Effiziente Algorithmen und Datenstrukturen I

Kapitel 1: Einleitung

Christian Scheideler

WS 2008

Übersicht

- Eingabekodierung
- Asymptotische Notation
- Maschinenmodelle
- Pseudocode
- Laufzeitanalyse
- Einige Beispiele

Effizienzmessung

- Ziel: Effiziente Algorithmen und Datenstrukturen

- Maß für Effizienz:
  - Algorithmus: Aufwand bzgl. Eingabegröße (Speicheraufwand für Eingabe)
  - Datenstruktur:
    Aufwand einer Operation bzgl. Datenstrukturgröße (Anzahl Elemente in Datenstruktur) bzw.
    Länge der Anfragesequenz auf leerer DS

Eingabekodierung

Was ist die Größe einer Eingabe?

Speicherplatz in Bits oder Wörtern, aber Vorsicht bei Kodierung!

Beispiel: Primfaktorisierung.

Gegeben: Zahl $x \in IN$

Gesucht: Primfaktoren von x, d.h. Primzahlen $p_1,..,p_k$ mit $x = \Pi_i \, p_i^{e_i}$

Bekannt als hartes Problem (wichtig für RSA!)

Eingabekodierung

Trivialer Algorithmus:
teste von $y=2$ bis x alle Zahlen, ob diese x teilen, und wenn ja, bestimme Rest von x

Laufzeit: $\sim x$ Rechenschritte (falls Teilertest und Division konstante Zeit benötigen)

- Unäre Kodierung von x (x Nullen als Eingabe): Laufzeit linear (in Eingabegröße)
- Binäre Kodierung von x ($\sim\log x$ Bits): Laufzeit exponentiell (in Eingabegröße)

Binäre Kodierung ergibt korrekte Laufzeitaussage.

Eingabekodierung

Eingabegrößen:

- Größe von Zahlen: binäre Kodierung
- Größe von Mengen/Folgen von Zahlen: oft lediglich Anzahl Elemente

Beispiel: Sortierung

Gegeben: Folge von Zahlen $a_1,\ldots,a_n \in \mathbb{N}$

Gesucht: sortierte Folge der Zahlen

Größe der Eingabe: n

Effizienzmessung

- I: Menge der Eingabeinstanzen
- $T: I \rightarrow IN$: Laufzeit des Algorithmus für Instanz I
- I_n: Menge der Instanzen der Größe n.

Wir interessieren uns für folgende Fälle:

- Worst case: $t(n)=\max\{T(i): i \in I_n\}$
- Best case: $t(n)=\min\{T(i): i \in I_n\}$
- Average case: $t(n) = 1/|I_n| \sum_{i \in I_n} T(i)$

Am interessantesten ist worst case.

Effizienzmessung

Warum worst case?

- “typischer Fall” schwer zu greifen, average case ist nicht unbedingt gutes Maß
- liefert Garantien für die Effizienz des Algorithmus (wichtig für Robustheit)

Exakte Formeln für t(n) sehr aufwändig!

Einfacher: asymptotisches Wachstum

Asymptotische Notation

Intuitiv: Zwei Funktionen $f(n)$ und $g(n)$ haben dasselbe Wachstumsverhalten, falls es Konstanten c und d gibt mit $c<f(n)/g(n)<d$ und $c<g(n)/f(n)<d$ für alle genügend große n.

Beispiel: n^2, $5n^2-7n$ und $n^2/10+100n$ haben dasselbe Wachstumsverhalten, da z.B.

$1/5<(5n^2-7n)/n^2<5$ und $1/5<n^2/(5n^2-7n)<5$

für alle $n\geq 2$.

Asymptotische Notation

Warum reichen genügend große n?

Ziel: Verfahren, die auch für große Instanzen noch effizient sind (d.h. sie skalieren gut).

