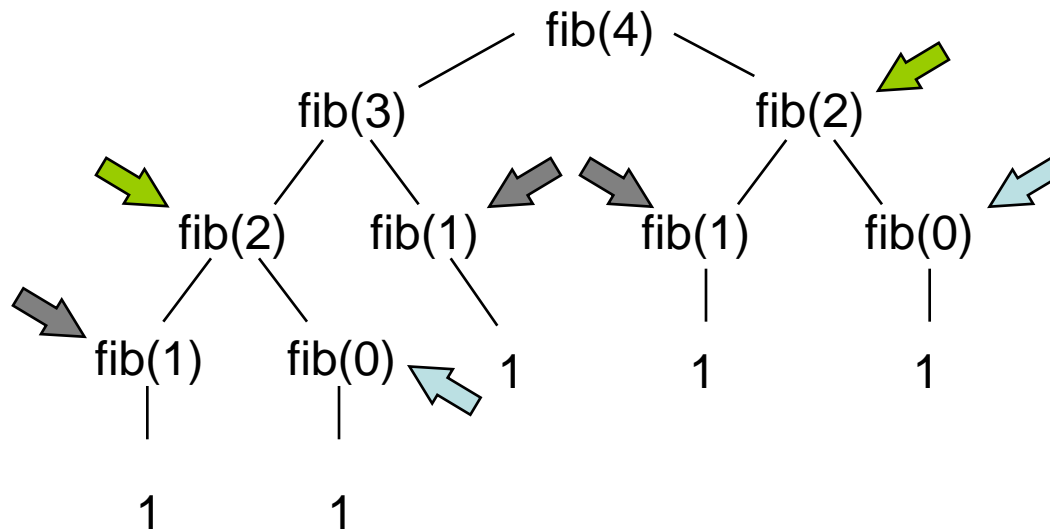
Idee *Dynamic Programming* (*Dynamisches Programmieren*)

nicht Start von Problemgröße n und Aufteilung bis Größe 1,
SONDERN

Start mit Lösung für Problemgröße 1, Kombination der berechneten Teillösungen bis eine Lösung für Problemgröße n erreicht wurde.

Beispiel: Berechnung der Fibonacci Zahlen

```
int fib(int n) {  
    if(n <= 1) return 1;  
    return fib(n-1) + fib(n-2)  
}
```




→	fib(2)	2x berechnet
→	fib(1)	3x berechnet
→	fib(0)	2x berechnet

Problem: Wiederholte Lösung eines Teilproblems

Lösungsweg

Beginn mit Berechnung für `fib(0)`, Anlegen einer Tabelle aller berechneten Werte und Konstruktion der neuen Werte aus den berechneten Tabelleneinträgen.



<code>fib(0)</code>	<code>fib(1)</code>	<code>fib(2)</code>	...	<code>fib(n)</code>
1	1	2	...	<code>fib(n-2)+fib(n-1)</code>

Werte werden aus der
Tabelle entnommen

```
int fib(int n) {  
    int F[MAXSIZE];  
    F[0] = 1; F[1] = 1;  
    for(i = 2; i <= n; i++) {  
        F[i] = F[i-1] + F[i-2];  
    }  
    return s;  
}
```

Beachte: Verbesserung der Laufzeit ABER zusätzlicher Speicherplatzbedarf

Ziel ist objektive Bewertung von Algorithmen

Kriterien

Effektivität

Ist das Problem lösbar, ist der Ansatz umsetzbar in ein Programm?

Korrektheit

Macht der Algorithmus was er soll?

Termination

Hält der Algorithmus an, besitzt er eine endliche Ausführungszeit?

Komplexität

Wie schnell ist der Algorithmus \Rightarrow Laufzeitkomplexität?

Wie strukturiert ist der Algorithmus \Rightarrow Strukturkomplexität?

Wie viel Speicherplatz braucht er Algorithmus \Rightarrow
Speicherplatzkomplexität?

Prinzip

Algorithmus kann als lauffähiges Computerprogramm formuliert werden.

effective \Rightarrow it does work

Problem

Formulierung

Transformation der Problembeschreibung in einen exekutierbaren Algorithmus

Frage: Gibt es überhaupt eine ausführbare Lösung zum gegebenen Problem?



Produziert der Algorithmus das gewünschte Ergebnis?

Zwei Vorgangsweisen möglich

Testen

Verifikation

Testen

Vollständiges Austesten meist nicht möglich

Statistischer Ansatz meist verfolgt, z.B. Pfadüberdeckung (Strukturelle Komplexität)

Bestenfalls „Falsifizierung“ erreichbar

Es können nur Fehler gefunden werden, aber es kann keine Korrektheit bewiesen werden

Mathematisch orientierte Verifikationstechniken erlauben es, die Korrektheit von Programmstücken in Abhängigkeit von Bedingungen an die Eingabedaten (Prämissen) zu beweisen

zwingt den Entwickler Entwurfsentscheidungen noch einmal nachzuvollziehen und hilft beim Auffinden logischer Fehler

Algorithmus muss verstanden werden

je größer ein Programmsystem, umso schwieriger die Verifikation (in der Praxis kaum von Bedeutung)

Beweisansatz abhängig vom Problem

Vollständige Induktion

Beispiel: Gauß'sche Summenformel

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

Ind. Anfang

$$\text{für } n = 1 \rightarrow 1 \cdot 2 / 2 = 1 \quad \checkmark$$

Ind. Voraussetzung

$$(1+2+\dots+n-1) = (n-1) \cdot n / 2$$

Ind. Schluss

$$\begin{aligned} (1+2+\dots+n-1)+n &= (n-1) \cdot n / 2 + n = \\ &= ((n-1) \cdot n + 2n) / 2 = (n^2 - n + 2n) / 2 = \\ &= n \cdot (n+1) / 2 \quad \checkmark \end{aligned}$$

verschiedene Ansätze

McCarthy, Naur, Floyd, Hoare, Knuth, Dijkstra, etc.

