Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Algorithmenparadigmen

Unter einem Paradigma versteht man unter anderem in der Wissenschaftstheorie ein »Denkmuster, welches das wissenschaftliche Weltbild einer Zeit prägt« – ein Algorithmenparadigma sollte daher ein Denkmuster darstellen, das die Formulierung und den Entwurf von Algorithmen und damit letztendlich von Programmiersprachen grundlegend prägt.

- Es gibt aber keine einheitliche Auffassung über die Anzahl existierender Algorithmenparadigmen und deren Abgrenzung.

Algorithmen

- Ein **Algorithmenparadigma** legt die Denkmuster fest, die einer Beschreibung eines Algorithmus zugrunde liegen.
- Ein **Algorithmus** ist eine Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

Es gibt zwei grundlegende Arten, Schritte von Algorithmen zu notieren (schießloich die Programmierklassen):

- Applikative Algorithmen sind eine Verallgemeinerung der Funktionsauswertung mathematisch notierter Funktionen.
  - In ihnen spielt Rekursion eine wesentliche Rolle.
- Imperative Algorithmen basieren auf einem einfachen Maschinenmodell mit gespeicherten und änderbaren Werten.
  - Hier werden primär Schleifen und Alternativen als Kontrollbausteine eingesetzt.
- In der Informatik werden weitere Paradigmen diskutiert: logisch, objektorientiert, agentenorientiert, parallel.

Applikative Algorithmen

Die Idee applikativer Algorithmen besteht darin, eine Definition zusammengesetzter **Funktionen** durch Terme mit Unbestimmten vorzunehmen. Eine einfache Funktionsdefinition in diesem Sinne kann wie folgt mathematisch notiert werden:

$$f(x) = 5x + 1$$

- Streng genommen erfüllt die Funktionsdefinition erst zusammen mit einem Auswertungsalgorithmus für Terme unsere Anforderungen an eine Algorithmensprache – erst die Termauswertung legt die Reihenfolge der atomar auszuführenden Berechnungsschritte fest.

Applikative Algorithmen

Applikative Algortihmen entsprechen den funktionalen Programmiersprachen und bestehen aus:

1. Termen als Ausdrücke für die Berechnung
2. Termen mit Unbestimmten
3. Funktionsdefinitione
4. Funktionsauswertung

Applikative Algorithmen

Terme

Gegeben sind im Folgenden zwei (unendliche, abzählbare) Mengen von Symbolen (als »Unbestimmte« bezeichnet):

```
x, y, z, … vom Typ int
q, p, r, … vom Typ bool
```

- Wir müssen nun die Definition von Termen auf Terme mit Unbestimmten erweitern.

1. Die int-Werte …, − 2, −1, 0, 1, … sind elementare nicht weiter zerlegbar int-Terme.
2. Sind t, u int-Terme, so sind auch (t + u), (t − u), (t · u), (t ÷ u), sign(t), abs(t) int-Terme.
3. Ist b ein bool-Term und sind t, u int-Terme, so ist auch `if b then t else u fi` ein int-Term.
4. Nur die durch diese Regeln gebildeten Zeichenketten sind int-Terme.

Def. 1. Definition von int-Termen und Regelwerk zur Berechnung

- Ein Term mit Unbestimmten wird analog zu Termen ohne Unbestimmte gebildet.

```
x, x − 2, 2x + 1, (x + 1)(y − 1)
```

Def. 2. Terme vom Typ int mit Unbestimmten

```
p, p ∧ true, (p ∨ true) = ⇒ (q ∨ false)
```

Def. 3. Terme vom Typ bool

Termkomposition

- Komplexeren Berechnungen kann man durch Zusammensetzen von Grundoperationen durchführen

$$7 + (9 + 4) \cdot 8 - 14$$
$$13 - \text{sign}(-17) \cdot 15$$

- Bei der Termbildung dienen Klammern und Prioritäten zur Festlegung der Auswertungsreihenfolge

- Für Algorithmensprachen gibt es eine weitere Art von Termen , die in normaler Arithmetik nicht eingesetzt werden (oder doch?).

- **Bedingte Terme** erlauben – analog dem Auswahloperator in der Pseudocode-Notation – die Auswahl zwischen zwei Alternativen basierend auf dem Test eines Prädikats. Notiert wird ein bedingter Term wie folgt:

```
if b then t else u fi
```

Funktionsdefinition

- Basierend auf der erweiterten Termdefinition, können wir nun eine Notation für die Funktionsdefinitionen festlegen.

