Algorithmen und Datenstrukturen

Jörg Schäfer
Jens Liebehenschel
Frankfurt University of Applied Sciences

Fachbereich 2 Informatik und Ingenieurwissenschaften
Computer Science and Engineering
http://www.informatik.fb2.frankfurt-university.de/~jschaefer/

Skript
Sommersemester 2024

Zusammenfassung

Das Modul 9 „*Algorithmen und Datenstrukturen*“ vermittelt die mit den Begriffen Algorithmus und abstrakte Datenstruktur verbundenen Kenntnisse, die zentral für die gesamte Informatik sind. Somit werden mit dem Kurs sowohl Schlüsselqualifikationen für die berufliche Tätigkeit als Informatiker, als auch Voraussetzungen für das Verständnis nahezu aller Folgekurse im Verlauf des Studiums vermittelt. Die Studierenden sollen die in dem Modul vermittelten Begriffe Algorithmen, Datenstrukturen, Komplexität etc. soweit verstanden haben, dass für einfache bis mittel-schwere Problemstellungen geeignete neue Datenstrukturen (aufbauend auf den in dem Kurs behandelten Standardstrukturen) gestaltet, Algorithmen zur Bearbeitung entwickelt und nach den gelernten Methoden dargestellt werden und Lösungsmöglichkeiten hinsichtlich Korrektheit, Komplexität und Eleganz beurteilt werden können.

Dieses Skript dient lediglich der Ergänzung zur Vorlesung und stellt keinen Ersatz für ein Lehrbuch dar.

Sprache

Dieses Skript ist nach den Regeln der neuen deutschen Rechtschreibung verfasst, die allerdings *nicht* durchgängig verwendet werden. Unter anderem aus Gründen der Lesbarkeit verwende ich in diesem Skript das *generische Maskulinum*. Ich habe mich außerdem bemüht, die für die Informatik heutzutage so typischen, unschönen Anglizismen zu vermeiden – dies ist mir jedoch nicht vollständig gelungen.

Danksagung

Ich möchte mich ganz herzlich bei den Tutoren des Sommersemesters 2012, Herrn David Klein, Herrn Jens Mäusezahl und Herrn Daniel Roß für Anregungen bedanken, die zur Verbesserung des Skripts geführt haben. Ganz besonders gilt mein Dank Herrn Nicolas Lupp, der dieses Skript im Jahre 2013 als Lehrbeauftragter verwendet hat. Herr Lupp hat mir viele hilfreiche Anmerkungen zum Skript gegeben sowie zahlreiche Verbesserungsvorschläge gemacht, die in die vorliegende Version eingegangen sind. Außerdem hat er zahlreiche Fehler gefunden, die ich beseitigt habe.

Ebenso gilt mein ganz besonderer Dank dem Kollegen und Koautor Jens Liebehenschel, der das (neue) Kapitel 14 beigefügt und ausserdem umfangreiche Verbesserungsvorschläge und Fehlerkorrekturen vorgeschlagen hat. Alle noch verbleibende Fehler gehen allein zu meinen Lasten.

Lizenz und Copyright

Feedback

Da ich stets daran interessiert bin, dieses Skript zu verbessern, bin ich über konstruktives Feedback jederzeit erfreut!

Jörg Schäfer, Frankfurt, 24.03.2016.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

Listings

Kapitel 1

Einführung und Motivation

> Wer fragt, ist ein Narr für eine Minute. Wer nicht fragt, ist ein Narr sein Leben lang.
>
> Einen Fehler begangen haben und ihn nicht korrigieren: Erst das ist ein Fehler.
>
> Konfuzius

> Ich höre und vergesse, ich sehe und behalte, ich handle und verstehe.
>
> Folklore (angeblich Konfuzius)

1.1 Organisation

1.1.1 Vorlesung, E-Learning, Übungen, Klausur

Vorlesung und Übungen

Das Modul ist wie folgt organisiert:

1. Es gibt zwei jeweils zweistündige Vorlesungen pro Woche, die Sie dem Stundenplan entnehmen.
2. Es gibt eine zweistündige Übung, die Einteilung wird durch die Dozenten bzw. Tutoren vorgenommen.
3. Sie erhalten die Credits durch Bestehen einer Klausur (s.u.).

Warnung: Der Stoff ist zu schwer, zu kompliziert und zu umfangreich, um *ausschließlich* durch *Zuhören* verstanden zu werden! Es ist *auf jeden Fall* notwendig, dass Sie Zeit in die Vor- und Nachbereitung der Lehrveranstaltung investieren.

Daher empfehlen wir *ausdrücklich*:

1. Besuchen Sie regelmäßig die Vorlesung *und* arbeiten Sie diese nach, und zwar das ganze Semester hindurch!
2. Lesen Sie neben Ihren Aufzeichnungen und diesem Skript auch Bücher (siehe Literaturliste)!
3. Bilden Sie Lerngruppen!
4. Die Übungen sind zum *Üben* da. Das bedeutet, dass Sie *immer* versuchen sollen, die Aufgaben zuhause *selbstständig* (oder in einer Lerngruppe gemeinsam) zu lösen! Arbeiten Sie in der Übung aktiv mit!
5. Bei Verständnisproblemen frühzeitig fragen!
6. Nutzen Sie das Know-how der Übungsleiter!

E-Learning

Diese Vorlesung ist keine *selbstständige* E-Learning-Veranstaltung, sondern verfolgt vielmehr ein Blended Learning Konzept. Daher gibt es Unterlagen, Übungsaufgaben und weiterführende Informationen im E-Learning System der Frankfurt University of Applied Sciences. Die Zugangsdaten werden in der Vorlesung (und in den Übungen) bekannt gegeben.

Klausur

Am Ende des Semesters findet die Klausur statt (90 min). Voraussetzungen für die Teilnahme an der Klausur bestehen keine, aber die Erfahrung hat gezeigt, dass nur die regelmäßige, aktive Teilnahme an den Übungen für Erfolg sorgt.

1.1.2 Lernziele

In dieser Vorlesung werden wir die folgenden Themen behandeln, und Sie (hoffentlich!) über die folgenden Begriffe etwas lernen:

- Darstellung von Algorithmen, Bausteine für Algorithmen (besonders Iteration, Rekursion)
- Analyse: Korrektheit von Algorithmen, Induktionsbeweise für iterative und rekursive Verfahren, Laufzeitbegriff, $\mathcal{O}$-Notation, Laufzeitanalyse
- Algorithmen zur Implementierung von Datentypen
- Balancierte Suchbäume (Grundlagen), einfache Hashverfahren
- Sortierverfahren (u.a. Quicksort, Heapsort, Mergesort)
- Grundbegriffe der Graphentheorie

- Ausgewählte grundlegende Algorithmen für Graphen: Breitensuche, Tiefensuche, Kürzester Weg (Dijkstra)
- Randomisierte Algorithmen
- Verteilte Algorithmen und Nebenläufigkeit: Mehrere Rechner arbeiten zusammen, kommunizieren, zeitliche Abfolge nicht vorhersagbar (nur Einführung)
- Entwurf von Algorithmen und Graphen, sowie Kenntnisse der wichtigsten Algorithmenparadigmen (Divide-and-Conquer, Dynamische Programmierung, Greedy, Backtracking, usw.)

1.1.3 FAQ

1. Ist der Stoff schwer?
   Antwort: Ja!
2. Ist der Stoff zu schwer?
   Antwort: Nein! Es gibt für alle Übungen überdies ein *Tutorenprogramm*, in dem erfahrene Tutoren Sie zusätzlich betreuen. Nutzen Sie daher diese Chance durch aktive Teilnahme!
3. Lohnt sich der Stoff?
   Antwort: Ja, auf alle Fälle! Algorithmen sind ein zentraler Bestandteil vieler innovativer Konzepte und Firmen in der IT. Wenn Sie Algorithmen beherrschen, dann werden Sie die Möglichkeit haben, an neuen Ideen mitzuwirken und interessante Jobs zu bekommen.
   Aus einem Standardtest für Telefoninterviews[1]:

   > Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

   Am Ende der Vorlesung können Sie so etwas[2]!
4. Was wird denn in dieser Veranstaltung vermittelt?
   Antwort: Wir vermitteln verschiedene "Bausteine". Sie werden verschiedene allgemeine und auch speziellere Algorithmen kennen lernen, aber auch Entwurfsprinzipien für Algorithmen. Außerdem Datenstrukturen mit verschiedenen Eigenschaften, die die Algorithmen ausnutzen können, um Probleme effizient zu lösen. Neben der Analyse von Algorithmen und Datenstrukturen geben wir Ihnen am Ende ein Beispiel, auf welche Weise Algorithmen und Datenstrukturen kombiniert werden können. Wie die "Bausteine" zusammen gesetzt werden, können wir

[1] Quelle: `https://blog.codinghorror.com/why-cant-programmers-program/`
[2] ca. 98% der interviewten Programmierer *mit Erfahrung* können so etwas nicht!

Ihnen nur ansatzweise vermitteln. Diese Erfahrung müssen Sie selbst machen, zum Beispiel indem Sie viel mit den Lehrinhalten "herumspielen".

5. Warum brauche ich als Informatiker(in) denn diese ganze Theorie?
   Antwort: Hier könnten sehr viele Antworten stehen, wir geben einige wenige:

   (a) In vielen Projekten – nicht nur aus dem technischen Bereich – muss Software optimiert werden. Dies betrifft meistens Laufzeit oder Speicherplatzbedarf. Dann benötigt man ein grundlegendes Verständnis von Algorithmen und Datenstrukturen samt der Analyse von Eigenschaften.

   (b) Viele Problemstellungen lassen sich viel effizienter lösen, wenn man sich zunächst Gedanken über die Lösung macht. Eine geeignete Kombination von Algorithmen und Datenstrukturen kann Probleme oft um Größenordnungen besser lösen als naive Umsetzungen. Dazu geben wir Ihnen am Anfang und am Ende des Skripts Beispiele.

   (c) Theoretische Zugänge zu Problemen helfen, einfache Lösungen zu finden. Oft tragen "theoretische Gedanken" zu effizienten Lösungen bei.

   (d) Viele praktische – und nicht nur theoretische – Probleme benötigen viel Mathematik und effiziente Algorithmen und Datenstrukturen. Beispielsweise 3D-Grafik und Animationen, die nicht nur für Computerspiele wichtig sind, sondern auch für viele andere professionelle Anwendungen. Ein anderes Beispiel sind die verschiedenen "Problemlöser", die auch in Computerspielen Anwendung finden.

   (e) Oft sollen in der Praxis Systeme gebaut werden, die immer verfügbar sind oder allen Angriffen widerstehen können. Das ist nicht möglich. Es ist wichtig, sich dessen bewusst zu sein, um mit den Auftraggebern diskutieren zu können. Sie werden unlösbare und "schwierige" Probleme kennen lernen.

6. Kann man denn heutzutage nicht ohne Kenntnis von Algorithmen und Datenstrukturen auskommen?
   Antwort: Wenn Sie lebenslänglich GUIs programmieren oder Systeme warten wollen, vielleicht ... Diese Jobs haben jedoch keine rosigen Zukunftsaussichten. Software-Entwickler hingegen, die Datenstrukturen und Algorithmen gut verstehen, können sich die Jobs aussuchen.

7. Muss ich programmieren?
   Antwort: Falsche Frage, Sie *dürfen* programmieren! Durch moderne Hardware und Technologien wie Hoch- und Skriptsprachen sind Sie

in der einmaligen Situation, dass Sie *alle* Algorithmen und Datenstrukturen ausprobieren und dadurch viel besser verstehen können als Generationen von Informatikern vor Ihnen. Diese Chance sollten Sie sich nicht entgehen lassen! Die Kenntnis von ausführbarem Pseudocode (Python) ist zudem klausurrelevant. Sie finden eine Liste von verfügbarem Beispielcode im Anhang und in Moodle!

8. Welche Vorkenntnisse sind notwendig?
   Antwort: Die nachstehenden Module:

   (a) Einführung in die Programmierung mit C, Teil I

   (b) Einführung in die Informatik

   (c) Algebra und Analysis

   In den parallel stattfindenden Modulen Einführung in die Programmierung mit C, Teil II, und Diskrete Mathematik werden die behandelten Themen praktisch umgesetzt bzw. deren mathematische Grundlagen behandelt. Im parallel stattfindenden Modul Theoretische Informatik werden Teile der hier behandelten Aspekte vom Standpunkt der Automaten und formalen Sprachen behandelt.

9. Welche Programmierkenntnisse werden erwartet?
   Antwort: Anfängerkenntnisse in C

1.1.4 Literatur

Die Vorlesung folgt ausgewähltem Material aus [CLR04] und [OW93] ergänzt durch ausgewähltes, aktuelles Material. Die Bücher sollten in der Bibliothek verfügbar sein; die wichtigsten Werke wurden zudem als *Semesterapparat* zusammengestellt, in dem die genannten Bücher für Sie *vorrätig* gehalten werden, so dass Sie jederzeit in der Bibliothek lernen können. Bitte von dieser Möglichkeit Gebrauch machen!

Als Klassiker für Algorithmen muss [Knu97a] [Knu97b] [Knu98] erwähnt werden. Zum Lernen kann er *nicht* empfohlen werden, da zu umfangreich und mathematisch, allerdings sollten Sie im Laufe Ihres Studiums – allein schon aus kulturellen Gründen – mal reingeschaut haben!

Manche Zeichnungen[3] in diesem Skript stammen von `http://xkcd.com/`.

1.1.5 Skript

Dieses Skript stellt keinen Ersatz für ein Lehrbuch dar, wird jedoch fortlaufend weiterentwickelt und kann Ihnen als Gedächtnisstütze dienen! Das Skript

[3] lizenziert unter einer "Creative Commons Attribution-NonCommercial 2.5 License"

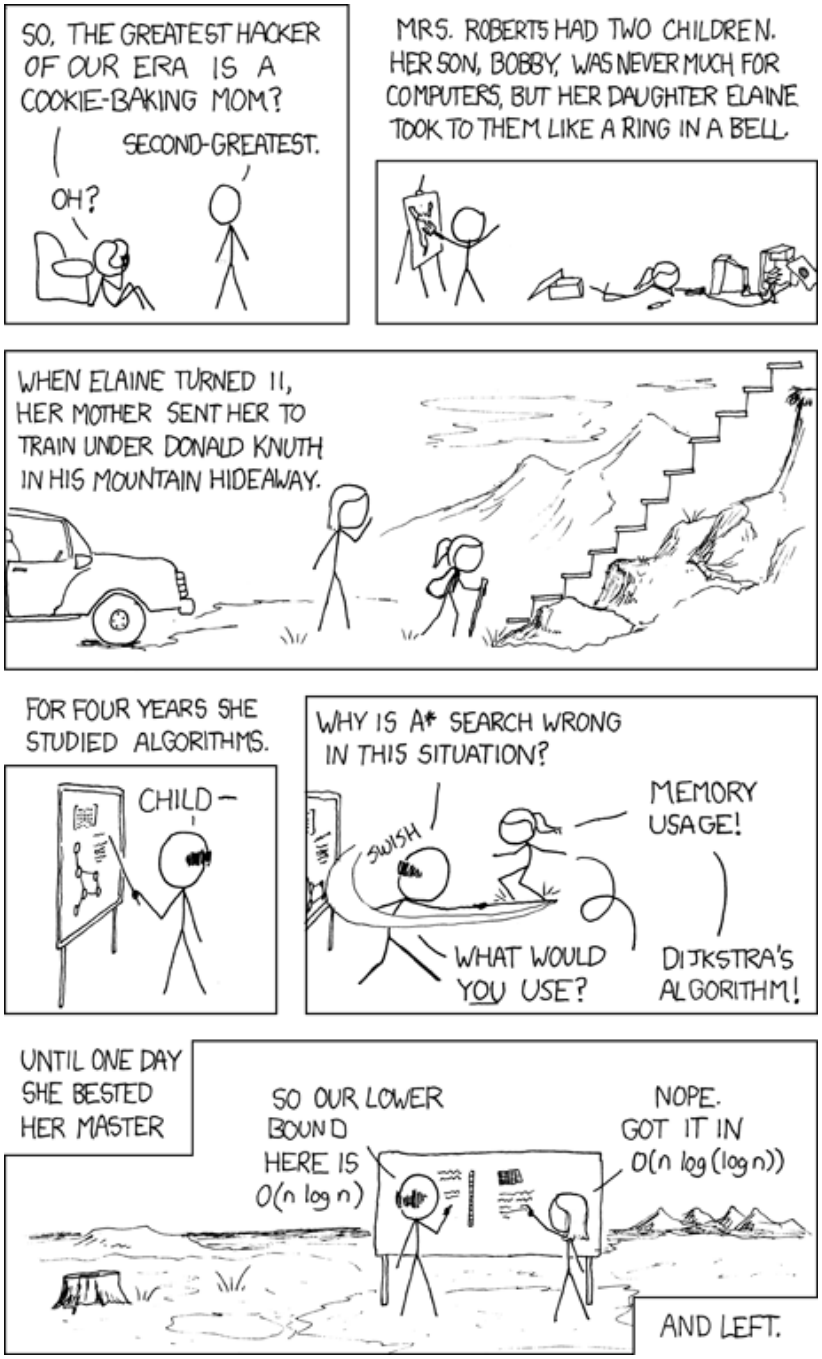


Abbildung 1.1: The Art of Algorithms, Quelle: http://xkcd.com/

beruht auf der genannten Literatur, für Fehler bin ich allein verantwortlich – Fehler und allgemeine Anmerkungen gerne an mich, damit ich das Skript verbessern kann!

Am Ende des Skriptes (auf S. 266) finden Sie ein Verzeichnis mit den wichtigsten hier verwendeten Notationen.

Bemerkung: Machen Sie sich zusätzlich zum Skript unbedingt *eigene* Notizen! Das ist äußerst sinnvoll und hilft Ihnen beim Verständnis des Lernstoffes.

1.2 Beziehung zu anderen Modulen

Algorithmen und Datenstrukturen bauen auf der „Einführung in die Informatik“ auf und sind einerseits Grundlage für das Verständnis Theorie Formaler Sprachen und Automaten und profitieren andererseits von Techniken, die in der Theorie Formaler Sprachen und Automaten – alles Themen des Moduls „Theoretische Informatik“ – entwickelt wurden.

1.3 Praxisbeispiel I: Textanalyse

Algorithmen, die mit Textanalyse zu tun haben, sind spätestens seit dem Siegeszug des WWW in aller Munde – ein Großteil des Erfolgs von Google besteht in der intelligenten Anwendung von Algorithmen zur Text bzw. Datenanalyse. Weitere Anwendungsgebiete finden sich in der Biologie, z. B. bei der Genomanalyse (unsere Gene sind ja auch nur ein – gar nicht so langer – Text).

Wir betrachten als Beispiel die folgende Aufgabe:

Einlesen eines Textes und bestimmen der häufigsten Wörter!

Das nachfolgende Listing 1.1 stellt eine naive Methode dar, die Wörter im Text zu zählen und verwendet als Datenstrukturen nur Felder (Arrays). (Falls Sie glauben sollten, dass das nur ein fiktives, pädagogisches Beispiel sein soll – solchen Code findet man *ständig* in der Praxis – geschrieben von unerfahrenen Programmierern, oder solchen, die ihr Studium wieder vergessen haben.) Die Laufzeit für die Analyse z. B. der Lutherbibel (ca. 764512 Wörter) beträgt ca. 4,5 Minuten auf einem 2.53 GHz Intel Core Duo 2 Prozessor.

Dasselbe Problem lässt sich eleganter lösen: durch Verwenden einer geeigneteren Datenstruktur (in diesem Fall einer Hashtabelle) und an diese Datenstruktur angepassten Algorithmen. Ein Listing für eine Skriptsprache Python ist in 1.2 dargestellt. Die Laufzeit für die Lutherbibel beträgt weniger als 1 Sekunde! Und das, obwohl Python eine „langsame“ Skriptsprache ist im Vergleich zu C! (Derselbe Algorithmus als C-Programm läuft in 0.35 Sekunden versus 262 Sekunden, ist also 748 Mal schneller[4]!)

[4]Source-Code gibt es im E-Learning und wird besprochen, sobald die Datenstruktur Hashtabelle behandelt wird

Listing 1.1: Naive Methode, Wörter zu zählen

```
1  #include <stdio.h>
2  #include <string.h>
3
4  # define MAX_WORDS 1000000
5
6  char *WORDS[MAX_WORDS];
7  int COUNT[MAX_WORDS];
8  int LAST = 0;
9
10 int find(char *word)
11 {
12     for (int i = 0; i < MAX_WORDS; i++) { // linear search
13         if(WORDS[i] != NULL && (strcmp(WORDS[i], word) == 0)){
14             return i;
15         }
16     }
17     return -1;
18 }
19
20 void add_naive(char *word)
21 {
22     int i = 0;
23     if((i = find(word)) != -1){ // word exists already
24         COUNT[i]++;
25     }
26     else { // new word
27         COUNT[LAST] = 1;
28         WORDS[LAST++] = word;
29     }
30 }
31
32 void print_count_naive()
33 {
34     int c, max_index, max = 0;
35     for (int i = 0; i < MAX_WORDS; i++) {
36         c = COUNT[i];
37         if(max < c){
38             max_index = i;
39             max = c;
40         }
41     }
42     printf("Word \"%s\" has max count %i",
43            WORDS[max_index], COUNT[max_index]);
44
45 }
```

Listing 1.2: Intelligentere Methode, Wörter zu zählen

```
1  import sys
2
3  def wc(filename):
4      wc = {}
5      f = open(filename, encoding='latin1')
6      for line in f.readlines():
7          words = line.split("␣")
8          for word in words:
9              if word in wc:
10                 wc[word] += 1
11             else:
12                 wc[word] = 1
13     f.close()
14     return wc
15
16 def get_count(counts):
17     max_count = 0
18     max_word = ""
19     for key in list(counts.keys()):
20         if counts[key] > max_count:
21             max_count = counts[key]
22             max_word = key
23     return max_word, max_count
```

Der Unterschied zwischen beiden Datenstrukturen bewirkt *lineares* Wachstumsverhalten der Laufzeit des Algorithmus im Gegensatz zu *quadratischem* und dieser Unterschied ist gewaltig, wie Abbildung 1.2 verdeutlicht.

Fazit: Die „richtige“ Wahl eines Algorithmus und der unterliegenden Datenstrukturen ist *enorm* wichtig!

1.4 Praxisbeispiel II: Suchen

Ein wohlbekanntes Problem ist das Suchproblem, z. B. in einem Telefonbuch (s. Abbildung 1.3).

Wenn man einen Namen k suchen möchte, so kann man wie folgt vorgehen:

1. Telefonbuch in der Mitte aufschlagen.
2. Falls Name $= k$, dann gefunden.
3. Kommt k vor oder nach dem Namen der oben steht?
4. Je nachdem, wie die Antwort lautet, mit der unteren oder oberen Hälfte des Telefonbuchs wie unter 1. weiterverfahren

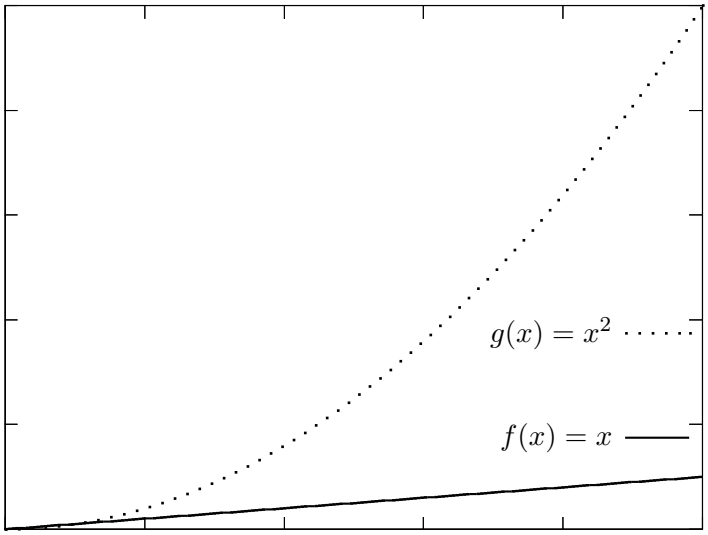


Abbildung 1.2: Quadratisches vs. lineares Wachstum

Abbildung 1.3: Telefonbuch, Quelle: `http://commons.wikimedia.org/wiki/File:Telefonbog_ubt-1.JPG`

Die formale Lösung für das Suchproblem gibt der Algorithmus BINARY-SEARCH 1.1, der auf einer geordneten Liste A operiert[5].

Der Algorithmus benutzt ein wichtiges Prinzip, mit dem Algorithmen konstruiert werden können: *Rekursion*!

Ein beispielhafter Ablauf für eine erfolgreiche Suche ist in Abbildung 1.4

[5]Der Algorithmus benutzt in Zeile 3 zur Berechnung von $(l + r)/2$ implizit die Gaußklammer oder Abrundungsfunktion, englisch "floor", d.h. es wird zur nächst kleineren ganzen Zahl abgerundet!

Algorithmus 1.1 BINARY-SEARCH(A, l, r, k)

```
1: if r < l then                          // Value not found!
2:     return false
3: mid ← (l + r)/2                        // using floor!
4: if k = A[mid] then                     // Value found!
5:     return true
6: else if k < A[mid] then                // Search left
7:     return BINARY-SEARCH(A, l, mid-1, k)
8: else                                   // Search right
9:     return BINARY-SEARCH(A, mid+1, r, k)
```

verdeutlicht[6].

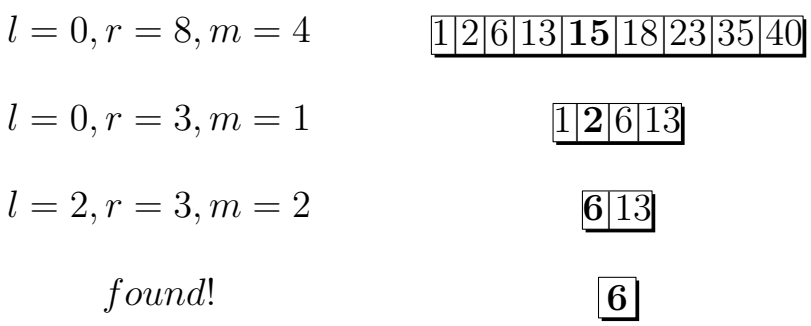


Abbildung 1.4: Binäre Suche für $k = 6$

Wichtige Eigenschaften eines Algorithmus

1. Korrektheit
2. Laufzeit

Lemma 1.4.1. *BINARY-SEARCH ist korrekt.*

Beweis. Wir müssen zeigen, dass BINARY-SEARCH abbricht und dass beim Abbruch der Wert **true** angenommen wird, falls das Element k in A existiert und sonst **false**.

Abbruch:
Abbruch ergibt sich daraus, dass bei jedem rekursiven Aufruf von BINARY-SEARCH die effektive Länge $d_i := r_i - l_i$ der Liste halbiert wird und aus der Tatsache, dass bei $d_i = 1$ die Suche beim nächsten Aufruf abgebrochen wird.

[6]Das Feld starte beim Index 0 und endet beim Index 8.

Korrektheit:

1. Sei $k \in A$: Es gilt die folgende Schleifeninvariante[7]: Bei jedem rekursiven Aufruf von BINARY-SEARCH liegt k im betrachteten Intervall $(l, \ldots, r)$, da die Liste A sortiert ist. Damit wird k in Zeile 4 mit Sicherheit gefunden, da die effektive Länge $d_i := r_i - l_i$ irgendwann 1 ist und somit $mid = 1$ das Element herausgreift.
2. Sei $k \notin A$: Zeile 4 ist immer falsch, da BINARY-SEARCH abbricht, muss also Zeile 2 ausgeführt werden.

□

Lemma 1.4.2. *BINARY-SEARCH benötigt maximal* $\lg n$ *Schritte, wobei* n *die Anzahl der Elemente in der geordneten Liste angibt.*

Beweis. In jedem Schritt i gilt für die effektive Länge $d_i := r_i - l_i$ der Liste: $d_i \leq n/2^i$, was man induktiv beweist. Bei Abbruch gilt $d_{i_a} \geq 1$ und somit $1 \leq n/2^{i_a}$ oder äquivalent $i_a \leq \lg n$. □

Neben der Worst Case Analyse (maximale Schrittzahl) sind auch die folgenden Maße interessant:

- Average Case: Durchschnittliche Zahl
- Best Case: Bestmögliche Anzahl

Best Case der Binärsuche ist offensichtlich dann gegeben, wenn man das Element in der „Mitte“ sucht: $Best(n) = 1$. Sei $k := \lg n$, dann gilt, dass im Durchschnitt $Avg(n) = k + k/n - 1$ Versuche benötigt werden[8].

Beweis. (Beweisskizze) Zur Vereinfachung nehmen wir an, dass $n = 2^k - 1$. Dies ist zwar nicht der Worst Case, aber einfach zu rechnen. Der Fehler ist für große n vernachlässigbar. Im ersten Durchgang kann 1 Element gefunden werden. Im zweiten Durchgang können 2, dann 4 usw. also im i-ten Durchgang 2^{i-1} Elemente gefunden werden. Damit gilt für den Mittelwert $Avg(n) = 1/n \sum_{i=1}^{k} i2^{i-1}$. Mit Induktion zeigt man $\sum_{i=1}^{k} i2^{i-1} = 2^k(k-1)+1$. Daraus folgt

$$Avg(n) = \frac{2^k(k-1)+1}{2^k-1} = \frac{2^k k}{2^k-1} - \frac{2^k}{2^k-1} + \frac{1}{2^k-1} = k + k/n - 1 \ .$$

□

[7] hier besser: Rekursionsinvariante

[8] Mit $\lg n$ wird hier und im folgenden der Logarithmus zur Basis Zwei bezeichnet.

Für große n gilt also $Avg(n) \approx k - 1$. Man benötigt also nur einen Versuch weniger als im Worst Case[9]!

Wenn das Telefonbuch nicht geordnet wäre, müsste man linear (ein Element nach dem anderen) suchen, dieser Algorithmus heißt LINEAR-SEARCH (1.2).

Algorithmus 1.2 LINEAR-SEARCH(A, k)

```
1: for j = 0 to length(A) − 1 do
2:     if k = A[j] then                    // Value found!
3:         return true
4: return false
```

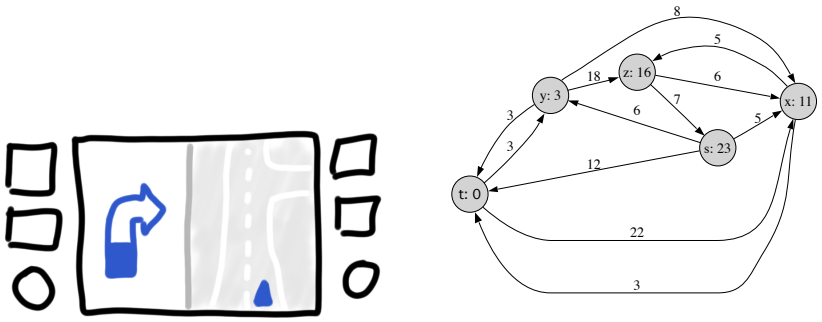
Lemma 1.4.3. *In einem Feld seien n paarweise verschiedenen Werte vorhanden. Für LINEAR-SEARCH dieses Feldes gilt für die erfolgreiche Suche*

- *Worst Case: Es werden n Vergleiche benötigt.*
- *Average Case: Es werden im Mittel $(n+1)/2$ Vergleiche benötigt.*
- *Best Case: Es wird 1 Vergleich benötigt.*

Beweis. Übung! □

1.5 Praxisbeispiel III: Navigation

Die praktische Relevanz von Algorithmen verdeutlicht Abbildung 1.5.



(a) Realität (b) Informatik

Abbildung 1.5: Kürzester Weg

[9] Als Informatiker überlege man sich, dass man durch k Entscheidungen 2^k Bits Information erhält; da die Zahl n genau k Bits enthält, ist das Ergebnis also nicht erstaunlich

Wir werden einige “Shortest Path” Algorithmen in einem späteren Kapitel besprechen!

1.6 Praxisbeispiel IV: Entscheidungen auswürfeln

Nehmen Sie an, Sie möchten Ja-/Nein-Entscheidungen unter Zuhilfenahme eines Würfels treffen. Ihnen stehen zwei verschiedenen Würfel zur Verfügung:

1. Ein Würfel, bei dem auf jeweils drei Seiten *Ja* und *Nein* steht.
2. Ein Würfel, der auf jeweils einer Seite *Ja* und *Nein* steht. Auf den anderen vier Seiten steht *Nochmals würfeln*, die Entscheidung wird also in die Zukunft geschoben.

Zu den wichtigen Rahmenbedingungen: Diese Würfel sind fair. Das bedeutet, dass die Wahrscheinlichkeit für das Liegenbleiben für alle Seiten gleich ist. Die Wahrscheinlichkeit, dass die Würfel auf einer Seite liegen bleiben, beträgt exakt $\frac{1}{6}$.

Welchen Würfel verwenden Sie? – Wenn Sie schnell ein Ergebnis haben möchten, dann sicherlich den ersten. Wenn Sie es spannend finden, mehrmals zu würfeln, und Spaß an Überraschungen haben, dann ist vielleicht der zweite Würfel Ihre Wahl.

Schauen wir uns jetzt die einfache Analyse der beiden Würfel an. Wir betrachten die Wahrscheinlichkeit der beiden Ereignisse *Ja gewürfelt* und *Nein gewürfelt* und die Anzahl der erforderlichen Würfe bis zu einer Entscheidung.

Für den ersten Würfel ist die Analyse ganz einfach. Die Wahrscheinlichkeit für das Eintreffen der beiden Ereignisse *Ja gewürfelt* und *Nein gewürfelt* ist jeweils $\frac{1}{2}$. Und natürlich erhalten wir unter den gegebenen Rahmenbedingungen eine Antwort nach genau einem Wurf.

Beim zweiten Würfel ist das nicht ganz offensichtlich. Nach Gefühl sind beide Entscheidungen ebenfalls gleichwahrscheinlich, da jeweils eine Seite für *Ja* und *Nein* steht und bei den anderen Seiten nochmals gewürfelt wird. Der Würfel kann schon im ersten Wurf eine Entscheidung liefern, aber es kann auch passieren, dass er nach einer beliebig gewählten hohen Zahl von Würfen noch keine Entscheidung liefert. Vermutlich werden es aber im Mittel nicht zu viele Würfe sein. Dieser Ansatz reicht bei der Analyse von Algorithmen und Datenstrukturen jedoch nicht aus. Gefühle, Intuition und Vermutungen können schnell in die Irre führen – auch wenn dies hier nicht der Fall ist. Jetzt berechnen wir die beiden Werte.

Zunächst widmen wir uns der Wahrscheinlichkeit des Ereignisses, dass *Ja*

gewürfelt wird. Dies schreiben wir jetzt mathematisch hin und rechnen es gleich aus. *Ja* kann im ersten Wurf gewürfelt werden oder im zweiten Wurf, wenn im ersten Wurf *Nochmals würfeln* erschien, usw. Oder allgemein: Im n-ten Wurf wird *Ja* gewürfelt und in den $n-1$ vorherigen Würfen *Nochmals würfeln*, $\forall n > 0$.

$$P(Ja) = \frac{1}{6} + \frac{4}{6} \cdot \frac{1}{6} + \left(\frac{4}{6}\right)^2 \cdot \frac{1}{6} + \ldots = \sum_{i \geq 0} \frac{1}{6} \cdot \left(\frac{4}{6}\right)^i = \frac{1}{6} \cdot \frac{1}{1 - \frac{4}{6}} = \frac{1}{2}$$

Natürlich gilt wegen der Rahmenbedingungen $P(Ja) = P(Nein)$.

Wie kommen wir auf die mittlere Anzahl von Würfen, bis der Würfel ein Ergebnis (*Ja* oder *Nein*) liefert?

- Die Wahrscheinlichkeit, dass der Würfel im ersten Wurf *Ja* oder *Nein* liefert, ist $\frac{2}{6}$.
- Die Wahrscheinlichkeit, dass der Würfel im zweiten Wurf *Ja* oder *Nein* liefert, ist $\frac{4}{6} \cdot \frac{2}{6}$. Im ersten Wurf muss das Ergebnis *Nochmals würfeln* gewesen sein, sonst hätte es keinen zweiten Wurf gegeben.
- Die Wahrscheinlichkeit, dass der Würfel im dritten Wurf *Ja* oder *Nein* liefert, ist $\left(\frac{4}{6}\right)^2 \cdot \frac{2}{6}$. In den vorherigen Würfen muss das Ergebnis jeweils *Nochmals würfeln* gewesen sein, sonst hätte es keinen dritten Wurf gegeben.
- ...

Auch dies können wir wieder als unendliche Summe schreiben und erhalten (dieses Mal mit weniger angegebenen Zwischenschritten) das Ergebnis, dass im Mittel (Average Case) das Ergebnis nach drei Würfen fest steht.

$$\frac{2}{6} \cdot 1 + \frac{4}{6} \cdot \frac{2}{6} \cdot 2 + \left(\frac{4}{6}\right)^2 \cdot \frac{2}{6} \cdot 3 + \ldots = \frac{2}{6} \left(\sum_{i \geq 0} i \left(\frac{4}{6}\right)^i + \sum_{i \geq 0} \left(\frac{4}{6}\right)^i \right) = 3$$

Bei diesen Berechnungen haben wir die *Geometrische Reihe* und die abgeleitete Geometrische Reihe verwendet. Beide Formeln gelten für $|z| < 1$.

$$\sum_{i \geq 0} z^i = \frac{1}{1-z} \text{ und } \sum_{i \geq 0} i z^i = \frac{z}{(1-z)^2}$$

Jetzt haben Sie gesehen, dass es beim zweiten Würfel, der (bei manchen Menschen) mehr Spaß verspricht, länger dauert, um ein Ergebnis zu erhalten. Dies ist ein Trade-off: Ein Faktor wird besser, während ein anderer

schlechter wird. In der Informatik gibt es einen klassischen Trade-off: Laufzeit und Speicherplatz. Beim Design von Algorithmen und Datenstrukturen werden Sie immer wieder Entscheidungen treffen, um einen der Aspekte zu verbessern. Oft verschlechtert sich der andere dadurch. Sie können sich also Laufzeit für Speicherplatz „kaufen“ und umgekehrt. Siehe dazu auch Abschnitt 14.1.1.

1.7 Weitere Praxisbeispiele

Algorithmen sind nicht nur (abstrakte) Mathematik, sondern haben direkten Einfluß auf unser Leben. Dieser Einfluß hat in den letzte Jahrzehnten durch die immer wichtiger werdende Kommunikation- und Datenverarbeitungstechnologie *immens* zugenommen. Algorithmen haben somit eine politische Bedeutung. Als Denkanstöße seien hier nur ein paar Zitate der letzten Jahre aufgeführt:

1.7.1 Algorithmen, Mustererkennung und „Terror“

> Im Dezember 2010 wurde ein Student im Regionalexpress von Kassel nach Frankfurt von zwei Bundespolizisten aufgefordert, seinen Ausweis vorzuzeigen. Er weigerte sich, da er annahm, allein wegen seiner schwarzen Haut angesprochen worden zu sein. Zwei Gerichtsverfahren später stand fest, dass er mit dieser Vermutung richtig lag – die Polizisten hatten bei ihrer „verdachtsunabhängigen Kontrolle“ gezielt nach Menschen gesucht, die ihnen als Ausländer erschienen waren. Sie arbeiteten nach einem bestimmten Muster.
>
> Die Polizisten sagten im Prozess aus, der Student sei ihnen aufgefallen, weil er dunkle Haut hatte, in einem voll besetzten Zug nicht saß, sondern durch den Gang ging, offensichtlich allein reiste und kein Gepäck besaß. Jedes einzelne dieser Merkmale ist harmlos, unbedeutend. Zusammen aber ergaben sie für die Polizisten das Muster „illegaler Einwanderer“. Andere Fakten interessierten die Beamten nicht – nicht sein deutscher Ausweis, nicht sein fehlerfreies Deutsch, nicht sein Auftreten.
>
> ...
>
> Einige amerikanische Städte wie Santa Cruz testen Algorithmen, um Autodiebe und Einbrecher zu fangen, noch bevor die eine Tür aufhebeln – vergleiche auch den Film Minority Report (`https://de.wikipedia.org/wiki/Minority_Report`).
>
> ...
>
> Kritiker dieser Technik fordern daher längst eine Ethik der Algorithmen, denn kontrollieren lassen sich diese nur durch Transparenz. Die will derzeit aber niemand, weder Google, noch Amazon,

noch die NSA. Was bedeutet, dass die Prämisse des demokratischen Strafrechtes nicht mehr gilt. Unschuldig bis zum Beweis der Schuld? Dieses Konzept kennen Algorithmen nicht.

(Kai Biermann, Quelle: `http://tinyurl.com/mt3ef76`)

1.7.2 Forderung nach „deutschen“ Algorithmen

CDU fordert „deutsche Algorithmen“ zur Verschlüsselung[10]

"Wir müssen hier auf deutsche Forschung, deutsche Algorithmen setzen.“

(Thomas Jarzombek, Quelle: `http://tinyurl.com/kveg9sm`)

1.7.3 Die Macht der Algorithmen

Über die „Macht der Algorithmen“ hat der Mathematiker Günter M. Ziegler in `http://page.math.tu-berlin.de/~mdmv/archive/18/mdmv-18-2-100.pdf` einen schönen Kurzartikel verfasst.

1.8 Fragen und Aufgaben zum Selbststudium

1. Programmieren Sie Algorithmen zur linearen und binären Suche in einem Array von Zahlen in C und testen diese! (Beachten Sie die Voraussetzung der Sortierung der Elemente bei der binären Suche.)
2. Berechnen Sie sowohl für die lineare als auch für die binären Suche den Best Case, den Worst Case und den Average Case für erfolgreiche und erfolglose Suche. (Beachten Sie, dass bei der Analyse der erfolglosen Suche das Verhältnis von den in der Liste vorhandenen Elementen zu allen möglichen Elementen eine Rolle spielt. Definieren Sie das in einer geeigneten Weise, damit Sie die Berechnungen durchführen können.)
3. Was müssen Sie berücksichtigen, wenn Sie lineare und binäre Suche analysieren möchten, also für erfolgreiche und erfolglose Suche eine gemeinsame Betrachtung anstellen? (Hinweis: Wonach wird wie oft gesucht?)
4. Vollziehen Sie die Rechnungen in Abschnitt 1.6 nach.
5. Machen Sie Experimente mit dem zweiten Würfel aus Abschnitt 1.6 und überprüfen Sie, wie nah Sie an den berechneten Wert 3 für die benötigte Anzahl von Würfen kommen.

[10] Das ist etwa so sinnvoll wie deutsche Physik, aber man kann von einem Netzpolitiker natürlich keine Sachkenntnis erwarten

Kapitel 2

Definition und Anforderungen an Algorithmen und Datenstrukturen

2.1 Der Begriff Algorithmus

2.1.1 Definition

Definition 2.1.1. *An algorithm is a finite, definite, effective procedure, with some output [Knu97a].*

Basierend auf dieser knappen Charakterisierung durch Knuth, definieren wir einen Algorithmus wie folgt:

Definition 2.1.2. *Ein Algorithmus A ist eine durch einen endlichen Text gegebene Verarbeitungsvorschrift, die zu jeder zulässigen Eingabe x aus einer Menge $\mathcal{X}$ eindeutig eine Reihe von Schritten vorschreibt. Wenn nach endlich vielen Schritten der Algorithmus anhält, erzeugt der Algorithmus ein Ergebnis $A(x)$.*

Man bemerke, dass in obiger Definition die Eigenschaft, nach endlichen Schritten zu terminieren, *nicht* Teil der Definition ist! Ein Algorithmus ist nach obiger Definition zudem immer *deterministisch*, d.h. beinhaltet keine zufälligen Elemente (im Unterschied zu *randomisierten* Algorithmen, s.u.).

Will man den Begriff des Algorithmus noch strenger formalisieren, so kann man das mithilfe des Begriffs der Turingmaschine tun:

Definition 2.1.3. *Eine Berechnungsvorschrift heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert.*

(Alternativ kann man Algorithmen nach Kleene auch mithilfe des sog. Lambda-Kalküls und des Begriffs der „rekursiven Funktion“ strenger formalisieren, was wir hier nicht behandeln.)

Algorithmen können in folgender Form spezifiziert werden:

- Umgangssprache
- Pseudocode
- Graphisch
- Programm in einer Programmiersprache
- Programm für formales Maschinenmodell
- Formale Spezifikation

Manche Algorithmen verwenden auch den Zufall:

Definition 2.1.4. *Ein* randomisierter *Algorithmus A ist ein Algorithmus, der (echte oder simulierte) Zufallsexperimente durchführen kann (siehe Abbildung 2.1).*

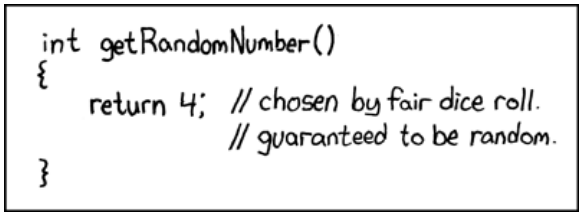


Abbildung 2.1: Random, Quelle: http://xkcd.com/

2.1.2 Typen von Algorithmen

Man unterscheidet die folgenden Typen von Algorithmen (s. [GS04])

1. applikative Algorithmen und
2. imperative Algorithmen.

Applikative Algorithmen sind in gewissem Sinne näher an der mathematischen Definition und basieren direkt auf der Theorie von Funktionen und ihren Auswertungen, während imperative Algorithmen ein Modell eines idealen Rechners unterstellen, das motiviert ist durch die heute verwendeten Registermaschinen. Die Literatur über Algorithmen behandelt überwiegend

Abbildung 2.2: Mohammad Kharazmi, Quelle:`http://en.wikipedia.org/wiki/File:Abu_Abdullah_Muhammad_bin_Musa_al-Khwarizmi_edit.png`

imperative Algorithmen. Applikative Algorithmen sind Grundlage für *Funktionale Sprachen* wie LISP, imperative werden in imperativen Programmiersprachen wie C, C++, Java verwendet. Aus Zeitgründen gehen wir in dieser Vorlesung nur auf imperative Algorithmen ein.

2.1.3 Ursprung

Der Begriff Algorithmus geht auf den persischen Mathematiker, Astronomen und Geographen, Mohammad Kharazmi (etwa 783-850, s. Abbildung 2.2)[1] zurück. Mohammad Kharazmi hatte ein Lehrbuch über das Rechnen mit indischen Ziffern (um 825) geschrieben. Die mittelalterliche lateinische Übersetzung beginnt mit den Worten *Dixit Algorismi* („Algorismi hat gesagt“). Im Mittelalter wurde daraus lateinisch *algorismus* als Bezeichnung für die Kunst des Rechnens mit arabischen Ziffern und als Titel für Schriften über diese Kunst.

2.2 Anforderungen an Algorithmen

Obgleich wir Algorithmen in dieser Vorlesung eher umgangssprachlich und durch (nicht vollständig formalisierten) Pseudocode darstellen, also *nicht*

[1] vollständiger Name: Abu Abdullah Mohammad Ibn Musa al-Khawarizmi

mit mathematischer Exaktheit, stellen wir die folgenden Bedingungen an einen Algorithmus:

1. Der Algorithmus muss *präzise* definiert sein einschließlich des Inputs und Outputs sowie jedes einzelnen Schrittes.
2. Es muss möglich sein, zu beweisen, dass der Algorithmus „korrekt“ ist.
3. Man muss über das Laufzeitverhalten räsonieren können.

2.3 Beispiel: Fakultät

Erinnerung: Die Fakultät $n!$ einer natürlichen Zahl n ist wie folgt definiert:

$$n! := \begin{cases} 1 \text{ falls } n = 0, \\ n\,(n-1)! \text{ falls } n > 0\ . \end{cases}$$

Ein Beispiel für einen einfachen Algorithmus zur Berechnung der Fakultät $n!$ einer natürlichen Zahl n ist im Algorithmus FACTORIAL 2.1 dargestellt[2]. Dieser Algorithmus verdeutlicht zugleich eins der wichtigsten Prinzipien zur

Algorithmus 2.1 FACTORIAL(n)

```
1: if n = 0 then
2:     return 1
3: else
4:     return n FACTORIAL(n − 1)
```

Konstruktion von Algorithmen: die Rekursion!

Aus dem Jargon File (`http://www.catb.org/~esr/jargon/`):

```
recursion: n.

     See recursion.
```

Rekursion ist kein ganz einfaches, aber fundamentales Konzept!

> “To understand recursion, one must first understand recursion.”
>
> “If you already know what recursion is, just remember the answer. Otherwise, find someone who is standing closer to Douglas Hofstadter than you are; then ask him or her what recursion is.” Andrew Plotkin

[2]Wir verwenden in diesem Skript zwei verschiedene Notationen zur Darstellung von Algorithmen, die im Kapitel 4 erklärt sind.

Es ist klar, dass es häufig zur Lösung von Problemen viele verschiedene mögliche Algorithmen gibt (die unterschiedliche Vor- und Nachteile haben können). So kann man die Fakultät $n!$ auch *iterativ* berechnen, wie im Algorithmus FACTORIAL-ITER 2.2 gezeigt.

Algorithmus 2.2 FACTORIAL-ITER(n)

```
1: k ← n
2: i ← 1
3: while k > 1 do
4:       i ← k · i
5:       k ← k − 1
6: return i
```

Lemma 2.3.1. *FACTORIAL und FACTORIAL-ITER sind korrekt.*

Beweis. (Es wird im folgenden vorausgesetzt, dass $n \in \mathbb{N}$, d.h. insbesondere dass $n \geq 0$ ist.)

Fall 1: FACTORIAL
Abbruch: FACTORIAL bricht ab, denn in jedem rekursiven Aufruf von FACTORIAL (Zeile 4) wird n um eins vermindert und Zeile 1 sorgt für die Abbruchbedingung.
Korrektheit: Induktion über n. Für $n = 0$ gilt offensichtlich

$$\text{FACTORIAL}(0) = 1 = 0!.$$

Induktionsschluß: $\text{FACTORIAL}(n+1) = (n+1) \cdot \text{FACTORIAL}(n) = (n+1) \cdot n!$ nach Induktionsannahme, also $\text{FACTORIAL}(n+1) = (n+1)!$ nach Definition der Fakultät.

Fall 2: FACTORIAL-ITER
Abbruch: FACTORIAL-ITER bricht ab, denn in jedem Schleifendurchlauf wird k um eins vermindert.
Korrektheit: Wir führen eine Schleifeninvariante (s.u.) ein. Wir betrachten den Ausdruck $k \cdot i$ in Zeile 4:

1. Durchlauf: $k \cdot i = n \cdot 1 = n$
2. Durchlauf: $k \cdot i = (n-1) \cdot n$
3. Durchlauf: $k \cdot i = (n-2) \cdot (n-1) \cdot n$
4. ...

Offensichtlich gilt für den j-ten Schleifendurchlauf $k_j \cdot i_j = n!/(k_j - 1)!$. Wir zeigen, dass der Ausdruck $k_j \cdot i_j \cdot (k_j - 1)! = i_j \cdot k_j!$ (mit $j = n - k + 1$) in jedem Durchlauf konstant bleibt und den Wert $n!$ hat. Für $j = 1$ gilt

$$i_1 \cdot k_1! = 1 \cdot n! = n!.$$

Mit Induktionsannahme $i_j \cdot k_j! = n!$ folgt

$$\begin{aligned} i_{j+1} \cdot k_{j+1}! &= k_j \cdot i_j \cdot (k_j - 1)! \quad \text{(Zeile 4 und 5)} \\ &= i_j \cdot k_j \cdot (k_j - 1)! \\ &= i_j \cdot k_j! = n! \quad \text{(Induktionsvoraussetzung)} \end{aligned}$$

Wenn die Abbruchbedingung erreicht ist, gilt aber $k_n = 1$, also

$$i_n = i_n \cdot k_n! = n!.$$

□

2.4 Beispiel: Sortieren durch Einfügen

In diesem Abschnitt untersuchen wir einen (etwas) komplexeren Algorithmus für das Sortieren, an dem wir einige grundlegende Prinzipien wie Schleifeninvarianten vertiefen und erste Betrachtungen zur Laufzeitanalyse vornehmen.

2.4.1 Algorithmus

Definition 2.4.1. *Sei eine Folge (Eingabe) von n Zahlen $(a_1, \ldots, a_n)$ gegeben. Die Zahlen a_i werden als Schlüssel bezeichnet. Eine Permutation (Umordnung) $(a'_1, \ldots, a'_n)$, auch $(a_{i_1}, \ldots, a_{i_n})$ bezeichnet, heißt Sortierung, wenn gilt*

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n.$$

Ein Algorithmus, der zu jeder Eingabe eine Sortierung berechnet, heißt Sortieralgorithmus.

Als Beispiel eines einfachen Sortieralgorithmus wählen wir das Sortieren durch Einfügen (siehe Algorithmus INSERTION-SORT 2.3 und Listing 2.4[3]), das dem Sortieren von Spielkarten nachempfunden ist [CLR04].

Angewandt auf die Eingangsfolge $a = (5, 2, 7, 4, 6, 1, 3)$ ergeben sich die 6 Sortierschritte in der Tabelle 2.1:

Hier kann man sehr gut sehen, wie die Größe des sortierten Feldes jeweils um ein Element pro Durchlauf der for-Schleife wächst.

2.4.2 Analyse

Definition 2.4.2. *Die Analyse von Algorithmen ist die wissenschaftliche Untersuchung der Korrektheit, der Performanz und des Ressourcenverbrauchs von Computerprogrammen.*

[3]Hier sehen Sie – wie im Kapitel 4 ausführlich ausgeführt – die zwei alternativen Darstellungen, die wir in diesem Skript verwenden.

Algorithmus 2.3 INSERTION-SORT(A)

1: **for** $j = 1$ to $length(A) - 1$ **do**
2: $key \leftarrow A[j]$
3: $i \leftarrow j - 1$
4: **while** $i >= 0$ **and** $A[i] > key$ **do**
5: $A[i + 1] \leftarrow A[i]$
6: $i \leftarrow i - 1$
7: $A[i + 1] \leftarrow key$

Algorithmus 2.4 INSERTION-SORT-P(A)

Listing 2.1 Sortieren durch Einfügen (Ausführbarer Pseudocode)

```
1  def insertsort(a):
2      for j in range(1, len(a)):
3          key = a[j]
4          i = j-1
5          while i >= 0 and a[i] > key:
6              a[i+1] = a[i]
7              i = i-1
8          a[i+1] = key
9      return a
```

Tabelle 2.1: Sortierschritte beim Insertion-Sort

0:	$(5, 2, 7, 4, 6, 1, 3)$
1:	$(2, 5, 7, 4, 6, 1, 3)$
2:	$(2, 5, 7, 4, 6, 1, 3)$
3:	$(2, 4, 5, 7, 6, 1, 3)$
4:	$(2, 4, 5, 6, 7, 1, 3)$
5:	$(1, 2, 4, 5, 6, 7, 3)$
6:	$(1, 2, 3, 4, 5, 6, 7)$

Also beantworten wir die folgenden Fragen:

- Wie kann man beweisen, dass etwas korrekt ist?
- Wie kann man Programme schnell machen?
- Wie kann man den Ressourcenverbrauch optimieren?

2.4.3 Korrektheit

Definition 2.4.3. *Sei S eine Schleife eines Algorithmus. Eine Eigenschaft E, die in jeder Iteration der Schleife wahr ist, heißt* Schleifeninvariante. *Damit erfüllt S:*

1. *Initialisierung: E ist vor der Iteration der Schleife wahr.*
2. *Fortsetzung: Wenn E vor der Iteration der Schleife wahr ist, ist E auch danach wahr.*

Für die Betrachtung von Algorithmen nützliche Schleifeninvarianten sind solche, die außerdem die folgende Eigenschaft besitzen:

3. *Terminierung: Bei Abbruch (oder Ende) der Schleife liefert die Invariante eine nützliche Eigenschaft (z. B. dass der Algorithmus korrekt ist).*

Lemma 2.4.1. *Schleifeninvariante: Zu Beginn jeder Iteration der* ***for****-Schleife in den Zeilen 1–7 im Algorithmus 2.3 besteht das Teilfeld* $A[0, \ldots, j-1]$ *aus den ursprünglich in* $A[0, \ldots, j-1]$ *enthaltenen Elementen, allerdings in geordneter Reihenfolge. Bei Abbruch sind* $A[0, \ldots, n-1]$ *sortiert.*

Beweis.

1. Es sei $j = 1$. Dann ist $A[0, \ldots, j-1] = A[0]$, also trivialerweise sortiert.
2. Wir betrachten $A[0, \ldots, j-1]$. Der Algorithmus verschiebt

   $$A_{j-1}, A_{j-2}, \ldots$$

   so weit nach rechts, bis A_j „korrekt“, d.h. sortiert eingesetzt ist.
3. Wenn die Schleife abbricht, ist $j = n$, also $A[0, \ldots, j-1]$ sortiert und somit auch die gesamte Sequenz $A[0, \ldots, n-1]$ sortiert.

□

2.4.4 Komplexität und Laufzeitverhalten

```
1: for j = 1 to length(A) − 1 do                 // (c, t) = (c_7, n)
2:     key ← A[j]                                // (c, t) = (c_1, n − 1)
3:     i ← j − 1                                 // (c, t) = (c_2, n − 1)
4:     while i >= 0 and A[i] > key do            // (c, t) = (c_5, Σ_{j=1}^{n−1} t_j)
5:         A[i + 1] ← A[i]                       // (c, t) = (c_3, Σ_{j=1}^{n−1} (t_j − 1))
6:         i ← i − 1                             // (c, t) = (c_4, Σ_{j=1}^{n−1} (t_j − 1))
7:     A[i + 1] ← key                            // (c, t) = (c_6, n − 1)
```

Notation: c_i gibt die (konstanten) Kosten (englisch “cost”) an, t (englisch “time”) die Anzahl der Ausführung der jeweiligen Zeile und t_j gibt an, wie oft die Schleifenbedingung in Zeile 4 für die j-te Schleife überprüft wird. Offensichtlich ist die Anzahl der Ausführungen des Schleifenrumpfes um eins kleiner (also $t_j - 1$) als die Anzahl der Überprüfungen der Bedingung in Zeile 4. Es gilt $1 \leq t_j \leq j + 1$ für $1 \leq j \leq n - 1$.

Damit ergibt sich die *Laufzeit* als Produkt aus Kosten und Zeit:

$$\begin{aligned} T(n) &= c_1(n-1) + c_2(n-1) + c_3 \sum_{j=1}^{n-1}(t_j - 1) + c_4 \sum_{j=1}^{n-1}(t_j - 1) \\ &+ c_5 \sum_{j=1}^{n-1} t_j + c_6(n-1) + c_7 n \end{aligned}$$

Im günstigen Fall, wenn das Feld bereits (aufsteigend) sortiert ist, gilt $A[j-1] \leq A[j] = key \quad \forall j \in \{1, \ldots, n-1\}$ (Zeile 4). Daher gilt $t_j = 1 \quad \forall j \in \{1, \ldots, n-1\}$ und somit

$$\begin{aligned} T(n) &= c_1(n-1) + c_2(n-1) + c_5(n-1) + c_6(n-1) + c_7 n \\ &= (c_1 + c_2 + c_5 + c_6 + c_7)n - (c_1 + c_2 + c_5 + c_6) \\ &= an + b, \end{aligned}$$

mit Konstanten a und b. Also ist in diesem Falle INSERT-SORT $\in \mathcal{O}(n)$, anschaulich: wächst *linear* mit n. (Diese Notation wird in Definition 3.1.2 noch erklärt.)

Im schlechtesten Fall, wenn das Feld absteigend sortiert ist, finden wir $A[i] > A[j] = key \quad \forall j \in \{1, \ldots, n-1\} \quad \forall i \in \{0, \ldots, j-1\}$. Somit gilt $t_j = j + 1 \quad \forall j \in \{1, \ldots, n-1\}$.

Mit der Gaußschen Summenformel

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

rechnen wir leicht nach, dass

$$\sum_{j=1}^{n-1} j = \frac{(n-1)n}{2}$$

und

$$\sum_{j=1}^{n-1}(j+1) = \frac{n(n+1)}{2} - 1$$

gelten und damit

$$\begin{aligned} T(n) &= c_1(n-1) + c_2(n-1) \\ &\quad + c_3\left(\frac{(n-1)n}{2}\right) + c_4\left(\frac{(n-1)n}{2}\right) \\ &\quad + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6(n-1) + c_7 n \\ &= an^2 + bn + c, \end{aligned}$$

mit Konstanten a, b und c. Also ist in diesem Falle INSERT-SORT $\in \mathcal{O}(n^2)$, anschaulich: wächst *quadratisch* mit n.

2.5 Der Begriff der Datenstruktur

2.5.1 Formale Definition

Bemerkung: Der nachfolgende Abschnitt ist – in mathematischer Hinsicht – relativ oberflächlich, versucht jedoch, das *Wesentliche* der Begriffsbildungen zu erfassen. Für Details sei auf die angegebene Literatur verwiesen! Im folgenden werden die Begriffe Datenstruktur und Datentyp – wie auch in der Literatur z.T. üblich – synonym verwendet!

Definition 2.5.1. *Eine Algebraische Struktur heißt ein geordnetes Paar $(A, (f_i)_{i\in I})$ mit der Trägermenge A und einer Familie von inneren, endlichstelligen Verknüpfungen $f_i : A^n \mapsto A$, die auch Grundoperationen heißen. Die Trägermenge A wird in der Informatik auch als* Sorte *bezeichnet.*

Definition 2.5.2. *Ein* Datentyp *(DT) besteht aus einer oder mehreren Mengen von Objekten und Operationen, die auf den Objekten dieser Mengen ausführbar sind. Die Kombination Objektmengen und Operationen wird auch als* Signatur *bezeichnet. Eine Signatur ist ein Paar (Sorten, Operationen), wobei Sorten Namen für die Objektmengen und Operationen Namen für die Operationen auf diesen Mengen repräsentieren. Die Signatur hat zunächst keinerlei inhaltliche Interpretation; sie beinhaltet keine Semantik (reine Syntax).*

Definition 2.5.3. *Ein* algebraischer Datentyp *(ADT)*[4] *besteht aus einem Datentyp zusammen mit einer inhaltlichen (semantischen) Interpretation der Objekt- und Operationsmengen. Die Konkretisierung der Operationsmenge führt zu Abstrakten Datentypen beziehungsweise Algebren.*

Definition 2.5.4. *Ein* konkreter Datentyp *ist ein abstrakter Datentyp, bei dem die Objekt- und Operationsmengen konkret – einschließlich des Wertebereichs – bestimmt sind.*

[4] Algebraic Data Type

In der Regel liegen in der Informatik *mehrsortige* Algebren vor, d.h. eine Algebra mit mehreren Sorten als Trägermenge.

Als Beispiel seien die natürlichen Zahlen $\mathbb{N}$ vereint mit den Wahrheitswerten, also $A := \mathbb{N} \cup \{TRUE, FALSE\}$ genannt mit den (Grund-) Operationen $+$, $-$, $*$, $\div$, die $\mathbb{N}$ auf sich selbst abbilden, $\neg$, $\wedge$, $\vee$, die die Wahrheitswerte $\{TRUE, FALSE\}$ auf sich selbst abbilden und $=$, $<$, $>$, $\leq$ und $\geq$, die $\mathbb{N}$ auf die Wahrheitswerte abbilden.

Oft sind nicht alle Grundoperationen für alle Werte definiert – man spricht von partiell definierten Operationen. Z. B. ist $x \div y$ nicht für alle Paare $(x, y) \in \mathbb{N} \times \mathbb{N}$ definiert und der Ausdruck $3 < TRUE$ ist sinnlos. Man benutzt dann oft den speziellen „Wert“ „Bottom“ $\perp$, um dennoch einen Wert zuweisen zu können, also z. B. $4 \div 3 \mapsto \perp$ oder $3 < TRUE \mapsto \perp$. So kann man die Operationen auf *allen* Werten definieren.

Objektorientierte Programmiersprachen unterstützen durch ihr *Typenkonzept* die Erstellung von ADTs, da hier Daten und Operationen gebunden und die Daten gekapselt werden können. Einige modulare Programmiersprachen wie Ada oder Modula-2 unterstützen ebenfalls die Erstellung von abstrakten Datentypen. Bei vielen imperativen und prozeduralen Programmiersprachen, insbesondere C, ist eine Kapselung nicht oder nur unzureichend möglich und der Entwickler muss darauf achten, dass jeweils die „richtigen“ Operationen auf die richtigen Datentypen angewandt werden.

Mit dem nachfolgend definierten Konzept der Termalgebra kann man zu jedem ADT Realisierungen konstruieren:

Definition 2.5.5. *Eine Termalgebra ist eine algebraische Struktur, die aus einer Signatur frei generiert wird. „Frei generiert“ bedeutet hier – vereinfacht ausgedrückt –, dass die Familie der inneren, endlichstelligen Verknüpfungen $f_i : A^n \mapsto A$ aus denjenigen endlichen, formalen Verknüpfungen – den Termen – der Signatur bestehen.*

Ein ADT kann also als eine Termalgebra aufgefasst werden.

Bemerkung: Man kann mithilfe einer Termalgebra stets ein Model, d.h. eine konkrete Realisierung, für einen ADT generieren, indem man die Operationen durch die Signatur generiert und durch Restklassen (Quotienten-) Bildung gleiche Terme miteinander identifiziert. Für Details sei auf z. B. [GS04] verwiesen.

Ein ADT hat kein eindeutiges Modell. Als Beispiel betrachten wir die Algebra der Wahrheitswerte $\{TRUE, FALSE\}$:

```
Bool
Operationen TRUE: -> Bool
            FALSE: -> Bool
```

```
End Bool
```

Die folgenden Modelle stellen beide valide Realisierungen der Algebra der Wahrheitswerte $\{TRUE, FALSE\}$ dar:

Modell 1:

$$\begin{aligned} \text{Mengen: } & \{TRUE, FALSE\} \to \mathbb{N} \\ \text{Funktionen: } & TRUE \to 1 \\ & FALSE \to 0 \end{aligned}$$

Modell 2:

$$\begin{aligned} \text{Mengen: } & \{TRUE, FALSE\} \to \{true, false\} \\ \text{Funktionen: } & TRUE \to true \\ & FALSE \to false \end{aligned}$$

Offensichtlich ist die erste Realisierung unnötig *groß*, da die meisten (unendlich viele) Elemente gar nicht benutzt werden.