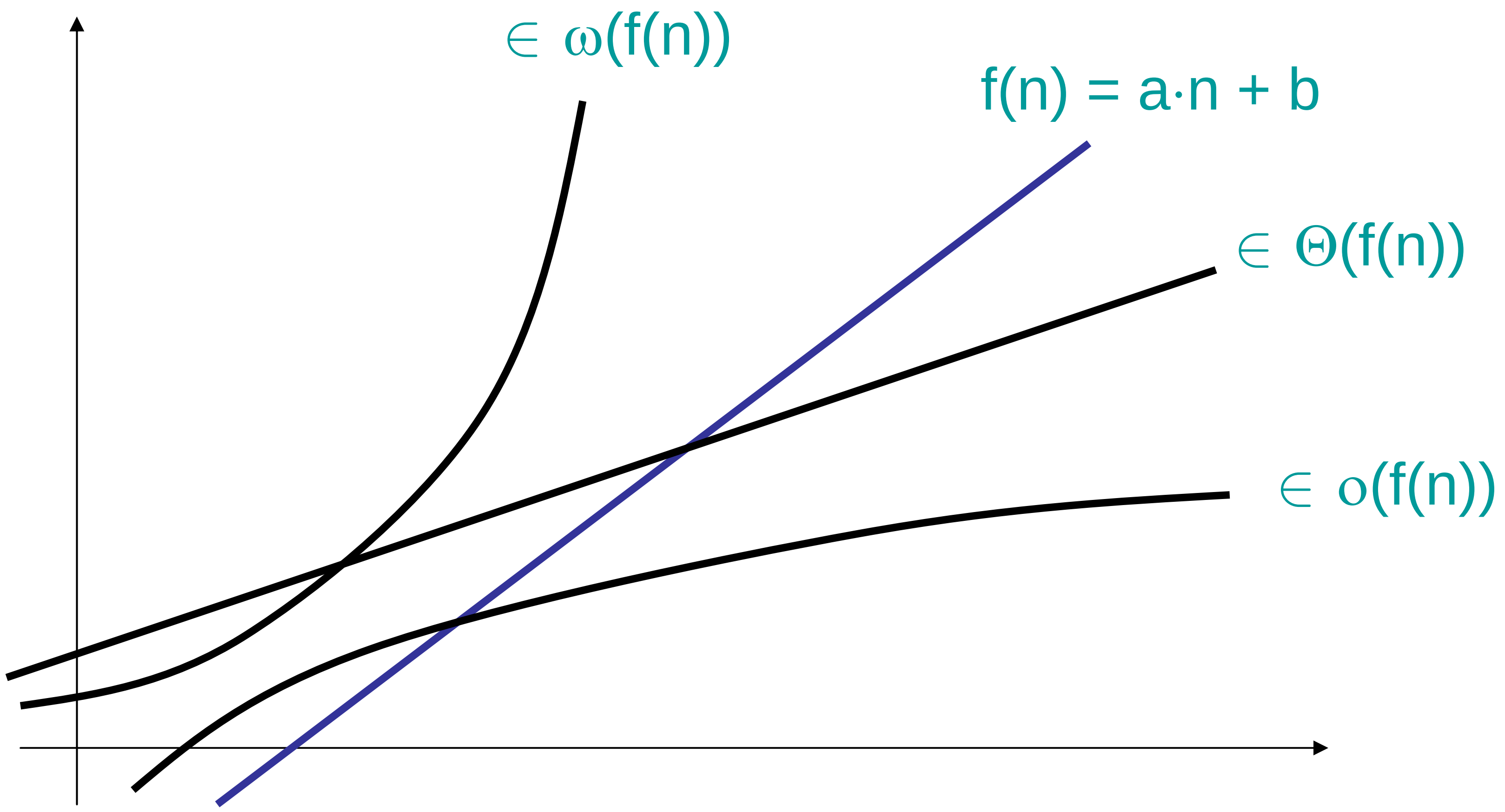
Folgende Mengen formalisieren asymptotisches Verhalten:

- $O(f(n))=\{ g(n) \mid \exists\, c>0\ \exists\, n_0>0\ \forall\, n>n_0: g(n) \leq c\cdot f(n) \}$
- $\Omega(f(n))=\{ g(n) \mid \exists\, c>0\ \exists\, n_0>0\ \forall\, n>n_0: g(n) \geq c\cdot f(n) \}$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- $o(f(n))=\{ g(n) \mid \forall\, c>0\ \exists\, n_0>0\ \forall\, n>n_0: g(n) \leq c\cdot f(n) \}$
- $\omega(f(n))=\{ g(n) \mid \forall\, c>0\ \exists\, n_0>0\ \forall\, n>n_0: g(n) \geq c\cdot f(n) \}$

Nur Funktionen f(n) (bzw. g(n)) mit $\exists\, N>0\ \forall\, n>N: f(n) > 0$!

Diese sollen schließlich Zeit-/Speicherschranken sein.

Asymptotische Notation



Beispiele

- n^2, $5n^2$-$7n$, $n^2/10 + 100n \in O(n^2)$
- $n \log n \in \Omega(n)$, $n^3 \in \Omega(n^2)$
- $\log n \in o(n)$, $n^3 \in o(2^n)$
- $n^5 \in \omega(n^3)$, $2^{2n} \in \omega(2^n)$

Asymptotische Notation

O-Notation auch als Platzhalter für eine Funktion:

- statt $g(n) \in O(f(n))$ schreiben wir auch $g(n) = O(f(n))$
- Für $f(n)+g(n)$ mit $g(n)=o(h(n))$ schreiben wir auch $f(n)+g(n) = f(n)+o(h(n))$
- Statt $O(f(n)) \subseteq O(g(n))$ schreiben wir auch $O(f(n)) = O(g(n))$

Beispiel: $n^3+n = n^3 + o(n^3) = (1+o(1))n^3 = O(n^3)$

O-Notationsgleichungen sollten nur von links nach rechts verstanden werden!

Rechenregeln für O-Notation

Lemma 1.1: Sei $p(n) = \sum_{i=0}^{k} a_i \cdot n^i$ mit $a_k>0$. Dann ist $p(n) \in \Theta(n^k)$.

Beweis:

Zu zeigen: $p(n) \in O(n^k)$ und $p(n) \in \Omega(n^k)$.

$p(n) \in O(n^k)$: Für $n \geq 1$ gilt

$$p(n) \leq \sum_{i=0}^{k} |a_i|\, n^i \leq n^k \sum_{i=0}^{k} |a_i|$$

Also ist Definition von $O()$ mit $c=\sum_{i=0}^{k} |a_i|$ und $n_0=1$ erfüllt.

