Kapitel 1

Einführung

1.1 Berechnungsprobleme und Algorithmen

Definition 1.1. Ein *Berechnungsproblem* ist eine Relation der Form $P \subseteq X \times Y$ so dass für alle $x \in X$ ein $y \in Y$ existiert mit $(x, y) \in P$.

Dabei stellen wir uns X als Menge der möglichen Eingaben vor, Y als Menge der möglichen Ausgaben und $(x, y) \in P$ als die Aussage "bei Eingabe x ist y eine korrekte Ausgabe". Oft werden wir statt Berechnungsproblem einfach nur Problem sagen. Ein Berechnungsproblem ist aber zu unterscheiden von einem mathematischen Problem, bei welchem es sich (üblicherweise) um eine Frage der Form "Ist die Aussage ... wahr?" handelt. Beispiele für Berechnungsprobleme sind:

Bestimmung des ggT
Eingabe: positive ganze Zahlen n_1 und n_2
Ausgabe: der größte gemeinsame Teiler von n_1 und n_2

Sortierproblem
Eingabe: eine endliche Folge ganzer Zahlen $(a_1, \ldots, a_n)$
Ausgabe: eine Permutation $(a_{\pi(1)}, \ldots, a_{\pi(n)})$ so dass $a_{\pi(1)} \leq \cdots \leq a_{\pi(n)}$

Linearisierung
Eingabe: eine endliche partiell geordnete Menge $(A, \leq)$
Ausgabe: eine totale Ordnung $a_1, \ldots, a_n$ der Elemente von A so dass $a_i \leq a_j \Rightarrow i \leq j$

Wie man am dritten der obigen Beispiele sehen kann, muss ein Berechnungsproblem nicht unbedingt eine eindeutige Lösung haben. Falls $P \subseteq X \times Y$ ein Problem ist, dann heißt jedes $x \in X$ *Instanz von* P, beispielsweise ist $(3, 7, 2, 5, 8)$ eine Instanz des Sortierproblems.

Definition 1.2. Ein *Algorithmus* ist eine wohldefinierte Rechenvorschrift.

Wir geben hier keine präzisere Definition des Begriffs Algorithmus. Dies zu tun würde im Wesentlichen auf die Definition einer Programmiersprache hinauslaufen und damit am Thema dieser Vorlesung vorbeigehen. Wir werden konkrete Algorithmen in *Pseudocode* angeben, d.h. in einer der Situation angepassten Mischung aus natürlicher Sprache und üblichen Anweisungen und Kontrollstrukturen einer Programmiersprache wie Schleifen, Verzweigungen, usw. Wir verlangen auch nicht formell dass jeder Algorithmus *terminiert*, d.h. dass er für jede Eingabe nach endlicher Zeit stoppt. Allerdings werden wir in dieser Vorlesung fast ausschließlich terminierende Algorithmen betrachten.

Wir kennen bereits viele Algorithmen, zum Beispiel Algorithmen zur Addition und Multiplikation zweier natürlicher Zahlen in Dezimaldarstellung, wie sie in der Volksschule gelehrt werden, den Algorithmus zur Division mit Rest von Polynomen, den euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen, das gaußsche Eliminationsverfahren zur Lösung linearer Gleichungssysteme, usw. Algorithmen sind aus der Mathematik nicht wegzudenken. Anders als bisher werden wir in dieser Vorlesung ein systematisches Studium von Algorithmen betreiben. Dieses beschränkt sich nicht auf die bloße Definition und Verwendung von Algorithmen, sondern untersucht Fragen wie z.B. "Wie effizient ist dieser Algorithmus?" "Wie ist das Verhältnis zwischen zwei Algorithmen?" "Welche Ansätze zur Entwicklung guter Algorithmen gibt es?" "Ist dieser Algorithmus der beste zur Lösung dieses Problems? In welchem Sinn ist er das?" usw. Algorithmen werden also von einem Mittel zur Lösung von Problemen zu einem Objekt unserer Untersuchungen.

Oft operieren Algorithmen auf umfangreichen Daten. Ein wichtiger Aspekt ist dann die Frage in welcher Form bzw. Struktur die Daten gespeichert werden. Man spricht in diesem Zusammenhang auch von *Datenstrukturen*. In dieser Vorlesung werden wir eine Reihe nützlicher und effizienter Datenstrukturen kennenlernen. Eine der einfachsten und gleichzeitig wichtigsten Datenstrukturen ist das *Datenfeld* (engl. *array*). Mathematisch handelt es sich dabei um eine endliche Folge. Die Elemente der Folge werden in aufsteigender Reihenfolge im Speicher abgelegt. Wir schreiben $A, B, \ldots$ für Datenfelder, $A[i]$ für das i-te Element, der kleinste Index eines Datenfelds ist 1, wir schreiben $A.\mathit{Länge}$ für die Länge des Datenfelds (d.h. für die Anzahl von Elementen, die es enthält). Die Notation $A.\mathit{Länge}$ wird in der Informatik verwendet, um das Attribut *Länge* des Objekts A zu notieren. Manche Attribute (wie dieses) können nur gelesen werden (engl. *read-only*), manche können auch geschrieben werden, d.h. in Zuweisungen verwendet werden. Was davon der Fall ist wird üblicherweise aus dem Kontext heraus klar sein.

