Algorithmen I

Dennis Hofheinz und Henning Meyerhenke

Übungen:
Christian Staudt und Christoph Striecks

Institut für theoretische Informatik
Web:
`https://crypto.iti.kit.edu/algo-sose15`

(Folien von Peter Sanders)

Organisatorisches

Vorlesungen:
Mo: 15:45–17:15
Mi: 14:00–14:45

Saalübung:
Mi: 14:45–15:30

Tutorium: wöchentlich, ab nächster Woche
Einteilung mittels Webinscribe
`https://webinscribe.ira.uka.de/`

Übungsblätter: wöchentlich
Ausgabe Mittwoch nach der Vorlesung/Übung
Abgabe Freitag 12:45 Uhr (9 Tage nach Ausgabe)

Organisatorisches

Sprechstunde (DH): Donnerstag, 10.30–11.30 Uhr
(jederzeit bei offener Tür oder nach Vereinbarung)

- Dennis Hofheinz, Raum 279

Sprechstunde (HM): **ab 18. Mai**: Montag, 13.00–14.00 Uhr
(oder nach Vereinbarung)

- Henning Meyerhenke, Raum 033

Organisatorisches

Mittsemesterklausur: zur Kontrolle

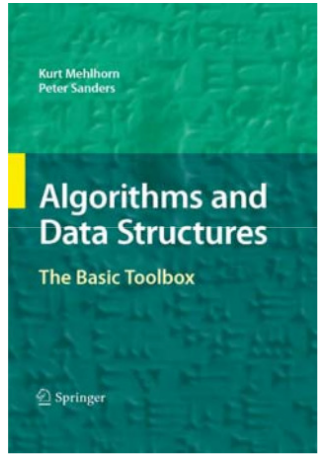
Abschlussklausur:

28.09.2015, 100% der Note

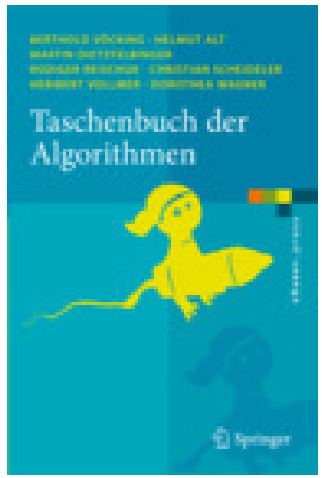
nächste Versuchsmöglichkeit: nach dem WS 15/16

Materialien

- Folien, Übungsblätter
- Diskussionsforum: Link siehe Homepage
- Buch:
  K. Mehlhorn, P. Sanders
  Algorithms and Data Structures — The Basic Toolbox
  vorl. Version der dt. Übersetzung online



- Taschenbuch der Algorithmen
  Springer 2008 (Unterhaltung / Motivation)



Weitere Bücher

- Algorithmen - Eine Einführung von Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, und Clifford Stein von Oldenbourg
- Algorithmen und Datenstrukturen von Thomas Ottmann und Peter Widmayer von Spektrum Akademischer Verlag
- Algorithmen kurz gefasst von Uwe Schöning von Spektrum Akad. Vlg., Hdg.

Algorithmus? Kann man das essen?