$p(n) \in \Omega(n^k)$: Für $n>2kA/a_k$ mit $A=\max_i |a_i|$ gilt

$$p(n) \geq a_k\, n^k - \sum_{i=0}^{k-1} A\, n^i \geq a_k\, n^k - k \cdot A\, n^{k-1} \geq a_k\, n^k/2$$

Also ist Definition von $\Omega()$ mit $c=a_k/2$ und $n_0=2kA/a_k$ erfüllt.

Rechenregeln für O-Notation

Nur Funktionen f(n) mit $\exists$ N>0 $\forall$ n>N: f(n)>0 !

Lemma 1.2:

(a) $c \cdot f(n) \in \Theta(f(n))$ für jede Konstante $c>0$

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

(c) $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

(d) $O(f(n)+g(n)) = O(f(n))$ falls $g(n) \in O(f(n))$

Ausdrücke auch korrekt für Ω statt O.

Vorsicht bei induktiver Anwendung von (d)!

Rechenregeln für O-Notation

Behauptung: $\sum_{i=1}^{n} i = O(n)$

"Beweis": Sei $f(n) = n+f(n-1)$ und $f(1)=1$.

Induktionsanfang: $f(1)=O(1)$.

Induktionsschluss: $f(n-1)=O(n-1)$ gezeigt.

Dann gilt:

$$f(n) = n + f(n-1) = n + O(n-1) = O(n+(n-1)) = O(n)$$

Also ist $f(n) = \sum_{i=1}^{n} i = O(n)$ natürlich falsch!

Also Vorsicht mit (d) in Induktionsbeweisen!

Rechenregeln für O-Notation

Lemma 1.3: Seien f und g stetig und differenzierbar. Dann gilt:

(a) Falls $f'(n) = O(g'(n))$, dann auch $f(n)=O(g(n))$

(b) Falls $f'(n) = \Omega(g'(n))$, dann auch $f(n)=\Omega(g(n))$

(c) Falls $f'(n) = o(g'(n))$, dann auch $f(n)=o(g(n))$

(d) Falls $f'(n) = \omega(g'(n))$, dann auch $f(n)=\omega(g(n))$

Der Umkehrschluss gilt im Allg. nicht!

Rechenregeln für O-Notation

Lemma 1.4: Sei h eine Funktion mit $h(n)>0$ für alle $n>N$ für ein N. Dann gilt:

(a) Falls $f(n) = O(g(n))$, dann ist auch $f(n)+h(n) = O(g(n)+h(n))$

(b) Falls $f(n) = O(g(n))$, dann ist auch $f(n)\cdot h(n) = O(g(n)\cdot h(n))$

Folgt aus Lemma 1.2.

Beispiel: $\log n=O(n) \Rightarrow n \log n = O(n^2)$

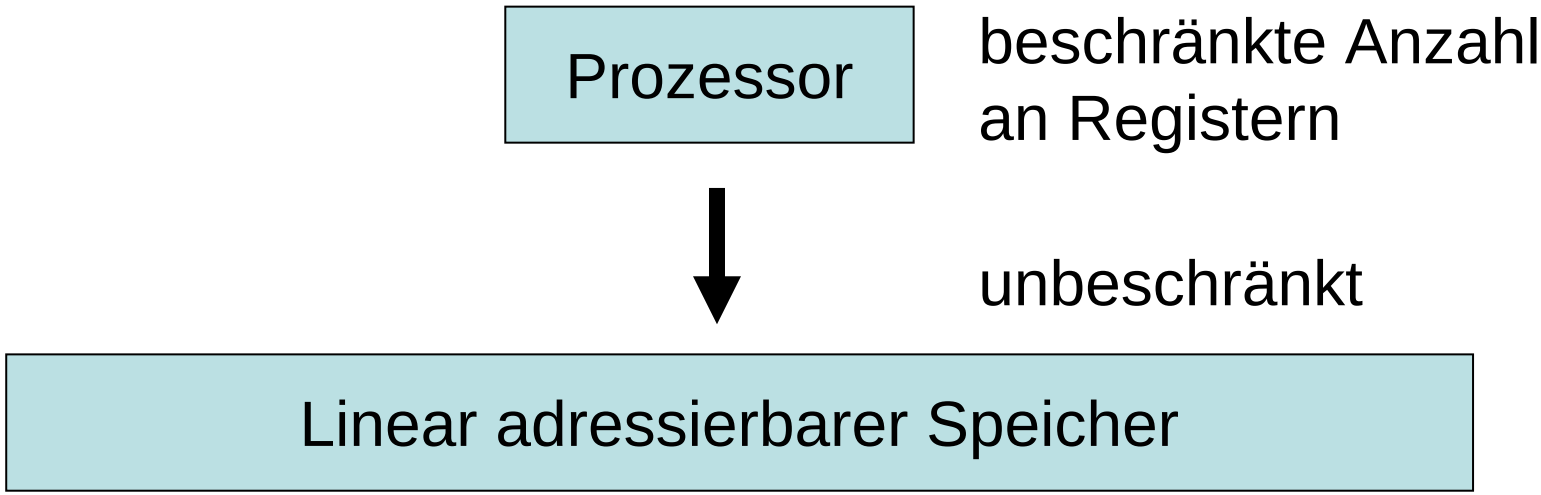
Rechenbeispiele

- Lemma 1.1:
  $n^3-3n^2+2n = O(n^3)$
- Lemma 1.1 und 1.2:
  $\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n} i) = O(n^2/2+n/2) = O(n^2)$
- Lemma 1.3:
  $1/n = O(1)$ und $(\log n)' = 1/n$, also ist
  $(\log n)' = O(n')$ und damit $\log n = O(n)$
- Lemma 1.4:
  aus $\log n = O(n)$ folgt $n \log n = O(n^2)$

Maschinenmodell

Was ist eigentlich ein Rechenschritt?