Sind $v_1, \ldots, v_n$ Unbestimmte vom Typ $\tau_1, \ldots, \tau_n$ (bool oder int) und ist $t(v_1, \ldots, v_n)$ ein Term, so heißt

$$f(v_1, \ldots, v_n) = t(v_1, \ldots, v_n)$$

eine Funktionsdefinition der Funktion f vom Typ τ. τ ist dabei der Typ des Terms t(v1, …, vn).

Def. 4. Funktionsdefinitionen

- f heißt Funktionsname,
- die $v_1, \ldots, v_n$ heißen formale Parameter und
- der Term $t(v_1, \ldots, v_n)$ heißt Funktionsausdruck.

```
foo(a,b) = a+b
```

Funktionen

Jede Funktion ist also bestimmt durch:

1. Einen Namen (oder ohne Name: Lambda Ausdruck)
2. Formalen Parametern
3. Funktionsausdruck (Funktionskörper)
4. Einer Eingabemenge $\mathbb{E}$ von Werten
5. Einer Ausgabemenge $\mathbb{A}$ von Werten
6. Einer Relation $\boldsymbol{r}$: $\mathbb{E} \rightarrow \mathbb{A}$

Auswertung von Funktionen

- Definierte Funktionen können mit konkreten Werten aufgerufen und ausgewertet werden.

- Eine Funktionsdefinition definiert eine Funktion mit der folgenden Signatur:

$$f : \tau_1 \times \ldots \times \tau_n \to \tau$$

- Sind nun $a_1, .., a_n$ Werte vom Typ $\tau_1, .., \tau_n$, so ersetzt man bei der Auswertung von $f(a_1, .., a_n)$ im definierenden Term jedes Vorkommen der Unbestimmten v_i durch den Wert a_i und wertet dann den entstehenden Term $t(a_1, .., a_n)$ aus.

- Die konkreten Werte $a_1, \ldots, a_n$ heißen aktuelle Parameter.

- Den Ausdruck $f(a_1, \ldots, a_n)$ bezeichnen wir als Funktionsaufruf.

Funktionale Komposition

- Terme sind Ausdrücke
- Aufrufe definierter Funktionen sind auch Terme
- Terme können also Funktionsaufrufe enthalten
- Argumente von Funktionen können wiederum Funktionsaufrufe sein

```
f(x, y) = if g(x, y) then h(x + y) else h(x - y) fi
g(x, y) = (x = y) ∨ odd(y)
h(x) = j(x + 1) · j(x - 1)
j(x) = 2x - 3
```

Das grundlegende Programmierparadigma von funktionalen Sprachen (Haskell, OCaML)

- Eine Funktionsdefinition f heißt rekursiv, wenn direkt (oder indirekt über andere Funktionen) ein Funktionsaufruf f(..) in ihrer Definition auftritt.

[addp]

$$
\begin{array}{lll}
f(1,2) & \mapsto & \textbf{if } g(1,2) \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\
 & \mapsto & \textbf{if } 1=2 \lor \textbf{odd}(2) \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\
 & \mapsto & \textbf{if } 1=2 \lor \textbf{false} \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\
 & \mapsto & \textbf{if } \textbf{false} \lor \textbf{false} \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\
 & \mapsto & \textbf{if } \textbf{false} \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\
 & \mapsto & h(1-2) \\
 & \mapsto & h(-1) \\
 & \mapsto & j(-1+1) \cdot j(-1-1) \\
 & \mapsto & j(0) \cdot j(-1-1) \\
 & \mapsto & j(0) \cdot j(-2) \\
 & \mapsto & (2 \cdot 0 - 3) \cdot j(-2) \\
 & \mapsto^* & (-3) \cdot (-7) \\
 & \mapsto & 21
\end{array}
$$

Abb. 1. Funktionsauswertung; obwohl scheinbar mehrschrittig wird die Berechnung nur durch Datenabhängigkeiten "gesteuert". Es findet eine schrittweise Termreduktion durch Einsetzen und Zusammenfassen sowie Berechnung mit grundlegenden Operationen statt.

Schließlich kann zusammengefasst werden:

Ein applikativer Algorithmus ist eine Liste von Funktionsdefinitionen:

$$f_1(v_{1,1},\ldots,v_{1,n1}) = t_1(v_{1,1},\ldots,v_{1,n_1}) \vdots f_m(v_{1,m},\ldots,v_{m,n1}) = t_m(v_{m,1},\ldots,v_{m,n_m})$$

Die erste Funktion f_1 wird wie beschrieben ausgewertet und ist die Bedeutung (Semantik) des Algorithmus (quasi die main Funktion).