Algorithmen werden verwendet um Berechnungsprobleme zu lösen. Damit meint man Folgendes:

Definition 1.3. Sei $P \subseteq X \times Y$ ein Berechnungsproblem und $\mathcal{A}$ ein Algorithmus. Wir sagen dass $\mathcal{A}$ *das Problem* P löst falls für jede Instanz $x \in X$ gilt dass $y = \mathcal{A}(x)$ die Eigenschaft $(x, y) \in P$ hat.

Oft wird aus dem Kontext heraus klar sein, welches Berechnungsproblem wir lösen wollen, dann sprechen wir einfach von der *Korrektheit* eines Algorithmus.

1.2 Korrektheitsbeweise

Ein *Korrektheitsbeweis* ist ein Beweis der zeigt, dass ein bestimmter Algorithmus ein bestimmes Berechnungsproblem löst. Um diesen Begriff zu illustrieren wollen wir zunächst ein einfaches Beispiel betrachten. Der folgende Algorithmus erhält ein Datenfeld (dessen Einträge Zahlen sind) als Eingabe und liefert eine Zahl als Ausgabe.

Algorithmus 1 Prozedur P

```
1: Prozedur P(A)
2:     m := 0
3:     Für i = 1, ..., A.Länge
4:         m := m + A[i]
5:     Ende Für
6:     Antworte m/A.Länge
7: Ende Prozedur
```

Mit etwas Programmiererfahrung lässt sich leicht erkennen, dass Algorithmus 1 das folgende Berechnungsproblem löst:

Arithmetisches Mittel
Eingabe: eine endliche Folge $a_1, \ldots, a_n$
Ausgabe: arithmetischer Mittelwert von $a_1, \ldots, a_n$

Wir wollen diese Aussage nun präzise formulieren und beweisen. Ein wesentlicher Aspekt von Korrekheitsaussagen und Korrektheitsbeweisen besteht darin, dass wir es mit zwei unterschiedlichen Sprachebenen zu tun haben: einerseits die Sprachebene des Programms oder Pseudocodes und die in ihm vorkommenden *Programmvariablen*, andererseits die übliche mathematische Sprache in der wir Aussagen *über* den Pseudocode formulieren. Dieser Algorithmus erhält als Eingabe ein Datenfeld A, er benutzt die lokale Variable m und den Schleifenzähler i, der im Körper der Schleife ebenfalls eine lokale Variable ist. Die Programmvariablen sind also A, m und i. Für den Rest der Diskussion dieses Beispiels werden wir für Programmvariablen `Schreibmaschinenschrift` verwenden, d.h. $\mathtt{A}, \mathtt{m}, \mathtt{i}$. Wir können also formulieren:

Satz 1.1. *Algorithmus 1 ist korrrekt, d.h. er löst das Berechnungsproblem "Arithmetisches Mittel", d.h. P(*$\mathtt{A}$*) antwortet mit* $\frac{\sum_{j=1}^{\mathtt{A}.Länge} \mathtt{A}[j]}{\mathtt{A}.Länge}$.

Man beachte dass in obiger Formel die Variable j *keine* Programmvariable ist sondern eine Variable der üblichen mathematischen Sprache. Um diese Aussage formal zu beweisen benutzen wir eine *Schleifeninvariante*. Eine Invariante für eine bestimmte Schleife ist eine logische Aussage I über jene Programmvariablen, die zu Beginn eines Durchlaufs der Schleife verfügbar sind, mit den folgenden Eigenschaften:

1. I ist zu Beginn des ersten Durchlaufs der Schleife wahr und
2. I wird von der Schleife erhalten, d.h. wenn sie zu Beginn des i-ten Durchlaufs wahr ist, dann ist sie auch zu Beginn des $i+1$-ten Durchlaufs wahr.

Damit ist eine Schleifeninvariante auch nach dem letzten Durchlauf der Schleife wahr. Sinnvollerweise wählt man die Invariante so, dass sie am Ende der Schleife eine Form hat, die für den weiteren Korrektheitsbeweis nützlich ist. Deshalb enthält ein Korrektheitsbeweis im Kontext einer Schleifeninvariante noch den weiteren Teil der

3. Verwendung der Invariante im Korrektheitsbeweis.

Auf diese Weise werden in einem Programmablauf mit n Durchläufen einer bestimmte Schleife zusätzlich zum Beginn und zum Ende des Programms noch $n+1$ weitere Zeitpunkte definiert:

1. der Beginn des 1. Schleifendurchlaufs, ..., n. der Beginn des n-ten Schleifendurchlaufs, $n+1$. der Beginn des $n+1$-ten Schleifendurchlaufs der die Schleife beendet. Es ist oft nützlich den Wert einer Programmvariablen $\mathtt{x}$ zum i-ten dieser Zeitpunkte mit $\mathtt{x}_i$ zu bezeichnen. Für Algorithmus 1 erhalten wir so die Werte $\mathtt{m}_k$, $\mathtt{A}_k$ und $\mathtt{i}_k$ für $k = 1, \ldots, n+1$ wobei $n = \mathtt{A}.\mathit{Länge}$. Nun sieht man leicht dass $\mathtt{A}_1 = \cdots = \mathtt{A}_{n+1}$ so dass wir den Index von $\mathtt{A}$ einfach weglassen. Außerdem gilt $\mathtt{i}_k = k$ für $k = 1, \ldots, n+1$.

Beweis von Satz 1.1. Unsere Schleifeninvariante ist:

$$I(\mathtt{A}, \mathtt{m}, \mathtt{i}) \colon \mathtt{m} = \sum_{j=1}^{\mathtt{i}-1} \mathtt{A}[j].$$

Zunächst weisen wir nach, dass diese zu Beginn des ersten Durchlaufs der Schleife wahr ist. Dann ist $k = 1$ und damit erhalten wir:

1. $\mathtt{m}_1 = 0 = \sum_{j=1}^{0} \mathtt{A}[j]$.

Im nächsten Schritt weisen wir nach, dass die Invariante durch die Schleife erhalten wird. Dazu nehmen wir (ähnlich einer Induktionshypothese) an dass die Invariante zu Beginn des i-ten Durchlaufs wahr ist, d.h. $\mathtt{m}_k = \sum_{j=1}^{\mathtt{i}_k - 1} \mathtt{A}[j]$, und erhalten:

2. $\mathtt{m}_{k+1} = \mathtt{m}_k + \mathtt{A}[\mathtt{i}_k] = \sum_{j=1}^{\mathtt{i}_k} \mathtt{A}[j] = \sum_{j=1}^{\mathtt{i}_{k+1} - 1} \mathtt{A}[j]$.

Um den Beweis abzuschließen betrachten wir den Beginn des $n+1$-ten Schleifendurchlaufs. Bei diesem wird nach Inkrementierung von $\mathtt{i}$ festgestellt, dass die Schleife zu beenden ist und damit wird die Prozedur mit Zeile 6 fortgesetzt. Als Antwort der Prozedur erhalten wir also

3. $\mathtt{m}_{n+1}/n = \frac{\sum_{j=1}^{n} \mathtt{A}[j]}{n}$

was die Behauptung zeigt. □

Wie man an obigem Beweis exemplarisch sehen kann handelt es sich bei der Verwendung von Schleifeninvarianten um eine struktierte Vorgehensweise zur Führung von Induktionsbeweisen über imperative Programme. Bei dieser Vorgehensweise verwendet man für jede in einem Algorithmus vorkommende Schleife eine Invariante. Wie sich diese Invarianten zueinander verhalten hängt vom Verhältnis der Schleifen zueinander ab, so entsprechen etwa zwei verschachtelte Schleifen einem verschalteten Induktionsbeweis mittels Schleifeninvarianten. Es ist möglich, in einem präzisen logischen Sinn zu zeigen, dass Beweise mittels Schleifeninvarianten ausreichend sind um Eigenschaften von imperativen Programmen nachzuweisen, d.h. dass jede wahre Aussage über ein Programm durch einen Beweis mit Schleifeninvarianten gezeigt werden kann.

Eines der am häufigsten auftretenden Berechnungsprobleme ist das Sortieren eines Datenfelds. Folglich sind Sortieralgorithmen sehr gründlich untersucht worden. Als nächstes Beispiel wollen wir hier einen ersten Sortieralgorithmus betrachten: Einfügesortieren (engl. *insertion sort*). Die Grundidee besteht darin, ein Datenfeld zu unterteilen in einen bereits sortierten Bereich (links) und einen noch nicht sortierten Bereich (rechts). Der sortierte Bereich wird dann sukzessive vergrößert, indem das erste Element des unsortierten Bereichs in den sortierten Bereich (an der richtigen Stelle) eingefügt wird. Am Ende ist der unsortierte Bereich leer und das Datenfeld damit vollständig sortiert. Die Idee kann wie folgt als Pseudocode notiert werden:

Algorithmus 2 Einfügesortieren

```
1: Prozedur EINFÜGESORTIEREN(A)
2:     Für j := 2, ..., A.Länge
3:         x := A[j]
4:         i := j − 1
5:         Solange i ≥ 1 und A[i] > x
6:             A[i + 1] := A[i]
7:             i := i − 1
8:         Ende Solange
9:         A[i + 1] := x
10:    Ende Für
11: Ende Prozedur
```

Beispiel 1.1. siehe `bsp.einfuegesortieren.pdf`.