“Rückwärtsbeweis”:

Strukturierter Ansatz, beruht auf *Prädikamentransformation*

Überprüfung, ob die Voraussetzungen (*weakest preconditions, wp*) an die Eingabedaten garantieren, dass das Programm in einem Zustand terminiert, der das gewünschte Programmziel erfüllt.

Man bezeichnet diese Programmziel, welches die Aufgabe des Programms beschreibt, als *Korrektheitsbedingung, KB*

Hierzu darf das Programm nur aus einfachen Zuweisungen, Vergleichen, while-Form Schleifen und Anweisungsfolgen bestehen. Alle anderen Konstrukte müssen übersetzt werden.

Beispiel

```
int sum(int n) {
    int s = 0;
    int i = 1;
    while(i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

Rückwärtsbeweis

Korrektheitsbedingung KB: $s = \sum_{i=1}^n i$

(I) Schleifeninvariante SI

$$SI: (s = \sum_{j=1}^{i-1} j) \wedge (i \leq n+1)$$

(1) $SI \wedge \neg Bed \Rightarrow KB$

$$s = (\sum_{j=1}^{i-1} j) \wedge (i \leq n+1) \wedge (i > n) \Rightarrow s = (\sum_{j=1}^{i-1} j) \wedge (i = n+1) \Rightarrow s = \sum_{j=1}^n j \Rightarrow KB$$

(2) $SI \wedge Bed \Rightarrow wp(\text{Schleifenblock} \mid SI)$

$$wp(s = s + i \mid wp(i = i + 1 \mid (s = \sum_{j=1}^{i-1} j) \wedge (i \leq n+1))) \Rightarrow wp(s = s + i \mid (s = \sum_{j=1}^i j) \wedge$$

$$\wedge (i+1 \leq n+1)) \Rightarrow (s + i = \sum_{j=1}^i j) \wedge (i \leq n) \Rightarrow (s = \sum_{j=1}^{i-1} j) \wedge (i \leq n) \equiv SI \wedge Bed$$

(II) Rest des Programms

$$wp(s = 0 \mid wp(i = 1 \mid SI)) \Rightarrow wp(s = 0 \mid (s = \sum_{j=1}^{1-1} j) \wedge (1 \leq n+1)) \Rightarrow$$

$$\Rightarrow (0 = \sum_{j=1}^0 j) \wedge (0 \leq n) \Rightarrow n \geq 0$$

Hält der Algorithmus an, d.h. besitzt er eine endliche Ausführungszeit?

Falls nicht klar, manchmal folgende Technik einsetzbar:

Man finde die bestimmende Größe oder Eigenschaft des Algorithmus der die folgenden 3 Charakteristiken erfüllt:

- Eine 1-1 Abbildung dieser Größe auf die ganzen Zahlen kann aufgestellt werden.

- Diese Größe ist positiv.

- Die Größe nimmt während der Ausführung des Algorithmus kontinuierlich ab (dekrementiert).

Idee:

Die gefundene Größe besitzt bei Algorithmusbeginn einen vorgegebenen positiven Startwert, der sich kontinuierlich verringert. Da die Größe nie negativ werden kann, folgt, dass der Algorithmus terminieren muss, bevor die Größe kleiner 0 ist.

Beispiel:

```
int sum(int n) {  
    if(n <= 0) return 0;  
    if(n == 1) return 1;  
    else  
        return n+sum(n-1);  
}
```

Größe n

Termination
bevor $n < 0$

Dekrement

Die *Laufzeitkomplexität* liefert Aussagen über das Laufzeitverhalten von Algorithmen in Abhängigkeit von der Problemgröße

Ziel

Algorithmen zu vergleichen

Ansatz

Messen der Ausführungszeit der einzelnen Anweisungen

Bestimmen, wie oft jede Anweisung beim Programmablauf ausgeführt wird

Summe berechnen

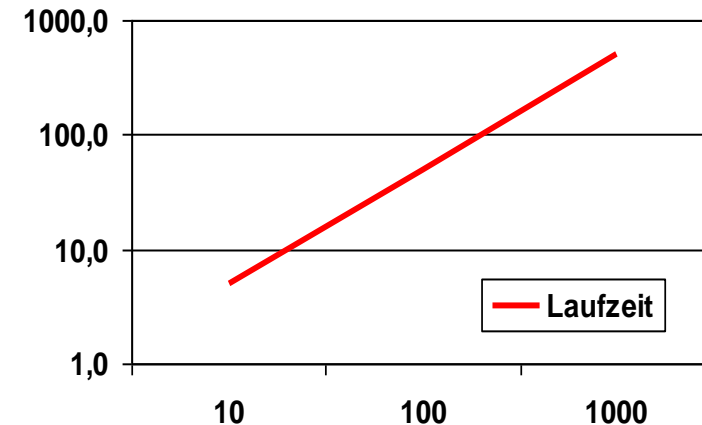
Problem

Ausführungszeiten abhängig von Maschinen- bzw. Systemarchitektur, Übersetzungsqualität des Compilers, etc.



<code>int sum(int n)</code>	Zeit in msec
<code>int s = 0;</code>	$T_1 \Rightarrow 0.1$
<code>int i = 1;</code>	$T_2 \Rightarrow 0.1$
<code>while(i <= n) {</code>	$T_3 \Rightarrow 0.3$
<code> s = s + i;</code>	$T_4 \Rightarrow 0.1$
<code> i = i + 1;}</code>	$T_5 \Rightarrow 0.1$
<code>return s;</code>	$T_6 \Rightarrow 0.1$
<code>}</code>	

Gemessene Zeiten im Programm



Berechnung der Laufzeit

$$f_{\text{sum}}(n) = T_1 + T_2 + n \cdot (T_3 + T_4 + T_5) + T_6$$

$$f_{\text{sum}}(10) = 0.1 + 0.1 + 10 \cdot (0.3 + 0.1 + 0.1) + 0.3 + 0.1 = 5.6$$

$$f_{\text{sum}}(100) = 0.1 + 0.1 + 100 \cdot (0.3 + 0.1 + 0.1) + 0.3 + 0.1 = 50.6$$

$$f_{\text{sum}}(1000) = 0.1 + 0.1 + 1000 \cdot (0.3 + 0.1 + 0.1) + 0.3 + 0.1 = 500.6$$

$$f_{\text{sum}}(n) = 0.6 + n \cdot (0.5)$$

Entspricht einem Ansatz des „Ausprobierens“

Nur punktuell möglich

bestimmte Problemgröße, Datenverteilung

Sonderfälle problematisch

Systemabhängig

Hardware, Prozessor, ...

Betriebssysteme, Compiler, Bibliotheken, ...

Lastabhängig

Frage schwer zu beantworten, ob „graduelle“ oder
„grundsätzliche“ Verbesserung erreichbar

Grundprinzip

