Algorithmen und Datenstrukturen
Kapitel 2: Divide & Conquer

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Divide-and-Conquer

- **Englisch:** *Divide and conquer*
- **Latein:** *Divide et impera*
- **Deutsch:** *Teile und herrsche*

Quelle: [2]

- **Bedeutung in der Historie**
  - Teile Volk bzw. Gruppierung in Untergruppen auf, um sie leichter zu beherrschen.

- **Bedeutung in der Informatik**
  - ***Divide***: Teile Problem in ***einfachere***, kleinere Teilprobleme auf.
  - ***Conquer***: Löse die Teilprobleme, ggfs. rekursiv, bis sie leicht lösbar sind.
  - ***Combine***: Berechne aus den Teillösungen die Gesamtlösung.

> ***Goethe:***
> *Entzwei und gebiete!*
> *Tüchtig Wort;*
> *[Verein und leite!*
> *Bessrer Hort]*

Inhalt

- **Türme von Hanoi**
- Maximum-Subarray-Problem
- Laufzeitanalyse von rekursiven Algorithmen
- Multiplikation großer Zahlen, Karatsuba [3]

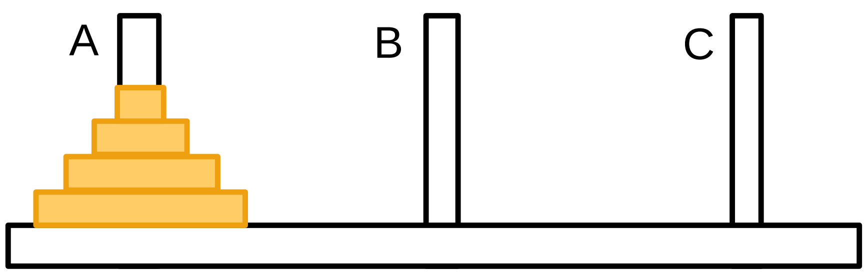
Türme von Hanoi

- 3 Stäbe (*A*, *B*, *C*), auf die mehrere verschieden große Scheiben mit einem Loch in der Mitte gelegt werden können.

- **Zu Beginn:** Alle Scheiben liegen auf *A*

- **Ziel des Spiels**
  - Verlege kompletten Scheibenstapel von *A* nach *B* oder *C*

- **Regeln**
  - Pro Zug ***nur*** 1 Scheibe von einem Stab auf einen anderen bewegen!
  - ***Keine*** größere Scheibe auf eine kleinere Scheibe legen!



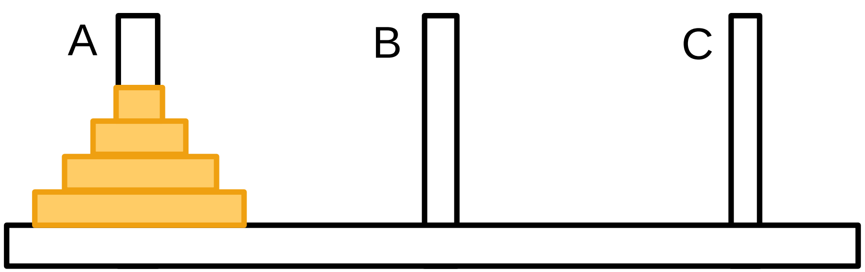
Wie löst man das Problem?

Türme von Hanoi: Divide-and-Conquer

- Um *n* Scheiben von *A* auf *C* zu versetzen
  - Bringe *n-1* Scheiben von *A* unter Verwendung von *C* auf *B*.
  - Führe dann Zug $A \rightarrow C$ aus
  - Bringe dann *n-1* Scheiben von *B* unter Verwendung von *A* auf den Stab *C.*

Anzahl Scheiben Start Ziel via

$$\underbrace{hanoi\,(n, A, C, B) =}_{\text{Gesamtproblem: } n \text{ Scheiben}} \underbrace{\begin{cases} A \rightarrow C & \textbf{falls n=1} \\ \text{hanoi}(n\text{-}1, A, B, C)\text{, dann } A \rightarrow C\text{, dann } hanoi(n\text{-}1, B, C, A) & \textbf{sonst} \end{cases}}_{\text{Kleinere Teilprobleme mit } n\text{-1 bzw. 1 Scheibe}}$$



Quellcode: HanoiRekursion.java

http://www.algomation.com/algorithm/towers-hanoi-recursive-visualization

Türme von Hanoi: Diskussion

- **Wie viele Züge** *Z(n)*, falls *n* Scheiben?
  - $Z(n) = 2^n - 1$
  - Errate Lösung, z.B. Ausprobieren in Java
  - Dann: Beweis durch Induktion.

