

# Zusammenfassung - Kryptographie

Marc Meier

20. Oktober 2015

Korrektheit und Vollständigkeit der Informationen sind nicht gewährleistet. Macht euch eigene Notizen oder ergänzt/korrigiert meine Ausführungen!

## Inhaltsverzeichnis

### 1 Grundlagen und Wiederholungen

1

Planung der Veranstaltung

#### 1. Wiederholung (ca 3-4 Termine)

- Probleme, Kodierung von Problemen, Laufzeit von Algorithmen
- P, PN, NPC...
- Grundlegende Probleme & Algorithmen

#### 2. Algorithmen für schwierige Probleme (ca. 15 Termine)

- parallele / randomisierte Algorithmen
- Approximationsalgorithmen
- parametrisierte Algorithmen

#### 3. Kryptographie (ca 10 Termine)

- Public-Key-Kryptographie

## 1 Grundlagen und Wiederholungen

### 1.1 Probleme, deren Kodierung und Laufzeit von Algorithmen

Anhand einiger Beispiele soll die Wichtigkeit der Kodierung der Probleme bzw. der Eingabe verdeutlicht werden.

#### 1.1.1 Sortieren

**Eingabemenge:** Natürliche Zahlen  $a_1, a_2, \dots, a_n$

**Ausgabe:** Eingabe aufsteigend sortiert

Beim Sortieren handelt es sich um ein *Suchproblem*. Beispiele für bekannte Sortieralgorithmen und deren Laufzeitkomplexität sind Bubblesort  $\mathcal{O}(n^2)$  und Mergesort  $\mathcal{O}(n \log n)$ <sup>1</sup>.

#### 1.1.2 Eingabelänge N

Die Länge der Eingabe entspricht *nicht* der Anzahl der zu sortierenden Elemente  $n$ . Dies ist aufgrund der Kodierung in eine maschinenlesbare Form der Fall. Für gewöhnlich verwenden heutige Computer eine Darstellung im Binärsystem. Daher gilt:

$$N := \sum_{i=1}^n \log a_i \quad (1)$$

---

<sup>1</sup>Sofern nicht anders angegeben entspricht  $\log n$  immer dem dualen Logarithmus von  $n$  (Basis 2)

Im Folgenden soll die Eingabe  $[11, 13, 113]$  kodiert werden. Für eine Turingmaschine mit dem Eingabealphabet  $\Sigma := \{0, 1, \$\}$  würde diese folgendermaßen aussehen:  $1011\$1101\$1110001$ . Um eine Lesbarkeit für Binärrechner zu ermöglichen, werden weiterhin folgende Ersetzungen durchgeführt:  $1 \rightarrow 11$   $0 \rightarrow 00$   $\$ \rightarrow 01$ . Die resultierende Kodierung wäre nun:  $11001111\mathbf{01}11110011\mathbf{01}11111100000011$ . Sei nun  $L$  die maximale Wortlänge, d.h. die größte Binärdarstellung der Eingabezahlen  $a_i$ .

$$L := \max_{i=1}^n \log a_i \quad (2)$$

Daraus folgt für Bubblesort:

$$\begin{aligned} \mathcal{O}(n^2) &= \mathcal{O}(L \cdot n^2) \quad | \text{ Algo. } n\text{-mal durchlaufen mit max. Eingabelänge } L \text{ als obere Abschätzung} \\ &= \mathcal{O}(N \cdot N^2) \quad | L \leq N (\text{offensichtlich wegen } N := \sum_{i=1}^n \log a_i) \\ &= \mathcal{O}(N^3) \quad | n \leq N^2 \end{aligned}$$

### 1.1.3 Primzahl

**Eingabemenge:** Eine natürliche Zahl  $n$

**Ausgabe:** Ist  $n$  eine Primzahl?

Es handelt sich um ein *Entscheidungsproblem*. Die Eingabelänge ist  $N := \log n$ . Ein naiver Algorithmus wird im Folgenden beschrieben.

```

1: if  $n = 2$  then
2:   return Ja
3: else
4:   for  $d := 2$  to  $n - 1$  do
5:     if  $n \bmod d = 0$  then
6:       return Nein
7:     else
8:       end if
9:   end for
10:  return Ja
11: end if

```

Die Komplexität lässt sich wie folgt herleiten: Für die Verzweigungen (Zeilen 1 und 5), sowie für die return-Statements in den Zeilen 2 und 6 ist eine beschränkte Komplexität  $\mathcal{O}(1)$  anzunehmen. Alle Statements in der for-Schleife werden  $n - 2$  mal wiederholt, die Komplexität ist  $\mathcal{O}(n)$ .

