

3. Beispiele

Korrektheit eines Algorithmus oder seiner Implementation zeigen,
Rekursion und Rekurrenzen
[Literaturangaben bei den Beispielen]

3.1 Altägyptische Multiplikation

Altägyptische Multiplikation – Ein Beispiel, wie man Korrektheit von Algorithmen zeigen kann.

Altägyptische Multiplikation

3

Berechnung von $11 \cdot 9$

| | | |
|---------------|--|--------------|
| 11 | | 9 |
| 22 | | 4 |
| 44 | | 2 |
| 88 | | 1 |
| <hr/> | | |
| 99 | | — |

| | | |
|---------------|--|--------------|
| 9 | | 11 |
| 18 | | 5 |
| 36 | | 2 |
| 72 | | 1 |
| <hr/> | | |
| 99 | | |

1. Links verdoppeln, rechts ganzzahlig halbieren.
2. Gerade Zahl rechts \Rightarrow Zeile streichen.
3. Übrige Zeilen links addieren.

³Auch bekannt als Russische Bauernmultiplikation

Vorteile

- Kurze Beschreibung, einfach zu verstehen.
- Effizient für Computer im Dualsystem: Verdoppeln = Left Shift, Halbieren = Right Shift

left shift $9 = 01001_2 \rightarrow 10010_2 = 18$

right shift $9 = 01001_2 \rightarrow 00100_2 = 4$

Fragen

- Für welche Eingaben liefert der Algorithmus das richtige Resultat (in endlicher Zeit)?
- Wie beweist man seine Korrektheit?
- Was ist ein gutes Mass für seine Effizienz?

Die Essenz

Wenn $b > 1$, $a \in \mathbb{Z}$, dann:

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Terminierung

$$a \cdot b = \begin{cases} a & \text{falls } b = 1, \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Rekursiv funktional notiert

$$f(a, b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a, \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

Als Funktion programmiert

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}
```

Korrektheit: Mathematischer Beweis

$$f(a, b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a \cdot \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

Zu zeigen: $f(a, b) = a \cdot b$ für $a \in \mathbb{Z}$, $b \in \mathbb{N}^+$.

Korrektheit: Mathematischer Beweis per Induktion

Sei $a \in \mathbb{Z}$, zu zeigen $f(a, b) = a \cdot b \quad \forall b \in \mathbb{N}^+$.

Anfang: $f(a, 1) = a = a \cdot 1$

Hypothese: $f(a, b') = a \cdot b' \quad \forall 0 < b' \leq b$

Schritt: $f(a, b') = a \cdot b' \quad \forall 0 < b' \leq b \stackrel{!}{\Rightarrow} f(a, b+1) = a \cdot (b+1)$

$$f(a, b+1) = \begin{cases} f\left(2a, \overbrace{\frac{b+1}{2}}^{0 < \cdot \leq b}\right) \stackrel{i.H.}{=} a \cdot (b+1) & \text{falls } b > 0 \text{ ungerade,} \\ a + f\left(2a, \underbrace{\frac{b}{2}}_{0 < \cdot < b}\right) \stackrel{i.H.}{=} a + a \cdot b & \text{falls } b > 0 \text{ gerade.} \end{cases}$$




[Code-Umformung: Endrekursion]

Die Rekursion lässt sich *endrekursiv* schreiben

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}

// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    int z=0;
    if (b%2 != 0){
        --b;
        z=a;
    }
    return z + f(2*a, b/2);
}
```



[Code-Umformung: Endrekursion \Rightarrow Iteration]

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    int z=0;
    if (b%2 != 0){
        --b;
        z=a;
    }
    return z + f(2*a, b/2);
}
```



```
int f(int a, int b) {
    int res = 0;
    while (b != 1) {
        int z = 0;
        if (b % 2 != 0){
            --b;
            z = a;
        }
        res += z;
        a *= 2; // neues a
        b /= 2; // neues b
    }
    res += a; // Basisfall b=1
    return res;
}
```

[Code-Umformung: Vereinfachen]

```
int f(int a, int b) {  
    int res = 0;  
    while (b != 1) {  
        int z = 0;  
        if (b % 2 != 0){  
            --b; → Teil der Division  
            z = a; → Direkt in res  
        }  
        res += z;  
        a *= 2;  
        b /= 2;  
    }  
    res += a; → in den Loop  
    return res;  
}
```



