Einführung in die Informatik

Algorithms

Eigenschaften von Algorithmen

Wolfram Burgard

Motivation und Einleitung

- In der Informatik sucht man im Normalfall nach **Verfahren zur Lösung von Problemen**.
- Eine zentrale Fragestellung ist wie man solche Verfahren beschreibt.
- Man ist meist daran interessiert **von einer konkreten Programmiersprache zu abstrahieren**,
- weil es **für ein- und dasselbe Problem unterschiedliche Programme mit verschiedenen Eigenschaften** (z.B. bezüglich Laufzeit) geben kann.
- Dabei stellt man an diese Verfahren noch bestimmte Anforderungen und bezeichnet sie als **Algorithmen**.
- **Programme** sind dann nur noch die **Umsetzung dieser Algorithmen in einer speziellen Programmiersprache**.

Der Begriff „Algorithmus“

- Der Begriff „Algorithmus“ geht auf den persischen Mathematiker und Astronomen Ibn Musa Al-Chwarismi zurück.
- Im 9. Jahrhundert hat er das Lehrbuch „Kitab al jabr w’ almuqabala“ („Regeln der Wiedereinsetzung und Reduktion“) geschrieben.
- Im folgenden werden wir verschiedene Fragestellungen untersuchen:
    1. Was sind Algorithmen?
    2. Wie erstellt man Algorithmen?
    3. Wie untersucht man Algorithmen?
    4. Wie beschreibt man Algorithmen?

Handlungsanweisungen

Im täglichen Leben begegnen uns **Handlungsanweisungen** aller Art, wie zum Beispiel die folgenden:

- Ärztliche Verordnung: Nimm dreimal täglich 15 Tropfen Asperix vor den Mahlzeiten.
- Waschanleitung: Bei 60 Grad waschen; Waschmittelzugabe in Abhängigkeit von der Wasserhärte nach Angaben des Herstellers.
- Einfahrvorschrift für Autos: Im 2. Gang nicht über 50 km/h, im 3. Gang nicht über 80 km/h, im 4. Gang nicht über 120 km/h; nach 1000 gefahrenen km Motor- und Getriebe Ölwechsel.
- Spielregel: ... bei einer 6 darf noch einmal gewürfelt werden ...
- Koch- oder Backrezepte
- ...

Aspekte von Handlungsanweisungen

Wir unterscheiden drei verschiedene Aspekte:

1. Der **Text einer Handlungsanweisung**,
2. der **Ausführende** und
3. die **Ausführung**.

Im Kontext der Informatik sind dies

1. der **Algorithmus**,
2. der **Prozessor** und
3. der **Prozess**.

Eigenschaften von Handlungsanweisungen

- Einzelne **Anweisungen** werden **stets in bestimmter Reihenfolge ausgeführt**.
- Diese kann mit der **textuellen Reihenfolge** der Beschreibung der Handlungsanweisungen übereinstimmen. Sie kann aber auch **von Bedingungen abhängig** gemacht werden.
- Bisweilen ist es auch **erlaubt**, **Handlungsanweisungen *nebenläufig***, d.h. nicht sequentiell oder nacheinander, sondern parallel oder gleichzeitig, auszuführen, d.h. die zeitliche Reihenfolge wird dann nicht festgelegt.
- Schließlich wird bei allen, auch bei Alltagsanweisungen, ein **Unterschied zwischen ihrer Beschreibung** im Text und den **Daten** gemacht.

Dies findet sich auch bei Algorithmen wieder.

Problematische Handlungsanweisungen (1)

1. Starte mit der Zahl 3.

2. Addiere 0,1.

3. Addiere 0,04.

4. Addiere 0,001.

5. Addiere 0,0005.

6. Addiere 0,00009.

7. ...

Diese Handlungsanweisung hat **keine endliche Länge**.

Problematische Handlungsanweisungen (2)

Zur Berechnung der dritten Wurzel einer Zahl x verfahre wie folgt:

1. Erfrage x.

2. Setze r auf 1.

3. Wiederhole

```
r := r - (r * r * r - x)/(3 * r * r);
```

Die **Ausführung** dieser Handlungsanweisung **hält nicht an**.

Handlungsanweisungen und Präzision

Aus einer Zubereitungsanleitung:

> „Die Suppe aus der Dose nach Vorschrift zubereiten. Sie können zum Schluss noch einige Spargelstückchen hinzugeben. Den Schinken in Streifen schneiden ...“

Aus der Spielanleitung zu „Hugo, das Schlossgespenst“:

> „... Der mutigste Spieler setzt zunächst einen seiner Gäste auf ein beliebiges freies Feld ... Wer von Euch jetzt schon zittert, darf beginnen.“

Anleitungen für Spiele oder Kochrezepte sind häufig **unpräzise**. Die **Ausführungsreihenfolge** einzelner Anweisungen ist nicht genau **festgelegt**. Auch **steht häufig nicht fest, wie begonnen wird**.

