Kapitel 7

Entwurf und Korrektheit von Algorithmen

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> *(Donald Knuth)*

7.1 Idee

Warnung!

Es gibt **keinen** (exakten) Algorithmus, um Algorithmen zu entwickeln.

Abstraktionsvermögen, Kreativität und Erfahrung tragen aber dazu bei, (gute) Algorithmen zu entwickeln. Jede dieser Eigenschaften kann durch Training verbessert werden.

- Jedoch verlangt jeder Algorithmus **Problemverständnis** und (mindestens) eine **Idee**

Problemlösen

Man beachte:

Algorithmenentwurf ist Problemlösen.

Allgemeines Vorgehen beim Problemlösen:

1. Verstehen Sie das Problem!
2. Was ist das Ziel? Was ist gegeben?
3. Modellieren Sie das Problem!
4. Lösen Sie das Modellproblem!
5. Interpretieren Sie die Lösung!

Problemverstehen

Merke

Ich habe ein Problem **dann und erst dann** verstanden, wenn ich es anderen erklären kann!

- **Interessante Fragen:**
  - Woraus genau besteht die Eingabe?
  - Was genau ist das gewünschte Ergebnis oder die gewünschte Ausgabe?
  - Kann ich ein Minimalbeispiel angeben, das von Hand zu lösen ist? Was passiert, wenn ich dies mache?
  - Brauche ich eine optimale, eine nahezu optimale oder nur irgendeine Lösung? Was heißt in diesem Problem „optimal“?
  - Wie groß ist eine typische Probleminstanz? 10 Objekte? 1000 Objekte? 1000000 Objekte?
  - Wie wesentlich ist Geschwindigkeit in meinem Problem? Muss es in Sekunden gelöst sein? Eine Minute? Eine Stunde? Ein Tag?
  - Wie kann das Problem in einer Grafik beschrieben werden?
  - Welche Gemeinsamkeiten hat das Problem mit anderen bekannten Problemen?
  - Was unterscheidet dieses Problem von anderen ähnlichen Problemen?

Suchen der Idee

- Niemand kann Ihnen abnehmen, die Lösungsidee zu finden
- Jedoch gibt es mehrere typische Ansätze:
  - **Invarianten** ➡ Herausfinden, was sich **nicht** ändert
  - **Induktion** ➡ Übertragung vom (hoffentlich einfachen) Einzelfall auf alle Fälle
  - **Abstraktion und Nachnutzung** ➡ Das Problem so **modellieren**, dass es mit anderen (bekannten) Problemen übereinstimmt
  - **Suchen** mit „brutaler Gewalt“ (***brute force***) ➡ Alle infrage kommenden Fälle untersuchen, ob sie die Lösung bieten
  - **Intelligentes Suchen** ➡ Wie „brute force“, aber vorher den Suchraum einschränken

Modellierung

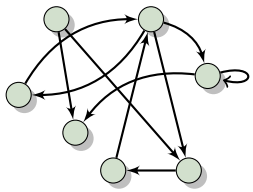
- Durch Modellierung kann ein Problem auf das Wesentliche beschränkt werden
- Als Modellierungsansätze sind vor allem grundlegende Konzepte aus der Mathematik geeignet:
  - **Mengen** (ungeordnete Elemente) und **Multimengen** (mehrfaches Vorkommen erlaubt)
  - **Permutationen** (geordnete Elemente)
  - **Hierarchien**/**Bäume**
  - **Graphen**
  - **Punkte** (Koordinaten in geometrischen Räumen)
  - **Polygone** (Regionen in geometrischen Räumen)
  - **Zeichenketten**

Exkurs: Graphen

Da Graphen eine wesentliche Methode zur Modellierung sind, wird hier eine (sehr) kurze Einführung in die Graphentheorie eingeschoben.

Falls Sie bereits mit Graphen vertraut sind, können Sie diese überspringen.

- **Graphen** sind ein häufig benutztes Mittel zur Modellierung von Sachverhalten
- Wir haben sie bereits ständig gebraucht, ohne sie zu benennen
- Vorteile von Graphen: gute grafische Representation
  - Knoten = Kreise, Punkte (engl.: *nodes, vertices*)
  - Kanten = Linien, Pfeile (engl.: *edges, arcs*)
- Ein ganzer Bereich der Mathematik beschäftigt sich mit Graphen ➡ **Graphentheorie**



Definition 7.1: Graph

Ein **Graph** ist eine Menge von **Knoten** $\mathbf{V} = \{v_1, v_2, \ldots, v_n\}$, die durch eine

(Multi-)Menge von **Kanten** $\{e_1, e_2 \ldots, e_m\}$ in Beziehung stehen können. Kanten können **ungerichtet** oder **gerichtet** sein. Ungerichtete Kanten werden als Knotenmengen $\{v_i, v_j\}$ aufgefasst, gerichtete Kanten als geordnete Knotenpaare (Tupel) (v_i, v_j).[1] Gerichtete Kanten werden in der Regel durch Pfeilspitzen repräsentiert.

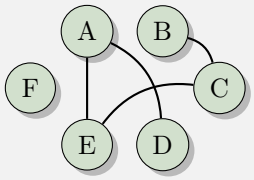
Kanten können Werte (Gewichte) zugewiesen werden, $w : \mathbf{E} \to \mathbf{R}$. In diesem Fall spricht man von einem **gewichteten Graph**.

