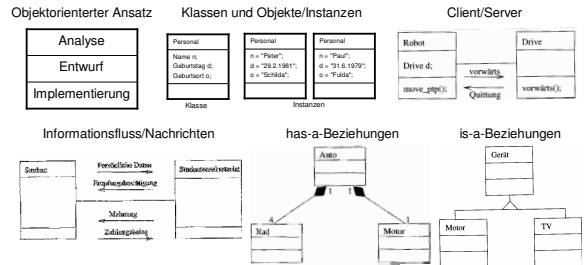


Algorithmen und algorithmische Sprachkonstrukte

Thomas Röfer

Begriff des Algorithmus
 Algorithmenaufbau
 Programmiersprachliche Grundkonzepte
 Iterative und rekursive Algorithmen
 Korrektheitsbeweise von Algorithmen

Rückblick „Objektorientierte Analyse und Entwurf“



Algorithmus

- Ein Algorithmus ist die Beschreibung einer Methode zur Lösung einer gegebenen Aufgabenstellung.
 - Der Begriff kann sehr allgemein gefasst werden – auch Kochrezepte und Gebrauchsanweisungen zum Zusammenbau von Geräten sind Algorithmen.
 - In der Informatik sind die natürlichen Algorithmen interessiert, die in programmierbare Funktionen münden.
- Beispiele
 - Elementare Rechenverfahren wie schriftliches Addieren, Subtrahieren, Multiplizieren und Dividieren
 - Ein Verfahren zur Umwandlung von Dezimalzahlen in Dualzahlen
 - (Der fundamentale Instruktionszyklus eines Prozessors)

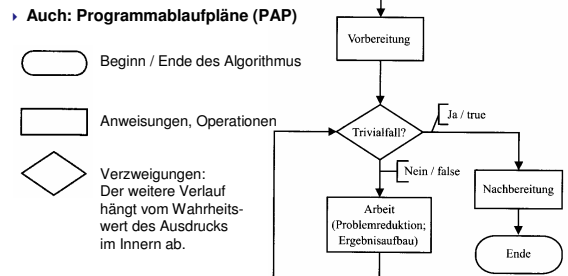
Algorithmus – Definition

- Spezifikation
 - Eingabespezifikation:** Eingabegrößen, Anforderungen an diese
 - Ausgabespezifikation:** Ausgabegrößen, Eigenschaften dieser
- Durchführbarkeit
 - Endliche Beschreibung:** Verfahren muss in endlichem Text vollständig beschrieben sein
 - Effektivität:** jeder Schritt muss effektiv ausführbar sein
 - Determiniertheit:** Verfahrensablauf ist zu jedem Zeitpunkt fest vorgeschrieben
- Korrektheit
 - partielle Korrektheit:** jedes berechnete Ergebnis genügt der Ausgabespezifikation, sofern die Eingaben der Spezifikation genügt haben
 - Terminierung:** Halt nach endlich vielen Schritten mit einem Ergebnis, sofern die Eingaben der Spezifikation genügt haben

Grundschema des Algorithmenaufbaus

- Name des Algorithmus und Liste der Parameter
- Spezifikationen des Ein-/Ausgabeverhaltens
- Schritt 1: Vorbereitung**
 - Einführung von Hilfsgrößen etc.
- Schritt 2: Trivialfall?**
 - Prüfe, ob ein einfacher Fall vorliegt. Falls ja, Beendigung mit Ergebnis.
- Schritt 3: Arbeit (Problemreduktion, Ergebnisaufbau)**
 - Reduziere die Problemstellung X auf eine einfachere Form X' , mit $X > X'$ bezüglich einer wohlfundierten Ordnung $>$. Baue entsprechend der Reduktion einen Teil des Ergebnisses auf.
- Schritt 4: Rekursion bzw. Iteration**
 - Rufe zur Weiterverarbeitung den Algorithmus mit dem reduzierten X' erneut auf (Rekursion), bzw. fahre mit X' bei Schritt 2 fort (Iteration).