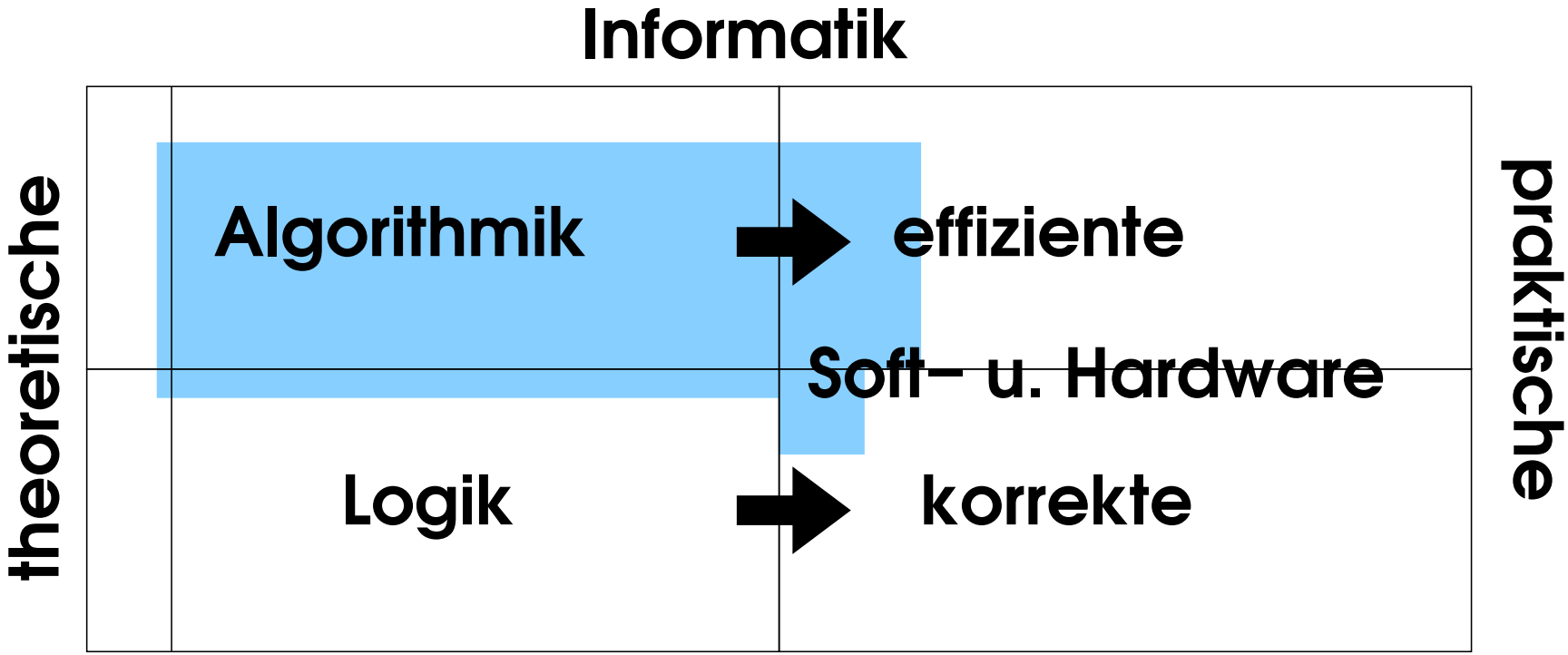
Pseudogriechische Verballhornung eines Namens,
der sich aus einer Landschaftsbezeichnung ableitet:
Al-Khwarizmi war persischer/usbekischer
Wissenschaftler (aus Khorasan) aber lebte in
Bagdad $\approx$ 780..840.
Machtzentrum des arabischen Kalifats
auf seinem Höhepunkt.

Er hat ein Rechenlehrbuch geschrieben.
$\rightsquigarrow$Algorithmus wurde zum Synonym für Rechenvorschrift.

Unter einem Algorithmus versteht man eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten.

