Algorithmen und Datenstrukturen

Teile und Herrsche:

Maximale Teilsumme ♦ Matrixmultiplikation

Algorithmen auf Sequenzen

(und Matrizen)

Maximale Teilsumme

Name: **Maximale Teilsumme** [5, -3, -3, 1, 4, -1, 2, -2]
Eingabe: Sequenz L ganzer Zahlen
Ausgabe: größte Summe von Elementen einer Teilsequenz

Teilsequenz:

- zusammenhängender Teil der Sequenz L
- bestimmt durch Paar von Indizes (i, j): $[a_i , a_{i+1} , \ldots, a_j]$, also Sequenz mit allen a_k mit $i \leq k \leq j$
- *in Python:* `a[i:j+1]`

Anwendung: maximaler Gewinn mit einer Aktie
in einem bestimmten Zeitraum

Maximale Teilsumme naiv: *Idee*

- alle Teilsummen systematisch durchprobieren **(Brute Force)**
- untere Grenze *i* von 0 bis len(*L*)-1
  → `i in range(0, len(L))`
- obere Grenze jeweils von *i* bis len(*L*)-1
  → `j in range(i, len(L))`

➢ Teilfolgen [0], [0,1], [0,1,2], ..., [0,1,..., len(*L*)-1], [1], [1,2], ..., [1,2,..., len(*L*)-1], [2], [2,3], ..., [len(*L*)-1]

➢ stets Summe bilden und Maximum merken

Maximale Teilsumme naiv

```
def maxTeilsumme_1(L):
    maxSumme = 0  # mindestens 0 (leere Teilsequenz)
    for i in range(0, len(L)): # untere Grenze
        for j in range(i, len(L)): # obere Grenze
            summe = 0
            for k in range(i, j+1): # Summe bilden
                summe += L[k]
            if summe > maxSumme:
                maxSumme = summe
    return maxSumme
```

Laufzeitanalyse

mit $n := \text{len}(L)$

```
def maxTeilsumme_1(L):
    maxSumme = 0                            O(1)
    for i in range(0, len(L)):              O(n) Loops
        for j in range(i, len(L)):          O(n) Loops
            summe = 0                       O(1)
            for k in range(i, j+1):         O(n) Loops
                summe += L[k]               O(1)
            if summe > maxSumme:            O(1)
                maxSumme = summe            O(1)
    return maxSumme                         O(1)
```

$\rightarrow O(n^3)$

Laufzeitanalyse … etwas genauer …

```
for i in range(0, len(L))
        for j in range(i, len(L))
               for k in range(i, j+1)
```

mit $n := \text{len}(L)$

i \ j	0	1	2	…	n-1
0	1	2	3	…	n
1		1	2	…	n-1
2			1	…	n-2
…				…	…
n-1					1

u

$$\sum_{u=1}^{n} \sum_{k=1}^{u} k$$

$$= \sum_{u=1}^{n} \frac{u(u+1)}{2}$$

$\in O(n^3)$

Entwurfsparadigma
Teile und Herrsche
(Divide and Conquer)

Teile und Herrsche - Idee

1. Zerlegung einer Eingabe der Größe *n* in mehrere (möglichst gleich große) Eingaben
2. Lösung des Problems für die kleineren Eingaben
3. Zusammensetzung der Lösungen für die kleineren Eingaben zur Lösung für die ursprüngliche Eingabe

- Lösung des Problems für die kleineren Eingaben auf die gleiche Weise, bis elementare Probleme entstehen

➢ **Rekursion**

Maximale Teilsumme Teile und Herrsche

- Zerlegung der Sequenz der Länge n (möglichst) in der Mitte (also etwa in zwei Sequenzen der Länge $n/2$)
- weitere Zerlegungen bis Sequenzen der Länge 1: $[a]$
  → maximale Teilsumme: $\max\{0, a\}$
- Zusammensetzung der Lösungen?!

[1,1,1,-1]

- Benötigen
  - maximale Teilsumme der linken und der rechten Teilfolge
  - maximale Summe rechter Randfolgen der linken Teilfolge und
  - maximale Summe linker Randfolgen der rechten Teilfolge

→ Maximum der beiden maximalen Teilsummen
und der Summe der maximalen Randfolgensummen

Maximale Teilsumme - Beispiel

max. Teilsumme links: **5**

max. Teilsumme rechts: **5**

[5, -3, -3, 1, 4, -1, 2, -2]

rechte Randfolgen:

1
-3, 1
-3, -3, 1
5, -3, -3, 1

linke Randfolgen:

4
4, -1
4, -1, 2
4, -1, 2, -2

1 ← → **5**

6

Randfolgen

Sei $L = [a_0, a_1, \ldots, a_s]$ eine beliebige Sequenz.