Flussdiagramme

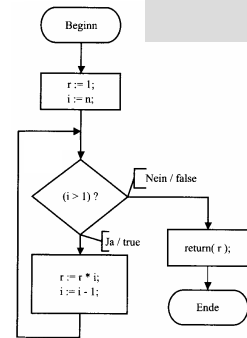


Beispiel: Fakultät

- ▶ **Mathematisch**
 - ▶ $n! = n \cdot (n-1) \cdot \dots \cdot 1, n \in \mathbb{N}, n > 0$
 - ▶ $0! = 1$
- ▶ **Beispiele**
 - ▶ Möglichkeiten im Lotto „6 aus 49“: $\frac{49!}{(49-6)! \cdot 6!}$
 - ▶ Möglichkeiten, 350 Studenten auf 450 Plätze zu verteilen: $\frac{450!}{(450-350)!}$
- ▶ **Allgemeinsprachliche Beschreibung**
 - ▶ Berechnung der Fakultät von n :
 - ▶ Die Fakultät von 0 bzw. 1 ist 1.
 - ▶ Ansonsten bilde das Produkt aller Zahlen zwischen 2 und n .

Beispiel: Fakultät

- ▶ **Algorithmus: factorial(n)**
 - ▶ Anforderung: $n \in \mathbb{N}$
 - ▶ Zusicherung: das Resultat ist $n!$
 - 1. Kopiere 1 nach r .
 - 2. Kopiere n nach i .
 - 3. Prüfe, ob i größer als 1.
 - 4. Falls nein, gib das Resultat r aus.
 - 5. Falls ja, multipliziere r mit i und ziehe 1 von i ab.
 - 6. Fahre mit Schritt 3 fort.

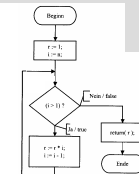


Programmiersprachliche Grundkonzepte

- ▶ **Datenspeicher und Zuweisungen**
 - ▶ *Variablen* und *Zuweisungen* zum Speichern von berechneten (Zwischen-)Ergebnissen
 - ▶ *Konstanten* zum Bezeichnen fester Werte, z.B. π
- ▶ **Ausdrücke**
 - ▶ *Boolsche* und *arithmetische Ausdrücke* zum Auswerten von Formeln und Bedingungen, z.B. $a + b$ oder $a > b$
- ▶ **Sequenzierungsanweisungen**
 - ▶ *Blöcke* zum Gruppieren von Daten und Anweisungen, z.B. in Java: `{ ... }`
 - ▶ *Funktionsaufrufe* zum mehrmaligen Wiederverwenden einmal definierter Algorithmen, z.B. $x = \sin(\alpha) + \sin(\beta)$
 - ▶ *Bedingte Anweisungen* zum Verzweigen im Programmfluss, z.B. `if(a > b) { r = a; } else { r = b; }`
 - ▶ *Iterationsanweisungen* zur Realisierung von Schleifen, z.B. `while(a > b) { a = a - b; }`
- ▶ **Hilfskonstrukte**
 - ▶ *Kommentare*: `/*` Kommentar `*/` oder `//` Kommentar bis Zeilenende

Sprünge (goto)

- ▶ **Zweck**
 - ▶ Fortsetzung des Programmflusses an anderer Stelle
- ▶ **Nachteile**
 - ▶ Sprünge führen zu sog. *Spaghetti-Code*
 - ▶ In Java gibt es kein `goto`
- ▶ **BASIC mit Zeilennummern**
 - ▶ 10 INPUT n
 - ▶ 20 r = 1
 - ▶ 30 i = n
 - ▶ 40 IF i > 1 THEN GOTO 70
 - ▶ 50 PRINT r
 - ▶ 60 END
 - ▶ 70 r = r * i
 - ▶ 80 i = i - 1
 - ▶ 90 GOTO 40



- ▶ **BASIC mit Sprungmarken**
 - ▶ INPUT n
 - ▶ r = 1
 - ▶ i = n
 - ▶ loop: IF i > 1 THEN GOTO reduct
 - ▶ PRINT r
 - ▶ END
 - ▶ reduct: r = r * i
 - ▶ i = i - 1
 - ▶ GOTO loop