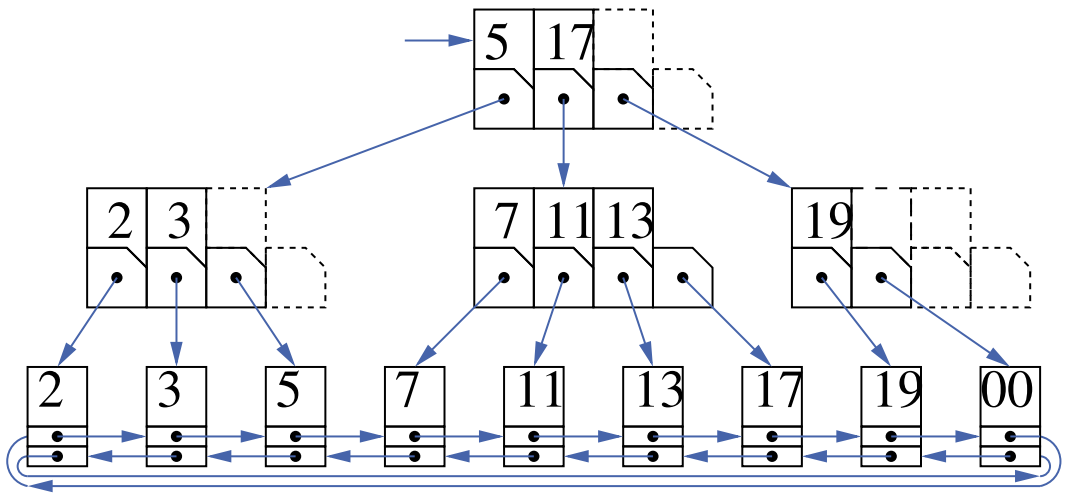
Algorithmik

Kerngebiet der (theoretischen) Informatik
mit direktem Anwendungsbezug



Datenstruktur

Ein Algorithmus bearbeitet Daten.
Wenn ein Teil dieser Daten eine (interessante) Struktur haben, nennen wir das Datenstruktur.
Immer wiederkehrende Datenstrukturen und dazugehörige Algorithmenteile
$\rightsquigarrow$ wichtige Grundwerkzeuge (Basic Toolbox)



Themenauswahl: Werkzeugkasten

Immer wieder benötigte

- Datenstrukturen
- Algorithmen
- Entwurfstechniken $\rightsquigarrow$ neue Algorithmen
- Analysetechniken $\rightsquigarrow$ Leistungsgarantien, objektiver Algorithmenvergleich

Jeder Informatiker braucht das $\rightsquigarrow$ Pflichtvorlesung

Inhaltsübersicht

Amuse Geule
Beispiel: Langzahl-Multiplikation

Schreibe Zahlen als Ziffernfolgen $a = (a_{n-1} \dots a_0)$, $a_i \in 0..B-1$.
Wir zählen

Volladditionen: $(c', s) := a_i + b_j + c$

Beispiel ($B = 10$): $9 + 9 + 1 = (1, 9)$

Ziffernmultiplikationen: $(p', p) := a_i \cdot b_j$

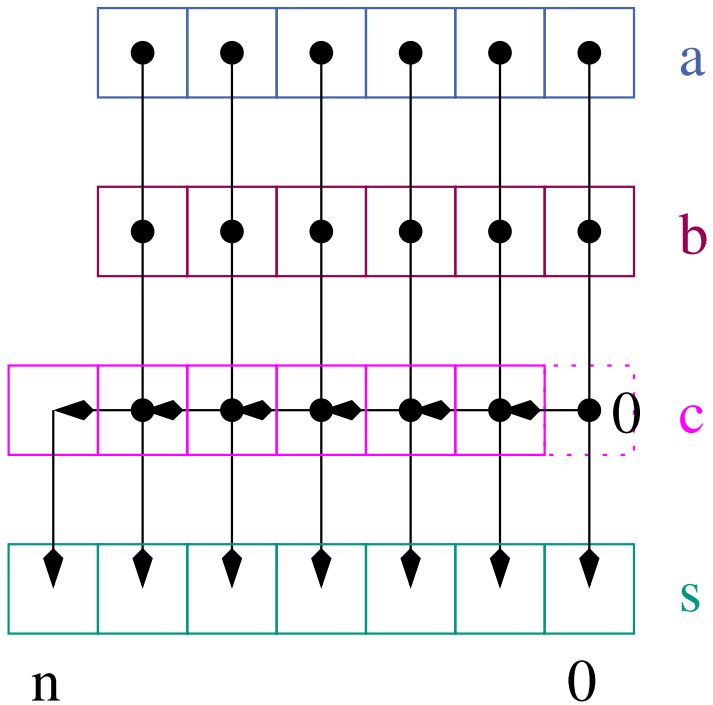
Beispiel ($B = 10$): $9 \cdot 9 = (8, 1)$

