Übersicht

3 Effizienz
- Effizienzmaße
- Rechenregeln für $\mathcal{O}$-Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Übersicht

3 Effizienz
- Effizienzmaße
- Rechenregeln für $\mathcal{O}$-Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Effizienzmessung

Ziel:

- Beschreibung der Performance von Algorithmen
- möglichst genau, aber in kurzer und einfacher Form

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

- Menge $\mathcal{I}$ der Instanzen
- Laufzeit des Algorithmus $T : \mathcal{I} \mapsto \mathbb{N}$

Problem: T sehr schwer exakt bestimmbar bzw. beschreibbar

Lösung: Gruppierung der Instanzen (meist nach Größe)

Eingabekodierung

Bei Betrachtung der Länge der Eingabe:

Vorsicht bei der Kodierung!

Beispiel (Primfaktorisierung)

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

Gesucht: Primfaktoren von x (Primzahlen $p_1, \ldots, p_k$ mit $x = \prod_{i=1}^{k} p_i^{e_i}$)

Bekannt als hartes Problem (wichtig für RSA-Verschlüsselung!)

Eingabekodierung - Beispiel Primfaktorisierung

Beispiel (Primfaktorisierung)

Trivialer Algorithmus

Teste von $y = 2$ bis $\lfloor \sqrt{x} \rfloor$ alle Zahlen, ob diese x teilen und wenn ja, dann bestimme wiederholt das Ergebnis der Division bis die Teilung nicht mehr ohne Rest möglich ist

Laufzeit: $\sqrt{x}$ Teilbarkeitstests und höchstens $\log_2 x$ Divisionen

- Unäre Kodierung von x (x Einsen als Eingabe):
  Laufzeit polynomiell bezüglich der Länge der Eingabe
- Binäre Kodierung von x ($\lceil \log_2 x \rceil$ Bits):
  Laufzeit exponentiell bezüglich der Länge der Eingabe

Eingabekodierung

Betrachtete Eingabegröße:

- Größe von Zahlen: Anzahl Bits bei binärer Kodierung
- Größe von Mengen / Folgen: Anzahl Elemente

Beispiel (Sortieren)

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

Gesucht: sortierte Folge der Zahlen

Größe der Eingabe: n

Manchmal Betrachtung von mehr Parametern:

- Größe von Graphen: Anzahl Knoten und Anzahl Kanten

Effizienzmessung

Sei $\mathcal{I}_n$ die Menge der Instanzen der Größe n eines Problems.

Effizienzmaße:

- Worst case:

$$t(n) = \max\{T(i) : i \in \mathcal{I}_n\}$$

- Average case:

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$

- Best case:

$$t(n) = \min\{T(i) : i \in \mathcal{I}_n\}$$

(Wir stellen sicher, dass max und min existieren und dass $\mathcal{I}_n$ endlich ist.)

Vor- und Nachteile der Maße

- worst case:
  liefert Garantie für die Effizienz des Algorithmus,
  evt. aber sehr pessimistische Abschätzung

- average case:
  beschreibt durchschnittliche Laufzeit, aber nicht unbedingt übereinstimmend mit dem “typischen Fall” in der Praxis,
  ggf. Verallgemeinerung mit Wahrscheinlichkeitsverteilung

- best case:
  Vergleich mit worst case liefert Aussage über die Abweichung innerhalb der Instanzen gleicher Größe,
  evt. sehr optimistisch

Exakte Formeln für $t(n)$ sind meist sehr aufwendig bzw. nicht möglich!

$\Rightarrow$ betrachte asymptotisches Wachstum ($n \to \infty$)

Wachstumsrate / -ordnung

- $f(n)$ und $g(n)$ haben gleiche Wachstumsrate, falls für große n das Verhältnis durch Konstanten beschränkt ist:

$$\exists c, d \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad c \leq \frac{f(n)}{g(n)} \leq d$$

- $f(n)$ wächst schneller als $g(n)$, wenn es für alle positiven Konstanten c ein n_0 gibt, ab dem $f(n) \geq c \cdot g(n)$ für $n \geq n_0$ gilt, d.h.,

$$\forall c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad f(n) \geq c \cdot g(n)$$

anders ausgedrückt: $$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Beispiel

n^2 und $5n^2 - 7n$ haben gleiche Wachstumsrate, da für alle $n \geq 2$ $1 \leq \frac{5n^2 - 7n}{n^2} \leq 5$ gilt. Beide wachsen schneller als $n^{3/2}$.

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$
$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$
$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben

$\Rightarrow$ Forderung: $\exists n_0 : \forall n \geq n_0 : f(n) > 0$

Manchmal auch: $\forall n : f(n) \geq 0$

Wachstumsrate / -ordnung

- Warum die Betrachtung der Wachstumsrate und die Forderung nur für genügend große n?

  Ziel effizienter Algorithmen: Lösung großer Probleminstanzen

  gesucht: Verfahren, die für große Instanzen noch effizient sind

  Für große n sind Verfahren mit kleinerer Wachstumsrate besser.

