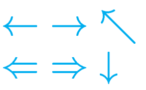
Algorithmen und Datenstrukturen
(Th. Ottmann und P. Widmayer)

Folien: Leistungsanalyse

Autor: Stefan Edelkamp

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg

1 Überblick



Überblick

Themen

Leistungsverhalten von Algorithmen

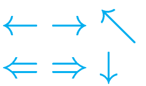
Laufzeitanalyse

Funktionenklassen

Beispiele

Mastertheorem

2 Themen



Algorithmen-Problembereiche

Datenstrukturen:

Listen, Stapel, Schlangen, Bäume,. . .

Problemlösetechniken

Divide-and-Conquer, Greedy, Dynamische Programmierung, vollständige Aufzählung, Backtracking, . . .

Benötigt werden

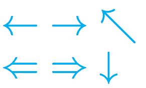
Sprache zur Formulierung von Algorithmen:

Pascal, Modula-2, C, C++, Java, . . .

Mathematisches Instrumentarium zur Messung der Komplexität (Zeit- und Platzbedarf)

Groß-O-Kalkül

3 Leistungsverhalten von Algorithmen



Effizienzanalyse:

Speicherplatzkomplexität: Wird primärer, sekundärer Speicherplatz effizient genutzt?

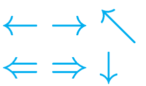
Laufzeitkomplexität: Steht die Laufzeit im akzeptablen/vernünftigen/optimalen Verhältnis zur Aufgabe?

Theorie: liefert untere Schranke, die für jeden Algorithmus gilt, der das Problem löst.

Spezieller Algorithmus liefert obere Schranke für die Lösung des Problems.

Erforschung von oberen und unteren Schranken: Effiziente Algorithmen und Komplexitätstheorie (Zweige der Theoretischen Informatik)

4 Laufzeitanalyse



Sei P ein gegebenes Problem und x Eingabe für P, $|x|$ Länge von x, und $T_P(x)$ die Laufzeit von P auf x.

Der beste Fall (best case): Oft leicht zu bestimmen, kommt in der Praxis jedoch selten vor.

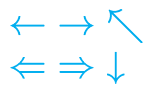
$$T_P(n) = \inf\{T_P(x) | \, |x| = n, x \text{ Eingabe für } P\}$$

Der schlechteste Fall (worst case): Liefert garantierte Schranken, meist leicht zu bestimmen. Oft zu pessimistisch

$$T_P(n) = \sup\{T_P(x) | \, |x| = n, x \text{ Eingabe für } P\}$$

Im amortisierten worst case wird der durchschnittliche Aufwand für eine schlechtestmögliche Folge von Eingaben gemessen (technisch anspruchsvoll).

Der Mittlere Fall



Der mittlere Fall (average case): Nicht leicht zu handhaben, für die Praxis jedoch relevant.

Sei $q_n(x)$ Wahrscheinlichkeitsverteilung, mit der Eingabe x mit Länge n auftritt.

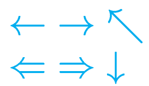
$T_{P,q_n}(n) = \sum_{|x|=n, x \text{ Eingabe für } P} T_P(x) q_n(x)$

Wichtig:

Worüber wird gemittelt?

Sind alle Eingaben der Länge n gleichwahrscheinlich?

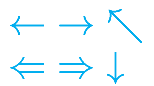
Messung des Leistungsverhaltens



1. Betrachte konkrete Implementierung auf konkreter Hardware. Miß Laufzeit für repräsentative Eingaben.
2. Berechne Verbrauch an Platz und Zeit für idealisierte Referenzmachine:
   a) Random Access Machine (RAM)
   b) Registermachine (RM)
   c) Turingmachine (TM)
3. Bestimme Anzahlen gewisser (teurer) Grundoperationen, z.B.
   # Vergleiche bzw. # Bewegung von Daten (beim Sortieren)
   # Multiplikationen / Divisionen (für numerische Verfahren)

Für 2. und 3.: Beschreibe Aufwand eines Algorithmus als Funktion der Größe des Inputs (kann verschieden gemessen werden).

