11 Divide-and-Conquer und Rekursionsgleichungen

Divide-and-Conquer

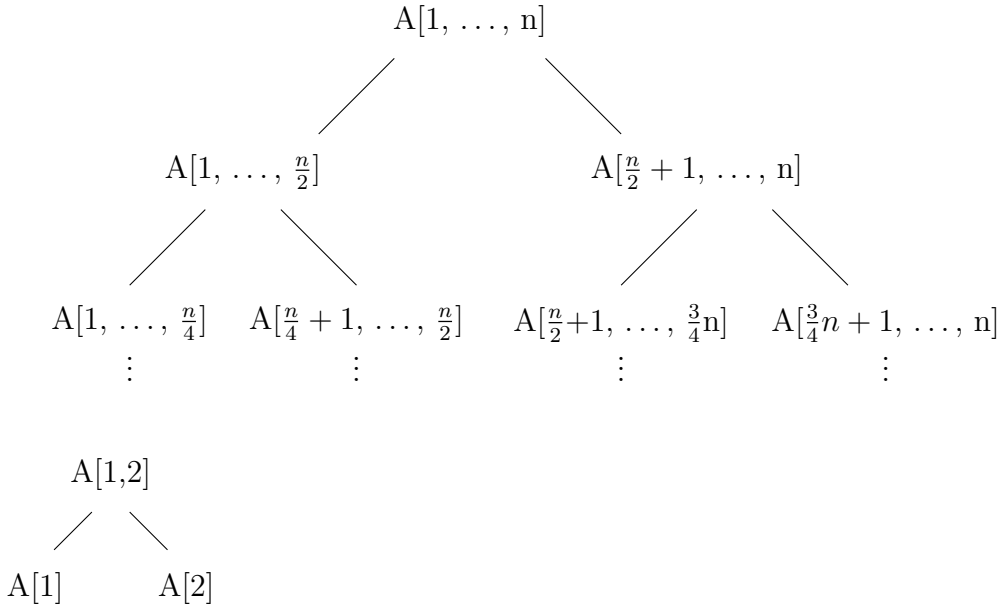
- Problem aufteilen in Teilprobleme
- Teilproblem (rekursiv) lösen
- Lösungen der Teilprobleme zusammensetzen

Typische Beispiele: Binäre Suche, Meregesort, Quicksort

Mergesort

```
Mergesort(A[1, ..., n]){
1. if (n==1) oder (n==0) return A;
2. B_1 = Mergesort (A[1, ..., n/⌊2⌋]);
3. B_2 = Mergesort (A[n/⌊2⌋+1, ..., n]); //2. + 3. divide-Schritte
4. return "Mischung von B_1 und B_2"   //Mischung bilden, conquer-Schritt
}
```

Aufrufbaum bei n = Zweierpotenz

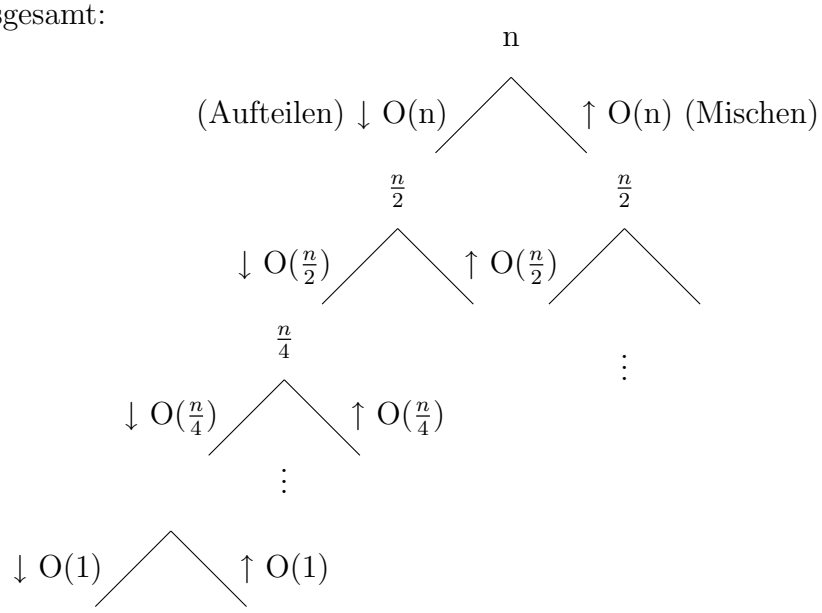


Tiefe: genau $log_2 n$

Laufzeit:

- Blätter: $O(n)$
- Aufteilen: beim m Elementen $O(m)$
- Zusammensetzen von 2-mal $\frac{n}{2}$ Elementen: $O(m)$

Damit insgesamt:



Für die Blätter: $n \cdot O(1) = O(n)$

Für das Aufteilen: $c \cdot n + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + \cdots + c \cdot 1 + c \cdot \frac{n}{2} + \ldots$ Wo soll das enden?

Wir addieren in einer anderen Reihenfolge:

1	$c \cdot n$
2	$+c \cdot \frac{n}{2} + c \cdot \frac{n}{2}$
3	$+c \cdot (\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4})$
4	$+c \cdot (\frac{n}{8} + \cdots + \frac{n}{8}$
⋮	
log n	$+c \cdot \underbrace{(1 + \cdots + 1)}_{\text{n-mal}}$

$= logn \cdot c \cdot n = O(n \cdot logn)$

Ebenso für das Mischen: $O(n \cdot logn)$

Insgesamt $O(n \cdot logn) + O(n) = O(n \cdot logn)$.

- Wichtig: Richtige Reihenfolge beim Zusammenaddieren. Bei Bäumen oft stufenweise!

- Die eigentliche Arbeit beim divide-and-conquer geschieht im Aufteilen und im Zusammenfügen. Der Rest ist rekusiv.
- Mergesort einfach bottom-up ohne Rekursion, da starrer Aufrufbaum.
  $A[1,2], A[3,4], \ldots, A[n-1,n]$ Sortieren
  $A[1,\ldots,4], A[5,\ldots,8], \ldots$ Sortieren
  $\vdots$
  Auch $O(n \cdot logn)$.
- Ebenfalls Heapsort sortiert in $O(n \cdot logn)$.
- Bubblesort, Insertion Sort, Selection Sort $O(n^2)$

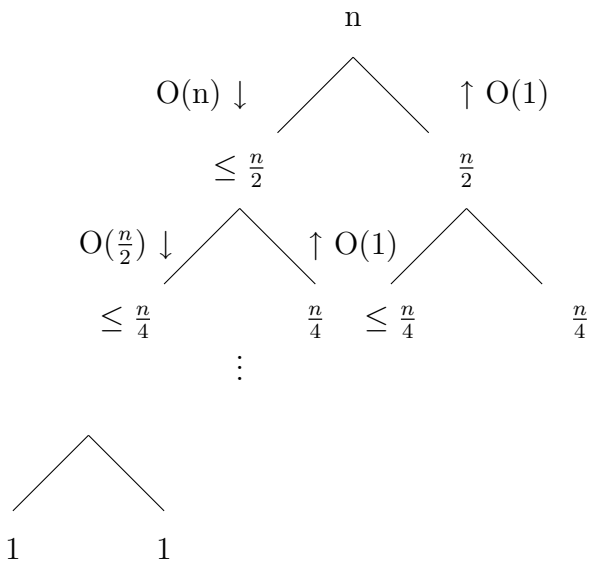
11.1 Algorithmus: Quicksort

Eingabe: array A[1, . . . , n] of „geordneter Datentyp“

```
1. if (n == m) return
2. a = ein A[i];
3. Lösche A[i] aus A.
4. for j=1 to n {
5.    if (A[j] ≤ a){
        A[j] als nächstes Element zu B1;
        break;
        }
6. if (A[j] > a){
     A[j] zu B2}
   }
7. A = B1 a B2
8. if (B1 ≠ ∅) Quicksort (B1)          // B1 als Teil von A
9. if (B2 ≠ ∅) Quicksort (B2)          // B2 als Teil von A
```

