Effizienz von Algorithmen

Eine Einführung

Michael Klauser

LMU

30. Oktober 2012

Ein einführendes Beispiel

Wie würdet ihr einen Stapel Karten Sortieren?

Ein einführendes Beispiel

Wie würdet ihr einen Stapel Karten Sortieren?

Sortieren-durch-Einfügen

Die Idee

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \cdots, a_n \rangle$.
Ausgabe: Eine Permutation $\langle a'_1, a'_2, \cdots, a'_n \rangle$ der Eingabefolge, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Prinzip

1. Finde den minimalen Wert in der Liste.
2. Vertausche ihn mit dem Wert an der ersten Position.
3. Wiederhole schritt zwei für den rest der Liste. (Beginnend mit der Letzten Startposition +1)

Ein paar Code Beispiele

Haskell

```
sSort :: (Ord a) => [a] -> [a]
sSort [ = [
sSort xs = m : sSort (delete m xs)
  where
      m = minimum xs]]
  -- delete löscht das erste Auftreten
  -- eines Elements in einer Liste
```

Python

```
def selectionsort(seq):
    for i in range(len(seq) - 1):
        k = i
        for j in range(i, len(seq)):
            if seq[j] < seq[k]:
                k = j
        seq[i], seq[k] = seq[k], seq[i]
```

C

```
void selectionsort( int anzahl, int daten[])
{
    int i, k, t, min;
    for( i = 0; i < anzahl-1; i++)
    {
        min = i;
        for( k = i+1; k < anzahl; k++)
        {
            if( daten[k] < daten[min])
                min = k;
        }
        t = daten[min];  // Tauschen
        daten[min] = daten[i];
        daten[i] = t;
    }
 }
```

Pseudocode

```
Data: Feld A[]
for  j=2 to len(A) do
    key = A[j];
    i = j + 1;
    while i > 0 && A[i]>key do
        A[i+1] = A[i];
        i = i -1
    end
end
```

Algorithm 1: Sortieren durch Einfügen als Pseudocode.

Definition des Systems

Für eine präzise mathematische Effizienzanalyse benötigen wir ein Rechenmodell, das definiert:

- Welche Datentypen existieren.
- Wie Daten abgelegt werden.
- Welche Operationen zulässig sind.
- Welche Zeit eine bestimmte Operation für einen definierten Datentyp benötigt.

Random Access Maschine

Formal ist ein Rechenmodell gegeben durch die Random Access Maschine, kurz RAM.

Ein RAM ist ein idealer 1-Prozessrechner mit einfachem aber unbegrenzt großem Speicher.

Basisoperationen und ihre Kosten

Definition

Als Basisoperationen bezeichnen wir:

- **Kontrolloperationen:** Wertübergaben, Verzweigungen, Programmaufrufe
- **Datenverwaltungsoperationen:** Laden, Ablegen, Kopieren
- **Arithmetische Operationen:** Multiplikation, Addition, Division, +1, -1, Modulo
- **Bitweise Operationen:** shift, or, and ,not

Vereinfachung

Wir nehmen an, dass jede Operation bei allen Operanden gleich viel Zeit benötigt bzw. Kostet.

Effizienz

Ein wichtiges Merkmal eines Algorithmus ist seine Effizienz, insbesondere wenn man für ein geg. Problem mehrere Algorithmen zur Auswahl hat.

Effizienz ist kein einfaches Kriterium, sonder besteht aus:

- Speicher-Effizienz: Wie viel Speicher braucht der Algorithmus.
- Subjective-Effizienz: Wie einfach ist Algorithmus zu erfassen.
- Laufzeit-Effizienz: Wie schnell ist der Algorithmus.

Speicher-Effizienz

Beschreibt den Speicherbedarf eines Algorithmus.
Kann analog zur Laufzeit-Effizienz behandelt werden

- Speicher bedarf pro Variable wird erhobenen. (Abhängig von der Implementierung)
- Garbage Collection spielt eine entscheidende Rolle.

Subjective-Effizienz

- Wie einfach ist der Algorithmus geistig zu erfassen.
- Wie natürlich löst er das Problem.

