Grundbegriffe, Komplexität-Analyse, Rekursion

Themenverzeichnis

Algorithmus
- Beispiele
- Gleichwertige Algorithmen

Komplexität
- Komplexitätsklassen
- Landau-Notation

Rekursion und Wiederholung (Iteration)
- Beispiele

Bestimmung der Komplexität am Beispiel der String-basierten Arithmetik
- Addition
- Multiplikation nach Schulbuch
- Karatzuba - Multiplikation

Algorithmen-Analyse mit Maschinenmodell und Pseudocode
- RAM
- Prozeduren und Funktionen
- Korrektheit – Nachweise

Binäre und exponentielle Suche

Grundlagen der Algorithmenanalyse
- Iterative Konstrukte (Schleifen)
- Rekursive Konstrukte

Was wird als Algorithmus bezeichnet?

In etwa: eine Verarbeitungsvorschrift ➔ der Begriff kann nicht ganz genau definiert werden.

Intuitiver Algorithmus-Begriff

Ein Algorithmus ist eine präzise (in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-Schritte.)

Beispiele von bekannten Algorithmen

1. Addition zweier positiver Dezimalzahlen (mit Überträgen)

$$\begin{array}{rcc} & 3 & 3 \\ + & 4 & 8 \\ & {\scriptstyle 1} & \\ \hline & 8 & 1 \end{array}$$

2. Test, ob eine gegebene natürliche Zahl eine Primzahl ist
3. Sortieren einer unsortierten Kartei (etwa lexikographisch)
4. Berechnung der Eulerschen Zahl $e = 2.7182\ldots$

Im Kontext der Software-Entwicklung ist es ein "Kochrezept".

Beispiele:

- der Algorithmus von Euklid zur Berechnung des größten gemeinsamen Teilers („ggT“) zweier positiver ganzer Zahlen.
- der Algorithmus „Quicksort“ zum Sortieren von Reihungen
- der Knuth-Morris-Pratt-Algorithmus zum Suchen einer (kürzeren) Zeichenkette in einem (längeren) Text
- Algorithmus zum Traversieren eines Baumes

In der Spieltheorie hat Algorithmus eine spezielle Bedeutung: eine Regel, welche sicher zum Sieg führt, wenn befolgt.

Euklid-Algorithmus in Java

```
static int ggtIterativ(int ersteZahl, int zweiteZahl) { // 2
  // requires3 ersteZahl > 0 && zweiteZahl > 0; ensures return > 0
  while (ersteZahl != zweiteZahl)
    if (ersteZahl > zweiteZahl)
      ersteZahl -= zweiteZahl; // 4
    else
      zweiteZahl -= ersteZahl;
  return ersteZahl;
}
```

Die Korrektheit des Verfahrens ist nicht auf den ersten Blick ersichtlich.
Ein Leitfaden für den Nachweis: die Menge von gemeinsamen Teilern bleibt bei jedem Schritt unverändert.

Typisches Merkmal eines Algorithmus: das eigentliche Ziel wird im Verfahren nicht erwähnt.
Das ist häufig der Fall, es muss oft ein Schema "blind befolgt" werden (Beispiel: aus einem Labyrinth kann immer der Ausweg gefunden werden, indem die Wand mit einer Hand berührt wird).

Beschleunigte Variante:

```
static int ggt2(int ersteZahl, int zweiteZahl) { // Zusicherungen ähnlich
  while (ersteZahl != 0 && zweiteZahl != 0)
    if (ersteZahl > zweiteZahl)
      ersteZahl = ersteZahl % zweiteZahl;
    else
      zweiteZahl = zweiteZahl % ersteZahl;
  return ersteZahl + zweiteZahl;
}
```

Es werden damit viele Einzelschritte gespart, wenn man z.B. mit **97** und **2** anfängt.

Eine „gerade aus“ – Variante führt zum gleichen Ziel:

```
static int ggt1(int ersteZahl, int zweiteZahl) {
  // requires ersteZahl > 0 && zweiteZahl > 0; ensures return > 0
  for (int i = Math.max(ersteZahl, zweiteZahl); i>0; i--) // ²
    if ((ersteZahl % i == 0) && (zweiteZahl % i == 0))
      return i;
  return 0; // wenn requires nicht erfüllt
}
```

Aus den obigen Beispielen ist klar ersichtlich, dass eine bestimmte Aufgabe mit mehreren unterschiedlichen Ansätzen gelöst werden kann.
Man spricht in diesem Fall von *gleichwertigen* Algorithmen.

Typisches Beispiel: Sortieren (Bubble Sort, Quick Sort usw. ➔ das sind verschiedene Methoden, die ihre Vor- und Nachteile haben).

Komplexität eines Algorithmus

Damit bezeichnet man die Form der Abhängigkeit der Laufzeit vom Verarbeitungsvolumen (nicht etwa die Komplexität der Programmstruktur o.ä.).

Man kann die Leistungsfähigkeit von Algorithmen nicht etwa wie folgt bewerten:
"Algorithmus 1 erledigt 100 Bestellungen in 10 Minuten, Algorithmus 2 schafft es in 6 Minuten"
➔ solche Aussagen hängen von der Hardware, Umgebung, Sprache usw. ab.

Auf zwei verschiedenen Plattformen kann sich die Laufzeit von Implementierungen eines Algorithmus wie folgt verhalten:

Plattform 1 ➔ $3.4\ n^2 + 16.7\ n + 98.2$

Plattform 2 ➔ $2.6\ n^2 + 25.8\ n + 76.4$

wobei **n** das zu verarbeitende Volumen darstellt (z.B. die Zahl von Verbuchungen).

Man kann festhalten:

(i) Für ausreichend grosse **n** –Werte sind nur die „führenden" (am schnellsten wachsenden) Glieder **$3.4\ n^2$** und **$2.6\ n^2$** relevant, der Rest kann vernachlässigt werden.

(ii) Die unterschiedlichen Konstanten **3.4** und **2.6** ergeben sich aus den jeweiligen Umgebungen.

(iii) Für die beiden Plattformen gilt gemeinsam, dass eine Verdopplung von **n** ca. eine Vervierfachung der Laufzeit zur Folge hat.

Demzufolge ist sinnvoll zu sagen, dass die Laufzeit des Algorithmus „die gleiche Wachstumsordnung wie **n^2** hat" oder „sich asymptotisch wie **n^2** verhält" ➔ in der sog. *Landau-Notation* heisst es „der Algorithmus gehört zur Komplexitätsklasse **$O(n^2)$**" oder „hat die Komplexität **$O(n^2)$**". Als „asymptotisches Verhalten" bezeichnet man dabei das Verhalten für ausreichend grosse **n** – Werte (damit übereinstimmend ist die Landau-Notation auch als *asymptotische Notation* und die Ermittlung der Komplexitätsklasse als *asymptotische Analyse* bekannt).

Bei komplexen Ausdrücken für den Zeitaufwand wird die Zugehörigkeit zu einer O-Klasse immer durch die dominierende Komponente bestimmt.

Komplexitätsklassen

$O(1)$	konstanter Aufwand
$O(\log n)$	logarithmischer Aufwand
$O(n)$	linearer Aufwand
$O(n \cdot \log n)$	
$O(n^2)$	quadratischer Aufwand
$O(n^k)$ für ein $k \geq 0$	polynomialer Aufwand
$O(2^n)$	exponentieller Aufwand

Die Komplexitätsklasse gibt Auskunft über den Zeitaufwand des Algorithmus an sich, unabhängig von der spezifischen Implementierung auf bestimmter Plattform.

Nur für ausreichend hohe **n** – Werte ergibt eine bessere O-Klasse auch immer kürzere Laufzeiten ➔ für kleine **n** – Werte kann durchaus ein Algorithmus mit schlechterer O-Klasse schneller sein.

