Vorlesungsskript

Theoretische Informatik III

Sommersemester 2010

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

19. Mai 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 $\Theta(n^2)$.

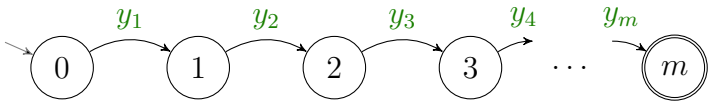
2.1.1 String-Matching mit endlichen Automaten

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:

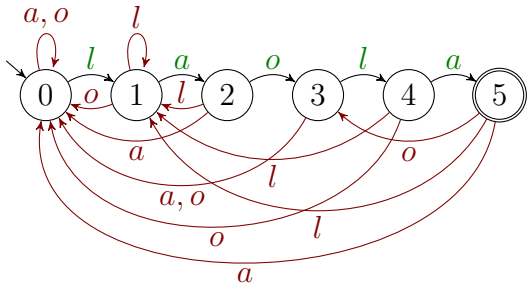
$$\hat{\delta}(0, x) = \max\{k \leq m \mid y_1 \cdots y_k \text{ ist Suffix von } x\}.$$

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

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

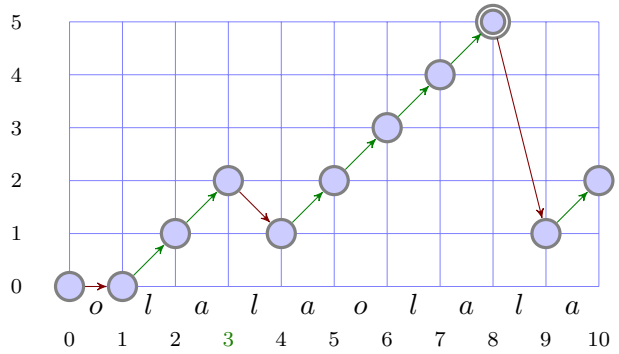
wobei $y_1 \cdots y_j$ das längste Präfix von $y = y_1 \cdots y_m$ ist, welches Suffix von $y_1 \cdots y_j a$ ist (siehe Übungen).

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“

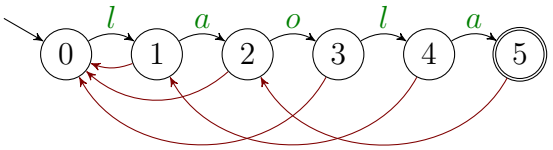
(also $y_{i+1} = a$ und somit $\tilde{p}a$ ein Präfix von y ist) oder der Zustand 0 erreicht wird.

- 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:*

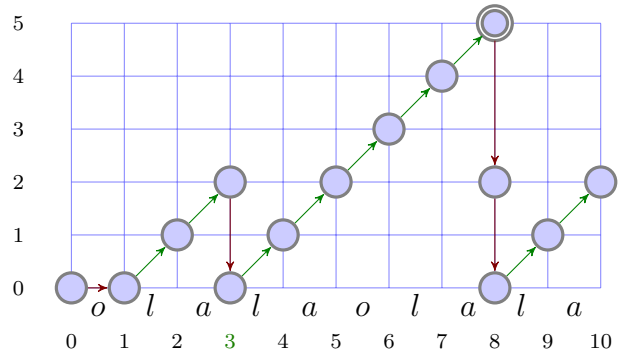


k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

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.

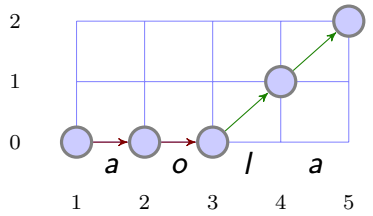
Nun kommen wir auf die Frage zurück, wie sich die Präfixfunktion π effizient berechnen lässt. Die Aufgabe besteht darin, für jedes Präfix $y_1 \cdots y_i$, $i \geq 1$, das längste echte Präfix zu berechnen, das zugleich Suffix von $y_1 \cdots y_i$ ist.

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* $l \leq r \wedge a_l \leq a \leq a_r$. Da nach Abbruch der while-Schleife zudem $l \geq r$ gilt, muss dann $l = r$ und $a = a_l$ sein. Dies zeigt die Korrektheit von `Binary-Search`.

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

2.3 Sortieralgorithmen

Wie wir im letzten Abschnitt gesehen haben, lassen sich Elemente in einer sortierten Folge sehr schnell aufspüren. Falls wir diese Operation öfters 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 *vergleichende Sortierverfahren* von den übrigen Sortierverfahren. Während erstere nur Ja-Nein-Fragen der Form „$a_i \leq a_j$?“ oder „$a_i < a_j$?“ stellen dürfen, können letztere auch die konkreten Zahlenwerte a_i der Folge abfragen. Vergleichsbasierte Verfahren benötigen im schlechtesten Fall $\Omega(n \log n)$ Vergleiche, während letztere unter bestimmten Zusatzvoraussetzungen sogar in Linearzeit arbeiten.