Die fehlerfreie Implementierung hängt vom Verständnis des Algorithmus abhängt, ebenso wie der Zeitaufwand für die Implementierung.

Naiver Grundansatz

Implementierung des Algorithmus auf dem System von Interesse.

- **Nicht Objective:** Da sowohl die Codierung des Algorithmus als auch das zugrunde liegende System (Compiler, Sprache, Hardware) die Laufzeit enorm beeinflussen können.
- **Hoher Arbeitsaufwand:** Mehrfach Implementation notwendig.

Wir suchen

Eine Möglichkeit, eine Aussage über das Laufzeitverhalten von Algorithmen, ohne deren Implementierung zu machen.

Ein neuer Ansatz

Wir zählen für einen gegebenen Satz von Eingaben alle Anweisungen j mit einer Zeit t_j, wobei t_j von der Art von j abhängt.

- Für das Ausführen einer Zeile Pseudocode ist ein konstanter Zeitaufwand notwendig.
- Verschieden Zeilen können einen unterschiedlichen Zeitaufwand aufweisen.

$\Rightarrow$ Jede Zeile i benötigt einen konstanten Zeitaufwand den wir Kosten c_j nennen.

Laufzeit von Sortieren durch Einfügen

```
   Data: Feld A[]
1  for  j=2 to len(A) do   /*c1  n */
2  |    key = A[j];                                   /*  c2   n − 1  */
3  |    i = j + 1;                                    /*  c3   n − 1  */
4  |    while i > 0 && A[i]>key do   /*c4 Σ_{i=2}^{n} t_j */
5  |    |   A[i+1] = A[i];                           /*  c5   Σ_{i=2}^{n}(t_j − 1)  */
6  |    |   i = i -1;                                /*  c6   Σ_{i=2}^{n}(t_j − 1)  */
7  |    end
8  |    A[i+1] =key ;                                 /*  c7   n − 1  */
9  end
```

Algorithm 2: Effizienz von Sortieren-durch-Einfügen.

t_j ist die Anzahl der Schleifendurchläufe für ein gegebenes j.

Die Laufzeit eines Algorithmus ist die Summe aller Laufzeiten.

Laufzeit von Sortieren durch Einfügen

Laufzeit $T(n)$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_j + c_5 \sum_{i=2}^{n} (t_j - 1) + c_6 \sum_{i=2}^{n} (t_j - 1) + c_7(n-1) \quad (1)$$

Der best mögliche Fall

Im best möglichen Fall ist $A[]$ sortiert.

Für jedes $j = 2, 3, 4, \cdots, n$ gilt in Zeile vier immer $A[i] < key \Rightarrow t_j = 1$ für alle j.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned} \tag{2}$$

Der schlechtest mögliche Fall

Im schlechtest möglichen Fall ist $A[]$ absteigend sortiert.

Jedes Element von $A[]$ muss mit jedem bereits sortierten Element verglichen werden.

$\Rightarrow t_j = j \; \forall \; j \in n$

Laufzeit im schlechtest möglichen Fall

Mit

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \tag{3}$$

und

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2} \tag{4}$$

schreiben wir

$$T(n) = c_1 n + c_2(n-1) + c_3(n+1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \tag{5}$$
$$+ c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4 + c_5 + c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 c_6}{2} + c_7\right) n \tag{6}$$
$$-(c_2 + c_3 + c_4 + c_7)$$

Der schlechteste Fall als Regelfall

Laufzeit im schlechtest Fall

$$T(n) \sim an^2 + bn + c \tag{7}$$

Warum der schlechteste Fall?

- Dieser Fall definiert eine obere Schranke.
- Der schlechteste Fall ist der Regelfall.
- Der mittlere Fall ist genauso schlecht wie der schlechtest.
  - Im mittleren Fall sind alle Elemente gleichverteilt in A.
  - d.h. die Hälfte der Elemente $A[1 \ldots j-1] < A[j]$

  $\Rightarrow$ $T(n) \sim an^2 + bn + c$

Wachstumsgrad

Eine weitere Abstraktion der Laufzeit führt zur sogenannten Wachstumsrate beziehungsweise zum Wachstumsgrad.

