

1.4 Entwurfsmethoden

Für den guten Entwurf kann man kaum strenge Regeln angeben, aber es gibt einige Prinzipien, die man je nach Zielsetzung einsetzen kann und die unterschiedliche Vorzüge haben. Wir werden folgende typische Methoden vorstellen:

1. Divide-and-Conquer
2. Dynamische Programmierung

1.4.1 Divide-and-Conquer-Strategie

Die Divide-and-Conquer-Strategie wird häufig benutzt, weil sie vielseitig ist und auf fast jedes Problem angewendet werden kann, das sich in kleinere Teilprobleme zerlegen läßt. Die Strategie besagt:

- **Divide:** Zerlege das Problem in Teilprobleme, solange bis es „elementar“ bearbeitet werden kann.
- **Conquer:** Setze die Teillösungen zur Gesamtlösung zusammen (merge).

Nach dieser Strategie entwickelte Algorithmen legen oft eine rekursive Implementierung nahe (gutes Beispiel: Baum-Durchlauf; schlechtes Beispiel: Fibonacci-Zahlen). Bei der Laufzeitanalyse solcher Algorithmen gelangt man meist zu Rekursionsgleichungen. Diese werden in Abschnitt 1.5 näher behandelt.

Beispiel: Gleichzeitige Bestimmung von MAX und MIN einer Folge

```

0  procedure MAXMIN (S);      [Aho et al. 74, Seite 61]
1  if ||S||=2 then
2    let S={a,b};
3    return (MAX (a, b), MIN (a, b));
4  else
5    divide S into two subsets  $S_1$  and  $S_2$ , each with half the elements;
6    (max1,min1) ← MAXMIN( $S_1$ );
7    (max2,min2) ← MAXMIN( $S_2$ );
8    return (MAX (max1, max2), MIN (min1, min2));
9  end;
10 end MAXMIN;
```

Sei $T(n)$ die Zahl der Vergleiche, dann gilt

$$T(n) = \begin{cases} 1 & n = 2 \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

Erklärung: $n = 2$: ein Vergleich (Zeile 3)
 $n > 2$: zwei Aufrufe von MAXMIN mit $\frac{n}{2}$ -elementigen Mengen (Zeilen 6 und 7) und zwei zusätzliche Vergleiche (Zeile 8)

Behauptung: $T(n) = \frac{3}{2}n - 2$

Verifikation:

$$\begin{aligned} T(2) &= \frac{3}{2} \cdot 2 - 2 = 1 \quad \checkmark \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 &= 2 \cdot \left(\frac{3}{2} \cdot \frac{n}{2} - 2\right) + 2 \\ &= \frac{3}{2}n - 2 = T(n) \quad \checkmark \end{aligned}$$

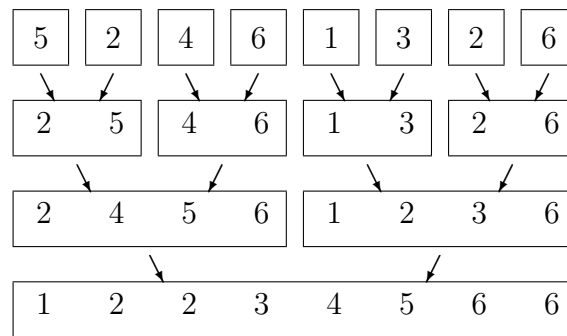
Zum Vergleich: Ein Algorithmus, der Minimum und Maximum separat bestimmt, benötigt $T(n) = 2n - 2$ Vergleiche.

