Vorlesungsskript

Algorithmische Diskrete Mathematik

PD Dr. Raymond Hemmecke

Version vom 28. April 2009

Korrekturen bitte an hemmecke@mathematik.tu-darmstadt.de

Inhaltsverzeichnis

1 Einführung **5**

1.1 Computer und Algorithmen 5

1.2 Ein elementares Beispiel 7

2 Kurze Einführung in die Graphentheorie **9**

2.1 Grundlegende Begriffe 9

2.2 Probleme auf Graphen 12

2.2.1 Kreis 12

2.2.2 Hamiltonischer Kreis 12

2.2.3 Aufspannende Bäume und Wälder 12

2.2.4 Matchings 12

2.2.5 Cliquen 12

2.2.6 Färbungsproblem 13

2.2.7 Kürzeste Wege 13

2.2.8 Traveling Saleman Problem (TSP) 13

3 Datenstrukturen **15**

3.1 Datentypen und Operationen, Datenstrukturen 15

3.2 Datenstrukturen zur Speicherung von Graphen 17

3.2.1 Kanten- und Bogenlisten 17

3.2.2 Adjazenzmatrizen 19

3.2.3 Adjazenzlisten 20

3.3 Depth-First Search Algorithmus 21

3.4 Beispiel für DFS Algorithmus 22

4 Komplexität, Speicherung von Daten **25**

4.1 Probleme, Komplexitätsmaße, Laufzeiten 25

4.2 Kodierungsschema 26

4.2.1 Ganze Zahlen 26

4.2.2 Rationale Zahlen . . . 26
4.2.3 Vektoren . . . 26
4.2.4 Matrizen . . . 26
4.3 Rechnermodell . . . 26
4.4 Asymptotische Notation . . . 28
4.4.1 Obere Schranken . . . 28
4.4.2 Untere Schranken . . . 30
4.5 Die Klassen $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-Vollständigkeit . . . 32
4.5.1 Entscheidungsprobleme . . . 32
4.5.2 Optimierungsprobleme . . . 34
4.6 Zwei wichtige Techniken zur Behandlung von kombinatorischen Optimierungsproblemen . . . 34
4.6.1 Binäre Suche . . . 35
4.6.2 Skalierungstechnik . . . 35

5 Kürzeste Wege **37**
5.1 Ein Startknoten, nichtnegative Gewichte . . . 37
5.2 Ein Startknoten, beliebige Gewichte . . . 39

6 Sortieren in Arrays **43**
6.1 Mergesort . . . 43
6.1.1 Mischen sortierter Arrays und der Algorithmus für Mergesort . . . 43
6.1.2 Die Analyse von Mergesort . . . 46
6.2 Beschleunigung durch Aufteilung: Divide-and-Conquer . . . 48
6.2.1 Aufteilungs-Beschleunigungssätze . . . 48
6.2.2 Multiplikation von Dualzahlen . . . 49
6.3 Quicksort . . . 51
6.3.1 Der Algorithmus . . . 51
6.3.2 Worst-Case Aufwand von Quicksort . . . 54
6.3.3 Mittlerer Aufwand von Quicksort . . . 55
6.4 Heapsort . . . 58

7 Untere Komplexitätsschranken für das Sortieren **65**
7.1 Das Entscheidungsbaum-Modell . . . 65
7.2 Analyse des Entscheidungsbaumes . . . 67

Kapitel 1

Einführung

Inhalt:

- Grundlagen des Entwurfs und der Analyse von Algorithmen
- Standardalgorithmen und Datenstrukturen
- Aspekte der Rechnernutzung: Zahldarstellung, Computerarithmetik, grundlegende Algorithmen der numerischen und diskreten Mathematik

1.1 Computer und Algorithmen

Was macht einen Computer revolutionär?

> Ein Computer ist Maschine, die geistige Routinearbeiten durchführt, indem sie einfache Operationen mit hoher Geschwindigkeit ausführt.
> Ein Computer kann nur solche Aufgaben erledigen, die durch einfache Operationen beschreibbar sind. Ferner muss man dem Computer mitteilen, wie die Aufgabe durchzuführen ist.

Definition 1.1.1 *(a) Ein* **Algorithmus** *ist eine in einer festgelegten Sprache abgefasste endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Verarbeitungsschritte zur Lösung einer gestellten Aufgabe.*

(b) Die konkrete Abarbeitung des Algorithmus nennt man **Prozess**. *Die Einheit, die den Prozess ausführt, heißt* **Prozessor**. *Ein Prozess besitzt zu jedem Zeitpunkt einen* **Zustand**, *der den aktuellen Stand der Ausführung angibt.*

Beispiele:

Prozess	Algorithmus	Typische Schritte im Algorithmus
Modellflugzeug bauen	Montageanleitung	leime Teil A an Flügel B
Kuchen backen	Kochrezept	rühre 3 Eier um

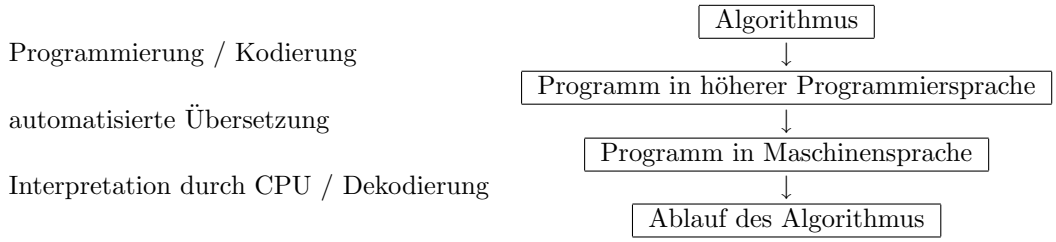
Merke: Ein Algorithmus ist eine Handlungsvorschrift und keine Problembeschreibung. Ein Prozessor ist nicht immer ein Computer. Vielmehr kann es auch ein Mensch oder ein Gerät sein.

⇒ Computer: spezieller Prozessor □

Ein typischer Computer hat 3 Hauptkomponenten:

Ein-, Ausgabegeräte	↔	Zentraleinheit CPU	↔	Speicher (memory)
Kommunikation		führt die Basisoperationen aus		enthält auszuführende Operationen des Alg., Daten und Objekte auf der die Operationen wirken

Die Ausführung eines Algorithmus auf einem Prozessor setzt voraus, dass der Prozessor den Algorithmus interpretieren kann.



- Ohne Algorithmus gibt es kein Programm.
- Algorithmen sind unabhängig von einer konkreten Programmiersprache und einem konkreten Computer.
- Programmiersprachen sind nur Mittel zum Zweck, um Algorithmen in Form von Prozessen auszuführen.

Aufgrund der grundlegenden Bedeutung von Algorithmen gibt es viele Gebiete der Angewandten Mathematik und Informatik, die sich mit Algorithmen beschäftigen im Rahmen der folgenden Schwerpunkte:

- Entwurf von Algorithmen
- Berechenbarkeit: Was kann durch einen Algorithmus berechnet werden?
- Komplexität von Algorithmen
- Korrektheit von Algorithmen

1.2 Ein elementares Beispiel

Problemstellung: Temperaturangaben in Fahrenheit sollen in Celsius umgerechnet werden
Algorithmus: basiert auf dem Zusammenhang der beiden Temperaturskalen:

(a) 0 Fahrenheit $= -16\frac{7}{9}$ Celsius

(b) 100 Fahrenheit $= 37\frac{7}{9}$ Celsius

(c) Die Temperatur in Celsius C ist eine lineare Funktion in der Temperatur in Fahrenheit F.

$$C = \frac{5}{9}(F - 32) \tag{1.1}$$

Dies ergibt den Algorithmus:

1. Einlesen von F.
2. Berechnung von C mittels Formel (1.1)
3. Ausgabe von C.

Programm in C:

```
#include <stdio.h>

/* Umwandlung von Fahrenheit in Celsius */

main() {
  float fahrenheit, celsius;
  printf("Temperatur in Fahrenheit eingeben\n");
  scanf("%f",&fahrenheit);
  celsius = (5.0/9.0)*(fahrenheit-32.0);
  printf("Temperatur in Celsius %6.1f\n",celsius);
}
```

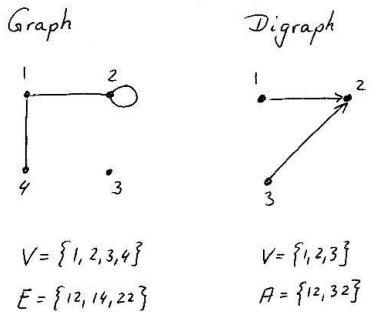
Kapitel 2

Kurze Einführung in die Graphentheorie

2.1 Grundlegende Begriffe

Ein (ungerichteter) **Graph** ist ein Paar $G = (V, E)$ disjunkter Mengen, wobei die Elemente von E 2-elementige Teilmengen (= ungeordnete Paare) von V sind.

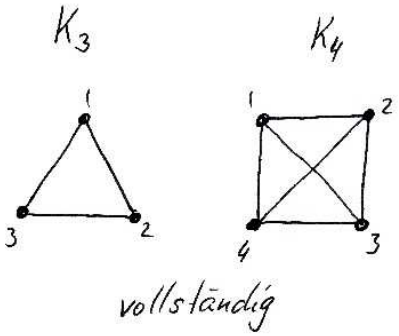
Ein **gerichteter Graph** (oder **Digraph**) ist ein Paar $G = (V, A)$ disjunkter Mengen mit $A \subseteq V \times V$, d.h. die Elemente von A sind geordnete Paare von Elementen aus V.



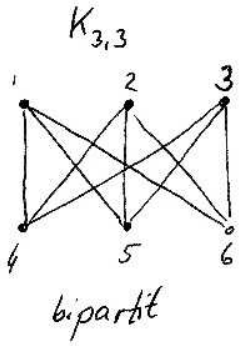
Die Elemente aus V heißen **Ecken** oder **Knoten**, die Elemente aus E **Kanten** und die Elemente aus A heißen **Bögen** oder **gerichtete Kanten**.

Eine Ecke v heißt mit einer Kante e **inzident**, wenn $v \in e$ gilt. Der **Grad** (oder die **Valenz**) einer Ecke v ist die Anzahl der mit v inzidenten Kanten.

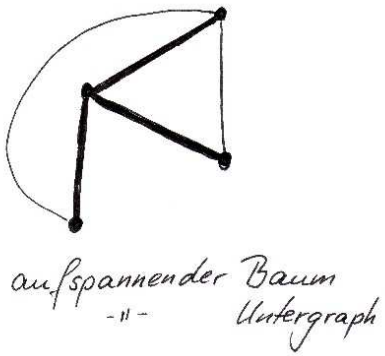
Zwei Ecken x, y von G heißen **adjazent** oder **benachbart** in G, wenn $xy \in E(G)$ ist. Sind je zwei Ecken von G benachbart, so heißt G **vollständig**.



Ein Graph heißt **bipartit**, wenn sich V disjunkt in zwei Teile V_1 und V_2 zerteilen lässt, so dass jede Kante in E einen Endknoten in V_1 und einen Endknoten in V_2 besitzt.



Ein Graph $G' = (V', E')$ heißt **Untergraph** von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.
$G' \subseteq G$ heißt **aufspannender Teilgraph** von G falls $V' = V$.



Ein **Weg** ist ein nichtleerer Graph $P = (V, E)$ der Form

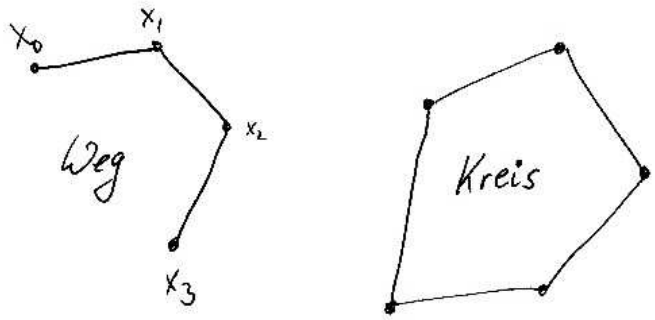
$$V = \{x_0, x_1, \ldots, x_k\} \quad E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\},$$

wobei die x_i paarweise verschieden sind.

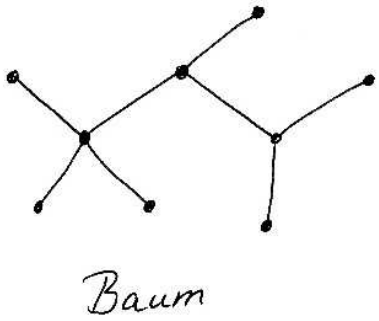
Ein **Kreis** ist ein nichtleerer Graph $P = (V, E)$ der Form

$$V = \{x_0, x_1, \ldots, x_{k-1}\} \quad E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_0\},$$

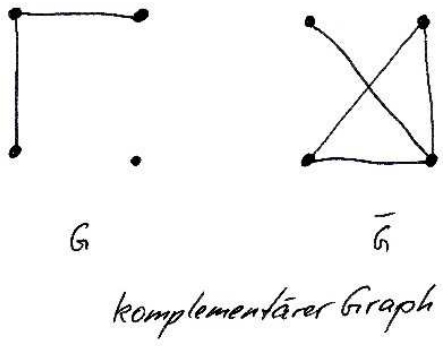
wobei die x_i paarweise verschieden sind.



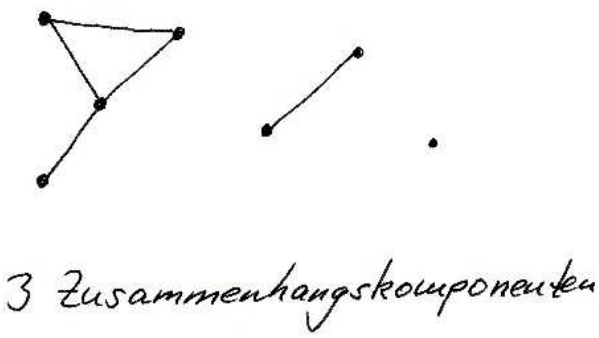
Ein **Baum** ist ein zusammenhängender Graph (= eine Zusammenhangskomponente), der keinen Kreis enthält. Ein **Wald** ist die (disjunkte) Vereinigung von Bäumen.



Der zu G **komplementäre Graph** $\bar{G}$ ist der Graph $\bar{G} = (V, E')$ wobei $ij \in E' \Leftrightarrow ij \notin E$, für $i, j \in V, i \neq j$.



Ein Graph G heißt **zusammenhängend**, falls es zu jedem Paar $v, w \in V$ einen Weg von v nach w in G gibt.



2.2 Probleme auf Graphen

2.2.1 Kreis

Problem: Entscheide für einen gegebenen Graphen $G = (V, E)$, ob er einen Kreis enthält. Falls ja, gib einen solchen an.

2.2.2 Hamiltonischer Kreis

Ein **hamiltonischer Kreis** ist ein Kreis, der jeden Knoten genau einmal durchläuft.

Problem: Entscheide für einen gegebenen Graphen $G = (V, E)$, ob er einen hamiltonischen Kreis enthält. Falls ja, gib einen solchen an.

2.2.3 Aufspannende Bäume und Wälder

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ einen aufspannenden Wald, d.h., finde für jede Zusammenhangskomponente von G einen aufspannenden Baum.

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ und gegebenen Gewichten c_{ij} für alle Kanten $ij \in E$ einen aufspannenden Wald mit minimalem/maximalen Gewicht.

2.2.4 Matchings

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Kanten aus E, so dass je zwei dieser Kanten nicht inzident sind.

2.2.5 Cliquen

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Knoten aus V, so dass je zwei dieser Knoten benachbart sind.

2.2.6 Färbungsproblem

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ die minimale Anzahl k von Farben, so dass sich die Knoten V mit einer der k Farben so färben lassen, dass benachbarte Knoten unterschiedlich gefärbt sind.

2.2.7 Kürzeste Wege

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ und gegebenen Gewichten/Längen c_{ij} für alle Kanten $ij \in E$ einen kürzesten Weg zwischen zwei gegebenen Knoten v und w aus V.

2.2.8 Traveling Saleman Problem (TSP)

Problem: Finde für einen gegebenen Graphen $G = (V, E)$ und gegebenen Gewichten/Längen c_{ij} für alle Kanten $ij \in E$ einen kürzesten hamiltonischen Kreis in G.

Kapitel 3

Datenstrukturen

3.1 Datentypen und Operationen, Datenstrukturen

Die Syntax einer algorithmischen Sprache beschreibt die formalen Regeln mit denen ein Algorithmus formuliert werden kann. Sie erklärt jedoch nicht die Bedeutung der Daten und die Operationen, die auf den Daten ausgeführt werden.

Definition 3.1.1 *Ein* **Datentyp** *(kurz: Typ) besteht aus*

- *dem Wertebereich des Typs und*
- *einer Menge von Operationen auf diesen Werten.*

Jede Programmiersprache verfügt über Standard-Datentypen.

Beispiel: Der C-Typ "int"

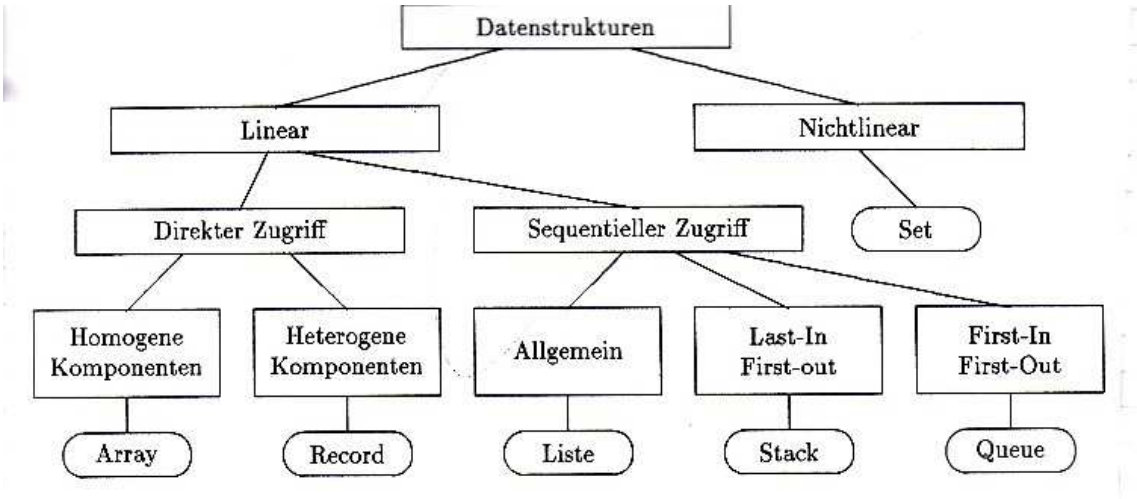
Wertebereich: $\{N_{min}, \ldots, 0, \ldots, N_{max}\}$, wobei N_{min} und N_{max} maschinenenabhängig sind

Operationen: $=$ (Zuweisung), $+$ (Addition), $-$ (Subtraktion), $*$ (Multiplikation), / (ganzzahlige Division), $==$ (Test auf Gleichheit), $!=$ (Test auf Ungleichheit)

Definition 3.1.2 Zusammengesetzte *oder* **strukturierte Datentypen** *setzen sich aus Datentypen gemäß bestimmter Strukturierungsmerkmale zusammen. Strukturierte Typen haben neben Wertebereich und Operationen*

- *Komponentendaten (das können strukturierte oder unstrukturierte Datentypen sein),*
- *Regeln, die das Zusammenwirken der Komponenten definieren.*

Lineare Datenstruktur*: hat eine vollständige Ordnung auf den Komponenten, d.h. es gibt eine erste und letzte Komponente und jede dazwischen hat einen Vor- und Nachfolger.*



Beispiel: Array
Kennzeichen:

- feste Komponentenzahl
- direkter Zugriff auf Komponenten mittels Indizes
- homogener Grundtyp

Zu den Operationen gehören:

- Ermittlung des Werts einer Komponente
- Zuweisung eines Werts an eine Komponente
- Zuweisung von Arrays
- Test auf Gleichheit

Weitere Datentypen sind z.B. Strings, Records, Listen, etc.

