4 Algorithmen und Datenstrukturen

Algorithmen sind Verfahren zur schrittweisen Lösung von Problemen. Sie können abstrakt, d.h. unabhängig von konkreten Rechnern oder Programmiersprachen, beschrieben werden. Gute Algorithmen sind oft das Ergebnis wissenschaftlicher Intuition und mathematischer Herleitungen. Die Umsetzung eines Algorithmus in ein lauffähiges Programm ist für einen Programmierer, der seine Programmiersprache beherrscht, eine „handwerkliche“ Tätigkeit.

Algorithmen bilden oft den Kern von Anwendungsprogrammen und sind entscheidend für dessen Güte. Kommerzielle Programme bestehen heute aber aus mehreren Komponenten:

- einer Benutzerschnittstelle (Fenster, Menüs, Maussteuerung etc.),
- einem Verwaltungsteil zum Lesen, Speichern und Aufbereiten von Daten,
- einer Sammlung von Algorithmen zur Berechnung und Manipulation von Daten.

Während die ersten beiden Komponenten meist den größten Teil des Programmcodes beanspruchen, sind sie oft aus Bausteinen zusammengesetzt, die von der Programmierumgebung fertig angeboten werden. Auch in diesen Teilen sind Algorithmen verborgen – etwa zur Darstellung und Manipulation von grafischen Objekten oder zum Zugriff auf Datenbanken – der Programmierer übernimmt diese aber und verlässt sich auf die angebotene Implementierung.

Vom Umfang her ist der zuletzt erwähnte Teil in vielen Fällen der geringste – aber gerade an dieser Stelle entscheidet sich, wie gut oder schlecht ein Programm ist. Die sorgfältige Auswahl der Algorithmen kann die Laufzeit eines Programms häufig drastischer reduzieren als eine raffinierte Programmiertechnik. Die Güte eines Algorithmus hängt dabei vor allem von zwei Faktoren ab,

- der Qualität der Ergebnisse und
- der Laufzeit.

Beide Kriterien muss der Programmierer kompetent beurteilen können. Für einen Algorithmus, der zu gewissen Eingabedaten ein Ergebnis berechnet, muss gefordert werden, dass die Ergebnisdaten immer *korrekt* sind. Dass dies nicht allein durch Testen zu gewährleisten ist, haben wir bereits im Kapitel über Verifikation, S. 187 ff., dargelegt. Insbesondere sollte ein Algorithmus dokumentiert sein. Komplizierte Stellen sollten zum Beispiel in Form von Kommentaren im Programmtext erklärt werden. Zu den kritischen Informationen gehören insbesondere Schleifeninvarianten und Überlegungen zur Terminierung.

Gewisse Probleme lassen sich in der Praxis nur näherungsweise lösen, dazu gehören viele numerische Aufgaben, aber auch Optimierungsprobleme, deren genaue Lösung unvertretbar

lange dauern würde. Hier kann die Abweichung des gelieferten Ergebnisses von dem wahren Wert ein Qualitätskriterium sein. *Heuristische Algorithmen* verwenden „Daumenregeln“ in der Hoffnung, für viele praktisch relevanten Probleme brauchbare Lösungen zu liefern. Hier kommen Gütekriterien ins Spiel, die sich nur in der Praxis bewähren können.

Das zweite Kriterium, die *Laufzeit* eines Algorithmus, muss ebenfalls vom Programmierer gut verstanden werden. Da Programmpakete während der Erstellung oft mit erdachten Beispieldaten getestet werden, kann ein Programm völlig versagen, wenn es zum ersten Mal mit den Datenmengen einer realen Anwendung konfrontiert wird. Aus diesem Grund muss der Programmierer vorab abschätzen können, wie sich eine Vergrößerung der zu bearbeitenden Daten auf die Laufzeit und auf den Speicherbedarf des Programms auswirkt.