Die exakten Werte der Ausführungszeiten sind uninteressant!

Über die *Ordnungsnotation* möchte man Aussagen treffen (siehe Ziel), dass Algorithmus A grob gesehen gleich schnell wie Algorithmus B ist.

Durch Beschreiben der Laufzeiten von A und B über Funktionen $f(n)$ und $g(n)$ wird diese Fragestellung auf Vergleich dieser Funktionen zurückgeführt.

Man betrachtet das *asymptotische Wachstum* der Ausführungszeiten der Algorithmen bei wachsender Problemgröße n .

d.h., wenn die Problemgröße gegen Unendlich geht

Big-O-Notation: Eine Funktion $f(n)$ heißt von der Ordnung $O(g(n))$, wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \leq c_0 g(n)$ für alle $n > n_0$.

liefert eine Obergrenze für die Wachstumsrate von Funktionen $f \in O(g)$, wenn f höchstens so schnell wie g wächst.

z.B.: $17n^2 \in O(n^2)$, $17n^2 \in O(2^n)$

Big- Ω -Notation: Eine Funktion $f(n)$ heißt von der Ordnung $\Omega(g(n))$, wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \geq c_0 g(n)$ für alle $n > n_0$.

liefert eine Untergrenze für die Wachstumsrate von Funktionen $f \in O(g)$, wenn f mindestens so schnell wie g wächst.

z.B.: $17n^2 \in \Omega(n^2)$, $2^n \in \Omega(n^2)$, $n^{37} \in \Omega(n^2)$

Θ -Notation: Das Laufzeitverhalten ist $\Theta(g(n))$ falls gilt:
 $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ (beschreibt das Laufzeitverhalten exakt)

Laufzeitschätzung: $f_{\text{sum}}(n) = T_1 + T_2 + n \cdot (T_3 + T_4 + T_5) + T_3 + T_6$

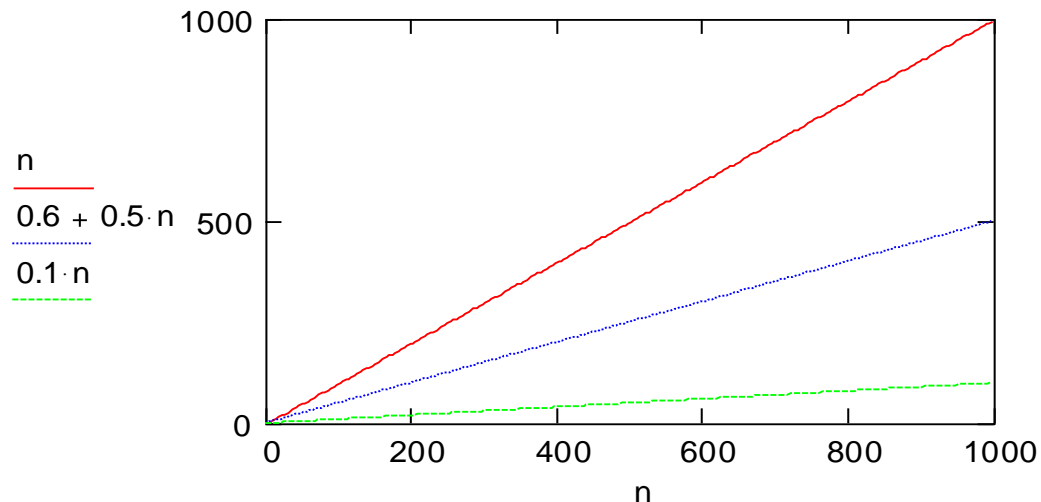
Messung: $T_1=0.1, T_2=0.1, T_3=0.3, T_4=0.1, T_5=0.1, T_6=0.1$

ergibt $f_{\text{sum}}(n) = 0.6 + n \cdot (0.5)$

$g(n)=n \Rightarrow$ Untergrenze: $0.1 \cdot g(n)$ Obergrenze: $h(n)=1 \cdot g(n)$

reale Werte, in der
Implementierung
gemessen

eine von vielen
möglichen