2.3.1 Sortieren durch Einfügen

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 `Insertion-Sort`$(a_1, \ldots, a_n)$

```
1  for i := 2 to n do
2    z := a_i
3    j := i − 1
4    while (j ≥ 1 ∧ a_j > z) do
5      a_{j+1} := a_j
6      j := j − 1
7    a_{j+1} := z
```

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: $a_{j+1}, \ldots, a_{i-1}$ wurden um jeweils ein Feld nach $a_{j+2}, \ldots, a_i$ verschoben und $z < a_{j+2}$.

Zusammen mit der Abbruchbedingung der **while**-Schleife folgt hieraus, dass z in Zeile 5 an der 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 $\sum_{i=2}^{n} \mathcal{O}(i-1) = \mathcal{O}(n^2)$ begrenzt.

Bemerkung 8.

- *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* $\sum_{i=2}^{n} \Theta(\frac{i-1}{2}) = \Theta(n^2)$.

2.3.2 Sortieren durch Mischen

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

Die Prozedur `Mergesort`(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`(l, m, r) zu einer sortierten Folge zusammenfügt.

Prozedur `Mergesort`(l, r)

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

Die Prozedur `Merge`(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 `Merge`(l, m, r)

```
1  allokiere Speicher fuer 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
6        B[i] := A[k]
7        k := k + 1
8     else if k > r then
9        B[i] := A[j]
10       j := j + 1
11    else if A[j] ≤ A[k] then
```

```
12        B[i] := A[j]
13        j := j + 1
14      else
15        B[i] := A[k]
16        k := k + 1
17  kopiere das Feld B[l...r] in das Feld A[l...r]
18  gib den Speicher fuer 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 (hierzu muss allerdings auch noch die Rekursion durch eine Schleife ersetzt werden).

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 offenbar 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.
- Im Fall $j \leq m$ gilt $B[i] \leq A[j]$, und im Fall $k \leq r$ gilt $B[i] \leq A[k]$.

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 $A[l \cdots r]$ der Länge $n = r - l + 1$ zu sortieren. Offensichtlich erfüllt $V(n)$ die Rekursionsgleichung

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

Dabei ist $M(n) \leq n - 1$ die Anzahl der Vergleiche, die `Merge` benötigt, um die beiden sortierten Felder $A[l \ldots m]$ und $A[m + 1 \ldots r]$ zu mischen. Falls $M(n) = n - 1$ und n eine Zweierpotenz ist, erhalten wir also die Rekursion

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

Für die Funktion $f(k) = V(2^k)$ gilt dann

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

Aus den ersten Folgengliedern $f(0) = 0$, $f(1) = 1$,

$$\begin{array}{lcll} f(2) & = & 2 + 2^2 - 1 & = 1 \cdot 2^2 + 1, \\ f(3) & = & 2 \cdot 2^2 + 2 + 2^3 - 1 & = 2 \cdot 2^3 + 1, \\ f(4) & = & 2 \cdot 2 \cdot 2^3 + 2 + 2^4 - 1 & = 3 \cdot 2^4 + 1 \end{array}$$

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.

Ist n eine beliebige Zahl und gilt nur $M(n) \leq n - 1$, so folgt

$$V(n) \leq n' \log_2 n' - n' + 1 \leq 2n \log_2(2n) + 1 = \mathcal{O}(n \log n),$$

wobei $n' < 2n$ die kleinste Zweierpotenz $\geq n$ ist. Da die Laufzeit $T(n)$ von `MergeSort` asymptotisch durch die Anzahl $V(n)$ der Vergleiche beschränkt ist, folgt $T(n) = \mathcal{O}(n \log n)$.

Satz 9. *`MergeSort` ist ein vergleichendes Sortierverfahren mit einer Laufzeit von $\mathcal{O}(n \log n)$.*

2.3.3 Lösen von Rekursionsgleichungen

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

$$T(n) = \sum_{i=1}^{\ell} T(n_i) + f(n) \quad \textit{mit} \quad n_i \in \{\lfloor \alpha_i n \rfloor, \lceil \alpha_i n \rceil\},$$

wobei $0 < \alpha_i < 1$, $i = 1, \ldots, \ell$, fest gewählte reelle Zahlen sind. Dann gilt im Fall $f(n) = \Theta(n^k)$ für ein $k \geq 0$:

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

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

Beispiel 11. *Die Anzahl $V(n)$ der Vergleiche von* `MergeSort` *erfüllt die Rekursion*

$$V(n) = V(\lfloor n/2 \rfloor) + V(\lceil n/2 \rceil) + n - 1,$$

d.h. $l = 2$, $\alpha_1 = \alpha_2 = 1/2$ und $f(n) = n - 1 = \Theta(n^k)$ für $k = 1$. Wegen $\sum_{i=1}^{\ell} \alpha_i^k = 1/2 + 1/2 = 1$ folgt daher $V(n) = \Theta(n \log n)$.

2.3.4 Eine untere Schranke für das Sortierproblem

Frage. *Wie viele Vergleichsfragen 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 $a_i < a_j$ (bzw. $\pi(i) < \pi(j)$) stellen, bis höchstens noch eine Permutation $\pi \in S_n$ mit den erhaltenen Antworten konsistent ist. Damit A möglichst viele Fragen stellen muss, 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

$$\lceil \log_2(n!) \rceil = n \log_2 n - n \log_2 e + \tfrac{1}{2} \log n + \Theta(1) = n \log_2 n - \Theta(n)$$

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

Satz 12. *Ein vergleichendes Sortierverfahren benötigt mindestens $\lceil \log_2(n!) \rceil$ Fragen, um eine Folge $(a_1, \ldots, a_n)$ von n Zahlen zu sortieren.*

Wir können das Verhalten von A auch durch einen Fragebaum B veranschaulichen, dessen Wurzel mit der ersten Frage von A markiert ist. Jeder mit einer Frage markierte Knoten hat zwei Kinder, die die Antworten ja und nein auf diese Frage repräsentieren. Stellt A nach Erhalt der Antwort eine weitere Frage, so markieren wir den

entsprechenden Antwortknoten mit dieser Frage. Andernfalls gibt A eine Permutation π der Eingabefolge aus und der zugehörige Antwortknoten ist 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 jede Eingabefolge zu einem anderen Blatt führt, hat B mindestens $n!$ Blätter. Folglich können wir in B einen Pfad der Länge $\lceil \log_2(n!) \rceil$ finden, indem wir jeweils in den Unterbaum mit der größeren Blätterzahl verzweigen.

Da also jedes vergleichende Sortierverfahren mindestens $\Omega(n \log n)$ Fragen benötigt, ist `Mergesort` asymptotisch optimal.

Korollar 13. *`MergeSort` ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.*

2.3.5 QuickSort

Ein weiteres Sortierverfahren, das den "Divide and Conquer"-Ansatz verfolgt, ist `QuickSort`. Im Unterschied zu `MergeSort` wird hier das Feld *vor* den rekursiven Aufrufen umsortiert. Als Folge hiervon bereitet die Zerlegung in Teilprobleme die Hauptarbeit, während das Zusammenfügen der Teillösungen trivial ist. Bei `MergeSort` ist es gerade umgekehrt.

Prozedur `QuickSort`(l, r)