Wie oben erwähnt, spielen multiplikative und additive Konstanten für die Bestimmung der O-Klasse keine Rolle ➔ daher gibt es keine O-Klasse **$O(6n^2)$** oder **$O(n^2 + 100)$** , es wäre dasselbe wie **$O(n^2)$**.

Der Zeitaufwand ist oft für verschiedene Instanzen von Input-Daten sehr unterschiedlich. Dementsprechend können sich aus der Komplexität-Analyse verschiedene Resultate für den besten, den schlechtesten und den mittleren Fall ergeben. Die Art der jeweiligen Anwendung bestimmt, welche von diesen Varianten massgeblich ist.
Analog zur obigen Zeitaufwand – Komplexität gibt es auch Speicherkomplexität für den Speicher – Verbrauch (ist weniger oft kritisch).

Formal-mathematisch kann man die asymptotische Wachstumsordnung in allgemeinerer Weise mit Hilfe von folgenden Beziehungen zwischen zwei Funktionen **f(n)** und **g(n)** definieren:

$$
\begin{aligned}
\mathrm{O}(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}, \\
\Omega(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}, \\
\Theta(f(n)) &= \mathrm{O}(f(n)) \cap \Omega(f(n)), \\
\mathrm{o}(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}, \\
\omega(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}.
\end{aligned}
$$

Die Bedeutungen der Symbole **O**, **Ω**, **Θ**, **o** und **ω** können wie folgt formuliert werden:

- **O** : **g(n)** wächst langsamer oder gleich schnell wie **f(n)**
- **Ω** : **g(n)** wächst schneller oder gleich schnell wie **f(n)**
- **Θ** : **g(n)** wächst gleich schnell wie **f(n)**
- **o** : **g(n)** wächst strikt langsamer als **f(n)**
- **ω** : **g(n)** wächst strikt schneller als **f(n)**

Für praxisbezogene Ermittlungen der Komplexität ist die **O** – Beziehung völlig ausreichend, in formalen Ableitungen und Beweisen sind auch die übrigen Beziehungen oft nützlich.

Die Zugehörigkeit einer Funktion **h(n)** zur Klasse **O(f(n))** wird korrekterweise mit

h(n) ∈ O(f(n))

formuliert: „**h(n)** ist ein Element der Menge **O(f(n))**“.

Häufig wird dafür jedoch eine vereinfachte Form

h(n) = O(f(n))

benutzt.

Beispiel: $\mathbf{7n^2 + 3n + 5 = O(n^2)}$

Für die oben definierten Beziehungen gilt:

$$
\begin{aligned}
cf(n) &= \Theta(f(n)), \textit{ für jede positive Konstante } c, \\
f(n)+g(n) &= \Omega(f(n)), \\
f(n)+g(n) &= \mathrm{O}(f(n)), \textit{ wenn } g(n) = \mathrm{O}(f(n)), \\
\mathrm{O}(f(n)) \cdot \mathrm{O}(g(n)) &= \mathrm{O}(f(n) \cdot g(n)).
\end{aligned}
$$

Aus diesen Regeln folgt auch unmittelbar die früher erwähnte Irrelevanz von multiplikativen Konstanten und Termen niedrigerer Ordnung (z.B. additive Konstanten) für die Bestimmung einer O-Klasse.

Rekursion und Wiederholung (Iteration)

Das Thema ist vom OOP-Kurs bekannt

Klassisches Beispiel: Berechnung der Fakultät **n!**

n! = 1 * 2 * 3 * .... * (n – 2) * (n – 1) * n

n! = (n – 1)! * n

Die untenstehende iterative Berechnung der Fakultät **n!**

```
public int fakultaetIterativ(int n)
{
    int f = 1;
    for (int i = 2; i <= n; i++) {
        f = f * i;
    }
    return f;
}
```

kann mit Rekursion alternativ wie folgt implementiert werden:

```
public int fakultaetRekursiv(int n)
{
    if (n < 2) {
        return 1;
    }
    else {
        return fakultaetRekursiv(n - 1) * n;
    }
}
```

Im Prinzip sind meistens beide Methoden (Iteration und Rekursion) möglich. Die Rekursion ist oft eleganter, kann aber aufwändiger sein, weil Methodenaufrufe Zeit und Speicherplatz beanspruchen.
Im obigen Beispiel ist die Speicherkomplexität der iterativen Lösung **O(1)**.
Bei der rekursiven Lösung ist sie allerdings **O(n)**, weil der Call-Stack bei jedem rekursiven Aufruf vergrössert wird.

Das nächste Beispiel thematisiert diese Art von Schwierigkeiten mit rekursiven Lösungen.

Fibonacci - Zahlen

Es ist eine Folge von natürlichen Zahlen, welche wie folgt rekursiv definiert wird:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2$$

Daraus ergibt sich:

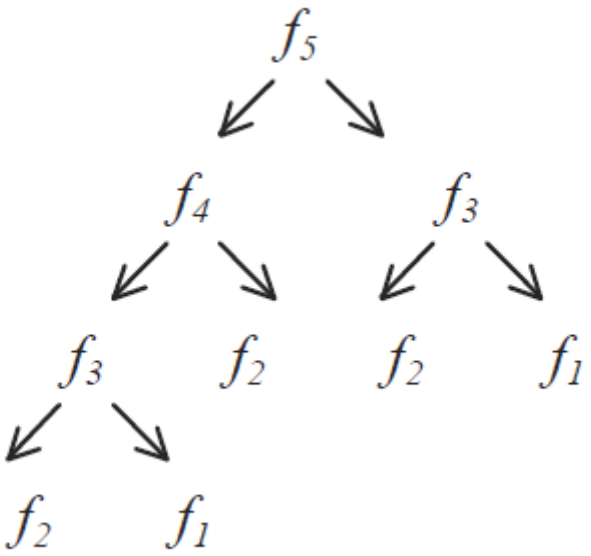
n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
f_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	…

Die obige Definition kann unmittelbar mit einem rekursiven Algorithmus umgesetzt werden:

```
static int fibonacciRekursiv(int n) {
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibonacciRekursiv(n-1) + fibonacciRekursiv(n-2);
}
```

Dieser Algorithmus hat allerdings die Komplexität **O(2^n)** (exponentiell) ➔ für grössere **n** –Werte ist er praktisch unbrauchbar.

Diese hohe Komplexität ist auf mehrfache (redundante) Berechnung von Vorgänger-Elementen zurückzuführen:



Abhilfe ➔ die einmal berechneten Elemente können für den späteren Gebrauch aufbewahrt werden:

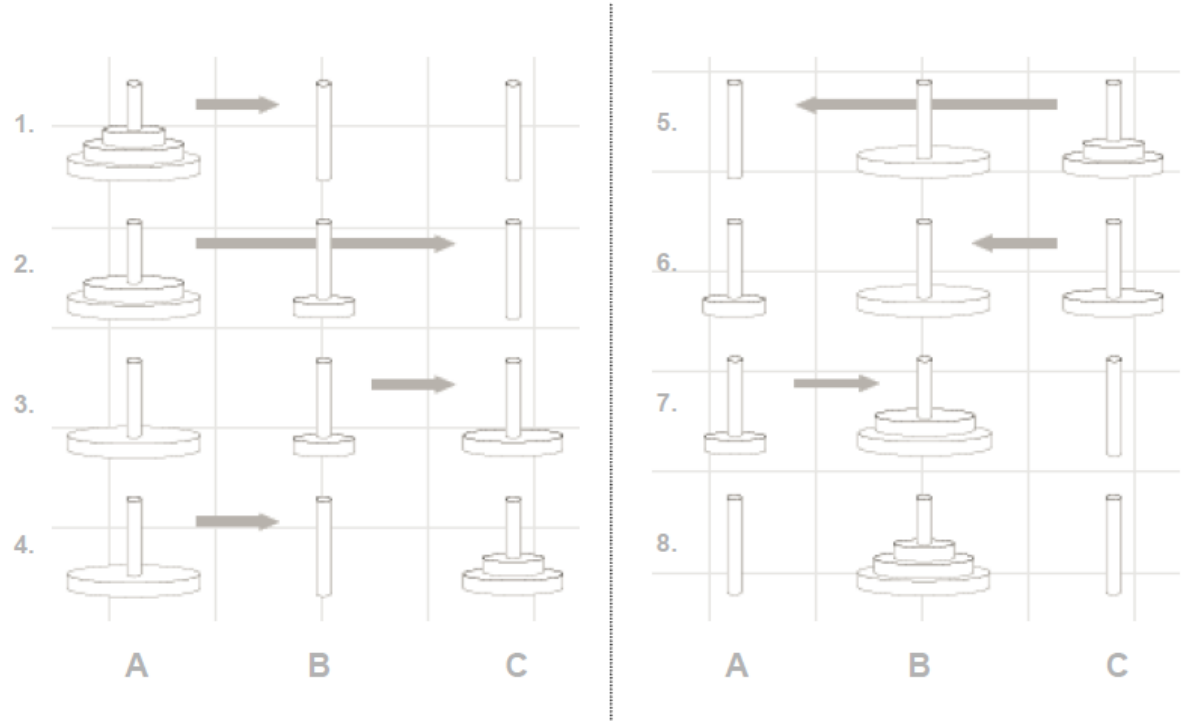
```
class FibonacciMitGedaechtnis {
  private long ged[]; // Gedächtnis
  public FibonacciMitGedaechtnis(int max) {
    ged = new long[max]; // 1
  }
  public long fibonacci(int n) {
     // requires 0 <= n < max == ged.length
    if (ged[n] != 0) // schon errechnet
      return ged[n];
    else if (n < 2) { // 0 oder 1
      ged[n] = n;
      return ged[n];
    }
    else
      ged[n] = fibonacci(n-1) + fibonacci(n-2);
    return ged[n];
  }
}
```

Ansonsten kann auch mit einer iterativen Lösung die **O(n)** – Komplexität erreicht werden:

```
static int fibonacciIterativ(int n) { // requires n >= 0
  if (n > 0) {
    int aktuelle = 1, temp = 1, vorherige = 0;
    for (int i = 1; i < n; i++) {
      temp = aktuelle;
      aktuelle = aktuelle + vorherige;
      vorherige = temp;
    }
    return aktuelle;
  }
  else
    return 0;
}
```

Türme von Hanoi

Ein anderes Beispiel, bei dem die Lösung ohne Rekursion – Einsatz viel schwieriger wäre.



Die oben abgebildeten drei Platten sollen in mehreren Zügen von A nach B befördert werden, wobei

(i) in einem Zug nur eine Platte verlegt werden darf

(ii) eine Platte darf nie auf einer kleineren liegen.

Leitidee der Lösung:

1. Die zwei oberen Platten von A nach C befördern mit Zwischenablage B
2. Die unterste Platte von A nach B verlegen
3. Die zwei oberen Platten von C nach B befördern mit Zwischenablage A

Der Schritt 2 ist trivial, die Schritte 1 und 3 sind identisch mit der ursprünglichen Aufgabe, wobei die Zahl der zu bewegenden Platten kleiner ist (2 anstatt 3).

Wenn die Verschiebung eines Turmes durch die Methode

Turm (anzahlPlatten, quelle, senke, zwischenablage)

dargestellt wird, können die Schritte 1 und 3 durch rekursive Aufrufe der gleichen Methode getätigt werden, wobei die letzteren Aufrufe ebenfalls weitere rekursive Aufrufe der gleichen Methode tätigen.

Daraus ergibt sich das folgende Schema:

```
Turm(3,A,B,C)
      Turm(2,A,C,B)
            Turm(1,A,B,C)
                  bewege A → B
            bewege A → C
            Turm(1,B,C,A)
                  bewege B → C
      bewege A → B
      Turm(2,C,B,A)
            Turm(1,C,A,B)
                  bewege C → A
            bewege C → B
            Turm(1,A,B,C)
                  bewege A → B
```

Bestimmung der Komplexität eines Algorithmus am Beispiel
der String-basierten Arithmetik („Langzahlarithmetik“)

Anmerkung: In der folgenden Analyse beschränken wir uns auf Additionen und Multiplikationen von zwei natürlichen Zahlen, die mit der gleichen Anzahl **n** von Ziffern in einem Zahlensystem mit der Basis **B** dargestellt werden. Es kann anschliessend gezeigt werden, dass die Resultate auch für den allgemeinen Fall von zwei unterschiedlich langen Operanden gelten.

Die Addition von zwei Zahlen:

$a_{n-1} \ldots a_1 \; a_0$	erster Operand
$b_{n-1} \ldots b_1 \; b_0$	zweiter Operand
$c_n \; c_{n-1} \ldots c_1 \; 0$	Überträge
$s_n \; s_{n-1} \ldots s_1 \; s_0$	Summe

Beispiel:

6 9 1 7	erster Operand
4 2 6 9	zweiter Operand
1 1 0 1 0	Überträge
1 1 1 8 6	Summe

Elementarschritt: Addition von drei Ziffern ==> ergibt zwei Ziffern, die höherwertigere dient als Übertrag

Es sind insgesamt **n** Elementarschritte (Elementaroperationen) notwendig und das Ergebnis ist eine Zahl mit max. **n + 1** Ziffern

Die Multiplikation von zwei Zahlen (nach Schulbuch):

$$a \cdot b = (a_{n-1} \ \ldots \ a_i \ \ldots \ a_0) \cdot (b_{n-1} \ \ldots \ b_j \ \ldots \ b_0)$$

Als Zwischenschritt untersuchen wir zuerst die Multiplikation des linken Operandes **a** mit einer Ziffer $\mathbf{b_j}$ der Zahl **b** :

$$(a_{n-1} \ldots a_i \ldots a_0) \cdot b_j \quad \longrightarrow \quad \begin{array}{r} c_{n-1}\ c_{n-2} \ \ldots\ c_i \quad c_{i-1} \ \ldots\ c_0\ 0 \\ d_{n-1} \ \ldots\ d_{i+1}\ d_i \quad \ldots\ d_1\ d_0 \\ \hline \text{Summe} \end{array}$$

wobei $a_i \cdot b_j = c_i \cdot B + d_i$ (z.B. $4 \cdot 8 = 3 \cdot 10 + 2$)

Für die Multiplikationen $\mathbf{a_i \cdot b_j}$ werden dabei **n** Elementaroperationen benötigt, für die anschliessende Addition

$$(c_{n-1} \ \ldots \ c_i \ \ldots \ c_0\ 0) + (d_{n-1} \ \ldots \ d_i \ \ldots \ d_0)$$

Ebenfalls **n** Elementaroperationen.