Beispiel: MergeSort

MergeSort ist ein einfaches Beispiel für ein nach der Divide-and-Conquer-Strategie entworfenes Sortierverfahren. Sei $F = a_1, \dots, a_n$ eine Folge von Zahlen. Dann funktioniert MergeSort so:

1. Teile F in zwei etwa gleichgroße Folgen $F_1 = a_1, \dots, a_{\lceil n/2 \rceil}$ und $F_2 = a_{\lceil n/2 \rceil + 1}, \dots, a_n$. (divide)
2. Sortiere F_1 und F_2 mittels MergeSort, falls $|F_1| > 1$ bzw. $|F_2| > 1$. (conquer)
3. Verschmelze F_1 und F_2 zu einer sortierten Folge. (merge)

Wir veranschaulichen das MergeSort-Verfahren (genauer „bottom-up“- oder auch „2-way-straight“-MergeSort) an einem Beispiel:



Sei $T(n)$ die Zahl der Operationen (im wesentlichen Vergleiche), dann gilt:

$$T(n) = \begin{cases} c_1 & , \text{ für } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n & , \text{ für } n > 1 \end{cases}$$

für geeignete Werte $c_1, c_2 > 0$. Dabei charakterisiert c_1 den Aufwand für die Lösung des trivialen Problems ($n = 1$), und $n \cdot c_2$ den Aufwand für das Verschmelzen zweier Listen.

Behauptung: $T(n) \leq (c_1 + c_2) \cdot n \lg n + c_1$

Beweis: Verifizieren der Rekursion mittels vollständiger Induktion von $\frac{n}{2}$ nach n .

- Induktionsanfang: $n = 1$ offensichtlich korrekt
- Induktionsschritt von $\frac{n}{2}$ nach n :

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n \\ &\leq 2 \left[(c_1 + c_2) \frac{n}{2} \lg \frac{n}{2} + c_1 \right] + c_2 n \\ &= (c_1 + c_2) n (\lg n - 1) + 2c_1 + c_2 n \\ &= (c_1 + c_2) n \lg n + 2c_1 - c_1 n \\ &= (c_1 + c_2) n \lg n + c_1 - c_1 (n - 1) \\ &\leq (c_1 + c_2) n \lg n + c_1 \end{aligned}$$

Exakte Analyse von MergeSort

Sei $T(n)$ die Zahl der Vergleiche. Die Aufteilung der Folge geschieht in zwei Teilfolgen mit $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Elementen.

Zur Verschmelzung werden $\lceil n/2 \rceil + \lfloor n/2 \rfloor - 1 = n - 1$ Vergleiche benötigt.

Daher ergibt sich folgende Rekursionsgleichung:

$$T(n) = \begin{cases} 0 & \text{für } n = 1 \\ n - 1 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) & \text{für } n > 1 \end{cases}$$

Für diese gilt [Mehlhorn, Seite 57]:

$$T(n) = n \cdot \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$$

1.4.2 Dynamische Programmierung

Die Bezeichnung „dynamische Programmierung“ geht auf Richard Bellman (1957) zurück:

- dynamisch = sequentiell
- Programmierung = Optimierung mit Nebenbedingungen (vgl. Lineare Programmierung)

Die dynamische Programmierung kann auf Optimierungsprobleme angewandt werden, deren Lösungen sich aus den optimalen Lösungen von Teilproblemen zusammensetzen (*Bellmansches Optimalitätsprinzip*). Das Grundprinzip ist, zunächst Lösungen für „kleine“ Elementarprobleme zu finden und daraus sukzessive Lösungen für immer „größere“ Probleme zu konstruieren, bis schließlich das eigentliche Gesamtproblem gelöst ist. Diese Vorgehensweise von kleinen zu immer größeren Teillösungen charakterisiert die dynamische Programmierung als „*bottom-up*“-Strategie und unterscheidet sie vom „*top-down*“-Ansatz der Divide-and-Conquer-Methode. (Anmerkung: Diese Unterscheidung ist nicht ganz zwingend; z.B. wird diese Fallunterscheidung schwierig bei der Methode der *dynamischen Programmierung mit Memoization*.)