Beispiel: Liste
Eine Liste ist eine lineare Datenstruktur. Ihre Komponenten werden **Items** oder **Listenelemente** genannt. Das erste Element heißt **Anfang** oder **Kopf (head)** der Liste, das letzte Element heißt **Ende (tail)**, Kennzeichen der Datenstruktur Liste sind:

- veränderliche Länge (Listen können wachsen und schrumpfen),
- homogene Komponenten (elementar oder strukturiert),
- sequentieller Zugriff auf Komponenten durch ein (impliziten) **Listenzeiger**, der stets auf ein bestimmtes Element der Liste zeigt, und nur immer ein Listenelement vor oder zurück gesetzt werden kann.

Ein Beispiel sind Güterwaggons eines Zuges an einer Verladestation (= Listenzeiger), die nur einen Waggon zur Zeit beladen kann.

Typische Listenoperationen sind das Einfügen in eine Liste, der sequentielle Übergang zum nächsten Element, das Löschen eines Elements, usw.

3.2 Datenstrukturen zur Speicherung von Graphen

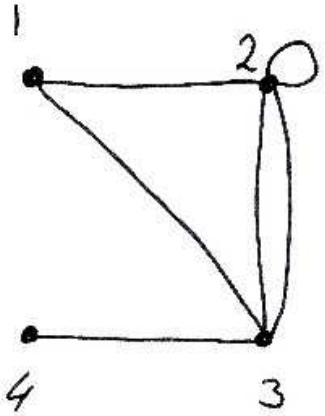
3.2.1 Kanten- und Bogenlisten

Die einfachste Art, einen Graphen oder Digraphen zu speichern, ist die Kantenliste für Graphen bzw. die Bogenliste für Digraphen. Ist $G = (V, E)$ ein Graph mit $n = |V|$ Knoten und $m = |E|$ Kanten, so sieht die Kantenliste wir folgt aus:

$$n, m, a_1, e_1, a_2, e_2, \ldots, a_m, e_m,$$

wobei a_i, e_i die beiden Endknoten der Kante i sind. Die Reihenfolge des Aufführens der Endknoten von i bzw. den Kanten ist beliebig. Bei Schleifen wird der Endknoten zweimal aufgelistet.

Beispiel:



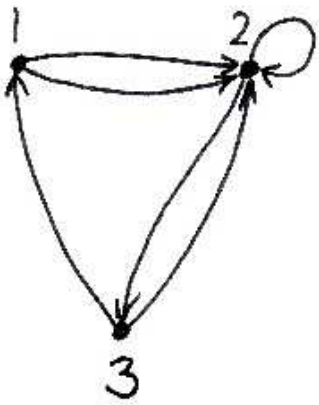
Dieser Graph hat die folgende Kantenliste:

$$4, 6, 1, 2, 2, 3, 4, 3, 3, 2, 2, 2, 1, 3.$$

Man beachte, dass hier die Reihenfolge der Knoten beliebig ist, da es sich um ungerichtete Kanten handelt. □

Bei der Bogenliste eines Digraphen verfahren wir genauso, müssen jedoch darauf achten, dass ein Bogen immer zunächst durch seinen Anfangs- und dann durch seinen Endknoten repräsentiert wird.

Beispiel:



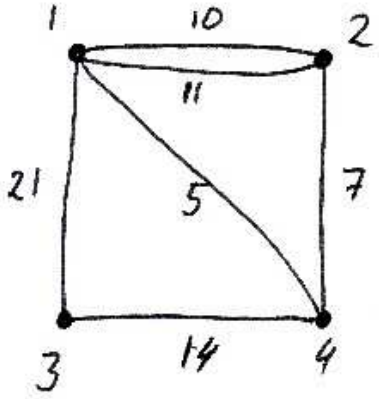
Die Bogenliste dieses Digraphen ist:

$$3, 6, 1, 2, 2, 3, 3, 2, 2, 2, 1, 2, 3, 1.$$

Man beachte, dass hier die Reihenfolge der Knoten fest ist, da es sich um gerichtete Kanten handelt. □

Haben die Kanten oder Bögen Gewichte, so repräsentiert man eine Kante (einen Bogen) entweder durch Anfangsknoten, Endknoten, Gewicht oder macht eine Kanten- bzw. Bogenliste wie oben und hängt an diese noch eine Liste mit den m Gewichten der Kanten $1, 2, \ldots, m$ an.

Beispiel:



Dieser gewichtete Graph ist in den beiden folgenden Kantenlisten mit Gewichten gespeichert:

$$4, 6, 1, 2, 10, 1, 2, 11, 2, 4, 7, 4, 3, 14, 3, 1, 21, 1, 4, 5$$

$$4, 6, 1, 2, 1, 2, 2, 4, 3, 4, 1, 4, 1, 3, 11, 10, 7, 14, 5, 21$$

Der Speicheraufwand einer Kanten- bzw. Bogenliste beträgt $2(m + 1)$ Speicherzellen, eine Liste mit Gewichten erfordert $3m + 2$ Zellen. □

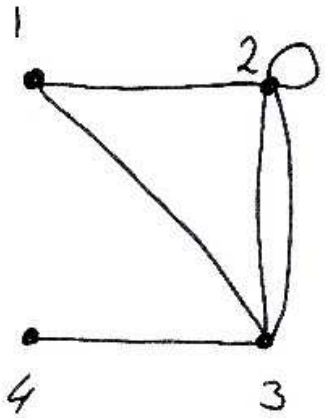
3.2.2 Adjazenzmatrizen

Definition 3.2.1 *Sei $G = (V, E)$ ein ungerichteter Graph mit $V = \{1, \ldots, n\}$, so ist die symmetrische Matrix $A \in \mathbb{R}^{n \times n}$ mit*

- $a_{ij} = a_{ji} =$ *Anzahl der Kanten, die i mit j, $i \neq j$, verbinden,*
- $a_{ii} = 2*$ *(Anzahl der Schleifen, die i enthalten), $i = 1, \ldots, n$,*

die **Adjazenzmatrix** *von G.*

Wegen der Symmetrie genügt es, die obere Dreiecksmatrix von A zu speichern.
Beispiel:



Die Adjazenzmatrix dieses Graphen ist:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

□

Hat G Kantengewichte und ist G einfach, so setzt man

- $a_{ij} =$ Gewicht der Kante, wenn $ij \in E$,
- $a_{ij} = 0, -\infty, +\infty$, anderenfalls.

Definition 3.2.2 *Die Adjazenzmatrix eines Digraphen $D = (V, A)$ mit $V = \{1, \ldots, n\}$ ohne Schleifen ist definiert durch*

- $a_{ii} = 0$, $i = 1, \ldots, n$,
- $a_{ij} =$ *Anzahl der Bögen in A mit Anfangsknoten i und Endknoten j.*

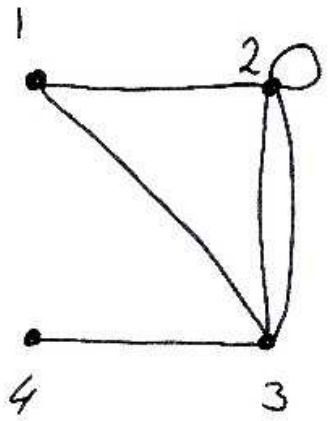
Bogengewichte werden wie im ungerichteten Fall behandelt.

Der Speicheraufwand für Adjazenzmatrizen beträgt n^2 Zellen (bzw. $\frac{n(n-1)}{2}$ für obere Dreiecksmatrix).

3.2.3 Adjazenzlisten

Definition 3.2.3 *Speichert man für einen Graphen $G = (V, E)$ die Anzahl von Knoten und für jeden Knoten seinen Grad und die Namen der Nachbarknoten, so nennt sich eine solche Datenstruktur* **Adjazenzliste** *von G.*

Beispiel:



Adjazenzliste:

	Knotennummer	Grad	Nachbarknoten
4	1	2	2, 3
	2	4	1, 3, 3, 2
	3	4	1, 2, 2, 4
	4	1	3

Die Liste der Nachbarknoten eines Knotens v heißt **Nachbarliste** von v. Jede Kante $ij \in E$, $i \neq j$, ist zweimal repräsentiert. Zur Speicherung von Gewichten kreiert man in der Regel eine Gewichtsliste.

Bei Digraphen geht man analog vor: nur speichert man lediglich die Nachfolger eines Knotens (jeder Bogen kommt also nur einmal vor).

Die Speicherung der Adjazenzliste eines Graphen erfordert $2n + 1 + 2m$ Speicherplätze, ihre Speicherung bei einem Digraphen erfordert $2n + 1 + m$ Speicherplätze.

Adjazenzlisten:

- kompakt, "gut für dünn besetzte Graphen",
- aber schlecht: Suchen, ob Kante ij in G enthalten ist.

3.3 Depth-First Search Algorithmus

(Tiefensuche: "so schnell wie möglich einen Knoten verlassen", beruht auf Datenstruktur Adjazenzliste)

Ziel: Berechnet für einen Graphen G die Anzahl der Komponenten, entscheidet, ob G Kreis enthält und findet für jede Komponente einen aufspannenden Baum

DFS-Algorithmus teilt die Kanten des Graphen in zwei Teilmengen. Kante xy heißt Vorwärtskante, wenn wir im Verfahren von einem markierten Knoten x entlang xy zum Knoten y gehen und y markieren. Anderfalls heißt xy Rückwärtskante.

Algorithmus: Depth-First Search

Input: Graph $G = (V, E)$ in Form einer Adjazenzliste, d.h. für jeden Knoten v ist eine Nachbarliste $N(v)$ gegeben.

Output: Kantenmenge T (= DFS-Baum, falls G zusammenhängend ist) und Kantenmenge B (mit $B \cup T = E$)

Alle Knoten seien unmarkiert.

1. Setze $T := \emptyset$, $B := \emptyset$.
2. Für alle $v \in V$ führe aus

   Ist v unmarkiert, dann **CALL SEARCH**(v).

   (Es existieren genau soviele Zusammenhangskomponenten in G wie hier **SEARCH**(v) aufgerufen wird.)

   END
3. Gib T und B aus.

Dieser Algorithmus enthält rekursiv folgendes Unterprogramm:

PROCEDURE SEARCH(v)

1. Markiere v.
2. Für alle Knoten $w \in N(v)$ führe aus:

   3. Ist w markiert und $vw \notin T$, setze $B := B \cup \{vw\}$.

   4. Ist w unmarkiert, setze $T := T \cup \{vw\}$ und **CALL SEARCH**(w).

   END

END SEARCH

Laufzeit des Verfahrens: $O(|V| + |E|)$

Es gilt:

(1) Die Menge aller Vorwärtskanten ist ein Wald von G, der in jeder Komponente einen aufspannenden Baum bildet.

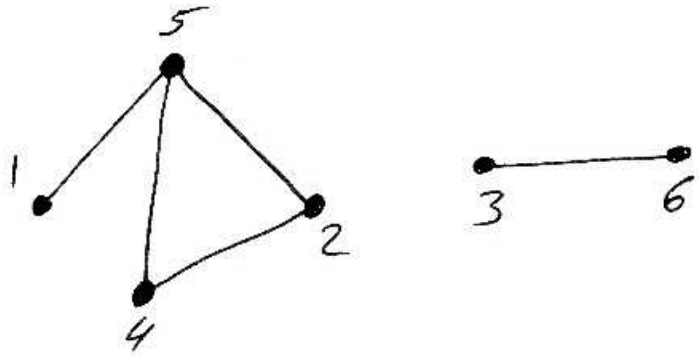
(2) G enthält einen Kreis $\Leftrightarrow B \neq \emptyset$.

Damit haben wir einen polynomialen Algorithmus zur Lösung des Entscheidungsproblems, ob G einen Kreis enthält.

DFS-Algorithmus ist konzipiert für die Datenstruktur einer Adjazenzliste!

3.4 Beispiel für DFS Algorithmus

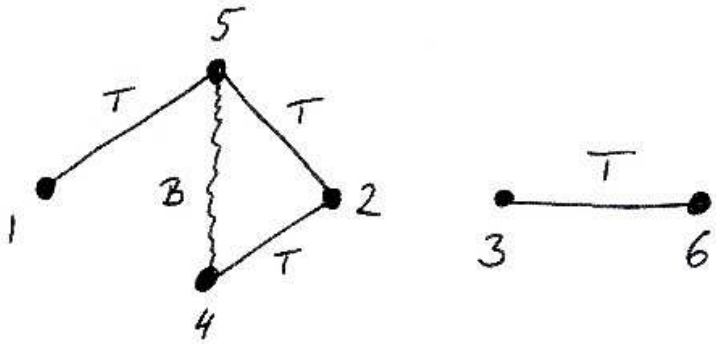
Aufgabe: Wende den DFS Algorithmus auf den folgenden Graphen an:



Dieser Graph hat die folgende Adjazenzliste:

	Knotennummer	Grad	Nachbarknoten
6	1	1	5
	2	2	4, 5
	3	1	6
	4	2	2, 5
	5	3	1, 2, 4
	6	1	3

Der Programmablauf auf der nächsten Seite wird die folgenden Kantenmarkierungen erzeugen. Die mit "T" markierten Kanten ergeben einen aufspannenden Baum. Da von der äußeren Schleife zweimal SEARCH aufgerufen wurde (für 1 und 3), hat der Graph 2 Zusammenhangskomponenten. Da $B \neq \emptyset$, besitzt der Graph einen Kreis (der durch jeweils die Kanten aus B geschlossen wurde).



Programmablauf

$T := \emptyset$, $B := \emptyset$

Wir müssen alle $v \in \{1, 2, 3, 4, 5, 6\}$ betrachten.

1 ist unmarkiert, also **CALL SEARCH**(1)

Markiere 1.

Wir müssen alle $v \in \{5\}$ betrachten.

5 ist unmarkiert, also $T := T \cup \{15\} = \{15\}$. **CALL SEARCH**(5)

Markiere 5.

Wir müssen alle $v \in \{1, 2, 4\}$ betrachten.

1 ist markiert, aber $51 \in T$

2 ist unmarkiert, also $T := T \cup \{52\} = \{15, 52\}$. **CALL SEARCH**(2)

Markiere 2.

Wir müssen alle $v \in \{4, 5\}$ betrachten.

4 ist unmarkiert, also $T := T \cup \{24\} = \{15, 52, 24\}$. **CALL SEARCH**(4)

Markiere 4.

Wir müssen alle $v \in \{5\}$ betrachten.

5 ist markiert und $45 \notin T$, also $B := B \cup \{45\} = \{45\}$.

5 ist markiert, aber $25 \in T$

4 ist markiert und $54 \notin T$, also $B := B \cup \{54\} = \{45\}$.

2 ist markiert

3 ist unmarkiert, also **CALL SEARCH**(3)

Wir müssen alle $v \in \{6\}$ betrachten.

6 ist unmarkiert, also $T := T \cup \{36\} = \{15, 52, 24, 36\}$. **CALL SEARCH**(6)

Markiere 6.

Wir müssen alle $v \in \{3\}$ betrachten.

3 ist markiert, aber $63 \in T$

4 ist markiert

5 ist markiert

6 ist markiert

END

Kapitel 4

Komplexität, Speicherung von Daten

4.1 Probleme, Komplexitätsmaße, Laufzeiten

In der Mathematik hat "Problem" viele Bedeutungen.

Definition 4.1.1 *Ein* **Problem** *ist eine allgemeine Fragestellung, bei der mehrere Parameter offen gelassen sind und für die eine Lösung oder Antwort gesucht wird.*

Ein Problem ist dadurch definiert, dass alle seine Parameter beschrieben werden und dass genau angegeben wird, welche Eigenschaften eine Antwort (Lösung) haben soll.

*Sind alle Parameter eines Problems mit expliziten Daten belegt, dann sprechen wir von einem Problembeispiel (***Instanz***).*

Beispiel:

TSP: Problem/Fragestellung: Finde einen kürzesten Hamiltonischen Kreis.

Offene Parameter: Anzahl Städte, Entfernungen

Definition 4.1.2 *Wir sagen, dass ein Algorithmus Problem* Π *löst, wenn er für jedes Problembeispiel* $I \in \Pi$ *eine Lösung findet.*

Ziel: Entwurf "effizienter" Algorithmen

Frage: Was ist Effizienz? → Komplexitätstheorie → Speicher- und Laufzeitkomplexität

Trivial: Laufzeit eines Algorithmus hängt ab von der "Größe" der Eingabedaten

Laufzeitanalyse erfordert eine Beschreibung, wie Problembeispiele dargestellt werden (Kodierungsschema).

4.2 Kodierungsschema

4.2.1 Ganze Zahlen

Ganze Zahlen werden binär dargestellt, d.h. wir schreiben

$$n = \pm \sum_{i=0}^{k} x_i \cdot 2^i,$$

mit $x_i \in \{0, 1\}$ und $k = \lfloor \log_2(|n|) \rfloor$.
D.h. die Kodierungslänge $\langle n \rangle$ einer ganzen Zahl n ist gegeben durch die Formel

$$\langle n \rangle = \lceil \log_2(|n| + 1) \rceil + 1 = \lfloor \log_2(|n|) \rfloor + 2,$$

wobei $+1$ wegen des Vorzeichens $+$ oder $-$.

4.2.2 Rationale Zahlen

Sei $r \in \mathbb{Q}$. Dann existieren $p \in \mathbb{Z}$ und $q \in \mathbb{Z}$, teilerfremd, mit $r = \frac{p}{q}$.

$$\langle r \rangle = \langle p \rangle + \langle q \rangle$$

4.2.3 Vektoren

Für $x = (x_1, \ldots, x_n)^\intercal \in \mathbb{Q}^n$ ist

$$\langle x \rangle = \sum_{i=1}^{n} \langle x_i \rangle.$$

4.2.4 Matrizen

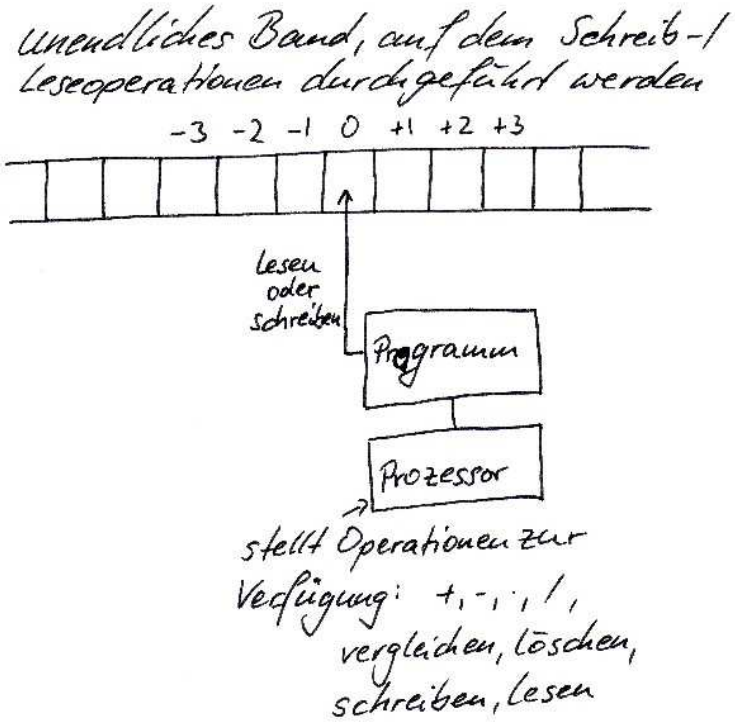
Für $A \in \mathbb{Q}^{m \times n}$ ist

$$\langle A \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle.$$

4.3 Rechnermodell

Nach Festlegung der Kodierungsvorschrift muss ein Rechnermodell entworfen werden, auf dem unsere Speicher- und Laufzeitberechnungen durchgeführt werden.