- Wir betrachten nur den führenden Term.
  $\Rightarrow$ an^2
- Wir verwerfen a, da diese Größe nur von Konstanten abhängt.
  $\Rightarrow$ n^2

Zur allgemeinen und formalen Darstellung der Wachstumsrate führen wir die Θ-Notation ein.

Die Θ Notation

Sie beschreibt das asymptotische Laufzeitverhalten eines Algorithmus im schlechtesten Fall.
Für eine gegeben Funktion g bezeichne wir mit $\Theta(g)$ die Menge der Funktionen

$$\Theta(g) = \{f : \exists c_i > 0 \in n_0 | 0 < c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n > n_0\} \quad (8)$$

Anmerkung

Im Allgemeinen wird mit der Θ Notation sehr schlampig umgegangen d.h. Sie wird z.B. auf $\mathbb{R}$ oder nur auf Teilmengen von $\mathbb{N}$ angewendet.
Des Weiteren verwendet man $g = \Theta(g)$ anstelle von $g \in \Theta(g)$

Die $\mathcal{O}$-Notation

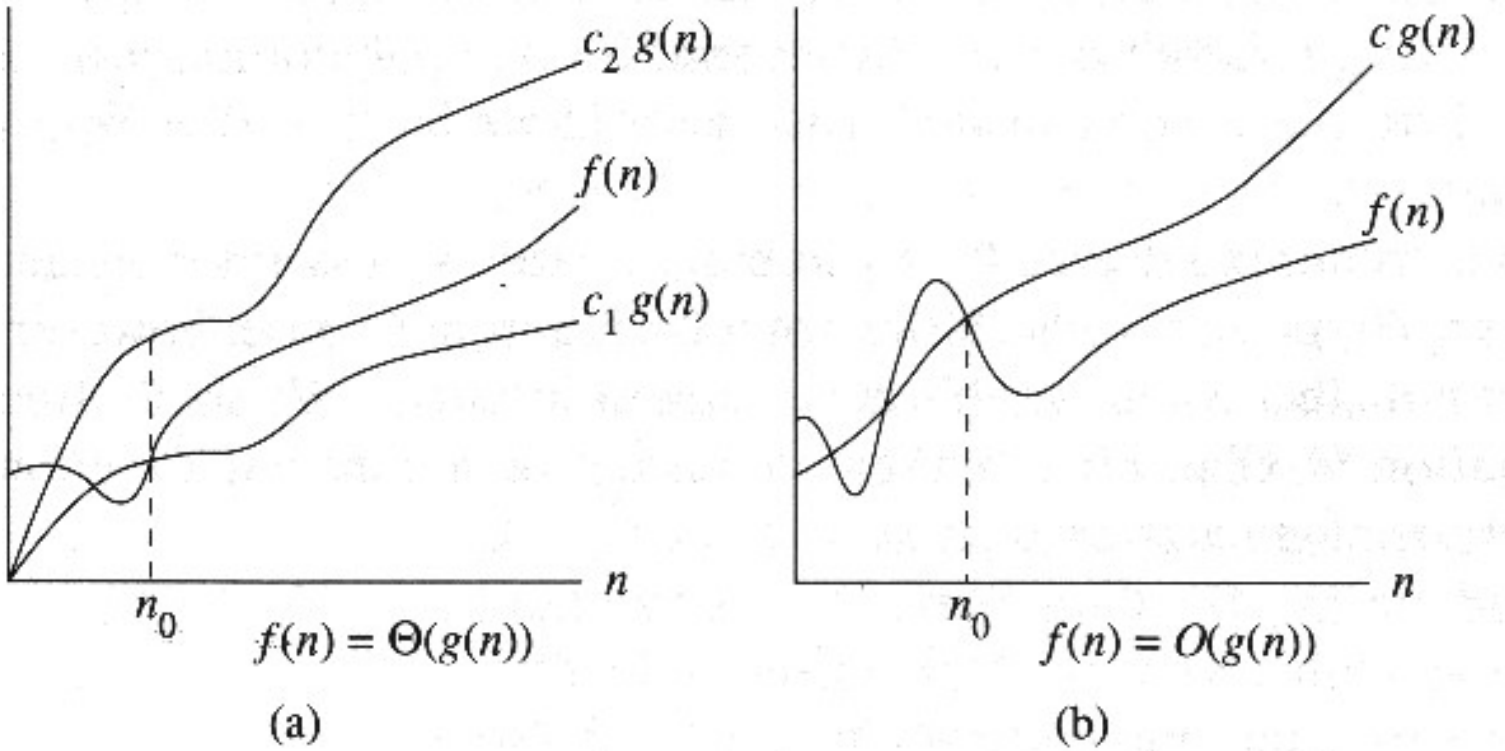
Definiert nur eine obere asymptotische Schranke.