Der Vollständigkeit halber sei auf die Problematik, die durch triviale Modelle entstehen, hingewiesen. So ist z. B. das folgende Modell eine triviale Realisierung der Algebra der Wahrheitswerte $\{TRUE, FALSE\}$:

$$\begin{aligned} \text{Mengen: } & \{TRUE, FALSE\} \to \{1\} \\ \text{Funktionen: } & TRUE \to 1 \\ & FALSE \to 1 \end{aligned}$$

Diese Realisierung ist aber in der Regel nicht gewünscht. Um sie auszuschließen, muß man zusätzliche Axiome (Operationen) mit einbeziehen, etwa $TRUE \neq FALSE$. Die Details sprengen den Rahmen dieses Skripts.

2.5.2 Beispiel: Integer

Beispiel: Datentyp Integer (kennen Sie aus Einführung in C!)

```
Simple Integer
Sorten int, bool
Operationen zero: -> int
```

```
            succ: int -> int
            +   : int x int -> int
            -   : int x int -> bool
            <   : int x int -> bool
End Simple Integer
```

Der Datentyp Integer wird zum abstrakten Datentyp Integer durch Zuordnung einer Semantik:

$$
\begin{aligned}
\text{Mengen: } & int \to \mathbb{N} \\
& bool \to \{TRUE, FALSE\} \\
\text{Funktionen: } & zero \to 0 \\
& succ \to x \mapsto x + 1 \\
& + \to (a, b) \mapsto a + b \\
& - \to (a, b) \mapsto a - b \\
& < \to (a, b) \mapsto a < b
\end{aligned}
$$

Bemerkung: Man muß im obigen Beispiel beachten, dass einige Operationen wie z. B. $-$ nicht für alle Argumentkombinationen definiert sind, oder das Modell um $\perp$ ergänzen.

2.5.3 Beispiel: Komplexe Datentypen

Mit Hilfe der eingebauten Datentypen können komplexere, zusammengesetzte ADTs konstruiert werden, z. B unter Verwendung des `struct` Schlüsselworts:

```
struct datum
 {
   int tag;
   char monat[10];
   int jahr;
 };
```

2.5.4 Beispiel: „Eingebaute“ Datentypen

Beispiel: „Eingebaute“ Datentypen in C, siehe Tabelle 2.2.

2.5.5 Beispiel: Objektorientierte Datentypen

Ein Codebeispiel aus der Crypto++ Library 5.6.2, einer C++ Bibliothek für Kryptographie:

Tabelle 2.2: C Datentypen

Type	Keyword	Bytes	Wertebereich
character	char	1	-128 ... 127
unsigned character	unsigned char	1	0 ... 255
integer	int	2	-32 768 ... 32 767
short integer	short	2	-32 768 ... 32 767
long	integer long	4	-2 147 483 648 ... 2 147 483 647
unsigned integer	unsigned int	2	0 ... 65 535
unsigned short integer	unsigned short	2	0 ... 65 535
unsigned long integer	unsigned long	4	0 ... 4 294 967 295
single-prec. float.-point	float	4	1.17E-38 ... 3.4E38
double-prec. float.-point	double	8	2.2E-308 ... 1.8E308

```
class Integer : public ASN1Object
{
public:
...
//! creates the zero integer
Integer();

//! copy constructor
Integer(const Integer& t);

//! convert from signed long
Integer(signed long value);
...
//!
bool IsZero() const {return !*this;}
//!
bool NotZero() const {return !IsZero();}
//!
bool IsNegative() const {return sign == NEGATIVE;}
//!
bool NotNegative() const {return !IsNegative();}
//!
bool IsPositive() const {return NotNegative() && NotZero();}
//!
bool NotPositive() const {return !IsPositive();}
//!
bool IsEven() const {return GetBit(0) == 0;}
//!
bool IsOdd() const {return GetBit(0) == 1;}
...

Integer&  operator=(const Integer& t);

//!
Integer&  operator+=(const Integer& t);
//!
Integer&  operator-=(const Integer& t);
//!
Integer&  operator*=(const Integer& t) {return *this = Times(t);}
//!
```

```
Integer&  operator/=(const Integer& t) {return *this = DividedBy(t);}
//!
Integer&  operator%=(const Integer& t) {return *this = Modulo(t);}
//!
Integer&  operator/=(word t)  {return *this = DividedBy(t);}
//!
Integer&  operator%=(word t)  {return *this = Modulo(t);}

//!
Integer&  operator<<=(unsigned int);
//!
Integer&  operator>>=(unsigned int);

...
//! \name UNARY OPERATORS
//@{
//!
bool operator!() const;
//!
Integer  operator+() const {return *this;}
//!
Integer  operator-() const;
//!
Integer& operator++();
//!
Integer& operator--();
//!
Integer  operator++(int) {Integer temp = *this; ++*this; return temp;}
//!
Integer  operator--(int) {Integer temp = *this; --*this; return temp;}
//@}
...
//! \name BINARY OPERATORS
//@{
//! signed comparison
/*! \retval -1 if *this < a
\retval  0 if *this = a
\retval  1 if *this > a
*/
int Compare(const Integer& a) const;

//!
Integer Plus(const Integer &b) const;
//!
Integer Minus(const Integer &b) const;
//!
Integer Times(const Integer &b) const;
//!
Integer DividedBy(const Integer &b) const;
//!
Integer Modulo(const Integer &b) const;
//!
Integer DividedBy(word b) const;
//!
```

```
word Modulo(word b) const;

//!
Integer operator>>(unsigned int n) const {return Integer(*this)>>=n;}
//!
Integer operator<<(unsigned int n) const {return Integer(*this)<<=n;}
//@}
...
}
```

2.5.6 Beispiel: Stapel

Abbildung 2.3 zeigt einen Datentyp *Stapel* (englisch "Stack") zusammen mit den grundlegenden Operationen, die ein Stack unterstützt: PUSH und POP. Ein Stack ist ein Last-In-First-Out (LIFO) Speicher und hat das folgende

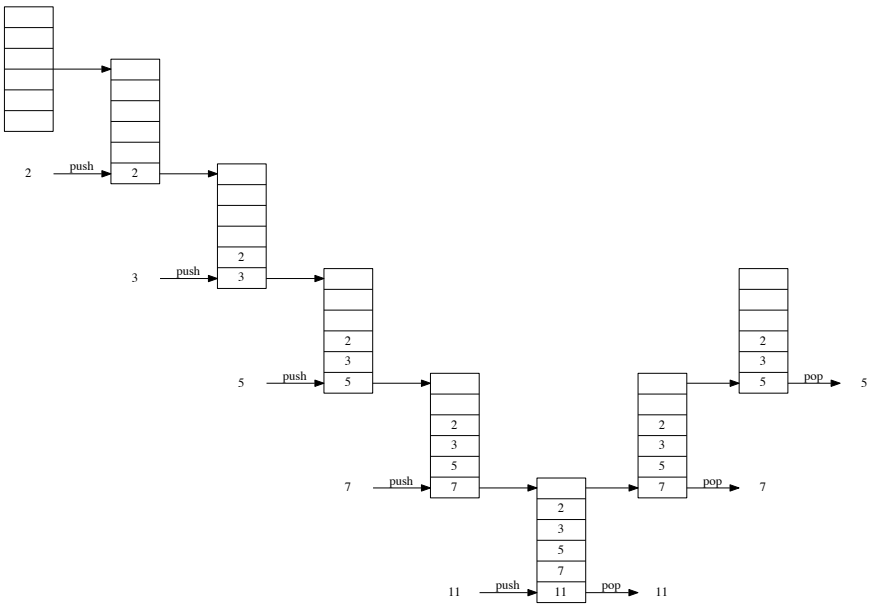


Abbildung 2.3: Stack

API, d.h. unterstützt die folgenden Operationen:

- `empty(())`: Leeren Stack erzeugen.
- `push(s, x)`: Neues Element x auf den Stack s legen.
- `pop(s)`: Oberstes Element des Stacks s (falls vorhanden) entfernen.
- `top(s)`: Oberstes Element des Stacks s ausgeben (falls vorhanden).
- `isempty(s)`: Ausgabe "true" falls Stack s leer, "false" sonst.

Die oben gegebene Beschreibung eines Stapels ist offensichtlich umgangssprachlich und nicht mathematisch exakt. Stattdessen lässt sich die Semantik eines Datentyps auch formal definieren:

- Typen:
  - $KEYS$
  - $STACKS$
  - $BOOLEAN$
- Operationen:
  - $PUSH\colon STACKS \times KEYS \to STACKS$
  - $POP\colon STACKS \to STACKS \times KEYS$
  - $TOP\colon STACKS \to KEYS$
  - $ISEMPTY\colon STACKS \to BOOLEAN$

Oder man definiert ein mathematisches Modell:

- Ein Stack ist ein n-Tupel $(a_1, \ldots, a_n)$
- $Keys := \mathcal{K} \neq \emptyset$
- $Stacks := S := \bigcup_{n \geq 0} \mathcal{K}^n$, entsprechend (beliebig langen) Sequenzen von Schlüsseln
- $Boolean := \{TRUE, FALSE\}$
- $PUSH((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$
- usw.

Zu guter Letzt lassen sich Datentypen auch axiomatisch definieren. Für alle Stapel (Stacks) s und alle Keys x soll gelten:

- $push : S \times \mathcal{K} \to S$
- $pop : S \to S \times \mathcal{K}$
- $top : S \to \mathcal{K}$
- $pop(push(s, x)) = (s, x)$
- $push(pop(s)) = s$
- $top(push(s, x)) = x$
- $isempty(push(s, x)) = false$
- $isempty(empty()) = true$
- usw.

Z. B. bedeuten $pop(push(s, x)) = (s, x)$ und $push(pop(s)) = s$, dass *push* und *pop* invers zueinander sind, mit anderen Worten wenn man ein Element auf dem Stack ablegt und es danach wieder mithilfe von *pop* vom Stack entfernt, dann ist der Gesamtzustand des Systems nicht verändert; ebenso, wenn man ein Element zuerst mithilfe von *pop* vom nicht leeren Stack entfernt und es danach wieder mit *push* auf den Stack legt. Der Mathematiker oder theoretische Informatiker sagt dann, dass das Diagramm 2.4 „kommutativ“ ist, d.h. es ist egal wie man die Pfeile kombiniert (*id* bezeichnet die Identität).

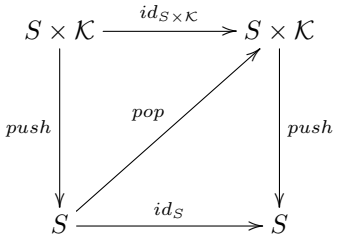


Abbildung 2.4: Stack Inverse $pop = push^{-1}$ und $push = pop^{-1}$

Dies ermöglicht u.a. die Anwendung formaler Methoden zum Beweis der Korrektheit von Implementierungen (s. z. B. [GS04]). Wir werden in dieser Vorlesung meist die umgangssprachliche, selten auch die mathematische Formulierung wählen.

2.5.7 Implementierungsaspekte

Es gibt verschieden Möglichkeiten, Stacks zu implementieren:

1. Mit Hilfe eines Feldes (Arrays) $A[0, \ldots, n-1]$ und einer Variable für die Stackhöhe $int : h$
2. Mit Hilfe verketteter Listen (kommt später)

Die Operationen des APIs sind dabei Funktionsaufrufe mit den entsprechenden Parametern (Übungsaufgabe!). Das API ist dabei bei prozeduralen Sprachen nicht gekapselt, d.h. der Aufrufer muss gewisse Konventionen in seinem Code befolgen. Aus diesem Grund verwendet man bei objekt-orientierten Programmiersprachen konsequent das sogenannte Kapselungsprinzip (eng. Information Hiding) an, bei dem der Datentyp als Klasse implementiert wird und das API als öffentlich sichtbare Methoden.

2.5.8 Analyse

Der Vorteil einer formalen Definition ist, dass man nun die Eigenschaften eines Datentyps mathematisch *beweisen* kann!

Korrektheit

Lemma 2.5.1. *Die Implementierung des Stapels (Stack) mithilfe eines Feldes realisiert das mathematische Modell, solange kein Laufzeitfehler „Speicherüberlauf“ auftritt.*

Beweis. Übung! □

Zeit- und Platzbedarf

Wir können leicht die Laufzeitanalyse vornehmen:

Lemma 2.5.2. *Die Implementierung des Stapels mithilfe eines Feldes realisiert jede Operation als $\mathcal{O}(1)$. Mit anderen Worten: die Operationen benötigen konstant lange Zeit unabhängig von der (befüllten) Größe des Stapels.*

Beweis. Übung! □

Ebenso offensichtlich gilt:

Lemma 2.5.3. *Die Implementierung des Stapels mithilfe eines Feldes erfordert $MAX \times k$ Speicherplatz, wobei MAX die Feldlänge angibt und k den Platzbedarf des im Stapel verwendeten Typs, sowie Platz für eine Variable für die Stackhöhe.*

Beweis. Übung! □

2.6 Zusammenhang Mathematik, Algorithmen, Datenstrukturen und Programmierung

Die nachstehende Grafik 2.5 verdeutlicht den Zusammenhang zwischen Mathematik, Algorithmen, Datenstrukturen und Programmierung.

2.7 Fragen und Aufgaben zum Selbststudium

1. Geben Sie zwei Definitionen des Begriffs Algorithmus!
2. Welche Anforderungen muss man an einen Algorithmus stellen?
3. Warum ist eine rekursive Definition nicht zirkulär? Was müssen Sie bei allen rekursiven Definitionen überprüfen?
4. Wie viele Schleifen enthält INSERTION-SORT?
5. Warum endet die erste Schleife in INSERTION-SORT bei $length(a) - 1$ und nicht bei $length(a)$ und warum startet man bei $j = 1$ und nicht bei $j = 0$?

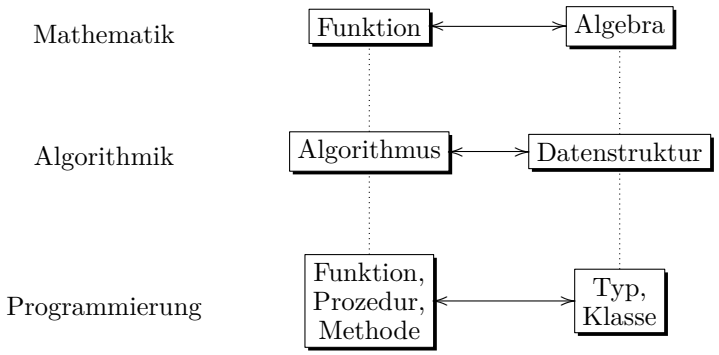


Abbildung 2.5: Zusammenhang Mathematik, Algorithmen, Datenstrukturen und Programmierung

6. Was macht Zeile 7 im Algorithmus 2.3 INSERTION-SORT und warum ist sie notwendig?
7. Sortieren Sie die Folge $a = (7, 13, 73, 4, 61, 2, 3)$ mithilfe von INSERTION-SORT per Hand und geben Sie jeden einzelnen Schritt an!
8. Geben Sie für die Sortierung (vorherige Aufgabe) an, von welcher Schleife die Daten gerade bearbeitet werden!
9. Schreiben Sie INSERTION-SORT so um, dass absteigend sortiert wird, die größten Elemente also am Anfang stehen!
10. Geben Sie Pseudocode für LINEAR-SEARCH an und beweisen Sie mithilfe einer Schleifeninvariante, dass Ihr Algorithmus korrekt ist!
11. Analysieren sie SELECTIONSORT (`http://de.wikipedia.org/wiki/Selectionsort`) in Bezug auf Korrektheit und Laufzeit!
12. Implementieren sie einen Stack mithilfe eines Feldes fester Größe! Was passiert, wenn die Anzahl der PUSH Operationen größer ist als die Feldgröße!
13. Beweisen Sie Lemma 2.5.1. Gehen Sie von einer axiomatischen Charakterisierung des Stacks aus, d.h. zeigen Sie, dass
    - $pop(push(s, x)) = (s, x)$
    - $push(pop(s)) = s$
    - $top(push(s, x)) = x$
    - $isempty(push(s, x)) = false$
    - $isempty(empty()) = true$

gilt!

14. Beweisen Sie Lemma 2.5.2!
15. Beweisen Sie Lemma 2.5.3! (Gehen Sie ähnlich vor, wie in 2.4.4.!)

Kapitel 3

Wachstum und Komplexitätsbetrachtungen

In diesem Kapitel präzisieren wir die Konzepte zur Analyse von Wachstum und Komplexität und entwickeln die dafür notwendige Notation. Das sogenannte $\mathcal{O}(n)$-Kalkül bzw. die $\mathcal{O}(n)$-Notation ist im englischen Sprachraum auch als "Big-O Notation" bekannt; im deutschen Sprachraum nennt man sie auch „Landausymbole" oder „Landaunotation" nach dem deutschen Zahlentheoretiker Edmund Landau. $\mathcal{O}$ stand ursprünglich für Omikron und als Symbol für Ordnung. Es wurde erstmals vom deutschen Zahlentheoretiker Paul Bachmann verwendet.

3.1 $\mathcal{O}(n)$-Notation

3.1.1 Intuition

Warnung: Dieser Abschnitt ist nicht immer mathematisch exakt, was aber beabsichtigt ist!

Zur Erinnerung: Beim Sortieren durch Einfügen galt für die Laufzeit im ungünstigen Fall die folgende Formel:

$$\begin{aligned} T(n) &= c_1(n-1) + c_2(n-1) \\ &\quad + c_3\left(\frac{(n-1)n}{2}\right) + c_4\left(\frac{(n-1)n}{2}\right) \\ &\quad + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6(n-1) + c_7 n \\ &= an^2 + bn + c, \end{aligned}$$

Zur Vereinfachung schreiben wir im folgenden stets:

$$T(n) = an^2 + bn + c \ .$$

Der genaue Wert der Konstanten a, b und c ist schwierig zu ermitteln und (u.a.) vom Rechner abhängig, aber – wie wir noch sehen werden – auch nicht besonders interessant!

Um eine Intuition zu entwickeln, betrachten wir weitere einfache Algorithmen, z. B. den Algorithmus 3.1.

Algorithmus 3.1 SCHLEIFE (n)

```
1: for j = 0 to n − 1 do
2:     FOO(j)
```

0	1	2	$\cdots\cdots$	$n-1$

Tabelle 3.1: Werte für Parameter j von FOO in Algorithmus 3.1

Unter der Annahme, dass FOO *nicht* von j abhängt, gilt für die Laufzeit T von SCHLEIFE: $T(n) = an + b$. (Warum?!)

Für den Algorithmus 3.2 gilt dagegen unter der Annahme, dass BAR *nicht* von i und j abhängt, für die Laufzeit T' von DOPPELSCHLEIFE:[1]

$$T'(n) = a'n^2 + b'n + c' \ .$$

Algorithmus 3.2 DOPPELSCHLEIFE (n)

```
1: for i = 0 to n − 1 do
2:     for j = 0 to n − 1 do
3:         BAR(i, j)
```

(0,0)	(0,1)	(0,2)	$\cdots$	(0,$n-1$)
(1,0)	(1,1)	(1,2)	$\cdots$	(1,$n-1$)
(2,0)	(2,1)	(2,2)	$\cdots$	(2,$n-1$)
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
($n-1$,0)	($n-1$,1)	($n-1$,2)	$\cdots$	($n-1$,$n-1$)

Tabelle 3.2: Werte für Parameter (i, j) von BAR in Algorithmus 3.2

Frage: Welcher Algorithmus ist „besser“, SCHLEIFE oder DOPPELSCHLEIFE?
Antwort: Das hängt davon ab, wie man „besser“ definiert!

[1] Warum das so ist, werden wir u.a. in den Übungen untersuchen!

Frage: Welcher Algorithmus ist „schneller“, SCHLEIFE oder DOPPEL-SCHLEIFE?
Antwort: Betrachten wir zunächst

$$T(n) = an + b$$
$$T'(n) = a'n^2 + b'n + c',$$

wobei die Konstanten a und a' positiv sind.

Lemma 3.1.1. *Für n genügend groß, gilt immer $T(n) < T'(n)$*

Beweis.

$$\begin{aligned} T'(n) - T(n) &= a'n^2 + b'n + c' - an - b \\ &= a'n^2 + (b' - a)n + (c' - b) \\ &= a'(n^2 + dn + e) \\ &= a' \left((n + d/2)^2 + e - d^2/4\right) \end{aligned} \tag{3.1}$$

mit $d := \frac{b'-a}{a'}$ und $e := \frac{c'-b}{a'}$ und positivem a'. Falls $d^2/4 - e \leq 0$ ist $T'(n) - T(n)$ immer positiv, und im anderen Fall $d^2/4 - e \geq 0$ gilt für

$$n > \left(\sqrt{d^2/4 - e} - d/2\right), \tag{3.2}$$

ebenfalls, dass $T'(n) - T(n)$ positiv ist. □

Wir sehen also, dass für „große“ n, d.h. für große Eingabewerte der Algorithmus 3.1 *immer* besser ist, und zwar unabhängig von den konkreten Konstanten (Rechner!) a, b, a', b' und c'! Das heißt, in einem gewissen Sinn sind die Konstanten unwichtig, was zählt ist nur der Ausdruck n bzw. n^2.

Ziel der $\mathcal{O}(n)$-Notation bzw. des $\mathcal{O}(n)$-Kalküls ist es, genau diese unwichtigen Elemente zu vernachlässigen und sich auf das Wesentliche zu konzentrieren!

Bevor wir die mathematische Definition in nachstehendem Abschnitt genauer anschauen, definieren wir anschaulich:

Definition 3.1.1. *Wir definieren die folgenden Komplexitätsklassen:*

1. *Ein Algorithmus $A(n)$ ist in $\mathcal{O}(1)$, wenn er konstante Zeit benötigt. Intuition: Berechnung hängt* nicht *von Größe n des Problems ab. Anschaulich: Ein doppelt so großes Problem benötigt auch im ungünstigsten Fall nicht länger.*

2. *Ein Algorithmus $A(n)$ ist in $\mathcal{O}(n)$, wenn er – in Abhängigkeit von n – maximal linear wächst.*
   Anschaulich: Ein doppelt so großes Problem benötigt auch im ungünstigsten Fall nur doppelt so lange.

3. *Ein Algorithmus $A(n)$ ist in $\mathcal{O}(n^2)$, wenn er – in Abhängigkeit von n – maximal quadratisch wächst.*
   Anschaulich: Ein doppelt so großes Problem benötigt auch im ungünstigsten Fall nur vier mal so lange.

3.1.2 Beispiele

Einfache Beispiele

Betrachten wir den Algorithmus 3.3! Er ist offensichtlich in $\mathcal{O}(1)$, da die Länge des Feldes A keine Rolle spielt und der Zugriff auf das erste Feld $A[0]$ konstante Zeit benötigt.

Algorithmus 3.3 HEAD(A)

```
1: return A[0]
```

Betrachten wir den Algorithmus 3.4! Er ist genau dann in $\mathcal{O}(1)$, wenn gilt: $length(A) \in \mathcal{O}(1)$ und dies hängt i.a. von der Implementierungssprache ab. (Die Funktion $length(A)$ kann z. B. auch in $\mathcal{O}(n)$ sein!)

Algorithmus 3.4 TAIL(A)

```
1: return A[length(A) − 1]
```

Betrachten wir den Algorithmus 3.5! Er ist offensichtlich in $\mathcal{O}(n)$, da die Schleife n mal durchlaufen wird.

Algorithmus 3.5 SCHLEIFE1(n)

```
1: for i = 0 to n − 1 do
2:     a ← i
3:     b ← a
4:     c ← a ∗ b
```

Betrachten wir den Algorithmus 3.6!

Die Laufzeit für die Zeilen 3 und 4 von Algorithmus 3.6 ist $a + \sum_{i=0}^{j}(a + b)$. Mit dem gleichen Ansatz kann die Laufzeit der äußeren Schleife berechnet

Algorithmus 3.6 SCHLEIFE2(n)

```
1: k ← 0                                  // c_1
2: for j = 0 to n − 1 do                  // a
3:       for i = 0 to j do                // a
4:             k ← k + i                  // b
5: return k                               // c_2
```

werden und mit $c := c_1 + c_2$ gilt:

$$\begin{aligned} T(n) &= c + a + \sum_{j=0}^{n-1} \left[a + a + \sum_{i=0}^{j} (a+b) \right] \\ &= n^2 \cdot \frac{a+b}{2} + n \cdot \frac{5a+b}{2} + a + c \ . \end{aligned}$$

Offensichtlich liegt die Laufzeit von SCHLEIFE2 somit in $\mathcal{O}(n^2)$.

Betrachten wir zu guter Letzt den Algorithmus 3.7!

Algorithmus 3.7 SCHLEIFE3(n)

```
1: k ← 0                                  // c_1
2: for j = 0 to n − 1 do                  // a
3:       for i = 0 to n − 1 do            // a
4:             k ← f(i, j)                // b
5: return k                               // c_2
```

Falls $f(i,j)$ in konstanter Zeit abläuft (präziser: sich durch eine Konstante b nach oben abschätzen läßt), ist er ebenfalls in $\mathcal{O}(n^2)$, denn für die Laufzeit T von SCHLEIFE3 gilt analog zu SCHLEIFE2:

$$\begin{aligned} T(n) &= c + a + \sum_{j=0}^{n-1} \left[a + a + \sum_{i=0}^{n-1} (a+b) \right] \\ &= n^2 \cdot (a+b) + n \cdot 2a + a + c \ . \end{aligned}$$

Wir werden im folgenden Regeln kennenlernen, um mit der $\mathcal{O}$-Notation rechnen zu können. Insbesondere, können wir komplizierte Fälle auf einfache zurückführen. So kann man z. B. unwichtige Dinge weglassen:

Lemma 3.1.2. *Für n genügend groß, gilt immer $T'(n) = a'n^2 + b'n + c' < Cn^2$ für eine positive Konstante C.*

Beweis. Beweis, der sich auf allgemeine Polynome übertragen läßt:

$$\begin{aligned} |a'n^2 + b'n + c'| &\leq |a'n^2| + |b'n| + |c'| \\ &\leq |a'|n^2 + |b'|n^2 + |c'|n^2 \\ &= \left(|a'| + |b'| + |c'|\right) n^2 =: Cn^2 \end{aligned} \tag{3.3}$$

□

Das heißt, dass nur die höchste Potenz interessant ist. Ebenso kann man die Landausymbole addieren. Übertragen auf die Programmierung bedeutet dies, dass die Gesamtlaufzeit sich durch die Addition der einzelnen Laufzeiten ergib, falls Algorithmen nacheinander (seriell) ablaufen.

Graphisch kann man sich diese Regeln verdeutlichen, wie in Abbildung 3.1 dargestellt!

Mit solchen Rechenregeln kann man später u.a. zeigen, dass die folgende Hierarchie von Funktionen, die man als Informatiker „intuitiv“ kennen sollte, gilt:

$$\begin{aligned} \mathcal{O}(\log n) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(\tfrac{n}{\log n}) &< \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n(\log n)^2) \\ &< \mathcal{O}(n^2) < \mathcal{O}(n^3) < \dots \\ &< \mathcal{O}(2^n) < \mathcal{O}(e^n) < \mathcal{O}(3^n) < \dots \\ &< \mathcal{O}(n!) \\ &< \mathcal{O}(n^n) \end{aligned}$$

Beispiele mit Rekursion

Algorithmen werden häufig rekursiv definiert bzw. benutzen Rekursion. Hierbei treten dann für die Laufzeit sogenannte Rekursionsgleichungen auf, die sich nicht so einfach lösen lassen wie die oben angegebene Beispiele.

Betrachten wir den „prototypischen“ rekursiven Algorithmus 3.8!

Algorithmus 3.8 RECURSIVE(A)

```
1: if BASECASE(A) then                    // BREAK
2:     return
3: A' ← REDUCE(A)                         // Reduce Problem
4: COMBINE(A, RECURSIVE(A'))              // Combine with Recursion Result
```

Rekursive Algorithmen können wir mit u.a. Rekursionsbäumen lösen, aber häufig auch einfach „raten“. Ein Hilfsmittel dazu sind Rekursionsgleichungen. Betrachten wir noch einmal die Fakultät (Algorithmus 3.9).

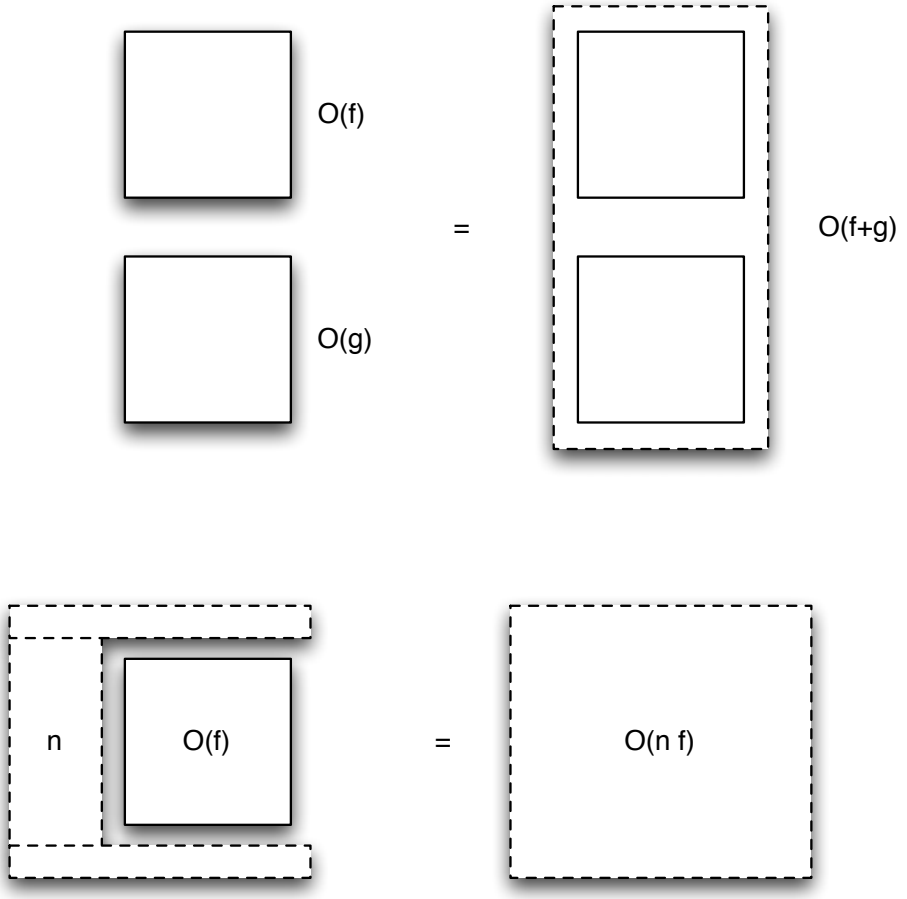


Abbildung 3.1: Big-O Regeln Komposition

Algorithmus 3.9 FACTORIAL(n)

```
1: if n = 0 then
2:     return 1
3: n' ← n − 1
4: return n FACTORIAL(n')
```

Für die Laufzeit $T(n)$ gilt die folgende Rekursionsgleichung (die Konstante b gibt den Aufwand der Multiplikation des Resultats des rekursiven Aufrufs mit n an, die Konstante a die Laufzeit von FACTORIAL(0)):

$$T(0) = a$$
$$T(n) = T(n-1) + b \qquad \forall n > 0$$

Die Lösung ist nicht schwierig, wenn man sich ein paar Beispiele anschaut (englisch “Unrolling a Recurrence Relation”):

$$
\begin{aligned}
T(0) &= a \\
T(1) &= T(0) + b = a + b \\
T(2) &= T(1) + b = a + b + b = a + 2b \\
T(3) &= T(2) + b = a + 2b + b = a + 3b \\
T(4) &= T(3) + b = a + 3b + b = a + 4b \\
&\dots \\
T(n) &= a + n\,b
\end{aligned}
$$

Also ist FACTORIAL(n) in $\mathcal{O}(n)$!

Bemerkung: Die Rekursion reduziert hier das Problem nur von n auf $n-1$, was nicht gut ist. Viel besser sind z. B. Rekursionen, die ein Problem halbieren:

$$
\begin{aligned}
T(0) &= a \\
T(n) &= T(n/2) + b
\end{aligned}
$$

Hierfür kann man zeigen, dass dann gilt: T ist in $\mathcal{O}(\lg n)$.

Graphisch kann man sich Rekursion und Laufzeit wie in den Abbildungen 3.2 und 3.3 dargestellt, verdeutlichen!

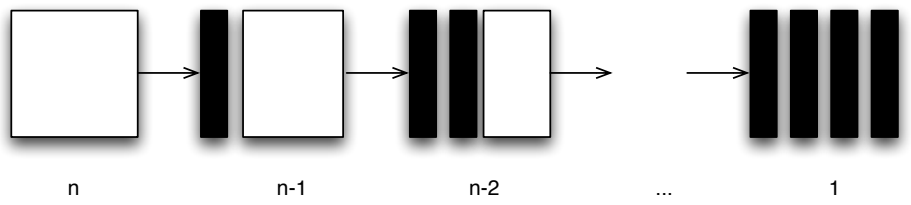


Abbildung 3.2: Big-O Rekursiv Linear

Interpretation

Bevor wir uns den genauen Definitionen der $\mathcal{O}$-Notation im Abschnitt 3.1.4 zuwenden, verfeinern wir zunächst die Anschauung noch einmal. Die $\mathcal{O}$-Notation gibt nicht exakt die Größenordnung an, sondern nur eine obere Schranke, die echte Größenordnung kann also kleiner sein. In der Praxis wird dennoch meist die $\mathcal{O}$-Notation verwendet. Die Angabe der exakten

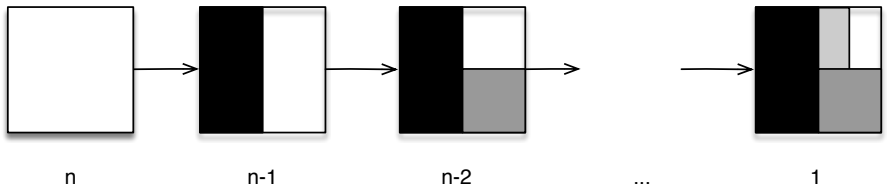


Abbildung 3.3: Big-O Rekursiv Logarithmisch

Größenordnung ist mittels der Θ-Notation möglich. Darauf greifen wir jetzt zurück, um exakter zu werden.

Die so definierten Komplexitätsklassen lassen sich interpretieren:

1. Ein Algorithmus $A(n)$ ist typischerweise in $\Theta(1)$, wenn er entweder gar nicht vom Input abhängt, oder die Datenstruktur so gegeben ist, dass man direkten (konstanten) Zugriff auf die fraglichen Elemente hat. Hier zeigt sich bereits, dass Datenstrukturen und Algorithmen eng zusammenhängen!
2. Ein Algorithmus $A(n)$ ist typischerweise in $\Theta(n)$, wenn er durch jedes Element einer Datenstruktur einmal (oder eine *konstante* Anzahl mal) durchgeht.
3. Ein Algorithmus $A(n)$ ist typischerweise in $\Theta(n^2)$, wenn er für jedes Element einer Datenstruktur etwas mit jedem anderen Element tut.
4. Ein Algorithmus $A(n)$ ist typischerweise in $\Theta(\log_a n)$, wenn er eine Datenstruktur durch Teilen rekursiv verkleinern kann, um seinen Job zu erledigen. Wenn er halbiert, ist er typischerweise in $\Theta(\lg n)$.
5. Ein Algorithmus $A(n)$ ist typischerweise in $\Theta(n!)$, wenn er alle Permutationen der Elemente ausprobieren muss.

Häufig vorkommende Beispiele sind in Tabelle 3.3 aufgeführt.

Tabelle 3.3: Beispiele

Klasse	Bedeutung	Anschauliche Erklärung	Beispiele für Laufzeiten
$f \in \Theta(1)$	f ist beschränkt	f überschreitet einen konstanten Wert nicht (ist also unabhängig vom Wert des Arguments).	Nachschlagen des n-ten Elementes in einem Feld
$f \in \Theta(\lg n)$	f wächst logarithmisch	f wächst ungefähr um einen konstanten Betrag, wenn sich das Argument verdoppelt.	Binary Search
$f \in \Theta(n)$	f wächst linear	f wächst ungefähr auf das Doppelte, wenn sich das Argument verdoppelt.	Suche im unsortierten Feld mit Einträgen (Bsp. lineare Suche)
$f \in \Theta(n \lg n)$	f hat superlineares Wachstum (aber kein quadratisches)	f wächst nur etwas schneller als linear	Fortgeschrittenere Sortier-Algorithmen (Mergesort, Heapsort)
$f \in \Theta(n^2)$	f wächst quadratisch	f wächst ungefähr auf das Vierfache, wenn sich das Argument verdoppelt.	Einfache Algorithmen zum Sortieren von Zahlen (INSERTION-SORT)
$f \in \Theta(2^n)$	f wächst exponentiell	f wächst ungefähr auf das Doppelte, wenn sich das Argument um eins erhöht	Brute Force Passwort Cracking
$f \in \Theta(n!)$	f wächst faktoriell	f wächst ungefähr um das $(n + 1)$-fache, wenn sich das Argument um eins erhöht.	Viele naive Brut-Force Algorithmen, Handlungsreisender

3.1.3 Warnungen

1. Man kann mit dem $\mathcal{O}(n)$-Kalkül nicht nur Laufzeiten, sondern auch andere interessante Parameter wie Speicherplatzbedarf abschätzen.
2. Außerdem muss man zwischen Best, Average und Worst Case unterscheiden. Hier finden Sie leider im Internet (und teilweise sogar in Büchern) falsche oder ungenaue Interpretationen.
3. In der Literatur wird Best, Average und Worst Case häufig gar nicht angegeben und Sie müssen die Bedeutung aus dem Kontext erschließen.
4. Weitere problemspezifische Aspekte werden in der Literatur manchmal nicht korrekt angegeben. Ein Beispiel ist beim Suchen, ob eine Suche mit offenem Such-Ergebnis oder die erfolgreiche bzw. erfolglose Suche betrachtet wird. Dies betrifft auch die Verteilung der zu suchenden Schlüssel wie das Verhältnis von vorhandenen zu nicht-vorhandenen Schlüsseln.

3.1.4 Definition

Mit diesem Abschnitt kehren wir wieder zurück zu den mathematisch *exakten* Definitionen und Aussagen.

Definition 3.1.2. *Sei $g\colon \mathbb{N} \to \mathbb{R}^+$ eine beliebige Funktion. Dann bezeichnet $\mathcal{O}(g(n))$ die Menge der folgenden Funktionen:*

$$\begin{aligned}\mathcal{O}(g(n)) := \{f(n) \quad | \quad & \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{so dass} \\ & 0 \leq f(n) \leq c\, g(n) \quad \forall n \geq n_0\}\end{aligned}$$

Man nennt $g(n)$ eine asymptotische obere Schranke *für $f(n)$. Häufig schreibt man auch einfach $\mathcal{O}(g)$ statt $\mathcal{O}(g(n))$.*

Beachten Sie, dass $\mathcal{O}$ Funktionen auf *Mengen von Funktionen* abbildet, denn $\mathcal{O}(g(n))$ bezeichnet für jedes g eine *Menge* von Funktionen.

Intuition: f wächst asymptotisch (d.h. für große n) und unter Vernachlässigung konstanter Faktoren höchstens so schnell wie g.

Definition 3.1.3. *Sei $g\colon \mathbb{N} \to \mathbb{R}^+$ eine beliebige Funktion. Dann bezeichnet $\Omega(g(n))$ die Menge der folgenden Funktionen:*

$$\begin{aligned}\Omega(g(n)) := \{f(n) \quad | \quad & \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{so dass} \\ & 0 \leq c\, g(n) \leq f(n) \quad \forall n \geq n_0\}\end{aligned}$$

Man nennt $g(n)$ eine asymptotische untere Schranke *für $f(n)$.*

Intuition: $f(n)$ ist asymptotisch und bis auf einen konstanten Faktor durch g nach unten beschränkt.

Definition 3.1.4. *Sei $g\colon \mathbb{N} \to \mathbb{R}^+$ eine beliebige Funktion. Dann bezeichnet $\Theta(g(n))$ die Menge der folgenden Funktionen:*

$$\Theta(g(n)) := \{f(n) \quad | \quad \exists\, c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{so dass} \\ 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n) \quad \forall n \geq n_0\}$$

Intuition: $f(n)$ ist asymptotisch und bis auf konstante Faktoren von derselben Größenordnung wie $g(n)$.

Lemma 3.1.3. $\Theta = \Omega \cap \mathcal{O}$.

Beweis. Übungsaufgabe! □

Definition 3.1.5. *Sei $g\colon \mathbb{N} \to \mathbb{R}^+$ eine beliebige Funktion. Dann bezeichnet $o(g(n))$ die Menge der folgenden Funktionen:*

$$o(g(n)) := \{f(n) \quad | \quad \forall c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{so dass} \\ 0 \leq f(n) < c\, g(n) \quad \forall n \geq n_0\}$$

Man nennt $g(n)$ eine asymptotisch scharfe *obere Schranke, $f(n)$ ist* asymptotisch kleiner *als $g(n)$.*

Intuition: $f(n)$ wächst asymptotisch streng langsamer als $g(n)$, das heißt $f(n)$ ist *asymptotisch vernachlässigbar.*

Lemma 3.1.4. *Es gilt:*

$$o(g) = \{f : \mathbb{N} \to \mathbb{R}^+, \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$$

Beweis. Übungsaufgabe! □

Definition 3.1.6. *Sei $g\colon \mathbb{N} \to \mathbb{R}^+$ eine beliebige Funktion. Dann bezeichnet $\omega(g(n))$ die Menge der folgenden Funktionen:*

$$\omega(g(n)) := \{f(n) \quad | \quad \forall c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{N}, \text{so dass} \\ 0 \leq c\, g(n) < f(n) \quad \forall n \geq n_0\}$$

Man nennt $g(n)$ eine asymptotisch scharfe *untere Schranke, $f(n)$ ist* asymptotisch größer *als $g(n)$.*

Intuition: $f(n)$ wächst asymptotisch streng schneller als $g(n)$, das heißt $f(n)$ ist *asymptotisch dominant.*

Beachten Sie, dass aus der Definition sofort für ein beliebiges g folgt: $\omega(g) \subseteq \Omega(g)$ und $o(g) \subseteq \mathcal{O}(g)$. Man kann sich außerdem leicht überlegen, dass echte Teilmengen vorliegen, da $\mathcal{O}$ und Ω reflexiv sind und ω und o nicht (s.u.), also g immer in $\mathcal{O}(g)$ und $\Omega(g)$ enthalten ist, aber nicht in $\omega(g)$ und $o(g)$.

Lemma 3.1.5. *Es gilt:*

$$\omega(g) = \{f : \mathbb{N} \to \mathbb{R}^+, \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\}$$

Beweis. Übungsaufgabe! □

Die Abbildung 3.4 veranschaulicht die Notation.

Zum Ende diese Abschnittes verlassen wir bezüglich der $\mathcal{O}$-Notation wieder die mathematisch exakten Definitionen. Zur Vereinfachung werden die in diesem Abschnitt definierten Mengen in der Literatur oft wie Funktionen behandelt. Wir verwenden die folgenden Schreibweisen synonym.

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) \quad &\text{und} \quad f(n) \in \mathcal{O}(g(n)) \\ \mathcal{O}(f(n)) = \mathcal{O}(g(n)) \quad &\text{und} \quad \mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n)) \end{aligned}$$

Somit kann man zum Beispiel auch schreiben:

$$f(n) = g(n) + \mathcal{O}(1) \quad \text{statt} \quad f(n) - g(n) \in \mathcal{O}(1)\ .$$

Offensichtlich müssen die Gleichungen in diesem Fall aber von links nach rechts gelesen werden und dürfen nicht umgeformt werden.

3.1.5 Rechenregeln

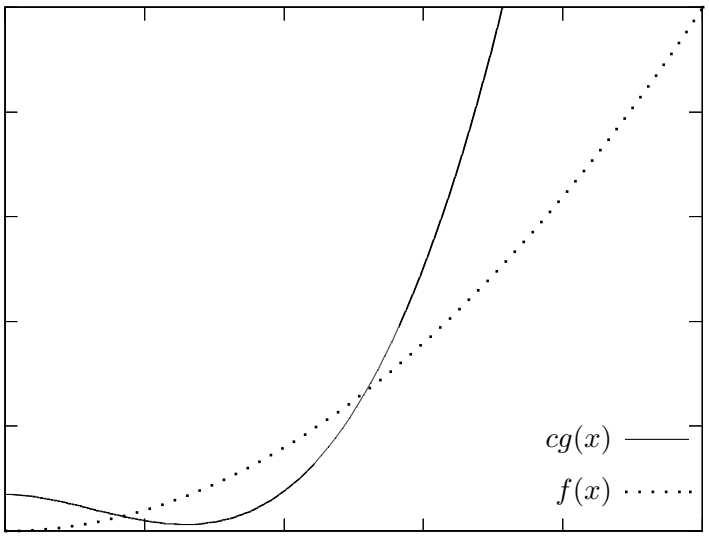
Lemma 3.1.6. *Die oben definierten Mengen haben die folgenden Eigenschaften:*

Transitivität:

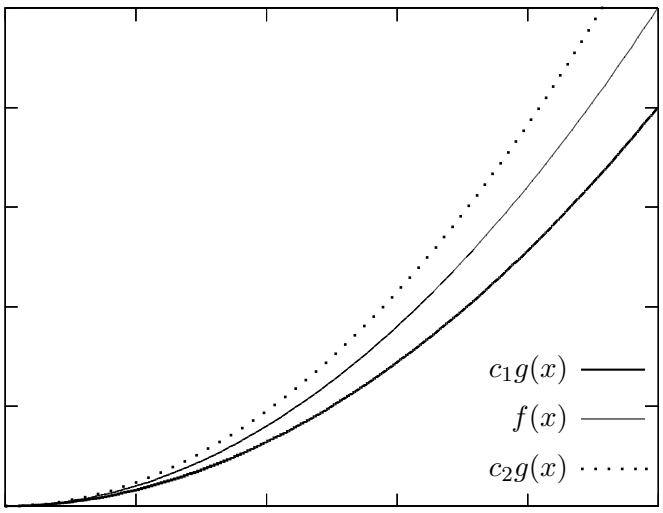
$$\begin{aligned} &\text{Aus } f(n) = \Theta(g(n)) \text{ und } g(n) = \Theta(h(n)) \text{ folgt } f(n) = \Theta(h(n)) \\ &\text{aus } f(n) = \mathcal{O}(g(n)) \text{ und } g(n) = \mathcal{O}(h(n)) \text{ folgt } f(n) = \mathcal{O}(h(n)) \\ &\text{aus } f(n) = \Omega(g(n)) \text{ und } g(n) = \Omega(h(n)) \text{ folgt } f(n) = \Omega(h(n)) \\ &\text{aus } f(n) = o(g(n)) \text{ und } g(n) = o(h(n)) \text{ folgt } f(n) = o(h(n)) \\ &\text{aus } f(n) = \omega(g(n)) \text{ und } g(n) = \omega(h(n)) \text{ folgt } f(n) = \omega(h(n)) \end{aligned}$$

Reflexivität:

$$\begin{aligned} f(n) &= \Theta(f(n)) \\ f(n) &= \mathcal{O}(f(n)) \\ f(n) &= \Omega(f(n)) \end{aligned}$$



(a) $f(n) = \mathcal{O}(g(n))$



(b) $f(n) = \Theta(g(n))$

Abbildung 3.4: Growth

Symmetrie:

$$f(n) = \Theta(g(n)) \quad \Leftrightarrow \quad g(n) = \Theta(f(n))$$

Transpositionssymmetrie:

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) \quad &\Leftrightarrow \quad g(n) = \Omega(f(n)) \\ f(n) = o(g(n)) \quad &\Leftrightarrow \quad g(n) = \omega(f(n)) \end{aligned}$$

Beweis. Übungsaufgabe! □

Daher kann man die folgende Analogie aufstellen:

$$\begin{aligned} f = \Theta(g) \quad &\widehat{=} \quad f = g \\ f = \mathcal{O}(g) \quad &\widehat{=} \quad f \leq g \\ f = \Omega(g) \quad &\widehat{=} \quad f \geq g \\ f = o(g) \quad &\widehat{=} \quad f < g \\ f = \omega(g) \quad &\widehat{=} \quad f > g \end{aligned}$$

Die Vergleichsoperatoren beziehen sich hierbei auf das Wachstumsverhalten.

Aufgrund dieser Eigenschaften kann man Funktionen gemäß ihrem Wachstumsverhalten in sogenannte *Äquivalenzklassen* einteilen. Somit sind etwa n^2 und $3n^2 + 4n + 7$ in der selben Äquivalenzklasse und unterscheiden sich *nicht* hinsichtlich ihres Wachstumsverhaltens. Man kann dabei eine der beiden Funktionen (in der Regel die „einfachere“) stellvertretend für alle aus der Klasse wählen.

Für das Rechnen mit der $\mathcal{O}(n)$ Notation gelten zudem einfache Rechenregeln:

Lemma 3.1.7. *Es gelten die folgenden Rechenregeln:*

1. *Konstante Faktoren weglassen:* $g(n) \in \mathcal{O}(d\, f(n)), d \in \mathbb{R}^+$, *dann gilt* $g(n) \in \mathcal{O}(f(n))$.
2. *Multiplikationsregel: Aus* $g_1(n) \in \mathcal{O}(f_1(n))$ *und* $g_2(n) \in \mathcal{O}(f_2(n))$ *folgt* $g_1(n)g_2(n) \in \mathcal{O}(f_1(n)f_2(n))$.
3. *Linearität: Aus* $g_1(n) \in \mathcal{O}(f_1(n))$ *und* $g_2(n) \in \mathcal{O}(f_2(n))$ *folgt* $g_1(n) + g_2(n) \in \mathcal{O}(f_1(n) + f_2(n)) = \mathcal{O}(\max(f_1(n), f_2(n))$ *(letzte Gleichung gilt, falls* max *definiert ist).*
4. *Weglassen von Termen „kleinerer Ordnung“: Sei* $f(n) > 0$ *für* $n > n_0$, $|g(n)| \in o(f(n))$ *und* $h(n) = f(n) + g(n)$ *dann folgt* $h(n) \in \mathcal{O}(f(n))$.

Analoge Aussagen gelten für Ω, ω und Θ. Für unendliche Summen kann die Linearitätsregel verallgemeinert werden, vorausgesetzt, dass die Terme uniform *beschränkt sind.*

Beweis. Übung! □

Nützlich ist auch das folgende Lemma für Grenzwerte:

Lemma 3.1.8. *Falls $f(n) > 0$ für $n > n_0$ und*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

für ein $c \in \mathbb{R}^+$, dann gilt $f(n) \in \mathcal{O}(g(n))$.

Beweis. Übung! □

Für Polynome gilt aus dem bisher gezeigten:

Lemma 3.1.9. *Sei f ein Polynom, d.h.*

$$f(n) = \sum_{i=0}^{k} a_i n^i = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0,$$

dann gilt falls $a_k > 0$

$$f \in \Theta(n^k).$$

Beweis. Übung! □

Lemma 3.1.10. *Es gilt die folgende Hierarchie von Funktionen (sollte man als Informatiker „intuitiv“ kennen!) (für genügend große n):*

$$\begin{aligned} \log n < \sqrt{n} < \frac{n}{\log n} &< n < n \log n < n(\log n)^2 \\ &< n^2 < n^3 < \ldots \\ &< 2^n < e^n < 3^n < \ldots \\ &< n! \\ &< n^n \end{aligned}$$

Das Symbol log *steht dabei hier für einen Logarithmus mit beliebiger Basis. Wegen der Bedeutung des Logarithmus zur Basis Zwei in der Informatik verwenden wir im folgenden $\lg n := \log_2 n$ (und nicht etwa zur Basis 10!).*

Beweis. Übung! □

3.2 Rekursionsgleichungen

Rekursionsgleichungen der folgenden Form sind für die Laufzeit- und Speicheranalyse von Algorithmen sehr wichtig:

$$T(n) = \begin{cases} \Omega(1), & \text{wenn } n = 1 \\ f(T(g(n))) + \Omega(g(n)), & \text{wenn } n > 1,\ g(n) < n \text{ und } f : \mathbb{N} \to \mathbb{R} \end{cases}$$

Zur Erinnerung: Für INSERTION-SORT hatten wir $T(n) = an^2 + bn + c$ explizit ausgerechnet. Wir können dies jedoch auch mit Hilfe von Rekursionsgleichungen ausrechnen. Eine rekursive Variante von INSERTION-SORT ist im Algorithmus 3.10 angegeben[2].

Algorithmus 3.10 INSERTION-SORT-REC-P(A)

Listing 3.1 Rekursive Version des Sortierens durch Einfügen (Ausführbarer Pseudocode)

```
1  def insertsortrec(j, a):
2      if j >= len(a):
3          return a
4      else:
5          key = a[j]
6          i = j-1
7          while i >= 0 and a[i] > key:
8              a[i+1] = a[i]
9              a[i] = key
10             i = i-1
11         return insertsortrec(j+1, a)
```

Wenn wir die Rekursion über die Variable j betrachten, gilt für diese:

$$T(j) = \begin{cases} B\,n + D, & \text{wenn } j = n \\ T(j+1) + B(j-1) + C, & \text{wenn } 1 \leq j < n, \end{cases}$$

wie man unmittelbar aus dem Pseudocode ablesen kann. Mit der Substitution $T'(i) := T(n-i) - B\,n$ erhält man daraus eine Rekursionsgleichung für die Laufzeit von T' in Abhängigkeit von $n := len(a)$:

$$T'(i) = \begin{cases} D, & \text{wenn } i = 0 \\ T'(i-1) + B(i-1) + C, & \text{wenn } 0 < i \leq n. \end{cases}$$

[2]Dabei wurde in Zeile 8 und 9 ein echter Dreieckstausch vorgenommen, d.h. im Gegensatz zum Pseudocode des Algorithmus 2.3 keine Optimierung vorgenommen.

Wenn Sie hier Verständnisprobleme haben, überlegen Sie sich, dass j eine aufsteigende Rekursionstiefe und gleichzeitig höhere Aufwände mit der while-Schleife bedeutet. M.a.W. der Arbeitsanteil der while-Schleife steigt mit Rekursionstiefe an.

Lemma 3.2.1. *Eine Lösung dieser Rekursiongleichung hat die Form* $T'(n) = \frac{a}{2}n^2 + an + \frac{3}{2}a$ *(mit* $a \in \mathbb{R}^+$*). Das bedeutet, dass* $T'(n) \in \Theta(n^2)$*. Und somit ist auch* $T \in \Theta(n^2)$*.*

Beweis. Mit Induktion über n. (Übung!) □

3.2.1 Substitutionsmethode

Das Lösen von Rekursionsgleichungen mit Hilfe von Raten, Einsetzen und Induktion nennt man auch *Substitutionsmethode*.

3.2.2 Variablenwechsel

Manchmal kann durch eine geänderte Wahl von Variablen ein Ausdruck vereinfacht und gelöst werden. Beispiel:

$$T(n) = 2T(\sqrt{n}) + \lg n$$

Durch Substitution $n \to n' := \lg n$ und $T'(n') := T(2^{n'})$ erhält man

$$T'(n') = 2T'(n'/2) + n'$$

mit Lösung $T'(n') \in \mathcal{O}(n' \lg n')$ und damit

$$T(n) = T(2^{n'}) = T'(n') \in \mathcal{O}(n' \lg n') = \mathcal{O}(\lg n \lg \lg n).$$

3.2.3 Mastertheorem

Eine weitere Methode Rekursionsgleichungen zu lösen, ist das Mastertheorem [CLR04] für Rekursionsgleichungen der Form $T(n) = aT(n/b)+f(n)$:

Theorem 3.2.1. *Seien* $a \geq 1$ *und* $b > 1$ *Konstanten, sei* $f(n)$ *eine Funktion* $\mathbb{N} \to \mathbb{R}$ *und für* $T(n)$ *gelte die Rekursionsgleichung* $T(n) = aT(n/b) + f(n)$*, wobei* n/b *als* $\lfloor n/b \rfloor$ *oder* $\lceil n/b \rceil$ *interpretiert wird. Dann gilt mit* $\gamma := \log_b a$*:*

1. *Wenn* $f(n) = \mathcal{O}(n^{\gamma-\epsilon})$ *für ein* $\epsilon > 0$*, dann gilt* $T(n) \in \Theta(n^\gamma)$
2. *Wenn* $f(n) = \Theta(n^\gamma)$*, dann gilt* $T(n) \in \Theta(n^\gamma \lg n)$
3. *Wenn* $f(n) = \Omega(n^{\gamma+\epsilon})$ *für ein* $\epsilon > 0$ *und* $af(n/b) \leq cf(n)$ *für* $c < 1$ *und* n *genügend groß, dann gilt* $T(n) \in \Theta(f(n))$

Beweis. Siehe [CLR04] □

3.2.4 Baummethode

Rekursionsgleichungen können auch mit Hilfe der *Baummethode* gelöst werden. Letztere werden wir z. B. für den MergeSort-Algorithmus noch anwenden, siehe Seite 91.

3.2.5 Erzeugendenfunktionen

Rekursionsgleichungen können auch durch den Einsatz von *Erzeugendenfunktionen* gelöst werden. Mit dieser mächtigen Technik können darüber hinaus viele Abzählprobleme gelöst werden, die bei der Analyse von Algorithmen und Datenstrukturen auftreten. Eine Erzeugendenfunktion ist formale Potenzreihe, deren Koeffizienten die abzuzählenden Werte sind.

Eine Vertiefung würde den Rahmen des Skriptes deutlich sprengen. Eine kurze Einführung und ein Anwendungs-Beispiel ist in [Neb12] gegeben. Dort wird gezeigt, wie aus der doch sehr einfachen Rekursionsgleichung für die Fibonacci-Zahlen (Definition in Kapitel 4.1) die folgende überraschende Formel für die n-te Fibonacci-Zahl berechnet wird:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) .$$

Überraschend ist diese Formel deswegen, weil für jede natürliche Zahl n auch der Wert von F_n immer eine natürliche Zahl ist. Eine schöne Übungsaufgabe ist der Beweis dieser Formel durch vollständige Induktion.

3.3 Komplexitätsmaße und -klassen

Bei der Analyse von Algorithmen kann man unterschiedliche Kriterien betrachten:

1. Laufzeit
2. Speicherverbrauch

Bei der Laufzeit und dem Speicherverbrauch wiederum kann man unterscheiden zwischen den folgenden Fällen:

1. Schlechtester Fall (Worst Case)
2. Bester Fall (Best Case)
3. Durchschnittlicher Fall (Average Case)

Während Bester und Schlechtester Fall die Laufzeit des Algorithmus unter besten bzw. schlechtesten Randbedingungen (in der Regel Funktion der Eingangsdaten) angeben, stellt der durchschnittliche Fall eine probabilistische Betrachtung dar und kann nur mit Begriffen der Wahrscheinlichkeitstheorie

analysiert und verstanden werden (s.u.). Die Bedeutung des Worst Case liegt darin, dass dieser eine *obere* Schranke darstellt. Mit anderen Worten wenn ein Algorithmus im Worst Case z. B. ein $\mathcal{O}(n^2)$ Verhalten hat, dann ist dies für *jeden* Durchlauf unabhängig von der Wahl der Eingabeparameter *garantiert*. Außerdem tritt der Worst Case bei vielen Algorithmen ziemlich oft auf. Der Average Case ist für viele Algorithmen genauso schlecht wie der Worst Case (zumindest asymptotisch). Für einige, wie z. B. Quick-Sort ist der Average Case jedoch viel besser und da der Worst Case hier nur sehr selten auftritt, ist eine Average Case-Analyse sinnvoll. Die Best Case-Analyse ist in der Regel ohne Bedeutung.

Meistens werden wir im folgenden also Worst- und/oder Average Case Angaben machen, und zwar in $\mathcal{O}(n)$-Notation. Die Bedeutung von $\mathcal{O}(n)$ liegt darin, dass sie es ermöglicht, Algorithmenklassen zu vergleichen und dass sie von der konkreten Implementierung und dem konkreten Computer abstrahiert. Das $\mathcal{O}(n)$-Konzept sorgt also dafür, dass unsere Analysen *unabhängig* von der verwendeten *Hardware* sind! Interessant sind nicht unwesentliche Konstanten und einzelne Koeffizienten, sondern das asymptotische Verhalten bei großen n. Man nennt diese Klassen auch *Komplexitätsklassen*.

3.4 Wachstum

Unter der Annahme, dass ein Rechner 10^{10} Operationen pro Sekunde ausführt[3], stellt Tabelle 3.4 die Zeitschranken für Algorithmen verschiedener Komplexitätsklassen dar – die Grafik 3.5[4] veranschaulicht das Wachstumsverhalten.

Tabelle 3.4: Zeitschranken für die Rechenzeit

Zeitschranke	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$f(n) = n$	1.0×10^{-09}	2.0×10^{-09}	3.0×10^{-09}	4.0×10^{-09}	5.0×10^{-09}
$f(n) = n \log n$	2.3×10^{-09}	6.0×10^{-09}	1.0×10^{-08}	1.5×10^{-08}	2.0×10^{-08}
$f(n) = n^2$	1.0×10^{-08}	4.0×10^{-08}	9.0×10^{-08}	1.6×10^{-07}	2.5×10^{-07}
$f(n) = 2^{\sqrt{n}}$	9.0×10^{-10}	2.2×10^{-09}	4.5×10^{-09}	8.0×10^{-09}	1.3×10^{-08}
$f(n) = 2^n$	1.0×10^{-07}	1.0×10^{-04}	1.1×10^{-01}	1.1×10^{02}	1.1×10^{05}
$f(n) = n!$	3.6×10^{-04}	2.4×10^{08}	2.7×10^{22}	8.2×10^{37}	3.0×10^{54}

$2^{\sqrt{n}}$ Zur Verdeutlichung: $2.4 \times 10^{08} \approx 182$ Jahre, $3.0 \times 10^{54} \approx 2 \times 10^{48}$ Jahre, was um ein Vielfaches höher als das Alter des Universums ist! Bereits anhand dieses einfachen Beispiels wird deutlich, wie wichtig gute Algorithmen sind und dass in vielen Fällen ein schnellerer Rechner *gar nichts* nützt!

[3] Das entspricht 10 GHz, d.h es wird eine Operation pro Takt ausgeführt.

[4] Man beachte die unterschiedlichen Skalierungen!

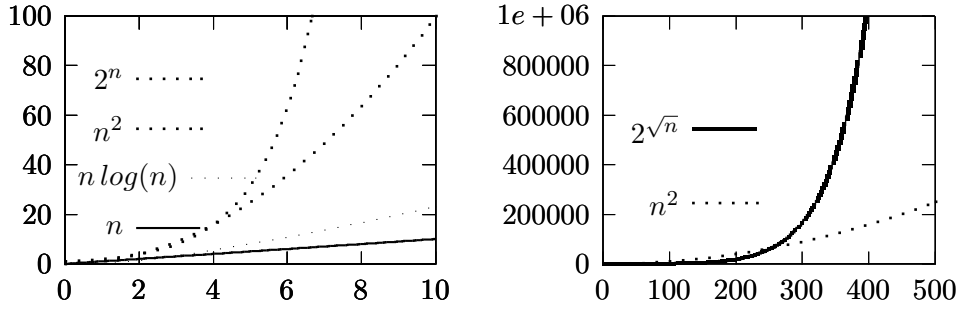


Abbildung 3.5: Growth

3.5 Probabilistische Betrachtung

Bei einer probabilistischen Betrachtung betrachten wir z. B. die Laufzeit T_A eines Algorithmus $A : X_i \to A(X_i)$ als Funktion der Eingangsparameter (Input) X_i, wobei diese als Zufallsvariablen aufgefasst werden. Zum Beispiel kann ein Sortieralgorithmus A als Funktion von $A : \mathbb{N}^n \to \mathbb{N}^n$ aufgefasst werden:

$$A : (x_1, \ldots, x_n) \to (x_{\pi_1}, \ldots, x_{\pi_n}),$$

wobei $\pi_i(n)$ eine Permutation bezeichnet. Wenn wir annehmen, dass die Eingabewerte einer bestimmten Verteilung gehorchen, so kann man die Laufzeit von A als Zufallsvariable analysieren und etwa den Erwartungswert ausrechnen. Wir werden dies bei einigen Algorithmen im folgenden tun, allerdings nehmen wir aus Gründen der Einfachheit dabei stets eine Gleichverteilung an.

3.6 P und NP

Eine für die praktische und theoretische Informatik wichtige Komplexitätsklasse stellen die *polynomialen* Algorithmen dar. Probleme, die in polynomialer Zeit lösbar sind, liegen in der Komplexitätsklasse **P** („P“ für Polynomial), Probleme, für die man potentiell (richtige oder falsche) Lösungen in polynomialer Zeit *verifizieren*, aber nicht notwendiger Weise Lösungen finden kann, liegen in der Komplexitätsklasse **NP** („NP“ für Nondeterministic[5] Polynomial). Es ist offensichtlich, dass gilt $\mathbf{P} \subseteq \mathbf{NP}$ (wieso? Übung!). Probleme in **P** sind also in $\mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$.

Das wichtigste ungelöste Problem lautet:

$$\mathbf{P} = \mathbf{NP}?$$

[5]Der Begriff “Nondeterministic” bezieht sich auf sogenannte nicht-deterministische Turing Maschinen – deren Erklärung hier zu weit führen würde.

Fast alle Informatiker gehen davon aus, dass **P** eine *echte* Teilmenge von **NP** darstellt, dass also gilt

$$\mathbf{P} \neq \mathbf{NP}$$

Das liegt u.a. auch an der Existenz von so genannten **NP**-vollständigen Problemen, s. [CLR04].

Probleme in **P** haben auch deshalb eine große Bedeutung, weil das k in $\mathcal{O}(n^k)$ meist recht klein ist und sich diese Probleme praktisch auch für große n gut lösen lassen, während Probleme, die nicht in **P** sind und z. B. sich nur in exponentieller Zeit lösen lassen, also Probleme in $\mathcal{O}(e^n)$ bereits für kleine n praktisch unlösbar werden.

