

Algorithmen und Datenstrukturen 1

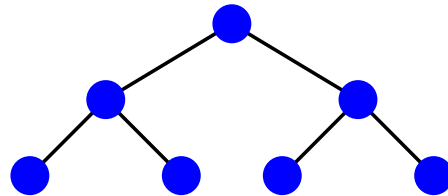
Prof. Dr. E. Rahm

Wintersemester 2001 / 2002

Universität Leipzig

Institut für Informatik

<http://dbs.uni-leipzig.de>



Zur Vorlesung allgemein

Vorlesungsumfang: 2 + 1 SWS

Vorlesungsskript

- im WWW abrufbar (PDF, PS und HTML)
- Adresse <http://dbs.uni-leipzig.de>
- ersetzt nicht die Vorlesungsteilnahme !
- ersetzt nicht zusätzliche Benutzung von Lehrbüchern

Übungen

- Durchführung in zweiwöchentlichem Abstand
- selbständige Lösung der Übungsaufgaben wesentlich für Lernerfolg
- Übungsblätter im WWW
- praktische Übungen auf Basis von Java
- Rechnerzeiten reserviert im NT-Pool (HG 1-68, Mo-Fr. nachmittags) und Sun-Pool (HG 1-46, vormittags und Mittwoch nachmittags)
- Detail-Informationen siehe WWW



Leistungsbewertung

Erwerb des Übungsscheins ADS1 (unbenotet)

- Fristgerechte Abgabe der Lösungen zu den gestellten Übungsaufgaben
- Übungsklausur Ende Jan./Anfang Feb.
- Zulassungsvoraussetzung ist korrekte Lösung der meisten Aufgaben und Bearbeitung aller Übungsblätter (bis auf höchstens eines)

Informatiker (Diplom), 3. Semester

- Übungsschein ADS1 zu erwerben (Voraussetzung für Vordiplomsklausur)
- Klausur über Modul ADS (= ADS1+ADS2) im Juli als Teilprüfung zur Vordiploms-Fachprüfung „Praktische Informatik“

Mathematiker / Wirtschaftsinformatiker: Übungsschein ADS1 erforderlich

Magister mit Informatik als 2. Hauptfach

- kein Übungsschein erforderlich
- Prüfungsklausur zu ADS1 + ADS2 im Juli
- Bearbeitung der Übungsaufgaben wird dringend empfohlen



Termine Übungsbetrieb

Ausgabe 1. Übungsblatt: Montag, 15. 10. 2001; danach 2-wöchentlich

Abgabe gelöster Übungsaufgaben bis spätestens Montag der übernächsten Woche, 11:15 Uhr

- vor Hörsaal 13 (Abgabemöglichkeit 11:00 - 11:15 Uhr)
- oder früher im Holz-Postkasten HG 3. Stock, Abt. Datenbanken
- Programmieraufgaben: dokumentierte Listings der Quellprogramme sowie Ausführung

6 Übungsgruppen

Nr.	Termin	Woche	Hörsaal	Beginn	Übungsleiter	#Stud.	Bemerkung
1	Mo, 15:15	B	SG 3-11	29.10.	Sosna	30	
2	Mo, 15:15	A	SG 3-11	5.11.	Sosna	30	Ausweichtermin wegen Dies A.: Di, 4.12., 11:15 Uhr, SG 3-07
3	Di, 11:15	B	SG 3-07	30.10.	Böhme	30	
4	Di, 11:15	A	SG 3-07	6.11.	Böhme	30	
5	Fr, 15:15	B	HS 20	2.11.	Müller	60	
6	Fr, 15:15	A	HS 20	9.11.	Müller	60	
7	Di, 9:15	B	SG 3-05	30.10.	Müller	30	

- Einschreibung über Online-Formular
- Aktuelle Infos siehe WWW



Ansprechpartner ADS1

Prof. Dr. E. Rahm

- während/nach der Vorlesung bzw. Sprechstunde (Donn. 14-15 Uhr), HG 3-56
- rahm@informatik.uni-leipzig.de

Wissenschaftliche Mitarbeiter

- Timo Böhme, boehme@informatik.uni-leipzig.de, HG 3-01
- Robert Müller, mueller@informatik.uni-leipzig.de, HG 3-01
- Dr. Dieter Sosna, dieter@informatik.uni-leipzig.de, HG 3-04