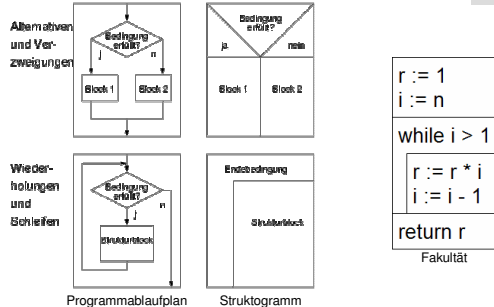
Strukturierte Verzweigungen

- ▶ **Verzweigung mit Sprüngen**
 - ▶ IF NOT(Bedingung) THEN GOTO else
 - ▶ // Anweisungen, falls Bedingung wahr
 - ▶ GOTO ende
 - ▶ else:
 - ▶ // Anweisungen, falls Bedingung falsch
 - ▶ ende:
 - ▶ // Weiter im Programmfluss...
- ▶ **Strukturierte Verzweigungen**
 - ▶ IF Bedingung THEN
 - ▶ // Anweisungen, falls Bedingung wahr
 - ▶ ELSE
 - ▶ // Anweisungen, falls Bedingung falsch
 - ▶ END IF
 - ▶ // Weiter im Programmfluss...
- ▶ **In Java**
 - ▶ if(Bedingung)
 - ▶ // Anweisung, falls Bedingung wahr
 - ▶ else
 - ▶ // Anweisung, falls Bedingung falsch
- ▶ **Tipps**
 - ▶ else-Zweig kann fehlen
 - ▶ Falls mehr als eine Anweisung: `{...}`
 - ▶ Einrückungen!

Strukturierte Schleifen

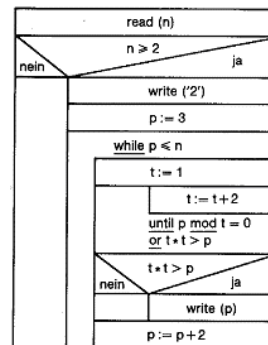
- ▶ **Schleifen mit Sprüngen**
 - ▶ loop:
 - ▶ IF NOT(Bedingung) THEN GOTO ende
 - ▶ // Anweisungen, solange Bedingung wahr
 - ▶ GOTO loop
 - ▶ ende:
 - ▶ // Weiter im Programmfluss...
- ▶ **Strukturierte Schleifen**
 - ▶ WHILE Bedingung DO
 - ▶ // Anweisungen, solange Bedingung wahr
 - ▶ END WHILE
 - ▶ // Weiter im Programmfluss...
- ▶ **In Java**
 - ▶ while(Bedingung)
 - ▶ // Anweisung, solange Bedingung wahr
- ▶ **Verschiedene Schleifen-Typen**
 - ▶ Abweisend: `while(...)` ...
 - ▶ Annehmend: `do ... while(...)`
 - ▶ Zählen: `for(...; ...; ...)` ...

Struktogramme (Nassi-Shneiderman)



Beispiel

- ▶ Primzahlberechnung



Beispiel: Fakultät in Java (iterativ)

- ▶ **In Java**

```

class Factorial
{
    static int factorial(int n)
    {
        // Vorbereitung
        int f = 1;
        int i = n;
        // Problemreduktion
        while(i > 1)
        {
            r = r * i;
            i = i - 1;
        }
        // Trivialfall
        return r;
    }
}
            
```
- ▶ **Testfälle**
 - ▶ Trivialfälle: $n = 0, 1$
 - ▶ Sonstige Fälle, z.B. $n = 3, 6$
- ▶ **Gültige Eingabewerte**
 - ▶ $n \in \{0 \dots 12\}$
 - ▶ Fakultät nur für natürliche Zahlen definiert
 - ▶ Maximal 12! lässt sich noch mit 32-Bit-Integers repräsentieren!
- ▶ **Tipp**
 - ▶ *static*-Funktionen sind *Klassen-Methoden*, d.h. sie beziehen sich nicht auf ein bestimmtes Objekt
 - ▶ Daher muss keine Instanz der Klasse existieren, um sie aufrufen zu können
 - ▶ Aufruf von außen: `Klassenname.methodenname(parameterliste)`