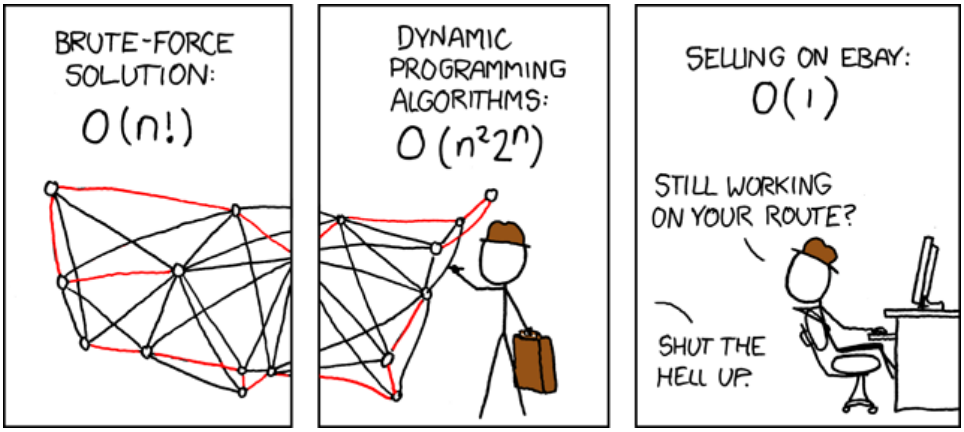


Abbildung 3.6: Erkennen von NP Problemen, Quelle: http://xkcd.com/

Es ist für jeden Informatiker in der Praxis wichtig, Probleme die *nicht* in **P** sind zu erkennen, damit man nicht versucht, eine Lösung (s. Abbildung 3.7) zu finden, die es gar nicht geben kann! Ein Beispiel für ein solches Problem sind die Türme von Hanoi, die wir in Abschnitt 9.1.3 kennenlernen werden. Einige NP-vollständige Probleme sind in [Kar72] zu finden.

3.7 Unlösbare Probleme

Es gibt auch Probleme, für die es *überhaupt* keine Lösung geben kann. Zu diesen gehört z. B. das berühmte *Halteproblem*. Gesucht ist ein Programm **PROVER**(P, x), das

1. als Eingabe ein *beliebiges* Programm **P** zusammen mit seinem Input x einliest,
2. das Programm in endlicher Zeit analysiert und

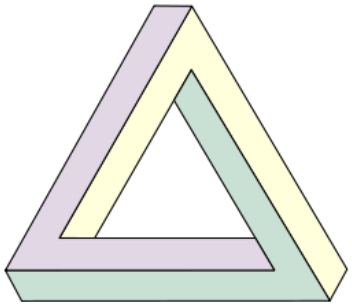


Abbildung 3.7: Mission Impossible, Quelle: `http://en.wikipedia.org/wiki/File:Penrose_triangle.svg`

3. als Ausgabe **true** zurückgibt, wenn das Programm **P** mit Input x nach endlicher Zeit anhält, oder **false**, falls das nicht der Fall ist.

Gesucht ist also ein universelles Testprogramm, der Traum jedes Programmierers (und Managers)! Der nächste Satz zeigt, dass es so ein Programm nicht geben kann.

Theorem 3.7.1. *Es gibt kein Programm **PROVER**, das Halteproblem kann nicht gelöst werden.*

Beweis. Angenommen **PROVER** existiert, dann definieren wir den nachstehend gezeigten Algorithmus CANTOR 3.11. CANTOR(P) hält also genau

Algorithmus 3.11 CANTOR(P)

```
1: if PROVER(P, P) then          // P hält mit Input P
2:     while true do             // CANTOR hält nicht
3:         do nothing
4: else                          // P hält nicht mit Input P
5:     return                    // CANTOR hält
```

dann, wenn **PROVER**(P, P) *falsch* ist – wenn **PROVER**(P, P) dagegen *wahr* ist, gerät CANTOR in eine unendliche Schleife. Nun betrachten wir CANTOR(CANTOR). Hält CANTOR(CANTOR) an?

1. Annahme CANTOR(CANTOR) hält an:
   Dann muss PROVER(CANTOR, CANTOR) **false** sein, dies bedeutet aber, dass CANTOR *nicht* anhält, Widerspruch!
2. Annahme CANTOR(CANTOR) hält nicht an:
   Dann muss PROVER(CANTOR, CANTOR) **true** sein, dies bedeutet aber, dass CANTOR *anhält*, Widerspruch!

Also muss die Annahme, dass **PROVER** existiert, falsch sein (Tertium non datur). □

Der Beweis benutzt den berühmten Diagonalisierungstrick von Cantor und ist philosophisch von großer Bedeutung – der ursprüngliche Beweis geht auf die bahnbrechenden Arbeiten von Alan Turing (1936) zurück.

Bemerkung: Häufig wird obiger Satz so interpretiert, dass Beweise, dass Programme korrekt sind, unmöglich seien. Das ist völlig falsch! Obiger Satz sagt lediglich, dass es kein allgemeines automatisiertes Verfahren für beliebige Programme geben kann. Für ein konkretes Programm können schlaue Informatiker häufig einen Beweis führen!

Bemerkung: Der berühmte Satz von Rice aus der theoretischen Informatik besagt sogar, dass es generell unmöglich ist, eine nicht-triviale Eigenschaft eines Algorithmus algorithmisch zu entscheiden. Er wird ganz ähnlich bewiesen wie der vorherige Satz (ebenfalls mit einem Diagonalisierungstrick).

3.8 Fragen und Aufgaben zum Selbststudium

1. Betrachten Sie Algorithmus 3.6: SCHLEIFE2. Finden Sie einen anderen Algorithmus, der den gleichen Wert wie SCHLEIFE2 berechnet, aber in $\mathcal{O}(1)$ läuft. Hinweis: Vereinfachen Sie dazu unter Zuhilfenahme von $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ und $\sum_{i=0}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}$ den von SCHLEIFE2 berechneten Ausdruck $\sum_{j=0}^{n-1} \sum_{i=0}^{j} i$.
2. Gegeben sei der folgende Algorithmus:

Algorithmus 3.12 REVERSE(A)

```
1: for i = 0 to ⌊(length(A)−2)/2⌋ do
2:     A[i], A[length(A) − i − 1] ← A[length(A) − i − 1], A[i]
```

   - Sie haben durch Laufzeitanalysen ermittelt, dass der Algorithmus 3.12 in $\mathcal{O}(n^2)$ läuft. Können Sie sich das erklären?
   - Wie können Sie diesen Algorithmus – gegebenenfalls unter Zuhilfenahme einer Datenstruktur – verbessern, so dass er in $\mathcal{O}(n)$ läuft.
   - Sie müssen für ein Programmsystem sehr oft Strings umdrehen, und eine Laufzeit in $\mathcal{O}(n)$ ist nicht akzeptabel. Wie können Sie diesen Algorithmus – gegebenenfalls unter Zuhilfenahme einer Datenstruktur – verbessern, so dass er in $o(n)$ läuft. Welche Laufzeit besitzt Ihr Algorithmus in Θ-Notation?

3. Zeigen Sie $\mathcal{O}(g(n)) \cap \omega(g(n)) = \emptyset$, d.h. die Schnittmenge aus $\mathcal{O}$ und ω ist leer!

4. f und g seien asymptotisch positiv, d.h. $\exists n_0$ so dass $\forall n > n_0$ $f(n)$ und $g(n)$ positiv sind. Zeigen Sie, dass $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$ gilt!

5. Beweisen Sie, dass für beliebiges $a \in \mathbb{R}$ und $b \in \mathbb{R}^+$ für alle $n \in \mathbb{N}$ $(n + a)^b \in \mathcal{O}(n^b)$ gilt!

6. Gilt $2^{n+1} \in \mathcal{O}(2^n)$?

7. Ist $2^{2n} \in \mathcal{O}(2^n)$?

8. Füllen Sie die folgende Tabelle aus, so dass Sie entweder $\in$ oder $\notin$ die leeren Felder eintragen.

	$o(n^2)$	$\mathcal{O}(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	$\omega(n^2)$
$17n\log(n) - 12n + 4$					
$4n^2 - 9n + 7\sqrt{n} - 25$					
$8n^2\sqrt{n} - 2n\log(n) + 6$					

   Sie können zur Vereinfachung die folgenden Vorüberlegungen anstellen.

   - Vergegenwärtigen Sie sich zunächst nochmals, was die Mengen o, $\mathcal{O}$, Θ, Ω und ω beinhalten.
   - Identifizieren Sie dann die relevanten Terme der Ausdrücke in der ersten Spalte. Hinweis: Es sind immer die Anteile mit dem asymptotisch größten Wachstum.

   An welchen Stellen kann Ihnen $\Theta = \Omega \cap \mathcal{O}$ aus Lemma 3.1.3 helfen?

9. Beantworten Sie die folgenden Fragen[6]

   - Ist $\lceil \lg n \rceil !$ polynomial beschränkt?
   - Ist $\lceil \lg \lg n \rceil !$ polynomial beschränkt?

10. Ist die folgende Aussage „Für beliebiges f gilt $f(n) \in \Theta(f(n/2))$.“ wahr? Falls ja, Beweis, falls nein, Gegenbeispiel! Falls nein, gibt es überhaupt Funktionen f, für die die Aussage gilt? (Hinweis: Betrachten Sie Funktionen wie in Lemma 3.1.10.)

11. Beweisen Sie Lemma 3.2.1.!

12. Zeigen Sie mithilfe des Mastertheorems, dass die folgende Rekursionsgleichung $T(n) = 2\,T(n/2) + cn$ die Lösung $T(n) = \Theta(n \lg n)$ hat!

[6] Benutzen Sie das folgende Lemma (Beweis?!): Eine Funktion f ist genau dann polynomial beschränkt, wenn gilt $\lg(f(n)) \in \mathcal{O}(\lg n)$.

13. Gegeben sei der Algorithmus 3.13:

Algorithmus 3.13 $Q(A)$

```
1:  n ← length(A)
2:  if n = 1 then
3:      return A[0]
4:  else
5:      r ← 0
6:      for i = 0 to ⌊n · lg(n)⌋ do
7:          r ← r + A[i mod n]
8:      for i = 0 to 3 do
9:          Q(A[⌊i·n/6⌋..⌊i·n/6 + (n−2)/2⌋])
10:     return r
```

- Begründen Sie, dass die Laufzeit $T(n)$ von $Q(A)$ sich wie folgt beschreiben lässt: $T(n) = 4\,T(n/2) + n \lg n$.
- Zeigen Sie mithilfe des Mastertheorems, dass diese Rekursionsgleichung die Lösung $T(n) = \Theta(n^2)$ besitzt.

Kapitel 4

Darstellungen von Algorithmen

Wir verwenden in dieser Vorlesung unterschiedliche Darstellungsmethoden für Algorithmen. Zunächst einmal kommen *textuelle* Darstellungsformen in Frage. In der Literatur wird – auch aus historischen Gründen – häufig Pseudocode verwendet, weshalb wir auch in diesem Skript Pseudocode-Notation einführen. Im Gegensatz zur Situation in den 50er und 60er Jahren des letzten Jahrhunderts existieren heute mächtige Hochsprachen, vor allem Skriptsprachen, mit deren Hilfe es möglich ist, anstelle von Pseudocode direkt ausführbaren Code zu verwenden. Wir werden daher auch von dieser Möglichkeit Gebrauch machen. Darüber hinaus lassen sich die meisten Algorithmen mit verhältnismäßig wenig Aufwand auch in C implementieren und da diese Programmiersprache einer der ersten des Curriculums ist, werden wir Algorithmen auch direkt in C definieren. Eine weitere, exzellente Form besteht in der Verwendung von funktionalen Programmiersprachen, für die Algorithmen zum Teil auch anders entworfen werden (müssen bzw. können). Leider können wir aus Zeitgründen hier davon keinen Gebrauch machen.

Wichtig ist, dass Sie sich im Laufe der Zeit eine gewisse Fertigkeit erwerben, den abstrakten Algorithmus hinter den Darstellungen zu „lesen“!

Neben den textuellen Darstellungsformen gibt es auch *graphische* Formen wie Struktogramme, Datenflussdiagramme und Programmablaufpläne. Auch davon werden wir in dieser Vorlesung Gebrauch machen.

4.1 Pseudocode

Pseudocode ist strukturiertes Deutsch oder Englisch. Neben Konstrukten, die Sie aus der Ablaufsteuerung von Programmiersprachen wie C kennen, ist es erlaubt, in Pseudocode auch „normales“ Deutsch oder Englisch zu verwenden, um einen Algorithmus zu verdeutlichen. Wir verwenden die folgenden Konventionen, die sich an [CLR04] anlehnen:

1. Blockstrukturen werden durch Einrücken verdeutlicht (im Gegensatz zu dem in den Pascalsprachfamilien üblichen **Begin** und **End** Konstruktionen).
2. Die Schleifenkonstrukte **while**, **for** und **repeat-until** haben ähnliche Semantik wie in C oder Java.
3. Als Kontrollstruktur wird **if-else** verwendet mit ähnlicher Semantik wie in C oder Java.
4. Kommentare werden durch `//` (wie in C) gekennzeichnet.
5. Mehrwertige Zuordnungen wie `i,j = 1,2` sind zulässig. Die Operationen werden quasi parallel ausgeführt, somit können zum Beispiel mittels `a,b = b,a` die Werte vertauscht werden.
6. Variablen sind lokal, es sei denn explizit anders gekennzeichnet.
7. Feldelemente eines Feldes (Arrays) A werden mit $A[i]$ gekennzeichnet.
8. Felder beginnen bei 0 und enden bei $n-1$, wenn n die Länge bezeichnet[1].
9. $A[0, \ldots, n-1]$ bezeichnet das gesamte Feld.
10. Parameterwerte werden “per Value” übergeben.
11. Rückgabewerte per **return** können mehrwertig sein.

Achtung: Insbesondere unterschiedliche Konventionen hinsichtlich des Startindexes eines Feldes können zu frustrierenden Fehlern führen (s. Abbildung 4.1). Meistens wird bei Pseudocodedarstellungen davon ausgegangen, dass Felder mit 1 starten. Da die meisten Programmiersprachen Felder bei 0 starten lassen, haben wir hier die Pseudocodedarstellung angepasst, damit Sie es leichter haben, den Pseudocode zu verstehen bzw. zu implementieren.

Ein Beispiel für (ineffizienten) Pseudocode zum Berechnen der sogenannten

[1] Die – wenigen – Ausnahmen, bei denen wir aus verschiedenen Gründen bei 1 beginnen zu zählen und bei n aufhören, sind *explizit* gekennzeichnet.

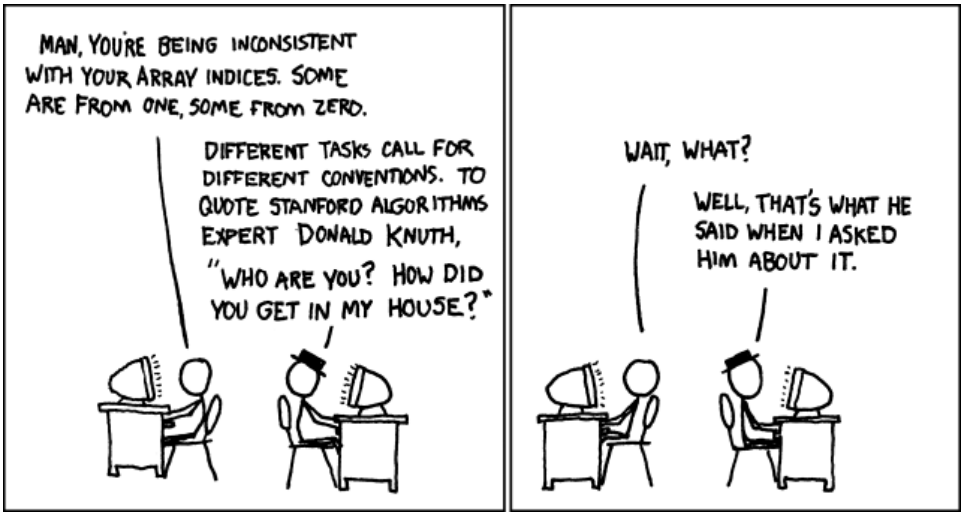


Abbildung 4.1: Indexkonventionen, Quelle: http://xkcd.com

Fibonacci-Zahlen F_n definiert wie folgt

$$F_n := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n > 1 \end{cases}$$

gibt der Algorithmus 4.1.

Algorithmus 4.1 FIB(n)

```
1: if n ≤ 1 then
2:     return n
3: else
4:     x ← FIB(n − 1)
5:     y ← FIB(n − 2)
6:     return x + y
```

Fibonacci-Zahlen gehen auf Leonardo da Pisa, auch Fibonacci genannt ($*$ um 1180 in Pisa; † nach 1241 in Pisa), zurück – ein bedeutender Mathematiker des Mittelalters. Fibonacci hat mit seinen Fibonacci-Zahlen 1202 das Wachstum einer Kaninchenpopulation beschrieben. (Die Reihe war aber schon in der indischen und westlichen Antike bekannt.)

Wir werden Fibonacci noch häufiger zur Illustration verschiedenster Themen verwenden, weil sie so schön einfach zu definieren sind, aber dennoch komplexes Verhalten aufweisen!

4.2 Ausführbarer Pseudocode

Wie in der Einleitung bemerkt, kann man anstelle von Pseudocode häufig „echten“ Code einer Hochsprache verwenden. Wir benutzen dazu Python (`http://www.python.org`). Sie brauchen Python *nicht* zu lernen, um die Beispiele zu verstehen, denn wie Sie sehen werden, besteht kaum ein Unterschied zwischen Pseudocode und Python. Daher wird Python oft auch als „ausführbarer“ Pseudocode bezeichnet. Das Fibonacci Beispiel ist als Algorithmus 4.2 im Listing 4.1 dargestellt.

Algorithmus 4.2 FIB-P(n)

Listing 4.1 Fibonacci (Ausführbarer Pseudocode)

```
1  def fib(n):
2      if n==1 or n==2:
3          return 1
4      else:
5          return fib(n-1) + fib(n-2)
```

Der wichtigste Unterschied besteht in der Angabe von Schleifenkonstrukten. In Python schreibt man

```
for i in range(n)
```

wenn man von 0 bis $n-1$ iterieren möchte.

Wenn Sie Python ausprobieren wollen, können sie Python für Ihre Plattform installieren (die meisten Linux-Distributionen und Mac OSX enthalten Python vorinstalliert) und die Grundzüge der Sprache lernen. Eine gute Einführung in Python (die viel mehr enthält, als wir hier brauchen) ist z. B. das Tutorial (`http://docs.python.org/3.1/tutorial/`) oder “Instant Python” (`http://hetland.org/writing/instant-python.html`), ein Online E-Book (`http://www.swaroopch.com/notes/Python`), sowie [ZD04] und [Het08].

In diesem Skript finden Sie viele Varianten der Algorithmen als ausführbaren Pseudocode. Diese sind mit dem Postfix „-P“[2] gekennzeichnet. In der Regel sind diese Varianten identisch mit dem Pseudocode in Pascal Notation; vereinzelt gibt es jedoch Abweichungen wie z. B. eine funktionalere Version o.ä.

4.3 C-Code

Wie erwähnt geben wir viele vorgestellten Algorithmen auch als C-Code an, insbesondere in den Übungen werden wir dies benutzen! Eine gutes

[2] „P“ für „Python“

Nachschlagewerk für Algorithmen in C ist [Sed90]. Das Fibonacci Beispiel ist im Listing 4.2 dargestellt.

Listing 4.2: Fibonacci (C-Code)

```
1  #include <stdio.h>
2
3  int FIB(int n)
4  {
5      if(n <= 1)
6          return n;
7      else
8          return FIB(n-1) + FIB(n-2);
9  }
10
11 int main(int argc, char *argv[])
12 {
13     if(argc != 2){
14         printf("Usage: fib n\n");
15         return 1;
16     }
17     int i = atoi(argv[1]);
18     printf("FIB(%i)=%i\n", i, FIB(i));
19     return 0;
20 }
```

4.4 Struktogramme

Nassi-Shneiderman-Diagramme (s. [NS73]) werden verwendet, um sowohl Programmabläufe als auch Algorithmen zu visualisieren. in der Praxis haben sie jedoch eine geringe Bedeutung und sie werden meist durch UML-Konstrukte ersetzt. Abbildung 4.2 gibt ein Beispiel für ein Nassi-Shneiderman-Diagramm des rekursiven Fibonacci-Algorithmus.

4.5 Programmablaufplan

Ein Programmablaufplan ist ein Ablaufdiagramm für ein Programm, das auch als Flussdiagramm (englisch “flowchart”) oder Programmstrukturplan bezeichnet wird. Es ist eine graphische Darstellung eines Algorithmus. Heutzutage werden dazu meisten UML-Aktivitätsdiagramme verwendet (eine gute Einführung ist z. B. [FS00]). Nachfolgend (Abbildung 4.3) wieder das Fibonacci Beispiel.

FIB — Fibonacci

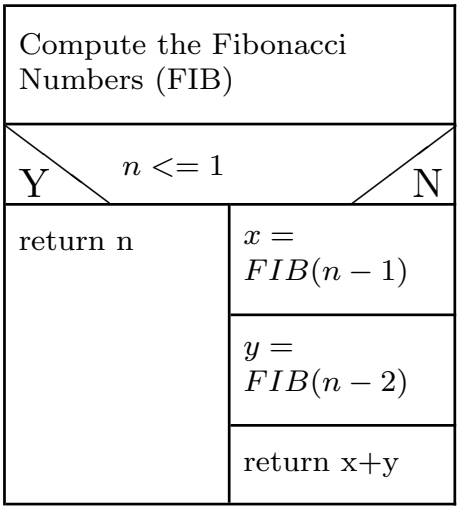


Abbildung 4.2: Nassi-Shneiderman Fibonacci

4.6 Datenflussdiagramme

Anstelle von Programmablaufplänen werden manchmal auch Datenflussdiagramme verwendet, wenn der Schwerpunkt auf den Daten und weniger auf dem verwendeten Algorithmus liegt. Datenflussdiagramme sind strukturell Aktivitätsdiagrammen weitgehend äquivalent.

4.7 Fragen und Aufgaben zum Selbststudium

1. Beschreiben Sie in Pseudo-Code Notation die Fakultäts-Funktion!
2. Beschreiben Sie in Pseudo-Code Notation einen Algorithmus BACHELOR-GRADE(G), der aus einer Liste G von Prüfungsnoten ihre Bachelor-Abschlussnote berechnet. Sie können dazu die tatsächliche Punktegewichtung aus der PO verwenden!
3. Berechnen Sie BACHELOR-GRADE(G) mit ausführbarem Pseudocode!
4. Berechnen Sie BACHELOR-GRADE(G) mit C-Code!
5. Zeichnen Sie das Struktogramm für BACHELOR-GRADE!
6. Zeichnen Sie einen Programmablaufplan für BACHELOR-GRADE!

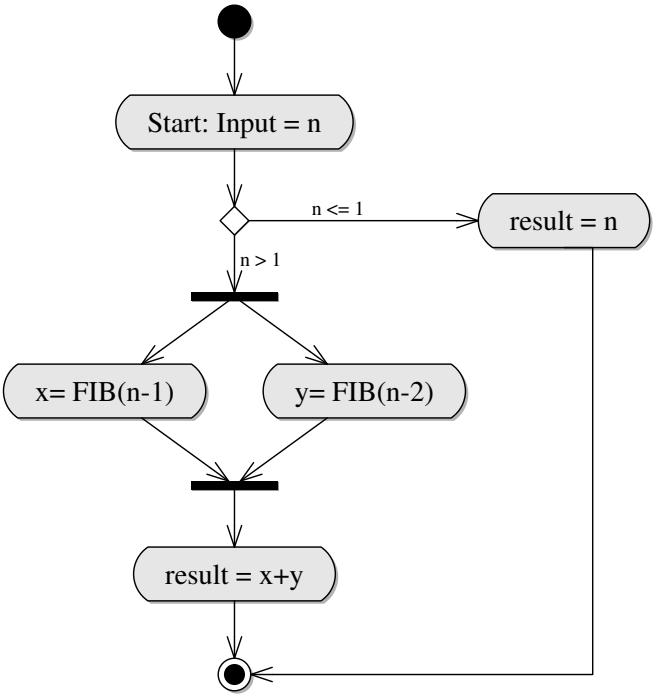


Abbildung 4.3: Programmablaufplan Fibonacci

7. Zeichnen Sie jeweils einen Programmablaufplan für jedes im nächsten Kapitel behandelte Sortierverfahren!

Kapitel 5

Sortieren

Diese Kapitel beinhaltet die wichtigsten Sortierverfahren. Eine schöne (Laufzeit-) Visualisierung der einzelnen Verfahren finden Sie auf der Webseite `http://www.sorting-algorithms.com`.

5.1 Mergesort

5.1.1 Algorithmus

MERGESORT wurde erstmals 1945 durch John von Neumann vorgestellt. Das Verfahren ist ein Beispiel für das so genannte *Divide-and-Conquer-Prinzip* (lat. Divide et impera! oder Teile-und-Herrsche-Prinzip). Die zu sortierende Folge wird zunächst in zwei Hälften aufgeteilt ("divide"), die jeweils für sich sortiert werden ("conquer"). Hier sehen Sie wieder Rekursion am Werk! Danach werden die sortierten Hälften zu einer insgesamt sortierten Folge verschmolzen ("combine").

Das Teile-und-Herrsche-Prinzip ist ein allgemeines Verfahren zur Problemlösung und in Abbildung 5.1 verdeutlicht. MERGESORT 5.1 stellt den Pseudocode dar. MERGESORT benutzt dabei den Algorithmus MERGE 5.2, den man sich anhand eines Beispiels verdeutlichen sollte (er funktioniert ähnlich wie das Sortieren zweier bereits sortierter Stapel von Spielkarten).

Der ausführbare Pseudocode ist in Listing 5.1 dargestellt.

Der Ablauf von MERGESORT ist in Abbildung 5.2 verdeutlicht – zusammengefügte Felder werden durch runde Klammern dargestellt.

5.1.2 Analyse

Lemma 5.1.1. *MERGESORT ist korrekt.*

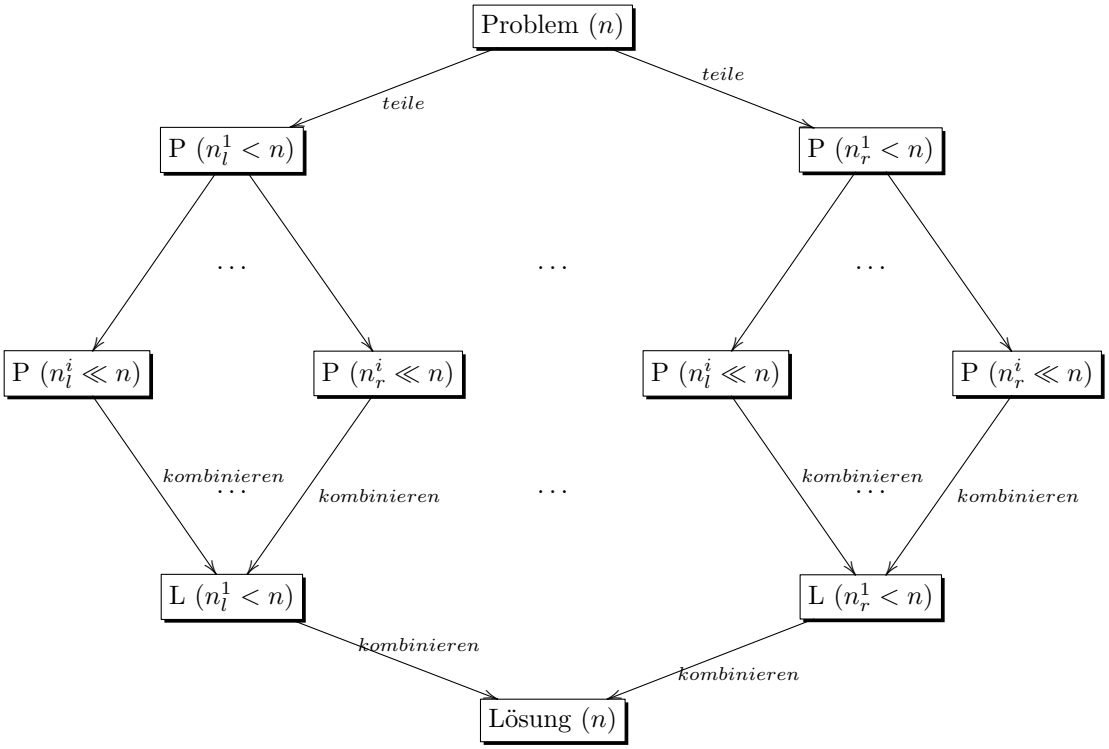


Abbildung 5.1: Teile-und-Herrsche-Prinzip

Algorithmus 5.1 MERGESORT(A)

```
1: if length(A) <= 1 then
2:     return A
3: middle ← ⌊length(A)/2⌋
4: left ← A[0, ..., middle − 1]
5: right ← A[middle, ..., length(A) − 1]
6: left ← MERGESORT(left)              // Recursion
7: right ← MERGESORT(right)            // Recursion
8: if left[last] > right[first] then
9:     return MERGE(left, right)       // Need to merge
10: else
11:    return (left, right)            // Nothing to merge, just append
```

Beweis. Da das Array A bei jedem Rekursionsschritt halbiert wird, ist klar, dass die Rekursion abbricht. Die Teilfelder $left$ und $right$ in Zeile 6 und 7 sind korrekt sortiert, falls wir annehmen können, dass MERGESORT für alle „kleineren“ Felder, d.h. für alle Felder A' mit $len(A') < len(A)$ korrekt ist. In Zeile 9 kombiniert MERGE jeweils zwei Teilfelder. Wenn wir

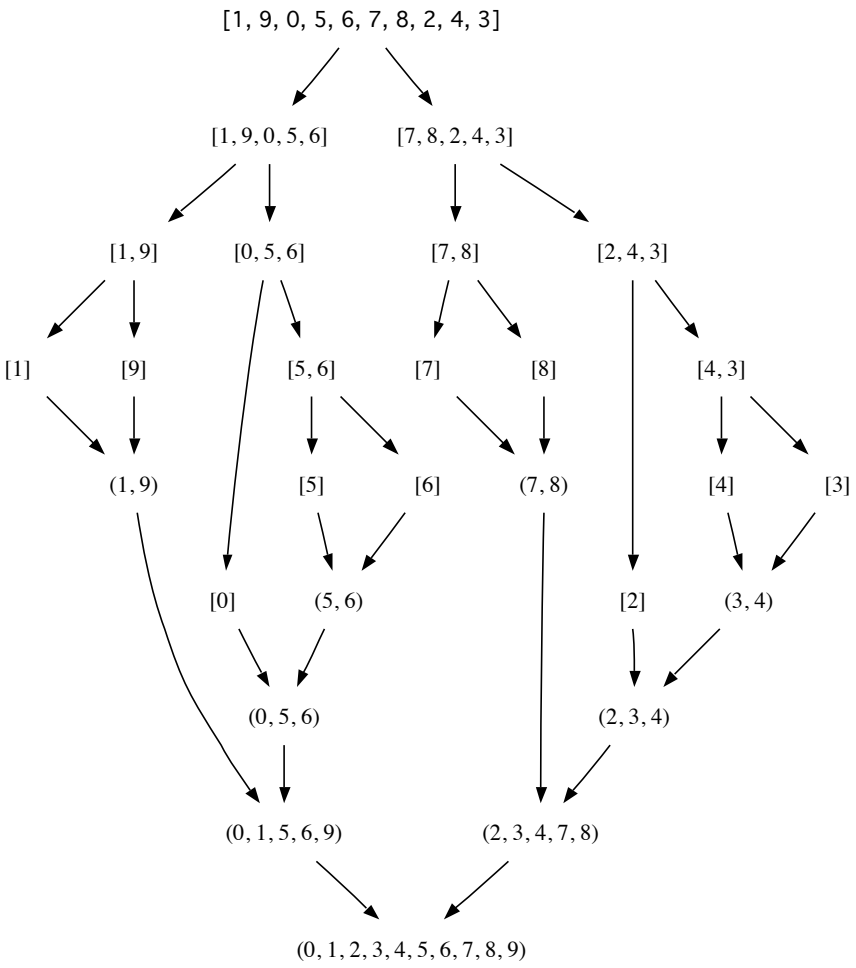


Abbildung 5.2: Ablauf von MERGESORT

Algorithmus 5.2 MERGE($left, right$)

```
1:  result ← ()                                   // Start with empty list
2:  while length(left) > 0 and length(right) > 0 do
3:      if left[0] <= right[0] then
4:          result ← (result, left[0])
5:          left ← left(1, ...)                   // Remove first element
6:      else
7:          result ← (result, right[0])
8:          right ← right(1, ...)                 // Remove first element
9:  if length(left) > 0 then
10:     result ← (result, left)                   // Right empty
11: else
12:     result ← (result, right)                  // Left empty
13: return result
```

also zeigen können, dass MERGE korrekt ist, dann haben wir die folgende Aussage bewiesen: Falls MERGESORT korrekt ist für alle Felder A' mit $len(A') < len(A)$, dann ist MERGESORT auch für A selbst korrekt. Mit Induktion über die Länge des Arrays folgt dann die Behauptung für alle Felder.

Wir müssen also nur noch zeigen, dass MERGE korrekt ist. Dazu benutzen wir die folgende Schleifeninvariante (für die while Schleife in Zeile 2):

In jeder Iteration i enthält das Feld "result" die geordneten Schlüssel $A'[1], \ldots, A'[i]$.

Das stimmt offensichtlich für den Anfang $i = 0$ und das leere Feld. Da nach Voraussetzung left und right geordnet sind, wird in Zeile 4 oder 7 genau ein nächstgrößeres Element hinzugefügt. Wenn entweder left oder right leer sind, bricht die Schleife ab, und wir haben damit $A'[1], \ldots, A'[n']$ mit $n' := n - \max(length(left), length(right))$ geordnet. Zeile 10 bzw. 12 stellt sicher, dass das übrig gebliebene, nicht-leere und sortierte Teilfeld dem Ergebnis $result$ hinzugefügt wird, so dass $result$ komplett sortiert ist. □

Zur Analyse der Laufzeit verwenden wir die im Kapitel 2 bereits erwähnte *Baummethode* (Abbildung 5.3) : Wie man sieht, erhält man in jeder Ebene $\sum = cn$ Beiträge. Die Baumtiefe beträgt $\lg n$, denn es gilt $2^k = n$ mit Baumtiefe k. Daher erhält man insgesamt die Kosten von

$$T(n) = cn \lg n$$

Damit haben wir das folgende Lemma gezeigt.

Lemma 5.1.2. *MERGESORT ist in* $\mathcal{O}(n \lg n)$.

Algorithmus 5.3 MERGESORT-P(A)

Listing 5.1 Merge Sort (Ausführbarer Pseudocode)

```
1  def mergesort(a):
2      if len(a) <= 1:
3          return a
4      middle = len(a)//2
5      left, right = a[0:middle], a[middle: len(a)]
6      left, right = mergesort(left), mergesort(right)
7      # compare last item from left with first item from right
8      if left[-1] > right[0]:
9          return merge(left, right)
10     else:
11         return left + right
12
13 def merge(left, right):
14     result = []
15     while len(left) > 0 and len(right) > 0:
16         if left[0] <= right[0]:
17             result.append(left[0])
18             left = left[1:]
19         else:
20             result.append(right[0])
21             right = right[1:]
22     if len(left) > 0: # append the rest
23         result += left
24     else:
25         result += right
26     return result
27
28 print(mergesort([1, 9, 0, 5, 6, 7, 8, 2, 4, 3]))
```

Ein alternativer Beweis nutzt die folgende Rekursionsgleichung aus

$$T(n) = T(n/2) + T(n/2) + \Theta(n),$$

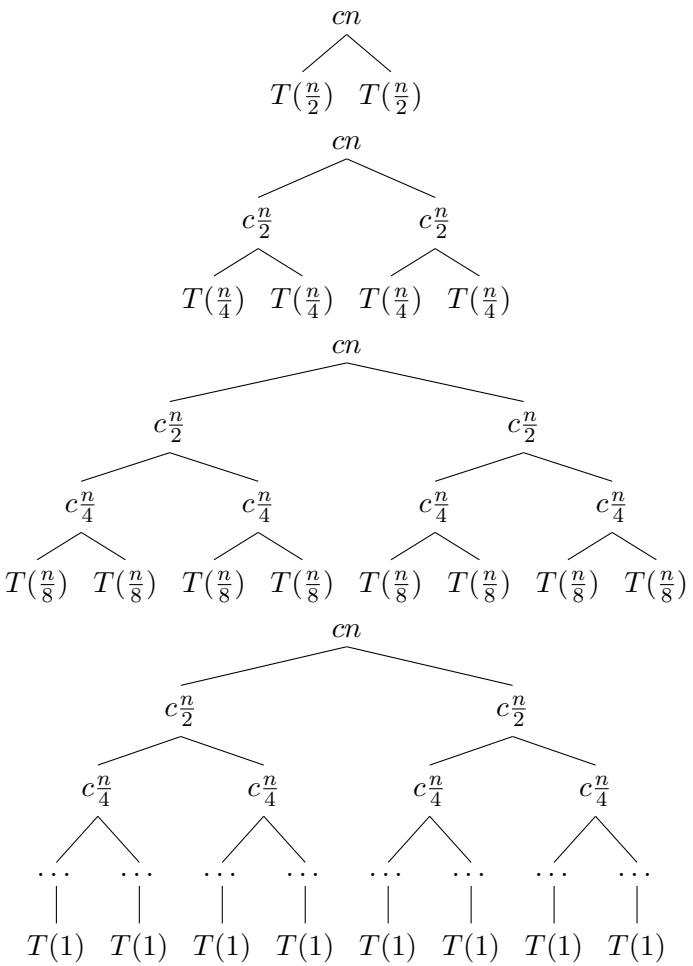
die man sofort aus der Definition ablesen kann, wenn man sich überlegt hat, dass MERGE in $\Theta(n)$ ist. Damit ergibt sich das Resultat aus dem Mastertheorem (Fall 2).

Außerdem gilt (ohne Beweis)

Lemma 5.1.3. *MERGESORT ist ein* stabiles Sortierverfahren*, d.h. die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, wird bewahrt.*

Definition 5.1.1. *Ein Algorithmus arbeitet in-place bzw. in situ, wenn er außer dem zur Speicherung der zu bearbeitenden Daten benötigten Speicher nur eine konstante, also von der zu bearbeitenden Datenmenge unabhängige*

Abbildung 5.3: Die Baummethode



Menge $\mathcal{O}(1)$ von Speicher benötigt. Um dies zu erreichen, überschreibt der Algorithmus also Eingabedaten mit Ausgabedaten.

Offenbar gilt:

Lemma 5.1.4. *MERGESORT sortiert* nicht *"in place".*

5.2 Quicksort

5.2.1 Algorithmus

Quicksort ist ein rekursiver, in-place und nicht-stabiler[1] Sortieralgorithmus, der nach dem Teile-und-Herrsche-Prinzip arbeitet. Er wurde ca. 1960 von C.

[1] jedenfalls in der üblichen, effizienten Implementierung von PARTITION (s.u.).

Antony R. Hoare entwickelt und existiert in vielen Varianten. Der Algorithmus hat den Vorteil, dass er schnell ist, da er über eine sehr kurze innere Schleife verfügt und ohne zusätzlichen Speicherplatz auskommt (abgesehen von dem für die Rekursion zusätzlichen benötigten Platz auf dem Aufruf-Stack).

Die Grund-Idee ist nach dem Teile-und-Herrsche-Prinzip simpel: Gegeben sei ein Feld $A[p, \ldots, r]$, dann

1. Divide: $A[p, \ldots, r]$ teilen in $A[p, \ldots, q-1]$ und $A[q+1, \ldots, r]$, so dass alle Elemente von $A[p, \ldots, q-1]$ kleiner sind als $A[q]$ oder gleich und alle von $A[q+1, \ldots, r]$ größer.
2. Conquer: Die beiden Teilfelder durch rekursive Aufrufe sortiert.
3. Combine: Da die Teilfelder bereits sortiert sind, ist nichts zu tun, da der Quicksort-Algorithmus in-place arbeitet.

Damit sieht der Pseudo-code von QUICKSORT sehr einfach aus, wie im Algorithmus QUICKSORT 5.4 dokumentiert.

Algorithmus 5.4 QUICKSORT(A)

```
1: if A is empty then
2:     return A
3: l ← length(A) − 1
4: pivot ← A[l]                                  // Take last element as pivot
5: less ← {A[i] | A[i] ≤ pivot, i ≠ l}
6: more ← {A[i] | A[i] > pivot, i ≠ l}
7: return (QUICKSORT(less), pivot, QUICKSORT(more))
```

Der ausführbare Pseudocode ist in Listing 5.2 dargestellt.

Programmiersprachen, die Manipulationen von Feldern und Listen direkt unterstützen, wie z. B. die meisten funktionalen Sprachen, können diesen Pseudocode direkt umsetzen, so auch Python, mit dem man (mit Einschränkungen) funktional programmieren kann, wie in Algorithmus 5.6 verdeutlicht (mit `a[-1]` wird in Python das letzte Element des Feldes bezeichnet, mit `a[:-1]` bekommt man in Python das Feld bis auf das letzte Element).

Um mit imperativen Programmiersprachen (wie C) die Idee von QUICKSORT umzusetzen, muss man das Aufteilen (englisch "partition") explizit programmieren. Der Pseudo-code von QUICKSORT sieht dann so aus, wie im Algorithmus QUICKSORT 5.7 dokumentiert. (Der korrekte Aufruf lautet: QUICKSORT$(A, 0, length(A) - 1)$.)

Der entscheidende Algorithmus ist hier PARTITION 5.8.

Algorithmus 5.5 QUICKSORT-P(A)

Listing 5.2 Quicksort (Ausführbarer Pseudocode)

```
1  def quicksort(a, p, r):
2      if p < r:
3          q = partition(a, p, r)
4          quicksort(a, p, q-1)
5          print(a[p:q-1])
6          quicksort(a, q+1, r)
7          print(a[q+1:r])
8      return a
9
10 def partition(a, p, r):
11     x = a[r] # pivot
12     i = p-1
13     for j in range(p, r):
14         if a[j] <= x:
15             i = i+1
16             a[i], a[j] = a[j], a[i]
17     a[i+1], a[r] = a[r], a[i+1]
18     return i+1
```

Die Funktionsweise von PARTITION, der ein sog. Pivotelement[2] zur Teilung des Feldes benutzt, mache man sich am besten anhand der Abbildung 5.4 klar!

5.2.2 Analyse

Lemma 5.2.1. *QUICKSORT ist korrekt.*

[2]Es gibt verschiedene Varianten von QUICKSORT, die unterschiedliche Strategien zur Bestimmung des Pivotelements benutzen!

Algorithmus 5.6 QUICKSORT-FUN-P(A)

Listing 5.3 Quicksort – Funktionale Version (Ausführbarer Pseudocode)

```
1  def qs(a):
2      if not a:
3          return []
4      else:
5          pivot = a[-1]
6          less = [x for x in a[:-1] if x <=  pivot]
7          more = [x for x in a[:-1] if x > pivot]
8          return qs(less) + [pivot] + qs(more)
```

Algorithmus 5.7 QUICKSORT(A, p, r)

```
1: if p < r then
2:     q ← PARTITION(A, p, r)
3:     QUICKSORT(A, p, q − 1)
4:     QUICKSORT(A, q + 1, r)
```

Algorithmus 5.8 PARTITION(A, p, r)

```
1: x ← A[r]                                   // Pivotelement
2: i ← p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i ← i + 1
6:         A[i], A[j] ← A[j], A[i]
7: A[i + 1], A[r] ← A[r], A[i + 1]
8: return i + 1
```

Beweis. Zu zeigen ist offensichtlich nur, dass PARTITION die gewünschten Eigenschaften hat, denn alles Weitere folgt bereits aus den eingehenden Erläuterungen zum Teile-und-Herrsche-Prinzip von QUICKSORT. Das sieht man aber anhand der folgenden Schleifeninvariante, die beschreibt, wie das Feld in drei Teile eingeteilt wird – die ersten zwei enthalten diejenigen Elemente, die kleiner bzw. größer sind als das Pivotelement, der dritte Teil besteht aus den bisher noch nicht analysierten Einträgen.

Bei Beginn jeder Iteration in den Zeilen 3-6 von PARTITION gilt:

1. Wenn $p \leq k \leq i$, dann $A[k] \leq x$ (x ist das Pivotelement).
2. Wenn $i + 1 \leq k \leq j - 1$, dann $A[k] > x$.
3. Wenn $k = r$, dann $A[k] = x$.

Dass dies in der Tat eine Schleifeninvariante ist verdeutlichen die Abbildungen 5.5 (1. Fall: $A[j] > x$) und 5.6 (2. Fall $A[j] \leq x$).

Wenn die Schleife „durch“ ist, gilt aber $j = r - 1$ und damit die gewünschte Unterteilung. □

Lemma 5.2.2. *Die Laufzeit von QUICKSORT ist im Worst-Case $\mathcal{O}(n^2)$, aber im “Normalfall” in $\mathcal{O}(n \lg n)$.*

Beweis. Der Worst-Case tritt dann ein, wenn PARTITION ein Feld der Länge n *immer* in ein Feld der Länge $n - 1$ und Null zerlegt (plus das Pivotelement). In diesem Fall gilt für die Laufzeit:

$$T(n) \quad = \quad T(n-1) + T(0) + \Theta(n)$$

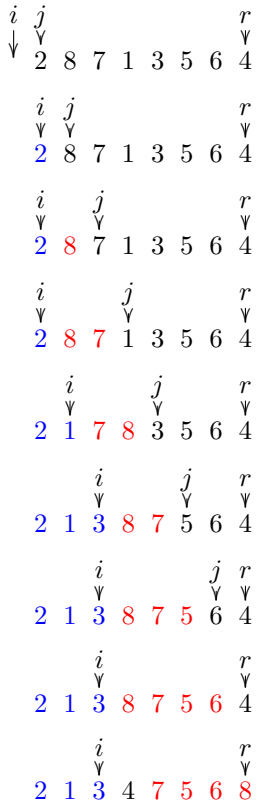


Abbildung 5.4: PARTITION (Rückgabe ist $q = i + 1$)

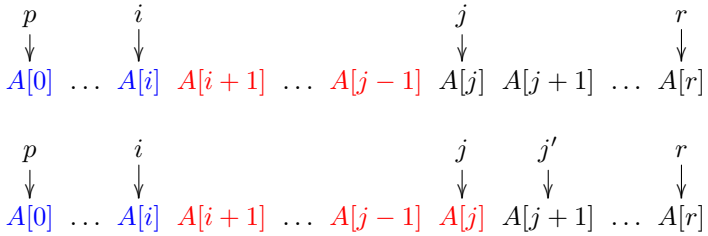


Abbildung 5.5: Schleifeninvariante Quicksort, 1. Fall: $A[j] > A[r] = x$ (mit $j' = j + 1$: Wert von j im nächsten Schleifendurchlauf)

$$= \quad T(n-1) + \Theta(n)$$

Dies hat als Lösung $T(n) = \Theta(n^2)$ (Übungsaufgabe zur Substitutionsmetho-

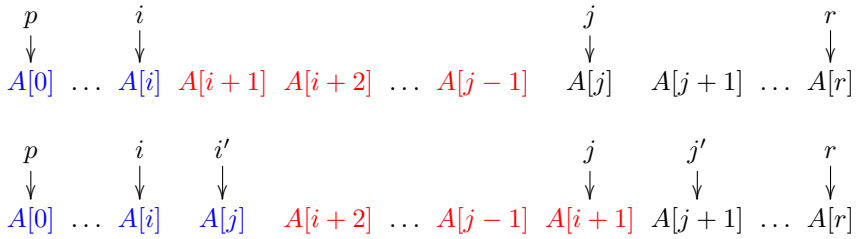


Abbildung 5.6: Schleifeninvariante Quicksort, 2. Fall: $A[j] \leq A[r] = x$ (mit $i' = i + 1$ und $j' = j + 1$: Werte im nächsten Schleifendurchlauf)

de).

Falls PARTITION ein Feld der Länge n *immer* in ein Feld der Länge $n/2$ zerlegt, dann gilt dagegen:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) \\ &= 2\,T(n/2) + \Theta(n) \end{aligned}$$

Dies hat als Lösung $T(n) = \Theta(n \lg n)$ nach dem Mastertheorem (Übungsaufgabe). □

Bemerkung: Man kann zeigen, dass bei jeder Partitionierung, die das Feld um einen Faktor verkleinert (und nicht nur eine konstante Anzahl von Feldern), die Laufzeit $\Theta(n \lg n)$ ist, d.h. z. B. selbst wenn man ein Feld der Länge 100 in 90 und 10 unterteilt (usw.).

In der Praxis zieht man QUICKSORT in vielen Fällen MERGESORT vor, weil die Laufzeit fast immer besser ist (zwar sind beide $\Theta(n \lg n)$, aber die Konstanten sind besser) und der Worst-Case fast nie eintritt. Beachten Sie jedoch, dass der Worst-Case (mit der in der obigen Version gewählten Strategie zur Bestimmung des Pivotelements) dann auftritt, wenn QUICKSORT ein Feld sortieren soll, das *bereits sortiert* ist. Da dieser Fall in der Praxis nicht ungewöhnlich ist, reduziert man die Wahrscheinlichkeit des Worst-Case bei Implementierungen von QUICKSORT durch *Randomisierung*, d.h. durch zufälliges Durchmischen der Einträge vor der Sortierung oder Auswahl des Pivotelementes aus einer Menge von Elementen!

5.3 Heapsort

5.3.1 Algorithmus

Heapsort (Haldensortierung) ist ein 1964 von Robert W. Floyd und J. W. J. Williams entwickeltes Sortierverfahren. Heapsort nutzt eine Datenstruktur aus, den Heap (deutsch Halde).

Definition 5.3.1. *Eine endliche Menge T von Knoten heißt* Binärbaum *genau dann wenn*

1. *die Menge T leer ist, oder*
2. *ein ausgezeichneter Knoten T_0, die Wurzel, existiert und genau zwei disjunkte Binärbäume, der linke T_l und der rechte Teilbaum T_r, existieren, so dass gilt $T = T_0 \cup T_l \cup T_r$.*

In diesem Kontext wird die leere Menge auch als Blatt bezeichnet. Ein Baum besteht also aus einer Wurzel, beliebig vielen inneren Elementen und darauf basierend endlich vielen Blättern. Ein Binärbaum ist ein vollständiger *Baum genau dann wenn alle Blätter die gleiche Tiefe haben, d.h. wenn die Entfernung sämtlicher Blätter zur Wurzel gleich ist.*

Definition 5.3.2. *Ein Heap ist ein* nahezu vollständiger *Binärbaum, der die* Max-Heap-Eigenschaft *(bzw.* Min-Heap-Eigenschaft*) besitzt: Jeder Knoten hat einen Schlüssel, der höchstens (bzw. mindestens) so groß ist wie der Schlüssel des Elternteils.* Nahezu vollständig *bedeutet, dass alle Ebenen mit Ausnahme der letzten vollständig sind, also alle Knoten enthalten. In der letzten Ebene müssen – von links beginnend – alle Blätter vorhanden sein bis zu einer Position. Ab einer Position dürfen keine weiteren Blätter mehr vorhanden sein.*

Heaps kann man sich also am besten als Bäume vorstellen, wie in Abbildung 5.7 dargestellt.

Heaps können nicht nur als Bäume definiert werden, sondern auch effizient als Felder. Letzteres besitzt den Vorteil, dass man die Elemente unter Ausnutzung der Binärstruktur direkt adressieren kann wie in Abbildung 5.8 und Abbildung 5.9 dargestellt. Das linke Element des Elements des Index i ergibt sich aus der Formel $2i$, das rechte aus der Formel $2i + 1$ (bzw. aus den Formeln $2i + 1$ und $2i + 2$, wenn die Feldindizees bei „0“ starten).

Wichtig ist, dass zu einem Heap mit Heapsize von n nur die ersten n Elemente des Feldes gehören, d.h. auch wenn das Feld Elemente $A[i]$ mit $i > n$ enthält, so gehören diese *nicht* zum Heap. Das ist eine praktische Konvention, weil man dann Elemente nicht entfernen muss und wir werden davon beim HEAPSORT 5.9 Algorithmus Gebrauch machen.

Die Idee des HEAPSORT Algorithmus besteht darin, dass das maximale

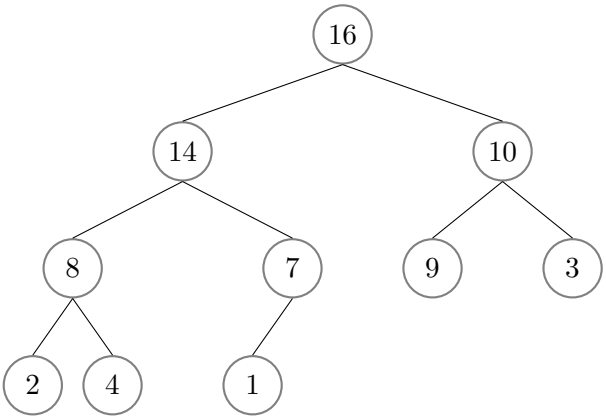


Abbildung 5.7: Ein Heap

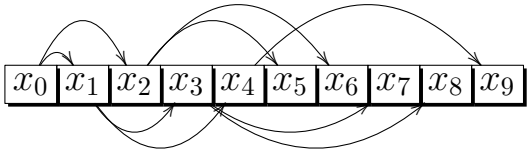


Abbildung 5.8: Heap als Feld

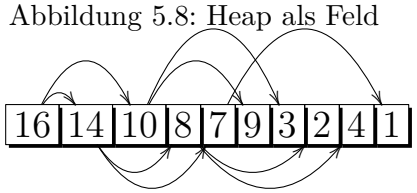


Abbildung 5.9: Heap aus Abbildung 5.7 als Feld

Element bei einem Max-Heap das erste ist (das Wurzel-Element). Wir können nun die Elemente sortieren, in dem wir das erste Element mit dem letzten vertauschen und den Heapsize um eins verringern. Danach steht das größte Element am Ende des Feldes, das wegen des verkleinerten Heapsize nicht mehr zum Heap gehört. Allerdings ist nach dem Tausch das erste Element nicht notwendigerweise das größte, also die Max-Heap-Eigenschaft u.U. verletzt. Dies korrigieren wir mit einer Prozedur MAX-HEAPIFY 5.10. Am Ende ist der Heap leer, und das Feld sortiert.

Algorithmus 5.9 HEAPSORT(A)

```
1: BUILD-MAX-HEAP(A)
2: for i = length(A) − 1 down to 1 do
3:     A[0], A[i] ← A[i], A[0]
4:     MAX-HEAPIFY(A, i, 0)
```

Der entscheidende Algorithmus ist MAX-HEAPIFY 5.10, der auch in BUILD-MAX-HEAP 5.11 benutzt wird, um einen Heap initial zu erzeugen. Wenn die Unterbäume des Knotens $A[i]$ des Heaps A die Max-Heap-Eigenschaft

Algorithmus 5.10 MAX-HEAPIFY(A, h, i)

```
1: l ← 2i + 1                          // lists start with 0
2: r ← 2i + 2                          // lists start with 0
3: if l < h and A[l] > A[i] then       // max of parent and left child
4:     max ← l
5: else
6:     max ← i
7: if r < h and A[r] > A[max] then     // compare right child with max
8:     max ← r
9: if max ≠ i then                     // correct max heap at i
10:    A[i], A[max] ← A[max], A[i]
11:    MAX-HEAPIFY(A, h, max)          // recursively repair at max
```

besitzen, der Knoten $A[i]$ jedoch kleiner als einer seiner Kinder ist, dann „repariert“ MAX-HEAPIFY dies, indem der Eintrag $A[i]$ mit dem Maximum der Teilbäume vertauscht wird (man spricht auch vom „Versickern“). Ggf. muss dieser Prozess wiederholt werden (Rekursion), bis die unterste Ebene des Heaps erreicht ist.

BUILD-MAX-HEAP erzeugt aus einem Feld einen Heap durch wiederholtes Aufrufen von MAX-HEAPIFY.

Algorithmus 5.11 BUILD-MAX-HEAP(A)

```
1: h ← length(A)
2: for i = ⌊h/2⌋ down to 1 do
3:     MAX-HEAPIFY(A, h, i-1)
```

Die Funktionsweise von MAX-HEAPIFY ist in Abbildung 5.10 illustriert.

Der ausführbare Pseudocode ist in Listing 5.4 dargestellt.

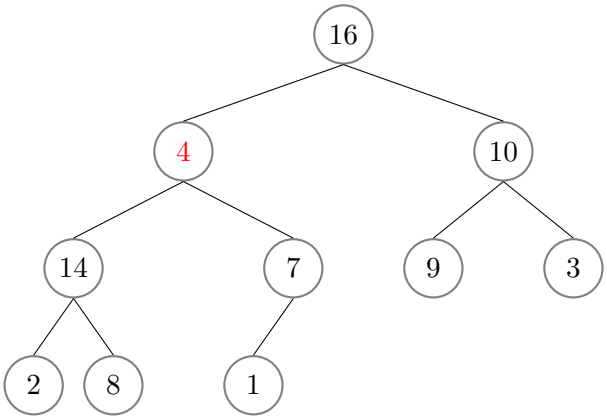
5.3.2 Analyse

Lemma 5.3.1. *HEAPSORT ist korrekt.*

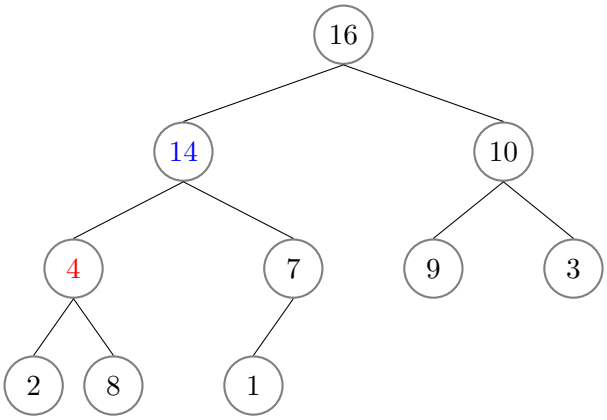
Beweis. (Skizze, für Details s. z. B. [CLR04])

HEAPSORT bricht „offensichtlich“ ab und ist genau dann korrekt, wenn MAX-HEAPIFY und BUILD-MAX-HEAP korrekt sind:

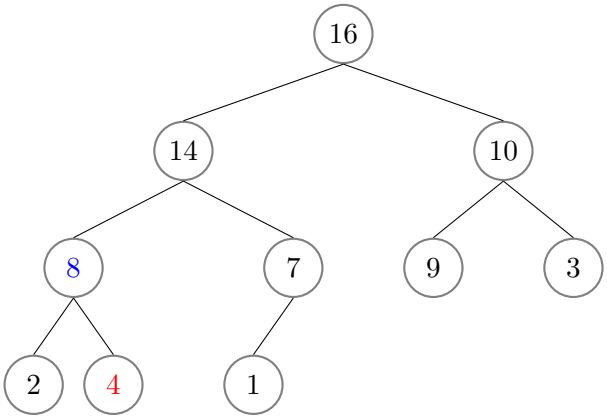
1. MAX-HEAPIFY ist korrekt, weil das Vertauschen und die Rekursion spätestens bei den Blättern des Heaps abbricht und damit die Heap-Eigenschaft wiederhergestellt wird.



(a) Knoten 1 (Wert 4) verletzt die Max-Heap-Eigenschaft



(b) MAX-HEAPIFY 1. Schritt



(c) MAX-HEAPIFY 2. Schritt

Abbildung 5.10: MAX-HEAPIFY

2. BUILD-MAX-HEAP hat die Schleifeninvariante: Beim Start jeder

Algorithmus 5.12 HEAPSORT-P(A)

Listing 5.4 Heapsort (Ausführbarer Pseudocode)

```
1  def heapify(a, heapsize, i):
2      l = 2*i+1 # lists start with 0!
3      r = 2*i+2 # lists start with 0!
4      if l < heapsize and a[l] > a[i]:
5          maximum = l
6      else:
7          maximum = i
8      if r < heapsize and a[r] > a[maximum]:
9          maximum = r
10     if maximum != i:
11         a[i], a[maximum] = a[maximum], a[i]
12         heapify(a, heapsize, maximum)
13
14 def build_heap(a):
15     for i in range(len(a)//2, 0, -1):
16         heapify(a, len(a), i-1)
17
18 def heapsort(a):
19     build_heap(a)
20     for i in range(len(a)-1, 0, -1):
21         a[0], a[i] = a[i], a[0]
22         heapify(a, i, 0)
```

Iteration (Zeile 2–3) ist jeder Knoten $i+1, i+2, \ldots, n$ die Wurzel eines Max-Heaps. Beim Schleifenabbruch ist aber damit A ein Max-Heap (es wird von unten nach oben die Max-Heap-Eigenschaft hergestellt).

□

Lemma 5.3.2. *HEAPSORT ist in* $\Theta(n \lg n)$.

Beweis. Wir zeigen zunächst, dass MAX-HEAPIFY in $\Theta(\lg n)$ ist. Dann gilt für die Laufzeit $T(n)$ von HEAPSORT *ohne* BUILD-MAX-HEAP

$$T(n) = \Theta(n)\,\Theta(\lg n) = \Theta(n \lg n),$$

da die Schleife (Zeilen 3–6) offensichtlich in $\Theta(n)$ ist.

Für MAX-HEAPIFY gilt die folgende Rekursionsgleichung:

$$T(n) = T\left(\frac{2}{3}n\right) + \Theta(1),$$

(Der Faktor 2/3 ergibt sich aus dem ungünstigen Fall, wenn die unterste Ebene des Heaps nur zur Hälfte gefüllt ist.)

Nach dem Mastertheorem (oder per Substitution) hat dies die Lösung $\Theta(\lg n)$.

BUILD-MAX-HEAP ist in $\mathcal{O}(n)$. Das kann man wie folgt einsehen. MAX-HEAPIFY hat eine Laufzeit von $\mathcal{O}(h)$, wobei h die Höhe des Heaps bezeichnet. Damit gilt für die Laufzeit $T(n)$ von BUILD-MAX-HEAP:

$$T(n) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil n/2^{h+1} \right\rceil \mathcal{O}(h) \leq \mathcal{O}\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \right)$$

Da außerdem $\sum_{h=0}^{\infty} h/2^h = 2$ gilt (ohne Beweis), ergibt sich dann insgesamt

$$T(n) \leq \mathcal{O}(n)$$

Und somit gilt für die Laufzeit $T(n)$ von HEAPSORT:

$$T(n) = \mathcal{O}(n) + \Theta(n \lg n) = \Theta(n \lg n)$$

□

5.4 Bubblesort

5.4.1 Algorithmus

Ein weit-verbreiterter Sortieralgorithmus ist BUBBLESORT 5.13.

Algorithmus 5.13 BUBBLESORT(A)

```
1: repeat
2:     swapped ← false
3:     for i = 0 to length(A) − 2 do
4:         if A[i] > A[i + 1] then
5:             A[i], A[i + 1] ← A[i + 1], A[i]
6:             swapped ← true
7: until not swapped
```

(Es gibt BUBBLESORT in verschiedenen Variationen – wundern Sie sich also nicht, wenn Sie in Lehrbüchern oder Programmbeispielen alternative Varianten finden!)

5.4.2 Analyse

Lemma 5.4.1. *BUBBLESORT ist korrekt.*

Beweis. Übungsaufgabe! Überlegen Sie sich dazu, dass BUBBLESORT abbricht, was man einsieht, wenn man bedenkt, dass die Anzahl der Inversionen immer um Eins vermindert wird, wenn die Variable `swapped` in BUBBLESORT auf `true` gesetzt wird. Überlegen Sie sich danach, was beim Abbruch vorher in der `for` Schleife überprüft wurde. □

Lemma 5.4.2. *Die Laufzeit von BUBBLESORT liegt im Worst Case in* $\Theta(n+d)$*, wobei* n *die Anzahl der Schlüssel ist und* d *die Anzahl der Inversionen*[3] *in der Liste zu sortierender Schlüssel.*

Beweis. Übungsaufgabe! Überlegen Sie sich dazu, dass ein umgekehrt sortiertes Array ein Worst Case ist, bei dem alle Anweisungen des Algorithmus maximal oft durchlaufen werden. Überlegen Sie sich dann, wie die Anzahl der Inversionen und die Schleifendurchläufe verknüpft sind. □

Lemma 5.4.3. *Der Worst Case von BUBBLESORT ist in* $\Theta(n^2)$*.*

Beweis. Übungsaufgabe! Überlegen Sie sich dazu, dass die Anzahl der Inversionen für den Worst Case $n(n-1)/2$ beträgt. □

5.5 Vergleich Sortierverfahren

Die nachstehende Tabelle 5.1 vermittelt einen Überblick über die verschiedenen betrachteten Sortierverfahren. In der Spalte "Speicher" ist der zusätzliche Speicherplatz angegeben, den der Algorithmus benötigt. Dieser kann sowohl für Daten als auch für Variablen wie Indizes genutzt werden. Außerdem ist zu berücksichtigen, dass Funktionsaufrufe auch Speicher auf dem Stack benötigen. Dies ist möglicherweise bei rekursiven Algorithmen relevant, und in der Tabelle 5.1 implizit enthalten.

5.6 Vergleichsbasierte Sortierverfahren

Bislang haben wir Sortierverfahren betrachtet, die Elemente sortieren, indem sie diese Elemente paarweise vergleichen und Vertauschungen der Elemente ausführen oder diese umkopieren. Diese Verfahren nennt man *vergleichsbasierte* Sortierverfahren.

MERGESORT und HEAPSORT sind optimale vergleichsbasierte Sortierverfahren, wie das Theorem 5.6.1 zeigt.