Die daraus resultierende Komplexität  $\mathcal{O}(n)$  ist äquivalent zu  $\mathcal{O}(2^{\log n})$ . Weil  $\log n$  der Eingabelänge  $N$  entspricht, ist die Komplexität entsprechend  $\mathcal{O}(2^N)$ , also exponentiell. Demnach ist der Algorithmus nicht effizient.

### 1.1.4 Cliquesuche im Graphen

**CLIQUE** (gehört zu den Entscheidungsproblemen)

**Eingabemenge:** Graph  $G = (V, E)$  und  $k \in \mathbb{N}$

**Ausgabe:** Hat  $G$  mindestens paarweise verbundene Knoten (eine Clique  $C$  mit  $\geq k$  Knoten)?

**CLIQUE (Suchproblem)** (gehört zu den Optimierungsproblemen)

**Eingabemenge:** Graph  $G = (V, E)$

**Ausgabe:** Bestimme eine Clique  $C$  in  $G$  mit maximaler Knotenzahl.

**Behauptung:** Wenn CLIQUE (Entscheidungsproblem) in polynomieller Zeit lösbar ist, so ist auch die Suchversion von CLIQUE in polynomieller Zeit lösbar.<sup>3</sup>

**Beweis**

<sup>3</sup>In polynomieller Zeit, abhängig von der Knotenzahl  $|V|$  des Graphen

$\Leftarrow$  Zur Eingabe  $(G,k)$  von CLIQUE bestimmen wir zunächst mit dem Algorithmus für CLIQUE-Suchversion eine größte Clique  $C$  von  $G$ . Es wird verglichen, ob  $|C| \geq k$  gilt. Ist dies der Fall, gibt der Algorithmus *Ja* zurück, andernfalls *Nein*. Die Laufzeit entspricht offensichtlich der Clique-Suchversion.

$\Rightarrow$  Betrachte folgende Zwischenversion für CLIQUE: Betrachte folgende Zwischenversion für CLIQUE<sup>4</sup>:

**CLIQUE-Zwischenversion:**

**Eingabe:** Graph  $G=(V,E)$

**Aufgabe:** Bestimme die Maximalzahl  $\omega(G)$  von paarweise verbundenen Knoten in  $G$

**Behauptung:** CLIQUE in polynomieller Zeit lösbar  $\Rightarrow^{(1)}$  CLIQUE-Zwischenversion in polynomieller Zeit  $\Rightarrow^{(2)}$  CLIQUE-Suchversion in polynomieller Zeit lösbar.

**zu (1):** Sei  $A$  ein Algorithmus für CLIQUE in polynomieller Zeit. Zur Eingabe  $G$  von CLIQUE-Zwischenversion wende  $A$  auf Eingaben  $(G,n), (G,n-1), \dots, (G,1)$  an! Ausgabe ist  $\omega(G) =$  erstes  $k$  mit  $A(G,k) = Ja$

```

1: for  $k := n$  to 1 do
2:   if  $A(G,k) = \text{'ja'}$  then return  $\omega(G) = k$ 
3:   end if
4: end for

```

Laufzeit:  $n \cdot \mathcal{O}(n^c) = \mathcal{O}(n^{c+1})$

**zu (2):** : Sei  $A$  ein polyzeit Algorithmus für Clique-Zwischenversion. d.h.  $A(G) = \omega(G)$  in  $\mathcal{O}(n^d)$  für eine Konstante  $d$ .

Zur Eingabe  $G$  von Clique-Suchversion wende folgendes an:

```

1: for jede Kante  $e \in E$  do
2:   if  $\omega(G - e) = \omega(G)$  then  $G := G - e$ 
3:   end if
4: end for
5: return  $C := V(G)$  ohne isolierte Knoten

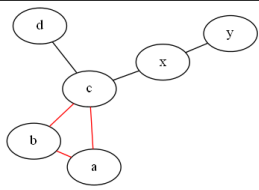
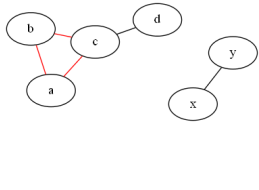
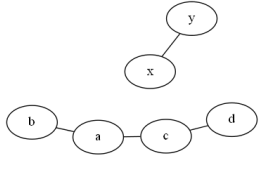
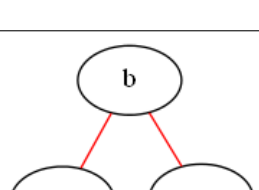
```