Def. 5. Applikativer Algorithmus

Applikative Algorithmen sind die Grundlage einer Reihe von universellen Programmiersprachen, wie APL, Lisp, Scheme, Haskell oder Scala etc. Diese Programmiersprachen werden als funktionale Programmiersprachen bezeichnet.

Ein applikativer Algorithmus muss nicht für alle Eingabewerte zu einem definierten Ergebnis führen. Das folgende Beispiel verdeutlicht dies.

```
f(x) = if x = 0 then 0 else f(x - 1) fi
```

- f(−1) → f(−2) → … Auswertung terminiert nicht!
- Eine nicht terminierende Berechnung führt als Vereinbarung zum Ergebnis ⊥ für undefiniert. Also gilt:

$$f(x) = \begin{cases} 0 & x \geq 0 \\ \perp & \end{cases}$$

- Eine (möglicherweise) für einige Eingabewertekombinationen undefinierte Funktion heißt partielle Funktion.

```
x! = x(x - 1)(x - 2) … 2 · 1 für x > 0
fac(x) = if x = 0 then 1 else x · fac(x - 1) fi
```

Bsp. 1. Beispiel für einen applikativen Algorithmus: Fakultät mit Rekursion

```
f0 = f1 = 1, fi = fi−1 + fi−2 für i > 0
fib(x) = if (x = 0) V (x = 1) then 1
         else fib(x − 2) + fib (x − 1) fi
```

Bsp. 2. Beispiel für einen applikativen Algorithmus: Fibonacci Zahlen mit Rekursion

```
f(x) = if x > 100 then x − 10 else f(f(x + 11)) fi
⇔
f(x) = if x > 100 then x − 10 else 91 fi
```

Bsp. 3. Noch ein kurioses Beispiel: Die McCarthys 91 Funktion und zwei Funktionen mit gleicher Ein- und Ausgabemenge und Relation

Imperative Algorithmen

Die imperativen Algorithmen bilden die verbreitetste Art, Algorithmen für Computer zu formulieren, da sie auf einem abstrakten Modell eines üblichen Rechners basieren.

- Wir hatten trotzdem zuerst applikative Algorithmen betrachtet, da diese mathematisch einfacher zu formalisieren und zu analysieren sind.

- Imperative Algorithmen basieren auf den Konzepten **Anweisung und Variable**.

    - Das imperative Paradigma ist sehr nah mit dem intuitiven Algorithmenbegriff verwandt und wird durch viele Programmiersprachen wie C, Pascal, Fortran, COBOL oder auch von Java realisiert.

Variablen

Variablen bilden die Speicherplätze für Werte:

- Eine Variable besteht aus einem Namen (z.B. X) und einem veränderlichen Wert. Jeder Variablen ist ein Typ zugeordnet.
- Ist t ein Term ohne Unbestimmte und w(t) sein Wert, dann bezeichnet man das Paar X :=t als Wertzuweisung. Ihre Bedeutung ist festgelegt durch:

```
Nach Ausführung von X:=t gilt: X = w(t).
```

- Vor der Ausführung der ersten Wertzuweisung gilt: $X = \perp$ (undefiniert).

Zustände

Die möglichen Zustände eines Rechners werden wie folgt festgelegt:

- Ist $\mathbb{X} = \{X1, X2, \dots\}$ eine Menge von Variablen(namen), von denen jede Werte aus der Wertemenge W annehmen kann (alle Variablen vom gleichen Typ), dann ist ein Zustand Z eine partielle Abbildung:

```
Z : 𝕏 → W (Zuordnung des momentanen Wertes).
```

- Ist $Z : \mathbb{X} \to W$ ein Zustand und wählt man eine Variable $X \in \mathbb{X}$ und einen Wert w vom passenden Typ, so ist der transformierte Zustand durch eine Änderung der Variablen X gegeben, d.h. es findet ein Zustandsübergang $Z_i \to Z_{i+1}$ statt!

- Mit einer Anweisung α wird ein Zustand Z in einen (anderen) Zustand $[\![\alpha]\!](Z)$ überführt.

- Um Werte, die Variablen als Zwischenergebnisse zugewiesen wurden, später wiederverwenden zu können, muss man Werte aufrufen bzw. auslesen können.

Ausdrücke

- Ausdrücke entsprechen im Wesentlichen den Termen einer applikativen Sprache, jedoch stehen an der Stelle von Unbekannten Variablen.