- **Implementierungsvarianten** von Divide-and-Conquer
  - ***Rekursiv:*** Funktion ruft sich selbst auf (hier verwendet)
  - ***Iterativ:*** Schrittweises Berechnen der Lösung mit Sprachelementen wie Schleifen, ohne dass sich eine Funktion selbst aufruft.

- ***Iterativer Algorithmus*** („*Bunemann und Levy“, 1980*)
  - `Wiederhole bis gesamter Stapel auf der Zielscheibe`
    - `Setze` **`kleinste, freie Scheibe`** `auf die Stange rechts von ihr (falls es rechts keine Stange mehr gibt, dann auf die linke Stange → Zyklus)`
    - `Setze die` **`zweitkleinste, freie Scheibe`** `auf die einzig mögliche Stange`
  - Benötigt ebenfalls $2^n - 1$ Züge

Publikums-Joker

Welche Aussagen ist **falsch**?

A. Das Türme von Hanoi-Problem lässt sich immer lösen.

B. Das Türme von Hanoi-Problem lässt sich sowohl rekursiv als auch iterativ lösen.

C. Türme von Hanoi erfordert zwingend eine gerade Anzahl an Zügen.

D. Die Laufzeit des Türme von Hanoi-Problems ist $\Theta(2^n)$.

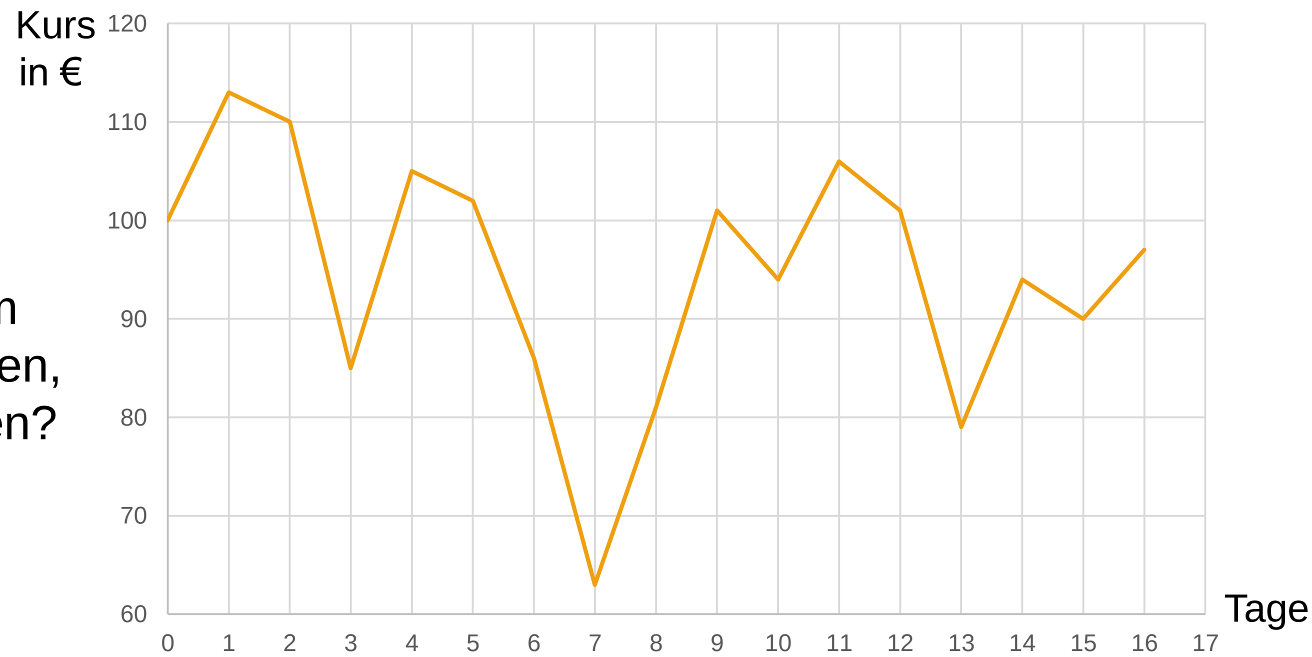
Inhalt

- Türme von Hanoi
- **Maximum-Subarray-Problem**
- Laufzeitanalyse von rekursiven Algorithmen
- Multiplikation großer Zahlen, Karatsuba [3]

Problemstellung

- **Börse**: Aktienkurs der letzten Tage bekannt.
- **Annahmen**
  - Man kann nur *einmal* am Tag (z.B. Mittags) kaufen.
  - Man kann nur *einmal* am Tag (z.B. Mittags) verkaufen.
  - Am Kauftag kann man nicht gleich wieder verkaufen.

Wann wäre der beste Tag zum Kaufen und Verkaufen gewesen, um den Gewinn zu maximieren?