**Hinweis:** Ich bin mir nicht sicher, ob es sich um das korrekte Beispiel aus der Vorlesung handelt.

**Laufzeit:**  $m \cdot \mathcal{O}(n^d) = \mathcal{O}(n^{d+2})$

---

<sup>4</sup> $\omega(G) = \max\{|C| : C \text{ ist eine Clique in } G\}$  ist die Cliquenzahl

$G$	$\omega(G) = 3$		Vollständiger Graph
$G_2 := G - cx$	$\omega(G) = 3$		Nach Entfernen der Kante $cx$ hat sich die Cliquenzahl nicht geändert. Die Kante $cx$ ist also nicht für die Clique notwendig.
$G_3 := G_2 - bc$	$\omega(G) = 2$		Nach Entfernen der Kante $bc$ hat sich die Cliquenzahl verringert. Offensichtlich wird $bc$ also für die Clique benötigt.
$C := \{a, b, c\}$	$\omega(G) = 3$		Im weiteren Verlauf wird (offensichtlich) noch die Kante $cd$ entfernt. Anschließend werden die isolierten Knoten $d$ , $x$ und $y$ entfernt und die Lösung $C := \{a, b, c\}$ zurückgegeben.

### 1.1.5 Gegenüberstellung „einfache Formulierung“ vs „formale Formulierung“

Die genaue Formulierung des Problems ist wichtig:

„Ist  $n$  prim?“

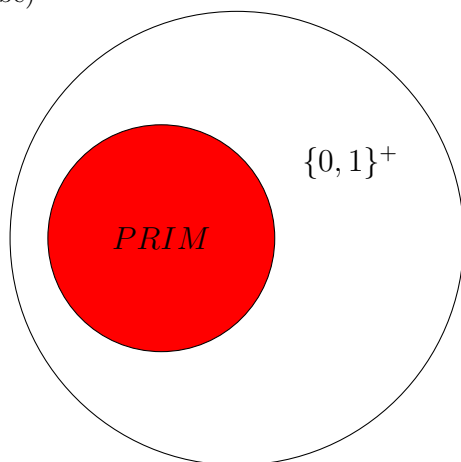
genauer:

**Eingabemenge:** Eine natürliche Zahl  $n$

**Ausgabe:** Ist  $n$  eine Primzahl?

**Formal:**  $PRIM^5 = \{bin(n) | n \in \mathbb{N} \text{ ist prim}\} \subset \{0, 1\}^+$

Venn-Diagramm  $PRIM \subset \{0, 1\}^+$  TODO (Und offensichtlich falsch, da ich noch keinen Plan von TIKZ habe)




---

<sup>5</sup> $bin(n)$  - Binärdarstellung von  $n$

## 1.2 $\mathbb{P}$ , NP, NP-vollständig

$\mathbb{P}$  = Menge aller *Entscheidungsprobleme*, für die ein (deterministischer) polynomialzeit-Algorithmus existiert (formal via Deterministischer Turing Maschine, „effizient lösbare Probleme“). Ein *Optimierungsproblem* ist lösbar, wenn die entsprechende Entscheidungsversion in  $\mathbb{P}$  ist.

### 1.2.1 Grundlegende Probleme in $\mathbb{P}$

- **Sortieren**
- **PRIM** - Allerdings nicht der vorgestellte Algorithmus. Effizienter Algorithmus wurde 2002 vorgestellt. Komplexität  $\mathcal{O}(\log n)^{12}$ , zur Zeit sogar  $\mathcal{O}(\log n)^6$  <https://de.wikipedia.org/wiki/AKS-Primzahltest>
- **2-SAT** (Modellierung mit Graphenproblem für Beweis)

#### Übrige Notizen

In der Praxis ermittelt man die Komplexität der Einfachheit halber mit uniformen Kostenmaß. Sortierung: Eingabe:  $a_1, a_2, \dots, a_n$  beziehungsweise  $\text{bin}(a_1) \# \text{bin}(a_2) \# \dots \# \text{bin}(a_n)$  mit Ersetzung Aufgabe: Sortiere Zahlen aufsteigend Mergesort:  $\mathcal{O}(n \cdot \log n)$  Problem -i Eingabelänge (siehe oben, hatten wir schon)