Kostenmaße



Einheitskostenmaß: Annahme, jedes Datenelement belegt unabhängig von seiner Größe denselben Speicherplatz (in Einheitsgröße).

Damit: Größe der Eingabe bestimmt durch Anzahl der Datenelemente

Beispiel: Sortierproblem

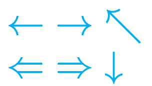
Logarithmisches Kostenmaß: Annahme, jedes Datenelement belegt einen von seiner Größe (logarithmisch) abhängigen Platz.

Größe der Eingabe bestimmt durch die Summe der Größen der Elemente.

(für $n > 0$ ist die # Bits zur Darstellung von n $= \lceil \log_2(n+1) \rceil$)

Beispiel: Zerlegung einer in Dualdarstellung gegebenen Zahl in Primfaktoren

Beispiel



Taktzahl (Anzahl Bitwechsel) eines Von-Neumann Addierwerks bei Addition von 1 zu einer in Binärdarstellung gegebenen Zahl i der Länge n, d.h. $0 \leq i \leq 2^{n-1}$. Sie beträgt 1 plus der Anzahl der Einsen am Ende der Binärdarstellung von i.

Schlechtester Fall:

Die worst case Rechenzeit beträgt $n + 1$ Takte.

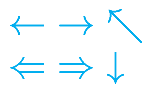
Beispiel:

Addition von 1 zu $111\ldots1$.

Mittlerer Fall

Wir nehmen eine Gleichverteilung auf der Eingabenmenge an. Es gibt 2^{n-k} Eingaben, die mit $(0, 1, \ldots, 1)$ enden, wobei am Ende $k - 1$ Einsen stehen. Hinzu kommt die Eingabe $i = 2^n - 1$ für die das Addierwerk $n + 1$ Takte benötigt.

Mittlerer Fall



Die average case Rechenzeit $T_a(n)$ beträgt also:

$T_a(n) = 2^{-n}(\sum_{1 \le k \le n} 2^{n-k} k + (n+1))$

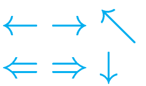
Es ist

$$\begin{aligned}
\sum_{1 \le k \le n} 2^{n-k} k &= n 2^{n-n} + \ldots + 2 \cdot 2^{n-2} + 1 \cdot 2^{n-1} \\
&= \quad 2^0 + \ldots + 2^{n-3} + 2^{n-2} + 2^{n-1} \\
&\quad +2^0 + \ldots + 2^{n-3} + 2^{n-2} \\
&\quad +2^0 + \ldots + 2^{n-3} \\
&\quad \vdots \\
&\quad +2^0 \\
&= (2^n - 1) + \ldots + (2^1 - 1) \\
&= 2^{n+1} - 2 - n.
\end{aligned}$$

Demnach ist
$T_a(n) = 2^{-n}(2^{n+1} - 2 - n + (n+1)) = 2 - 2^{-n}$.

Es genügen also im Durchschnitt 2 Takte um eine Addition von 1 durchzuführen.

5 Funktionenklassen



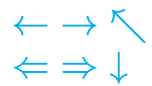
Groß-O-Notation: O-, Ω- und Θ- Notation beschreiben obere, untere bzw. genaue Schranken.

Idee: Beim Aufwand kommt es auf Summanden und konstante Faktoren letztendlich nicht an.

Gründe:
- Interesse an asymptotischen Verhalten für große Eingaben,
- Genaue Analyse oft technisch sehr aufwendig/unmöglich
- Lineare Beschleunigungen sind leicht möglich (Tausch von Hard- und Software)

Ziel: Komplexitätsmessungen mit Hilfe von Funktionenklassen, z.B. $O(f)$ Menge von Funktionen, die in der Größenordnung von f sind.

Landau'sche O-Notation



$f = O(g)$ (in Worten: f wächst nicht schneller als g), wenn $\exists c \in I\!R^+, n_0 \in I\!N, \forall n \geq n_0 : f(n) \leq cg(n)$.

$f = \Omega(g)$ (f wächst mindestens so schnell wie g), wenn $g = O(f)$ ist.

$f = \Theta(g)$ (f und g sind von der gleichen Wachstumsordnung), wenn $f = O(g)$ und $g = O(f)$ gelten.