Viele Programmierer verwenden unnötig viel Zeit auf sinnlose Optimierungen ihrer Programme, und verpassen dann den entscheidenden Punkt, der mehr als alles andere die Laufzeit bestimmt. Über Programmierkunststücke, wie jüngst gesehen:

```
while ( !(next=find(i++))) ;
```

kann man nur den Kopf schütteln. Wenn der Zeitvorteil dieser Konstruktion gegenüber einer klaren, einfachen Lösung überhaupt messbar sein sollte, er wird nie die Zeit wieder einbringen, die der Programmierer mit dem „Erfinden“ dieser Monstrosität verschwendet hat – ganz zu schweigen von der Verwirrung für jeden, der solchen Code einmal pflegen muss. Sinnvoll wäre es, ein Programm sauber und klar zu schreiben, und zum Schluss mit geeigneten Werkzeugen, etwa einem *profiler,* gezielt zu analysieren, an welchen Stellen sich eine Optimierung überhaupt lohnen könnte.

Algorithmen sind meist von der *Repräsentation* der Daten, also von ihrer Strukturierung, abhängig. Ob eine Kundenliste ungeordnet in einer Datei gespeichert ist, oder ob sie in einem Array im Hauptspeicher gehalten wird, hat entscheidenden Einfluss auf die Auswahl geeigneter Suchalgorithmen. Je nach Anwendung muss der Programmierer entscheiden, ob es sich lohnt, die Liste zu ordnen, bevor mehrere Such- oder Einfügeoperationen vorzunehmen sind. Eventuell zahlt es sich sogar aus, die Liste nach mehreren Kriterien zu sortieren. Aus diesen Gründen kann man *Algorithmen* nur im Zusammenhang mit (passenden) *Datenstrukturen* behandeln. Für viele häufig wiederkehrende Probleme der Programmierpraxis sind gute Algorithmen und Datenstrukturen bekannt. Deren Kenntnis muss zum Repertoire jedes Informatikers gehören.

Glücklicherweise können wir uns bei vielen Problemstellungen an analogen Situationen des täglichen Lebens orientieren. Wie suche ich in einem Telefonbuch geschickt nach einem Teilnehmer? Wie ordne ich einen Stapel von Briefen nach ihrer Postleitzahl? Wie füge ich eine Spielkarte in eine bereits geordnete „Hand“ ein? Oft hilft es, bekannte Strategien in ein Programm zu übernehmen. Darüber hinaus gibt es auch pfiffige Sortierstrategien, die kein Pendant in unserer täglichen Erfahrung besitzen. Ein Beispiel hierfür wird *heapsort* sein.

4.1 Suchalgorithmen

Gegeben sei eine Sammlung von Daten. Wir suchen nach einem oder mehreren Datensätzen mit einer bestimmten Eigenschaft. Dieses Problem stellt sich zum Beispiel, wenn wir im Telefonbuch die Nummer eines Teilnehmers suchen. Zur raschen Suche nutzen wir aus, dass die Einträge geordnet sind, z.B. nach

Name, Vorname, Adresse.

Wenn wir Namen und Vornamen wissen, finden wir den Eintrag von Herrn Müller sehr schnell durch *binäres Suchen*: Dazu schlagen wir das Telefonbuch in der Mitte auf und vergleichen den gesuchten Namen mit einem Namen auf der aufgeschlagenen Seite. Ist dieser zufällig gleich dem gesuchten Namen, so sind wir fertig. Ist er in der alphabetischen Ordnung größer, brauchen wir für den Rest der Suche nur noch die erste Hälfte des Telefonbuches zu berücksichtigen, ansonsten nur die zweite Hälfte.