1945 entwaft John von Neumann die RAM (random access machine).



Maschinenmodell

Speicher:

- Unendlich viele Speicherzellen s[0],s[1],s[2],…
- Speicherzellen können Daten oder Befehle speichern
- Jede Speicherzelle kann eine polynomiell in n (Eingabegröße) beschränkte Zahl speichern (dafür $O(\log n)$ Bits)

Linear adressierbarer Speicher

Maschinenmodell

Prozessor:

- Beschränkte Anzahl an Registern $R_1,\dots,R_k$
- Instruktionszeiger zum nächsten Befehl im Speicher
- Befehlssatz (jede Instruktion eine Zeiteinheit):
  $R_i{:=}s[R_j]$: lädt Inhalt von $s[R_j]$ in R_i
  $s[R_j]{:=}R_i$: speichert Inhalt von R_i in $s[R_j]$
  $R_i{:=}c$ für eine Konstante c
  $R_i{:=}R_j \circ R_k$: binäre Rechenoperation
  $\circ \in \{+, -, \cdot, \oplus,$ div, mod, $\wedge, \vee, \dots\}$ oder
  $\circ \in \{<, \leq, =, \geq, >\}$: 1: wahr, 0: falsch
  $R_i := \circ\, R_j$: unäre Rechenoperation, $\circ \in \{-, \neg\}$
  JZ x, R_i : if $R_i{=}0$ then jump to memory pos x

Maschinenmodell

RAM-Modell:

- Grundlage für die ersten Computer
- Prinzip gilt auch heute

Aber: exponentielle Leistungssteigerungen haben Speicherhierarchien und Multicore-Prozessoren eingeführt, für die das RAM-Modell angepasst werden muss.

Herausforderungen an Algorithm Engineering!

Weitere Maschinenmodelle

Speicherhierarchie:

Register

L1-Cache

L2-Cache

Hauptspeicher

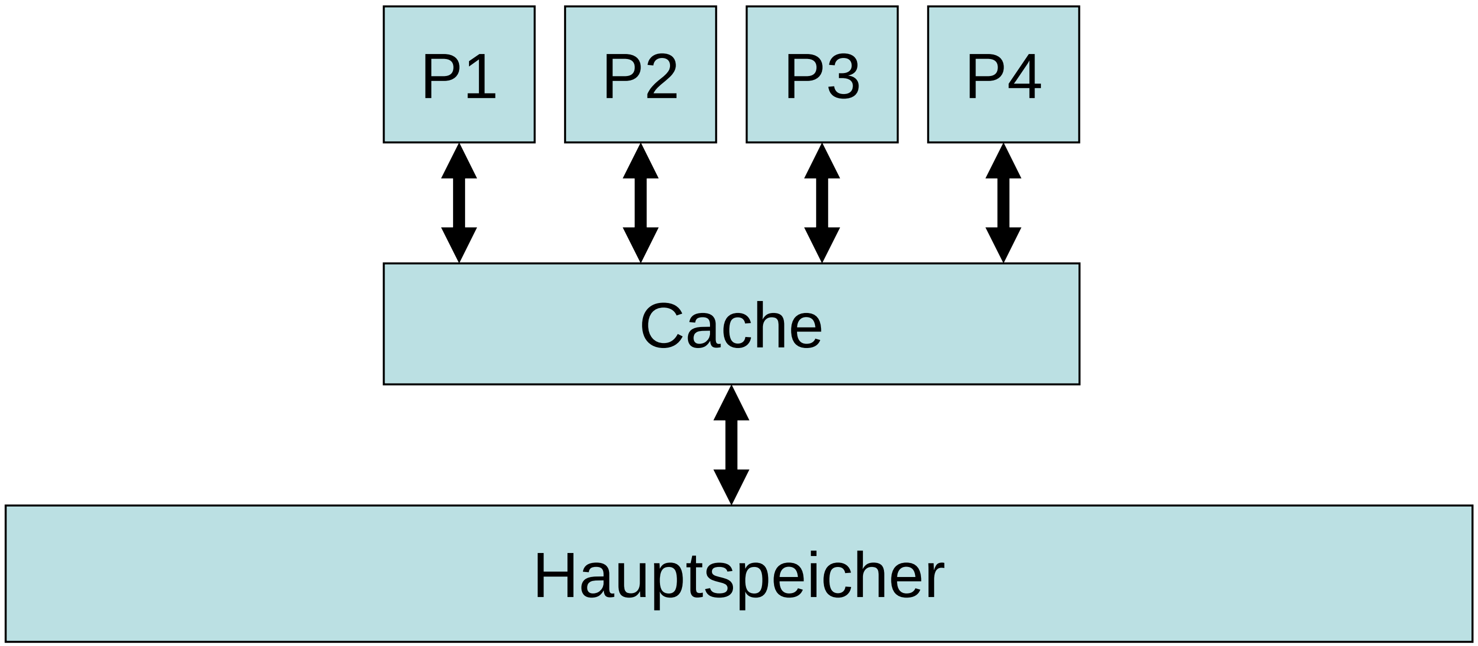
Festplatte

schneller

langsamer

Weitere Maschinenmodelle

Multicore-Prozessoren:



Weitere Maschinenmodelle

- Boolesche Schaltkreise (Hardware)
- Turingmaschine (Komplexitätstheorie)
- Quantencomputer (Kryptographie)
- DNA-Computer
- …

Pseudo-Code

Maschinencode sehr umständlich.