In der Komplexitätstheorie: Turing-Maschine (ein ganz normaler Computer)



Ablauf eines Algorithmus auf der Turing-Maschine:

Algorithmus A soll Problembeispiel I des Problems Π lösen. Alle Problembeispiele liegen in kodierter Form vor. Die Anzahl der Speicherplätze, die nötig sind, um I vollständig anzugeben, heißt **Inputlänge**, $\langle I \rangle$.

Der Algorithmus liest die Daten und beginnt dann, Operationen auszuführen, d.h. Zahlen zu berechnen, zu speichern, zu löschen.

Die Anzahl der Speicherplätze, die während der Ausführung des Algorithmus mindestens einmal benutzt werden, nennen wir **Speicherbedarf** von A zur Lösung von I. Die Laufzeit von A zur Lösung von I ist die Anzahl elementarer Operationen. Dazu zählen $+$, $-$, $*$, $/$, Vergleich, Löschen, Schreiben, Lesen von rationalen Zahlen.

Dies ist jedoch zu unpräzise! Zur Darstellung der entsprechenden Zahlen werden mehrere Bits benötigt. $\rightarrow$ Für jede Operation müssen wir mehrere Bits zählen.

Definition 4.3.1 Laufzeit *von A zur Lösung von I ist definiert durch die Anzahl elementarer Operationen, die A ausgeführt hat, um I zu lösen, multipliziert mit der größten Kodierungslänge der während der Berechnung aufgetretenen ganzen oder rationalen Zahl.*

Definition 4.3.2 ("worst-case Laufzeit") *(a) Sei A ein Algorithmus zur Lösung eines Problems Π. Die Funktion $f_A : \mathbb{N} \rightarrow \mathbb{N}$,*

$$f_A(n) = \max\{\textit{Laufzeit von } A \textit{ zur Lösung von } I : I \in \Pi, \langle I \rangle \leq n\}$$

heißt **Laufzeitfunktion** *von A.*

(b) Die Funktion $s_A : \mathbb{N} \to \mathbb{N}$ *definiert durch*

$$s_A(n) = \max\{\textit{Speicherbedarf von } A \textit{ zur Lösung von } I : I \in \Pi, \langle I \rangle \leq n\}$$

heißt **Speicherplatzfunktion** *von A.*

(c) Der Algorithmus A hat eine **polynomiale Laufzeit** *(kurz: A ist ein polynomialer Algorithmus), wenn es ein Polynom* $p : \mathbb{N} \to \mathbb{N}$ *gibt mit*

$$f_A(n) \leq p(n) \quad \textit{für alle } n \in \mathbb{N}.$$

Wir sagen f_A *ist von der* **Ordnung** *höchstens* n^k *(geschrieben* $f_A \in O(n^k)$*), falls das Polynom p den Grad k hat.*

(d) Der Algorithmus A hat polynomialen Speicherbedarf, falls es ein Polynom $q : \mathbb{N} \to \mathbb{N}$ *gibt mit*

$$s_A(n) \leq q(n) \quad \textit{für alle } n \in \mathbb{N}.$$

Übungsaufgabe: Berechne die Laufzeit- und Speicherplatzfunktion des folgenden Algorithmus:

Input: ganze Zahl n

FOR $i = 1$ to $\langle n \rangle$ DO

    $s := n \cdot n \cdot n$;

END

Gib s aus. □

4.4 Asymptotische Notation

4.4.1 Obere Schranken

Seien $f, g : \mathbb{N} \to \mathbb{R}$. Dann ist $f \in O(g)$ (gesprochen: "f ist in groß Oh von g"), wenn zwei positive Konstanten $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt:

$$|f(n)| \leq c|g(n)|.$$

Eine andere gebräuchliche Schreibweise hierfür ist $f = O(g)$.

Nehmen wir an, wir ermitteln die Rechenzeit $f(n)$ für einen bestimmten Algorithmus. Die Variable n kann z.B. die Anzahl der Ein- und Ausgabewerte sein, ihre Summe, oder auch die Größe eines dieser Werte. Da $f(n)$ maschinenabhängig ist, genügt eine a priori Analyse nicht. Jedoch kann man mit Hilfe einer a priori Analyse ein g bestimmen, so dass $f \in O(g)$.

Wenn wir sagen, dass ein Algorithmus eine Rechenzeit $O(g)$ hat, dann meinen wir damit folgendes: Wenn der Algorithmus auf unterschiedlichen Computern mit den gleichen Datensätzen läuft, und diese Größe n haben, dann werden die resultierenden Laufzeiten immer kleiner sein als eine Konstante mal $|g(n)|$. Bei der Suche nach der Größenordnung von f werden wir darum bemüht sein, das kleinste g zu finden, so dass $f \in O(g)$ gilt.

Ist f z.B. ein Polynom, so gilt:

Satz 4.4.1 *Für ein Polynom* $f(n) = a_m n^m + \ldots + a_1 n + a_0$ *vom Grade m gilt* $f \in O(n^m)$.

Beweis. Wir benutzen die Definition von $f(n)$ und eine einfache Ungleichung:

$$\begin{aligned} |f(n)| &\leq |a_m|n^m + \ldots + |a_1|n + |a_0| \\ &\leq \left(|a_m| + \frac{|a_{m-1}|}{n} + \ldots + \frac{|a_0|}{n^m}\right) n^m \\ &\leq (|a_m| + |a_{m-1}| + \ldots + |a_0|)\, n^m, n \geq 1 \end{aligned}$$

Setzt man $c = |a_m| + |a_{m-1}| + \ldots + |a_0|$ und $n_0 = 1$, so folgt unmittelbar die Behauptung. □

Die asymptotische Notation vernachlässigt Konstanten und Terme niedrigerer Ordnung (wie z.B. die Terme $a_k n^k$ mit $k < m$ im Polynom). Dafür gibt es zwei gute Gründe:

- Für große n ist die Größenordnung allein maßgebend. Z.B. ist bei $10^4 n$ und n^2 die erste Laufzeit für große n ($n \geq 10^4$) zu bevorzugen.
- Konstanten und Terme niedrigerer Ordnung hängen von vielen Faktoren ab, z.B. der gewählten Sprache oder der verwendeten Maschine, und sind daher meist nicht maschinenunabhängig.

Um Größenordnungen unterscheiden zu können, gibt es die o-Notation (lies: "klein oh Notation"). Sei $g : \mathbb{N} \to \mathbb{R}$. Dann bezeichnet

$$o(g) := \{f : \mathbb{N} \to \mathbb{R} : \forall c > 0\ \exists n_0 \in \mathbb{N} \text{ mit } f(n) < c \cdot g(n)\ \forall n \geq n_0\}$$

die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$, für die zu jeder positiven Konstanten $c \in \mathbb{R}$ ein $n_0 \in \mathbb{N}$ existiert, so dass für alle $n \geq n_0$ die Ungleichung $f(n) < c \cdot g(n)$ gilt. Gehört eine Funktion $f : \mathbb{N} \to \mathbb{R}$ zu dieser Menge, so schreibt man $f \in o(g)$ (gesprochen: "f ist in klein oh von g"). f heißt dann von (echt) kleinerer Größenordnung als g. Statt $f \in o(g)$ schreibt man gelegentlich auch $f = o(g)$.

Die gängigsten Größenordnungen für Rechenzeiten sind:

$$\begin{aligned} O(1) &< O(\log n) \\ &< O(n) \\ &< O(n \log n) \\ &< O(n^2) \\ &< O(n^k) \text{ für } k \in \mathbb{N} \text{ fest}, k \geq 3 \\ &< O(n^{\log n}) \\ &< O(2^n) \end{aligned}$$

Dabei bedeutet $O(f) < O(g)$, dass $f \in o(g)$, f also von kleinerer Größenordnung als g ist.

$O(1)$ bedeutet, dass die Anzahl der Ausführungen elementarer Operationen unabhängig von der Größe des Inputs durch eine Konstante beschränkt ist. Die ersten Größenordnungen haben eine wichtige Eigenschaft gemeinsam: sie sind durch ein Polynom beschränkt. (Sprechweisen: **polynomial beschränkt**, **polynomial**, **schnell**, **effizient**.) $O(n)$, $O(n^2)$ und $O(n^3)$ sind selbst Polynome, die man–bzgl. ihrer Grade–**linear**, **quadratisch** und **kubisch** nennt.

Es gibt jedoch keine ganze Zahl m, so dass n^m eine Schranke für 2^n darstellt, d.h. $2^n \notin O(n^m)$ für jede feste ganze Zahl m. Die Ordnung von 2^n ist $O(2^n)$.

Man sagt, dass ein Algorithmus mit der Schranke $O(2^n)$ einen **exponentiellen** Zeitbedarf hat. Für große n ist der Unterschied zwischen Algorithmen mit exponentiellem bzw. durch ein Polynom begrenztem Zeitbedarf ganz beträchtlich. Es ist eine große Leistung, einen Algorithmus zu finden, der statt eines exponentiellen einen durch ein Polynom begrenzten Zeitbedarf hat.

Die nachfolgende Tabelle zeigt, wie die Rechenzeiten der sechs typischen Funktionen anwachsen, wobei die Konstante gleich 1 gesetzt wurde. Wie man feststellt, zeigen die Zeiten vom Typ $O(n)$ und $O(n \log n)$ ein wesentlich schwächeres Wachstum also die anderen. Für sehr große Datenmengen ist dies oft das einzig noch verkraftbare Wachstum.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Folgendes Beispiel verdeutlicht den drastischen Unterschied zwischen polynomialen und exponentiellem Wachstum. Es zeigt, dass selbst enorme Fortschritte in der Rechnergeschwindigkeit bei exponentiellen Laufzeitfunktionen hoffnungslos versagen.

Beispiel: (Vergleich der Größenordnungen polynomial/expenentiell) Sei $f(n)$ die Laufzeitfunktion für ein Problem, und sei n_0 die Problemgröße, die man mit der jetzigen Technologie in einer bestimmten Zeitspanne t lösen kann. (Z.B. Berechnen kürzester Wege in einem Graphen mit n Knoten, $t = 60$ Minuten. Dann ist n_0 die Größe der Graphen, für die in einer Stunde die kürzesten Wege berechnet werden können.)

Wir stellen jetzt die Frage, wie n_0 wächst, wenn die Rechner 100 mal so schnell werden. Sei dazu n_1 die Problemgröße, die man auf den schnelleren Rechnern in der gleichen Zeitspanne t lösen kann.

Offenbar erfüllt n_0 die Gleichung $f(n_0) = t$ bei der alten Technologie, und $f(n_0) = t/100$ bei der neuen Technologie. Da n_1 bei der neuen Technologie $f(n_1) = t$ erfüllt, ergibt sich

$$f(n_1) = 100 \cdot f(n_0).$$

Ist die Laufzeitfunktion $f(n)$ polynomial, etwa $f(n) = n^k$, k fest, so folgt

$$n_1^k = 100 \cdot n_0^k, \text{ also } n_1 = \sqrt[k]{100} \cdot n_0,$$

d.h., man kann jetzt um den Faktor $\sqrt[k]{100}$ größere Probleme in der derselben Zeit lösen.

Ist dagegen die Laufzeitfunktion $f(n)$ exponentiell, etwa $f(n) = 2^n$, so folgt

$$2^{n_1} = 100 \cdot n_0, \text{ also } n_1 = n_0 + \log 100,$$

d.h., man kann jetzt nur eine um den **additiven** Term $\log 100$ größere Probleme in derselben Zeit lösen. Der Fortschritt macht sich also kaum bemerkbar.

4.4.2 Untere Schranken

Die O-Notation dient der Beschreibung oberer Schranken. Größenordnungen für untere Schranken werden mit der Ω-Notation ausgedrückt.

Seien $f, g : \mathbb{N} \to \mathbb{R}$. Dann ist $f \in \Omega(g)$ (gelesen: "f ist in Omega von g"), wenn zwei positive Konstanten $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt:

$$|f(n)| \geq c|g(n)|.$$

Manchmal kommt es vor, dass für die Laufzeit $f(n)$ eines Algorithmus gilt: $f \in \Omega(g)$ und $f \in O(g)$. Dafür benutzen wir folgende Schreibweise. Seien $f, g : \mathbb{N} \to \mathbb{R}$.

Dann ist $f \in \theta(g)$ (gelesen: "f ist in theta von g"), wenn es positive Konstanten $c_1, c_2 \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ gilt:

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|.$$

Falls $f \in \theta(g)$ gilt, dann ist g sowohl eine obere, als auch eine untere Schranke für f. Das bedeutet, dass die beiden Extremfälle–der beste und der schlechteste Fall–die gleiche Zeit brauchen (bis auf einen konstanten Faktor).

Beispiel: (Sequentielle Suche) Sei $f(n)$ die Anzahl von Vergleichen bei der sequentiellen Suche in einem unsortierten Array mit n Komponenten. Dann ist $f \in O(n)$, da man ja mit n Vergleichen auskommt. Andererseits muss man auch jede Komponente überprüfen, denn ihr Wert könnte ja der gesuchte Wert sein. Also $f \in \Omega(n)$ und damit $f \in \theta(n)$. □

Beispiel: (Matrixmultiplikation) Bei der Matrixmultiplikation von $n \times n$ Matrizen ergibt sich die Berechnung eines Eintrags c_{ij} von $C = A \cdot B$ gemäß $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. Sie erfordert also n Multiplikationen und $n - 1$ Additionen.

Insgesamt sind für ganz C also n^2 Einträge c_{ij} zu berechnen, und somit $n^2(n+n-1) = 2n^3 - n^2 = O(n^3)$ arithmetische Operationen insgesamt auszuführen.

Da jeder Algorithmus für die Matrixmultiplikation n^2 Einträge berechnen muss, folgt andererseits, dass jeder Algorithmus zur Matrixmultiplikation von zwei $n \times n$ Matrizen $\Omega(n^2)$ Operationen benötigt.

Es klafft zwischen $\Omega(n^2)$ und $O(n^3)$ also noch ein "Komplexitätslücke". Die schnellsten bekannten Algorithmen zur Matrixmultiplikation kommen mit $O(n^{2,376})$ Operationen aus.

Bezüglich der eingeführten o-Notation gilt:

Lemma 4.4.2 *$f \in o(g)$ genau dann, wenn $f \in O(g)$ und $f \notin \Omega(g)$.*

Beweis. "⇒": Sei $f \in o(g)$. Nach Definition ist dann $f = O(g)$ und nicht $g \in O(f)$. Zu zeigen ist also noch $f \notin \Omega(g)$). Angenommen, dies sei doch der Fall. Dann folgt

$$\begin{array}{lll} f \in \Omega(g) & \Rightarrow^{\text{Def}} & \exists c_1 > 0, n_1 \in \mathbb{N} \text{ mit } |f(n)| \geq c_1|g(n)|, \forall n \geq n_1 \\ & \Rightarrow & \exists c_1 > 0, n_1 \in \mathbb{N} \text{ mit } |g(n)| \leq \frac{1}{c_1}|f(n)|, \forall n \geq n_1 \\ & \Rightarrow & \exists c := \frac{1}{c_1} > 0, n_0 := n_1 \in \mathbb{N} \text{ mit } |g(n)| \leq c|f(n)|, \forall n \geq n_0 \\ & \Rightarrow^{\text{Def}} & g \in O(f) \end{array}$$

Dies ist ein Widerspruch dazu, dass $g \notin O(f)$.

"⇐": Sei $f \in O(g)$ und $f \notin \Omega(g)$. Zu zeigen ist, dass $f \in o(g)$, d.h. (nach Definition) $f \in O(g)$ (was nach Voraussetzung erfüllt ist) und $g \notin O(f)$. Angenommen, dies sei doch der Fall. Dann folgt

$$\begin{array}{lll} f \in O(g(n)) & \Rightarrow^{\text{Def}} & \exists c_2 > 0, n_2 \in \mathbb{N} \text{ mit } |g(n)| \leq c_2|f(n)|, \forall n \geq n_2 \\ & \Rightarrow & \exists c_2 > 0, n_2 \in \mathbb{N} \text{ mit } |f(n)| \geq \frac{1}{c_2}|g(n)|, \forall n \geq n_2 \\ & \Rightarrow & \exists c := \frac{1}{c_2} > 0, n_0 := n_2 \in \mathbb{N} \text{ mit } |f(n)| \geq c|g(n)|, \forall n \geq n_0 \\ & \Rightarrow^{\text{Def}} & f \in \Omega(g) \end{array}$$

im Widerspruch zur Voraussetzung $f \notin \Omega(g)$. □

4.5 Die Klassen $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-Vollständigkeit

4.5.1 Entscheidungsprobleme

Ein **Entscheidungsproblem** ist ein Problem, das nur zwei mögliche Antworten besitzt, "ja" oder "nein".

Beispiele: Ist n eine Primzahl? Enthält G einen Kreis? □

Informell: Klasse aller Entscheidungsprobleme, für die ein polynomialer Lösungsalgorithmus existiert, wird mit $\mathcal{P}$ bezeichnet.

Diese Definition ist informell: Gegeben ein Kodierungsschema E und ein Rechnermodell M. Π sei ein Entscheidungsproblem, wobei jede Instanz aus Π durch Kodierungsschema E kodierbar sei. Π gehört zur Klasse $\mathcal{P}$ (bzgl. E und M), wenn es einen auf M implementierbaren Algorithmus zur Lösung aller Problembeispiele aus Π gibt, dessen Laufzeitfunktion auf M polynomial ist.

Beispiele:

- Enthält G einen Kreis? "einfach"
- Enthält G einen hamiltonischen Kreis? $\rightarrow \mathcal{NP}$ "schwieriger"

Informell: Ein Entscheidungsproblem gehört zur Klasse $\mathcal{NP}$, wenn es folgende Eigenschaften hat: Ist die Antwort "ja" für $I \in \Pi$, dann kann Korrektheit dieser Aussage in polynomialer Laufzeit überprüft werden.

Beispiel: "Enthält G einen hamiltonischen Kreis?" □

Definition 4.5.1 *Ein Entscheidungsproblem Π gehört zu $\mathcal{NP}$, wenn es die folgende Eigenschaften hat:*

(a) Für jedes Problembeispiel $I \in \Pi$, für das die Antwort "ja" lautet, gibt es mindestens ein Objekt Q, mit dessen Hilfe die Korrektheit der "ja"-Antwort überprüft werden kann.

(b) Es gibt einen Algorithmus, der Problembeispiele $I \in \Pi$ und Zusatzobjekte Q als Input akzeptiert und der in einer Laufzeit, die polynomial in $\langle I \rangle$ ist, überprüft, ob Q ein Objekt ist, aufgrund dessen Existenz eine "ja"-Antwort für I gegeben werden muss.

$\Rightarrow \langle Q \rangle$ polynomial in $\langle I \rangle$

Die Probleme "Hat ein Graph G einen Kreis?", "Hat ein Graph G einen hamiltonischen Kreis?" sind somit in $\mathcal{NP}$. Hat nämlich G einen Kreis oder hamiltonischen Kreis, so wählen wir einen solchen als Objekt Q. Dann entwerfen wir einen polynomialen Algorithmus, der für einen Graphen G und eine zusätzliche Kantenmenge Q entscheidet, ob Q ein Kreis oder hamiltonischer Kreis von G ist.