$$\mathcal{O}(g) = \{f : \exists c_i > 0 \in n_0 | 0 \leq f(n) \leq c_2 g(n) \forall n > n_0\} \tag{9}$$

Anmerkung

Aus $f(n) = \Theta(g(n))$ folgt $f(n) = \mathcal{O}(g(n))$.

Abbildung: Vergleich der Θ- und $\mathcal{O}$-Notation. [Stasys Jukna, Uni-Frankfurt]



Beispiel:

$$\frac{1}{2}n^2 - 3n = \Theta(n^2) \tag{10}$$

Für $n > n_0$ gilt

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \tag{11}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \tag{12}$$

Für $n \geq 1 \Rightarrow c_2 \geq \frac{1}{2} \wedge$ Für $n \geq 7 c_1 \leq \frac{1}{14}$
$\Rightarrow \frac{1}{2}n^2 - 3n = \Theta(n^2)$

Θ für Sortieren-durch-Einfügen

Sortieren-durch-Einfügen

$$T(n) \sim an^2 + bn + c \Rightarrow \Theta(n^2) \tag{13}$$

$$c_1 = \frac{a}{4}\ ;\ c_2 = \frac{7a}{4}\ ;\ n_0 = 2\max\left(\frac{|b|}{a}, \sqrt{\frac{|c|}{a}}\right) \tag{14}$$

Im Allgemeinen gilt für ein Polynom

$$p(n) = \sum_{i=0}^{d} a_i n^i \tag{15}$$

$$p(n) = \Theta(n^d) \tag{16}$$

wenn a_i = konstant und $a_i > 0$.

Sortieren-durch-Mischen

Das Sortierverfahren Sortieren durch Mischen erzeugt eine sortierte Folge durch Verschmelzen sortierter Teilstücke.
Das Verfahren beruht auf dem Divide-and-Conquer-Prinzip und ruft sich rekursiv selbst auf.

Teilen-und-Beherrsche

Ein Teile-und-Beherrsche Algorithmus zerlegt das eigentliche Problem so lange in kleinere und einfachere Teilprobleme, bis diese lösbar sind. Anschließend wird aus diesen Teillösungen eine Lösung für das Gesamtproblem konstruiert.
Das Paradigma von Teile-und-Beherrsche umfasst drei grundlegende Schritte auf **jeder** Rekursionsebene.

1. **Teile** das Problem in mehrere Teilprobleme auf, die kleinere Instanzen des typgleichen Problems sind.
2. **Beherrsche** die Teilprobleme rekursiv. (Wenn die Teilprobleme klein genug sind werden Sie direkt gelöst.)
3. **Vereinige** die Lösung der Teilprobleme zur Lösung des Gesamtproblems.

Sortieren-durch-Mischen

Prinzip

1. Die zu sortierende Folge werden zunächst in zwei Teilfolgen zu je $\frac{n}{2}$ Elemente aufgeteilt.
2. Die zwei Teilfolgen werden rekursive durch Sortieren-durch-Mischen sortiert.
3. Die sortierten Hälften zu einer insgesamt sortierten Folge verschmolzen (Merge).

Die Rekursion bricht ab, wenn die zu sortierende Teilfolge die Länge 1 hat.