Zunächst müssen wir uns mit einem *Entscheidungsbaum* beschäftigen, der in alle möglichen Fällen Elemente mit minimaler Anzahl Vergleiche sortiert. In Anlehnung an [CLR04] muss zunächst ein optimaler Entscheidungsbaum für die Sortierung gefunden werden. Eine Analyse der minimalen Höhe des Entscheidungsbaums (dies ist die optimale Anzahl der Elementvergleiche zur Sortierung) und aller möglichen Permutationen der Elemente führt schließlich zum Ergebnis.

[3] Als Inversion wird ein Paar $A[i], A[j]$ mit $i < j$ bezeichnet, für das gilt: $A[i] > A[j]$.

Tabelle 5.1: Vergleich Sortierverfahren

Name	Laufzeit			Speicher	Stabil	Inplace	Methode	Kommentar
	Avg.	Worst	Best					
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	Ja	Ja	Paartausch	Einfache Implementierung, gut verständlich.
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	Ja	Ja	Einsetzen	Einfache Implementierung, gut verständlich.
Mergesort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n)$	Ja	Nein	Merging	Merge benötigt zusätzliches Feld.
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(\lg n)$	Nein	Ja	Größtes Element in Wurzel des Heap	Zwei Phasen: Aufbau des Heaps und Entfernung des jeweils größten Elements aus dem Heap.
Quicksort	$\Theta(n \lg n)$	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(\lg n)$	implementierungsabhängig	Ja	Partitionierung	Speicher $\Theta(\lg n)$ wird benötigt wegen Rekursion – bei naiver Implementierung sogar $\Theta(n)$.

Vergleichsbasierte Sortierverfahren können Information über die Elemente nur durch paarweise Vergleiche erhalten. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass alle Elemente paarweise verschieden sind.

Ein optimaler Entscheidungsbaum für die Elemente a_1, a_2 und a_3 ist in Abbildung 5.11 gegeben. Entscheidungen müssen in den inneren Knoten des Baumes getroffen werden. Dies sind wie dort ersichtlich binäre Entscheidungen, weil ein Vergleich entweder wahr (weiter beim linken Kindknoten) oder falsch (weiter beim rechten Kindknoten) sind. Die neu erlangte Information über die zu sortierenden Elemente steht an der Kante zum Kindknoten. In den Blättern ist schließlich die sortierte Reihenfolge der Schlüssel angegeben. Es kann übrigens mehrere optimale Entscheidungsbäume geben, je nachdem in welcher (optimalen) Reihenfolge die Elemente verglichen werden, hier ist also keine Eindeutigkeit gegeben.

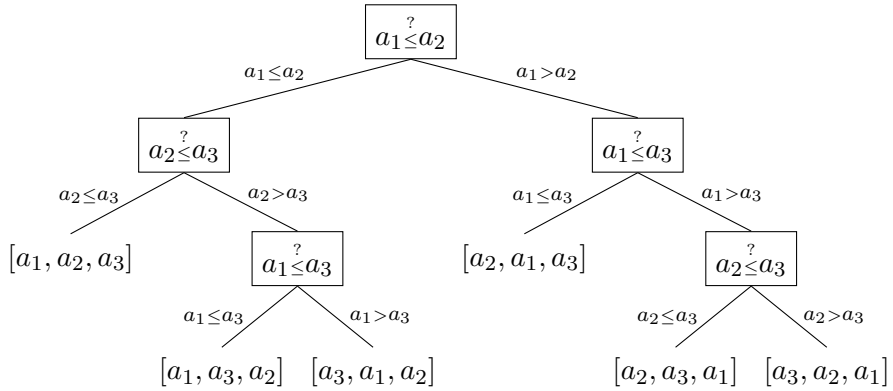


Abbildung 5.11: Entscheidungsbaum zum Sortieren mit einer minimalen Anzahl von Schlüssel-Vergleichen für drei Elemente a_1, a_2 und a_3

Wir betrachten das Beispiel $[a_1, a_2, a_3] = [6, 4, 5]$. Es werden mit dem Entscheidungsbaum in Abbildung 5.11 drei Vergleiche benötigt, um diese Liste zu sortieren.

1. Der erste Vergleich liefert $a_1 > a_2$, also gibt es noch drei Möglichkeiten für a_3, nämlich kleinstes Element, zwischen den beiden anderen oder größtes Element.
2. Der zweite Vergleich liefert $a_1 > a_3$, somit ist klar, dass a_1 das größte Element ist. Unklar ist aber, in welcher Reihenfolge sich a_2 und a_3 befinden müssen.
3. Diese Entscheidung liefert der letzte Vergleich mit $a_2 \leq a_3$. Also ist die korrekt sortierte Folge $[a_2, a_3, a_1] = [4, 5, 6]$.

Für die Sortierung von $[a_1, a_2, a_3] = [7, 8, 9]$ werden mit dem Entscheidungsbaum in Abbildung 5.11 lediglich zwei Vergleiche benötigt.

Theorem 5.6.1. *Jedes Sortierverfahren, das Vergleiche verwendet, benötigt im Worst Case wenigstens* $\Omega(n \lg n)$ *Vergleiche.*

Beweis. Ein optimaler Entscheidungsbaum zum Sortieren von n Elementen muss $n!$ Blätter besitzen, denn es muss möglich sein, alle möglichen $n!$ Permutationen der n Elemente mit Hilfe der Vergleiche des Baumes zu sortieren. Wir machen hierbei keine Aussage über die Reihenfolge der Vergleiche, also die Struktur des Baumes, das ist für den Beweis nicht relevant. Die Höhe des Baumes (Anzahl Kanten) gibt die Anzahl der minimal notwendigen Vergleiche an. Diese bezeichnen wir mit $v(n)$.

Nun besitzt ein vollständiger Binärbaum der Höhe h genau 2^h Blätter.

Es ist jetzt der Binärbaum mit der kleinsten Höhe gesucht, der in den Blättern

alle $n!$ Permutationen aufnehmen kann, also

$$n! \leq 2^{v(n)} \quad \Rightarrow \quad v(n) \geq \lg(n!) \ .$$

Mit *Stirlings Formel* $n! = (\frac{n}{e})^n \sqrt{2\pi n} \left(1 + \Theta\left(\frac{1}{n}\right)\right)$ ergibt sich:

$$\begin{aligned} \lg(n!) &= \lg\left(\left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right)\right) \\ &\sim \lg\left(\left(\frac{n}{e}\right)^n \sqrt{2\pi n}\right) \ , n \to \infty \\ &= n(\lg(n) - \lg(e)) + \lg(\sqrt{2\pi n}) \ , n \to \infty \ . \end{aligned}$$

Somit erhalten wir für $n \to \infty$ insgesamt

$$v(n) \geq \lg(n!) = n\lg(n) - n\lg(e) + \lg(\sqrt{2\pi n}) = \Omega(n\lg(n))$$

und damit die untere Schranke für die Anzahl der Vergleiche beim Sortieren von n Elementen. □

Diese untere Schranke wird durch MERGESORT und HEAPSORT auch angenommen. Es ist also eine asymptotisch scharfe Schranke.

5.7 Sortieren in linearer Zeit

Zuvor haben wir gezeigt, dass Sortieren mittels vergleichsbasierten Sortierverfahren wenigstens $\Omega(n \lg n)$ Schlüssel-Vergleiche benötigt. In diesem Kapitel stellen wir den *Countingsort* vor, der in linearer Zeit sortiert. Offensichtlich kann dies dann kein vergleichsbasiertes Sortierverfahren sein. Es wird eine Annahme über die Wertemenge der Schlüssel getroffen: Die Größenordnung der Wertemenge darf höchstens so groß sein wie die Anzahl der zu sortierenden Schlüssel.

Der Countingsort sortiert durch Zählen der kleineren oder gleichgroßen Schlüssel (für jedes zu sortierende Element) in einem Hilfsfeld und anschließendes Umsortieren in ein weiteres Feld. Die Umsortierung wird realisiert, indem in dem Hilfsfeld immer die Position der Schlüssel in der Sortierung zu finden sind. Das Hilfsfeld wird mit jedem Schreiben eines Elements aktualisiert.

5.7.1 Algorithmus Countingsort

Der Countingsort 5.14 ist aus [CLR04] übernommen, wurde jedoch an unsere Konvention angepasst, dass Feldindizes immer bei 0 beginnen.

Algorithmus 5.14 COUNTINGSORT(A, B, k)

```
1: for i = 0 to k do
2:     C[i] = 0
3: for j = 0 to length(A) − 1 do
4:     C[A[j]] = C[A[j]] + 1
   // C[i] enthält nun die Anzahl der Elemente = i
5: for i = 1 to k do
6:     C[i] = C[i] + C[i − 1]
   // C[i] enthält nun die Anzahl der Elemente ≤ i
7: for j = length(A) − 1 downto 0 do
8:     B[C[A[j]] − 1] = A[j]
9:     C[A[j]] = C[A[j]] − 1
```

Mit der Eingabe

$$A = \boxed{2}\boxed{5}\boxed{3}\boxed{0}\boxed{2}\boxed{3}\boxed{0}\boxed{3}$$

ergibt sich im nach dem Schleifendurchlauf durch A:

$$C = \boxed{2}\boxed{0}\boxed{2}\boxed{3}\boxed{0}\boxed{1}$$

Es gibt also zwei Mal den Schlüssel 0, kein Mal den Schlüssel 1, ... und ein Mal den Schlüssel 5. Nach dem nächsten Schleifendurchlauf enthält C die Anzahl der Schlüssel, die kleiner oder gleich dem Index von C sind.

$$C = \boxed{2}\boxed{2}\boxed{4}\boxed{7}\boxed{7}\boxed{8}$$

Dies entspricht der jeweiligen Position der Elemente (der Schlüssel ist der Index von C) in der Sortierung. In der letzten Schleife werden die Schlüssel aus A ins Ausgabefeld B an die berechnete Position kopiert. Mit jedem kopierten Element wird die Position um eins verringert, weil ein eventuell weiteres Element mit gleichem Schlüssel links vom gerade umkopierten Element einsortiert werden muss. Am Ende (nachdem alle Elemente von A nach B kopiert worden sind) sieht C wie folgt aus:

$$C = \boxed{0}\boxed{2}\boxed{2}\boxed{4}\boxed{7}\boxed{7}$$

Eine weitere Detaillierung des Beispiels ist in [CLR04] zu finden.

5.7.2 Analyse Countingsort

Lemma 5.7.1. *Der Countingsort ist korrekt.*

Beweis. Übungsaufgabe. □

Lemma 5.7.2. *Countingsort ist im Best- und im Worst-Case in* $\Theta(n)$*, falls* $(\exists k)(\forall i \in [0 : n-1])(0 \leq A[i] \leq k)$ *und* $k \in \mathcal{O}(n)$*.*

Beweis. Offensichtlich. Bei Betrachtung der Anzahl der Schleifendurchläufe sieht man sofort, dass Beiträge in $\Theta(n)$ anfallen, jedoch keine Beiträge größer als $\mathcal{O}(n)$, sofern $k \in \mathcal{O}(n)$. □

Die Schlüssel müssen sich nicht notwendigerweise im Intervall $[0 : k]$ befinden, auch $[l : l+k]$ ist möglich, solange $k \in \mathcal{O}(n)$. Natürlich muss der Algorithmus dann angepasst werden.

Darüber hinaus ist festzustellen: Countingsort ist ein stabiles Sortierverfahren. Dies ergibt sich unmittelbar aus dem Umkopieren der Elemente von A nach B und mit der Veränderung der möglichen nächsten Position für ein Element mit gegebenem Schlüssel (gespeichert im Feld C).

Der Einsatzbereich von Countingsort ist jedoch stark eingeschränkt durch die Voraussetzung, dass die Wertemenge der Schlüssel sich in einem bestimmten Bereich befinden muss, der von der Größenordnung der Anzahl der zu sortierenden Elemente entspricht. Dies ist bei den meisten Problemen in der Praxis nicht gegeben. Meist ist die Wertemenge der Schlüssel deutlich größer als die Anzahl der der zu sortierenden Elemente.

5.7.3 Algorithmus Radixsort

Der Countingsort ist nicht unbedingt geeignet, um größere Zahlen zu sortieren, weil die Wertemenge der Schlüssel üblicherweise sehr groß ist. Somit wäre viel Speicher erforderlich und damit die Laufzeit auch in dieser inakzeptablen Größenordnung.

Der Radixsort 5.15 [CLR04] ist ein Algorithmus, der zur Sortierung von Lochkarten verwendet wurde. Er löst dieses Problem. Die Idee des Radixsort ist einfach: Es wird nur immer eine Stelle des Schlüssels sortiert. Die intuitive Herangehensweise wäre die Sortierung der ersten Stelle, dann rekursiv die weiteren Stellen. In diesem Fall müsste man aber in jeder Rekursionsebene immer wieder viele Teilintervalle merken. Dies würde bei Lochkarten vielen Stapeln entsprechen, die man in der richtigen Reihenfolge halten muss. Dies war früher nicht akzeptabel, obwohl es für einen Computer heute kein Problem wäre. Statt der intuitiven Reihenfolge sortiert der Radixsort die Schlüssel beginnend von hinten.

Wir geben ein einfaches Beispiel an, bei dem dreistellige Zahlen auf diese Weise sortiert werden. Zuerst wird nach der Einerstelle, dann der Zehnerstelle und abschließend nach der Hunderterstelle sortiert. In der Abbilding 5.12 sind die bereits sortierten Teile fett gedruckt.

839		72**0**		7**20**		**329**
657		35**5**		3**29**		**355**
457		43**6**		4**36**		**436**
329	→	65**7**	→	8**39**	→	**457**
436		45**7**		3**55**		**657**
720		83**9**		6**57**		**720**
355		32**9**		4**57**		**839**

Abbildung 5.12: Ablauf des Radixsort

So ergibt sich der folgende einfache Algorithmus, der auf ein stabiles Sortierverfahren zurückgreifen muss, weil sonst die bereits vorhandene Teilsortierung in weiteren Durchläufen wieder zerstört würde. Hier kann zum Beispiel der Countingsort verwendet werden. d ist die Anzahl der zu sortierenden Stellen, wobei d die höchstwertige Stelle und 1 die niederwertigste Stelle im Schlüssel bezeichnet.

Algorithmus 5.15 RADIXSORT(A, d)

```
1: for i = 1 to d do
2:     Sortiere nach der i-ten Stelle mit einem stabilen Sortierverfahren
```

Wir sind bei Algorithmus 5.15 von einer festen Länge der Schlüssel ausgegangen. Für den Radixsort ist es aber notwendig, dass alle Schlüssel die gleiche Länge besitzen. Dies ist in der Praxis gegebenenfalls nicht erfüllt. Sofern dies nicht der Fall ist, muss die gleiche Länge hergestellt werden. Zahlen müssen natürlich links mit Nullen aufgefüllt werden, um die natürliche Sortierung zu erhalten. Zeichenketten müssen rechts mit Zeichen aufgefüllt werden, die kleiner als das kleinste vorhandene Zeichen sind, um die natürliche lexikographische Ordnung zu erhalten. Beim Einsatz des Radixsorts in der Praxis muss dies berücksichtigt werden.

5.7.4 Analyse Radixsort

Lemma 5.7.3. *Der Radixsort ist korrekt.*

Beweis. Übungsaufgabe. □

Lemma 5.7.4. *Falls die n zu sortierenden Elemente d Stellen besitzen und jede Stelle k verschiedene Werte annehmen kann, dann ist die Laufzeit von Radixsort in $\Theta(d(n+k))$, sofern der stabile Sortieralgorithmus in $\Theta(n+k)$ läuft. Für Countingsort ist diese Bedingung erfüllt.*

Beweis. Übungsaufgabe. □

5.8 Abschließende Bemerkungen

Sie haben nun verschiedene Sortieralgorithmen kennen gelernt. Auch wenn wir nicht alle Algorithmen im Detail analysiert haben, sollten Sie einen Eindruck gewonnen haben, welcher Algorithmus für welches Einsatzgebiet sinnvoll ist. Die Sortieralgorithmen sind alle sehr gut untersucht. Details sind beispielsweise in [CLR04] und [Knu98] zu finden. Der für die Praxis wichtige Average-Case ist meist am schwierigsten zu berechnen. Er hängt von der Verteilung der Eingaben ab, wir sind (wie sonst auch üblich) von einer Gleichverteilung ausgegangen.

BUBBLESORT ist zum Beispiel ein sehr einfach zu implementierender Sortieralgorithmus, aber *kein* effizienter. BUBBLESORT ist sogar schlechter als INSERTION-SORT (vergleiche Übungsaufgabe)! Dass BUBBLESORT so bekannt ist, dürfte an seinem lustigen Namen liegen – es gibt *keinen* Grund, BUBBLESORT in der Praxis einzusetzen. Ein noch schlechterer Algorithmus ist übrigens BOGOSORT (`http://de.wikipedia.org/wiki/Bogosort`) mit mittlerer Laufzeit $\Theta(n\,n!)$ und sogar Worst-Case $T(n) = \infty$.

Üblicherweise fährt man mit bekannten und erprobten Sortieralgorithmen sehr gut. Sollten Sie ganz spezielle Anforderungen an ein Sortierverfahren haben, muss genau überlegt werden, ob ein erprobtes Verfahren angepasst wird. Hier ist auf jeden Fall ein sehr gutes Testen erforderlich, insbesondere für Randbedingungen wie leere Listen oder Abbruchkriterien für Rekursionen.

Im folgenden Kapitel stellen wir Ihnen einfache und auch komplexere Datenstrukturen vor und darauf basierende Algorithmen. Wenn Sie die Lösung zu einem Problem finden müssen, dann liegt es an Ihnen, die geeigneten Algorithmen und Datenstrukturen zu finden. Unter anderem kann es erforderlich sein, eine einfache Datenstruktur zu erweitern, um Algorithmen effizient zu machen.

Solche Algorithmen-Designs sind definitiv fortgeschrittene Themen aus dem Bereich Software-Engineering, die den Rahmen des Skripts sprengen würden.

5.9 Fragen und Aufgaben zum Selbststudium

1. Sortieren Sie das folgende Feld $A = (0, 6, 4, 13, 12, 7, 1, 10, 23, 13)$ „per Hand“ mit Hilfe von MERGESORT!
2. Sortieren Sie das folgende Feld $A = (0, 6, 4, 13, 12, 7, 1, 10, 23, 13)$ „per Hand“ mit Hilfe von QUICKSORT!
3. Sortieren Sie das folgende Feld $A = (0, 6, 4, 13, 12, 7, 1, 10, 23, 13)$ „per

Hand“ mit Hilfe von HEAPSORT!

4. Sortieren Sie das folgende Feld $A = (0, 6, 4, 13, 12, 7, 1, 10, 23, 13)$ „per Hand“ mit Hilfe von BUBBLESORT!

5. Sortieren Sie das folgende Feld $A = (0, 6, 4, 3, 2, 7, 1, 0, 2, 1, 2, 7, 3)$ „per Hand“ mit Hilfe von COUNTINGSORT!

6. Was passiert bei MERGESORT, wenn das Feld bereits sortiert ist?

7. Wann ist QUICKSORT optimal, wann schlecht?

8. Wie wird das Feld durch PARTITION bei QUICKSORT aufgeteilt, wenn das Feld aus lauter *identischen* Einträgen besteht?

9. Modifizieren Sie QUICKSORT so, dass in *absteigender* Reihenfolge sortiert wird!

10. Zeigen Sie, dass QUICKSORT eine Laufzeit von $\mathcal{O}(n^2)$ hat, wenn alle Elemente unterschiedlich und bereits sortiert sind!

11. Wieviele Elemente gibt es in einem HEAP der Höhe h mindestens und wieviele Elemente höchstens?

12. Kreditkartenzahlungen, die von vielen verschiedenen über das Land verstreuten Kartenterminals eingehen, sollen nach dem Transaktionsdatum sortiert werden. Die eingehenden Zahlungen von verschiedenen Terminals sind meistens bereits vorsortiert, da in den Kartenterminals ebenfalls nach Transaktionsdatum sortiert wird. Daher sind die zu sortierenden Daten „fast“ sortiert. Begründen Sie, warum hier vermutlich INSERTION-SORT gegenüber QUICKSORT die bessere Wahl darstellen könnte!

13. Zeigen Sie, dass ein Binärbaum mit n inneren Knoten genau $n + 1$ Blätter enthält, $n \geq 0$.

14. Zeigen Sie, dass in einem Unterbaum eines MAX-HEAPs die Wurzel des Unterbaums das größte Element des Unterbaums enthält!

15. Muss ein MAXHEAP A – als Feld interpretiert – schon sortiert sein? Geben Sie ggf. ein Beispiel an!

16. Ein Feld A sei absteigend sortiert. Ist dieses Feld ein MAXHEAP? Begründen Sie Ihre Antwort!

17. Illustrieren Sie durch Zeichnen der entsprechenden Bäume die Arbeitsweise von BUILD-MAX-HEAP angewendet auf das Feld

$$(5, 3, 17, 10, 84, 19, 6, 22, 9)!$$

18. Schreiben Sie eine iterative Version von MAX-HEAPIFY, d.h. sorgen Sie dafür, dass der rekursive Aufruf in Zeile 11 durch eine Schleifenkonstruktion ersetzt wird! (Hinweis: Führen Sie eine boolsche Variable *finished* ein und lassen Sie den ganzen Algorithmus in einer Schleife `while finished != true` laufen und setzen Sie dann an geeigneter Stelle finished auf `false`.)

19. Berechnen Sie die Komplexitätsklasse von MAX-HEAPIFY in der iterativen Version und vergleichen Sie mit der ursprünglichen, rekursiven!

20. Warum ist BUBBLESORT *kein* gutes Sortierverfahren?

21. Zeigen Sie, dass BUBBLESORT in $\mathcal{O}(n^2)$ ist! (Hinweis: Verfahren Sie analog zur Argumentation für INSERTION-SORT.)

Kapitel 6

Lineare Datenstrukturen

6.1 Der Begriff der Datenstruktur

Zur Erinnerung: In Kapitel 2 haben wir *abstrakte Datentypen* (ADTs) definiert als Datentypen zusammen mit einer inhaltlichen (semantischen) Interpretation der Objekt- und Operationsmengen. Außerdem haben wir *konkrete Datentypen* als abstrakte Datentypen definiert, bei denen die Objekt- und Operationsmengen konkret einschließlich des Wertebereichs bestimmt sind. In diesem und dem nächsten Kapitel werden wir einige wichtige ADTs einführen und untersuchen. Wie wir in Kapitel 1 gesehen haben, kann die richtige Wahl des Datentyps entscheidend sein für Eigenschaften eines Algorithmus sein. So konnten wir in Kapitel 1 durch Wahl eines „intelligenten“ Datentyps, in diesem Fall einer Hashtabelle (s.u.), die Geschwindigkeit der Häufigkeitsanalyse von Wörtern gegenüber der naiven Version (Felder) wesentlich steigern.

Ein weiteres Beispiel liefern Suchverfahren. BINARY-SEARCH aus Kapitel 1 ist in $\mathcal{O}(\lg n)$ während LINEAR-SEARCH in $\Theta(n)$ ist. Der Unterschied ist auch hier die unterliegende Datenstruktur.

> Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.
>
> Fred Brooks [Bro75]

> Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.
>
> [...] Smart data structures and dumb code works a lot better than the other way around.
>
> Eric Raymond [Ray01]

Objektorientierte Programmiersprachen unterstützen durch das Typenkonzept

die Erstellung von ADTs, da hier Daten und Operationen an Typen[1] gebunden werden, die Daten geschützt werden können (private Attribute) und die zulässigen Operationen (Methoden) festgelegt werden können. Einige prozedurale Programmiersprachen wie Ada oder Modula-2 (Pascal) unterstützen ebenfalls gezielt die Erstellung von ADTs. Bei den meisten Sprachen lässt sich die Semantik nur informell als Kommentartext beschreiben.

6.2 Dynamische Datenstrukturen

Wir haben bisher *Felder* kennengelernt, d.h. Datenstrukturen mit *fester* Länge. Für viele Zwecke ist dies unzureichend, da ein Algorithmus auf unterschiedlich großen, oft zur Kompilierzeit unbekannten Datenmengen operieren muss[2]. Datenstrukturen, die mit unterschiedlichen Eingabegrößen zurecht kommen und daher *veränderbar* sein müssen, werden *dynamisch* genannt.

In diesem Kapitel werden verschiedene lineare Datenstrukturen behandelt, die allesamt dynamisch sind, u.a.

1. Verkette Listen
2. Stapel
3. Warteschlangen
4. Hashtabellen

Während viele dynamische[3] Programmiersprachen Konstrukte für dynamische Datenstrukturen und Mengen besitzen, müssen diese insbesondere für statisch typisierte, kompilierte Sprachen wie C durch Bibliotheken bereitgestellt werden. Bei Sprachen wie C, die keine automatische Speicherverwaltung (Garbage Collector) besitzen, muss sorgfältig auf ebendiese geachtet werden. Üblicherweise wird dazu explizites Zeiger- (Pointer-) und Memorymanagement verwendet.

Die Objekte, die in dynamischen Datenstrukturen gespeichert werden, weisen häufig einen *Schlüssel* und *Satelliten-*, d.h. Nutzdaten auf. Für dynamische Datenstrukturen sind die folgenden Operationen nützlich:

- SEARCH(S, k): Suche für eine Menge S den Schlüssel k (“key”).

[1] in der Regel die Klassen der Sprache – eine leider für OO-Sprachen typische Vermischung des Typkonzepts mit dem Implementierungskonzept

[2] Wir haben dies z. B. beim Eingangsbeispiel des Wörterzählens gesehen, bei dem bei der naiven Lösung einerseits eine große Speicherverschwendung durch feste Felder vorkam, andererseits die Korrektheit bei großen Textmengen *nicht* garantiert werden konnte.

[3] oder funktionale!

- INSERT(S, x): Einfügen des Objektes, auf das x zeigt, in eine Menge S.
- DELETE(S, x): Löschen des Objektes, auf das x zeigt.
- MINIMUM(S): Finde das kleinste Element (wenn das Sinn ergibt, d.h. die Elemente sich ordnen lassen).
- MAXIMUM(S): Finde das größte Element (wenn das Sinn ergibt, d.h. die Elemente sich ordnen lassen).
- SUCCESSOR(S, x): Gib den Zeiger auf das nächste Element („hinter“ x) zurück.
- PREDECESSOR(S, x): Gib den Zeiger auf das vorherige Element („vor“ x) zurück.

6.3 Verkettete Listen

Eine verkettete Liste ist eine Datenstruktur in der die Objekte in linearer Reihenfolge angeordnet sind. Die Anordnung entspricht meist der Reihenfolge des Einfügens und steht nicht notwendiger Weise in einer Beziehung zu einer möglichen Ordnungsstruktur der Objekte der Liste. Abbildung 6.1 verdeutlicht eine solche Struktur (NIL bezeichnet hier und im folgenden einen Zeiger auf „Nichts“).

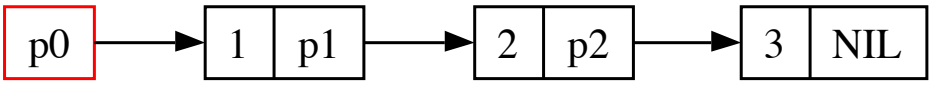


Abbildung 6.1: Verkettete Liste

6.3.1 Suchen in Verketteten Listen

Der Suchalgorithmus für Verkettete Listen ist in 6.1 ausgeführt.

Algorithmus 6.1 LIST-SEARCH(L, k)

```
1: x ← L.head
2: while x ≠ NIL and key[x] ≠ k do
3:       x ← x.next
4: return x
```

Wie man leicht sieht, ist LIST-SEARCH im Worst-Case in $\Theta(n)$, wobei n die Länge der Liste bezeichnet.

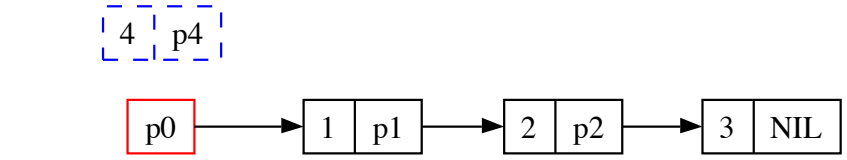
6.3.2 Einfügen in Verketteten Listen

Der Einfügealgorithmus für Verkettete Listen ist in 6.2 ausgeführt.

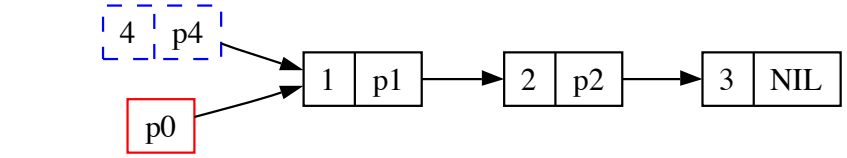
Algorithmus 6.2 LIST-INSERT(L, x)

1: $x.next \leftarrow L.head$
2: $L.head \leftarrow x$

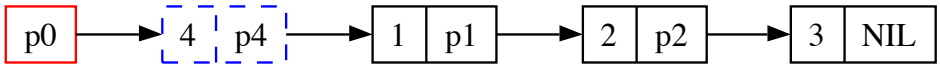
Wie man leicht sieht, ist LIST-INSERT in $\mathcal{O}(1)$. Abbildung 6.2 veranschaulicht das Einfügen eines Elements in eine verkettete Liste.



(a) Schritt Eins



(b) Schritt Zwei



(c) Schritt Drei

Abbildung 6.2: Einfügen in eine Verkette Liste

6.3.3 Löschen aus Verketteten Listen

Übungsaufgabe!

6.3.4 Doppelt Verkettete Listen

Manchmal sind doppelt verkettete Listen nützlich. Sie erfordern zwei Zeiger: *previous* und *next* wie Abbildung 6.3 verdeutlicht.

Der Einfügealgorithmus für Doppelt Verkettete Listen ist in 6.3 ausgeführt, falls das neue Element am Anfang der Liste eingefügt wird.

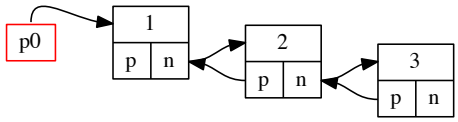


Abbildung 6.3: Doppelt Verkettete Liste

Algorithmus 6.3 DL-LIST-INSERT(L, x)

1: $x.next \leftarrow L.head$
2: **if** $L.head \neq NIL$ **then**
3: $\quad L.head.prev \leftarrow x$
4: $L.head \leftarrow x$
5: $x.prev \leftarrow NIL$

6.3.5 Löschen aus Doppelt Verketteten Listen

Übungsaufgabe!

6.3.6 Exkurs: Verkettete Listen in C

In C lassen sich Verkettete Listen leicht mit Hilfe von Zeigern und Strukturen implementieren, wie das Codefragment in Listing 6.1 und 6.2 zeigt.

Listing 6.1: Verkettete Liste – C-Code Header

```
1  typedef struct Item Item;
2
3  struct Item{
4      char *value;
5      struct Item *next;
6  };
7
8  Item *newItem(char *word);
9  Item *addfront(Item *listp, Item *newp);
10 Item *addend(Item *listp, Item *newp);
11 Item *deletefirst(Item *listp, char *word);
12 Item *lookup(Item *listp, char *word);
13 void apply(Item *listp, void (*fn)(Item*, void*  ), void * arg);
```

Listing 6.2: Verkettete Liste – C-Code

```
1  Item *newItem(char *word){
2      Item *itemp;
3      itemp = (Item *) malloc(sizeof(Item));
4      itemp->value = word;
5      itemp->next=NULL;
6      return itemp;
```

```
 7  }
 8
 9  Item *addfront(Item *listp, Item *newp){
10      newp->next = listp;
11      return newp;
12  }
13
14  Item *addend(Item *listp, Item *newp){
15      Item *p;
16      if(listp == NULL)
17          return newp;
18      for (p = listp; p->next != NULL; p = p->next)
19          ;
20      p->next = newp;
21      return listp;
22  }
```

Wie man sieht muss der Aufrufende (englisch "Caller") den Listenkopf explizit verwalten – die Abstraktion ist hier nur unvollkommen. Eine bessere Abstraktion kann man in C++ mit OO-Techniken erzielen.

In der Praxis ist es notwendig, den Speicher dynamisch zu verwalten, da das Erzeugen neuer Listenelemente einerseits Speicher erfordert und andererseits beim Löschen oder Entfernen von Elementen aus der Liste Speicher wieder freizugeben ist. Sprachen *ohne* automatische Speicherverwaltung (englisch "Garbage Collection") erfordern ein manuelles Verwalten. Ein Beispiel für diese Speicherverwaltung ist im C-Listing 6.3 gegeben. Bei einer manuellen Verwaltung ist es wichtig, festzulegen, wer die Verantwortung für die Verwaltung hat – der Aufrufende der Bibliothek oder die Bibliothek selbst. In unserem Beispiel erzeugt die Bibliothek selbst den benötigten Platz beim Aufruf `newItem` und gibt ihn bei der Operation `deletefirst` etwa wieder frei. Dies unterstützt die Vermeidung von Speicherlecks, indem die Speicherverwaltung im Datentyp selbst und damit für den Benutzer transparent[4] vorgenommen wird.

6.3.7 Vergleich Verkettete Listen und Felder

Für ein gegebenes Problem können häufig Verkettete Listen *oder* Felder verwendet werden. Die Entscheidung erfordert Erfahrung im Software-Engineering, ist jedoch nicht willkürlich. Die nachstehenden Fakten mögen dabei helfen. Siehe dazu auch Tabelle 6.1.

Die Vorteile von Verketteten Listen gegenüber Feldern:

- maximale Größe der Liste muss nicht im voraus bekannt sein, dadurch ggf. geringerer Speicherbedarf

[4] d.h ohne die Notwendigkeit, selbst dafür sorgen zu müssen.

Listing 6.3: Speicherverwaltung

```
1  Item *deletefirst(Item *listp, char *word)
2  {
3      Item *p, *prev;
4      prev = NULL;
5
6      for (p =listp; p != NULL; p = p->next) {
7          if(strcmp(p->value, word) == 0) {
8              if(prev == NULL)
9                  listp = p->next;
10             else {
11                 prev->next = p->next;
12             }
13             free(p);
14             return listp;
15         }
16         prev = p;
17     }
18     printf("%s", "list element not found!");
19     return NULL;
20 }
```

- hohe Flexibilität hinsichtlich der Datenstrukturen (die Elemente können z. B. von unterschiedlichem Typ sein)
- maximale Größe nur von der Speichergröße abhängig

Die Nachteile von Verketteten Listen gegenüber Feldern:

- kein wahlfreier Zugriff auf das n-te Element
- unterschiedliche Zugriffszeiten auf verschiedene Elemente (erstes vs. letztes)
- erhöhter Speicherbedarf, da nicht nur die Nutzdaten, sondern auch die Zeiger abgelegt werden müssen

Die nachstehende Tabelle 6.1 vergleicht die Komplexität von Operationen bei Feldern und Verketteten Listen.

6.4 Stapel

Stapel und *Warteschlangen* (englisch "*Stacks*" und "*Queues*") sind wichtige dynamische Datenstrukturen, die sich mit Listen und Feldern implementieren lassen. Üblich sind aus Effizienzgründen Implementierungen mit Feldern, die allerdings „Stapelüberlauf"-Ausnahmen auslösen können, wenn zu viele Objekte auf den Stapel gepackt werden. Ein Stapel hat das folgende API:

Tabelle 6.1: Vergleich Felder und Listen

Operation	Verk. Liste	Feld	Bemerkung
Insertion Anfang	$\Theta(1)$	$\Theta(n)$	Feld $\Theta(n)$, da Elemente (nach hinten) verschoben werden müssen, bevor eingefügt wird.
Deletion Anfang	$\Theta(1)$	$\Theta(n)$	Feld $\Theta(n)$, da Elemente (nach vorn) verschoben werden müssen.
Insertion Ende	$\Theta(n)$	$\Theta(1)$	Feld $\Theta(1)$, nur dann wenn „Ende“ bekannt ist!
Deletion Ende	$\Theta(n)$	$\Theta(1)$	Feld $\Theta(1)$, nur dann wenn „Ende“ bekannt ist!
Lookup	$\Theta(n)$	$\Theta(n)$	
Index i	$\Theta(i)$	$\Theta(1)$	
Verschwendeter Platz	$\Theta(n)$	$\Theta(n)$	Listen verschwenden Platz für Zeiger, Felder für nicht verwendete Elemente

- PUSH(S, x): Legt das Objekt, auf das x zeigt auf dem Stapel ab. Erzeugt eine Ausnahme, wenn der Stapel voll ist.
- POP(S): Nimmt ein Objekt vom Stapel. Erzeugt Ausnahme, wenn der Stapel leer ist.

Ein Stapel ist also ein *LIFO* Last-In/First-Out Speicher. Ein Stapel kann leicht mit Hilfe eines Feldes implementiert werden. Angenommen, das Feld enthält die Elemente $(15, 1, 5, 9)$ und man legt (“pusht”) 17 und 3 auf den Stapel und entnimmt danach 3. Die folgende Abbildung 6.4 verdeutlicht diese Ereignisse.

Der Pseudocode für PUSH und POP ist in 6.4 bzw. 6.5 ausgeführt.

Algorithmus 6.4 PUSH(S, x)

1: $S.top \leftarrow S.top + 1$
2: $S[S.top] \leftarrow x$

Dabei wird die Hilfsprozedur EMPTY 6.6 verwendet.

Produktivcode für PUSH muss auf alle Fälle noch auf StackOverflow abprüfen (und ggf. eine Fehlerbehandlung durchführen), siehe Algorithmus 6.7, wobei MAX die maximale Stapelgröße kennzeichnet, die üblicher Weise durch die Länge des vorab allozierten Feldes gegeben ist. Die PUSH und POP-

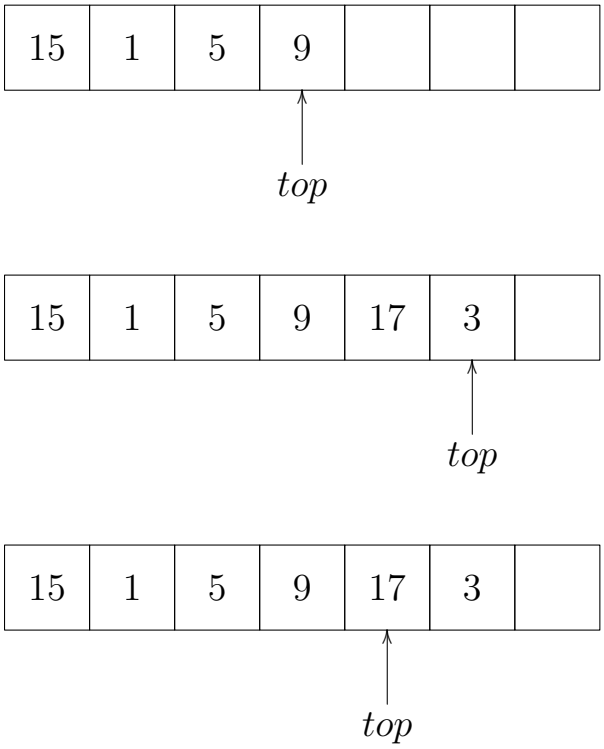


Abbildung 6.4: Stapel

Algorithmus 6.5 POP(S)

```
1: if EMPTY(S) then
2:     return "ERROR: underflow"
3: else
4:     S.top ← S.top − 1
5:     return S[S.top + 1]
```

Algorithmus 6.6 EMPTY(S)

```
1: if S.top = −1 then
2:     return true
3: else
4:     return false
```

Operationen auf Stapeln sind in $\mathcal{O}(1)$.

Der Vorteil von Abstrakten Datentypen ist es, dass die Implementierung ausgetauscht werden kann, ohne dass sich abhängiger Clientcode ändert. So können wir z. B. in obiger Implementierung leicht das Feld durch eine verkette

Algorithmus 6.7 PUSH-PROD(S, x)

```
1: if S.top ≥ MAX − 1 then
2:     return ERROR: Stack overflow
3: else
4:     S.top ← S.top + 1
5:     S[S.top] ← x
```

Liste austauschen. Dadurch bekommt unser Stapel „unbegrenzt“ viel Platz. Der Preis, den wir dafür zahlen ist eine geringfügig schlechtere Performanz, da das Einfügen in eine Liste mehr Zeit erfordert als das Speichern in einem Feld. Der Pseudocode ist in 6.8 und 6.9 verdeutlicht.

Algorithmus 6.8 PUSH(L, x)

```
1: LIST-INSERT(L, x)
```

Algorithmus 6.9 POP(L)

```
1: item ← L.head
2: LIST-REMOVE(L, L.head)
3: return item
```

(Hierbei ist allerdings sorgfältig darauf zu achten, dass der Speicher für *item* bei der `pop` Operation *nicht* freigegeben wird, da der Aufrufer *item* in der Regel benutzen möchte. Die oben angegeben C-Implementierung wäre also zu modifizieren. Hier wird deutlich, welche Koordinierungsprobleme bei der manuellen Speicherverwaltung auftreten können. In der Praxis muss stets sichergestellt werden, dass der Speicher wieder freigegeben wird – entweder durch sorgfältige Programmierung (und entsprechende Sprachmuster und Software-Engineering Disziplin) oder durch eine geeignete (semi-) automatisierte Speicherverwaltung.

6.5 Warteschlangen

6.5.1 Grundlagen

Eine Warteschlange ist ein *FIFO* First-In/First-Out Speicher. Ein Einfügen wird als ENQUEUING, ein Entnehmen als DEQUEUING bezeichnet. Abbildung 6.5 verdeutlicht die Funktion von Warteschlangen, TAIL bezeichnet dabei das Ende, HEAD den Anfang der Warteschlange.

1. Eine Warteschlange ist mit $(15, 6, 9, 8, 4)$ gefüllt
2. Es werden 17, 3 und 5 eingestellt. Man beachte dass TAIL zyklisch um das Feld läuft.

3. Es wird 15 entnommen.

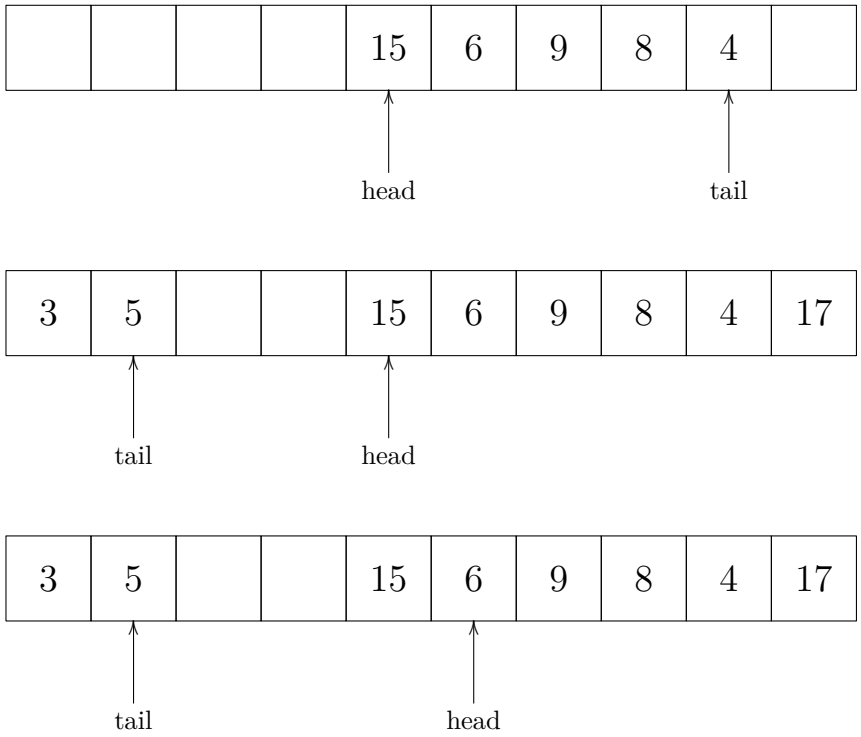


Abbildung 6.5: Warteschlange

Operationen auf Warteschlangen sind in $\mathcal{O}(1)$.

6.5.2 Prioritätswarteschlangen

Eine Prioritätswarteschlange oder Vorrangwarteschlange ist eine Datenstruktur für die Speicherung von Elementen x, die einen Schlüssel k (die Priorität) zugeordnet haben und im Falle einer Max-Prioritätswarteschlange folgende Eigenschaften erfüllen (Min-Prioritätswarteschlangen analog):

1. INSERT(S, x): fügt ein Element x in die Menge S ein, Notation $S \leftarrow S \cup \{x\}$.
2. MAXIMUM(S): gibt das[5] maximale Element, d.h. das Element mit dem größten Schlüssel zurück.
3. EXTRACT $-$ MAX(S): entfernt das[6] maximale Element.

[5]Im Falle von mehreren Elementen mit dem maximalen Schlüssel (d.h. gleichen Schlüssel) wird eines zurückgegeben (ggf. in FIFO Abfolge).

[6]Im Falle von mehreren Elementen mit dem maximalen Schlüssel (d.h. gleichen Schlüssel)

Abbildung 6.6: Warteschlange, Quelle: `http://en.wikipedia.org/wiki/File:Northern_Rock_Queue.jpg`

4. INCREASE − KEY(S, x, k): erhöht den Wert des Schlüssels von x auf k (Annahme: k ist mindestens so groß wie der aktuelle Wert.)

Max-Prioritätswarteschlangen sind von großer Bedeutung für Job-Scheduling Anwendungen. Min-Prioritätswarteschlangen finden Anwendung bei Simulationen (s. dazu auch [CLR04]) oder bei Bäumen (s.u.).

Wir können Max- und Min-Prioritätswarteschlangen mit einem Heap implementieren, wie in Abbildung 6.7 gezeigt.

Bemerkung: Im folgenden wird aus Gründen der Einfachheit die Unterscheidung zwischen den Elementen x und ihren Schlüsseln k aufgehoben und die Elemente mit ihren Schlüsseln identifiziert, mit anderen Worten es werden nur (ganze) Zahlen als Elemente betrachtet, die zugleich die Priorität kennzeichnen. In der Praxis unterscheidet man natürlich zwischen dem Element x und seinem Schlüssel k. So kann das Element x z. B. einen Unix-Prozess mit Prozess-Id PID kennzeichnen, und der Schlüssel ist dann z. B. die Prozesspriorität. (Die Prioritäten von Prozessen können z. B. mit dem Unix Kommando `nice` geändert werden.)

Die nachstehenden Algorithmen 6.10, 6.11, 6.12, 6.13 und 6.14 implementieren das Protokoll einer Max-Prioritätswarteschlange.

Man mache sich insbesondere die Funktionsweise von HEAP-INCREASE-KEY und MAX-HEAP-INSERT anhand einer Zeichnung (Heap!) klar (Übung)!

wird eines entfernt (ggf. in FIFO Abfolge).

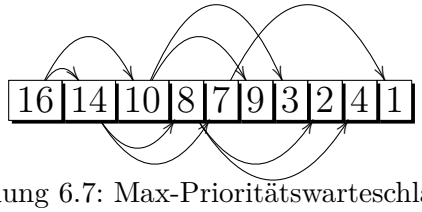


Abbildung 6.7: Max-Prioritätswarteschlange

Algorithmus 6.10 HEAP-MAX(A)

1: **if** $A.heapsize < 1$ **then**
2: **return** ERROR: "Heap underflow"
3: **return** $A[0]$

Algorithmus 6.11 HEAP-EXTRACT-MAX(A)

1: **if** $A.heapsize < 1$ **then**
2: **return** ERROR: "Heap underflow"
3: $max \leftarrow A[0]$
4: $A.heapsize \leftarrow A.heapsize - 1$ // decrease before since
5: $A[0] \leftarrow A[A.heapsize]$ // heapsize $\in [1, \ldots, n]$
6: MAX-HEAPIFY($A, 0$)
7: **return** max

Algorithmus 6.12 PARENT(A, i)

1: **if** $i \leq 0$ **or** $i \geq A.heapsize$ **then**
2: **return** ERROR: "No parent"
3: **else**
4: **return** $(i-1)/2$ // Floor function

Algorithmus 6.13 HEAP-INCREASE-KEY(A, i, key)

1: **if** $i < 0$ **or** $i \geq A.heapsize$ **then**
2: **return** ERROR: "Index out of range"
3: **if** $key < A[i]$ **then**
4: **return** ERROR: "key must be bigger than current key"
5: $A[i] \leftarrow key$
6: **while** $i > 0$ **and** $A[PARENT(i)] < A[i]$ **do**
7: $A[i], A[PARENT(i)] \leftarrow A[PARENT(i)], A[i]$
8: $i \leftarrow PARENT(i)$ // "Bubbling Up"

Eine Implementierung einer Min Priority Queue mit einem Heap ist im Listing 6.4 veranschaulicht.

Listing 6.4: Min Priority Queue mit Heap (Ausführbarer Pseudocode)

```
1 import sys
2 INF = sys.maxsize # TODO should do better
```

Algorithmus 6.14 MAX-HEAP-INSERT(A, key)

1: **if** $A.heapsize \geq MAX\_HEAPSIZE - 1$ **then**
2: **return** ERROR: "Heap overflow"
3: $A[A.heapsize] \leftarrow -\infty$ // "Insert minimal key"
4: HEAP-INCREASE-KEY($A, A.heapsize, key$)
5: $A.heapsize \leftarrow A.heapsize + 1$

```
3
4  def parent(i):
5      return (i-1)//2
6
7  def left(i):
8      return 2*i+1 # lists start with 0!
9
10 def right(i):
11     return 2*i+2 # lists start with 0!
12
13 def heapify(a, heapsize, i):
14     l = left(i)
15     r = right(i)
16     if l < heapsize and a[l] < a[i]:
17         minimum = l
18     else:
19         minimum = i
20     if r < heapsize and a[r] < a[minimum]:
21         minimum = r
22     if minimum != i:
23         a[i], a[minimum] = a[minimum], a[i]
24         heapify(a, heapsize, minimum)
25
26 def build_heap(a):
27     heap_size = len(a)
28     for i in range(len(a)//2, 0, -1):
29         heapify(a, heap_size, i-1)
30
31 def min_heap(a):
32     return a[0]
33
34 def extract_min(a):
35     if len(a) == 0:
36         return None
37     minimum = a[0]
38     del a[0]
39     build_heap(a)
40     return minimum
41
42 def insert_heap(a, x):
```

```
43      a.append(x)
44      decrease_key_heap(a, len(a)-1, x)
45
46  def decrease_key_heap(a, i, k):
47      if k > a[i]:
48          raise Exception("new key bigger than current key")
49      a[i] = k
50      while i > 0 and a[parent(i)] > a[i]:
51          a[parent(i)], a[i] = a[i], a[parent(i)]
52          i = parent(i)
53
54  # Tests with intermediate output:
55  lists = [[8,1,3,16,9], [5,2,7,4,6,1,3], [1,2,3,4],
56     [5,4,3,2,1], [8,9,0,6,4,7,1,3,2,5] ]
57  for l in lists:
58      print("To be sorted:", l)
59      build_heap(l)
60      print("After build_heap:", l)
61      insert_heap(l, 7)
62      print("After insert_heap(7):", l)
63      print("min_heap:", min_heap(l))
64      for i in range(len(l)):
65          print(l)
66          print("min_heap:", extract_min(l))
67      print()
```

6.6 Hashtabellen

6.6.1 Motivation

Für viele Anwendungen sind Datenstrukturen nützlich, die nur die folgenden Operationen möglichst effizient unterstützen:

- INSERT,
- SEARCH und
- DELETE

Bisher haben wir Elemente entweder direkt adressiert, nämlich durch einen Index wie bei Feldern, oder gar nicht – wie bei Verketteten Listen. Letzteres hat auf jeden Fall den Nachteil, dass ein Suchen in $\mathcal{O}(n)$ ist. Aber auch bei direkter Adressierung ist das Suchen in $\mathcal{O}(n)$, wenn man den Index nicht kennt, sondern nach einem bestimmten Wert sucht. Ideal wären Lineare Datenstrukturen, die immer oder wenigstens im „Normalfall“ konstante Suchzeiten benötigen. Solche Datenstrukturen existieren und heißen Hashtabellen. Erreicht wird dies durch *indirekte Adressierung* über eine sogenannte Hashfunktion.

Direkte Adressierung ist in Abbildung 6.8 dargestellt. Die Menge möglicher Schlüsselwerte sei mit $\mathcal{K}$ bezeichnet, die Schlüssel mit k_i und deren zugeordnete Werte mit v_i.

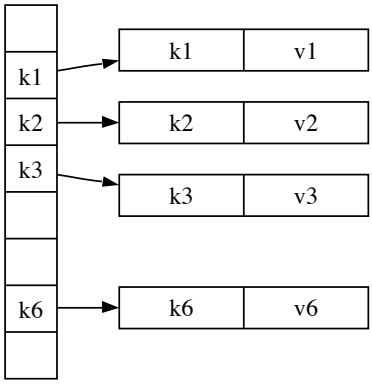


Abbildung 6.8: Hashtabelle – Direkte Adressierung

Wie man sieht, werden die Schlüssel $1:1$ (Speicher-) Plätzen in einem Feld zugeordnet. Wenn die Menge der zu Schlüssel groß ist, führt die direkte Adressierung zu einer u.U. unannehmbaren Speicherplatzverschwendung (z.B. wie beim Countingsort, Algorithmus 5.14) und ist damit oft völlig unpraktikabel. Im Gegenzug dazu verwendet man bei der indirekten Adressierung eine *Hashfunktion* $h : \mathcal{K} \to h(\mathcal{K})$. Die Grundidee der Hash-Verfahren besteht darin, aus dem zu speichernden Schlüsselwert die Adresse im Speicher zu *berechnen*, an der dieses Element gespeichert wird.

> "Any problem in computer science can be solved with another level of indirection" David Wheeler

Sei etwa $\mathcal{K}$ der Bereich, aus dem die Schlüsselwerte stammen, und sei $S \subseteq \mathcal{K}$ eine zu speichernde Menge mit $|S| = n$. (Wir bezeichnen mit $|S|$ die Anzahl der Elemente der Menge S). Zur Speicherung der Elemente von S benutzen wir eine Menge von Behältern $B_0, B_1, \ldots, B_{m-1}$. Im allgemeinen gilt $|\mathcal{K}| \gg m$ (d.h. $|\mathcal{K}|$ ist sehr viel größer als m). Infolgedessen kann es zu Kollisionen kommen, d.h. es müssen u.U. mehrere Elemente aus $\mathcal{K}$ in einem Behälter gespeichert werden. Die Abbildung 6.9 verdeutlicht dieses Verfahren ebenso, wie den notwendigen Mechanismus zum Auflösen von Kollisionen. Hier wurde eine Verkettete Liste für die Überläufer verwendet. Wir untersuchen diese Verfahren in den nächsten Abschnitten en detail.

6.6.2 Hashfunktionen

Hashfunktionen bilden eine potentiell sehr große Menge von Schlüsseln auf eine in der Regel kleinere Menge von Werten ab (Ausnahme: Perfektes

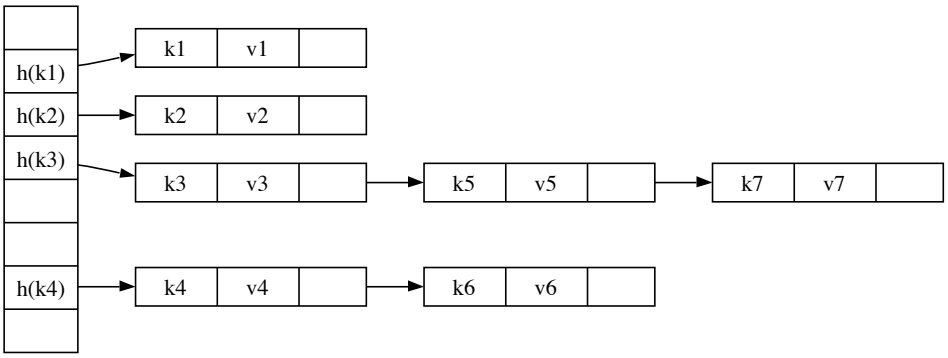


Abbildung 6.9: Hashtabelle – Indirekte Adressierung

Hashing, s.u.).

Definition 6.6.1. *Wir bezeichnen die Menge der möglichen Schlüssel mit* $\mathcal{K}$. $S \subseteq \mathcal{K}$ *ist eine zu speichernde Menge mit* $|S| = n$, *die Zielmenge wird mit* $\mathcal{Z}$ *bezeichnet und nimmt ganzzahlige Werte von* 0 *bis* $m-1$ *an, d.h.* $\mathcal{Z} := \{0, \ldots, m-1\}$ *und* $|\mathcal{Z}| = m$. *Die Funktion* $h : \mathcal{K} \to \mathcal{Z}$ *heißt* Hashfunktion. *Aus mathematischen Gründen kann eine derartige Funktion* h *im allgemeinen*[7] *nicht injektiv sein, d.h. es wird Schlüssel* k_1 *und* k_2 *geben mit* $k_1 \neq k_2$ *und* $h(k_1) = h(k_2)$. *Dies bezeichnet man als* Kollision. *Das Verhältnis* $n/|\mathcal{K}|$ *heißt* Schlüsseldichte. *Der Wert* $\alpha := n/m$ *heißt der* Belegungsfaktor *der Hashtabelle.*

Beispiel: Wir wollen die Bundesländer auf 13 Behälter verteilen. Als einfaches Beispiel einer Hash-Funktion verwenden wir dabei die folgende:

$$hash(w) = \sum_{i=1}^{length(w)} ascii(w_i) \bmod 13$$

Damit ergibt sich die in Tabelle 6.2 dargestellte Verteilung. Obwohl noch Behälter frei sind, kommt es zu mehreren Kollisionen.

In Tabelle 6.3 sieht man, wie viele Bundesländer in die einzelnen Behälter gekommen sind.

Wie groß ist die Wahrscheinlichkeit von Kollisionen? Sei h eine *uniforme* Hash-Funktion, die die Schlüssel $s \in S$ gleichmäßig auf alle m Behälter verteilt, d.h. für $0 \leq i < m$ ist

$$P(h(s) = i) = 1/m$$

Dann gilt

$$P_{Kollision} \quad = \quad 1 - P_{KeineKollision}$$

[7] also falls $n > m$

Tabelle 6.2: Hashcodes der Bundesländer

Baden-Württemberg	11
Bayern	11
Berlin	6
Brandenburg	12
Bremen	3
Hamburg	8
Hessen	3
Mecklenburg-Vorpommern	8
Niedersachsen	1
Nordrhein-Westfalen	8
Rheinland-Pfalz	2
Saarland	0
Sachsen	7
Sachsen-Anhalt	2
Schleswig-Holstein	0
Thüringen	5

Tabelle 6.3: Verteilung der Bundesländer auf die Behälter

Behälter	0	1	2	3	4	5	6	7	8	9	10	11	12
Anzahl Bundesländer	2	1	2	2	0	1	1	1	3	0	0	2	1

$$P_{KeineKollision} = \prod_{i=1}^{n} P_i,$$

wobei P_i die Wahrscheinlichkeit dafür ist, dass der i-te Schlüssel auf einen freien Platz kommt, wenn die Schlüssel $1, \ldots, i-1$ auch alle auf freie Plätze gekommen sind. Nun ist $P_1 = 1$, $P_2 = (m-1)/m$, usw. also allgemein $P_i = (m-i+1)/m$, und somit

$$P_{Kollision} = 1 - \frac{m(m-1)\ldots(m-n+1)}{m^n}$$

Für $m = 365$ ergeben sich die folgenden Kollisionswahrscheinlichkeiten:

- n = 22: $P_{Kollision} \approx 0,475$
- n = 23: $P_{Kollision} \approx 0,507$

Dies ist als das sogenannte „Geburtstagsparadoxon“ bekannt: Sind mehr als 23 Personen zusammen, so haben mit mehr als 50% Wahrscheinlichkeit mindestens zwei von ihnen am selben Tag Geburtstag. Für das Hashing bedeutet das, dass Kollisionen praktisch unvermeidbar sind.

Für die Wahl guter Hashfunktionen hat man die folgenden Anforderungen:

- Sie sollte *surjektiv* sein, d.h. alle Werte der Zielmenge sollten erfasst werden.
- Gleichverteilung: $\mathcal{K}$ sollte auf $\mathcal{Z}$ gleichförmig abgebildet werden.
- Sie sollte möglichst einfach zu berechnen sein.
- Sie sollte möglichst wenig Kollisionen erzeugen.

Die Eigenschaft *injektiv* zu sein, kann im allgemeinen *nicht* erfüllt werden, da die Menge der Schlüssel mächtiger (größer) als die Zielmenge ist. Lediglich bei perfektem Hashing (s.u.), bei dem die Menge der Schlüssel vorab, d.h. zur Kompilierzeit bekannt ist, kann man Injektivität erreichen.

Damit hängt die Eigenschaft einer Hashfunktion „gut“ zu sein, wesentlich von den statistischen Eigenschaften der Schlüsselmenge ab! Für die weitere Analyse nehmen wir o.B.d.A. an, dass $\mathcal{K} = \mathbb{N}$, d.h. dass wir uns auf natürliche Zahlen als Schlüssel beschränken können. Für weiterführendes mathematisches Material zu Hashfunktionen sei auf [Knu97b] verwiesen!

Divisionsmethode

Bei der Divisionsmethode verwendet man Hashfunktionen der Form

$$h(k) = k \bmod m. \tag{6.1}$$

Dabei ist es wichtig, bestimmte Werte von m zu vermeiden. So sind zum Beispiel Zweierpotenzen $m = 2^l$ ungünstig, da $h(k)$ dann nur die lowest-order bits auswertet. Üblicherweise verwendet man Primzahlen die nicht in der Nähe einer Zweierpotenz 2^l liegen. Angenommen man wollte 2000 Strings speichern und nicht mehr als 3 Werte in einem Behälter ablegen, dann böte sich $m = 701$ an, da prim und in der Nähe von $2000/3$ und nicht nahe an einer Zweierpotenz.

Multiplikationsmethode

Bei der Multiplikationsmethode verwendet man Hashfunktionen der Form

$$h(k) = \lfloor (m(k\,A \bmod 1)) \rfloor, \tag{6.2}$$

wobei $0 < A < 1$ ($(k\,A \bmod 1)$ ist der Nachkommaanteil). Für m kann man eine Zweierpotenz wählen (schnell), für A wird oft (nach Knuth [Knu98]) $A \approx (\sqrt{5} - 1)/2$ gewählt.

Universelles und Perfektes Hashing

Definition 6.6.2. Perfektes Hashing *bezeichnet eine Hashfunktion, die keine Kollisionen verursacht.*

Eine perfekte Hashfunktion ist injektiv. Dies ist natürlich nur dann möglich, wenn man alle Schlüssel vorab kennt und dann die Hashfunktion entsprechend auswählen kann. Von Interesse ist dies z. B. für Compiler für Symboltabellen o.ä.

Definition 6.6.3. Universelles Hashing *bezeichnet eine Technik, Hashfunktionen zufällig auszuwählen, um den Worst-Case unwahrscheinlich zu machen.*

Natürlich muss die einmal gewählte Hashfunktion über die Lebenszeit der Hashtabelle verwendet werden, und kann nicht beim Auftreten bestimmter Schlüssel verändert werden. Für Details sei auf [CLR04] verwiesen.

6.6.3 Kollisionsauflösung durch Verkettete Listen

Kollisionsauflösung durch Verkettete Listen basiert auf einer einfachen Idee: Wenn es zu Kollisionen kommt, dann werden die Elemente in einer Verketteten Liste im selben Behälter abgespeichert, d.h. der Behälter enthält einen Zeiger auf den Beginn (Kopf) der Verketteten Liste, wie in Abbildung 6.9 verdeutlicht. Damit ergeben sich die folgenden Operationen bzw. Algorithmen 6.15, 6.16 und 6.17.

Algorithmus 6.15 CHAINED-HASH-INSERT(T, x)

1: insert x at head of list $T[h(x.key)]$

Algorithmus 6.16 CHAINED-HASH-SEARCH(T, x)

1: search for x with key $k := x.key$ in list $T[h(k)]$

Algorithmus 6.17 CHAINED-HASH-DELETE(T, x)

1: delete x from list $T[h(x.key)]$

Analyse

Der Worst-Case einer Hashtabelle mit Kollisionsauflösung durch Verkettete Listen ist schlecht, da im Worst-Case *alle* Elemente in der Liste sein können – also nur ein Behälter verwendet wird. Daher ist der Worst-Case für die Suche $\Theta(n)$ genauso schlecht wie bei einer Verketteten Liste.

Der Durchschnittsfall ist allerdings wesentlich besser, wie das nachstehende Theorem beweist.

Theorem 6.6.1. *Im Durchschnitt dauert eine erfolglose Suche bei einer Hashtabelle mit Verketteten Listen und Uniformem Hashing, d.h. bei dem die Hashwerte gleichverteilt sind,* $\Theta(1+\alpha)$*, wobei* $\alpha = n/m$ *den Belegungsfaktor bezeichnet (mit* n *Schlüsseln und Tabellengröße* m*).*

Beweis. Der Erwartungswert für die Zeit einer nicht erfolgreichen Suche für einen Wert k ist die zu erwartende Zeit, die man benötigt, um bis zum Ende der Liste $T[h(k)]$ zu suchen. Der Erwartungswert für deren Länge beträgt aber $E[n_{h(k)}] = \alpha$, was man wie folgt sieht. Sei

$$X_i := \begin{cases} 1 & h(k_i) = h(k), \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt $n_j = \sum_{i=1}^{n} X_i$ für alle j und somit $E[n_{h(k)}] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1/m = n/m$. Da die Berechnung von h konstant ist ($\Theta(1)$), ergibt sich die Gesamtzeit als $\Theta(1+\alpha)$. □

Erfolgreiche Suchen sind etwas schwieriger zu analysieren, da nicht jede Liste gleich wahrscheinlich ist. Jedoch gilt auch hier der folgende Satz:

Theorem 6.6.2. *Im Durchschnitt dauert eine erfolgreiche Suche bei einer Hashtabelle mit Verketteten Listen und Uniformen Hashing, d.h. bei dem die Hashwerte gleichverteilt sind,* $\Theta(1+\alpha)$*, wobei* $\alpha = n/m$ *den Belegungsfaktor bezeichnet.*

Beweis. Siehe [CLR04]. □

Man kann daher die grundlegende Idee von Hashing auch wie folgt verstehen. Man unterteilt die Menge S in m disjunkte Untermengen $S_1, \ldots, S_m$, so dass jede Untermenge eine zu erwartenden Größe von n/m bekommt. Dann muss man bei einer Suchanfrage mit dem Schlüssel k lediglich die Untermenge, die zu $h(k)$ gehört untersuchen. Daher betragen die Suchkosten $\mathcal{O}(n/m)$. Wenn wir die Zahl der Behälter mit n wachsen lassen, wenn also gilt $n \in \mathcal{O}(m)$, dann gilt sogar $\mathcal{O}(n/m) = \mathcal{O}(n)/m = \mathcal{O}(m)/m = \mathcal{O}(1)$. Im Durchschnitt können wir also sämtliche Operationen in konstanter Zeit ausführen!

