Grundlagen der Algorithmen und Datenstrukturen

Kapitel 2

Christian Scheideler + Helmut Seidl

SS 2009

Übersicht

- Eingabekodierung
- Asymptotische Notation
- Maschinenmodell
- Java
- Laufzeitanalyse
- Einige Beispiele

Effizienzmessung

Hauptziel: Effiziente Algorithmen

Exakte Spezifikation der Laufzeit eines Algorithmus (bzw. einer DS-Operation):

- I: Menge der Instanzen
- $T:I \rightarrow IN$: Laufzeit des Algorithmus für Instanz

Problem: T SEHR schwer exakt bestimmbar!

Lösung: wir gruppieren Instanzen in Gruppen "ähnlicher Instanzen"

Einfachste Strategie: Gruppiere nach Größe

Eingabekodierung

Aber was ist die Größe einer Instanz?

Speicherplatz in Bits oder Wörtern, aber Vorsicht bei Kodierung!

Beispiel: Primfaktorisierung.

Gegeben: Zahl $x \in \mathbb{N}$

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: ~x Teilbarkeitstests und Divisionen

- Unäre Kodierung von x (x Einsen als Eingabe): Anzahl der Operationen linear (in Eingabegröße)
- Binäre Kodierung von x (~log x Bits): Laufzeit exponentiell (in Eingabegröße)

Binäre Kodierung ergibt korrekte Laufzeitaussage.

Eingabekodierung

Geeignete Eingabekodierungen:

- Größe von Zahlen: binäre Kodierung
- Größe von Mengen/Folgen von Zahlen: oft reicht die Anzahl der 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

Für ein gegebenes Problem sei I_n die 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 der 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 aufwendig!

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).

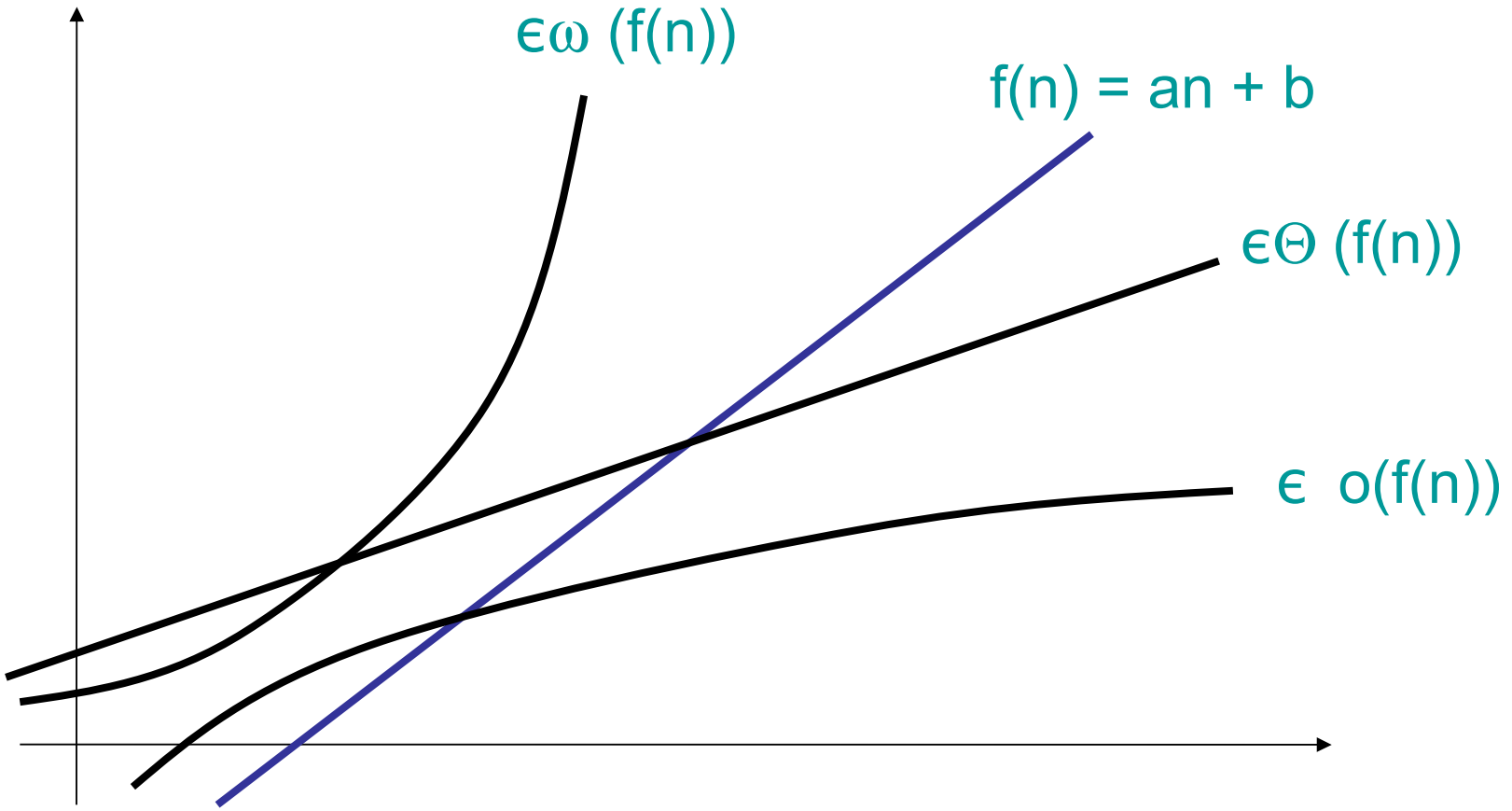
Folgende Mengen formalisieren asymptotisches Verhalten:

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

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 2.1: Sei $p(n) = \sum_{i=0}^{k} a_i n^i$ 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 $n>1$ 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

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

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 2.2:

- c f(n) = Θ (f(n)) für jede Konstante c>0
- O(f(n))+O(g(n)) = O(f(n)+g(n))
- O(f(n)) O(g(n)) = O(f(n) g(n))
- O(f(n)+g(n)) = O(f(n)) falls g(n)=O(f(n))

Ausdrücke auch korrekt für Ω statt O.

Vorsicht bei induktiver Anwendung der Regeln!

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:

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

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

Also Vorsicht mit Induktionsbeweisen!

Rechenregeln für O-Notation

Lemma 2.3: Seien f und g stetig und differenzierbar. Dann gilt:

- Falls $f'(n) = O(g'(n))$, dann auch $f(n)=O(g(n))$
- Falls $f'(n) = \Omega(g'(n))$, dann auch $f(n)=\Omega(g(n))$
- Falls $f'(n) = o(g'(n))$, dann auch $f(n)=o(g(n))$
- Falls $f'(n) = \omega(g'(n))$, dann auch $f(n)=\omega(g(n))$

Der Umkehrschluss gilt im Allg. nicht!

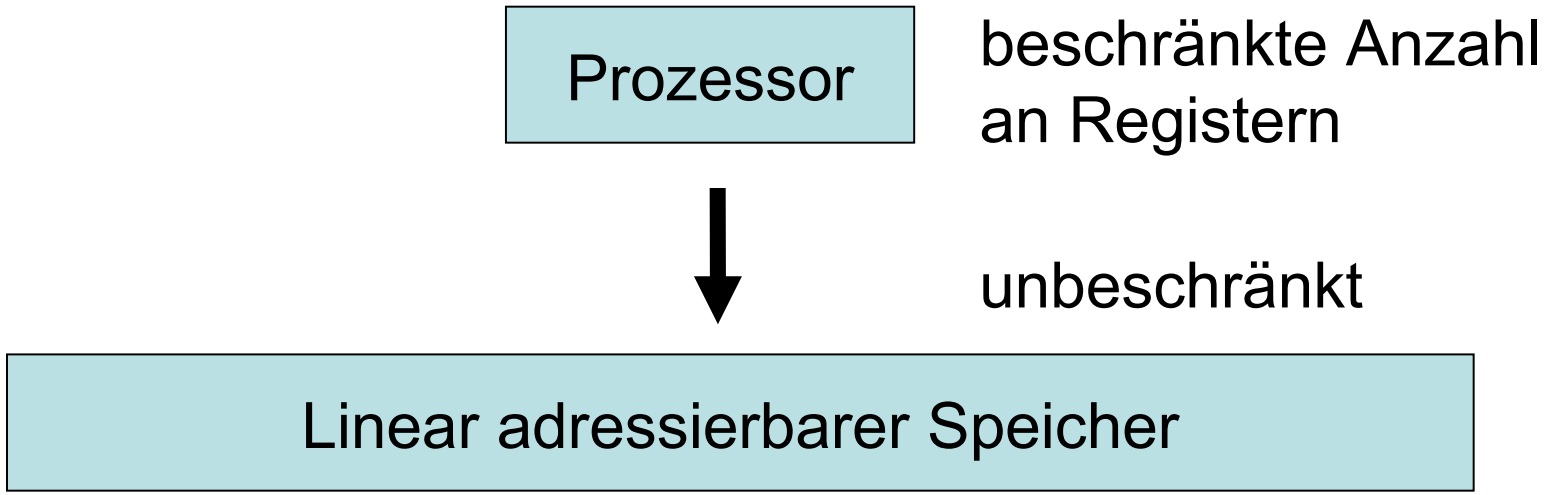
Rechenbeispiele

- Lemma 2.1:
  $n^3-3n^2+2n = O(n^3)$
- Lemma 2.1:
  $O(\sum_{i=1}^{n} i) = O(n^2/2+n/2)=O(n^2)$
- Lemma 2.3:
  $1/n = O(1)$ und $(\log n)' = 1/n$, also ist
  $\log n = O(n)$
- Lemma 2.2:
  aus $\log n = O(n)$ folgt $n \log n = O(n^2)$

Maschinenmodell

Was ist eigentlich ein Rechenschritt?

1945 entwirft 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 $R_1,\ldots,R_k$
- 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
  $R_i:=R_j$ op R_k : binäre Rechenoperation
  op $\in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, \ldots\}$ oder
  op $\in \{<, \leq, =, >, \geq\}$: 1: wahr, 0: falsch
  $R_i :=$ op R_j : unäre Rechenoperation, op $\in \{-, \neg\}$

Maschinenmodell

Prozessor (Forts.):

- Befehlssatz (jede Instruktion eine Zeiteinheit):

  jump x : springe an die Position x

  jumpz x R_i : falls R_i=0 springe an die Position x

  jumpi R_j : springt die Adresse aus R_j an.

- entspricht Assembler-Code einer realen Maschine!

Maschinenmodell

RAM-Modell:

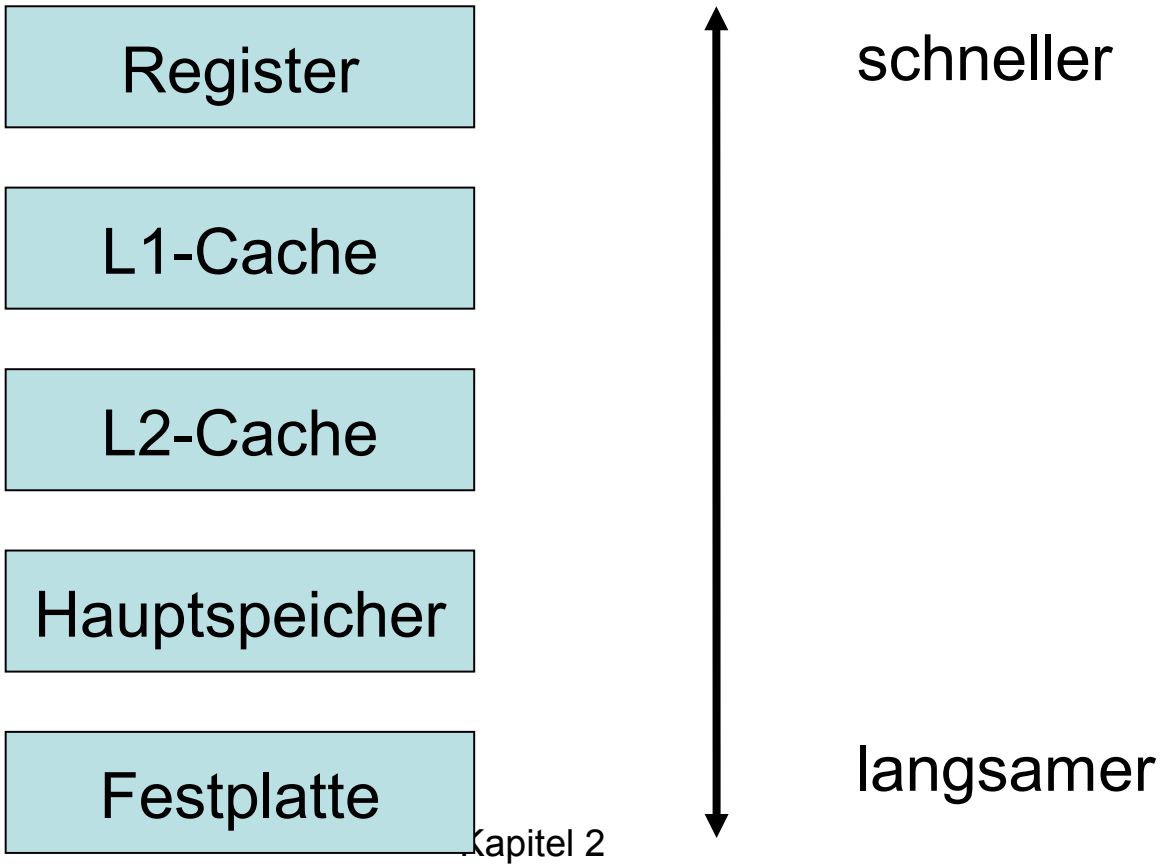
- Grundlage für die ersten Computer
- Prinzip gilt auch heute