Studentische Hilfskräfte

- Tilo Dietrich, TiloDietrich@gmx.de
- Katrin Starke, katrin.starke@gmx.de
- Thomas Tym, mai96iwe@studserv.uni-leipzig.de

Web-Angelegenheiten:

- S. Jusek, juseks@informatik.uni-leipzig.de, HG 3-02



Vorläufiges Inhaltsverzeichnis

1. Einführung
 - Komplexität von Algorithmen
 - Bestimmung der Zeitkomplexität
 - Das Prinzip "Teile und Herrsche"
2. Einfache Suchverfahren (Arrays)
3. Verkettete Listen, Stacks und Schlangen
4. Sortierverfahren
 - Elementare Verfahren
 - Shell-Sort, Heap-Sort, Quick-Sort
 - Externe Sortierverfahren
5. Allgemeine Bäume und Binärbäume
 - Orientierte und geordnete Bäume
 - Binärbäume (Darstellung, Traversierung)
6. Binäre Suchbäume
7. Mehrwegbäume



Literatur

Das intensive Literaturstudium zur Vertiefung der Vorlesung wird dringend empfohlen. Auch Literatur in englischer Sprache sollte verwendet werden.

T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, Reihe Informatik, Band 70, BI-Wissenschaftsverlag, 3. Auflage, Spektrum-Verlag, 1996

M.A. Weiss: Data Structures & Algorithm Analysis in Java. Addison-Wesley 1999, 2. Auflage 2002

Weitere Bücher

- *V. Claus, A. Schwill*: Duden Informatik, BI-Dudenverlag, 2. Auflage 1993
- *D.A. Knuth*: The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973
- *R. Sedgewick*: Algorithmen. Addison-Wesley 1992
- *G. Saake, K. Sattler*: Algorithmen und Datenstrukturen - Eine Einführung mit Java. dpunkt-Verlag, 2002
- *A. Solymosi, U. Gude*: Grundkurs Algorithmen und Datenstrukturen. Eine Einführung in die praktische Informatik mit Java. Vieweg, 2000, 2. Auflage 2001



Einführung

Algorithmen stehen im Mittelpunkt der Informatik

Wesentliche Entwurfsziele bei Entwicklung von Algorithmen:

- Korrektheit
- Terminierung
- Effizienz

Wahl der Datenstrukturen v.a. für Effizienz entscheidend

Abstrakte Datentypen (ADTs): Zusammenfassung von Algorithmen und Datenstrukturen

Vorlesungsschwerpunkte:

- Entwurf von effizienten Algorithmen und Datenstrukturen
- Analyse ihres Verhaltens



Komplexität von Algorithmen

funktional gleichwertige Algorithmen weisen oft erhebliche Unterschiede in der Effizienz (Komplexität) auf

Wesentliche Maße:

- Rechenzeitbedarf (Zeitkomplexität)
- Speicherplatzbedarf (Speicherplatzkomplexität)

Programmlaufzeit von zahlreichen Faktoren abhängig

- Eingabe für das Programm
- Qualität des vom Compiler generierten Codes und des gebundenen Objektprogramms
- Leistungsfähigkeit der Maschineninstruktionen, mit deren Hilfe das Programm ausgeführt wird
- Zeitkomplexität des Algorithmus, der durch das ausgeführte Programm verkörpert wird

Bestimmung der Komplexität

- Messungen auf einer bestimmten Maschine
- Aufwandsbestimmungen für idealisierten Modellrechner (Bsp.: Random-Access-Maschine oder RAM)
- Abstraktes Komplexitätsmaß zur asymptotischen Kostenschätzung in Abhängigkeit zur Problemgröße (Eingabegröße) n



Bestimmungsfaktoren der Komplexität

Zeitkomplexität T ist i.a. von "Größe" der Eingabe n abhängig

Beispiel: $T(n) = a \cdot n^2 + b \cdot n + c$

Verkleinern der Konstanten b und c

$$T_1(n) = n^2 + n + 1$$

$$T_2(n) = n^2$$

n	1	2	3	10	20	100	1000
$T_1(n)$	3	7	13	111	421	10101	1001001
$T_2(n)$	1	4	9	100	400	10000	1000000
T_1/T_2	3	1.75	1.44	1.11	1.05	1.01	1.001