$f = o(g)$ (f wächst langsamer als g), wenn $f(n)/g(n)$ eine Nullfolge ist.

$f = \omega(g)$ (f wächst schneller als g), wenn $g = o(f)$ ist.

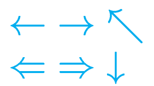
In Ottmann/Widmayer

$$O(g) = \{f | \exists a > 0, b > 0, \forall n : f(n) \leq ag(n) + b\}$$

und

$$\Omega(g) = \{g | \exists c > 0, \exists \infty \text{ viele } n : f(n) \geq cg(n)\}$$

Bemerkungen



Schreibweise: $f = O(g)$ statt $f \in O(g)$

Gleichungsfolgen: In der O-Notation müssen stets von links nach rechts gelesen werden. So ist

$$10n^2 + n = O(n^2) = o(n^3)$$

sinnvoll, während

$$10n^2 + n = o(n^2) = O(n^3)$$

ebenso sinnlos ist wie

$$10n^2 + n = O(n^2) = \Omega(n^2)$$

Minimalität: Die angegebene Größenordnung muß nicht minimal gewählt sein (s.o.)

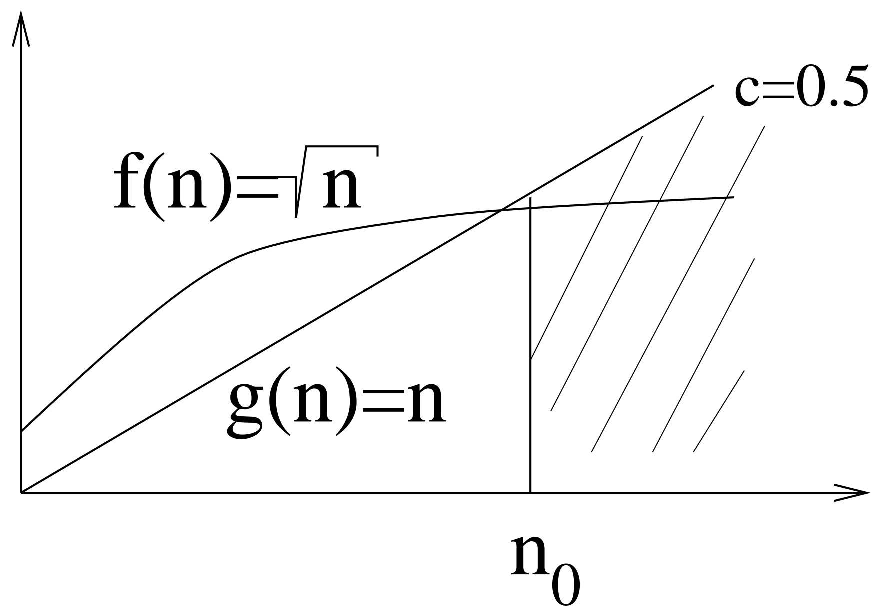
Asymptotik: Wie groß n sein muß bleibt unklar

Konstanten: Die Konstanten c und n_0 (bzw. a und b) haben für kleine n großen Einfluß.

$$f(n) \in O(g(n))$$

$$\Leftrightarrow \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \forall n \geq n_0 : f(n) \leq cg(n)$$

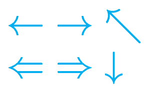
$$\Leftarrow \lim_{n \to \infty} f(n)/g(n) = c$$



Die Regel von L'Hospital

$$\lim_{n \to \infty} f(n)/g(n) = c \Leftarrow \lim_{n \to \infty} f'(n)/g'(n) = c$$

Weitere Eigenschaften



1. Für monoton steigende Funktionen $f \not\equiv 0$ sind beide Definitionen für Groß-O äquivalent.

2. Es gibt Funktionen in denen beide Definitionen von Ω voneinander abweichen (erste Definition ist schärfer).

3. Für alle k ist $n^k \in o(2^n)$

4. Es seien p_1 und p_2 Polynome vom Grad d_1 bzw. d_2, wobei die Koeffizienten von n^{d_1} und n^{d_2} positiv sind. Dann gilt:

   a) $p_1 \in \Theta(p_2) \Leftrightarrow d_1 = d_2$

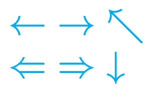
   b) $p_1 \in o(p_2) \Leftrightarrow d_1 < d_2$

   b) $p_1 \in \omega(p_2) \Leftrightarrow d_1 > d_2$

5. Für alle $k > 0$ und $\epsilon > 0$ gilt $\log^k n = o(n^\epsilon)$

6. $2^{n/2} = o(2^n)$

Beweis: Übungsaufgabe. Gleichzeitig ein Beleg, daß die Analysis-Vorlesung Anwendung hat.