Charakteristisch für die dynamische Programmierung ist es, eine Hilfsgröße zu definieren, die die Lösungen der Teilprobleme beschreibt und für diese Größe eine Rekursionsgleichung (recurrence relation, DP equation) zu formulieren. Die Rekursionsgleichung beschreibt, wie die Teillösungen zusammengesetzt sind. Konkret umfaßt ein auf dynamischer Programmierung basierender Algorithmus folgende Schritte:

- Teilprobleme bearbeiten
- **Teilergebnisse in Tabellen eintragen**
- Zusammensetzen der Gesamtlösung

Beispiel: Matrix-Kettenprodukte (matrix chain problem)

Gegeben seien zwei (reelwertige) Matrizen $B \in \mathbb{R}^{l \times m}$ und $C \in \mathbb{R}^{m \times n}$. Für die Elemente der Produktmatrix $A = B \cdot C$ gilt:

$$a_{i,k} = \sum_{j=1}^m b_{i,j} c_{j,k}$$

Zur Berechnung des Produkts von B und C sind offenbar $l \cdot m \cdot n$ (skalare) Multiplikationen und Additionen erforderlich.

Bei mehreren Matrizen hängt der Wert des (Ketten-)Produkts bekanntlich nicht von der Klammerung, also davon in welcher Reihenfolge die Matrixmultiplikationen ausgewertet werden, ab (Assoziativgesetz). Der Rechenaufwand ist aber sehr wohl von der Klammerung abhängig, wie das folgende Beispiel belegt:

Konkretes Beispiel: Seien $M_i, i = 1, \dots, 4$ reelle Matrizen, der Dimensionen $10 \times 20, 20 \times 50, 50 \times 1$ und 1×100 . Wir betrachten den Rechenaufwand zur Berechnung des Matrizenproduktes $M_1 \cdot M_2 \cdot M_3 \cdot M_4$ gemessen in der Zahl der dafür notwendigen (skalaren) Multiplikationen für zwei verschiedene Klammerungen:

1.	$M_1 \cdot \underbrace{\underbrace{(M_2 \cdot \underbrace{(M_3 \cdot M_4)}_{[50,1,100]})}_{[20,50,100]}}_{[10,20,100]}$	5000	Operationen	
		+	100000	Operationen
		+	20000	Operationen
			125000	Operationen
2.	$\underbrace{\underbrace{(M_1 \cdot \underbrace{(M_2 \cdot M_3)}_{[20,50,1]})}_{[10,20,1]}}_{[10,1,100]} \cdot M_4$	1000	Operationen	
		+	200	Operationen
		+	1000	Operationen
			2200	Operationen

Allgemeine Problemstellung

Gegeben sei eine Folge r_0, r_1, \dots, r_N von natürlichen Zahlen, die die Dimensionen von N Matrizen $M_i \in \mathbb{R}^{r_{i-1} \times r_i}$ beschreibt. Es soll das Matrix-Kettenprodukt $M_1 \cdot M_2 \cdot \dots \cdot M_N$ berechnet werden. Das Problem besteht nun darin, die Klammerung für diesen Ausdruck zu finden, die die Anzahl der (skalaren) Multiplikationen minimiert.

Eine vollständige Suche (exhaustive search), die alle möglichen Klammerungen durchprobiert, hätte eine Komplexität, die exponentiell mit der Zahl N der Matrizen wächst.

Ansatz mittels dynamischer Programmierung

Definiere zunächst eine Hilfsgröße m_{ij} als die minimale Zahl der Operationen zur Berechnung des Teilprodukts

$$M_i \cdot M_{i+1} \cdot \dots \cdot M_j \quad \text{mit} \quad 1 \leq i \leq j \leq N$$

Um die Rekursionsgleichung für m aufzustellen, zerlegen wir dieses Produkt an der Stelle k mit $i \leq k < j$ in zwei Teile:

$$\underbrace{(M_i \cdot M_{i+1} \cdot \dots \cdot M_k)}_{m_{i,k}} \cdot \underbrace{(M_{k+1} \cdot M_{k+2} \cdot \dots \cdot M_j)}_{m_{k+1,j}} + r_{i-1} \cdot r_k \cdot r_j \quad \text{Operationen}$$