Abstraktion: "Maximum Subarray"

- ***Umwandeln*** in ein "Array-Problem"
  - Kursdifferenz*:* `A[i] = <Preis am Tag i> - <Preis am Tag i-1>`
  - Gesucht: ***Teilarray***, bei dem die ***Summe*** der Elemente ***maximal*** ist.
  - Wie sieht Array *A* für die ersten 6 Tage der Kursentwicklung aus?

Tag	0	1	2	3	4	5	6	…
Preis	100	113	110	85	105	102	86	…
Änderung		13	?	?	?	?	?	?

- **Naheliegende Ansätze** funktionieren nicht. Warum?
  - Kauf bei Minimalkurs UND Verkauf bei Maximalkurs
  - ENTWEDER Kauf bei Minimalkurs ODER Verkauf bei Maximalkurs

Maximum Subarray: Version 1.0

- **Ziel:** Algorithmus,
  - der das Problem möglichst effizient löst.
  - der gut mit Größe des Eingabearrays skaliert.

- **"Brute Force":** Berechne Summe für **alle** Subarrays
  - Iteriere über alle Subarrays
  - Berechne jeweils die Summe

- ***Analyse der Laufzeit***
  - Wie viele Subarrays gibt es? $\binom{n}{2} = \Theta(n^2)$.
  - Was kostet Summenberechnung bei Array mit max. Länge *n*? $O(n)$
  - Gesamt: $O(n^3)$.

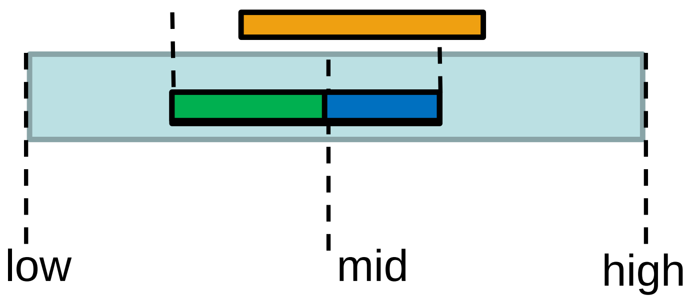
- Geht es schneller?

Version 2.0: Divide-and-Conquer

- ***Divide*: ** Teile Array *A* in 2 Teilarrays möglichst gleicher Länge auf
  - Links: `A[low…mid]`
  - Rechts: `A[(mid+1)…high]`

- ***Conquer***: Löse "Maximum Subarray" für diese beiden Teilarrays

- ***Combine:*** Die Gesamtlösung ist die "beste Lösung" aus
  - ***Fall A***: Linke Lösung, `A[low…mid]` *oder*
  - ***Fall B:*** Rechte Lösung, `A[(mid+1)…high]` *oder*
  - ***Fall C:*** Lösung, die über die Grenze (= *mid*) geht.

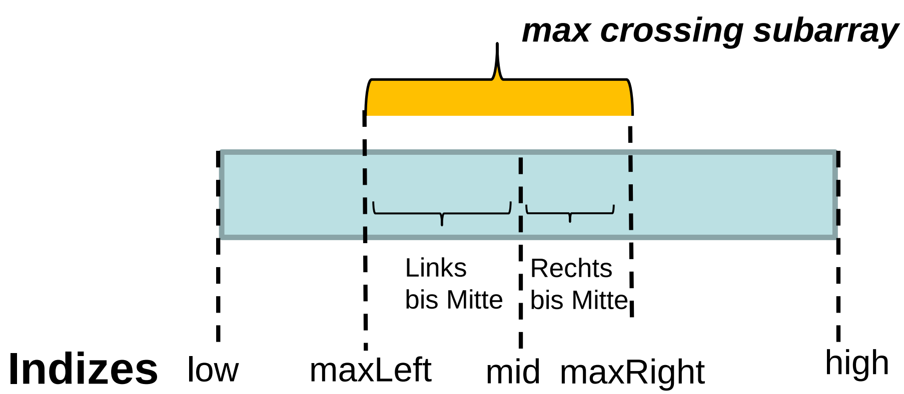


Mögliche Lagen des gesuchten maximalen Subarrays

Version 2.0: Divide-and-Conquer, Fall C

- **Fall C:** ***Kein*** Teilproblem, sondern ***anderes*** Problem!
  - Eine solche Lösung muss (!) die Mitte enthalten.
  - Heißt im Folgenden: "***Maximum Crossing Subarray***"

- **Idee**
  - Max. Crossing Subarray enthält Mitte $A[mid]$ und besteht aus 2 Teilen.
    - **Links** bis zur Mitte: $\mathrm{A}[maxLeft..mid]$ mit $low \leq maxLeft \leq mid$ und
    - **Rechts** ab der Mitte: $\mathrm{A}[(mid+1)..maxRight]$ mit $mid < maxRight \leq high$.
  - Bestimme beide Teile, indem man von der Mitte nach links (rechts) läuft und jeweils die Summe "aktualisiert" (s. nächste Folie).