Besser: Pseudo-Code oder Programmiersprache.

Variablendeklarationen:

v: T : Variable v vom Typ T

v=x: T : wird vorinitialisiert mit Wert x

Variablentypen:

- integer, boolean, char
- Pointer to T: Zeiger auf Element vom Typ T
- Array[i..j] of T: Feld von Elementen von i bis j vom Typ T

Pseudo-Code

Allokation und Deallokation von Speicher:

- v:= allocate Array[1..n] of T
- dispose v

Befehlssatz: (C: Bedingung, I,J: Anweisungen)

- v:=A : v erhält Ergebnis von Ausdruck A
- if C then I else J
- repeat I until C,   while C do I
- for v:=a to e do I
- foreach $e \in S$ do I
- return v

Laufzeitanalyse

Was wissen wir?

- O-Kalkül ($O(f(n)), \Omega(f(n)), \Theta(f(n)), \ldots$)
- RAM-Modell
  (load, store, jump,…)
- Pseudo-Code
  (if then else, while do, allocate/dispose,…)

Wie analysieren wir damit Programme?

Laufzeitanalyse

Berechnung der worst-case Laufzeit:

- T(I) sei worst-case Laufzeit für Instruktion I
- T(el. Zuweisung) = O(1), T(el. Vergleich) = O(1)
- T(return x) = O(1)
- T(allocate/dispose) = O(1) (nicht offensichtlich!)
- T(I;I´) = T(I) + T(I´)
- T(if C then I else I´) = O(T(C) + max{T(I),T(I´)})
- T(for i:=a to b do I) = $O(\sum_{i=a}^{b} T(I))$
- T(repeat I until C) = $O(\sum_{i=1}^{k} (T(C)+T(I)))$
  (k: Anzahl Iterationen)
- T(while C do I) = $O(\sum_{i=1}^{k} (T(C)+T(I)))$

Laufzeitanalyse

Vorsicht mit der Annahme
$T(\text{el. Zuweisung/Vergleich}) = O(1)$

Beispiel:
Eingabe: $n \in \mathbb{N}$
$x:=2$
for $i:=2$ to n do $x:=x\cdot x$
return x

Am Ende ist $x=2^{2\wedge(n-1)}$, benötigt also 2^{n-1} Bits, was 2^{n-1} Zeit für die Ausgabe benötigt. Bei Annahme oben wäre Zeitaufwand für Ausgabe nur $O(1)$!

Beipiel: Vorzeichenausgabe

Gegeben: Zahl $x \in IR$

Algorithmus Signum(x):

if $x<0$ then return -1

If $x>0$ then return 1

Return 0

Wir wissen:

$T(x<0) = O(1)$

$T(\text{return } -1) = O(1)$

$T(\text{if } B \text{ then } I) = O(T(B)+ T(I))$

Also ist $T(\text{if } x<0 \text{ then return } -1) = O(1+1) = O(1)$

Beipiel: Vorzeichenausgabe

Gegeben: Zahl $x \in IR$

Algorithmus Signum(x):

if $x<0$ then return -1	$O(1)$
if $x>0$ then return 1	$O(1)$
return 0	$O(1)$

Gesamtlaufzeit: $O(1+1+1)=O(1)$

Beispiel: Minimumsuche

Gegeben: Zahlenfolge in A[1],…,A[n]

Minimum Algorithmus:

min := ∞	$O(1)$
for i:=1 to n do	$O(\sum_{i=1}^{n} T(I))$
if A[i]<min then min:=A[i]	$O(1)$
return min	$O(1)$

Laufzeit: $O(1 + (\sum_{i=1}^{n} 1) + 1) = O(n)$

Beispiel: Sortieren

Gegeben: Zahlenfolge in A[1],…,A[n]

Bubblesort Algorithmus:

```
for i:=1 to n-1 do
   for j:= n-1 downto i do
      if A[j]>A[j+1] then
         x:=A[j]
         A[j]:=A[j+1]
         A[j+1]:=x
```

$O(\sum_{i=1}^{n-1} T(I))$

$O(\sum_{j=i}^{n-1} T(I))$

$O(1 + T(I))$

$O(1)$

$O(1)$

$O(1)$

Laufzeit: $O(\sum_{i=1}^{n-1} \sum_{j=i}^{n-1} 1)$

Beispiel: Sortieren

Gegeben: Zahlenfolge in A[1],…,A[n]

Bubblesort Algorithmus:

```
for i:=1 to n-1 do
  for j:= n-1 downto i do
    if A[j]>A[j+1] then
      x:=A[j]
      A[j]:=A[j+1]
      A[j+1]:=x
```

$$\sum_{i=1}^{n-1} \sum_{j=i}^{n-1} 1$$
$$= \sum_{i=1}^{n-1} (n-i)$$
$$= \sum_{i=1}^{n-1} i$$
$$= n(n-1)/2$$
$$= O(n^2)$$