Addition

c=0 : Digit // carry / Überlauf

for $i := 0$ **to** $n-1$ **do** $(c, s_i) := a_i + b_i + c$

$s_n := c$



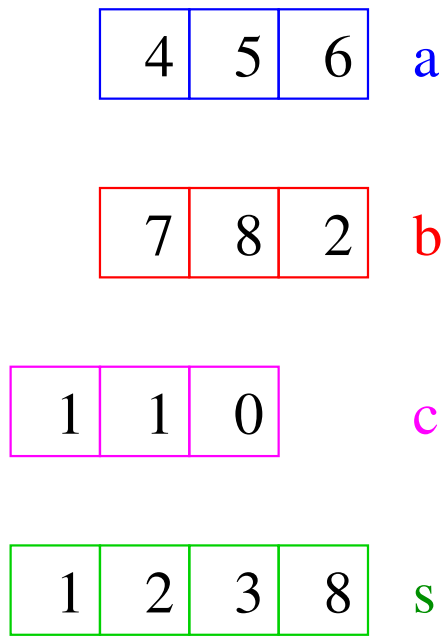
Satz: Addition von n-Ziffern-Zahlen braucht n Ziffern-Additionen.

Beispiel

$c{=}0$: Digit // carry / Überlauf

for $i := 0$ **to** $n-1$ **do** $(c, s_i) := a_i + b_i + c$

$s_n := c$



Exkurs: Pseudocode

- Kein C/C++/Java Menschenlesbarkeit vor Maschinenlesbarkeit
- Eher Pascal + Mathe − begin/end Einrückung trägt Bedeutung

Zuweisung: :=

Kommentar: //

Ausdrücke: volle Mathepower $\{i \geq 2 : \neg\exists a, b \geq 2 : i = ab\}$

Deklarationen: $c{=}0$: Digit

Tupel: $(c, s_i) := a_i + b_i + c$

Schleifen: **for** , **while** , **repeat** ...**until** ,...

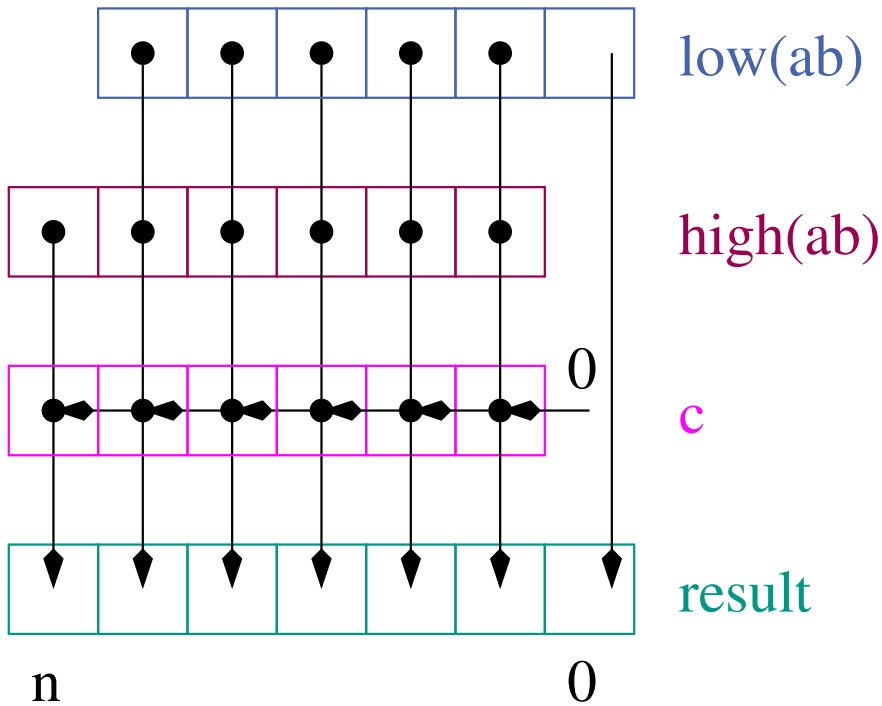
uvam: Buch Abschnitt 2.3, hier: just in time und on demand
if , Datentypen, Klassen, Speicherverwaltung

