Vorlesungsskript

Theoretische Informatik III

Sommersemester 2008

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

2. Mai 2008

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der VL ThI 2 standen folgende Themen im Vordergrund:

- 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? (Algorithmik)

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 (Churchsche These bzw. 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.

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 $f(n) = O(g(n))$, 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 $f(n) = O(g(n))$ ist, dass f „nicht wesentlich schneller“ als g wächst. Formal bezeichnet der Term $O(g(n))$ die Klasse aller Funktionen f, die obige Bedingung erfüllen. Die Gleichung $f(n) = O(g(n))$ drückt also in Wahrheit eine Element-Beziehung $f \in O(g(n))$ aus. O-Terme können auch auf der linken Seite vorkommen. In diesem Fall wird eine Inklusionsbeziehung ausgedrückt. So steht $n^2 + O(n) = O(n^2)$ für die Aussage $\{n^2 + f \mid f \in O(n)\} \subseteq O(n^2)$.

Beispiel 2

- $7\log(n) + n^3 = O(n^3)$ ist richtig.
- $7\log(n)n^3 = O(n^3)$ ist falsch.
- $2^{n+O(1)} = O(2^n)$ ist richtig.
- $2^{O(n)} = O(2^n)$ 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 $g(n) = O(f(n))$, 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 $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$, d.h. f und g wachsen ungefähr gleich schnell.

Kapitel 2

Suchprobleme

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-Sequenz- analyse.

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 5 NAIV-STRING-MATCHER

1 **Eingabe:** Text $x = x_1 \cdots x_n$ und Muster $y = y_1 \cdots y_m$
2 $V \leftarrow \emptyset$
3 **for** $i := 0$ **to** $n - m$ **do**
4 **if** $x_{i+1} \cdots x_{i+m} = y_1 \cdots y_m$ **then** $V \leftarrow V \cup \{i\}$ **end**
5 **Ausgabe:** V

Die Korrektheit von `naiv-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 `naiv-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 $O(nm) = O(n^2)$. Für Eingaben der Form $x = a^n$ und $y = a^{\lfloor n/2 \rfloor}$ ist die Laufzeit tatsächlich $\Theta(n^2)$.

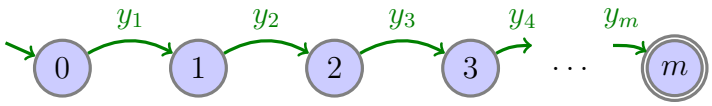
2.2 String-Matching mit endlichen Automaten

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 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.
- 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

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

M_y speichert also in seinem Zustand die maximale Präfixlänge k, für die $y_1 \cdots y_k$ ein Suffix der gelesenen Eingabe ist:

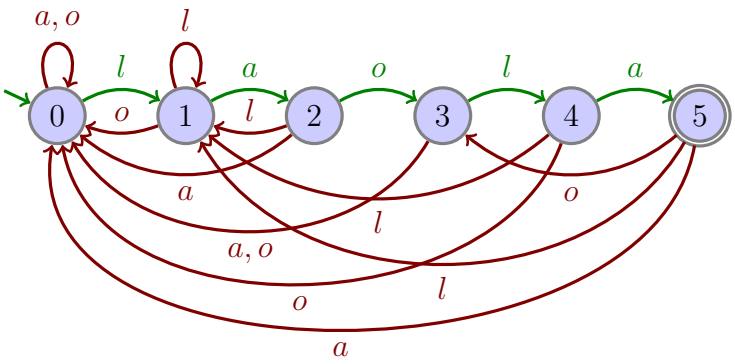
$$\hat{\delta}(0, x) = \max\{k \leq m \mid y_1 \cdots y_k \text{ ist Suffix von } x\}.$$

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

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

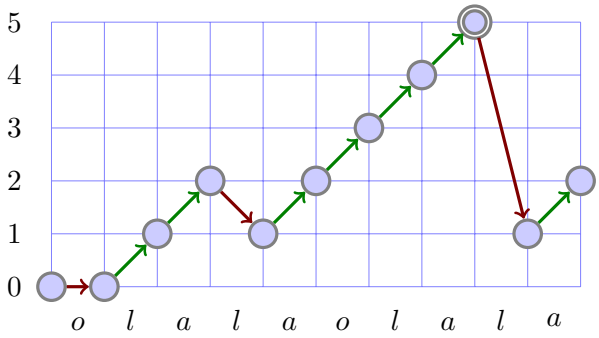
wobei $y_1 \cdots y_j$ das längste Präfix von $y = y_1 \cdots y_m$ ist, welches Suffix von $y_1 \cdots y_j a$ ist (siehe Übungen).

