Merkblatt Algorithmen

Prof. Dr. Udo Hahn, Erik Fäßler, Johannes Hellrich, Franz Matthies, Sven Büchel
28. März 2017

Dieses Dokument beschäftigt sich mit den grundlegenden Elementen der Algorithmisierung bzw. Programmierung. Die hier vorliegenden Beispiele sollen dem Verständnis dienen und das Lösen der Aufgaben erleichtern. Alle Punkte wurden in der Übung bereits besprochen und werden hier lediglich zum Nachsehen wiedergegeben.

Inhaltsverzeichnis

---

1 Einstieg

Algorithmen sind Abfolgen von gültigen Anweisungen, häufig zum Lösung eines bestimmten Problems. Im Bereich der Sprachverarbeitung sind solche Probleme etwa das automatische Berechnen von Wortarten oder eines Syntax-Baumes aus einem Eingabesatz. Algorithmen sind in aller Regel dafür vorgesehen, von Computern ausgeführt zu werden.

Ein Computer ist jedoch keine Vorraussetzung um sich mit Algorithmen auseinanderzusetzten. Die Handlungsanweisen, aus denen sie sich zusammensetzen können (im Prinzip) auch von Menschen ausgeführt werden, wie wir im Vergleich der Übung feststellen werden. Beispiele für solche zur Ausführung von Menschen gedachten Algorithmen findet man viele im Mathematikunterricht in der Schule (etwa schriftliches Dividieren, Dreisatz, Ableiten von Funktionen, usw.).

Damit die Anweisungen eines Algorithmus von einem Computer ausgeführt werden können, müssen diese in einer konkreten *Programmiersprache* (z.B. Java, Python oder C) verfasst sein. Der Algorithmus ist gewissermaßen die Abstraktion über so ein konkretes Stück Programmcode, sodass derselbe Algorithmus in verschiedenen Programmiersprachen umgesetzt (*implementiert*) werden kann und dabei jedes Mal leicht unterschiedlich aussehen wird.

Um den Rahmen der Veranstaltung nicht völlig zu sprengen beschäftigen wir uns hier nicht mit konkreten Programmiersprachen, sondern nur den mit abstrakten Algorithmen. Da wir diese trotzdem irgendwie aufschreiben müssen, ist es üblich, sogenannten *Pseudocode* zu verwenden. Dabei handelt es sich um eine vereinfachte Quasi-Programmiersprache,

sodass wie uns nicht mit technischen Details einer echten Programmiersprache beschäftigen müssen.[1] Die Algorithmen, die wir so schreiben, lassen sich daher unkompliziert in verschiedenen Programmiersprachen implementieren. Nachfolgend werden die Spezifikationen unseres Pseudocodes sowie Grundkonzepte der Algorithmik vorgestellt.

Den ersten Befehl, den wir kennnenlernen, ist der `print`-Befehl. Dieser ermöglicht es uns, Ergebnisse von (zum Beispiel) Berechnungsanweisungen *ausgeben* zu lassen. Das bedeutet vereinfacht gesagt, dass diese auf unserem Bildschirm erscheinen, wenn wir den Algorithmus auf einem Computer ausführen würden:

```
print 1+1 //Hier wuerde "2" auf dem Bildschirm erscheinen.
```

Bei den beiden Strichen `//` handelt es sich um Kommentar-Markierungen. Das heißt, alles was in der Zeile rechts von diesen Strichen steht gehört nicht zum eigentlichen Algorithmus, sondern dient nur zu dessen Erläuterung. Ein Computer würde diesen Teil einfach ignorieren.

Pseudocode, genauso wie echter Programmcode, ist in Zeilen gegliedert. Dabei können mehrere Anweisungen in einer Zeile stehen (so wird im obigen Beispiel erst die Anweisung gegeben, 1 und 1 zu addieren, und dann erst die Anweisung, dass Ergebnis dieser Berechnung auszugeben). Man sollte jedoch versuchen, nicht zuviele Anweisungen in eine Zeile zu setzen, sondern den Code durch die Zeilen in sinnvolle Einheiten zu gliedern. Andernfalls wird Code schnell unleserlich.

