

Algo Zusammenfassung

Tim Nogga

Contents

Chapter 1	Einleitung	Page 2
1.1	Insertionsort	2
1.2	Größenordnungen	2
Chapter 2	Methoden zum Entwurf von Algorithmen	Page 4
2.1	Divide and Conquer	4
	Binary Search — 4 • Merge Sort — 5 • Divide-and-Conquer am Beispiel von Strassen — 5 • Master Theorem — 6	
2.2	Greedy Algorithmen	7
	Optimale Auswahl von Aufgaben — 8	

Chapter 1

Einleitung

1.1 Insertionsort

Algorithm 1: Insertion-Sort

Input: $\text{int}[] \text{ a}$
Output: a sortiert

```
1 for  $\text{int } j = 1; j < a.length; j++$  do
2    $\text{int } x = a[j];$ 
3    $\text{int } i = j - 1;$ 
4   while  $i \geq 0 \wedge a[i] > x$  do
5      $a[i + 1] = a[i];$ 
6      $i = i - 1;$ 
7    $a[i + 1] = x;$ 
8 return  $a;$ 
```

Theorem 1.1.1

Die Ausgabe von Insertionsort ist stets eine aufsteigend sortierte Permutation der Eingabe.

Theorem 1.1.2

Die Laufzeit von Insertionsort ist in $O(n^2)$

1.2 Größenordnungen

Definition 1.2.1: Größenordnungen

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ zwei Funktionen

- ① $f(n) \in \mathcal{O}(g(n))$ falls $\exists c > 0, n_0 \in \mathbb{N}$ mit $f(n) \leq c \cdot g(n) \forall n \geq n_0$
- ② $f(n) \in \Omega(g(n))$ falls $\exists c > 0, n_0 \in \mathbb{N}$ mit $f(n) \geq c \cdot g(n) \forall n \geq n_0$
- ③ $f(n) \in \Theta(g(n))$ falls $f(n) \in \mathcal{O}(g(n))$ und $f(n) \in \Omega(g(n))$
- ④ $f(n) \in o(g(n))$ falls $\forall c > 0 \exists n_0 \in \mathbb{N}$ mit $f(n) \leq c \cdot g(n) \forall n \geq n_0$
- ⑤ $f(n) \in \omega(g(n))$ falls $\forall c > 0 \exists n_0 \in \mathbb{N}$ mit $f(n) \geq c \cdot g(n) \forall n \geq n_0$

Note:-

Es gilt für die exakten schranken $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, wenn $f(n) \in o(g(n))$ und umgekehrt, wenn $f(n) \in \omega(g(n))$

Chapter 2

Methoden zum Entwurf von Algorithmen

2.1 Divide and Conquer

2.1.1 Binary Search

Algorithm 2: Binary Search

Input: $\text{int}[] \text{ a}, \text{int } x, \text{int } l, \text{int } r$
Output: Bool

```
1 if  $l > r$  then
2   | return false;
3 int  $m = \lfloor \frac{l+r}{2} \rfloor$ ;
4 if  $a[m] < x$  then
5   | return binarySearch(a, x, m + 1, r);
6 if  $a[m] > x$  then
7   | return binarySearch(a, x, l, m - 1);
```

Theorem 2.1.1

Die Laufzeit von Binary Search ist in $O(\log n)$

Note:-

Erklärung:

- $l > r$ bedeutet, dass das gesuchte Element nicht in der Liste ist
- $a[m] < x$ bedeutet, dass das gesuchte Element rechts von m ist
- $a[m] > x$ bedeutet, dass das gesuchte Element links von m ist

Geht rekursiv weiter, bis das Element gefunden ist.