Version 2.0: Divide-and-Conquer, Fall C

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)

    leftSum = -∞
    sum = 0, maxLeft = mid
    for i = mid downto low
        sum = sum + A[i]
        if sum > leftSum
            leftSum = sum
            maxLeft = i
    rightSum = -∞
    sum = 0, maxRight = mid + 1
    for j = mid+1 to high
        sum = sum + A[j]
        if sum > rightSum
            rightSum = sum
            maxRight = j
    return (maxLeft, maxRight, leftSum + rightSum)
```

Wende Algorithmus auf das Teilarray A[low..high] an. Die Mitte liegt bei Index "mid".

"Links bis zur Mitte"

"Rechts bis zur Mitte"

Berechne Lage (linker und rechter Grenzindex) und Summe des Max.Crossing Subarrays

Laufzeit: $\boldsymbol{\Theta(n)}$

Warum?

Quellcode: MaximumSubarrayRecursive.java

Version 2.0: Divide-and-Conquer

Berücksichtige nun alle Fälle!

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
    if high == low
       return (low, high, A[low]) // only 1 element

    else mid = ⌊(low+high)/2⌋
       (leftLow, leftHigh, leftSum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
       (rightLow, rightHigh, rightSum)=
          FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
       (crossLow, crossHigh, crossSum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)

       if leftSum ≥ rightSum and leftSum ≥ crossSum
          return  (leftLow, leftHigh, leftSum)
       elseif rightSum ≥ leftSum and rightSum ≥ crossSum
          return (rightLow, rightHigh, rightSum)
       else return (crossLow, crossHigh, crossSum)
```

Wende Algorithmus auf das Teilarray A[low..high] an!

Fall A: linke Lösung

Fall B: rechte Lsg.

2 rekursive Aufrufe!

Fall C

$\boldsymbol{\Theta(n)}$

Welche Lösung ist die beste?

Quellcode: MaximumSubarrayRecursive.java

Version 2.0: Divide-and-Conquer, Analyse

Annahmen

- Größe des Eingabearrays ist eine 2er Potenz
- $T(n)$: Laufzeit des Algorithmus bei einem Subarray mit *n* Elementen.

Terminierung

- *high* == *low*, d.h. *n* = 1 ⇒ $T(n) = \Theta(1)$

Rekursion

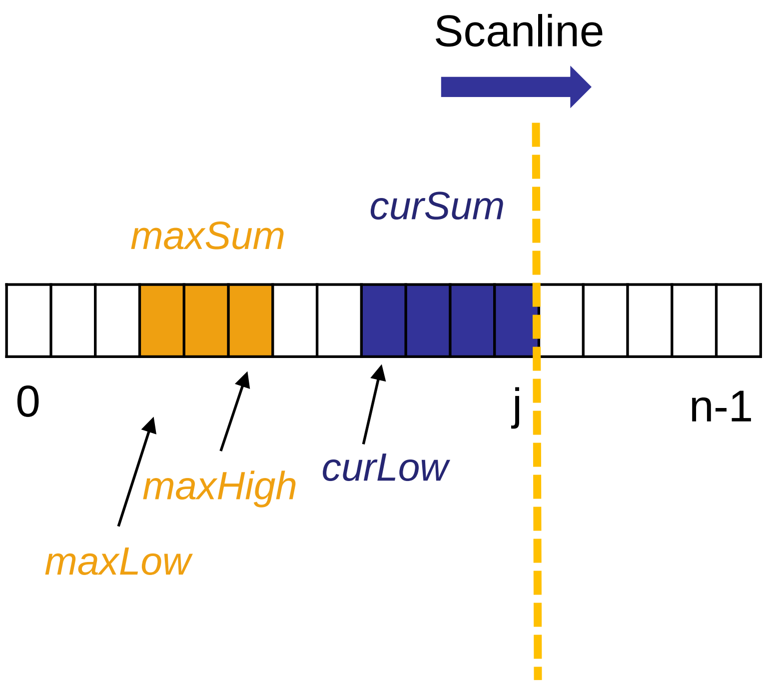
- ***Divide***: Aufteilen benötigt $\Theta(1)$
- ***Conquer***: 2 Teilprobleme mit *n*/2 Elementen ⇒ $2 \cdot T(\frac{n}{2})$
- ***Combine:*** Aufruf von FIND-MAX-CROSSING-SUBARRAY $\Theta(n)$ und eine konstante Anzahl an Tests ⇒ $\Theta(n) + \Theta(1)$
- **Rekursionsgleichung** zur Abschätzung der Laufzeit: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$
- Lösung der Rekursionsgleichung: $T(n) = \boldsymbol{\Theta(n \log n)}$
  - Siehe später!