Aufgrund der Definition von m_{ij} erhalten wir die folgende Rekursionsgleichung durch Minimierung über die Split-Stelle k :

$$m_{i,j} = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j\} & j > i \end{cases}$$

Implementierung

Um zu einem lauffähigen Programm zu gelangen, muß noch festgelegt werden, in welcher Reihenfolge die $m_{i,j}$ ausgewertet werden. Zur Berechnung des Aufwands für ein Kettenprodukt der Länge l benötigt man gemäß obiger Rekursionsgleichung nur Ergebnisse für kürzere Produkte. Daher beginnt man mit einzelnen Matrizen, berechnet dann den Aufwand für alle Produkte von zwei Matrizen, dann den für Dreierprodukte, usw.:

Längenindex	auszuwertende Terme	
$l = 0$	$m_{i,i}$	$\forall i = 1, \dots, N$
$l = 1$	$m_{i,i+1}$	$\forall i = 1, \dots, N - 1$
$l = 2$	$m_{i,i+2}$	$\forall i = 1, \dots, N - 2$
...		
l	$m_{i,i+l}$	$\forall i = 1, \dots, N - l$
...		
$l = N - 2$	$m_{i,i+N-2}$	$\forall i = 1, 2$
$l = N - 1$	$m_{1,N}$	

Dies läßt sich dann als Programm (Pseudo-Code) folgendermaßen formulieren:

```

VAR r : ARRAY [0..N] OF INTEGER ;
    m,s : ARRAY [1..N],[1..N] OF INTEGER ;
BEGIN
  FOR i := 1 TO N DO
    m[i,i] = 0 ;
  END ;
  FOR l := 1 TO N - 1 DO          (* Laengenindex *)
    FOR i := 1 TO N - l DO      (* Positionsindex *)
      j := i + l ;
      m[i,j] := min_{i ≤ k < j} { m[i,k] + m[k+1,j] + r[i-1] · r[k] · r[j] };
      s[i,j] := arg min_{i ≤ k < j} { m[i,k] + m[k+1,j] + r[i-1] · r[k] · r[j] };
    END ;
  END ;
END ;

```

Ergebnisse

- $m[1, N]$: Unter diesem Eintrag in der Tabelle m findet man die Anzahl der minimal nötigen Operationen.
- $s[1, N]$: Unter diesem Eintrag der Tabelle s ist die *Split-Stelle* für die äußerste Klammerung (Matrixprodukt der größten Länge) vermerkt. Für die weitere Klammerung muß man unter den Einträgen für die entsprechenden Indexgrenzen nachschauen.
- Die Laufzeitkomplexität des vorgestellten Algorithmus ist $O(N^3)$.

1.4.3 Memoization: Tabellierung von Zwischenwerten

Die mögliche Ineffizienz direkter rekursiver Implementierung kann man oft durch Memoization vermeiden, indem man Zwischenwerte in Tabellen speichert.

Beispiel: Fibonacci-Zahlen

```

CONST N_max = 100 ;
VAR   F := ARRAY [0..N_max] OF INTEGER {-1,..} ;

PROCEDURE Fibonacci(n : CARDINAL) : CARDINAL =
BEGIN
  IF F[n] < 0 THEN
    IF n <= 1 THEN
      F[n] := n ;
    ELSE
      F[n] := Fibonacci(n-1) + Fibonacci(n-2) ;
    END ;
  END ;
  RETURN F[n] ;
END Fibonacci ;

```

Dieses Programm hat den Vorteil, daß es die Definition der Fibonacci-Zahlen aus Abschnitt 1.2.5 genau widerspiegelt, aber dennoch (bis auf einen konstanten Faktor) genauso effizient ist, wie die dort vorgestellte iterative Lösung.

Auch das Problem der besten Klammerung von Matrixkettenprodukten kann mittels Memoization implementiert werden (Übungsaufgabe).

1.5 Rekursionsgleichungen

Dieses Kapitel beschäftigt sich mit Rekursionsgleichungen (Differenzgleichungen) wie sie bei der Laufzeitanalyse insbesondere rekursiver Algorithmen auftreten.