```
1  if l < r then m := Partition(l, r)
2     QuickSort(l, m − 1)
3     QuickSort(m + 1, r)
```

Die Prozedur `QuickSort`(l, r) sortiert ein Feld $A[l \ldots r]$ wie folgt:

- Zuerst wird die Funktion `Partition`(l, r) aufgerufen.
- Diese wählt ein *Pivotelement*, welches sich nach dem Aufruf in $A[m]$ befindet, und sortiert das Feld so um, dass gilt:

$$A[i] \leq A[m] \leq A[j] \text{ für alle } i, j \text{ mit } l \leq i < m < j \leq r. \qquad (*)$$

- Danach werden die beiden Teilfolgen $A[l \ldots m-1]$ und $A[m+1 \ldots r]$ durch jeweils einen rekursiven Aufruf sortiert.

Die Funktion `Partition`(l, r) *pivotisiert* das Feld $A[l \ldots r]$, indem sie

- $x = A[r]$ als Pivotelement wählt,
- die übrigen Elemente mit x vergleicht und dabei umsortiert und
- den neuen Index $i + 1$ von x zurückgibt.

Prozedur `Partition`(l, r)

```
1  i := l − 1
2  for j := l to r − 1 do
3     if A[j] ≤ A[r] then
4        i := i + 1
5        if i < j then
6           vertausche A[i] und A[j]
7  if i + 1 < r then
8     vertausche A[i + 1] und A[r]
9  return(i+1)
```

Unter der Voraussetzung, dass die Funktion `Partition` korrekt arbeitet, d.h. nach ihrem Aufruf gilt $(*)$, folgt die Korrektheit von `QuickSort` durch einen einfachen Induktionsbeweis über die Länge $n = r - l + 1$ des zu sortierenden Arrays.

Die Korrektheit von `Partition` wiederum folgt leicht aus folgender Invariante für die for-Schleife:

$$A[k] \leq A[r] \text{ für } k = l, \ldots, i \text{ und } A[k] > A[r] \text{ für } k = i+1, \ldots, j. \quad (**)$$

Da nämlich nach Ende der for-Schleife $j = r - 1$ ist, garantiert die Vertauschung von $A[i+1]$ und $A[r]$ die Korrektheit von `Partition`.

Wir müssen also nur noch die Gültigkeit der Schleifeninvariante (∗∗) nachweisen. Um eindeutig definierte Werte von j vor und nach jeder Iteration der for-Schleife zu haben, ersetzen wir diese durch eine semantisch äquivalente while-Schleife:

Prozedur `Partition`(l, r)