Version 3.0: Lineare Laufzeit

- Maximum Subarray ist sogar in $\boldsymbol{\Theta(n)}$ bestimmbar.
    - Implementierung + Details: siehe Übung!

- **Idee: „Scanline“**
    - Lese jeden Wert des Arrays nur ***einmal*** und zwar von links nach rechts.
    - Speichere **Zusatzinfo** (=bisherige Maxima) Variablen.

- **Variablen**
    - ***j***: Index der aktuellen Leseposition
    - Aktueller Wissensstand bzgl. Maximum Subarray
        - ***maxLow*** und ***maxHigh***: Linker und rechter Index
        - ***maxSum***: Summe der Werte
    - Maximum Subarray, das an aktueller Leseposition *j* **endet.**
        - ***curLow:*** Linker Index
        - ***curSum***: Summe der Werte



Übung: Führe Scanline auf [-2, 3, 9, -11, 4] durch!

Version 3.0 (siehe Übung)

Pseudocode ist unvollständig und wird in Übung implementiert und besprochen!

```
MAX-SUBARRAY-SCANLINE(A)
    n = A.length
    maxSum = -∞
    curSum = -∞
    for j = 0 to n − 1
        if curSum > 0
            // curSum aktualisieren
        else
            // EdgeMaxSubarray neu beginnen
            // nur aktuelles Element bildet EdgeMaxSubArray

        if curSum > maxSum
            // EdgeMaxSubbaray ist besser als bisher
            // gefundenes MaximumSubArray

    return (maxLow, maxHigh, maxSum)
```

Eingabearray

Index des linken Rands (Lage)

Index des rechten Rands (Lage)

Summe der Werte in Teilarray

Publikums-Joker

Welche der folgenden Aussagen ist **falsch**?

A. Divide-and-Conquer liefert ***eine*** gültige Lösung des Problems.

B. Divide-and-Conquer liefert ***nicht*** die bestmögliche Lösung bzgl. der Laufzeit.

C. Bei der Divide-and-Conquer Lösung erfolgen ***3 rekursive*** Aufrufe.

D. Es kann ***keine*** schnellere Lösung des Problems als O(n) geben.

Inhalt

- Türme von Hanoi
- Maximum-Subarray-Problem
- **Laufzeitanalyse von rekursiven Algorithmen**
- Multiplikation großer Zahlen, Karatsuba [3]

Wdh.: Maximum Subarray - Divide-and-Conquer

FIND-MAXIMUM-SUBARRAY(*A, low, high*)

```
if high == low
   return (low, high, A[low])     // base case
```

$\boldsymbol{\Theta}(\mathbf{1})$

```
else mid = ⌊(low+high)/2⌋
   (leftLow, leftHigh, leftSum) =
      FIND-MAXIMUM-SUBARRAY(A, low, mid)
   (rightLow, rightHigh, rightSum)=
      FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
```

2 rekursive Aufrufe $2\,T(\frac{n}{2})$!

```
   (crossLow, crossHigh, rightSum) =
      FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
```

$\boldsymbol{\Theta}(\boldsymbol{n})$

```
   if leftSum ≥ rightSum and leftSum ≥ crossSum
      return  (leftLow, leftHigh, leftSum)
   elseif rightSum ≥ leftSum and rightSum ≥ crossSum
      return (rightLow, rightHigh, rightSum)
   else return (crossLow, crossHigh, crossSum)
```

$\boldsymbol{\Theta}(\mathbf{1})$

Laufzeit von rekursiven Algorithmen

Laufzeitberechnung

- Laufzeit lässt sich bei Divide & Conquer oft rekursiv beschreiben.
  - Beispiel: Maximum Subarray Problem
  - $T(n) = \begin{Bmatrix} 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & f\text{ür } n > 1 \\ \Theta(1) & f\text{ür } n = 1 \end{Bmatrix}$
- Wie erhält man eine ***geschlossene*** Form für diese Funktion?
  - Hier: $T(n) = \Theta(n \log n)$