Beispiel 6 Für das Muster $y = laola$ ergibt sich folgender DFA M_y:



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 7 DFA-STRING-MATCHER

1 **Eingabe:** Text $x = x_1 \cdots x_n$ und Muster $y = y_1 \cdots y_m$
2 konstruiere den DFA $M_y = (Z, \Sigma, \delta, 0, m)$ mit $Z = \{0, \ldots, m\}$
3 $V \leftarrow \emptyset$

4 $k \leftarrow 0$
5 **for** $i := 1$ **to** n **do**
6 $k \leftarrow \delta(k, x_i)$
7 **if** $k = m$ **then** $V \leftarrow V \cup \{i - m\}$ **end**
8 **Ausgabe:** 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 $O(n)$ 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 $O(k^2) = O(m^2)$ berechenbar. Dies führt auf eine Laufzeit von $O(\|\Sigma\|m^3)$ für die Konstruktion von M_y und somit auf eine Gesamtlaufzeit von $O(\|\Sigma\|m^3 + n)$. Tatsächlich lässt sich M_y sogar in Zeit $O(\|\Sigma\|m)$ konstruieren.

2.3 Der Knuth-Morris-Pratt-Algorithmus

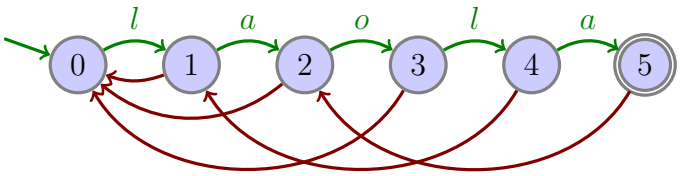
Durch eine Modifikation des Rücksprungmechanismus' lässt sich die Laufzeit von `DFA-String-Matcher` auf $O(n+m)$ verbessern. Hierz vergegenwärtigen wir uns folgende Punkte:

- Tritt im Zustand k ein Mismatch $a \neq y_{k+1}$ auf, so ermittelt M_y das längste Präfix p von $y_1 \cdots y_k$ ist, das zugleich Suffix von $y_1 \cdots y_k a$ ist, und springt in den Zustand $k' = |p|$.
- Im Fall $k' \neq 0$ hat p also die Form $p = p'a$, wobei p' sowohl echtes Präfix als auch echtes Suffix von $y_1 \cdots y_k$ ist.
- Die Idee beim KMP-Algorithmus ist nun, unabhängig von a auf das nächst kleinere Präfix $\tilde{p}$ von $y_1 \cdots y_k$ zu springen, das auch Suffix von $y_1 \cdots y_k$ ist.
- Dies wird solange wiederholt, bis das Zeichen a „passt“ (also $\tilde{p}a$ Präfix von y ist) oder der Zustand 0 erreicht wird.

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:

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

Beispiel 8 Für das Muster $y = laola$ ergibt sich folgende Präfixfunktion π:

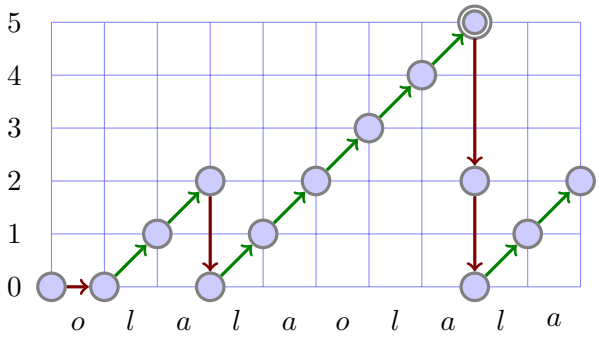


k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

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 $k \geq 1$, 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.

Algorithmus 9 KMP-STRING-MATCHER

1 **Eingabe:** Text $x = x_1 \cdots x_n$ und Muster $y = y_1 \cdots y_m$

2 $\pi \leftarrow$ `KMP-Prefix`(y)
3 $V \leftarrow \emptyset$
4 $k \leftarrow 0$
5 **for** $i := 1$ **to** n **do**
6 **while** $(k > 0 \wedge x_i \neq y_{k+1})$ **do** $k \leftarrow \pi(k)$ **end**
7 **if** $x_i = y_{k+1}$ **then** $k \leftarrow k + 1$ **end**
8 **if** $k = m$ **then** $V \leftarrow V \cup \{i - m\}$; $k \leftarrow \pi(k)$ **end**
9 **Ausgabe:** V

Die Korrektheit von `KMP-String-Matcher` ergibt sich einfach daraus, dass `KMP-String-Matcher` 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 $2n = O(n)$ 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 $O(m)$ berechnet werden.