Auch die Frage "Ist $n \in \mathbb{N}$ eine zusammengesetzte Zahl?" ist in $\mathcal{NP}$, denn liefern wir als "Objekt" zwei Zahlen $\neq 1$, deren Produkt n ist, so ist n keine Primzahl. Die Überprüfung der Korrektheit besteht somit in diesem Fall aus einer einzigen Multiplikation.

Bemerkungen:

- Obige Definition sagt nicht aus, wie Q zu finden ist!

- Laufzeit des Algorithmus in (b) ist polynomial in $\langle I \rangle$.
- Da der Algorithmus Q lesen muss, ist $\langle Q \rangle$ polynomial in $\langle I \rangle$.
- Auf die Frage "Hat die Gleichung $x^2 + y^2 = n$ eine Lösung in ganzen Zahlen?" ist $x = y = n\sqrt{0.5}$ kein geeignetes Zusatzobjekt.
- Definition ist unsymmetrisch in "ja" und "nein". $\mathcal{NP}$ impliziert nicht, dass auf die "nein"-Antwort ein polynomialer Algorithmus zur Überprüfung der Korrektheit dieser Aussage existiert.
- Probleme, die Negationen von Problemen aus $\mathcal{NP}$ sind, gehören zur Klasse co$\mathcal{NP}$. Zum Beispiel: "Hat G keinen Kreis?", "Ist $n \in \mathbb{N}$ eine Primzahl?" Man weiß, dass beide Probleme zur Klasse $\mathcal{NP} \cap$ co$\mathcal{NP}$ gehören. Vom Problem "Hat G keinen hamiltonischen Kreis?" weiß man nicht, ob es zu $\mathcal{NP}$ gehört.
- $\mathcal{NP}$: Nichtdeterministisch polynomialer Algorithmus: grob gesagt: Algorithmen, die am Anfang raten können, also einen nicht-deterministischen Schritt ausführen können (Orakel befragen), z.B. "Rate hamiltonischen Kreis". Gibt es keinen, STOP. Anderenfalls überprüfe Korrektheit des Objekts und antworte mit "ja".
- $\mathcal{P} \subseteq \mathcal{NP} \cap$ co$\mathcal{NP}$
- Offene Fragen:

$$\begin{aligned} \mathcal{P} &= \mathcal{NP} \cap \text{co}\mathcal{NP} \\ \mathcal{NP} &= \text{co}\mathcal{NP} \\ \mathcal{P} &\neq \mathcal{NP} \end{aligned}$$

Nächster Schritt: Charakterisierung besonders schwieriger Probleme in $\mathcal{NP}$.

Definition 4.5.2 *Gegeben seien zwei Entscheidungsprobleme* Π *und* Π'*. Eine polynomiale Transformation von* Π *in* Π' *ist ein polynomialer Algorithmus, der, gegeben ein (kodiertes) Problembeispiel* $I \in \Pi$*, ein (kodiertes) Problembeispiel* $I' \in \Pi'$ *produziert, so dass folgendes gilt:*

Die Antwort auf I *ist genau dann "ja", wenn die Antwort auf* I' *"ja" ist.*

Wenn Π in Π' polynomiell transformierbar ist und es einen polynomialen Algorithmus zur Lösung von Π' gibt, dann kann jede Instanz von Π auch polynomial gelöst werden.

Definition 4.5.3 *Ein Entscheidungsproblem* Π *heipt* $\mathcal{NP}$**-vollständig**, *falls* $\Pi \in \mathcal{NP}$ *und falls jedes andere Problem aus* $\mathcal{NP}$ *polynomial in* Π *transformiert werden kann.*

Jedes $\mathcal{NP}$-vollständige Entscheidungsproblem hat also folgende Eigenschaft. Falls Π in polynomialer Zeit gelöst werden kann, dann kann auch jedes andere Problem aus $\mathcal{NP}$ in polynomialer Zeit gelöst werden; in Formeln:

$$\Pi \quad \mathcal{NP} - \text{vollständig und } \Pi \in \mathcal{P} \Rightarrow \mathcal{P} = \mathcal{NP}.$$

Diese Eigenschaft zeigt, dass–bzgl. polynomialer Lösbarkeit–kein Problem in $\mathcal{NP}$ schwieriger ist als ein $\mathcal{NP}$-vollständiges.

Existieren $\mathcal{NP}$-vollständige Probleme? Ja.

Beispiele:

- 3-SAT: $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\ldots) \wedge \ldots = 1$
- "Ist $n \in \mathbb{N}$ prim?"

  $\in \mathcal{P}$?

  $\in \mathcal{NP} \cap$ co$\mathcal{NP}$?

  $\mathcal{NP}$-vollständig?

  Diese Fragen wurden 2003 von drei Indern geklärt. "Ist $n \in \mathbb{N}$ prim?" ist in $\mathcal{P}$.

4.5.2 Optimierungsprobleme

Bisher haben wir lediglich Entscheidungsprobleme betrachtet. Was ist mit Optimierungsproblemen?

Sei O ein Maximierungsproblem (Minimierungsproblem). Man legt zusätzlich zur Instanz I noch eine Schranke B fest und stellt die Frage:

Gibt es für I eine Lösung, deren Wert nicht kleiner (größer) als B ist?

Damit ist ein Optimierungsproblem polynomial in ein Entscheidungsproblem transformierbar.

Definition 4.5.4 *Ein Optimierungsproblem O heißt $\mathcal{NP}$-schwer, falls es ein $\mathcal{NP}$-vollständiges Entscheidungsproblem Π gibt, so dass Pi in polynomialer Zeit gelöst werden kann, wenn O in polynomialer Zeit gelöst werden kann.*

Beispiele:

- Rucksackproblem
- Maschinenbelegung
- stabile Menge maximaler Kardinalität
- Clique maximaler Kardinalität
- minimale Knotenüberdeckung

4.6 Zwei wichtige Techniken zur Behandlung von kombinatorischen Optimierungsproblemen

Die Techniken sind

(a) binäre Suche

(b) Skalierungstechnik

und werden jeweils an einem Beispiel erläutert.

4.6.1 Binäre Suche

Oft kann ein Optimierungsproblem in ein Entscheidungsproblem transformiert werden. Eine Lösung des letzteren kann zu einem polynomialen Verfahren für das Optimierungsproblem werden.

Beispiel:

TSP: Instanz I ist charakterisiert durch eine Distanzmatrix $c_{ij} \in \mathbb{Z} \forall i, j \in V$, $|V| = n$. Problem: "Finde einen bzgl. c minimalen Hamiltonischen Kreis in K_n."

TSP-Entscheidungsproblem: Input wie beim TSP, $B \in \mathbb{Z}$. Problem: "Existiert eine TSP-Tour der Länge $\leq B$?"

Lemma 4.6.1 *Die Existenz eines polynomialen Algorithmus zur Lösung des TSP-Entscheidungsproblems impliziert die Existenz eines polynomialen Algorithmus zur Lösung des TSP.*

Beweis. Die Länge eines beliebigen hamiltonischen Kreises in K_n ist ein ganzzahliger Wert im Intervall $[ns, nt]$, wobei $s = \min\{c_{ij} : i, j \in V, i \neq j\}$ und $t = \max\{c_{ij} : i, j \in V, i \neq j\}$.

Rufe TSP-Entscheidungsproblem mit $B = \lfloor \frac{n(s+t)}{2} \rfloor$ auf, passe obere Schranke nt oder untere Schranke ns je nach Ergebnis an und iteriere.

Insgesamt reichen demnach $\log(nt - ns)$ Iterationen aus. □

Setzen wir voraus, dass das Problem zu entscheiden, ob in einem Graphen $G = (V, E)$ ein hamiltonischer Kreis existiert, $\mathcal{NP}$-vollständig ist, so sieht man, dass das TSP $\mathcal{NP}$-schwer ist.

Beweis. Sei $G = (V, E)$ mit $|V| = n$. Setze $B := n$ und $c_{ij} = 1$, wenn $ij \in E$, und $c_{ij} = 2$, wenn $ij \notin E$. Rufe TSP-Entscheidungsproblem mit Input c und B auf. □

4.6.2 Skalierungstechnik

Sei $\mathcal{F} = \{x \in \{0,1\}^n : Ax \leq b\}$.

Gegeben sei ein Problem Π, wobei jede Instanz I von Π beschrieben werde durch Input $c \in \mathbb{Z}_+^n$, A_I, b_I. Sei $\mathcal{F}_I = \{x \in \{0,1\}^n : A_I x \leq b_I\}$.

Definition 4.6.2 (Orakel (AUG)) *Gegeben $c \in \mathbb{Z}_+^n$, ein Punkt $x^0 \in \mathcal{F}_I$. Finde $x^1 \in \mathcal{F}_I$ mit $c^\intercal x^1 < c^\intercal x^0$, oder versichere, dass kein solches x^1 existiert.*

Definition 4.6.3 (Orakel (OPT)) *Gegeben $c \in \mathbb{Z}^n$, ein Punkt $x^0 \in \mathcal{F}_I$. Finde $y \in \mathcal{F}_I$ mit $c^\intercal y = \min\{c^\intercal z : z \in \mathcal{F}_I\}$.*

Satz 4.6.4 *Es existiert ein Algorithmus, welcher (OPT) löst unter $O(n \log(C))$ Aufrufen eines Orakels zur Lösung von (AUG), wobei $C = \max\{|c_j| : j = 1, \ldots, n\}$.*

Beweis. (a) Wir zeigen zunächst, dass wir $c \geq 0$ annehmen können. Mit $c \in \mathbb{Z}^n$ assoziieren wir $\tilde{c} \in \mathbb{N}^n$ gemäß

$$\tilde{c}_j = \begin{cases} c_j, & \text{wenn } c_j \geq 0 \\ -c_j, & \text{sonst.} \end{cases}$$

Dann gilt $x \in \mathcal{F}_I \Leftrightarrow \tilde{x} \in \tilde{\mathcal{F}}_I$ mit

$$\tilde{x}_j = \begin{cases} x_j, & \text{wenn } c_j \geq 0 \\ 1 - x_j, & \text{sonst.} \end{cases}$$

$$\tilde{\mathcal{F}}_I = \{\tilde{x} : x \in \mathcal{F}_I\}$$

Daher ist $x \in \mathcal{F}_I$ eine minimale Lösung bzgl. c genau dann, wenn $\tilde{x} \in \tilde{\mathcal{F}}_I$ minimale Lösung bzgl. $\tilde{c}$. Ferner gilt für $t \in \mathbb{Z}^n$, $c^\intercal t < 0$, dass $\tilde{c}^\intercal \tilde{t} < 0$ mit

$$\tilde{t}_j = \begin{cases} t_j, & \text{wenn } c_j \geq 0 \\ 1 - t_j, & \text{sonst.} \end{cases}$$

(b) Sei $c \geq 0$. Sei M die Anzahl Bits zur Darstellung von C. Für $k = 1, \ldots, M$ definieren wir Probleme

$$P_k \quad \min\{c(k)^\intercal x : x \in \mathcal{F}_I\}.$$

$c(k) \in \mathbb{N}^n$ ist der Vektor, welcher aus c entsteht, indem wir die k führenden Bits von c wählen, d.h. $c(k) = (c_1(k), \ldots, c_n(k))$.

Algorithmus: Bit-Skalierung

1. Sei $x^0 \in \mathcal{F}_I$.

2. Für $k = 1, \ldots, M$ führe durch:

   2.1. Löse P_k mit Hilfe von (AUG) und Anfangslösung x^{k-1}.

   2.2. Sei x^k optimale Lösung von P_k.

Da P_M dem Problem (OPT) entspricht, ist der obige Algorithmus korrekt. Es verbleibt, eine Analyse über seine Laufzeit durchzuführen.

Da x^{k-1} optimal für $c(k-1)$ und x^k optimal für $c(k)$ sind, gilt:

$$0 \geq c(k)^\intercal (x^k - x^{k-1}) = [2c(k-1) + c^k]^\intercal (x^k - x^{k-1}) \geq (c^k)^\intercal (x^k - x^{k-1}) \geq -n,$$

wobei c^k der $0-1$-Vektor ist, der für jede Komponente das k-te Bit der entsprechenden Komponente von c hat. (Beachte, dass $2c(k-1)^\intercal (x^k - x^{k-1}) \geq 0$, da x^{k-1} optimal für $c(k-1)$ ist.)

D.h. für jedes k durchlaufen wir Schritt 2.1. höchstens n mal. Da $M = O(\log(C))$ folgt die Behauptung. □

Kapitel 5

Kürzeste Wege

Gegeben ein Digraph $D = (V, A)$ mit Bogengewichten $c(a)$.

Aufgabe: Finde einen Weg W von einem Knoten zu allen anderen oder zu einem bestimmten mit $c(W)$ minimal.

Problem:

- negative Gewichte
- Hamiltonische-Wege Problem

Unterscheidung der Algorithmen:

- nichtnegative Gewichte
- auch negative Gewichte möglich

5.1 Ein Startknoten, nichtnegative Gewichte

Idee: Dekomposition eine kürzesten (s, w)-Weges in einen kürzesten (s, u)-Weg plus kürzesten (u, w)-Weg.

Algorithmus: Dijkstra-Algorithmus

Input: Digraph $D = (V, A)$, Gewichte $c(a) \geq 0$ für alle $a \in A$, ein Knoten $s \in V$ (und ein Knoten $t \in V \setminus \{s\}$).

Output: Kürzeste gerichtete Wege von s nach v für alle $v \in V$ und ihre Länge (bzw. ein kürzester (s, t)-Weg).

Datenstrukturen:

$$\begin{array}{lcl} \text{DIST}(v) & = & \text{Länge des kürzesten } (s, v)\text{-Weges} \\ \text{VOR}(v) & = & \text{Vorgänger von } v \text{ im kürzesten } (s, v)\text{-Weg} \end{array}$$

0. Setze:

$$\begin{array}{lcl} \mathrm{DIST}(s) & = & 0 \\ \mathrm{DIST}(v) & = & +\infty \text{ für alle } v \in V \setminus \{s\} \\ \mathrm{VOR}(v) & = & s \text{ für alle } v \in V \setminus \{s\} \end{array}$$

1. Alle Knoten seien unmarkiert.

2. Bestimme einen unmarkierten Knoten u, so dass $\mathrm{DIST}(u) = \min\{\mathrm{DIST}(v) : v \text{ unmarkiert}\}$. Markiere u. (Falls $u = t$, gehe zu 5.)

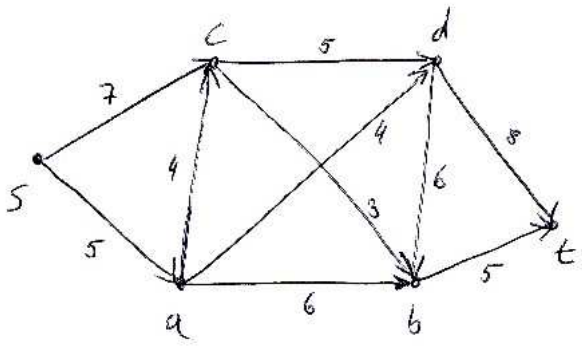
3. Für alle unmarkierten Knoten v mit $uv \in A$ führe aus:

$$\begin{array}{rcl} \text{Falls } \mathrm{DIST}(v) & > & \mathrm{DIST}(u) + c(uv) \text{ setze:} \\ \mathrm{DIST}(v) & := & \mathrm{DIST}(u) + c(uv) \text{ und } \mathrm{VOR}(v) := u. \end{array}$$

4. Sind noch nicht alle Knoten markiert, gehe zu 2.

5. Für alle markierten Knoten v ist $\mathrm{DIST}(v)$ die Länge des kürzesten (s, v)-Weges. Falls v markiert ist und $\mathrm{DIST}(v) < +\infty$, so ist $\mathrm{VOR}(v)$ der Vorgänger von v in einem kürzesten (s, v)-Weg, d.h. durch Rückwärtsgehen bis s kann ein kürzester (s, v)-Weg bestimmt werden. (Brechen wir das Verfahren nicht in Schritt 2 ab und gilt am Ende $\mathrm{DIST}(v) = +\infty$, so heißt das, dass es in D keinen (s, v)-Weg gibt.)

Beispiel.



	$DIST(s)$	$DIST(a)$	$DIST(b)$	$DIST(c)$	$DIST(d)$	$DIST(t)$	$Unmarkiert$
Start	0	∞	∞	∞	∞	∞	$\{s, a, b, c, d, t\}$
Markiere s	0	$5s$	∞s	$7s$	∞s	∞s	$\{a, b, c, d, t\}$
Markiere a	0	5	$11a$	7	$9a$	∞	$\{b, c, d, t\}$
Markiere c	0	5	$10c$	7	9	∞	$\{b, d, t\}$
Markiere d	0	5	10	7	9	$17d$	$\{b, t\}$
Markiere b	0	5	10	7	9	$15b$	$\{t\}$
(Markiere t)							$\emptyset$

Demnach ist der kürzeste Weg von s nach t

$$s \to c \to b \to t$$

und hat Länge 15.

Satz 5.1.1 *Der Dijkstra-Algorithmus arbeitet korrekt.*

Beweis. Über Induktion nach der Anzahl k der markierten Knoten zeigen wir: Ist v markiert, so enthält DIST(v) die Länge eines kürzesten (s, v)-Weges. Ist v unmarkiert, so enthält DIST(v) die Länge eines kürzesten (s, v)-Weges, wobei nur markierte Knoten als innere Knoten zugelassen sind.

$k = 1$: Dann ist nur s markiert und die Behauptung gilt sicherlich. Sei die Behauptung also richtig für $k \geq 1$ markierte Knoten.

Das Verfahren habe in Schritt 2 einen $(k+1)$-sten Knoten u markiert. Nach Ind.-voraussetzung ist DIST(u) die Länge eines kürzesten (s, u)-Weges, der als innere Knoten nur die ersten k markierte Knoten benutzen darf. Angenommen, es existiere ein kürzerer Weg P von s nach u. Dann muß er einen Bogen von einem markierten Knoten zu einem unmarkierten enthalten. Sei vw der erste derartige Bogen auf dem Weg P. Der Teilweg $\bar{P}$ von P von s nach w benutzt dann nur markierte Knoten als innere Knoten. Deshalb gilt $\text{DIST}(w) \leq c(\bar{P}) \leq c(P) < \text{DIST}(u)$. Aber dies widerspricht $\text{DIST}(u) \leq \text{DIST}(w)$ entsprechend der Auswahl von u in Schritt 2.

Es bleibt zu zeigen: Für die derzeit unmarkierten Knoten v ist DIST(v) die Länge eines kürzesten (s, v)-Weges, der nur markierte Knoten als innere hat. In Schritt 3 wird die Länge eines (s, v)-Weges über markierte Knoten verschieden von u verglichen mit der Länge eines (s, v)-Weges über markierte Knoten, der als vorletzten Knoten den Knoten u enthält. Der vorletzte Knoten auf einem kürzesten (s, v)-Weg ist aber entweder u oder ein anderer (vor u) markierter Knoten. Daraus folgt die Behauptung. □

In der Datenstruktur VOR merken wir uns zu jedem Knoten v seinen Vorgänger in einem kürzesten (s, v)-Weg. Einen kürzesten (s, v)-Weg erhält man dann gemäß:

$$v \quad \text{VOR}(v) \quad \text{VOR}(\text{VOR}(v)) \quad \ldots \quad \text{VOR}(\ldots \text{VOR}(v) \ldots) = s$$

Offensichtlich ist durch VOR( ) eine Arborenszenz definiert.

