Vorlesungsskript

Theoretische Informatik III

Sommersemester 2010

Prof. Dr. Johannes Köbler
Humboldt-Universität zu Berlin
Lehrstuhl Komplexität und Kryptografie

22. April 2010

Inhaltsverzeichnis

1 Einleitung

In den Vorlesungen ThI 1 und ThI 2 standen folgende Themen im Vordergrund:

- Mathematische Grundlagen der Informatik, Beweise führen, Modellierung Aussagenlogik, Prädikatenlogik
- Welche Probleme sind lösbar? (Berechenbarkeitstheorie)
- Welche Rechenmodelle sind adäquat? (Automatentheorie)
- Welcher Aufwand ist nötig? (Komplexitätstheorie)

Dagegen geht es in der VL ThI 3 in erster Linie um folgende Frage:

- Wie lassen sich eine Reihe von praktisch relevanten Problemstellungen möglichst effizient lösen?
- Wie lässt sich die Korrektheit von Algorithmen beweisen und wie lässt sich ihre Laufzeit abschätzen?

Die Untersuchung dieser Fragen lässt sich unter dem Themengebiet Algorithmik zusammenfassen.

Der Begriff *Algorithmus* geht auf den persischen Gelehrten Muhammed Al Chwarizmi (8./9. Jhd.) zurück. Der älteste bekannte nicht-triviale Algorithmus ist der nach *Euklid* benannte Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen (300 v. Chr.). Von einem Algorithmus wird erwartet, dass er jede *Problemeingabe* nach endlich vielen Rechenschritten löst (etwa durch Produktion einer Ausgabe). Ein Algorithmus ist ein „Verfahren“ zur Lösung eines Entscheidungs- oder Berechnungsproblems, das sich prinzipiell auf einer Turingmaschine implementieren lässt (Church-Turing-These).

Die Registermaschine

Bei Aussagen zur Laufzeit von Algorithmen beziehen wir uns auf die Registermaschine (engl. random access machine; RAM). Dieses Modell ist etwas flexibler als die Turingmaschine, da es den unmittelbaren Lese- und Schreibzugriff (random access) auf eine beliebige Speichereinheit (Register) erlaubt. Als Speicher stehen beliebig viele Register zur Verfügung, die jeweils eine beliebig große natürliche Zahl speichern können. Auf den Registerinhalten sind folgende arithmetische Operationen in einem Rechenschritt ausführbar: Addition, Subtraktion, abgerundetes Halbieren und Verdoppeln. Unabhängig davon geben wir die Algorithmen in Pseudocode an. Das RAM-Modell benutzen wir nur zur Komplexitätsabschätzung.

Die Laufzeit von RAM-Programmen wird wie bei TMs in der Länge der Eingabe gemessen. Man beachte, dass bei arithmetischen Problemen (wie etwa Multiplikation, Division, Primzahltests, etc.) die Länge einer Zahleingabe n durch die Anzahl $\lceil \log n \rceil$ der für die Binärkodierung von n benötigten Bits gemessen wird. Dagegen bestimmt bei nicht-arithmetischen Problemen (z.B. Graphalgorithmen oder Sortierproblemen) die Anzahl der gegebenen Zahlen die Länge der Eingabe.

Asymptotische Laufzeit und Landau-Notation

Definition 1. *Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{R}^+$. Wir schreiben $f(n) = \mathcal{O}(g(n))$, falls es Zahlen n_0 und c gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Die Bedeutung der Aussage $f(n) = \mathcal{O}(g(n))$ ist, dass f „nicht wesentlich schneller“ als g wächst. Formal bezeichnet der Term $\mathcal{O}(g(n))$ die Klasse aller Funktionen f, die obige Bedingung erfüllen. Die Gleichung $f(n) = \mathcal{O}(g(n))$ drückt also in Wahrheit eine Element-Beziehung $f \in \mathcal{O}(g(n))$ aus. O-Terme können auch auf

der linken Seite vorkommen. In diesem Fall wird eine Inklusionsbeziehung ausgedrückt. So steht $n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$ für die Aussage $\{n^2 + f \mid f \in \mathcal{O}(n)\} \subseteq \mathcal{O}(n^2)$.

Beispiel 2.