In applikativen (funktionalen) Algorithmen können Unbestimmte wie in einem Gleichungssystem "parallel" im Sinne von bewachten Ausdrücken verwendet werden, d.h. erst ein Term der eine Unbestimmte *x* verwendet, und dann folgend ein Term der die Unbestimmte berechnet, oder mehrere Terme die die Unbestimmten bestimmen. In imperativen Sprachen gibt es eine strikte Sequenz: Erst berechnen, dann verwenden.

```
Applikativ                                   Imperativ
===============                              =========
y=x-t   fac n | n===1 = 1                    a:=1;          fac(n) {
t=a+b         | n > 1 = n *fac(n-1)          b:=2;            if (n<0) then return null;
a=1           | n<1   = ⊥                    t:=a+b;           else if (n==1) return 1:
b=2                                          y:=x-t;          else return n*fac(n-1)
                                                            }
```

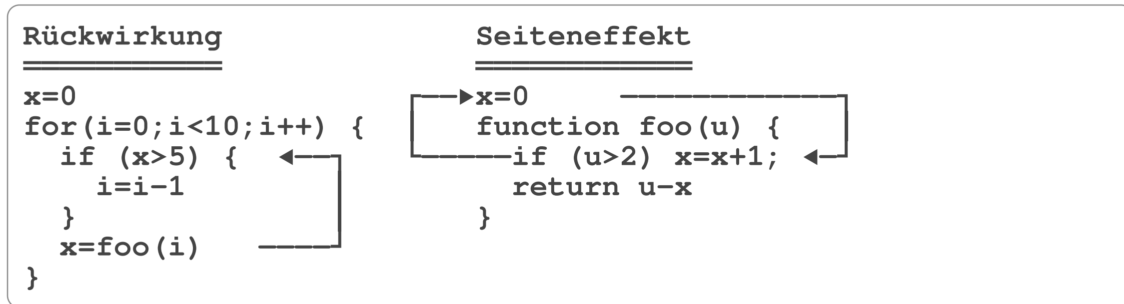
Bsp. 4. Terme mit Unbestimmten (funktional) und Ausdrücke mit Variablen

Ausdrücke

Die Auswertung von Termen mit Variablen ist zustandsabhängig. An der Stelle der Variablen wird ihr Wert im gegebenen Zustand gesetzt. Imperative Algorithmen könne Seiteneffekte haben, d.h. dann dass ein Berechnugn mit den gleichen Eingabe- zu verschiedenen Ausgabewerten führt!

- Man unterscheidet gebundene und freie Variablen.
  - Gebundene Variablen sind in Java Klassenvariablen, oder Methoden/Funktionsparameter
  - Frei Variablen sind solche die außerhalb eines lokalen Kontext definiert und überall verändert werden können!
  - Schleifen verwenden Variablen die u.U. zuerst gelesen und dann verändert werden (Rückwirkung)

Seiteneffekte und Rückwirkung



Bsp. 5. Beispiele für Rückwirkung und Seiteneffekte

Komplexe Anweisungen

Die komplexen Anweisungen imperativer Algorithmen bilden eine Untermenge der intuitiven Algorithmenbausteine.

- Die Folge oder Sequenz bildet den ersten Baustein imperativer Algorithmen: Sind α1 und α2 Anweisungen, so ist α1;α2 auch eine Anweisung.

1. Die Bedeutung der Sequenz ist durch die folgende Zustandstransformation festgelegt:

```
⟦α1;α2⟧ = ⟦α2⟧(⟦α1⟧(Z))
```

2. Die Auswahl oder Selektion bildet den zweiten Baustein: Sind α1 und α2 Anweisungen und B ein boolescher Ausdruck, so ist

```
if B then α1 else α2 fi
```

eine Anweisung.

Komplexe Anweisungen

3. Die letzte und für imperative Algorithmen charakteristische Kontrollstruktur ist die Wiederholung oder Iteration: Ist α eine Anweisung und B ein boolescher Ausdruck, so ist

```
while B do α od
```

eine Anweisung. Die Iteration wird wie bei den intuitiven Algorithmen oft als Schleife bezeichnet.

Programme

- Ein Programm $\mathbb{P}$ bildet eine Eingabe- auf eine Ausgabemenge ab, ist letztlich auch wieder eine Funktion, die aus Anweisungen α besteht die den zustand Z eines Programms durch Änderungen von Variablen verändert.