Einfache Regeln:
$f = O(f)$
$O(O(f)) = O(f)$
$kO(f) = O(f)$ für konstantes k
$O(f + k) = O(f)$ für konstantes k

Additionsregel: $O(f) + O(g) = O(\max\{f, g\})$
Multiplikationsregel: $O(f) \cdot O(g) = O(f \cdot g)$.

Beweis (Additionsregel und Multiplikationsregel): Seien $\exists c_1, c_2 \in I\!R^+$ und $n_1, n_2 \in I\!N$ mit $T_1(n) \leq c_1 f(n)$ für alle $n \geq n_1$ und $T_2(n) \leq c_2 g(n)$ für alle $n \geq n_2$.

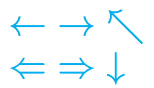
Dann gilt für alle $n \geq n_0$:

Additionsregel: $T_1(n) + T_2(n) \leq c \cdot \max\{f(n), g(n)\}$, wobei $c = c_1 + c_2$ und $n_0 = \max\{n_1, n_2\}$ ist.

Multiplikationsregel: $T_1(n)T_2(n) \leq cf(n)g(n)$, wobei $c = c_1 c_2$ und $n_0 = \max\{n_1, n_2\}$ ist.

Additionsregel für die Komplexität der Hintereinanderausführung der Programme.
Multiplikationsregel für die Komplexität von ineinandergeschachtelten Schleifen.

Hierarchie von Größenordnungen



$O(1)$: konstante Funktionen

$O(\log n)$: Logarithmische Funktionen

$O(\log^2 n)$: Quadratisch logarithmische Funktionen

$O(n)$: Lineare Funktionen

$O(n \log n)$: - keine spezielle Bezeichnung -

$O(n^2)$: quadratische Funktionen

$O(n^3)$: kubische Funktionen

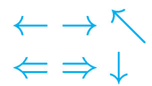
$O(n^k)$: polynomielle Funktionen (für k fest, beliebig)

genauer: f heißt polynomiell beschränkt, wenn es ein Polynom p mit $f = O(p)$ gibt.

$O(2^n)$: exponentielle Funktionen

genauer: f wächst exponentiell, wenn es ein $\epsilon > 0$ mit $f = \Theta(2^{n^\epsilon})$ gibt.

Skalierbarkeiten



Annahme: Rechenschritt 0.001 Sekunden. Maximale Eingabelänge bei gegebener Rechenzeit.

Laufzeit T(n)	1 Sekunde	1 Minute	1 Stunde
n	1000	60000	3600000
$n \log n$	140	4895	204094
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Maximale Eingabelänge in Abhängigkeit der Technologie ($\log 10p = \log p + \log 10 \approx \log p$, $3.16^2 \approx 10, 2.15^3 \approx 10, 2^{3.3} \approx 10$)

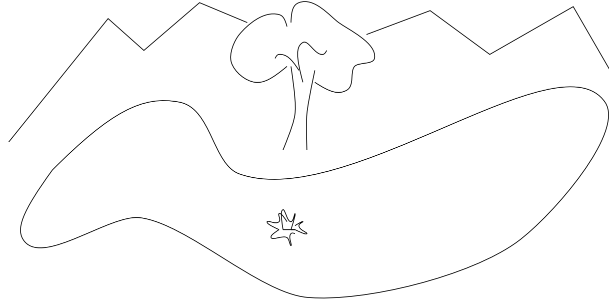
Laufzeit T(n)	alt	neu (d.h. 10-mal schneller)
n	p	$10p$
$n \log n$	p	(fast 10)p
n^2	p	$3.16p$
n^3	p	$2.15p$
2^n	p	$p + 3.3$