Algorithmus 10 `KMP-Prefix`

1 **Eingabe:** Muster $y = y_1 \cdots y_m$
2 $\pi(1) \leftarrow 0$
3 $k \leftarrow 0$
4 **for** $i := 2$ **to** m **do**

5 **while** $(k > 0 \wedge y_i \neq y_{k+1})$ **do** $k \leftarrow \pi(k)$ **end**
6 **if** $y_i = y_{k+1}$ **then** $k \leftarrow k + 1$ **end**
7 $\pi(i) \leftarrow k$
8 **Ausgabe:** π

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

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

2.4 Durchsuchen von Mengen

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 (in diesem Fall ist $\{\{a_1, \ldots, a_n\}\}$ eine *Multimenge*). Gesucht sind dann evtl. alle Indizes i mit $a_i = a$.
Durch eine sequentielle Suche lässt sich das Problem in Zeit $O(n)$ lösen.

Algorithmus 11 SEQUENTIAL-SEARCH

1 **Eingabe:** Eine Zahlenfolge $a_1, \ldots, a_n$ und eine Zahl a
2 $i \leftarrow 0$
3 **repeat**
4 $i \leftarrow i + 1$
5 **until** $(i = n \vee a = a_i)$
6 **Ausgabe:** i falls $a_i = a$ bzw. Fehlermeldung, falls $a_i \neq a$

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 12 BINARY-SEARCH

Eingabe: Eine Zahlenfolge $a_1, \ldots, a_n$ und eine Zahl a
  $l \leftarrow 1$
  $r \leftarrow n$
  while $l < r$ **do**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    if $a \leq a_m$ **then** $r \leftarrow m$ **else** $l \leftarrow m + 1$ **end**
  end
Ausgabe: i falls $a_l = a$ bzw. Fehlermeldung, falls $a_l \neq a$

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 $O(\log n)$.

2.5 Sortieren von Zahlenfolgen

Wie wir im letzten Abschnitt gesehen haben, lassen sich Elemente in einer sortierten Folge sehr schnell aufspüren. Falls wir also diese Operation sehr oft ausführen müssen, bietet es sich an, die Zahlenfolge zu sortieren.

Sortierproblem

Gegeben: Eine Folge $a_1, \ldots, a_n$ von natürlichen Zahlen.

Gesucht: Eine Permutation $a_{i_1}, \ldots, a_{i_n}$ dieser Folge mit $a_{i_j} \leq a_{i_{j+1}}$ für $j = 1, \ldots, n-1$.

Man unterscheidet auf Vergleichen basierende Sortierverfahren von den übrigen Sortierverfahren. Während erstere nur Anfragen der Form $a_i \leq a_j$ stellen dürfen, können letztere auch die konkreten Zahlenwerte a_i der Folge verwerten. Vergleichsbasierte Verfahren benötigen im schlechtesten Fall $\Omega(n \log n)$ Vergleiche, während letztere unter bestimmten Zusatzvoraussetzungen sogar in Linearzeit arbeiten.

Ein einfacher Ansatz, eine Zahlenfolge zu sortieren, besteht darin, sequentiell die Zahl a_i ($i = 2, \ldots, n$) in die bereits sortierte Teilfolge $a_1, \ldots, a_{i-1}$ einzufügen.

Algorithmus 13 INSERTION-SORT

```
1   Eingabe: Eine Zahlenfolge a_1, ..., a_n
2     for i := 2 to n do
3       z ← a_i
4       j ← i − 1
5       while (j ≥ 1 ∧ a_j > z) do
6         a_{j+1} ← a_j
7         j ← j − 1
8       end
9       a_{j+1} ← z
10  Ausgabe: a_1, ..., a_n
```

Die Korrektheit von `Insertion-Sort` lässt sich induktiv durch den Nachweis folgender Schleifeninvarianten beweisen:

- Nach jedem Durchlauf der **for**-Schleife sind $a_1, \ldots, a_i$ sortiert.
- Nach jedem Durchlauf der **while**-Schleife gilt $z < a_k$ für $k = j + 2, \ldots, i$.