Exkurs vom Exkurs: Wieso nicht C++/Java-like?

- Klare Unterscheidung von Programmcode
- viele redundante `()[];`
- C `for` ist sehr low level
- == ist unschön während := für Zuweisung klarer ist
- C Logik/Bitoperatoren sind kryptischer als $\wedge$ etc.
- Wir verwendenen C++/Java-Notation wo dies sinnvoll ist
  $// \ ++ \ --, - \ += \ -=$
- Mathenotation ist oft mächtiger

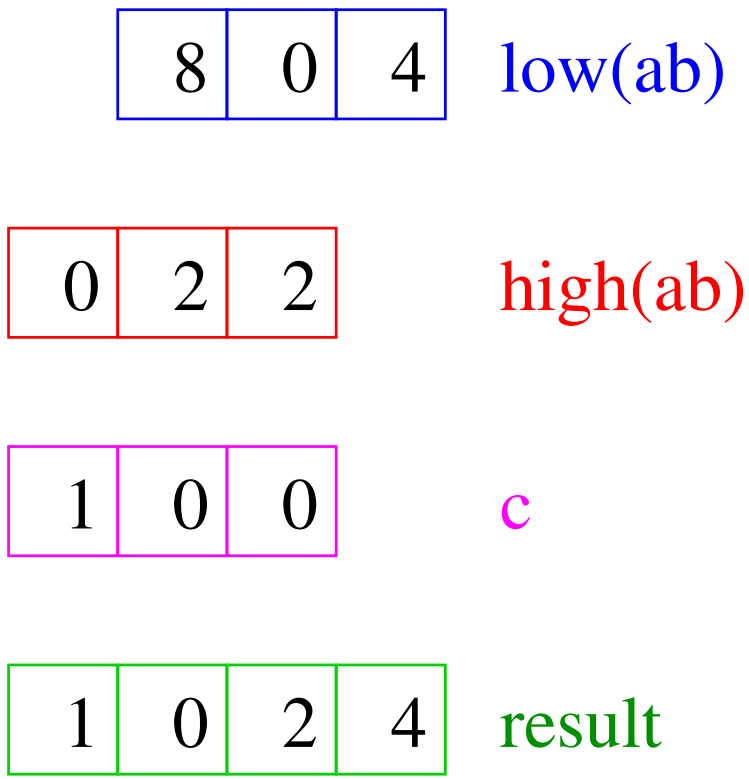
Ziffernmultiplikation

Function numberTimesDigit(a : **Array** $[0..n-1]$ **of** Digit, b : Digit)



Beispiel

numberTimesDigit(256,4)



Ziffernmultiplikation

Function numberTimesDigit(a : **Array** $[0..n-1]$ **of** Digit, b : Digit)
    result : **Array** $[0..n]$ **of** Digit
    c=0 : Digit // carry / Überlauf
    (h', ℓ):= $a[0] \cdot b$ // Ziffernmultiplikation
    result[0]:= ℓ
    for $i := 1$ **to** $n-1$ **do** // $n-1$ Iterationen
        (h, ℓ):= $a[i] \cdot b$ // Ziffernmultiplikation
        $(c, \text{result}[i])$:= $c + h' + \ell$ // Ziffernaddition
        h':= h
    result[n]:= $c + h'$ // Ziffernaddition, kein Überlauf?!
    return result