Verbessern der Konstanten a nach a'

$$\lim_{n \rightarrow \infty} \frac{a \cdot n^2 + b \cdot n + c}{a' \cdot n^2 + b' \cdot n + c'} = \frac{a}{a'}$$

Wesentlich effektiver: Verbesserung im Funktionsverlauf!
(Wahl eines anderen Algorithmus mit günstigerer Zeitkomplexität)



Asymptotische Kostenmaße

Festlegung der Größenordnung der Komplexität in Abhängigkeit der Eingabegröße: Best Case, Worst Case, Average Case

Meist Abschätzung oberer Schranken (Worst Case): *Groß-Oh-Notation*

Zeitkomplexität $T(n)$ eines Algorithmus ist von der Größenordnung n , wenn es Konstanten n_0 und $c > 0$ gibt, so daß für alle Werte von $n > n_0$ gilt

$$T(n) \leq c \cdot n$$

- man sagt "T(n) ist in $O(n)$ " bzw. " $T(n) \in O(n)$ " oder " $T(n) = O(n)$ "

Allgemeine Definition:

Klasse der Funktionen $O(f)$, die zu einer Funktion (Größenordnung) f gehören ist $O(f) = \{g \mid \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$

Ein Programm, dessen Laufzeit oder Speicherplatzbedarf $O(f(n))$ ist, hat demnach die Wachstumsrate $f(n)$

- Beispiel: $f(n) = n^2$ oder $f(n) = n \cdot \log n$
- $f(n) = O(n \log n) \rightarrow f(n) = O(n^2)$, jedoch gilt natürlich $O(n \log n) \neq O(n^2)$



Asymptotische Kostenmaße (2)

Beispiel: $6n^4 + 3n^3 - 7n \in O(n^4)$

- zu zeigen: $6n^4 + 3n^3 - 7n \leq c n^4$ für ein c und alle $n > n_0$
 $\rightarrow 6 + 3/n - 7/n^4 \leq c$
- Wähle also z.B. $c = 9, n_0 = 1$

Groß-Omega-Notation: $f \in \Omega(g)$ oder $f = \Omega(g)$ drückt aus, daß f mindestens so stark wächst wie g (untere Schranke)

- Definition: $\Omega(g) = \{h \mid \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : h(n) \geq c g(n)\}$
- alternative Definition (u.a. Ottmann/Widmayer):
 $\Omega(g) = \{h \mid \exists c > 0 : \exists \text{ unendlich viele } n : h(n) \geq c g(n)\}$

Exakte Schranke: gilt für Funktion f sowohl $f \in O(g)$ als auch $f \in \Omega(g)$, so schreibt man $f = \Theta(g)$

- f aus $\Theta(g)$ bedeutet also: die Funktion g verläuft ab einem Anfangswert n_0 im Bereich $[c_1 g, c_2 g]$ für geeignete Konstanten c_1, c_2



Wichtige Wachstumsfunktionen

Kostenfunktionen

- $O(1)$ konstante Kosten
- $O(\log n)$ logarithmisches Wachstum
- $O(n)$ lineares Wachstum
- $O(n \log n)$ n-log n-Wachstum
- $O(n^2)$ quadratisches Wachstum
- $O(n^3)$ kubisches Wachstum
- $O(2^n)$ exponentielles Wachstum

Wachstumsverhalten

log n	3	7	10	13	17	20
\sqrt{n}	3	10	30	100	300	1000
n	10	100	1000	10⁴	10⁵	10⁶
n log n	30	700	10 ⁴	10 ⁵	2 · 10 ⁶	2 · 10 ⁷
n ²	100	10 ⁴	10 ⁶	10 ⁸	10 ¹⁰	10 ¹²
n ³	1000	10 ⁶	10 ⁹	10 ¹²	10 ¹⁵	10 ¹⁸
2 ⁿ	1000	10 ³⁰	10 ³⁰⁰	10 ³⁰⁰⁰	10 ³⁰⁰⁰⁰	10 ³⁰⁰⁰⁰⁰



Problemgröße bei vorgegebener Zeit

Komplexität	1 sec	1 min	1 h
$\log_2 n$	2 ¹⁰⁰⁰	2 ⁶⁰⁰⁰⁰	-
n	1000	60000	3600000
n log ₂ n	140	4893	20000
n ²	31	244	1897
n ³	10	39	153
2 ⁿ	9	15	21

Größe des größten Problems, das in 1 Stunde gelöst werden kann:

Problemkomplexität	aktuelle Rechner	Rechner 100x schneller	1000x schneller
n	N ₁	100 N ₁	1000 N ₁
n ²	N ₁	10 N ₂	32 N ₂
n ³	N ₃	4.6 N ₃	10 N ₃
n ⁵	N ₄	2.5 N ₄	4 N ₄
2 ⁿ	N ₅	N ₁ + 7	N ₁ + 10
3 ⁿ	N ₆	N ₆ + 4	N ₆ + 6



Leistungsverhalten bei kleiner Eingabegröße

Asymptotische Komplexität gilt vor allem für große n

bei kleineren Probleme haben konstante Parameter wesentliche Einfluß

Verfahren mit besserer (asympt.) Komplexität kann schlechter abschneiden als Verfahren mit schlechter Komplexität

Alg.	T(n)	Bereiche von n mit günstigster Zeitkomplexität
A ₁	186182 log ₂ n	n > 2048
A ₂	1000 n	1024 ≤ n ≤ 2048
A ₃	100 n log ₂ n	59 ≤ n ≤ 1024
A ₄	10 n ²	10 ≤ n ≤ 58
A ₅	n ³	n = 10
A ₆	2 ⁿ	2 ≤ n ≤ 9



Zeitkomplexitätsklassen

Drei zentrale *Zeitkomplexitätsklassen* werden unterschieden

Algorithmus A mit Zeitkomplexität T(n) heißt:

linear-zeitbeschränkt $T(n) \in O(n)$

polynomial-zeitbeschränkt $\exists k \in \mathbb{N}$, so daß $T(n) \in O(n^k)$

exponentiell-zeitbeschränkt $\exists k \in \mathbb{N}$, so daß $T(n) \in O(k^n)$

exponentiell-zeitbeschränkte Algorithmen im allgemeinen (größere n) nicht nutzbar

Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, gelten als unlösbar (intractable)



Berechnung der (Worst-Case-) Zeitkomplexität

elementare Operationen (Zuweisung, Ein-/Ausgabe): $O(1)$

Summenregel:

- $T_1(n)$ und $T_2(n)$ seien die Laufzeiten zweier Programmfragmente P_1 und P_2 ; es gelte $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$.
- Für die Hintereinanderausführung von P_1 und P_2 ist dann $T_1(n) + T_2(n) \in O(\max(f(n), g(n)))$

Produktregel, z.B. für geschachtelte Schleifenausführung von P_1 und P_2 :

$$T_1(n) \cdot T_2(n) \in O(f(n) \cdot g(n))$$

Weitere Konstrukte

- Fallunterscheidung: Kosten der Bedingungsanweisung ($= O(1)$) + Kosten der längsten Alternative
- Schleife: Produkt aus Anzahl der Schleifendurchläufe mit Kosten der teuersten Schleifenausführung
- rekursive Prozeduraufrufe: Produkt aus Anzahl der rekursiven Aufrufe mit Kosten der teuersten Prozedurausführung



Beispiel zur Bestimmung der Zeitkomplexität

```
void proz0 (int n) {
    proz1();
    proz1();
    for (int i=1; i <= n; i++) {
        proz2();
        proz2();
        proz2();
        for (int j=1; j <= n; j++) {
            proz3();
            proz3();
        }
    }
}
```



Beispiel: Berechnung der maximalen Teilsumme

Gegeben: Folge F von n ganzen Zahlen. Gesucht: Teilfolge von $0 \leq i \leq n$ aufeinander folgenden Zahlen in F , deren Summe maximal ist

Anwendungsbeispiel: Entwicklung von Aktienkursen (tägliche Änderung des Kurses). Maximale Teilsumme bestimmt optimales Ergebnis

Tag	1	2	3	4	5	6	7	8	9	10
Gewinn/Verlust (Folge)	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

Lösungsmöglichkeit 1:

```
int maxSubSum1 (int [ ] a) {
    int maxSum = 0; // leere Folge
    for (int i = 0; i < a.length; i++)
        for (int j = i; j < a.length; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++)
                thisSum += a[ k ];
            if ( thisSum > maxSum ) maxSum = thisSum;
        }
    return maxSum;
}
```