2.1.2 Merge Sort

Algorithm 3: Merge Sort

Input: int[] a, int l, int r
Output: a sortiert

```
1 if l < r then
2   int m = ⌊  $\frac{l+r}{2}$  ⌋;
3   mergeSort(a, l, m);
4   mergeSort(a, m + 1, r);
5   merge(a, l, m, r);
```

Algorithm 4: Merge

Input: int[] a, int l, int m, int r
Output: a sortiert

```
1 int [] l = new int[m - l + 1];
2 int [] r = new int[r - m];
3 for int i = 0; i < l.length; i++ do
4   l[i] = a[l + i];
5 for int i = 0; i < r.length; i++ do
6   r[i] = a[m + 1 + i];
7 int iL = 0;
8 int iR = 0;
9 int iA = l;
10 while iL < m-l + 1 and iR < r - m do
11   if l[iL] ≤ r[iR] then
12     a[iA] = l[iL];
13     iL++;
14     iA++;
15   else
16     a[iA] = r[iR];
17     iR++;
18     iA++;
19   while iL < m-l + 1 do
20     a[iA] = l[iL];
21     iL++;
22     iA++;
23   while iR < r - m do
24     a[iA] = r[iR];
25     iR++;
26     iA++;
```

Theorem 2.1.2

Die Laufzeit von Merge Sort ist in $O(n \log n)$

2.1.3 Divide-and-Conquer am Beispiel von Strassen

Note:-

Wenn man die Laufzeit von Strassen zu Simpleren Divide and Conquer Algorithmen vergleicht, wie Simple Product, ist die Laufzeit von Strassen besser

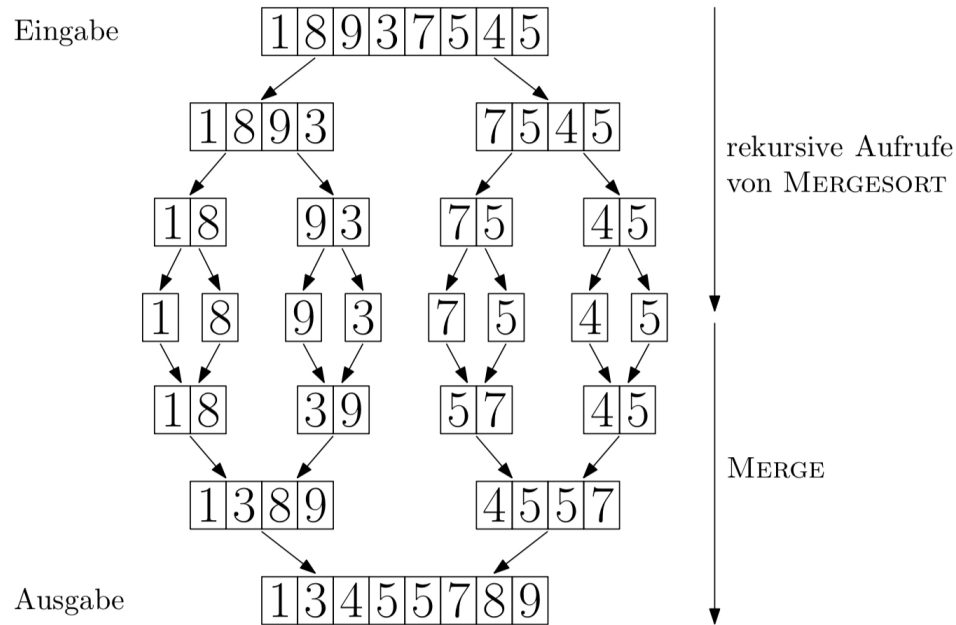


Figure 2.1: Mergesort

Theorem 2.1.3

Die Laufzeit von Strassen ist in $O(n^{\log_2 7})$

Proof: Siehe Seite 25 skript (Kein Bock das zu Texen) folgt aber aus $T(n) = 7T(n/2) + O(n^2)$

⊗

2.1.4 Master Theorem

Theorem 2.1.4 Mastertheorem

Seien $a \geq 1, b > 1$, Konstanten und sei $f: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ eine Funktion. Ferner sei $T: \mathbb{N} \rightarrow \mathbb{R}$ definiert durch $T(1) = \Theta(1)$ und

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

für alle $n > 1$. Die Funktion T kann wie folgt beschrieben werden:

- $T(n) = O(n^{\log_b a})$ falls $f(n) = O(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0$
- $T(n) = \Theta(n^{\log_b a} \log n)$ falls $f(n) = \Theta(n^{\log_b a})$
- Falls $T(n) = \Omega(n^{\log_b a + \epsilon})$ für eine Konstante $\epsilon > 0$ und falls $af(\frac{n}{b}) \leq cf(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n , dann gilt $T(n) = \Theta(f(n))$

Note:-

Im Prinzip werden nur die Funktionen n und $n^{\log_b a}$ verglichen. Im ersten Fall wächst f langsamer im zweiten gleich schnell. Verglichen zum ersten Fall führt das zu einem zusätzlichen $\log n$. Im dritten Fall wächst f schneller als $n^{\log_b a}$, also ist die Lösung $\Theta(f(n))$

Lemma 1. Seien $a \geq 1, b > 1$, Konstanten und sei $f: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ eine Funktion. Sei $T(n) = \Theta(O(n^{\log_b(a)})) + g(n)$ mit $g(n) \stackrel{\text{def}}{=} \sum_{i=0}^{\log_b(n)-1} a^i f(\frac{n}{b^i})$. Betrachte die Funktion beschränkt auf Werte von n mit $n = b^j$ für $j \in \mathbb{N}$.

- Falls $f(n) = O(n^{\log_b(a) - \epsilon})$ für eine Konstante $\epsilon > 0$, dann $g(n) = O(n^{\log_b(a)})$
- Falls $f(n) = \Theta(n^{\log_b(a)})$, dann $g(n) = \Theta(n^{\log_b(a)} \log n)$

- Falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und falls $af(\frac{n}{b}) \leq cf(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n , dann gilt $g(n) = \Theta(f(n))$

Note:-

Die Funktion $g(n)$ ist die Summe der Kosten der rekursiven Aufrufe. Das Lemma beschreibt das Mastertheorem für den Fall, dass n eine Potenz von b ist. Die Höhe des Rekursionsbaumes beträgt $\log_b(n)$. Logischerweise ist die gesamte Laufzeit die Summe über alle Knoten für jedes Level, wobei sich die Anzahl an Knoten mit der Höhe skaliert mit a^h hinzu kommt dann noch das die Kosten pro Level noch mit der Summe $\sum_{i=0}^{h-1} f(\frac{n}{b^i})$ gegeben ist. Also entsteht daher die Formel für die gesamte Laufzeit mit $\Theta(a^h) + \sum_{i=0}^{h-1} f(\frac{n}{b^i})$

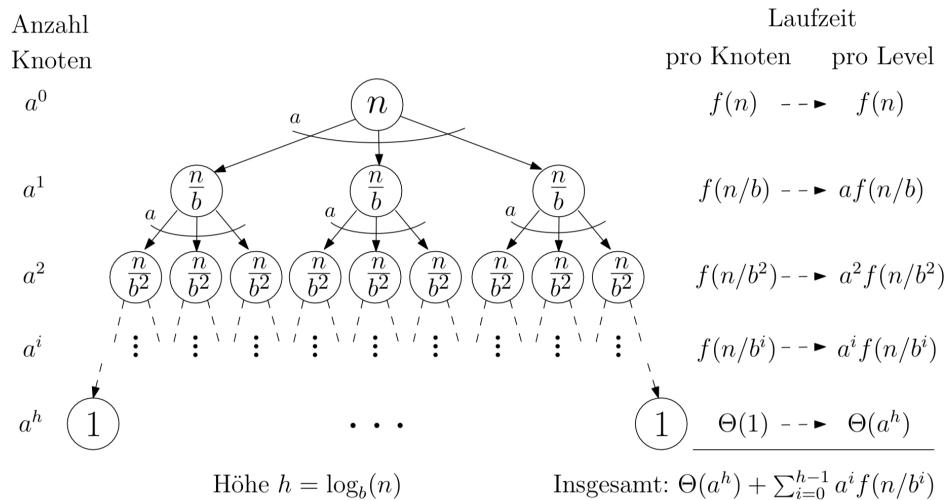


Figure 2.2: Visualized Lemma 1

2.2 Greedy Algorithmen

Definition 2.2.1: Greedy Algorithmen

Greedy Algorithmen sind Algorithmen die immer den besten lokalen Schritt wählen.