Ein Algorithmus wird von oben nach unten, Zeile für Zeile abgearbeitet. Mehrere Ausgaben erscheinen dabei nacheinander (meist jeweils in einer eigenen Zeile) auf dem Bildschirm.

```
print 1+1 //Ausgabe 2
print 7-4 //Ausgabe 3
print "Hello, World!" //Ausgabe "Hello, World!"
```

2 Variablen

Den Begriff der *Variablen* (*„Veränderbare“*) kennt man aus dem Mathematikunterricht. Eine Funktion $f(x) = x^2 + 5$ enthält die Variable x. Diese Variable kann alle Werte zwischen $-\infty$ und $+\infty$ annehmen, die Funktion gibt für jedes unterschliedliche x einen Funktionswert aus. Wichtig ist, dass x unterschiedliche Werte annehmen kann, also nicht unbedingt nur einem Wert entspricht. Man weist in der Mathematik auch Werte zu. Dazu werden meist Phrasen wie „Sei x gleich 7“ verwendet. Ab diesem Zeitpunkt enthält die Variable x den Wert 7, bis ihr ein anderer Wert zugewiesen wird.

Dieses Konzept einer Variablen lässt sich direkt auf Variablen in Programmiersprachen übertragen. „Sei x gleich 7“ würde beispielsweise mit `x = 7` ausgedrückt. Das Zeichnen der Funktion $f(x) = x^2 + 5$ könnte so aussehen:

```
for (x in [−∞,... +∞])
      zeichne den Punkt (x, x^2 + 5) in das Koordinatensystem
```

Hier durchläuft x alle Werte von negativ Unendlich bis positiv Unendlich. Für jeden Wert wird der Funktionswert in das Koordinatensystem eingetragen (Natürlich würde das obenstehende Programm in Wahrheit niemals fertig werden!).

In Programmiersprachen werden Variablen häufig schlicht als „Behälter“ verwendet, um sich Zahlen (oder Zeichenketten und weiteres) zu „merken“.

Beispiele:

Einfache Zuweisungen: Die Zahl rechts neben dem Pfeil wird der Variable zugewiesen

```
x <- 5
y <- 7
z <- 8
```

Einfache Rechenschritte (mit den Variablenwerten von oben):

```
x <- 3 * z          // x beinhaltet nun den Wert 3 * z = 3 * 8 = 24
x <- x / 2          // x beinhaltet seinen eigenen Wert durch zwei geteilt
z <- x + y          // z = x + y = 12 + 7 = 19
```

[1] Unser Pseudocode hat starke Ähnlichkeit mit der Programmiersprache Python.

3 Datentypen und Operationen

Im Mathematikunterricht nehmen Variablen meistens nur Zahlenwerte an (z.B. ganze Zahlen, rationale Zahlen, relle Zahlen, usw.). Für Algorithmen zur Verarbeitung von natürlicher Sprache braucht man jedoch noch weitere "Arten von Werten", zum Beispiel Zeichenketten, um eine sprachliche Äußerungen festhalten zu können. Solche unterschiedlichen *Datentypen* wie etwa Zahlen und Zeichenketten zeichnen sich zum einen durch einen unterschiedlichen Wertebereich aus (z.B. gibt es keine Zahl "Haus", obwohl dies eine tadellose Zeichenkette ist). Zum anderen unterscheiden sie sich durch die für sie definierten *Operationen*. Z.B. lassen sich Zahlen wie gewohnt miteinander multiplizieren, für zwei Zeichenketten ist dies jedoch offensichtlich nicht ohne weiteres möglich ("Haus"$\times$ "Baum"$=$?). Der Einfachheit halber unterscheiden wir hier (nur) die folgenden Datentypen:

- Zahlen
- Wahrheitswerte (Booleans)
- Zeichenketten (Strings)
- Listen

3.1 Booleans

Werte des Datentyp Wahrheitswert (oder Boolean) sind entweder wahr (`TRUE`) oder falsch (`FALSE`). Auf Booleans kann man die aussagenlogischen Operationen "und", "oder" und "nicht" (`AND, OR, NOT`) anwenden:

```
var1 <- TRUE
var2 <- FALSE
print var1 //Ausgabe TRUE
print var1 AND var2 //Ausgabe FALSE
print var1 OR var 2 //Ausgabe TRUE
print NOT var1 //Ausgabe FALSE

//Verkettungen mehrer Operationen sollten
//zur Uebersichtlichkeit geklammert werden
print NOT(var1 OR var2) //Ausgabe FALSE
```