Daraus folgt:

Lemma 1.2. *Die Multiplikation einer n-ziffrigen und einer einziffrigen natürlichen Zahl kann mit 2n Elementaroperationen durchgeführt werden. Das Ergebnis ist eine Zahl mit $n+1$ Ziffern.*

Beispiel:

$$\begin{array}{rcl} 5\ 6\ 7\ 8 \cdot 4 & \Longrightarrow & \begin{array}{r} 2\ 2\ 2\ 3\ 0 \\ \quad 0\ 4\ 8\ 2 \\ \hline 2\ 2\ 7\ 1\ 2 \end{array} \end{array}$$

Wenn wir für das obige Zwischenresultat $\mathbf{a \cdot b_j}$ die Notation

$$a \cdot b_j = p_j = (p_{j,n}\ p_{j,n-1}\ p_{j,n-2}\ \ldots\ p_{j,2}\ p_{j,1}\ p_{j,0})$$

einführen, kann das Zielprodukt $\mathbf{a \cdot b}$ (**n** Ziffern mal **n** Ziffern) wie folgt als Summe von Teilprodukten dargestellt werden:

$$\begin{array}{rrrrrrr}
 & & p_{0,n} & p_{0,n-1} & \cdots & p_{0,2} & p_{0,1}\ p_{0,0} \\
 & p_{1,n} & p_{1,n-1} & p_{1,n-2} & \cdots & p_{1,1} & p_{1,0} \\
p_{2,n} & p_{2,n-1} & p_{2,n-2} & p_{2,n-3} & \cdots & p_{2,0} & \\
 & & \cdots & & & & \\
\hline
p_{n-1,n} & \cdots & p_{n-1,3} & p_{n-1,2} & p_{n-1,1} & p_{n-1,0} &
\end{array}$$

Summe der n Teilprodukte

Beispiel:

$$\begin{array}{r}
5\ 6\ 7\ 8\ \cdot\ 4\ 3\ 2\ 1 \\
\hline
5\ 6\ 7\ 8 \\
1\ 1\ 3\ 5\ 6 \\
1\ 7\ 0\ 3\ 4 \\
2\ 2\ 7\ 1\ 2 \\
\hline
2\ 4\ 5\ 3\ 4\ 6\ 3\ 8
\end{array}$$

Die Erzeugung von **n** Teilprodukten nimmt (nach Lemma 1.2) $\mathbf{n \cdot 2n = 2n^2}$ Elementaroperationen in Anspruch. Die Summe der ermittelten Teilprodukte wird sinnvollerweise mit einer Iterationsschleife berechnet, welche die Zeilen der Reihe nach zu einem „Akkumulator“ addiert:

```
akku := a · b_0
for j := 1 to n – 1 do
    akku := akku + a · b_j · B^j
```

(**B** ist die Basis des verwendeten Zahlensystems, im vorherigen Beispiel **B = 10** . Die Multiplikation mit $\mathbf{B^j}$ entspricht der Linksverschiebung von Teilprodukt-Zeilen, die mit absteigender Reihenfolge wächst.)

Aus den obigen Abbildungen geht klar hervor, dass von der Addition einer Zeile zum Akkumulator maximal **n + 1** Ziffern betroffen sind.
Die Bildung der Summe verbraucht somit maximal $\mathbf{(n-1)\cdot(n+1) = n^2-1}$ und die ganze Multiplikation $\mathbf{2n^2+n^2-1 = 3n^2-1}$ Elementaroperationen.
Demzufolge hat die Schulbuch-Multiplikation die Komplexitätsklasse $\mathbf{O(n^2)}$.

Für ausreichend grosse **n** – Werte hat somit die Verdoppelung von **n** eine Vervierfachung der Rechenzeit zur Folge, was man auch mit einer Messung überprüfen kann:

n	T_n [s]	$T_n/T_{n/2}$
8	0.00000469	
16	0.0000154	3.28527
32	0.0000567	3.67967
64	0.000222	3.91413
128	0.000860	3.87532
256	0.00347	4.03819
512	0.0138	3.98466
1024	0.0547	3.95623
2048	0.220	4.01923
4096	0.880	4
8192	3.53	4.01136
16384	14.2	4.01416
32768	56.7	4.00212
65536	227	4.00635
131072	910	4.00449

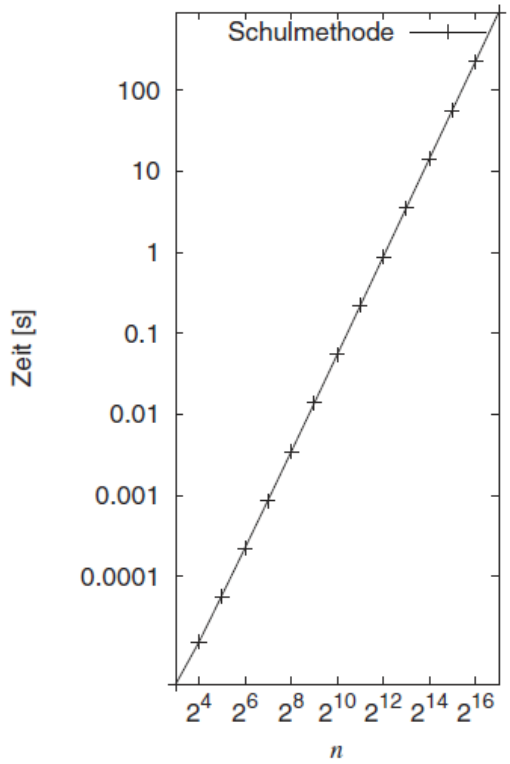


Abb. 1.1. Die Rechenzeit der Schulmethode für die Multiplikation von zwei Zahlen mit je n Ziffern. Die drei Spalten der Tabelle *links* geben n, die Rechenzeit T_n der C++-Implementierung aus Abschnitt 1.7 und das Verhältnis $T_n/T_{n/2}$ an. Die Graphik *rechts* zeigt $\log T_n$ als Funktion von $\log n$; dabei sehen wir im Wesentlichen eine Gerade. Man beachte, dass aus $T_n = \alpha n^\beta$ für konstante Faktoren α und β folgt, dass $T_n/T_{n/2} = 2^\beta$ und $\log T_n = \beta \log n + \log \alpha$ gilt, d. h., $\log T_n$ ist eine lineare Funktion von $\log n$ mit Steigung β. In unserem Fall beträgt die Steigung 2, wie man mit einem Lineal nachkontrollieren kann.

Karatzuba - Multiplikation

Jeder Operand wird zerlegt in einen höchstwertigen und einen niedrigstwertigen Teil:

$$a = a_1 \cdot B^k + a_0 \qquad b = b_1 \cdot B^k + b_0$$

wobei $k = \text{floor}(n / 2)$, damit beide Teile ca. die gleiche Länge haben.
Das Produkt $\mathbf{a \cdot b}$ lässt sich damit wie folgt umformen

$$\begin{aligned} a \cdot b &= a_1 \cdot b_1 \cdot B^{2k} + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot B^k + a_0 \cdot b_0 \\ &= a_1 \cdot b_1 \cdot B^{2k} + ((a_1 + a_0) \cdot (b_1 + b_0) - (a_1 \cdot b_1 + a_0 \cdot b_0)) \cdot B^k + a_0 \cdot b_0 \,. \end{aligned}$$

Die Produktbildung erfolgt danach in folgenden Schritten:

(a) Zerlege a und b in a_1, a_0, b_1 und b_0.
(b) Berechne die drei Produkte

$$p_2 = a_1 \cdot b_1, \quad p_0 = a_0 \cdot b_0, \quad p_1 = (a_1 + a_0) \cdot (b_1 + b_0).$$

(c) Addiere die Produkte aus (b), passend ausgerichtet, so dass sich das Produkt $a \cdot b$ ergibt, d. h., berechne $a \cdot b$ gemäß der Formel

$$a \cdot b = (p_2 \cdot B^{2k} + p_0) + (p_1 - (p_2 + p_0)) \cdot B^k.$$

(Die erste Summe wird durch Zusammensetzen gebildet, nicht durch eine wirkliche Addition.)