Abb. 4.1: Suche in einem Telefonbuch

Wir verfahren danach weiter wie vorher, schlagen also bei der Mitte der ersten Hälfte auf und vergleichen wieder den gesuchten mit einem gefundenen Namen und so fort. Dieser Algorithmus heißt *binäre Suche*. Bei einem Telefonbuch mit ca. 1000 Seiten Umfang kommen wir damit nach höchstens 10 Schritten zum Ziel, bei einem Telefonbuch mit 2000 Seiten, nach 11 Schritten. Noch schneller geht es, wenn wir die Anfangsbuchstabenmarkierung auf dem Rand des Telefonbuches ausnutzen. Diese Idee werden wir später unter dem Namen *Skip-Liste* (siehe S. 370), wieder antreffen.

Wenn wir umgekehrt eine Telefonnummer haben und mithilfe des Telefonbuches herausbekommen wollen, welcher Teilnehmer diese Nummer hat, bleibt uns nichts anderes übrig, als der Reihe nach alle Einträge zu durchsuchen. Diese Methode heißt *lineare Suche*. Bei einem Telefonbuch mit 1000 Nummern müssen wir im Schnitt 500 Vergleiche durchführen, im schlimmsten Falle gar 1000. Müssen wir diese Art von Suche öfters durchführen, so empfiehlt sich eine zusätzliche Sortierung (einer Kopie) des Telefonbuches nach der Rufnummer, so dass wir wieder binär suchen können.

4.1.1 Lineare Suche

Allgemein lässt sich das *Suchproblem* wie folgt formulieren:

Suchproblem: *In einem Behälter A befinden sich eine Reihe von Elementen. Prüfe, ob ein Element $e \in A$ existiert, das eine bestimmte Eigenschaft $P(e)$ erfüllt.*

„Behälter“ steht hier allgemein für Strukturen wie: *Arrays, Dateien, Mengen, Listen, Bäume, Graphen, Stacks, Queues, etc.* Wenn nichts Näheres über die Struktur des Behälters oder die Platzierung der Elemente bekannt ist, dann müssen wir folgenden Algorithmus anwenden:

> *Entferne der Reihe nach Elemente aus dem Behälter, bis dieser leer ist oder ein Element mit der gesuchten Eigenschaft gefunden wurde.*

Wir können diesen Algorithmus bereits programmähnlich formulieren, wenn wir folgende Grundoperationen als gegeben annehmen:

- Prüfen, ob der Behälter leer ist: *istLeer*,
- Ergreifen und Entfernen eines Elementes aus dem Behälter: *nimmEines*.

Wir nehmen an, dass diese Operationen in einer geeigneten Klasse *Behälter* definiert sind, die auch die Elemente verwaltet. Damit ergibt sich der folgende in Java formulierte Algorithmus für die Lineare Suche. Er gibt entweder ein Element mit der gesuchten Eigenschaft zurück oder die Null-Referenz, falls nichts zu finden war.

```
class Behälter<Element> {
    boolean istLeer(){ ..... }
    Element nimmEines(){ ..... }
}
...
boolean P(Element e){ ... }

Element linSuche(Behälter<Element> beh){
    while(! beh.istLeer()){
        Element e = beh.nimmEines();
        if (P(e)) return e;
    }
    return null;
}
```

In diesem Programmfragment entnehmen wir dem Behälter der Reihe nach alle Elemente solange bis wir ein Element mit der gesuchten Eigenschaft gefunden haben. In der Praxis würde man die entnommenen Elemente natürlich nicht wegwerfen, sondern z.B. als entnommen markieren, in einen Hilfsbehälter bewegen, oder dergleichen.

Behälter Datentypen in Java entsprechen den Klassen des *Collection Framework*. Für diese und zusätzlich für alle Arrays ist die verkürzte *for*-Schleife anwendbar (siehe S. 263, ff.). Solche Behälter können daher mit dem dem folgenden Schema durchsucht werden:

```
boolean P(Element e){ ..... }

Element linSuche(Collection<E> beh){
    for(Element e : beh)
        if (P(e)) return e;
    return null; }
```

Typischerweise will man aber nicht das Element selbst, sondern seine Position in dem Behälter. In manchen objektorientierten Sprachen (z.B. Smalltalk) verfeinert man die Collection Hierarchie noch zu der Unterhierarchie „Indexed Collection“. In Java muss man auf spezielle Klassen wie *ArrayList* oder auf Arrays zurückgreifen. Leider hilft uns hier die verkürzte *for*-Schleife nicht mehr, da sie nur die Elemente, nicht aber deren Indizes liefert.