Diese Methodik ist eine klassische Anwendung des sogenannten “Space-Time Trade-offs”, d.h. um bessere Performanz zu erzielen wird Platz geopfert („verschwendet“).

Implementierung

Nachstehende C-Code Fragmente des Listings 6.5 und 6.6 verdeutlichen eine (rudimentäre) Implementierung einer Hashtabelle, die eine Verkettete Liste für die Überläufer (bei Kollisionen) verwendet.

Listing 6.5: Hashtabelle mit Liste für Überläufer – C-Code Header

```
1  typedef struct Item Item;
2
3  struct Item{
4      char *key;
5      char *value;
6      struct Item *next;
7  };
8
9  int insert(Item* item);
10 Item* search(char *key);
11 int delete(char *key);
12 void print_table();
```

Listing 6.6: Hashtabelle mit Liste für Überläufer – C-Code

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include "hash.h"
5
6  enum {
7      NHASH = 43,      /* size of hashtable array */
8      MULTIPLIER = 31 /* multiply bits */
9  };
10
11
12 Item *statetab[NHASH]; /* hash table of items */
13
14 unsigned int hash(char *key)
15 {
16     unsigned int h;
17     unsigned char *p;
18     h = 0;
19     for(p = (unsigned char *) key; *p != '\0'; p++)
20         h = MULTIPLIER * h + *p;
21     return h % NHASH;
22
23 }
24
25 int insert(Item* item)
26 {
27     Item *ip;
28     int h = hash(item->key);
29     if((ip = statetab[h]) == NULL){
30         item->next = NULL;
31     }
32     else{
33         Item *ip_tmp; /* test whether item already exists */
34         ip_tmp = ip;
35         while (ip_tmp != NULL) {
36             if(strcmp(item->key, ip_tmp->key) == 0)
37                 return -1; /* error, item already exists */
38             ip_tmp = ip_tmp->next;
```

```
39          }
40          /* collision: insert at front of list */
41          item->next = ip;
42      }
43      statetab[h] = item;
44      return 0;
45  }
46
47  Item* search(char *key)
48  {
49      Item *ip;
50      int h;
51      h = hash(key);
52      for(ip = statetab[h]; ip != NULL; ip = ip->next) {
53          if(strcmp(key, ip->key) == 0)
54              break;
55      }
56      return ip;
57  }
58
59  int delete(char *key)
60  {
61      /* Exercise! */
62      return -1;
63  }
64
65  void print_table()
66  {
67      Item *ip;
68      for(int i = 0; i < NHASH; i++) {
69          for (ip = statetab[i]; ip != NULL; ip = ip->next){
70              printf("[%s: %s], ", ip->key, ip->value);
71          }
72      }
73  }
```

6.6.4 Kollisionsauflösung durch Offene Adressierung

Idee

Bei der *Offenen Adressierung*

- benötigt man keine Verketteten Listen
- werden alle Elemente direkt in der Tabelle gespeichert
- enthält jeder Tabelleneintrag entweder ein Element oder NIL
- ist der Belegungsfaktor α nie größer als 1, d.h.
- kann die Hashtabelle „voll“ werden[8]

[8] in diesem Fall kann man keine neuen Elemente speichern, oder die Tabelle muss vergrößert werden – das kann bei der Kollisionsauflösung durch Verkettete Listen nicht

So kann man Platz sparen und mit dem gewonnenen Platz ggf. eine größere Tabelle anlegen und so die Wahrscheinlichkeit für Kollisionen verringern.

Implementiert wird Offene Adressierung in dem man sukzessive *probiert*, ob ein Behälter frei ist und falls nicht, den nächsten freien *berechnet*. Dieses Verfahren wird auch als *Sondieren* (englisch "probing") bezeichnet. Wir erweitern dazu die Hashfunktion h so, dass die Probesequenz mit berücksichtigt wird:

$$h : \mathcal{K} \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}.$$

Damit erhalten wir für jeden Schlüssel k die folgende Probesequenz

$$(h(k,0), \ldots, h(k,m-1)) .$$

Die Hashfunktion soll so konstruiert sein, dass diese Probesequenz eine Permutation der Werte $\{0, \ldots, m-1\}$ herstellt und somit garantiert ist, dass bei wiederholter Anwendung *alle* Behälter ausprobiert werden, der Schlüssel k also eingefügt werden kann, wenn überhaupt noch irgendwo Platz in der Tabelle ist. Der Algorithmus HASH-INSERT 6.18 verdeutlicht das Verfahren (in diesem Pseudocode sind zur Vereinfachung die Werte x, die eingefügt werden, identisch mit dem Schlüssel k).

Algorithmus 6.18 HASH-INSERT(T, k)

```
1:  i ← 0
2:  repeat
3:      j ← h(k, i)
4:      if T[j] = NIL then
5:          T[j] ← k
6:          return j
7:      else
8:          i ← i + 1
9:  until i = m
10: return Error: "Hash table overflow"
```

Der Algorithmus HASH-SEARCH 6.19 verdeutlicht das Verfahren bei der Suche.

Löschen eines Eintrags k ist bei Offener Adressierung schwierig, da wir nicht einfach den Eintrag k entfernen können. Täten wir dieses, so würde HASH-SEARCH die nach k eingefügten Schlüssel nicht finden können. In der Praxis speichert man daher oft einen expliziten `DELETED` Wert und modifiziert HASH-SEARCH entsprechend (Übungsaufgabe!).

Es gibt mehrere Verfahren die Hashfunktion h wie oben angegeben zu erweitern.

---

passieren.

Algorithmus 6.19 HASH-SEARCH(T, k)

```
1: i ← 0
2: repeat
3:     j ← h(k, i)
4:     if T[j] = k then
5:         return j
6:     i ← i + 1
7: until T[j] = NIL or i = m
8: return NIL
```

1. Lineares Sondieren
2. Quadratisches Sondieren
3. Double Hashing

Beim Linearen Sondieren kommt es zu einer „Klumpenbildung“, d.h. in der Tabelle sind immer wieder mehrere benachbarte Felder belegt. Dies verschlechtert die Laufzeit der Operationen Einfügen und Suchen. Die beiden anderen Verfahren haben das Ziel, die „Klumpenbildung“ durch bessere Verteilung zu vermeiden. Wir behandeln hier nur Lineares Sondieren, für die anderen Verfahren siehe [CLR04].

Definition 6.6.4. *Gegeben sei eine Hashfunktion h'. Mit Linearem Sondieren bezeichnet man die folgende Hashfunktion:*

$$h(k, i) := (h'(k) + i) \bmod m.$$

Dass Lineares Sondieren alle Behälter ausprobiert, ist aus der Definition offensichtlich. Es ist auch ein einfach zu implementierendes Verfahren. Allerdings hat es die Schwäche, dass es zur Cluster- und Sequenzbildung neigt: Folgt ein Behälter auf eine Sequenz von belegten Behältern, so ist die Wahrscheinlichkeit, dass er belegt wird größer (alle „Vorgängerkollisionen“ werden auf diesen „geschoben“)!

Beispiel

Wir verdeutlichen Offene Adressierung anhand des oben eingeführten Beispiels der Bundesländer. Wenn wir für h Lineares Sondieren wählen und die Bundesländer in der alphabetischen Reihenfolge einfügen ergibt sich folgendes Bild:

1. Baden-Württemberg, Hashcode 11: landet auf Index 11 ($i = 0$)
2. Bayern, ebenfalls Hashcode 11: Kollision, Erhöhung des Index ($i = 1$) und landet somit auf Index 12

3. Berlin, Hashcode 6: keine Kollision, landet somit auf Index 6
4. Brandenburg, Hashcode 12: Kollision, Erhöhung des Index ($i = 1$) und landet somit auf Index 0 (zyklisch!)
5. Bremen, Hashcode 3: keine Kollision, landet somit auf Index 3
6. usw.
7. Nach 13 Bundesländern sieht die Hashtabelle T so aus:

$$\begin{aligned} T \quad = \quad & (\text{Brandenburg, Niedersachsen, Rheinland-Pfalz,} \\ & \text{Bremen, Hessen, Saarland, Berlin,} \\ & \text{Sachsen, Hamburg, Mecklenburg-Vorpommern,} \\ & \text{Nordrhein-Westfalen, Baden-Württemberg, Bayern}) \end{aligned}$$

   und ist somit vollständig gefüllt.
8. Für alle nachfolgenden Bundesländer (Sachsen-Anhalt, Schleswig-Holstein und Thüringen) lautet dann die Nachricht „Error: Overflow“.

6.6.5 Einsatzgebiete

Nachstehend einige wenige Beispiele für die zahlreichen Anwendungsfälle für Hashtabellen:

- Assoziative Arrays (auch Maps, Lookup-Table, Dictionary oder Wörterbuch).
- Datenbanken: Hashtabellen werden als Index für Tabellen verwendet (Hashindex kann unter günstigen Bedingungen zu idealen Zugriffszeiten führen).
- Implementierung von Mengen (Sets)
- Implementierung von Objekten bei OO-Sprachen (z. B. Python)
- Caches
- Symboltabellen für Compiler oder Interpreter oder für gemeinsam genutzte Strukturen, bekannt als Hash Consing oder Persistente Datenstrukturen – immer wichtiger für Concurrent Programming (Multi-Cores!)
- Praktisch alle nicht-trivialen Anwendungen. In der Praxis werden Hashtabellen zur Beschleunigung von Algorithmen genutzt. Der Preis dafür ist zusätzlicher Speicherplatz – der klassische Trade-off in der Informatik.

6.7 Fragen und Aufgaben zum Selbststudium

1. Erklären Sie den Unterschied zwischen statischen und dynamischen Datenstrukturen!
2. Geben Sie fünf Beispiele für dynamische Datenstrukturen!
3. Schreiben Sie Pseudo-Code für einen Stapel und eine Warteschlange!
4. Beschreiben Sie das Resultat der folgenden Operationen auf einem STACK (Zeichnung o.ä.)!
   (a) PUSH$(S, 5)$
   (b) PUSH$(S, 6)$
   (c) PUSH$(S, 1)$
   (d) POP(S)
   (e) PUSH$(S, 10)$
   (f) POP(S)
5. Ein mit Hilfe eines Feldes implementierter Stack hat nur endlich viel Platz. Implementieren Sie eine alternative Version (Pseudo-Code) mit Hilfe einer Verketteten Liste, die diese Beschränkung nicht aufweist!
6. Überlegen Sie, ob es sinnvoll wäre, eine Verkettete Liste zu sortieren! Wird die Suchzeit im Fall der erfolgreichen und der erfolglosen Suche besser? Wirkt sich das auf die Suchzeit in $\mathcal{O}$-Notation aus?
7. Definieren Sie einen Algorithmus LIST-DELETE, der Löschen für eine Verkettete Liste ermöglicht!
8. Definieren Sie einen Algorithmus LIST-DELETE, der Löschen für eine Doppelt-Verkettete Liste ermöglicht!
9. Eine Verkette Liste ist in C implementiert. Beim Aufruf von LIST-DELETE(L, x) wird der Speicher, der von x belegt wird, mit Hilfe von `free` freigegeben. Was halten Sie von dieser Implementierung?
10. Erklären Sie den Algorithmus HEAP-INCREASE-KEY und beschreiben Sie, wofür er eingesetzt wird!
11. Beschreiben Sie in Ihren eigenen Worten, welche Faktoren die Performanz von Hashtabellen wesentlich beeinflussen!
12. Eine Hashfunktion h bilde Schlüssel auf ein Feld der Länge m ab. Angenommen, die Hashfunktion ist uniform, wie wahrscheinlich ist eine Kollision der ersten zwei eingefügten Elemente?

13. Eine Hashfunktion h bilde Schlüssel auf ein Feld der Länge m ab. Wieviele Elemente kann die Hashtabelle *höchstens* kollisionsfrei aufnehmen?

14. Wodurch können Kollisionen bei Hashtabellen aufgelöst werden? Geben Sie zwei verschiedene Verfahren an!

15. Geben Sie Pseudo-Code für das Löschen von Elementen in einer Hashtabelle mit Kollisionsauflösung durch Verkettete Listen an!

16. Fügen Sie die folgende Sequenz in eine Hashtabelle mit sieben Buckets „per Hand“ ein: $(6, 29, 17, 14, 21, 4, 12, 25, 5)$. Dafür sollen Sie die folgende Hashfunktion verwenden:

$$h(k) = k \bmod 7$$

    Zeichnen Sie Ihre Lösung!

17. Geben Sie zwei gebräuchliche Verfahren an, Hashfunktionen zu konstruieren!

Kapitel 7

Nicht-lineare Datenstrukturen

Die bisher untersuchten linearen Datenstrukturen reichen für viele einfache Probleme aus. Es gibt aber bei komplexen Problemen die Notwendigkeit, kompliziertere Datenstrukturen zu konstruieren, um die Nichtlinearität des Problemraums abbilden zu können. Diese nicht-linearen Datenstrukturen sind Gegenstand dieses Kapitels.

7.1 Graphen

Graphen sind sehr wichtige Datenstrukturen in der Informatik. Einsatzgebiete für Graphen sind vielfältig, u.a.

- Abhängigkeiten
- Optimierungsprobleme
- U-Bahnpläne
- Karten und Navigation
- Routingalgorithmen (Internet, Mobilfunk)
- WWW
- usw.

Die mathematische Definition eines Graphen abstrahiert von den (unwesentlichen) Details und reduziert das Problem damit auf den algorithmischen Kern.

7.1.1 Definition und Darstellung

Definition 7.1.1. *Ein* Graph *G ist ein Tupel (V, E) mit einer Menge von* Knoten *V (englisch "vertex/vertices" oder "Node") und einer Menge E von* Kanten *(englisch "edge/edges"). Dabei ist E in* ungerichteten *Graphen eine Teilmenge aller 2-elementigen Teilmengen von V, bei* gerichteten *Graphen eine Teilmenge des kartesischen Produktes.*

Bei gerichteten Graphen kommt es also auf die Richtung der Verbindungskante an, bei ungerichteten nicht. Die Abbildungen 7.1 und 7.2 verdeutlichen die übliche graphische Darstellung von Graphen.

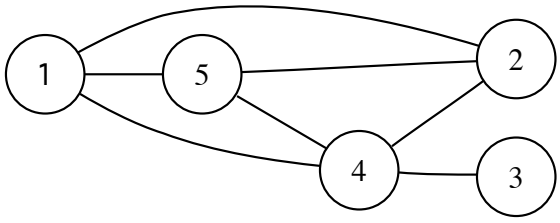


Abbildung 7.1: Ungerichteter Beispielgraph

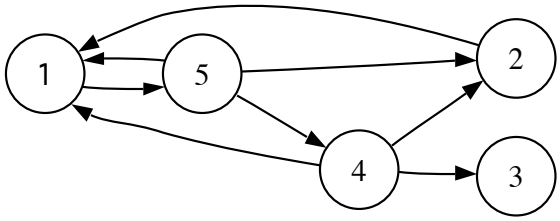


Abbildung 7.2: Gerichteter Beispielgraph

Definition 7.1.2. *Der* Knotengrad *eines Knotens $v \in V$ in einem ungerichteten Graphen $G = (V, E)$ ist die Anzahl der Kanten $e \in E$, die mit v verbunden sind.*

Bei gerichteten Graphen müssen die Richtungen der Kanten berücksichtigt werden. Man spricht dann vom *Eingangsgrad* und *Ausgangsgrad*.

Man kann Graphen durch ihre *Adjazenzmatrix*[1] M, siehe Tabellen 7.1 und 7.2, darstellen. Für ungerichtete Graphen gilt $M^T = M$.

Eine alternative Darstellungsart ist die Darstellung als Adjazenz-Liste, wie in Abbildung 7.3 dargestellt. Diese Darstellung wird häufig auch direkt als Im-

[1] von lateinisch adiacens „dabeiliegend".

	1	2	3	4	5
1	0	1	0	1	1
2	1	0	0	1	1
3	0	0	0	1	0
4	1	1	1	0	1
5	1	1	0	1	0

Tabelle 7.1: Adjazenzmatrix des Graphen aus Abbildung 7.1

$\nearrow$	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	0
3	0	0	0	0	0
4	1	1	1	0	0
5	1	1	0	1	0

Tabelle 7.2: Adjazenzmatrix des Graphen aus Abbildung 7.2

plementierungsstrategie verwendet. Adjazenz-Listen können unterschiedlich implementiert werden. In dieser Abbildung sehen Sie ein Feld mit Zeigern auf Felder der adjazenten Knoten.

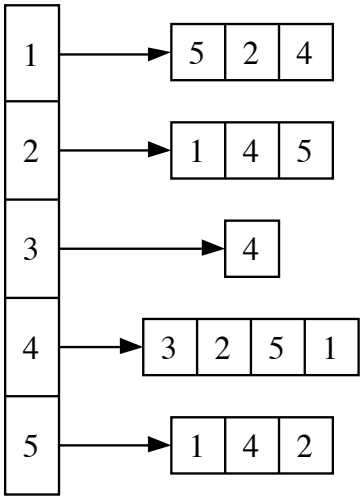


Abbildung 7.3: Listendarstellung des Graphen aus Abbildung 7.1

Beide Darstellungsformen haben ihre Vor- und Nachteile hinsichtlich Speicher- und Zugriffseffizienz. In der Praxis werden auch beide Formen verwendet. Für die Auswahl einer geeigneten Datenstruktur für gegebene Graphen betrachten wir die Tabelle 7.3. Sofern nicht explizit aufgeführt, gelten die Angaben für den Best, Average, und Worst Case. Offensichtlich gilt $\Theta(|V| + |E|) = \Theta(|E|)$

für zusammenhängende Graphen.

	Adjazenzmatrix	Adjazenzliste
Speicherplatz bzw. Besuchen aller Knoten über die Datenstruktur	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Zugriff auf Kante zwischen Knoten u und v	$\Theta(1)$	$\Theta(Adj[u])$: BC: $\Theta(1)$ AC: $\Theta(\|E\|/\|V\|)$ WC: $\Theta(\|E\|)$
Zugriff auf alle adjazenten Knoten zu einem Knoten u	$\Theta(\|V\|)$	

Tabelle 7.3: Speicherplatz- und Laufzeit-Eigenschaften von Adjazenzmatrix- und Adjazenzlisten-Darstellung von Graphen

7.1.2 Implementierungsaspekte

Ein Beispiel, wie Graphen in C strukturiert implementiert werden können, ist in Listing 7.5 dargestellt. Eine beliebte C/C++ Bibliothek, die auch Unterstützung für Graphen enthält, ist die boost Bibliothek (s. `http://www.boost.org/`).

7.1.3 Wege in Graphen

Wir betrachten zunächst zwei einfache Aufgabenstellungen für Graphen: Finden von Rundwegen. Als *Rundweg* bezeichnen wir einen Weg, der bei einem Knoten beginnt und auch endet.

- Rundwege in Graphen, die jede Kante genau einmal durchlaufen.
- Rundwege in Graphen, die jeden Knoten genau einmal besuchen.

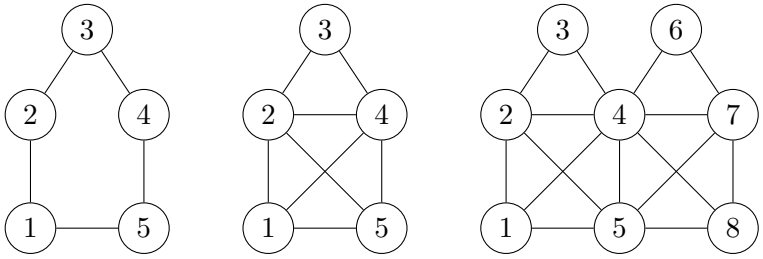
Auch wenn die beiden Aufgabenstellungen einfach sind und sehr ähnlich aussehen, gibt es gravierende Unterschiede in der Komplexität der Lösung, wie wir noch sehen werden.

Rundwege über alle Kanten

Definition 7.1.3. *Ein* Eulerkreis *eines zusammenhängenden ungerichteten Graphen $G = (V, E)$ ist ein Zyklus, der jede Kante aus G genau einmal durchläuft und am Ende wieder beim Startknoten ankommt.*

Offensichtlich werden bei einem Eulerkreis die Knoten von G (in Abhängigkeit ihres Knotengrades) gegebenenfalls mehrfach durchlaufen.

Abbildung 7.4 enthält drei Graphen. Die Abbildung 7.5 enthält die Versuche, einen Eulerkreis in den entsprechenden Graphen der Abbildung 7.4 zu zeichnen.



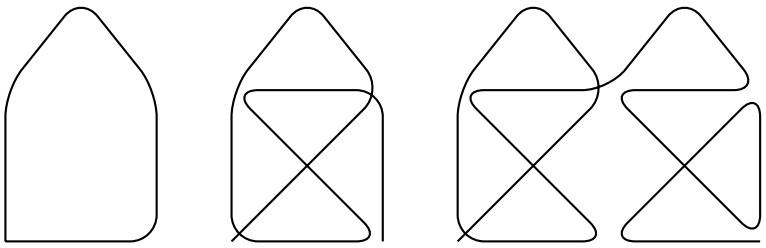
(a) Haus 1 (b) Haus 2 (c) Doppelhaus

Abbildung 7.4: Graphen: Haus vom Nikolaus

Der erste Graph (a) enthält offensichtlich einen Eulerkreis: $1 \to 2 \to 3 \to 4 \to 5 \to 1$. Ein Start an jedem anderen Knoten wäre auch möglich gewesen, da ein Eulerkreis ein geschlossener Kantenzug ist.

Beim zweiten Graphen (b) gibt es keinen Eulerkreis. Jedoch ist es möglich, jede Kante genau einmal zu durchlaufen, wenn man bei Knoten 1 oder 5 startet. Der Kantenzug endet dann im jeweils anderen Knoten, sofern alle Kanten durchlaufen werden.

Beim dritten Graphen (c) gibt es auch keinen Eulerkreis. Darüber hinaus gibt es auch keinen Kantenzug, der alle Kanten genau einmal durchläuft, auch wenn er in einem vom Startknoten unterschiedlichen Knoten endet. Wenn man den Graphen modifiziert (in der Abbildung wurde die Kante zwischen den Knoten 4 und 5 eliminiert), dann können alle anderen Kanten durchlaufen werden, wie in der Abbildung 7.5 (c) erkennbar.

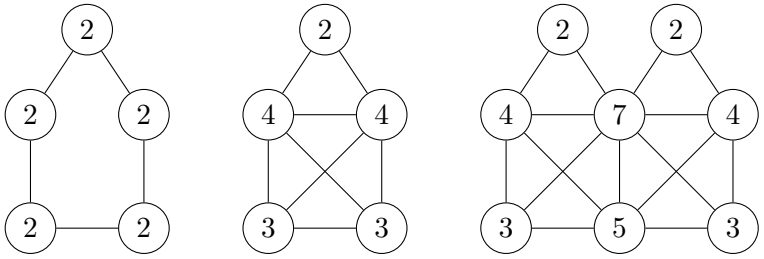


(a) Haus 1 (b) Haus 2 (c) Doppelhaus

Abbildung 7.5: Zeichnen: Haus vom Nikolaus

Um den Zusammenhang zwischen dem Graphen und der Existenz eines Eulerkreises zu erkennen, betrachten wir Abbildung 7.6. Dort sind die Knoten mit ihren jeweiligen Knotengraden markiert. Man kann vermuten, dass die Existenz von Eulerkreisen etwas mit den Knotengraden zu tun hat.

In (a) gibt es nur gerade Knotengrade, ein Eulerkreis existiert.



(a) Haus 1 (b) Haus 2 (c) Doppelhaus

Abbildung 7.6: Graphen: Haus vom Nikolaus mit Knotengraden

In (b) existiert kein Eulerkreis. Wenn man jedoch eine weitere Kante zwischen den Knoten 1 und 5 einführen würde, dann wäre ein Eulerkreis vorhanden. In diesem Fall wären auch alle Knotengrade gerade.

In (c) in Abbildung 7.5 wurde bereits eine Kante weggelassen. Nach Einführung einer Kante zwischen den Knoten 1 und 8 würde ein Eulerkreis existieren. Auch in diesem Fall wären alle Knotengrade gerade.

Lemma 7.1.1. *In einem zusammenhängenden ungerichteten Graphen existiert genau dann ein Eulerkreis, wenn die Knotengrade aller Knoten des Graphen gerade sind.*

Beweis. Übungsaufgabe. □

Die Konstruktion eines Eulerkreises ist mit dem Algorithmus des Mathematikers Carl Hierholzer in linearer Zeit (basierend auf der Anzahl von Kanten) möglich. Auch wenn wir den Algorithmus hier nicht behandeln, wollen wir die Idee kurz angeben.

Man beginnt mit der Konstruktion eines geschlossenen Kantenzuges. Falls nun noch nicht durchlaufene Kanten vorhanden sind, dann konstruiert man einen weiteren geschlossenen Kantenzug aus noch nicht durchlaufenen Kanten an einem Knoten, der im bisherigen Kantenzug enthalten ist. Diese beiden Kantenzüge werden dann zu einem Kantenzug verbunden. Dies wird fortgesetzt, solange noch nicht alle Kanten durchlaufen worden sind.

An diesem Verfahren erkennt man auch die Beweisidee zum Lemma 7.1.1: Jeder Knoten besitzt einen geraden Knotengrad. Falls er noch nicht besucht wurde, ist der Knotengrad 0. Ist er in einem Kantenzug enthalten, dann ist sein Knotengrad $2, 4, 6, \ldots$, weil die Anzahl der Wege in den Knoten hinein gleich der Anzahl der Wege aus dem Knoten heraus sein muss. Bei jedem Besuch eines Knotens kommen zwei bislang nicht durchlaufene Kanten hinzu.

Königsberger Brückenproblem

Mit diesem Vorwissen können wir das *Königsberger Brückenproblem* lösen. Leonard Euler untersuchte, ob ein Weg im früheren Königsberg existiert, der jede der sieben Brücken über den Fluss Pregel genau einmal enthält. Außerdem fragte er sich, ob ein Rundweg möglich wäre.

Dazu betrachten wir in der Abbildung 7.7 auf der linken Seite ein Modell von Königsberg zur damaligen Zeit und auf der rechten Seite den entsprechenden Graphen. Die Knoten sind die vier Teile der Stadt und die Kanten sind die Brücken.

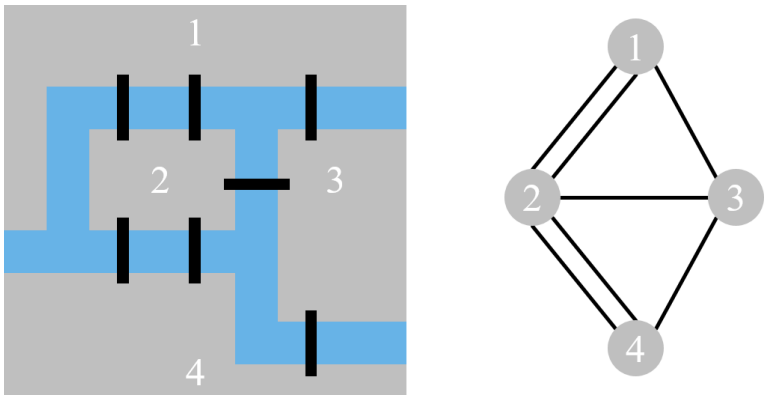


Abbildung 7.7: Königsberger Brückenproblem

Der Graph besitzt vier Knoten – alle mit ungeraden Knotengraden. Anhand von Lemma 7.1.1 wissen wir, dass kein Eulerkreis existiert. Man kann anhand des Graphen sehen, dass kein Weg existiert, der genau einmal über jede Brücke führt.

Rundwege über alle Knoten

Definition 7.1.4. *Ein* Hamiltonkreis *eines zusammenhängenden ungerichteten Graphen G ist ein Zyklus, der jeden Knoten aus G genau einmal besucht und am Ende wieder beim Startknoten ankommt.*

Theorem 7.1.1. *Die Entscheidung, ob ein ungerichteter Graph einen Hamiltonkreis enthält, ist NP-vollständig.*

Beweis. [CLR04]. □

Somit ist die Lösung des Problems, in einem zusammenhängenden ungerichteten Graphen einen Eulerkreis zu finden “leicht”, hingegen schon die Entscheidung, ob ein Hamiltonkreis existiert, “schwer”.

7.1.4 Breitensuche und Tiefensuche

Breitensuche

Breitensuche (eng. Breadth First Search oder BFS) stellt ein systematisches Verfahren dar, Knoten ausgehend von einem Startknoten s zu entdecken und ist im Algorithmus BFS 7.1 und Abbildung 7.8 dargestellt. Dieser Algorithmus setzt einen zusammenhängenden Graphen voraus. Der Algorithmus benutzt die bereits kennengelernte lineare Datenstruktur Warteschlange bzw. Queue. (Hier und im folgenden wird mit π der Vorgängerknoten bezeichnet.) Breitensuche ist ein „archetypischer“ Graphalgorithmus, der grundlegend für viele andere Algorithmen ist. Der Algorithmus entdeckt neue Knoten gleichmäßig verlaufend an einer Grenze zwischen bekannten und neuen Knoten. Ähnliche Ideen werden bei vielen Graphalgorithmen verwendet, wie z. B. bei dem noch zu untersuchenden DIJKSTRA Algorithmus für kürzeste Wege. BFS erzeugt durch den Teilgraph der Vorgängerknoten einen den Graph *aufspannenden Baum*, wie ebenfalls in Abbildung 7.8 verdeutlicht.

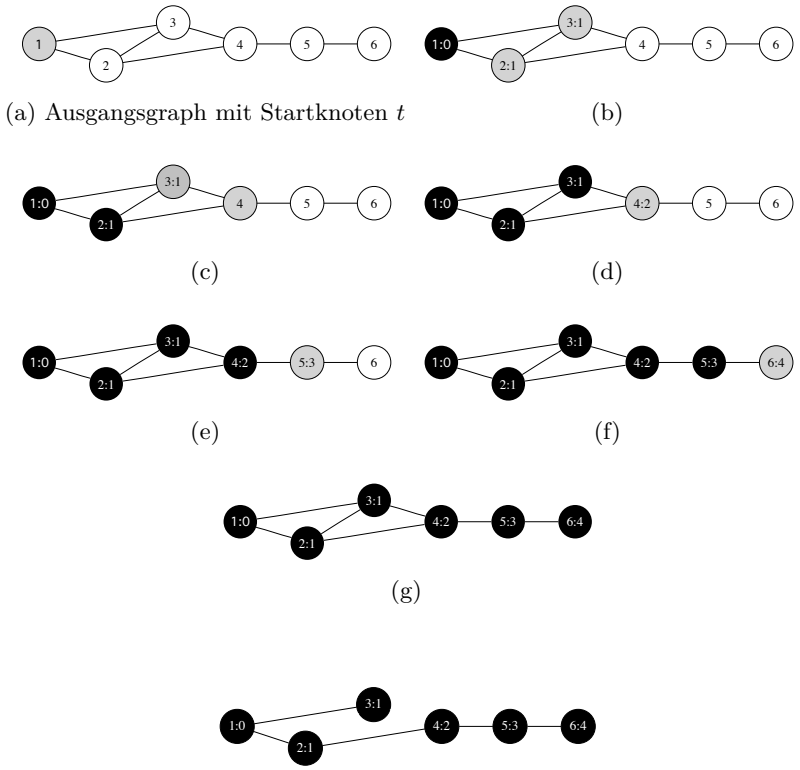
Algorithmus 7.1 BFS (G, s)

```
 1: for each vertex u ∈ G.V \ {s} do
 2:     u.color ← WHITE
 3:     u.d ← ∞
 4:     u.π ← NIL
 5: s.color ← GRAY
 6: s.d ← 0
 7: s.π ← NIL
 8: Q ← ∅                          // Q: zu bearbeitende Knoten
 9: ENQUEUE(Q, s)                  // Beginn mit Startknoten
10: while Q ≠ ∅ do
11:     u ← DEQUEUE(Q)
12:     for each ν ∈ G.Adj[u] do
13:         if ν.color = WHITE then
14:             ν.color ← GRAY
15:             ν.d ← u.d + 1
16:             ν.π ← u
17:             ENQUEUE(Q, ν)
18:     u.color ← BLACK
```

Eine Implementierung in C ist im Listing 7.4 dargestellt.

Laufzeit

Lemma 7.1.2. *BFS ist in $\mathcal{O}(|V| + |E|)$, wobei $|S|$ die Anzahl der Mengenelemente einer Menge S bezeichnet.*



(a) Ausgangsgraph mit Startknoten t (b)

(c) (d)

(e) (f)

(g)

(h) Spannbaum

Abbildung 7.8: Funktionsweise BFS

Beweis. Die Queue-Operationen benötigen $\mathcal{O}(|V|)$, da jeder Knoten $\nu \in V$ einmal in die Queue eingestellt und entnommen wird. Die Operationen, die die Adjazenzlisten betreffen, sind in $\mathcal{O}(|E|)$, also zusammen $\mathcal{O}(|V|+|E|)$. □

Bemerkung: Wenn der Graph nahezu vollständig ist, dann ergibt sich $\mathcal{O}(|V|^2)$, da dann $|E| = \mathcal{O}(|V|^2)$.

Tiefensuche

Tiefensuche (eng. Depth First Search oder DFS) stellt ein weiteres, systematisches Verfahren dar, Knoten ausgehend von einem Startknoten s zu entdecken und ist im Algorithmus DFS 7.2 und DFS-VISIT 7.3 dargestellt. Der Algorithmus versucht, zunächst möglichst „tief“ in den Graphen einzudringen, um dann, wenn es nicht mehr weitergeht, zurückzugehen (Rückverfolgung, englisch “back tracking”). In diesem Algorithmus werden die Zeitpunkte des erstmaligen Besuchs der Knoten gespeichert (*time*). Die hier nicht enthaltene

Distanz kann analog des Algorithmus 7.1 eingebaut werden.

DFS kann eingesetzt werden, um Eigenschaften eines Graphen zu ermitteln, wie z. B. die Existenz und den Ort von Zyklen. Abbildung 7.9 veranschaulicht einen Beispieldurchlauf. Auch DFS konstruiert – wie BFS – einen oder mehrere[2] Spannbäume.

Algorithmus 7.2 DFS (G)

```
1: for each vertex u ∈ G.V do
2:     u.color ← WHITE                    // Knoten noch nicht besucht
3:     u.π ← NIL
4: time ← 0
5: for each vertex u ∈ G.V do
6:     if u.color = WHITE then
7:         DFS-VISIT(G, u)
```

Algorithmus 7.3 DFS-VISIT (G, u)

```
1: time ← time + 1
2: u.d ← time                             // Knoten gefunden
3: u.color ← GRAY                         // Knoten in Bearbeitung
4: for each vertex ν ∈ G.Adj[u] do
5:     if ν.color = WHITE then
6:         ν.π ← u
7:         DFS-VISIT(G, ν)
8: u.color ← BLACK                        // Knoten fertig bearbeitet
9: time ← time + 1
10: u.f ← time
```

Basierend auf der in Listing 7.5 dargestellten Datenstruktur für Graphen kann man DFS wie in Listing 7.6 dargestellt implementieren.

Laufzeit

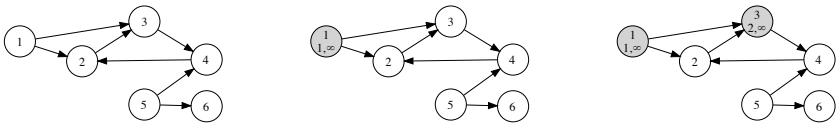
Lemma 7.1.3. *DFS ist in* $\Theta(|V| + |E|)$.

Beweis. Ohne Beweis (s. z. B. [CLR04]). □

7.1.5 Kürzester Weg – Dijkstra und A*

Definition 7.1.5. *Ein gewichteter Graph* $G = (V, E)$ *ist ein gerichteter Graph* G *mit einer Gewichtsfunktion* $w : E \to \mathbb{R}$.

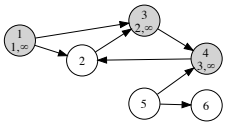
[2] falls mehrere Zusammenhangskomponenten vorhanden sind



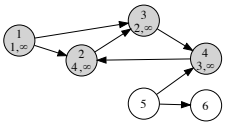
(a)

(b)

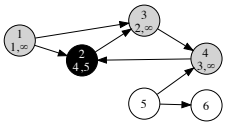
(c)



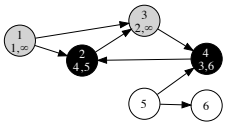
(d)



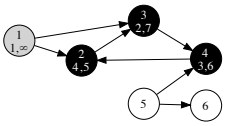
(e)



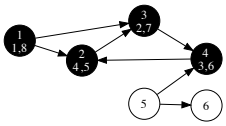
(f)



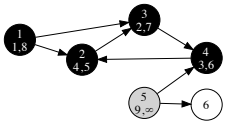
(g)



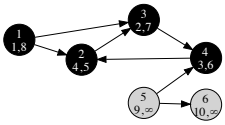
(h)



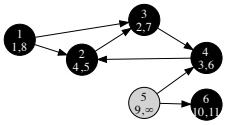
(i)



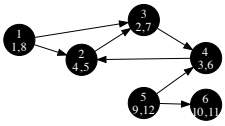
(j)



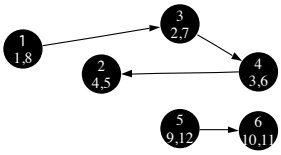
(k)



(l)



(m)



(n) Spannbaum

Abbildung 7.9: Funktionsweise DFS

Man kann sich diese Gewichtsfunktion vorstellen als Abstand oder Kosten (z. B. Benzinverbrauch). Man kann zwar auch negative Gewichte untersuchen; wir wollen das hier jedoch nicht tun und so betrachten wir im folgenden *ausschließlich* positive w mit $w : E \to \mathbb{R}^+$.

Relax!

Grundlegend für viele Algorithmen, die kürzeste Weg o.ä. in Graphen berechnen, ist die Prozedur RELAX (s. Algorithmus 7.4 und Abbildung 7.10). Sie besteht darin, dass man *Abschätzungen* über den kürzesten Weg fortlaufend verfeinert. Zunächst wird der Graph initialisiert, wie in der Prozedur INITIALIZE 7.5 ($\nu.\pi$ bezeichnet wieder den Vorgänger von ν) ausgeführt. Dann testet man, ob der bisher gefundene Schätzwert verbessert werden kann.

Algorithmus 7.4 RELAX (u, ν, w)

1: **if** $\nu.d > u.d + w(u, \nu)$ **then**
2: $\quad \nu.d \leftarrow u.d + w(u, \nu)$
3: $\quad \nu.\pi \leftarrow u$

(a) Vorher (b) Nachher

Abbildung 7.10: Relax Prozedur liefert für Knoten B einen kürzeren Weg

Algorithmus 7.5 INITIALIZE (G, s)

1: **for** each vertex $\nu \in G.V$ **do**
2: $\quad \nu.d \leftarrow \infty$
3: $\quad \nu.\pi \leftarrow NIL$
4: $s.d = 0$

Dijkstra

Wir können den kürzesten Weg nun mithilfe des DIJKSTRA Algorithmus (s. 7.6)[3] berechnen. Dabei sei noch einmal daran erinnert, dass dieser Algorith-

[3]Abweichend von Cormen [CLR04] (und effizienter) untersuchen wir in Zeile 7 nur diejenigen Knoten, die *nicht* im "Closed Set" S sind, da sich die Werte dieser Knoten durch

mus nur funktioniert, wenn sämtliche Gewichte *positiv* sind! Wir verwenden

Algorithmus 7.6 DIJKSTRA(G, w, s)

```
1: INITIALIZE(G, s)
2: S ← ∅
3: Q ← G.V                                  // Initialize Q with all V ∈ G
4: while Q ≠ ∅ do
5:     u ← EXTRACT-MIN(Q)
6:     S ← S ∪ {u}
7:     for each vertex ν ∈ G.Adj[u] and ν ∉ S do
8:         RELAX(u, ν, w)
```

für Q eine Prioritätswarteschlange. Das ist nicht unbedingt notwendig, da man z. B. auch eine sortierte Liste hätte verwenden können, aber für die Laufzeit des Algorithmus günstiger.

Der Algorithmus berechnet die kürzesten Abstände von s zu jedem Knoten. Der Weg selbst läßt sich aus der Liste der Vorgänger berechnen, wie der Algorithmus DIJKSTRA-SHORTEST-PATH 7.7 ausführt[4] (falls der Leser den Vorgängerbaum vermissen sollte – dieser wird ja implizit in RELAX mit berechnet, s. Algorithmus 7.4 und die Vorgänger stehen daher *nach* Aufruf von DIJKSTRA im Graphen zur Verfügung!).

Algorithmus 7.7 DIJKSTRA-SHORTEST-PATH $(G, w, start, end)$

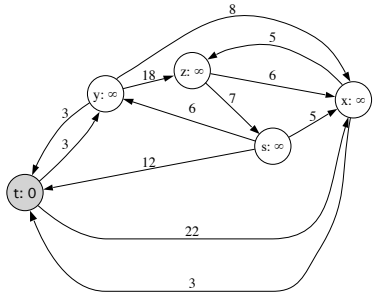
```
Require: DIJKSTRA(G, w, start)              // Vorbedingung
1: PATH ← {}
2: while true do
3:     PATH ← APPEND(PATH, end)
4:     if end = start then
5:         BREAK
6:     end ← end.π                          // Den Vorgänger nehmen
7: return REVERSE(PATH)                     // Die Liste umdrehen
```

Der Algorithmus ist in den Abbildungen 7.11 und 7.12 veranschaulicht (ausführbarer Pseudocode ist im Listing 7.7 im Abschnitt „Ausführbarer Pseudo-Code“ aufgeführt).

Der Dijkstra-Algorithmus ist ein schönes Beispiel für einen so genannten „gierigen“ (englisch “Greedy-”) Algorithmus, da er für den jeweils nächsten Startpunkt (Zeile 5) für seine Analyse denjenigen Knoten auswählt, der am nächsten ist. Dies ist möglich, weil das kürzeste-Weg-Problem die *optimale Teilstruktur* hat:

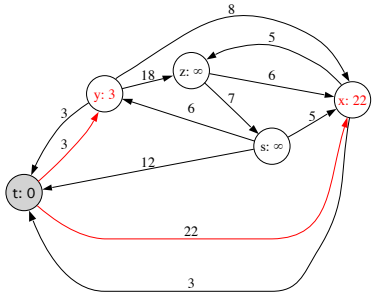
RELAX sowieso nicht mehr verändern.

[4] Der triviale Algorithmus REVERSE ist hier nicht ausgeführt.



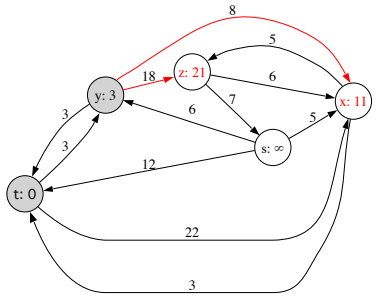
$Q = \{(s:\infty), (x:\infty), (y:\infty), (z:\infty)\}$
$S = \{(t:0)\}$

(a) Ausgangsgraph mit Startknoten t



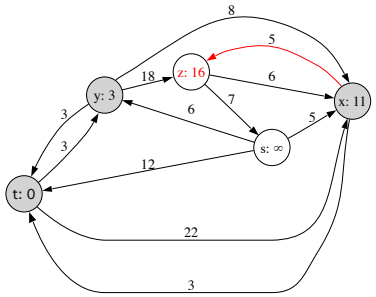
$Q = \{(s:\infty), (x:22), (y:3), (z:\infty)\}$
$S = \{(t:0)\}$

(b) y und x werden erreicht



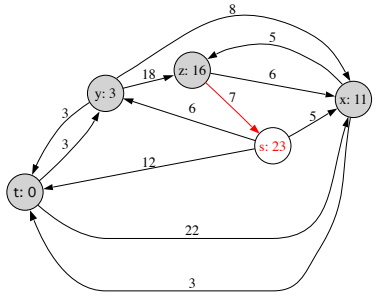
$Q = \{(s:\infty), (x:11), (z:21)\}$
$S = \{(t:0), (y:3)\}$

(c) z wird erreicht und x aktualisiert



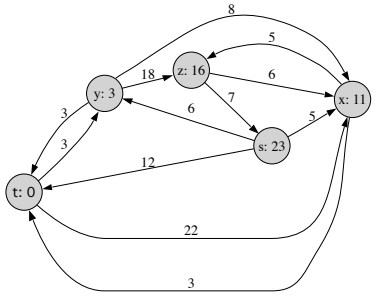
$Q = \{(s:\infty), (z:16)\}$
$S = \{(t:0), (y:3), (x:11)\}$

(d) z wird aktualisiert



$Q = \{(s:23)\}$
$S = \{(t:0), (y:3), (x:11), (z:16)\}$

(e) s wird erreicht



$Q = \emptyset$
$S = \{(t:0), (y:3), (x:11), (z:16), (s:23)\}$

(f) Ende

Abbildung 7.11: Funktionsweise des Dijkstra-Algorithmus

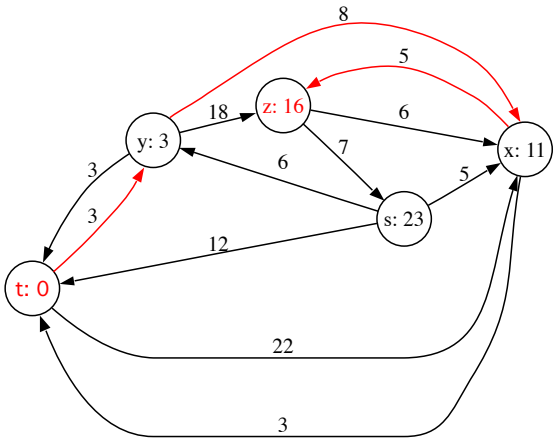


Abbildung 7.12: Kürzester Weg von t nach z

Lemma 7.1.4. *Sei $G = (V, E)$ gewichteter, gerichteter Graph mit Gewichtsfunktion $w : G \to \mathbb{R}^+$ und $p = (\nu_0, \ldots, \nu_k)$ ein kürzester Weg, dann ist $p = (\nu_i, \nu_{i+1}, \ldots, \nu_j)$ für beliebiges i, j mit $0 \leq i \leq j \leq k$ ein kürzester Weg.*

Beweis. Übung! □

Analyse

Korrektheit

Der Schlüssel zum Verständnis des Dijkstra-Algorithmus liegt in der optimalen Teilstruktur des Kürzeste-Weg-Problems sowie in einigen grundlegenden Eigenschaften von Graphen.

Lemma 7.1.5. *Der Dijkstra Algorithmus ist korrekt.*

Beweis. Zu zeigen ist

1. Der Algorithmus terminiert.
2. Nach Abbruch sind die Abstände korrekt berechnet.
3. Die Abfolge der Vorgänger liefert den kürzesten Weg.

Die erste Eigenschaft ist klar, denn Zeile 5 extrahiert jeweils genau einen Knoten pro Schleifendurchlauf aus der Prioritätswarteschlange Q. Da in Q nach Zeile 3 nichts mehr eingefügt ist, terminiert 4. Die zweite Eigenschaft ergibt sich aus folgender Überlegung:

Der Algorithmus verwaltet eine Schleifeninvariante $Q = V \setminus S$. Wir können uns S als „bekannte“ und Q als noch unbekannte zu testende Knoten vorstellen. Von den Knoten in S ist die Entfernung zu s bekannt und ändert sich nicht mehr, von den restlichen Knoten ist die tatsächliche Entfernung noch unbekannt; am Anfang beträgt also die Entfernung ∞, später – falls der Knoten bereits in „Reichweite“ von S, also auf dem Rand liegt – trägt sie den letzten bekannten, *vorläufigen* Wert. Abbildung 7.13 veranschaulicht den Rand.

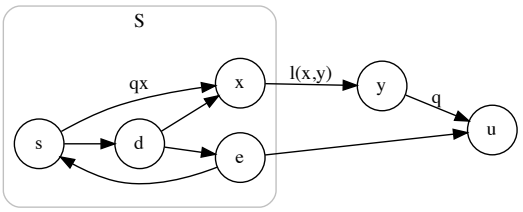


Abbildung 7.13: Der Rand: Bekannte und unbekannte Knoten

Wir führen nun eine Induktion über S aus – wir behaupten, dass für alle Knoten ν in S gilt $\nu.d$ ist der kürzester Weg vom Start. Das ist für $|S| = 1$ offensichtlich, denn der Startknoten hat $s.d = 0$. Für den Induktionsschritt $|S| = n$ auf $|S| = n + 1$ überlegen wir wie folgt. Der neue Knoten u ist der einzige, für den wir zeigen müssen, dass $u.d$ der kürzeste Weg ist. Wir nehmen an, dass das nicht der Fall ist, dann gibt es einen Weg q von s zu u für dessen Länge $l(q)$ gilt:

$$l(q) < u.d \tag{7.1}$$

Sei nun xy die erste Kante auf dem Weg von q, die S verläßt, wobei $x \in S$ sei, dann bezeichnen wir mit q_x den Weg vom Start s zu x. Es gilt dann:

$$l(xy) + l(q_x) \leq l(q) \tag{7.2}$$

Da $x.d$ der kürzeste Weg ist (Induktionsannahme!), haben wir

$$x.d \leq l(q_x). \tag{7.3}$$

Da aber y in der Adjazenzliste von $x \in S$ liegt, wurde y „relaxt“ und wir wissen, dass

$$y.d \leq x.d + l(xy). \tag{7.4}$$

Schlussendlich wissen wir aber auch, dass

$$u.d \leq y.d, \tag{7.5}$$

da der Algorithmus immer das minimale Element aus der Min-Prioirty-Queue entfernt. Kombinieren wir nun die Gleichungen 7.1– 7.5, so erhalten wir

$$u.d > l(q) \geq l(q_x) + l(xy) \geq x.d + l(xy) \geq y.d \geq u.d, \tag{7.6}$$

was ein Widerspruch ist. Also ist die Annahme, dass ein kürzerer Weg existiert falsch. (Man beachte, dass hier die Tatsache ausgenützt wird, dass die Funktion w positiv ist!) Da am Ende alle Knoten in S sind, ist der Beweis vollständig.

Die dritte Eigenschaft ergibt sich unmittelbar aus der Tatsache, dass der Vorgängergraph aus den kürzesten Wegen zusammengesetzt ist, wenn er aus Relaxationsschritten, für die $\nu.d = \delta(s, \nu)$ gilt, entstanden ist. □

Laufzeit

Die Laufzeit ergibt sich aus den folgenden Beobachtungen:

1. Jeder Knoten $u \in V$ wird zur Initialisierung genau einmal betrachtet.
2. Jeder Knoten $u \in V$ wird zur Menge Q genau einmal hinzugefügt und wieder entnommen.
3. Jeder Knoten $u \in V$ wird zur Menge S genau einmal hinzugefügt.
4. Die WHILE-Schleife läuft für jeden Knoten $u \in V$, also $|V|$ mal.
5. Daher wird EXTRACT-MIN $|V|$ mal aufgerufen.
6. Jede Kante ν der Adjazenzliste $Adj[u]$ wird genau einmal (gerichtete Graphen) oder zweimal (ungerichtete Graphen) in Zeile 7-8 des Algorithmus 7.6 untersucht.
7. Die FOR-Schleife läuft somit insgesamt $|E|$ mal für gerichtete Graphen und $2|E|$ mal für ungerichtete Graphen.
8. RELAX besitzt konstante Laufzeit.

In diesen Betrachtungen wurde die Laufzeit von Operationen auf der Prioritätswarteschlange wie EXTRACT-MIN noch nicht berücksichtigt. Die Performanz des Algorithmus hängt wesentlich von der Umsetzung der Prioritätswarteschlange Q ab.

1. Erster Fall: Prioritätswarteschlange implementiert als Feld.
   (a) INSERT und DECREASE-KEY sind $\Theta(1)$ für Feld-basierte Prioriätswarteschlangen.
   (b) EXTRACT-MIN ist $\Theta(|V|)$, da nach dem zu entnehmenden Element gesucht werden muss.

   Damit ergibt sich die Gesamtlaufzeit als $\Theta(|E| + |V|^2) = \Theta(|V|^2)$, da $|E| = \Theta(|V|^2)$.
2. Zweiter Fall: Prioritätswarteschlange implementiert als Heap-basierte Prioriätswarteschlange (siehe Abschnitt 7.6).
   (a) Zeit, den Heap zu bauen ist $\Theta(|V|)$.
   (b) DECREASE-KEY sind $\Theta(\lg(|V|))$.
   (c) EXTRACT-MIN ist $\Theta(\lg(|V|))$.

   Damit ergibt sich die Gesamtlaufzeit als $\Theta\left(|V| + (|V| + |E|)\lg(|V|)\right) = \Theta(|E|\lg(|V|))$ (Letztere Gleichheit gilt nur für zusammenhängende Graphen.)

Welches Verfahren ist nun besser? Dies hängt offensichtlich davon ab, ob $\Theta(|E|\lg(|V|)) < \Theta(|V|^2)$ ist oder äquivalent, ob $|E| \in o(|V|^2/\lg|V|)$. Letzteres gilt für die meisten sogenannten „dünnen“ (engl “sparse”) Graphen.

Alternative Algorithmen

Es gibt natürlich noch viele andere Kürzester Weg Algorithmen, s. u.a. [Sed90][5].

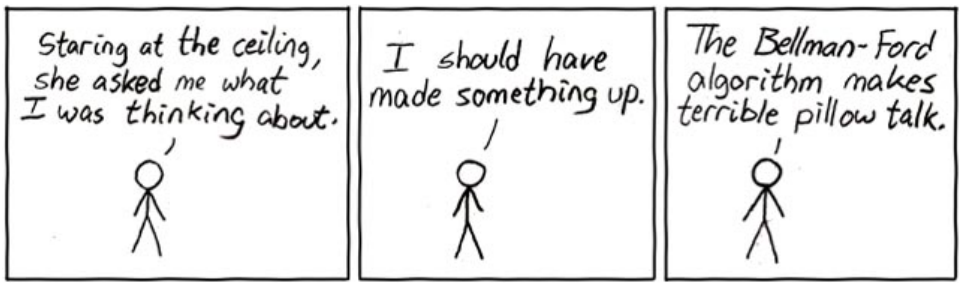


Abbildung 7.14: Pillowtalk, Quelle: http://xkcd.com

[5] die z. B. auch mit negativen Gewichten funktionieren oder die kürzesten Wege zwischen allen Knoten eines Graphen berechnen

A* Algorithmus

Die Grundidee des A* Algorithmus ist es, anstelle lediglich die Kosten für den Weg vom Start s bis zum gerade untersuchten Knoten u zu untersuchen, eine *Abschätzung* für den verbleibenden Weg vorzunehmen, und zwar mit Hilfe einer sogenannten Heuristik-Funktion $h : V \times V \to \mathbb{R}^+$. Mit anderen Worten, wir minimieren die Funktion f, die die Gesamtkosten vom Start s zum Ziel e angibt:

$$f(u) := w(s, u) + h(u, e)$$

Eine häufig verwendete Heuristikfunktion ist die (euklidische) Abstandsfunktion. Der A* Algorithmus sucht also – im Gegensatz zum Dijkstra – nicht blind, sondern besitzt Kenntnis über das Ziel. Man spricht daher auch von einem „informierten“ Suchalgorithmus. Man kann nicht beliebige Schätzfunktionen verwenden, sondern nur solche, für die gilt[6]:

1. h ist *zulässig*, d.h. h darf den Abstand niemals überschätzen und
2. h ist *monoton* (oder konsistent), d.h. h erfüllt die Relation $h(x, e) < w(x, y) + h(y, e)$.

Der Pseudocode ist in 7.8 angegeben. Wie man sieht, entspricht er weitgehend dem Dijkstra, insbesondere sind die Zeilen 10-13 identisch mit der RELAX Prozedur 7.4, wenn man als Heuristik $h \equiv 0$ nimmt. In diesem Sinn kann man also beim Dijkstra von einem degenerierten A* Algorithmus sprechen[7]. Der andere Unterschied besteht darin, dass der Dijkstra stets alle Knoten untersucht, während der A* wegen Zeile 8 abbricht, sobald der Zielknoten gefunden wurde. Die Berechnung des Weges erfolgt dann mit Hilfe des Backtrackings „rückwärts“ wie beim Dijkstra, siehe Algorithmus 7.7! (Falls beim Endknoten kein Vorgänger eingetragen ist so existiert kein Pfad – z.B. weil der Graph nicht zusammenhängend ist – und es muss ein Fehler ausgegeben werden.)

Für den Beweis der Korrektheit des A* Algorithmus sei auf die Literatur [PH72] verwiesen.

Das informierte Verhalten des A* Algorithmus wirkt auf den Betrachter fast wie intelligentes Vorgehen. Schöne Visualisierungen finden sich u.a. in

- `http://www.geosimulation.de/methoden/a_stern_algorithmus.htm`
- `https://www.youtube.com/watch?v=19h1g22hby8`

Der A* Algorithmus ist ein praktisch anwendbarer Algorithmus und wird in handelsüblichen Navigationsgeräten und Routenplanern verwendet.

[6] Jedenfalls in der hier angegeben Variante des A* Algorithmus.

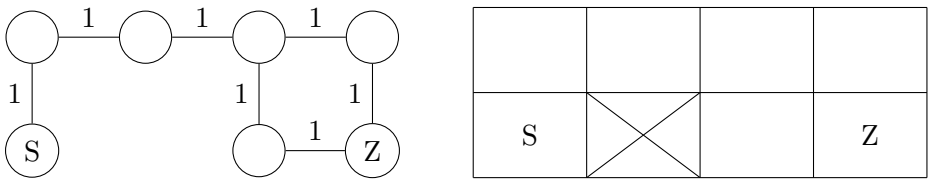
[7] Ebenso ist ein Algorithmus GREEDY, der ausschließlich die Heuristik verwendet, d.h. bei dem $g \equiv 0$ ist, denkbar. Dieser ist schnell, aber findet nicht immer den kürzesten Weg, wie in Abbildung 7.16 verdeutlicht.

Algorithmus 7.8 A*(G, w, h, s, e)

```
1:  INITIALIZE(G, s)
2:  S ← ∅                                   // Closed list
3:  Q ← G.V                                 // Initialize open list Q with all V ∈ G
4:  while Q ≠ ∅ do
5:      u ← EXTRACT-MIN(Q)                  // Extract node with minimum f value
6:      S ← S ∪ {u}
7:      if u = e then
8:          BREAK                           // Found path to end node
9:      for each vertex ν ∈ G.Adj[u] and ν ∉ S do
10:         if ν.d > u.d + w(u, ν) then
11:             ν.d ← u.d + w(u, ν)         // Update costs from start
12:             ν.f ← ν.d + h(ν, e)         // Update heuristics to end
13:             ν.π ← u
```

Bei den vielen Vorzügen, die der A* Algorithmus besitzt, stellt sich die Frage, ob und warum man den Algorithmus von Dijkstra einsetzen sollte. Der Vorteil, den der Algorithmus von Dijkstra bietet, ist dass er – im Gegensatz zu A* – keine Kenntnis des Zielorts besitzen muss. Das ist gar nicht so esoterisch wie es sich anhört, so kann z.B. eine Suche bei einem Computerspiel nach einem „Schatz“ in einem unbekannten Terrain vorgenommen werden und erst nach Auffinden des Schatzes ist der kürzeste Weg zu berechnen.

Zur Vereinfachung verwenden wir eine Matrix-Repräsentation eines Graphen. Die Kosten von einem Feld ins horizontal oder vertikal benachbarte Feld betragen immer 1. Diagonal existieren keine Verbindungen. In der Matrix gesperrte Felder (mit zwei diagonalen Linien durchgestrichen) existieren im Graphen nicht und sind daher auch für den Algorithmus nicht vorhanden. Ein Beispiel ist in Abbildung 7.15 zu finden mit Startknoten S und Zielknoten Z.



(a) Graph (b) Matrix-Repräsentation

Abbildung 7.15: Matrix-Repräsentation eines Graphen

Basierend auf Abbildung 7.15 verdeutlicht Abbildung 7.16 noch einmal die unterschiedlichen Suchstrategien. Beachten Sie, dass gesperrte Felder hier mit einem „X“ gekennzeichnet sind und die Pfeile hier zur besseren

Veranschaulichung in Richtung des Nachfolgers zum Ziel gerichtet sind, und nicht zum Vorgänger, wie dies in den Algorithmen realisiert wird. Beim A* Algorithmus werden die kursiv dargestellten Felder nur dann erreicht, wenn das jeweils darunter befindliche Feld in Zeile 5 des Algorithmus 7.8 entnommen wird.

6	5	4→	3→	2↓
5	4	3↑	X	1↓
4	3	2↑	X	e
5→	4→	3↑	X	1
s↑	X	X	X	2
7	6	5	4	3

(a) GREEDY

4	5	6	7	8
3	4	5	X	9
2	3	4	X	e
1	2	3	X	7↑
s↓	X	X	X	6↑
1→	2→	3→	4→	5↑

(b) DIJKSTRA

4+6	*5+5*	*6+4*		
3+5	4+4	5+3	X	
2+4	3+3	4+2	X	e
1+5	2+4	3+3	X	7+1↑
s↓	X	X	X	6+2↑
1+7→	2+6→	3+5→	4+4→	5+3↑

(c) A*

Abbildung 7.16: Unterschiedliche Kürzeste Wege Strategien

Der Organismus Physarum polycephalum kann auch kürzeste Wege finden und ist ein Einzeller, siehe `https://distribution.arte.tv/fiche/Blob__un_genie_sans_cerveau`, `https://en.wikipedia.org/wiki/Physarum_polycephalum`

7.2 Bäume

Bäume gehören zu den wichtigsten Datenstrukturen in der Informatik. Sie sind spezielle Graphen und kommen u.a. als

- Syntaxbäume
- Entscheidungsbäume
- Suchbäume

vor. Ein Verständnis der grundlegenden Eigenschaften von Bäumen ist daher für Informatiker unerlässlich. Zusammen mit den Bäumen werden wir auch Algorithmen zum Durchsuchen (Traversieren) von Bäumen vorstellen.

7.2.1 Definition und elementare Eigenschaften

Man kann Bäume als spezielle *Graphen* definieren:

Definition 7.2.1. *Ein Graph $G = (V, E)$ heißt* topologischer Baum*, wenn eine der folgenden äquivalenten Aussagen wahr ist:*

- *Zwischen je zwei Knoten von G gibt es genau einen Weg.*
- *G ist zusammenhängend und es gilt $|E| = |V| - 1$. (Es gibt immer eine Kante weniger als Knoten.)*
- *G enthält keinen Zyklus und es gilt $|E| = |V| - 1$.*
- *G ist zusammenhängend und azyklisch.*

Beachten Sie, dass in der Definition 7.2.1 die Struktur des Baumes definiert ist, aber nicht seine Wurzel. Jeder Knoten könnte nach dieser Definition die Wurzel sein.

Da viele Algorithmen die Wurzel als Einstiegspunkt in die Datenstruktur Baum benötigen, definieren wir Bäume nach [Knu97a] mit folgender *rekursiven* Definition 7.2.2:

Definition 7.2.2. *Eine endliche Menge T von Knoten heißt* Baum *genau dann wenn*

1. *Ein ausgezeichneter Knoten existiert, die* Wurzel*, und*
2. *die anderen Knoten in m disjunkte Mengen $(T_1, \ldots, T_m)$ aufgeteilt werden können, von denen jede wiederum ein Baum ist. Diese werden* Unterbäume *genannt.*

Diese rekursive Definition passt gut zu den rekursiven Algorithmen, mit denen wir Bäume konstruieren und traversieren werden. Abbildung 7.17 zeigt eine graphische Darstellung eines Baums. Neben der Wurzel sind die *Blätter* ausgezeichnet, die diejenigen Konten repräsentieren, die keine Unterbäume enthalten. Man beachte, dass die Unterbäume nicht notwendigerweise geordnet sind, ist die Ordnung entscheidend, spricht man von *geordneten* Bäumen.

Für weiterführende Literatur sei auf [CLR04] sowie auf das Modul „Diskrete Mathematik“ verwiesen!

7.2.2 Operationen

Man kann allgemeine Bäume strukturiert durchsuchen (traversieren). Die dabei verwendeten Algorithmen unterscheiden sich jedoch nicht wesentlich von den Algorithmen zur Traversierung von Binärbäumen, die wir im folgenden vorstellen werden.

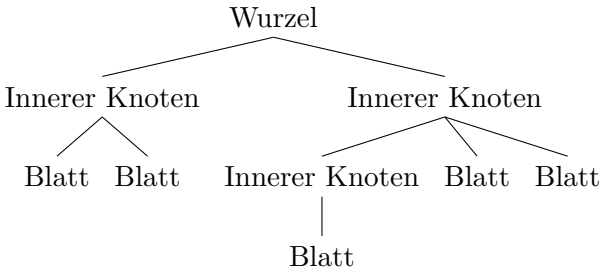


Abbildung 7.17: Ein Baum

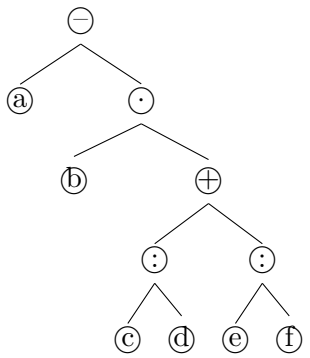


Abbildung 7.18: Ein Binärbaum

7.3 Binärbäume

7.3.1 Definition und elementare Eigenschaften

Wir haben Binärbäume schon benutzt, um Heaps zu definieren. Zur Erinnerung:

Definition 7.3.1. *Eine endliche Menge T von Knoten heißt* Binärbaum *genau dann wenn*

1. *die Menge T leer ist, oder*
2. *ein ausgezeichneter Knoten existiert, die* Wurzel *und*
3. *genau zwei disjunkte Binärbäume existieren, der* linke *und der* rechte Teilbaum.

Abbildung 7.18 zeigt eine graphische Darstellung eines Binärbaums. (Man beachte, dass Binärbäume, technisch gesehen, keine Spezialfälle von Bäumen sind, da sie leer sein können und da der linke und der rechte Baum *ausgezeichnet* sind.)

Der in Abbildung 7.18 gezeigte Baum entspricht dem algebraischen Aus-

druck

$$a - b\left(\frac{c}{d} + \frac{e}{f}\right)$$

was bereits ein Indiz dafür ist, wie nützlich Binärbäume sein können.

Definition 7.3.2. *Ein* binärer Suchbaum *ist ein binärer Baum, bei dem für jeden Knoten gilt: die Knoten des linken Teilbaums haben nur kleinere (oder gleiche*[8]*) Werte und die Knoten des rechten Teilbaums nur größere (oder gleiche) Werte als der Knoten selbst.*

Somit sind binäre Suchbäume eher von links nach rechts geordnet, während Heaps eher von oben nach unten geordnet sind.

Ein Beispiel für einen binären Suchbaum ist in Abbildung 7.19 dargestellt[9].

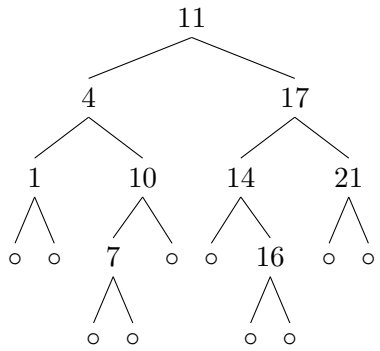


Abbildung 7.19: Binärer Suchbaum

7.3.2 Traversieren

Binäre Suchbäume kann man leicht durchsuchen bzw. systematisch durchqueren ("Walk the Tree"), wie der folgende Algorithmus 7.9 verdeutlicht.

Algorithmus 7.9 INORDER-TREEWALK(x)