- $7\log(n) + n^3 = \mathcal{O}(n^3)$ *ist richtig.*
- $7\log(n)n^3 = \mathcal{O}(n^3)$ *ist falsch.*
- $2^{n+\mathcal{O}(1)} = \mathcal{O}(2^n)$ *ist richtig.*
- $2^{\mathcal{O}(n)} = \mathcal{O}(2^n)$ *ist falsch (siehe Übungen).*

◁

Es gibt noch eine Reihe weiterer nützlicher Größenvergleiche von Funktionen.

Definition 3. *Wir schreiben $f(n) = o(g(n))$, falls es für jedes $c > 0$ eine Zahl n_0 gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Damit wird ausgedrückt, dass f „wesentlich langsamer“ als g wächst. Außerdem schreiben wir

- $f(n) = \Omega(g(n))$ *für $g(n) = \mathcal{O}(f(n))$, d.h. f wächst mindestens so schnell wie g)*
- $f(n) = \omega(g(n))$ *für $g(n) = o(f(n))$, d.h. f wächst wesentlich schneller als g, und*
- $f(n) = \Theta(g(n))$ *für $f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$, d.h. f und g wachsen ungefähr gleich schnell.*

2 Suchen und Sortieren

2.1 Suchen von Mustern in Texten

In diesem Abschnitt betrachten wir folgende algorithmische Problemstellung.

String-Matching (StringMatching):

Gegeben: Ein Text $x = x_1 \cdots x_n$ und ein Muster $y = y_1 \cdots y_m$ über einem Alphabet Σ.

Gesucht: Alle Vorkommen von y in x.

Wir sagen *y kommt in x an Stelle i vor*, falls $x_{i+1} \cdots x_{i+m} = y$ ist. Typische Anwendungen finden sich in Textverarbeitungssystemen (emacs, grep, etc.), sowie bei der DNS- bzw. DNA-Sequenzanalyse.

Beispiel 4. *Sei $\Sigma = \{A, C, G, U\}$.*

Text x = AUGACGAUGAUGUAGGUAGCGUAGAUGAUGUAG,
Muster y = AUGAUGUAG.

Das Muster y kommt im Text x an den Stellen 6 und 24 vor. ◁

Bei naiver Herangehensweise kommt man sofort auf folgenden Algorithmus.

Algorithmus `naive-String-Matcher`(x, y)

```
1  Input: Text x = x_1 ··· x_n und Muster y = y_1 ··· y_m
2    V := ∅
3    for i := 0 to n − m do
```

4 **if** $x_{i+1} \cdots x_{i+m} = y_1 \cdots y_m$ **then**
5 $V := V \cup \{i\}$
6 **Output:** V

Die Korrektheit von `naive-String-Matcher` ergibt sich wie folgt:

- In der **for**-Schleife testet der Algorithmus alle potentiellen Stellen, an denen y in x vorkommen kann, und
- fügt in Zeile 4 genau die Stellen i zu V hinzu, für die $x_{i+1} \cdots x_{i+m} = y$ ist.

Die Laufzeit von `naive-String-Matcher` lässt sich nun durch folgende Überlegungen abschätzen:

- Die **for**-Schleife wird $(n - m + 1)$-mal durchlaufen.
- Der Test in Zeile 4 benötigt maximal m Vergleiche.

Dies führt auf eine Laufzeit von $\mathcal{O}(nm) = \mathcal{O}(n^2)$. Für Eingaben der Form $x = a^n$ und $y = a^{\lfloor n/2 \rfloor}$ ist die Laufzeit tatsächlich $\Theta(n^2)$.