Wenn das Element nicht gefunden wird können wir einen nicht existenten Index zurückgeben, für Java-Arrays bietet sich -1 an, oder wir erzeugen eine Ausnahme (siehe S. 270 ff):

```
int linSuche(Element[] a)throws NichtGefunden{
  for (int i=0; i < a.length; i++)
    if (P(a[i])) return i;
  throw new NichtGefunden();
}
```

Schlimmstenfalls müssen wir den ganzen Behälter durchsuchen, bis wir das gewünschte Element finden. Hat dieser *N* Elemente, so werden wir bei einer zufälligen Verteilung der Daten erwarten, nach ca. *N/2* Versuchen das gesuchte Element gefunden zu haben. In jedem Fall ist die Anzahl der Zugriffe proportional zur Anzahl der verschiedenen Elemente.

Arrays als Behälter werden in den folgenden Such- und Sortier-Algorithmen eine besondere Rolle spielen. In Java ist die Indexmenge eines Arrays *a* mit *n* = *a.length* Elementen stets das Intervall [0 ... *n*–1] = [0 ... *a.length*–1]. Besonders für die rekursiven Algorithmen wird es sich als günstig herausstellen, wenn wir sie so verallgemeinern, dass sie nicht nur ein ganzes Array, sondern auch einen *Abschnitt* (engl. *slice*) eines Arrays sortieren können. Unter dem Abschnitt *a*[*lo* ... *hi*] verstehen wir dabei den Teil des Arrays *a*, dessen Indizes aus dem Teilintervall [*lo* ... *hi*] sind, also *a*[*lo*], *a*[*lo*+1], ... , *a*[*hi*]. Wir setzen dabei $0 \le lo \le hi \le a.length-1$ voraus. Ein Aufruf, z.B. einer Suchroutine, erhält dann als Parameter neben dem Array *a* auch die Abschnittsgrenzen, wie z.B. in

```
binSearch(a, lo, hi).
```

Das komplette Array wird mit dem Aufruf `binSearch(a,0,a.length-1)` durchsucht.

4.1.2 Exkurs: Runden, Logarithmen und Stellenzahl

Ist *r* eine reelle Zahl, so bezeichnet man mit $\lfloor r \rfloor$ die größte ganze Zahl, die kleiner oder gleich *r* ist, und mit $\lceil r \rceil$ die kleinste ganze Zahl größer oder gleich *r*, also z.B. $\lfloor 3.14 \rfloor = 3$ und $\lceil 3.14 \rceil = 4$. Meistens ist aufrunden das gleiche wie *abrunden*+1, also $\lceil r \rceil = \lfloor r \rfloor + 1$, ausser wenn *r* ganzzahlig ist, denn dann ist $\lceil r \rceil = r = \lfloor r \rfloor$.

Für eine Zahl *n* und eine Basiszahl *d* ist ($log_d\, n$), der *Logarithmus* von *n zur Basis d*. Das ist diejenige reelle Zahl *r* mit $d^r = n$. Benötigt *n* bei Darstellung zur Basis *d* genau *k* Stellen, also $n = (z_{k-1} \ldots z_0)_d$ mit $z_{k-1} \neq 0$, so folgt $d^{k-1} \le n < d^k$, also $(k\text{-}1) \le log_d\, n < k$. Daraus folgt $\lfloor log_d\, n \rfloor + 1 = k$, so dass gilt:

$\lfloor log_d\, n \rfloor + 1$ ist die Anzahl der Stellen von *n* bei der Darstellung zur Basis *d*.