```
1  i := l − 1
2  j := l − 1
3  while j < r − 1 do
4     j := j + 1
5     if A[j] ≤ A[r] then
6        i := i + 1
7        if i < j then
8           vertausche A[i] und A[j]
9  if i + 1 < r then
10    vertausche A[i + 1] und A[r]
11 return(i+1)
```

Nun lässt sich die Invariante (∗∗) leicht induktiv beweisen.

Induktionsanfang: Vor Beginn der while-Schleife gilt die Invariante, da i und j den Wert $l - 1$ haben.

Induktionsschritt: Zunächst wird j hochgezählt und dann $A[j]$ mit $A[r]$ verglichen.

Im Fall $A[j] > A[r]$ behält die Invariante ihre Gültigkeit, da nur j hochgezählt wird und i unverändert bleibt.

Im Fall $A[j] \leq A[r]$ wird auch i hochgezählt. Ist nun $i = j$, so folgt $A[i] = A[\,j] \leq A[r]$ und die Invariante behält ihre Gültigkeit. Ist dagegen $i < j$, so gilt nach Zeile 6 $A[i] > A[r]$. *Nach* der Vertauschung in Zeile 8 ist also $A[i] \leq A[r]$ und $A[j] > A[r]$, so dass die Invariante auch hier erhalten bleibt.

Als nächstes schätzen wir die Laufzeit von `QuickSort` im schlechtesten Fall ab. Dieser Fall tritt ein, wenn sich das Pivotelement nach jedem Aufruf von `Partition` am Rand von A (d.h. $m = l$ oder $m = r$) befindet. Dies führt nämlich dazu, dass `Partition` der Reihe nach mit Feldern der Länge $n, n-1, n-2, \ldots, 1$ aufgerufen wird. Da `Partition` für die Umsortierung eines Feldes der Länge n genau $n - 1$ Vergleiche benötigt, führt `QuickSort` insgesamt die maximal mögliche Anzahl

$$V(n) = \sum_{i=1}^{n}(i-1) = \binom{n}{2} = \Theta(n^2)$$

von Vergleichen aus. Dieser ungünstige Fall tritt insbesondere dann ein, wenn das Eingabefeld A bereits (auf- oder absteigend) sortiert ist.

Im *besten Fall* zerlegt das Pivotelement das Feld dagegen jeweils in zwei gleich große Felder, d.h. $V(n)$ erfüllt die Rekursion

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

Diese hat die Lösung $V(n) = n \log_2 n - \Theta(n)$ (vgl. die worst-case Abschätzung bei `MergeSort`).

Es gibt auch Pivotauswahlstrategien, die in linearer Zeit z.B. den Median bestimmen. Dies führt auf eine Variante von `QuickSort` mit einer Laufzeit von $\Theta(n \log n)$ bei *allen* Eingaben. Allerdings ist die Bestimmung des Medians für praktische Zwecke meist zu aufwändig.

Bei der Analyse des Durchschnittsfalls gehen wir von einer zufälligen Eingabepermutation $A[1 \ldots n]$ der Folge $1, \ldots, n$ aus. Dann ist die Anzahl $V(n)$ der Vergleichsanfragen von `QuickSort` eine Zufallsvariable. Wir können $V(n)$ als Summe $\sum_{1 \leq i < j \leq n} X_{ij}$ folgender Indikatorvariablen darstellen:

$$X_{ij} = \begin{cases} 1, & \text{falls die Werte } i \text{ und } j \text{ verglichen werden,} \\ 0, & \text{sonst.} \end{cases}$$

Ob die Werte i und j verglichen werden, entscheidet sich beim ersten Aufruf von `Partition`(l, r), bei dem das Pivotelement $x = A[r]$ im Intervall

$$I_{ij} = \{i, \ldots, j\}$$

liegt. Bis zu diesem Aufruf werden die Werte im Intervall I_{ij} nur mit Pivotelementen außerhalb von I_{ij} verglichen und bleiben daher im gleichen Teilfeld $A[l \ldots r]$ beisammen. Ist das erste Pivotelement x in I_{ij} nun nicht gleich i oder j, dann werden i und j nicht miteinander verglichen. Das liegt daran dass im Fall $i < x < j$ die Werte i und j bei diesem Aufruf in zwei verschiedene Teilfelder getrennt werden ohne miteinander verglichen zu werden.

Die Werte i und j werden also genau dann verglichen, wenn das erste Pivotelement x im Intervall I_{ij} den Wert i oder j hat. Da die Eingabe eine Zufallsfolge ohne Mehrfachvorkommen ist, nimmt x jeden Wert in I_{ij} mit Wahrscheinlichkeit $1/(j-i+1)$ an. Daher findet mit Wahrscheinlichkeit $p_{ij} = 2/(j-i+1)$ ein Vergleich zwischen den Werten i und j statt.

Der Erwartungswert von $V(n) = \sum_{1 \le i < j \le n} X_{ij}$ berechnet sich nun zu

$$\begin{aligned} E[V(n)] &= \sum_{1 \le i < j \le n} \underbrace{E[X_{ij}]}_{p_{ij}} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\le \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{2}{k} \le 2 \sum_{i=1}^{n-1} \log n = \mathcal{O}(n \log n). \end{aligned}$$

Damit ist die durchschnittliche Laufzeit von `QuickSort` $\mathcal{O}(n \log n)$. Dass dies für vergleichende Sortierverfahren asymptotisch optimal ist, wird in den Übungen gezeigt.

Satz 14. *`QuickSort` ist ein vergleichendes Sortierverfahren mit einer im Durchschnitt asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.*

Unabhängig davon nach welcher (deterministischen) Strategie das Pivotelement gewählt wird, wird es immer Eingabefolgen geben, für die `QuickSort` $\binom{n}{2}$ Vergleiche benötigt. Eine Möglichkeit, die Effizienz von `QuickSort` im Durchschnittsfall auf den schlechtesten Fall zu übertragen, besteht darin, eine *randomisierte* Auswahlstrategie für das Pivotelement anzuwenden.

Die Prozedur `RandomQuickSort`(l, r) arbeitet ähnlich wie `QuickSort`. Der einzige Unterschied besteht darin, dass als Pivotelement ein zufälliges Element aus dem Feld $A[l \ldots r]$ gewählt wird.

Prozedur `RandomQuickSort`(l, r)