Komplexitätsbestimmung

Anzahl Durchläufe der äussersten Schleife: n

mittlere Schleife: berücksichtigt alle Teilfolgen beginnend ab Position i

- Anzahl: $n, n-1, n-2, \dots, 1$
- Mittel: $n+1/2$

Anzahl Teilfolgen: $\sum i = (n^2+n)/2$

innerste Schleife: Addition aller Werte pro Teilfolge

$$\begin{aligned} \# \text{Additionen: } \sum i(n+1-i) &= \sum i \cdot n + \sum i - \sum i^2 \\ &= n \cdot (n^2+n)/2 + (n^2+n)/2 - n/6 \cdot (n+1) \cdot (2n+1) \\ &= n^3/6 + n^2/2 + n/3 \end{aligned}$$

Zeitkomplexität:



Maximale Teilsumme (2)

Lösungsmöglichkeit 2:

```
int maxSubSum2 (int [ ] a) {
    int maxSum = 0; // leere Folge
    for (int i = 0; i < a.length; i++) {
        int thisSum = 0;
        for (int j = i; j < a.length; j++) {
            thisSum += a[j];
            if (thisSum > maxSum) maxSum = thisSum;
        }
    }
    return maxSum;
}
```

Zeitkomplexität:



Maximale Teilsumme (3)

Lösungsmöglichkeit 3:

```
int maxSubSum3 (int [ ] a) {
    int maxSum = 0;
    int thisSum = 0;
    for (int i = 0, j = 0; j < a.length; j++) {
        thisSum += a[j];
        if (thisSum > maxSum) maxSum = thisSum;
        else if (thisSum < 0) {
            i = j + 1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```

Zeitkomplexität:

gibt es Lösungen mit besserer Komplexität?



Rekursion vs. Iteration

für viele Probleme gibt es sowohl rekursive als auch iterative Lösungsmöglichkeiten

Unterschiede bezüglich

- Einfachheit, Verständlichkeit
- Zeitkomplexität
- Speicherkomplexität

Beispiel: Berechnung der Fakultät $n!$

```
int fakRekursiv (int n) { // erfordert n > 0
    if (n <= 1) return 1;
    else return n * fakRekursiv (n-1);
}

int fakIterativ (int n) { // erfordert n > 0
    int fak = 1;
    for (int i = 2; i <= n; i++) fak *= i;
    return fak;
}
```

- Zeitkomplexität
- Speicherkomplexität



Berechnung der Fibonacci-Zahlen

Definition

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$

rekursive Lösung verursacht exponentiellen Aufwand

```
int fibRekursiv (int n) { // erfordert n > 0
    if (n <= 0) return 0;
    else if (n == 1) return 1;
    else return fibRekursiv (n-2) + fibRekursiv (n-1)
}
```

iterative Lösung mit linearem Aufwand möglich

- z.B. Speichern der bereits berechneten Fibonacci-Zahlen in Array
- Alternative *fibIterativ*

```
int fibIterativ (int n) { // erfordert n > 0
    if (n <= 0) return 0;
    else {
        int aktuelle = 1, vorherige = 0, temp = 1;
        for (int i = 1; i < n; i++) {
            temp = aktuelle;
            aktuelle += vorherige;
            vorherige = temp;
        }
        return aktuelle;
    }
}
```



Das Prinzip "Teile und Herrsche" (Divide and Conquer)

Komplexität eines Algorithmus läßt sich vielfach durch Zerlegung in kleinere Teilprobleme verbessern

Lösungsschema

1. *Divide*: Teile das Problem der Größe n in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn $n > 1$ ist; sonst löse das Problem der Größe 1 direkt.
2. *Conquer*: Löse die Teilprobleme auf dieselbe Art (rekursiv).
3. *Merge*: Füge die Teillösungen zur Gesamtlösung zusammen.



Beispiel: Sortieren einer Liste mit n Elementen

einfache Sortierverfahren: $O(n^2)$

Divide-and-Conquer-Strategie:

```

Modul      Sortiere (Liste) (* Sortiert Liste von n Elementen *)
Falls n > 1 dann
  Sortiere (erste Listenhälfte)
  Sortiere (zweite Listenhälfte)
  Mische beide Hälften zusammen.
    