Satz 5.1.2 *Sei $D = (V, A)$ ein Digraph mit nichtnegativen Bogengewichten und $s \in V$. Es gibt eine Arboreszenz B mit Wurzel s, so dass für jeden Knoten $v \in V$, für den es einen (s, v)-Weg in D gibt, der (eindeutig bestimmte) gerichtete Weg in B von s nach v ein kürzester (s, v)-Weg ist.*

Anzahl Operationen des Dijkstra-Algorithmus: $O(|V|^2)$

5.2 Ein Startknoten, beliebige Gewichte

Das Problem, einen kürzesten Weg in einem Digraphen mit beliebigen Bogengewichten zu bestimmen, ist trivialerweise äquivalent zum Problem, einen längsten Weg in einem Digraphen mit beliebigen Gewichten zu finden. Wäre dies in polynomialer Zeit lösbar, so könnte man auch das $\mathcal{NP}$-vollständige Problem lösen, zu entscheiden, ob ein Digraph einen gerichteten hamiltonischen Weg hat.

Dijkstra-Algorithmus: funktioniert nur für nichtnegative Gewichte!

Jetzt: beliebige Gewichte, aber der Digraph ist azyklisch. In diesem Fall können die Knoten so numeriert werden, dass $(u, v) \in A \Rightarrow u < v$. D.h. wir haben eine Ordnung auf den Knoten, die während des Verfahrens benutzt werden kann.

Algorithmus: Moore-Bellman-Algorithmus für azyklische Digraphen

Input: Azyklischer Digraph $D = (V, A)$, Gewichte $c(a)$ für alle $a \in A$ (auch negative Gewichte sind zugelassen), ein Knoten $s \in V$.

Output: Kürzeste gerichtete Wege von s nach v für alle $v \in V$ und ihre Länge.

Datenstrukturen:

DIST(v), VOR(v) $\forall v \in V$. O.B.d.A. nehmen wir an, dass $V = \{1, 2, \ldots, n\}$ gilt und alle Bögen die Form (u, v) mit $u < v$ haben.

Setze:

$$\begin{array}{lcl} \text{DIST}(s) & = & 0 \\ \text{DIST}(v) & = & c(sv) \text{ für alle } v \in V \text{ mit } sv \in A \\ \text{DIST}(v) & = & +\infty \text{ für alle } v \in V \text{ mit } sv \notin A \\ \text{VOR}(v) & = & s \text{ für alle } v \in V \setminus \{s\} \end{array}$$

FOR $v = s + 2$ TO n DO

FOR $u = s + 1$ TO $v - 1$ DO

Falls $(u, v) \in A$ und DIST(u) + $c(uv)$ <DIST(v)

setze DIST(v) :=DIST(u) + $c(uv)$ und VOR(v) := u.

END

END

Falls DIST(v) $< \infty$, so enthält DIST(v) die Länge des kürzesten gerichteten Weges von s nach v, und aus VOR kann ein kürzester (s, v)-Weg entnommen werden. Falls DIST(v) $= +\infty$, so existiert in D kein (s, v)-Weg.

Satz 5.2.1 *Der Moore-Bellman-Algorithmus funktioniert für beliebige azyklische Digraphen mit beliebigen Gewichten.*

Beweis. Nach Voraussetzung haben alle Bögen in D die Form (u, v) mit $u < v$. Deshalb enthält jeder (s, v)-Weg mit $v > s$ als innere Knoten nur solche Knoten u mit $s < u < v$.

Durch Induktion nach $v = s + 1, \ldots, n$ zeigen wir, dass DIST(v) die Länge eines kürzesten (s, v)-Weges ist.

$v = s + 1$ o.k.

Sei die Behauptung richtig für alle Knoten $s + 1, \ldots, v$. Betrachte den Knoten $v + 1$. Ein kürzester $s, v + 1$-Weg besteht entweder nur aus dem Bogen $(s, v + 1)$ (falls vorhanden) oder aber er führt zu einem Knoten $s + 1 \leq u \leq v$ und benutzt anschließend die Kante $u, v + 1$. Dieses Minimum wir jedoch in der innersten Schleife berechnet. Daraus folgt die Behauptung. □

Anzahl Operationen des Moore-Bellman-Algorithmus: $O(|V|^2)$

Wichtig für spätere Kapitel ist eine Variante des Moore-Bellman-Algorithmus, welche für beliebige Gewichte und beliebige Digraphen funktioniert, sofern es keinen gerichteten **negativen** Kreis gibt.

Diese Variante heißt Yen-Variante und hat Laufzeit $O(|V|^3)$.

Wichtig: Mit dem Verfahren kann man tatsächlich testen, ob D einen negativen gerichteten Kreis enthält! Ein solcher wird in polynomialer Zeit gefunden.

Algorithmus: Yen-Variante

Input: Azyklischer Digraph $D = (V, A)$, Gewichte $c(a)$ für alle $a \in A$ (auch negative Gewichte sind zugelassen), ein Knoten $s \in V$.

Output: Kürzeste gerichtete Wege von s nach v für alle $v \in V$ und ihre Länge.

Datenstrukturen:

DIST(v), VOR(v) $\forall v \in V$. Zur Vereinfachung der Darstellung nehmen wir an, dass $V = \{1, 2, \ldots, n\}$ und $s = 1$ gilt.

Setze:

$$\begin{array}{lcl} \text{DIST}(s) & = & 0 \\ \text{DIST}(v) & = & +\infty \text{ für alle } v \in V \setminus \{s\} \\ \text{VOR}(v) & = & s \text{ für alle } v \in V \setminus \{s\} \end{array}$$

FOR $m = 0$ TO $n - 2$ DO

Falls m gerade:

FOR $v = 2$ TO n DO

FOR $u = 1$ TO $v - 1$ DO

Falls $(u, v) \in A$ und DIST$(u) + c(uv) <$DIST(v)

setze DIST$(v) :=$DIST$(u) + c(uv)$ und VOR$(v) := u$.

END

END

Falls m ungerade:

FOR $v = n - 1$ DOWNTO 1 BY -1 DO

FOR $u = n$ DOWNTO $v + 1$ BY -1 DO

Falls $(u, v) \in A$ und DIST$(u) + c(uv) <$DIST(v)

setze DIST$(v) :=$DIST$(u) + c(uv)$ und VOR$(v) := u$.

END

END

END

Falls DIST$(v) < \infty$, so enthält DIST(v) die Länge eines kürzesten (s, v)-Weges und aus VOR(v) kann wie üblich ein kürzester (s, v)-Weg entnommen werden. Falls DIST$(v) = +\infty$, so gibt es in D keinen (s, v)-Weg.

Satz 5.2.2 *Die Yen-Variante des Moore-Bellman-Algorithmus arbeitet korrekt, falls D keinen negativen gerichteten Kreis enthält.*

Beweis. (Skizze) DIST wird geeignet interpretiert. Sei wie im Algorithmus $s = 1$ und $V = \{1, 2, \ldots, n\}$. Ein Bogen mit (u, v) mit $u < v$ heißt **Aufwärtsbogen**, ein Bogen mit (u, v) mit $u > v$ heißt **Abwärtsbogen**. Ein Richtungswechsel auf einem (s, v)-Weg tritt auf, wenn auf einen Aufwärtsbogen ein Abwärtsbogen folgt oder umgekehrt.

Da $s = 1$, ist der erste Bogen immer ein Aufwärtsbogen.

Wir bezeichnen mit DIST(v, m) den Inhalt von DIST(v) nach Beendigung der m-ten Iteration der äußeren Schleife.

Behauptung: Für alle $v \in V \setminus \{s\}$ und $0 \leq m \leq n-2$ ist $\text{DIST}(v, m)$ der kürzeste aller (s, v)-Wege mit höchstens m Richtungswechseln.

Daraus folgt für $m = n - 2$ der Satz.

Die Behauptung für $m = 0$ folgt aus der Korrektheit des Moore-Bellman-Algorithmus.

Induktion nach m. Die Behauptung gelte für m. Z.z. ist sie für $m + 1$.

Fall 1. $m + 1$ ungerade ($m + 1$ gerade geht analog).

Der erste Bogen eines jeden Weges von s nach v ist Aufwärtsbogen. $m + 1$ ungerade $\Rightarrow$ Für einen Weg mit genau $m + 1$ Richtungswechslen ist der letzte Bogen ein Abwärtsbogen.

Nun führt man Induktion nach $v = n, n - 1, \ldots, 2$ durch. Jeder (s, n)-Weg ended mit Aufwärtsbogen. $\Rightarrow \text{DIST}(n, m+1) = \text{DIST}(n, m)$. Gelte die Behauptung $\forall v = n, \ldots, u$. Betrachte Knoten $u-1$. Der kürzeste Weg von s nach $u - 1$ mit genau $m + 1$ Richtungswechseln endet mit Abwärtsbogen $(w, u - 1)$ mit $w > u - 1$ bei $m + 1$ Richtungswechseln.

Der kürzeste Weg mit höchstens $m + 1$ Richtungswechseln berechnet sich dann gemäß der Formel, die im Algorithmus berechnet wird:

$$\min\{\text{DIST}(u - 1, m), \min_{w=u,\ldots,n}\{\text{DIST}(w, m + 1) + c(w, u - 1)\}\}$$

Erster Term: kürzester Weg mit höchstens m Richtungswechseln. Zweiter Term: Da hier der Weg $P(s, w)$, wenn er $m+1$ Richtungswechsel hatte mit Abwärtskante endet, hat auch der Weg $P(s, w) \cup (w, n - 1)$ nur $m + 1$ Richtungswechsel. Hatte $P(s, w)$ nur m Richtungswechsel, so hat $P(s, w) \cup (w, u - 1)$ höchstens $m + 1$ Richtungswechsel. □

Bemerkung: Sei D stark zusammenhängend ($\Rightarrow \exists$ gerichtete Wege von s zu jedem Knoten). Dann enthält D einen negativen Kreis genau dann, wenn bei einer zusätzlichen Ausführung der äußeren Schleife ($m = n - 1$) der Wert $\text{DIST}(v)$ für mindestens einen Knoten $v \in V$ geändert wird.

Kapitel 6

Sortieren in Arrays

Sortieralgorithmen gehören zu den am häufigsten angewendeten Algorithmen in der Datenverarbeitung. Man hatte daher bereits früh ein großes Interesse an der Entwicklung möglichst effizienter Sortieralgorithmen. Zu diesem Thema gibt es umfangreiche Literatur, nahezu jedes Buch über Algorithmen und Datenstrukturen beschäftigt sich mit Sortieralgorithmen, da sie besonders geeignet sind, Anfängern Programmiermethodik, Entwurf von Algorithmen, und Aufwandsanalyse zu lehren.

Wir erläutern die Sortieralgorithmen vor folgendem Hintergrund. Gegeben ist ein Datentyp **Item**.

```
struct Item {
  int key;
  // data components
};
```

Objekte dieses Typs sind "Karteikarten", z.B. aus einer Studentenkartei. Jede Karteikarte enthält neben den Daten (data components) einen Schlüssel (key) vom Typ int, z.B. die Matrikelnummer, nach denen sortiert oder gesucht werden kann. Die gesamte Kartei wird durch ein Array **vec** mit Grundtyp Item dargestellt. Wir reden daher im weiteren auch von **Komponenten** oder **Elementen** statt Karteikarten.

Die Wahl von int als Schlüsseltyp ist willkürlich. Hier kann jeder andere Typ verwendet werden, für den eine vollständige Ordnungsrelation definiert ist, also zwischen je zwei Werten a, b genau eine der Relationen $a < b$, $a = b$, $a > b$ gilt. Dies können z.B. auch Strings mit der lexikographischen Ordnung sein (Meier $<$ Müller), nur müsste dann der "eingebaute" Vergleich "$<$" von ganzen Zahlen durch eine selbstdefinierte Funktion zum Vergleich von Strings ersetzt werden.

6.1 Mergesort

Mergesort teilt das zu sortierende Array in zwei gleichgroße Teilfolgen, sortiert diese (durch rekursive Anwendung von Mergesort auf die beiden Teile) und mischt die sortierten Teile zusammen.

6.1.1 Mischen sortierter Arrays und der Algorithmus für Mergesort

Wir betrachten zunächst das Mischen von zwei bereits sortierten Arrays. Seien dazu v_1 und v_2 bereits sortierte Arrays der Länge m bzw. n mit Komponenten vom Typ Item. Diese sollen in das

Array v der Länge $m+n$ verschmolzen werden.

Dazu durchlaufen wir v_1 und v_2 von links nach rechts mit zwei Indexzeigern i und j wie folgt:

1. Initialisierung: $i = 0$, $j = 0$, $k = 0$;
2. Wiederhole Schritt 3 bis $i = m$ oder $j = n$.
3. Falls $v_1[i]$.key< $v_2[i]$.key, so kopiere $v_1[i]$ an die Position k von v und erhöhe i und k um 1. Anderenfalls kopiere $v_2[i]$ an die Position k von v und erhöhe i und k um 1.
4. Ist $i = m$ und $j < n$, so übertrage die restlichen Komponenten von v_2 nach v.
5. Ist $j = n$ und $i < m$, so übertrage die restlichen Komponenten von v_1 nach v.

Bei jedem Wiedereintritt in die Schleife 3 gilt die Invariante

$$\begin{array}{lllllllll} v[0].\text{key} & \leq & \ldots & \leq & v[k-1].\text{key} & & & & \\ & & & & v[k-1].\text{key} & \leq & v_1[i].\text{key} & \leq \ldots \leq & v_1[m-1].\text{key} \\ & & & & v[k-1].\text{key} & \leq & v_2[j].\text{key} & \leq \ldots \leq & v_2[n-1].\text{key} \end{array}$$

Hieraus folgt sofort, dass v am Ende aufsteigend sortiert ist.

Beispiel. Betrachten wir die Arrays

$$v_1 = [12, 24, 53, 63] \quad \text{und} \quad v_2 = [18, 25, 44, 72].$$

Die dazugehörige Folge der Werte von i, j, k, und v bei jedem Wiedereintritt in die Schleife 3 ist in der nachfolgenden Tabelle angegeben. Am Ende der Schleife ist $i = 4$ und Schritt 4 des Algorithmus wird ausgeführt, d.h. der Rest von v_2, also die 72, wird nach v übertragen.

i	j	k	$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$	$v[6]$	$v[7]$
1	0	1	12	–	–	–	–	–	–	–
1	1	2	12	18	–	–	–	–	–	–
2	1	3	12	18	24	–	–	–	–	–
2	2	4	12	18	24	35	–	–	–	–
2	3	5	12	18	24	35	44	–	–	–
3	3	6	12	18	24	35	44	53	–	–
4	3	7	12	18	24	35	44	53	63	–

Sei $C(m, n)$ die maximale Anzahl von Schlüsselvergleichen und $A(m, n)$ die maximale Anzahl von Zuweisungen von Komponenten beim Mischen.

Vergleiche treten nur in der Schleife 3 auf, und zwar genau einer pro Durchlauf. Da die Schleife maximal $n + m - 1$ mal durchlaufen wird, gilt

$$C(m, n) \leq m + n - 1.$$

Im folgenden sei **Merge** eine Funktion zum Mischen sortierter Arrays. Damit ergibt sich eine einfache Variante von Mergesort:

```
void MergeSort(Item v[], int first, int last) {
  int middle;
  if (first < last) {
    middle = (first+last)/2;     // divide v into 2 equal parts
```

```
    MergeSort(v,first,middle);   // sort the first part
    MergeSort(v,middle+1,last);  // sort the second part
    Merge(v,first,middle,last);  // merge the 2 sorted parts
  } // endif
}
```

Der Aufruf

```
MergeSort(a,0,n-1);
```

sortiert dann ein Array

```
Item a[n];
```

im gesamten Bereich von 0 bis $n-1$.

Die Korrektheit von MergeSort ergibt sich sofort durch vollständige Induktion nach der Anzahl n =last−first+1 der zu sortierenden Komponenten.

Ist $n = 1$, also last=first (Induktionsanfang), so wird im Rumpf von MergeSort nichts gemacht und das Array v ist nach Abarbeitung von MergeSort trivialerweise im Bereich first..last sortiert.

Ist $n > 1$, so sind first..middle und middle+1..last Bereiche mit weniger als n Elementen, die also nach Induktionsvoraussetzung durch die Aufrufe MergeSort(v,first,middle) und MergeSort(v,middle+1,last) korrekt sortiert werden. Die Korrektheit von Merge ergibt dann die Korrektheit von MergeSort.

Für das Standardbeispiel

$$a = [63, 24, 12, 53, 72, 18, 44, 35]$$

ergibt der Aufruf MergeSort(a,0,7) dann den in der folgenden Tabelle dargestellten Ablauf. Dabei beschreiben die Einrücktiefe die Aufrufhierarchie (Rekursionsbaum), und die Kästen die bereits sortieren Teile des Arrays.

MergeSort(a,0,7)	63	24	12	53	72	18	44	35
MergeSort(a,0,3)	63	24	12	53	72	18	44	35
MergeSort(a,0,1)	63	24	12	53	72	18	44	35
MergeSort(a,0,0)	63	24	12	53	72	18	44	35
MergeSort(a,1,1)	63	24	12	53	72	18	44	35
Merge(a,0,0,1)	24	63	12	53	72	18	44	35
MergeSort(a,2,3)	24	63	12	53	72	18	44	35
MergeSort(a,2,2)	24	63	12	53	72	18	44	35
MergeSort(a,3,3)	24	63	12	53	72	18	44	35
Merge(a,2,2,3)	24	63	12	53	72	18	44	35
Merge(a,0,1,3)	12	24	53	63	72	18	44	35
MergeSort(a,4,7)	12	24	53	63	72	18	44	35
MergeSort(a,4,5)	12	24	53	63	72	18	44	35
MergeSort(a,4,4)	12	24	53	63	72	18	44	35
MergeSort(a,5,5)	12	24	53	63	72	18	44	35
Merge(a,4,4,5)	12	24	53	63	18	72	44	35
MergeSort(a,6,7)	12	24	53	63	18	72	44	35
MergeSort(a,6,6)	12	24	53	63	18	72	44	35
MergeSort(a,7,7)	12	24	53	63	18	72	44	35
Merge(a,6,6,7)	12	24	53	63	18	72	35	44
Merge(a,4,5,7)	12	24	53	63	18	35	44	72
Merge(a,0,3,7)	12	18	24	35	44	53	63	72

6.1.2 Die Analyse von Mergesort

Wir ermitteln nun den worst-case Aufwand $C(n)$ für die Anzahl der Vergleiche und $A(n)$ für die Anzahl der Zuweisungen von Mergesort beim Sortieren eines Arrays mit n Komponenten.

Aus dem rekursiven Aufbau des Algorithmus ergeben sich folgende Rekursionsgleichungen für $C(n)$:

$$\begin{aligned} C(2) &= 1 \\ C(2n) &= 2 \cdot C(n) + C(n,n) \quad \text{für } n > 1. \qquad (*) \\ &= 2 \cdot C(n) + 2n - 1 \end{aligned}$$

Begründung: Das Sortieren eines 2-elementigen Arrays erfordert einen Vergleich. Das Sortieren eines Arrays der Länge $2n$ erfordert den Aufwand für das Sortieren von 2 Arrays der Länge n (rekursive Aufrufe von MergeSort für die beiden Teile), also $2 \cdot C(n)$, plus den Aufwand $C(n,n)$ für das Mischen (Aufruf von Merge).

