Algorithmen und Datenstrukturen

Peter Rossmanith

Theoretische Informatik, RWTH Aachen

13. April 2018

Organisatorisches

Vorlesung:
Peter Rossmanith (Raum AH II, Informatikzentrum)

Sprechstunde: Mittwochs 10:15–11:00

Übung:
Jan Dreier, Philipp Kuinke
`tcs-teaching@cs.rwth-aachen.de`

Nur diese E-Mailadresse!!!!

Organisatorisches

Vorlesung: Freitags, 8:30–10:00 Uhr, AH II

Globalübung+Fragestunde: Mittwochs, 14:15–15:45, R5 (ab 25.04.2018)

Tutorübungen: (ab 23.04.2018)

- Montags 16:15–17:00 und 17:00–17:45
- Mittwochs 16:15–17:00
- Donnerstags 14:15–15:00 und 15:00–15:45

Anmeldung zu Übungsgruppen siehe Link auf Webseite

Ausgabe des Übungsblatts in den Tutorübungen

Alle Materialen auf Webseite `tcs.rwth-aachen.de/lehre/DA/SS2018/`

Organisatorisches

Klausur:

14.8.2018, zwischen 8:00 und 11:30 Uhr (1. Klausur)
26.9.2018, zwischen 8:00 und 11:30 Uhr (2. Klausur)

Anmeldung: zur "Prüfung Datenstrukturen und Algorithmen"
Abmeldung: über Campus-Office

Zulassungskriterien:

50% der Punkte aus den Hausaufgaben

50% der Punkte aus selbstständigen Präsenzübungen während der Tutorübungen (ab 23.04.2018)

Erstellung der Hausaufgaben in Gruppen von bis zu n Personen
und Abgabe in der folgenden Tutorübung. Die Konstante n wird in Kürze festgelegt.

Übersicht

Übersicht

Einordnung

Etwas Vorwissen kann bei Algorithmen und Datenstrukturen helfen. Diese Vorlesungen sind nützlich:

- Programmierung
- Diskrete Strukturen
- Analysis
- Stochastik

Einordnung

Vorlesungen, die Datenstrukturen und Algorithmen ergänzen:

- Berechenbarkeit und Komplexität
- Formale Systeme, Automaten, Prozesse
- Numerisches Rechnen
- Effiziente Algorithmen

Übersicht

Drei sehr umfassende Bücher


T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
Introduction to Algorithms.
The MIT Press, 2d edition, 2001.


T. Ottmann and P. Widmayer.
Algorithmen und Datenstrukturen.
Spektrum Verlag, 2002.


K. Mehlhorn and P. Sanders.
Algorithms and Data Structures: The Basic Toolbox.
Springer, 2008.

Weitere interessante Bücher


A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.


S. Skiena.
The Algorithm Design Manual.
Telos, 1997.

Übersicht

Komplexität von Algorithmen

Eine zentrale Frage:

> *Zwei verschiedene Algorithmen lösen dasselbe Problem.*
> *Welcher ist schneller?*

Unterscheide:

1. Die Laufzeit eines konkreten Algorithmus
2. Die Geschwindigkeit mit der sich ein Problem lösen läßt
   → Komplexitätstheorie

Komplexität von Algorithmen

Eine zentrale Frage:

> *Zwei verschiedene Algorithmen lösen dasselbe Problem.*
> *Welcher ist schneller?*

Unterscheide:

1. Die Laufzeit eines konkreten Algorithmus
2. Die Geschwindigkeit mit der sich ein Problem lösen läßt
   → Komplexitätstheorie

Komplexität von Algorithmen

Eine zentrale Frage:

> *Zwei verschiedene Algorithmen lösen dasselbe Problem.*
> *Welcher ist schneller?*

Unterscheide:

1. Die Laufzeit eines konkreten Algorithmus
2. Die Geschwindigkeit mit der sich ein Problem lösen läßt
   $\rightarrow$ Komplexitätstheorie

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf dieser Eingabe schneller.