```

Kosten $T(n) = 2 \cdot T(n/2) + c \cdot n$ $T(1) = d$

Diese rekursives Gleichungssystem (Rekurrenzrelation) hat geschlossene

Lösung $T(n) = c \cdot n \cdot \log_2 n + d \cdot n$

=> Sortieralgorithmus in $O(n \log n)$



Beispiel 2: Maximale Teilsumme

rechtes Randmaximum einer Folge

- rechte Randfolge von F = Teilfolge von F , die bis zum rechten Rand (Ende) von F reicht
- rechtes Randmaximum von F : maximale Summe aller rechten Randfolgen
- analog: linke Randfolge, linkes Randmaximum

Beispiel: $F = (+3, -2, +5, -20, +3, +3)$

rekursiver (Divide-and-Conquer-) Algorithmus für maximale Teilsumme

- falls Eingabefolge F nur aus einer Zahl z besteht, nimm Maximum von z und 0
- falls F wenigstens 2 Elemente umfasst:
 - zerlege F in etwa zwei gleich große Hälften *links* und *rechts*
 - bestimme maximale Teilsumme, ml , sowie rechtes Randmaximum, rR , von *links*
 - bestimme maximale Teilsumme, mr , sowie linkes Randmaximum, lR , von *rechts*
 - das Maximum der drei Zahlen ml , $rR+lR$, und mr ist die maximale Teilsumme von F



Multiplikation zweier n -stelliger Zahlen

Standardverfahren aus der Schule: $O(n^2)$

$$\begin{array}{r}
 5432 \cdot 1995 \\
 \hline
 5432 \\
 48888 \\
 48888 \\
 27160 \\
 \hline
 10836840
 \end{array}$$

Verbesserung: Rückführung auf Multiplikation von 2-stelligen Zahlen

$$\begin{array}{|c|c|} \hline A & B \\ \hline 54 & 32 \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline C & D \\ \hline 19 & 95 \\ \hline \end{array}$$

$$\begin{array}{r}
 AC = 54 \cdot 19 = 1026 \\
 (A + B) \cdot (C + D) - AC - BD = 86 \cdot 114 - 1026 - 3040 = 5738 \\
 BD = 32 \cdot 95 = 3040 \\
 \hline
 10836840
 \end{array}$$



Multiplikation (2)

Prinzip auf n-stellige Zahlen verallgemeinerbar

Kosten

- drei Multiplikationen von Zahlen mit halber Länge
- Aufwand für Addition und Subtraktion proportional zu n:

$$T(n) = 3T(n/2) + c \cdot n \quad T(1) = d$$

- Die Lösung der Rekurrenzrelation ergibt sich zu

$$T(n) = (2c + d)n^{\log_3 3} - n$$

- Kosten proportional zu $n^{\log_3 3}$ ($O(n^{1.59})$)



Problemkomplexität

Komplexität eines Problems: Komplexität des besten Algorithmus'

Aufwand typischer Problemklassen

Komplexität	Beispiele
$O(1)$	einige Suchverfahren (Hashing)
$O(\log n)$	allgemeinere Suchverfahren (Binärsuche, Baum-Suchverfahren)
$O(n)$	sequentielle Suche, Suche in Texten; maximale Teilsumme einer Folge, Fakultät, Fibonacci-Zahlen
$O(n \log n)$	Sortieren
$O(n^2)$	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume), Multiplikation Matrix-Vektor (einfach)
$O(n^3)$	Matrizen-Multiplikation (einfach)
$O(2^n)$	viele Optimierungsprobleme, Türme von Hanoi, Acht-Damen-Problem

theoretisch nicht lösbare algorithmische Probleme: Halteproblem, Gleichwertigkeit von Algorithmen

nicht-algorithmische Probleme



Zusammenfassung

Komplexität / Effizienz wesentliche Eigenschaft von Algorithmen

meist asymptotische Worst-Case-Abschätzung in Bezug auf Problemgröße n

- Unabhängigkeit von konkreten Umgebungsparametern (Hardware, Betriebssystem, ...)
- asymptotisch „schlechte“ Verfahren können bei kleiner Problemgröße ausreichen

wichtige Klassen: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, ... $O(2^n)$

zu gegebener Problemstellung gibt es oft Algorithmen mit stark unterschiedlicher Komplexität

- unterschiedliche Lösungsstrategien
- Raum vs. Zeit: Zwischenspeichern von Ergebnissen statt mehrfacher Berechnung
- Iteration vs. Rekursion

Bestimmung der Komplexität aus Programmfragmenten

allgemeine Lösungsstrategie: Divide-and-Conquer (Teile und Herrsche)