Grenzen



daraus folgt bezüglich der Ordnung des Algorithmus

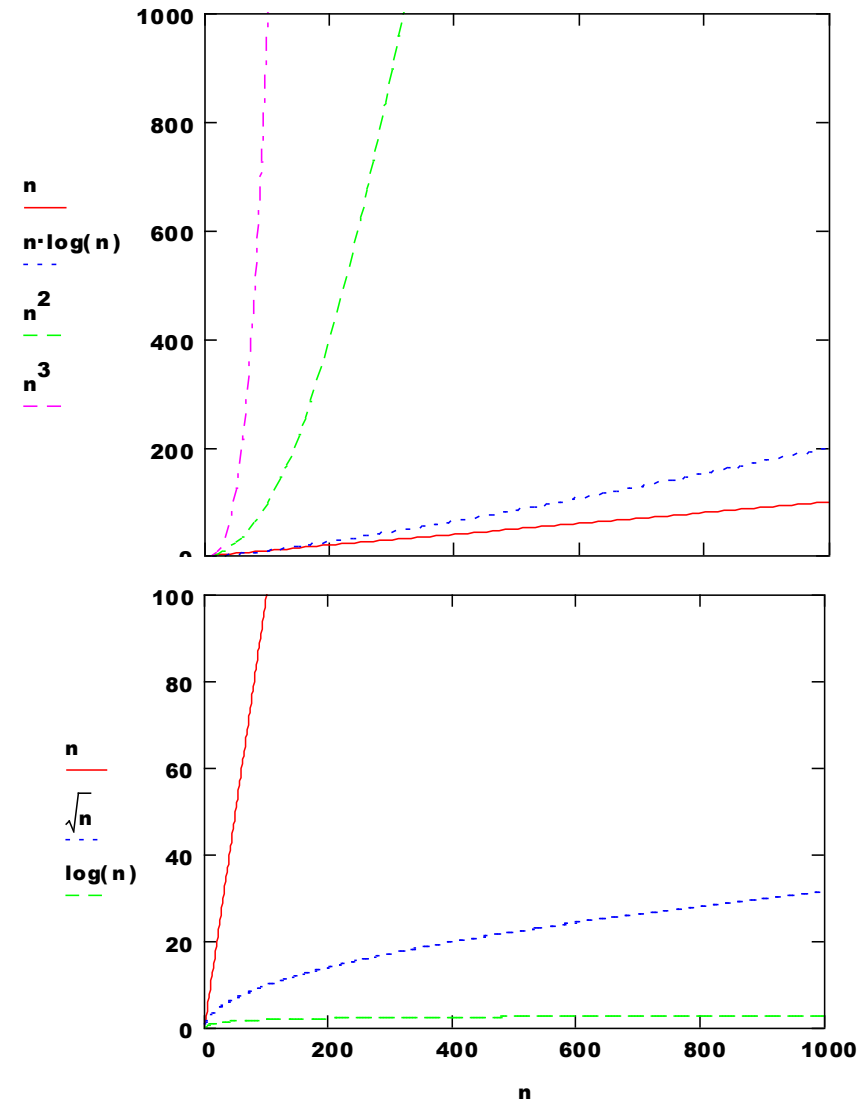
$f_{\text{sum}}(n) \in O(n)$ und weiters $f_{\text{sum}}(n) \in \Omega(n)$, d.h. $f_{\text{sum}}(n) \in \Theta(n)$.

Die daraus für unser Beispiel ableitbare Aussage lautet, dass die Laufzeit des Algorithmus direkt proportional zur Problemgröße n ist

Laufzeitenvergleich

Annahme vorgegebene
Problemgröße und
unterschiedliches
Laufzeitverhalten

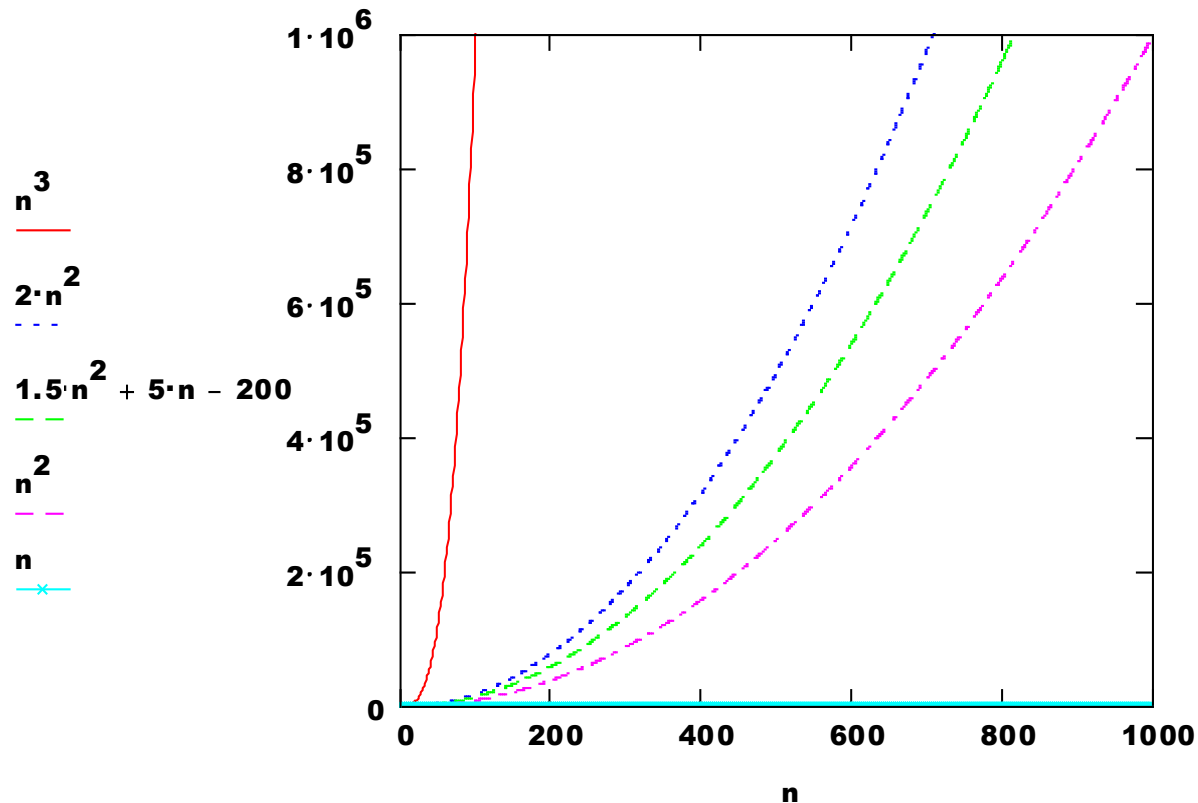
Ordnung	Laufzeit
$\log n$	1.2×10^{-5} sec
\sqrt{n}	3.2×10^{-4} sec
n	0.1 sec
$n \log n$	1.2 sec
$n \sqrt{n}$	6.5 sec
n^2	2.8 h
n^3	31.7 a
2^n	über 1 Jahrhundert



Beschreibung des Laufzeitverhaltens

durch obere $O(n)$ und untere Schranke $\Omega(n)$

z.B.: $T(n) = 1.5n^2 + 5n - 200 \Rightarrow O(n^2)$, da $n^2 \leq T(n) \leq 2n^2$



Prinzipien der mathematischen Analyse

Definition der Problemgröße n

Finden eines geeigneten Problemparameters der die Problem-größe beschreibt. Dieser charakterisiert die Belastung ($= f(n)$)

worst-case versus *average-case*

worst-case: Verhalten im schlechtesten Fall, maximal zu erwartender Aufwand

average-case: \emptyset -Verhalten, Erwartungswert des Aufwandes

Laufzeitanalyse über $O(n)$ und $\Omega(n)$

Ignoriere konstante Faktoren

Merke nur höchste Potenzen

oft schwierig zu bestimmen

Ignoriere konstante Faktoren

Konstante Werte werden auf den Faktor 1 reduziert, d.h.

$$T(n) = 13 * n \Rightarrow O(n)$$

$$T(n) = \log_2(n) \Rightarrow O(\log(n)), \text{ da } \log_x(n) = \log_a(n) / \log_a(x)$$

Merke nur höchste Potenz

Ignoriere alle anderen Potenzen außer der höchsten

$$T(n) = n^2 - n + 1 \Rightarrow O(n^2)$$

Daher in Kombination:

$$T(n) = 13*n^2 + 47*n - 11*\log_2(n) + 50000 \Rightarrow O(n^2)$$