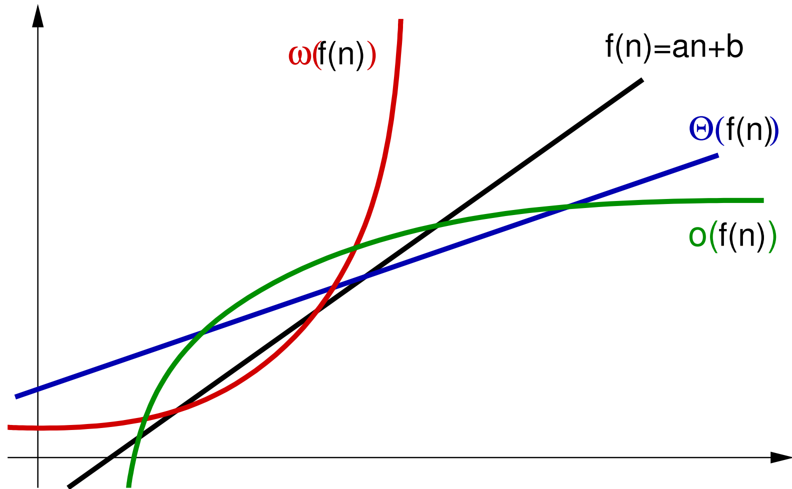
- Warum Verzicht auf konstante Faktoren?

  Unser Maschinenmodell ist nur eine Abstraktion von echten Computern und kann die reale Laufzeit sowieso nur bis auf konstante Faktoren bestimmen.

  Daher ist es meistens sinnvoll, Algorithmen mit gleicher Wachstumsrate erstmal als gleichwertig zu betrachten.

- außerdem: Laufzeitangabe durch einfache Funktionen

Asymptotische Notation



Asymptotische Notation

Beispiel

- $5n^2 - 7n \in \mathcal{O}(n^2)$, $n^2/10 + 100n \in \mathcal{O}(n^2)$, $4n^2 \in \mathcal{O}(n^3)$
- $5n^2 - 7n \in \Omega(n^2)$, $n^3 \in \Omega(n^2)$, $n \log n \in \Omega(n)$
- $5n^2 - 7n \in \Theta(n^2)$
- $\log n \in o(n)$, $n^3 \in o(2^n)$
- $n^5 \in \omega(n^3)$, $2^{2n} \in \omega(2^n)$

Asymptotische Notation als Platzhalter

- statt $g(n) \in \mathcal{O}(f(n))$ schreibt man oft auch $g(n) = \mathcal{O}(f(n))$
- für $f(n) + g(n)$ mit $g(n) \in o(h(n))$ schreibt man auch $f(n) + g(n) = f(n) + o(h(n))$
- statt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$ schreibt man auch $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$

Beispiel

$$n^3 + n \quad = \quad n^3 + o(n^3) \quad = \quad (1 + o(1))n^3 \quad = \quad \mathcal{O}(n^3)$$

$\mathcal{O}$-Notations"gleichungen" sollten nur von links nach rechts gelesen werden!

Übersicht

3. Effizienz
   - Effizienzmaße
   - Rechenregeln für $\mathcal{O}$-Notation
   - Maschinenmodell
   - Laufzeitanalyse
   - Durchschnittliche Laufzeit
   - Erwartete Laufzeit

Wachstumsrate von Polynomen

Lemma

Sei p ein Polynom der Ordnung k bzgl. der Variable n, also

$$p(n) = \sum_{i=0}^{k} a_i \cdot n^i \quad \textit{mit} \quad a_k > 0.$$

Dann ist

$$p(n) \in \Theta(n^k).$$

Wachstumsrate von Polynomen

Beweis.

Zu zeigen: $p(n) \in \mathcal{O}(n^k)$ und $p(n) \in \Omega(n^k)$

$p(n) \in \mathcal{O}(n^k)$:

Für $n \geq 1$ gilt:

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

Also ist die Definition

$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

mit $c = \sum_{i=0}^{k} |a_i|$ und $n_0 = 1$ erfüllt.

Wachstumsrate von Polynomen

Beweis.

$p(n) \in \Omega(n^k)$:

$$A = \sum_{i=0}^{k-1} |a_i|$$

Für positive n gilt dann:

$$p(n) \geq a_k n^k - A n^{k-1} = \frac{a_k}{2} n^k + n^{k-1} \left(\frac{a_k}{2} n - A \right)$$

Also ist die Definition

$$\Omega(f(n)) = \{ g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$$

mit $c = a_k / 2$ und $n_0 > 2A / a_k$ erfüllt. □

Rechenregeln für $\mathcal{O}$-Notation

Für Funktionen $f(n)$ (bzw. $g(n)$) mit $\exists n_0\ \forall n \geq n_0 :\ f(n) > 0$ gilt:

Lemma

- *$c \cdot f(n) \in \Theta(f(n))$ für jede Konstante $c > 0$*
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Die Ausdrücke sind auch korrekt für Ω statt $\mathcal{O}$.