Analyse: $1 + (n-1) = n$ Multiplikationen, $(n-1) + 1 = n$ Additionen

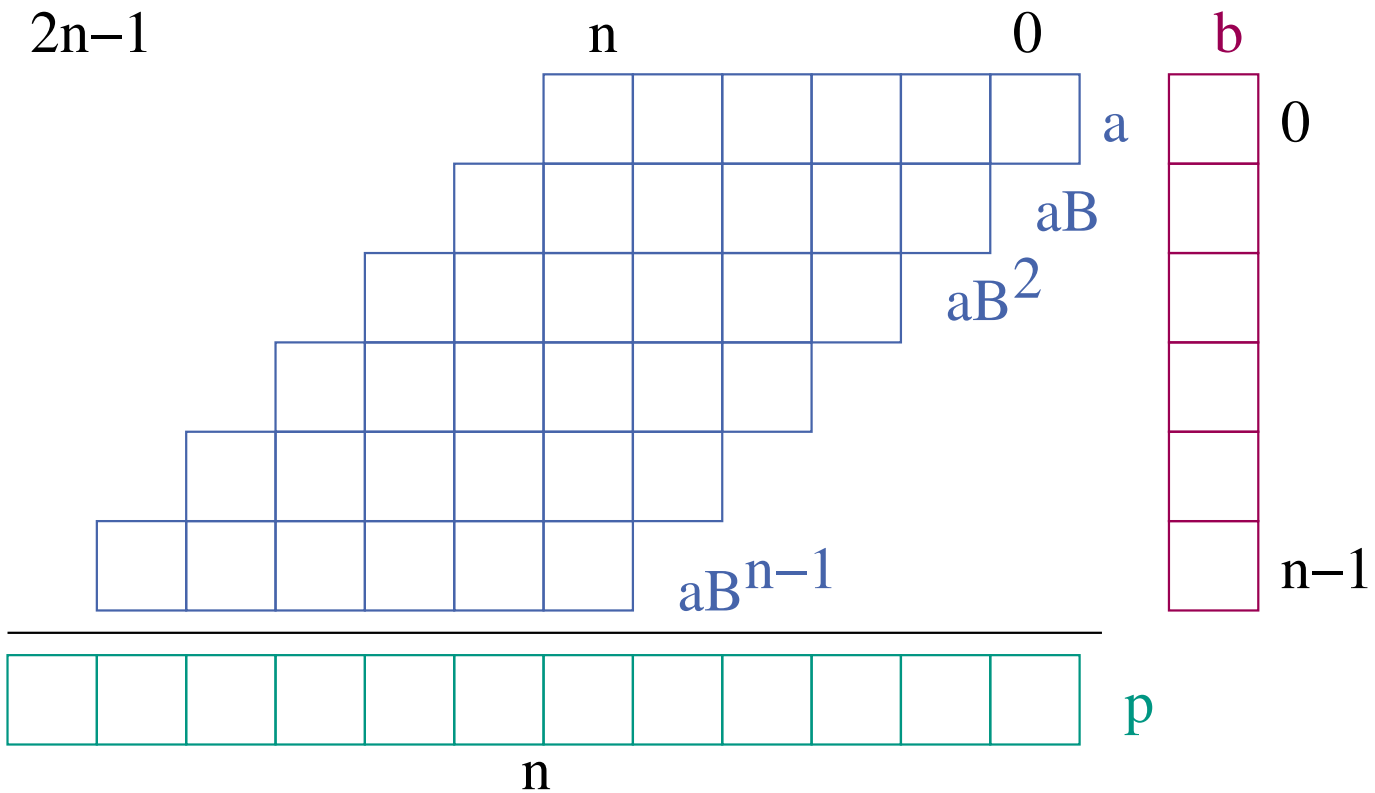
Schulmultiplikation

$p{=}0 \;\; : \mathbb{N}$ // Langzahl

for $j := 0$ **to** $n-1$ **do**

// Langzahladdition, Langzahl mal Ziffer, Schieben:

$p := p + a \cdot b[j] \cdot B^j$



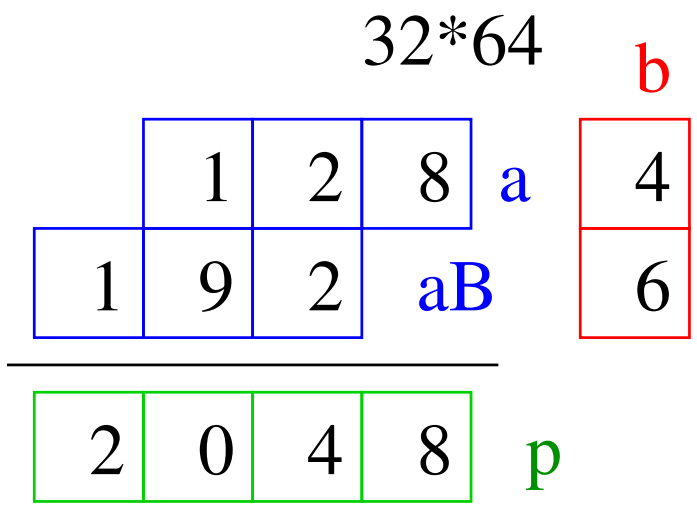
Schulmultiplikation Beispiel

p=0 $: \mathbb{N}$ // Langzahl

for $j := 0$ **to** $n-1$ **do**

// Langzahladdition, Langzahl mal Ziffer, Schieben:

$p := p + a \cdot b[j] \cdot B^j$



Schulmultiplikation Analyse

$p{=}0 \ : \mathbb{N}$
for $j := 0$ **to** $n-1$ **do**
  $p := p$ // $n+j$ Ziffern (außer bei $j=0$)
    $+$ // $n+1$ Ziffernadditionen (optimiert)
    $a \cdot b[j]$ // je n Additionen/Multiplikationen
    $\cdot B^j$ // schieben (keine Ziffernarithmetik)

Insgesamt:
n^2 Multiplikationen
$n^2 + (n-1)(n+1) = 2n^2 - 1$ Additionen
$3n^2 - 1 \leq 3n^2$ Ziffernoperationen

Exkurs O-Kalkül, die Erste

$$\mathrm{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

Idee: Konstante Faktoren (und Anfangsstück) ausblenden

- \+ Operationen zählen $\rightsquigarrow$ Laufzeit — welche Ops.?
- \+ Rechnungen vereinfachen
- \+ Interpretation vereinfachen
- ? Werfen wir zuviel Information weg — ?

Beispiel: Schulmultiplikation braucht Zeit $\mathrm{O}(n^2)$

Ergebnisüberprüfung

später an Beispielen

Ein rekursiver Algorithmus

Function recMult(a, b)

assert a und b haben n Ziffern, sei $k = \lceil n/2 \rceil$

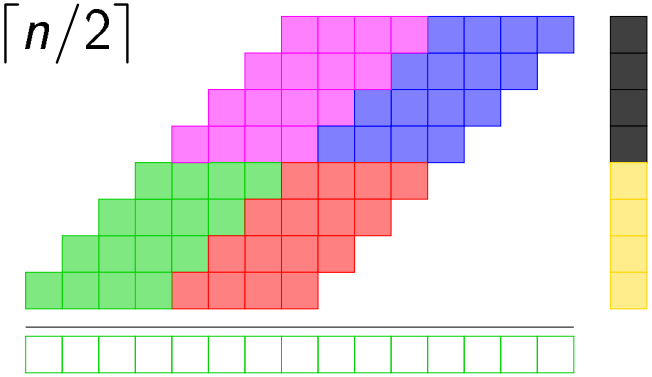
if $n = 1$ **then return** $a \cdot b$

Schreibe a als $a_1 \cdot B^k + a_0$

Schreibe b als $b_1 \cdot B^k + b_0$

return

$$\text{recMult}(a_1, b_1) \cdot B^{2k} +$$
$$(\text{recMult}(a_0, b_1) + \text{recMult}(a_1, b_0)) \cdot B^k +$$
$$\text{recMult}(a_0, b_0)$$