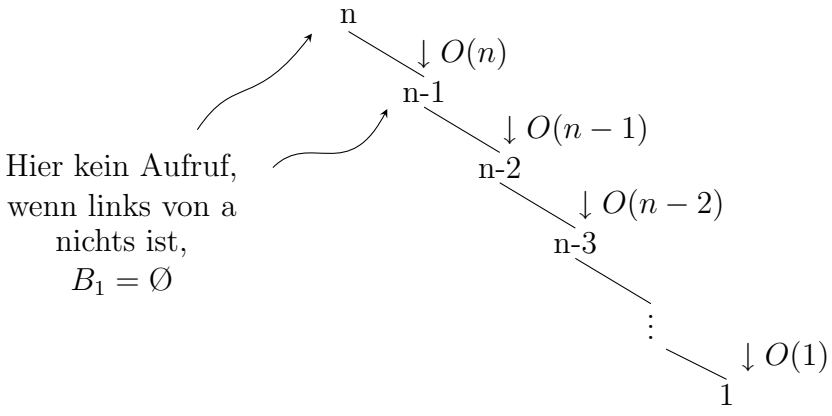
Beachte:
Die Prozedur Partition aus dem 1. Semester erlaubt es, mit dem array A alleine auszukommen.

Prozedurbäume:



Hier $O(n \cdot logn)$: $O(n) + 2 \cdot O(\frac{n}{2}) + 4 \cdot O(\frac{n}{4}) + \cdots + n \cdot O(1) + \underbrace{O(n)}_{\text{Blätter}}$

Aber auch:



Laufzeit:

$$\begin{aligned} n + n - 1 + \cdots + 3 + 2 + 1 = \frac{n \cdot (n-1)}{2} &= O(n^2) \quad \text{(Aufteilen)} \\ 1 + 1 + \cdots + 1 + 1 + 1 &= O(n) \quad \text{(Zusammensetzen)} \end{aligned}$$

Also: $O(n^2)$ und auch $\Omega(n^2)$.

Wieso Quicksort trotzdem in der Praxis häufig? In der Regel tritt ein gutartiger Baum auf, und wir haben $O(n \cdot logn)$. Vergleiche immer mit festem a, a in einem Register $\Longrightarrow$ schnelleres Vergleichen. Dagegen sind beim Mischen von Mergesort öfter beide Elemente des Vergleiches neu.

Aufrufbaum von Quicksort vorher nicht zu erkennen. Keine nicht-rekursive Implementierung wie bei Mergesort. Nur mit Hilfe eines (Rekursions-) Kellers.

Noch einmal zu Mergesort. Wir betrachten den Fall, dass n eine Zweierpotenz ist, dann $\frac{n}{2}, \frac{n}{4}, \ldots, 1 = 2^0$ ebenfalls. Sei $T(n)$ = worst-case-Zeit bei $A[1, \ldots, n]$. Dann gilt für ein geeignetes c:

$$\begin{aligned} T(n) &\leq c \cdot n + 2 \cdot T(\frac{n}{2}) \\ T(1) &\leq c \end{aligned}$$

- Einmaliges Teilen O(n)
- Mischen O(n)
- Aufrufverwaltung hier O(n)

Betrachten nun

$$\begin{aligned} T(n) &= c \cdot n + 2 \cdot T(\frac{n}{2}) \\ T(1) &= c \end{aligned}$$

Dann durch Abwickeln der rekursiven Gleichung:

$$\begin{aligned} T(n) &= c \cdot n + 2 \cdot T(\frac{n}{2}) \\ &= c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 2 \cdot 2 \cdot T(\frac{n}{4}) \\ &= c \cdot n + c \cdot n + 4 \cdot c \cdot \frac{n}{2} + 2 \cdot 2 \cdot 2 \cdot T(\frac{n}{8}) \\ &\vdots \\ &= c \cdot n \cdot logn + 2 \cdot 2 \cdot \cdots \cdot 2 \cdot T(1) \\ &= c \cdot n \cdot logn + c \cdot n \\ &= O(c \cdot nlogn) = O(n \cdot logn) \end{aligned}$$

Schließlich noch ein strenger Induktionsbeweis, dass $(T(n) = O(n \cdot logn)$.

Induktionsanfang: $T(1) = O(1 \cdot log1) = 0$! stimmt so nicht. Also fangen wir bei $T(2)$ an. $T(2) \leq d \cdot 2$ für ein d geht.

