Algorithmen & Datenstrukturen

Herbst 2018

Vorlesung 5

Algorithmenentwurf: Maximum SubArray

Viele Algorithmen operieren "induktiv".
D.h. sie lösen zuerst, rekursiv, ein oder mehrere Probleme kleinerer Größe und konstruieren daraus die endgültige Lösung

Für ein Problem gibt es im allgemeinen mehrere Algorithmen. Uns interessiert der (asymptotisch) effizienteste.
- Laufzeit (Anzahl elementarer Ops)
- Speicherbedarf

Das Optimum ist die (Laufzeit/Speicher) Komplexität des Problems.

Problem: Maximum SubArray

gegeben: n Zahlen $a_1, \ldots, a_n \in \mathbb{Z}$
finde: Teilstück mit maximaler Summe ≥ 0,
also finde i, j mit $1 \leq i \leq j \leq n$
so daß $S = \sum_{k=i}^{j} a_k$ maximal ist

Falls alle Summen negativ sind, $S = 0$.

Beispiel: ($n = 8$)

7 −11 15 110 −23 −3 127 −1

Lösung durch Hingucken $S = 226$

z.B. es könnten Änderungen der Aktienkurse sein: Wann war der beste Zeitraum für Kauf (i) und Verkauf (j)?

Algorithmus 1 (naiv):

Idee: alle Intervalle ausprobieren

für $i = 1 \dots n$ (alle Anfänge)
  für $j = i \dots n$ (alle Enden)
    $S = \sum_{k=i}^{j} a_k$ (berechne Summe)
    merke maximales S ← wieso konnte ich das in der Laufzeitanalyse weglassen?

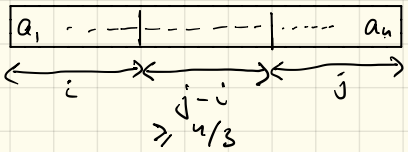
elementare Operation: Addition

Kosten: $\sum_{i=1}^{n} \sum_{j=i}^{n} (j-i) \leq \sum_{i=1}^{n} \sum_{j=1}^{n} n = n^3$

$\geq \sum_{i=1}^{n/3} \sum_{j=\frac{2}{3}n+1}^{n} (j-i) \geq \sum_{i=1}^{n/3} \sum_{j=2n/3+1}^{n} n/3 = \frac{n^3}{27}$

$\Rightarrow$ Laufzeit $\leq O(n^3)$ (Schranke ist scharf)

Abschätzung nach unten visuell:



In Java sieht das etwa so aus:

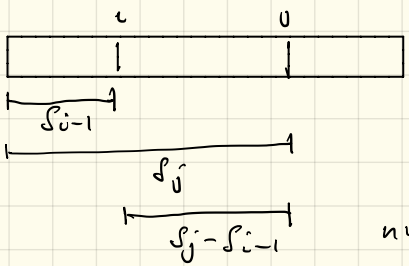
```
M = 0;
for (i=1 ; i<=n ; i = i+1)          // Anfang
   for (j=i; j<=n ; j = j+1) {      // Ende
      S = 0;
      for (k=i; k<=j ; k = k+1)     // berechne Summe
         S = S + a[k];
         if (S > M)                 // update Maximum
            M = S;
   }
```

Algorithmus 2 (Präfixsummen)

Neue Algorithmen design method: geschickte Vorberechnung von Termen die sich danach durch häufige Benutzung amortisiert.

Terme hier: "Präfixsummen" $S_k = \sum_{i=1}^{k} a_i$, $k = 1..n$

Nutzung:



nur eine Operation!

$S_0 = 0$

für $i = 1 \ldots n$
  $S_i = S_{i-1} + a_i$
} Vorberechnung: $O(n)$

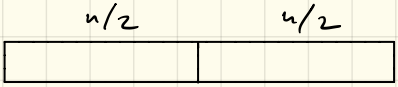
für $i = 1 \ldots n$
  für $j = i \ldots n$
    $S = S_j - S_{i-1}$
    merke maximales S
} alle Summen: $O(n^2)$

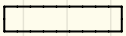
Laufzeit: $O(n^2)$

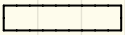
Algorithmus 3 (divide-and-conquer)

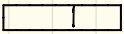
divide-and-conquer: eine Form von Design durch Induktion (löse kleine Probleme → löse grosses Problem)

Idee: zerlege in 2 Hälften



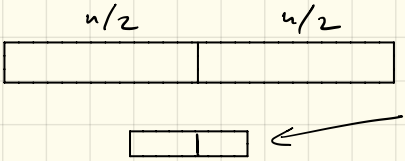
Fall 1:  Lösung ganz links

Fall 2:  " ganz rechts

Fall 3:  " über die Mitte

Fall 1 & 2: gleiches Problem für $n/2$
→ löse rekursiv
(das bedeutet, dass auch die Hälften wieder zerlegt werden usw.)