```
1: if x ≠ NIL then
2:     INORDER-TREEWALK(x.left)
3:     print x.key                    // or do something else with x
4:     INORDER-TREEWALK(x.right)
```

[8] Ob man gleiche Werte erlaubt und ob diese links oder rechts eingefügt sind, ist reine Konvention und für alles weitere belanglos – man muss sich nur für eine Variante entscheiden und konsequent beibehalten!

[9] Fehlende Elemente (Kinder) werden durch das Zeichen ∘ dargestellt.

(Man kann diesen Algorithmus leicht auf allgemeine Bäume verallgemeinern, siehe Fragen und Aufgaben zum Selbststudium.)

Eine häufig verwendete Technik beim Entwurf von Datenstrukturen besteht darin, dass man die Funktion, die man bei der Traversierung des Baums ausführen möchte – im obigen Beispiel also die Funktion "print" *nicht* hardkodiert, sondern einen Zeiger auf eine Funktion auswertet. Damit kann die Datenstruktur flexibel je nach Situation mit unterschiedlichen Auswertungsfunktionen bestückt werden – eine Anwendung des Strategie-Entwurfmusters [GHJV95].

Lemma 7.3.1. *Wenn x die Wurzel eines Baums mit n Knoten bezeichnet, dann ist INORDER-TREEWALK von $\Theta(n)$ Komplexität.*

Beweis. Sei $T(n)$ die Laufzeit von INORDER-TREEWALK. Da alle n Knoten besucht werden, gilt $T(n) \in \Omega(n)$. Wir müssen also zeigen $T(n) \in \mathcal{O}(n)$. Es gilt $T(0) =: c$, $c > 0$, denn T benötigt konstante Zeit für den Test $x \neq NIL$. Sei $n > 0$, der linke Teilbaum habe k Knoten, der rechte also $n - k - 1$. Dann gilt:

$$T(n) \leq T(k) + T(n - k - 1) + d, \quad d > 0.$$

d bezeichnet eine obere Schranke für den Rumpf des Algorithmus ohne die rekursiven Aufrufe. Diese Rekursionsgleichung hat die Lösung $T(n) \leq (c + d)\, n + c$, wie wir mit Induktion beweisen: Für $n = 0$ gilt $T(0) = c \leq (c+d)\, 0 + c$. Angenommen, die Ungleichung gilt für alle $n' < n$ (Induktionsvoraussetzung), dann folgt

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &\leq ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c. \end{aligned}$$

□

In-order ist am gebräuchlichsten, aber es existieren noch "Pre-" und "Post-order-" Algorithmen (7.10 und 7.11).

Algorithmus 7.10 PREORDER-TREEWALK(x)

```
1: if x ≠ NIL then
2:     print x.key                          // or do something else with x
3:     PREORDER-TREEWALK(x.left)
4:     PREORDER-TREEWALK(x.right)
```

Selbstverständlich sind auch PREORDER-TREEWALK und POSTORDER-TREEWALK von $\Theta(n)$.

Algorithmus 7.11 POSTORDER-TREEWALK(x)

```
1: if x ≠ NIL then
2:     POSTORDER-TREEWALK(x.left)
3:     POSTORDER-TREEWALK(x.right)
4:     print x.key                    // or do something else with x
```

7.3.3 Suchen

Zur Illustration verschiedener Suchalgorithmen für Binäre Suchbäume benutzen wir den in Abbildung 7.20 dargestellten, einfachen Binären Suchbaum.

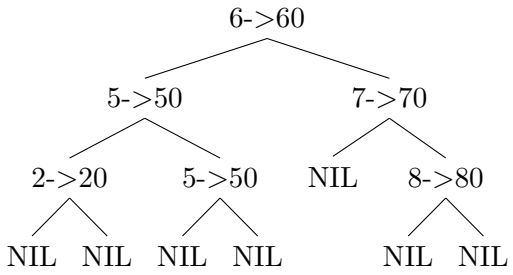


Abbildung 7.20: Ein Binärer Suchbaum

Eine mögliche Repräsentierung von Binärbäumen sind *Tupel*. Wir verwenden im folgenden Listing 7.1 eine Darstellung mit einem 4-er Tupel a_i:

1. a_0: enthält den Schlüssel.
2. a_1: enthält den Wert.
3. a_2: enthält den linken Teilbaum.
4. a_3: enthält den rechten Teilbaum.

Der in Abbildung 7.20 dargestellte Baum hat also folgende Darstellung:

```
(6, 60,
    (5, 50,
        (2, 20, None, None),
        (5, 50, None, None)
    ),
    (7, 70,
        None,
        (8, 80, None, None)
    )
)
```

Binäre Suchbäume lassen sich exzellent durchsuchen, wie der Algorithmus 7.12 verdeutlicht.

Algorithmus 7.12 TREE-SEARCH(x, k)

1: **if** $x = NIL$ **or** $k = x.key$ **then**
2: **return** x
3: **if** $k < x.key$ **then**
4: **return** TREE-SEARCH$(x.left, k)$
5: **else**
6: **return** TREE-SEARCH$(x.right, k)$

Damit läßt sich eine Suche einfach implementieren wie in Listing 7.1 dargestellt.

Algorithmus 7.13 TREE-SEARCH-P(x, k)

Listing 7.1 Tree Search – Version 1 (Ausführbarer Pseudocode)

```
1  class Node():
2      def __init__(self, key, value=0, left=None, right=None):
3          self.key = key
4          self.value = value
5          self.left = left
6          self.right = right
7
8  def search_binary_tree(node, key):
9      if node is None:
10         return None          # key not found
11     if key < node.key:
12          return search_binary_tree(node.left, key)
13     elif key > node.key:
14          return search_binary_tree(node.right, key)
15     else:                    # key is equal to node key
16          return node.value   # found key
17
18 # walk the tree in-order (prints tree "mirrored")
19 def print_tree(node, indent=""):
20     if node is not None:
21         indent += "␣␣␣␣"
22         print_tree(node.left, indent)
23         print("%s%s:␣%s" %(indent[:-4], node.key, node.value))
24         print_tree(node.right, indent)
25     else:
26         print("%s%s" %(indent, "NIL"))
```

Darstellungen mit Hilfe von Tupeln sind in Skriptsprachen und funktionalen Programmiersprachen sehr gebräuchlich. In prozeduralen oder objektorientierten Sprachen wird eher eine Darstellung mit Hilfe von Abstrakten

Datentypen verwendet. Man kann einen Knoten etwa wie folgt als Struktur, Klasse oder Record implementieren:

Key	Schlüssel
Value	Wert
Left	Zeiger auf Wurzel des linken Teilbaums
Right	Zeiger auf Wurzel des rechten Teilbaums

Damit läßt sich eine Suche ebenso einfach implementieren wie in Listing 7.2 dargestellt. Ebenso kann man leicht mittels einer rekursiv definierten Routine `constructTree` zwischen den beiden Versionen konvertieren.

Algorithmus 7.14 TREE-SEARCH-V2-P(x, k)

Listing 7.2 Tree Search – Version 2 (Ausführbarer Pseudocode)

```
1  def search_binary_tupletree(tupletree, key):
2      if tupletree is None:
3          return None              # key not found
4      if key < tupletree[0]:
5           return search_binary_tree(tupletree[2], key)
6      elif key > tupletree[0]:
7           return search_binary_tree(tupletree[3], key)
8      else:                        # key is equal to node key
9           return tupletree[1]     # found key
10
11 # construct tree from tuple representation
12 def constructTree(tpl):
13     left, right = None, None
14     if tpl[2] is not None:
15         left = constructTree(tpl[2])
16     if tpl[3] is not None:
17         right= constructTree(tpl[3])
18     return Node(tpl[0], tpl[1], left, right)
19
20
21 tupletree = (6, 60,
22                 (5, 50,
23                     (2, 20, None, None),
24                     (5, 50, None, None)),
25                 (7, 70,
26                     None,
27                     (8, 80, None, None)))
28
29 tree = constructTree(tupletree)
30 print("in-oder:")
31 print_tree(tree)
32 print(search_binary_tupletree(tupletree, 6))
```

Eine iterative Version der Suche ist im Algorithmus TREE-SEARCH-ITERATIVE 7.15 gezeigt – die Rekursion wird „abgerollt“.

Algorithmus 7.15 TREE-SEARCH-ITERATIVE(x, k)

```
1: while x ≠ NIL and k ≠ x.key do
2:     if k < x.key then
3:         x ← x.left
4:     else
5:         x ← x.right
6: return x
```

Der Algorithmus TREE-MINIMUM 7.16 liefert das minimale Element in einem Baum.

Algorithmus 7.16 TREE-MINIMUM(x)

```
1: while x.left ≠ NIL do
2:     x ← x.left
3: return x
```

Analog liefert TREE-MAXIMUM 7.17 das maximale Element in einem Baum.

Algorithmus 7.17 TREE-MAXIMUM(x)

```
1: while x.right ≠ NIL do
2:     x = x.right
3: return x
```

Lemma 7.3.2. *TREE-MINIMUM und TREE-MAXIMUM sind korrekt und $\mathcal{O}(h)$, wobei h die Höhe des Baums bezeichnet.*

Beweis. Übung! □

7.3.4 Einfügen

Wir können Knoten in Bäume mit Hilfe der Prozedur TREE-INSERT 7.18 einfügen, die in Abbildung 7.21 veranschaulicht ist (die Abbildung zeigt das Einfügen des Elements 12 in den Baum).

7.3.5 Löschen

Übungsaufgabe!

Algorithmus 7.18 TREE-INSERT(T, z)

```
 1: y, x ← NIL, T.root
 2: while x ≠ NIL do                    // search predecessor for new key
 3:     y ← x
 4:     if z.key < x.key then
 5:         x ← x.left
 6:     else
 7:         x ← x.right
 8: if y = NIL then                     // y is predecessor of new key
 9:     T.root ← z                      // Tree T was empty
10: else if z.key < y.key then
11:     y.left ← z
12: else
13:     y.right ← z
```

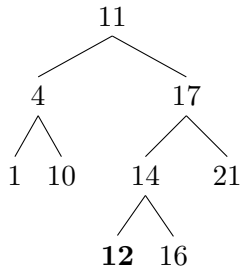


Abbildung 7.21: TREE-INSERT

7.4 Zufällig aufgebaute Bäume

7.4.1 Zufall und Form

Mit Hilfe der Prozedur TREE-INSERT können wir Bäume iterativ aufbauen. Dabei kommt es wesentlich auf die Reihenfolge der Einfügungen an. Angenommen, wir fügen in der folgenden (zufälligen) Reihenfolge ein: $(14, 33, 1, 25, 12, 4)$, dann ergibt sich der in Abbildung 7.22 gezeigte Baum.

Lemma 7.4.1. *Seien alle Schlüssel der Menge* $\{1, \ldots, 50\}$ *gleichwahrscheinlich. Dann ist die mittlere Suchzeit – gemessen in der Anzahl der Schlüsselvergleiche – für den Baum in Abbildung 7.22*

- *bei einer erfolgreichen Suche* $\frac{5}{2} = 2{,}5$ *und*
- *bei einer erfolglosen Suche* $\frac{31}{11} \approx 2{,}82$.

Beweis. Berechnung der Anzahl der erforderlichen Schlüsselvergleiche für

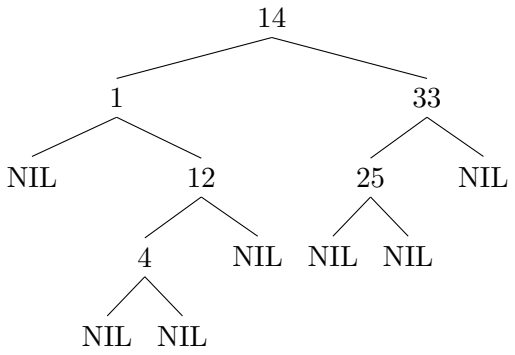


Abbildung 7.22: Ein zufällig aufgebauter Binärer Suchbaum

jeden der gleichwahrscheinlichen Schlüssel.

- Erfolgreiche Suche: Die Wurzel besteht aus einem Schlüssel, in der Ebene darunter befinden sich zwei, in der nächsten Ebene ebenfalls zwei, und in der untersten Ebene ein Schlüssel. Die Ebene, also die Anzahl der Knoten auf dem Weg zum gesuchten Schlüssel, entspricht der Suchzeit.
$$\frac{1 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + 1 \cdot 4}{6} = \frac{15}{6}$$

- Erfolglose Suche: Wir betrachten die Schlüssel, die nicht im Baum vorhanden sind. Für die Schlüssel 2 und 3 werden vier Schlüsselvergleiche benötigt, für 5 bis 11 ebenfalls. Für die Schlüssel 13, 15 bis 24 und 26 bis 32 werden drei Schlüsselvergleiche benötigt und für 34 bis 50 zwei.
$$\frac{9 \cdot 4 + 18 \cdot 3 + 17 \cdot 2}{44} = \frac{124}{44}$$

□

Angenommen, wir fügen die gleichen Schlüssel in der folgenden, geordneten Reihenfolge ein: $(1, 4, 12, 14, 25, 33)$, dann ergibt sich der in Abbildung 7.23 gezeigte Baum. Der Baum ist zu einer Liste degeneriert.

Lemma 7.4.2. *Seien alle Schlüssel der Menge* $\{1, \ldots, 50\}$ *gleichwahrscheinlich. Dann ist die mittlere Suchzeit – gemessen in der Anzahl der Schlüsselvergleiche – für den Baum in Abbildung 7.23*

- *bei einer erfolgreichen Suche* $\frac{7}{2} = 3{,}5$ *und*
- *bei einer erfolglosen Suche* $\frac{223}{44} \approx 5{,}07$.

Beweis. Analog Beweis zu Lemma 7.4.1. □

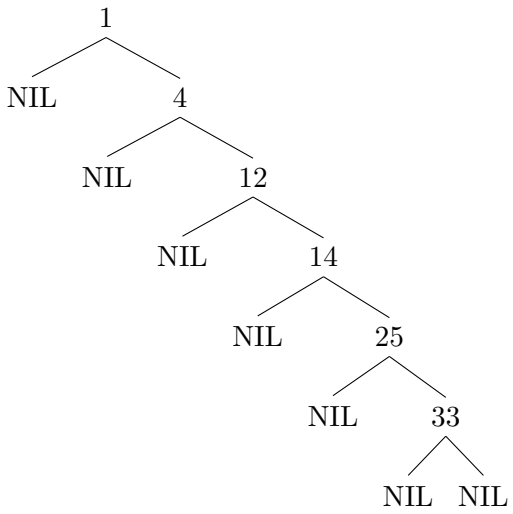


Abbildung 7.23: Ein durch Einfügen sortierter Schlüssel aufgebauter Binärer Suchbaum

Kennzeichnend für die Güte eines Baums ist also u.a. das Verhältnis der Höhe zur Anzahl der Knoten sowie die Höhe der einzelnen Äste zueinander. Man kann versuchen, durch verschiedene Maßnahmen die Qualität eines Baums zu verbessern bzw. zu erhalten, siehe Abschnitt „Balancierte Bäume“.

7.4.2 Analyse

Theorem 7.4.1. *Der Erwartungswert für die Höhe des zufällig durch Einfügen von n Schlüsseln aufgebauten Baums ist $\mathcal{O}(\lg(n))$.*

Beweis. [CLR04], Seite 300. □

7.5 Balancierte Bäume

Wie wir gesehen haben, können durch Einfügen (und Löschen) Bäume mit unerwünschten Eigenschaften erzeugt werden. Daher kann man Techniken verwenden, die verhindern, dass Bäume z. B. zu Listen degenerieren.

7.5.1 Rot-Schwarz-Bäume

Rot-Schwarz-Bäume bestehen – wie der Name sagt – aus rot und schwarz eingefärbten Knoten.

Definition 7.5.1. *Ein Binärer Suchbaum hat die Rot-Schwarz-Eigenschaft genau dann, wenn gilt:*

1. *Jeder Knoten ist entweder rot oder schwarz.*

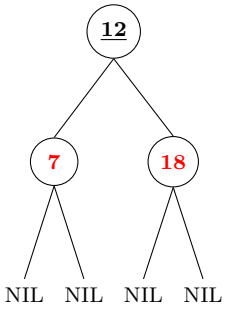


Abbildung 7.24: Ein einfacher Rot-Schwarz-Baum

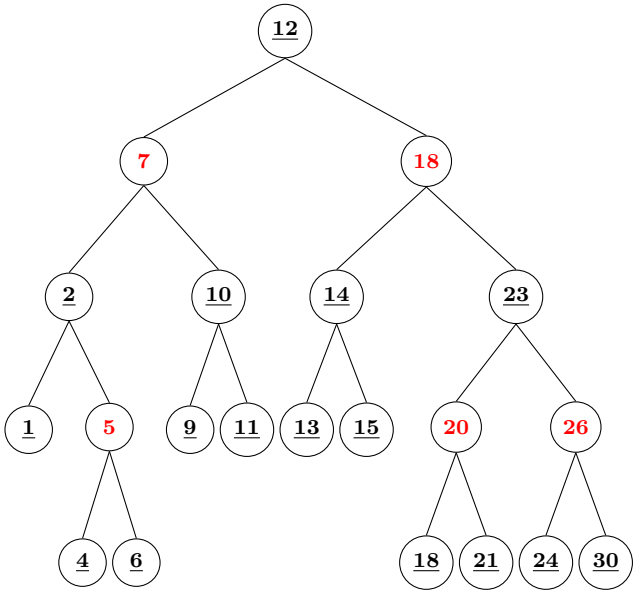


Abbildung 7.25: Ein weiterer Rot-Schwarz-Baum

2. *Die Wurzel ist schwarz.*
3. *Jedes Blatt ist schwarz*[10].
4. *Wenn ein Knoten rot ist, dann sind beide Kinder schwarz.*
5. *Für jeden Knoten enthalten die Pfade vom Knoten zu* allen *Blättern gleich viele schwarze Knoten*[11].

Abbildung 7.24 und Abbildung 7.25[12] zeigen Beispiele für Rot-Schwarz-Bäume.

[10] Es ist Konvention, die NIL Knoten als schwarz gefärbte Blätter aufzufassen.

[11] Die Anzahl der schwarzen Knoten in diesen Pfaden wird auch als *Schwarzhöhe* bezeichnet.

[12] Aus Gründen der Übersichtlichkeit wurden hier die schwarzen! NIL Blätter weggelassen!

Ein Rot-Schwarz-Baum kann *nicht* degenerieren:

Lemma 7.5.1. *Ein Rot-Schwarz-Baum mit n internen Knoten hat eine Höhe von maximal $2\lg(n+1)$.*

Beweis. Siehe [CLR04]. □

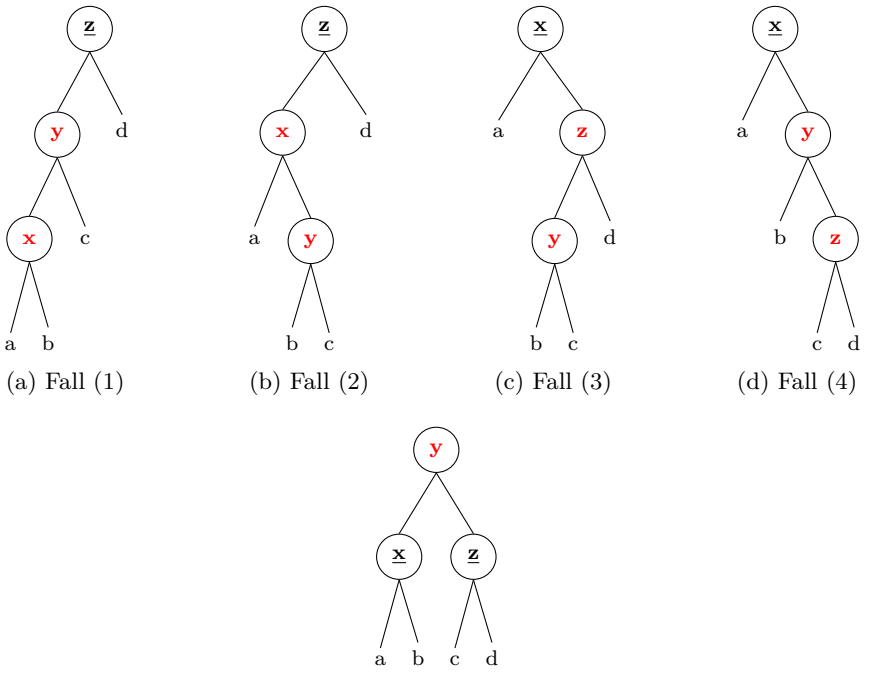
Natürlich sind Einfüge- und Löschoperationen für Rot-Schwarz-Bäume komplizierter, da man ggf. die Rot-Schwarz-Eigenschaft wiederherstellen muss – ein Prozess, der *Rotation* genannt wird (s. [CLR04]).

Wir behandeln im folgenden nur die Einfügeoperation und diese – abweichend etwa von der Darstellung im Cormens Buch [CLR04] – *erheblich* vereinfacht. Die nachfolgende Idee basiert auf der Darstellung von Okasaki [Oka99] und ist durch funktionale Programmiersprachen inspiriert. Die Grundidee kann jedoch auch bei imperativen Sprachen wie C verwendet werden. Der Grund für das erheblich kompliziertere Verfahren in [CLR04] liegt in Optimierungsverfahren, die für imperative Programmiersprachen Sinn ergeben, nicht jedoch für funktionale. Das hier vorgestellte Verfahren ist aber erheblich übersichtlicher und verdeutlicht die Grundidee besser. Da die erzielten Performanzverbesserungen das asymptotische Verhalten nicht beeinflussen – dieses ist bei beiden Verfahren immer $\mathcal{O}(\lg(n))$ – läßt sich der nachstehende Algorithmus auch praktisch verwenden.

Grundlegend für die Funktionsweise ist die in Abbildung 7.27 dargestellte Rotation, die in *allen* vier Fällen identisch den Baum „repariert“ und das Problem nach oben „verschiebt“, bis es entweder an einem schwarzen Knoten terminiert oder rekursiv an der Wurzel angekommen ist, die dann einfach schwarz gefärbt wird. Die dargestellten Fälle (1) bis (4) stellen die Situation dar, in der entweder durch Einfügen des Knotens x (Fall (1)), des Knotens y (Fall (2) und Fall (3)) bzw. des Knotens z (Fall (4)) oder durch rekursives „nach oben schieben“ eines Knotens eine reparaturbedürftige Situation entstanden ist: Es liegt eine Verletzung der Rot-Schwarz-Eigenschaft 4.) „Wenn ein Knoten rot ist, sind beide Kinder schwarz“ vor. Alle vier Fälle werden auf gleiche Art und Weise repariert. Dabei kann eine Verletzung der Rot-Scharz-Eigenschaften an der Wurzel des Teilbaums ausgelöst werden. Das ist jedoch nicht weiter schlimm, denn durch mehrfaches, rekursives Anwenden von BALANCE wird diese Situation bis maximal zur Wurzel des Baums verschoben, wo sie durch Schwarzfärben der Wurzel aufgelöst wird.

Das Einfügen in einen Rot-Schwarz-Baum erfolgt in zwei Schritten. Zunächst wird der Algorithmus 7.18 TREE-INSERT für Binärbäume verwendet, um ein Element x an der richtigen Stelle einzusetzen, und zwar stets als rotes Element. Danach wird der Algorithmus 7.19 BALANCE[13] aufgerufen, um

[13]Dabei ist zu beachten, dass im Algorithmus mit x der einzusetzende Knoten und mit



(a) Fall (1) (b) Fall (2) (c) Fall (3) (d) Fall (4)

(e) Nach Rebalancing

Abbildung 7.26: Rot-Schwarz-Rebalancing

die ggf. verletzte Rot-Schwarz-Eigenschaft wiederherzustellen. Man beachte, dass TREE-INSERT den Baum (rekursiv oder iterativ) herabsteigt und BALANCE den Baum vom eingesetzten Blatt x wieder heraufsteigt, und zwar solange wie notwendig (u.U. bis zur Wurzel, wo man ggf. die Wurzel schwarz färbt, s. Zeile 3 des Algorithmus 7.19). Damit gilt also auch, dass das Einfügen in Rot-Schwarz-Bäume in $\mathcal{O}(\lg(n))$ ist. Der Hilfsalgorithmus MAKE-CHILD(x, y) sorgt dafür, dass der Knoten x den Knoten y als Elternteil bekommt (und somit y den Knoten x als linkes oder rechtes Kind).

Sie finden ein Implementierungsbeispiel für einen Rot-Schwarz-Baum in Python im Code `redblacktree.py` (s. E-Learning).

7.5.2 AVL Bäume

Eine weitere Möglichkeit Degeneration zu verhindern, besteht natürlich darin, explizit vorzuschreiben, dass die Höhen der linken und rechten Teilbäume nicht zu stark differieren dürfen.

y sein Elternkonten (Parent) bezeichnet wird. Diese Notation ist nur für die Fälle Fall (1) und Fall (2) identisch mit Abbildung 7.26. (Die Notation in Abbildung 7.26 wurde so gewählt um die Auflösung identisch zu bezeichnen.)

Algorithmus 7.19 BALANCE(x)

```
 1: y ← x.parent
 2: if y = Nil then
 3:     x.color ← BLACK
 4:     return x
 5: z ← y.parent
 6: if z = Nil then
 7:     y.color ← BLACK
 8:     return y
 9: if x.color = RED and y.color = RED then
10:     // Rot-Schwarz-Eigenschaft wieder herstellen
11:     if x.key < y.key and y.key < z.key then
12:         x.color, z.color ← BLACK        // Fall 1
13:         z.left, y.right ← y.right, z
14:         MAKE-CHILD(y, z.parent)
15:         z.parent ← y
16:         return BALANCE(y)
17:     else if x.key > y.key and y.key < z.key then
18:         y.color, z.color ← BLACK        // Fall 12
19:         y.right, x.left ← x.left, y
20:         z.left, x.right ← x.right, z
21:         MAKE-CHILD(x, z.parent)
22:         z.parent, y.parent ← x
23:         return BALANCE(y)
24:     else if x.key < y.key and y.key ≥ z.key then
25:         y.color, z.color ← BLACK        // Fall 3
26:         y.left, x.right ← x.right, y
27:         z.right, x.left ← x.left, z
28:         MAKE-CHILD(x, z.parent)
29:         z.parent, y.parent ← x
30:         return BALANCE(y)
31:     else if x.key > y.key and y.key ≥ z.key then
32:         x.color, z.color ← BLACK        // Fall 4
33:         z.right, y.left ← y.left, z
34:         MAKE-CHILD(y, z.parent)
35:         z.parent ← y
36:         return BALANCE(y)
37: else
38:     return BALANCE(y)                   // Rekursiver Aufruf
```

Definition 7.5.2. *Ein Baum ist ein AVL-Baum*[14] *genau dann, wenn die Differenz der Höhen der linken und rechten Teilbäume nicht größer ist als* 1.

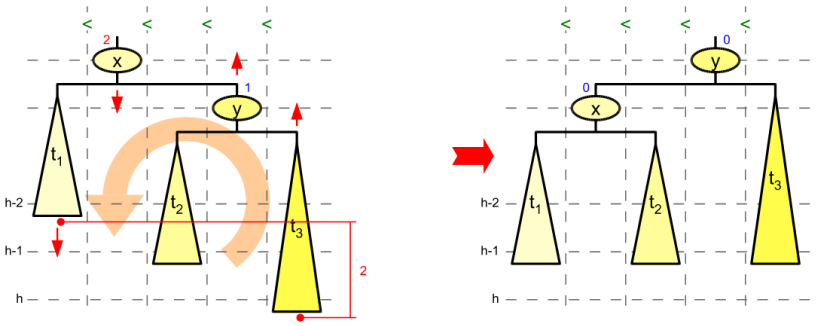


Abbildung 7.27: AVL Rotation, Quelle: `http://commons.wikimedia.org/wiki/File:AVL-double-rl.png`

7.6 Haufen und Halden (Heaps)

Wir haben Haufen bereits beim Heapsort-Algorithmus kennengelernt.

7.6.1 Implementierungsaspekte

Bisher haben wir Heaps und Prioritätswarteschlangen nur vom algorithmischen Standpunkt aus betrachtet und dabei *wesentliche* Implementierungsaspekte vernachlässigt. So sind wir bisher davon ausgegangen, dass die Elemente der Prioritätswarteschlange bzw. des (unterliegenden) Heaps als Schlüssel die natürlichen Zahlen haben und darüber hinaus keine Attribute enthalten. In der Praxis wollen wir Heaps und Prioritätswarteschlangen natürlich dazu benutzen, die für Applikationen oder Betriebssysteme wichtigen Objekte zu verwalten. Daher müssen wir die Datenstruktur so entwerfen, dass zusätzliche Attribute abgespeichert und verwaltet werden können. Folgende Aspekte sind dabei zu berücksichtigen:

1. Die Prioritätswarteschlange bzw. der Heap muss in der Lage sein, eine Datenstruktur zu speichern, die es ermöglicht, zusätzliche Attribute zu speichern. Dies kann geschehen indem
   (a) eine Referenz (Zeiger) oder
   (b) eine Struktur (anstelle lediglich einer natürlichen Zahl) abgespeichert wird.

[14] Der Name AVL leitet sich von den Erfindern Adelson-Velsky und Landis ab, die diese Datenstruktur 1962 entwickelt haben.

2. Die Anwendungen müssen einen Zugriff (englisch “handle”) auf das Element der Prioritätswarteschlange bzw. des Heaps haben.
3. Ggf. (anwendungsabhängig) muss die Prioritätswarteschlange bzw. der Heap eine Möglichkeit haben, auf ein Element anhand eines Attributes (anstelle des Index) zugreifen zu können. (Wir haben diese Datenstruktur u.a. beim Dijkstra-Algorithmus einsetzen können – in der Version der Prioritätswarteschlange Q mit Heap.)

Aus all diesen Gründen ist die Implementierung einer Prioritätswarteschlange bzw. eines Heaps anspruchsvoll und in der Praxis existieren viele verschiedene Implementierungen derselben unterliegenden Algorithmen für unterschiedliche Datenstrukturen und APIs. So können Prioritätswarteschlangen bzw. Heaps als verkettete Listen auftauchen, aber auch als Prioritätswörterbücher, die zusätzliche Strukturen wie Hashmaps verwenden.

Ein weiterer, in der Praxis wichtiger Gesichtspunkt ist die Entscheidung, das API prozedural (so wie wir es hier weitgehend halten) zu gestalten oder objektorientiert. Bei einem objektorientierten Entwurf werden meistens die unterliegenden Datenstrukturen in ein Typen- bzw. Klassendesign überführt und die Algorithmen als Methoden direkt in den Klassen untergebracht oder in externen Klassen, z. B. unter Verwendung des Strategieentwurfsmusters (s. [GHJV95]).

7.7 Mengen

Mengen sind wichtige Datentypen, die wir hier aber aus Zeitgründen nicht behandeln. Mengen müssen so implementiert werden, dass ein Element nur maximal einmal in der Menge vorkommen kann.

7.8 Einsatzgebiete von Komplexen Datenstrukturen

Die hier vorgestellten komplexen Datenstrukturen sind die Grundlage für viele Verfahren und Technologien wie Parser und Compiler, Datenbanken, Optimierungsverfahren etc. pp.

7.9 C-Code

Listing 7.3: Datenstrukturen für Graphen (BFS) – C-Code

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define MAXSIZE 64
4 #define INF 9999
```

```
5
6  typedef struct Node Node;
7  typedef struct Edge Edge;
8
9  enum {
10     WHITE = 0,
11     GRAY = 1,
12     BLACK = 2,
13 };
14
15 struct Node {
16     int id;      /* id */
17     int d;       /* distance */
18     int color;   /* color */
19     Node *pi;    /* previous node */
20     Edge *edge; /* adjacency list (edges) */
21 };
22
23 struct Edge {
24     Edge *next; /* next element in list */
25     Node *to;   /* node, edge points to */
26 };
27
28 Node *q[MAXSIZE];
29 int front = -1;
30 int rear = -1;
31
32 void enqueue(Node *node)
33 {
34     if (rear == MAXSIZE-1){
35         printf("Queue Overflow\n");
36     }
37     else {
38         if (front == -1) /* Queue is initially empty */
39             front = 0;
40         rear++;
41         q[rear] = node;
42     }
43 }
44
45 Node* dequeue()
46 {
47     Node *node;
48     if (front == -1 || front > rear)
49     {
50         printf("Queue Underflow\n");
51     }
52     else
53     {
54         node = q[front];
55         front++;
56         return node;
57     }
58     return node;
```

```
59  }
60
61  int qisempty(){
62      return front > rear;
63  }
64
65  Node *addNode(int id)
66  {
67      Node *node = (Node *) malloc(sizeof(Node));
68      node->id = id;
69      node->d = INF;
70      node->color = WHITE;
71      node->pi = NULL;
72      node->edge = NULL;
73      printf("Added node %i\n", node->id);
74      return node;
75  }
76
77  Edge *addEdge(Node *nodeFrom, Node *nodeTo)
78  {
79      Edge *ep = (Edge *) malloc(sizeof(Edge));
80      ep->to = nodeTo;
81      ep->next = NULL;
```

Listing 7.4: BFS – C-Code (Algorithmus)

```
1   void bfs(Node *node)
2   {
3       node->color = GRAY;
4       node->d = 0;
5       node->pi = NULL;
6       enqueue(node);
7       while(!qisempty()){
8           Node *np = dequeue();
9           Edge *edge = np->edge;
10          while(edge != NULL){
11              Node *tp = edge->to;
12              if(tp->color == WHITE){
13                  tp->color == GRAY;
14                  tp->d = np->d + 1;
15                  tp->pi = np;
16                  enqueue(tp);
17              }
18              edge = edge->next;
19          }
20          np->color = BLACK;
21          Node* pre = NULL;
22          int pi = 0;
23          if(pre = np->pi)
24              pi = pre->id;
25          printf("Discovered node %i, distance d: %i, predecessor: %i\n",
26                  np->id, np->d, pi);
27      }
28  }
```

```
29
30
31  int main (int argc, const char * argv[]) {
32      printf("Hello, BFS!\n");
33      Node *np1, *np2, *np3, *np4, *np5, *np6;
34      np1 = addNode(1);
35      np2 = addNode(2);
36      np3 = addNode(3);
37      np4 = addNode(4);
38      np5 = addNode(5);
39      np6 = addNode(6);
40
41      addEdge(np1, np2);
42      addEdge(np1, np3);
43      addEdge(np2, np3);
44      addEdge(np3, np4);
45      addEdge(np4, np2);
46      addEdge(np4, np5);
47      addEdge(np5, np6);
48      bfs(np1);
49      return 0;
50  }
```

Listing 7.5: Datenstrukturen für Graphen (DFS) – C-Code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Node Node;
5  typedef struct Edge Edge;
6
7  enum {
8      WHITE = 0,
9      GRAY = 1,
10     BLACK = 2,
11 };
12
13 struct Node {
14     int id;      /* id */
15     int d;       /* distance */
16     int td;      /* time discovered */
17     int tf;      /* time finished */
18     int color;   /* color */
19     Node *pi;    /* previous node */
20     Edge *edge; /* adjacency list (edges) */
21 };
22
23 struct Edge {
24     Edge *next; /* next element in list */
25     Node *to;   /* node, edge points to */
26 };
27
28 Node *addNode(int id)
29 {
```

```
30      Node *node = (Node *) malloc(sizeof(Node));
31      node->id = id;
32      node->d = 0;
33      node->td = 0;
34      node->tf = 0;
35      node->color = WHITE;
36      node->pi = NULL;
37      node->edge = NULL;
38      printf("Added node %i\n", node->id);
39      return node;
40  }
41
42  Edge *addEdge(Node *nodeFrom, Node *nodeTo)
43  {
44      Edge *ep = (Edge *) malloc(sizeof(Edge));
45      ep->to = nodeTo;
46      ep->next = NULL;
47      Edge *edge = nodeFrom->edge;
48      if(edge)
49          ep->next = edge;
50      nodeFrom->edge = ep;
51      printf("Added edge from %i to %i\n", nodeFrom->id, nodeTo->id);
52      return ep;
53  }
```

Listing 7.6: DFS – C-Code (Algorithmus)

```
1   void dfsvisit(Node *node, int *time)
2   {
3       (*time)++;
4       node->td = *time;
5       node->color = GRAY;
6       printf("Node %i, visited at t: %i\n", node->id, node->td);
7
8       Edge *edge = node->edge;
9       while(edge != NULL){
10          Node *to = edge->to;
11          if(to->color == WHITE){
12              to->pi = node;
13              dfsvisit(to, time);
14          }
15          edge = edge->next;
16      }
17      node->color = BLACK;
18      (*time)++;
19      node->tf = *time;
20      Node* pre = NULL;
21      int pi = 0;
22      if((pre = node->pi))
23          pi = pre->id;
24      printf("Node %i, discovered at t: %i, finished at t: %i, predecessor: %i\n",
25              node->id, node->td, node->tf, pi);
26  }
27
```

```
28 void dfs(Node *node[], int len, int *time)
29 {
30     for(int i = 0; i < len; i++){
31         Node *np = node[i];
32         if(np->color == WHITE){
33             dfsvisit(node[i], time);
34         }
35     }
36 }
37
38
39 int main (int argc, const char * argv[]) {
40     printf("Hello DFS!\n");
41
42     Node *np1, *np2, *np3, *np4, *np5, *np6;
43     np1 = addNode(1);
44     np2 = addNode(2);
45     np3 = addNode(3);
46     np4 = addNode(4);
47     np5 = addNode(5);
48     np6 = addNode(6);
49
50     addEdge(np1, np2);
51     addEdge(np1, np3);
52     addEdge(np2, np3);
53     addEdge(np3, np4);
54     addEdge(np4, np2);
55     addEdge(np5, np4);
56     addEdge(np5, np6);
57     Node *npa[6] = {np1, np2, np3, np4, np5, np6};
58     int time = 0;
59     dfs(npa, 6, &time);
60     return 0;
61 }
```

7.10 Ausführbarer Pseudo-Code

Listing 7.7: DIJKSTRA

```
1  import sys
2  import math
3  INF = math.inf
4
5  def parent(i):
6      return i//2
7
8  def left(i):
9      return 2*i+1 # lists start with 0!
10
11 def right(i):
12     return 2*i+2 # lists start with 0!
13
14 def heapify(a, h, heapsize, i):
15     l = left(i)
16     r = right(i)
17     if l < heapsize and a[l][1] < a[i][1]:
```

```
18          minimum = l
19      else:
20          minimum = i
21      if r < heapsize and a[r][1] < a[minimum][1]:
22          minimum = r
23      if minimum != i:
24          h[a[i][0]], h[a[minimum][0]] = minimum, i
25          a[i], a[minimum] = a[minimum], a[i]
26          heapify(a, h, heapsize, minimum)
27
28  def build_heap(a, h):
29      for i in range(len(a)):
30          h[a[i][0]] = i
31      heap_size = len(a)
32      for i in range(len(a)//2, 0, -1):
33          heapify(a, h, heap_size, i-1)
34
35  def min_heap(a, h):
36      return a[0]
37
38  def extract_min(a, h):
39      if len(a) == 0:
40          return None
41      minimum = a[0]
42      del h[a[0][0]]
43      del a[0]
44      build_heap(a, h)
45      return minimum
46
47  def insert_heap(a, h, x):
48      a.append([x[0], INF])
49      h[x[0]] = len(a)-1
50      decrease_key_heap(a, h, x)
51
52  def decrease_key_heap(a, h, x):
53      i = h[x[0]]
54      k = x[1]
55      if k > a[i][1]:
56          raise Exception("new␣key␣bigger␣than␣current␣key")
57      a[i][1] = k
58      while i > 0 and a[parent(i)][1] > a[i][1]:
59          a[parent(i)], a[i] = a[i], a[parent(i)]
60          h[a[i][0]] = i
61          i = parent(i)
62
63  # the Dijkstra algorithm
64  def dijkstra(G, start, end=None):
65      D = {} # distance
66      P = {} # dictionary of predecessors
67      for vertex in list(G.keys()):
68          D[vertex] = INF
69      D[start] = 0
70      Q = []
71      h = {}
72      for vertex in list(G.keys()):
73          if vertex == start:
74              insert_heap(Q, h, [vertex, 0])
75          else:
76              insert_heap(Q, h, [vertex, INF])
77      while len(Q) != 0:
78          print(Q)
79          print(D)
```

```
80          u = extract_min(Q, h)
81          for v in G[u[0]]: # relax
82              uvLength = D[u[0]] + G[u[0]][v]
83              if uvLength < D[v]:
84                  D[v] = uvLength
85                  P[v] = u
86                  if v in h:
87                      print("relaxing %s" % v)
88                      decrease_key_heap(Q, h, [v, uvLength])
89                  else:
90                      insert_heap(Q, h, [v, uvLength])
91      return D, P
92
93  def shortestPath(G, start, end):
94      D, P = dijkstra(G, start, end)
95      Path = []
96      while 1:
97          Path.append(end)
98          if end == start: break
99          end = P[end][0]
100     Path.reverse()
101     return Path
102
103 # generates dot presentation from Graph
104 def print_graph(G):
105     print("digraph states {\nsize=\"4,3\";\nrankdir=LR;\nnode [shape=circle];")
106     for node in list(G.keys()):
107         print("%s [label = \"%s\"];" % (node, node))
108     for nodeFrom in list(G.keys()):
109         for nodeTo in list(G[nodeFrom].keys()):
110             print("%s -> %s [label = \"%s\"];" % (nodeFrom, nodeTo, G[nodeFrom][nodeTo]))
111     print("}")
112
113 # Graph from Cormen et al, p. 599
114 G = {'s':{'y': 5, 't': 10}, 'y': {'t': 3, 'z': 1, 'x': 9}, 't': {'y': 2, 'x': 1},
115          'x': {'z': 4}, 'z': {'x': 6, 's':7} }
116 G2 = {'s':{'y': 6, 't': 12, 'x': 5}, 'y': {'t': 3, 'z': 18, 'x': 8},
117           't': {'y': 3, 'x': 22}, 'x': {'z': 5, 't': 3}, 'z': {'x': 6, 's':7} }
118 #print_graph(G)
119 #print_graph(G2)
120
121 print(dijkstra(G2, 't'))
122 print(shortestPath(G2, 't', 'z'))
123 print(print_graph(G))
```

7.11 Fragen und Aufgaben zum Selbststudium

1. Geben Sie drei Beispiele für nicht-lineare Datenstrukturen an!
2. Begründen Sie die Einträge in Tabelle 7.3.
3. Überlegen Sie, welche Kanten im Graphen 7.7 eliminiert oder hinzugefügt werden müssten, damit ein Kantenzug, der alle Kanten genau einmal durchläuft, und ein Eulerkreis entsteht.
4. Geben Sie einen Pseudocode für TREE-DELETE(T, z) an!
   Tipp: Unterscheiden Sie die Fälle
   (a) z hat keine Kinder

(b) z hat nur ein Kind

(c) z hat zwei Kinder (schwieriger Fall)

5. Geben Sie ein Beispiel dafür an, dass die Prozedur TREE-INSERT anstelle eines Baums eine Verkettete Liste erzeugt!

6. Schreiben Sie eine rekursive Version von TREE-INSERT!

7. Sortieren Sie ein ein Feld A wie folgt:

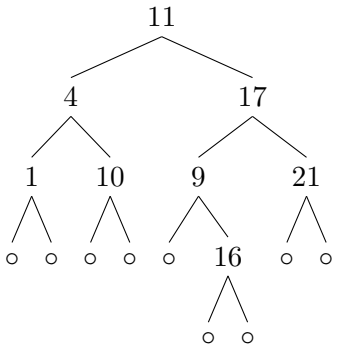
   (a) Erzeugen Sie mit Hilfe von TREE-INSERT einen binären Suchbaum

   (b) Geben Sie mit Hilfe des Algorithmus INORDER-TREE-WALK alle Elements aus

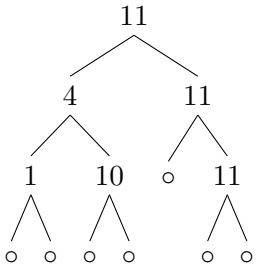
   Zeigen Sie, dass der Algorithmus korrekt ist! (Tipp: Überlegen Sie sich, wann genau *print* $x.key$ für ein Element aufgerufen wird!)

8. Im obigen Fall, welche Komplexität hat der Algorithmus im Worst und welche im Best Case?

9. Ist der folgende Baum ein Binärer Suchbaum?



10. Ist der folgende Baum ein Binärer Suchbaum?



11. Erstellen Sie mit Hilfe der Prozedur TREE-INSERT(T, z) (Algorithmus 7.8) aus folgenden Elementen einen Baum

$$(29, 42, 16, 33, 23, 25, 37, 50, 17, 51, 10, 30, 14, 45, 7) \ .$$

12. Wie sieht der Baum aus wenn folgende Elemente mit der Prozedur TREE-INSERT(T, z) eingefügt werden

$$(2, 8, 19, 20, 35, 51, 55, 60, 79)?$$

13. Wiederholen Sie die beiden vorherigen Aufgaben mit einem Rot-Schwarz-Baum! Welche Unterschiede stellen Sie fest?

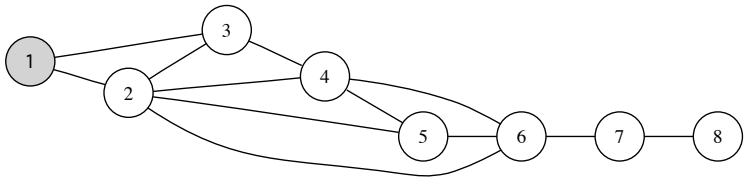
14. Machen Sie sich klar, warum die in dargestellte Rotation bzw. der Algorithmus BALANCE die Rot-Schwarz-Eigenschaft wiederherstellt!

15. Verallgemeinern Sie die Algorithmen INORDER-, POSTORDER- und PREORDER-TREEWALK auf beliebige Bäume!

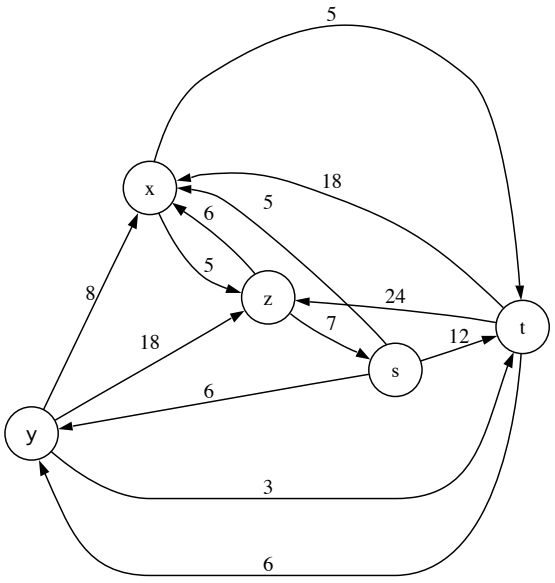
16. Beschreiben Sie die Idee des BFS Algorithmus in eigenen Worten!

17. Beschreiben Sie die Idee des DFS Algorithmus in eigenen Worten!

18. Führen Sie den BFS für den folgenden Graphen „per Hand“ aus!



19. Berechnen Sie den kürzesten Weg zwischen s und t für den folgenden Graphen mit Hilfe des Dijkstra-Algorithmus „per Hand“!



Kapitel 8

Dynamische Programmierung

8.1 Rückblick Fibonacci

Wir haben bereits in einem früheren Kapitel die Fibonacci-Zahlen *rekursiv* berechnet. Der Algorithmus ist aber sehr ineffizient, wie das nächste Lemma zeigt!

Lemma 8.1.1. *FIB ist in der Komplexitätsklasse* $\mathcal{O}(F_n)$.

Beweis. Übung! □

F_n wächst nämlich exponentiell – das sieht man mit der Formel von Moivre-Binet, aus der folgt

$$F_n = \lfloor 1/\sqrt{5}\Phi^n + 1/2 \rfloor,$$

mit $\Phi = (1 + \sqrt{5})/2)$; dabei hängt Φ mit dem goldenen Schnitt zusammen! Beispiel: $F_{45} = 1.134.903.170$, d.h. man ist schon in der Größenordnung von einer Milliarde Berechnungsschritten!

Warum ist FIB so ineffizient? Antwort: Weil bereits bekannte Resultate immer wieder (neu) berechnet werden, wie Abbildung 8.1 verdeutlicht!

Wir können dies vermeiden, in dem wir *bereits bekannte* Lösungen *speichern* und das Endergebnis so systematisch aufbauen. Dies ist in der deutlich effizienteren Variante FIB-ARRAY (s. Algorithmus 8.1) gezeigt, die ein Feld verwendet, in dem die bereits bekannten Fibonacci-Zahlen gespeichert werden.

Dieser Algorithmus ist effizienter, wie das nächste Lemma zeigt:

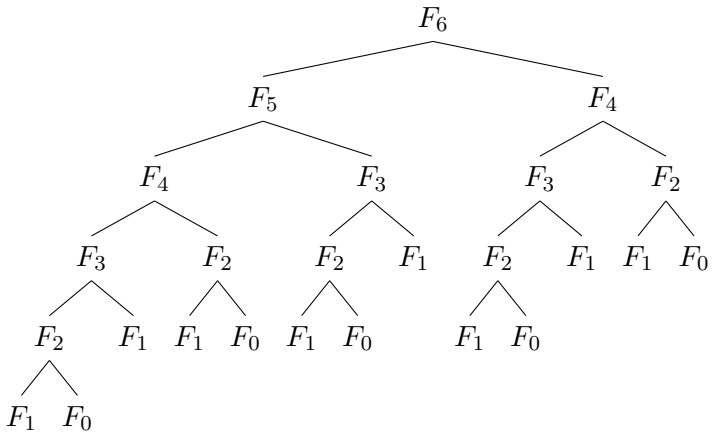


Abbildung 8.1: Fibonacci Berechnungsbaum über Rekursion

Algorithmus 8.1 FIB-ARRAY(n)

```
1: a[0], a[1] ← 0, 1
2: i ← 2
3: while i ≤ n do
4:     a[i] ← a[i − 1] + a[i − 2]
5:     i ← i + 1
6: return a[n]
```

Lemma 8.1.2. *Die Laufzeit von FIB-ARRAY ist in der Komplexitätsklasse* $\mathcal{O}(n)$.

Beweis. Übung! □

Die Fibonacci-Zahlen lassen sich also in linearer[1] Zeit berechnen! Das obige Verfahren geht aber verschwenderisch mit dem Speicherplatz um, da das Feld sämtliche Zahlen F_i für $i < n$ berechnet. Eine genaue Analyse zeigt, dass man sich nur die jeweils beiden letzten Berechnungen merken muss, wenn man lediglich an F_n interessiert ist. Diese Variante ist in FIB-ITER (s. Algorithmus 8.2) dargestellt.

Die hier entwickelten Ideen sind fundamental für die sogenannte *dynamische Programmierung*, die wir in diesem Kapitel behandeln werden.

[1] Es geht sogar in $\mathcal{O}(\ln n)$ – das behandeln wir hier aber nicht!

Algorithmus 8.2 FIB-ITER(n)

```
1: if n ∈ {0, 1} then
2:     return n
3: a, b ← 1, 1
4: while n > 2 do
5:     c ← a + b
6:     a, b ← b, c
7:     n ← n − 1
8: return b
```

8.2 Longest Common Subsequence (LCS) als Beispiel für Dynamische Programmierung

Der Begriff Dynamische Programmierung wurde in den 1940er Jahren Richard Bellman [Bel58] eingeführt, der diese Methode u.a. auf die Regelungstheorie anwendete. Dynamische Programmierung kann dann erfolgreich eingesetzt werden, wenn das Optimierungsproblem aus vielen gleichartigen Teilproblemen (s.u.) besteht, und wenn eine[2] optimale Lösung des Problems sich aus den optimalen Lösungen der Teilprobleme zusammensetzen läßt. Zuerst werden die optimalen Lösungen der kleinsten Teilprobleme berechnet und diese dann zu einer Lösung eines größeren Teilproblems zusammengesetzt. Dabei werden einmal berechnete Teilergebnisse gespeichert, um sie bei nachfolgenden Berechnungen wiederverwenden zu können anstatt sie jedes Mal neu zu berechnen.

Ein schönes Beispiel für den Einsatz der Technik der dynamischer Programmierung ist das LCS Verfahren zur Bestimmung von übereinstimmenden Teilsequenzen.

8.2.1 Charakterisierung einer LCS

In biologischen Anwendungen möchte man häufig eine DNA von zwei (verschiedenen) Organismen vergleichen. Dazu müssen Sequenzen der Form

TGCGTCCATT und ACCGTTGCGTCCAGCTGC

miteinander verglichen werden.

Definition 8.2.1. *Seien $X = (x_1, \ldots, x_n)$ und $Z = (z_1, \ldots, z_k)$ Sequenzen*[3]*. Z ist* Teilsequenz *von X genau dann wenn es eine steigende Sequenz $(i_1, \ldots, i_k)$ von Indizes von X gibt mit*

$$x_{i_j} = z_j \quad \forall j \in (1, \ldots, k)$$

[2] Es kann mehrere optimale Lösungen geben!

[3] Man beachte, dass die Sequenzen bei 1 starten!

Beispiel: Die Sequenz $Z = CGAT$ ist eine Teilsequenz von

$$X = ACCGTCGAGGCGT$$

mit der zugehörigen Indexsequenz $(2, 4, 8, 13)$.

Definition 8.2.2. *Wenn X, Y und Z gegeben sind und Z sowohl von X als auch von Y Teilsequenz ist, sprechen wir von einer* gemeinsamen Teilsequenz. *Eine Sequenz Z ist eine* längste gemeinsame Teilsequenz *(englisch "Longest Common Subsequence" oder LCS), wenn jede andere gemeinsame Teilsequenz nicht länger als Z ist. Man beachte, dass es mehrere LCS geben kann.*

Beispiel: $X = ACGT$ und $Y = AGCTA$ haben zwei LCS, ACT und AGT.

Theorem 8.2.1 (Optimale Teilstruktur einer LCS)**.** *Seien $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ Sequenzen und sei $Z = (z_1, \ldots, z_k)$ eine LCS von X und Y.*

1. *Falls $x_m = y_n$, dann gilt $z_k = x_m = y_n$ und Z_{k-1} ist eine LCS von X_{m-1} und Y_{n-1}.*
2. *Falls $x_m \neq y_n$, dann folgt aus $z_k \neq x_m$, dass Z eine LCS von X_{m-1} und Y ist.*
3. *Falls $x_m \neq y_n$, dann folgt aus $z_k \neq y_n$, dass Z eine LCS von X und Y_{n-1} ist.*

Beweis. Es gilt

1. Wenn $z_k \neq x_m$ wäre, dann könnten wir $x_m = y_n$ an Z anhängen im Widerspruch zu Z ist LCS. Also gilt $z_k = x_m = y_n$. Z.z. $Z_{k-1} := (z_1, \ldots, z_{k-1})$ ist eine LCS von X_{m-1} und Y_{n-1}. Falls das nicht gilt, dann gäbe es eine gemeinsame Teilsequenz W von X_{m-1} und Y_{n-1} mit Länge größer als $k - 1$. Wir hängen $x_m = y_n$ an und erhalten eine gemeinsame Teilsequenz von X und Y mit Länge größer als k, Widerspruch!
2. Wenn $z_k \neq x_m$, dann ist Z gemeinsame Teilsequenz von X_{m-1} und Y. Gäbe es gemeinsame Teilsequenz W von X_{m-1} und Y mit Länge größer als k, dann wäre W auch gemeinsame Teilsequenz von X_m und Y, Widerspruch!
3. Analog.

□

8.2.2 Rekursionsgleichung

Wir bezeichnen mit $c[i,j]$ die Länge einer LCS von X_i und Y_j[4].

Lemma 8.2.1. *Es gilt die Rekursionsgleichung*

$$c[i,j] = \begin{cases} 0, & \textit{wenn } i = 0 \textit{ oder } j = 0 \\ c[i-1,j-1]+1, & \textit{wenn } i,j > 0 \textit{ und } x_i = y_j \\ \max(c[i-1,j], c[i,j-1]), & \textit{wenn } i,j > 0 \textit{ und } x_i \neq y_j \end{cases}$$

Beweis. Folgt unmittelbar aus der optimalen Teilstruktur! □

Wie ersichtlich und in Abbildung 8.1 veranschaulicht hängt $c[i,j]$ nur von drei benachbarten Feldern ab.

$c[i-1,j-1]$	$c[i-1,j]$
$c[i,j-1]$	$c[i,j]$

Tabelle 8.1: Ausschnitt aus der Matrix zur LCS-Berechnung

8.2.3 Berechnung einer LCS

Mithilfe der Rekursionsgleichung könnte man leicht einen rekursiven Algorithmus aufstellen. Dieser könnte aber – abhängig von der Eingabe – wegen des dritten Falls der Rekursionsgleichung für $c[i,j]$ exponentielles Laufzeitverhalten, was man wie folgt sehen kann. Aus der Rekursionsgleichung folgt für die Laufzeit die folgende Rekursionsgleichung, wenn man $n := i + j$ setzt:

$$T(n) = 2\,T(n-1) + \Theta(1),$$

woraus man mit Hilfe der Substitutionsmethode oder über Iteration die Lösung $T(n) \in \Theta(2^n)$ berechnet.

Da aber insgesamt nur $|X| \cdot |Y|$ Werte berechnet werden müssen, geht das bei einem Problem mit dieser Rekursion besser, in dem man die sogenannte *Dynamische Programmierung* einsetzt.

Nachfolgend zunächst der Pseudocode des Algorithmus LCS-LENGTH 8.3. Er verwendet zwei Tabellen c und b. In c wird jeweils die oben definierte Länge gespeichert, und zwar von $(0,0)$ aus startend nach rechts und aufsteigend. Die Hilfstabelle b speichert Informationen über die Fallunterscheidung (grob gesprochen, speichern wir dort einen Zeiger auf dasjenige Element, das wir in obiger Fallunterscheidung ausgewählt haben):

Der ausführbare Pseudocode ist in Listing 8.1 dargestellt.

[4]Hier und im folgenden beginnen die Sequenzen wieder mit 0 und enden mit $n-1$ bzw. $k-1$.

Algorithmus 8.3 LCS-LENGTH(X,Y)

1: $m \leftarrow length[X]$
2: $n \leftarrow length[Y]$
3: **for** $i = 0$ to m **do**
4: $\quad c[i, 0] \leftarrow 0$
5: **for** $j = 1$ to n **do**
6: $\quad c[0, j] \leftarrow 0$
7: **for** $i = 1$ to m **do** // Matrix zeilenweise aufbauen
8: $\quad$ **for** $j = 1$ to n **do**
9: $\quad\quad$ **if** $x_{i-1} = y_{j-1}$ **then**
10: $\quad\quad\quad c[i, j] \leftarrow c[i-1, j-1] + 1$
11: $\quad\quad\quad b[i, j] \leftarrow$ "$\nwarrow$"
12: $\quad\quad$ **else if** $c[i-1, j] \geq c[i, j-1]$ **then**
13: $\quad\quad\quad c[i, j] \leftarrow c[i-1, j]$
14: $\quad\quad\quad b[i, j] \leftarrow$ "$\uparrow$"
15: $\quad\quad$ **else**
16: $\quad\quad\quad c[i, j] \leftarrow c[i, j-1]$
17: $\quad\quad\quad b[i, j] \leftarrow$ "$\leftarrow$"

Algorithmus 8.4 LCS-LENGTH-P(X,Y)

Listing 8.1 LCS-Länge (Ausführbarer Pseudocode)

```
1  def lcs_length(x, y):
2      m, n = len(x) + 1, len(y) + 1
3      c = [[0 for j in range(n)] for i in range(m)]
4      b = [["" for j in range(n)] for i in range(m)]
5      for i in range(1, m):
6          for j in range(1,n):
7              if x[i-1] == y[j-1]:
8                  c[i][j] = c[i-1][j-1] + 1
9                  b[i][j] = "NW"  # Upleft
10             elif c[i-1][j] >= c[i][j-1]:
11                 c[i][j] = c[i-1][j]
12                 b[i][j] = "NO"  # Up
13             else:
14                 c[i][j] = c[i][j-1]
15                 b[i][j] = "WE"  # Left
16     return c, b
```

Mit den Sequenzen $X = TGCGTCCAT$ und $Y = TACGTGCGCT$ (und der Lösung $LCS = TCGTCCT$) erhält man die Tabellen c (s. 8.2) und b (s. 8.3).

Man kann nun die Tabelle b verwenden, um eine LCS TCGTCCT zu konstruieren, wie in Tabelle 8.4 dargestellt.

Tabelle 8.2: LCS Länge c

	y_j	T	A	C	G	T	G	C	G	C	T
x_i	0	0	0	0	0	0	0	0	0	0	0
T	0	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	2	2	2	2	2	2	2
C	0	1	1	2	2	2	2	3	3	3	3
G	0	1	1	2	3	3	3	3	4	4	4
T	0	1	1	2	3	4	4	4	4	4	5
C	0	1	1	2	3	4	4	5	5	5	5
C	0	1	1	2	3	4	4	5	5	6	6
A	0	1	2	2	3	4	4	5	5	6	6
T	0	1	2	2	3	4	4	5	5	6	7

Tabelle 8.3: Hilfstabelle b fürs Backtracing

	y_j	T	A	C	G	T	G	C	G	C	T
x_i											
T		↖	←	←	←	↖	←	←	←	←	↖
G		↑	↑	↑	↖	←	↖	←	↖	←	←
C		↑	↑	↖	↑	↑	↑	↖	←	↖	←
G		↑	↑	↑	↖	←	↖	↑	↖	←	←
T		↖	↑	↑	↑	↖	←	←	↑	↑	↖
C		↑	↑	↖	↑	↑	↑	↖	←	↖	↑
C		↑	↑	↖	↑	↑	↑	↖	↑	↖	←
A		↑	↖	↑	↑	↑	↑	↑	↑	↑	↑
T		↖	↑	↑	↑	↖	↑	↑	↑	↑	↖

Dazu muss man „rückwärts“ gehen, wie der Pseudocode 8.5 zeigt. Der Algorithmus muss wie folgt aufgerufen werden: PRINT-LCS(b, X, m, n).

Algorithmus 8.5 PRINT-LCS(b, X, i, j)

```
1: if i = 0 or j = 0 then
2:     return
3: if b[i, j] = "↖" then
4:     PRINT-LCS(b, X, i − 1, j − 1)
5:     PRINT x_i
6: else if b[i, j] = "↑" then
7:     PRINT-LCS(b, X, i − 1, j)
8: else                                   // b[i, j] = "←"
9:     PRINT-LCS(b, X, i, j − 1)
```

Ausführbarer Pseudocode ist in Listing 8.2 dargestellt.

Tabelle 8.4: Gesamtdarstellung

	y_j	T	A	C	G	T	G	C	G	C	T
x_i	0	0	0	0	0	0	0	0	0	0	0
T	0	1 ↖	1 ←	1 ←	1 ←	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ↖
G	0	1 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ↖	2 ←	2 ↖	2 ←	2 ←
C	0	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	2 ↑	3 ↖	3 ←	3 ↖	3 ←
G	0	1 ↑	1 ↑	2 ↑	3 ↖	3 ←	3 ↖	3 ↑	4 ↖	4 ←	4 ←
T	0	1 ↖	1 ↑	2 ↑	3 ↑	4 ↖	4 ←	4 ←	4 ↑	4 ↑	5 ↖
C	0	1 ↑	1 ↑	2 ↖	3 ↑	4 ↑	4 ↑	5 ↖	5 ←	5 ↖	5 ↑
C	0	1 ↑	1 ↑	2 ↖	3 ↑	4 ↑	4 ↑	5 ↖	5 ↑	6 ↖	6 ←
A	0	1 ↑	2 ↖	2 ↑	3 ↑	4 ↑	4 ↑	5 ↑	5 ↑	6 ↑	6 ↑
T	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑	5 ↑	5 ↑	6 ↑	7 ↖

Algorithmus 8.6 PRINT-LCS-P(b, X, i, j)

Listing 8.2 PRINT-LCS-P (Ausführbarer Pseudocode)