Induktionsschluss:

$$\begin{aligned} T(n) &= c \cdot n + 2 \cdot T(\frac{n}{2}) \\ &\leq c \cdot n + 2 \cdot d \cdot \frac{n}{2} \\ &\leq c \cdot n + 2 \cdot d \cdot \frac{n}{2} \cdot (logn - 1) \\ &= c \cdot n + d \cdot nlogn - d \cdot n \\ &\leq d \cdot nlogn \qquad \text{geht, wenn } c \geq d \text{ gewählt ist.} \end{aligned}$$

Wie sieht das bei Quicksort aus?
$T(n)$ = worst-case-Zeit von Quicksort auf $A[1, \ldots, n]$. Dann

$$\begin{aligned} T(n) &\leq c \cdot n + T(n-1) \\ T(n) &\leq c \cdot n + T(1) + T(n-2) \\ T(n) &\leq c \cdot n + T(2) + T(n-3) \\ &\vdots \\ T(n) &\leq c \cdot n + T(n-2) + T(1) \\ T(n) &\leq c \cdot n + T(n-1) \end{aligned}$$

Also

$$\begin{aligned} T(1) &\leq c \\ T(n) &\leq c \cdot n + \max(\{T(n-1)\} \cup \{T(i) + T(n-i-1) | 1 \leq i \leq n-1\}) \end{aligned}$$

Dann $T(n) \leq dn^2$ für d geeignet.

Induktionsanfang: ✓

Indukstionsschluss:

$$\begin{aligned} T(n) &\leq c \cdot n + Max(\{d(n-1)^a\} \cup \{\underbrace{d \cdot i^2 + d(n-i-1)^2}_{i+n-i-1-n-1} | 1 \leq i \leq n-1\}) \\ &\leq c \cdot n + d(n-1)^2 \\ &\leq d \cdot n^2 \qquad \text{für } d \geq c \end{aligned}$$

Aufgabe: Stellen Sie die Rekursionsgleichung für eine rekursive Version der binären Suche auf und schätzen sie diese bestmöglich ab.

11.2 Multiplikation großer Zahlen

Im Folgenden behandeln wir das Problem der Multiplikation großer Zahlen. Bisher haben wir angenommen: Zahlen in ein Speicherwort $\Longrightarrow$ arithmetische Ausdrücke in O(1).

Man spricht vom uniformen Kostenmaß. Bei größeren Zahlen kommt es auf den Multiplikationsalgorithmus an. Man misst die Laufzeit in Abhängigkeit von der # Bits, die der Rechner verarbeiten muss. Das ist bei einer Zahl n etwa $log_2 n$ (genauer für $n \geq 1 \llcorner log_2 n \lrcorner + 1$).

Man misst nicht in der Zahl selbst!

Die normale Methode, zwei Zahlen zu addieren, lässt sich in O(n) Bitoperationen implementieren bei Zahlen der Länge n:

$$\begin{array}{rcccc} & a_1 & \ldots & a_n \\ + & b_1 & \ldots & b_n \\ \hline & & \ldots & a_n + b_n \end{array}$$

Multiplikation in $O(n^2)$:

$$(a_1 \ldots a_n) \cdot (b_1 \ldots b_n) =$$

$$\begin{array}{rl} (a_1 \ldots a_n) \cdot b_1 & O(n) \\ (a_1 \ldots a_n) \cdot b_2 & O(n) \\ & \vdots \\ (a_1 \ldots a_n) \cdot b_n & O(n) \\ \hline \end{array}$$

Summenbildung

$n-1$ Additionen mit Zahlen der Länge $\leq 2 \cdot n \Longrightarrow O(n^2)$!

Mit divide-and-conquer geht es besser! Nehmen wir zunächst einmal an, n ist eine Zweierpotenz. Dann

$$\begin{array}{l} \overbrace{a_1 \ldots a_n}^{a:=} = \overbrace{a_1 \ldots a_{\frac{n}{2}}}^{a':=} \overbrace{a_{\frac{n}{2}-1} \ldots a_n}^{a'':=} \\ \underbrace{b_1 \ldots b_n}_{b:=} = \underbrace{b_1 \ldots b_{\frac{n}{2}}}_{b':=} \underbrace{b_{\frac{n}{2}-1} \ldots b_n}_{b'':=} \end{array}$$