Beispiel: Binäre Suche

Gegeben: Zahl x und ein sortiertes Array A[1],…,A[n]

Binäre Suche Algorithmus:

l:=1; r:=n	$O(1)$
while l < r do	$O(\sum_{i=1}^{k} T(I))$
m:=(r+l) div 2	$O(1)$
if A[m] = x then return m	$O(1)$
if A[m] < x then l:=m+1	$O(1)$
else r:=m-1	$O(1)$
return l	$O(1)$

$O(\sum_{i=1}^{k} 1) = O(k)$

Beispiel: Binäre Suche

Gegeben: Zahl x und ein sortiertes Array $A[1],\ldots,A[n]$

Binäre Suche Algorithmus:

```
l:=1; r:=n
while l < r do
   m:=(r+l) div 2
   if A[m] = x then return m
   if A[m] < x then l:=m+1
                else r:=m-1
return l
```

$O(\sum_{i=1}^{k} 1) = O(k)$

Was ist k ?? $\rightarrow$ Zeuge

$s_i = (r-l+1)$ in Iteration i

$s_1 = n$, $s_{i+1} <= s_i/2$

$s_i < 1$: fertig

Also ist $k <= \log n + 1$

Beispiel: Bresenham Algorithmus

(x,y):=(0,R)	O(1)
F:=1-R	O(1)
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)	O(1)
while x<y do	$O(\sum_{i=1}^{k} T(I))$
x:=x+1	
if F<0 then	
F:=F+2·x - 1	alles
else	O(1)
F:=F+2·(x-y)	
y:=y-1	
plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)	
plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)	

$O(\sum_{i=1}^{k} 1) = O(k)$

Beispiel: Bresenham Algorithmus

```
(x,y):=(0,R)
F:=1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x:=x+1
    if F<0 then
        F:=F+2·x - 1
    else
        F:=F+2·(x-y)
        y:=y-1
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

Zeuge:

$\phi(x,y) = y-x$

Monotonie: verringert sich um >=1 pro while-Runde
Beschränktheit: while-Bed.

Beispiel: Bresenham Algorithmus

```
(x,y):=(0,R)
F:=1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x:=x+1
    if F<0 then
        F:=F+2·x - 1
    else
        F:=F+2·(x-y)
        y:=y-1
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

Zeuge:

$\phi(x,y) = y-x$

Anzahl Runden:

$\phi_0(x,y) = R$, $\phi(x,y)>0$

$\rightarrow$ maximal R Runden

Beispiel: Fakultät

Gegeben: natürliche Zahl n

Algorithmus Fakultät(n):

if $n=1$ then return 1 $\quad$ $O(1)$

$\quad$ else return $n \cdot$ Fakultät($n-1$) $\quad$ $O(1 + ??)$

Laufzeit:

- $T(n)$: Laufzeit von Fakultät(n)
- $T(n) = T(n-1) + O(1)$, $T(1) = O(1)$

Beispiel: Binäre Suche

Eingabe: $x \in IN$ und $A[1],\dots,A[n]$ für ein $n \in IN$

Aufruf mit BinSuche(x,1,n).

Algorithmus BinSuche(x,l,r)

```
if l>=r then return A[l]
m:=(l+r) div 2
if A[m]=x then return x
if A[m]<x then return BinSuche(x,m+1,r)
if A[m]>x then return BinSuche(x,l,m-1)
```

Laufzeit:

- $T(n)$: worst-case Laufzeit von Binsuche(x,l,r) mit $n=r-l+1$
- $T(n) = T(\lfloor n/2 \rfloor)+O(1)$, $T(1)=O(1)$

Beispiel: Multiplikation

Rekursive Version der Schulmethode:

Gegeben: zwei n-stellige Zahlen a, b zur Basis 2, n gerade

Sei $a=a_1\cdot 2^k+a_0$ und $b=b_1\cdot 2^k+b_0$ mit $k=n/2$

Dann gilt:

$$
\begin{aligned}
a \cdot b &= (a_1\cdot 2^k+a_0) \cdot (b_1\cdot 2^k+b_0) \\
&= a_1\cdot b_1\, 2^{2k} + (a_1\cdot b_0 + a_0\cdot b_1)\, 2^k + a_0\cdot b_0
\end{aligned}
$$

4 rekursive Aufrufe zur Multiplikation $n/2$-stelliger Zahlen

Beispiel: Multiplikation

Annahme: $|a|=|b|=n=2^c$ für ein $c \in IN$

Algorithmus Produkt(a,b):
if $|a|=|b|=1$ then return $a \cdot b$
else
  $k:=|a|/2$
  $a_0:= a$ div 2^k ; $a_1:= a$ mod 2^k
  $b_0:= b$ div 2^k ; $b_1:= b$ mod 2^k
  return Produkt(a_1,b_1)$\cdot 2^{2k}$ + (Produkt(a_1,b_0) + Produkt(a_0,b_1))$\cdot 2^k$ + Produkt(a_0,b_0)

mod, div, + und $\cdot 2^k$ auf n-Bit Zahlen: Aufwand $O(n)$

Zeitaufwand: $T(n) = 4 \cdot T(n/2) + O(n)$, $T(1) = O(1)$

Beispiel: Seltsam

Algorithmus Seltsam(S)
```
if |S|=1 then return x     // S={x}
S':= erste |S|/5 Elemente in S
S'':= letzte 3|S|/4 Elemente in S
a:=Seltsam(S')
b:=Seltsam(S'')
return a+b
```