Beispiel

$$\boxed{10}\boxed{01}\cdot\boxed{19}\boxed{84}=$$
$$10\cdot 19\cdot 10000+$$
$$(10\cdot 84+1\cdot 19)\cdot 100+$$
$$1\cdot 84=$$
$$1985984$$

Analyse

Function recMult(a,b) // $T(n)$ Ops

  assert a und b haben n Ziffern, sei $k = \lceil n/2 \rceil$

  if $n = 1$ **then return** $a \cdot b$ // 1 Op

  Schreibe a als $a_1 \cdot B^k + a_0$ // 0 Ops

  Schreibe b als $b_1 \cdot B^k + b_0$ // 0 Ops

  return

    recMult$(a_1,b_1) \cdot B^{2k} +$ // $T(n/2) + 2n$ Ops

    (recMult(a_0,b_1)+recMult$(a_1,b_0))B^k +$ // $2T(n/2) + 2n$ Ops

    recMult(a_0,b_0) // $T(n/2) + 2n$ Ops

Also $T(n) \leq 4T(n/2) + 6n$

Übung: Wo kann man hier $\approx 2n$ Ops sparen?

Analyse

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 6 \cdot n & \text{if } n \geq 2. \end{cases}$$

$\longrightarrow$ (Master-Theorem, stay tuned)

$$T(n) \in \Theta\left(n^{\log_2 4}\right) \subseteq \mathrm{O}\left(n^2\right)$$

Aufgabe:

Zeigen Sie durch vollständige Induktion, dass
$T(n) \leq 7n^2 - 6n$
, falls n eine Zweierpotenz ist

Exkurs: Algorithmen-Entwurfsmuster

Im Buch: siehe auch Index!

Schleife: z. B. Addition

Unterprogramm: z. B. Ziffernmultiplikation, Addition

Teile und Herrsche: (lat. divide et impera, engl. divide and conquer)
Aufteilen in eins oder mehrere, kleinere Teilprobleme,
oft rekursiv

Es kommen noch mehr: greedy, dynamische Programmierung,
Metaheuristiken, Randomisierung,...

Karatsuba-Ofman Multiplikation[1962]

Beobachtung: $(a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_0 b_0 + a_1 b_0 + a_0 b_1$

Function recMult(a, b)

assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

if $n = 1$ **then return** $a \cdot b$

Schreibe a als $a_1 \cdot B^k + a_0$

Schreibe b als $b_1 \cdot B^k + b_0$

$c_{11} :=$ recMult(a_1, b_1)

$c_{00} :=$ recMult(a_0, b_0)

return

$$c_{11} \cdot B^{2k} +$$
$$(\text{recMult}((a_1 + a_0), (b_1 + b_0)) - c_{11} - c_{00})B^k$$
$$+ c_{00}$$

Beispiel

$$\boxed{10}\boxed{01}\cdot\boxed{19}\boxed{84}=$$
$$10\cdot 19\cdot 10000+$$
$$((10+1)\cdot(19+84)-10\cdot 19-1\cdot 84)\cdot 100+$$
$$1\cdot 84=$$
$$1985984$$

Analyse

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 3 \cdot T(\lceil n/2 \rceil) + 10 \cdot n & \text{if } n \geq 2. \end{cases}$$

$\longrightarrow$ (Master-Theorem)

$$T(n) = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.58}\right)$$