$100n < \lceil n^3/125 \rceil$ ab $n > 111$

$10n < .5n \log n$ ab $n > 2^{20}$

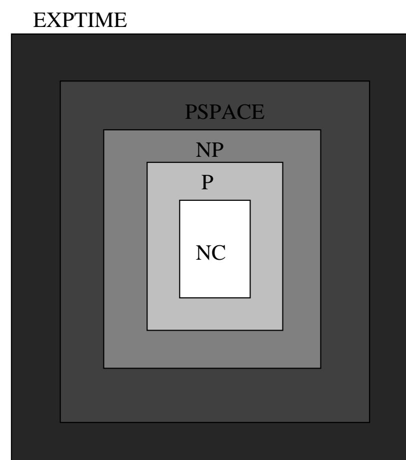
Exponentielle Algorithmen

Seerosenbeispiel:

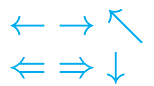


Schachbrett (Reis), Zinseszins, Börse (Kybernetik), Zellteilung (Qualitativ-Quantitativ), Radioaktiver Zerfall (Exponentiell langsam)

In der Theoretischen Informatik wird die Grauzone zur Lösung von Problemen durch Algorithmen mit garantiert polynomiellen und garantiert exponentiellen Laufzeiten analysiert.



Zeitaufwand für Programmstück A



1. A ist einfache Zuweisung (read, write, ...):
$cost(A) = const \in O(1)$

2. A ist Folge von Anweisung: Additionsregel anwenden

3. A ist `if`-Anweisung:

a) `if (cond) B;` $cost(A) = cost(cond) + cost(B)$

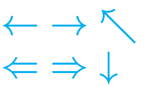
b) `if (cond) B; else C`
$cost(A) = cost(cond) + \max\{cost(B), cost(C)\}$

4. A ist eine Schleife:

$cost(A) = \sum_{\text{Umlauf } i}\{cost(\text{Anweisungen } i) + cost(\text{Terminierungsbedingung } i)\}$

oft

$cost(A) = \#\text{Umläufe} \cdot \{cost(\text{Anweisungen}) + cost(\text{Terminierungsbedingung})\}$



Analyse von Bubble-Sort: Sortiert die Zahlen im Array `a[1..n]` durch wiederholtes Vertauschen

Algorithmus:

```
repeat
  for i= 1..n-1
    if (a[i] > a[i+1])
      swap(a[i],a[i+1])
until (keine Vertauschungen mehr notwendig)
```

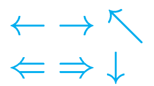
Abschätzen der Anzahl von Vergleichen:

Worst Case Innere Schleife: $n-1$ Vergleiche, maximal n Durchläufe (Minimum an `a[n]`) der äusseren Schleife $\Rightarrow O(n^2)$

Best Case Array ist bereits sortiert: Ende nach einem Durchlauf $\Rightarrow O(n)$

Average Case Fakt: Minimum wandert pro Durchlauf der äußeren Schleife 1 Schritt nach links. Annahme: Jede Position für Min gleichwahrscheinlich. $\Rightarrow$ im Durchschnitt werden $\frac{1 \cdot O(n) + 2 \cdot O(n) + \ldots + n \cdot O(n)}{n} = O(n^2)$ viele Vergleiche ausgeführt.

Das Maxsummenproblem

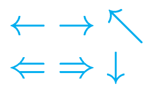


Problem: Finde ein Index-Paar (i, j) in einem Array `a[1..n]` von ganzen Zahlen für das $f(i, j) = a_i + \ldots + a_j$ maximal ist. Als Rechenschritte zählen arithmetische Operationen und Vergleiche.

Der naive Algorithmus:

Es werden alle $f(i, j)$ Werte berechnet werden und daraus der maximale f-Wert ermittelt. Offensichtlich genügen zur Berechung von $f(i, j)$ genau $j - i$ Additionen. Der Algoritmus startet mit $\mathsf{max} \leftarrow f(1, 1)$ und aktualisiert max wenn nötig.