Es gibt aber viele andere mögliche Eingaben.

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf dieser Eingabe schneller.

Es gibt aber viele andere mögliche Eingaben.

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf dieser Eingabe schneller.

Es gibt aber viele andere mögliche Eingaben.

Worst-Case–Komplexität von Algorithmen

Lösung:

Die Laufzeit eines Algorithmus ist nicht eine Zahl, sondern eine Funktion $\mathbf{N} \to \mathbf{N}$.

Sie gibt die Laufzeit des Algorithmus für jede Eingabelänge an.

Die Laufzeit für Eingabelänge n ist die schlechteste Laufzeit aus allen Eingaben mit Länge n.

Wir nennen dies die Worst-Case–Laufzeit.

Worst-Case–Komplexität von Algorithmen

Lösung:

Die Laufzeit eines Algorithmus ist nicht eine Zahl, sondern eine Funktion $\mathbf{N} \to \mathbf{N}$.

Sie gibt die Laufzeit des Algorithmus für jede Eingabelänge an.

Die Laufzeit für Eingabelänge n ist die schlechteste Laufzeit aus allen Eingaben mit Länge n.

Wir nennen dies die Worst-Case–Laufzeit.

Sequentielle Berechenbarkeitshypothese

Church'sche These:

> *Die Menge der algorithmisch lösbaren Probleme ist für alle vernünftigen Berechnungsmodelle identisch.*

→ Vorlesung Berechenbarkeit und Komplexität

„Theorem“

Sequentielle Berechenbarkeitshypothese:

Die benötigte Laufzeit für ein Problem auf zwei unterschiedlichen sequentiellen Berechenbarkeitsmodellen unterscheidet sich nur um ein Polynom.

Ein beliebiges Polynom ist für uns zu grob!

Sequentielle Berechenbarkeitshypothese

Church'sche These:

> *Die Menge der algorithmisch lösbaren Probleme ist für alle vernünftigen Berechnungsmodelle identisch.*

→ Vorlesung Berechenbarkeit und Komplexität

„Theorem“

Sequentielle Berechenbarkeitshypothese:

Die benötigte Laufzeit für ein Problem auf zwei unterschiedlichen sequentiellen Berechenbarkeitsmodellen unterscheidet sich nur um ein Polynom.

Ein beliebiges Polynom ist für uns zu grob!

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die Random Access Machine (RAM).

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die Random Access Machine (RAM).

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die Random Access Machine (RAM).

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Eine RAM besteht aus:

1. Einem Speicher aus Speicherzellen 1, 2, 3,...
2. Jede Speicherzelle kann beliebige Zahlen speichern
3. Der Inhalt einer Speicherzelle kann als Anweisung interpretiert werden
4. Einem Prozessor der einfache Anweisungen ausführt
5. Anweisungen: Logische und Arithmetische Verknüpfungen, (bedingte) Sprünge
6. Jede Anweisung benötigt konstante Zeit

Grenzen des RAM-Modells

- Jede Speicherzelle enthält beliebige Zahlen:
  Unrealistisch, wenn diese sehr groß werden
- Jede Anweisung benötigt gleiche Zeit:
  Unrealistisch, wegen Caches, Sprüngen

Daumenregel: Bei n Speicherzellen, $O(\log n)$ bits verwenden.

```
void quicksort(int N)
{
    int i,j,l,r,k,t;
    l=1; r=N;
    if(N>M)
      while(1) {
        i=l-1; j=r; k=a[j];
        do {
            do { i++; } while(a[i]<k);
            do { j--; } while(k<a[j]);
            t=a[i]; a[i]=a[j]; a[j]=t;
        } while(i<j);
        a[j]=a[i]; a[i]=a[r]; a[r]=t;
        if(r-i ≥ i-l) {
            if(i-l>M) { push(i+1,r); r=i-1; }
            else if(r-i>M) l=i+1;
            else if(stack_is_empty) break;
            else pop(l,r);
        }
        else
            if(r-i>M) { push(l,i-1); l=i+1; }
            else if(i-l>M) r=i-1;
            else if(stack_is_empty) break;
            else pop(l,r);
      }
    for(i=2;i ≤ N;i++)
        if(a[i-1]>a[i]) {
            k=a[i]; j=i;
            do { a[j]=a[j-1]; j--; } while(a[j-1]>k);
            a[j]=k;
        }
}
```

Quicksort – Ein Beispiel

Statt einer Worst-Case–Analyse führen wir eine Average-Case–Analyse durch.