3.2 Zahlen

Für die Übung fassen wir alle numerische Ausdrücke wie ganze Zahlen (1,2,3...), negative Zahlen (-4) sowie Gleitkommazahlen (17,4123) unter dem Datentyp *Zahl* zusammen. Die erlaubten Operationen sind zum einen die Grundrechenarten (+, –, *, /) sowie die Vergleichsoperationen =, $\neq$ $<$, $\leq$, $>$, $\geq$. Das Ergebnis bei der Anwendung der Grundrechenarten ist wiederum vom Typ Zahl, bei den Vergleichsoperatoren ist das Ergebnis vom Typ Boolean. Beispiel:

```
foo <- 7
bar <- 3
foobar <- foo + bar
print foobar //Ausgabe 10
print foo < foobar //Ausgabe TRUE
```

3.3 Strings

Wir arbeiten oft mit Zeichenketten/Strings, also Folgen von Buchstaben oder anderen Zeichen. Diese werden durch Anführungszeichen kenntlich gemacht. Dabei sind Strings die eine Zahl enthalten von der Zahl selbst zu unterscheiden, im folgenden Beispiel etwa ist der Wert von y ungleich dem von z.

```
x <- "Baum"
y <- "8"
z <- 8
```

Für Strings gilt, dass wir sie durch + **konkatinieren** (verbinden) können, im folgenden nimmt z den Wert "12" an.

```
x <- "1"
y <- "2"
z <- x + y
```

Zusätzlich können wir auch ihre **Länge**, also die Anzahl der Zeichen aus denen sie bestehen, abfragen. Dazu benutzen wir entweder eine Funktion `length(String)` (unten mehr zu Funktionen) oder Betragsstriche `||String||`.

Wir überprüfen, ob zwei Strings identisch sind, mit dem Gleichheitsoperator (=):

```
"foo" = "bar" //Ausgabe FALSE
"foo" = "foo" //Ausgabe TRUE
```

Die einzelnen Zeichen einer Zeichenkette sind durchnummeriert. Wie in der Informatik üblich, unterscheidet sich diese Zählung von der alltäglich Herangehensweise, da sie mit 0 anfängt. Das heißt, von der Zeichenkette “Haus” ist das *nullte* Element “H”, das *erste* Elemente “a”, das *zweite* Element “u”, usw. In Pseudocode sieht dieses Beispiel so aus:

```
x <- "Haus"
print x[0] //Ausgabe: "H"
print x[1] //Ausgabe: "a"
print x[2] //Ausgabe: "u"
```

Das heißt die Abfrage einzelner Elemente einer Zeichenkette ist über die Syntax `Zeichenkette[Index]` möglich. Zusätzlich ist es möglich, durch **negative Indizes** einfacher auf die hinteren Elemente eines Strings zuzugreifen. Dabei ist –1 der Index des letzten Elements, –2 der Index des vorletzten Elements, usw:

```
var <- "Computerlinguistik ist toll!"
var1 <- var[-1]
print var1 //Ausgabe "!"
```

Um längere Teilketten (Substrings) abzurufen, kann man sogenanntes **Slicing** verwenden. Z.B.

```
x <- "Baumhaus"
print x[0:4] //Ausgabe: "Baum"
```

Das heißt mit der Syntax `Zeichenkette[BeginnIndex:EndIndex]` lässt sich auf die Teilkette von `BeginnIndex` bis `EndIndex` zugreifen (wobei `Zeickenkette[EndIndex]` das erste Zeichen ist, dass *nicht* mehr zur Teilkette gehört).