```
// pre: b>0  
// post: return a*b  
int f(int a, int b) {  
    int res = 0;  
    while (b > 0) {  
        if (b % 2 != 0)  
            res += a;  
        a *= 2;  
        b /= 2;  
    }  
    return res;  
}
```

Korrektheit: Argumentation mit Invarianten!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

Hier gilt $x = a \cdot b + res$

Wenn hier $x = a \cdot b + res \dots$

... dann auch hier $x = a \cdot b + res$
 b gerade

Hier gilt $x = a \cdot b + res$

Hier gilt $x = a \cdot b + res$ und $b = 0$

Also $res = x$.

Zusammenfassung

Der Ausdruck $a \cdot b + res$ ist eine Invariante.

- Werte von a, b, res ändern sich, aber die Invariante bleibt "im Wesentlichen" unverändert: Invariante vorübergehend durch eine Anweisung zerstört, aber dann darauf wieder hergestellt. Betrachtet man solche Aktionsfolgen als atomar, bleibt der Wert tatsächlich invariant
- Insbesondere erhält die Schleife die Invariante (*Schleifeninvariante*), sie wirkt dort wie der Induktionsschritt bei der vollständigen Induktion
- Invarianten sind offenbar mächtige Beweishilfsmittel!

[Weiteres Kürzen]

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    while (b > 0) {
        if (b % 2 != 0){
            res += a;
            --b;
        }
        a *= 2;
        b /= 2;
    }
    return res;
}
```



```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    while (b > 0) {
        res += a * (b%2);
        a *= 2;
        b /= 2;
    }
    return res;
}
```

[Analyse]