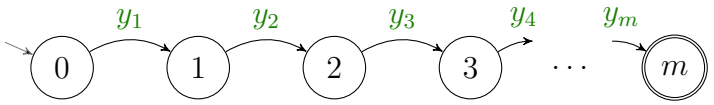
2.1.1 String-Matching mit endlichen Automaten

Durch die Verwendung eines endlichen Automaten lässt sich eine erhebliche Effizienzsteigerung erreichen. Hierzu konstruieren wir einen DFA M_y, der jedes Vorkommen von y in der Eingabe x durch Erreichen eines Endzustands anzeigt. M_y erkennt also die Sprache

$$L = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}.$$

Konkret konstruieren wir $M_y = (Z, \Sigma, \delta, 0, m)$ wie folgt:

- M_y hat $m + 1$ Zustände, die den $m + 1$ Präfixen $y_1 \cdots y_k$, $k = 0, \ldots, m$, von y entsprechen, d.h. $Z = \{0, \ldots, m\}$.
- Liest M_y im Zustand k das Zeichen y_{k+1}, so wechselt M_y in den Zustand $k + 1$, d.h. $\delta(k, y_{k+1}) = k + 1$ für $k = 0, \ldots, m - 1$:



- Falls das nächste Zeichen a nicht mit y_{k+1} übereinstimmt (engl. *mismatch*), wechselt M_y in den Zustand

$$\delta(k, a) = \max\{j \leq m \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}.$$

Der DFA M_y speichert also in seinem Zustand die maximale Länge k eines Präfixes $y_1 \cdots y_k$ von y, das zugleich ein Suffix der gelesenen Eingabe ist:

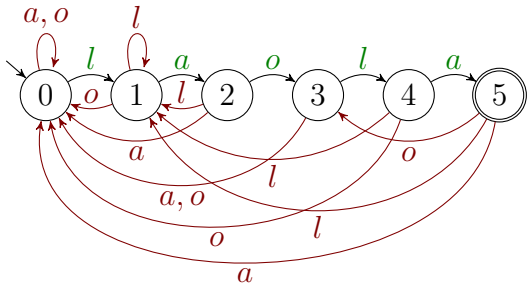
$$\hat{\delta}(0, x) = \max\{k \leq m \mid y_1 \cdots y_k \text{ ist Suffix von } x\}.$$

Die Korrektheit von M_y folgt aus der Beobachtung, dass M_y isomorph zum *Äquivalenzklassenautomaten* M_{R_L} für L ist. M_{R_L} hat die Zustände $[y_1 \cdots y_k]$, $k = 0, \ldots, m$, von denen nur $[y_1 \cdots y_m]$ ein Endzustand ist. Die Überführungsfunktion ist definiert durch

$$\delta([y_1 \cdots y_k], a) = [y_1 \cdots y_j],$$

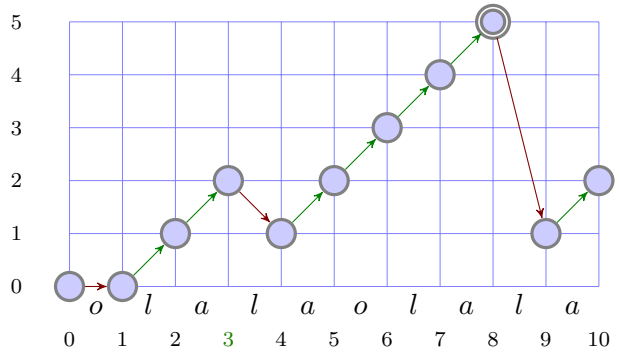
wobei $y_1 \cdots y_j$ das längste Präfix von $y = y_1 \cdots y_m$ ist, welches Suffix von $y_1 \cdots y_j a$ ist (siehe Übungen).

Beispiel 5. *Für das Muster $y = laola$ hat M_y folgende Gestalt:*



δ	0	1	2	3	4	5
a	0	2	0	0	5	0
l	1	1	1	4	1	1
o	0	0	3	0	0	3

M_y macht bei der Suche nach dem Muster $y = laola$ im Text $x = olalaolala$ folgende Übergänge:



◁

Insgesamt erhalten wir somit folgenden Algorithmus.

Algorithmus `DFA-String-Matcher`(x, y)

```
1  Input: Text x = x_1 ··· x_n und Muster y = y_1 ··· y_m
2    konstruiere den DFA M_y = (Z, Σ, δ, 0, m)
3    V := ∅
4    k := 0
5    for i := 1 to n do
6      k := δ(k, x_i)
7      if k = m then V := V ∪ {i − m}
8  Output: V
```

Die Korrektheit von `DFA-String-Matcher` ergibt sich unmittelbar aus der Tatsache, dass M_y die Sprache

$$L(M_y) = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}$$

erkennt. Folglich fügt der Algorithmus genau die Stellen $j = i - m$ zu V hinzu, für die y ein Suffix von $x_1 \cdots x_i$ (also $x_{j+1} \cdots x_{j+m} = y$) ist.