Diese Syntax lässt sich noch weiter vereinfachen, wenn der Anfang bzw. das Ende des Teilstrings mit dem Anfang bzw. dem Ende der ursprünglichen Zeichenkette zusammenfallen:

```
x <- "Haus"
print x[:2] //Ausgabe: "Ha"
print x[2:] //Ausgabe: "us"
```

3.4 Listen

Eine Liste ist eine Datenstruktur, die *mehrere* unterschiedliche Werte speichern kann, und wird durch eckige Klammern (`[]`) deutlich markiert. Die Elemente der Liste müssen dabei nicht zwingend alle den gleichen Datentyp haben. Das heißt in *einer* Liste können sowohl Zahlen, Booleans, Strings und sogar andere Listen gespeichert sein. Hinsichtlich der erlaubten Operationen verhalten sich Listen quasi identisch zu Strings—Konkatenation, Abfrage der Länge, Zugriff auf Elemente durch Indizes und Slicing funktioniert wie schon von Strings bekannt:

```
//Initialisierung einer Listenvariable
foo <- [1, 2, "1", "2", "Haus", [1,2,3], FALSE]
```

```
//Zugriff auf einzelne Elemente
print foo[1] //Ausgabe 2
print foo[2] //Ausgabe "1"
print foo[5] //Ausgabe [1,2,3]
print foo[5][0] //Ausgabe 1; Sonderfall

//Konkatenation
bar <- [TRUE]
print foo + bar
    //Ausgabe [1, 2, "1", "2", "Haus", [1,2,3], FALSE, TRUE]

//Slicing
print foo[2:4] //Ausgabe ["1", "2"]
```

4 Kontrollstrukturen

Beim Lesen bzw. Schreiben von Programmen im Pseudecode ist zu beachten, dass der Programmfluss stets von oben nach unten, Zeile für Zeile statt findet. Das heißt, die Reihenfolge der Anweisungen beeinflusst das Ergebnis eines Algorithmus entscheidend:

Algorithmus 1:

```
1. x <- 4
2. y <- 9
3.
4. y <- y + x
5. x <- x * 2
6. print y //Ausgabe 13
```

Algorithmus 2:

```
1. x <- 4
2. y <- 9
3.
4. x <- x * 2
5. y <- y + x
6. print y //Ausgabe 17
```

Am Ende von Algorithmus 1 beinhaltet die Variable y den Wert 13, x den Wert 8. Am Ende von Algorithmus 2 beinhaltet y jedoch den Wert 17, während x wieder eine 8 repräsentiert. Der Grund für den Unterschied liegt in der Reihenfolge der Anweisungen. In der Zeile, in der y ein neuer Wert zugewiesen wird (4. bzw. 5. Zeile) ist es wichtig, welchen Wert x zu *diesem Zeitpunkt* enthält. Wird zuerst die Zweisung an y getätigt und dann x verdoppelt (erste Sequenz), so bleibt die Zuweisung an y von der Verdopplung unberührt. Bei der y-Zuweisung in der zweiten Sequenz wurde x bereits verdoppelt und geht somit auch verdoppelt in y ein.

Die *Kontrollstrukturen* `if / else`, `for` und `while` werden hingegen verwendet, um den *Programmfluss zu ändern*. Sie werden also immer dann gebraucht, wenn nicht einfach jede Zeile von oben nach unten nacheinander ausgeführt werden soll.

Jede `if`-, `for`- oder `while`-Anweisung bezieht sich auf einen Block von Anweisungen. Es gehören die Anweisungen zu einem Block, die direkt unter der entsprechenden Anweisung *eingerückt* sind.

4.1 if / else

Die if / else - Struktur („wenn / ansonsten“) bietet eine Möglichkeit, um den normalen Programmfluss eines Programms anzupassen. Die Struktur stellt die Ausführung *ihres Blocks* unter eine Bedingung. `if x > 3` bedeutet „wenn die Variable x den Wert 3 überschreitet – und nur dann – wird der folgende Anweisungsblock ausgeführt“. Ist das nicht der Fall, werden die entsprechenden Anweisungen *übersprungen*, wie im vorigen Abschnitt erklärt wurde. Ein `if` kann dabei ohne `else` vorkommen:

```
if x > 3
     x <- x + 5
     print x
<weitere Anweisungen>
```

Nur wenn x größer ist als 3 wird die Variable um 5 erhöht und anschließend ausgegeben. Ist die Variable kleiner oder gleich 3 passiert *nichts*. Alle Anweisungen nach dem `if`-Block werden auf jeden Fall ausgeführt (es sei denn natürlich, dort finden sich weitere Kontrollstrukturen).