Analyse Naiver Algorithmus



Es gibt j Paare der Form $(\cdot, j)$

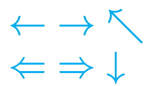
Vergleiche:
$V_1(n) = (\sum_{1 \leq j \leq n} j) - 1 = n(n+1)/2 - 1$

Additionen:

$$
\begin{aligned}
A_1(n) &= \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} (j - i) \\
&= \sum_{1 \leq i \leq n} \sum_{1 \leq k \leq n-i} k \\
&= \sum_{1 \leq i \leq n} \sum_{1 \leq k \leq i} k \\
&= \sum_{1 \leq i \leq n} i(i+1)/2 \\
&= \frac{1}{2}\left(\sum_{1 \leq i \leq n} i^2 + \sum_{1 \leq i \leq n} i\right) \\
&= \frac{1}{2}\left(\frac{1}{6}(n-1)n(2(n-1)+1) + \frac{1}{2}(n-1)n\right) \\
&= 1/6n^3 - 1/6n
\end{aligned}
$$

Zusammen:
$T_1(n) = V_1(n) + A_1(n) = 1/6n^3 + 1/2n^2 + 1/3n - 1$

Der etwas bessere Algorithmus



Der naive Ansatz berechnet $a_1 + a_2$ für $f(1,2), f(1,3), \ldots, f(1,n)$, also $(n-1)$-mal.

Besser geht's mit folgender Erkenntnis. Es gilt:

$$f(i, j+1) = f(i,j) + a_{j+1}$$

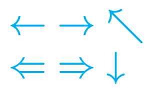
Damit braucht man für alle $f(i,\cdot)$-Werte genau $(n-i)$ Additionen.

Vergleiche: $V_2(n) = V_1(n) = n(n+1)/2 - 1$

Additionen:
$A_2(n) = \sum_{1 \le i \le n}(n-i) = \sum_{1 \le k \le n-1} k = n(n-1)/2$

Zusammen: $T_2(n) = V_2(n) + A_2(n) = n^2 - 1.$

Divide-And-Conquer



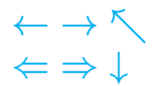
Annahme: $n = 2^k$. Unterteilung der zu untersuchenden Paare (i, j) in drei Klassen:

- $1 \leq i, j \leq n/2$
- $1 \leq i \leq n/2 < j \leq n$
- $n/2 < i \leq j \leq n$

Wenn wir die Probleme für die drei Klassen gelöst haben, erhalten wir den “Gesamtsieger” in 3 Vergleichen.

Das erste und dritte Problem sind vom gleichen Typ, wie das Ausgangsproblem und werden rekursiv mit dem gleichen Ansatz gelößt.

Analyse Divide-and-Conquer



Für das zweite Problem gibt es eine effiziente direkte Lösung. Betrachte:

$g(i) = a_i + \ldots + a_{n/2}$ und $h(j) = a_{n/2+1} + \ldots + a_j$

Dann gilt: $f(i,j) = g(i) + h(j)$ und: Um f zu maximieren, reicht es aus g und h einzeln zu maximieren.

Berechne nacheinander

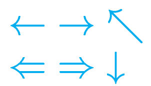
$$\begin{aligned} g(n/2) &= a_{n/2}, \\ g(n/2-1) &= a_{n/2-1} + a_{n/2}, \\ \ldots, \\ g(1) &= a_1 + \ldots + a_{n/2} \end{aligned}$$

und danach den max. g Wert in $(n/2-1)$ Vergleichen und $(n/2-1)$ Additionen.

Berechne den maximalen h analog.

Damit ergeben sich für das zweite Problem insgesamt $n-1$ Additionen und $n-2$ Vergleiche, also $2n-3$ Operationen, obwohl die Klasse $n^2/4$ Paare (i,j) enthält.