Die Eingabe besteht aus den Zahlen $1, \ldots, n$ in zufälliger Reihenfolge.

Wie lange benötigt Quicksort im Durchschnitt zum Sortieren?

Es sind $O(n \log n)$ Schritte auf einer RAM.

Für eine genauere Analyse, müssen wir das Modell genau festlegen.

Quicksort – Ein Beispiel

Statt einer Worst-Case–Analyse führen wir eine Average-Case–Analyse durch.

Die Eingabe besteht aus den Zahlen $1, \ldots, n$ in zufälliger Reihenfolge.

Wie lange benötigt Quicksort im Durchschnitt zum Sortieren?

Es sind $O(n \log n)$ Schritte auf einer RAM.

Für eine genauere Analyse, müssen wir das Modell genau festlegen.

```
sw −4(r29),r30
add r30,r0,r29
sw −8(r29),r31
subui r29,r29,#464
sw 0(r29),r2
sw 4(r29),r3
sw 8(r29),r4
sw 12(r29),r5
...
sw 40(r29),r12
lw r11,(r30)
lw r9,4(r30)
slli r1,r11,#0x2
lhi r8,((_a+4)>>16)&0xffff
addui r8,r8,(_a+4)&0xffff
lhi r12,((_a)>>16)&0xffff
addui r12,r12,(_a)&0xffff
add r7,r1,r12
sgt r1,r11,r9
beqz r1,L8
addi r4,r30,#−408
add r10,r0,r4
L11:
addi r3,r8,#−4
L51:
add r2,r0,r7
lw r5,(r7)
L15:
addi r3,r3,#4
L49:
lw r1,(r3)
slt r1,r1,r5
bnez r1,L49
addi r3,r3,#4
addi r3,r3,#−4
addi r2,r2,#−4
L50:
lw r31,(r2)
slt r1,r5,r31
bnez r1,L50
addi r2,r2,#−4
addi r2,r2,#4
lw r6,(r3)
sw (r3),r31
sltu r1,r3,r2
bnez r1,L15
sw (r2),r6
lw r12,(r3)
sw (r2),r12
lw r12,(r7)
sw (r3),r12
sub r1,r7,r3
srai r5,r1,#0x2
sub r1,r3,r8
srai r2,r1,#0x2
sge r1,r5,r2
beqz r1,L24
sw (r7),r6
sgt r1,r2,r9
beqz r1,L25
addi r1,r3,#4
sw (r4),r1
addi r4,r4,#4
sw (r4),r7
addi r4,r4,#4
j L11
addi r7,r3,#−4
L25:
sgt r1,r5,r9
beqz r1,L34
addi r8,r3,#4
j L51
addi r3,r8,#−4
L24:
sgt r1,r5,r9
beqz r1,L32
addi r1,r3,#−4
sw (r4),r8
addi r4,r4,#4
sw (r4),r1
addi r4,r4,#4
j L11
addi r8,r3,#4
L32:
sgt r1,r2,r9
bnez r1,L11
addi r7,r3,#−4
L34:
seq r1,r4,r10
bnez r1,L8
addi r4,r4,#−4
lw r7,(r4)
addi r4,r4,#−4
j L11
lw r8,(r4)
L8:
slli r1,r11,#0x2
lhi r2,((_a)>>16)&0xffff
addui r2,r2,(_a)&0xffff
add r7,r1,r2
addi r3,r2,#8
sleu r1,r3,r7
beqz r1,L40
addi r4,r2,#4
L42:
lw r1,(r4)
lw r2,(r3)
sgt r1,r1,r2
beqz r1,L41
add r5,r0,r2
add r2,r0,r3
addi r31,r3,#−4
L44:
lw r12,(r31)
sw (r2),r12
addi r31,r31,#−4
lw r1,(r31)
sgt r1,r1,r5
bnez r1,L44
addi r2,r2,#−4
sw (r2),r5
L41:
addi r3,r3,#4
sleu r1,r3,r7
bnez r1,L42
addi r4,r4,#4
L40:
lw r2,0(r29)
lw r3,4(r29)
lw r4,8(r29)
...
lw r12,40(r29)
lw r31,−8(r30)
add r29,r0,r30
jr r31
lw r30,−4(r30)
```