Mit der `else`-Anweisung kann mitgeteilt werden, was passieren soll, falls die `if`-Bedingung *nicht* zutrifft.

```
if x > 3
     x <- x + 5
     print x
else
     x <- x + 10
     print "x wurde um 10 erhöht"
<weitere Anweisungen>
```

Falls x nun also nicht größer als 3 ist, wird eine 10 hinzuaddiert und eine Meldung ausgegeben, dass die Erhöhung um 10 statt gefunden hat. Der Block unter dem `if` wird *nicht* ausgeführt, wohl aber der Block unter dem `else`. Die „weiten Anweisungen“ sind nach wie vor unberührt von der `if / else`-Struktur.

4.2 Schleifen

Schleifen werden eingesetzt, um einen Anweisungsblock mehrmals zu *wiederholen*. Wir lernen hier die beiden geläufigsten Formen, die `for`- und die `while`–Schleife, kennen.

4.2.1 `for`-Schleife

Die `for`-Schleife wird verwendet, um über den Inhalt einer Liste zu *iterieren*. Im *Schleifenkopf* wird dazu eine *Schleifenvariable* eingeführt (*deklariert*) die nacheinander die verschiedenen Werte der Liste durchläuft. Dabei wird jedes Mal der *Schleifenkörper* ausgeführt. Die Schleife ist abgearbeitet, wenn alle Elemente der Liste durchlaufen wurden. Beispiel:

```
summe <- 0
for i in [1,2,3,4,5]
      summe <- summe + i
print i
```

Die *Schleifenvariable* i nimmt jeden Wert in der Liste [1,2,3,4,5] an. Für jeden dieser Werte wird der Schleifenkörper ausgeführt. Ersetzt man im obigen Schleifenkörper in jedem Durchlauf das i durch den aktuellen Wert des Durchlaufs, erhält man die Anweisungen

```
summe <- summe + 1
summe <- summe + 2
summe <- summe + 3
summe <- summe + 4
summe <- summe + 5
```

Es wird also die Summenformel von 5 berechnet. Wichtig ist sehen, dass die „weiteren Anweisungen“ erst nach Beendigung der Schleife ausgeführt werden. Der Programmfluss dreht sich so lange „im Kreis“ und wiederholt immer wieder den Schleifenkörper, bis i auch den Wert des letzten Listenelementes angenommen hat. Erst dann wird die Schleife verlassen.

Wichtig ist zu verinnerlichen, dass die Liste, die durchlaufen wird, nicht nur aufsteigen geordnete Zahlen enthalten darf, sondern beliebige Werte. So gibt der Algorithmus nacheinander die Strings “Hello”, “World” und “!” aus. Auch die Schleifenvariable muss nicht immer mit i bezeichnet werden:

```
for z in ["Hello", "World", "!"]
  print z
```

Für den Fall, dass die Liste, die durchlaufen wird, eine Liste auf- oder absteigend sortierten Zahlen ist, können wir vereinfacht `Anfang...Ende` schreiben:

```
//zaehlt von 99 auf 0 runter
for a in 99...0
  print a

//zaehlt von 1 bis 10 rauf
for b in 1...10
  print b
```

4.2.2 `while`-Schleife

Die zweite Schleifenform, die wir hier verwenden, ist die `while`-schleife. Sie besitzt keine Zählvariable sondern lediglich eine Bedingung. Der Schleifenkörper einer `while`-Schleife wird so lange wiederholt, wie die Bedingung zutrifft (ähnlich wie die `if`-Anweisung, mit dem Unterschied, dass bei `if` der Anweisungsblock nur ein einziges mal ausgeführt wird).

Beispiel:

```
summe <- 1
while summe < 7
      summe <- summe + 2
<weitere Anweisungen>
```

Es ist wichtig zu sehen, dass diese Schleife nur deshalb irgendwann aufhört, weil die Bedingung schließlich nicht mehr zutreffen wird. Dies geschieht durch die Erhöhung der Variable `summe`, deren Wert in der Bedingung abgefragt wird. Mit anderen Worten: Damit eine `while`-Schleife jemals beendet wird, muss der Schleifenkörper Anweisungen enthalten, die die Schleifenbedingung irgendwann – am besten, wenn die Arbeit der Schleife getan ist – ungültig machen. Auch hier gilt: So lange die Schleife nicht beendet wurde, werden die „anderen Anweisungen“ nicht ausgeführt. Das heißt, so lange die Schleifenbedingung wahr ist, werden die Anweisungen nach der Schleife nicht betrachtet.