Beispiel 7.1

Anton, Bernd, Claudia, Dora, Ernst und Franziska seien bei dem Social-Web-Dienst „FaceVZ“ angemeldet. Es sind jeweils miteinander „befreundet“[2]:

- Anton und Dora
- Bernd und Claudia
- Anton und Ernst
- Claudia und Ernst

Dieser Sachverhalt lässt sich mit folgenden (ungerichteten) Graphen modellieren:



Beispiel 7.2

Anna, Brigitte, Charlotte, Denise, Eva und Friederike gehen ab und zu gemeinsam in eine Kneipe. Wenn, was häufiger vorkommt, die eine der Damen ihr Portemonnaie vergessen hat, zahlt eine andere für sie.

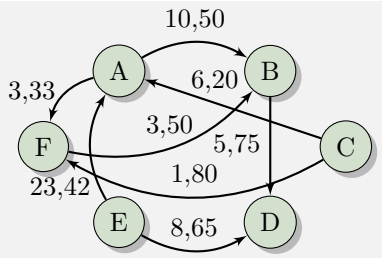
Am Ende des Jahres ergibt sich folgendes Schuldverhältnis:

- Anna schuldet Brigitte 10,50€ und Friederike 3,33€
- Brigitte schuldet Denise 5,75€
- Claudia schuldet Anna 6,20€ und Friederike 1,80€
- Eva schuldet Anna 23,42€ und Denise 8,65€
- Friederike schuldet Brigitte 3,50€

Die ist darstellbar in einem gerichteten und gewichteten Graphen:

[1] Häufig wird auch bei ungerichteten Graphen die Tupel-Schreibweise genutzt und $(v_i, v_j) \equiv (v_j, v_i)$ vorausgesetzt.

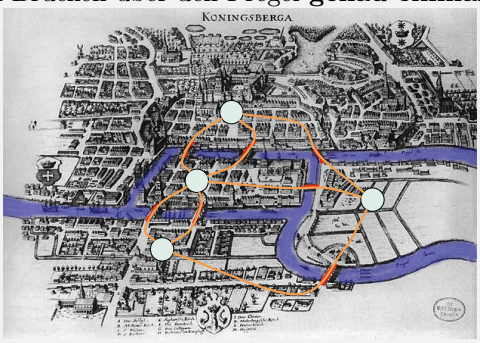
[2] Was immer das auch bedeuten mag.



- Durch die Reduzierung auf das (Graphen-)modell abstrahiert sich die Problemstellung

Beispiel 7.3: Königsberger Brückenproblem

Problem aus dem frühen 18. Jahrhundert: Gibt es in Königsberg einen Weg, bei dem man alle sieben Brücken über den Pregel **genau einmal** überquert?



Leonhard Euler löste dieses Problem 1736 mit Hilfe der Graphentheorie.

- Das **Modell** des Graphen ist von der **Datenstruktur** zu unterscheiden
- Beispielsweise kennt C keinen Datentyp für Graphen ➡ dieser muss ggf. selbst definiert werden

Achtung!

Nur weil ein Problem mit einem bestimmten Ansatz **modelliert** wurde, bedeutet dies **nicht** unbedingt, dass auch eine entsprechende Datenstruktur für diesen Ansatz gebraucht wird.

Falls tatsächlich eine solche Datenstruktur gebraucht wird, gibt es verschiedene Möglichkeiten, z.B.:

- **Adjazenzmatrix**: zweidimensionales Array, bei dem jedes Element $a_{i,j}$ beschreibt, ob vom Knoten v_i zum Knoten v_j eine Kante vorhanden ist oder welches Gewicht diese hat

```
enum {NumNodes=5};
bool a[NumNodes][NumNodes];
     :
a[0][1]= true;
// =  edge (v0,v1) exists
```

- Eignet sich besonders für gerichtete Graphen
- **Verbundstypen und Zeiger**: Knoten können durch struct-Typen und Kanten durch darin enthaltene Zeiger modelliert werden

```
enum{ MaxDegree = 6};
struct node;
typedef struct node{
  int id;
  struct node *in[MaxDegree];
  struct node *out[MaxDegree];
} node_t;
```

- Eignet sich besonders für Graphen mit variabler Knotenanzahl

Beliebige andere Ansätze sind denkbar...

- Da die Graphentheorie ein ziemlich altes Gebiet der Mathematik ist, gibt es viele Sätze und Standardalgoritmen, die man direkt für die Lösung eines vorliegenden Problems nutzen kann
- Beispiele:
  - Satz über die Existenz von Euler-Wegen und -Kreisen
  - Kürzeste Wege in einem gewichteten Graphen
  - Maximaler Fluss zwischen zwei Knoten in einen gewichteten Graph
  - Effektivität von Suchen in Bäumen
- Eine kompakte Darstellung von Graphenalgorithmen bietet z.B. [Bra94]

Von der Idee zum Algorithmus

- Hat man eine Idee gefunden, muss der Algorithmus entwickelt werden
- Vorgehen der **schrittweisen Verfeinerung** (**Top-Down-Methode**)
  1. Beschreibe die Grobidee zur Lösung des Problems
  2. Zerlege das Problem in Unterprobleme
  3. Verfahre mit jedem Unterproblem entsprechend
- Wann beenden? ➡ Wenn **ausführbare Einzelschritte** gefunden sind
- Fähigkeiten des ausführenden Systems müssen bekannt sein

7.2 Spezifikation

- Ein Algorithmus muss irgendwie beschrieben werden
- Eine Variante ist eine konkrete Implementation in einer konkreten Programmiersprache
- **Problem:** Eine Programmiersprache kennt nicht alle möglichen Konzepte

Beispiel 7.4

C kennt keine Datentypen und Operationen für mathematische Mengen. Mengenoperationen würden in C kompliziert aussehen und daher evtl. die **Idee** eines Algorithmus verschleiern.