Lemma 6.1.1 *Für $n = 2^q$ hat die Rekursionsgleichung (*) die Lösung*

$$C(2^q) = (q-1)2^q + 1.$$

Beweis. Der Beweis erfolgt durch Induktion nach q. Ist $q = 1$, so ist $C(2^1) = 1$ und $(q-1)2^q+1 = 1$ (Induktionsanfang). Also sei die Behauptung richtig für 2^r mit $1 \leq r \leq q$. Wir schließen jetzt auf

$q+1$:

$$\begin{aligned} C(2^{q+1}) &= 2 \cdot C(2^q) + 2 \cdot 2^q - 1 \quad \text{Rekursionsgleichung} \\ &= 2 \cdot [(q-1)2^q + 1] + 2 \cdot 2^q - 1 \quad \text{Induktionsvoraussetzung} \\ &= (q-1)2^{q+1} + 2 + 2^{q+1} - 1 \\ &= q \cdot 2^{q+1} + 1 \end{aligned}$$

□

Bezüglich der Anzahl $A(n)$ der Zuweisungen von Arraykompoenten ergibt sich analog:

$$A(2n) = 2 \cdot A(n) + \text{ Zuweisungen in Merge}$$

In Merge werden zunächst die Teile von v nach v_1 und v_2 kopiert. Dies erfordert $2n$ Zuweisungen. Für das Mergen sind dann wieder $A(n,n) = 2n$ Zuweisungen erforderlich. Also ergibt sich die Rekursionsgleichung

$$\begin{aligned} A(2) &= 4 \\ A(2n) &= 2 \cdot A(n) + 4n \quad \text{für } n > 1. \end{aligned}$$

Der gleiche Lösungsansatz liefert für $n = 2^q$:

$$A(n) = (q+1)2^q.$$

Wir erhalten damit folgenden Satz.

Satz 6.1.2 *Mergesort sortiert ein Array mit n Komponenten mit $O(n \log n)$ Vergleichen und Zuweisungen.*

Beweis. Sei $2^{q-1} < n \leq 2^q$. Dann gilt:

$$\begin{aligned} C(n) &\leq C(2^q) = (q-1)2^q + 1 \\ &< \log_2 n \cdot 2^q + 1 \\ &< (\log_2 n) \cdot 2n + 1 \\ &= 2n \log_2 n + 1 = O(n \log n) \end{aligned}$$

Analog:

$$\begin{aligned} A(n) &\leq A(2^q) = (q+1)2^q \\ &< (\log_2 n + 2)2^q \\ &< (\log_2 n + 2) \cdot 2n \\ &= 2n \log_2 n + 4n = O(n \log n) \end{aligned}$$

□

Betrachten wir zum Abschluss noch den Rekursionsaufwand und die Rekursionstiefe. Für $n = 2^q$ ist die Rekursionstiefe gerade $q = \log_2 n$. Für beliebige n ergibt sich wegen $2^{q-1} < n \leq 2^q$ eine Rekursionstiefe von höchstens $\log_2 2n = \log_2 n + 1$.

Die Anzahl der rekursiven Aufrufe ergibt sich als Summe entlang der Schichten des Rekursionsbaums zu

$$\sum_{i=0}^{q} 2^i = 2^{q+1} - 1 < 4n - 1 = O(n).$$

Rekursionsaufwand und Rekursionstiefe halten sich somit in vernünftigen Grenzen.

6.2 Beschleunigung durch Aufteilung: Divide-and-Conquer

Mergesort ist ein typisches Beispiel für die sogenannte "Beschleunigung durch Aufteilung". Dieses Prinzip tritt oft bei der Konzeption von Algorithmen auf. Daher hat man Interesse an einer allgemeinen Aussage über die Laufzeit in solchen Situationen.

6.2.1 Aufteilungs-Beschleunigungssätze

Gegeben ist ein Problem der Größe $a{\cdot}n$ mit der Laufzeit $f(a{\cdot}n)$. Dieses zerlegt man in b Teilprobleme der Laufzeit $f(n)$. Ist die Laufzeit für das Aufteilen respektive Zusammenfügen der Teillösungen $c \cdot n$, so ergibt sich die folgende Rekursionsgleichung und der folgende Satz.

Satz 6.2.1 *Seien $a > 0$, b, c natürliche Zahlen und sei folgende Rekursionsgleichung gegeben:*

$$\begin{aligned} f(1) &= \frac{c}{a} \\ f(a \cdot n) &= b \cdot f(n) + c \cdot n \quad \textit{für } n = a^q, q > 1. \end{aligned}$$

Dann gilt

$$f(n) \in \begin{cases} O(n) & , \textit{ falls } a > b \\ O(n \log n) & , \textit{ falls } a = b \\ O(n^{\log_a b}) & , \textit{ falls } a < b \end{cases}$$

Beweis. Für $n = a^q$ gilt

$$f(n) = \frac{c}{a} n \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i .$$

Dies zeigt man durch Induktion über q. Für $q = 0$ ist die Summe 0 und daher $f(1) = \frac{c}{a}$. Die Behauptung sei nun für q gezeigt. Dann ergibt sich im Induktionsschluss auf $q + 1$:

$$\begin{aligned} f(a^{q+1}) &= f(a \cdot a^q) \\ &= b \cdot f(a^q) + c \cdot a^q \quad \text{Rekursionsgleichung} \\ &= b \cdot \frac{c}{a} \cdot a^q \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i + c \cdot a^q \quad \text{Induktionsvoraussetzung} \\ &= \frac{c}{a} \cdot a^{q+1} \frac{b}{a} \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i + \frac{c}{a} \cdot a^{q+1} \\ &= \frac{c}{a} \cdot a^{q+1} \sum_{i=0}^{q} \left(\frac{b}{a}\right)^{i+1} + \frac{c}{a} \cdot a^{q+1} \\ &= \frac{c}{a} \cdot a^{q+1} \left(\sum_{i=1}^{q} \left(\frac{b}{a}\right)^i + 1 \right) \\ &= \frac{c}{a} \cdot a^{q+1} \sum_{i=0}^{q+1} \left(\frac{b}{a}\right)^i \end{aligned}$$

Also gilt

$$f(n) = \frac{c}{a} n \sum_{i=0}^{\log_a n} \left(\frac{b}{a}\right)^i .$$

Wir betrachten jetzt 3 Fälle:

Fall 1: $a > b$. Dann ist

$$\frac{b}{a} < 1 \Rightarrow \sum_{i=0}^{\log_a b} \left(\frac{b}{a}\right)^i < \sum_{i=0}^{\infty} \left(\frac{b}{a}\right)^i.$$

Die letzte Summe ist eine geometrische Reihe mit Wert $k_1 = \frac{1}{1-\frac{b}{a}} = \frac{a}{a-b}$. Also ist

$$f(n) < \frac{c \cdot k_1}{a} n \Rightarrow f(n) \in O(n).$$

Fall 2: $a = b$. Dann ist

$$f(n) = \frac{c}{a} n \sum_{i=0}^{\log_a n} 1 = \frac{c}{a} n(\log_a n + 1) = \frac{c}{a} n \log_a n + \frac{c}{a} n$$

Für $n \geq a$ ist $\log_a n \geq 1$ und daher

$$f(n) \leq \frac{c}{a} n \log_a n + \frac{c}{a} n \log_a n = 2\frac{c}{a} n \log_a n = \left(\frac{2c}{a} \log_a 2\right) \cdot n \log_2 n \in O(n \log_2 n).$$

Fall 3: $a < b$. Dann ist

$$\begin{aligned} f(n) &= \frac{c}{a} n \sum_{i=0}^{\log_a n} \left(\frac{b}{a}\right)^i \\ &= \frac{c}{a} a^q \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i \quad \text{da } n = a^q \\ &= \frac{c}{a} \sum_{i=0}^{q} b^i a^{q-i} = \frac{c}{a} \sum_{i=0}^{q} b^{q-i} a^i \\ &= \frac{c}{a} b^q \sum_{i=0}^{q} \left(\frac{a}{b}\right)^i \\ &< \frac{c}{a} b^q \sum_{i=0}^{\infty} \left(\frac{a}{b}\right)^i. \end{aligned}$$

Wie im Fall 1 ist $\sum_{i=0}^{\infty} \left(\frac{a}{b}\right)^i$ eine geometrische Reihe mit Wert $k_2 = \frac{b}{b-a}$. Also ist

$$f(n) < \frac{ck_2}{a} b^q = \frac{ck_2}{a} b^{\log_a n} = \frac{ck_2}{a} n^{\log_a b} \in O(n^{\log_a b}).$$

□

Offenbar ist der Fall 2 gerade der auf Mergesort zutreffende Fall.

Ein weiteres Beispiel für Divide-and-Conquer ist die Multiplikation von Dualzahlen.

6.2.2 Multiplikation von Dualzahlen

Als weitere Anwendung betrachten wir die Multiplikation von zwei n-stelligen Dualzahlen. Die traditionelle Methode erfordert $\Theta(n^2)$ Bit-Operationen. Durch Aufteilung und Beschleunigung erreicht

man $O(n^{\log_2 3}) = O(n^{1,59})$ Operationen. Dies ist nützlich bei der Implementation der Multiplikation belieb langer Dualzahlen, z.B. in Programmpaketen, die mit den Standard long int Zahlen nicht auskommen.

Seien x, y zwei n-stellige Dualzahlen, wobei n eine Zweierpotenz sei. Wir teilen x, y in zwei $\frac{n}{2}$-stellige Zahlen $x = a2^{n/2} + b$ und $y = c2^{n/2} + d$. Dann ist

$$xy = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd.$$

Man hat die Multiplikation also auf 4 Multiplikationen von $\frac{n}{2}$-stelligen Zahlen und einige Additionen und Shifts (Multiplikationen mit $2^{n/2}$ bzw. 2^n), die nur linearen Aufwand erfordern, zurückgeführt. Die führt zur Rekursionsgleichung

$$T(n) = 4T(n/2) + c_0 n,$$

mit der Lösung $T(n) = \Theta(n^2)$, also ohne Gewinn gegenüber der traditionellen Methode.

Die Anweisungen

$$\begin{aligned} u &:= (a+b)(c+d) \\ v &:= ac \\ w &:= bd \\ z &:= v2^n + (u - v - w)2^{n/2} + w \end{aligned}$$

führen jedoch zur Berechnung von $z = xy$ mit 3 Multiplikationen von Zahlen der Länge $\frac{n}{2}$ bzw. $\frac{n}{2} + 1$, da bei $a + b$ bzw. $c + d$ ein Übertrag auf die $(\frac{n}{2} + 1)$-te Position entstehen könnte. Ignorieren wir diesen Übertrag, so erhält man

$$T(n) = 3T(n/2) + c_1 n$$

mit der gewünschten Lösung $T(n) = \Theta(n^{\log_2 3})$.

Um den Übertrag zu berücksichtigen, schreiben wir $a + b$ und $c + d$ in der Form $a + b = \alpha 2^{n/2} + \bar{a}$ und $c + d = \gamma 2^{n/2} + \bar{c}$ mit den führenden Bits α, γ und den $\frac{n}{2}$-stelligen Resten $\bar{a}, \bar{c}$. Dann ist

$$(a + b)(c + d) = \alpha\gamma 2^n + (\alpha\bar{a} + \gamma\bar{c})2^{n/2} + \bar{a}\bar{c}.$$

Hierin tritt nur **ein** Produkt von $\frac{n}{2}$-stelligen Zahlen auf (nämlich $\bar{a}\bar{c}$. Der Rest sind Shifts bzw. lineare Operationen auf $\frac{n}{2}$-stelligen Zahlen (z.B. $\alpha\bar{a}$).

Daher erhält man insgesamt die Rekursionsgleichung

$$T(n) = 3T(n/2) + c_2 n,$$

wobei $c_2 n$ folgenden Aufwand enthält:

Additionen $a + b$, $c + d$:	$2 \cdot \frac{n}{2}$
Produkt $\alpha\gamma$:	1
Shift $\alpha\gamma$ auf $\alpha\gamma 2^n$:	n
Produkte $\alpha\bar{a}$, $\gamma\bar{c}$:	$2 \cdot \frac{n}{2}$
Addition $\alpha\bar{a} + \gamma\bar{c}$:	$\frac{n}{2} + 1$
Shift $\alpha\bar{a} + \gamma\bar{c}$ auf $(\alpha\bar{a} + \gamma\bar{c})2^{n/2}$:	$\frac{n}{2}$
Shift v auf $v2^n$:	n
Addition $u - v - w$:	$2(\frac{n}{2} + 1)$
Shift $u - v - w$ auf $(u - v - w)2^{n/2}$:	$\frac{n}{2}$
Addition zu z:	$2n$

Dieser Aufwand addiert sich zu $8,5n + 2 < 9n$ für $n \geq 2$. Also kann c_2 als 9 angenommen werden. Also Lösung erhält man nach dem Aufteilungsbeschleunigungssatz

$$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1,59}).$$

Der "Trick" bestand also darin, auf Kosten zusätzlicher Additionen und Shifts, eine "teure" Multiplikation von $\frac{n}{2}$-stelligen Zahlen einzusparen. Die rekursive Anwendung dieses Tricks ergibt dann die Beschleunigung von $\Theta(n^2)$ auf $\Theta(n^{1,59})$. Für die normale Computerarithmetik ($n = 32$) zahlt sich dieser Trick nicht aus, jedoch bedeutet er für Computerarithmetiken mit beliebigstelligen Dualzahlen, die meist softwaremäßig realisiert werden, eine wichtige Beschleunigung.

Das Verfahren läßt sich natürlich auch im Dezimalsystem anwenden. Wir geben ein Beispiel für $n = 4$:

$$x = 4217 \quad y = 5236$$

Dann ist

$$\begin{array}{lllllllll} a & = & 42, & b & = & 17 & \text{und} & a+b & = & 59; \\ c & = & 52, & d & = & 36 & \text{und} & c+d & = & 88. \end{array}$$

Es folgt

$$\begin{aligned} u &= (a+b)(c+d) = 55 \cdot 88 = 5192 \\ v &= ac = 42 \cdot 52 = 2184 \\ w &= bd = 17 \cdot 36 = 612 \\ xy &= v \cdot 10^4 + (u - v - w) \cdot 10^2 + w \\ &= 2184 \cdot 10^4 + 2396 \cdot 10^2 + 612 \\ &= 21.840.000 + 239.600 + 612 \\ &= 22.080.212 \end{aligned}$$

Auf ähnliche Weise wie die Multiplikation von Zahlen läßt sich auch die Multiplikation (großer) $n \times n$ Matrizen beschleunigen. Hier erhält man die Rekursionsgleichung

$$T(2n) = 7T(n) + 14n$$

mit der Lösung $\Theta(n^{\log_2 7}) = \Theta(n^{2,81})$, also eine Beschleunigung gegenüber der normalen Methode mit dem Aufwand $\Theta(n^3)$. Hier lassen sich noch weitere Beschleunigungen erzielen. Der momentane "Rekord" steht bei $O(n^{2,39})$.

6.3 Quicksort

Quicksort basiert (im Gegensatz zu Mergesort) auf **variabler** Aufteilung des Eingabearrays. Es wurde 1962 von Hoare entwickelt. Es benötigt zwar im Worst Case $\Omega(n^2)$ Vergleiche, im Mittel jedoch nur $O(n \log n)$ Vergleiche, und ist aufgrund empirischer Vergleiche allen anderen $O(n \log n)$ Sortierverfahren überlegen.

6.3.1 Der Algorithmus

Wir geben zunächst eine Grobbeschreibung von Quicksort an.

1. Gegeben ist $v[]$ mit vSize Komponenten.

2. Wähle eine beliebige Komponente v[pivot].

3. Zerlege das Array v in zwei Teilbereiche $v[0] \ldots v[k-1]$ und $v[k+1] \ldots v$[vSize−1] mit

   a) $v[i]$.key< v[pivot].key für $i = 0, \ldots, k-1$

   a) $v[i]$.key= v[pivot].key

   a) $v[j]$.key> v[pivot].key für $i = k+1, \ldots,$ vSize−1

4. Sofern ein Teilbereich aus mehr als einer Komponente besteht, so wende Quicksort rekursiv auf ihn an.

Die Korrektheit des Algorithmus folgt leicht durch vollständige Induktion. Die Aufteilung erzeugt Arrays kleinerer Länge, die nach Induktionsvoraussetzung durch die Aufrufe von Quicksort in Schritt 4 korrekt sortiert werden. Die Eigenschaften 3a)-3c) ergeben dann die Korrektheit für das ganze Array.

In unserem Standardbeispiel ergibt sich, falls man stets die mittlere Komponente wählt (gekennzeichnet durch *) die in der folgenden Abbildung dargestellte Folge von Zuständen (jeweils nach der Aufteilung). Die umrahmten Bereiche geben die aufzuteilenden Bereiche an.

Input	**63**	**24**	**12**	**53***	**72**	**18**	**44**	**35**
1. Aufteilung	**18**	**24**	**12***	**35**	**44**	**53**	**72***	**63**
2. Aufteilung	**12**	**24**	**18***	**35**	**44**	**53**	**63**	**72**
3. Aufteilung	**12**	**18**	**24**	**35***	**44**	**53**	**63**	**72**
Output	**12**	**18**	**24**	**35**	**44**	**53**	**63**	**72**

Wir betrachten nun die Durchführung der Aufteilung im Detail. Da sie rekursiv auf stets andere Teile des Arrays v angewendet wird, betrachten wir einen Bereich von loBound bis hiBound.

Grobbeschreibung der Aufteilung

1. Gegeben ist v und der Bereich zwischen loBound und hiBound.

2. Wähle eine Komponente v[pivot].

3. Tausche v[pivot] mit v[loBound].

4. Setze Indexzeiger loSwap auf loBound+1 und hiSwap auf hiBound.

5. Solange loSwap<hiSwap wiederhole:

   5.1 Inkrementiere ggf. loSwap solange, bis v[loSwap].key> v[loBound].key.

   5.2 Dekrementiere ggf. hiSwap solange, bis v[hiSwap].key< v[loBound].key.

   5.3 Falls loSwap<hiSwap, so vertrausche v[loSwap] und v[hiSwap].

6. Tause v[loBound] und v[hiSwap].

Die folgende Abbildung illustriert diese Aufteilung. l und h geben die jeweilige Position von loSwap und hiSwap an.

Input	63	24	12	53	72	18	44	35
Schritt 2	63	24	12	53^*	72	18	44	35
Schritt 3	53^*	24	12	63	72	18	44	35
Schritt 4	53^*	24^l	12	63	72	18	44	35^h
Am Ende von 5.1	53^*	24	12	63^l	72	18	44	35^h
Am Ende von 5.2	53^*	24	12	63^l	72	18	44	35^h
Schritt 5.3	53^*	24	12	35^l	72	18	44	63^h
Am Ende von 5.1	53^*	24	12	35	72^l	18	44	63^h
Am Ende von 5.2	53^*	24	12	35	72^l	18	44^h	63
Schritt 5.3	53^*	24	12	35	44^l	18	72^h	63
Am Ende von 5.1	53^*	24	12	35	44	18	72^{lh}	63
Am Ende von 5.2	53^*	24	12	35	44	18^h	72^l	63
Schritt 6	18	24	12	35	44	53^*	72	63

Bei jedem Eintritt in Schleife 5 gilt die Invariante

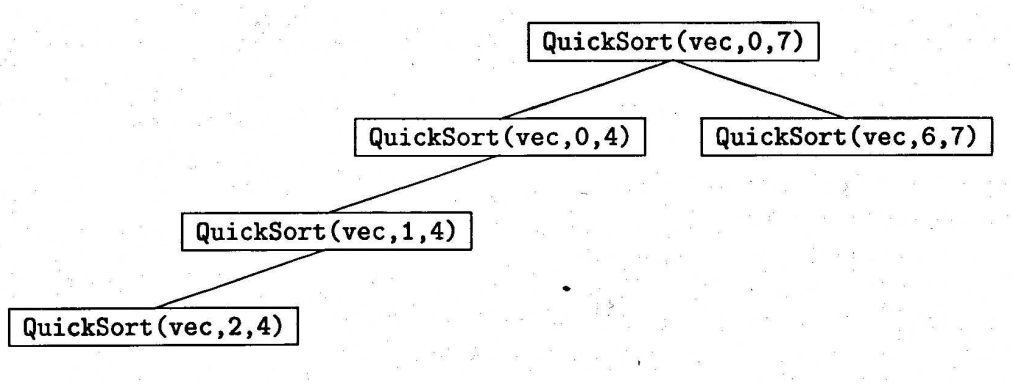
$$\begin{aligned} v[i] \leq v[\text{loBound}].\text{key} \quad &\text{für} \quad i = \text{loBound} \ldots \text{loSwap} - 1, \\ v[j] > v[\text{loBound}].\text{key} \quad &\text{für} \quad j = \text{hiSwap} + 1 \ldots \text{hiBound}, \\ \text{loSwap} < \text{hiSwap} \quad &\Rightarrow \quad v[\text{loSwap}].\text{key} \leq v[\text{loBound}].\text{key} \leq v[\text{hiSwap}].\text{key} \end{aligned}$$

Beim Austritt gilt zusätzlich

$$\text{loSwap} \geq \text{hiSwap}, \quad v[\text{hiSwap}].\text{key} \leq v[\text{loBound}].\text{key},$$

so dass der Tausch in Schritt 6 die Pivot-Komponente (derzeit gespeichert an der Stelle loBound) genau an die richtige Stelle tauscht.