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    while (b > 0) {
        res += a * (b%2);
        a *= 2;
        b /= 2;
    }
    return res;
}
```

Altägyptische Multiplikation entspricht der Schulmethode zur Basis 2.

$$\begin{array}{r} 1\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 1 \quad (9) \\ 1\ 0\ 0\ 1 \quad (18) \\ \hline 1\ 1\ 0\ 1\ 1 \\ 1\ 0\ 0\ 1 \quad (72) \\ \hline 1\ 1\ 0\ 0\ 0\ 1\ 1 \quad (99) \end{array}$$

Effizienz

Frage: Wie lange dauert eine Multiplikation von a und b ?

- Mass für die Effizienz

- Gesamtzahl der elementaren Operationen: Verdoppeln, Halbieren, Test auf "gerade", Addition
- Im rekursiven wie im iterativen Code: maximal 6 Operationen pro Aufruf bzw. Durchlauf

- Wesentliches Kriterium:

- Anzahl rekursiver Aufrufe oder
- Anzahl Schleifendurchläufe(im iterativen Fall)

- $\frac{b}{2^n} \leq 1$ gilt für $n \geq \log_2 b$. Also nicht mehr als $6 \lceil \log_2 b \rceil$ elementare Operationen.

3.2 Schnelle Multiplikation von Zahlen

[Ottman/Widmayer, Kap. 1.2.3]

Beispiel 2: Multiplikation grosser Zahlen

Primarschule:

| | | | | | |
|----------|----------|---|----------|----------|--------------|
| <i>a</i> | <i>b</i> | | <i>c</i> | <i>d</i> | |
| 6 | 2 | · | 3 | 7 | |
| <hr/> | | | | | |
| | | | 1 | 4 | <i>d · b</i> |
| | | 4 | 2 | | <i>d · a</i> |
| | | | 6 | | <i>c · b</i> |
| | 1 | 8 | | | <i>c · a</i> |
| <hr/> | | | | | |
| = | 2 | 2 | 9 | 4 | |

$2 \cdot 2 = 4$ einstellige Multiplikationen. \Rightarrow Multiplikation zweier n -stelliger Zahlen: n^2 einstellige Multiplikationen

Beobachtung

$$\begin{aligned}ab \cdot cd &= (10 \cdot a + b) \cdot (10 \cdot c + d) \\&= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\&\quad + 10 \cdot b \cdot d + b \cdot d \\&\quad + 10 \cdot (a - b) \cdot (d - c)\end{aligned}$$

Verbesserung?

| <i>a</i> | <i>b</i> | | <i>c</i> | <i>d</i> | |
|----------|----------|---|----------|----------|---------------------|
| 6 | 2 | · | 3 | 7 | |
| <hr/> | | | | | |
| | | | 1 | 4 | <i>d · b</i> |
| | | | 1 | 4 | <i>d · b</i> |
| | | | 1 | 6 | $(a - b) · (d - c)$ |
| | | | 1 | 8 | <i>c · a</i> |
| | 1 | 8 | | | <i>c · a</i> |
| <hr/> | | | | | |
| = | 2 | 2 | 9 | 4 | |

→ 3 einstellige Multiplikationen.

Grosse Zahlen

$$6237 \cdot 5898 = \underbrace{62}_{a'} \underbrace{37}_{b'} \cdot \underbrace{58}_{c'} \underbrace{98}_{d'}$$

Rekursive / induktive Anwendung: $a' \cdot c'$, $a' \cdot d'$, $b' \cdot c'$ und $c' \cdot d'$ wie oben berechnen.

→ $3 \cdot 3 = 9$ statt 16 einstellige Multiplikationen.

Verallgemeinerung

Annahme: zwei n -stellige Zahlen, $n = 2^k$ für ein k .

$$\begin{aligned}(10^{n/2}a + b) \cdot (10^{n/2}c + d) &= 10^n \cdot a \cdot c + 10^{n/2} \cdot a \cdot c \\ &+ 10^{n/2} \cdot b \cdot d + b \cdot d \\ &+ 10^{n/2} \cdot (a - b) \cdot (d - c)\end{aligned}$$

Rekursive Anwendung dieser Formel: Algorithmus von Karatsuba und Ofman (1962).

Algorithmus Karatsuba Ofman

Input: Zwei n -stellige ($n > 0$) ganze positive Zahlen x und y mit dezimalen Ziffern $(x_i)_{1 \leq i \leq n}$ und $(y_i)_{1 \leq i \leq n}$

Output: Produkt $x \cdot y$

if $n = 1$ **then**

| **return** $x_1 \cdot y_1$

else

Sei $m := \lfloor \frac{n}{2} \rfloor$

Unterteile $a := (x_1, \dots, x_m)$, $b := (x_{m+1}, \dots, x_n)$, $c := (y_1, \dots, y_m)$,

$d := (y_{m+1}, \dots, y_n)$