- **rechte Randfolge von *L*:** $[a_r, a_{r+1}, \ldots, a_s]$ für ein $0 \leq r \leq s$
- *Summe* der rechten Randfolge: $\sum_{k=r}^{s} a_k$
- **linke Randfolge von *L*:** $[a_0, a_1, \ldots, a_r]$ für ein $0 \leq r \leq s$
- *Summe* der linken Randfolge: $\sum_{k=0}^{r} a_k$

Berechnung linker Randsummen

```
def liRandMax(L,i,j):
    # pre: 0 <= i <= j < len(L)
    # post: maximale Summe einer linken Randfolge
    # der Teilfolge von i bis j (einschließlich i,j)
    max = 0
    sum = 0
    for k in range(i,j+1):  # k = i, i+1, ..., j
        sum += L[k]
        if (sum > max):
            max = sum
    return max
```

Berechnung rechter Randsummen

```
def reRandMax(L,i,j):
    # pre: 0 <= i <= j < len(L)
    # post: maximale Summe einer rechten Randfolge
    # der Teilfolge von i bis j (einschließlich i,j)
    max = 0
    sum = 0
    for k in range(j,i-1,-1):  # k = j, j-1, ..., i
        sum += L[k]
        if (sum > max):
            max = sum
    return max
```

Berechnung der maximalen Teilsumme

```
def maxTeilsumme(L,i,j):
    # pre: 0 <= i <= j < len(L)
    # post: maximale Teilsumme der Teilfolge
    # von i bis j (einschließlich i,j)
    if (i == j):   # nur ein Element
        if L[i] > 0: return L[i]
        else: return 0
    else:
        m = (i + j)//2   # Mitte von L
```

(Fortsetzung von else)

```
maxLinks = maxTeilsumme(L,i,m)
maxRechts = maxTeilsumme(L,m+1,j)
maxMisch = reRandMax(L,i,m)
           + liRandMax(L,m+1,j)

return max(maxLinks, maxRechts, maxMisch)
```

Maximale Teilsumme rekursiv

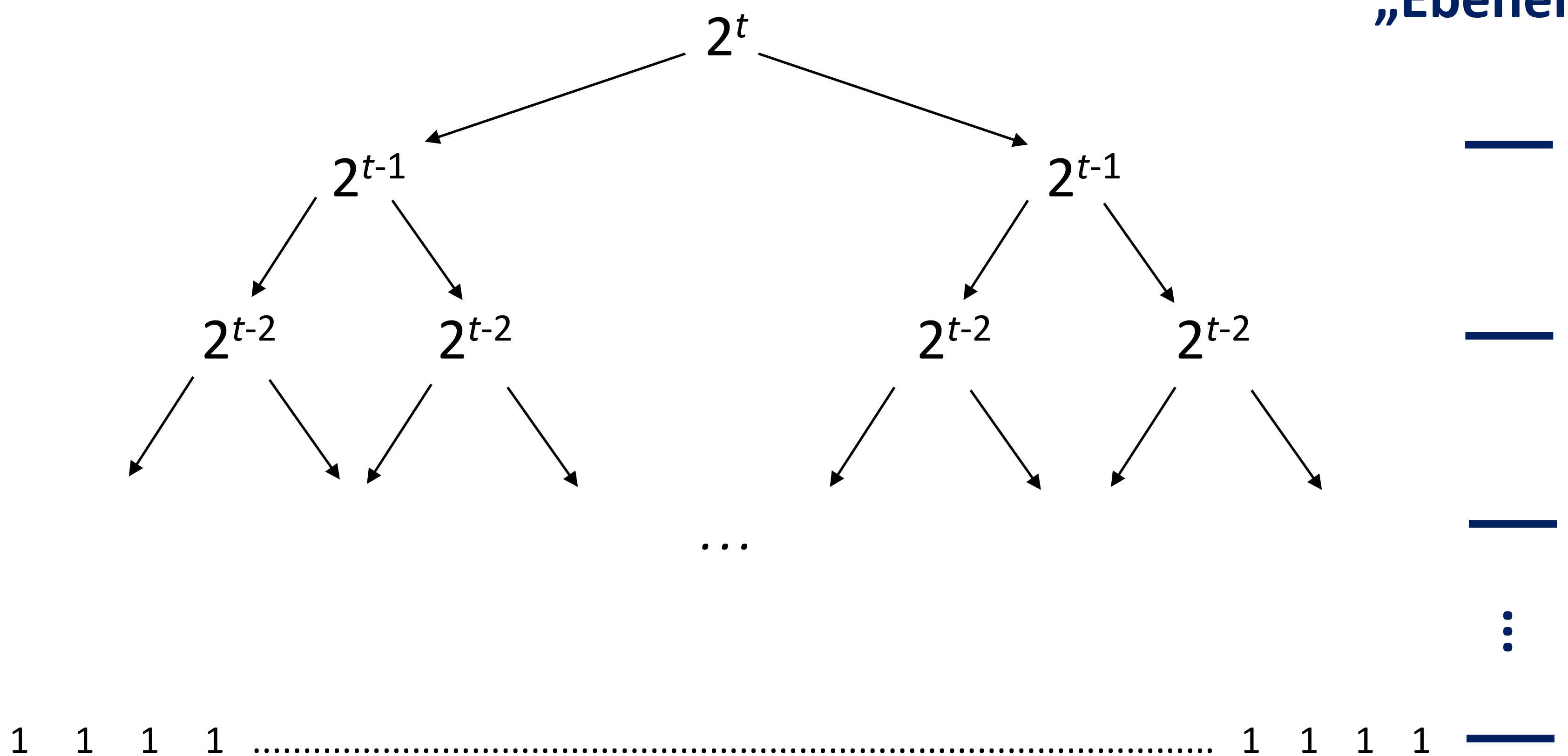
- Aufruf von **`maxTeilsumme(L,i,j)`** für die gesamte Folge:

```
def maxTeilsumme_rek(L):
    # pre: L ist nicht leer
    # post: maximale Teilsumme von L
    return maxTeilsumme(L,0,len(L)-1)
```