Allgemeinheit

Betrachten Sie die folgende Handlungsanweisung:

Um in das Glottertal zu kommen,

- verlassen Sie die Georges-Köhler-Allee und biegen Sie links ab.
- ...
- Fahren Sie dann geradeaus auf die B3 Richtung Emmendingen und
- biegen Sie hinter Gundelfingen auf die B294 Richtung Waldkirch ab.
- Nehmen Sie dann die erste Ausfahrt und biegen Sie rechts ab.
- ...

Diese **Handlungsanweisung** ist präzise, führt aber nur dann zum Ziel, wenn man in Freiburg an der Technischen Fakultät startet.

Sie ist **so spezifisch**, dass sie **nur ein einziges Problem löst**. Für einen Algorithmus **fehlt** ihr **die notwendige Allgemeinheit**.

Eine intuitive Definition des Algorithmenbegriffs

Definition: *Ein Algorithmus ist eine präzise, endliche Verarbeitungsvorschrift, die genau festlegt, wie die Instanzen einer Klasse von Problemen gelöst werden. Ein Algorithmus liefert eine Funktion (Abbildung), die festlegt, wie aus einer zulässigen Eingabe die Ausgabe ermittelt werden kann.*

Eigenschaften von Algorithmen (1)

Finitheit: Die Beschreibung des Verfahrens ist von endlicher Länge (*statische* Finitheit) und zu jedem Zeitpunkt der Abarbeitung des Algorithmus hat der Algorithmus nur endlich viele Ressourcen belegt (*dynamische* Finitheit).

Terminierung: Verarbeitungsvorschriften, die nach Durchführung endlich vieler Schritte (Operationen) zum Stillstand kommen, heißen *terminierend*.

In der Informatik spielen aber auch viele nichtterminierende Programme eine große Rolle. Sie werden beispielsweise zur Prozessteuerung, Datenübertragung in Netzen und Mensch-Maschine Kommunikation benutzt. Man spricht in diesem Kontext auch von reaktiven Systemen.

Eigenschaften von Algorithmen (2)

Effektivität: Die Wirkung einer einzelnen Anweisung eines Algorithmus ist eindeutig festgelegt.

Determinismus: Liegt die Reihenfolge, in der die einzelnen Schritte einer Verarbeitungsvorschriften ausgeführt werden, eindeutig fest, hängt sie also nur von den Eingabedaten ab, so spricht man von *deterministischen* Algorithmen.

Daneben spielen in der Theorie auch *nicht-deterministische* und in der Praxis zunehmend auch *randomisierte*, d.h. von einem zufälligen Ereignis abhängige, Verarbeitungsvorschriften eine Rolle.

Determinismus und Determiniertheit

Determinismus: Liegt die Reihenfolge, in der die einzelnen Schritte einer Verarbeitungsvorschriften ausgeführt werden, eindeutig fest, hängt sie also nur von den Eingabedaten ab, so spricht man von *deterministischen* Algorithmen.

Determiniertheit: Verarbeitungsvorschriften sind *determiniert*, wenn sie bei gleichen Parametern und Startwert stets das gleiche Resultat liefern.

Bemerkung 1: Jede deterministische Verarbeitungsvorschrift ist auch determiniert. Jedoch ist nicht jede determinierte Verarbeitungsvorschrift auch deterministisch.

Bemerkung 2: Es gibt auch randomisierte Verfahren, bei denen das Ergebnis zu einem gewissen Grad auf Zufall beruht. Bei diesen Verfahren ist jedoch die Nachvollziehbarkeit gegeben, d.h. bei gleichen Parametern für den Zufall gibt es den gleichen Output.

Wie beschreibt man Algorithmen?

- Es gibt eine Vielzahl von Techniken, um Algorithmen zu beschreiben.
- Hierzu gehören beispielsweise umgangssprachliche Formulierungen, spezielle, abstrakte Maschinenmodelle, wie z.B. Register- oder Turingmaschinen, aber auch spezielle Sprachen.
- Im folgenden wollen wir zunächst ausgehen von einer Formulierung durch so genannte **`while`-Programme**.