[[PROG]]:		$W_1 \times \ldots \times W_n \mapsto V_1 \times \ldots \times V_m$
$[\![\text{PROG}]\!](w_1, \ldots, w_n)$	=	$(Z(Y_1), \ldots, Z(Y_m))$
wobei Z	=	$[\![\alpha]\!](Z_0)$,
$Z_0(X_i)$	=	$w_i,\ i = 1, \ldots, n$,
und $Z_0(Y)$	=	$\perp$ für alle Variablen
		$Y \neq X_i,\ i = 1, \ldots, n$

Zusammenfassung

Kurz gefasst lassen sich imperative Algorithmen wie folgt charakterisieren:

Die Algorithmenausführung besteht aus einer Folge von Basisschritten, genauer von Wertzuweisungen. Diese Folge wird mittels Selektion und Iteration basierend auf booleschen Tests über dem Zustand konstruiert. Jeder Basisschritt definiert eine elementare Transformation des Zustands. Die Semantik des Algorithmus ist durch die Kombination all dieser Zustandstransformationen zu einer Gesamttransformation festgelegt.

Deduktive Algorithmen

Das sogenannte logische Paradigma führt uns zu den deduktiven Algorithmen.

- Deduktive Algorithmen basieren auf logischen Aussagen und sind die Grundlage von Programmiersprachen wie PROLOG (für PROgramming in LOGic).
- Logische Algorithmen bestehen aus einer Reihe von logischen Aussagen und einem Auswertungsalgorithmus für Anfragen.

Daher sind sie nicht in dem Sinne Algorithmen, wie wir sie bisher betrachtet haben – erst die Kombination der drei Bestandteile, logisches Programm, Auswertungsalgorithmus und konkrete Anfrage, legen eine Berechnungsfolge fest.

Während die meisten sonstigen Programmiersprachen applikative und imperative Elemente mischen, werden aufgrund dieser Unterschiede deduktive Elemente nur in speziellen logischen Programmiersprachen eingesetzt.

```
%% sudoku.pl
:- use_module(library(clpfd)).

sudoku(Puzzle) :-
    flatten(Puzzle, Tmp), Tmp ins 1..9,
    Rows = Puzzle,
    transpose(Rows, Columns),
    blocks(Rows, Blocks),
    maplist(all_distinct, Rows),
    maplist(all_distinct, Columns),
    maplist(all_distinct, Blocks),
    maplist(label, Rows).

blocks([A,B,C,D,E,F,G,H,I], Blocks) :-
    blocks(A,B,C,Block1), blocks(D,E,F,Block2), blocks(G,H,I,Block3),
    append([Block1, Block2, Block3], Blocks).

blocks([], [], [], []).
blocks([A,B,C|Bs1],[D,E,F|Bs2],[G,H,I|Bs3], [Block|Blocks]) :-
    Block = [A,B,C,D,E,F,G,H,I],
    blocks(Bs1, Bs2, Bs3, Blocks).
```

Bsp. 6. Sudoko Löser in Prolog mit Randbedingungslöser. That's all folks.

Genetische Algorithmen

Genetische Algorithmen sind durch Prinzipien der Informationsverarbeitung in der Natur inspiriert.

- Die DNS kann als Analogon zu einem Programm aufgefasst werden, indem Gene als ein in einem 4-Buchstaben-Alphabet kodiertes Programm interpretiert werden, das den Bauplan etwa eines Tieres (inklusive einiger Aspekte des Verhaltens!) festgelegt.

- In der Natur gibt es keine Programmierer, die Fortpflanzung mit der Kombination aus männlichem und weiblichem Erbgut zusammen mit Mutationen verändert das Erbgut; die natürliche Auslese führt zur Ausprägung von Eigenschaften und letztendlich zu neuen Arten.

  - Durch Auslese driftet der Gen-Pool langfristig in eine Richtung, die besser an die auslesenden Umweltbedingungen angepasst ist.

Genetische Algorithmen

Umgesetzt in die Denkweise der Informatik, arbeitet ein genetischer Algorithmus wie folgt:

1. Ein Pool von Phänotypen repräsentiert Individuen, die eine bestimmte Problemklasse (näherungsweise) zu lösen vermögen.
2. Die Lösungsstrategie jedes Phänotyps ist in seinem Erbgut kodiert (entsprechend den Chromosomen in der Natur). Die Strategie kann beispielsweise als Bitfolge fester Länge dargestellt sein.
3. Eine Generation von Individuen wird auf Probleme der Problemklasse angesetzt. Einzelne Phänotypen werden betreffend der Qualität der Lösungen (oft Fitness genannt) bewertet.
4. Erreicht die Fitness eines Phänotyps einen vorgegebenen Schwellwert, kann dieses Programm als Löser für Probleme der Problemklasse eingesetzt werden.
5. Entspricht noch kein Phänotyp den Anforderungen, so wird eine neue Generation von Phänotypen aus den bisherigen Phänotypen berechnet. Die Fitness der bisherigen Phänotypen bestimmt dabei, inwieweit diese als Elternteil einbezogen werden. Das Erbgut eines Kindes kann durch Kombination des Erbguts zweier Eltern und / oder durch Mutation berechnet werden.