Sortieren-durch-Mischen

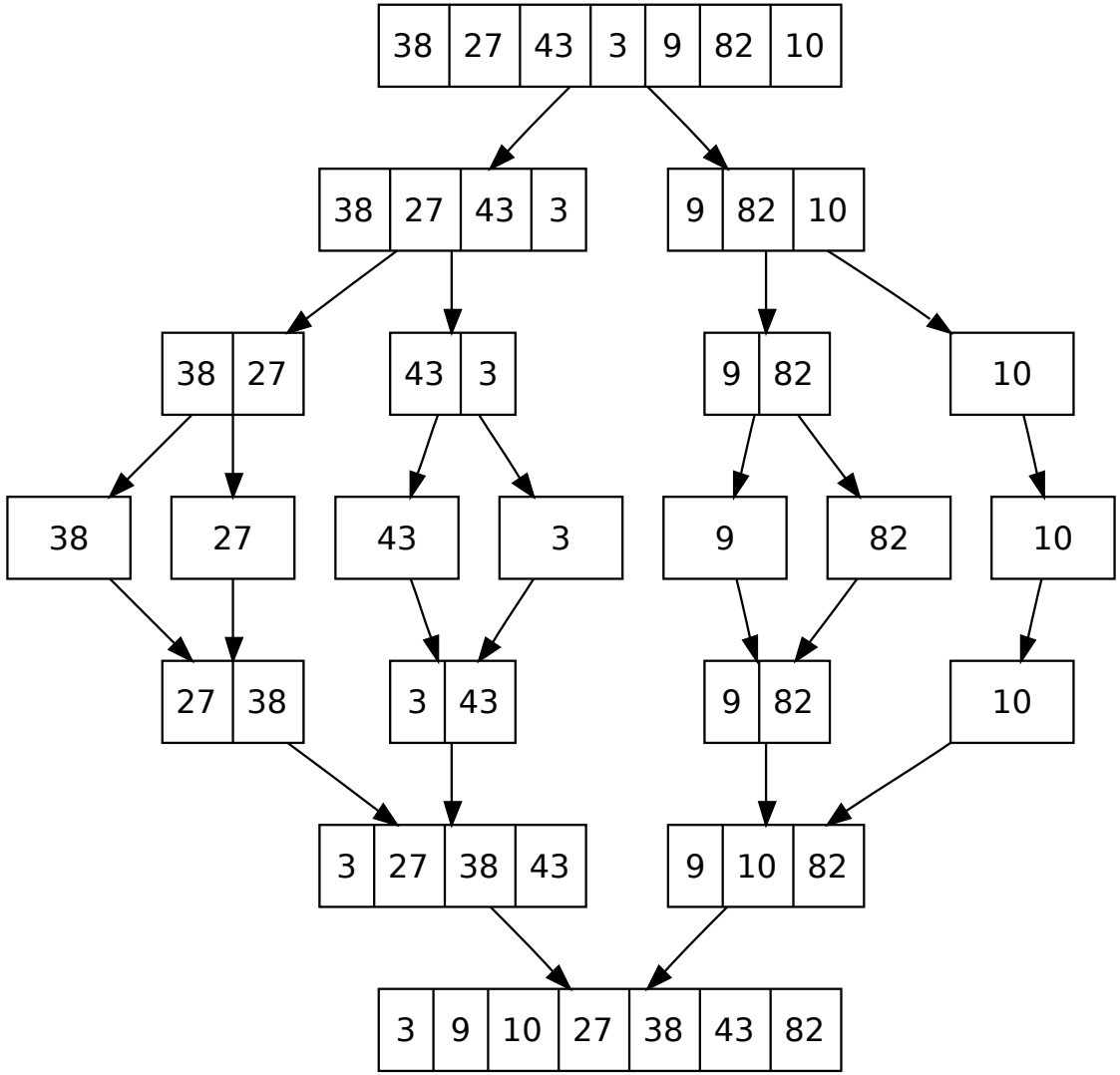


Abbildung: Sortieren durch Mischen

Die Merge-Funktion

Die zentrale Operation des Sortieren-durch-Mischen Algorithmus ist das Mischen der **sortierten** Folgen.
Wir Mischen mithilfe der `MERGE(A,p,q,r)` Funktion.