`While`-Programme (1)

- `While`-Programme erlauben es **Variablen** zu verwenden und darin Werte oder Zwischenergebnisse abzuspeichern.
- Wir verwenden beispielsweise `a, b, x, wert`, ... zur Bezeichnung von Speicherzellen, die beliebige (ganze) Zahlen aufnehmen können.
- Neben Variablen die Zahlen aufnehmen können gibt es auch so genannte **boolesche Variablen**. Eine boolesche Variable hat nur die **beiden Zustände wahr und falsch** (`true` und `false`). Meist benutzt man boolesche Ausdrücke um Bedingungen zu modellieren.

`While`-Programme (2)

Elementare Anweisungen: Sie sind entweder die **leere Anweisung** (`skip`) oder die **Zuweisung**, die es erlaubt, eine Speicherzelle `x` mit dem Wert eines arithmetischen Ausdrucks zu füllen:

```
x := t
```

Dabei ist also `t` ein aus Variablen, Operatoren (wie t, *, /, ...), Funktionszeichen und Klammern zusammengesetzter Ausdruck.

```
x := (y + 17) * 3

u := (u + x) / 2 * x

B := (x > 5)
```

Eine Wertzuweisung `u := t` setzt `u` auf den Wert von `t` und terminiert anschließend.

While-Programme (3)

Komposition: Bei der (sequentiellen) **Komposition** `s1;s2` zweier Programme `s1` und `s2` wird zunächst `s1` und nach der Terminierung von `s1` dann `s2` ausgeführt.

Selektion: Die Selektion ist eine **bedingte Anweisung** der Form

```
if B then s1 else s2 end
```

Zunächst wird der boolesche Ausdruck `B` ausgewertet. Wenn `B` wahr ist, wird `s1` ausgeführt, andernfalls `s2` ausgeführt.

Iteration: Die Ausführung einer **Schleife**

```
while B do s1 end
```

beginnt mit der Auswertung des booleschen Ausdrucks `B`. Wenn `B` falsch ist, terminiert die Schleife sofort. Ist `B` wahr, wird `s1` ausgeführt. Nachdem `s1` terminiert, wird der ganze Vorgang wiederholt.

Beispiel (1)

```
x := 23;
y := 17;
while x != 0 do
   x := x - 1;
   y := y + 1;
end
```

Am Ende der Ausführung dieses Algorithmus steht in der Variablen `y` der Wert `40 = 23+17`.

Beispiel (2)

```
wert := 1;
while n > 0 do
    wert := 2 * wert;
    n := n - 1;
end
```

Das Programmstück berechnet offenbar 2^n, falls anfangs $n > 0$ war.

Offensichtlich gibt es zwischen `while`- und `Java`–Programmen eine starke Ähnlichkeit.

Genau genommen sind **`while`-Programme** eine **Untermenge von Java**.

Was können Computer berechnen?

- Programme sind die Umsetzung von Algorithmen in einer Programmiersprache.
- Java-Programme erfüllen oft Eigenschaften von Algorithmen, sie haben eine endliche Länge und sie sind präzise formuliert.
- Darüber hinaus haben wir festgelegt, dass Algorithmen eine Funktion realisieren.
- Es stellt sich nun die Frage, was Algorithmen eigentlich alles berechnen können, oder bezogen auf Computerprogramme, für welche Probleme man ein Programm entwickeln kann.
- Die Aussage „jedes Problem ist lösbar“ ist leider nicht auf die Informatik übertragbar.
- Vielmehr gilt eher das Gegenteil, d.h. „fast nichts“ ist mit Computern lösbar.

Warum ist fast nichts berechenbar?

Die Aussage, dass „fast nichts“ berechenbar, also mit Computern lösbar ist, ergibt sich nun aus den folgenden zwei Teilaussagen:

1. Die **Menge der Algorithmen ist abzählbar**.

2. Es gibt **überabzählbar viele Funktionen mit Argumenten und Werten im Bereich der natürlichen Zahlen**.

Warum ist die Menge der Algorithmen abzählbar?

Dass die **Menge der Algorithmen abzählbar** ist, folgt einfach daraus, dass **jedes** `while`**-Programm durch einen endlichen Text beschrieben sein muss**.

Wir können nun die **Texte der** `while`**-Programme zunächst der Länge nach und Texte gleicher Länge lexikographisch ordnen**; das liefert uns dann eine Aufzählung der `while`-Programme und damit aller Algorithmen.

Hinweis: Entsprechendes gilt natürlich auch für Java-Programme. Bei 256 verschiedenen Zeichen gibt es 256^{10} mögliche Texte der Länge 10, wobei allerdings nur wenige davon gültige Java-Programme sind. Die Aufzählung aller gültigen Java-Programme der Länge 10 mit einem naiven Verfahren würde somit zwar lange dauern, aber sie wäre möglich.