```
1  if l < r then
2     m := RandomPartition(l, r)
3     RandomQuickSort(l, m − 1)
4     RandomQuickSort(m + 1, r)
```

Prozedur `RandomPartition`(l, r)

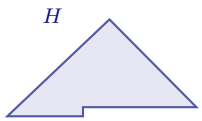
```
1  guess randomly j ∈ {l, ..., r}
2  if j < r then
3     vertausche A[j] und A[r]
4  return(Partition(l, r))
```

Es ist nicht schwer zu zeigen, dass sich `RandomQuickSort` bei *jeder* Eingabefolge $A[l, \ldots, r]$ gleich verhält wie `QuickSort` bei einer zufälligen Permutation dieser Eingabefolge (siehe Übungen). Daher ist die erwartete Laufzeit von `RandomQuickSort` auch im *schlechtesten Fall* durch $\mathcal{O}(n \log n)$ beschränkt, falls die Zahlenwerte paarweise verschieden sind.

Satz 15. *`RandomQuickSort` ist ein randomisiertes vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen erwarteten Laufzeit von $\mathcal{O}(n \log n)$.*

2.3.6 HeapSort

`HeapSort` benutzt als Datenstruktur einen so genannten *Heap*, um ein Feld zu sortieren.



Definition 16. *Ein* ***Heap*** H *mit n Knoten ist ein geordneter Binärbaum nebenstehender Form. Das heißt,*

- *H hat in Tiefe $i = 0, 1, \ldots, \lfloor \log_2 n \rfloor - 1$ jeweils die maximale Anzahl von 2^i Knoten und*
- *in Tiefe $\lfloor \log_2 n \rfloor$ sind alle Knoten linksbündig angeordnet.*

Zudem ist jeder Knoten v mit einer Zahl $H[v]$ beschriftet, deren Wert mindestens so groß ist wie die Werte der Kinder von v (sofern vorhanden).

Ein Heap H mit n Knoten lässt sich in einem Feld $H[1, \ldots, n]$ speichern. Dabei gilt:

- Das linke Kind von Knoten i hat den Index $\texttt{left}(i) = 2i$.
- Das rechte Kind von Knoten i hat den Index $\texttt{right}(i) = 2i + 1$.
- Der Elternknoten von Knoten i hat den Index $\texttt{parent}(i) = \lfloor i/2 \rfloor$.

Die Heap-Eigenschaft lässt sich nun wie folgt formulieren. Für alle Knoten $i \in \{1, \ldots, n\}$ gilt

$$(2i \leq n \Rightarrow H[i] \geq H[2i]) \wedge (2i + 1 \leq n \Rightarrow H[i] \geq H[2i + 1]).$$

Da die Knoten im Intervall $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$ keine Kinder haben, ist für sie die Heap-Eigenschaft automatisch erfüllt.

Ist $H[1, \ldots, n]$ ein Heap, dann repräsentiert auch jedes Anfangsstück $H[1, \ldots, r]$, $1 \leq r \leq n$, einen Heap H_r mit r Knoten. Zudem ist für $1 \leq i \leq r \leq n$ der Teilbaum von H_r mit Wurzel i ein Heap, den wir mit $H_{i,r}$ bezeichnen.

Da die Wurzel $H[1]$ eines Heaps den größten Wert haben muss, können wir eine in einem Feld $H[1, \ldots, n]$ gespeicherte Zahlenfolge sortieren, indem wir H zuerst zu einem Heap umsortieren und dann sukzessive

- die Wurzel $H[1]$ mit dem letzten Heap-Element vertauschen,
- den rechten Rand des Heaps um ein Feld nach links verschieben (also die vormalige Wurzel des Heaps herausnehmen) und
- die durch die Ersetzung der Wurzel verletzte Heap-Eigenschaft wieder herstellen.

Sei $H[1, \ldots, n]$ ein Feld, so dass der Teilbaum $H_{i,r}$ die Heap-Eigenschaft in allen Knoten bis auf seine Wurzel i erfüllt. Dann stellt die Prozedur $\texttt{Heapify}(i, r)$ die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ her.

Prozedur $\texttt{Heapify}(i, r)$

```
1  if (2i ≤ r) ∧ (H[2i] > H[i]) then
2     x := 2i
3  else
4     x := i
5  if (2i + 1 ≤ r) ∧ (H[2i + 1] > H[x]) then
6     x := 2i + 1
7  if x > i then
8     vertausche H[x] und H[i]
9     Heapify(x, r)
```

Unter Verwendung der Prozedur `Heapify` ist es nun leicht, ein Feld zu sortieren.

Algorithmus `HeapSort`

