

# Algo Zusammenfassung

Tim Nogga

# Contents

<b>Chapter 1</b>	<b>Einleitung</b>	<b>Page 2</b>
1.1	Insertionsort	2
1.2	Größenordnungen	2
<b>Chapter 2</b>	<b>Methoden zum Entwurf von Algorithmen</b>	<b>Page 4</b>
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 • Rucksackproblem mit Teilbaren Aufgaben — 9	

# Chapter 1

## Einleitung

### 1.1 Insertionsort

---

**Algorithm 1:** Insertion-Sort

---

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

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

---

#### 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

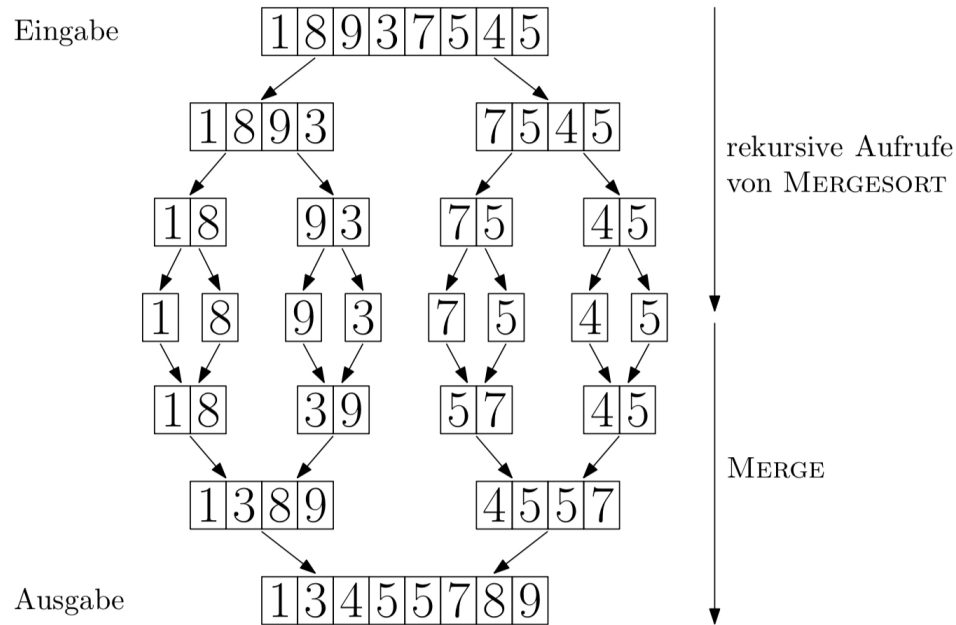


Figure 2.1: Mergesort

### 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  $\log n$  führt. 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  $\Theta(a^h) + \sum_{i=0}^{h-1} f(\frac{n}{b^i})$

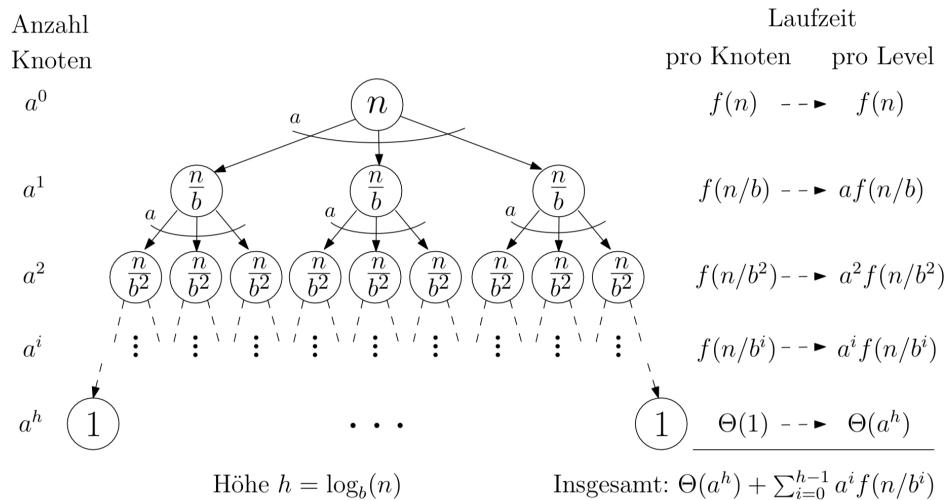


Figure 2.2: Visualized Lemma 1

## 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 Defenition 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, das 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  $\sum_{i \in M} y_i$

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

### Note:-

Intervall Scheduling: Jede Aufgabe hat eine Startzeit  $s_i \geq 0$  und eine Endzeit  $f_i$ , wobei  $s_i < f_i$  gilt.

Zudem steht ein Prozessor zur Verfügung der nur eine Aufgabe gleichzeitig bearbeiten kann.

Die Aufgabe ist es nun eine möglichst große Teilmenge von Aufgaben zu finden, die sich nicht überschneiden.

### Definition 2.2.2: Intervall Scheduling Formales Ziel

Gesucht ist eine Teilmenge  $S' \subseteq S$ , sodass für jedes  $i, j \in S'$  gilt  $i \neq j$  und  $[s_i, f_i) \cap [s_j, f_j) = \emptyset$

### Note:-

Es liegt nahe das es einen Greedy Algorithmus gibt welcher das Problem löst (Im Skript kommen noch Algorithmen die nicht klappen)

### Algorithm 5: Greedy Ende

```

1  $S^* = \emptyset$ 
2 while  $S \neq \emptyset$  do
3   Wähle die Aufgabe  $i$  mit dem frühesten Ende
4    $S^* = S^* \cup \{i\}$ 
5    $S = S \setminus \{i\}$  Keine Ahnung nicht im Skript aber kommt mir logisch vor
6   Entferne alle Aufgaben die sich mit  $i$  überschneiden
```

Für den Beweis von den Algorithmus wird folgendes Lemma benötigt.