- $A[]$ ist ein sortiertes Feld.
- p, q und r sind Indizes des Feldes A, wobei $p \leq q < r$ gilt.
- Die Teilfelder $A[p \ldots q]$ und $A[q+1 \ldots r]$ sind bereits Sortiert.

Pseudocode der Merge Funktion

```
MERGE(A,p,q,r);
n_1 = q -p +1;
n_2 = r -q;
L = Feld[1 . . . n_1 + 1];
R = Feld[1 . . . n_2 + 1] ;
for  i=1 to n_1 do
  |    L[i] = A[p+i-1];
end
for j = 1 to n_2 do
  |    R[j] = A[q+j]
end
L[n_1 + 1] = ∞;
L[n_2 + 1] = ∞;
i =1;
j =1;
for k = p to r do
  |    if L[i] ≤ R[j] then
  |      |    A[k] = L[i];
  |      |    i = i+1;
  |    else
  |      |    A[k] = R[j];
  |      |    j=j+1;
  |    end
end
```

Algorithm 3: Sortieren durch Einfügen als Pseudocode.

Funktionsweise der Merge Funktion am Beispiel eines Kartenspiels

Wir starten mit zwei Kastenstapeln, die verdeckt auf dem Tisch liegen. Die kleinste Karte liegt obenauf.

1. Wir decken die obersten Karten auf.
2. Wir vergleichen die aufgedeckten Karten.
3. Die kleiner (größere) Karte wird separate abgelegt.
4. Wir decken wider eine neue Karte.
5. Wir wiederholen dies schritte bis es nur noch einen Stapel gibt.

Da wir nur immer die obersten Karten vergleichen und somit nur n Grundschritte ausfuhren gilt

$$T_{\text{MERGE}} = \Theta(n)$$

Die MERGE-SORT Funktion

Die MERGE-SORT(Sortieren durch Mischen) Funktion benutzt die MERGE Funktion als Unterroutine.

MERGE-SORT(A,p,r);
if $p < r$ **then**
  $q = \frac{p+r}{2}$;
  MERGE-SORT(A,p,q) ;
  MERGE-SORT(A,q+1,r) ;
  MERGE(A,p,q,r) ;
end

Analyse von Sortieren-durch-Mischen

Bei Algorithmen, die sich selbst aufrufen können wir die Laufzeit als **Rekursionsgleichung**.
Wenn die Größe des Teilproblems hinreichend klein ist, dann benötigt die direkte Lösung eine konstante Zeit.
$\Rightarrow \Theta(1)$

- Die Aufteilung des Gesamtproblems führt zu a Teilproblemen der Größe $\frac{1}{b}$.
- Im Fall von Sortieren-durch-Mischen gilt $a = b = 2$.

$\Rightarrow$ Das lösen eines Teilproblems der Größe $\frac{n}{b}$ dauert $T\left(\frac{n}{b}\right)$.

$\Rightarrow$ Das lösen von a Teilproblemen benötigt somit $aT\left(\frac{n}{b}\right)$.

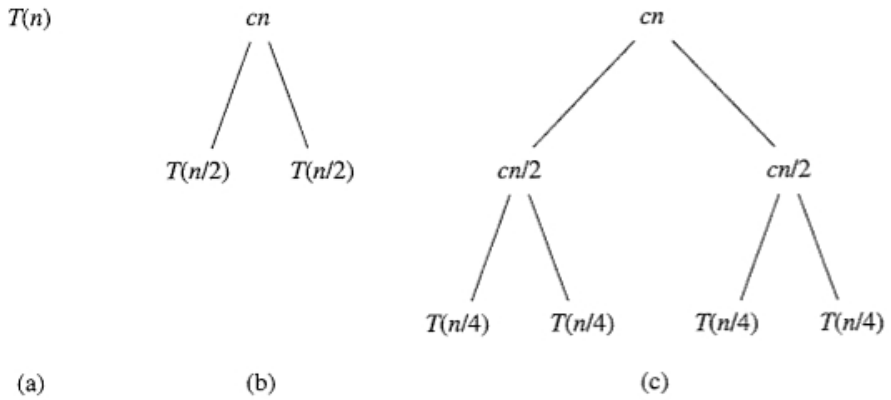
- Wir nehmen an das die Aufteilung die Zeit $D(n)$ benötigt und die Zusammenführung die Zeit $C(n)$.