Laufzeit:

- T(n): Laufzeit für $|S|=n$
- $T(n)=T(n/5)+T(3n/4) + O(1)$, $T(1)=O(1)$

Master-Theorem

Theorem 1.5: Für positive Konstanten a,b,c und d mit $n=b^k$ für ein $k\in\mathbb{N}$ sei

$$T(n) = a \qquad \text{falls } n=1$$

$$T(n) = c \cdot n + d \cdot T(n/b) \qquad \text{falls } n>1$$

Dann gilt

$$T(n) = \Theta(n) \qquad \text{falls } d<b$$

$$T(n) = \Theta(n \log n) \qquad \text{falls } d=b$$

$$T(n) = \Theta(n^{\log_b d}) \qquad \text{falls } d>b$$

Master-Theorem

Behauptung: Sei $n=b^k$ für ein $k\in IN_0$. Dann ist

$$T(n) = cn \sum_{i=0}^{k-1} (d/b)^i + a\cdot d^k$$

Beweis: vollständige Induktion

$k=0$: $T(n) = a$ (wahr)

$k-1 \rightarrow k$:

$$T(n) = cn + d\cdot T(n/b)$$
$$= cn + d\cdot (c(n/b) \sum_{i=0}^{k-2} (d/b)^i + a\cdot d^{k-1})$$
$$= cn \sum_{i=0}^{k-1} (d/b)^i + a\cdot d^k$$

Master-Theorem

Sei $n=b^k$ für ein $k \in \mathbb{N}_0$. Dann ist

$$T(n) = cn \sum_{i=0}^{k-1} (d/b)^i + a \cdot d^k$$

Drei Fälle:

- $d<b$: $\sum_{i=0}^{k-1} (d/b)^i = \Theta(1)$, also $T(n)=\Theta(n)$
- $d=b$: $\sum_{i=0}^{k-1} (d/b)^i = \Theta(k)$, also $T(n)=\Theta(n \cdot k)$ mit $k=\log_b n$
- $d>b$: $\sum_{i=0}^{k-1} (d/b)^i = \Theta((d/b)^k)$, also $T(n)=\Theta(n \cdot (d/b)^k)=\Theta(d^k)=\Theta(n^{\log_b d})$

Beispiele:

- Fakultät:
  $T(n)=T(n-1)+c,\ T(1)=d$
  Lösung: $T(n) = c\cdot(n-1) + d$ (vollst. Ind.)
- BinSuche ($n=2^k$):
  $T(n)=T(n/2)+c,\ T(1)=d$
  Lösung: $T(n) = c \log n + d$ (vollst. Ind.)
- Multiplikation ($n=2^k$):
  $T(n) = 4\cdot T(n/2) + c\cdot n,\ T(1)=d$
  Lösung: $T(n) = \Theta(n^2)$ (Mastertheorem)

Rekursionsauflösung

Verschiedene Verfahren:

- Vollständige Induktion: einfachste Methode für Rekursionsauflösung, falls bereits richtige Vermutung (welche entweder über Abwicklung der Rekursion über mehrere Schritte oder zumindest über die Hypothese $T(n)=n^k$ für ein unbekanntes k näherungsweise bestimmt werden kann)
- Erzeugendenfunktionen
- Charakteristische Polynome
- …

Rekursionsauflösung

Beispiel: Algorithmus Seltsam
$T(n)=T(n/5)+T(3n/4)+c,\ T(1)=d$

Ziel: suche kleinstes k, so dass $T(n)<n^k$.

Für dieses k muss gelten:

$$n^k > (n/5)^k + (3n/4)^k + c$$
$$\Leftrightarrow\ 1 > (1/5)^k + (3/4)^k + c/n^k$$

Also konstantes k gesucht mit $1>(1/5)^k+(3/4)^k$.

Dieses k ist mit binärerer Suche zu ermitteln.

Ergebnis: $k\approx 0.92$

Average Case Laufzeit

Beispiel: Inkrement einer großen Binärzahl, die in A[0],…,A[n-1] gespeichert ist (A[n]=0)

Algorithmus Inc(A):

```
i:=0
while true do
   if A[i]=0 then A[i]:=1; return
   A[i] := 0
   i := i+1
```

Durchschnittliche Laufzeit für Zahl der Länge n?