Analyse Divide-and-Conquer



Rekursionsgleichung (2 Vergleiche um Maximum der Gruppen zu finden)

$$\begin{aligned}
T_3(1) &= 0 \\
T_3(2^k) &= 2T_3(2^{k-1}) + 2 \cdot 2^k - 1 \\
&= 4T_3(2^{k-2}) + 2(2 \cdot 2^{k-1} - 1) + 2 \cdot 2^k - 1 \\
&= 4T_3(2^{k-2}) + (2^{k+1} - 2) + (2^{k+1} - 1) \\
&= 8T_3(2^{k-1}) + (2^{k+1} - 4) + (2^{k+1} - 2) \\
&\quad +(2^{k+1} - 1) \\
&= \ldots
\end{aligned}$$

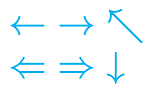
Nun raten wir die Lösung der Rekursionsgleichung

$$T_3(2^k) = 2^l T_3(2^{k-l}) + \textstyle\sum_{1 \le i \le l}(2^{k+1} - 2^{i-1})$$

und verifizieren die Vermutung mit einem Induktionsbeweis. Wir benutzen, da wir $T_3(1)$ kennen, die spezielle Lösung für $l = k$

$$\begin{aligned}
T_3(2^k) &= 0 + \sum_{1 \le i \le k} (2^{k+1} - 2^{i-1}) \\
&= 2k2^k - (2^k - 1) \\
&= (2k - 1)2^k + 1 = (2 \log n - 1)n + 1
\end{aligned}$$

Der clevere Algorithmus



Scanline-Prinzip: Wir wollen Problem in $[1..n]$ lösen und wollen die Eingabe nur einmal von links nach rechts lesen. Dafür müssen wir genügend Information bereitstellen:

Wir verwalten dabei nach dem Lesen von a_k in max den größten Wert von $f(i,j)$ aller Paare (i,j) für $1 \leq i \leq j \leq k$.

Für $k = 1$ setzen wir max auf a_1.

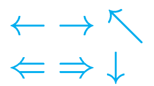
Konkurrenten sind Paare (i,j) mit $k < i$ und Paare (i,j) mit $i \leq k < j$. Über die erste Menge wissen wir noch nichts, da wir erst $a_1, \ldots, a_k$ gelesen haben. Vom Divide-and-Conquer Algorithmus wissen wir, daß für die zweite Menge nur die g-Werte

$$g(i) = a_i + \ldots + a_k$$

interessant sind. Deshalb verwalten wir zusätzlich

$$\text{max}^* = \text{max}_{1 \leq i \leq k}\{g(i) \mid \text{mit } g(i) = a_i + \ldots + a_k\}.$$

Aktualisierung und Analyse



Sei nun a_{k+1} gelesen. Wir erhalten die neuen g-Werte

$g_{neu}(i) = g_{alt}(i) + a_{k+1}$, für $1 \leq i \leq k$
$g_{neu}(i) = a_{k+1}$, für $i = k + 1$

Also $\max^*_{neu} = \max\{\max^*_{alt} + a_{k+1}, a_{k+1}\}$

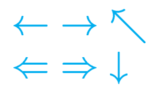
Für $\max_{neu}$ kommen folgende Paare (i, j) in Frage:

$1 \leq i \leq j \leq k$ (maximaler Wert $\max_{alt}$)
$1 \leq i \leq k, j = k + 1$ (maximaler Wert $\max^*_{neu}$)

Also kann $\max_{neu}$ mit einem Vergleich aus $\max_{alt}$ und $\max^*_{neu}$ berechnet werden.

Bei der Verarbeitung von a_k, $2 \leq k \leq n$, genügen also 3 Operationen, demnach ist $T_4(n) = 3n - 3$.

Zusammestellung der Ergebnisse



$$T_1(n) = 1/6n^3 + 1/2n^2 + 1/3n - 1$$
$$T_2(n) = n^2 - 1$$
$$T_3(n) = (2\log n - 1)n + 1$$
$$T_4(n) = 3n - 3$$

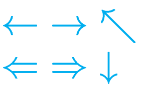
n	$T_1(n)$	$T_2(n)$	$T_3(n)$	$T_4(n)$
2^2	19	15	13	9
2^4	815	255	113	45
2^6	45759	4095	705	189
2^8	2829055	65535	3841	765
2^{10}	179481599	1048575	19457	3069
2^{15}	$> 5 \cdot 10^{12}$	$\approx 10^9$	950273	98301

Die Rechenzeiten richten sich nach den Größenordnungen $(n^3, n^2, n\log n, n)$ und nicht nach den Vorfaktoren.