Berechne rekursiv $A := a \cdot c$, $B := b \cdot d$, $C := (a - b) \cdot (d - c)$

Berechne $R := 10^n \cdot A + 10^m \cdot A + 10^m \cdot B + B + 10^m \cdot C$

| **return** R

Analyse

$M(n)$: Anzahl einstelliger Multiplikationen.

Rekursive Anwendung des obigen Algorithmus \Rightarrow Rekursionsgleichung:

$$M(2^k) = \begin{cases} 1 & \text{falls } k = 0, \\ 3 \cdot M(2^{k-1}) & \text{falls } k > 0. \end{cases} \quad (\text{R})$$

Teleskopieren

Iteratives Einsetzen der Rekursionsformel zum Lösen der Rekursionsgleichung.

$$\begin{aligned}M(2^k) &= 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) \\ &= \dots \\ &\stackrel{!}{=} 3^k \cdot M(2^0) = 3^k.\end{aligned}$$

Beweis: Vollständige Induktion

Hypothese $H(k)$:

$$M(2^k) = F(k) := 3^k. \quad (\text{H})$$

Behauptung:

$H(k)$ gilt für alle $k \in \mathbb{N}_0$.

Induktionsanfang $k = 0$:

$$M(2^0) \stackrel{R}{=} 1 = F(0). \quad \checkmark$$

Induktionsschritt $H(k) \Rightarrow H(k + 1)$:

$$M(2^{k+1}) \stackrel{R}{=} 3 \cdot M(2^k) \stackrel{H(k)}{=} 3 \cdot F(k) = 3^{k+1} = F(k + 1). \quad \checkmark$$



Vergleich

Primarschulmethode: n^2 einstellige Multiplikationen.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Beispiel: 1000-stellige Zahl: $1000^2/1000^{1.58} \approx 18$.

Bestmöglicher Algorithmus?

Wir kennen nun eine obere Schranke $n^{\log_2 3}$.

Es gibt praktisch (für grosses n) relevante, schnellere Algorithmen. Beispiel: Schönhage-Strassen-Algorithmus (1971) basierend auf schneller Fouriertransformation mit Laufzeit $\mathcal{O}(n \log n \cdot \log \log n)$. Die beste obere Schranke ist nicht bekannt. ⁴

Untere Schranke: n . Jede Ziffer muss zumindest einmal angeschaut werden.

⁴Im März 2019 haben David Harvey and Joris van der Hoeven einen, praktisch noch irrelevanten, $\mathcal{O}(n \log n)$ Algorithmus vorgestellt. Man vermutet, dass $n \log n$ die beste obere Grenze ist, hat es aber noch nicht bewiesen.

Anhang: Asymptotik mit Additionen und Shifts

Bei jeder Multiplikation zweier n -stelliger Zahlen kommt auch noch eine konstante Anzahl Additionen, Subtraktionen und Shifts dazu

Additionen, Subtraktionen und Shifts von n stelligen Zahlen kosten $\mathcal{O}(n)$

Daher ist die asymptotische Laufzeit eigentlich (mit geeignetem $c > 1$, $d > 0$) bestimmt durch die folgende Rekurrenz

$$T(n) = \begin{cases} 3 \cdot T\left(\frac{1}{2}n\right) + c \cdot n & \text{falls } n > 1 \\ d & \text{sonst} \end{cases}$$

Anhang: Asymptotik mit Additionen und Shifts

Annahme: $n = 2^k$, $k > 0$

$$\begin{aligned}T(2^k) &= 3 \cdot T(2^{k-1}) + c \cdot 2^k \\&= 3 \cdot (3 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\&= 3 \cdot (3 \cdot (3 \cdot T(2^{k-3}) + c \cdot 2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\&= 3 \cdot (3 \cdot (\dots(3 \cdot T(2^{k-k}) + c \cdot 2^1)\dots) + c \cdot 2^{k-1}) + c \cdot 2^k \\&= 3^k \cdot d + c \cdot 2^k \sum_{i=0}^{k-1} \frac{3^i}{2^i} = 3^k \cdot d + c \cdot 2^k \frac{\frac{3^k}{2^k} - 1}{\frac{3}{2} - 1} \\&= 3^k(d + 2c) - 2c \cdot 2^k\end{aligned}$$

Somit $T(2^k) = 3^k(d + 2c) - 2c \cdot 2^k \in \Theta(3^k) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$.

3.3 Maximum Subarray Problem

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]

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.

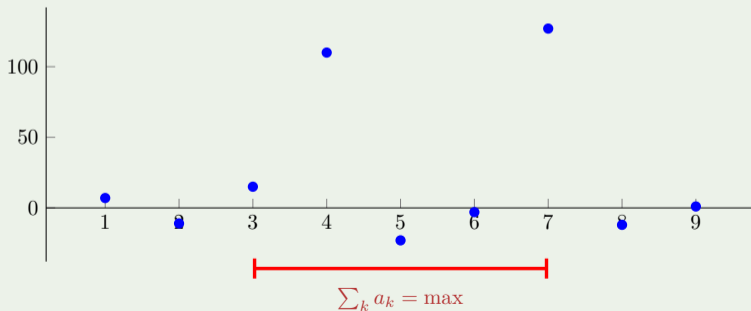
Maximum Subarray Problem

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