1.5.1 Sukzessives Einsetzen

Viele Rekursionsgleichungen lassen sich durch *sukzessives Einsetzen* lösen. (Bei den Beispielen 2 – 5 wird der Einfachheit halber angenommen, daß n eine Zweierpotenz ist: $n = 2^k$, $k \in \mathbb{N}$.)

Beispiel 1:

$$\begin{aligned}
 T(n) &= T(n-1) + n \quad , \quad n > 1 \quad (T(1) = 1) \\
 &= T(n-2) + (n-1) + n \\
 &= \dots \\
 &= T(1) + 2 + \dots + (n-2) + (n-1) + n \\
 &= \frac{n(n+1)}{2}
 \end{aligned}$$

Beispiel 2:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + 1 \quad , \quad n > 1 \quad (T(1) = 0) \\
 &= T\left(\frac{n}{4}\right) + 1 + 1 \\
 &= T\left(\frac{n}{8}\right) + 1 + 1 + 1 \\
 &= \dots \\
 &= T(1) + \lg n \\
 &= \lg n
 \end{aligned}$$

Beispiel 3:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + n \quad , \quad n > 1 \quad (T(1) = 0) \\
 &= T\left(\frac{n}{4}\right) + \frac{n}{2} + n \\
 &= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n \\
 &= \dots \\
 &= T(1) + \dots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n \\
 &= 2(n-1) \quad \text{für } n > 1
 \end{aligned}$$

Beispiel 4:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \quad , \quad n > 1 \quad (T(1) = 0) \\
\frac{T(n)}{n} &= \frac{T\left(\frac{n}{2}\right)}{n/2} + 1 \\
&= \frac{T\left(\frac{n}{4}\right)}{n/4} + 1 + 1 \\
&= \dots \\
&= \lg n \\
T(n) &= n \lg n
\end{aligned}$$

Beispiel 5:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 1 \quad , \quad n > 1 \quad (T(1) = 1) \\
&= 2[2T\left(\frac{n}{4}\right) + 1] + 1 \\
&= 4T\left(\frac{n}{4}\right) + 3 \\
&= 8T\left(\frac{n}{8}\right) + 7 \\
&= \dots \\
&= n \cdot T(1) + n - 1 \\
&= 2n - 1 \\
\text{Verifiziere: } &= 2 \cdot (n - 1) + 1 \quad \checkmark
\end{aligned}$$

1.5.2 Master-Theorem für Rekursionsgleichungen

Bei der Analyse von Algorithmen, die nach der Divide-and-Conquer-Strategie entworfen wurden, treten meist Rekursionsgleichungen der folgenden Form auf:

$$T(n) = \begin{cases} 1 & \text{für } n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + d(n) & n > 1 \end{cases}$$

mit Konstanten $a \geq 1$ und $b > 1$ und einer positiven Funktion $n \mapsto d(n)$. Statt $T\left(\frac{n}{b}\right)$ kann auch $T(\lfloor \frac{n}{b} \rfloor)$ oder $T(\lceil \frac{n}{b} \rceil)$ stehen.

Dann gilt für $d(n) \in O(n^\gamma)$ mit $\gamma > 0$:

$$T(n) \in \begin{cases} O(n^\gamma) & a < b^\gamma \\ O(n^\gamma \log_b n) & \text{für } a = b^\gamma \\ O(n^{\log_b a}) & a > b^\gamma \end{cases}$$

Anmerkung: Es gibt eine allgemeinere Form des Master-Theorems, die allerdings auch etwas umständlicher ist [Cormen et al., S. 62 f.].

Beweis

in drei Schritten:

- Beweis für $d(n) = n^\gamma$ und Potenzen von b ($n = b^k$, $k \in \mathbb{N}$)
- Erweiterung von $d(n) = n^\gamma$ auf allgemeine Funktionen $n \mapsto d(n)$
- Erweiterung von Potenzen von b auf beliebige n .