```
1  Input: Ein Feld H[1, ..., n]
2  for i := ⌊n/2⌋ downto 1 do
3     Heapify(i, n)
4  for r := n downto 2 do
5     vertausche H[1] und H[r]
```

6 `Heapify`$(1, r-1)$

Wir setzen zunächst voraus, dass die Prozedur `Heapify` korrekt arbeitet. D.h. `Heapify`(i, r) stellt die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ her, falls $H_{i,r}$ die Heap-Eigenschaft höchstens in seiner Wurzel i nicht erfüllt. Unter dieser Voraussetzung folgt die Korrektheit von `HeapSort` mittels folgender Schleifeninvarianten, die sich sehr leicht verifizieren lassen.

Invariante für die erste for-Schleife (Zeilen 1 – 2):
Für $j = i, \ldots, n$ ist der Teilbaum $H_{j,n}$ ein Heap.

Nach Beendigung dieser Schleife (d.h. $i = 1$) ist demnach $H_{1,n}$ ein Heap.

Invariante für die zweite for-Schleife (Zeilen 3 – 5):
$H[r], \ldots, H[n]$ enthalten die $n - r + 1$ größten Feldelemente in sortierter Reihenfolge und der Teilbaum $H_{1,r-1}$ ist ein Heap.

Am Ende der zweiten for-Schleife (d.h. $r = 2$) enthält also $H[2, \ldots, n]$ die $n - 1$ größten Elemente in sortierter Reihenfolge, d.h. $H[1, \ldots, n]$ ist sortiert.

Als nächstes zeigen wir die Korrektheit von `Heapify`. Sei also $H[1, \ldots, n]$ ein Feld, so dass der Teilbaum $H_{i,r}$ die Heap-Eigenschaft in allen Knoten bis auf seine Wurzel i erfüllt. Dann müssen wir zeigen, dass `Heapify`(i, r) die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ herstellt.

`Heapify`(i, r) bestimmt den Knoten $x \in \{i, 2i, 2i + 1\}$ mit maximalem Wert $H(x)$. Im Fall $x = i$ erfüllt der Knoten i bereits die Heap-Eigenschaft. Ist x dagegen eines der Kinder von i, so vertauscht `Heapify` die Werte von i und x. Danach ist die Heap-Eigenschaft höchstens noch im Knoten x verletzt. Daher folgt die Korrektheit von `Heapify` durch einen einfachen Induktionsbeweis über die Rekursionstiefe.

Es ist leicht zu sehen, dass `Heapify`(i, r) maximal $2h(i)$ Vergleiche benötigt, wobei $h(i)$ die Höhe des Knotens i in $H_{1,r}$ ist. Daher ist die Laufzeit von `Heapify`(i, r) durch $\mathcal{O}(h(i)) = \mathcal{O}(\log r)$ beschränkt.

Für den Aufbau eines Heaps H der Tiefe $t = \lfloor \log_2 n \rfloor$ wird `Heapify` in der ersten for-Schleife für höchstens

- $2^0 = 1$ Knoten der Höhe $h = t$,
- $2^1 = 2$ Knoten der Höhe $h = t - 1$,

  $\vdots$
- 2^{t-1} Knoten der Höhe $h = t - (t - 1) = 1$

aufgerufen. Für $h = 1, \ldots, t$ sind das also höchstens 2^{t-h} Knoten der Höhe h. Da `Heapify` für einen Knoten der Höhe h höchstens $2h$ Vergleichsfragen stellt, benötigt der Aufbau des Heaps maximal

$$V_1(n) \leq 2 \sum_{h=1}^{t} h 2^{t-h} \leq 2 \sum_{h=1}^{t} h \frac{n}{2^h} < 2n \sum_{h=1}^{\infty} \frac{h}{2^h} = 4n$$

Vergleiche. Für den Abbau des Heaps in der zweiten for-Schleife wird `Heapify` genau $(n - 1)$-mal aufgerufen. Daher benötigt der Abbau des Heaps maximal

$$V_2(n) \leq 2(n-1)\lfloor \log_2 n \rfloor \leq 2n \log_2 n$$

Vergleiche.

Satz 17. *`HeapSort` ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.*

Die Floyd-Strategie

Die *Floyd-Strategie* benötigt beim Abbau des Heaps im Durchschnitt nur halb so viele Vergleiche wie die bisher betrachtete *Williams-Strategie*. Die Idee besteht darin, dass `Heapify`$(1, r)$ beginnend mit der Wurzel $i_0 = 1$ sukzessive die Werte der beiden Kinder des aktuellen Knotens i_j vergleicht und jeweils zu dem Kind i_{j+1} mit dem größeren Wert absteigt, bis nach $t \leq \lfloor \log_2 r \rfloor$ Schritten ein Blatt i_t erreicht wird.

Nun geht `Heapify` auf diesem Pfad bis zum ersten Knoten i_j mit $H[i_j] \geq H[1]$ zurück und führt auf den Werten der Knoten $i_j, i_{j-1}, \ldots, i_0$ einen Ringtausch aus, um die Heap-Eigenschaft herzustellen. Dies erfordert

$$t + (t - j + 1) = 2t - j + 1$$

Vergleiche (im Unterschied zu $2j$ Vergleichen bei der Williams-Strategie). Da sich der Knoten i_j, an dessen Stelle der Wurzelknoten eingefügt wird, im Mittel sehr weit unten im Baum befindet (d.h. $t - j = \mathcal{O}(1)$), spart man auf diese Art asymptotisch die Hälfte der Vergleiche.

2.3.7 BucketSort

Die Prozedur `BucketSort` sortiert n Zahlen $a_1, \ldots, a_n$ aus einem Intervall $[a, b)$ wie folgt (z.B. für $n = 10$, $a = 0$ und $b = 100$):

1. Erstelle für $j = 1, \ldots, n$ eine Liste L_j für das halb offene Intervall $I_j = \left[a + (j-1)\frac{b-a}{n}, a + j\,\frac{b-a}{n}\right) = [10(j-1), 10j)$.
2. Bestimme zu jedem Element a_i das Intervall I_j, zu dem es gehört, und füge es in die entsprechende Liste L_j ein.
3. Sortiere jede Liste L_j.
4. Füge die sortierten Listen L_j wieder zu einer Liste zusammen.

Im schlechtesten Fall kommen alle Schlüssel in die gleiche Liste. Dann hat `BucketSort` dieselbe asymptotische Laufzeit wie das als Unterroutine verwendete Sortierverfahren. Sind dagegen die zu sortierenden Zahlenwerte im Intervall $[a, b)$ (annähernd) gleichverteilt, so ist die durchschnittliche Laufzeit von `BucketSort` $\Theta(n)$. Dies gilt sogar, wenn als Unterroutine ein Sortierverfahren der Komplexität $\mathcal{O}(n^2)$ verwendet wird.

Wir schätzen nun die erwartete Laufzeit von `BucketSort` ab, wobei wir annehmen, dass die Folgenglieder a_i im Intervall $[a, b)$ unabhängig gleichverteilt sind. Sei X_i die Zufallsvariable, die die Länge der Liste L_i beschreibt. Dann ist X_i binomialverteilt mit Parametern n und $p = 1/n$. Also hat X_i den Erwartungswert

$$E[X_i] = np = 1$$

und die Varianz

$$V[X_i] = np(1-p) = 1 - 1/n < 1.$$

Wegen $V[X_i] = E[X_i^2] - E[X_i]^2$ ist $E[X_i^2] = V[X_i] + E[X_i]^2 < 2$. Daher folgt für die erwartete Laufzeit $T(n)$ von `BucketSort`:

$$T(n) = \mathcal{O}(n) + E\left[\sum_{i=0}^{n-1} \mathcal{O}(X_i^2)\right] = \mathcal{O}\left(n + \sum_{i=0}^{n-1} E[X_i^2]\right) = \mathcal{O}(n).$$

2.3.8 CountingSort

Die Prozedur `CountingSort` sortiert eine Zahlenfolge, indem sie zunächst die Anzahl der Vorkommen jedes Wertes in der Folge und daraus die Rangzahlen $C[i] = \|\{j \mid A[j] \leq i\}\|$ der Zahlenwerte $i = 0, \ldots, k$ bestimmt. Dies funktioniert nur unter der Einschränkung, dass die Zahlenwerte natürliche Zahlen sind und eine Obergrenze k für ihre Größe bekannt ist.

Algorithmus `CountingSort`