Fall 3: entscheidende Beobachtung



Algorithmus: $n=1$: $a_1 > 0 \Rightarrow \Pi = a_1$
$a_1 \leq 0 \Rightarrow \Pi = 0$

$n > 1$:
- teile in der Mitte — $O(1)$
- löse Teilprobleme links ($\Rightarrow \Pi_1$) und rechts ($\Rightarrow \Pi_2$) — $2T(\frac{n}{2})$
- berechne beste Randstücke, setze zusammen ($\Rightarrow \Pi_3$) — $O(n)$
- $\Pi = \max(\Pi_1, \Pi_2, \Pi_3)$ — $O(1)$

Laufzeitanalyse: $T(1) = c$ (Konstante)
$T(n) = 2T(\frac{n}{2}) + a \cdot n$, a Konstante

Lösung durch „Beleskopieren": $n = 2^k$: äquivalent asymptotisch nichts

$$T(n) = 2T(\tfrac{n}{2}) + an \qquad \text{(Annahme: } n = 2^k\text{)}$$

$$= 2\left(2T(\tfrac{n}{4}) + a\tfrac{n}{2}\right) + an = 4T(\tfrac{n}{4}) + 2an$$

$$= 4\left(2T(\tfrac{n}{8}) + a\tfrac{n}{4}\right) + 2an = 8T(\tfrac{n}{8}) + 3an$$

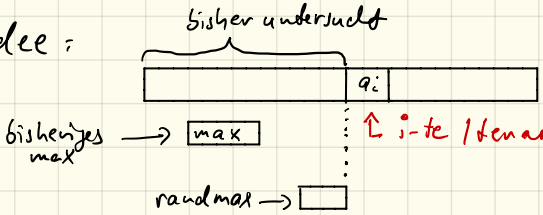
$$\vdots$$

$$= nT(1) + \log_2(n)\, an = O(n \log n)$$

ganz formal: beweise durch Induktion!

Algorithmus 4 (induktiv von links nach rechts)

Idee:



Beobachtung:
ein neues max kann nur durch randmax + a_i entstehen

```
randmax = 0
max = 0
für i = 1..n
    randmax = randmax + a_i
    wenn randmax > max
        max = randmax
    wenn randmax ≤ 0
        randmax = 0
```

} O(1)

Laufzeit O(n)

Beispiel: m = max, r = randmax

Iterationen

0	1	2	3	4	5	6	7	8
	7	-11	15	110	-23	-3	127	-1
r=0	r=7	r=0	r=15	r=125	r=102	r=99	r=226	r=225
m=0	m=7	m=7	m=15	m=125	m=125	m=125	m=226	m=226

Komplexität des Problems

Behauptung: ein korrekter Algorithmus muss alle Elemente a_i ansehen

[etwas formaler: Algo korrekt $\Rightarrow$ schaut alle a_i an]

Beweis: indirekt, d.h. wir zeigen

[Algo schaut nicht alle a_i an $\Rightarrow$ Algo ist nicht korrekt]

Also, nehmen wir an er schaut nicht alle a_i an:

Fall 1: Algo berechnet Lösung S die a_i enthält
$\rightarrow$ setze $a_i = -\infty$ (oder $-n \max_{j \neq i} |a_j| - 1$)
Damit ist $S < 0$ und daher falsch

Fall 2: Algo berechnet Lösung ohne a_i
$\rightarrow$ setze $a_i = +\infty$ (oder $n \max_{j \neq i} |a_j| + 1$)
Damit ist $S < a_i$ und daher falsch $\square$

Asymptotische Laufzeitanalyse (Teil 2)

Laufzeitanalyse eines Algorithmus

1.) Bestimme elementare Operationen
(z.B. *, +, Vergleich, push, pop, ...) Annahme: diese kosten $O(1)$