Quicksort auf dem DLX-Prozessor

Die Laufzeit im Durchschnitt beträgt

$$10A_N + 4B_N + 2C_N + 3D_N + 3E_N + 2S_N + 3N + 6,$$

Schritte, wobei

$$\begin{aligned}
A_N &= \frac{2N - M}{M + 2} \\
B_N &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2} \\
C_N &= N + 1 + 2(N+1)(H_{N+1} - H_{M+2}) \\
D_N &= (N+1)(1 - 2H_{M+1}/(M+2)) \\
E_N &= \frac{1}{6}(N+1)M(M-1)/(M+2) \\
S_N &= (N+1)/(2M+3) - 1.
\end{aligned}$$

und $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$.

O-Notation

Definition

$$O(f) = \left\{ g \colon \mathbf{N} \to \mathbf{R} \;\middle|\; \exists c \in \mathbf{R}^+ \, \exists N \in \mathbf{N} \, \forall n \geq N \colon |g(n)| \leq c|f(n)| \right\}$$
$$\Omega(f) = \left\{ g \colon \mathbf{N} \to \mathbf{R} \;\middle|\; \exists c \in \mathbf{R}^+ \, \exists N \in \mathbf{N} \, \forall n \geq N \colon |g(n)| \geq c|f(n)| \right\}$$
$$\Theta(f) = \left\{ g \colon \mathbf{N} \to \mathbf{R} \;\middle|\; \exists c_1, c_2 \in \mathbf{R}^+ \, \exists N \in \mathbf{N} \, \forall n \geq N \colon c_1|f(n)| \leq |g(n)| \leq c_2|f(n)| \right\}.$$

Informell:

- $g = O(f) \;\approx\;$ g wächst langsamer als f
- $g = \Omega(f) \;\approx\;$ g wächst schneller als f
- $g = \Theta(f) \;\approx\;$ g wächst so schnell wie f

Theorem

Es seien $f, g : \mathbf{N} \to \mathbf{R}$ *zwei Funktionen und* $c \in \mathbf{R}$ *eine Konstante. Dann gilt das folgende:*

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$
7. $f(O(1)) = O(1)$

Theorem

Es seien $f, g : \mathbf{N} \to \mathbf{R}$ *zwei Funktionen und* $c \in \mathbf{R}$ *eine Konstante. Dann gilt das folgende:*

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$
7. $f(O(1)) = O(1)$

Theorem

Es seien $f, g\colon \mathbf{N} \to \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante. Dann gilt das folgende:

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$
7. $f(O(1)) = O(1)$

Theorem

Es seien $f, g\colon \mathbf{N} \to \mathbf{R}$ *zwei Funktionen und* $c \in \mathbf{R}$ *eine Konstante. Dann gilt das folgende:*

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$
7. $f(O(1)) = O(1)$

Theorem

Es seien $f, g\colon \mathbf{N} \to \mathbf{R}$ *zwei Funktionen und* $c \in \mathbf{R}$ *eine Konstante. Dann gilt das folgende:*