```
1  def printlcs(b, x, i, j):
2      if i == 0 or j == 0:
3          return ""
4      if b[i][j] == "NW":
5          return printlcs(b, x, i-1, j-1) + x[i-1]
6      elif b[i][j] == "NO":
7          return printlcs(b, x, i-1, j)
8      else:
9          return printlcs(b, x, i, j-1)
```

8.2.4 Analyse

Die Komplexität von LCS-LENGTH ergibt sich aus dem nachstehenden Lemma:

Lemma 8.2.2. *Die Laufzeit von LCS-LENGTH beträgt* $\Theta(mn)$.

Beweis. Folgt unmittelbar aus den verschachtelten Schleifen. □

8.2.5 Optimierung

Der obige Algorithmus läßt sich noch hinsichtlich Speicherverbrauch verbessern. Zunächst bemerkt man, dass jeder Eintrag $c[i, j]$ nur von drei weiteren Einträgen abhängt, nämlich von $c[i-1, j-1]$, $c[i-1, j]$ und $c[i, j-1]$. Daher benötigt man die vollständige Tabelle b gar nicht, sondern nur diese Elemente. Somit kann der Platzbedarf reduziert werden. Am asymptotischen Platzbedarf ändert dies jedoch nichts, da man immer noch die Tabelle c benötigt. Wenn man nur die Länge der LCS benötigt kann man auch den Platzbedarf durch c auf $\Theta(m+n)$ reduzieren; für Details sei auf [CLR04] verwiesen.

8.3 Schnittoptimierung

Ein weiteres schönes Beispiel für eine Dynamische Programmierung ist die Schnittoptimierung, bei der es darum geht ein Gut, wie z. B. einen Draht so in n Teile zu zerlegen, dass die Summe der Verkaufspreise für die Einzelteile maximiert wird. Für Details sei auf [CLR04] und auf den Beispielcode `cutrod.py` (s. E-Learning) verwiesen.

8.4 Theorie Dynamischer Programmierung

Die Theorie und abstrakten Grundprinzipien der Dynamischen Programmierung vertiefen wir in Kapitel 9.

8.5 Fragen und Aufgaben zum Selbststudium

1. Gegeben seien die Sequenzen $X = (A, B, C, B, D, A, B)$ und $Y = (B, C, D, B)$, sowie $Z = (B, D, A, B)$. Zeigen Sie, dass sowohl Y als auch Z Teilsequenzen von X sind und geben Sie die jeweiligen Indexsequenzen an!

2. Gegeben seien die Sequenzen $X = (A, B, C, B, D, A, B)$ und $Y = (B, D, C, A, B, A)$. Welche Länge hat eine längste gemeinsame Teilsequenz von X und Y? Geben Sie (mindestens) eine LCS an!

3. Bestimmen Sie eine LCS der Sequenzen $X = (1, 0, 0, 1, 0, 1, 0, 1, 1)$ und $Y = (0, 1, 0, 1, 1, 0, 1, 1, 0, 1)$!

4. Lesen Sie unmittelbar aus der Tabelle 8.3. eine LCS für der Sequenzen $X = TGCGTCCA$ und $Y = TACGTGC$ ab!

5. In Zeile 14 im Algorithmus 8.3 LCS-LENGTH(X, Y) wird für alle $c[i-1, j] \geq c[i, j-1]$ ein "$\uparrow$", in Zeile 17 für $c[i-1, j] < c[i, j-1]$ ein "$\leftarrow$" gesetzt, und im Algorithmus 8.4 PRINT-LCS(b, X, i, j) wird dies dann benutzt, um *eine* LCS auszugeben. Modifizieren Sie beide Algorithmen so, dass *alle* möglichen LCS ausgegeben werden. Hinweis: Sie sollten dazu nicht nur ein "$\uparrow$", ein "$\leftarrow$" oder ein "$\nwarrow$" speichern, sondern *alle* Möglichkeiten, falls $c[i-1, j] = c[i, j-1]$. Für die PRINT-LCS Prozedur ist es dann am einfachsten, wenn Sie eine Rekursion verwenden.

Kapitel 9

Algorithmusdesign

In diesem Kapitel rekapitulieren wir noch einmal das bisher gelernte und fassen die Entwurfs- (Design-) Techniken für Algorithmen zusammen. Außerdem illustrieren wir die Techniken an ausgewählten Beispielen.

9.1 Entwurfsprinzipien

In diesem Kapitel stellen wir allgemeine Entwurfsprinzipien für Algorithmen vor. Da die Datenstrukturen häufig die Algorithmen prägen oder überhaupt erst ermöglichen, ist ein wesentlicher Bestandteil des Entwurfes von Algorithmen auch der Entwurf der zugrunde liegenden Datenstrukturen.

Weiterführende Literatur zum Thema Entwurfsprinzipien sind zum Beispiel [Ott98] und [Lev05]. Aber auch in Kapitel 14 lernen Sie, wie man Schritt für Schritt einen Algorithmus und die dazugehörigen Datenstrukturen designt.

9.1.1 Verfeinerung

Das Prinzip der Verfeinerung haben wir schon mehrfach angewendet. Es besteht in der strukturierten, schrittweise erfolgten Detaillierung und Präzisierung:

1. Algorithmus „umgangssprachlich“ aufschreiben, dabei „schwierige“ Punkte und Details zunächst weglassen.
2. Jeden einzelnen Punkt schrittweise präzisieren, bis „vernünftiger“ Pseudocode dasteht (oder sogar ausführbarer Code).
3. Dazu häufig Ersetzung von Prosa durch (Unter-) Funktionsaufrufe bzw. Prozeduren.

9.1.2 Entwurfsmuster

Das schrittweise Verfeinern erklärt natürlich nur, wie man vorgehen soll, wenn man bereits ein Idee hat. Wie kommt man überhaupt aber auf die *Idee* für einen Algorithmus? Natürlich hilft die Erfahrung und das Studium von bekannten Lösungen. Ein guter Algorithmusdesigner kann auf diesen Fundus zurückgreifen und erkennt schnell, mit welchen Techniken er oder sie an ein Problem herangehen kann. Dabei hilft die Kenntnis von Entwurfsmustern – codifizierten Techniken, die wir im folgenden vorstellen.

9.1.3 Rekursion

Ein wichtiges Werkzeug ist die *Rekursion*. Mit ihrer Hilfe kann man viele Probleme elegant formulieren, selbst dann, wenn man die Lösung noch gar nicht verstanden hat! Als Beispiel sei das uralte Problem der Türme von Hanoi genannt. Ziel ist es einen Turm aus mehreren gelochten Scheiben unterschiedlicher Größe, die auf einer Stange A aufgefädelt sind, von A nach B zu bewegen. Dabei darf man eine Hilfsstange C verwenden. Dabei sind lediglich zwei Bedingungen zu beachten. Zu einem Zeitpunkt darf immer nur eine Scheibe bewegt werden und eine größere Scheibe darf niemals auf einer kleineren liegen.

Die Lösungsstrategie ist keineswegs offensichtlich, aber man erkennt, dass man, wenn man das Problem für $n-1$ Scheiben lösen kann, es auch für n Scheiben lösen kann:

> „Vermutlich wurde das Spiel 1883 vom französischen Mathematiker Édouard Lucas erfunden. Er dachte sich dazu die Geschichte aus, dass indische Mönche im großen Tempel zu Benares, im Mittelpunkt der Welt, einen Turm aus 64 goldenen Scheiben versetzen müssten, und wenn ihnen das gelungen sei, wäre das Ende der Welt gekommen. In der Geschichte lösen die Mönche das Problem folgendermaßen: Der älteste Mönch erhält die Aufgabe, den Turm aus 64 Scheiben zu versetzen. Da er die komplexe Aufgabe nicht bewältigen kann, gibt er dem zweitältesten Mönch die Aufgabe, die oberen 63 Scheiben auf einen Hilfsplatz zu versetzen. Er selbst (der Älteste) würde dann die große letzte Scheibe zum Ziel bringen. Dann könnte der Zweitälteste wieder die 63 Scheiben vom Hilfsplatz zum Ziel bringen. Der zweitälteste Mönch fühlt sich der Aufgabe ebenfalls nicht gewachsen. So gibt er dem drittältesten Mönch den Auftrag, die oberen 62 Scheiben zu transportieren, und zwar auf den endgültigen Platz. Er selbst (der Zweitälteste) würde dann die zweitletzte Scheibe an den Hilfsplatz bringen. Schließlich würde er wieder den Drittältesten beauftragen, die 62 Scheiben vom Zielfeld zum

Hilfsplatz zu schaffen. Dies setzt sich bis zum 64. Mönch (dem Jüngsten) fort, der die obenauf liegende kleinste Scheibe alleine verschieben kann. Da es 64 Mönche im Kloster gibt und alle viel Zeit haben, können sie die Aufgabe in endlicher, wenn auch sehr langer Zeit erledigen." (Quelle: Wikipedia)

Daher löst der Algorithmus TOWERS-OF-HANOI 9.1[1] offenbar das Problem. Eine Implementierung in C enthält das Listing 9.1. Der erste Aufruf für n Scheiben lautet towersofhanoi(n,1,2,3);.

Algorithmus 9.1 TOWERS-OF-HANOI($T[1,\ldots,n], A, B, C$)

1: **if** $n > 0$ **then**
2: TOWERS-OF-HANOI($T[1,\ldots,n-1], A, C, B$)
3: MOVE DISK $T[n]$ FROM A TO B
4: TOWERS-OF-HANOI($T[1,\ldots,n-1], C, B, A$)

Listing 9.1: Türme von Hanoi – C-Code

```
1  #include <stdio.h>
2
3  void towersofhanoi(int i, int from, int to, int using)
4  {
5      if (i > 0) {
6          towersofhanoi(i-1, from, using, to);
7          printf ("move␣%d␣-->␣%d\n", from, to);
8          towersofhanoi(i-1, using, to, from);
9      }
10 }
11
12 int main (int argc, const char * argv[]) {
13     printf("Hello,␣Towers␣of␣Hanoi!\n");
14     towersofhanoi(3, 1, 2, 3);
15     return 0;
16 }
```

Lemma 9.1.1. *Der Algorithmus 9.1 benötigt $2^n - 1$ Scheibenbewegungen, um einen Turm mit n Scheiben auf eine andere Stange zu verschieben.*

Beweis. Übungsaufgabe, vollständige Induktion. □

Es gibt auch keinen besseren Algorithmus, der das Problem mit weniger Scheibenbewegungen löst. Daher ist das Problem der Türme von Hanoi nur mit exponentiellem Aufwand zu lösen.

[1] Die Argumente A, B zeigen an, dass von A nach B verschoben werden soll. Dabei wird die Hilfsstange C verwendet.

Sofern das Verschieben einer Scheibe eine Sekunde dauert, sind in Tabelle 9.1 die benötigten Zeiten für verschiedenen Anzahlen von Scheiben angegeben.

Scheiben	5	10	20	30	40	60	64
Benötigte Zeit	31 Sek.	17 Min.	12 Tage	34 Jahre	348 Jhd.	36,6 Mrd. Jahre	585 Mrd. Jahre

Tabelle 9.1: Benötigte Zeit zum Lösen des Türme von Hanoi Problems

Aus Performanz- oder Speicherplatzgründen wird man bei rekursiv definierten Algorithmen häufig eine iterative Lösung suchen. Man kann unter gewissen Bedingungen mit Hilfe sogenannter Akkumulatorfunktionen diese Umwandlung vornehmen lassen; einige Compiler machen das sogar automatisch, Stichwort „End-Rekursion“ (englisch “Tail-Recursion”). Im allgemeinen ist es jedoch nicht einfach, rekursive Lösungen in iterative zu verwandeln. Häufig kann man dazu Datenstrukturen wie Stapel oder Warteschlangen benutzen.

Ein Beispiel für die Auflösung von Rekursion ist im Listing 9.2 dargestellt; die Variable c übernimmt dabei die Rolle des Akkumulators.

Listing 9.2: Fibonacci – Ausführbarer Pseudo-Code

```
1  def fib(n):
2      if n==1 or n==2:
3          return 1
4      else:
5          return fib(n-1) + fib(n-2)
6
7
8
9  # iterative version of fib simulates call stack with
10 # explicit stack but uses accumulator ("+")
11 # this is as inefficient as recursive version, of course
12 def fib_iter(n):
13     s = []
14     s.append(n)
15     res = 0
16     while len(s) != 0:
17         a = s.pop()
18         if a==1 or a==2:
19             res = res + 1
20         else:
21             s.append(a-1)
22             s.append(a-2)
23     return res
```

Rekursion ist nicht auf den rekursiven Aufruf *derselben* Funktion beschränkt

– Funktionen können sich auch *wechselseitig* aufrufen, wie das im Beispiel "Even-odd" gezeigte Listing 9.3 verdeutlicht, das bestimmt, ob eine Zahl gerade oder ungerade ist.

Listing 9.3: Even-odd – Ausführbarer Pseudo-Code

```
1  def odd(n):
2      if n==0:
3          return False
4      else:
5          return even(n-1)
6
7  def even(n):
8      if n==0:
9          return True
10     else:
11         return odd(n-1)
```

9.2 Entwurfsmuster

9.2.1 Teile-und-Herrsche (Divide and Conquer)

Das Entwurfsmuster *Teile-und-Herrsche* ist eng mit der Idee der Rekursion verwandt und in Pseudocode 9.2 für den (generischen) Algorithmus DIVIDE-AND-CONQUER dargestellt.

Algorithmus 9.2 DIVIDE-AND-CONQUER(P)

```
1: if PROBLEM P is small then
2:     return explicit Solution S_e
3: else
4:     decompose P into smaller P_1, ..., P_n
5:     for i = 1 to n do
6:         S_i ← DIVIDE-AND-CONQUER(P_i)
7:     merge Solutions S_i into S
8:     return S
```

Wir haben das Teile-und-Herrsche-Prinzip u.a. bei der Konstruktion des Algorithmus MERGE-SORT gesehen.

9.2.2 Gierige (Greedy) Algorithmen und Optimale Teilstruktur

Gierige Algorithmen versuchen ein Problem dadurch zu lösen, dass in jedem Schritt die beste „gierigste" Lösung ausgesucht wird. Gierige Algorithmen

versuchen also ein globales (Optimierungs-) Problem durch lokale Optimierung zu lösen. Die so gefundene Lösung ist nicht immer optimal und daher muss man sich explizit davon überzeugen, dass der gierige Algorithmus korrekt ist. Wesentlich dafür ist, dass das Problem die optimale Teilstruktur hat, die wir schon im Kapitel 7 und 8 benutzt haben. Lokale Optima müssen nicht unbedingt identisch mit globalen Optima sein, wie Abbildung 9.1 veranschaulicht.

Definition 9.2.1. *Optimalitätsprinzip von Bellman: Ein Problem hat die optimale Teilstruktur genau dann, wenn jede optimale Lösung des Problems, optimale Lösungen für die Teilprobleme beinhaltet.*

Als Beispiel seien die im Kapitel 7 aufgetretene optimale Teilstruktur für kürzeste-Weg-Problem und die im Kapitel 8 benutzte optimale Teilstruktur für LCS genannt.

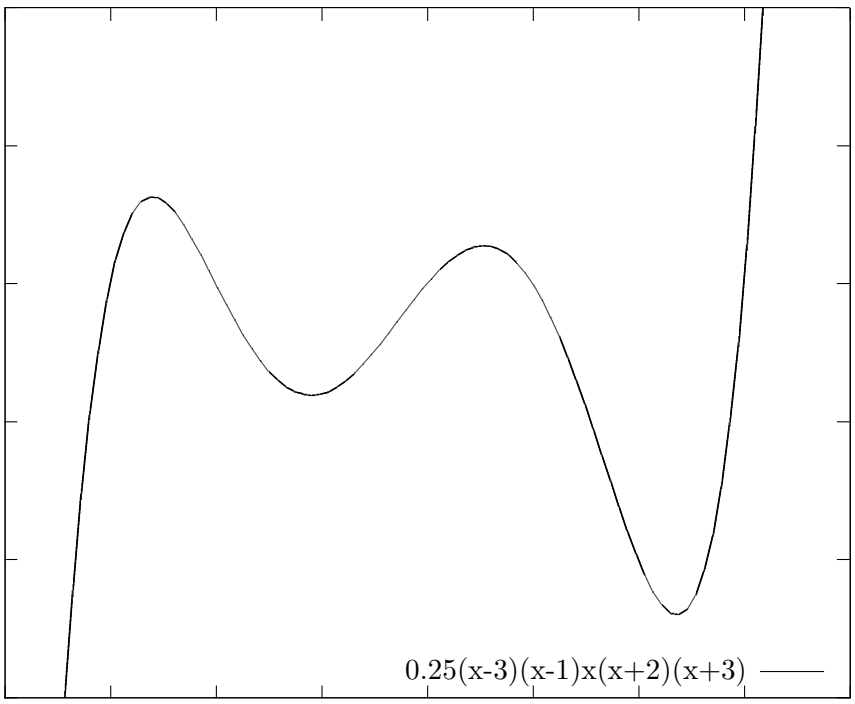


Abbildung 9.1: Lokale Optima

Ein Beispiel für einen Gierigen Algorithmus, den wir schon kennen, ist der DIJKSTRA Algorithmus.

Aktivitätsauswahl

Ein weiteres schönes Beispiel für einen Gierigen Algorithmus ist die Aktivitätsauswahl (Activity Selection), bei der es darum geht, maximal viele nicht-überlappende Aktivitäten (etwa eines Projektplans) auszuwählen. Für Details sei auf [CLR04] und auf Listing 9.4 verwiesen.

Listing 9.4: Activity Selection (Ausführbarer Pseudocode)

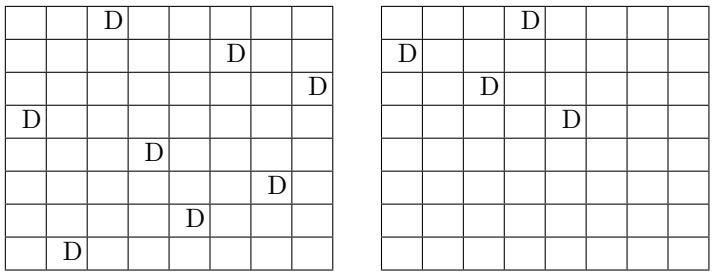
```
1  import math
2
3  def activity_sel(a, i, n):
4      m = i+1
5      l = []
6      while m <= n and a[m][0] < a[i][1]: # search for first activity
7          m = m+1
8      if m < n:
9          l += [m] # for result, store index only
10         return  l + activity_sel(a, m, n)
11     else:
12         return []
13
14 def adjust(a):
15     # add s_0 and s_n+1, math.inf acts as infinity
16     return [[0,0]] + a + [[len(a), math.inf]]
17
18 a = [[1,4],[3,5],[0,6],[5,7],[3,8],[5,9],[6,10],[8,11],[8,12],[2,13],[12,14]]
19 print(activity_sel(adjust(a), 0, len(a)+1))
```

9.2.3 Backtracking

Das Backtracking realisiert eine systematische Suchtechnik mit der ein vorgegebener Lösungsraum durchsucht werden kann. Der Name Backtracking kommt daher, dass man bei der Suche (z. B. in einem Labyrinth) in Sackgassen geraten kann. Der Backtrackingprozess sieht dann vor, systematisch zurückzugehen, bis man die nächste, noch nichtuntersuchte Lösung angehen kann. Wesentlich ist hier, dass wir wissen, dass sobald eine Sackgasse sichtbar ist, das „Weitergehen“ keinen Sinn hat und der gesamte Zweig verworfen werden kann. Als Beispiel für eine Anwendung des Backtracking verwenden wir das Acht-Damen-Problem. Ziel ist es acht Damen auf einem Schachbrett so zu platzieren, dass keine die andere bedroht, siehe Abbildung 9.2[2].

Der Algorithmus PLACE-QUEEN 9.3 löst das Acht-Damen-Problem. Er ist ein Depth-First Algorithmus (man mache sich das anhand einiger Beispiele klar); Backtracking findet in Zeile 9 statt. Man mache sich auch klar, dass

[2]Eine Sackgasse liegt im zweiten Fall vor, weil keine Dame in der zweiten Reihe (von unten) plaziert werden kann.



(a) Lösung (b) Sackgassse

Abbildung 9.2: Acht Damenproblem

die Lösung systematisch auf mögliche oder nicht mögliche Lösungen eines kleineren Problems (Damen auf kleineren Feldern) zurückgeführt wird.

Algorithmus 9.3 PLACE-QUEEN(board, col)

```
1: if board is full then
2:     PRINT SOLUTION
3:     return TRUE
4: for row = 1 to 8 do
5:     if position (row, col) is safe then
6:         put new queen on (row, col)      // Place queen
7:         if PLACE-QUEEN(board, col+1) then
8:             // Recursion
9:             return TRUE
10:        else                              // Backtracking
11:            // Remove problem
12:            remove i's queen on (row, col)
13: return FALSE                             // Base Case – No Solution
```

Backtracking findet eine natürliche Anwendung in Spielprogrammen und Planungs-, Konfigurations- und Optimierungsproblemen. Zur Laufzeit eines Backtracking-Algorithmus entstehen sogenannte *Lösungsbäume*.

9.2.4 Dynamische Programmierung

Die Technik der Dynamischen Programmierung haben wir bereits in Kapitel 8 (Berechnung einer LCS) kennengelernt. Der Anwendungsbereich der Dynamischen Programmierung sind ähnliche Probleme wie für Gierige Algorithmen, allerdings kann Dynamische Programmierung gerade dann angewandt werden, wenn die gierigen (Teil-) Lösungen nicht direkt zur optimalen globalen Lösung führen, also etwa die gierigen (Teil-) Lösungen nicht automatisch (Teil-) Lösungen des globalen Problems sind. Dynamische Programmierung läßt

sich anwenden, wenn die beiden folgenden Bedingungen erfüllt sind:

1. Optimale Teilstruktur – haben wir bereits bei Gierigen Algorithmen kennengelernt.
2. Überlappende Teilprobleme, d.h. dass wir bei der rekursiven Lösung der Teilprobleme größere Bereiche haben, die „überlappen“.

Das Prinzip der Dynamischen Programmierung nutzt insbesondere die zweite Bedingung aus, um einmal errechnete Parameter für weitere Teilprobleme wieder zu verwenden, in dem diese z. B. in einer Tabelle gespeichert werden anstelle diese immer wieder neu zu berechnen. So können dann Lösungen für das globale Problem sukzessive aus den Teillösungen aufgebaut werden. Bei der LCS liegen Überlappende Teilprobleme vor, denn wenn man die LCS von X und Y finden möchte, dann muss man

- die LCS von X und Y_{n-1} und
- die LCS von X_{m-1} und Y finden.

Diese Teilprobleme haben aber das gemeinsame Teilproplem LCS von X_{m-1} und Y_{n-1}, usw. Damit konnten wir die Berechnung von $c[i, j]$ auf die Berechnung $c[i-1, j]$, $c[i, j-1]$ und $c[i-1, j-1]$ zurückführen.

9.3 Entwurfsprinzipien Datenstrukturen

Wir können an dieser Stelle dieses Thema *nicht* abschließend behandeln – dies ist z. B, den Modulen über Objekt-orientierte Programmiersprachen vorbehalten. Die folgenden Aspekte sollte man jedoch auf alle Fälle adressieren:

- Man verwende – wo möglich – bestehende, bekannte Datenstrukturen (für die ggf. Bibliotheken existieren).
- Man modifiziere – falls nötig – bestehende, bekannte Datenstrukturen um diejenigen Attribute und Konstrukte, die für das vorliegende Problem spezifisch sind.
- Man setze aus bekannten Datenstrukturen neue zusammen (Kombinationsprinzip).
- Falls der gewünschte Datentyp von Grund auf neu implementiert werden muss, wende man die bewährten Designprinzipen aus dem Software Engineering an wie z. B. Kapselung, Information Hiding etc. an. Insbesondere sind den Anforderungen der gewählten Zielsprache Rechnung zu tragen (z. B. Speicherverwaltung).
- Wann immer man bei der Formulierung eines Algorithmus bemerkt, dass er „umständlicher“ als eigentlich notwendig ist, überprüfe man,

ob man durch Wahl einer alternativen Datenstruktur den (Pseudo-) Code vereinfachen kann.

9.4 Fallstudien

Wir betrachten zwei Fallstudien, um die Theorie zu verdeutlichen.

9.4.1 Markov

Wir wollen ein Programm schreiben, dass

1. einen Text einliest,
2. die statistische Häufigkeit von Wortkombinationen ermittelt,
3. basierend auf dieser Statistik einen zufälligen Text generiert, der dieselbe Statistik aufweist.

Dieses Beispiel ist Kernighan und Pike [KP99] entnommen.

Neben dem Verständnis der mathematischen Grundlagen ist ganz entscheidend, welche Datenstrukturen man verwendet. Kernighan und Pike haben sich für Hashtabellen entschieden (warum? Übung!). Der Code für die Hashtabellen ist im Listing 9.5, der Aufruf im Listing 9.6 dokumentiert.

Listing 9.5: Markov – C-Code Datenstruktur

```
1  State *statetab[NHASH]; /* hash table of states */
2
3  unsigned int hash(char *s[NPREF])
4  {
5      unsigned int h;
6      unsigned char *p;
7      int i;
8
9      h = 0;
10     for (i = 0; i < NPREF; i++)
11         for(p = (unsigned char *) s[i]; *p != '\0'; p++)
12             h = MULTIPLIER *h + *p;
13     return h % NHASH;
14
15 }
16
17 State* lookup(char *prefix[NPREF], int create)
18 {
19     State *sp;
20     int h, i;
21     h = hash(prefix);
22     for(sp = statetab[h]; sp != NULL; sp = sp->next) {
23         for(i = 0; i < NPREF; i++)
24             if(strcmp(prefix[i], sp->pref[i]) != 0)
25                 break;
```

```
26          if (i == NPREF) {/* found */
27              return sp;
28          }
29      }
30      if(create) {
31          /* TODO should check return value */
32          sp = (State * ) malloc(sizeof(State));
33          for(i = 0; i < NPREF; i++)
34              sp->pref[i] = prefix[i];
35          sp->suf = NULL;
36          sp->next = statetab[h];
37          statetab[h] = sp;
38      }
39      return sp;
40  }
41
42  void add(char *prefix[NPREF], char *suffix)
43  {
44      State *sp;
45      sp = lookup(prefix, 1); /* create if not found */
46      Suffix *suf;
47      suf = (Suffix *) malloc(sizeof(Suffix));
48      suf->word = suffix;
49      suf->next = sp->suf;
50      sp->suf = suf;
51      /* shift words left in prefix */
52      memmove(prefix, prefix +1, (NPREF -1) * sizeof(prefix[0]));
53      prefix[NPREF-1] = suffix;
54  }
```

Listing 9.6: Markov Aufruf – C-Code main

```
1   /*
2    *  main.c
3    *  markov
4    *
5    *  Created by Joerg Schaefer on 1/25/10.
6    *  Based on Brian W. Kernighan and Rob Pike. The Practice of Programming. Addison-
7    *  Copyright 2010-2020 FRA-UAS All rights reserved.
8    *
9    */
10  #include <stdio.h>
11  #include <stdlib.h>
12  #include <string.h>
13  #include <time.h>
14  #include "state.h"
15
16  char NONWORD[] = "\n";
17
18  void build(char *prefix[NPREF]);
19  void generate(int words);
20
21  int main (int argc, const char * argv[])
22  {
```

```
23      clock_t start,end;
24      double timeelapsed;
25      start = clock();
26
27      int i, nwords = MAXGEN;
28      printf("START\n");
29      char *prefix[NPREF];
30
31      srand ( time(NULL) );
32
33      for(i = 0; i < NPREF; i++)
34          prefix[i] = NONWORD;
35
36      build(prefix);
37      add(prefix, NONWORD);
38      end = clock();
39      timeelapsed = (end - start)/(double)CLOCKS_PER_SEC;
40      start = clock();
41      generate(nwords);
42      end = clock();
43      printf("\n\nbuilding hashtable took %.8f seconds\n", timeelapsed);
44
45      timeelapsed = (end - start)/(double)CLOCKS_PER_SEC;
46      printf("generating text took %.8f seconds\n", timeelapsed);
47      return 0;
48  }
49
50  void build(char *prefix[NPREF])
51  {
52      char buf[100];
53      printf("Please enter words, finish with CTRL-D\n");
54      while (scanf("%s", buf) == 1){
55          add(prefix, strdup(buf));
56      }
57
58  }
59
60  void generate(int nwords)
61  {
62      State *sp;
63      Suffix * suf;
64      char *prefix[NPREF], *w;
65      int i, nmatch;
66
67      for(i = 0; i < NPREF; i++)
68          prefix[i] = NONWORD;
69
70      for(i = 0; i < nwords; i++) {
71          sp = lookup(prefix, 0);
72          nmatch = 0;
73          for(suf = sp->suf; suf != NULL; suf = suf->next)
74              if(rand() % ++nmatch == 0)
75                  w = suf->word;
76          if(strcmp(w, NONWORD) == 0)
```

```
77                  break;
78              printf("%s␣", w);
79              memmove(prefix, prefix+1, (NPREF-1) * sizeof(prefix[0]));
80                  prefix[NPREF-1] = w;
81          }
82      }
```

9.4.2 Sudoku

Sudoku (japanisch Sudoku, kurz für Suji wa dokushin ni kagiru), wörtlich ungefähr "Es ist am besten, wenn die Zahl alleine bleibt") ist ein Logikrätsel. In der üblichen Version ist es das Ziel, ein 9x9-Gitter mit den Ziffern 1 bis 9 so zu füllen, dass jede Ziffer in jeder Spalte, in jeder Zeile und in jedem Block (3x3-Unterquadrat) nur einmal vorkommt (eine für Informatiker geeignete Version ist in Abbildung 9.3, eine "korrekte" Verallgemeinerung in Abbildung 9.4 abgebildet).

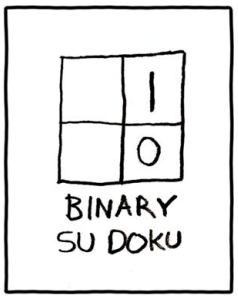


Abbildung 9.3: Binary Sudoku, Quelle: http://xkcd.com

1

(a) 1x1 Sudoku

1	2	3	4
3	4	1	2
2	3	4	1
4	1	2	3

(b) 2x2 Sudoku

Abbildung 9.4: Korrekte 1x1 und 2x2 Sudokus

Ausgangspunkt ist ein Gitter, in dem bereits mehrere Ziffern vorgegeben sind.

Wir wollen eine Lösung für Sudoku-Rätsel schreiben. Die mit Abstand beste Quelle einschließlich Erklärung wie man eine derart elegante Lösung finden kann, findet sich bei Peter Norvig (`http://norvig.com/sudoku.html`). Wir haben sein Programm in den Listings 9.7 und 9.8 abgedruckt.

Listing 9.7: Sudoku

```
1  ## Solve Every Sudoku Puzzle
2
3  ## See http://norvig.com/sudoku.html
4
5  ## Ported to Python 3.0 using python-modernize -w sudo.py
6
7  ## Throughout this program we have:
8  ##   r is a row,    e.g. 'A'
9  ##   c is a column, e.g. '3'
10 ##   s is a square, e.g. 'A3'
11 ##   d is a digit,  e.g. '9'
```

```
##   u is a unit,   e.g. ['A1','B1','C1','D1','E1','F1','G1','H1','I1']
##   g is a grid,   e.g. 81 non-blank chars, e.g. starting with '.18...7...
##   values is a dict of possible values, e.g. {'A1':'123489', 'A2':'8', ...}

from __future__ import print_function
from six.moves import zip
def cross(A, B):
    return [a+b for a in A for b in B]

rows = 'ABCDEFGHI'
cols = '123456789'
digits   = '123456789'
squares  = cross(rows, cols)
unitlist = ([cross(rows, c) for c in cols] +
            [cross(r, cols) for r in rows] +
            [cross(rs, cs) for rs in ('ABC','DEF','GHI') for cs in ('123','456','789')])
units = dict((s, [u for u in unitlist if s in u])
             for s in squares)
peers = dict((s, set(s2 for u in units[s] for s2 in u if s2 != s))
             for s in squares)

def search(values):
    "Using depth-first search and propagation, try all possible values."
    if values is False:
        return False ## Failed earlier
    if all(len(values[s]) == 1 for s in squares):
        return values ## Solved!
    ## Chose the unfilled square s with the fewest possibilities
    _,s = min((len(values[s]), s) for s in squares if len(values[s]) > 1)
    return some(search(assign(values.copy(), s, d))
                for d in values[s])

def assign(values, s, d):
    "Eliminate all the other values (except d) from values[s] and propagate."
    if all(eliminate(values, s, d2) for d2 in values[s] if d2 != d):
        return values
    else:
        return False

def eliminate(values, s, d):
    "Eliminate d from values[s]; propagate when values or places <= 2."
    if d not in values[s]:
        return values ## Already eliminated
    values[s] = values[s].replace(d,'')
    if len(values[s]) == 0:
        return False ## Contradiction: removed last value
    elif len(values[s]) == 1:
        ## If there is only one value (d2) left in square, remove it from peers
        d2, = values[s]
        if not all(eliminate(values, s2, d2) for s2 in peers[s]):
            return False
    ## Now check the places where d appears in the units of s
    for u in units[s]:
        dplaces = [s for s in u if d in values[s]]
        if len(dplaces) == 0:
            return False
        elif len(dplaces) == 1:
            # d can only be in one place in unit; assign it there
            if not assign(values, dplaces[0], d):
                return False
    return values
```

Listing 9.8: Sudoku Hilfsroutinen

```
def parse_grid(grid):
    "Given a string of 81 digits (or .0-), return a dict of {cell:values}"
    grid = [c for c in grid if c in '0.-123456789']
    values = dict((s, digits) for s in squares) ## Each square can be any digit
    for s,d in zip(squares, grid):
        if d in digits and not assign(values, s, d):
            return False
    return values

def solve_file(filename, sep='\n', action=lambda x: x):
    "Parse a file into a sequence of 81-char descriptions and solve them."
    results = [action(search(parse_grid(grid)))
               for grid in open(filename).read().strip().split(sep)]
    print(("## Got %d out of %d" % (
          sum((r is not False) for r in results), len(results))))
    return results

def printboard(values):
    "Used for debugging."
```

```
20      width = 1+max(len(values[s]) for s in squares)
21      line = '\n' + '+'.join(['-'*(width*3)]*3)
22      for r in rows:
23          print(''.join(values[r+c].center(width)+(c in '36' and '|' or '')
24                        for c in cols) + (r in 'CF' and line or '')))
25      # print
26      return values
27
28  def all(seq):
29      for e in seq:
30          if not e: return False
31      return True
32
33  def some(seq):
34      for e in seq:
35          if e: return e
36      return False
37
38  if __name__ == '__main__':
39      solve_file("top95.txt", action=printboard)
40
41  ## References used:
42  ## http://www.scanraid.com/BasicStrategies.htm
43  ## http://www.krazydad.com/blog/2005/09/29/an-index-of-sudoku-strategies/
44  ## http://www2.warwick.ac.uk/fac/sci/moac/currentstudents/peter_cock/python/sudoku/
```

Norvig nutzt hier ganz geschickt

- Clevere Datenstrukturen
- Wechselseitig rekursive Aufrufe (wie bei Even-odd s.o.)
- Constraint Propagation
- Backtracking (modifizierte Version)

9.5 Fragen und Aufgaben zum Selbststudium

1. Erklären Sie in eigenen Worten die Grundidee, die den folgenden Begriffen zugrunde liegt:
   (a) Teile-und-Herrsche,
   (b) Gierige Algorithmen,
   (c) Dynamische Programmierung,
   (d) Gemeinsame Teilprobleme,
   (e) Backtracking und
   (f) Constraint Propagation!
2. Markov: Kompilieren Sie das Programm `Markov` und generieren Sie Texte damit!
3. Markov: Warum verwendet das Programm `Markov` Hashtabellen? Welche Daten werden in der Hashtabelle abgelegt?
4. Markov: Zeichnen Sie ein Ablaufdiagramm für das Programm `Markov` und erläutern Sie, wie es funktioniert!

5. Sudoku: Welche Datenstrukturen verwendet `sudo.py`?
6. Sudoku: Erklären Sie das Zusammenspiel von `assign` und `eliminate`!
7. Sudoku: Zeichnen Sie ein Ablaufdiagramm für die Prozedur `search` insbesondere im Zusammenspiel mit `assign` und `eliminate`!
8. Sudoku: Welche Rekursionen gibt es in `sudo.py` und welche Abbruchbedingungen existieren für diese?
9. Überlegen Sie sich einen Algorithmus zur effizienten Berechnung von x^n bezüglich der Anzahl der Multiplikationen, wobei n eine natürliche Zahl ist. Dieser Algorithmus muss natürlich besser sein als sukzessive Multiplikation $x^n = x \cdot \ldots \cdot x$. Berechnen Sie die Anzahl der Multiplikationen! Bei welchen n ist Ihr Algorithmus optimal, wo ist er schlechter? Ist Ihr Algorithmus im Worst Case besser als sukzessive Multiplikation?

Kapitel 10

Parallele Algorithmen

Die folgenden, aktuellen Entwicklungen führen zu verstärktem Interesse an parallelen Algorithmen:

1. Multi-core Maschinen
2. Generelle Verbreitung von Verteilten Systemen
3. Cloud Computing

10.1 Grundbegriffe

10.1.1 Historischer Rückblick

Bereits 1967 formulierte Gene Amdahl das nach ihm benannte Amdahlsche Gesetz (s. [Amd67]):

Lemma 10.1.1. *Wenn ein Algorithmus den Anteil a von parallelisierbaren Arbeitsschritten aufweist, dann gilt für die maximale Beschleunigung Q auf einem Rechner mit P Prozessoren gegenüber einem Einzel CPU Rechner:*

$$\begin{aligned} & Q & := & \frac{1}{(1-a)+a/P} \\ \Rightarrow \quad & Q & \leq & \frac{1}{(1-a)} \wedge \lim_{P\to\infty} Q = \frac{1}{(1-a)} \end{aligned}$$

Beweis. offensichtlich. □

Beispiel 1: In einem Programm sei die Hälfte parallelisierbar, also $a = 0,5$. Es steht ein Rechner mit $P = 2$ Prozessoren zur Verfügung. Dann beträgt die benötigte Rechenzeit 75%, verglichen mit einer Ein-Prozessor-Maschine. Stünden unendlich viele Prozessoren zur Verfügung, dann gilt: $\lim_{P\to\infty} Q = 2$.

Eine Verkürzung der Rechenzeit über eine Halbierung hinaus ist also nicht möglich.

Beispiel 2: Mit besseren Rahmenbedingungen, 80% parallelisierbar, 8 Prozessoren kann die Rechenzeit auf 30% reduziert werden. Mit unendlich vielen Prozessoren wäre: $Q = 5$, also wird noch immer $\frac{1}{5}$ der Rechenzeit einer Ein-Prozessor-Maschine benötigt.

Einerseits hat das Amdahlsche Gesetz einige optimistischen Annahmen.

- Der oder die parallelisierbaren Teile sind beliebig teilbar, d.h. sie können so geschnitten werden, dass die Laufzeit aller Teile identisch ist.
- Es gibt keine Split- und Rekombinationskosten.
- Die Laufzeit ist vorhersagbar, z.B. unabhängig von den Eingaben.

Jeder, der sich schon intensiver mit Parallität in der Software auseinander gesetzt hat, wird dies bestätigen können.

Andererseits ist das Amdahlsche Gesetz auch eine pessimistischste Abschätzung, da es davon ausgeht, dass die nicht-parallelisierbaren Anteile konstant sind – eine Annahme, die, wie wir noch sehen werden, nicht unbedingt zutreffend sein muss. Ausserdem geht es davon aus, dass die Problemgröße unverändert bleibt, das Ziel ist also ist, ein Problem in möglichst kurzer Zeit zu berechnen. Für den Fall, in der gleichen Zeit ein *größeres* Problem zu berechnen, ergeben sich günstigere Voraussetzungen, da der parallele Anteil sehr hoch werden kann. Je länger eine Berechnung dauert, desto weniger fällt die Zeit für die Initialisierung und abschließende Synchronisierung ins Gewicht. Eine Ergänzung zu Amdahls Gesetz formulierte John Gustafson – das Gustafsons Gesetz (`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.6348`):

> „We feel that it is important for the computing research community to overcome the “mental block” against massive parallelism imposed by a misuse of Amdahl’s speedup formula“.

10.1.2 Rechnermodell

Um sinnvolle Aussagen über Parallelisierbarkeit und verwandte Dinge treffen zu können, benötigen wir ein Berechnungsmodell. Ein fruchtbares Modell eingeführt von Eager et al. (s. [EZL89]) stellen wir im folgenden vor. Es basiert auf der Darstellung des Berechnungsbaums als gerichteter, azyklischer Graph (englisch “Directed Acyclic Graph” oder DAG), wie z. B. in Abbildung 10.1 dargestellt.

Man beachte, dass die Kanten eine *Teilordnung* $\leq$ für die Knoten N_i definieren: $N_i < N_j :\Leftrightarrow \exists\, p(i,j)$, wobei $p(i,j)$ einen Pfad von N_i nach N_j bezeichnet.

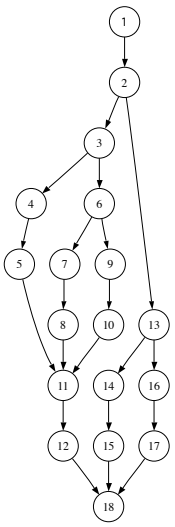


Abbildung 10.1: DAG

In dem Beispiel gilt dann $1 < 2$, $6 < 12$ – Sprechweise N_i kommt „vor“ N_j. Gibt es keinen Weg, dann laufen N_i und N_j parallel ab, wir verwenden als Notation dafür $\|$ – zum Beispiel sind 4 und 9 parallel: $4 \| 9$.

10.1.3 Work Span Gesetze

Mit diesem einfachen Modell lassen sich bereits interessante Aussagen formulieren und ableiten:

Definition 10.1.1. *Der Aufwand (englisch “WORK”) einer parallelen Berechnung ist definiert als die Laufzeit der Berechnung auf einem Prozessor P.*

Nehmen wir zur Vereinfachung jetzt und im folgenden immer an, dass alle Berechnungen Einheitszeit verlangen, dann ist im Beispiel Abbildung 10.1 also WORK = 18.

Definition 10.1.2. *Die Spanne (englisch “SPAN”) einer parallelen Berechnung ist definiert als die längste Laufzeit einer Teilberechnung entlang eines Pfades im DAG, mit anderen Worten*

$$SPAN := \textit{Anzahl der Knoten entlang des kritischen Pfades von } DAG$$

Im Beispiel Abbildung 10.1 ist also SPAN = 9 (kritischer Pfad: $1 \to 2 \to 3 \to 6 \to 7 \to 8 \to 11 \to 12 \to 18$).

Definition 10.1.3. *Die Laufzeit eines parallelen Algorithmus auf P Prozessoren wird mit T_P bezeichnet. Die Laufzeit eines parallelen Algorithmus auf*

einem einzigen Prozessor wird mit T_1 bezeichnet[1].

Bemerkung: Die Laufzeit eines Algorithmus hängt nicht nur von WORK und SPAN ab, sondern auch davon, ob (genügend viele) Prozessoren vorhanden sind, um die Parallelisierbarkeit auszuschöpfen, d.h. jeden Zweig auf einen Prozessor verteilen zu können. Wenn eine (idealisierte) Maschine mit unendlich vielen Prozessoren gegeben wäre, dann würde die Laufzeit offensichtlich identisch mit dem SPAN sein, daher wird der SPAN meist auch mit T_∞ bezeichnet.

Lemma 10.1.2. *Das Arbeitsgesetz (englisch "work-law") besagt:*

$$T_P \geq T_1/P$$

Beweis. Ein Rechner mit P Prozessoren kann in einem Schritt P Berechnungen ausführen, also in der Zeit T_P genau PT_P Berechnungen. Insgesamt sind T_1 Berechnungen auszuführen. Parallel sind mindestens so viele Berechnungen auszuführen wie notwendig, also $PT_P \geq T_1$ oder äquivalent $T_P \geq T_1/P$. □

Lemma 10.1.3. *Das Zeitspannengesetz (englisch "span-law") besagt:*

$$T_P \geq T_\infty$$

Beweis. Ein Rechner mit P Prozessoren ist langsamer als ein (idealisierter) Rechner mit unendlich vielen Prozessoren (der Rechner mit unendlich vielen Prozessoren kann immer den mit P Prozessoren emulieren). □

Man kann mit dem Arbeits- und Zeitspannengesetz zeigen, dass für einen idealisierten Parallelrechner gilt:

$$T_P \leq T_1/P + T_\infty. \tag{10.1}$$

Für den formalen Beweis sei auf [CLR04] verweisen. Gleichung (10.1.) kann man jedoch auch intuitiv verstehen, wenn man die Berechnungsschritte einteilt in solche, die auf sämtlichen P Prozessoren verteilt werden können und solche für die das nicht gilt. Für erstere gilt für einen idealen Parallelrechner $PT_P \leq T_1$; für letztere $T_P \leq T_\infty$.

Definition 10.1.4. *Als Beschleunigung (englisch "speed-up") S definiert man*

$$S := T_1/T_P$$

Interpretation: Die Beschleunigung S gibt an, wie viele mal schneller der parallelisierte Algorithmus auf P Prozessoren gegenüber demjenigen auf einem Prozessor ist.

[1] Gemäß der obigen Definition ist also T_1 identisch mit WORK.

Lemma 10.1.4. *Es gilt*

$$S \leq P$$

Beweis. Aus dem Arbeitsgesetz folgt $T_P \geq T_1/P$ also $P \geq T_1/T_P$. □

Die Beschleunigung kann also nicht größer sein als P (intuitiv klar). In der Regel ist sie wegen Overhead geringer (z.T. viel geringer).

Definition 10.1.5. *Wenn $S = \Theta(P)$ dann spricht man von* linearer *Beschleunigung, wenn $S = P$ von* perfekter *linearer Beschleunigung (optimaler Fall).*

Definition 10.1.6. *Der Quotient $Q := T_1/T_\infty$ heißt Parallelismus (englisch "parallism").*

Interpretation:

1. Q gibt die durchschnittliche[2] Dauer (Arbeit) entlang des kritischen Wegs an (den parallelisierbaren Teil) – das folgt unmittelbar aus der Definition von T_1 und SPAN.
2. Q gibt die maximal mögliche Beschleunigung an ($Q = T_1/T_\infty \geq T_1/T_P = S$).
3. Der optimale Fall von perfekter linearer Beschleunigung kann nicht erreicht werden, wenn mehr als Q Prozessoren vorhanden sind (es ist also sinnlos, noch mehr Prozessoren hinzuzufügen, sobald man Q erreicht hat). Um das einzusehen, nehmen wir an, dass $P > Q = T_1/T_\infty > T_1/T_P = S$. Die Beschleunigung ist also echt kleiner als Q. Gilt sogar $P \gg Q$ dann folgt $S \ll P$; man erreicht also immer weniger Vorteil durch Hinzunehmen von Prozessoren.

Für das Eingangsbeispiel (Abbildung 10.1) gilt:

1. WORK = 18
2. SPAN = 9
3. Daher beträgt der Parallelismus $Q = 2$ (das ist nicht viel).

10.2 Parallelisierungsmodell

Wir führen zwei neue Schlüsselwörter ein **spawn** und **sync** (s. [CLR04]), mit deren Hilfe wir in Pseudocode parallele Algorithmen beschreiben können. Wenn einer Anweisung **spawn** vorangestellt wird, dann *kann* sie parallel zu den nachfolgenden ausgeführt werden. Eine **sync** Anweisung unterbricht

[2] wir hatten ja zur Vereinfachung bisher immer eine Einheitszeit angenommen

die Anwendung solange, bis zuvor eventuell aufgerufene Anweisungen ein Ergebnis geliefert haben.

Eine kommerzielle Implementierung dieser am MIT entwickelten Ideen ist im Cilk++ Runtime System (RTS) für C++ implementiert (`http://www.cilk.com`).

10.3 Beispiel

Wir illustrieren die Konzepte wieder am bekannten Fibonacci-Beispiel.

10.3.1 P-Fibonacci

Wir wollen den bekannten rekursiven Algorithmus FIB 10.1 parallelisieren[3]:

Algorithmus 10.1 FIB(n)

```
1: if n ≤ 1 then
2:     return n
3: else
4:     x ← FIB(n − 1)
5:     y ← FIB(n − 2)
6: return x + y
```

Mithilfe der Schlüsselwörter ein **spawn** und **sync** können wir FIB parallelisieren wie im Algorithmus P-FIB 10.2 dargestellt.

Algorithmus 10.2 P-FIB(n)

```
1: if n ≤ 1 then
2:     return n
3: else
4:     x ← spawn P-FIB(n − 1)
5:     y ← P-FIB(n − 2)
6: sync
7: return x + y
```

Der Ablauf der Berechnung ist dabei in Abbildung 10.2 dargestellt. Die Parallelisierungs- und Synchronisationsaufrufe sind mit **spawn** und **sync** gekennzeichnet und es wurde ein **cont** d.h. “Continuation” Knoten eingeführt, der der Fortsetzung (englisch “Continuation”) nach dem Parallelisierungsaufruf

[3]Wie wir gezeigt haben, ist dieser Algorithmus sehr schlecht, aber hier kommt es nur auf das Prinzip an!

spawn in Zeile 4 des Algorithmus vorangeht, und alle Funktionsaufrufe (englisch "call") bis zum "sync" beinhaltet – hier nur einer in Zeile 5.

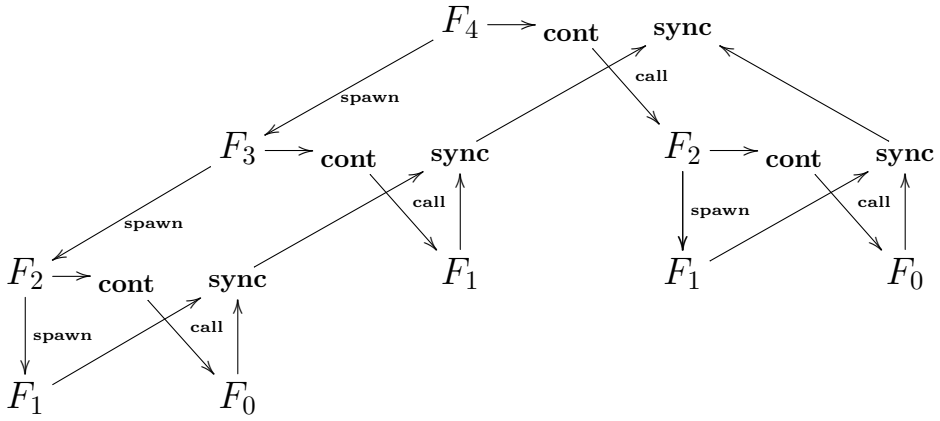


Abbildung 10.2: Berechnungsablauf

10.3.2 Analyse

Wir betrachten zunächst P-FIB(4), siehe Abbildung 10.3.

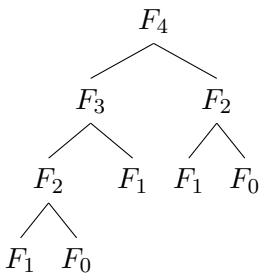


Abbildung 10.3: Fibonacci Berechnungsbaum für FIB(4)

Dann gilt für den Aufwand WORK $T_1 = 17$ und für die Spanne SPAN $T_\infty = 8$ (siehe Abbildung 10.2) und daher für die Parallelität $Q = T_1/T_\infty = 2.125$. Das bedeutet, dass P-FIB(4) schlecht parallelisiert und auf einem Parallelrechner mit 200 Prozessoren kaum schneller läuft als auf einem mit nur zwei Prozessoren.

Diese Betrachtung ändert sich jedoch mit wachsenden n rasch. Der Aufwand WORK oder T_1 von P-FIB ist einfach zu berechnen, da identisch mit der seriellen Prozedur FIB. Daher gilt

$$T_1(n) = T(n) = \Theta(\Phi^n),$$

mit $\Phi = (1 + \sqrt{5})/2$. Für die Spanne SPAN oder T_∞ gilt:

$$\begin{aligned} T_\infty(n) &= \max\left(T_\infty(n-1), T_\infty(n-2)\right) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1), \end{aligned}$$

denn bei den parallel laufenden Aktivitäten können wir das Maximum nehmen. Die Lösung dieser Gleichung lautet $T_\infty(n) = \Theta(n)$.

Beweis. Trivial □

Damit ergibt sich für die Parallelität Q von P-FIB:

$$Q(n) = T_1(n)/T_\infty(n) = \Theta(\Phi^n)/\Theta(n) = \Theta(\Phi^n/n).$$

Q wächst exponentiell, daher parallelisiert P-FIB *ausgezeichnet* – es ist aber immer noch ein schlechter Algorithmus...

10.3.3 Mergesort

Offensichtlich eignen sich Algorithmen, die das Teile-und-Herrsche-Prinzip bzw. Divide and Conquer verwenden, gut zum Parallelisieren. Ein Algorithmus, der auf Divide and Conquer beruht, ist MERGESORT, den wir bereits kennen. Eine naive parallelisierte Version ist im Algorithmus 10.3 dargestellt. Eine genaue Analyse zeigt (s. [CLR04]), dass $Q = \Theta(\lg n)$. Das

Algorithmus 10.3 P-MERGESORT(A)

```
1:  if length(A) <= 1 then
2:      return A
3:  middle ← ⌊length(A)/2⌋
4:  left ← A[0, ..., middle − 1]
5:  right ← A[middle, ..., length(A) − 1]
6:  left ← spawn MERGESORT(left)        // Recursion
7:  right ← MERGESORT(right)            // Recursion
8:  sync
9:  if left_last > right_first then
10:     return MERGE(left, right)       // Need to merge
11: else
12:     return (left, right)            // Nothing to merge, just append
```

ist sehr schlecht, denn wenn wir z. B. 1 Million Elemente sortieren möchten ist $\lg 10^6 \approx 20$, d.h. wir werden lineare Beschleunigung nur für ein Dutzend Prozessoren erreichen und danach findet keine Skalierung mehr statt. Der Grund liegt im seriellen MERGE. Man kann aber eine parallele Version für MERGE, P-MERGE einführen und damit $\mathcal{O}(n/\lg^2 n)$ erreichen, was *viel* besser ist.

10.4 Praxis

In diesem Kapitel konnten wir – der Kürze eines Semesters geschuldet – nur eine elementare Einführung in das Thema geben. Für weiterführende Literatur sei auf [CLR04] und [EZL89] verwiesen.

In der Praxis spielen Fragen der Parallelisierbarkeit eine immer wichtigere Rolle – nicht zuletzt hervorgerufen durch Phänomene wie Multicore-Architekturen und Cloud-Computing; in Bezug auf letzteres seien hier nur die folgenden wichtigen Anwendungsfälle genannt:

- MapReduce
- Hadoop
- Disco

10.5 Fragen und Aufgaben zum Selbststudium

1. Machen Sie sich die Bedeutung des Parameters Q im Amdahlschen Gesetz ($Q = \frac{1}{(1-a)+a/P} \leq \frac{1}{(1-a)}$) klar, insbesondere anhand der trivialen Beispiele $a = 0$, 1 oder $1/2$!

2. Angenommen, Sie parallelisieren P-FIB durch **spawn** in Zeile 5 anstelle von Zeile 4 im Algorithmus 10.2. Berechnen Sie WORK, SPAN und Parallelität für diesen Fall und vergleichen Sie mit den Berechnungen im Skript!

3. Könnte man im Algorithmus 10.3 P-MERGESORT in Zeile 7 ein *zusätzliches* **spawn** verwenden? Was änderte sich dadurch; ist dies sinnvoll?

4. Professor Schlau hat einen Algorithmus A, für den gilt:

$$\begin{aligned} T_1^A &= 2000 \\ T_\infty^A &= 10 \end{aligned}$$

   Professor Schlau verbessert diesen Algorithmus so, dass für den neuen Algorithmus A' folgendes gilt:

$$\begin{aligned} T_1^{A'} &= 1000 \\ T_\infty^{A'} &= 15 \end{aligned}$$

   Berechnen Sie die Laufzeit mit Hilfe der Formel[4]

$$T_P = T_1/P + T_\infty$$

[4] d.h. wir verstehen die Ungleichung (10.1) als Abschätzung für die Laufzeit!

für $P = 32$ Prozessoren und $P = 512$ Prozessoren! Was sind Ihre Schlussfolgerungen?

Kapitel 11

String-Matching

11.1 Einführung

11.1.1 Motivation

String Matching bezeichnet ein Verfahren, einen zu suchenden String oder ein Stringmuster (englisch "Pattern") in einem Text zu finden. Ein String kann dabei eine beliebige Zeichenkette darstellen. Offensichtlich ist diese Funktionalität vielfältig anwendbar, u.a.

- Texteditoren
- Textverarbeitungsprogramme
- Suchfunktionen von Betriebssystemen
- Indizierung von Webseiten (Suchmaschinen)
- DNA - Sequenzierung
- ...

Effiziente String Matching Verfahren stellen somit eine wichtige Klasse von Algorithmen dar.

11.1.2 Terminologie

Um im folgenden die Algorithmen präzise definieren und analysieren zu können, führen wir die nachstehende Terminologie ein.

Definition 11.1.1. *Wir definieren die folgenden Begriffe:*

- *Ein* Alphabet *ist ein endlicher Zeichenvorrat* Σ *von Zeichen oder Buchstaben (englisch "Character").*

- *Ein* String T *ist ein Feld* $A[1, \ldots n]$ *der Länge* n *mit* $A[i] \in \Sigma$*, Notation* $T \in \Sigma^*$*. (Die Menge aller möglichen Zeichenketten (alle Wörter) wird also mit* $T \in \Sigma^*$ *bezeichnet.) Man beachte, dass wir hier den Index von* 1 *bis* n *laufen lassen!*
- *Ein* Muster P *(englisch "Pattern") ist ein Feld der Länge* $m \leq n$*.*
- *Ein Muster* P *trifft mit* Verschiebung *(englisch "Shift")* s *in* T *auf genau dann, wenn* $0 \leq s \leq n-m$ *und* $T[s+1, \ldots, s+m] = P[1, \ldots, m]$
- *Mit* P_k *bezeichnen wir Präfix des Musters bis zum Index* $k \leq m$*, also*

$$P_k := P[1, \ldots, k] \ .$$

Definition 11.1.2. *Das* String-Matching *Problem besteht im Auffinden aller möglichen Verschiebungen eines Musters* P *in einem Text (String)* T*, wie in Abbildung 11.1 verdeutlicht*

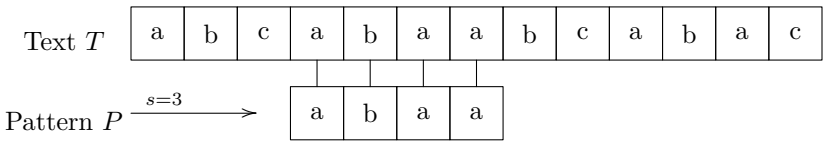


Abbildung 11.1: Beispiel String-Matching Problem

Definition 11.1.3. *Ein String* $w \in \Sigma^*$ *heißt* Präfix[1] *eines Strings* $x \in \Sigma^*$*, genau dann wenn* $x = wy$ *für ein* $y \in \Sigma^*$*. Notation:* $w \sqsubset x$*. Ein String* $w \in \Sigma^*$ *heißt* Suffix *eines Strings* $x \in \Sigma^*$*, genau dann wenn* $x = yw$ *für ein* $y \in \Sigma^*$*. Notation:* $w \sqsupset x$*.*

11.2 Algorithmen

11.2.1 Naiver Algorithmus

Ein einfacher, naiver String-Matching Algorithmus, der sozusagen Brute-Force verwendet, ist NAIVE-STRING-MATCHING 11.1.

11.2.2 Analyse

Lemma 11.2.1. *NAIVE-STRING-MATCHING benötigt* $\Theta((n-m+1)\, m) = \Theta(n^2)$ *Zeit im Worst Case. Der Worst Case tritt auf, wenn* $m = \Theta(n)$*.*

Beweis. Betrachte $T = a^n = aaa \ldots a$ (n Wiederholungen von a) und $P = a^m$. Für jede der $n - m + 1$ möglichen Verschiebungen muss die implizite

[1] englisch "Prefix"

Algorithmus 11.1 NAIVE-STRING-MATCHING(T, P)

```
1: n ← T.length
2: m ← P.length
3: for s = 0 to n − m do
4:     if P[1, ..., m] = T[s + 1, ..., s + m] then
5:         print Pattern occurred with Shift s
```

Schleife in Zeile 4 m mal durchlaufen werden. Wenn wir $m = n/2$ wählen, folgt die Behauptung. □

NAIVE-STRING-MATCHING ist *nicht* optimal, weil der Algorithmus jedes Mal „vergisst“, was er bereits über den Stringvergleich weiß und so bei jedem Nicht-Match (Fehler) von vorne anfangen muss (Zeile 4).

11.2.3 String Matching und FSMs

Exkurs FSMs

Endliche Automaten (englisch “Finite State Machines” oder FSMs) sind aus der theoretischen Informatik bekannt und sehr nützlich in der Praxis. Wir werden im folgenden einen Anwendungsfall sehen.

Definition 11.2.1. *Ein endlicher Automat ist ein 5-Tupel* $(Q, q_0, A, \Sigma, \delta)$ *mit*

1. Q *ist eine endliche Menge von Zuständen (States).*
2. q_0 *ist der Anfangszustand.*
3. $A \subseteq Q$ *ist eine Untermenge von* akzeptierenden *Zuständen*[2]*.*
4. Σ *ist ein Alphabet.*
5. δ *ist eine Funktion von* $Q \times \Sigma \to Q$*. Sie heißt* Übergangsfunktion *(englisch “Transition Function”).*

Ein Beispiel für einen endlichen Automaten, der überprüft, ob ein Text T eine gerade oder ungerade Anzahl von Buchstaben enthält, ist in Abbildung 11.2 gegeben.

Man kann die Übergangsfunktion von FSMs auch mithilfe einer Tabelle 11.1 anstelle eines Graphen darstellen.

Einen etwas komplizierteren Automaten veranschaulicht Abbildung 11.3. Er entscheidet, ob das aktuelle Zeichen sich in einem $C/C++$-Kommentar befindet.

[2] häufig nur genau einer!

Abbildung 11.2: FSM Gerade / Ungerade (Der akzeptierender Zustand ist “even”)

Tabelle 11.1: Übergangsfunktion für Gerade / Ungerade

	$c \in \Sigma$ beliebig
even	odd
odd	even

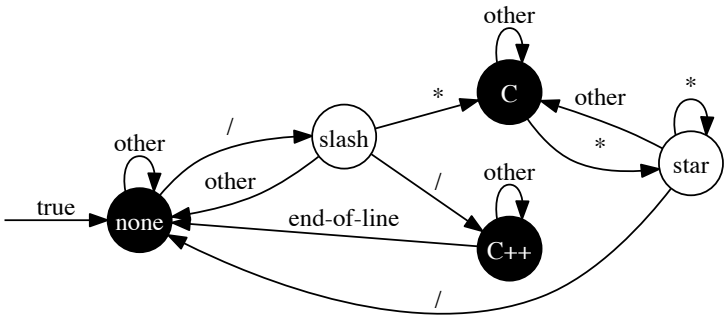


Abbildung 11.3: Automat für $C/C++$ Kommentare

Diese beiden Beispiel verdeutlichen bereits die Leistungsfähigkeit von Automaten. Sie kennen endliche Automaten vielleicht auch aus der Theorie oder Praxis von regulären Ausdrücken (englisch “regular expressions”).

Abbildung 11.4 illustriert eine FSM, die erkennt, ob im Text das Wort “nano” mindestens einmal vorkommt.

Wenn man alle Vorkommen des Wortes “nano” erkennen möchte, dann sind le-

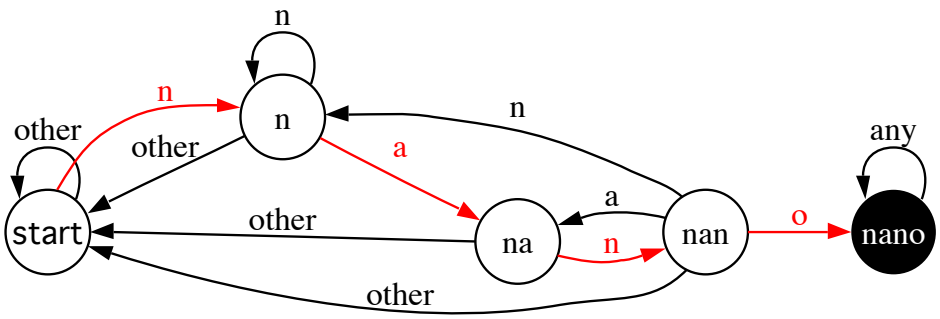


Abbildung 11.4: FSM für “nano” Erkennung

diglich drei Modifikationen an der FSM in Abbildung 11.4 erforderlich:

1. Entfernen der Kante “any” von “nano” nach “nano”.
2. Einführen einer Kante “n” von “nano” nach “n”.
3. Einführen einer Kante “other” von “nano” nach “start”.

Jedes Mal, wenn der Zustand “nano” erreicht wird, wurde das Wort erkannt.

Algorithmus FSM-MATCHER

Die Grundidee des Algorithmus FSM-MATCHER 11.2 ist nun, einen Automaten zu konstruieren, der „weiss“, wo man sich in einer Zeichenkette befindet und so den String-Match effizienter durchzuführen. Natürlich muss man zuvor diesen Automaten konstruieren.

Algorithmus 11.2 FSM-MATCHER(T, δ, m)

```
1: n ← T.length
2: q ← 0
3: for i = 1 to n do
4:     q ← δ(q, T[i])
5:     if q = m then
6:         print  Pattern occurred with Shift i − m
```

Die Übergangsfunktion können wir mithilfe des Algorithmus 11.3 berechnen. Wichtig ist, dass diese Berechnung *nicht* vom Eingabetext T, sondern *nur* vom Muster P anhängt! Im folgenden bezeichnet für einen Buchstaben c der

Ausdruck $P_q c$ denjenigen String, der durch „Anhängen“ von c an P_q entsteht. Der Algorithmus COMPUTE-TRANSITION-FUNCTION berechnet die

Algorithmus 11.3 COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1: m ← P.length
2: for q = 0 to m do
3:     for each c ∈ Σ do
4:         k ← min(m + 1, q + 2)
5:         repeat
6:             k ← k − 1
7:         until P_k ⊐ P_q c
8:         δ(q, c) ← k
9: return δ
```

Übergangsfunktion, in dem in den Zeilen 2 und 3 alle Zustände q und alle Buchstaben c des Alphabets durchgegangen werden und in den Zeilen 4–8 $\delta(q, c)$ auf den größten Wert gesetzt wird, so dass $P_k \sqsupset P_q c$, also P_k ein Suffix von $P_q c$ ist.