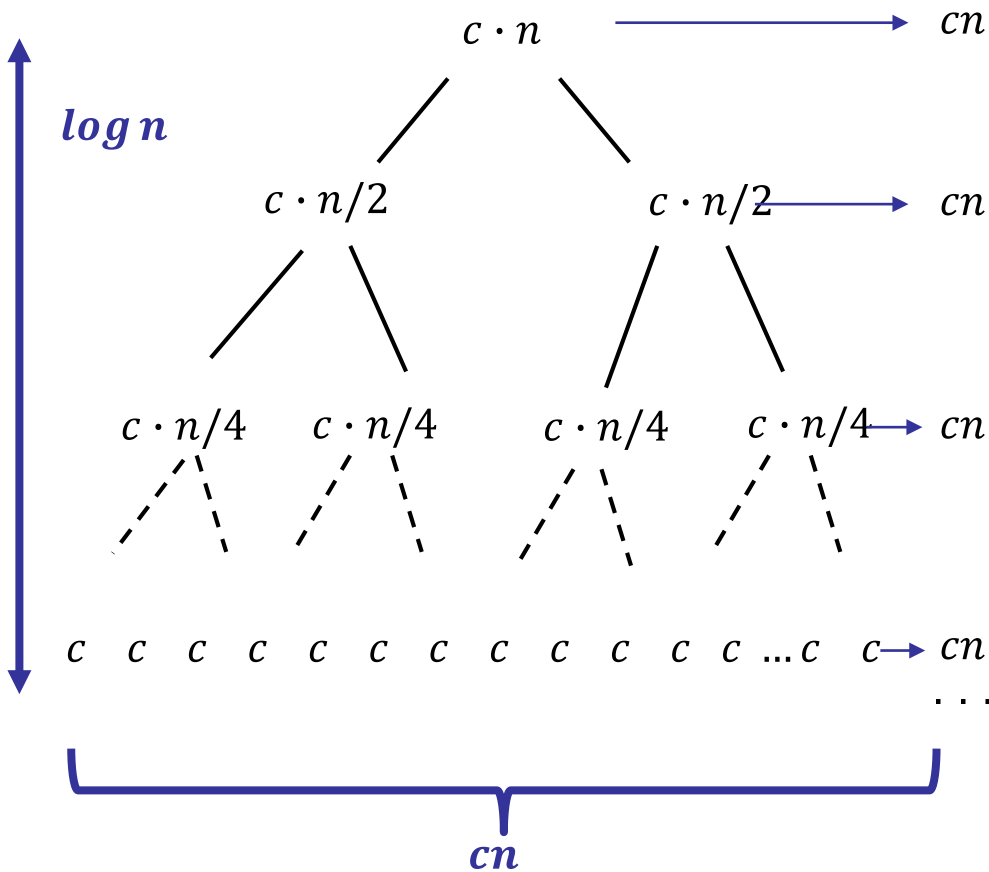
2 Möglichkeiten zur Bestimmung der geschlossenen Form

- *Rekursionsbaum* erraten + *Induktionsbeweis*
- *Mastertheorem (nicht Teil der Vorlesung)*

Rekursionsbaum

- **Annahme**
  - *c*: Konstante, die *Terminierungsfall* **und** *Divide*- und *Combine*-Komponenten beschreibt.
  - $T(n) = \begin{cases} 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n & \text{für } n > 1 \\ c & \text{für } n = 1 \end{cases}$
- **Höhe des Baumes:** $\log n$
- **In jeder Zeile Kosten** $c \cdot n$
- **Lösung**
  - $T(n) = \boldsymbol{\Theta(n \log n)}$
  - Erraten anhand Rekursionsbaum
  - Beweis durch Induktion (hier weggelassen)