Die Laufzeit von `DFA-String-Matcher` ist die Summe der Laufzeiten für die Konstruktion von M_y und für die Simulation von M_y bei Eingabe x, wobei letztere durch $\mathcal{O}(n)$ beschränkt ist. Für δ ist eine Tabelle mit $(m+1)\|\Sigma\|$ Einträgen

$$\delta(k, a) = \max\{j \leq k+1 \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}$$

zu berechnen. Jeder Eintrag $\delta(k, a)$ ist in Zeit $\mathcal{O}(k^2) = \mathcal{O}(m^2)$ berechenbar. Dies führt auf eine Laufzeit von $\mathcal{O}(\|\Sigma\|m^3)$ für die Konstruktion von M_y und somit auf eine Gesamtlaufzeit von $\mathcal{O}(\|\Sigma\|m^3 + n)$. Tatsächlich lässt sich M_y sogar in Zeit $\mathcal{O}(\|\Sigma\|m)$ konstruieren.

2.1.2 Der Knuth-Morris-Pratt-Algorithmus

Durch eine Modifikation des Rücksprungmechanismus' lässt sich die Laufzeit von `DFA-String-Matcher` auf $\mathcal{O}(n+m)$ verbessern. Hierzu vergegenwärtigen wir uns folgende Punkte:

- Tritt im Zustand k ein Mismatch $a \neq y_{k+1}$ auf, so ermittelt M_y das längste Präfix p von $y_1 \cdots y_k$, das zugleich Suffix von $y_1 \cdots y_k a$ ist, und springt in den Zustand $j = \delta(k, a) = |p|$.
- Im Fall $j > 0$ hat p also die Form $p = p'a$, wobei $p' = y_1 \cdots y_{j-1}$ sowohl echtes Präfix als auch echtes Suffix von $y_1 \cdots y_k$ ist. Zudem gilt $y_j = a$.
- Die Idee beim KMP-Algorithmus ist nun, bei einem Mismatch unabhängig von a auf das nächst kleinere Präfix $\tilde{p} = y_1 \cdots y_i$ von $y_1 \cdots y_k$ zu springen, das auch Suffix von $y_1 \cdots y_k$ ist.
- Stimmt nach diesem Rücksprung das nächste Eingabezeichen a mit y_{i+1} überein, so wird dieses gelesen und der KMP-Algorithmus erreicht (nach einem kleinen Umweg über den Zustand i) den Zustand $i + 1 = j$, in den auch M_y wechselt.
- Andernfalls springt der KMP-Algorithmus nach derselben Methode solange weiter zurück, bis das nächste Eingabezeichen a „passt“

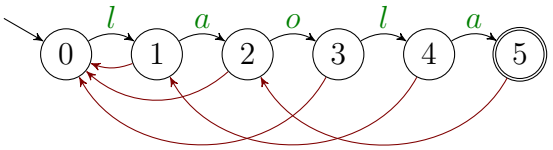
(also $y_{i+1} = a$ und somit $\tilde{p}a$ ein Präfix von y ist) oder der Zustand 0 erreicht wird.

- In beiden Fällen wird a gelesen und der Zustand $\delta(k, a)$ angenommen.

Der KMP-Algorithmus besucht also alle Zustände, die auch M_y besucht, führt aber die Rücksprünge in mehreren Etappen aus. Die Sprungadressen werden durch die so genannte *Präfixfunktion* $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ ermittelt:

$$\pi(k) = \max\{0 \le j \le k-1 \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k\}.$$

Beispiel 6. *Für das Muster $y = laola$ ergibt sich folgende Präfixfunktion π:*

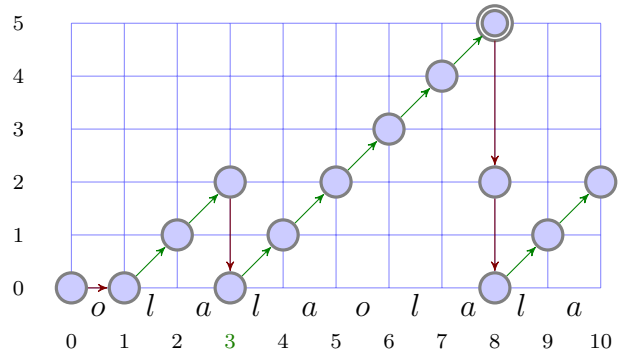


k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

Wir können uns die Arbeitsweise dieses Automaten wie folgt vorstellen:

1. *Erlaubt das nächste Eingabezeichen einen Übergang vom aktuellen Zustand k nach $k+1$, so führe diesen aus.*
2. *Ist ein Übergang nach $k+1$ nicht möglich und $k \ge 1$, so springe in den Zustand $\pi(k)$ ohne das nächste Zeichen zu lesen.*
3. *Andernfalls (d.h. $k = 0$ und ein Übergang nach 1 ist nicht möglich) lies das nächste Zeichen und bleibe im Zustand 0.*

Der KMP-Algorithmus macht bei der Suche nach dem Muster $y = laola$ im Text $x = olalaolala$ folgende Übergänge:



◁

Auf die Frage, wie sich die Präfixfunktion π möglichst effizient berechnen lässt, werden wir später zu sprechen kommen. Wir betrachten zunächst das Kernstück des KMP-Algorithmus, das sich durch eine leichte Modifikation von `DFA-String-Matcher` ergibt.

`DFA-String-Matcher`(x, y)

```
1  Input: Text x_1 ··· x_n
       und Muster y_1 ··· y_m
2   konstruiere M_y
3   V := ∅
4   k := 0
5   for i := 1 to n do
6     k := δ(k, x_i)
7     if k = m then
8       V := V ∪ {i − m}
9  Output: V
```

`KMP-String-Matcher`(x, y)

```
1  Input: Text x_1 ··· x_n und
       Muster y_1 ··· y_m
2   π := KMP-Prefix(y)
3   V := ∅
4   k := 0
5   for i := 1 to n do
6     while (k > 0 ∧ x_i ≠ y_{k+1}) do
7       k := π(k)
8     if x_i = y_{k+1} then k := k + 1
9     if k = m then
10      V := V ∪ {i − m}, k := π(k)
11 Output: V
```

Die Korrektheit des Algorithmus `KMP-String-Matcher` ergibt sich einfach daraus, dass er den Zustand m an genau den gleichen Textstellen besucht wie `DFA-String-Matcher`, und somit wie dieser alle Vorkommen von y im Text x findet.

Für die Laufzeitanalyse von `KMP-String-Matcher` (ohne die Berechnung von `KMP-Prefix`) stellen wir folgende Überlegungen an.

- Die Laufzeit ist proportional zur Anzahl der Zustandsübergänge.
- Bei jedem Schritt wird der Zustand um maximal Eins erhöht.
- Daher kann der Zustand nicht öfter verkleinert werden als er erhöht wird (*Amortisationsanalyse*).
- Es gibt genau n Zustandsübergänge, bei denen der Zustand erhöht wird bzw. unverändert bleibt.
- Insgesamt finden also höchstens $2n = \mathcal{O}(n)$ Zustandsübergänge statt.

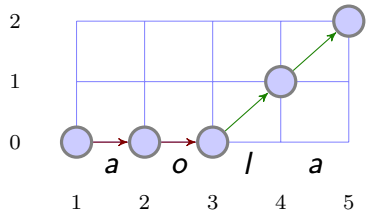
Nun kommen wir auf die Frage zurück, wie sich die Präfixfunktion π effizient berechnen lässt. Die Aufgabe besteht darin, für jedes Präfix $y_1 \cdots y_i$, $i \geq 1$, das längste echte Präfix zu berechnen, das zugleich Suffix von $y_1 \cdots y_i$ ist.

Die Idee besteht nun darin, mit dem KMP-Algorithmus das Muster y im Text $y_2 \cdots y_m$ zu suchen. Dann liefert der beim Lesen von y_i erreichte Zustand k gerade das längste Präfix $y_1 \cdots y_k$, das zugleich Suffix von $y_2 \cdots y_i$ ist (d.h. es gilt $\pi(i) = k$). Zudem werden bis zum Lesen von y_i nur Zustände kleiner als i erreicht. Daher sind die π-Werte für alle bis dahin auszuführenden Rücksprünge bereits bekannt und π kann in Zeit $\mathcal{O}(m)$ berechnet werden.