Gesucht: Teilstück $[i, j]$, $1 \leq i \leq j \leq n$ mit maximaler positiver Summe

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

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



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 3

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

Beweis:

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

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**

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

Analyse

Theorem 4

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

Beweis:

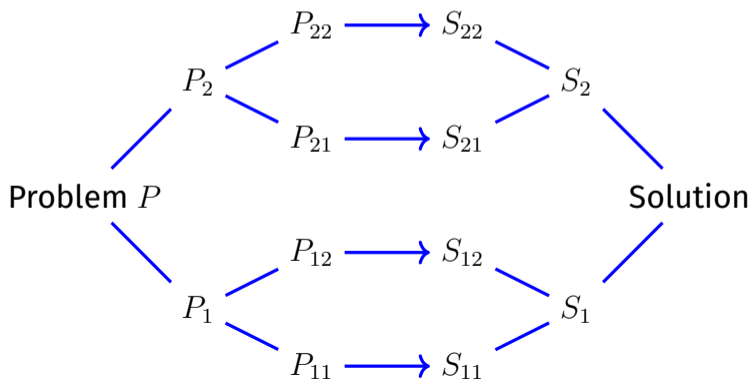
$$\sum_{i=1}^n 1 + \sum_{i=1}^n \sum_{j=i}^n 1 = n + \sum_{i=1}^n (n - i + 1) = n + \sum_{i=1}^n i = \Theta(n^2)$$



divide et impera

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

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



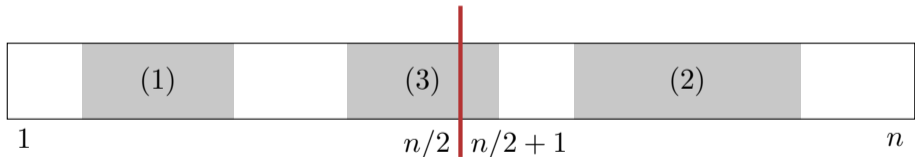
Maximum Subarray – Divide

- 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



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

Analyse

Theorem 5

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

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 $n = 2^k$:

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

Lösung:

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

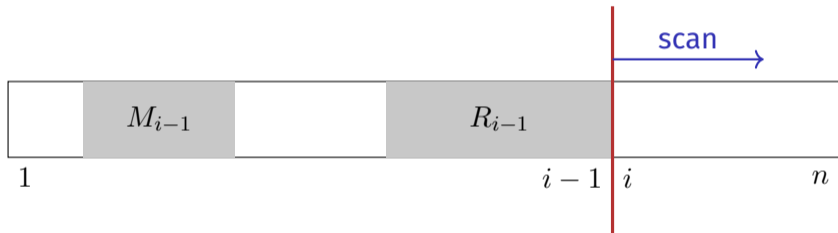
also

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



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).

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

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 ;

Analyse

Theorem 6

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

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.

Komplexität des Maximum Subarray Sum Problems

Theorem 7

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

3.4 Anhang

Herleitung und Wiederholung einiger mathematischen Formeln

Logarithmen

$$\log_a y = x \Leftrightarrow a^x = y \quad (a > 0, y > 0)$$

$$\log_a(x \cdot y) = \log_a x + \log_a y$$

$$a^x \cdot a^y = a^{x+y}$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$\frac{a^x}{a^y} = a^{x-y}$$

$$\log_a x^y = y \log_a x$$

$$a^{x \cdot y} = (a^x)^y$$

$$\log_a n! = \sum_{i=1}^n \log i$$

$$\log_b x = \log_b a \cdot \log_a x$$

$$a^{\log_b x} = x^{\log_b a}$$

Letzteres sieht man durch Einsetzen von $x \rightarrow a^{\log_a x}$

Summen

$$\sum_{i=0}^n i = \frac{n \cdot (n + 1)}{2} \in \Theta(n^2)$$

Trick

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

Summen

$$\sum_{i=0}^n i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Trick:

$$\sum_{i=1}^n i^3 - (i-1)^3 = \sum_{i=0}^n i^3 - \sum_{i=0}^{n-1} i^3 = n^3$$

$$\sum_{i=1}^n i^3 - (i-1)^3 = \sum_{i=1}^n i^3 - i^3 + 3i^2 - 3i + 1 = n - \frac{3}{2}n \cdot (n+1) + 3 \sum_{i=0}^n i^2$$

$$\Rightarrow \sum_{i=0}^n i^2 = \frac{1}{6}(2n^3 + 3n^2 + n) \in \Theta(n^3)$$

Kann einfach verallgemeinert werden: $\sum_{i=1}^n i^k \in \Theta(n^{k+1})$.

Geometrische Reihe

$$\sum_{i=0}^n \rho^i \stackrel{!}{=} \frac{1 - \rho^{n+1}}{1 - \rho}$$

$$\begin{aligned} \sum_{i=0}^n \rho^i \cdot (1 - \rho) &= \sum_{i=0}^n \rho^i - \sum_{i=0}^n \rho^{i+1} = \sum_{i=0}^n \rho^i - \sum_{i=1}^{n+1} \rho^i \\ &= \rho^0 - \rho^{n+1} = 1 - \rho^{n+1}. \end{aligned}$$

Für $0 \leq \rho < 1$:

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{1 - \rho}$$