Der Cantorsche Diagonalschluss

Wir müssen nun noch zeigen, dass es überabzählbar viele Funktionen $f : \mathcal{N} \to \mathcal{N}$ gibt.

Dazu nehmen wir an $f_1, f_2, f_3, \ldots$ sei eine Aufzählung aller totalen Funktionen von den natürlichen Zahlen in sich. Dann definieren wir eine neue Funktion $f : \mathcal{N} \to \mathcal{N}$ wie folgt:

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

Da f total ist, muss sie an irgendeiner Stelle in der oben genannten Aufzählung $f_1, f_2, f_3, \ldots$ vorkommen, d.h.: $\exists k \forall x : f(x) = f_k(x)$. Dann gilt insbesondere für das Argument $x = k$:

$$f(k) = f_k(k) = f_k(k) + 1$$

Dieser Widerspruch kann offensichtlich nur dadurch aufgelöst werden, dass die Annahme der Abzählbarkeit falsch ist.

Berechenbarkeit

- Die Menge der totalen Funktionen von den natürlichen Zahlen in die natürlichen Zahlen ist nicht abzählbar.
- Da die Menge der Algorithmen abzählbar ist, muss es deutlich mehr Funktionen als Algorithmen geben.
- Dies ist natürlich eine reine Existenzaussage.
- Sie liefert uns noch kein einziges Beispiel eines ganz konkreten Problems, das mit Hilfe von Rechnern nicht oder präziser **prinzipiell nicht lösbar** ist.
- Eine Funktion, für die es keinen Algorithmus gibt, heißt **nicht berechenbar**.
- Wie sieht nun eine solche, nicht berechenbare Funktion aus?

Das Halteproblem

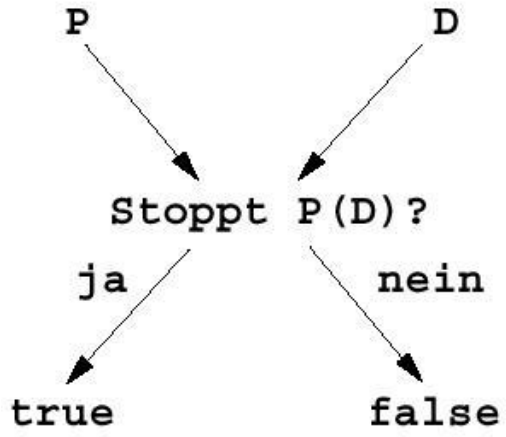
- Eines der größten Probleme bei Softwareerstellung sind Programmfehler (und insbesondere Laufzeitfehler).
- Programmfehler können enorme Kosten verursachen.
- Mögliche Folgen solcher Fehler (wie z.B. der Jahr-2000-Fehler) sind
  - falschen Ergebnissen (Rentenbescheide für Säuglinge),
  - unvorhergesehenen Programmabbrüchen (Ausnahmefehler) oder gar
  - Endlosschleifen.
- Es wäre natürlich naheliegend mithilfe von Computerprogrammen zu prüfen, ob ein gegebenes Programm einen solchen Fehler enthält.
- Wir betrachten hier das dritte Problem, d.h. die Frage, ob es ein Programm gibt, dass für ein beliebiges anderes Programm entscheidet, ob es für eine bestimmte Eingabe in eine Endlosschleife gerät oder nicht.
- Dieses Problem heißt das **Halteproblem**.

Das Halteproblem

- Die Lösung des Halteproblems wäre von großem wirtschaftlichen Nutzen:
  - Es könnte viel Rechenzeit gespart werden.
  - Die Anwender wären zufriedener, weil sie sich nie mehr fragen müssten, „ob da jetzt noch einmal etwas passiert“.
- Man könnte daher vermuten, dass weltweit zahlreiche Menschen daran arbeiten, ein Programm zur Lösung des Halteproblems zu entwickeln.
- Falls das so wäre, würden sie ihre Arbeitszeit verschwenden, denn das **Halteproblem ist unentscheidbar**, d.h. es kann kein entsprechendes Programm geben.
- Anders ausgedrückt: **Die Funktion, die `true` ausgibt, falls ein beliebiges Programm für eine gegebene Eingabe hält, und `false`, falls nicht, ist nicht berechenbar**.