Teil a: Für $\frac{n}{b^i}$ statt n gilt speziell für $i = 1, 2, 3, \dots$:

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right)$$

Sukzessives Einsetzen liefert:

$$\begin{aligned} T(n) &= a \cdot T\left(\frac{n}{b}\right) + d(n) \\ &= a \cdot \left[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right) \right] + d(n) \\ &= a^2 \cdot T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\ &= a^3 \cdot T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\ &= \dots \\ &= a^k T\left(\frac{n}{b^k}\right) + \sum_{j=0}^{k-1} a^j d\left(\frac{n}{b^j}\right) \end{aligned}$$

Wir wählen einen speziellen Wert für k :

$$k_n := \log_b n, \quad \text{d.h. } n = b^{k_n}$$

und erhalten damit

$$T(n) = a^{k_n} \cdot T(1) + \sum_{j=1}^{k_n-1} a^j d(b^{k_n-j})$$

Wir wählen speziell:

$$n \mapsto d(n) = n^\gamma \text{ mit } \gamma > 0$$

Dann gilt:

$$\begin{aligned}
 \sum_{j=0}^{k_n-1} a^j d(b^{k_n-j}) &= \sum_{j=0}^{k_n-1} a^j b^{\gamma k_n} b^{-\gamma j} \\
 &= b^{\gamma k_n} \cdot \sum_{j=0}^{k_n-1} \left(\frac{a}{b^\gamma}\right)^j \\
 \text{falls } a \neq b^\gamma : &= b^{\gamma k_n} \cdot \frac{\left(\frac{a}{b^\gamma}\right)^{k_n} - 1}{\frac{a}{b^\gamma} - 1} \\
 &= \frac{a^{k_n} - b^{\gamma k_n}}{\frac{a}{b^\gamma} - 1} \\
 \text{falls } a = b^\gamma : &= b^{\gamma k_n} \cdot \sum_{j=0}^{k_n-1} 1 \\
 &= k_n \cdot b^{\gamma k_n}
 \end{aligned}$$

Beachte:

$$\begin{aligned}
 a^{k_n} &= a^{\log_b n} \\
 &= (b^{\log_b a})^{\log_b n} \\
 &= (b^{\log_b n})^{\log_b a} \\
 &= n^{\log_b a}
 \end{aligned}$$

1. Fall $a < b^\gamma$: geometrische Reihe konvergiert für $k_n \rightarrow \infty$:

$$\begin{aligned}
 T(n) &\leq a^{k_n} + b^{\gamma k_n} \cdot \frac{-1}{\frac{a}{b^\gamma} - 1} \\
 &= n^{\log_b a} + n^\gamma \cdot \underbrace{\frac{1}{1 - \frac{a}{b^\gamma}}}_{>0 \text{ und unabh. von } n}
 \end{aligned}$$

Wegen $\log_b a < \gamma$ folgt:

$$T(n) \in O(n^\gamma)$$

2. Fall $a = b^\gamma$: Wegen $\gamma = \log_b a$ ist

$$\begin{aligned}
 T(n) &= a^{k_n} + b^{\gamma k_n} \cdot k_n \\
 &= n^{\log_b a} + n^\gamma \cdot \log_b n \\
 &= n^\gamma + n^\gamma \cdot \log_b n
 \end{aligned}$$

Also:

$$T(n) \in O(n^\gamma \log_b n)$$

3. Fall $a > b^\gamma$:

$$T(n) = a^{k_n} + \frac{a^{k_n} - (b^\gamma)^{k_n}}{\frac{a}{b^\gamma} - 1}$$

Wegen $a^{k_n} > (b^\gamma)^{k_n}$ folgt:

$$\boxed{T(n) \in O(n^{\log_b a})}$$

Teil b: Erweiterung auf allgemeine Funktion $d(n)$:

$$T(n) = a^{k_n} \cdot T(1) + \sum_{j=0}^{k_n-1} a^j d(b^{k_n-j})$$

$d(n) \in O(n^\gamma)$ heißt: Für genügend großes n gilt:

$$d(n) < c \cdot n^\gamma \text{ für geeignetes } c > 0$$

Damit kann man $T(n)$ nach oben abschätzen:

$$T(n) \leq a^{k_n} \cdot T(1) + c \cdot b^{\gamma k_n} \cdot \sum_{j=0}^{k_n-1} \left(\frac{a}{b^\gamma}\right)^j$$