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$
7. $f(O(1)) = O(1)$

Theorem

Es seien $f, g\colon \mathbf{N} \to \mathbf{R}$ *zwei Funktionen und* $c \in \mathbf{R}$ *eine Konstante. Dann gilt das folgende:*

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$
7. $f(O(1)) = O(1)$

Theorem

Es seien $f, g\colon \mathbf{N} \to \mathbf{R}$ *zwei Funktionen und* $c \in \mathbf{R}$ *eine Konstante. Dann gilt das folgende:*

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$ *falls* $|f(n)|$ *monoton steigt*
7. $f(O(1)) = O(1)$

Wie beweisen wir eine Gleichung der Form

$$O(f(n)) = O(g(n))$$

oder allgemein eine Gleichung, mit O-Notation auf der linken *und* rechten Seite?

In Wirklichkeit ist es $O(f(n)) \subseteq O(g(n))$.

Wir beweisen:
Für jedes $\hat{f}(n) = O(f(n))$ gilt $\hat{f}(n) = O(g(n))$.

Wie beweisen wir eine Gleichung der Form

$$O(f(n)) = O(g(n))$$

oder allgemein eine Gleichung, mit O-Notation auf der linken *und* rechten Seite?

In Wirklichkeit ist es $O(f(n)) \subseteq O(g(n))$.

Wir beweisen:
Für jedes $\hat{f}(n) = O(f(n))$ gilt $\hat{f}(n) = O(g(n))$.

Wir beweisen $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$:

Beweis.

Sei $\hat{f}(n) = O(f(n))$ und $\hat{g}(n) = O(g(n))$ beliebig.
Dann gilt

$$|\hat{f}(n)| \leq c|f(n)| \text{ und } |\hat{g}(n)| \leq c|g(n)|$$

für ein c und große n.

$$\Rightarrow |\hat{f}(n)| + |\hat{g}(n)| \leq c(|f(n)| + |g(n)|)$$

Daraus folgt $O(|f(n)|) + O(|g(n)|) = O(|f(n) + g(n)|)$.
Es gilt aber $O(f(n)) = O(|f(n)|)$ und $O(g(n)) = O(|g(n)|)$. □

Java

Wir verwenden oft Java für Datenstrukturen und Algorithmen.

Die Vorlesung ist aber von der Programmiersprache unabhängig.

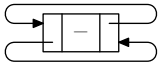
Lernziel sind die einzelnen Algorithmen und Datenstrukturen, nicht ihre Umsetzung in Java.

Für das Verständnis der Vorlesung sind Kenntnisse in einer objektorientierten Sprache notwendig.

Übersicht

Der Konstruktor muß den Listenkopf head erzeugen.
Der Vorgänger und Nachfolger von head ist head selbst.

Der Konstruktor muß den Listenkopf head erzeugen.
Der Vorgänger und Nachfolger von head ist head selbst.



Java

```
public class ADList⟨K, D⟩ extends AbstractMap⟨K, D⟩ {
  private Listnode⟨K, D⟩ head;
  public ADList() {
    head = new Listnode⟨K, D⟩(null, null);
    head.pred = head;
    head.succ = head;
  }
```

Java

```
public class Listnode⟨K, D⟩ {
    K key;
    D data;
    Listnode⟨K, D⟩ pred, succ;
    Listnode(K k, D d) {
        key = k; data = d; pred = null; succ = null; }
    void delete() {
        pred.succ = succ; succ.pred = pred; }
    void copy(Listnode⟨K, D⟩ n) {
        key = n.key; data = n.data; }
    void append(Listnode⟨K, D⟩ newnode) {
        newnode.succ = succ; newnode.pred = this;
        succ.pred = newnode; succ = newnode; }
```

Java

```
public void append(K k, D d) {
    head.pred.append(new Listnode<K, D>(k, d));
}
```

Java

```
public void prepend(K k, D d) {
    head.append(new Listnode<K, D>(k, d));
}
```

Java

```
public void delete(K k) {
  Listnode<K, D> n = findnode(k);
  if(n ≠ null) n.delete();
}
```