- **Abhilfe:** Beschreibung des Algorithmus in natürlicher Sprache
- **Problem:** Mangel an Exaktheit
- Deshalb werden Algorithmen (wenn die algorithmische Idee betont werden soll) in **Spezifikationssprachen** beschrieben
- Es gibt sehr viele Spezifikationssprachen
- Viele dienen nur dazu, die **Anforderungen** an den Algorithmus zu beschreiben ➡ für uns (hier) nicht interessant
- Es gibt **formale** und halbformale Spezifikationssprachen
  - Formale: **Automatisch** verarbeitbar, z.B. in Theorem Beweisern ➡ häufig schwer zu lesen[3]
  - Halbformale: Dient nur zur **Kommunikation** über Algorithmen

[3] ...für Menschen 😉

- Wir nutzen die wahrscheinlich gängigste halbformale Spezifikationssprache für imperative Algorithmen: **Pseudocode**

Pseudocode

- Jeder von Ihnen hat schon Pseudocode gesehen[4]
- Ähnlich PASCAL
- Legt einen Satz von Anweisungen fest
- Lässt beliebige mathematische Terme zu
- Lässt Sätze in natürlicher Sprache zu
  - Sollte möglichst wenig gebraucht werden
  - Müssen mit Verb beginnen und keine Nebensätze haben

Hinweis

Da Pseudocode niemals ausgeführt wird, gibt es eine Vielzahl verschiedener Formen.

In der Regel ist Pseudocode aber stets intuitiv lesbar, wenn man jemals mit einer imperativen Programmiersprache zu tun gehabt hat.

Pseudocode: Syntax und Layout

Anweisungen in Pseudocode:

- Auswertung eines Ausdrucks und **Speicherung** des Wertes in einer Variablen

```
var ← expression
```

- **Verzweigung**

```
if condition then
    statement(s)
end if
```

oder

[4] Oder er/sie hat die Hausaufgaben nicht vollständig gelesen.

```
if condition then
    statement(s)
else
    statement(s)
end if
```

- **While-Schleife**

```
while condition do
    statement(s)
end while
```

- **Repeat-Until-Schleife**[5]

```
repeat
    statement(s)
until condition
```

- **For-Schleife**

```
for expression do
    statement(s)
end for
```

- **Eingabe/Vorbedingung**

```
Require: requirement(s)
```

- **Ausgabe/Nachbedingung**

```
Ensure: postcondition(s)
```

- **Kommentare**

```
[statement]                                    ▷ comment
```

oder

[5] ...entspricht `do ... while` in C.

```
[statement]          { comment }
```

- **Unteralgorithmen**

```
procedure NAME(parameter)
    statement(s)
    [return expression]
end procedure
```

- **Aufruf**

```
NAME(parameter)
```

oder

```
call NAME(parameter)
```

Beispiel

- Beispiel „Euklidscher Algorithmus“ (ähnlich zu Aufgabe 6 auf Aufgabenblatt 3)

```
Require: A, B ∈ N, A > 0 ∧ B > 0
Ensure: a = b = gcd(A, B)                    ▷ gcd: greatest common divisor

  a ← A; b ← B
  while a ≠ b do
      if a < b then
          b ← b − a
      else
          a ← a − b
      end if
  end while
```

- Künftig werden wir Algorithmen häufig in Pseudocode angeben, wenn wir von konkreten Datenstrukturen abstrahieren können

Korrektheit

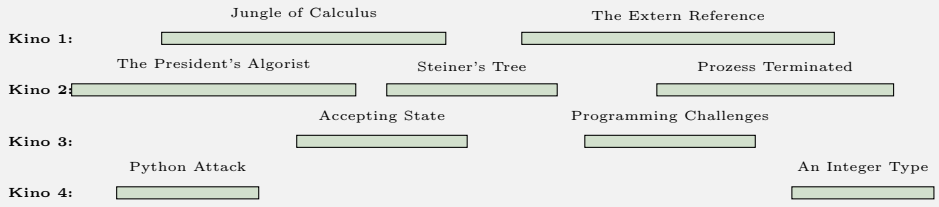
- Wenn ein Ansatz gefunden wurde, muss seine **Korrektheit** geprüft werden
- Zunächst sollte die Korrektheit der **Idee** geprüft werden, dann die anschließend die des konkreten Algorithmus
- Die Korrektheit einer Idee kann nicht formal geprüft werden
- Folgendes sollte überdacht werden:
  - Funktioniert die Idee allgemein oder nur in einem Spezialfall?
  - Gibt es Spezialfälle, in denen die Idee nicht funktioniert? Sind diese Fälle relevant?
  - Versuche, ein Gegenbeispiel zu finden
  - Extremfälle helfen

Beispiel

Beispiel 7.5

Sie sind Fan oder sogar Juror bei einem Filmfestival. Sie wollen soviel Filme wie möglich anschauen, die aber leider z.T. in verschiedenen Kinos laufen und sich zeitlich überschneiden.