Beweis der Unentscheidbarkeit des Halteproblems

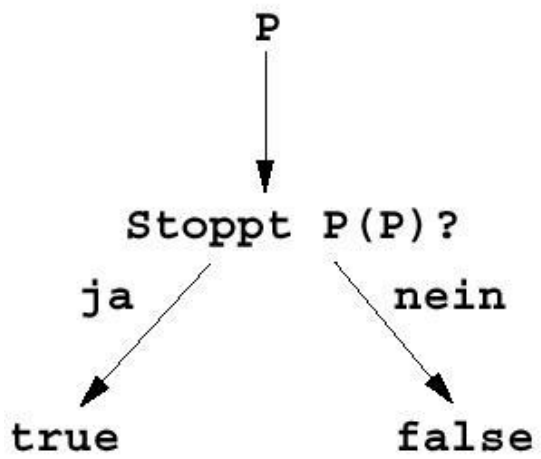
- Nehmen wir für einen Moment an, wir hätten ein Programm `Stopp-Tester` zur Lösung des Halteproblems.
- `Stopp-Tester` hat offensichtlich zwei Eingaben: Das zu überprüfende Programm `P` sowie die Eingabe `D` für `P`.
- `Stopp-Tester(P, D)` soll nun den Wert `true` ausgeben, wenn `P`, ausgeführt mit den Eingabedaten `D` halten würde. Andernfalls soll es `false` zurückliefern.



Spezialisierung von `Stopp-Tester`

- Unser Programm `Stopp-Tester` ist uns etwas zu allgemein, denn wir können es mit beliebigen Eingabedaten `D` für `P` aufrufen.
- Wir betrachten stattdessen das Programm `Stopp-Tester-Neu`, das als Eingabedaten für `P` den Programmtext von `P` selbst verwendet.
- D.h. `Stopp-Tester-Neu` hätte folgende Implementierung.

```
boolean Stopp-Tester-Neu(String P){
    return Stopp-Tester(P, P);
}
```