Nun ist

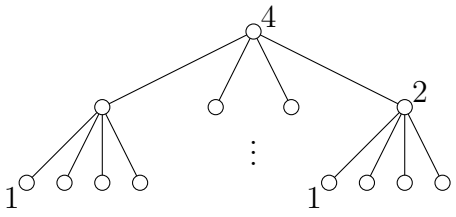
$$a \cdot b = \underbrace{(a' \cdot 2^{\frac{n}{2}} + a'')}_{a=} \cdot \underbrace{b' \cdot 2^n 2 + b''}_{b=}$$
$$= \underbrace{a'}_{\frac{n}{2}} \cdot \underbrace{b'}_{\frac{n}{2}} \cdot 2^n + \underbrace{a'}_{\frac{n}{2}} \cdot \underbrace{b''}_{\frac{n}{2}} \cdot 2^{\frac{n}{2}} + a'' \cdot b' \cdot 2^{\frac{n}{2}} + a'' \cdot b''$$

Mit den 4 Produkten rekursiv weiter. Das Programm sieht in etwa so aus:

```
Mult(a, b, n){                                  //a=a_1, ...a_n, b=b_1, ..., b_n

  1. if (n == 1) return a · b                   // n Zweierpotenz
  2. a', a'', b', b'' wie oben.                 // Divide
  3. m_1:= Mult(a', b', n/2);
  4. m_2:= Mult(a', b'', n/2);
  5. m_3:= Mult(a'', b, n/2);
  6. m_4:= Mult(a'', b'', n/2);
  7. return (m_1 · 2^n + (m_2+m_3) · 2^(n/2) + m_4)  //Conquer
}
```

Aufrufbaum, n = 4:



Tiefe $\log_2 4$

Laufzeit? Bei Mult(a,b,n) braucht der divide-Schritt und die Zeit für die Aufrufe selbst zusammen $O(n)$. Der conquer-Schritt erfolgt auch in $O(n)$. Zählen wir wieder pro Stufe:

1. Stufe $d \cdot n$ (von $O(n)$)
2. Stufe $4 \cdot d \cdot \frac{n}{2}$
3. Stufe $4 \cdot 4 \cdot d \cdot \frac{n}{4}$

$\vdots$ $log_2 n-$te Stufe $4^{log_2 n - 1} \cdot \cdot \frac{n}{4^{log_2 n} \cdot d}$

Blätter: $4^{log_2 n} \cdot d$

Das gibt:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{log_2 n} 4^i \cdot d \cdot \frac{n}{2^i} \\ &= d \cdot n \cdot \sum_{i=0}^{log_2 n} 2^i \\ &= d \cdot n \cdot \frac{2^{log_2 n+1} - 1}{2-1} \\ &= O(n^2) \qquad \text{mit } \sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1} \end{aligned}$$

für alle $x \neq 1$ (x>0, x<0 egal!), geometrische Reihe.

Betrachten wir die induzierte Rekursionsgleichung. Annahme: n = Zweierpotenz. (Beachte: $2^{\lfloor log_2 n \rfloor} \leq n \leq 2^{\lfloor log_2 n \rfloor + 1}$, d.h. zwischen n und $2n$ existiert eine Zweierpotenz.)

$$\begin{aligned} T(1) &= d \\ T(n) &= dn + 4 \cdot T(\frac{n}{2}) \end{aligned}$$

Versuchen $T(n) = O(n^2)$ durch Induktion zu zeigen.

1. Versuch: $T(n) \leq d \cdot n^2$

Induktionsanfang: ✓

Induktionsschluss:

$$\begin{aligned} T(n) &= dn + 4 \cdot T(\frac{n}{2}) \\ &\leq d \cdot n + dn^2 > dn^2 \qquad \text{Induktion geht nicht!} \end{aligned}$$

Müssen irgendwie das dn unterbringen.

2. Versuch $T(n) \leq 2dn^2$ gibt ein Induktionsschluss.

$$T(n) \leq dn + 2dn^2 > 2dn^2$$

3. Versuch $T(n) \leq 2dn^2 - dn$

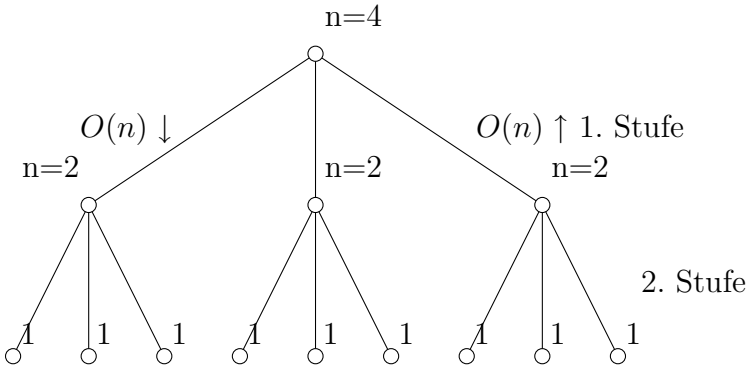
Induktionsanfang: $T(1) \leq 2d - d = d$✓