Wie sollte Ihr allgemeines Vorgehen aussehen, damit Sie möglichst viele Filme sehen?



Probleme dieses Typs nennt man *Schedulingprobleme* (Planungsprobleme). Sie sind ein häufig auftretender Problemtyp.

- **Modellierung**:
  - **Gegeben** ist eine Menge I von Intervallen
  - **Gesucht** ist die größte Untermenge nichtüberschneidender Intervalle
- ~~**1. Idee:** EarliestJobs~~
  - Wähle den Film, der als erster anfängt
  - Streiche alle überlappenden Filme

  - Starte von vorn bis kein Film übrig

- ~~**2. Idee:** ShortestJobs~~
  - Wähle den kürzesten Film
  - Streiche alle überlappenden Filme
  - Starte von vorn bis kein Film übrig
- **3. Idee**: ExhaustiveSearch ➡ **funktioniert**
  - Konstruiere alle Mengen von Filmen
  - Werfe alle Mengen weg, in denen sich mindestens zwei Filme überlappen
  - Wähle die Menge(n) mit den meisten Filmen aus

  Bei n Filmen gibt es 2^n verschiede Mengen, die betrachtet werden müssen.
- **4. Idee:** EarliestCompletion ➡ **funktioniert effizient**
  - Wähle den Film, der zuerst beendet ist
  - Streiche alle überlappenden Filme
  - Starte von vorn bis kein Film übrig
- Betrachten weiteres Beispiel

Beispiel 7.6: Traveling Salesman

Ein Vertreter (Handlungsreisender) muss in verschiedene Städte fahren und anschließend nach Hause zurückkehren.

Wie muss er seine Route planen, damit die zurückzulegend Gesamtstrecke möglichst kurz ist?

- **Modellierung**:
  - Nutzen gewichteten Graphen
  - Knoten sind Städte
  - Kanten zwischen zwei Knoten (Städten) haben Entfernung als Kantengewicht
- ~~**1. Idee:** NearestNeighbor~~
  - Wähle als erstes Ziel die nächstliegende Stadt aus
  - Wiederhole solange, bis keine unbesuchte Stadt übrig ist
  - Kehre nach Haus zurück
- ~~**2. Idee:** ClosestPair~~

  - Wähle die beiden Städte aus, die am nächsten sind
  - Streiche alle Städte, die bereits zwei Verbindungen haben
  - Wiederhole, bis alle Städte verbunden sind

- **3. Idee**: ExhaustiveSearch ➡ **funktioniert** (theoretisch)
  - Konstruiere alle möglichen Wege
  - Wähle den kürzesten aus
- Ein auf dieser Idee basierender Algorithmus ist **extrem** langsam
  - Er muss bei n Städten $\frac{n-1}{2}!$ Wege vergleichen
  - Bei $n = 30$ sind dies 4.420.880.996.869.850.977.271.808.000.000 ($\approx 4{,}4 \cdot 10^{30}$) verschiedene Wege
- Der derzeit[6] schnellste Computer der Welt „Sunway TaihuLight“ würde für ein Problem mit 30 Städten mehrere Millionen(!) Jahre arbeiten!
- Konzept funktioniert nur bei **sehr** kleinen Problemen

Merke

Für dieses Problem gibt es keine effektivere Lösung.

Man begnügt sich daher in der Praxis mit Näherungslösungen. Ein solcher Algorithmus, der keine korrekte Lösung garantiert, heißt **Heuristik**.

Algorithmenkorrektheit

- Auch der fertige Algorithmus muss auf seine **Korrektheit** geprüft werden
- Mitunter beinhaltet die Idee gleich einen kompletten Algorithmus, so dass gleich der konkrete Algorithmus geprüft werden kann
- Prinzipiell gibt es **zwei** Möglichkeiten:
  - **Erschöpfendes Testen**
    Ausschluss fehlerhaften Verhaltens einer Implementation des Algorithmus durch Ausführung (Test) **mit jeder möglichen Instanz der Eingabe**
  - **Korrektheitsbeweis**
    Nachweis korrekten Verhaltens durch Nutzung von **mathematischen** Methoden

[6] September 2016

Probleme

Merke

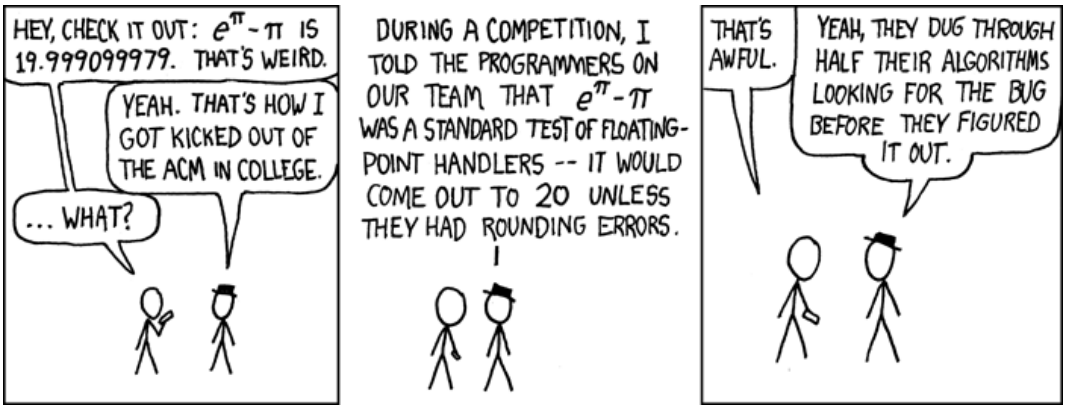
Erschöpfendes Testen ist in der Regel **unmöglich**, da der Eingaberaum meist sehr groß oder sogar unendlich ist.