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

Herausforderungen an Algorithm Engineering!

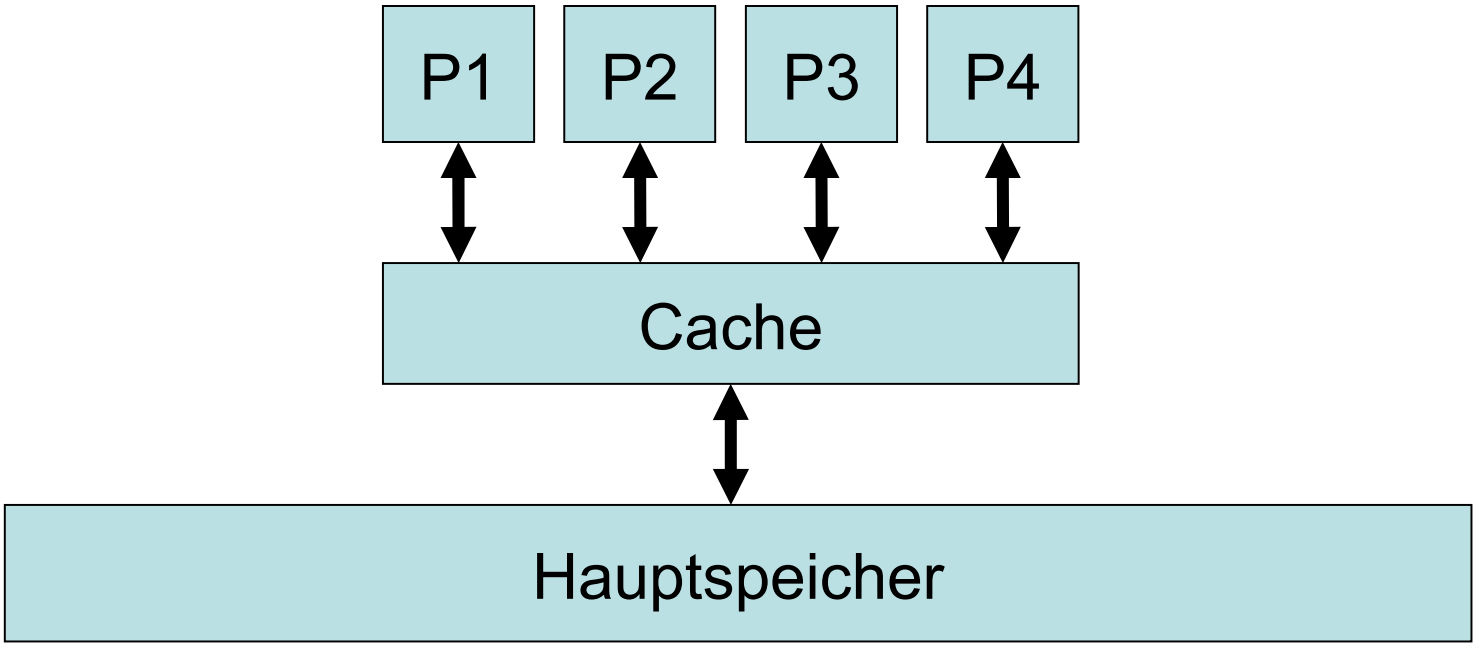
Maschinenmodell

Speicherhierarchie:



Maschinenmodell

Multicore-Prozessoren:



Java

Maschinencode sehr umständlich.

