

2. Effizienz von Algorithmen

Effizienz von Algorithmen, Random Access Machine Modell, Funktionenwachstum, Asymptotik [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Effizienz von Algorithmen

Ziele

- Laufzeitverhalten eines Algorithmus maschinenunabhängig quantifizieren.
- Effizienz von Algorithmen vergleichen.
- Abhängigkeit von der Eingabegrösse verstehen.

73

74

Technologiemodell

Random Access Machine (RAM)

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit.
- Elementare Operationen: Rechenoperation (+, -, ·, ...) , Vergleichsoperationen, Zuweisung / Kopieroperation, Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.
- Datentypen: Fundamentaltypen wie grössenbeschränkte Ganzzahl oder Fließkommazahl.

75

Grösse der Eingabedaten

Typisch: Anzahl Eingabeobjekte (von fundamentalem Typ).
Oftmals: Anzahl Bits für eine *vernünftige* / *kostengünstige* Repräsentation der Daten.

76

Asymptotisches Verhalten

Genauere Laufzeit lässt sich selbst für kleine Eingabedaten kaum voraussagen.

- Betrachten das asymptotische Verhalten eines Algorithmus.
- Ignorieren alle konstanten Faktoren.

Beispiel

Eine Operation mit Kosten 20 ist genauso gut wie eine mit Kosten 1.
Lineares Wachstum mit Steigung 5 ist genauso gut wie lineares Wachstum mit Steigung 1.

77

Oberflächlich

Verwende die asymptotische Notation zur Kennzeichnung der Laufzeit von Algorithmen

Wir schreiben $\Theta(n^2)$ und meinen, dass der Algorithmus sich für grosse n wie n^2 verhält: verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

79

2.2 Funktionenwachstum

\mathcal{O} , Θ , Ω [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

78

Genauer: Asymptotische obere Schranke

Gegeben: Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

Definition:

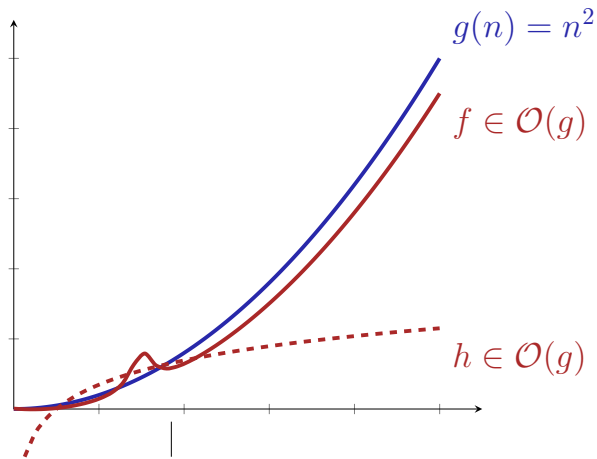
$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$$

Schreibweise:

$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

80

Anschauung



Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

81

82

Eigenschaft

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

Umkehrung: Asymptotische untere Schranke

Gegeben: Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

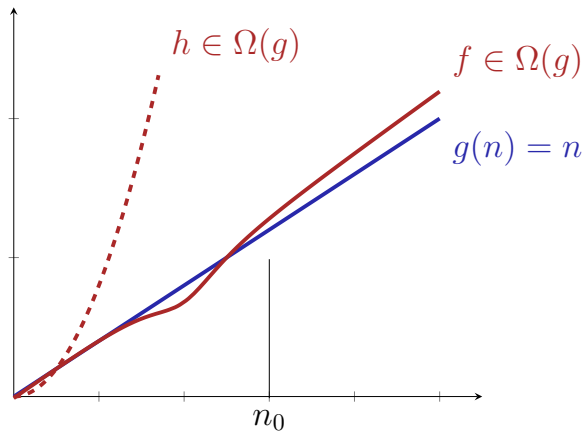
Definition:

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}$$

83

84

Beispiel



Asymptotisch scharfe Schranke

Gegeben Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

Definition:

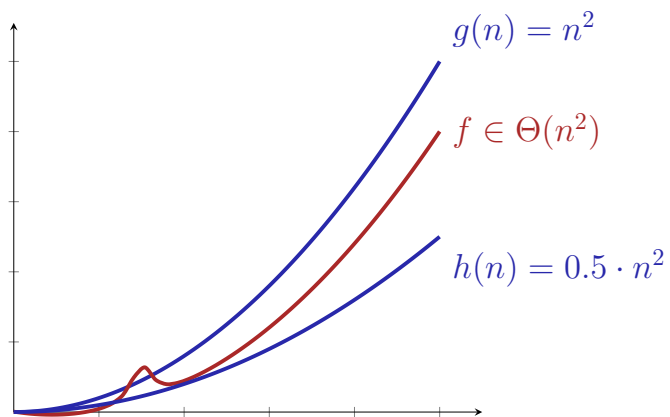
$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Einfache, geschlossene Form: Übung.

85

86

Beispiel



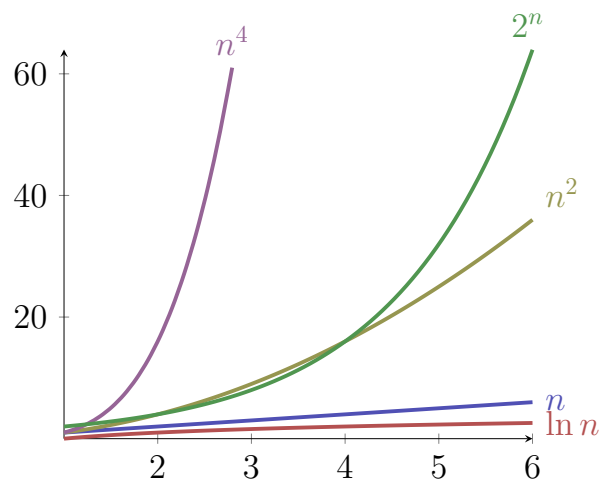
Wachstumsbezeichnungen

$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log \log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n \log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(2^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