Vorsicht, der letzte heißt dann

- $\Omega(f(n) + g(n)) = \Omega(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Aber: Vorsicht bei induktiver Anwendung!

Induktions"beweis"

Behauptung:

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

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

Ind.anfang: $f(1) = \mathcal{O}(1)$

Ind.vor.: Es gelte $f(n-1) = \mathcal{O}(n-1)$

Ind.schritt: Dann gilt

$$f(n) = n + f(n-1) = n + \mathcal{O}(n-1) = \mathcal{O}(n)$$

Also ist

$$f(n) = \sum_{i=1}^{n} i = \mathcal{O}(n)$$

FALSCH!

Ableitungen und $\mathcal{O}$-Notation

Lemma

Seien f und g differenzierbar.

Dann gilt

- *falls $f'(n) \in \mathcal{O}(g'(n))$, dann auch $f(n) \in \mathcal{O}(g(n))$*
- *falls $f'(n) \in \Omega(g'(n))$, dann auch $f(n) \in \Omega(g(n))$*
- *falls $f'(n) \in o(g'(n))$, dann auch $f(n) \in o(g(n))$*
- *falls $f'(n) \in \omega(g'(n))$, dann auch $f(n) \in \omega(g(n))$*

Umgekehrt gilt das im Allgemeinen nicht!

Rechenbeispiele für $\mathcal{O}$-Notation

Beispiel

- 1. Lemma:
  - $n^3 - 3n^2 + 2n \in \mathcal{O}(n^3)$
  - $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2/2 + n/2) = \mathcal{O}(n^2)$
- 2. Lemma:
  Aus $\log n \in \mathcal{O}(n)$ folgt $n \log n \in \mathcal{O}(n^2)$.
- 3. Lemma:
  - $(\log n)' = 1/n$, $(n)' = 1$ und $1/n \in \mathcal{O}(1)$.
    $\Rightarrow \log n \in \mathcal{O}(n)$

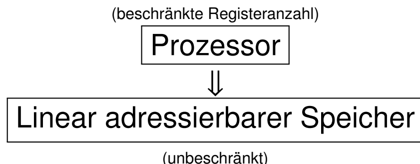
Übersicht

3 Effizienz
- Effizienzmaße
- Rechenregeln für $\mathcal{O}$-Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Abstraktion durch Maschinen-/Rechnermodelle

- 1936 Turing-Maschine: kann nicht auf beliebige Speicherzellen zugreifen, nur an der aktuellen Position des Lese-/Schreibkopfs
- 1945 J. von Neumann u.a.: Entwurf des Rechners EDVAC (Electronic Discrete Variable Automatic Computer)
  Programm und Daten teilen sich einen gemeinsamen Speicher
- 1963 John Shepherdson, Howard Sturgis (u.a.):

Random Access Machine (RAM)



RAM: Aufbau

Prozessor:

- beschränkte Anzahl an Registern $R_1, \ldots, R_k$
- Instruktionszeiger zum nächsten Befehl

Programm:

- nummerierte Liste von Befehlen
  (Adressen in Sprungbefehlen entsprechen dieser Nummerierung)

Eingabe:

- steht in Speicherzellen $S[1], \ldots, S[R_1]$

Modell / Reale Rechner:

- unendlicher / endlicher Speicher
- Abhängigkeit / Unabhängigkeit der Größe der Speicherzellen von der Eingabegröße

RAM: Speicher

- unbeschränkt viele Speicherzellen (words) $S[0], S[1], S[2], \ldots$, von denen zu jedem Zeitpunkt nur endlich viele benutzt werden
- beliebig große Zellen führen zu unrealistischen Algorithmen

$\Rightarrow$ Jede Speicherzelle darf bei Eingabelänge n eine Zahl mit $\mathcal{O}(\log n)$ Bits speichern.
(Für konstant große Zellen würde man einen Faktor $\mathcal{O}(\log n)$ bei der Rechenzeit erhalten.)

$\Rightarrow$ gespeicherte Werte stellen polynomiell in Eingabelänge n beschränkte Zahlen dar (sinnvoll für Array-Indizes; bildet auch geschichtliche Entwicklung $4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ Bit ab)

Begrenzter Parallelismus:

- sequentielles Maschinenmodell, aber
- Verknüpfung logarithmisch vieler Bits in konstanter Zeit

RAM: Befehle

Annahme:

- Jeder Befehl dauert genau eine Zeiteinheit.
- Laufzeit ist Anzahl ausgeführter Befehle

Befehlssatz:

- Registerzuweisung:
  $R_i := c$ (Konst. an Register), $R_i := R_j$ (Register an Register)
- Speicherzugriff:
  $R_i := S[R_j]$ (lesend), $S[R_j] := R_i$ (schreibend)
- Arithmetische / logische Operationen:
  $R_i := R_j \text{ op } R_k$ (binär: op $\in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, <, \leq, =, \geq, >\}$),
  $R_i := \text{op } R_j$ (unär: op $\in \{-, \neg\}$)
- Sprünge:
  jump x (zu Adresse x), jumpz x R_i (bedingt, falls $R_i = 0$),
  jumpi R_j (zu Adresse aus R_j)

Das entspricht Assembler-Code von realen Maschinen!