Genetische Algorithmen

- Genetische Algorithmen sind typischerweise einsetzbar für **Optimierungsaufgaben**, in denen eine sehr gute Näherungslösung, aber nicht notwendigerweise die optimale Lösung gesucht wird (es gibt Probleme, für die die optimale Lösung nicht mit vertretbarem Aufwand berechenbar ist, Komplexitätsklasse NP!).

```
Initialisiere erste Generation G;

do
  BesteFitness = max { Fitness(x) |x in G };
  if BesteFitness = 0 then break;
  Bestimme G′ aus G als Eltern der neuen Generation G;
  // etwa durch n Turniere aus je k zufällig aus G gewählten x
  G = {};
  for each x in G′ do
    with probability 0.7 do x = mutate(x) od;
     with probability 0.2
       do wähle y aus G′ aus;
       x = crossover(x, y);
       od;
    G = G union { x };
  od;
od;
return Kandidaten x aus G mit Fitness(x) = BesteFitness;
```

Alg. 1. Grundprinzip genetischer Algorithmen

Entwurfsprinzipien

Der Entwurf von Algorithmen und damit von Programmen ist eine konstruktive und kreative Tätigkeit, die den Entwerfer immer wieder vor neue Herausforderungen stellt – neben der reinen Funktionalität sind ja auch Fragen der Laufzeitkomplexität und andere, nichtfunktionale Anforderungen zu berücksichtigen.

Die automatische Ableitung eines optimalen Algorithmus aus einer Beschreibung der Anforderungen ist prinzipiell nicht automatisierbar!

ad-dpunkt 8.1

TODO: ad-dpunkt: Greedy (S. 226), DaC (S. 231), Backtracking (S. 234), Dyn. Programmierung (S. 242)

Schrittweise Verfeinerung

Die Vorgehensweise der schrittweisen Verfeinerung basiert auf folgendem Ablauf:

1. Beginne mit einer kurzen Folge abstrakter Lösungsschritte (d.h. etwa Anweisungen in Pseudocode)
2. Verfeinere die Anweisungen Schritt für Schritt in detailliertere Anweisungen, zunächst in verfeinerten Pseudocode und letztendlich in konkrete Algorithmenschritte
3. Fertig, wenn alle Anweisungen in der Zielsprache (z.B. Java) formuliert sind

In der Terminologie der Programmierung entspricht der Verfeinerungsschritt der Formulierung von Algorithmen auf einer abstrakten Stufe, bei der statt ausführbarer Einzelschritte Prozeduraufrufe eingesetzt werden, deren Bedeutung dann durch die Ausprogrammierung der Prozedur konkretisiert wird.

Fakorisierung, d.h. Aufteilung eines Programms oder einer großen Funktion in viele kleine Funktionen ist ebenfalls schrittweise Verfeinering von Algorithmen.

Schrittweise Verfeinerung

Aber schrittweise Verfeinerung ist Übgergang von abstrakter in konkrete Notation mit einer Programmiersparche.

- Es sollte so lange wie möglich während der Verfeinerung eine geeignete Notation (Pseudocode, natürliche Sprache) genutzt werden (so abstrakt wie möglich, so detailliert wie nötig).
- Jeder Verfeinerungsschritt ist mit einer Entwurfsentscheidung verbunden, die spätere Eigenschaften des Programms wie **Geschwindigkeit oder Speicherplatzbedarf** beeinflusst.
- Das bedeutet, dass unter Umständen derartige Entscheidungen später revidiert werden müssen.
    - Zim Beispiel ist dies etwa die Wahl des Sortierverfahrens oder auch die Entscheidung, dass die Eingabe für das Programm eine bereits sortierte Folge sein muss.

Schrittweise Verfeinerung

- Neben dem Ablauf müssen auch die zu verarbeitenden Daten schrittweise verfeinert werden, und auch Datenstrukturen angepasst werden.

Auch Datenstrukturen sowie Klassen können verfeinert (also refaktorisiert) werden.

Einsatz von Algorithmenmustern

> Die Idee des Einsatzes von Algorithmenmustern besteht darin, generische algorithmische Muster für bestimmte Problemklassen zu entwickeln und diese dann jeweils an eine konkrete Aufgabe anzupassen.