Example 2.2.1 (Wechselgeldproblem)

Das Wechselgeldproblem stellt die Münzen schritt für schritt zusammen, die am nächsten an den Restbetrag herankommen. Wie viel noch fehlt wird sich in der Variable z' gemerkt.

Theorem 2.2.1

Der Greedy Algorithmus löst das Wechselgeldproblem optimal. (Was ist wenn die Währung doof ist, siehe nächste Note)

Note:-

Für richtige Währungen wenn man Quatsch Währungen wählt lässt sich schnell ein Gegenbeispiel konstruieren z.B. 1,3,4 jetzt 6 als Betrag. Also wird halt die 4 gewählt weil die am größten ist dementsprechend wird in den nächsten 2 Schritten 2 mal die 1 gewählt, aber die 3 und 3 wären besser gewesen.

Ich mach einfach mal nen Beweis für die Vibes dazu, keine Ahnung gerade Bock drauf steht aber auch so im Skript (in Schöner siehe Seite 29).

Proof. Sei $z \in \mathbb{N}$ ein beliebiger Betrag, welcher genau erreicht werden soll. Für $i \in M := \{1, 2, 5, 10, 20, 50, 100, 200\}$ mit $x_i \in \mathbb{N}_0$ Wegen der Def von Greedy Algorithmen, das immer der Lokal beste Schritt gewählt wird folgt das die Ungleichung mit $i \in M, i > 1$ gilt

$$\sum_{j \in M, j < i} jx_j < i$$

Das j ist hierbei die Münze die gerade betrachtet wird, diese ist immer kleiner als i , da sonst das i gewählt werden würde, was gegen die Definition von Greedy Algorithmen verstößt, da es eine bessere Lösung gibt.

Zusammen mit der Ausgangsbedingung das der zu erreichende Betrag z immer $\sum_{i \in M} ix_i$ ist. Nun lässt sich hierdrüber folgende Induktion aufbauen.

$z = 1$ trivialerweise gilt die Aussage.

Die Aussage gilt für alle $z \in \mathbb{N}$ mit $z > 1$ betrachte

$$\sum_{j \in M, j < i} jx_j < i$$

für ein i maximal $\leq z$ muss mindestens eine Münze vom Wert i enthalten sein, da sonst der Betrag nicht erreicht werden kann.

Die Behauptung folgt auch für $z' = z - i$ da $z' \leq z$ gilt.

Also ist die Lösung vom Greedy Algorithmus mit der Ungleichung erfüllt.

Nun lässt sich annehmen, dass sich eine optimale Lösung finden lässt. Wenn man alle Münzen in M durchgeht lässt sich immer jede Münze durch eine Kombination von Münzen mit kleinerem Wert ersetzen.

Formaler sei $y_i \in \mathbb{N}_0$ mit $\sum_{i \in M} iy_i = z$ und kleinstmöglicher Zahl $\sum i \in My_i$

Sei die Lösung in Optimaler Weise $x_i \in \mathbb{N}_0$ Zum Beispiel gilt $y_1 \leq 1$, da zwei 1 Cent Münzen durch eine 2 Cent Münze ersetzt werden können.

Daraus folgt die Ungleichung $1 \cdot y_1 < 2$

Analog folgt für $y_2, y_2 \leq 2$, da sich 3 2 Cent Münzen durch eine 5 Cent Münze und eine 1 Cent Münze ersetzen lassen.

Daraus folgt die Ungleichung $1 \cdot y_1 + 2 \cdot y_2 < 5$

→ Dies gilt für alle $i \in M$

☺

2.2.1 Optimale Auswahl von Aufgaben