Maschinenmodell

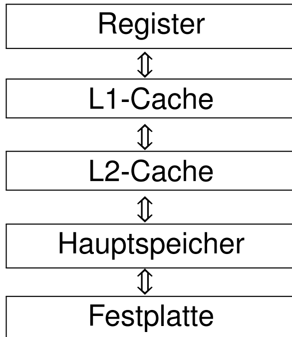
RAM-Modell

- Modell für die ersten Computer
- entspricht eigentlich der Harvard-Architektur (separater Programmspeicher)
- Random Access Stored Program (RASP) Modell entspricht der von Neumann-Architektur und hat große Ähnlichkeit mit üblichen Rechnern

Aber: Speicherhierarchie erfordert ggf. Anpassung des Modells

⇒ Algorithm Engineering, z.B. External-Memory Model

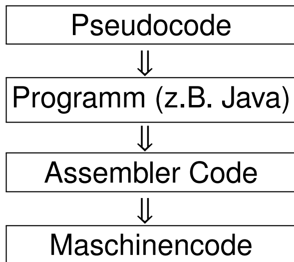
Speicherhierarchie



External-Memory Model

- begrenzter schneller Speicher mit M Zellen
- unbegrenzter (langsamer) externer Speicher
- I/O-Operationen transferieren B aufeinanderfolgende Worte

Pseudocode / Maschinencode



- Assembler/Maschinencode schwer überschaubar
- besser: Programmiersprache wie Pascal, C++, Java, ...
- oder: informal als Pseudocode in verständlicher Form

$a := a + bc \quad \Rightarrow \quad R_1 := R_b * R_C; \quad R_a := R_a + R_1$

R_a, R_b, R_c: Register, in denen a, b und c gespeichert sind

if (C) I else J $\quad \Rightarrow \quad$ eval(C); jumpz sElse R_c; trans(I); jump sEnd; trans(J)

eval(C): Befehle, die die Bedingung C auswerten und das Ergebnis in Register R_c hinterlassen
trans(I), trans(J): übersetzte Befehlsfolge für I und J

sElse, sEnd: Adresse des 1. Befehls in trans(J) bzw. des 1. Befehls nach trans(J)

Übersicht

3 Effizienz

- Effizienzmaße
- Rechenregeln für $\mathcal{O}$-Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Laufzeitanalyse / worst case

Berechnung der worst-case-Laufzeit:

- $T(I)$ sei worst-case-Laufzeit für Konstrukt I

- $T(\text{elementare Zuweisung}) = \mathcal{O}(1)$
- $T(\text{elementarer Vergleich}) = \mathcal{O}(1)$
- $T(\textbf{return } x) = \mathcal{O}(1)$
- $T(\textbf{new } \text{Typ}(\ldots)) = \mathcal{O}(1) + \mathcal{O}(T(\text{Konstruktor}))$
- $T(I_1; I_2) = T(I_1) + T(I_2)$
- $T(\textbf{if } (C)\ I_1 \textbf{ else } I_2) = \mathcal{O}(T(C) + \max\{T(I_1), T(I_2)\})$
- $T(\textbf{for}(i = a; i < b; i{+}{+})\ I) = \mathcal{O}\left(\sum_{i=a}^{b-1}(1 + T(I))\right)$
- $T(e.m(\ldots)) = \mathcal{O}(1) + T(ss)$, wobei ss Rumpf von m

Beispiel: Vorzeichenausgabe

Funktion signum(x)

Eingabe : Zahl $x \in \mathbb{R}$
Ausgabe : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

```
if x < 0 then
  return -1
if x > 0 then
  return 1
return 0
```

Wir wissen:

$$T(x < 0) = \mathcal{O}(1)$$
$$T(\textbf{return } \textit{-1}) = \mathcal{O}(1)$$
$$T(\textbf{if } (C)\ I) = \mathcal{O}(T(C) + T(I))$$

Also: $T(\textbf{if } (x < 0) \textbf{ return } -1) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$

Beispiel: Vorzeichenausgabe

Funktion signum(x)

Eingabe : Zahl $x \in \mathbb{R}$
Ausgabe : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

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

$$\mathcal{O}(1+1+1) = \mathcal{O}(1)$$

Beispiel: Minimumsuche

Funktion minimum(A, n)