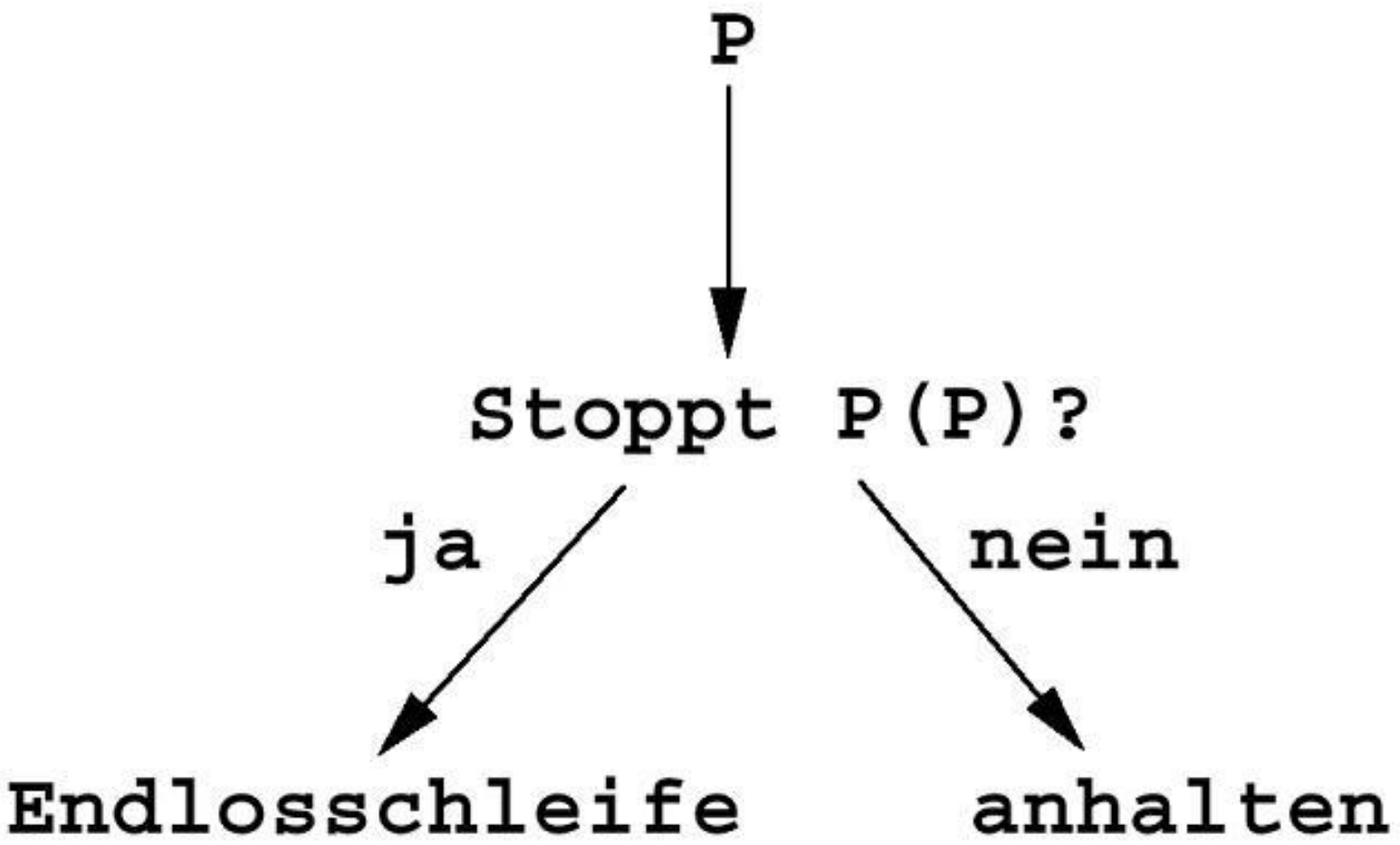
Das Programm `Spassig`

- Offensichtlich existiert `Stopp-Tester-Neu`, wenn das Programm `Stopp-Tester` existiert.
- Wir nutzen jetzt `Stopp-Tester-Neu` um ein weiteres Programm zu schreiben, welches wir `Spaßig` nennen.

```
void Spassig(String P){
    if (Stopp-Tester-Neu(P))
        while (true);
    else
        return;
}
```

- Für `Spaßig` gilt somit, dass es in eine **Endlosschleife** gerät, wenn **`P` angewendet auf sich selbst (seinen eigenen Text) anhält**.
- **Hält `P`** hingegen **nicht**, **hält** dafür **`Spassig`**.