Average Case Laufzeit

Beispiel: Inkrement einer großen Binärzahl, die in $A[0],\dots,A[n-1]$ gespeichert ist ($A[n]=0$)

Analyse: sei I_n = {n-bit Zahlen}

- Für ½ der Zahlen $(x_{n-1},\dots,x_0) \in I_n$ ist $x_0 = 0$
  $\rightarrow$ 1 Schleifendurchlauf
- Für ¼ der Zahlen $(x_{n-1},\dots,x_0)$ ist $(x_1,x_0) = (0,1)$
  $\rightarrow$ 2 Schleifendurchläufe
- Für $½^i$ der Zahlen ist $(x_i,\dots,x_0) = (0,1,\dots,1)$
  $\rightarrow$ i Schleifendurchläufe

Average Case Laufzeit

Beispiel: Inkrement einer großen Binärzahl, die in $A[0],\dots,A[n-1]$ gespeichert ist ($A[n]=0$)

Analyse: sei I_n = {n-bit Zahlen}

Average case Laufzeit $T(n)$:

$$
\begin{aligned}
T(n) &= (1/|I_n|) \sum_{i \in I_n} T(i) \\
&= (1/|I_n|) \sum_{i=1}^{n} (|I_n|/2^i)\, O(i) \\
&= \sum_{i=1}^{n} O(i/2^i) \\
&= O\left(\sum_{i=1}^{n} i/2^i \right) = O(1)
\end{aligned}
$$

Zahlen

Durchläufe

Average Case Laufzeit

Problem: Average case Laufzeit mag nicht korrekt die “gewöhnliche” durchschnittliche Laufzeit wiedergeben, da tatsächliche Eingabeverteilung stark von uniformer Verteilung abweichen kann.

Eingabeverteilung bekannt:

korrekte durchschnittl. Laufzeit berechenbar, aber oft schwierig

Erwartete Laufzeit

Eingabeverteilung folgt Wahrscheinlichkeits-verteilung. Jede Eingabe wird zufällig und unabhängig von früheren Eingaben gemäß Wahrscheinlichkeitsverteilung gewählt:

Average Case Laufzeit $\rightarrow$ Erwartete Laufzeit

Erwartete Laufzeit

Beispiel: Suche in unsortierter Liste

$\rightarrow \boxed{16} \rightarrow \boxed{9} \rightarrow \boxed{4} \rightarrow \boxed{23} \rightarrow \boxed{18} \rightarrow$

Heuristic: Move-to-Front

Nach jeder erfolgreichen Suche, füge das gefundene Element vorne in die Liste ein.

D.h. Search(4) ergibt

$\rightarrow \boxed{4} \rightarrow \boxed{16} \rightarrow \boxed{9} \rightarrow \boxed{23} \rightarrow \boxed{18} \rightarrow$

Erwartete Laufzeit

Analyse:

- I_n: n Search-Operationen
- s_i: Position von Element i in Liste (1: vorne)
- p_i: Wahrscheinlichkeit für Search(i)

Erwartete Laufzeit für Search(i) bei zufälligem i:

$$O(\sum_i p_i s_i)$$

Erwartete Laufzeit $T(n)$ bei statischer Liste:

$$T(n) = \sum_{c \in I_n} \underbrace{p(c)}_{\text{Wkeit für Instanz c}} \; \underbrace{t(c)}_{\text{Laufzeit für Instanz c}} = O(\sum_{j=1}^n \sum_i p_i s_i)$$

Erwartete Laufzeit

Was ist die optimale Anordnung?

Lemma 1.6: Eine Anordnung ist optimal, wenn für alle Elemente i,j mit $s_i<s_j$ gilt $p_i \geq p_j$.

O.B.d.A. sei $p_1>=p_2>=\ldots>=p_m$ (m:# Elemente)

- Opimale Anordnung: $s_i = i$
- Optimale erwartete Laufzeit: $Opt = \sum_i p_i\, i$

Theorem 1.7: Erwartete Laufzeit von Move-to-Front ist max. $2 \cdot Opt$ für genügend großes n.

Beweis von Theorem 2.7

Betrachte zwei feste Elemente i und j

- t: aktuelle Operation
- t_0: letzte Suchoperation auf i oder j

- $\Pr[A \mid (A \vee B)] = \Pr[A] / \Pr[A \vee B]$
- $\Pr[$Search(j) bei $t_0] = p_j / (p_i+p_j)$

Beweis von Theorem 2.7

Betrachte festes Element i

- Zufallsvariable $X_j \in \{0,1\}$:
  $X_j = 1 \Leftrightarrow j$ vor i in der Liste
- Listenposition von i: $1+\sum_j X_j$
- $E[X_j] = 0\cdot \Pr[X_j=0] + 1\cdot \Pr[X_j=1]$
  $= \Pr[$Search(j) später als $i] = p_j/(p_i+p_j)$
- $E[$Listenpos$] = E[1+\sum_j X_j] = 1+\sum_j E[X_j]$
  $= 1+\sum_j p_j/(p_i+p_j)$

Beweis von Theorem 2.7

Erwartete Laufzeit für Operation t für genügend großes t:

$$
\begin{aligned}
T_{MTF} &= \sum_i p_i \,(1+\sum_{j\neq i} p_j/(p_i+p_j)\,) \\
&= \sum_i p_i \;+\; \sum_{i\neq j} (p_i\, p_j)/(p_i+p_j) \\
&= \sum_i p_i \;+ 2 \sum_{j<i} (p_i\, p_j)/(p_i+p_j) \\
&= \sum_i p_i \,(1 + 2 \sum_{j<i} p_j/(p_i+p_j)\,) \\
&< \sum_i p_i \,(1 + 2 \sum_{j<i} 1\,) \\
&= \sum_i p_i\, 2i = 2 \cdot \mathrm{Opt}
\end{aligned}
$$

Nächstes Kapitel

Höhere Datenstrukturen.

Wir starten mit Priority Queues.