Eingabe : Zahlenfolge in $A[0], \ldots, A[n-1]$
n: Anzahl der Zahlen

Ausgabe : Minimum der Zahlen

Code	Laufzeit
min $= A[0]$;	$\mathcal{O}(1)$
for $(i = 1; i < n; i{+}{+})$ **do**	$\mathcal{O}(\sum_{i=1}^{n-1}(1 + T(I)))$
if $A[i] <$ min **then** min $= A[i]$;	$\mathcal{O}(1)$ ↗
return min	$\mathcal{O}(1)$

$$\mathcal{O}\left(1 + \left(\sum_{i=1}^{n-1} 1\right) + 1\right) = \mathcal{O}(n)$$

Beispiel: BubbleSort

Sortieren durch Aufsteigen

Vertausche in jeder Runde in der (verbleibenden) Eingabesequenz (hier vom Ende in Richtung Anfang) jeweils zwei benachbarte Elemente, die nicht in der richtigen Reihenfolge stehen

Beispiel

5	10	19	1	14	3
5	10	19	1	3	14
5	10	1	19	3	14
5	1	10	19	3	14
1	5	10	19	3	14

1	5	10	3	19	14
1	5	3	10	19	14
1	**3**	5	10	19	14
1	3	**5**	10	14	19
1	3	5	**10**	**14**	**19**

Beispiel: Sortieren

Prozedur BubbleSort(A, n)

Eingabe : n: Anzahl der Zahlen
$A[0], \ldots, A[n-1]$: Zahlenfolge

Ausgabe : Sortierte Zahlenfolge A

Code	Laufzeit
for $(i = 0; i < n-1; i{+}{+})$ **do**	$\mathcal{O}(\sum_{i=0}^{n-2} T(I_1))$
for $(j = n-2; j \geq i; j{-}{-})$ **do**	$\mathcal{O}(\sum_{j=i}^{n-2} T(I_2))$
if $A[j] > A[j+1]$ **then**	$\mathcal{O}(1 + T(I_3))$
$x = A[j]$;	$\mathcal{O}(1)$
$A[j] = A[j+1]$;	$\mathcal{O}(1)$
$A[j+1] = x$;	$\mathcal{O}(1)$

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

Beispiel: Sortieren

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

Beispiel: Binäre Suche

Prozedur BinarySearch(A, n, x)

Eingabe : n: Anzahl der (sortierten) Zahlen
$A[0], \ldots, A[n-1]$: Zahlenfolge
x: gesuchte Zahl

Ausgabe : Index der gesuchten Zahl

Code	Laufzeit
$\ell = 0;$	$\mathcal{O}(1)$
$r = n - 1;$	$\mathcal{O}(1)$
while $(\ell \leq r)$ **do**	$\mathcal{O}(\sum_{i=1}^{k} T(I))$
$\quad m = \lfloor (r + \ell)/2 \rfloor;$	$\mathcal{O}(1) \uparrow$
$\quad$ **if** $A[m] == x$ **then return** m;	$\mathcal{O}(1)$
$\quad$ **if** $A[m] < x$ **then** $\ell = m + 1;$	$\mathcal{O}(1)$
$\quad$ **else** $r = m - 1;$	$\mathcal{O}(1)$
return -1	$\mathcal{O}(1)$

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

Beispiel: Binäre Suche

Aber: Wie groß ist die Anzahl der Schleifendurchläufe k?

Größe des verbliebenen Suchintervalls $(r - \ell + 1)$ nach Iteration i:

$$\begin{aligned} s_0 &= n \\ s_{i+1} &\leq \lfloor s_i/2 \rfloor \end{aligned}$$

Bei $s_i < 1$ endet der Algorithmus.

$$\Rightarrow \quad k \leq \log_2 n$$

Gesamtkomplexität: $\mathcal{O}(\log n)$

Beispiel: Bresenham-Algorithmus

Algorithmus Bresenham1: zeichnet einen Kreis

```
x = 0;   y = R;                                                   O(1)
plot(0, R); plot(R, 0); plot(0, −R); plot(−R, 0);                 O(1)
F = 5/4 − R;                                                      O(1)
while x < y do                                                    O(∑_{i=1}^k T(I))
    if F < 0 then
        F = F + 2 ∗ x + 1;
    else
        F = F + 2 ∗ x − 2 ∗ y + 2;                                alles O(1)
        y = y − 1;
    x = x + 1;
    plot(x, y); plot(−x, y); plot(−y, x); plot(−y, −x);
    plot(y, x); plot(y, −x); plot(x, −y); plot(−x, −y);
```

Wie groß ist Anzahl Schleifendurchläufe k?