- Durch **teilweises** Testen kann das Vertrauen in die Korrektheit eines Programmes gestärkt werden, es ersetzt aber keinen Nachweis
- Die gute Auswahl von Testfällen ist schwierig

Merke

Korrektheitsbeweise auf Implementationsebene sind i.d.R. wegen mangelnder Formalisierung/Formalisierbarkeit der Implementation **schwierig oder unmöglich**.

- Ersatzweise wird der Korrektheitsbeweis auf einer **abstrakteren Ebene** durchgeführt ➡ Fehler bei Umsetzung in konkrete Implementation möglich
- Abhilfe können (bis zu einem gewissen Maß) Tools, die selbst formal bewiesen sind, schaffen.
  - Allgemein ist dies aber nicht möglich ➡ **Gödelscher Unvollständigkeitssatz**
- Beschäftigen uns hier mit Beweisen (nicht Testen)



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
http://xkcd.com/217/

7.3 Beweise

- Was ist ein Beweis?

Beweis – informale Definition

Ein Beweis ist eine Herleitung einer Aussage aus bereits bewiesenen Aussagen und/oder Grundannahmen (Axiomen).

- Wir kennen (aus der Mathematik) nur wenige (auch kombinierbare) Beweismethoden
  - Deduktion
  - Vollständige Fallunterscheidung (Aufzählung)
  - Vollständige/transfinite Induktion
  - Indirekte Beweise
- Betrachten diese Methoden kurz

Deduktion

- Deduktion ist der „klassische“ Beweis durch Kombination von Prämissen

Beispiel 7.7

- **Prämisse 1:** Alle Menschen sind sterblich
- **Prämisse 2**: Alle Könige sind Menschen
- **Folgerung:** Alle Könige sind sterblich

- Die Korrektheit der Prämissen muss entweder axiomatisch gegeben oder bereits bewiesen sein

Vollständige Fallunterscheidung

- Bei endlich vielen Varianten kann jede einzeln untersucht werden
- Ist eine Aussage für jede Variante wahr, ist sie insgesamt wahr

Beispiel 7.8

- **Behauptung:** Alle ungeraden Zahlen im Intervall $[2^1, 2^3]$ sind Primzahlen.

$$\forall i = 2n + 1, n \in \mathbf{N}, i \in [2^1, 2^3], prim(i)$$

- **Fälle:** Im Intervall sind nur die Zahlen 3, 5 und 7 ungerade

  - 3 ist prim
  - 5 ist prim
  - 7 ist prim
- **Folgerung:** Behauptung stimmt

Anmerkung:

Die Operatoren $\forall$ und $\exists$ sind aussagelogische **Quantoren** (oder **Quantifikatoren**):

$\forall$:
- **Bedeutung**: Aussage gilt für alle Elemente einer Menge
- **Sprechweise**: „für alle“
- **Beispiel**: $\forall x \in \{1, \ldots, 4\}, stat(x)$ ➡„Für alle Werte x im Bereich von 1 bis 4 gilt Aussage $stat(x)$“ $\forall x \in \{1, \ldots, 4\}, stat(x)$ ist äquivalent zu $stat(1) \land stat(2) \land stat(3) \land stat(4)$

$\exists$:
- **Bedeutung**: Es gibt ein Element in einer Menge, für Aussage gilt
- **Sprechweise**: „es existiert“, „es gibt“
- **Beispiel**: $\exists x \in \{1, \ldots, 4\}, stat(x)$ ➡„Für mindestens ein x im Bereich von 1 bis 4 gilt Aussage $stat(x)$“ $\exists x \in \{1, \ldots, 4\}, stat(x)$ ist äquivalent zu $stat(1) \lor stat(2) \lor stat(3) \lor stat(4)$

Vollständige/transfinite Induktion

- **Induktionsanker** (auch: Induktionsanfang, IA) ➡ Zeigen, dass eine Aussage für einen speziellen (kleinsten) Fall (meist: $n = 0$ oder $n = 1$) gilt
- **Induktionsschritt** (IS)
  - **Induktionsvoraussetzung** (IV): Annahme, dass Aussage für ein $n = k$ gilt
  - **Induktionsbehauptung** (IB): Behauptung, dass Aussage für $n = k + 1$ gilt
  - Beweis, dass Induktionsbehauptung aus Induktionsvoraussetzung gilt
- **Induktionsschluss** ➡ Folgerung, dass Aussage für alle (unendlich viele) Fälle ab dem ersten Fall gilt
- Detaillierter in LV „Mathematik I“