### Lemma 2.2.1

Es sei  $S$  eine Menge von Aufgaben und es sei  $i \in S$  eine Aufgabe mit dem frühesten Ende. Dann gibt es eine Optimale Auswahl  $S' \subseteq S$  von paarweise nicht kollidierenden Aufgaben mit  $i \in S'$

#### Note:-

Das Lemma sagt basically nur aus das es eine optimale Lösung gibt.

*Proof.* Genauer Beweis im Skript seite 32. Aber man definiert sich eine Optimale Menge  $S^*$  in diese fügt man dann eine Aufgabe  $i$  ein. Die Menge  $S^*$  ist dann immer noch optimal und es gibt keine Überschneidungen, da  $i$  vor oder gleich mit  $j$  der vorher kürzesten Aufgabe aufhört.  $j$  hat sich halt obviously auch nicht überschneiden, weil dann wär es nicht optimal gewesen. ☺

### Theorem 2.2.2

Der Algorithmus GreedyEnde wählt für jede Instanz eine größtmögliche Menge von paarweise nicht kollidierenden Aufgaben aus.

*Proof.* Lässt sich recht entspannt über ne invariante Zeigen:

In Zeile 2 gilt immer das  $S^*$  sich zu einer optimalen Lösung erweitert.

In der ersten Iteration gilt die Invariante, da  $S^*$  leer ist und  $S$  noch alle Aufgaben enthält. Sei nun die Invariante zu Beginn eines Schleifendurchlaufs erfüllt. Dann gibt es eine optimale Auswahl  $\hat{S} \subseteq S^* \cup S$  von Aufgaben, die alle Aufgaben aus  $S^*$  und gegebenenfalls zusätzliche Aufgaben aus  $S$  enthält. Außerdem kollidieren Aufgaben aus  $S^*$  nicht mit Aufgaben aus  $S$ . Das heißt, bei der Frage, welche Aufgaben aus  $S$  den Aufgaben aus  $S^*$  noch hinzugefügt werden müssen, um eine optimale Auswahl  $\hat{S}$  zu erhalten, spielen die Aufgaben aus  $S^*$  keine Rolle. Die Menge  $\hat{S} \setminus S^*$  dieser Aufgaben ist eine größtmögliche Teilmenge von paarweise nicht kollidierenden Aufgaben aus  $S$ . Gemäß Lemma 2.2.1 existiert eine solche Menge, die die Aufgabe  $i \in S$  mit dem frühesten Fertigstellungszeitpunkt  $f_i$  aller Aufgaben aus  $S$  enthält. Genau diese Aufgabe fügt der Greedy-Algorithmus am Ende der Menge  $S^*$  hinzu. Anschließend werden aus  $S$  alle Aufgaben entfernt, die mit dieser Aufgabe  $i$  kollidieren. Dies garantiert zusammen, dass auch am Anfang der nächsten Iteration die Invariante wieder erfüllt ist. ☺

### Definition 2.2.3: Laufzeit Greedy Ende

Die Laufzeit des Algorithmus GreedyEnde beträgt  $O(n \log n)$ , wobei  $n = |S|$  die Anzahl der Aufgaben in der Eingabe bezeichnet. Sind die Aufgaben bereits aufsteigend nach ihrem Fertigstellungszeitpunkt sortiert, so beträgt die Laufzeit  $O(n)$ .

#### Note:-

Es läuft halt  $n$  mal durch und man kann es in  $O(n \log n)$  sortieren.

## 2.2.2 Rucksackproblem mit Teilbaren Aufgaben

Das Rucksack, wie wir es Behandeln (anscheinend kommt mehr in Algo 2) besteht aus einer Menge von Objekten  $O = \{1, \dots, n\}$ , wobei jedes Objekt  $i \in O$  ein Gewicht  $w_i \in \mathbb{N}$  und einen Nutzen  $p_i \in \mathbb{N}$  besitzt.

Es soll eine Teilmenge der Objekte gefunden werden, die in den Rucksack passt und unter dieser Bedingung maximalen Nutzen besitzt. Jetzt wo es ein bisschen anders ist:

Wir dürfen das Objekt Teilen für den maximalen Nutzen.

---

**Algorithm 6:** Greedy Rucksack

---

```
1 Sortiere die Objekte nach dem Nutzen pro Gewicht  $p_i/w_i$  in absteigender Reihenfolge.
2 for  $int\ i = 1; i \leq n; i++$  do
3    $x[i] = 0;$ 
4  $int\ i = 1;$ 
5 while  $t > 0$  and  $i \leq n$  do
6   if  $t \geq w_i$  then
7      $x_i = 1$ 
8      $t = t - w_i$ 
9   else
10     $x[i] = t/w[i], t = 0$ 
11     $i++;$ 
12 return  $x_1, \dots, x_n$ 
13
```

**Theorem 2.2.3**

Der Algorithmus GreedyRucksack liefert für jede Instanz eine optimale Lösung in  $O(n \log n)$  Zeit.

14

---