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

Beispiel: Bresenham-Algorithmus

- Betrachte dazu die Entwicklung der Werte der Funktion

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

- Anfangswert: $\varphi_0(x, y) = R$
- Monotonie: verringert sich pro Durchlauf um mindestens 1
- Beschränkung: durch die **while**-Bedingung $x < y$ bzw. $0 < y - x$

$\Rightarrow$ maximal R Runden

Beispiel: Fakultätsfunktion

```
Funktion fakultaet(n)
Eingabe  : n ∈ ℕ₊
Ausgabe  : n!
if (n == 1) then                      O(1)
  └ return 1                            O(1)
else
  └ return n * fakultaet(n − 1)         O(1 + ...?)
```

- $T(n)$: Laufzeit von fakultaet(n)
- $T(1) = \mathcal{O}(1)$
- $T(n) = T(n-1) + \mathcal{O}(1)$

$\Rightarrow$ $T(n) = \mathcal{O}(n)$

Übersicht

3 Effizienz

- Effizienzmaße
- Rechenregeln für $\mathcal{O}$-Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Average Case Complexity

Uniforme Verteilung:
(alle Instanzen gleichwahrscheinlich)

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$

Tatsächliche Eingabeverteilung kann in der Praxis aber stark von uniformer Verteilung abweichen.

Dann

$$t(n) = \sum_{i \in \mathcal{I}_n} p_i \cdot T(i)$$

Aber: meist schwierig zu berechnen!

Beispiel: Binärzahl-Inkrementierung

Prozedur increment(A)

Eingabe : Array A mit Binärzahl in $A[0] \dots A[n-1]$,
in $A[n]$ steht eine 0

Ausgabe : inkrementierte Binärzahl in $A[0] \dots A[n]$

```
i = 0;
while (A[i] == 1) do
    A[i] = 0;
    i = i + 1;
A[i] = 1;
```

Durchschnittliche Laufzeit für Zahl mit n Bits?

Binärzahl-Inkrementierung: Analyse

- $\mathcal{I}_n$: Menge der n-Bit-Instanzen
- Für die Hälfte (also $\frac{1}{2}|\mathcal{I}_n|$) der Zahlen $x_{n-1} \dots x_0 \in \mathcal{I}_n$ ist $x_0 = 0$
  $\Rightarrow$ 1 Schleifendurchlauf
- Für die andere Hälfte gilt $x_0 = 1$.
  Bei diesen gilt wieder für die Hälfte (also $\frac{1}{4}|\mathcal{I}_n|$) $x_1 x_0 = 01$
  $\Rightarrow$ 2 Schleifendurchläufe
- Für den Anteil $(\frac{1}{2})^k$ der Zahlen gilt $x_{k-1} x_{k-2} \dots x_0 = 01 \dots 1$
  $\Rightarrow$ k Schleifendurchläufe

Durchschnittliche Anzahl Schleifendurchläufe:

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i) = \frac{1}{|\mathcal{I}_n|} \sum_{k=1}^{n} \frac{|\mathcal{I}_n|}{2^k} \cdot k = \sum_{k=1}^{n} \frac{k}{2^k} \stackrel{?}{=} \mathcal{O}(1)$$

Binärzahl-Inkrementierung: Abschätzung

Lemma

$$\sum_{k=1}^{n} \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$$

Beweis

Induktionsanfang:

Für $n = 1$ gilt: $$\sum_{k=1}^{1} \frac{k}{2^k} = \frac{1}{2} \leq 2 - \frac{1+2}{2^1} \quad \checkmark$$

Induktionsvoraussetzung:

Für n gilt: $$\sum_{k=1}^{n} \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$$

Binärzahl-Inkrementierung: Abschätzung

Beweis.

Induktionsschritt: $n \to n+1$

$$
\begin{aligned}
\sum_{k=1}^{n+1} \frac{k}{2^k} &= \left(\sum_{k=1}^{n} \frac{k}{2^k} \right) + \frac{n+1}{2^{n+1}} \\
&\leq 2 - \frac{n+2}{2^n} + \frac{n+1}{2^{n+1}} \qquad \text{(laut Ind.vor.)} \\
&= 2 - \frac{2(n+2)}{2^{n+1}} + \frac{n+1}{2^{n+1}} = 2 - \frac{2n+4-n-1}{2^{n+1}} \\
&= 2 - \frac{n+3}{2^{n+1}} \\
&= 2 - \frac{(n+1)+2}{2^{n+1}}
\end{aligned}
$$

Übersicht

3. Effizienz
   - Effizienzmaße
   - Rechenregeln für $\mathcal{O}$-Notation
   - Maschinenmodell
   - Laufzeitanalyse
   - Durchschnittliche Laufzeit
   - Erwartete Laufzeit

Zufallsvariable

Definition

Für einen Wahrscheinlichkeitsraum mit Ergebnismenge Ω nennt man eine Abbildung $X : \Omega \mapsto \mathbb{R}$ (numerische) Zufallsvariable.

Eine Zufallsvariable über einer endlichen oder abzählbar unendlichen Ergebnismenge heißt diskret.

Der Wertebereich diskreter Zufallsvariablen

$$W_X := X(\Omega) = \{x \in \mathbb{R} \mid \exists \omega \in \Omega \text{ mit } X(\omega) = x\}$$

ist ebenfalls endlich bzw. abzählbar unendlich.