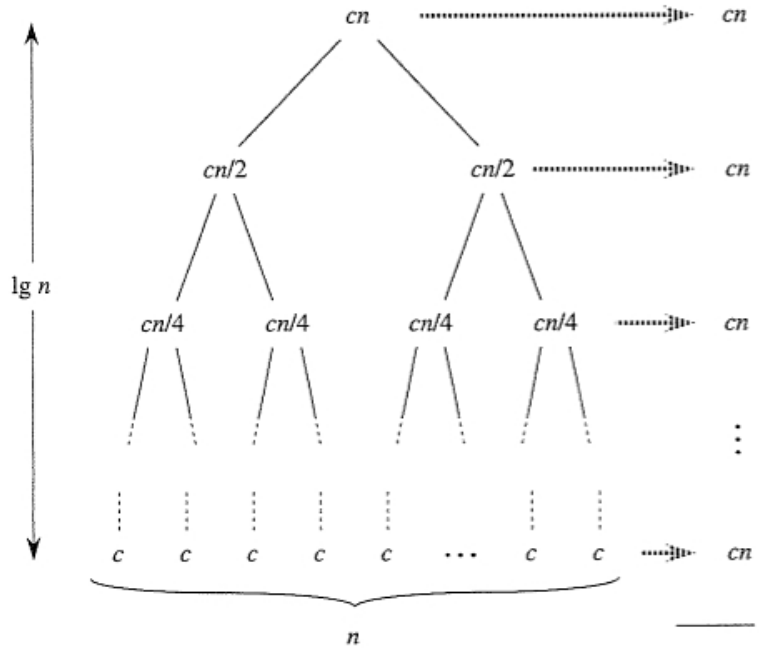
Analyse von Sortieren-durch-Mischen

$$\Rightarrow T(n) = \begin{cases} \Theta(1) & \text{falls} n = 1 \\ 2T(n/2) + \Theta(n) & \text{falls} n > 1 \end{cases} \tag{17}$$

Die Rekursionsgleichung ergibt
$T(n) = \Theta(n \lg(n))$

Rekursionsbaum für Sortieren-durch-Mischen





Vergleich von verschieden Ordnungen

$\Theta(1)$:	konstanter Aufwand, unabhängig von n
$\Theta(n)$:	linearer Aufwand (z.B. Einlesen von n Zahlen)
$\Theta(n \ln n)$:	Aufwand von Sortieren-durch-Mischen
$\Theta(n^2)$:	quadratischer Aufwand
$\Theta(n^k)$:	polynominaler Aufwand(k= *konstant*)
$\Theta(k^n)$:	exponentieler Aufwand(k= *konstant*)
$\Theta(n!)$:	Fakultativer Aufwand (z.B. Bestimmung aller Permutationen von n Elementen)

Konkreter Vergleich von Laufzeiten verschiedener Ordnungen

Annahme: Ein Schritt dauert $1\mu s$.

n=	10	20	30	40	50	60
n	$10\mu s$	$20\mu s$	$30\mu s$	$40\mu s$	$50\mu s$	$60\mu s$
n^2	$100\mu s$	$400\mu s$	$900\mu s$	1,6ms	2.5ms	3.6ms
n^3	1ms	8ms	27 ms	64ms	125ms	216ms
2^n	1ms	1s	18min	13d	36a	36,6ka
3^n	59ms	58min	6,5a	385ka	100Ma	10^6Ma
$n!$	3.62s	77ka	10Ea	$10^{16}Ea$		

Quellen

- Algorithms Robert Sedgewick
- Algorithms in a Nutshell George T. Heineman
- Algorithmen Eine Einfuerung H.Cormen
- Entwurf und Abalyse von Algorithmen M. Nebel