87

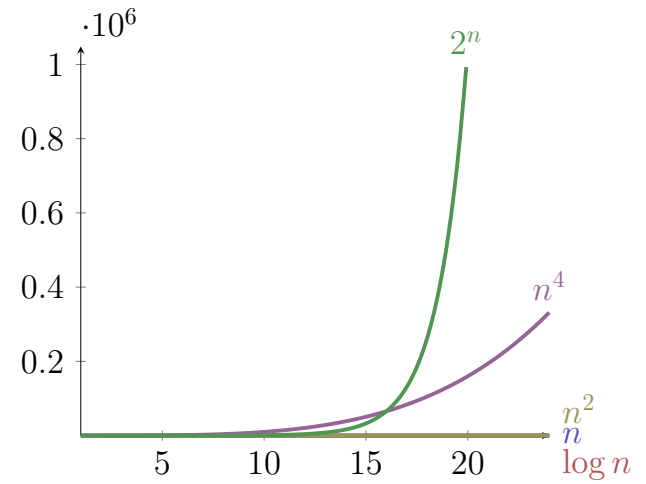
88

Kleine n



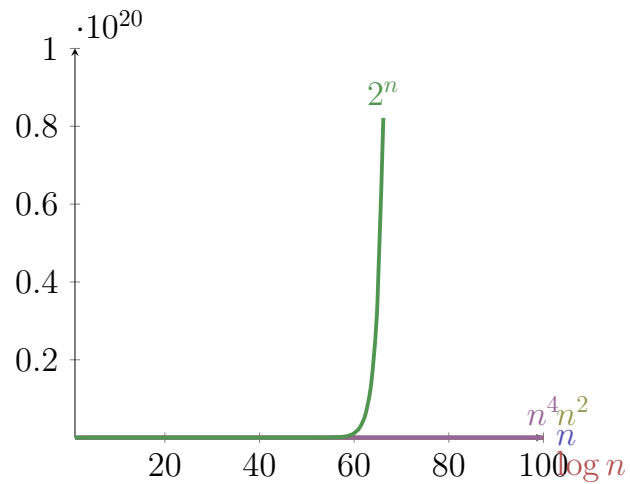
89

Grössere n



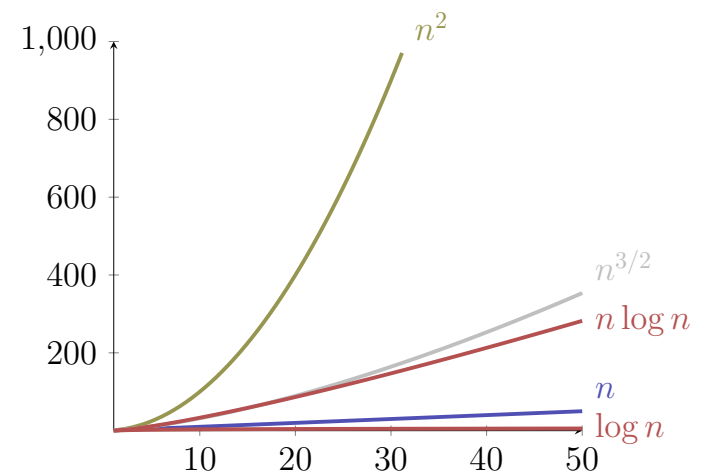
90

“Grosse” n



91

Logarithmen!



92

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$	10^{14} Jahrh.	$\approx \infty$	$\approx \infty$	$\approx \infty$

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine! Wenn ich heute ein Problem der Grösse n lösen kann, dann kann ich mit einer 10 oder 100 mal so schnellen Maschine...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \rightarrow n^{10}$	$n \rightarrow n^{100}$
n	$n \rightarrow 10 \cdot n$	$n \rightarrow 100 \cdot n$
n^2	$n \rightarrow 3.16 \cdot n$	$n \rightarrow 10 \cdot n$
2^n	$n \rightarrow n + 3.32$	$n \rightarrow n + 6.64$

93

94

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
 $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
Konstanten weglassen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \xrightarrow{n \rightarrow \infty} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ ist korrekt
- $\Theta(n) \subseteq \Theta(n^2)$ ist falsch: $n \notin \Omega(n^2) \supset \Theta(n^2)$

Nützlich

Theorem

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ zwei Funktionen. Dann gilt:

- 1 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subsetneq \mathcal{O}(g)$.
- 2 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C > 0$ (C konstant) $\Rightarrow f \in \Theta(g)$.
- 3 $\frac{f(n)}{g(n)} \xrightarrow{n \rightarrow \infty} \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

95

96

Zur Notation

Übliche Schreibweise

$$f = \mathcal{O}(g)$$

ist zu verstehen als $f \in \mathcal{O}(g)$.

Es gilt nämlich

$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

Beispiel

$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$ aber natürlich $n \neq n^2$.

97

Komplexität

Komplexität eines Problems P : minimale (asymptotische) Kosten über alle Algorithmen A , die P lösen.

Komplexität der Elementarmultiplikation zweier Zahlen der Länge n ist $\Omega(n)$ und $\mathcal{O}(n^{\log_3 2})$ (Karatsuba Ofman).

Exemplarisch:

Problem	Komplexität	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
		↑	↑	↑
Algorithmus	Kosten ²	$3n - 4$	$\mathcal{O}(n)$	$\Theta(n^2)$
		↓	↓	↓
Programm	Laufzeit	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$

²Anzahl Elementaroperationen

99

Algorithmen, Programme und Laufzeit

Programm: Konkrete Implementation eines Algorithmus.

Laufzeit des Programmes: messbarer Wert auf einer konkreten Maschine. Kann sowohl nach oben, wie auch nach unten abgeschätzt werden.

Beispiel

Rechner mit 3 GHz. Maximale Anzahl Operationen pro Taktzyklus (z.B. 8). \Rightarrow untere Schranke.

Einzelne Operation dauert mit Sicherheit nie länger als ein Tag \Rightarrow obere Schranke.

Asymptotisch gesehen stimmen die Schranken überein.

98

3. Algorithmenentwurf

Maximum Subarray Problem [Ottman/Widmayer, Kap. 1.3]

Divide and Conquer [Ottman/Widmayer, Kap. 1.2.2. S.9; Cormen et al, Kap. 4-4.1]

100

Algorithmenentwurf

Induktive Entwicklung eines Algorithmus: Zerlegung in Teilprobleme, Verwendung der Lösungen der Teilproblem zum Finden der endgültigen Lösung.

Ziel: Entwicklung des asymptotisch effizientesten (korrekten) Algorithmus.