Schreibweise: $\Pr[X = x] := \Pr\left[X^{-1}(x)\right] = \sum_{\omega \in \Omega \mid X(\omega) = x} \Pr[\omega]$

Zufallsvariable

Beispiel

Wir ziehen aus einem Poker-Kartenspiel mit 52 Karten (13 von jeder Farbe) eine Karte.

Wir bekommen bzw. bezahlen einen bestimmten Betrag, je nachdem welche Farbe die Karte hat, z.B. 4 Euro für Herz, 7 Euro für Karo, –5 Euro für Kreuz und –3 Euro für Pik.

Wenn wir ein As ziehen, bekommen wir zusätzlich 1 Euro.

$$\Omega = \{\heartsuit A, \heartsuit K, \ldots, \heartsuit 2, \diamondsuit A, \diamondsuit K, \ldots, \diamondsuit 2, \clubsuit A, \clubsuit K, \ldots, \clubsuit 2, \spadesuit A, \spadesuit K, \ldots, \spadesuit 2\}.$$

X sei der Geldbetrag den wir bekommen bzw. bezahlen.

$$W_X = \{-5, -4, -3, -2, 4, 5, 7, 8\}$$

$$\Pr[X = -3] = \Pr[\spadesuit K] + \ldots + \Pr[\spadesuit 2] = 12/52 = 3/13$$

Erwartungswert

Definition

Für eine diskrete Zufallsvariable X ist der Erwartungswert definiert als

$$\mathbb{E}[X] := \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega]$$

sofern $\sum_{x \in W_X} |x| \cdot \Pr[X = x]$ konvergiert (absolute Konvergenz).

Bei endlicher Ereignismenge und gleichwahrscheinlichen Ereignissen entspricht der Erwartungswert dem Durchschnitt:

$$\mathbb{E}[X] = \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \frac{1}{|\Omega|} = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} X(\omega)$$

Erwartungswert

Beispiel

(Beispiel wie zuvor)

$$\mathbb{E}[X] = 4 \cdot \frac{12}{52} + 5 \cdot \frac{1}{52} + 7 \cdot \frac{12}{52} + 8 \cdot \frac{1}{52} + (-5) \cdot \frac{12}{52} + (-4) \cdot \frac{1}{52} + (-3) \cdot \frac{12}{52} + (-2) \cdot \frac{1}{52} = \frac{43}{52}$$

Wir bekommen also im Erwartungswert $\frac{43}{52}$ Euro pro gezogener Karte.

Grundlagen zu diskreter Wahrscheinlichkeitstheorie findet man z.B. in folgendem Buch:

Th. Schickinger, A. Steger
Diskrete Strukturen – Band 2
(Wahrscheinlichkeitstheorie und Statistik)
Springer-Verlag, 2001.

Erwartungswert

Beispiel

Münze werfen, bis sie zum ersten Mal Kopf zeigt

Zufallsvariable k: Anzahl der Versuche

k ungerade: Spieler bezahlt etwas an die Bank
k gerade: Spieler bekommt etwas von der Bank

Zufallsvariable X: Gewinnbetrag der Bank

Variante 1: Spieler bezahlt / bekommt k Euro
$\mathbb{E}[X]$ existiert (absolute Konvergenz)

Variante 2: Spieler bezahlt / bekommt 2^k Euro
$\mathbb{E}[X]$ existiert nicht (keine Konvergenz)

Variante 3: Spieler bezahlt / bekommt $\frac{2^k}{k}$ Euro
$\mathbb{E}[X]$ existiert nicht (Konvergenz, aber keine absolute)

Erwartungswert zusammengesetzter Zufallsvariablen

Satz (Linearität des Erwartungswerts)

Für Zufallsvariablen $X_1, \ldots, X_n$ und

$$X := a_1 X_1 + \ldots + a_n X_n$$

mit $a_1, \ldots, a_n \in \mathbb{R}$ gilt

$$\mathbb{E}[X] = a_1 \mathbb{E}[X_1] + \ldots + a_n \mathbb{E}[X_n].$$

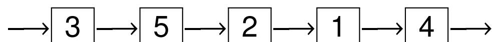
Interessant ist für uns vor allem der einfache Fall:

$$X := X_1 + \ldots + X_n$$

mit

$$\mathbb{E}[X] = \mathbb{E}[X_1] + \ldots + \mathbb{E}[X_n].$$

Beispiel: Suche in statischer Liste



- gegeben: Liste mit Elementen $1, \ldots, m$
- search(i): lineare Suche nach Element i ab Listenanfang

  s_i Position von Element i in der Liste (1$\hat{=}$Anfang)

  p_i Wahrscheinlichkeit für Zugriff auf Element i

Erwartete Laufzeit der Operation search(i) mit zufälligem i:

$$\mathbb{E}[T(\text{search}(i))] = \mathcal{O}\left(\sum_i p_i s_i\right)$$

Erwartete Laufzeit $t(n)$ für n Zugriffe bei statischer Liste:

$$t(n) = \mathbb{E}[T(n \times \text{search}(i))] = n \cdot \mathbb{E}[T(\text{search}(i))] = \mathcal{O}\left(n \sum_i p_i s_i\right)$$

Beispiel: Suche in statischer Liste

Optimale Anordnung?