```
void bubble(Element v[], int n) {  
    int i, j;  
    for(i = n-1; i >= 1; i--)  
        for(j = 1; j <= i; j++)  
            if(v[j-1] > v[j])  
                swap(v[j-1], v[j]);  
}
```

Problemgröße n

T_1

T_2

T_3

T_4

T_5

Annahme

$$T_1 = \dots T_5 = 1$$

Informeller Ansatz:

Gleichung finden

$$T(n) = T_1 + (n-1) \cdot (T_2 + (n-1) \cdot (T_3 + T_4 + T_5))$$

$$T(n) = 1 + (n-1) \cdot (1 + (n-1) \cdot (1 + 1 + 1)) = 3n^2 - 5n + 3$$

Ignoriere konstante Faktoren

$$T(n) = n^2 - n + 1$$

Merke höchste Potenz

$$T(n) = O(n^2) = \Omega(n^2) = \Theta(n^2)$$

worst-case =
best-case =
average case

Einfache Rekursion

```
int sum(int n) {  
    if (n <= 0) return 0;    T1  
    if (n == 1) return 1;    T2  
    return n + sum(n-1);    T3+T(n-1)  
}
```

Lösungsansatz Rekurrenzgleichung:

$$T(n) = T_1 + T_2 + T_3 + T(n-1)$$

$$T(n) = T(n-1) + 1 \quad T(0)=T(1)=1$$

$$T(n) = T(n-2) + 1 + 1$$

...

$$T(n) = 1 + \underbrace{\dots + 1 + 1}_n$$

$$T(n) = O(n)$$

Einige wichtige Rekurrenzen

$$T(n) = T(n-1) + n \Rightarrow \\ \Rightarrow T(n) = n \cdot (n+1) / 2 = O(n^2)$$

$$T(n) = T(n/2) + 1 \Rightarrow T(n) = \log_2 n = O(\log n)$$

$$T(n) = T(n/2) + n \Rightarrow T(n) = 2 \cdot n = O(n)$$

$$T(n) = 2 \cdot T(n/2) + n \Rightarrow \\ \Rightarrow T(n) = n \cdot \log_2 n = O(n \cdot \log n)$$

$$T(n) = 2 \cdot T(n/2) + 1 \Rightarrow T(n) = 2^n - 1 = O(n)$$

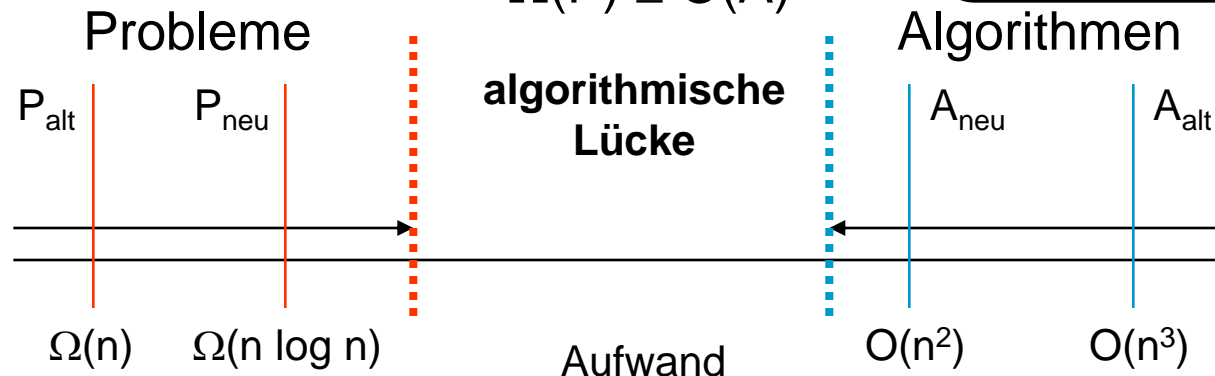
Eine *algorithmische Lücke* für ein Problem existiert, falls für die bekannten problemlösenden Algorithmen A gilt, dass ihr Aufwand größer als der ableitbare Lösungsaufwand für das Problem P ist.

$O(A)$ dient zur Beschreibung des Algorithmus

$\Omega(P)$ dient zur Beschreibung des Problems

$$\Omega(P) \leq O(A)$$

Ziel:
die algorithmische
Lücke zu schließen,
d.h. $\Omega(P) = O(A)$



Geschlossene Lücke

Suchen in sortierter Sequenz, $T(A) = O(\log(n)) = \Omega(\log(n))$

Sortieren, $T(A) = O(n \cdot \log(n)) = \Omega(n \cdot \log(n))$

Offene Lücke

Graphenisomorphie, $T(A_{naiv})$ aus $O(|V|!)$

1.3.6 Laufzeitanalyse von rekursiven Programmen



Die Laufzeitanalyse von rekursiven Programmen ist meist nicht-trivial

Laufzeitverhalten eines rekursiven Programms lässt sich durch eine Rekurrenz beschreiben

Beispiel: Mergesort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

wobei folgende Lösung angegeben wurde

$$T(n) = \Theta(n \log n)$$

Es existieren unterschiedliche Lösungsansätze, wie z.B.

Substitutionsmethode

Erraten einer asymptotischen Grenze und Beweis dieser Grenze durch Induktion

Iterationsmethode

Umwandlung der Rekurrenz in eine Summe und Anwendung von Techniken zur Grenzwertberechnung von Summen

Mastermethode

Liefert Grenzen für Rekurrenzen der Form

$T(n) = aT(n/b) + f(n)$, wobei $a \geq 1$, $b > 1$ und $f(n)$ ist eine gegebene Funktion

Das *Master Theorem* stellt ein „Kochrezept“ zur Bestimmung des Laufzeitverhaltens dar

Vereinfachte Form (generelle Version Cormen et al.)

Es seien $a \geq 1$, $b > 1$ und $c \geq 0$ Konstante

Wenn $T(n)$ durch $aT(n/b) + \Theta(n^c)$ definiert ist,
wobei n/b entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ ist,

Dann besitzt $T(n)$ den folgenden asymptotischen Grenzwert

Fall 1: wenn $c < \log_b a$ (d.h. $b^c < a$) dann $T(n) = \Theta(n^{\log_b a})$

Fall 2: wenn $c = \log_b a$ (d.h. $b^c = a$) dann $T(n) = \Theta(n^c \log n)$

Fall 3: wenn $c > \log_b a$ (d.h. $b^c > a$) dann $T(n) = \Theta(n^c)$

Binäres Suchen