Zusammen mit der Abbruchbedingung der **while**-Schleife folgt hieraus, dass z in Zeile 5 an der jeweils richtigen Stelle eingefügt wird.

Da zudem die **while**-Schleife für jedes $i = 2, \ldots, n$ höchstens $(i-1)$-mal ausgeführt wird, ist die Laufzeit von `Insertion-Sort` durch $\sum_{i=2}^{n} O(i-1) = O(n^2)$ begrenzt.

Bemerkung 14

- Ist die Eingabefolge $a_1, \ldots, a_n$ bereits sortiert, so wird die **while**-Schleife niemals durchlaufen. Im *besten Fall* ist die Laufzeit daher $\sum_{i=2}^{n} \Theta(1) = \Theta(n)$.
- Ist die Eingabefolge $a_1, \ldots, a_n$ dagegen absteigend sortiert, so wandert z in $i-1$ Durchläufen der **while**-Schleife vom Ende an den Anfang der bereits sortierten Teilfolge $a_1, \ldots, a_i$. Im *schlechtesten Fall* ist die Laufzeit also $\sum_{i=2}^{n} \Theta(i-1) = \Theta(n^2)$.
- Bei einer zufälligen Eingabepermutation der Folge $1, \ldots, n$ wird z im Erwartungswert in der Mitte der Teilfolge $a_1, \ldots, a_i$ eingefügt. Folglich beträgt die (erwartete) Laufzeit im *durchschnittlichen Fall* ebenfalls $\sum_{i=2}^{n} \Theta(\frac{i-1}{2}) = \Theta(n^2)$.

Wir können eine Zahlenfolge auch sortieren, indem wir sie in zwei Teilfolgen zerlegen, diese durch rekursive Aufrufe sortieren und die sortierten Teilfolgen wieder zu einer Liste zusammenfügen.

Diese Vorgehensweise ist unter dem Schlagwort “Divide and Conquer” (auch “divide et impera”, also “teile und herrsche”) bekannt. Dabei wird ein Problem gelöst, indem man es

- in mehrere Teilprobleme aufteilt,
- die Teilprobleme rekursiv löst, und
- die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems zusammenfügt.

Die Prozedur `Mergesort`(A, l, r) sortiert ein Feld $A[l \ldots r]$, indem sie

- es in die Felder $A[l \ldots m]$ und $A[m+1 \ldots r]$ zerlegt,
- diese durch jeweils einen rekursiven Aufruf sortiert, und
- die sortierten Teilfolgen durch einen Aufruf der Prozedur `Merge`(A, l, m, r) zu einer sortierten Folge zusammenfügt.

Prozedur 15 MERGESORT(A, l, r)

```
1  if l < r then
2    m ← ⌊(l + r)/2⌋
3    Mergesort(A, l, m)
4    Mergesort(A, m + 1, r)
5    Merge(A, l, r, m)
```

Die Prozedur `Merge`(A, l, m, r) mischt die beiden sortierten Felder $A[l \ldots m]$ und $A[m+1 \ldots r]$ zu einem sortierten Feld $A[l \ldots r]$.

Prozedur 16 MERGE(A, l, m, r)

```
1  allokiere Speicher für ein neues Feld B[l ... r]
2  j ← l
3  k ← m + 1
4  for i := l to r do
5    if j > m then B[i] ← A[k]; k ← k + 1
6    else if k > r then B[i] ← A[j]; j ← j + 1
7    else if A[j] ≤ A[k] then B[i] ← A[j]; j ← j + 1
8    else B[i] ← A[k]; k ← k + 1 end
9  kopiere das Feld B[l ... r] in das Feld A[l ... r]
10 gib den Speicher für B wieder frei
```

Man beachte, dass `Merge` für die Zwischenspeicherung der gemischten Folge zusätzlichen Speicher benötigt. `Mergesort` ist daher kein *"in place"-Sortierverfahren,* welches neben dem Speicherplatz für die Eingabefolge nur konstant viel zusätzlichen Speicher belegen darf. Zum Beispiel ist `Insertion-Sort` ein "in place"-Verfahren.

Auch `Mergesort` kann als ein "in place"-Sortierverfahren implementiert werden, falls die zu sortierende Zahlenfolge nicht als Array, sondern als mit Zeigern verkettete Liste vorliegt.