Da die Aufteilung (im Unterschied zu Mergesort) variabel ist, hat der Rekursionsbaum bei Quicksort im Allgemeinen Teilbäume unterschiedlicher Höhe. Im Standardbeispiel ergibt sich der in der folgenden Abbildung dargestellte Baum.



Man sieht, dass der Baum am tiefsten wird, wenn als Vergleichselement jeweils das kleinste oder größte Element des aufzuteilenden Bereichs gewählt wird. In diesem Fall entartet der Rekursionsbaum zu einer Liste (keine Verzweigungen). Die Rekursionstiefe kann also bis zu $n-1$ betragen.

Wir zeigen durch Induktion, dass auch die Anzahl $R(n)$ der rekursiven Aufrufe bei einem Arraybereich der Länge n höchstens $n-1$ beträgt.

Für $n = 1$ erfolgt kein Aufruf, also gilt $R(1) = 0$ (Induktionsanfang). Für $n > 1$ bewirkt der erste Aufruf eine Aufteilung in Bereiche mit n_1 und n_2 Komponenten, wobei $n_1 + n_2 = n - 1$ gilt, da die Vergleichskomponente wegfällt. Also sind $n_1, n_2 < n$ und man erhält

$$\begin{aligned} R(n) &= 1 + R(n_1) + R(n_2) \\ &\leq 1 + (n_1 - 1) + (n_2 - 1) \quad \text{Induktionsvoraussetzung} \\ &= n_1 + n_2 - 1 \\ &\leq n - 1, \end{aligned}$$

also $R(n) \leq n - 1$.

6.3.2 Worst-Case Aufwand von Quicksort

Vergleiche von Schlüsseln treten bei Quicksort nur bei den Aufteilungen auf. Dabei muss jeder andere Schlüssel mit dem Vergleichsschlüssel verglichen werden, also erfolgen bei einem Bereich von n Komponenten $n-1$ Vergleiche.

Wird nun jeweils der größte bzw. kleinste Schlüsselwert als Vergleichsschlüssel gewählt, so verkleinert sich der Bereich jeweils nur um ein Element und man erhält

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \quad \text{Vergleiche.}$$

Also gilt für die Worst-Case Anzahl $C(n)$ von Vergleichen

$$C(n) = \Omega(n^2).$$

Entsprechend ergibt sich für die Worst-Case Anzahl $A(n)$ von Zuweisungen

$$A(n) = \Omega(n^2).$$

6.3.3 Mittlerer Aufwand von Quicksort

Qicksort ist also im Worst-Case schlecht. Erfahrungsgemäß ist Quicksort aber sehr schnell im Vergleich zu anderen $\Omega(n^2)$ Sortierverfahren wie Bubblesort u. a. Dies liegt daran, dass der Worst-Case nur bei wenigen Eingabefolgen auftritt.

Man wird daher Quicksort gerechter, wenn man nicht den Worst-Case betrachtet, sondern den Aufwand über all möglichen Eingabefolgen mittelt, also den **mittleren Aufwand** $\bar{C}(n)$ bei Gleichverteilung aller $n!$ Reihenfolgen der Schlüssel $1, 2, \ldots, n$ betrachtet. Gleichverteilung bedeutet hier, dass jede Reihenfolge (Permutation) der Werte $1, \ldots, n$ mit dem gleichen Gewicht (nämlich 1) in das Mittel eingeht.

Sei Π die Menge aller Permutationen von $1, \ldots, n$. Für $\pi \in \Pi$ sei $C(\pi)$ die Anzahl von Vergleichen, die Quicksort benötigt, um π zu sortieren. Dann ist

$$\bar{C}(n) = \frac{1}{n!} \sum_{\pi \in \Pi} C(\pi).$$

Wir werden jetzt $\bar{C}(n)$ nach oben abschätzen. Dafür teilen wir die Menge Π aller Permutationen in die Mengen Π_1, ..., Π_n, wobei

$$\Pi_k = \{\pi \in \Pi : \text{ das Vergleichselement hat den Wert } k\}.$$

Für $n = 3$ ergibt sich $\Pi_1 = \{213, 312\}$, $\Pi_2 = \{123, 321\}$ und $\Pi_3 = \{132, 231\}$.

In Π_k ist das Vergleichselement fest vorgeschrieben, die anderen Komponenten können jedoch in jeder Reihenfolge auftreten. Also ist

$$|\Pi_k| = (n-1)! \quad \text{für} \quad k = 1, \ldots, n.$$

Für alle $\pi \in \Pi_k$ ergibt die erste Aufteilung in Quicksort die Teilarrays bestehend aus einer Permutation π_1 von $1, 2, \ldots, k-1$ und einer Permuation π_2 von $k+1, \ldots, n$ (da ja das Vergleichselement gerade k ist).

$Z(\pi)$ sei die Anzahl der Vergleiche mit der π in die Teile π_1 und π_2 zerlegt wird. Dann ist für alle $\pi \in \Pi_k$

$$C(\pi) = Z(\pi) + C(\pi_1) + C(\pi_2).$$

Dabei ist $Z(\pi) \leq n$. Summiert man über alle $\pi \in \Pi_k$, so ergibt sich wegen $|\Pi_k| = (n-1)!$

$$\sum_{\pi \in \Pi_k} C(\pi) = \sum_{\pi \in \Pi_k} Z(\pi) + \sum_{\pi \in \Pi_k} C(\pi_1) + \sum_{\pi \in \Pi_k} C(\pi_2) =: S_1 + S_2 + S_3.$$

Hierin ist

$$S_1 \leq \sum_{\pi \in \Pi_k} n = (n-1)! \cdot n = n!.$$

Wenn π alle Permutationen aus Π_k durchläuft, entstehen bei π_1 alle Permuationen von $1, \ldots, k-1$, und zwar jede $(n-1)!/(k-1)!$ mal, da Π_k ka insgesamt $(n-1)!$ Permutationen enthält. Also ist

$$\begin{aligned} S_2 &= \frac{(n-1)!}{(k-1)!} \sum_{\pi_1 \text{ Permutation von } 1,\ldots,k-1} C(\pi_1) \\ &= (n-1)! \bar{C}(k-1). \end{aligned}$$

Entsprechend folgt

$$S_3 = (n-1)! \bar{C}(n-k).$$

Durch Zusammensetzen aller Gleichungen bzw. Ungleichungen ergibt sich

$$\begin{aligned}
\bar{C}(n) &= \frac{1}{n!}\sum_{\pi\in\Pi} C(\pi) \\
&= \frac{1}{n!}\sum_{k=1}^{n}\sum_{\pi\in\Pi_k} C(\pi) \\
&\leq \frac{1}{n!}\sum_{k=1}^{n}[n! + (n-1)!\bar{C}(k-1) + (n-1)!\bar{C}(n-k)] \\
&= \frac{n!}{n!}\sum_{k=1}^{n} 1 + \frac{(n-1)!}{n!}\sum_{k=1}^{n}\bar{C}(k-1) + \frac{(n-1)!}{n!}\sum_{k=1}^{n}\bar{C}(n-k) \\
&= n + \frac{1}{n}\sum_{k=1}^{n}\bar{C}(k-1) + \frac{1}{n}\sum_{k=1}^{n}\bar{C}(k-1) \\
&= n + \frac{2}{n}\sum_{k=1}^{n}\bar{C}(k-1).
\end{aligned}$$

Wir haben damit eine Rekursionsgleichung für $\bar{C}(n)$ gefunden. Beachtet man noch die Anfangswerte

$$\bar{C}(0) = \bar{C}(1) = 0, \bar{C}(2) = 1,$$

so ist

$$\bar{C}(n) \leq n + \frac{2}{n}\sum_{k=2}^{n-1}\bar{C}(k) \quad \text{für} \quad n \geq 2.$$

Lemma 6.3.1 *Für die Lösung $r(n)$ der Rekursionsgleichung*

$$r(n) = n + \frac{2}{n}\sum_{k=2}^{n-1} r(k) \quad \textit{für} \quad n \geq 2.$$

mit den Anfangswerten $r(0) = r(1) = 0$, $r(2) = 1$ gilt für alle $n \geq 2$

$$r(n) \leq c \cdot n \cdot \ln n \quad \textit{mit} \quad c = 2.$$

Beweis. Der Beweis erfolgt durch vollständige Induktion nach n.

Induktionsanfang: Für $n = 2$ ist $r(2) = 1$. Andererseits ist $c \cdot 2\ln 2 \approx 1,39$. Also gilt der Induktionsanfang.

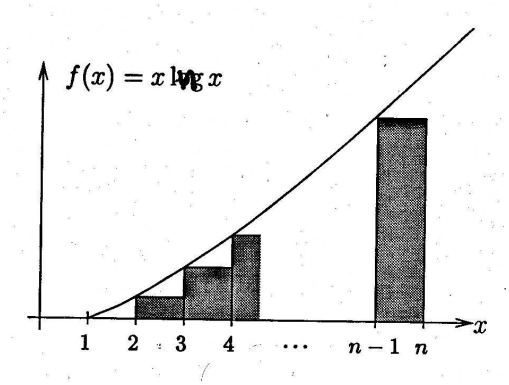
Induktionsvoraussetzung: Die Behauptung gelte für $2, 3, \ldots, n-1$.

Schluss auf n:

$$\begin{aligned}
r(n) &= n + \frac{2}{n}\sum_{k=2}^{n-1} r(k) \\
&\leq n + \frac{2}{n}\sum_{k=2}^{n-1} c \cdot k \ln k \quad \text{nach Induktionsvoraussetzung}
\end{aligned}$$

Um diesen Ausdruck weiter nach oben abzuschätzen, betrachten wir die Funktion $f(x) = x \ln x$. Dann ist $\sum_{k=2}^{n-1} k \ln k$ gerade der Flächeninhalt der schraffierten Fläche unter der Kurve $f(x)$, siehe nachfolgende Abbildung. Also gilt

$$\begin{aligned}
\sum_{k=2}^{n-1} k \ln k &\leq \int_2^n x \ln x \, \mathrm{dx} \\
&= \frac{x^2}{2} \ln x \Big|_2^n - \int_2^n \frac{x}{2} \quad \text{(partielle Integration)} \\
&= \frac{n^2}{2} \ln n - 2 \ln 2 - \left(\frac{n^2}{4} - 1 \right) \\
&\leq \frac{n^2}{2} \ln n - \frac{n^2}{4}
\end{aligned}$$



Hieraus folgt

$$\begin{aligned}
r(n) \leq n + \frac{2}{n} \sum_{k=2}^{n-1} c \cdot k \ln k &= n + \frac{2c}{n} \sum_{k=2}^{n-1} k \ln k \\
leq \quad & n + \frac{2c}{n} \left(\frac{n^2}{2} \ln n - \frac{n^2}{4} \right) \\
= \quad & n + c \cdot n \cdot \ln n - \frac{c}{2} n \\
= \quad & c \cdot n \cdot \ln n \quad \text{wegen} \quad c = 2.
\end{aligned}$$

□

Aus dem Lemma folgt:

Satz 6.3.2 *Für die mittlere Anzahl $\bar{C}(n)$ der Vergleiche zum Sortieren eines n-elementigen Arrays mit Quicksort gilt*

$$\bar{C}(n) = O(n \log n).$$

Beweis. Aus $\bar{C}(n) \leq r(n)$ und dem Lemma folgt

$$\bar{C}(n) \leq 2 \cdot n \ln n = 2 \cdot n \cdot \frac{\log n}{\log e} = \frac{2}{\log e} \cdot n \cdot \ln n$$

für all $n \geq 2$. Also ist $\bar{C}(n) = O(n \log n)$ mit der O-Konstanten $2/\log e \approx 2,89$. □

Entsprechend kann man für die mittlere Anzahl $\bar{A}(n)$ von Zuweisungen beweisen, dass

$$\bar{A}(n) = O(n \log n).$$

Quicksort arbeitet also im Mittel beweisbar sehr schnell, und dies wird auch in allen Laufzeituntersuchungen bestätigt.

Quicksort ist der Sortieralgorithmus, den man verwenden sollte.

6.4 Heapsort

Heapsort basiert im Gegensatz zu Mergesort und Quicksort nicht auf dem Prinzip der Aufteilung, sondern nutzt eine spezielle Datenstruktur (Heap), mit der wiederholt auf das größte Element eines Arrays zugegriffen wird.

Definition 6.4.1 *Ein* **Heap** *(priority queue) ist eine abstrakte Datenstruktur mit folgenden Kennzeichen:*

Wertebereich: Eine Menge von Werten des homogenen Komponententyps.

Operationen:

a) Einfügen einer Komponente

b) Zugriff auf die Komponente mit maximalem Wert

c) Entfernen einer Komponente

d) Änderung des Werts einer Komponente

Grobstruktur von Heapsort

Gegeben ein Array a mit n Komponenten.

1 Initialisiere den Heap mit den Komponenten von a.

2 FOR $i = n - 1$ DOWNTO 0 DO

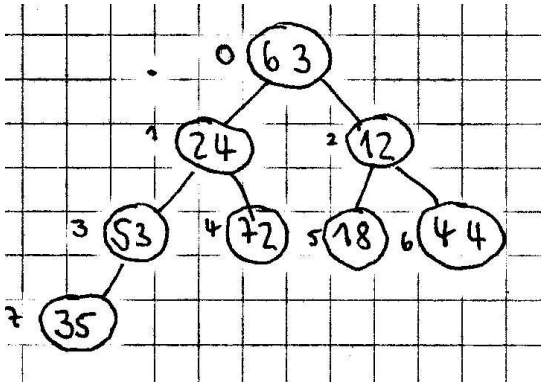
- 2.1 Greife auf das maximale Element des Heaps zu.
- 2.2 Weise diesen Wert der Arraykomponente $a[i]$ zu.
- 2.3 Entferne das größte Element aus dem Heap und aktualisiere ihn.

Korrektheit des Verfahrens: Klar.

Ziel: Implementation des Heaps, so dass die Operationen a)-d) schnell ausgeführt werden können.

Konkret:

- Initialisierung des Heaps in $O(n)$
- Zugriff auf das größte Element in $O(1)$
- Entfernen des größten Elements in $O(\log n)$



Idee zur Verwirklichung des Ziels

Man stelle sich das Array a als binären Baum vor.

Z.B. für $a = [63, 24, 12, 53, 72, 18, 44, 35]$

Mit dieser Interpretation gilt sofort:

- $a[0]$ = Wurzel des Baums
- $a[2i+1]$ = linker Sohn von $a[i]$
- $a[2i+2]$ = rechter Sohn von $a[i]$

Definition 6.4.2 *Wir sagen, dass das Array a mit n Komponenten die* **Heapeigenschaft** *hat, falls gilt:*

$$\left.\begin{array}{lcl} a[i] & \geq & a[2i+1] \\ a[i] & \geq & a[2i+2] \end{array}\right\} \forall i \quad mit \quad \begin{array}{lcl} 2i+1 & \leq & n-1, \\ 2i+2 & \leq & n-1. \end{array}$$

Bemerkung: Hat das Array a die Heapeigenschaft, so ist $a[0]$ das maximale Element des Arrays und entlang jeden Weges von einem Blatt zu der Wurzel sind die Schlüsselwerte aufsteigend sortiert. Demnach kann auf das größte Element von a in $O(1)$ zugegriffen werden.

Um die Heapeigenschaft eines Arrays a herzustellen, benötigen wir eine Unterroutine.

Heapify(a,L,U):

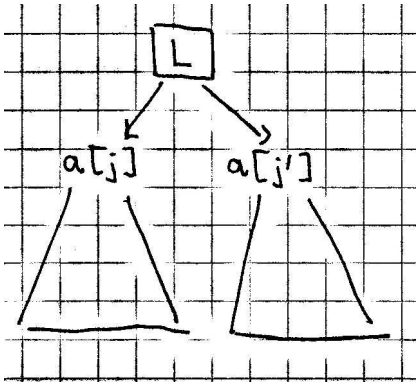
Input: a hat die Heapeigenschaft im Bereich $L+1, \ldots, U$.

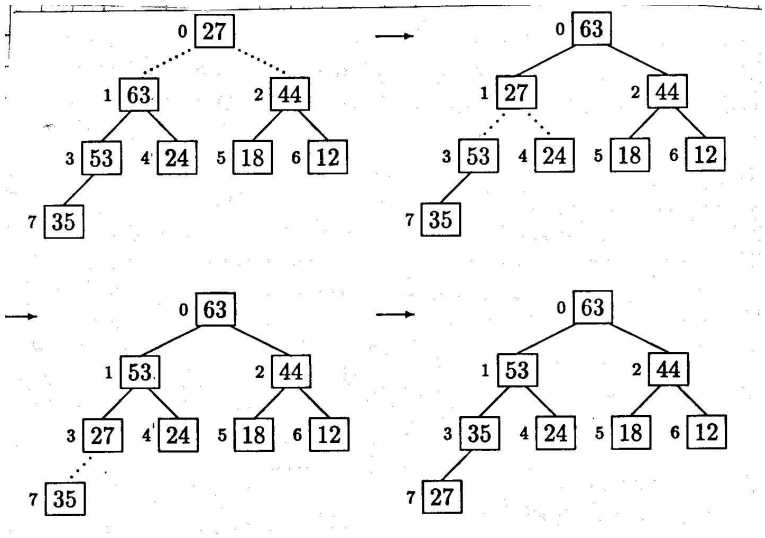
Output: a hat die Heapeigenschaft im Bereich $L, \ldots, U$.

1. Bestimme den Sohn j von L mit maximalem Wert.
2. Falls $a[j] > a[L]$, tausche $a[L]$ mit $a[j]$.
3. Rufe Heapify(a,j,U) auf.

Korrektheit von Heapify(a,L,U):

Nach Schritt 2 gilt: $a[L] \geq a[j]$ und $a[L] \geq a[j']$. Ferner gilt die Heapeigenschaft im Teilbaum mit Wurzel j'. Sie könnte jedoch im Teilbaum mit Wurzel j verletzt sein, was durch den rekursiven Aufruf in Schritt 3 jedoch repariert wird. □





Beispiel.

Im folgenden bezeichne die Überprüfung und ggf. die Herstellung der Heapeigenschaft für einen Knoten mit seinen Söhnen eine **Prüfaktion**.

Bemerkung: Der Aufruf von Heapify(a,L,U) benötigt $\lceil \log (U - L) \rceil$ Prüfaktionen.