```
int bs (int x, int z[], int l, int r) {  
    if (l > r) // Schwelle, kein Element vorhanden  
        return -1; // 0 verboten, da gültiger Indexwert!  
    else {  
        int m = (l + r) / 2;  
        if (x == z[m]) // gefunden!  
            return m;  
        else if (x < z[m])  
            return bs(x, z, l, m-1);  
        else // x > z[m]  
            return bs(x, z, m+1, r);  
    }  
}
```

Fall 1: wenn $c < \log_b a$ dann $T(n) = \Theta(n^{\log_b a})$
Fall 2: wenn $c = \log_b a$ dann $T(n) = \Theta(n^c \log n)$
Fall 3: wenn $c > \log_b a$ dann $T(n) = \Theta(n^c)$

Laufzeit: $T(n) = T(n/2) + 1 = T(n/2) + \Theta(1)$

$a = 1, b = 2, c = 0$

Fall 2, da $c = \log_b a \Rightarrow 0 = \log_2 1$, ergibt $T(n) = \Theta(\log n)$

Beispiel: Mergesort

$$T(n) = 2T(n/2) + n = 2T(n/2) + \Theta(n)$$

$$a = 2, b = 2, c = 1$$

Fall 2, da $c = \log_b a \Rightarrow 1 = \log_2 2$, ergibt $T(n) = \Theta(n \log n)$

Fall 1: wenn $c < \log_b a$ dann $T(n) = \Theta(n^{\log_b a})$

Fall 2: wenn $c = \log_b a$ dann $T(n) = \Theta(n^c \log n)$

Fall 3: wenn $c > \log_b a$ dann $T(n) = \Theta(n^c)$

Beispiel: $T(n) = 4T(n/2) + n = 4T(n/2) + \Theta(n)$

$$a = 4, b = 2, c = 1$$

Fall 1, da $c < \log_b a \Rightarrow 1 < \log_2 4$, ergibt $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

Beispiel: $T(n) = T(n/2) + n = T(n/2) + \Theta(n)$

$$a = 1, b = 2, c = 1$$

Fall 3, da $c > \log_b a \Rightarrow 1 > \log_2 1$, ergibt $T(n) = \Theta(n^1) = \Theta(n)$

Bewertende Aussage über den strukturellen Aufbau des
Programms und der Programmteile untereinander, d.h.
Bewertung des Programmierstils

interne Attribute

Allgemein angenommen, dass Programmierstil mit den zu erwartenden
Softwarewartungskosten korreliert.

Aussagen über die Qualität eines Softwareproduktes (externe
Attribute)

Fehleranfälligkeit
Wartungsaufwand
Kosten

Annahme:
interne Attribute sind mit externen
Attributen korreliert!

These

komplexes Programm \Rightarrow hohe Kosten
klares Programm \Rightarrow geringere Kosten

„When you can measure what you are speaking about and express it in numbers you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.“

Lord Kelvin

Ziel Metriken (Maßzahlen) zu finden, die den Aufbau, Struktur, Stil eines *Moduls* (Programmstücks) bewerten und vergleichen lassen.

Anforderungen

Gültigkeit

Misst tatsächlich was es vorgibt zu messen

Einfachheit

Resultate sind einfach verständlich und interpretierbar

Sensitivität

Reagiert ausreichend auf unterschiedliche Ausprägungen

Robustheit

Reagiert nicht auf im Zusammenhang uninteressante Eigenschaften

Fokus der Messung ist das Software-Modul

Definition schwierig – Kann Funktion, Methode, Klasse, etc. sein

Die *Intra-modulare Komplexität* beschreibt die Komplexität eines einzelnen SW Moduls

Modul Komplexität (intern)

LOC, Line-of-codes (simpel)

NCSS, non commenting source statements

McCabe (zyklomatische Komplexität), ...

Kohäsion (extern)

Henry-Kafura Metrik (Informationsfluss), ...

Die *Inter-modulare Komplexität* beschreibt Komplexität zwischen Modulen - *Kupplung*

Fenton und Melton, ...

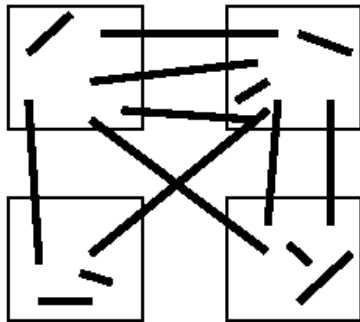
Kupplung

Misst Komplexität der Beziehungen zwischen Moduln

Kohäsion

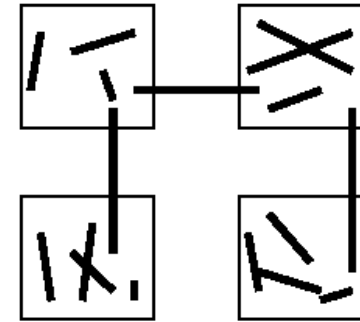
Misst Informationsfluss von und nach Außen abhängig von der Modulgröße

Meist Beziehung zwischen Kupplung und Kohäsion



Starke Kupplung

Schwache Kohäsion



Schwache Kupplung

Starke Kohäsion

(üblicherweise das Ziel guter SW-Entwicklung)

Die *Metrik von McCabe* ist ein Maß zur Beurteilung der Modul Komplexität

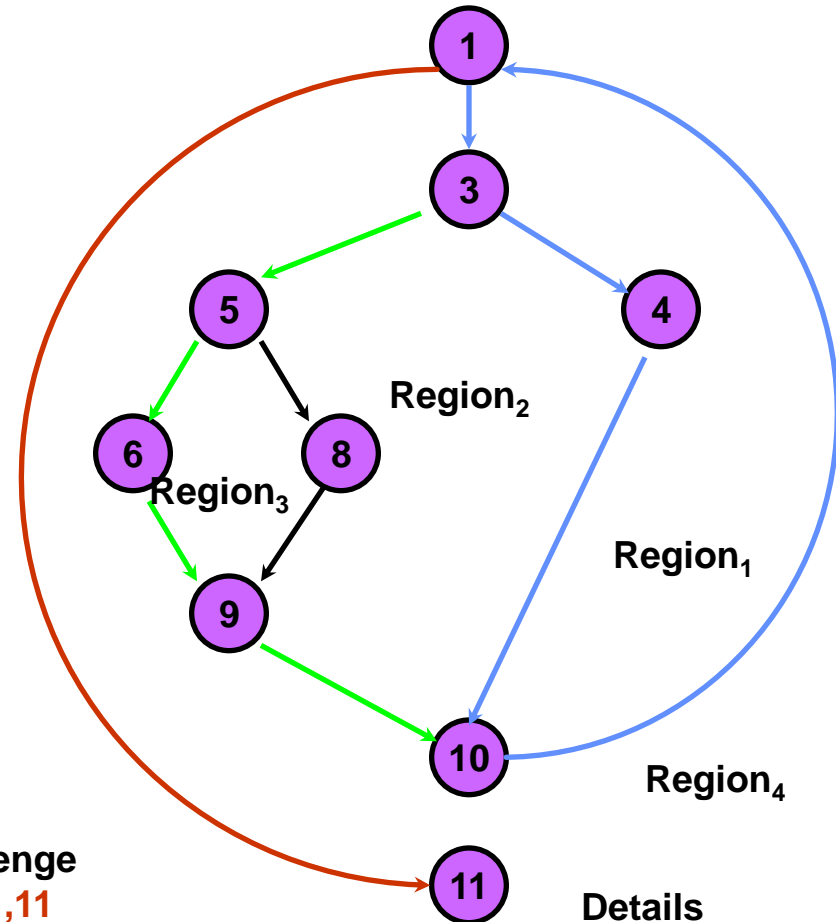
Basiert auf der *zyklomatischen Komplexität* V = die Anzahl der unabhängigen Pfade in einem Programmgraph

Bei einem unabhängigen Pfad wird mindestens eine 'neue' Kante im Programmablaufplan beschriftet.

Erfahrungswerte für die zyklomatische Komplexität

V(G)	Einschätzung im Normalfall
< 5	einfach
5-10	normal
> 10	komplex, sollte restrukturiert werden
> 20	schwer verständlich, wahrscheinlich fehlerhaft