$\Rightarrow$ wenn für alle Elemente i, j mit $p_i > p_j$ gilt, dass $s_i < s_j$, d.h. die Elemente nach Zugriffswahrscheinlichkeit sortiert sind

o.B.d.A. seien die Indizes so, dass $p_1 \geq p_2 \geq \ldots \geq p_m$

- Optimale Anordnung: $s_i = i$
- Optimale erwartete Laufzeit: $\text{opt} = \sum_i p_i \cdot i$

Einfach: wenn die Zugriffswahrscheinlichkeiten bekannt sind
$\Rightarrow$ optimale erwartete Laufzeit durch absteigende Sortierung nach p_i

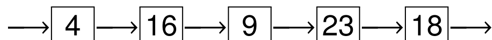
Problem: was wenn die Wahrscheinlichkeiten p_i unbekannt sind?

Beispiel: Suche in selbstorganisierender Liste

Move-to-Front Rule:

Verschiebe nach jeder erfolgreichen Suche das gefundene Element an den Listenanfang

Bsp.: Ausführung von search(4) ergibt



Beispiel: Suche in selbstorganisierender Liste

Erwartete Laufzeit $t(n)$ bei dynamischer Liste:

$$\mathbb{E}[T(\text{search}(i))] = \mathcal{O}\left(\sum_i p_i \cdot \mathbb{E}[s_i]\right)$$

Satz

Ab dem Zeitpunkt, wo auf jedes Element mindestens einmal zugegriffen wurde, ist die erwartete Laufzeit der search-Operation unter Verwendung der Move-to-Front Rule höchstens $2 \cdot \text{opt}$.

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Betrachte zwei feste Elemente i und j

t_0 Zeitpunkt der letzten Suchoperation auf i oder j

- bedingte Wahrscheinlichkeit: $\Pr[A \mid B] = \frac{\Pr[A \wedge B]}{\Pr[B]}$
- $\Pr[C \mid (C \vee D)] = \frac{\Pr[C \wedge (C \vee D)]}{\Pr[C \vee D]} = \frac{\Pr[C]}{\Pr[C \vee D]}$
- $\Pr[\text{search}(j) \text{ bei } t_0 \mid \text{search}(i \vee j) \text{ bei } t_0] = \frac{p_j}{p_i + p_j}$
- mit Wsk. $\frac{p_i}{p_i + p_j}$ steht i vor j und mit Wsk. $\frac{p_j}{p_i + p_j}$ steht j vor i

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Betrachte nun nur ein festes Element i

- Definiere Zufallsvariablen $X_j \in \{0,1\}$ für $j \neq i$:

$$X_j = 1 \quad \Leftrightarrow \quad j \text{ vor } i \text{ in der Liste}$$

- Erwartungswert:

$$\begin{aligned}
\mathbb{E}[X_j] &= 0 \cdot \Pr[X_j = 0] + 1 \cdot \Pr[X_j = 1] \\
&= \Pr[\text{letzte Suche nach } i\,/\,j \text{ war nach } j] \\
&= \frac{p_j}{p_i + p_j}
\end{aligned}$$

Beispiel: Suche in selbstorganisierender Liste

Beweis.

- Listenposition von Element i: $1 + \sum_{j \neq i} X_j$
- Erwartungswert der Listenposition von Element i:

$$\begin{aligned}
\mathbb{E}[s_i] &= \mathbb{E}\left[1 + \sum_{j \neq i} X_j\right] \\
&= 1 + \mathbb{E}\left[\sum_{j \neq i} X_j\right] = 1 + \sum_{j \neq i} \mathbb{E}\left[X_j\right] \\
\mathbb{E}[s_{i,\text{MTF}}] &= 1 + \sum_{j \neq i} \frac{p_j}{p_i + p_j}
\end{aligned}$$

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Erwartete Laufzeit der search-Operation:

$$\begin{aligned}
\mathbb{E}[T_{\text{MTF}}] &= \sum_i p_i \left(1 + \sum_{j \neq i} \frac{p_j}{p_i + p_j} \right) \\
&= \sum_i \left(p_i + \sum_{j \neq i} \frac{p_i p_j}{p_i + p_j} \right) = \sum_i \left(p_i + 2 \sum_{j<i} \frac{p_i p_j}{p_i + p_j} \right) \\
&= \sum_i p_i \left(1 + 2 \sum_{j<i} \frac{p_j}{p_i + p_j} \right) \leq \sum_i p_i \left(1 + 2 \sum_{j<i} 1 \right) \\
&\leq \sum_i p_i \cdot (2i - 1) < \sum_i p_i \cdot 2i = 2 \cdot \text{opt}
\end{aligned}$$