Analyse: Aufrufe von `maxTeilsumme_rek`

Annahme: L hat Länge $n = 2^t$

$O(\log n)$ „Ebenen“



Analyse: Arbeit je Ebene

- in Ebene mit Teilsequenzen der Länge $2^{t\text{-}k}$: **2^k mal:**

```
if (i == j):
        if L[i] > 0: return L[i]
        else: return 0
else:
        m = (i + j)//2
```

***O*(1)**

```
        maxLinks = maxTeilsumme(L,i,m)
        maxRechts = maxTeilsumme(L,m+1,j)
```

***O*(1)** (+ Arbeit in nächster Ebene)

```
        maxMisch = reRandMax(L,i,m)
                   + liRandMax(L,m+1,j)
```

$O(2^{t\text{-}k})$

```
        return max(maxLinks, maxRechts, maxMisch)
```

***O*(1)**

Analyse: Gesamtaufwand

- in Ebene mit 2^k Teilsequenzen der Länge 2^{t-k} :

  $\boldsymbol{O(2^k \cdot 2^{t-k}) = O(2^t)}$ *also* $\boldsymbol{O(n)}$

- bei $\log n$ vielen Ebenen somit

$$\boxed{O(n \log n)}$$

Analyse: Rekursionsgleichung

- Zeit $t(n)$ für Aufruf mit Sequenz der Länge $n > 1$:

$$t(n) = 2\ t(n/2) + d \cdot n$$
$$t(1) = C$$

- Für $n = 2^k$ gilt

$$\begin{aligned} t(2^k) &= 2\ t(2^{k-1}) + d \cdot 2^k \\ &= 2[2t(2^{k-2}) + d \cdot 2^{k-1}] + d \cdot 2^k \\ &= 4t(2^{k-2}) + 2d \cdot 2^k \\ &= 4[2t(2^{k-3}) + d \cdot 2^{k-2}] + 2d \cdot 2^k \\ &= 8t(2^{k-3}) + 3d \cdot 2^k \end{aligned}$$

Analyse: Rekursionsgleichung

- Zeit $t(n)$ für Aufruf mit Sequenz der Länge $n > 1$:

$$t(n) = 2\, t(n/2) + d \cdot n$$
$$t(1) = C$$

- Für $n = 2^k$ gilt

$$\begin{aligned} t(2^k) &\overset{\text{I.V.}}{=} 2^j\, t(2^{k-j}) + jd \cdot 2^k \quad (0 \le j \le k) \\ &= 2^j\, [2t(2^{k-j-1}) + d \cdot 2^{j-k}] + jd \cdot 2^k \\ &= 2^{j+1}\, t(2^{k-j-1}) + (j+1)\, d \cdot 2^k \\ &= 2^k\, t(2^{k-k}) + kd \cdot 2^k \\ &= n\, t(1) + \log_2 n \cdot d \cdot n \\ &\in O(n \log n) \end{aligned}$$