Beispiel:

a)

$$\begin{array}{cccccccc} & a & & & \cdot & & b & \\ a_1 & & a_0 & & & b_1 & & b_0 \\ 5 & 6 & 7 & 8 & \cdot & 4 & 3 & 2 & 1 \end{array}$$

b)

$p2 = a_1 \cdot b_1 = 56 \cdot 43 = 2408$

$p0 = a_0 \cdot b_0 = 78 \cdot 21 = 1638$

$p1 = (a_1 + a_0) \cdot (b_1 + b_0) = 134 \cdot 64 = 8576$

c)

$$\begin{array}{cccccccc} 5 & 6 & 7 & 8 & \cdot & 4 & 3 & 2 & 1 \\ & 2 & 4 & 0 & 8 & & & & \\ & & & & & 1 & 6 & 3 & 8 \\ & & & 8 & 5 & 7 & 6 & & \\ & & & -2 & 4 & 0 & 8 & & \\ & & & -1 & 6 & 3 & 8 & & \\ \hline & 2 & 4 & 5 & 3 & 4 & 6 & 3 & 8 \end{array}$$

Das Vorgehen benötigt 3 Multiplikationen von Faktoren der Länge ca. **n / 2** und 5 effektive Additionen. Die Halbierung der Länge der Multiplikationsfaktoren wirkt sich für grosse **n** – Werte stärker aus als die zusätzlichen Operationen. Für **n < 4** ist allerdings die Schulbuchmethode schneller.
Für steigende **n** – Werte ist das oben beschriebene Schema zunehmend im Vorteil, seine Komplexität bleibt jedoch **$O(n^2)$**.

Weitere wesentliche Verbesserung wird erzielt, wenn das Karatzuba-Verfahren rekursiv angewendet wird, d.h. wenn die obigen Teilzahlen **a_0, a_1, b_0, b_1** wieder auf gleiche Art zweigeteilt und miteinander multipliziert werden. Dabei wird beim Unterschreiten der Länge **4** zur Schulbuchmethode gewechselt.

Eine genaue Analyse zeigt, dass für die Rechenaufwände von zwei benachbarten Rekursion-Stufen eine sog. Rekurrenz-Beziehung gilt:

Lemma 1.6. *Sei $T_K(n)$ die maximale Anzahl von Elementaroperationen, die der Karatsuba-Algorithmus benötigt, wenn er auf natürliche Zahlen mit n Ziffern angewandt wird. Dann gilt:*

$$T_K(n) \leq \begin{cases} 3n^2 & \text{für } n \leq 3, \\ 3 \cdot T_K(\lceil n/2 \rceil + 1) + 8n & \text{für } n \geq 4. \end{cases}$$

Der Fall **n <= 3** folgt aus der Analyse der Schulbuch-Methode. Im anderen Fall ist vor allem der Faktor **3** von Bedeutung: dieser ergibt sich direkt aus der Tatsache, dass das Verfahren **3** Multiplikationen mit Teilzahlen durchführt. Der Term **8n** zeichnet für das Zusammenfügen von Zwischenresultaten.

Wenn man den Ausdruck **$3 \cdot T_K(\text{ceil}(n/2) + 1) + 8n$** näherungsweise mit **$3 \cdot T_K(n/2) + 8n$** ersetzt, kann mit einer einfachen Auswertung gezeigt werden, dass der lineare Term **8n** bei hohen **n** – Werten zum schneller wachsenden Gesamtwert des Ausdrucks praktisch nichts beiträgt und vernachlässigt werden kann:

n	$T_K(n)$	8n	$8n / T_K(n)$
2^{10}	322601	8192	0.025
2^{15}	81849067	262144	0.0032
2^{20}	19999948049	8388608	0.00042
2^{25}	4.86353 E12	268435456	0.000055
2^{30}	1.18195 E15	8589934592	0.0000073

Das führt zur vereinfachten Beziehung

$$T_K(n) = 3 \cdot T_K(n/2)$$

Wenn man für **$T_K(n)$** die polynomiale **n**-Abhängigkeit der Form **n^β** voraussetzt, folgt daraus

$$n^\beta = 3 \cdot n^\beta / 2^\beta \quad ==> \quad 2^\beta = 3 \quad ==> \quad \beta = \log 3$$

Somit wird mit der rekursiven Variante eine bessere Komplexität erreicht:

Satz 1.7 *Sei $T_K(n)$ die maximale Anzahl von Elementaroperationen, die der Karatsuba-Algorithmus für die Multiplikation zweier natürlicher Zahlen mit n Ziffern benötigt. Dann gilt: $T_K(n) \leq 153n^{\log 3}$ für alle n.*

Mit dem Näherungswert von **log 3 = 1.585** bedeutet das für grosse **n** - Werte im Vergleich zu **$O(n^2)$** eine wesentliche Verbesserung.

Eine zusätzliche Optimierung erreicht man durch die Variation der Schwelle **n_0**, bei der die Rekursion abgebrochen und die verbleibende Multiplikation nach Schulbuch durchgeführt wird.

Der optimale **n_0** – Wert kann für eine bestimmte Implementierung und Plattform durch Messungen bestimmt werden:

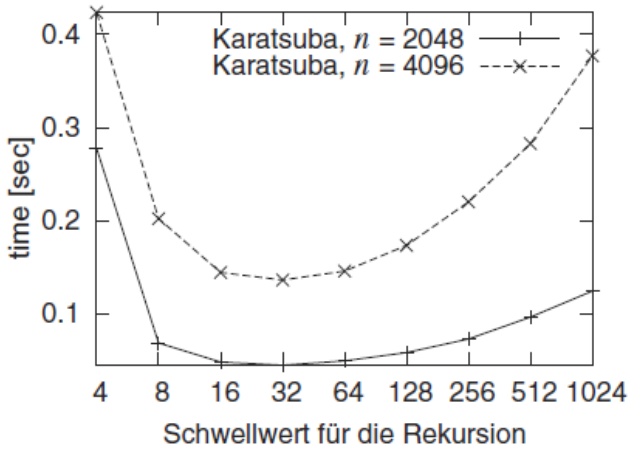


Abb. 1.4. Die Rechenzeit des Karatsuba-Algorithmus (in Sekunden) in Abhängigkeit von der Rekursionsschwelle n_0. Dargestellt sind die Zeiten für die Multiplikation von Zahlen mit 2048 Ziffern bzw. Zahlen mit 4096 Ziffern. Das Minimum ist bei $n_0 = 32$.

Algorithmen-Analyse mit Maschinenmodell und Pseudocode

RAM = Eine Variante des Von Neumann – Schema:

- Unbegrenzter Speicher, aber nur ein begrenzter Teil M ist zur bestimmten Zeit direkt adressierbar (z.B. M = 2^{32} Speicherzellen)
- Registersatz für Zwischenergebnisse, die Registerbreite typisch 64, 96, 128 .. Bits (ein Mehrfaches von log M = 32)
- Befehlssatz mit einfacher Arithmetik und Ablaufsteuerung (Sprünge)
- Bei der Analyse wird angenommen, dass jeder Befehl die gleiche Ausführungszeit beansprucht (damit ignoriert man z.B. die Cache-Zwischenspeicherung, Parallelverarbeitung mit Mehrkernprozessoren, Multi-Threading usw.)