Beispiel 7.9

- **Behauptung:** $\boxed{\sum_{i=1}^{n}(2i-1) = n^2}$ ➡ Summe der ersten n ungeraden Zahlen ist n^2
- **IA**: $n = 1$ ➡ $\sum_{i=1}^{1}(2i-1) = 2-1 = 1$ ➡ w.A.
- **Induktionsschritt:** ***IV:*** $\sum_{i=1}^{k}(2i-1) = k^2$, ***IB:*** $\sum_{i=1}^{k+1}(2i-1) = (k+1)^2$ ***Beweis:*** $\sum_{i=1}^{k+1}(2i-1) = \sum_{i=1}^{k}(2i-1) + (2(k+1)-1) = k^2 + 2k + 1 = (k+1)^2$
- **Induktionsschluss:** Behauptung gilt für alle $n \geq 1$

Indirekter Beweis

- Annahme des **Gegenteils** der Behauptung
- Unter Nutzung lediglich der Annahme, von Axiomen und bewiesenen Aussagen einen **Widerspruch** finden
- Folgerung dass Annahme falsch und folglich Behauptung wahr ist

Beispiel 7.10

- **Behauptung:** Es gibt keine größte Primzahl
- **Annahme** A**:** Es gibt eine größte Primzahl
- **Beweis:**
  - Aus A folgt, dass es nur endlich viele Primzahlen $p_1, \ldots, p_n$ gibt (B)
  - Bilden $q = 1 + \prod_{i=1}^{n} p_i$
  - Fallunterscheidung:
    1. q ist prim ➡ q ist größer als p_n ➡ Widerspruch zu A
    2. q hat Primteiler ➡ diese sind nicht in $p_1, \ldots, p_n$ ➡ Widerspruch zu B und damit zu A
- **Schlussfolgerung:** Annahme A führt zum Widerspruch, folglich ist die Behauptung wahr

Beweis von Algorithmen

- Fragestellung ist ähnlich wie beim Entwurf

  - Was muss eigentlich gezeigt werden? ➡ verlangt Klarheit über Spezifikation
  - Was ist bekannt?
- Was muss gezeigt werden:
  - ➡ Algorithmus sortiert **für jede gültige Eingabe** korrekt
- Auf welchen Annahmen kann aufgebaut werden?
  - ➡ Jeder Einzelschritt (Pseudocode oder Programmiersprache) wird gemäß seiner Spezifikation ausgeführt
  - Z.B. bewirkt $x \leftarrow x + 1$, dass Variable x um eins erhöht wird

Definition

- Man unterscheidet zwei Arten von „Korrektheit“

Definition 7.2: Partielle Korrektheit

Ein Algorithmus ist **partiell korrekt**, wenn er für eine spezifizierte erfüllte Vorbedingung (Q) bei einer **eventuellen Terminierung** eine spezifizierte Nachbedingung (R) erreicht; d.h. (R) ist nach Ausführung erfüllt.

Definition 7.3: Totale Korrektheit

Ein Algorithmus ist **total korrekt**, wenn er partiell korrekt ist und terminiert.

- Mit anderen Worten:
  - Ein partiell korrekter Algorithmus liefert das korrekte Ergebnis, *falls* er jemals fertig wird
  - Ein total korrekter Algorithmus liefert nach endlicher Zeit das korrekte Ergebnis

Beispiel „Sortieren“

- In Kapitel 1 wurde im Beispielalgorithmus 12 ein Sortieralgorithmus (Bubblesort) vorgestellt
- Die Korrektheit des Problems „Sortieren“ kann unabhängig vom Algorithmus ausgedrückt werden

Problem Sortieren

Eingabe: Folge von Elementen $(e_1, e_{2,\ldots,e_n})$ mit Ordnungsrelation („$\leq$“)

Ausgabe: Permutation $(e'_1, e'_2, \ldots, e'_n)$ von $(e_1, e_2, \ldots, e_n)$, so dass $e'_1 \leq e'_2 \leq \ldots \leq e'_n$

- **Beispiel**: Sortieren einer Sequenz natürlicher Zahlen
  - **Eingabe**: $8, 15, 3, 14, 7, 6, 18, 19$
  - **Ausgabe**: $3, 6, 7, 8, 14, 15, 18, 19$
- Der Bubblesort-Algorithmus (BSORT) sieht in Pseudocode z.B. so aus:

Algorithm BSORT

Require: $e_1, \ldots, e_n$
Ensure: $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

```
1: repeat
2:     changed ← false;
3:     for i ← 1, ..., n − 1 do
4:         if e_i > e_{i+1} then
5:             SWAP(e_i,e_{i+1})
6:             changed ← true
7:         end if
8:     end for
9: until changed = false
```

- Die partielle Korrektheit ist leicht zu beweisen:
  - Wenn BSORT terminiert, muss $changed = \text{false}$ gelten
  - $\rightsquigarrow$ für kein $i \in \{1, \ldots, n-1\}$ darf $e_i > e_{i+1}$ gelten
  - $\rightsquigarrow \forall i \in \{1, \ldots, n-1\}$, $e_i \leq e_{i+1}$ □
- Totale Korrektheit heben wir uns für später auf 🙂

Alternative Problemlösung: Insertion Sort

- Bubblesort ist nur eine mögliche Lösung für das Sortierproblem
- Betrachten Alternative: „Insertion Sort“ (INSSORT)