Rekursionsbaum:
„Expandiere rekursive Aufrufe“



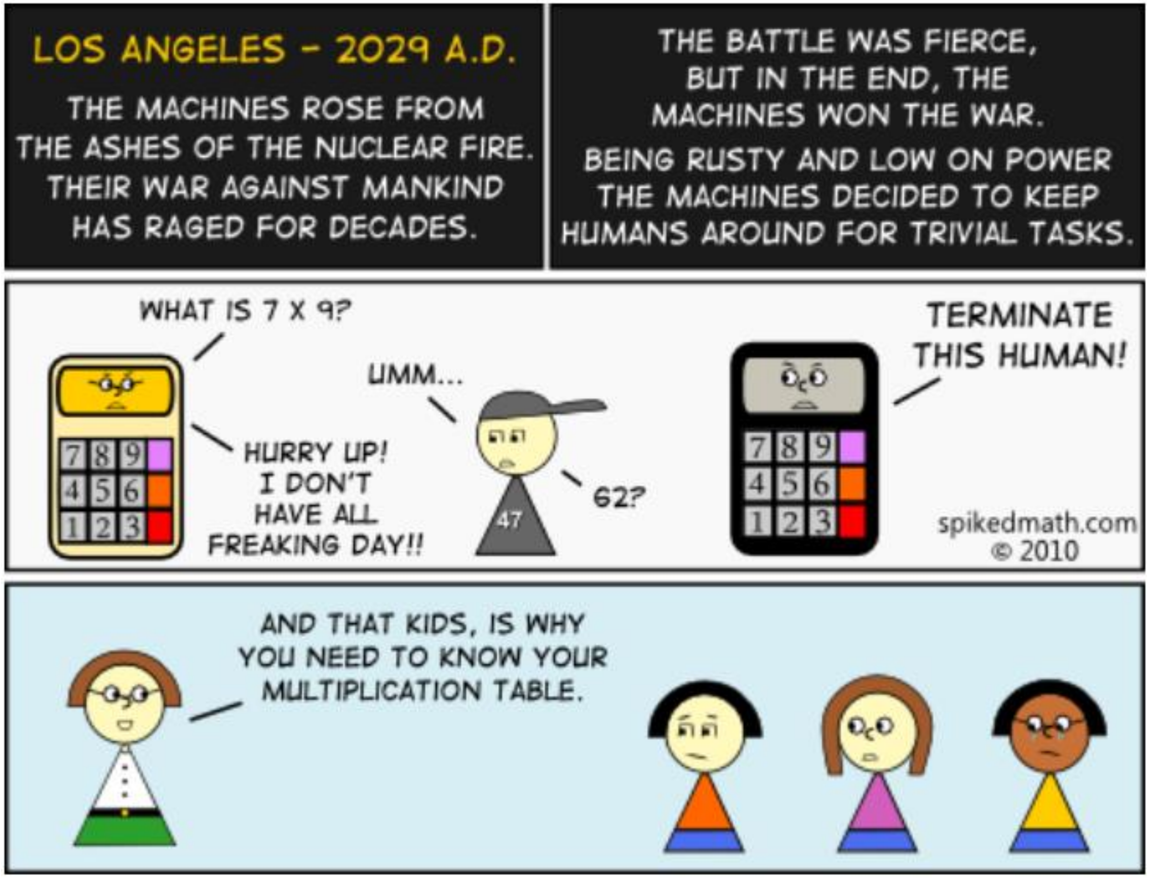
Gesamt: $\approx cn \log n$

Inhalt

- Türme von Hanoi
- Maximum-Subarray-Problem
- Laufzeitanalyse von rekursiven Algorithmen
- **Multiplikation großer Zahlen, Karatsuba [5]**

Motivation

- Wie multipliziert man ***effizient*** 2 große Zahlen?
- ***Grundidee***
  - Divide & Conquer
  - Minimale Anzahl rekursiver Aufrufe
- ***Annahme***
  - Beide Zahlen haben ***gleich viele*** Ziffern.
  - Die Anzahl der Ziffern beider Zahlen sei ***n*** (***n-stellige Zahl***)
  - ***n*** ist eine 2er Potenz, z.B. 8, 16, 32



Quelle: [4]

Multiplikation von 2-stelligen Zahlen (n = 2)

- ***Schulmethode***:
  - Ansatz: Ziffer * Ziffer und Additionen
- ***Grundoperationen***
  - Einstellige Multiplikationen: z.B. 2 * 5
  - Einstellige Additionen, z.B. 2 + 3
- Wie viele Multiplikationen werden benötigt?
  - Hier: 4 einstellige Multiplikationen
  - Allgemein: $\boldsymbol{O(n^2)}$ Multiplikationen falls 2 n-stellige Zahlen multipliziert werden sollen.
  - Hinweis: Additionen werden vernachlässigt.
- Geht es mit weniger Multiplikationen?

a = pq, **b** = r s

	78 * 21
2*8	0160
2*7	1400
1*8	0008
1*7	0070
+	1638

Schulmethode "Langform"

	78 * 21
2*78	156
1*78	78
+	1638

Schulmethode "Kurzform"

Schulmethode im Detail für n = 2

- Zahl zerlegt in Ziffern
  - $a = 10p + q$
  - $\mathrm{b} = 10r + s$

- Mit dieser Darstellung ergibt sich:
  - $a \cdot b = 100(p \cdot r) + 10(p \cdot s + q \cdot r) + q \cdot s$
  - ***4 Multiplikationen***

- Die Karatsuba-Methode kommt dagegen mit 3 Multiplikationen aus!

a **b**
pq r s

	78 * 21
2*8	16
2*7	14
1*8	08
1*7	07
+	1638

Schulmethode "Langform"

Karatsuba für n = 2

- Berechne die folgenden ***3 Werte***
  - $u = p \cdot r$
  - $v = q \cdot s$
  - $w = (q - p) \cdot (s - r)$
    - "Differenz der Ziffern"
    - Kann negativ werden!
- Warum ist das ***korrekt***? Zeige:
  - $100 \cdot u + 10 \cdot (u + v - w) + 1 \cdot v = (10p + q) \cdot (10r + s)$
- Beobachtung bzgl. Laufzeit:
  - Nur noch ***3 Multiplikationen!***
  - Overhead: Mehr Additionen und Subtraktionen!

Karatsuba

	a	b
	pq	r s

	78 * 21
100u	14 00
10u	140
10v	080
v	08
	1628
-10w	-010
	1638

Achtung: $-(-1) = 1$!

Karatsuba für $\boldsymbol{n = 4 = 2^2}$

- Führe die Multiplikation zweier 4-stelligen Zahlen auf die Multiplikation von drei *2*-stelligen Zahlen zurück.