RAM – Befehlssatz:

- $R_i := S[R_j]$ *lädt* den Inhalt der Speicherzelle, deren Index in Register R_j steht, in Register R_i.
- $S[R_j] := R_i$ *schreibt* den Inhalt von Register R_i in die Speicherzelle, deren Index in Register R_j steht.
- $R_i := R_j \odot R_h$ führt auf den Inhalten von Registern R_j und R_h eine binäre Operation $\odot$ aus und speichert das Ergebnis in Register R_i. Dabei gibt es für „$\odot$“ eine Reihe von Möglichkeiten. Die *arithmetischen* Operationen sind wie üblich $+$, $-$ und $*$; sie interpretieren die Registerinhalte als ganze Zahlen. Die Operationen **div** und **mod**, ebenfalls für ganze Zahlen, liefern den Quotienten bzw. den Rest bei der ganzzahligen Division. Die *Vergleichsoperationen* $\leq$, $<$, $>$ und $\geq$ für ganze Zahlen liefern als Ergebnis einen Wahrheitswert, also *true* ($= 1$) oder *false* ($= 0$). Daneben gibt es auch die bitweise auszuführenden Operationen $|$ (logisches Oder, OR), & (logisches Und, AND) und $\oplus$ (exklusives Oder, XOR); sie interpretieren Registerinhalte als Bitstrings. Die Operationen $\gg$ (Rechtsshift) und $\ll$ (Linksshift) interpretieren das erste Argument als Bitstring und das zweite als nichtnegativen Verschiebewert. Die *logischen* Operationen $\wedge$ und $\vee$ verarbeiten die *Wahrheitswerte* 1 and 0. Wir können auch annehmen, dass es Operationen gibt, die die in einem Register gespeicherten Bits als Gleitkommazahl interpretieren, d. h. als endlichen Näherungswert für eine reelle Zahl.
- $R_i := \odot R_j$ führt auf Register R_j eine *unäre* Operation $\odot$ aus und speichert das Ergebnis in Register R_i. Dabei sind die Operationen $-$ (für ganze Zahlen), $\neg$ (logische Negation für Wahrheitswerte, NOT) und $\sim$ (bitweise Negation für Bitstrings) vorgesehen.
- $R_i := C$ weist dem Register R_i einen *konstanten* Wert C zu.
- JZ k, R_i setzt die Berechnung in Programmzeile k fort, wenn Register R_i den Inhalt 0 hat (*Verzweigung* oder *bedingter Sprung*), andernfalls in der nächsten Programmzeile.[3]
- J k setzt die Berechnung in Programmzeile k fort (*unbedingter Sprung*).

Pseudocode

Eine weniger detaillierte Analyse des Zeitverhaltens kann mit Hilfe von Pseudocode durchgeführt werden, indem für jede Pseudocode-Anweisung die gleiche Ausführungsdauer angenommen wird.

Für die Pseudocode-Notation gibt es keinen allgemein gültigen Standard, die Schreibweise lehnt sich wahlweise an verschiedene höhere Sprachen an. Im Lehrmittel werden Elemente der Pascal-Syntax benutzt.

Beispiele:

```
x = 10 : N        // Deklaration einer Variablen x vom Typ „natürliche Zahl“,  ==>
                  //  ==> initialisiert mit 10

a : Array [1..5] of Z     // Deklaration eines Arrays a von ganzen Zahlen mit der Länge 5

c : Class age : N, income : N  end    // Deklaration einer Variablen c von einem ==>
                                      // ==> benutzerdefinierten Typ (Klasse)


//  Summe der Quadrate von natürlichen Zahlen  1 bis 20 :

i := 1
summe := 0

while  i <= 20 do                  // Iterationsschleife
    summe := summe + i * i
    i := i + 1
```

Prozeduren und Funktionen

Die iterative Fakultät-Berechnung als Beispiel:

```
Function  factorialIter(n : N) : N
    f = 1 : N
    while  n > 1 do
        f := f * n;
        n := n - 1;

    return f
```

Die Kopfzeile der obigen Deklaration besagt, dass die Funktion *factorialIter* einen Parameter des Typs *„natürliche Zahl“* entgegennimmt und einen Wert des gleichen Typs zurückgibt (Prozeduren werden mit *Procedure* anstelle von *Function* deklariert und geben keine Werte zurück).

Parameter können „*by value*“ oder „*by reference*“ übergeben werden. Im ersteren Fall wird der Parameterwert am Anfang (vor dem ersten Rumpf-Befehl) bitweise kopiert in den privaten Speicherbereich der Funktion/Prozedur, die anschliessend nur mit diesem Duplikat so arbeitet, als wäre das eine lokale Variable. Im letzteren Fall bekommt die Funktion/Prozedur die Referenz (= Adresse) des übergebenen Parameterwertes und benutzt nachher diese Referenz, um auf den Wert zuzugreifen – somit arbeitet sie effektiv mit der gleichen Variablen wie der Aufrufer unter einem anderen Variablennamen. (Aus der Sicht des Aufrufers folgt daraus: bei Übergabe einer Variablen „*by reference*“ kann sich ihr Wert nach dem Aufruf geändert haben, bei Übergabe „*by value*“ nicht.)
Im Pseudocode werden per Konvention „primitive“ Datentypen (Zahlen, Bool’sche Variablen) „*by value*“ übergeben, Objekte (inkl. Arrays) „*by reference*“ (die gleiche Konvention gilt in der C-Sprache).

Anmerkung: In Java werden Parameter nur „*by value*“ übergeben. Da aber Variablen, welche Objekte(Arrays) darstellen bereits (von Anfang an) Referenzen **sind**, wirkt sich die Übergabe „*by value*“ gleich aus, wie wenn sie „*by reference*“ erfolgen würde.

Beim RAM-Code - Aufruf einer nicht-rekursiven Funktion/Prozedur kann der Aufruf-Befehl am einfachsten durch den aufgerufenen Rumpf ersetzt werden ==> „Inline-Aufruf“.
Andernfalls (bei rekursiven Routinen) muss dafür *Rekursionsstack* benutzt werden.

Beispiel:

Function $factorial(n) : \mathbb{N}$
  if $n = 0$ **then return** 1 **else return** $n \cdot factorial(n-1)$

```
factorial :                                  // erster Befehl von factorial
R_arg := RS[R_r − 1]                         // lade n in Register R_arg
JZ thenCase, R_arg                           // springe zum then-Fall, falls n = 0
RS[R_r] := aRecCall                          // else-Fall; Rückkehradresse für rek. Aufruf
RS[R_r + 1] := R_arg − 1                     // Parameter ist n − 1
R_r := R_r + 2                               // erhöhe Stackzeiger
J factorial                                  // starte rekursiven Aufruf
aRecCall :                                   // Rückkehradresse für rek. Aufruf
R_result := RS[R_r − 1] ∗ R_result           // speichere n ∗ factorial(n − 1) ins Resultatregister
J return                                     // gehe zur Rücksprung-Vorbereitung
thenCase :                                   // Code für then-Fall
R_result := 1                                // speichere 1 ins Resultatregister
return :                                     // Code für Rücksprung
R_r := R_r − 2                               // gib Aktivierungssatz frei
J RS[R_r]                                    // springe zur Rückkehradresse
```

Abb. 2.2. Eine rekursive Funktion *factorial* zur Berechnung der Fakultätsfunktion und der entsprechende RAM-Code. Der RAM-Code gibt das Resultat $n!$ in Register R_{result} zurück.

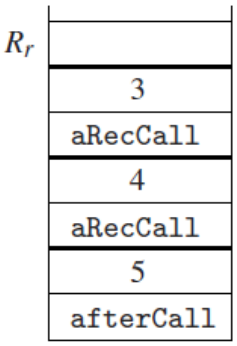


Abb. 2.3. Rekursionsstack eines Aufrufs $factorial(5)$, wenn die Rekursion den Aufruf $factorial(3)$ erreicht hat.

Bei jedem rekursiven Aufruf wird zum Rekursionsstack ein „Aktivierungssatz“ addiert, welcher die übergebene Parameterliste, lokale Variablen und eine Rücksprungadresse beinhaltet. Der Stack wird aufgebaut, bis die tiefste Rekursionsstufe (n = 0) erreicht ist und anschliessend stufenweise abgebaut. Es ist zu beachten, dass die effektive Berechnung des Ergebnisses in diesem Beispiel während der Abbauphase erfolgt: Zwischenresultate werden von Stufe zu Stufe im Register $\mathbf{R_{Result}}$ „nach oben gereicht“.