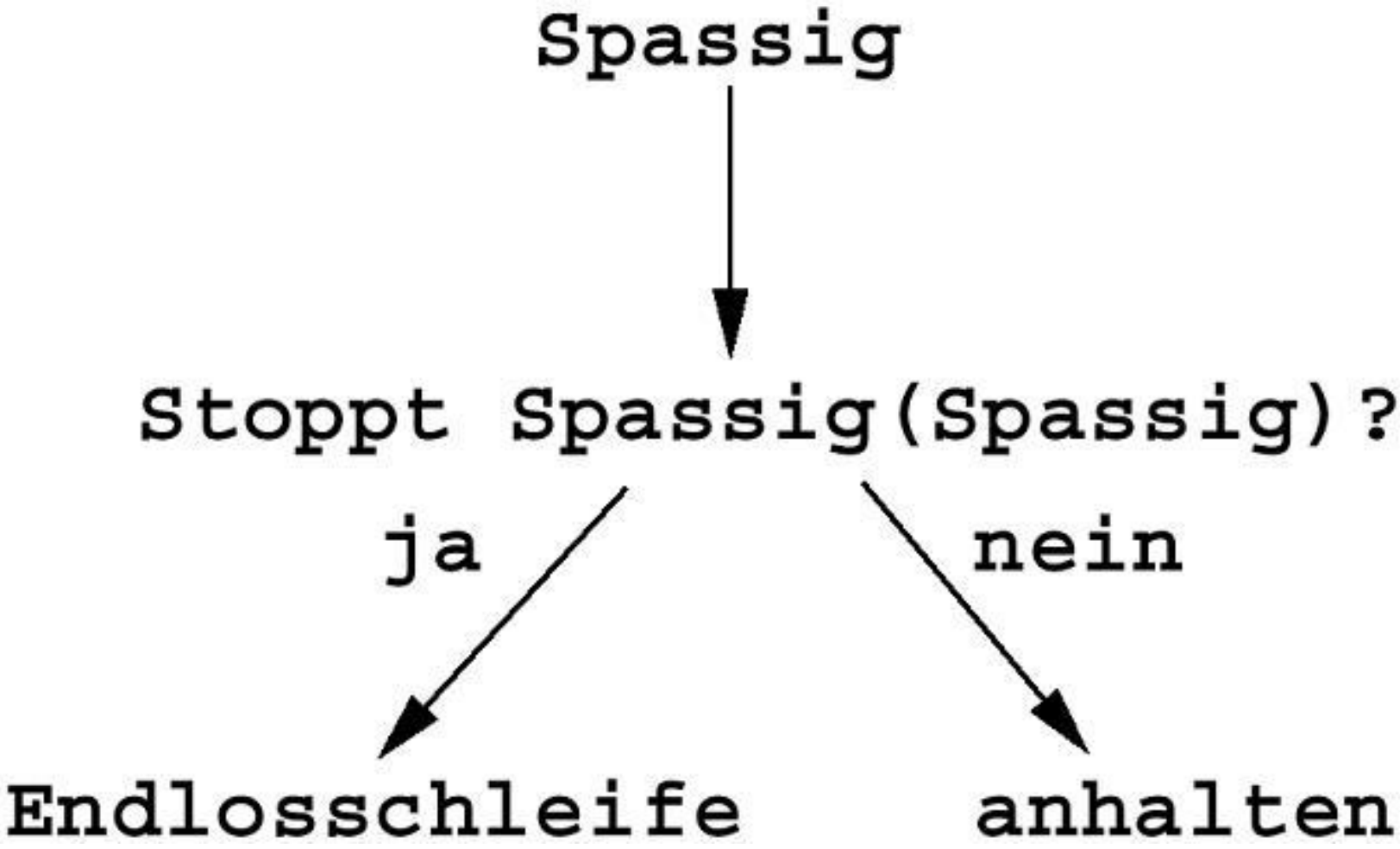
Arbeitsweise des Programms `Spassig`



Herbeiführen des Widerspruchs

- Um jetzt den Beweis durchzuführen, untersuchen wir, was passiert, wenn wir **`Spassig` auf sich selbst anwenden**. Dies ist zulässig, weil `Spassig` selbst wieder ein Programm ist.
- Dies würde aber bedeuten:
  - **Wenn `Spassig(Spassig)` anhält**, **dann gerät `Spassig`** angewendet auf sich selbst **in eine Endlosschleife.**
  - Andernfalls, **wenn `Spassig(Spassig)` nicht anhält**, **stoppt `Spaßig`** angewendet auf sich selbst.
- Beides ist ein **Widerspruch**, der nur dadurch aufgelöst werden kann, dass **`Spassig`** und damit auch **`Stopp-Tester` nicht existieren** kann.

Anwendung von `Spaßig` auf sich selbst



Prinzip dieses Beweises

Die Vorgehensweise ist folgendermaßen:

1. Annehmen, dass ein Programm `StoppTester` geschrieben werden kann.
2. Dieses Programm nutzten, um ein neues Programm `Spassig` zu schreiben.
3. Zeigen, dass `Spassig` eine undenkbare Eigenschaft hat (es kann weder halten noch endlos laufen).
4. Folgern, dass die Annahme in Schritt 1 falsch gewesen sein muss.

Das Halteproblem ist somit leider unentscheidbar.

Weitere unlösbare Probleme

- Natürlich gibt es einige einfache Programme, für die einfach zu entscheiden ist, ob sie für beliebige Eingaben terminieren.
- Der oben angegebene Beweis sagt lediglich, dass es **im Allgemeinen unentscheidbar** ist.
- Ein bekanntes Beispiel für unentscheidbare Probleme aus der Mathematik sind die Frage der Lösbarkeit diophantischer Gleichungen.
- Eine diophantische Gleichung ist eine Gleichung der Form $P(x, y, ...) = 0$ wobei P ein Polynom mit ganzzahligen Koeffizienten über den Variablen $x, y, ...$ ist und man sich für die ganzzahligen Lösungen interessiert.
- Ein typisches Beispiel für eine diophantische Gleichung ist

$$7x^2 - 5xy - 3y^2 + 2x + 4y - 11 = 0$$

Weitere nicht berechenbare Probleme

- Mit dem Halteproblem haben wir ein konkretes Problem kennengelernt, für das es keine algorithmische Lösung geben kann.
- Im Folgenden werden wir untersuchen, welche weiteren nicht-berechenbaren Probleme es gibt.
- Allerdings war die Beweismethode per Diagonalschluss für das Halteproblem schwer zu verstehen.
- Eine klassische Methode in der Informatik ist nun, die Unentscheidbarkeit des Halteproblems zu nutzen, um die Unentscheidbarkeit eines gegebenen Problems zu beweisen.
- Diese Technik nennt man **Reduktion des Halteproblems**.

Das Totalitätsproblem und Reduktionsbeweise

- Das Halteproblem ist die Frage, ob ein gegebenes Programm `P` angesetzt auf eine gegebene Eingabe `D` hält oder nicht.
- Das Totalitätsproblem ist die Frage, ob ein gegebenes Programm `P` für alle Eingaben `D` anhält.
- In diesem Fall sagen wir, `P` ist total.
- Um die Unentscheidbarkeit des Totalitätsproblems zu beweisen, **reduzieren wir das Halteproblem auf das Totalitätsproblem**.
- Die Idee einer solchen Reduktion ist zu zeigen, dass wir das Halteproblem lösen könnten, wenn wir das gerade betrachtete Problem (hier also das Totalitätsproblem) lösen könnten.