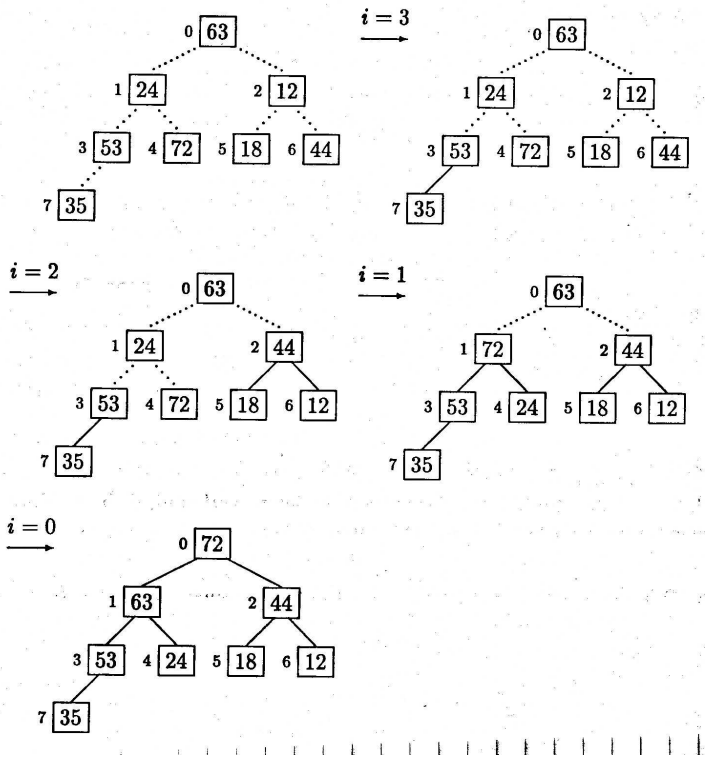
Die Routine Heapify lässt sich zum Herstellen der Heapeigenschaft eines Arrays a nutzen:

CreateHeap(a,n)

FOR $i = \lfloor \frac{n-1}{2} \rfloor$ DOWNTO 0 DO

Rufe auf Heapify(a,i,n-1).

Beispiel.



Lemma 6.4.3 *CreateHeap(a,n) erfordert $O(n)$ Prüfaktionen.*

Beweis. Sei $n \in \mathbb{N}$, so dass $2^k \leq n \leq 2^{k+1} - 1$. Dann gilt:

Schicht k	:	keine Prüfaktionen
Schicht $k-1$	:	eine Prüfaktion
	$\vdots$	
Schicht $k-i$	:	i Prüfaktionen
	$\vdots$	
Schicht 0	:	k Prüfaktionen

$$
\begin{aligned}
\Rightarrow \#\ \text{Prüfaktionen} &\leq 2^{k-1} \cdot 1 + \ldots + 2^{k-i} \cdot i + \ldots + 2^0 \cdot k \\
&= \sum_{i=1}^{k} 2^{k-i} \cdot i \\
&= \sum_{i=1}^{k} \frac{2^k}{2^i} \cdot i \\
&= 2^k \sum_{i=1}^{k} \frac{i}{2^i} \\
&= 2n, \quad \text{da} \quad 2^k \leq n \quad \text{und} \quad \sum_{i=1}^{\infty} \frac{i}{2^i} = 2
\end{aligned}
$$

□

Somit ergibt sich folgender Algorithmus und der zugehörige Satz:

Algorithmus **Heapsort**

Input: a,n

1 CreateHeap(a,n)

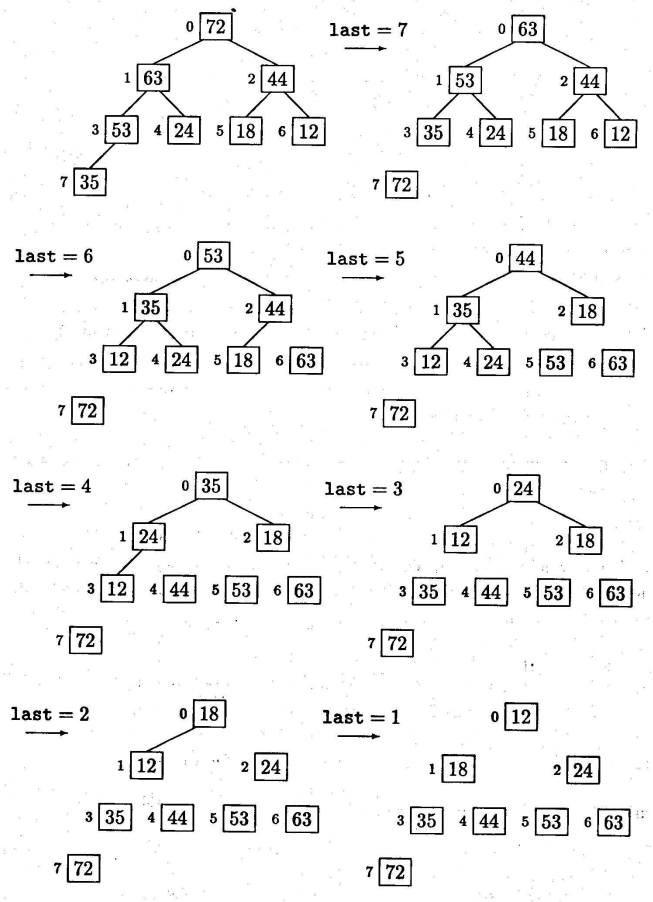
2 FOR $u = n - 1$ DOWNTO 0

2.1 temp= $a[0]$, $a[0] = a[u]$, $a[u]$ =temp

2.2 Heapify(a,0,u-1)

Satz 6.4.4 *Heapsort arbeitet korrekt und benötigt $O(n \log n)$ Prüfaktionen. ($\Rightarrow O(n \log n)$ Vergleiche und Zuweisungen)*

Beispiel.


0 72
1 63
2 44
3 53
4 24
5 18
6 12
7 35
last = 7
0 63
1 53
2 44
3 35
4 24
5 18
6 12
7 72
last = 6
0 53
1 35
2 44
3 12
4 24
5 18
6 63
7 72
last = 5
0 44
1 35
2 18
3 12
4 24
5 53
6 63
7 72
last = 4
0 35
1 24
2 18
3 12
4 44
5 53
6 63
7 72
last = 3
0 24
1 12
2 18
3 35
4 44
5 53
6 63
7 72
last = 2
0 18
1 12
2 24
3 35
4 44
5 53
6 63
7 72
last = 1
0 12
1 18
2 24
3 35
4 44
5 53
6 63
7 72

Kapitel 7

Untere Komplexitätsschranken für das Sortieren

Die besten bisher kennengelernten Sortierverfahren für Arrays haben einen Aufwand von $O(n \log n)$ im Worst-Case (Mergesort, Heapsort), bzw. im Average-Case (Quicksort).

Es stellt sich nun die Frage, ob es noch bessere Sortierverfahren geben kann. Dies kann tatsächlich der Fall sein, wenn man zusätzliche Informationen über die Schlüsselmenge hat, wie das Verfahren **Bucketsort** zeigt. Basieren die Algorithmen jedoch nur auf paarweisen Vergleichen von Schlüsseln, so gilt:

Satz 7.0.5 *[Untere Schranken für das Sortieren mit Vergleichen] Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen von Schlüsseln basiert, braucht zum Sortieren eines n-elementigen Arrays sowohl im Worst-Case, als auch im Mittel (bei Gleichverteilung) $\Omega(n \log n)$ Vergleiche.*

Dieser Satz zeigt, dass Mergesort, Heapsort und Quicksort bzgl. der Größenordnung **optimal** sind, und dass Laufzeitunterschiede höchstens der O-Konstanten zuzuschreiben sind.

Man beachte noch einmal den Unterschied zu den bisher gemachten $O(\ldots)$-Abschätzungen für ein Problem. Diese haben wir dadurch erhalten, dass ein konkreter Algorithmus, der das Problem löst, analysiert wurde. Die im Satz formulierte $\Omega(\ldots)$-Abschätzung bezieht sich jedoch auf **alle möglichen** Sortierverfahren (bekannte und unbekannte). Sie macht also eine Aussage über eine **Klasse von Algorithmen** statt über einen konkreten Algorithmus und ist damit von ganz anderer Natur.

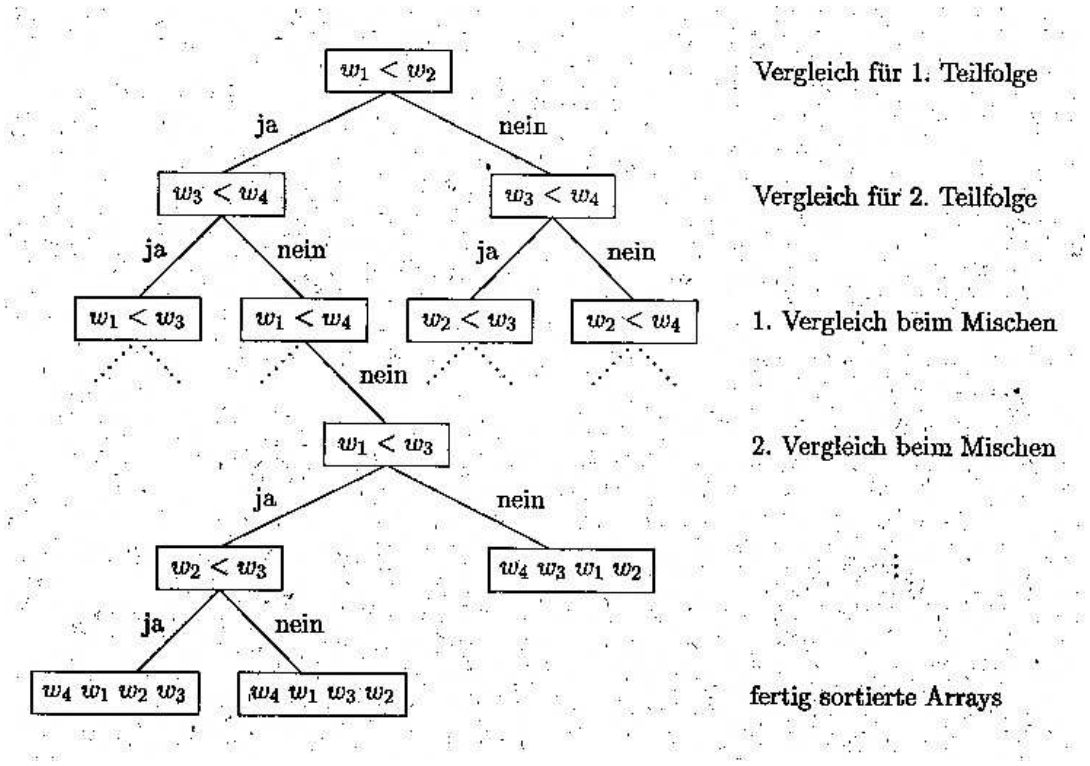
7.1 Das Entscheidungsbaum-Modell

Zum Beweis des Satzes werden wir die Tatsache nutzen, dass jeder deterministische Sortieralgorithmus, der nur auf paarweisen Vergleichen von Schlüsseln basiert, durch einen **Entscheidungsbaum** wie folgt beschrieben werden kann.

- Innere Knoten des Entscheidungsbaums sind Vergleiche im Algorithmus.
- Blätter des Baumes sind die sortierten Arrays, also $n!$ bei n zu sortierenden Elementen.

- Ein Weg von der Wurzel bis zu einem Blatt entspricht im Algorithmus der Folge der angestellten Vergleiche. Dabei wird vereinbart, dass beim Weitergehen nach **links** bzw. **rechts** der letzte Vergleich richtig (**true**) bzw. falsch (**false**) ist.

Als Beispiel betrachten wir einen Ausschnitt des Entscheidungsbaumes für das Sortieren von w_1, w_2, w_3, w_4 mit Mergesort. Es erfolgt also zunächst die Aufteilung in die Teilfolgen w_1, w_2 und w_3, w_4. Diese werden dann sortiert und gemischt. Es entsteht der in der folgenden Abbildung gezeigte Baum.



Satz 7.1.1 *Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen basiert, erzeugt einen solchen Entscheidungsbaum T.*

Beweis. Da der Algorithmus deterministisch ist, hat er für jede Eingabefolge denselben **ersten** Vergleich zwischen Arraykomponenten. Dieser bildet die **Wurzel** des Entscheidungsbaumes. In Abhängigkeit vom Ausgang des Vergleiches ("<" oder ">") ist der nächste Vergleich wiederum eindeutig bestimmt. Die Fortsetzung dieser Argumentation liefert für jede Eingabefolge eine endliche Folge von Vergleichen, die einem Weg von der Wurzel bis zu einem Blatt (sortierte Ausgabe) entspricht. □

Wir überlegen nun, wie wir den Worst-Case- bzw. Average-Case-Aufwand $C(n)$ bzw. $\bar{C}(n)$ des Algorithmus im Baum T ablesen können. Sei dazu $h(v)$ die Höhe des Knoten v im Baum T, $h(T)$ die Höhe von T, und $H(T) := \sum_{h \text{ Blatt}} h(v)$ die sogenannte **Blätterhöhensumme** von T.

Lemma 7.1.2 *Sei T der Entscheidungsbaum für den Algorithmus A und $C(n)$ bzw. $\bar{C}(n)$ die Worst-Case- bzw. Average-Case- (bei Gleichverteilung) Anzahl von Vergleichen bei n zu sortierenden Komponenten. Dann gilt:*

a) $C(n) = \max\limits_{v \text{ Blatt}} h(v) = h(T)$,

b) $\bar{C}(n) = \frac{1}{n!} \sum\limits_{v \text{ Blatt von } T} h(v) = \frac{1}{n!} H(T)$.

Beweis. Die Anzahl der Vergleiche, um zu einer sortierten Ausgabe v zu kommen, ist gerade $h(v)$. Also folgt a) direkt. Da wir Gleichverteilung der $n!$ verschiedenen Eingabereihenfolgen (und damit Ausgabereihenfolgen) annehmen, folgt b). □

7.2 Analyse des Entscheidungsbaumes

Die Abschätzung von $C(n)$ bzw. $\bar{C}(n)$ nach unten reduziert sich also auf die Abschätzung der Höhe bzw. der Blätterhöhensumme eines binären Baumes mit $n!$ Blättern nach unten. Dazu zeigen wir folgendes Lemma.

Lemma 7.2.1 *Sei T ein binärer Baum mit b Blättern. Dann gilt:*

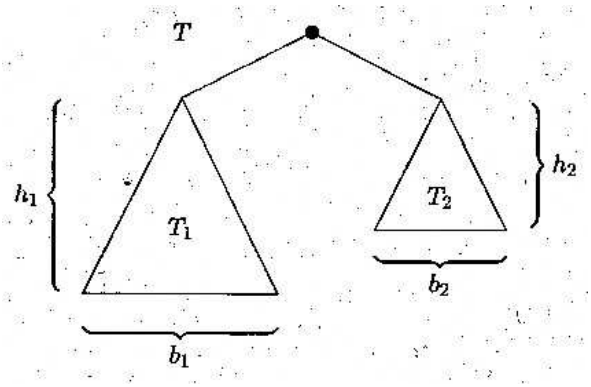
a) $h(T) \geq \log b$,

b) $H(T) \geq b \cdot \log b$.

Beweis. Der Beweis wird in beiden Fällen durch vollständige Induktion nach der Höhe $h(T)$ von T geführt.

Ist $h(T) = 0$, so besteht T nur aus der Wurzel, die zugleich ein Blatt ist. Also ist $b = 1$ und $\log b = 0 = h(T)$. Entsprechend ist $H(T) = \sum_{h \text{ Blatt}} h(v) = 0$ und $b \cdot \log b = 0$.

Es gelten nun a), b) für $h(T) = 0, 1, \ldots, h-1$ (Induktionsvoraussetzung). Zum Schluss auf h betrachte man die beiden Teilbäume T_1 und T_2 von T, wobei einer leer sein kann, vgl. folgende Abbildung.



Jeder der Teilbäume hat eine geringere Höhe als h, also trifft auf T_1 und T_2 die Induktionsvoraussetzung zu. Sei b_i die Anzahl der Blätter und h_i die Höhe von T_i, $(i = 1, 2)$, uns ei o.B.d.A. $b_1 \geq b_2$. Dann gilt:

$$\begin{aligned} h(T) = 1 + \max\{h_1, h_2\} &\geq 1 + h_1 \\ &\geq 1 + \log b_1 \quad \text{Induktionsvoraussetzung} \\ &= \log 2 + \log b_1 \\ &= \log (2b_1) \\ &= \log (b_1 + b_2), \quad (\text{da } b_1 \geq b_2) \\ &= \log b. \end{aligned}$$

Dies beweist a). Im Fall b) gilt

$$H(T) = b + H(T_1) + H(T_2),$$

da in T jedes Blatt gegenüber T_1 und T_2 eine um 1 größere Höhe hat. Auf T_1 und T_2 ist die Induktionsvoraussetzung anwendbar und es folgt

$$\begin{aligned} H(T) &\geq b + b_1 \log b_1 + b_2 \log b_2 \quad \text{Induktionsvoraussetzung} \\ &= b + b_1 \log b_1 + (b - b_1) \log (b - b_1). \end{aligned}$$

Da wir nicht genau wissen, wie groß b_1 ist, fassen wir die rechte Seite als von Funktion von $x = b_1$ auf und suchen ihr Minimum.

Also ist

$$H(T) \geq b + \min_{x \in [1,b]} [x \log x + (b - x) \log (b - x)].$$

Eine Kurvendiskussion zeigt, dass $f(x) := x \log x + (b - x) \log (b - x)$ das Minimum auf $[1, b]$ bei $x = b/2$ annimmt. Damit ist

$$\begin{aligned} H(T) &\geq b + \frac{b}{2} \log \frac{b}{2} + \frac{b}{2} \log \frac{b}{2} \\ &= b + b \log \frac{b}{2} \\ &= b + b(\log b - \log 2) \\ &= b + b(\log b - 1) \\ &= b \log b. \end{aligned}$$

□

Für die endgültige Abschätzung benötigen wir noch eine Abschätzung von $n!$ nach unten. Es ist

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1 \geq n \cdot (n-1) \cdot \ldots \cdot \lceil n/2 \rceil \geq \lceil n/2 \rceil^{\lfloor n/2 \rfloor + 1} \geq (n/2)^{n/2}.$$

Nach diesen Vorbereitungen kommen wir jetzt zum

Beweis von Satz 7.0.5. Sei T der Entscheidungsbaum zum gegebenen Sortieralgorithmus A. Bei einem Inputarray der Länge n hat T $n!$ Blätter. Es folgt

$$\begin{aligned} C(n) &= h(T) \\ &\geq \log n! \\ &\geq \log\left[(n/2)^{n/2}\right] \\ &= \frac{n}{2}\log\frac{n}{2} \\ &= \frac{n}{2}\log n - \frac{n}{2} \\ &= \frac{n}{3}\log n + \frac{n}{6}\log n - \frac{n}{2} \\ &\geq \frac{n}{3}\log n \quad \text{für} \quad \frac{n}{6}\log n \geq \frac{n}{2}, \text{also} \quad n \geq 8 \\ &= \Omega(n\log n). \end{aligned}$$

Entsprechend ist

$$\begin{aligned} \bar{C}(n) &= \frac{1}{n!}H(T) \\ &\geq \frac{1}{n!}(n!\log n!) \\ &= \log n! \\ &= \Omega(n\log n) \quad \text{(wie oben)}. \end{aligned}$$

□

Dieses Ergebnis, bzw. genauer die Ungleichungen

$$C(n) \geq \log n! \quad \text{und} \quad \bar{C}(n) \geq \log n!$$

werden auch als **informations-theoretische Schranken** für das Sortieren bezeichnet. Sie lassen sich anschaulich folgendermaßen interpretieren. Jeder Sortieralgorithmus muss zwischen $n!$ Möglichkeiten (den sortierten Reihenfolgen) unterscheiden und muss daher $\log n!$ Bits an Information sammeln. Ein Vergleich ergibt höchstens ein Bit an Information.