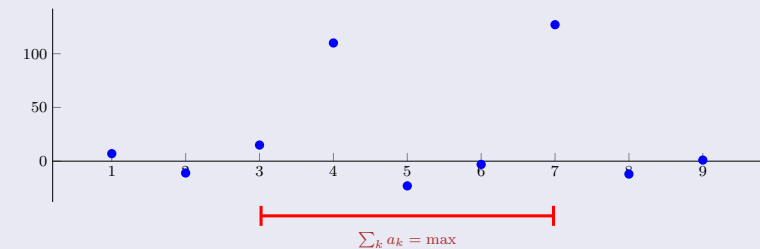
Effizienz hinsichtlich der Laufzeitkosten (# Elementaroperationen) oder / und Speicherbedarf.

Maximum Subarray Problem

Gegeben: ein Array von n rationalen Zahlen (a_1, \dots, a_n) .

Gesucht: Teilstück $[i, j]$, $1 \leq i \leq j \leq n$ mit maximaler positiver Summe $\sum_{k=i}^j a_k$.

Beispiel: $a = (7, -11, 15, 110, -23, -3, 127, -12, 1)$



101

102

Naiver Maximum Subarray Algorithmus

Input : Eine Folge von n Zahlen (a_1, a_2, \dots, a_n)

Output : I, J mit $\sum_{k=I}^J a_k$ maximal.

$M \leftarrow 0; I \leftarrow 1; J \leftarrow 0$

for $i \in \{1, \dots, n\}$ **do**

for $j \in \{i, \dots, n\}$ **do**

$m = \sum_{k=i}^j a_k$

if $m > M$ **then**

$M \leftarrow m; I \leftarrow i; J \leftarrow j$

return I, J

Analyse

Theorem

Der naive Algorithmus für das Maximum Subarray Problem führt $\Theta(n^3)$ Additionen durch.

Beweis:

$$\begin{aligned} \sum_{i=1}^n \sum_{j=i}^n (j-i+1) &= \sum_{i=1}^n \sum_{j=0}^{n-i} (j+1) = \sum_{i=1}^n \sum_{j=1}^{n-i+1} j = \sum_{i=1}^n \frac{(n-i+1)(n-i+2)}{2} \\ &= \sum_{i=0}^n \frac{i \cdot (i+1)}{2} = \frac{1}{2} \left(\sum_{i=1}^n i^2 + \sum_{i=1}^n i \right) \\ &= \frac{1}{2} \left(\frac{n(2n+1)(n+1)}{6} + \frac{n(n+1)}{2} \right) = \frac{n^3 + 3n^2 + 2n}{6} = \Theta(n^3). \end{aligned}$$

103

104

Beobachtung

$$\sum_{k=i}^j a_k = \underbrace{\left(\sum_{k=1}^j a_k \right)}_{S_j} - \underbrace{\left(\sum_{k=1}^{i-1} a_k \right)}_{S_{i-1}}$$

Präfixsummen

$$S_i := \sum_{k=1}^i a_k.$$

Maximum Subarray Algorithmus mit Präfixsummen

Input : Eine Folge von n Zahlen (a_1, a_2, \dots, a_n)

Output : I, J mit $\sum_{k=I}^J a_k$ maximal.

```

 $S_0 \leftarrow 0$ 
for  $i \in \{1, \dots, n\}$  do // Präfixsumme
   $S_i \leftarrow S_{i-1} + a_i$ 
 $M \leftarrow 0; I \leftarrow 1; J \leftarrow 0$ 
for  $i \in \{1, \dots, n\}$  do
  for  $j \in \{i, \dots, n\}$  do
     $m = S_j - S_{i-1}$ 
    if  $m > M$  then
       $M \leftarrow m; I \leftarrow i; J \leftarrow j$ 
    
```

105

106

Analyse

Theorem

Der Präfixsummen Algorithmus für das Maximum Subarray Problem führt $\Theta(n^2)$ Additionen und Subtraktionen durch.