Außer wenn $log_d\, n$ ganzzahlig ist, also außer für $n = d^k$ kann man die Formel vereinfachen:

$\lceil log_d\, n \rceil$ ist die Anzahl der Stellen von n bei der Darstellung zur Basis d.

Das ist unabhängig von der Repräsentation, gilt also für Dezimalzahlen genauso wie für Binärzahlen, z.B. haben wir $log_{10}\, 5678 = 3{,}754$ und $log_2\, (110111)_2 = log_2\, 54 = (log_{10}\, 54) / (log_{10}\, 2) = 1{,}732/0{,}301 = 5{,}754$. Durch Aufrunden erhalten wir (fast) immer die exakte Anzahl der Stellen, also z.B.: $\lceil log_{10}\, 5678 \rceil = 4$, $\lceil log_2\, (110111)_2 \rceil = 6$. Ausnahmen sind nur die exakten Potenzen: $\lceil log_{10}\, 999 \rceil = 3$, aber $\lceil log_{10}\, 1000 \rceil = 3$. Danach stimmt es wieder: $\lceil log_{10}\, 1001 \rceil = 4$, etc.

Der Logarithmus als Anzahl der Ziffern gibt uns eine gute Intuition für diese bei Schülern nicht gerade beliebte Funktion. Teilen wir eine Binärzahl durch 2, so verliert sie eine Stelle. Daher können wir eine beliebige Zahl n höchstens $\lceil log_2\, n \rceil$ oft halbieren. Diese Überlegung ist auch wichtig für die folgende Diskussion der folgenden *binären Suche*.

4.1.3 Binäre Suche

Wenn auf dem Element-Datentyp eine Ordnung definiert ist und die Elemente entsprechend ihrer Ordnung in einem Array gespeichert sind, dann nennt man das Array *geordnet* oder *sortiert*. Genauer sei a ein Array, das n Elemente enthält, und „≤“ eine Ordnung, die auf dem Datentyp *Element* definiert ist. a heißt dann *geordnet* (bzw. *sortiert*), wenn gilt:

$$\forall i\colon 0 \le i < n-1\ .\ a[i] \le a[i+1]$$

Für die Suche in solchen sortierten Arrays können wir die binäre Suche einsetzen, wie wir sie vom Telefonbuch her kennen. Dazu sei x das gesuchte Element. Wir fragen also nach einem Index i mit $a[i] = x$.

Wie bei der Namenssuche im Telefonbuch wollen wir den Bereich, in dem sich das gesuchte Element noch befinden kann, in jedem Schritt auf die Hälfte verkleinern. Dazu verallgemeinern wir das Problem dahingehend, dass wir i in einem beliebigen Indexbereich $[min...max]$ des Arrays a suchen, angefangen mit $min = 0$ und $max = n-1$. Neben $0 \le min, max < n$ soll stets die folgende Invariante gelten:

$$\exists i.\ (0 \le i \le n-1 \wedge a[i] = x) \Rightarrow \exists i.\ (min \le i \le max \wedge a[i] = x)$$

Mit anderen Worten: Wenn das gesuchte Element x überhaupt in dem Array vorhanden ist, dann muss es (auch) im Abschnitt $a[min...max]$ zu finden sein.

Der Algorithmus funktioniert folgendermaßen: Wenn $min > max$, breche ab, x ist nicht vorhanden, ansonsten wähle irgendeinen Index m mit $min \le m \le max$:

- Falls $x = a[m]$ gilt, sind wir fertig; wir geben m als Ergebnis zurück;
- falls $x < a[m]$, suche weiter im Bereich $min...m-1$, setze also $max = m - 1$;
- falls $x > a[m]$, suche weiter im Bereich $m+1 \ldots max$, setze also $min = m + 1$.

Für den Index m zwischen min und max nimmt man am besten einen Wert nahe der Mitte, also z.B. $m = (min + max)/2$. Auf diese Weise halbiert sich in jedem Schritt der noch zu betrachtende Bereich, und damit der Aufwand für die Lösung des Problems.