```
1  for i := 0 to k do C[i] := 0
2  for j := 1 to n do C[A[j]] := C[A[j]] + 1
3  for i := 1 to k do C[i] := C[i] + C[i − 1]
4  for j := 1 to n do
5     B[C[A[j]]] := A[j]
6     C[A[j]] := C[A[j]] − 1
7  for j := 1 to n do A[j] := B[j]
```

Satz 18. *`CountingSort` sortiert n natürliche Zahlen der Größe höchstens k in Zeit $\Theta(n+k)$ und Platz $\Theta(n+k)$.*

Korollar 19. *`CountingSort` sortiert n natürliche Zahlen der Größe $\mathcal{O}(n)$ in linearer Zeit und linearem Platz.*

2.3.9 RadixSort

RadixSort sortiert d-stellige Zahlen $a = a_d \cdots a_1$ eine Stelle nach der anderen, wobei mit der niederwertigsten Stelle begonnen wird.

Algorithmus `RadixSort`

```
1  for i := 1 to d do
2    sortiere A[1,...,n] nach der i-ten Stelle
```

Hierzu sollten die Folgenglieder möglichst als Festkomma-Zahlen vorliegen. Zudem muss in Zeile 2 „stabil“ sortiert werden.

Definition 20. *Ein Sortierverfahren heißt **stabil**, wenn es die relative Reihenfolge von Elementen mit demselben Wert nicht verändert.*

Es empfiehlt sich, eine stabile Variante von `CountingSort` als Unterroutine zu verwenden. Damit `CountingSort` stabil sortiert, brauchen wir lediglich die for-Schleife in Zeile 4 in der umgekehrten Reihenfolge zu durchlaufen:

```
1  for i := 0 to k do C[i] := 0
2  for j := 1 to n do C[A[j]] := C[A[j]] + 1
3  for i := 1 to k do C[i] := C[i] + C[i-1]
4  for j := n downto 1 do
5    B[C[A[j]]] := A[j]
6    C[A[j]] := C[A[j]] - 1
7  for j := 1 to n do A[j] := B[j]
```


Satz 21. *`RadixSort` sortiert n d-stellige Festkomma-Zahlen zur Basis b in Zeit $\Theta(d(n+b))$.*

`RadixSort` sortiert beispielsweise n $\mathcal{O}(\log n)$-stellige Binärzahlen in Zeit $\Theta(n \log n)$. Wenn wir r benachbarte Ziffern zu einer „Ziffer“ $z \in \{0, \ldots, b^r - 1\}$ zusammenfassen, erhalten wir folgende Variante von `RadixSort`.

Korollar 22. *Für jede Zahl $1 \leq r \leq d$ sortiert `RadixSort` n d-stellige Festkomma- Zahlen zur Basis b in Zeit $\Theta(d/r(n + b^r))$.*

Wählen wir beispielsweise $r = \lceil \log_2 n \rceil$, so erhalten wir für $d = \mathcal{O}(\log n)$-stellige Binärzahlen eine Komplexität von

$$\Theta\left(d/r(n + 2^r)\right) = \Theta(n + 2^r) = \Theta(n).$$

2.3.10 Vergleich der Sortierverfahren

Folgende Tabelle zeigt die Komplexitäten der betrachteten vergleichsbasierten Sortierverfahren.

	`Insertion-Sort`	`MergeSort`	`Quick-Sort`	`Heap-Sort`
worst-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
average-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Speicherplatz	$\Theta(1)$	$\Theta(n)$ bzw. $\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
stabil	ja	ja	nein	nein

Wir fassen auch die wichtigsten Eigenschaften der betrachteten Linearzeit-Sortierverfahren zusammen.

- `BucketSort`: Im Durchschnitt linearer Zeitverbrauch, falls die n Zahlen in einem Intervall $[a, b)$ gleichverteilt sind.

- `CountingSort`: Sogar im schlechtesten Fall lineare Zeit, falls die Werte natürliche Zahlen sind und $\mathcal{O}(n)$ nicht übersteigen.
- `RadixSort`: Bitweises Sortieren in linearer Zeit, falls die zu sortierenden Zahlen in Festkomma-Darstellung nicht mehr als $\mathcal{O}(\log n)$ Bit haben.

2.4 Datenstrukturen für dynamische Mengen

Viele Algorithmen benötigen eine Datenstruktur für dynamische Mengen. Eine solche Datenstruktur S sollte im Prinzip *beliebig viele* Elemente aufnehmen können. Die Elemente $x \in S$ werden dabei anhand eines *Schlüssels* $k = \texttt{key}(x)$ identifiziert. Auf die Elemente $x \in S$ wird meist nicht direkt, sondern mittels *Zeiger* (engl. *pointer*) zugegriffen.

Typische Operationen, die auf einer dynamische Mengen S auszuführen sind:

`Insert`(S, x)**:** Fügt x in S ein.

`Remove`(S, x)**:** Entfernt x aus S.

`Search`(S, k)**:** Gibt für einen Schlüssel k (einen Zeiger auf) das Element $x \in S$ mit $\texttt{key}(x) = k$ zurück, falls ein solches Element existiert, und `nil` sonst.

`Min`(S)**:** Gibt das Element in S mit dem kleinsten Schlüssel zurück.

`Max`(S)**:** Gibt das Element in S mit dem größten Schlüssel zurück.

`Prec`(S, x)**:** Gibt das Element in S mit dem nach x nächstkleineren Schlüssel zurück (bzw. `nil`, falls x das Minimum ist).

`Succ`(S, x)**:** Gibt das Element in S mit dem nach x nächstgrößeren Schlüssel zurück (bzw. `nil`, falls x das Maximum ist).

2.4.1 Verkettete Listen

Die Elemente einer verketteten Liste sind in linearer Reihenfolge angeordnet. Das erste Element der Liste L ist $\texttt{head}(L)$. Jedes Element x „kennt“ seinen Nachfolger $\texttt{next}(x)$. Wenn jedes Element x auch seinen Vorgänger $\texttt{prev}(x)$ kennt, dann spricht man von einer *doppelt verketteten Liste*.

Die Prozedur $\texttt{L-Insert}(L, x)$ fügt ein Element x in eine verkettete Liste L ein.

Prozedur $\texttt{L-Insert}(L, x)$

```
1  next(x) := head(L)
2  head(L) := x
```

Die Prozedur $\texttt{DL-Insert}(L, x)$ fügt ein Element x in eine doppelt verkettete Liste L ein.

Prozedur $\texttt{DL-Insert}(L, x)$

```
1  next(x) := head(L)
2  if head(L) ≠ nil then
3     prev(head(L)) := x
4  head(L) := x
5  prev(x) := nil
```

Die Prozedur $\texttt{DL-Remove}(L, x)$ entfernt wieder ein Element x aus einer doppelt verketteten Liste L.

Prozedur $\texttt{DL-Remove}(L, x)$

```
1  if x ≠ head(L) then
2     next(prev(x)) := next(x)
3  else
4     head(L) := next(x)
5  if next(x) ≠ nil then
6     prev(next(x)) := prev(x)
```

Die Prozedur `DL-Search`(L, k) sucht ein Element x mit dem Schlüssel k in der Liste L.

Prozedur `DL-Search`(L, k)

```
1  x := head(L)
2  while x ≠ nil and key(x) ≠ k do
3     x := next(x)
4  return(x)
```

Es ist leicht zu sehen, dass `DL-Insert` und `DL-Remove` konstante Zeit $\Theta(1)$ benötigen, während `DL-Search` eine lineare (in der Länge der Liste) Laufzeit hat.