Korrektheit-Nachweise

Es gibt Methoden, um die grundsätzliche Korrektheit von Algorithmen in vielen Fällen mit einem hohen Grad der Verlässlichkeit zu prüfen (keine Methode kann natürlich eine lückenlose Korrektheit des Konzepts und der Code-Umsetzung garantieren, zumal auch die Prüfungen fehleranfällig sind).

Solche Prüfungen sind hauptsächlich bei komplexeren iterativen und rekursiven Abläufen nützlich, wo alle möglichen Szenarios nicht einfach überblickbar und erfassbar sind.

Zu solchen Prüfungszwecken werden dabei an kritischen Stellen des Vorganges *Zusicherungen* und *Invarianten* – Kontrollen eingeschoben.

Beispiel:

Function *power*$(a : \mathbb{R};\ n_0 : \mathbb{Z}) : \mathbb{R}$
  assert $n_0 \geq 0$ // negative Exponenten können nicht bearbeitet werden
  $p = a : \mathbb{R};\quad r = 1 : \mathbb{R};\quad n = n_0 : \mathbb{N}$ // es gilt $p^n r = a^{n_0}$
  while $n > 0$ **do**
    invariant $p^n r = a^{n_0}$
    if *n ist ungerade* **then** $n{-}{-};\ r := r \cdot p$ // zwischen Zuweisungen ist Invariante verletzt
    else $(n, p) := (n/2, p \cdot p)$ // parallele Zuweisung erhält Invariante aufrecht
  assert $r = a^{n_0}$ // Folgerung aus der Invarianten und $n = 0$
  return r

Abb. 2.4. Ein Algorithmus zur Berechnung nichtnegativer ganzzahliger Potenzen von reellen Zahlen.

In der obigen Berechnung wir das allgemeine Vorgehen bei Korrektheit-Nachweisen anschaulich dargestellt.
Vor der Iterationsschleife wird die *Vorbedingung* $\mathbf{n_0 >= 0}$ geprüft und in jedem Durchlauf wird sichergestellt, dass die Invariante $\mathbf{p^n r = a^{n_0}}$ erhalten bleibt.
Aus der Abbruchbedingung **n = 0** und der Invariante ergibt sich dann zwingend die Zusicherung der *Nachbedingung* $\mathbf{r = a^{n_0}}$.

Die Zusicherungen der Invariante und der Nachbedingung sind nur als „gedankliche Prüfungen“ im Algorithmus-Entwurf vermerkt, wobei die Vorbedingung sinnvollerweise auch in die Code-Umsetzung zu übernehmen ist.

Die geeignete Invariante (oder auch mehrere Invarianten) muss bei jedem Algorithmus individuell bestimmt („ertastet“) werden, es gibt keine allgemein gültige Vorgehensweise.

Eine spezielle Kategorie stellen Datenstruktur-Invarianten dar: sie gewährleisten die Konsistenz von Datenobjekten.

Zur Vollständigkeit eines Korrektheit-Nachweises muss auch sichergestellt werden, dass die Abbruch-Bedingung tatsächlich erreicht wird und der Algorithmus demzufolge *terminiert* (d.h. verfängt sich nicht z.B. in einer endlosen Schleife).

Übung

(i) Wie kann man die Terminierung der Funktion *power* im obigen Beispiel sicherstellen?

(ii) Welchen Vorteil bietet die Funktion *power* im Vergleich zu einer $\mathbf{n_0}$ – fachen Multiplikation?

(iii) Wie könnte man die Korrektheit des Euklid-Algorithmus zur ggT – Bestimmung nachweisen?

Ein *Zertifikat* ist im Kontext von Korrektheit-Nachweisen die zusätzliche Information, welche ein Algorithmus in bestimmten Fällen zwecks Erleichterung der Prüfung von Zusicherungen ermittelt.

Binäre und exponentielle Suche

In einem streng aufsteigend geordneten Array **a[1..n]** soll die Position eines Suchwertes **x** ermittelt werden.
(Streng aufsteigend geordnet bedeutet: **a[1] < a[2] < … < a[n]** .)
Die binäre Suche befolgt einen rekursiven Ansatz (“teile und herrsche”).
Zuerst wird ein Element **a[m]** in der Mitte des Arrays bestimmt und mit **x** verglichen. Im Fall der Übereinstimmung kann die Suche beendet werden.
Wenn andernfalls **x < a[m]** gilt, kann der Suchbereich auf die linke Hälfte des Arrays **[1..(m-1)]** reduziert werden und der Suchvorgang wird dort rekursiv fortgesetzt, analog bei **x > a[m]** erfolgt der nächste rekursive Schritt im Bereich **[(m+1)..n]**.
Wenn der Suchwert im Array nicht existiert, werden die benachbarten Indizes **i** und **i + 1** vom nächstkleineren und nächstgrösseren Element zurückgegeben (**a[i] < x < a[i+1]**). Damit diese Regel auch für **x < a[1]** und **x > a[n]** befolgt werden kann, erweitert man den Indexbereich um „fiktive“ Indexe **0** und **n + 1**, die im Algorithmus als Indexe zugelassen sind (mit Ausschluss von entsprechenden effektiven Element-Zugriffen).
Im folgenden Pseudocode-Schema sind **l** und **r** die Grenzen des aktuell bearbeiteten Indexbereiches.

```
(ℓ,r) := (0,n+1)
while true do
    invariant (I)                                   // d. h. Invariante (I) gilt hier
    if ℓ+1 = r then return ( "(a[ℓ] < x < a[ℓ+1]" )
    m := ⌊(r+ℓ)/2⌋                                  // ℓ < m < r
    s := compare(x,a[m])          // −1 falls x < a[m], 0 falls x = a[m], +1 falls x > a[m]
    if s = 0 then return ( "x steht in a[m]" )
    if s < 0
        then r := m                                 // a[ℓ] < x < a[m] = a[r]
        else ℓ := m                                 // a[ℓ] = a[m] < x < a[r]
```

Abb. 2.5. Binäre Suche nach x in einem geordneten Array $a[1..n]$. (Eingerahmte Zahlen werden als Werte in die Textausgabe eingebaut.)

Die festgelegte Invariante (I) sichert zu, dass die Bereichsgrenzen **l** und **r** unterschiedlich sind (d.h. der Bereich „degeneriert“ nicht zu einem Index) und die Grenzelemente **a[l]** und **a[r]** schliessen den Suchwert **x** ein:

$$0 \leq \ell < r \leq n+1 \quad \text{und} \quad a[\ell] < x < a[r] \qquad \text{(I)}$$

Die maximale Anzahl von notwendigen rekursiven Durchläufen (d.h. im schlechtesten Fall, wenn die Suche erfolgslos ist) kann unmittelbar bestimmt werden, wenn **n** eine Zweierpotenz ist: $\mathbf{n = 2^k}$. In diesem Fall wird der Suchbereich in jedem Durchlauf genau halbiert, d.h. der Exponent **k** um **1** verkleinert. Somit sind insgesamt **k = log n** Durchläufe notwendig, woraus sich die Komplexität **O(log n)** ergibt.
Wenn **n** keine Zweierpotenz ist, werden höchstens so viele Durchläufe beansprucht wie bei der nächstgrösseren Zweierpotenz, d.h. im Einklang mit der gleichen Komplexität **O(log n).**

Anwendungen des obigen Verfahrens sind nicht auf Arrays beschränkt, es kann in gleicher Weise für Stützpunkte einer monotonen Funktion **f** eingesetzt werden ==> dabei werden die Array-Elemente **a[i]** mit den Funktionswerten **f(i)** ersetzt.
Eine Erweiterung der vorherigen Suchaufgabe ergibt sich, wenn die obere Bereichsgrenze nicht festgelegt ist, wodurch die Suche auf beliebig hohe **i**-Werte ausgedehnt werden kann (damit nimmt man freilich in Kauf, dass eine erfolgslose Suche nicht terminiert).
In diesem Fall muss in einer Vorphase zuerst die fehlende obere Grenze bestimmt werden, indem man eine exponentiell wachsende Indexfolge (z.b. $\mathbf{2^k}$ oder $\mathbf{10^k}$) so lange prüft, bis der erste geeignete Kandidat gefunden wurde. Damit kann anschliessend die binäre Suche erfolgen so wie vorher beschrieben. Dieses kombinierte Verfahren ist auch als *exponentielle Suche* bekannt.