Besser: Programmiersprache wie Java.

Variablendeklarationen:

T v; : Variable v vom Typ T

T v = x; : wird vorinitialisiert mit Wert x

Variablentypen:

- int, boolean, char, double, ...
- Klassen T, Interfaces I,
- T [n]: Feld von Elementen von 0 bis n-1 vom Typ T

Java-Programme

Allokation und Deallokation von Speicherobjekten:

- v = new T(v_1,...,v_k); // implizit wird Konstruktor für T
  // aufgerufen

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

- v=A; Variable v erhält Ergebnis von Ausdruck A
- if (C) I else J
- do I while (C); while (C) I
- for (v=a; v<e; v++) I
- return v;

Laufzeitanalyse

Was wissen wir?

- O-Kalkül ($O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$, …)
- RAM-Modell
  (load, store, jump,…)
- Java
  (if-else, while, new,…)

Wie analysieren wir damit Programme?

Laufzeitanalyse

Berechnung der worst-case Laufzeit:

- T(I) sei worst-case Laufzeit für Konstrukt I
- T(el. Zuweisung) = O(1), T(el. Vergleich) = O(1)
- T(return x) = O(1)
- T(new T(...)) = O(1) + O(T(Konstruktor))
- T(I;I´) = T(I) + T(I´)
- T(if (C) I else I´) = O(T(C) + max{T(I),T(I´)})
- T(for(i=a; i<b; i++) I) = O($\sum_{i=a}^{b-1}$ (1+T(I)))
- T(e.m(...)) = O(1) + T(ss) falls ss Rumpf von m

Beipiel: Vorzeichenausgabe

Gegeben: Zahl $x \in IR$

Algorithmus signum(x):

if (x<0) return -1;

If (x>0) return 1;

return 0;

Wir wissen:

T(x<0) = O(1)

T(return -1) = O(1)

T(if (B) I) = O(T(B)+ T(I))

Also ist T(if (x<0) return -1) = O(1+1) = O(1)

Beipiel: Vorzeichenausgabe

Gegeben: Zahl $x \in IR$

Algorithmus signum(x):

if (x<0) return -1;	O(1)
if (x>0) return 1;	O(1)
return 0;	O(1)

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

Beispiel: Minimumsuche

Gegeben: Zahlenfolge in A[0],…,A[n-1]

Minimum Algorithmus:

```
min = A[0];                          O(1)
for (i= 1;i<n;i++)                   O(∑_{i=1}^{n-1} (1+T(I)))
    if (A[i]<min) min=A[i];          O(1)
return min;                          O(1)
```

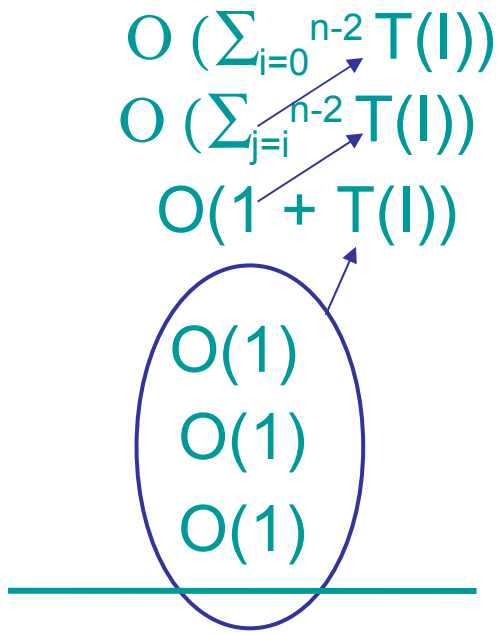
Laufzeit: $O(1 + (\sum_{i=1}^{n-1} 1) + 1) = O(n)$

Beispiel: Sortieren

Gegeben: Zahlenfolge in A[0],…,A[n-1]