Unter der Voraussetzung, dass `Merge` korrekt arbeitet, können wir per Induktion über die Länge $n = r - l + 1$ des zu sortierenden Arrays die Korrektheit von `Mergesort` wie folgt beweisen:

$n = 1$**:** In diesem Fall tut `Mergesort` nichts, was offensichtlich korrekt ist.

$n \rightsquigarrow n + 1$**:** Um eine Folge der Länge $n+1 \geq 2$ zu sortieren, zerlegt sie `Mergesort` in zwei Folgen der Länge höchstens n. Diese werden durch die rekursiven Aufrufe nach IV korrekt sortiert und von `Merge` nach Voraussetzung korrekt zusammengefügt.

Die Korrektheit von `Merge` lässt sich leicht induktiv durch den Nachweis folgender Invariante für die FOR-Schleife beweisen:

- Nach jedem Durchlauf enthält $B[l \cdots i]$ die $i - l + 1$ kleinsten Elemente aus $A[l \cdots m]$ und $A[m + 1 \cdots r]$ in sortierter Reihenfolge.
- Hierzu wurden die ersten $j - 1$ Elemente von $A[l \cdots m]$ und die ersten $k - 1$ Elemente von $A[m + 1 \cdots r]$ nach B kopiert.

Nach dem letzten Durchlauf (d.h. $i = r$) enthält daher $B[l \cdots r]$ alle $r - l + 1$ Elemente aus $A[l \cdots m]$ und $A[m + 1 \cdots r]$ in sortierter Reihenfolge, womit die Korrektheit von `Merge` bewiesen ist.

Um eine Schranke für die Laufzeit von `Mergesort` zu erhalten, schätzen wir zunächst die Anzahl $V(n)$ der Vergleiche ab, die `Mergesort` (im schlechtesten Fall) benötigt, um ein Feld $A[l \cdots m]$ der Länge $n = r - l + 1$ zu sortieren. Wir bezeichnen die Anzahl der Vergleiche, die `Merge` benötigt, um die beiden sortierten Felder $A[l \ldots m]$ und $A[m + 1 \ldots r]$ zu mischen, mit $M(n)$. Dann erfüllt $V(n)$ die Rekursionsgleichung

$$V(n) = \begin{cases} 0, & \text{falls } n = 1, \\ V(\lfloor n/2 \rfloor) + V(\lceil n/2 \rceil) + M(n), & \text{sonst.} \end{cases}$$

Offensichtlich benötigt `Merge` $M(n) = n - 1$ Vergleiche. Falls n also eine Zweierpotenz ist, genügt es, folgende Rekursion zu lösen:

$$V(1) = 0 \text{ und } V(n) = 2V(n/2) + n - 1, n \geq 1.$$

Hierzu betrachten wir die Funktion $f(k) = V(2^k)$. Dann gilt

$$f(0) = 0 \text{ und } f(k) = 2f(k - 1) + 2^k - 1, k \geq 1.$$

Aus den ersten Folgengliedern $f(0) = 0$, $f(1) = 1$, $f(2) = 2 + 2^2 - 1 = 1 \cdot 2^2 + 1$, $f(3) = 2 \cdot 2^2 + 2 + 2^3 - 1 = 2 \cdot 2^3 + 1$ und $f(4) = 2 \cdot 2 \cdot 2^3 + 2 + 2^4 - 1 = 3 \cdot 2^4 + 1$ lässt sich vermuten, dass $f(k) = (k-1) \cdot 2^k + 1$ ist. Dies lässt sich leicht durch Induktion über k verifizieren, so dass wir für V die Lösungsfunktion $V(n) = n \log_2 n - n + 1$ erhalten.

Frage 17 *Wie viele Fragen der Form $a_i \leq a_j$ benötigt ein vergleichender Sortieralgorithmus A mindestens, um eine Folge $(a_1, \ldots, a_n)$ von n Zahlen zu sortieren?*

Zur Beantwortung dieser Frage betrachten wir alle $n!$ Eingabefolgen $(a_1, \ldots, a_n)$ der Form $(\pi(1), \ldots, \pi(n))$, wobei $\pi \in S_n$ eine beliebige Permutation auf der Menge $\{1, \ldots, n\}$ ist. Um diese Folgen korrekt zu sortieren, muss A solange Fragen der Form $a_i \leq a_j$ (bzw. $\pi(i) \leq \pi(j)$) stellen, bis höchstens noch eine Permutation $\pi \in S_n$ mit den erhaltenen Antworten konsistent ist. Damit A möglichst viele Fragen benötigt, beantworten wir diese so, dass mindestens die Hälfte der verbliebenen Permutationen mit unserer Antwort konsistent ist (*Mehrheitsvotum*). Diese Antwortstrategie stellt sicher, dass nach i Fragen noch mindestens $n!/2^i$ konsistente Permutationen übrig bleiben. Daher muss A mindestens

$$\lceil \log_2(n!) \rceil = n \log_2 n - n \log_2 e + \Theta(\log n) = n \log_2 n - \Theta(n)$$

Fragen stellen, um die Anzahl der konsistenten Permutationen auf Eins zu reduzieren.