4.3 Schachtelung von Anweisungsblöcken

Kontrollstrukturen können ohne weiteres geschachtelt werden. Das heißt, eine Kontrollstruktur befindet sich im Anweisungsblock einer anderen Struktur.

Beispiel:

```
for i in [1,2,3,4,5,6,7,8,9,10]
      if i = 5 then
            print "Die fünf wurde erreicht!"
      else
            print "Das ist keine fünf."
```

Für jedes i (also $1, 2, 3, \ldots, 10$) wird der Schleifenkörper ausgeführt. dieser besteht aus einer `if / else`-Struktur. Zum Verständis kann es helfen, den Schleifenblock losgelöst zu betrachten:

```
if i = 5 then
      print "Die fünf wurde erreicht!"
else
      print "Das ist keine fünf."
```

„Ist die Variable i gleich 5, so gib das aus, ansonsten gib aus, dass keine 5 vorliegt“. Man muss sich klar machen, dass diese Anweisungen einfach für jedes i zwischen 1 und 10 ausgeführt werden, wenn sie als Anweisungsblock der obigen `for`-Schleife vorkommen.

5 Funktionen

Funktionen erlauben es uns Abläufe, die wir identisch oder sehr ähnlich an verschiedenen Stellen brauchen, an einem Ort zu sammeln. Wir definieren eine Fuktion mit ihrem Namen, ihrem Input (auch Parameter oder Argumente genannt) und Output (auch Rückgabewert genannt) einmal, können sie danach aber immer wieder aufrufen. Die folgende Beispielsfuntion gibt als Output den zu sich selbst addierten Input zurück:

```
def beispielsfunktion(↓input, ↑output)
        output ← input + input
```

Sie gibt beispielsweise beim Aufruf `beispielsfunktion("string")` den Wert `"stringstring"` zurück.
Stehen mehrere Funktionsaufrufe in einer Zeile werden sie von links nach rechts bearbeitet, wenn mehrere Funktionsaufrufe ineinander geschachtelt sind, werden sie von innen nach außen bearbeitet – das Vorgehen entspricht somit dem beim Lösen einer Gleichung wie $x = 1*(2+2)$. Beispielsweise würde bei `funktion1(funktion2("string"))` zuerst `funktion2("string")` ausgeführt und dann `funktion1` mit deren Ergebnis.
Achtung:
Wird eine Funktion später wieder aufgerufen (oder eine rekursive Funktion mehrfach nacheinander), so sind die Variablen völlig unabhängig voneinander. Somit gilt: `beispielsfunktion(1) + beispielsfunktion(1)` $= 4$ und nicht etwa 6.

6 Abschließendes zum Arbeiten mit Pseudocode

Wie eingangs schon erwähnt, arbeiten wir hier mit Pseudocode und keiner real existierenden Programmiersprache. Das hat den Vorteil, das man Algorithmen nah an der eigenen Intuition schreiben kann. So wäre auch die Anweisung

```
if x größer als 5 then
      print "Die Variable ist größer als 5!"
```

gültig. Ein Test, ob eine Variable durch 7 teilbar ist oder nicht könnte mit

```
if x % 7 = 0 then
      print "Die Variable durch 7 teilbar!"
```

als auch mit

```
if x ist durch 7 teilbar then
      print "Die Variable durch 7 teilbar!"
```

ausgedrückt werden. Hauptsache ist Verständlichkeit sowie die Lösung des Hauptproblems durch algorithmische Strukturen. Falls beispielsweise die Länge einer Liste bestimmt werden muss, um eine höhere Aufgabe zu lösen, so kann geschrieben werden

```
liste <- [1, 4, 23]
länge <- length(liste)    // oder "länge von liste", "len(liste)",...
```

Es kann sich so auf die Hauptaufgaben fokussiert werden.