Beweis:

$$\sum_{i=1}^n 1 + \sum_{i=1}^n \sum_{j=i}^n 1 = n + \sum_{i=1}^n (n - i + 1) = n + \sum_{i=1}^n i = \Theta(n^2)$$

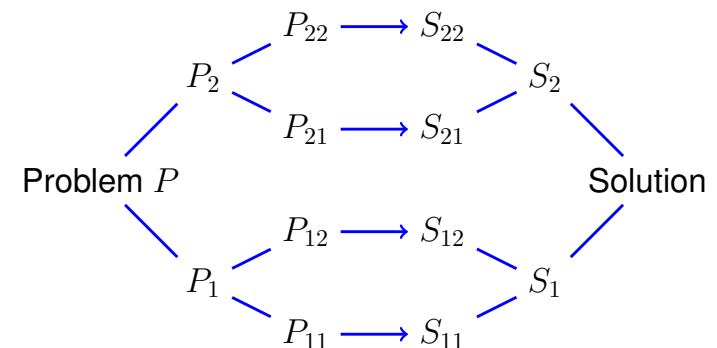
■

107

divide et impera

Teile und (be)herrsche (engl. divide and conquer)

Zerlege das Problem in Teilprobleme, deren Lösung zur vereinfachten Lösung des Gesamtproblems beitragen.



108

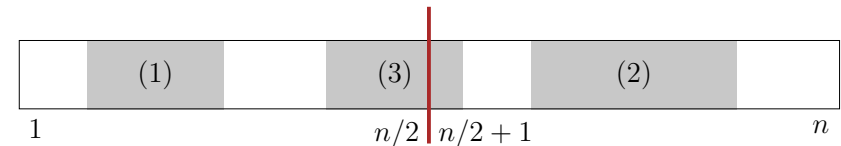
Maximum Subarray – Divide

- Divide: Teile das Problem in zwei (annähernd) gleiche Hälften auf:
 $(a_1, \dots, a_n) = (a_1, \dots, a_{\lfloor n/2 \rfloor}, a_{\lfloor n/2 \rfloor + 1}, \dots, a_n)$
- Vereinfachende Annahme: $n = 2^k$ für ein $k \in \mathbb{N}$.

Maximum Subarray – Conquer

Sind i, j die Indizes einer Lösung \Rightarrow Fallunterscheidung:

- 1 Lösung in linker Hälfte $1 \leq i \leq j \leq n/2 \Rightarrow$ Rekursion (linke Hälfte)
- 2 Lösung in rechter Hälfte $n/2 < i \leq j \leq n \Rightarrow$ Rekursion (rechte Hälfte)
- 3 Lösung in der Mitte $1 \leq i \leq n/2 < j \leq n \Rightarrow$ Nachfolgende Beobachtung



109

110

Maximum Subarray – Beobachtung

Annahme: Lösung in der Mitte $1 \leq i \leq n/2 < j \leq n$

$$\begin{aligned}
 S_{\max} &= \max_{\substack{1 \leq i \leq n/2 \\ n/2 < j \leq n}} \sum_{k=i}^j a_k = \max_{\substack{1 \leq i \leq n/2 \\ n/2 < j \leq n}} \left(\sum_{k=i}^{n/2} a_k + \sum_{k=n/2+1}^j a_k \right) \\
 &= \max_{1 \leq i \leq n/2} \sum_{k=i}^{n/2} a_k + \max_{n/2 < j \leq n} \sum_{k=n/2+1}^j a_k \\
 &= \max_{1 \leq i \leq n/2} \underbrace{S_{n/2} - S_{i-1}}_{\text{Suffixsumme}} + \max_{n/2 < j \leq n} \underbrace{S_j - S_{n/2}}_{\text{Präfixsumme}}
 \end{aligned}$$

Maximum Subarray Divide and Conquer Algorithmus

Input : Eine Folge von n Zahlen (a_1, a_2, \dots, a_n)

Output : Maximales $\sum_{k=i}^j a_k$.

if $n = 1$ **then**

return $\max\{a_1, 0\}$

else

 Unterteile $a = (a_1, \dots, a_n)$ in $A_1 = (a_1, \dots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \dots, a_n)$

 Berechne rekursiv beste Lösung W_1 in A_1

 Berechne rekursiv beste Lösung W_2 in A_2

 Berechne grösste Suffixsumme S in A_1

 Berechne grösste Präfixsumme P in A_2

 Setze $W_3 \leftarrow S + P$

return $\max\{W_1, W_2, W_3\}$

111

112

Analyse

Theorem

Der Divide and Conquer Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n \log n)$ viele Additionen und Vergleiche durch.