Grundlagen der Algorithmenanalyse

Fragestellungen, die im Vordergrund stehen:

- Berechenbarkeit
- Terminierung
- Komplexität

In Algorithmen mit komplexen iterativen und rekursiven Strukturen sind diese Eigenschaften oft nicht unmittelbar erkennbar und ihre Ermittlung benötigt eine ausführliche formal-mathematische Analyse (sehr oft ist auch keine Lösung bekannt).

Iterative Konstrukte (Schleifen)

Als Beispiel kann die Berechnung des folgenden Ausdrucks dienen:

$$\sum_{k=1}^{n} \sum_{i=k}^{n} k * i$$

```
k := 1
a := 0
while  k <=  n  do
    i := k
    while  i <=  n  do
        a := a + k * i
        i := i + 1
    k := k + 1
return a
```

Die Berechnung besteht aus zwei ineinander geschachtelten Schleifen.
Die aufwändigste Operation ist die Multiplikation **k * i** , somit nehmen wir die Anzahl dieser Multiplikationen als den massgeblichen Indikator für den Rechenaufwand an.
Die innere Schleife benötigt im kürzesten Fall (**k = n**) einen Durchlauf, im längsten Fall (**k = 1**) sind **n** Durchläufe notwendig. Somit kann „im Durchschnitt“ der Aufwand der inneren Schleife als ca. **n / 2** bewertet werden, wobei dieser Aufwand in der äusseren Schleife **n** mal getätigt wird. Das ergibt einen geschätzten Gesamtaufwand von **n * n / 2 = n^2 / 2** , woraus sich die Komplexität **O(n^2)** ergibt.

Rekursive Konstrukte

Wir betrachten hier das typische Rekursion-Schema, in dem aus einer Rekursionsebene die Unterstufe **d** – mal (in **d** Aufruf-Instanzen) getätigt wird, wobei die Verarbeitungsgrösse der Oberstufe **n** beträgt und die Unterstufe einen Bruchteil **n / b** dieser Grösse verarbeitet.

Für eine einfache lineare Rekurrenz-Beziehung zwischen dem Aufwand **r(n)** der Oberstufe und dem Aufwand **r(n/b)** der Unterstufe kann gezeigt werden:

Satz 2.5 (Mastertheorem (einfache Form)) *Für positive Konstanten a, b, c und d, und $n = b^k$ für eine natürliche Zahl k, sei die folgende Rekurrenz gegeben:*

$$r(n) = \begin{cases} a & \text{für } n = 1\,, \\ d \cdot r(n/b) + cn & \text{für } n > 1\,. \end{cases}$$

Dann gilt:

$$r(n) = \begin{cases} \Theta(n) & \text{für } d < b\,, \\ \Theta(n \log n) & \text{für } d = b\,, \\ \Theta\left(n^{\log_b d}\right) & \text{für } d > b\,. \end{cases}$$

Der Gesamtaufwand der Oberstufe beinhaltet neben dem Unterstufen-Aufwand von **d** Aufruf-Instanzen noch den Term **cn**, welcher für den Zusammenbau der Aufruf-Resultate zeichnet.

Die Situation kann wie folgt dargestellt werden:

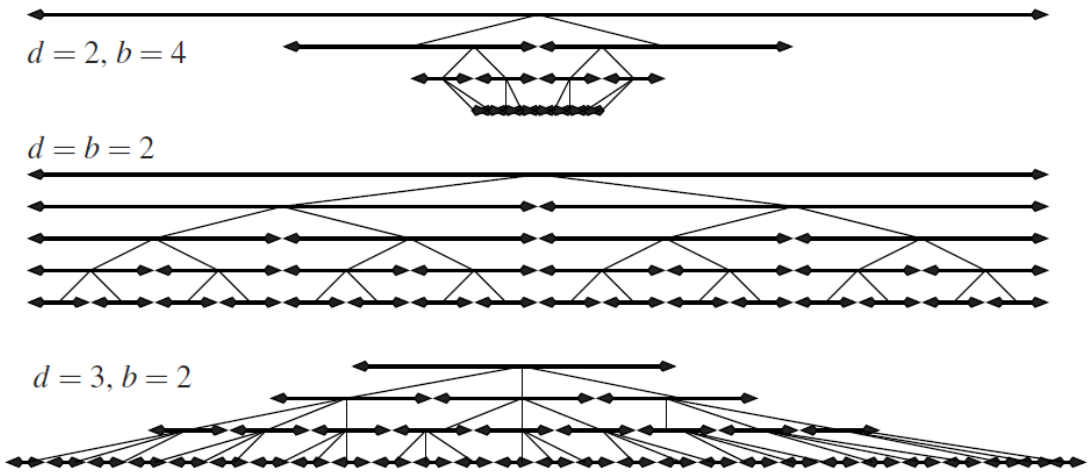


Abb. 2.6. Beispiele für die drei Fälle im Mastertheorem. Instanzen werden durch horizontale Strecken mit Doppelpfeilen angedeutet. Die Länge einer solchen Strecke steht für die Größe der Instanz, und die Teilinstanzen, die aus einer Instanz entstehen, sind in der darunterliegenden Ebene dargestellt. Der obere Teil der Abbildung stellt den Fall $d = 2$ und $b = 4$ dar, bei dem aus einer Instanz zwei Teilinstanzen mit einem Viertel der Größe erzeugt werden – die Gesamtgröße der Teilinstanzen ist nur die Hälfte der Größe der Instanz. Der mittlere Teil der Abbildung stellt den Fall $d = b = 2$ dar, und der untere Teil den Fall $d = 3$ und $b = 2$.

Für eine allgemeinere Form der Rekurrenz-Beziehung (siehe 2.1 unten) kann eine ähnliche Aussage (Satz 2.5) bewiesen werden.

$$r(n) \leq \begin{cases} a & \text{, wenn } n \leq n_0, \\ cn^s + d \cdot r(\lceil n/b \rceil + e_n) & \text{, wenn } n > n_0. \end{cases} \tag{2.1}$$

Dabei sind $a > 0$, $b > 1$, $c > 0$, $d > 0$ und $s \geq 0$ konstante reelle Zahlen, und die e_n, $n > n_0$, sind ganze Zahlen mit $-\lceil n/b \rceil < e_n \leq e$, für eine ganze Zahl $e \geq 0$.

Satz 2.5 (Mastertheorem (allgemeine Form)) *Wenn $r(n)$ die Rekurrenzungleichung (2.1) erfüllt, dann gilt:*

$$r(n) = \begin{cases} \mathrm{O}(n^s) & \text{für } d < b^s, \text{ d. h. } \log_b d < s\,, \\ \mathrm{O}(n^s \log n) & \text{für } d = b^s, \text{ d. h. } \log_b d = s\,, \\ \mathrm{O}\left(n^{\log_b d}\right) & \text{für } d > b^s, \text{ d. h. } \log_b d > s\,. \end{cases}$$