Algorithm INSSORT

Require: $e_1, \ldots, e_n$
Ensure: $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

```
1: for j ← 2, ..., n do
2:     key ← e_j
3:     i ← j − 1                          ▷ Verschiebe alle Elemente
4:     while (i > 0) ∧ (e_i > key) do     ▷   e_1, ..., e_{j−1}, die größer
5:         e_{i+1} ← e_i                  ▷   als key sind, nach rechts
6:         i ← i − 1
7:     end while
8:     e_{i+1} ← key                      ▷ Fülle die Lücke mit key
9: end for
```

- Intuitiv ist der Algorithmus „Insertion Sort“ vermutlich leicht zu verstehen
- Versuchen uns an einem Beweis der Korrektheit
- Benutzen dazu einige Lemmata

Lemma 1. *Wenn die Schleife 4 – 7, terminiert, ist entweder (a) $i = 0$ oder (b) $e_i \leq key$*

- **Beweis:**
  i startet mit einem Wert > 0 und wird schrittweise verkleinert, Rest folgt direkt aus der Schleifenbedingung

Lemma 2. *Wenn die Elemente $e_1, \ldots, e_{j-1}$ vor Eintritt in die Schleife 4 – 7 geordnet, so ist danach (a) die Sequenzen $e_1, \ldots, e_i$ und (b) die Sequenz $e_{i+1}, \ldots, e_j$ geordnet.*

- **Beweis:**
  (a) Die Sequenz $e_1, \ldots, e_i$ wird nicht geändert
  (b) Fallunterscheidung:
  - Die Schleife wird nicht betreten $\rightsquigarrow i = j - 1 \rightsquigarrow$ Sequenz $e_{i+1}, \ldots, e_j$ besteht aus einem Element e_j, ist also geordnet
  - Sonst: Entspricht der vorherigen Sequenz $e_i, \ldots, e_{j-1}$, die ja geordnet war

Merke:

Ordnung ist eine **Invariante** der Schleife!

Lemma 3. *Wenn vor dem Block 2 – 8 die Sequenz $e_1, \ldots, e_{j-1}$ sortiert war, ist anschließend die Sequenz $e_1, \ldots, e_j$ sortiert.*

- **Beweis:** Fallunterscheidung, i nach Schleife:

- $i = 0$: Laut Schleifenbedingung $\forall k \in \{1, \ldots, j-1\}, key < e_k$; außerdem laut Lemma 2 ist Sequenz $e_1, \ldots, e_j$ sortiert

- $i > 0, i < j-2$: Laut Lemma 1 ist $key \geq e_i$; außerdem laut Schleifenbedingung $\forall k \in \{i+2, j-1\}, key < e_k \rightsquigarrow e_1, \ldots, e_i, e_{i+1} = key, \ldots, e_j$ ist sortiert

- $i = j-1$: Schleife nicht durchlaufen $\rightsquigarrow key \geq e_{i-1} \rightsquigarrow e_1, \ldots, e_j$ ist sortiert

$key \geq e_i$ laut Lemma 1

Theorem 4. Insertion Sort *sortiert eine Sequenz.*

- **Beweis:** Induktion über j:

  - **IA**: $j = 1$ Einersequenz ist immer sortiert

  - **IS**: Wenn $e_1, \ldots, e_k$ sortiert ist, ist anschließend auch $e_1, \ldots, e_{k+1}$ sortiert laut Lemma 3

  - **Schlussfolgerung**: Theorem gilt für alle endlichen Sequenzen

Korrektheitskalküle

- Beweise, wie der angeführte Bubblesort-Beweis, sind *ad hoc*

- Es gibt spezielle Kalküle (**Kalkül ➡** formales System zum Ziehen logischer Schlüsse) für die Korrektheit von Programmen

- Beispiele

  - **Hoare-Kalkül** (auch: **Floyd-Hoare-Kalkül**)

  - **wp-Kalkül** (Edsger W.Dijkstra)

- Benutzung von Tripeln: {Vorbedingung} `Programmcode` {Nachbedingung}

- **Axiomatische** Regeln: $\frac{\text{Prämisse}}{\text{Folgerung}}$

- Beispiel **Kompositionsregel**: $\frac{\{Q\}\ P1\ \{R\} \qquad \{R\}\ P2\ \{S\}}{\{Q\}\ P1; P2\ \{S\}}$

- Mehr in LV „Verlässliche Systeme“

Terminierung

- **Partielle** Korrektheit wird bewiesen unter der Bedingung, dass der Code Terminiert
- Für **totale** Korrektheit muss also noch **Terminierung** bewiesen werden

Kritisch bei

- **Rekursion**
  - Die Rekursion muss nach einer endlichen Anzahl abbrechen, d.h. die Rekursionsbasis erreichen.
- **Schleifen**
  - Die Schleifenbedingung muss nach einer endlichen Anzahl von Iterationen *false* werden.
  - Es muss sichergestellt sein, dass auch der Schleifenrumpf terminiert (in jeder Iteration)

Terminierungsfunktion

- Zum Beweis der Terminerung einer Schleife muss eine **Terminierungsfunktion** τ angeben werden:

$$\tau : V \to \mathbb{N}$$

  - V ist eine Teilmenge der Ausdrücke über die Variablenwerte der Schleife
- Die Terminierungsfunktion muss folgende Eigenschaften besitzen:
  - Ihre Werte sind natürliche Zahlen (einschließlich 0)
  - Jede Ausführung des Schleifenrumpfs **verringert** ihren Wert (streng monoton fallend)
  - Die Schleifenbedingung ist *false*, wenn $\tau = 0$.