Auf diese Ungleichung kann man dieselben Überlegungen wie im Teil a anwenden und erhält dieselben Ergebnisse in Abhängigkeit von γ .

Teil c: Erweiterung auf allgemeine n [Cormen et al., S. 70]

$$T(n) = aT(\lceil \frac{n}{b} \rceil) + d(n)$$

Sukzessives Einsetzen liefert die Argumente $n, \lceil \frac{n}{b} \rceil, \lceil \frac{\lceil \frac{n}{b} \rceil}{b} \rceil, \dots$

Definiere:

$$n_i := \begin{cases} n & \text{für } i = 0 \\ \lceil \frac{n_{i-1}}{b} \rceil & \text{für } i > 0 \end{cases}$$

Ungleichung: $\lceil x \rceil \leq x + 1$

$$\begin{aligned} n_0 &\leq n \\ n_1 &\leq \frac{n}{b} + 1 \\ n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1 \\ \text{allgemein: } n_i &\leq \frac{n}{b^i} + \sum_{j=0}^{i-1} \frac{1}{b^j} \\ &\leq \frac{n}{b^i} + \frac{b}{b-1} \end{aligned}$$

Für $T(n)$ kann man dann zeigen:

$$T(n) \leq a^{\lfloor \log_b n \rfloor - 1} \cdot T(1) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j d(n_j)$$

und schätzt dann weiter nach oben ab (mühsame Rechnung).

1.5.3 Rekursionsungleichungen

Oft ist es einfacher, statt einer Gleichung eine Ungleichung zu betrachten.

Beispiel: Mergesort – Beweis durch vollständige Induktion.

Konstruktive Induktion

Eine spezielle Variante der vollständigen Induktion, die sog. *konstruktive Induktion*, wird benutzt, um zunächst noch unbekannte Konstanten zu bestimmen.

Beispiel: Fibonacci-Zahlen

$$f(n) = \begin{cases} n & \text{für } n \leq 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

Behauptung: Für $n \geq n_0$ gilt:

$$f(n) \geq a \cdot c^n$$

mit Konstanten $a > 0$ und $c > 0$, die noch zu bestimmen sind.

Induktionsschritt:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &\geq ac^{n-1} + ac^{n-2} \end{aligned}$$

Konstruktiver Teil: Wir bestimmen ein $c > 0$ mit der Forderung

$$ac^{n-1} + ac^{n-2} \stackrel{!}{\geq} a \cdot c^n$$

Umformen ergibt:

$$c + 1 \geq c^2$$

oder

$$c^2 - c - 1 \leq 0$$

Lösung:

$$c \leq \frac{1 + \sqrt{5}}{2}$$

(Negative Lösung scheidet aus.)

Induktionsanfang: Wähle $a > 0$ und n_0 geeignet.

Beispiel zur Warnung

Der exakte Wert der Konstanten in der Ungleichung ist wichtig. Wir betrachten dazu folgendes Beispiel:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$

Falsch ist der folgende „Beweis“:

$$\begin{aligned} \text{Behauptung: } T(n) &\leq c \cdot n \in O(n) && \boxed{\text{Falsch!}} \\ \text{„Beweis“: } T(n) &\leq 2 \cdot (c \cdot \lfloor \frac{n}{2} \rfloor) + n \\ &\leq cn + n \\ &\in O(n) \end{aligned}$$

Der Fehler:

- Behauptet wurde: $T(n) \leq c \cdot n$
- Gezeigt wurde: $T(n) \leq (c + 1)n$