Prozedur `KMP-Prefix`(y)

```
1  π(1) := 0
2  k := 0
3  for i := 2 to m do
4     while (k > 0 ∧ y_i ≠ y_{k+1}) do k := π(k)
5     if y_i = y_{k+1} then k := k + 1
6     π(i) := k
7  return(π)
```

Beispiel 7. *Die Verarbeitung des Musters* $y =$ *laola durch* `KMP-Prefix` *ergibt folgendes Ablaufprotokoll:*



k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

◁

Wir fassen die Laufzeiten der in diesem Abschnitt betrachteten String-Matching Algorithmen in einer Tabelle zusammen:

Algorithmus	Vorverarbeitung	Suche	Gesamtlaufzeit
naiv	0	$\mathcal{O}(nm)$	$\mathcal{O}(nm)$
DFA (einfach)	$\mathcal{O}(\|\Sigma\|m^3)$	$\mathcal{O}(n)$	$\mathcal{O}(\|\Sigma\|m^3 + n)$
DFA (verbessert)	$\mathcal{O}(\|\Sigma\|m)$	$\mathcal{O}(n)$	$\mathcal{O}(\|\Sigma\|m + n)$
Knuth-Morris-Pratt	$\mathcal{O}(m)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.2 Durchsuchen von Zahlenfolgen

Als nächstes betrachten wir folgendes Suchproblem.

Element-Suche

Gegeben: Eine Folge $a_1, \ldots, a_n$ von natürlichen Zahlen und eine Zahl a.

Gesucht: Ein Index i mit $a_i = a$ (bzw. eine Fehlermeldung, falls $a \notin \{a_1, \ldots, a_n\}$ ist).

Typische Anwendungen finden sich bei der Verwaltung von Datensätzen, wobei jeder Datensatz über einen eindeutigen Schlüssel (z.B. *Matrikelnummer*) zugreifbar ist. Bei manchen Anwendungen können die Zahlen in der Folge auch mehrfach vorkommen. Gesucht sind dann

evtl. alle Indizes i mit $a_i = a$. Durch eine sequentielle Suche lässt sich das Problem in Zeit $\mathcal{O}(n)$ lösen.

Algorithmus `Sequential-Search`

```
1  Input: Eine Zahlenfolge a_1, ..., a_n und eine Zahl a
2    i := 0
3    repeat
4      i := i + 1
5    until (i = n ∨ a = a_i)
6  Output: i, falls a_i = a bzw. Fehlermeldung, falls
       a_i ≠ a
```

Falls die Folge $a_1, \ldots, a_n$ sortiert ist, d.h. es gilt $a_i \leq a_j$ für $i \leq j$, bietet sich eine *Binärsuche* an.

Algorithmus `Binary-Search`

```
1  Input: Eine Zahlenfolge a_1, ..., a_n und eine Zahl a
2    l := 1
3    r := n
4    while l < r do
5      m := ⌊(l + r)/2⌋
6      if a ≤ a_m then r := m else l := m + 1
7  Output: l, falls a_l = a bzw. Fehlermeldung, falls
       a_l ≠ a
```

Offensichtlich gibt der Algorithmus im Fall $a \notin \{a_1, \ldots, a_n\}$ eine Fehlermeldung aus. Im Fall $a \in \{a_1, \ldots, a_n\}$ gilt die *Schleifeninvariante* $a_l \leq a \leq a_r$. Daher muss nach Abbruch der **while**-Schleife $a = a_l$ sein. Dies zeigt die Korrektheit von `Binary-Search`.

Da zudem die Länge $l - r + 1$ des Suchintervalls $[l, r]$ in jedem Schleifendurchlauf mindestens auf $\lfloor (l - r)/2 \rfloor + 1$ reduziert wird, werden höchstens $\lceil \log n \rceil$ Schleifendurchläufe ausgeführt. Folglich ist die Laufzeit von `Binary-Search` höchstens $\mathcal{O}(\log n)$.