➡ τ ist obere Schranke für die noch ausstehende Anzahl von Schleifendurchläufen.

Terminierungsregel

- Ist eine Terminierungsfunktion bekannt, kann die **Terminierungsregel** genutzt werden

$$\frac{\{\tau = k\}P\{\tau < k\} \;\; \{\tau = 0\} \Rightarrow \neg B \;\; P \text{ terminiert}}{\texttt{while}(B)\, P;\ \{\text{Terminierung}\}}$$

- Mit anderen Worten: **wenn** die Terminierungsfunktion streng monoton fallend ist, **und** (spätestens) ihr Wert 0 ein Ende der Schleife erzwingt, **und** der Schleifenkörper terminiert, **dann** terminiert eine Schleife

Es ist also zu zeigen:

1. ***Streng monotones Fallen von*** τ
2. ***Die Implikation der Nichterfüllung der Schleifenbedingung*** B ***bei niedrigsten*** τ
3. ***Die Terminierung des Rumpfs*** P

Beispiel: Quadratzahl

```
1 /* Calculates the square of a nonnegative integer number a */
2
3 /*  {Q: 0 ≤ a} */
4 y= 0;
5 x= 0;
6 while (y != a) {
7   x= x+2*y+1;
8   y= y+1;
9 }
10 /* {R: x = a²} */
```

- Wähle als Terminierungsfunktion $\tau = a - y$:
  1. τ wird in jedem Durchgang dekrementiert, da y inkrementiert wird und a konstant bleibt.
  2. Wenn $\tau = 0$ folgt $y = a$, d.h. die Schleifenbedingung $y \neq a$ ist falsch.
  3. Schleifenrumph enthält keine Schleifen/Rekursionen/Gotos... Terminierung trivial

➡ Die Schleife terminiert

Terminierung bei Rekursion

- Beim Beweis der Terminierung von Rekursionen kann genauso wie bei Schleifen vorgegangen werden
- Auch hier wird eine Terminierungsfunktion τ konstruiert, die mit zunehmender Rekursionstiefe kleiner wird.
- Es muss gelten:
  1. Die Werte von sind natürliche Zahlen (inkl. 0)
  2. Bei jedem Aufruf der Methode (rechte Seite der Rekursionsgleichung) wird der Wert von echt kleiner.
  3. Abbruch bei (spätestens) $\tau = 0$ erzwungen
- Beispiel: **Fibonacci-Zahlen**

```
1 /* Computes Fibonacci n-th number */
2 int fib(int n)
3 {
4   if (n<3) return 1;
5   else return (fib(n-1)+fib(n-2));
6 }
```

- In diesem Fall können wir als Terminierungsfunktion $\tau = n$ wählen.

Probleme bei Terminierung

- Betrachten folgendes Beispiel:

Algorithm GOLDBACH

Require: $x = 4$
Ensure: $x > 4$

1: **while** $x =$ Summe zweier Primzahlen **do**
2: $\quad x \leftarrow x + 2$
3: **end while**

- Terminiert dieses Schleife?
- **Goldbachsche Vermutung**: Bis heute nicht allgemein bewiesen, aber bis 10^{18} geprüft

Achtung!

Beweise für Terminierung sind **nicht** immer möglich!

- Diese Aussage ist wiederum bewiesen (Terminierungproblem)

Aufgaben

Aufgabe 7.1

Betrachten Sie folgenden „Beweis“ und finden Sie den Fehler!

$$
\begin{aligned}
20 &= 20 && |\cdot -1\\
-20 &= -20\\
16-36 &= 25-45\\
4\cdot 4-4\cdot 9 &= 5\cdot 5-5\cdot 9\\
4\cdot 4-2\cdot 4\cdot \frac{9}{2} &= 5\cdot 5-2\cdot 5\cdot \frac{9}{2} && |x \overset{def}{=} \frac{9}{2}\\
4^2-2\cdot 4\cdot x &= 5^2-2\cdot 5\cdot x && |+x^2\\
4^2-2\cdot 4\cdot x+x^2 &= 5^2-2\cdot 5\cdot x+x^2 && |(a-b)^2=a^2-2ab+b^2\\
(4-x)^2 &= (5-x)^2 && |\sqrt{\ }\\
4-x &= 5-x && |+x\\
4 &= 5 && \square
\end{aligned}
$$

Aufgabe 7.2: Trinominos

Ein Quadrat setzt sich aus $2^n \cdot 2^n - 1$ kleinen weißen Quadraten ($n \in \mathbf{N}$) und einem schwarzen Quadrat zusammen (Beispiel mit $n = 3$ siehe linke Abbildung). Ein Trinomino-Stein ist ein L-förmiger Stein, der drei kleine Quadrate abdeckt (siehe rechte Abbildung).

Beweisen Sie, dass die weißen Quadrate vollständig mit Trinomino-Steinen abgedeckt werden können, ohne dass sich Steine überlappen, Steine aus dem großen Quadrat ragen oder das schwarze Quadrat abgedeckt wird!