Analyse

Input : Eine Folge von n Zahlen (a_1, a_2, \dots, a_n)

Output : Maximales $\sum_{k=i'}^{j'} a_k$.

if $n = 1$ **then**

$\Theta(1)$ **return** $\max\{a_1, 0\}$

else

$\Theta(1)$ Unterteile $a = (a_1, \dots, a_n)$ in $A_1 = (a_1, \dots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \dots, a_n)$

$T(n/2)$ Berechne rekursiv beste Lösung W_1 in A_1

$T(n/2)$ Berechne rekursiv beste Lösung W_2 in A_2

$\Theta(n)$ Berechne grösste Suffixsumme S in A_1

$\Theta(n)$ Berechne grösste Präfixsumme P in A_2

$\Theta(1)$ Setze $W_3 \leftarrow S + P$

$\Theta(1)$ **return** $\max\{W_1, W_2, W_3\}$

113

114

Analyse

Rekursionsgleichung

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T(\frac{n}{2}) + a \cdot n & \text{falls } n > 1 \end{cases}$$

Analyse

Mit $n = 2^k$:

$$\bar{T}(k) = \begin{cases} c & \text{falls } k = 0 \\ 2\bar{T}(k-1) + a \cdot 2^k & \text{falls } k > 0 \end{cases}$$

Lösung:

$$\bar{T}(k) = 2^k \cdot c + \sum_{i=0}^{k-1} 2^i \cdot a \cdot 2^{k-i} = c \cdot 2^k + a \cdot k \cdot 2^k = \Theta(k \cdot 2^k)$$

also

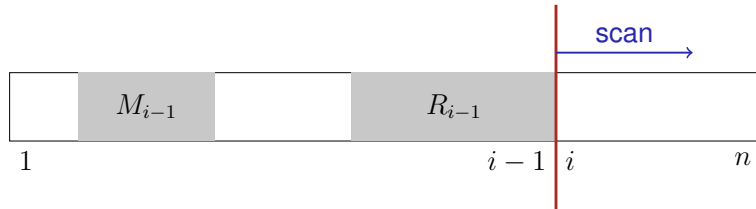
$$T(n) = \Theta(n \log n)$$

115

116

Maximum Subarray Sum Problem – Induktiv

Annahme: Maximaler Wert M_{i-1} der Subarraysumme für (a_1, \dots, a_{i-1}) ($1 < i \leq n$) bekannt.



a_i : erzeugt höchstens Intervall am Rand (Präfixsumme).

$$R_{i-1} \Rightarrow R_i = \max\{R_{i-1} + a_i, 0\}$$

117

Induktiver Maximum Subarray Algorithmus

Input : Eine Folge von n Zahlen (a_1, a_2, \dots, a_n) .

Output : $\max\{0, \max_{i,j} \sum_{k=i}^j a_k\}$.

$M \leftarrow 0$

$R \leftarrow 0$

for $i = 1 \dots n$ **do**

$R \leftarrow R + a_i$

if $R < 0$ **then**

$R \leftarrow 0$

if $R > M$ **then**

$M \leftarrow R$

return M ;

118

Analyse

Theorem

Der induktive Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n)$ viele Additionen und Vergleiche durch.

119

Komplexität des Problems?

Geht es besser als $\Theta(n)$?

Jeder korrekte Algorithmus für das Maximum Subarray Sum Problem muss jedes Element im Algorithmus betrachten.

Annahme: der Algorithmus betrachtet nicht a_i .

- 1 Lösung des Algorithmus enthält a_i . Wiederholen den Algorithmus mit genügend kleinem a_i , so dass die Lösung den Punkt nicht enthalten hätte dürfen.
- 2 Lösung des Algorithmus enthält a_i nicht. Wiederholen den Algorithmus mit genügend grossem a_i , so dass die Lösung a_i hätten enthalten müssen.

120

Komplexität des Maximum Subarray Sum Problems

Theorem

Das Maximum Subarray Sum Problem hat Komplexität $\Theta(n)$.

Beweis: Induktiver Algorithmus mit asymptotischer Laufzeit $\mathcal{O}(n)$.

Jeder Algorithmus hat Laufzeit $\Omega(n)$.

Somit ist die Komplexität $\Omega(n) \cap \mathcal{O}(n) = \Theta(n)$. ■