- Man versucht also für eine allgemeine Problemklasse, zum Beispiel das Finden einer kostenoptimalen Lösung in einem großen Lösungsraum (etwa das Finden kürzester Wege), eine Muster-Implementierung zu finden und diese dann an das konkrete Problem anzupassen.

Einsatz von Algorithmenmustern

Diese Grundidee kann man auf verschiedene Weise konkretisieren:

- Das Lösungsverfahren wird an einem möglichst einfachen Vertreter der Problemklasse vorgeführt und dokumentiert. Der Entwerfer versteht die Problemlösungsstrategie und überträgt diese auf sein Programm.
- Eine Bibliothek von Mustern (»Design Pattern«, »best practice«-Strategien) wird genutzt, um einen abstrakten Programmrahmen zu generieren. Die freien Stellen dieses Programmrahmens werden dann problemspezifisch ausgefüllt.
- Moderne Programmiersprachen benutzen parametrisierte Algorithmen und Vererbung mit Überschreiben, um Algorithmenmuster als lauffähige Programme zur Verfügung zu stellen und diese dann an ein konkretes Problem anzupassen.

Problemreduzierung durch Rekursion

> Rekursive Algorithmen sind ein für die Informatik spezifischer Lösungsansatz, der in den klassischen Ingenieurwissenschaften nicht entwickelt werden konnte – ein mechanisches Bauteil kann physikalisch nicht sich selbst als Bauteil enthalten, während ein Programm sich durchaus selbst aufrufen kann.

- Rekursive Programmierung ist daher in der Regel nicht aus dem Alltagswissen ableitbar, sondern muss als Technik erlernt und geübt werden.

Formale Algorithmenmodelle sind auch ohne rekursive Aufrufe bereits berechnungsuniversell.

Problemreduzierung durch Rekursion

- Man könnte daher meinen, Rekursion als solche ist eigentlich nicht notwendig. Tatsächlich formen Compiler rekursive Programme in nichtrekursiven Maschinencode um (Schleifen).

Allerdings ist das rekursive Anwenden einer Problemlösungsstrategie auf Teilprobleme ein Algorithmenmuster, mit dem man bestimmte Problemklassen einfach realisieren kann.

```
==== Linear ====             ==== Rekursiv ====

function fac(n) {            function fac(n) {
  y=1                          if (n>1) return n*fac(n-1)
  for(i=2;i≤n;i++) {           else return 1
    y=y*i                    }
  }
  return y
}
```

Algorithmenmuster: Greedy

Greedy steht hier für gierig. Das Prinzip gieriger Algorithmen ist es, in jedem Teilschritt so viel wie möglich zu erreichen. Wir werden das Prinzip an einem kleinen Beispiel verdeutlichen und danach ein realistisches Problem und dessen »gierige« Lösung behandeln.

Problemklasse, für die sie angewendet werden können:

- Gegeben ist eine feste Menge von Eingabewerten.
- Es gibt eine Menge von Lösungen, die aus Eingabewerten aufgebaut sind.
- Alle Lösungen lassen sich schrittweise aus partiellen Lösungen, beginnend bei der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen.
- Es existiert eine Bewertungsfunktion für partielle und vollständige Lösungen.
- Gesucht wird die bzw. eine optimale Lösung bezüglich der Bewertungsfunktion.

Greedy-Algorithmen berechnen nicht für alle derartigen Probleme tatsächlich die optimale Lösung.

Rekursion: Divide-and-conquer

> Das bereits mehrfach behandelte Prinzip der Rekursion wendet bei der Lösung eines Problems die Gesamtlösungsstrategie innerhalb der algorithmischen Lösung rekursiv auf ein geändertes Problem an.

- Typischerweise erfolgt diese erneute Anwendung des Gesamtverfahrens auf ein vereinfachtes Problem, um eine Terminierung zu garantieren.
- Das Prinzip »Divide-and-conquer«, deutsch »Teile und herrsche«, basiert nun darauf, in einem Schritt **eine große Aufgabe in mehrere kleinere Aufgaben zu teilen** und diese rekursiv zu bearbeiten – also ein klassischer Einsatz des Rekursionsprinzips.

Rekursion: Divide-and-conquer

Vorgehensweise:

1. Zerlege das gegebene Problem in mehrere getrennte Teilprobleme,
   - löse diese einzeln
   - und setze die Lösungen des ursprünglichen Problems aus den Teillösungen zusammen.
2. Wende dieselbe Technik auf jedes der Teilprobleme an, dann auf deren Teilprobleme usw., bis die Teilprobleme klein genug sind, dass man eine Lösung explizit angeben kann.
3. Strebe an, dass jedes Teilproblem von derselben Art ist wie das ursprüngliche Problem, so dass es mit demselben Algorithmus gelöst werden kann.