Bubblesort Algorithmus:

```
for (i=0; i<n-1; i++)
    for (j=n-2; j≥i; j--)
       if (A[j]>A[j+1]) {
          x=A[j];
          A[j]=A[j+1];
          A[j+1]=x;
          }
```



Laufzeit: $O\left(\sum_{i=0}^{n-2} \sum_{j=i}^{n-2} 1\right)$

Beispiel: Sortieren

Gegeben: Zahlenfolge in A[0],…,A[n-1]

Bubblesort Algorithmus:

```
for (i=0; i<n-1; i++)
    for (j=n-2; j≥i ; j--)
       if (A[j]>A[j+1]) {
          x=A[j];
         A[j]=A[j+1];
         A[j+1]=x;
       }
```

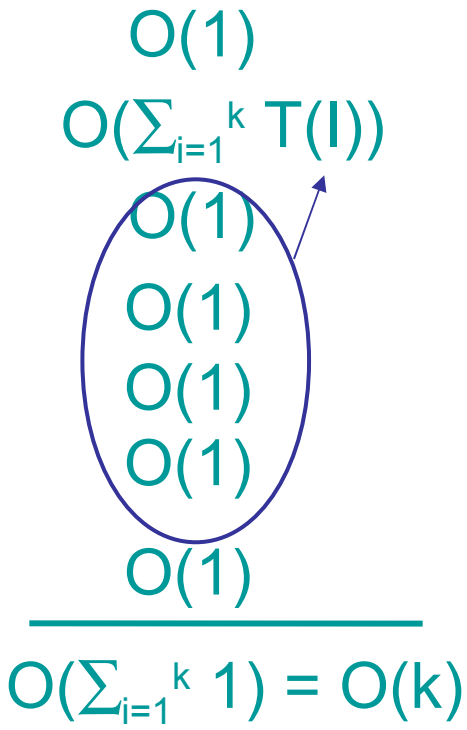
$$\sum_{i=0}^{n-2} \sum_{j=i}^{n-2} 1$$
$$= \sum_{i=0}^{n-2} (n-i-1)$$
$$= \sum_{i=1}^{n-1} i$$
$$= n(n-1)/2$$
$$= O(n^2)$$

Beispiel: Binäre Suche

Gegeben: Zahl x und ein sortiertes Array A[0],…,A[n-1]

Binäre Suche Algorithmus:

```
l=0; r=n-1;
while (l < r) {
    m=(r+l) / 2;
    if (A[m] == x) return m;
    if (A[m] < x) l=m+1;
    else r=m-1;
}
return -1;
```



Beispiel: Binäre Suche

Gegeben: Zahl x und ein sortiertes Array A[0],…,A[n-1]

Binäre Suche Algorithmus:

```
l=0; r=n-1;
while (l ≤ r) {
    m=l+(r-l) / 2;
    if (A[m] == x) return m;
    if (A[m] < x) l=m+1;
    else r=m-1;
    }
return l;
```

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

Was ist k ?? Zeuge:

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

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

$s_i < 1$: fertig

Also ist $k \leq \log n + 1$

Beispiel: Bresenham Algorithmus

```
x = 0; y = R;                                                    O(1)
F = -R+0.25;                                                     O(1)
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0);                    O(1)
while (x<y) {                                                    O(∑_{i=1}^{k} T(I))
    x = x+1;
    F = F+2x-1;                                                  alles
    if (F≥0) {
        y = y-1;                                                 O(1)
        F = F-2y;
    }
    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(∑_{i=1}^{k} 1) = O(k)
```

Beispiel: Bresenham Algorithmus

```
x = 0; y = R;
F = -R+0.25;
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0);
while (x<y) {
    x = x+1;
    F = F+2x-1;
    if (F≥0) {
        y = y-1;
        F = F-2y;
    }
    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:

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

Monotonie: verringert sich um ≥1 pro while-Runde

Beschränktheit: while-Bed.

Beispiel: Bresenham Algorithmus

```
x = 0; y = R;
F = -R+0.25;
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0);
while (x<y) {
    x = x+1;
    F = F+2x-1;
    if (F≥0) {
        y = y-1;
        F = F-2y-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:

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

Anzahl Runden:

$\varphi_0(x,y) = R$, $\varphi(x,y)>0$

$\Rightarrow$ maximal R Runden

Beispiel: Fakultät

Gegeben: natürliche Zahl n

Algorithmus fakultät(n):

```
if (n==1) return 1;                          O(1)
        else return n * fakultät(n-1);       O(1 + ??)
```

Laufzeit:

- T(n): Laufzeit von fakultät(n)
- T(n) = T(n-1) + O(1), T(1) = O(1)
- Wir schließen: T(n) = O(n)

Average Case Laufzeit

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

Algorithmus inc(A):

```
i=0;
while (true) {
    if (A[i]==0) { A[i] =1; return; }
    A[i] = 0;
    i = i+1;
}
```

Durchschnittliche Laufzeit für Zahl der Länge n?

Average Case Laufzeit

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

Analyse: sei I_n = {n-bit Zahlen}

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

Average Case Laufzeit

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

Analyse: sei I_n = {n-bit Zahlen}

Average case Laufzeit T(n):

$$T(n) = (1/|I_n|) \sum_{i \in I_n} T(i)$$

$$= (1/|I_n|) \sum_{i=1}^{n} (|I_n|/2^i)\, O(i)$$

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

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

Zahlen

Durchläufe

Average Case Laufzeit

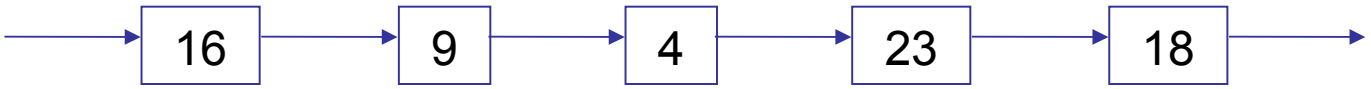
Problem: Average case Laufzeit mag nicht korrekt die “gewöhnliche” durchschnittliche Laufzeit wiedergeben, da tatsächliche Eingabeverteilung stark von uniformer Verteilung abweichen kann.

Wahrscheinlichkeitsverteilung bekannt:

korrekte durchschnittl. Laufzeit berechenbar, aber oft schwierig

Average Case Laufzeit

Beispiel: Suche in unsortierter Liste



Heuristik: Move-to-Front

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

D.h. search(4) ergibt

Average Case 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:

$$T(n) = \sum_{c \in I_n} p(c)\, t(c) = O(\sum_{j=1}^{n} \sum_i p_i s_i)$$

Wkeit für Instanz c

Laufzeit für Instanz c

Average Case Laufzeit

Was ist die optimale Anordnung?

Lemma 2.6: Eine Anordnung ist optimal, wenn für alle Elemente i,j mit $s_i<s_j$ gilt $p_i>p_j$.

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

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

Theorem 2.7: Erwartete Laufzeit von Move-to-Front ist max. 2 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: $1+\sum_j X_j$
- $E[X_j] = 0 \cdot \Pr[X_j=0] + 1 \cdot \Pr[X_j=1]$
  $= \Pr[\text{letzte Suchop auf } j] = p_j/(p_i+p_j)$
- $E[\text{Listenpos}] = E[1+\sum_j X_j] = 1+\sum_j E[X_j]$
  $= 1+\sum_{j \neq i} p_j/(p_i+p_j)$

Beweis von Theorem 2.7

Erwartete Laufzeit für Operation t für genügend großes t:

$$C_{MTF} = \sum_i p_i \, (1+\sum_{j \neq i} p_j/(p_i+p_j) \,)$$

$$= \sum_i (p_i \; + \; \sum_{j \neq i} (p_i \, p_j)/(p_i+p_j))$$

$$= \sum_i (p_i \; + 2 \sum_{j<i} (p_i \, p_j)/(p_i+p_j))$$

$$= \sum_i p_i \, (1 + 2 \sum_{j<i} p_j/(p_i+p_j) \,)$$

$$< \sum_i p_i \, (1 + 2 \sum_{j<i} 1)$$

$$\leq \sum_i p_i \, 2i = 2 \cdot Opt$$

Nächstes Kapitel

Thema: Repräsentation von Sequenzen als Felder und verkettete Listen

- Was ist eine Sequenz?
- Repräsentation als Feld und amortisierte Analyse
- Repräsentation als verkettete Liste
- Stapel (Stacks) und Schlangen (Queues)