Programmentwicklung in BlueJ

- ▶ **Schema**
 1. Prototyp erstellen, der zumindest die richtige Signatur hat und sich compilieren lässt
 2. Testklasse samt der Testfälle erstellen (Trivialfall + weitere)
 3. Prototyp zu vollständiger Implementierung erweitern, immer wieder testen
- ▶ **Vorteile**
 - ▶ Automatisches Testen ist viel einfacher als manuelles Testen
 - ▶ Wahl der Testfälle muss wohlüberlegt sein
 - ▶ Testen ist Pflicht!
 - ▶ *Ohne dokumentierte Tests gibt's Punktabzug*

Rekursion

- ▶ **Rekursiver Ansatz der Problemlösung**

Versucht, ein Problem $P(X)$ nach folgendem Schema zu lösen:

 - ▶ **Basis:** Direkte Lösung, falls Problemstellung (Eingabe) X einfacher Natur ist (Trivialfall)
 - ▶ **Schritt:** Führe eine Lösung für das Problem $P(X)$ für komplexere Problemstellungen X durch einen Schritt der Problemreduktion auf die Lösung des gleichen Problems für eine einfachere Problemstellung $P(X')$ zurück. Dabei muss $X > X'$ gelten für eine wohlfundierte Ordnungsrelation $>$.
- ▶ **Beispiel**

$$factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$$

Beispiel: Fakultät in Java (rekursiv)

- ▶ **In Java**

```

class Factorial
{
    static int factorial(int n)
    {
        if(n == 0)
        {
            // Trivialfall
            return 1;
        }
        else
        {
            // Problemreduktion
            return n * factorial(n - 1);
        }
    }
}
            
```

Beweis rekursiver Algorithmen

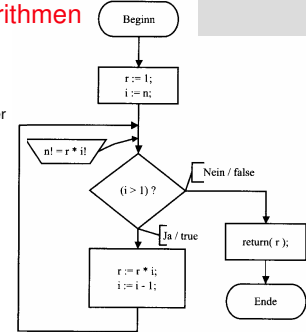
- Beweisverfahren**
 - Rekursive Algorithmen können durch vollständige Induktion bewiesen werden
 - Beweis**
 - Behauptung: Die Funktion

$$factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$$
 berechnet die Fakultät von n , d.h. $factorial(n) = n!$
 - Induktionsanfang**
 - $factorial(0) = 1 = 0!$
 - Induktionsschritt**
 - $factorial(n) = n!$

$$= n \cdot (n-1)! = n \cdot factorial(n-1)$$
- q.e.d.

Beweis iterativer Algorithmen

- Verifikation nach Floyd**
 - Finde eine geeignete Formel $F(V)$ und zeige, dass sie eine Schleifeninvariante am Anfang der Schleife ist; bezeichne $F(V)$ mit $INV(V)$
 - Zeige, dass aus der Eingabespezifikation folgt, dass $INV(V)$ vor dem ersten Schleifendurchgang gültig ist
 - Zeige, dass nach dem letzten Schleifendurchgang aus $INV(V)$ und der unerfüllten Fortsetzungsbedingung der Schleife die Ausgabespezifikation folgt



Beweis iterativer Algorithmen

- Schleifeninvariante**
 - $n! = r \cdot i!$
- Vor der Schleife**
 - $n! = r \cdot i!$
 - $= 1 \cdot n!$
 - $= n!$
- Invarianz in der Schleife**
 - Zu Beginn: $n! = r \cdot i!$
 - $i' = r' \cdot i, i' = i - 1$
 - Nach einem Durchlauf:
 - $n! = r' \cdot i!$
 - $= (r \cdot i) \cdot (i-1)!$
 - $= r \cdot i \cdot (i-1)!$
 - $= r \cdot i!$
- Nach der Schleife**
 - $n! = r \cdot i!$
 - $= r \cdot 1$
 - $= r$