```
0: void foo (int a) {  
1:   while (a < limit) {  
2:     doit1();  
3:     if(check1(a))  
4:       doit2();  
5:     else { if(check2(a))  
6:       doit3();  
7:     else  
8:       doit4();  
9:     } // end else  
10:  } // end while  
11: } // end foo
```



Basis Menge

Pfad 1: 1,11

Pfad 2: 1,3,4,10,1,11

Pfad 3: 1,3,5,6,9,10,1,11

Pfad 4: 1,3,5,8,9,10,1,11

Details

Kanten = 11

Knoten = 9

Bedingungsknoten = 3

Mehrere Berechnungsmöglichkeiten

1. Die Anzahl der Regionen im Programmgraph G
2. $V(G) = E - N + 2$ (E = Anz. d. Kanten, N = Anz. d. Knoten)
3. $V(G) = P + 1$ (P = Anzahl der binären Bedingungsknoten)

Zyklomatische Komplexität des Beispiels

1. Regionen = 4
2. $V(G) = 11 - 9 + 2 = 4$
3. $V(G) = 3 + 1 = 4$

In unserem Beispiel ist die magische Zahl 4, d.h.
„einfaches“ Programm (Metrik McCabe < 5)

Maß zur Bestimmung der Kohäsion

Beschreibt die funktionale Stärke des Moduls; zu welchem Grad die
Modulkomponenten dieselbe Aufgabe erfüllen

Die *Henry-Kafura Metrik* (Sallie Henry and Dennis Kafura)
basiert auf Zusammenhang zwischen Modulkomplexität und
Verbindungskomplexität zwischen Modulen

Maß für Modulkomplexität

LOC

NCSS

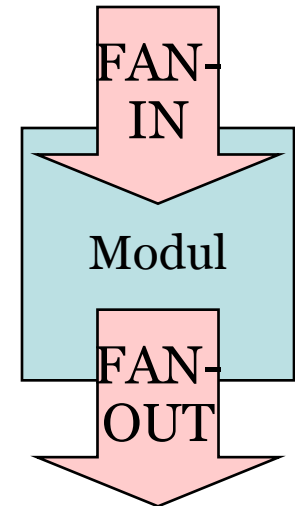
McCabe

Quantifizierung des lesenden und verändernden Zugriffs des Moduls auf die Umgebung

Zählt die Datenflüsse zwischen Moduln

FAN-IN_m: „Anzahl der Module die m verwenden“
genauer: Anzahl der Datenflüsse, die im Modul m terminieren + Anzahl der Datenstrukturen, aus denen der Modul m Daten ausliest

FAN-OUT_m: „Anzahl der Module die m verwendet“
genauer: Anzahl der Datenflüsse, die vom Modul m ausgehen + Anzahl der Datenstrukturen, die der Modul m verändert



Henry-Kafura Formel

$$C_{im} * (FAN-IN_m * FAN-OUT_m)^2$$

C_{im} ... Modulkomplexität (z.B. LOC, McCabe, ...)

Von Henry und Kafura auf Unix Code angewendet

Annahme: Zusammenhang zwischen Komplexität und
Änderungshäufigkeit (Fehlerkorrektur) einer Prozedur

165 Prozeduren untersucht, Patch-Information aus Newsgroups

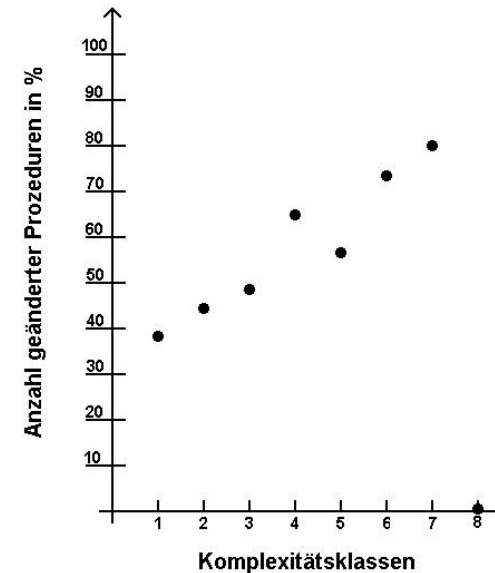
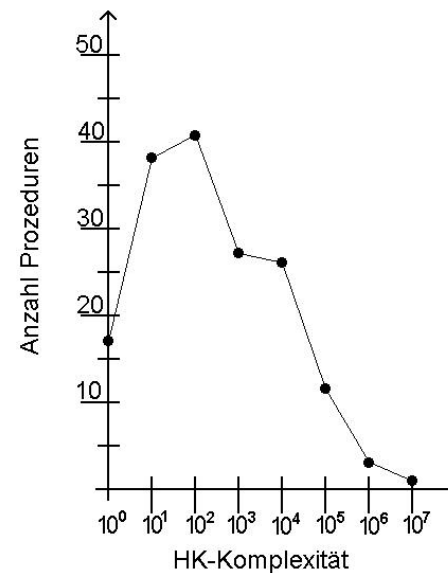
Annahme bestätigt:

Änderungen nehmen mit
Komplexitätsklasse zu

Schwächen von HK:

Abh. vom Datenfluss, bei
einem FAN = 0, gesamt 0
auch bei hoher
Modulkomplexität

Wiederverwendbarkeit wird durch
hohen FAN Wert „bestraft“



Fenton und Melton Metrik ist ein Maß zur Bestimmung der Kupplung, d.h. die Unabhängigkeit zwischen Moduln

Globale Kupplung eines Programms wird abgeleitet aus den Kupplungswerten zwischen allen möglichen Modulpaaren

Kupplungstypen

Binäre Relationen definiert auf Paaren von Moduln x, y

Nach dem Grad der „Unerwünschtheit“ geordnet

0. No coupling: keine Kommunikation zwischen x und y
1. Data coupling: Kommunikation über Parameter (Daten)
2. Stamp coupling: akzeptieren selben Record-Typ als Parameter
3. Control coupling: Kommunikation über Parameter (Kontrolle)
4. Common coupling: Zugriff auf dieselbe globale Datenstruktur
5. Content coupling: x greift direkt auf interne Struktur von y zu (ändert Daten, Anweisungen)

Fenton-Maß für die Kupplung zwischen 2 Moduln

$$c(x,y) = i + n/(n+1)$$

i ist der schlechteste Kupplungstyp zwischen Modul x und y

n ist die Anzahl der „Kupplungen“ vom Typ i

Maß $C(S)$ für die globale Kupplung eines Systems S bestehend aus n Moduln D_1, \dots, D_n

$C(S) = \text{Median der Menge der Kupplungswerte aller Modulpaare}$

Algorithmenparadigmen

Greedy, Divide and Conquer, Dynamic Programming

Analyse und Bewertung von Algorithmen

Effektivität, Korrektheit, Termination

Laufzeitkomplexität

Ordnungsnotation, Algorithmische Lücke, Master Theorem

Strukturkomplexität

McCabe, Henry-Kafura, Fenton