Rekursion: Divide-and-conquer

```
function DIVANDCONQ (P: problem) {
  if (P is klein) then
    return f(P) // direkte Berechnung
  else {
    {P1,P2,...,Pk} = split(P)
    R1=DIVANDCONQ ( P1 )
    R2=DIVANDCONQ ( P2 )
    ..
    Rk=DIVANDCONQ ( Pk )
    return merge(R1,R2,...,Rk)
  }
}
```

Alg. 2. Grundprinzip DaC

- Dieses Muster muss natürlich noch um die Repräsentation und Übergabe der Teillösungen verfeinert werden.

Rekursion: Backtracking

> Das Backtracking ist ein weiteres wichtiges Algorithmenmuster für Such- und Optimierungsprobleme. Das Backtracking realisiert eine allgemeine **systematische Suchtechnik**, die einen vorgegebenen **Lösungsraum komplett** bearbeitet.

Backtracking basiert auf dem vollständigen Durchlauf eines Lösungsraums, so dass die gesuchte Lösung tatsächlich erreicht wird.

- Dabei müssen Konfigurationen betrachtet werden, die jeweils in einem Schritt zu einer neuen Konfiguration erweitert werden können.
- Auch muss entscheidbar sein, ob wir eine Lösung (quasi den Käse) auch tatsächlich gefunden haben.

Rekursion: Backtracking

Formal bedeutet es, dass für unsere Problemstellung die folgenden Voraussetzungen gelten müssen:

- KF ist die Menge von Konfigurationen K.
- K0 ist die Anfangskonfiguration.
- Für jede Konfiguration Ki kann die Menge der direkten Erweiterungen $K_{i,1}$, .. , $K_{i,ni}$ bestimmt werden.
- Für jede Konfiguration ist entscheidbar, ob sie eine Lösung ist.

Backtracking ist ein rekursives Verfahren, das mit dem Aufruf BACKTRACK(K0), also mit der Anfangskonfiguration, gestartet wird.

Rekursion: Backtracking

```
function FindeLösung (Stufe, Vektor) {
  1. wiederhole, solange es noch neue Teil-Lösungsschritte gibt:
     a) wähle einen neuen Teil-Lösungsschritt;
     b) falls Wahl gültig ist:
               I) erweitere Lösungsvektor um Wahl;
              II) falls Lösungsvektor vollständig ist, return true; // Lösung gefunden!
        sonst:
                  falls (FindeLösung(Stufe+1, Vektor)) return true; // Lösung!
                  sonst mache Wahl rückgängig; // Sackgasse (Backtracking)!
  2. Da es keinen neuen Teil-Lösungsschritt gibt: return false // Keine Lösung!
}
```

Alg. 3. Backtracking-Muster

Komplexität

- Die Tiefensuche und somit auch Backtracking haben im schlechtesten Fall mit $O(z^N)$ und einem Verzweigungsgrad $z > 1$ eine exponentielle Laufzeit.
- Je größer die Suchtiefe N, desto länger dauert die Suche nach einer Lösung. Daher ist das Backtracking primär für Probleme mit einem kleinen Lösungsbaum geeignet.

Es gibt jedoch Methoden, mit welchen die Zeitkomplexität eines Backtracking-Algorithmus verringert werden kann. Diese sind unter anderem:

- Heuristiken
- Akzeptanz von Näherungslösungen und Fehlertoleranz
- Durchschnittliche Eingabemenge
- Randomisierung ("Simuliertes Abkühlen")

Dynamische Programmierung

Dynamische Programmierung vereint Aspekte der drei bisher vorgestellten Algorithmenmuster.

- Vom Ansatz der Greedy-Algorithmen wird die Wahl optimaler Teillösungen übernommen,
- von Divide-and-conquer und Backtracking die rekursive Herangehensweise basierend auf einem Konfigurationsbaum.

Während Divide-and-conquer-Verfahren unabhängige Teilprobleme durch rekursive Aufrufe lösen, werden bei der dynamischen Programmierung abhängige Teilprobleme optimiert gelöst, indem mehrfach auftretende Teilprobleme nur einmal gelöst werden.

Dynamische Programmierung

Der Anwendungsbereich der dynamischen Programmierung sind Optimierungsprobleme analog zu Greedy-Algorithmen – mit dynamischer Programmierung werden aber insbesondere Probleme bearbeitet, bei denen die Greedy-Vorgehensweise nicht zu optimalen Lösungen führt.