Analyse: Rekursionsgleichung

- Zeit $t(n)$ für Aufruf mit Sequenz der Länge $n > 1$:

$$t(n) = 2\, t(n/2) + d \cdot n$$

- *Allgemein* für rekursive Algorithmen:

$$t(n) = a\, t(n/b) + c(n)$$

  - a – Anzahl der Aufrufe für die kleineren Eingaben
  - n/b – Größe der kleineren Eingaben
  - $c(n)$ – Aufwand zum Zerlegen der Eingabe und zur Kombination der Lösung aus den Teillösungen

Lösung der Rekursionsgleichung

Master-Theorem: Für $a > 0$, $b > 1$, $d > 0$ und $c(n) \in O(n^\alpha)$ hat die Rekursion

$$t(n) = \begin{cases} a\, t(n/b) + c(n) & \text{falls } n > 1 \\ d & \text{falls } n = 1 \end{cases}$$

die Lösungen

$$t(n) \in \begin{cases} O(n^\alpha) & \text{falls } a < b^\alpha \\ O(n^\alpha \log n) & \text{falls } a = b^\alpha \\ O(n^{\log_b a}) & \text{falls } a > b^\alpha \end{cases}$$

Hier:

$a = b = 2$,

$\alpha = 1$

$\rightarrow$ **$O(n \log n)$**

(*s. z.B. Cormen, Leiserson, Rivest, Stein: Algorithmen – Eine Einführung. Oldenburg-Verlag, 2004.*)

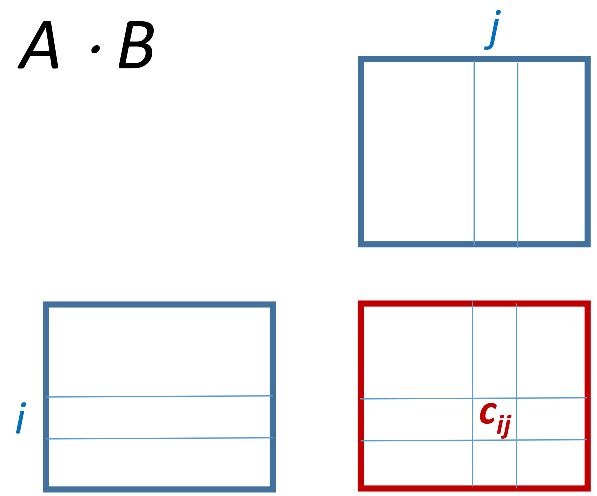
Schnelle Matrizenmultiplikation

Matrizenmultiplikation (klassisch)

Eingabe: zwei $n \times n$ Matrizen $A = (a_{ij})$ und $B = (b_{ij})$

Ausgabe: $C = (c_{ij})$ mit $C = A \cdot B$

$$c_{ij} = \sum_{1 \leq k \leq n} a_{ik}\, b_{kj}$$



→ Berechnung von n^2 Werten c_{ij} jeweils mit Aufwand $O(n)$

→ $\boldsymbol{O(n^3)}$

Teile und Herrsche (intuitiv)

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

mit

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Aufwand für jede Addition: $(n/2)^2$

→ Aufwand für das Zerlegen und die Konstruktion der Lösung: $\approx d \cdot n^2$

Analyse

$$t(n) = \begin{cases} 8\, t(n/2) + d_1 n^2 & \text{falls} \quad n > 2 \\ d_2 & \text{sonst} \end{cases}$$

$$\rightarrow \boldsymbol{t(n) \in O(n^{\log_2 8}) = O(n^3)}$$

Strassen-Algorithmus

- In $t(n) = 8\, t(n/2) + dn^2$ wird der Faktor 8 durch die Anzahl der Multiplikationen von Matrizen der Größe $\frac{n}{2} \times \frac{n}{2}$ verursacht.
- *Idee*: Reduzieren auf 7 solcher Multiplikationen
- Volker Strassen: $\boldsymbol{t(n) \in O(n^{\log_2 7}) = O(n^{2,81})}$
- weitere Verbesserungen existieren;
  bisher bester Algorithmus: $\boldsymbol{O(n^z)}$ **mit** $\boldsymbol{z < 2{,}373}$

Konstruktion nach Strassen

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22}) \cdot B_{11}$$
$$R = A_{11} \cdot (B_{12} - B_{22})$$
$$S = A_{22} \cdot (B_{21} - B_{11})$$
$$T = (A_{11} + A_{12}) \cdot B_{22}$$
$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$
$$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

(*Näheres z.B. in Meinel, Ch.: Effiziente Algorithmen. Fachbuchverlag Leipzig, 1991.*)