Es sollte sich das beruhigende Gefühl breitmachen, daß es sich lohnt clever zu sein.

Ottmann/Widmayer: Berechnungen in O-Notationen

7 Mastertheorem



Eine Kochbuchmethode zum Bestimmen des asymptotischen Wachstums von Funktionen, die durch Rekursionsgleichungen gegeben sind:

$$T(n) = \begin{cases} c & \text{falls } n \leq d \\ aT(n/b) + f(n) & \text{falls } n \geq d. \end{cases}$$

mit ganzzahligen Konstanten $c \geq 1$ und $d \geq 1$ und reelen Konstanten $a \geq 1$ und $b > 1$ und einer reellen Funktion $f(n) \geq 0$ für $n \geq d$. Unterscheide drei Fälle:

1. $(\exists \epsilon > 0)\ f(n) \in O(n^{\log_b a - \epsilon}) \Rightarrow$

$$T(n) \in \Theta(n^{\log_b a})$$

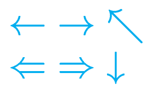
2. $(\exists k \geq 0)\ f(n) \in \Theta(n^{\log_b a} (\log n)^k) \Rightarrow$

$$T(n) \in \Theta(n^{\log_b a} (\log n)^{k+1})$$

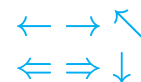
3. $(\exists \epsilon > 0)\ f(n) \in O(n^{\log_b a + \epsilon})$ und
   $(\exists \delta < 1)\ (n \geq d)\ af(n/b) \leq \delta f(n) \Rightarrow$

$$T(n) \in \Theta(f(n))$$

Beispiele Mastertheorem



1. $T(n) = 4T(n/2) + n$; $a = 4$, $b = 2$, $f(n) = n$.
   Mit $n^{\log_2 4} = n^2$ ist
   $$T(n) = \Theta(n^2),$$
   da Fall 1 gilt: $n = f(n) \in O(n^{2-\epsilon})$ für $\epsilon = 1$.
   Genau: $T(n) = 2n^2 - n$
2. $T(n) = T(n/2) + 1$; $a = 1$, $b = 2$, $f(n) = 1$. Mit $n^{\log_2 1} = 1$ ist
   $$T(n) = \Theta(\log n),$$
   da Fall 2 mit $k = 0$ gilt.
   Genau: $T(n) = \log n + 1$
3. $T(n) = 2T(n/2) + n \log n$; $a = 2$, $b = 2$, $f(n) = n \log n$. Mit $n^{\log_2 2} = n$ ist
   $$T(n) = \Theta(n \log^2 n),$$
   da Fall 2 mit $k = 1$ gilt.
4. $T(n) = T(n/3) + n \log n$; $a = 1$, $b = 3$, $f(n) = n$. Mit $n^{\log_3 1} = 1$ ist
   $$T(n) = \Theta(n),$$
   da Fall 3 gilt: $f(n) = \Omega(n^{0+\epsilon})$ für $\epsilon = 1$ und $af(n/b) = n/3 = f(n)/3$ für $\delta = 1/3$



Überblick, 2

Aktualisierung und Analyse, 29
Analyse Divide-and-Conquer, 26, 27
Analyse Naiver Algorithmus, 23

Beispiel, 9
Beispiele, 21
Beispiele Mastertheorem, 32
Bemerkungen, 13

Das Maxsummenproblem, 22
Der clevere Algorithmus, 28
Der etwas bessere Algorithmus, 24
Der Mittlere Fall, 6
Divide-And-Conquer, 25

Exponentielle Algorithmen, 19

Funktionenklassen, 11

Hierarchie von Größenordnungen, 17

Kostenmaße, 8

Landau'sche O-Notation, 12
Laufzeitanalyse, 5
Leistungsverhalten von Algorithmen, 4

Mastertheorem, 31
Messung des Leistungsverhaltens, 7
Mittlerer Fall, 10

O-Kalkül, 16

Skalierbarkeiten, 18

Themen, 3

Visualisierung, 14

Weitere Eigenschaften, 15

Zeitaufwand für Programmstück A, 20
Zusammestellung der Ergebnisse, 30