2.) (asymptotische) Laufzeit ist deren Anzahl in O-Notation im worst-case
z.B. $T(n) = O(n^2)$ über alle Eingaben
↑ Grösse der Eingabe

O-Notation

Wir betrachten positive Funktionen
$f, g : \mathbb{N} \longrightarrow \mathbb{R}^+ \quad (0 \notin \mathbb{R}^+)$

Asymptotisch obere Schranke

$$O(g(n)) = \{ f(n) \mid \exists c > 0 : f(n) \leq c\, g(n),\ n \geq 1 \}$$

$f(n) \leq O(g(n))$ Schreibweise für $f(n) \in O(g(n))$

Bild (für die Intuition):

"alle Funktionen die höchstens quadratisch wachsen"

$f(n) \in O(g(n))$ ist eine Relation auf den Funktionen

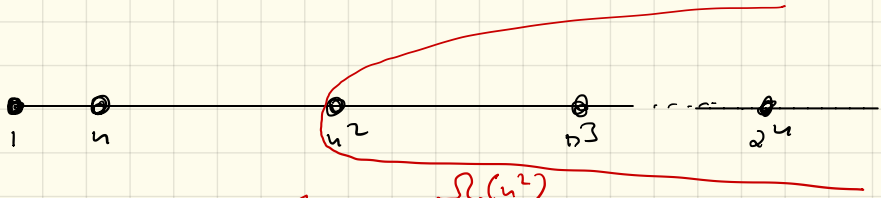
Asymptotisch untere Schranke

$$\Omega(g(n)) = \{ f(n) \mid \exists c > 0 : f(n) \geq c\, g(n),\ n \geq 1 \}$$

Omega

$f(n) \geq \Omega(g(n))$ für $f(n) \in \Omega(g(n))$

Bild:



"alle Funktionen die mindestens quadratisch wachsen"

$n^3 \geq \Omega(n^2)$

$2n \log n + 5 \geq \Omega(n \log n)$

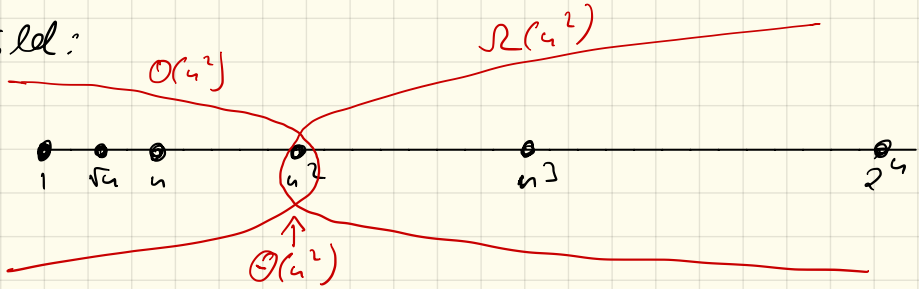
etc.

Asymptotisch genaues Wachstum

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

$f(n) = \Theta(g(n))$ für $f(n) \in \Theta(g(n))$

Bild:

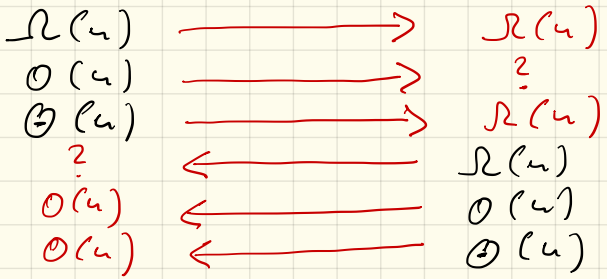


Anwendung auf die vier Max-Subarray Algorithmen:

Algo 1: $T(n) = \Theta(n^3)$
Algo 2: $T(n) = \Theta(n^2)$
Algo 3: $T(n) = \Theta(n \log n)$
Algo 4: $T(n) = \Theta(n)$

Da das Problem $\Omega(n)$ ist $\Rightarrow$ Max-Subarray hat Komplexität $\Theta(n)$

Komplexität eines Problems | Laufzeit irgend eines Algorithmus der Problem löst



Noch einmal: alle Analysen sind für den worst case! D.h. wenn ein Algorithmus nur in einem Fall z.B. $\Theta(n^2)$ braucht und sonst $\Theta(n)$ ist seine Laufzeit $\Theta(n^2)$.