Induktionsschluss:

$$\begin{aligned} T(n) &= dn + 4 \cdot T(\frac{n}{2}) \\ &\leq dn + 4(2d(\frac{n^2}{4} - d\frac{n}{2}) \\ &= dn + 2dn^2 - 2dn \\ &= 2dn^2 - dn \end{aligned}$$

Vergleiche auch Seite 164, wo $log_2 \frac{n}{2} = log_2 n - 1$ wichtig ist.

Nächstes Ziel: Verbesserung der Multiplikation durch nur noch drei rekursive Aufrufe mit jeweils $\frac{n}{2}$ vielen Bits. Analysieren wir zunächst die Laufzeit.



$log_2 n$ divide-and-conquer-Schritte + Zeit an n Blättern.

1. Stufe $d \cdot n$,
2. Stufe $3 \cdot d \cdot \frac{n}{2}$,
3. Stufe $3 \cdot 3 \cdot d \cdot \frac{n}{4}, \ldots,$
$\vdots$
$log_2 n - te$ Stufe Blätter $3^{log_2 n} \cdot d \cdot 1$

Dann ist

$$\begin{aligned}
T(n) &= \sum_{i=0}^{log_2 n} 3^i \cdot d \cdot \frac{n}{2^i} \\
&= d \cdot n \cdot \sum (\frac{3}{2})^i \\
&= d \cdot n \cdot \frac{(\frac{3}{2})^{log_2 n+1} - 1}{\underbrace{\frac{3}{2} - 1}_{=\frac{1}{2}}} \\
&\leq \not{2} \cdot d \cdot n \cdot \frac{3}{\not{2}} \cdot (\frac{3}{2})^{log_2 n} \\
&= 3 \cdot d \cdot n \cdot 2^{log_2(\frac{3}{2}) \cdot log_2 n} \\
&= 3 \cdot d \cdot n \cdot n^{\overbrace{log_2(\frac{3}{2})}^{<1}} \\
&= 3 \cdot d \cdot n^{log_2 3} = O(n^{1.59})
\end{aligned}$$

Also tatsächlich besser als $O(n^2)$.

Fassen wir zusammen:

2 Aufrufe mit $\frac{n}{2}$, linearer Zusatzaufwand

$$\begin{aligned}
T(n) &= d \cdot n \cdot \sum_{i=0}^{log_2 n} \underbrace{\frac{2^i}{2^i}}_{=1} \\
&= O(n \cdot logn) \qquad \text{(Auf jeder Stufe, d.h. 2 Aufrufe, halbe Größe)}
\end{aligned}$$

3 Aufrufe

$$\begin{aligned}
T(n) &= d \cdot n \sum (\frac{3}{2})^i \qquad \text{(3 Aufrufe, halbe Größe)} \\
&= O(d \cdot n^{log_2 3})
\end{aligned}$$

4 Aufrufe

$$\begin{aligned}
T(n) &= d \cdot n \cdot \sum (\frac{4}{2})^i \qquad \text{(4 Aufrufe, halbe Größe)} \\
&= d \cdot n \cdot n \\
&= O(n^2).
\end{aligned}$$

Aufgabe: Stellen Sie für den Fall der drei Aufrufe die Rekursionsgleichungen auf und beweisen Sie $T(n) = O(n^{log_2 3})$ duch eine ordnungsgemäße Induktion.