Wir verdeutlichen die Vorgehensweise an einem Beispiel. Sei $P = ababaca$. Dann ist der zugehörige FSM in Abbildung 11.5 bzw. Tabelle 11.2 dargestellt[3].

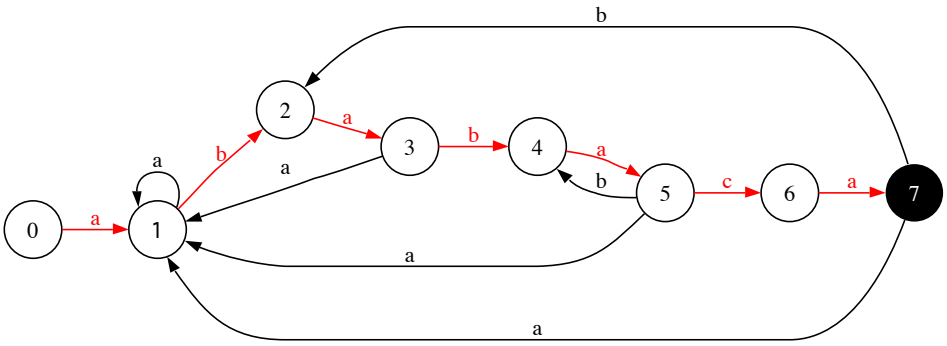


Abbildung 11.5: Übergangsfunktion für $P = ababaca$

Als Beispiel betrachten wir den Zustand $q = 5$, siehe auch Abbildung 11.6.

Wenn als nächstes Zeichen ein c kommt, dann gehen wir in den Zustand $q = 6$ über (nicht dargestellt). Wenn als nächstes Zeichen ein b kommt, dann

[3]Dabei sind aus Gründen der Übersichtlichkeit alle trivialen Zustandsübergänge in den Grundzustand weggelassen, also ist z. B. der fehlende Pfeil für ein c im Zustand 6 so zu lesen, dass der Automat dann in den Zustand 0 zurückspringt

Tabelle 11.2: Übergangsfunktion für $P = ababaca$

	Input			
State	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

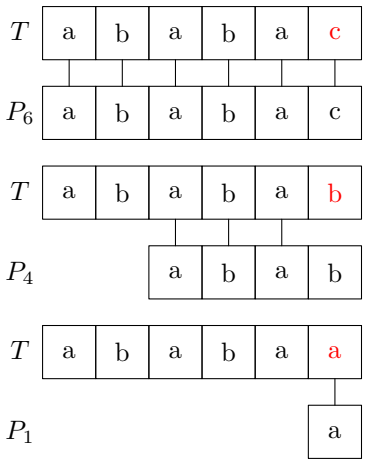


Abbildung 11.6: $q = 5$

haben wir zwar keinen Match. Der Automat weiss aber, dass das den Zustand $q = 4$ (und nicht etwa $q = 0$) impliziert, da die Sequenz *abab* immer noch valide ist. Kommt dagegen ein a, so müssen wir bis zum initialen ersten a verschieben, also in den Zustand $q = 1$ zurückfallen.

Der FSM-MATCHER kann ebenso wie die Übergangsfunktion COMPUTE--TRANSITION-FUNCTION mit wenigen Zeilen Python implementiert werden, wie das Listing 11.1 zeigt. Dabei wird die Übergangsfunktion als Tabelle (Dictionary, siehe Variable `d`) realisiert.

Die Ausgabe der Übergangsfunktion für das Muster *ababaca* und das Alphabet *abc* lautet (hier sortiert):

```
{(0, 'a'): 1, (0, 'b'): 0, (0, 'c'): 0,
 (1, 'a'): 1, (1, 'b'): 2, (1, 'c'): 0,
```

```
(2, 'a'): 3, (2, 'b'): 0, (2, 'c'): 0,
(3, 'a'): 1, (3, 'b'): 4, (3, 'c'): 0,
(4, 'a'): 5, (4, 'b'): 0, (4, 'c'): 0,
(5, 'a'): 1, (5, 'b'): 4, (5, 'c'): 6,
(6, 'a'): 7, (6, 'b'): 0, (6, 'c'): 0,
(7, 'a'): 1, (7, 'b'): 2, (7, 'c'): 0}
```

Algorithmus 11.4 FSM-MATCHER-P(T, δ, m)

Listing 11.1 FSM-MATCHER-P (Ausführbarer Pseudocode)

```
1  def compute_transition_function(p, s):
2      d = {}
3      m = len(p)
4      for q in range(0, m+1):
5          for a in s:
6              k = min(m+1, q+2)
7              suffix = False
8              while not suffix:
9                  k = k-1
10                 pa = p[:q] + a
11                 if pa[q-k+1: q+1] == p[:k]:
12                     suffix = True # P_k is suffix of P_q
13             d[(q,a)] = k
14     return d
15
16 def fsm_matcher(t, d, m):
17     n = len(t)
18     q = 0
19     for i in range(n):
20         q = d[(q, t[i])]
21         if q == m:
22             print("match␣at:␣%i" % (i - m + 1))
```

11.2.4 Analyse

Lemma 11.2.2. *FSM-MATCHER ist in* $\Theta(n)$.

Beweis. Die Schleife in Zeile 3 wird offensichtlich n mal durchlaufen. Da der der Zugriff auf den Zustand q in Zeile 4 in $\mathcal{O}(1)$ ist, folgt die Behauptung. □

Lemma 11.2.3. *COMPUTE-TRANSITION-FUNCTION ist in* $\mathcal{O}(m^3|\Sigma|)$, *also unabhängig von der Länge der Eingabe* n.

Beweis. Die Schleife in Zeile 2 liefert m Beiträge, die Schleife in Zeile 3 $|\Sigma|$, die innere Schleife in Zeilen 5–7 liefert höchstens $m + 1$ und der Vergleich in Zeile 7 kann im ungünstigen Fall m Buchstaben erfordern. □

Bemerkung:

1. Die Kosten für die Konstruktion des Automaten treten nur *einmal* auf.
2. Man kann den Automaten aber „cleverer“ konstruieren und nur $\Theta(m|\Sigma|)$ benötigen, wie wir im nächsten Abschnitt sehen werden.

11.2.5 Knuth-Morris-Pratt

Der Knuth-Morris-Pratt Algorithmus verbessert den naiven Algorithmus dadurch, dass er Informationen, die beim Vergleich entstehen, nicht wegwirft, sondern geschickt Eigenschaften des Musters selbst ausnutzt, um überflüssige Vergleiche zu überspringen. Man kann ihn auch als „Simulation“ des FSM Algorithmus verstehen, wie wir im folgenden noch sehen werden. Die Grundidee ist, dass man, wenn man einen partiellen Match von j Buchstaben an der Stelle i gefunden hat, man daraus schließen kann, wie die Positionen $T[i], \ldots, T[i+j-1]$ aussehen. Dazu berechnet man im Voraus die so genannte *Präfixfunktion*. Sie vergleicht das Muster mit sich selbst.

Definition 11.2.2. *Der Überlapp (englisch “overlap”) zweier Strings x und y ist der längste String, der zugleich Suffix von x und Präfix von y ist.*

Definition 11.2.3. *Die Präfixfunktion π eines Musters P berechnet den längsten Überlapp des Musters mit sich selbst.*

Als Beispiel berechnen wir die Präfixfunktion des Strings *ababaca*. Es gilt offenbar: $\pi(ababaca) = aba$.

Die Präfixfunktion lässt sich durch den COMPUTE-PREFIX-FUNCTION Algorithmus 11.5 berechnen, womit sich dann der eigentliche Algorithmus KMP-MATCHER 11.6 leicht implementieren lässt.

Algorithmus 11.5 COMPUTE-PREFIX-FUNCTION(P)

```
 1: m ← P.length
 2: π[1, ..., m] ← new Array
 3: π[1] ← 0
 4: k ← 0
 5: for q = 2 to m do
 6:     while k > 0 and P[k + 1] ≠ P[q] do
 7:         k ← π[k]
 8:     if P[k + 1] = P[q] then
 9:         k ← k + 1
10:     π[q] ← k
11: return π
```

Wir überlegen uns die Funktionsweise anhand eines Beispiels. Die Präfixtabelle für das Muster *ababaca* ist in Tabelle 11.3 dargestellt, die zu-

Algorithmus 11.6 KMP-MATCHER(T, P)

```
1:  n ← T.length
2:  m ← P.length
3:  π ← COMPUTE-PREFIX-FUNCTION(P)
4:  q ← 0
5:  for i = 1 to n do
6:      while q > 0 and P[q + 1] ≠ T[i] do
7:          q ← π[q]
8:      if P[q + 1] = T[i] then
9:          q ← q + 1
10:     if q = m then
11:         print  Pattern occurred with shift i − m
12:         q ← π[q]
```

gehörigen Zustandsübergänge des KMP-MATCHERS in Abbildung 11.7 verdeutlicht.

Tabelle 11.3: Präfixtabelle

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi(i)$	0	0	1	2	3	0	1

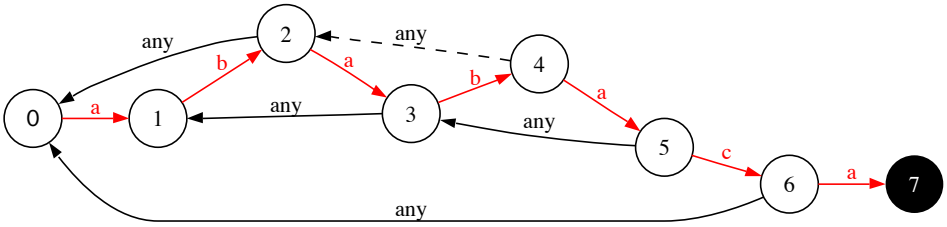


Abbildung 11.7: Zustandsübergänge

Als Beispiel nehmen wir den Index 5. Angenommen der Text hat an der Stelle $i = 6$ ein b und kein c. Dann liegt die Situation vor, die in Abbildung 11.8 dargestellt ist. Wenn dagegen der Text an der Stelle $i = 6$ ein a und kein c hat, dann liegt die Situation vor, die in Abbildung 11.9 dargestellt ist. Wir wissen, dass an der Stelle $i = 6$ kein Match vorliegt. Aufgrund der Präfixtabelle wissen wir aber, dass die längste Sequenz die Suffix und Präfix ist 3 beträgt, d.h. die Sequenz *aba* ist an den Stellen $i = 3$, 4 und 5 gematcht. Wir können also das Muster P um 2 Stellen verschieben und mit den Vergleichen fortfahren.

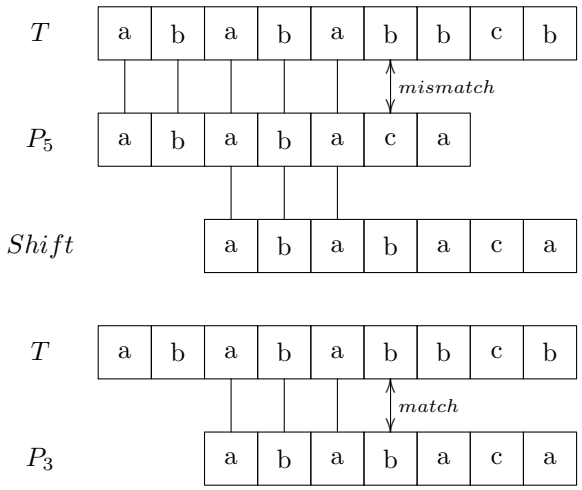


Abbildung 11.8: Präfix im Zustand P_5 zu P_3

Wie man sieht ist dieses Verfahren dem FSM-Verfahren sehr ähnlich. Anstelle jedoch direkt die Zustände in einer Tabelle nachzuschauen, werden beim KMP Verfahren die Zustände "on-the-fly" mithilfe der Präfixfunktion berechnet. Wie das obige Beispiel verdeutlicht ist dieses Verfahren u.U. ineffizienter, da man sich schrittweise von z. B. P_5 zu P_1 „durchhangelt", anstelle den Übergang direkt zu berechnen. Der FSM kann dies, weil er die Information, welches Zeichen als nächstes kommt, berücksichtigt – deshalb hat COMPUTE-TRANSITION-FUNCTION auch Σ als Eingabeparameter. Die Präfixfunktion dagegen ist vom Alphabet unabhängig und muss daher zur Laufzeit, in Abhängigkeit, welches Zeichen als nächstes kommt, diese Information berechnen. Man kann dies als eine Variante der Dynamischen Programmierung auffassen.

11.2.6 Analyse

Lemma 11.2.4. *KMP-MATCHER ist korrekt.*

Der Beweis dieser Tatsache lässt sich darauf zurückführen, dass KMP die FSM simuliert und wir die Korrektheit von FSM bereits gezeigt haben. Um zu beweisen, dass KMP FSM simuliert, muss man zeigen, dass COMPUTE-PREFIX-FUNCTION die Präfixfunktion korrekt berechnet. Dies ist jedoch überraschend kompliziert und wir verweisen daher auf [CLR04].

Lemma 11.2.5. *COMPUTE-PREFIX-FUNCTION ist in* $\Theta(m)$.

Beweis. Anhand des Pseudo-Codes könnte man vermuten, dass der Algorithmus schlechtere Laufzeiteigenschaften hat, da eine innere Schleife vorkommt.

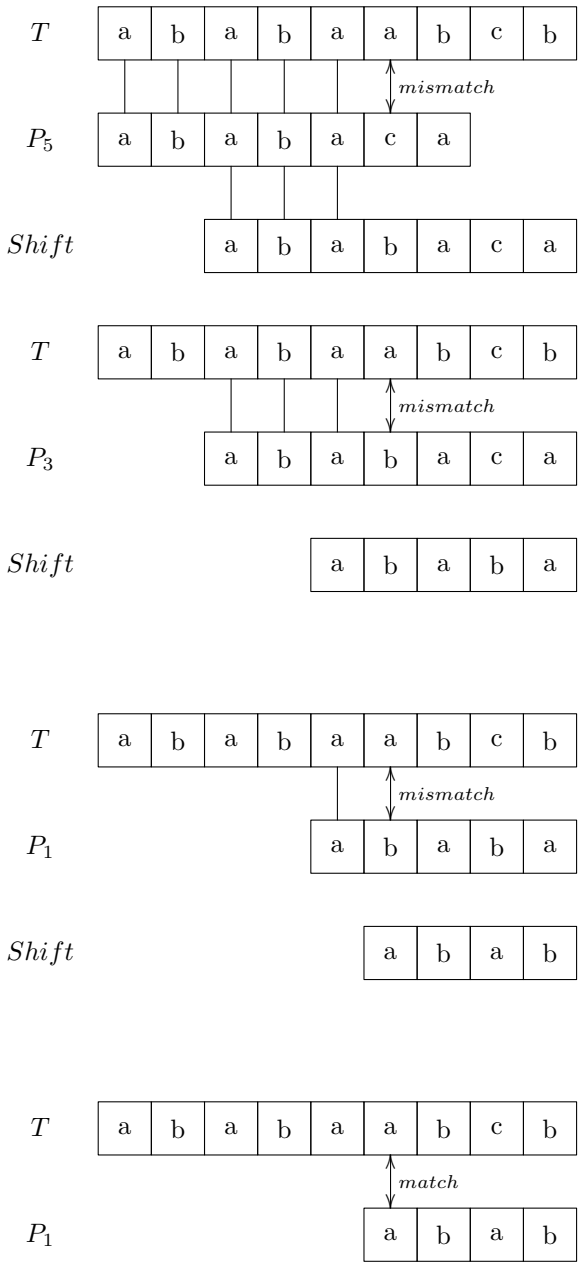


Abbildung 11.9: Präfix im Zustand P_5 zu P_1

Wir zeigen, dass dies nicht der Fall ist. Die Variable k startet bei 0 und wird in Zeile 9 erhöht. Man kann mithilfe vollständiger Induktion sehen, dass Zeile 7 k nicht erhöhen kann, sondern dass vielmehr $\pi[q] < q$ für alle q gilt. Außerdem wird k niemals negativ. Damit kann die while Schleife (Zeile 6)

maximal so häufig durchlaufen werden, wie Zeile 9 ausgeführt wird, nämlich $m-1$ mal (ansonsten würde k negativ). □

Lemma 11.2.6. *KMP-MATCHER (ohne COMPUTE-PREFIX-FUNCTION) ist in* $\Theta(n)$.

Beweis. Analog zu oben. □

Lemma 11.2.7. *Die Gesamtlaufzeit von KMP-MATCHER ist in* $\Theta(n+m)$.

Beweis. Folgt aus den beiden vorherigen Aussagen. □

11.2.7 Bewertung

Somit ist KMP ein – gegenüber dem naiven Algorithmus – deutlich effizienterer Algorithmus. In der Praxis ist bei natürlichen Sprachen (Alphabet) der naive Algorithmus jedoch häufig gar nicht schlecht, da die Fehler „schnell“ auftreten und somit der Worst-Case irrelevant ist. Man sieht das auch daran, dass das Worst Case Beispiel (a^n) recht „künstlich“ ist. Daher ist in der Praxis der Vorteil des KMP und FSM geringer als man annimmt. Allerdings sind FSMs effizient: Jeder Buchstabe wird genau einmal angeschaut und FSMs müssen nicht im Text zurückgehen – ein wesentlicher Vorteil, wenn man Daten von seriellen Medien (z. B. Bandlaufwerken) liest. Und FSMs können auch kompliziertere Automaten etwa für reguläre Ausdrücke realisieren.

11.3 Weitere Verfahren

Es gibt noch andere Verfahren zum String-Matching, z. B. der Rabin-Karp Algorithmus oder Boyer-Moore, die ganz anders aufgebaut sind, als die hier vorgestellten Algorithmen. Rabin-Karp verwendet ein Hash-Verfahren, Boyer-Moore geht das Muster *von hinten* durch.

11.4 Fragen und Aufgaben zum Selbststudium

1. Berechnen Sie die Präfixfunktion π für das Muster

   $$ababbbabbababbbabbbab!$$

2. Vollziehen Sie die Beweisführung im Lemma 11.2.5. anhand eines Beispiels nach!

3. NAIVE-STRING-MATCHING ist in $\mathcal{O}(n^2)$, KMP in $\mathcal{O}(n)$. Überlegen Sie sich, wo der n^2 Anteil für NAIVE-STRING-MATCHING herkommt und inwiefern für natürliche Sprachen der Unterschied zwischen NAIVE-STRING-MATCHING und KMP relevant ist. (Tipp: Überlegen Sie sich, ob ein Muster a^m mit $m \in \mathcal{O}(n)$ ein realistischer Anwendungsfall ist, und wie es für realistische Anwendungsfälle aussehen würde!)

Kapitel 12

Pagerank

Für die Theorie des Pagerank Algorithmus siehe u.a. `http://www.ams.org/samplings/feature-column/fcarc-pagerank`. Nachstehend im Listing 12.1 eine Implementierung in Python.

Listing 12.1: Page Rank (Ausführbarer Pseudocode)

```
def matrixMult(A, B):
    nrows = len(A)
    ncols = len(B[0])
    C = [[0 for i in range(ncols)] for j in range(nrows)]
    for i in range(nrows):
        for j in range(ncols):
            for k in range(len(A[0])):
                C[i][j] += A[i][k]*B[k][j]
    return C

def power(A, k):
    B = A
    for i in range(k-1):
        B = matrixMult(B, A)
    return B

def vec2matrix(v):
        return [[x,] for x in v]

def printMatrix(m):
    for i in range(len(m)):
        print(m[i])

if __name__ == '__main__':
        H = [[0,0,0,0,0,0, 1/3.0, 0],
             [0.5,0,0.5,1/3.0,0,0, 0, 0],
             [0.5,0,0,0,0,0, 0, 0],
             [0,1,0,0,0,0, 0, 0],
             [0,0,0.5,1/3.0,0,0, 1/3.0, 0],
             [0,0,0,1/3.0,1/3.0,0, 0, 0.5],
             [0,0,0,0,1/3.0, 0,0, 0.5],
             [0,0,0,0,1/3.0,1, 1/3.0, 0]]
        print("Google Matrix:")
        printMatrix(H)
        print("EV")
        printMatrix(matrixMult(power(H, 40),
             vec2matrix([1/8.0,1/8.0,1/8.0,1/8.0,1/8.0,1/8.0,1/8.0,1/8.0])))
```

Kapitel 13

Ausblick Fortgeschrittene Themen

Mögliche Themen (nicht behandelt):

1. Amortisierte Analyse
2. AVL-Bäume
3. Branch-and-bound
4. B-Trees
5. Computational Geometry
6. Fibonacci Heaps
7. FSMs
8. Kürzeste Wege zwischen allen Knoten eines Graphen
9. Lineare Optimierung
10. Minimum Spanning Trees und Algorithmen
11. Min-Max-Problem
12. Probabilistische Verfahren
13. Rucksackprobleme
14. Satz von Rice
15. Skiplisten
16. Algorithmus Handelsreisender
17. ...

Kapitel 14

Und ... – war es das jetzt schon?

Ja und Nein!

Ja, weil Sie jetzt die wichtigsten Grundlagen zu Algorithmen und Datenstrukturen kennen. Neben Algorithmen zur Suche, zum Sortieren sowie Algorithmen auf Graphen samt deren Analyse haben Sie Entwurfsprinzipien für Algorithmen kennen gelernt. Die Algorithmen geben an, welche Berechnungsschritte auf den Daten ausgeführt werden müssen, um ein Problem zu lösen. Wir haben diverse Datenstrukturen zur Organisation von Daten besprochen und Operationen auf diesen Datenstrukturen analysiert. Die Datenstrukturen repräsentieren die Information und ermöglichen dem jeweiligen Algorithmus den Zugriff in geeigneter Form.

Nein, weil neben dem Wissen über Algorithmen und Datenstrukturen mit ihren Eigenschaften und Einsatzbereichen zwar die Grundlagen zum Entwurf (Design) guter Algorithmen vorhanden sind. Eine weitere wichtige Voraussetzung ist die *Erfahrung beim Design*. Dies kann schwieriger vermittelt werden. Erfahrung muss man machen. Wenn Sie schon mit den Algorithmen und Datenstrukturen "herumgespielt" haben, können Sie stolz auf Ihre ersten Erfahrungen zurückblicken.

In diesem Kapitel wollen wir Ihnen ermöglichen, weitere Erfahrungen zu sammeln. Algorithmen und verwendete Datenstrukturen sind meist eng miteinander verzahnt. Da Designs zur Lösung "schwierigerer" Probleme oft nur Personen mit viel Erfahrung gelingen, möchten wir ein komplexeres Problem mit Ihnen gemeinsam lösen.

Dazu werden wir den Algorithmus und die benötigten Datenstrukturen schrittweise entwerfen. Insbesondere wird dabei die wechselseitige Abhängigkeit zwischen Algorithmus und Datenstrukturen klar.

Wie sehen nun solche Abhängigkeiten von Algorithmus und Datenstruktur aus?

- In Abschnitt 1.3 haben wir gesehen, dass eine effiziente Datenstruktur einen großen Einfluss auf die Laufzeit eines Algorithmus haben kann. Der Algorithmus in naiver Umsetzung benötigt eine Laufzeit, die quadratisch in der Größe der Eingabe ist. Bei Einsatz einer Hashtabelle ist die Laufzeit lediglich linear in der Größe der Eingabe.
- Wir erinnern und ans Suchen. Wenn ein Algorithmus eine effiziente Such-Funktion umsetzen soll, dann benötigen wir Voraussetzungen: Entweder eine sortierte statt einer unsortierten Liste oder sogar vorbereitete Datenstrukturen wie Hashtabellen.
- Wenn Daten in einer verketteten Liste gegeben sind, dann können viele Sortier-Algorithmen mit dieser Datenstruktur nicht effizient arbeiten.

Dazu passt der Titel eines alten Buches zum dem Thema Algorithmen und Datenstrukturen sehr gut: *Algorithms + Data Structures = Programs* [Wir76]. Programme bestehen zwar nicht aus der "Summe" der Algorithmen und Datenstrukturen, aber aus einer geeigneten Kombination.

Abschließend wollen wir also in diesem Kapitel das Zusammenspiel von Algorithmen und Datenstrukturen nochmals etwas genauer beleuchten, damit Sie passende Lösungen für gestellte Probleme entwickeln können.

Hier gibt es bewusst kleine Vorgriffe auf die Module *Software Engineering – Analysis* und *Software Engineering – Design*. Jedoch steht natürlich das Modul Algorithmen und Datenstrukturen im Vordergrund.

14.1 Problemverständnis

Eine Lösung für ein nicht verstandenes Problem zu entwickeln, ist wenig sinnvoll. Daher muss am Anfang der Entwicklung hinreichend viel Zeit in die Analyse des Problems investiert werden. Es ist meist nicht ausreichend, wenn Sie wissen, was funktional zu machen ist, beispielsweise "Sortieren von Datensätzen". Auch die Rahmenbedingungen der Entwicklung müssen bekannt sein, zum Beispiel:

- Wie viele Eingabedaten wird es geben, also wie "groß" ist das Problem?
- Welche Eigenschaften haben die Eingabedaten? Sind alle Eingabedaten gleichwahrscheinlich?
- Wieviel Speicher steht zur Verfügung?
- Welche Rechenzeiten sind gewünscht oder gerade noch akzeptabel? Wie sieht es im Mittel und im schlechtesten Fall aus?

- Welche Abweichung von einer optimalen Lösung ist akzeptabel?
- Wie lange darf die Entwicklung einer Lösung dauern?

Der Grund für die Notwendigkeit des Problemverständnisses ist offensichtlich. Unterschiedliche Algorithmen lösen das gleiche Problem in unterschiedlicher Güte, also zum Beispiel mit unterschiedlichen Speicherplatzbedarf und unterschiedlicher Laufzeit. Wir möchten einen Algorithmus finden, der die für uns wichtigen Ziele gut erfüllt.

Wir haben verschiedene Algorithmen kennen gelernt, die das gleiche Problem in unterschiedlicher Güte lösen. Beispielsweise sind die naiven Sortieralgorithmen verständlicher und leichter analysierbar, während die komplexeren eine bessere Laufzeit haben. Es ist immens wichtig, sich dieser Problematik bewusst zu sein, um geeignete Algorithmen und Datenstrukturen entwerfen zu können.

14.1.1 Trade-offs

Beim Design von Systemen und Software existieren eigentlich immer Zielkonflikte, also sich widersprechende Ziele wie Ressourcenbedarf, Verständlichkeit, Kosten und Entwicklungszeit.

Übertragen auf Algorithmen und Datenstrukturen könnten wir zum Beispiel Folgendes fordern: Wir hätten gerne eine Lösung eines gegebenen Problems mit einem einfachen Algorithmus, der nur rudimentäre Datenstrukturen verwendet. Außerdem soll er sehr gut verständlich (und damit auch wartbar) sein und selbstverständlich fast keinen Speicher verbrauchen und in kürzester Zeit unabhängig von der Größe der Eingabe mit der Berechnung fertig sein. Ach ja, hätten wir fast vergessen: Natürlich soll die Entwicklung innerhalb kürzester Zeit abgeschlossen sein und das Programm keine Fehler haben.

Diese “Maximalforderung” ist nicht realisierbar, höchstens für extrem einfache Problemstellungen mit wenigen Daten. Die verschiedenen Güte-Ziele widersprechen sich.

Daher muss dieser Zielkonflikt aufgelöst werden. Dazu ist es erforderlich, die oben genannten Rahmenbedingungen möglichst genau zu kennen.

Wenn nicht alle Ziele gleichzeitig erfüllt werden können (und das ist in der Praxis immer so), muss ein Kompromiss gefunden werden. Solche Trade-off-Entscheidungen müssen bewusst getroffen werden. Dies veranschaulichen wir Ihnen anhand eines kleinen (aber feinen) Beispiels.

Der klassische Trade-off der Informatik ist das Abwägen zwischen den beiden Ressourcen Laufzeit und Speicher. Oft können Algorithmen und Datenstrukturen so geändert werden, dass die Laufzeit besser wird, jedoch mehr

Speicherplatz erforderlich ist. Oder der Speicherplatzbedarf wird verringert auf Kosten der Laufzeit. Es ist also oft möglich Speicher für Laufzeit (oder auch umgekehrt) zu "kaufen". Ein gutes Beispiel ist ein Algorithmus, der dynamisches Programmieren verwendet. Er speichert Zwischenergebnisse ab, damit diese zukünftig nicht erneut berechnet werden müssen. Somit wird die Laufzeit verbessert auf Kosten des benötigten Speicherplatzes.

Die Wichtigkeit der einzelnen Ziele gehört also zu den wesentlichen Anforderungen an die Lösung.

Mit einem guten Verständnis der Problemstellung sind Sie gut gerüstet zum Entwurf der Lösung für das Problem.

14.2 Vom Problem zur Lösung

Als erstes richten wir unseren Blick auf das verstandene Problem und fragen uns: Ist es überhaupt möglich, eine Lösung für das Problem zu finden?

Für die meisten Probleme, die in der Praxis gelöst werden müssen, existiert eine Lösung. Vorsicht ist geboten, wenn qualitative Eigenschaften zu 100% erfüllt werden müssen. Beispielsweise sind 100 % *sichere* (sowohl *Safety* als auch *Security*) oder 100% *verfügbare Systeme* nicht realisierbar.

Wir haben aber auch andere unlösbare Probleme kennen gelernt, wie zum Beispiel das Halteproblem, siehe Abschnitt 3.7.

Und dann gibt es noch die Klasse der **NP**-vollständigen Probleme (Abschnitt 3.6). Diese sind zwar *theoretisch* für beliebige Problemgrößen lösbar, jedoch nicht in akzeptabler Rechenzeit, somit *praktisch* für "große" Problemgrößen unlösbar. Daher muss in der Praxis bei entsprechenden Problemgrößen häufig auf Heuristiken oder Approximationen zurückgegriffen werden. Dies ist schon beim Design des Algorithmus und seiner Datenstrukturen zu berücksichtigen.

14.3 Design der Lösung

Wie schon oben angesprochen ist das Design von Algorithmen und Datenstrukturen und deren Zusammenspiel alles andere als trivial. Auf jeden Fall müssen Algorithmen und Datenstrukturen gemeinsam entworfen werden, damit sie sich sinnvoll ergänzen. Die Datenstrukturen müssen den Zugriff auf die geforderten Daten in der geforderten Güte ermöglichen. Der Algorithmus muss in der Lage sein, mit der zur Verfügung stehenden Schnittstelle effizient zu arbeiten.

Natürlich sollten generelle Regeln beim Software-Design berücksichtigt werden, drei wichtige Konzepte lauten:

- *Separation of concerns* – Versuchen Sie, trennbare Aspekte auch im Algorithmus und den Datenstrukturen zu trennen, sofern dies möglich ist.
- *Keep it simple* – Je einfacher, desto besser, falls die Anforderungen erfüllt werden. Oft findet man durch Nachdenken einfachere, verständlichere und effizientere Lösungen.
- *No premature optimization* – Auf jeden Fall sollten Optimierungen nur dort durchgeführt werden, wo sie wirklich benötigt werden. Wenn zum Beispiel früh in der Entwicklung (ohne Grund) versucht wird, Speicherplatz zu sparen, könnten spätere Laufzeitprobleme wegen dieser Optimierung die Folge sein.

Hier muss der Fokus stark darauf gerichtet sein, wie der Algorithmus die Lösung in der erforderlichen Güte bereitstellen kann. Wenn der Algorithmus beispielsweise für einzelne Entscheidungen zu wenig Information besitzt, dann kann überlegt werden, ob man zum Beispiel eine Vorberechnung macht und speichert, damit der Algorithmus dann effizienter arbeiten kann. Sollte der Zugriff auf einzelne Elemente zu lange dauern, dann wäre zu überlegen, welche Datenstruktur einen schnelleren Zugriff ermöglicht. In diesem Skript haben Sie viele Beispiele kennen gelernt, in denen sich Algorithmen und Datenstrukturen ergänzen, auch wenn nicht immer explizit darauf hingewiesen wurde.

Der Schlüssel zum guten Design ist das Wissen über Algorithmen und Datenstrukturen und deren Eigenschaften. Wie oben angesprochen hilft Erfahrung sehr viel, um die richtigen Ideen und geeigneten Ansätze zur Kombination unterschiedlicher Algorithmen und Datenstrukturen zu finden.

14.3.1 Analyse der Lösung

Sie haben in der Veranstaltung verschiedene Analysenarten für Algorithmen und Datenstrukturen kennen gelernt. Auch haben Sie sicher schon ein Gefühl entwickelt, welche Ansätze zu welchen Anforderungen passen. Natürlich muss in der Design-Phase schon berücksichtigt werden, was für den zu entwickelnden Algorithmus besonders wichtig ist.

Sollten Sie eine Analyse durchführen, muss ein wichtiger Punkt berücksichtigt werden: In Artikeln, Büchern und auch diesem Skript trifft man für Analysen meist eine Annahme: Gleichwahrscheinlichkeit (zum Beispiel der Eingabe-Daten), siehe auch Kapitel 3.5. Dies gilt aber in der Praxis nicht immer und kann für die Analyse (oder schon vorher für das Design) relevant sein. Dazu zwei Beispiele:

- Lottozahlen: Zwar sollten bei der Ziehung alle Zahlen gleichwahrscheinlich sein, jedoch gilt dies nicht für die getippten Zahlen.
- Die Verteilung der Geburtstage auf die Tage des Jahres wie schon in Abschnitt 6.6.2 erwähnt.

Die Vertiefung dieses Themas würde den Rahmen des Skripts sprengen, jedoch wollten wir dieses praxisrelevante Thema nicht unerwähnt lassen.

Es ist auch wichtig, die Rahmenbedingungen für akzeptable Laufzeiten im mittleren oder schlechtesten Fall zu kennen. Falls die Laufzeit im schlechtesten Fall nicht akzeptabel ist, kann man zum Beispiel analysieren, wie wahrscheinlich der schlechteste Fall ist. Wenn man Glück hat, tritt er vielleicht bei den möglichen Eingabedaten gar nicht auf. Falls doch, muss eine andere Lösung gesucht werden. Dies muss natürlich auch in die Analyse einfließen.

14.4 Beispiel

Ein schönes Beispiel für das Ineinandergreifen von Algorithmus und Datenstrukturen ist eine algorithmische Lösung des Schiebespiels in Abbildung 14.1. Diese Schiebespiel kennt man unter vielen unterschiedlichen Namen wie *Klotski*, *Luzifix* und *Quo vadis*. Die *Start-Konstellation* ist im linken Teil der Abbildung gezeigt. Das Ziel ist es, den roten Stein durch Schiebevorgänge der Steine nach unten in die Mitte zu bewegen, wobei immer alle Steine im Spielfeld verbleiben müssen. Die zu erreichende *Ziel-Konstellation* ist im rechten Teil der Abbildung 14.1 gezeigt. Es ist dort lediglich der rote Stein vorhanden, die Positionen der anderen Steine sind unerheblich.

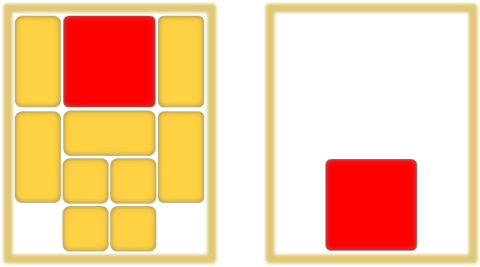


Abbildung 14.1: Start-Konstellation und Ziel-Konstellation des Schiebespiels

Nun wollen wir einen Algorithmus entwickeln, der eine minimale Anzahl von Schiebevorgängen von der Start-Konstellation zur Ziel-Konstellation findet und natürlich den zugehörigen Weg, um das Schiebespiel zu lösen. Zunächst dazu noch ein paar Definitionen.

Das Spielfeld ist ein *Raster*, dessen einzelne *Felder* durch die Größe des kleinsten Steins gegeben sind. Das Spielfeld besteht also aus fünf Zeilen und

vier Spalten. Ein *Schiebevorgang* ist die Bewegung von einem Stein um ein Feld im Raster. Als *Konstellation* bezeichnen wir einen Zustand, bei dem alle Steine am Raster ausgerichtet sind.

Der *Lösungsgraph* ist der Graph, der alle Konstellationen enthält mitsamt der notwendigen Schiebevorgänge, um von einer Konstellation zu einer anderen zu kommen. Inwieweit dieser Graph im Algorithmus wirklich aufgebaut werden muss, sehen wir gleich.

Zur Veranschaulichung finden Sie in der Abbildung 14.2 einen kleinen Ausschnitt aus dem Lösungsgraphen. In der Mitte oben befindet sich die Start-Konstellation. Außenherum die vier möglichen Nachfolgerkonstellationen. An den blauen Pfeilen sind die entsprechenden Steine und Schieberichtungen notiert. Für alle vier Nachfolger-Konstellationen gibt es wiederum Nachfolger-Konstellationen. Es lässt sich leicht erkennen, dass der komplette Lösungsgraph sehr groß ist.

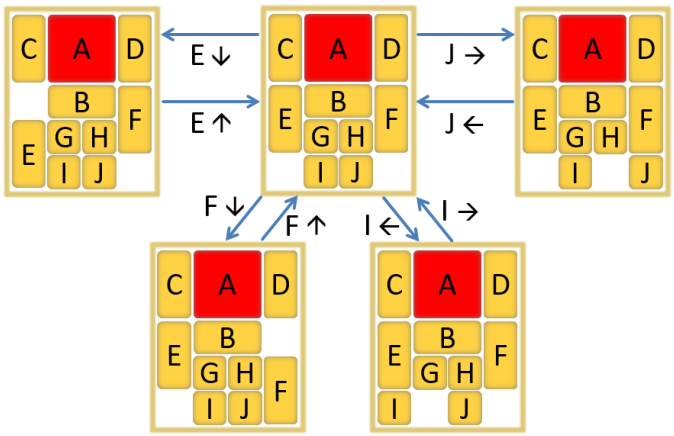


Abbildung 14.2: Übergänge von und zur Start-Konstellation im Lösungsgraph

14.5 Gedanken zum Algorithmus

Für den Algorithmus gibt es mindestens zwei Ansätze. Wir könnten das Schiebespiel (theoretisch) genauer analysieren – solche Analysen können sehr aufwendig werden. Wir würden (eventuell) Wissen über die Konstellationen und deren "Grade der Optimalität" erhalten. Dies könnte genutzt werden, um den Algorithmus zu entwickeln.

Wir wählen hier einen anderen Ansatz: Der Algorithmus soll die Möglichkeiten der Schiebevorgänge intelligent durchspielen, und dabei eine optimale Lösung finden. Es könnte mehrere gleich gute Lösungen geben; wir sind mit einer optimalen zufrieden, sollte es mehrere geben.

Damit ist klar, dass wir die möglichen Schritte in einer Form durchlaufen müssen, die auf jeden Fall zu einer optimalen Lösung führt.

14.5.1 Greedy-Ansatz

Zunächst hört sich das nach einer Greedy-Strategie (siehe Abschnitt 9.2.2) an. Dieser Ansatz ist jedoch hier nicht der richtige, da wir keine Bewertungsfunktion haben, also nicht wissen, wie "weit" es von einer gegebenen Konstellation bis zu einer optimalen Lösung ist. Wir kennen nämlich keine optimalen Teillösungen, da wir das Spiel nicht (theoretisch) analysiert haben. Siehe dazu Optimalitätsprinzip von Bellman in Definition 9.2.1. Dennoch wird unser Algorithmus für jede Konstellation nur einen der kürzest-möglichen Lösungswege finden.

14.5.2 Backtracking

Ein anderer Ansatz könnte Backtracking (siehe Abschnitt 9.2.3) sein. Der Algorithmus beginnt dann, virtuell Steine auf dem Spielfeld zu verschieben, bis eine Lösung gefunden ist. Wenn dies der Fall ist, macht der Algorithmus Backtrack-Schritte und sucht nur noch bessere Lösungen. Der Nachteil bei dem Verfahren ist, dass der gesamte Lösungsgraph abgearbeitet werden muss. Jedoch kann die Suche nach optimalen Lösungen abgebrochen werden, wenn keine bessere Lösung gefundenen werden kann als die beste bislang bekannte. Bei diesem Ansatz muss man unter anderem berücksichtigen, dass der Algorithmus nicht in unendliche Schleifen gerät, falls zum Beispiel ein Stein immer wieder hin- und hergeschoben wird, ohne neue Konstellationen zu finden.

Auf diesem Weg würde man in jedem Fall eine optimale Lösung finden, jedoch erscheint dies ineffizient. Unter Umständen wird eine optimale Lösung sogar bereits am Anfang gefunden. Allerdings wissen wir zu diesem Zeitpunkt nicht, dass diese wirklich optimal ist. Diese Erkenntnis haben wir erst, wenn alle Konstellationen, die zu einer optimalen Lösung führen können, analysiert sind. Somit muss der Algorithmus alle Teile des Lösungsgraphen kennen, die noch zu besseren Lösungen führen könnten.

Backtracking impliziert ja eine Tiefensuche (siehe Abschnitt 7.1.4) im Baum der Konstellationen. Der Algorithmus muss immer wieder bestehende Konstellationen überprüfen und sogar gegebenenfalls den Weg zu bekannten Konstellationen ändern. Mit diesem Ansatz kommen wir hier anscheinend nicht weiter, da viele Konstellationen des Lösungsgraphen immer wieder betrachtet und geändert werden müssen. Unser Ziel ist das Finden eines Ansatzes, der direkt mögliche optimale Teillösungen identifiziert und abspeichert.

Wir haben damit bereits eine wichtige Eigenschaft des Algorithmus gefunden:

Eine Konstellation muss nur für den optimalen Fall (also mit möglichst wenigen Schiebevorgängen) betrachtet werden. Somit wäre keine Änderung an der Anzahl der Schiebevorgänge und am besten Weg zu der Konstellation notwendig.

Und wir haben eine weitere Erkenntnis gewonnen: Wir müssen für jede Konstellation nur *eine* Vorgänger-Konstellation kennen, vom Lösungsgraph wird also nur ein Teil benötigt, ein *Lösungsbaum*.

14.5.3 Breitensuche

Die Idee des Lösungsbaumes greifen wir direkt wieder auf. Wir traversieren (beziehungsweise erzeugen) den Baum jetzt aber nicht mit der Tiefensuche, sondern mit der Breitensuche, siehe Abschnitt 7.1.4.

Die Idee dahinter ist einfach: In jeder Ebene i des Lösungsbaumes befinden sich alle Konstellationen, die mit der minimalen Anzahl i von Schiebevorgängen aus der Start-Konstellation erreicht werden können.

Somit befindet sich in Ebene 0 nur die Start-Konstellation. In Ebene i sind alle Konstellationen, die direkt aus einer Konstellation der Ebene $i-1$ durch einen Schiebevorgang erreicht werden können, jedoch noch nicht im Baum vorhanden sind. Insbesondere befinden sich in Ebene 1 alle Konstellationen, die direkt aus der Start-Konstellation durch einen Schiebevorgang erreicht werden können.

In der Abbildung 14.3 sehen wir oben (Ebene 0) die Start-Konstellation als Wurzel des Lösungsbaumes. In der Ebene 1 befinden sich ihre aus Abbildung 14.2 bekannten Nachfolger-Konstellationen. In der Ebene 2 sind alle neuen Nachfolger-Konstellationen der links stehenden Konstellation aus der Ebene 1 angegeben.

Jede Konstellation muss natürlich nur beim ersten Finden in den Lösungsbaum aufgenommen werden. Wenn im Algorithmus eine Konstellation erneut gefunden wird, dann kann die Anzahl der Schiebevorgänge nicht geringer sein, somit muss diese nicht weiter betrachtet werden.

Am Beispiel der Abbildung 14.3 wird klar, dass die Start-Konstellation nicht mehr als Nachfolger-Konstellation der Konstellationen in Ebene 1 aufgeführt wird. Ebenfalls können zwei Konstellationen in der Ebene 2 auf jeweils einem weiteren Weg erreicht werden. Auch dies wird im Lösungsbaum ebenfalls nicht erneut abgespeichert. Auch die Verbindung wird nicht abgespeichert. In der Abbildung ist dieser Zusammenhang durch die grauen Pfeile symbolisiert.

Nachdem die Idee für den Algorithmus geboren ist, müssen wir nun über die Datenstrukturen nachdenken. Es ergeben sich folgende Anforderungen:

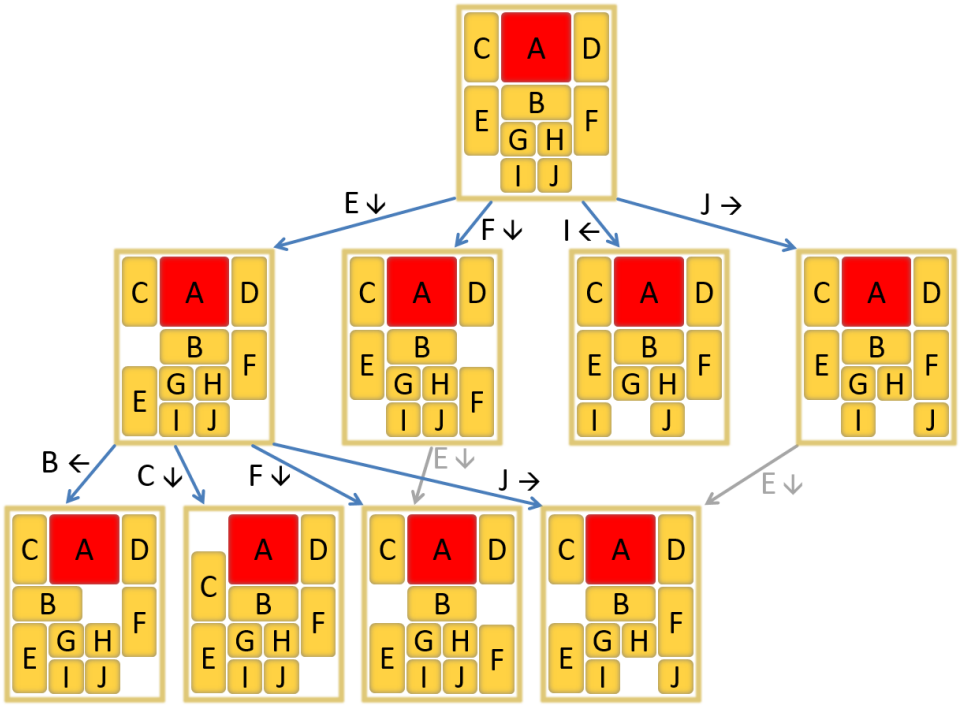


Abbildung 14.3: Nachfolger-Konstellationen im Lösungsbaum

- Die Konstellation muss abgespeichert werden können.
- Der Lösungsbaum mit den Konstellationen muss abgespeichert werden können, so dass die Anzahl der Schiebevorgänge und der Weg von der Start-Konstellation ermittelt werden können.
- Eine Datenstruktur zur Verwaltung der zu bearbeitenden Konstellationen für die Breitensuche muss vorhanden sein.
- Eine Datenstruktur zur schnellen Überprüfung, ob Konstellationen bereits vorhanden sind, muss existieren.

14.6 Gedanken zu den Datenstrukturen

Auf Basis der Anforderungen entwerfen wir jetzt die Datenstrukturen. Wir überlegen also, welche Datenstrukturen geeignet sind, die Anforderungen zu erfüllen und mit dem Algorithmus zusammen zu arbeiten.

Wir wollen uns jedoch nicht auf die Umsetzung in einer Programmiersprache fokussieren. Deshalb beschreiben wir die erforderlichen Datenstrukturen nicht in einem hohen Detaillierungsgrad, aber dennoch für eine Umsetzung hinreichend genau.

14.6.1 Konstellation

Die Konstellation müssen wir als einen Datentyp definieren und mehrfach im Speicher anlegen können. Darin muss abgespeichert werden, an welchen Positionen sich die einzelnen Spielsteine befinden. Dazu wird jeweils eine Variable pro Spielstein und freiem Feld in der Konstellation verwendet. Die Position (also Zeile und Spalte) des Spielsteines wird in einer Zahl abgespeichert, zum Beispiel zeilenweise. Somit wäre das Feld links oben Position 0 und das Feld rechts unten Position 19.

Natürlich wäre es möglich, *einen* Typ von Steinen oder freien Feldern nicht abzuspeichern, da sich diese Positionen ja aus den anderen ergeben. Dies würde Speicherplatz einsparen, jedoch deutlich auf Kosten der Geschwindigkeit gehen, da die nicht gespeicherten Positionen immer wieder neu berechnet werden müssten.

Ein wichtiger Punkt muss noch erwähnt werden. Wenn dies naiv implementiert wird, dann gibt es für den Algorithmus unterschiedlich aussehende Konstellationen, die jedoch identisch sind. Dies machen wir uns anhand des Beispiels der Variablen für die freien Felder klar. Wenn das *erste* freie Feld sich links unten befindet und das *zweite* rechts unten, dann könnte auch das *erste* links und das *zweite* rechts unten sein. Derzeit wären dies zwei unterschiedliche Konstellationen. Dies würde zu vielen Duplikaten führen – das wollen wir nicht. Im Übrigen gilt dies für alle identisch großen Steine mit der gleichen Ausrichtung.

Damit keine Duplikate in den Konstellationen auftreten, werden die Variablen für jede Art des Steins und freie Felder die sortierten Positionen beinhalten. Beispielsweise enthält die *erste* Variable für freie Felder das Feld mit der *kleineren* und die *zweite* das mit der *größeren* Position.

14.6.2 Lösungsbaum

Durch die Breitensuche wird der Lösungsbaum ebenenweise aufgebaut. Auf diesem Weg sind in den Baum aufgenommene Konstellationen immer optimal bezüglich der Anzahl der Schiebevorgänge von der Start-Konstellation. Sie müssen jedoch nicht auf einem optimalen Weg zur Ziel-Konstellation liegen.

Der Lösungsbaum enthält die Konstellationen und einen Verweis auf die jeweilige Vorgänger-Konstellation. Das Prinzip ist in der Abbildung 14.3 ersichtlich. Jedoch werden im Lösungsbaum die Verweise andersherum abgespeichert. Jede Konstellation enthält einen Verweis auf den Vorgänger mit der Information, welcher Schiebevorgang auszuführen ist.

Somit ist es für jede Konstellation (insbesondere die gefundene Lösung) möglich, die kleinste Anzahl von Schiebevorgängen von der Start-Konstellation

und die Reihenfolge und Steine der Schiebevorgänge zu finden.

Somit muss der Lösungsbaum nur aufgebaut werden, bis die erste Lösung gefunden ist. Dann ist dies eine optimale Lösung, gegebenenfalls die einzige. Da wir nur an einer optimalen Lösung interessiert sind, kann der Algorithmus die Suche abbrechen. Wenn wir wissen möchten, wieviele optimale Lösungen es gibt, dann müsste der Algorithmus lediglich die Ebene mit der gefundenen Lösung komplett abarbeiten. In der nächsten Ebene kann es keine optimale Lösung mehr geben, weil ein Schiebevorgang mehr erforderlich wäre.

14.6.3 Abspeicherung der noch unbearbeiteten Konstellationen

Da wir eine Breitensuche durchführen ist dieser Teil des Algorithmus einfach. Bei der Breitensuche suchen wir die Nachfolger-Konstellationen der "ältesten" noch nicht bearbeiteten Konstellation. Diese werden die in die Menge der noch nicht bearbeiteten Konstellationen aufgenommen. Die benötigte Datenstruktur ist offensichtlich eine Warteschlange, siehe Abschnitt 6.5. Die Elemente werden in der Reihenfolge ihres Einfügens wieder entnommen.

Selbstverständlich könnte dies auch in den Lösungsbaum integriert werden. Jedoch ist es für die Verständlichkeit des Algorithmus besser, die Warteschlange als separate Datenstruktur zu verwenden.

14.6.4 Überprüfung auf bereits vorhandene Konstellation

Nun fehlt uns noch eine Datenstruktur, die effizient prüft, ob eine Konstellation bereits vorhanden ist. Dazu speichern wir die Konstellation in einer Hashtabelle (siehe Abschnitt 6.6) ab. Für die dazu benötigte Hashfunktion gibt es viele Möglichkeiten. Es ist möglich, die Positionen der zehn Steine und zwei freien Felder bijektiv wie folgt in eine 64-bit Zahl zu codieren. Ein Stein oder freies Feld erhält die Position, an der sich seine linke obere Ecke befindet. Es gibt 12 Steine und freie Felder und maximal 20 mögliche Positionen pro Stein oder freiem Feld, somit $20^{12} = 4,096 \cdot 10^{15}$ Möglichkeiten. In eine vorzeichenlose 64-bit Zahl können $2^{64} \approx 1,8 \cdot 10^{19}$ Werte codiert werden.

Die Codierung der Konstellation ergibt sich zum Beispiel aus $\sum_{i=0}^{11} p_i \cdot 20^i$, wobei die p_i die Positionen der Steine und freien Felder repräsentieren mit $0 \leq p_i \leq 19$, $0 \leq i \leq 11$.

Für größere Steine gibt es weniger Möglichkeiten, dies haben wir der Einfachheit halber aber nicht ausgenutzt, um Speicher zu sparen. Ebenso könnte durch die oben angesprochene Sortierung der Speicherbedarf noch weiter optimiert werden, was aber auch nicht notwendig ist, da eine 64-bit Zahl pro Konstellation akzeptabel ist.

Dies könnte natürlich auch in die Datenstruktur zur Abspeicherung des Lösungsbaumes integriert werden. Jedoch wird es auch hier wegen der besseren Verständlichkeit des Algorithmus separat umgesetzt.

Selbstverständlich könnte auch hier jedes Mal der Lösungsbaum (ohne Verwendung einer Hashtabelle) durchlaufen werden, dies würde jedoch zu starkem Laufzeitzuwachs (nicht akzeptabel!) führen, jedoch würde das Programm weniger Speicher benötigen.

14.6.5 Ausgabe der Schritte in der richtigen Reihenfolge

Da der Lösungsbaum die Konstellationen mit der jeweiligen Vorgänger-Konstellation enthält, kann der Weg von einer optimalen Ziel-Konstellation bis hin zur Start-Konstellation gefunden werden. Dies ist jedoch die falsche Reihenfolge, genau verkehrt herum.

Wenn die Schritte in der richtigen Reihenfolge ausgegeben werden sollen, gibt es mehrere Möglichkeiten. Beispielsweise kann der Lösungsbaum von der gefundenen optimalen Lösung bis zur Start-Konstellation mittels einer rekursiven Funktion durchlaufen werden. Hinter dem rekursiven Aufruf wird dann der aktuelle Schritt ausgegeben. Auf diesem Weg wird die Ausgabe in der richtigen Reihenfolge – von der Start-Konstellation bis zur Ziel-Konstellation – durchgeführt.

Wir werden im Algorithmus einen anderen Weg beschreiten. Wenn eine Lösung gefunden ist, werden alle Konstellationen von der Lösung bis zur Start-Konstellation in einem Stack (siehe Abschnitt 6.4) gespeichert und am Ende ausgegeben. Der Stack kehrt automatisch die Reihenfolge um.

Jetzt wurden zwei Dinge deutlich. Zunächst der enge Zusammenhang zwischen dem Algorithmus und den verwendeten Datenstrukturen, aber auch der Zusammenhang der Datenstrukturen untereinander. Des Weiteren sind die verschiedenen Möglichkeiten angesprochen worden, wie mit den Design-Zielen umgegangen wird. Dazu haben wir verschiedene Trade-offs angegeben.

Natürlich könnten weitere Optimierungen durchgeführt werden. Beispielsweise könnten bei der Suche der Nachfolger-Konstellationen zunächst die betrachtet werden, die eher zum Ziel führen, zum Beispiel wenn der rote Stein wird nach unten bewegt wird. Oder es werden Teile vom Lösungsbaum nicht mehr beachtet, wenn eine optimale Lösung nicht mehr gefunden werden kann. Diese Optimierungen würden jedoch ein genaueres Problemverständnis voraussetzen, und dagegen hatten wir uns ja entschieden. Außerdem läuft der Algorithmus so schnell, dass keine weitere Optimierung notwendig ist.

14.7 Algorithmus

Nun wollen wir zum Abschluss den Algorithmus im Pseudocode darstellen. Hier muss jetzt entschieden werden, welcher Anteil der Funktionalität wo implementiert wird. Beispielsweise kann das Suchen der möglichen Nachfolger-Konstellationen direkt im Algorithmus umgesetzt werden oder in der Klasse zum Abspeichern der Konstellationen, wenn die Programmiersprache dieses Konzept unterstützt. Da dies auch Abhängigkeiten von den Möglichkeiten der eingesetzten Programmiersprache hat, gehen wir hier nicht im Detail darauf ein. Stattdessen präsentieren wir eine Lösung, aus der der Ablauf des Algorithmus gut erkennbar ist. Komplexe Operationen wie das Suchen von Nachfolger-Konfigurationen sind im Code nicht enthalten.

Der Algorithmus ist im Listing 14.1 dargestellt. Der Fokus liegt ausschließlich auf dem eigentlichen Algorithmus. Daher sind keine Definitionen der Variablen enthalten. Auch die Prüfungen von "Spezialfällen" wie Überprüfung, ob die Start-Konstellation bereits die Ziel-Konstellation ist, wurden weggelassen. Genauso wie Konsistenzchecks (zum Beispiel Invarianten), die beim Entwurf von Algorithmen immer hilfreich sind. Außerdem haben wir die Datenstrukturen bewusst nicht mit den üblichen sprechenden Namen versehen, sondern einfach den Namen der Datenstruktur verwendet.

14.7.1 Güte des Algorithmus

Um Ihnen ein besseres Gefühl für den in Java implementierten Algorithmus zu vermitteln: Die optimale Lösung dieses Schiebespiels benötigt 114 Schiebevorgänge. Dazu wurden knapp 24.000 Konstellationen berechnet, von denen sich knapp 100 noch in der Warteschlange befanden, also zum Zeitpunkt des Findens einer optimalen Lösung noch nicht weiter betrachtet worden sind.

Mit einer kleinen Modifikation (Programm findet nie eine Lösung) berechnet das Programm alle aus der initialen Konstellation erreichbaren Konstellationen, insgesamt 25.955. Wenn man nun annimmt, dass keine optimierte Abspeicherung der Konstellationen durchgeführt worden wäre, also dass die Vertauschung von Steinen gleicher Größe zu neuen Konstellationen führt, kommen wir auf 29.900.160 Konstellationen. Auch wenn diese vielleicht nicht alle erreicht werden können, sieht man hier deutlich, wie wichtig die optimierte Abspeicherung der Konstellation war.

Eine Laufzeitmessung wurde nicht durchgeführt. Die Laufzeit ist jedoch auf einem Intel Core i7 Prozessor (Notebook aus dem Jahr 2015) so schnell, dass unmittelbar nach dem Start des Programmes die Ausgabe der Lösung erfolgt.

Neben den oben beschriebenen Optimierungspotenzialen gibt es viele weitere.

Listing 14.1: Algorithmus zum Finden einer optimalen Lösung für das Schiebespiel

```
1  algoSlidingPuzzle() {
2     // add start-constellation to queue
3     // and to found constellations
4     queue.add(initalConstellation);
5     hashtable.add(initalConstellation);
6
7     // dequeue constellations until queue not empty
8     // and solution not found
9     while (!queue.isEmpty() && !solutionFound) {
10       constellation = queue.remove();
11
12       // find possible successor constellations
13       // for current constellation
14       successorConstellations =
15          constellations.getAllSuccessorConstellations();
16
17       // check all successor constellations
18       while (!successorConstellations.isEmpty()
19              && !solutionFound) {
20          constellation = successorConstellations.remove(0);
21
22          // if new constellation is a solution, we are done
23          if (constellation.isSolution()) {
24             solutionFound = true;
25          }
26          else {
27             // check whether constellation was already found
28             // if not: add constellation to queue
29             // and to found constellations
30             if (!hashtable.contains(constellation)) {
31                queue.add(constellation);
32                hashtable.add(constellation);
33             }
34          }
35       }
36    }
37
38    // output solution in reverse order
39    if (solutionFound) {
40       while (constellation.predecessor != null) {
41          stack.push(constellation);
42          constellation = constellation.predecessor;
43       }
44       stack.push(constellation);
45
46       while (!stack.isEmpty()) {
47          output(stack.pop());
48       }
49    }
50 }
```

Diese müssen aber nicht umgesetzt werden, weil sowohl die Rechenzeit als auch der Speicherplatzbedarf im akzeptablen Bereich liegen. Somit erfüllt der Algorithmus die geforderten Anforderungen, er löst das gestellte Problem in der geforderten Güte.

Die konkrete Implementierung des Programmes ist eine schöne Übungsaufgabe.

Zum Schluss wünschen wir Ihnen viel Erfolg beim Verstehen der Problemstellungen und Entwickeln von guten Lösungen durch Kombinieren geeigneter Algorithmen und Datenstrukturen. Und natürlich viel Spaß beim Herumexperimentieren mit Algorithmen und Datenstrukturen während der Entwicklung und dem Kombinieren verschiedener Ansätze zum Finden optimaler Lösungen.

Index

Beispielcode

Der nachstehend aufgeführte Beispielcode erfordert Python 3.0 aufwärts:

1. activityselection.py
2. binarytree.py
3. binsearch.py
4. bubblesort.py
5. countingsort.py
6. cutrod.py
7. dijkstra.py
8. fib.py
9. growth.py
10. hash.py
11. heapsort.py
12. insertsort.py
13. kmp.py
14. priorityqueue.py
15. lcs.py
16. lcsRecursive.py
17. linkedlist.py
18. markov.py
19. mergesort.py
20. pagerank.py
21. priorityqueue.py

22. quicksort.py
23. recursion.py
24. schleife2.py
25. binsearch.py
26. sudo.py
27. wordcount.py

Der nachstehend aufgeführte Beispielcode erfordert einen gcc Compiler für C[1] und sollte auf allen gängigen Plattformen lauffähig sein:

1. bfs
2. bubblesort
3. dfs
4. hash
5. heapsort
6. insertsort
7. list
8. markov
9. mergesort
10. misc
11. queue
12. quicksort
13. towershanoi
14. tree
15. wordcount

[1] Aufruf: In der Regel `gcc main.c`, ggf. andere Quelldateien linken.

Notationsverzeichnis

Notation	Erklärung
$\mathbb{N}$, $\mathbb{N}_0$	Menge der natürlichen Zahlen ohne / mit Null
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{R}$, $\mathbb{R}^+$	Menge der reellen / positiven reellen Zahlen
$\emptyset$	Leere Menge, Menge ohne Elemente
$\lvert A\rvert$	Kardinalität der Menge A
$A \setminus B$	Menge der Elemente aus A ohne die Elemente in B
$\lfloor x \rfloor$	Größte ganze Zahl $\leq x$
$\lceil x \rceil$	Kleinste ganze Zahl $\geq x$
$\min(i, j)$	Minimum von i und j
$\max(i, j)$	Maximum von i und j
$i \bmod j$	i modulo j, Divisionsrest, $i \bmod j = i - \left\lfloor \frac{i}{j} \right\rfloor \cdot j$
$n!$	Fakultät von $n = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$
$\binom{i}{j}$	Binomialkoeffizient, $\binom{i}{j} = \binom{i}{i-j} = \frac{i!}{j!\,(i-j)!}$
$\lg(z)$	Logarithmus von z zur Basis 2
$\mathcal{O}$	$\mathcal{O}$-Notation, siehe Seite 65
Ω	Ω-Notation, siehe Seite 65
Θ	Θ-Notation, siehe Seite 66
o	o-Notation, siehe Seite 66
ω	ω-Notation, siehe Seite 66
$(a, \ldots, z)$	Tupel
$[a, \ldots, z]$	Liste
$\{a, \ldots, z\}$	Menge
$[a : b]$	Zahlenintervall von a bis b
$A[i]$, A_i	i-ter Wert des Feldes A, i-tes Zeichen des Wortes A
$A[i, j]$, $A_{i,j}$	Wert in Zeile i, Spalte j der Matrix A
$A[i, \ldots, j]$	Teilfeld vom Index i bis Index j des Feldes A
$P(E)$, P_E	Wahrscheinlichkeit für das Eintreffen des Ereignisses E
$E[x]$	Erwartungswert von x
$\sqsubset$, $\sqsupset$	Präfix, Suffix

Siehe auch die speziellen Erläuterungen zum Pseudocode auf Seite 82.

Literaturangaben

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[Bel58] Richard Bellman. Dynamic programming and stochastic control processes. *Information and Control*, 1(3):228–239, 1958.

[Bro75] Fred P. Brooks, Jr. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM.

[CLR04] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Algorithmen - Eine Einführung.* Oldenbourg Wissensch.Vlg, 1 edition, 2004.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Computers*, 38(3):408–423, 1989.

[FS00] Martin Fowler and Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[GS04] K.-U. Sattler G. Saake. *Algorithmen und Datenstrukturen*, volume 2. Auflage. dpunkt, 2004.

[Het08] Magnus Lie Hetland. *Beginning Python.* Apress, Berkely, CA, USA, 2008.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[Knu97a] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 1997.

[Knu97b] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 1997.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3 (3rd ed.): Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 1998.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[Lev05] Anany Levitin. *The Design and Analysis of Algorithms*. Addison Wesley, 2005.

[Neb12] Markus Nebel. *Entwurf und Analyse von Algorithmen*. Springer-Vieweg, 2012.

[NS73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, 1973.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999.

[Ott98] Thomas Ottmann. *Prinzipien des Algorithmenentwurfs*. Spektrum Akademischer Verlag, 1998.

[OW93] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen, 2. Auflage*, volume 70 of *Reihe Informatik*. Bibliographisches Institut, 1993.

[PH72] B. Raphael P.E. Hart, N.J. Nilsson. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter 37*, pages 28–29, 1972.

[Ray01] Eric Raymond. *The Cathedral & the Bazaar*. O'Reilly, 2001.

[Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[ZD04] John M. Zelle and Ph. D. *Python Programming: An Introduction to Computer Science*. Franklin Beedle & Associates, 2004.