Wir können das Verhalten von A auch durch einen Fragebaum B veranschaulichen, dessen Wurzel mit der ersten Frage von A markiert ist. Jeder Knoten v, der mit einer Frage markiert ist, hat genau zwei Nachfolger, die für die beiden möglichen Antworten ja und nein stehen. Stellt A nach Erhalt der Antwort eine weitere Frage, so markieren wir den zugehörigen Antwortknoten mit dieser Frage. Falls A keine weitere Fragen stellt und die mittels π permutierte Eingabefolge ausgibt, so ist der zugehörige Antwortknoten ein Blatt, das wir mit π markieren. Nun ist leicht zu sehen, dass die Tiefe von B mit der Anzahl $V(n)$ der von A benötigten Fragen im schlechtesten Fall übereinstimmt. Da B mindestens $n!$ Blätter hat, können wir leicht einen Pfad der Länge $\lceil \log_2(n!) \rceil$ finden, indem wir jeweils in den Unterbaum mit der größeren Blätterzahl verzweigen.

Wir fassen unsere Beobachtungen zur Komplexität von `MergeSort` zusammen: Falls n eine Zweierpotenz ist, stellt `MergeSort` im schlechtesten Fall $V(n) = n \log_2 n - n + 1$ Fragen. Ist n keine Zweierpotenz, so können wir die Anzahl der Fragen durch $V(n') \leq V(2n) = O(V(n))$ abschätzen, wobei $n' \leq 2n$ die kleinste Zweierpotenz größer als n ist. Es ist leicht zu sehen, dass die Laufzeit $T(n)$ von `MergeSort` asymptotisch durch die Anzahl $V(n)$ der Vergleiche beschränkt ist, d.h. es gilt $T(n) = O(V(n))$. Da zudem jedes vergleichende Sortierverfahren mindestens $\Omega(n \log n)$ Fragen benötigt, haben wir folgenden Satz bewiesen.

Satz 18 *`MergeSort` ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $O(n \log n)$.*

Im Allgemeinen liefert der “Divide and Conquer”-Ansatz einfach zu implementierende Algorithmen mit einfachen Korrektheitsbeweisen. Die Laufzeit $T(n)$ erfüllt dann eine Rekursionsgleichung der Form

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n \text{ „klein“ ist,} \\ D(n) + \sum_{i=1}^{\ell} T(n_i) + C(n), & \text{sonst.} \end{cases}$$

Dabei ist $D(n)$ der Aufwand für das Aufteilen der Probleminstanz und $C(n)$ der Aufwand für das Verbinden der Teillösungen. Um solche Rekursionsgleichungen zu lösen, kann man oft eine Lösung „raten“ und per Induktion beweisen. Mit Hilfe von Rekursionsbäumen lassen sich Lösungen auch „gezielt raten“. Eine asymptotische Abschätzung liefert folgender Hauptsatz der Laufzeitfunktionen (Satz von Akra & Bazzi).

Satz 19 (Mastertheorem) *Sei $T : \mathbb{N} \to \mathbb{N}$ eine Funktion der Form*

$$T(n) = \sum_{i=1}^{\ell} T(n_i) + f(n),$$

mit $\ell, n_1, \ldots, n_\ell \in \mathbb{N}$ und $n_i \in \{\lfloor \alpha_i n \rfloor, \lceil \alpha_i n \rceil\}$ für fest gewählte reelle Zahlen $0 < \alpha_i < 1$ für $i = 1, \ldots, \ell$. Dann gilt im Fall $f(n) = \Theta(n^k)$ mit $k \geq 0$:

$$T(n) = \begin{cases} \Theta(n^k), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k < 1, \\ \Theta(n^k \log n), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k = 1, \\ \Theta(n^c), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k > 1, \end{cases}$$

wobei c Lösung der Gleichung $\sum_{i=1}^{\ell} \alpha_i^c = 1$ ist.