- Für $\boldsymbol{n = 4 = 2^2}$
  - Berechne: $u = p \cdot r$
  - Berechne: $\mathrm{v} = q \cdot s$
  - Berechne: $\mathrm{w} = (q - p) \cdot (s - r)$
  - $a \cdot b = 10^4 u + 10^2 (u + v - w) + v$

$$\underbrace{\overbrace{5\;6}^{\substack{\mathbf{p}\\ n/2}}\;\overbrace{7\;8}^{\substack{\mathbf{q}\\ n/2}}}_{n\text{ Ziffern}} \quad * \quad \underbrace{\overbrace{4\;3}^{\substack{\mathbf{r}\\ n/2}}\;\overbrace{2\;1}^{\substack{\mathbf{s}\\ n/2}}}_{n\text{ Ziffern}}$$

- Konkretes Zahlenbeispiel: 5678*4321
  - Weiteres Beispiel, siehe Übung!

Karatsuba für Werte $\boldsymbol{n = 2^k}$ mit $\boldsymbol{k > 2}$

- ***Divide-and-Conquer***
  - Multiplikation von ***zwei*** *n*-stelligen Zahlen wird rekursiv auf Multiplikation von ***drei*** *n*/2-stellige Zahlen zurückgeführt.
  - Multiplikation von ***zwei*** *n/2*-stelligen Zahlen wird rekursiv auf Multiplikation von ***drei*** *n*/4-stellige Zahlen zurückgeführt.
  - usw.
  - Terminierung: Multiplikation von 1-stelligen Zahlen.

- ***Allgemeiner Kombinationsschritt:***
  - *n*-stellige Zahlen in *n*/2-stellige Zahlen *p*, *q*, *r, s* zerlegt (wie vorher)
  - Kombination: $a \cdot b = \ 10^n u + 10^{\frac{n}{2}}(u + v - w) + v$

- ***Abschätzung der Laufzeit***: $T(n) = 3 \cdot T\left(\frac{n}{2}\right) = n^{\mathrm{ld}3} = \boldsymbol{n^{1{,}58}}$
  - Signifikante Verbesserung im Vergleich zu $O(n^2)$ der Schulmethode!

Karatsuba: Code

KARATSUBA(*a*, *b*) a,b: large integer values to be multiplied

```
n = max{a.bitLength(), b.bitLength)
if (n <= threshold)
    return a * b;                  // use regular multiplication

//otherwise Karatsuba: compute p, q, r, s from a
// a = 2^(n/2) * p + q and  b = 2^(n/2) * r + s
n2 = n ÷ 2 + n % 2        // compute n/2 (divide by 2, round up)
p = a ≫ n2;               // p = a ÷ 2^(n/2) (division expressed as shift operation)
q = a − (p ≪ n2);         // q = a − 2^(n/2) · p
r = b ≫ n2;               // r = b ÷ 2^(n/2)
s = b − (r ≪ n2);         // s = b − 2^(n/2) · r

// compute subexpressions u, v, w, recursion!
u = karatsuba(p,r)
v = karatsuba(q,s)
v = karatsuba(p − q, s − r)

// combination phase
return u · 2^(2*n2) + (u + v − w)2^(n2) + v;
```

return result of multiplication

Quellcode: Karatsuba.java

Publikums-Joker

Welche der folgenden Aussagen bzgl. Karatsuba ist **falsch**?

A. Die Multiplikation nach Karatsuba kann man für das folgende Problem anwenden: 5432 * 890.

B. Die Multiplikation nach Karatsuba kann man für das folgende Problem anwenden: 543 * 123.

C. Die Multiplikation nach Karatsuba benötigt genauso so viele Additionen wie die klassische Schulmethode.

D. Karatsuba sollte man nicht für einfache Multiplikationen anwenden.

Zusammenfassung

- Divide & Conquer
  - Wichtiges Prinzip zum Entwickeln (effizienter) Algorithmen
  - Klassisches Beispiel: Türme von Hanoi

- Maximum-Subarray-Problem
  - Divide-and-Conquer muss nicht das schnellste sein.

- Laufzeitanalyse von rekursiven Algorithmen

- Karatsuba-Algorithmus: Multiplikation großer Zahlen [3]
  - Klassische Divide-and-Conquer-Anwendung

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] http://dilbert.com/strip/2010-06-11 (abgerufen am 07.10.2016)

[3] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 1.2.3, 5. Auflage, Spektrum Akademischer Verlag, 2012. (eBook)

[4] http://spikedmath.com/326.html (abgerufen am 08.10.2016)

[5] Vöcking et al. Taschenbuch der Algorithmen, Kapitel 11 (Karatsuba)   Springer Verlag, 2008