Zurück zur Multiplikation.

$$\overbrace{a_1 \dots a_n}^{a:=} = \overbrace{a_1 \dots a_{\frac{n}{2}}}^{a':=} \cdot \overbrace{a_{\frac{n}{2}+1} \dots a_n}^{a'':=}$$
$$\overbrace{b_1 \dots b_n}^{b:=} = \overbrace{b_1 \dots b_{\frac{n}{2}}}^{b':=} \cdot \overbrace{b_{\frac{n}{2}+1} \dots b_n}^{b'':=}$$

Wie können wir einen Aufruf einsparen? Erlauben uns zusätzliche Additionen: Wir beobachten zunächst allgemein:

$$(x-y)\cdot(u-v) = x(u-v) + y\cdot(u-v) = x\cdot u - x\cdot v + y\cdot u - y\cdot v$$

(eine explizite und 4 implizite Multiplikationen (aber in Summe))

Wir versuchen jetzt $a'b'' + a''b'$ auf einen Schlag zu ermitteln:

$$(a'-a'')\cdot(b''-b') = a'b'' - a'b' + a''b'' + a''b'$$

Und: $a'b', a''b''$ brauchen wir sowieso und

$$a'b'' + a''b' = (a'-a'')\cdot(b''-b') + a'b' + a''b''.$$

Das Programm: $\vdots$

```
m1 = Mult(a', b', n/2)
m3 = Mult(a'', b'', n/2)
if (a' - a'' < 0, b''-b' < 0)
    m2 = Mult(a''-a', b'-b'', n/2)//|a'' − a'|, |b' − b''| < 2^(n/2−1)
if (a' - a'' ≥ 0, b''-b' < 0)
⋮
return (m1 · 2^n + (m2 + m1 + m3) · 2^(n/2) + m3
                   Zeit O(n/2)
```

Damit für die Zeit $T(1) \leq d$

$$T(n) \leq d\cdot n + 3\cdot T(\frac{n}{2}) \Leftarrow O(n^{log_2 3}).$$

Also Addition linear, nicht bekannt für Multiplikation.

Wir gehen jetzt wieder davon aus, dass die Basisoperationen in $O(1)$ Zeit durchzuführen sind.

Erinnern wir uns an die Matrizenmultiplikation: quadratische Matrizen.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

und allgemein haben wir bei 2 $n \times n$-Matrizen:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{i=1}^{n} a_{1i}b_{i1} & \sum_{i=1}^{n} a_{1i}b_{i2} \dots & \sum_{i=1}^{n} a_{1i}b_{in} \\ \vdots & & \\ \sum_{i=1}^{n} a_{ni}b_{i1} & \dots & \sum_{i=1}^{n} a_{ni}b_{in} \end{pmatrix}$$

Laufzeit ist, pro Eintrag des Ergebnisses: n Multiplikationen, $n-1$ Additionen, also $O(n)$. Bei n^2 Einträgen $O(n^3)(= O(n^2)^{\frac{3}{2}})$. Überraschend war seinerzeit, dass es besser geht mit divide-and-conquer.

Wie könnte ein divide-and-conquer-Ansatz funktionieren? Teilen wir die Matrizen einmal auf:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$A_{i,j}$ sind $\frac{n}{2} \times \frac{n}{2}$ - Matrizen (wir nehmen wieder an, n 2-er-Potenz)

$$\begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{pmatrix} \sum_{i=1}^{n} a_{1i}b_{i1} & \dots & \sum_{i=1}^{n} a_{1i}b_{in} \\ \vdots & & \\ \sum_{i=1}^{n} a_{ni}b_{i1} & \dots & \sum_{i=1}^{n} a_{ni}b_{in} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Es ist

$$C_11 = \begin{pmatrix} \sum_{i=1}^{n} a_{1i}b_{i1} & \dots & \sum_{i=1}^{n} a_{1i}b_{i\frac{n}{2}} \\ \vdots & & \vdots \\ \sum_{i=1}^{n} a_{\frac{n}{2}i}b_{i1} & \dots & \sum_{i=1}^{n} a_{\frac{n}{2}i}b_{i\frac{n}{2}} \end{pmatrix}$$

Es ist

$$A_{11} \cdot B_{11} = \begin{pmatrix} \sum_{i=1}^{n} a_{1i}b_{i1} & \dots & \sum_{i=1}^{n} a_{1i}b_{i\frac{n}{2}} \\ \vdots & & \vdots \\ \sum_{i=1}^{n} a_{\frac{n}{2}i}b_{i1} & \dots & \sum_{i=1}^{n} a_{\frac{n}{2}i}b_{i\frac{n}{2}} \end{pmatrix}$$