Reduktion des Halteproblems auf das Totalitätsproblem (1)

- Um zu zeigen, dass das Totalitätsproblem unentscheidbar ist, nehmen wir an, dass es ein Programm `Total-Tester` gäbe, welches entscheiden kann, ob ein Programm total ist, d.h. für alle Eingaben anhält.
- Wir nutzen dieses Programm `Total-Tester` nun, um für beliebige Programme `P` und `D` das Halteproblem zu lösen.
- Angenommen, wir hätten ein Programm `P` und gegebene Daten `D`, für die wir das Halteproblem lösen müssten. Wir konstruieren für `P` und `D` nun das folgende Programm:

```
void SpassigPD(String D1){
    wendePaufDan(P, D);
}
```

Reduktion des Halteproblems auf das Totalitätsproblem (2)

- Dieses Programm ignoriert die Eingabe `D1` und wendet stattdessen `P` auf die Eingabe D.
- Offensichtlich gilt, dass `SpassigPD` genau dann für alle Eingaben `D1` anhält, wenn `P` angesetzt auf `D` hält.
- `Total-Tester` wäre somit in der Lage, das Halteproblem für beliebige `P` und `D` zu lösen.
- Dies ist jedoch ein Widerspruch zur Unentscheidbarkeit des Halteproblems.

Das Äquivalenzproblem

- Beim Äquivalenzproblem geht es um die Frage, ob zwei Programme für die gleiche Eingabe stets die gleiche Ausgabe liefern, d.h. ob sie dieselbe Funktion berechnen.
- Auch bei diesem Problem verwendet man wie beim Totalitätsproblem eine Reduktion des Halteproblems.
- Wir nehmen wie oben an, dass wir ein Programm `Äquivalenz-Test` haben, das uns zwei Programme auf Äquivalenz testen kann.
- Betrachten wir nun das folgende Programm, welches den konstanten Wert `13` ausgibt:

```
void Thirteen(String D){
    System.out.println(13);
}
```

Die Reduktion

- Um die Unentscheidbarkeit des Äquivalenzproblems zu zeigen, konstruieren wir ein Programm `ThirteenPD`, welches genau dann `13` ausgibt, wenn ein beliebiges, von uns gewähltes Programm `P` angesetzt auf die Daten `D` hält.

```
void ThirteenPD(String D1){
    wendePaufDan(P, D);
    System.out.println(13);
}
```

- Offensichtlich gilt, dass beide Programme genau dann äquivalent sind, wenn `P` angesetzt auf `D` anhält.
- Damit müsste `Äquivalenz-Test` das Halteproblem lösen können, was im Widerspruch zur Unentscheidbarkeit des Halteproblems steht.

Rice's Theorem

- Wir haben oben gelernt, dass es verschiedene **Probleme** gibt, die **mit Algorithmen nicht lösbar** sind.
- Es scheint sogar so zu sein, dass **nahezu alle interessanten Eigenschaften über Algorithmen unentscheidbar** sind.
- **Rice's Theorem** bestätigt dies.
- Es besagt, dass **alle nicht trivialen** (nicht einfachen) **Eigenschaften von Algorithmen unentscheidbar** sind.
- Als trivial gelten beispielsweise Eigenschaften wie,
  - ob der Algorithmus eine bestimmte Länge hat oder
  - ob er eine bestimmte Zeichenkette enthält.
- Nicht-trivial sind hingegen die Eigenschaften, ob er eine bestimmte Ausgabe erzeugt, ob eine bestimmte Anweisung ausgeführt wird etc.

Korrektheit

- Oben haben wir diskutiert, dass es **Probleme** gibt, die sich **im allgemeinen einer Lösung durch Algorithmen entziehen**.
- Interessanterweise waren dies gerade **wichtige Eigenschaften** von Programmen, wie z.B. die **Terminierung** oder die **Äquivalenz**.
- Dennoch gibt es einige **Techniken**, mit deren Hilfe man z.B. die **Korrektheit von Algorithmen untersuchen** kann.
- In der Praxis ist man an verschiedenen Eigenschaften eines gegebenen Verfahrens interessiert:

  Terminierung: Hier will man zeigen, dass das Verfahren immer anhält.

  Partielle Korrektheit: Dies bedeutet, dass ein Verfahren, sofern es anhält, immer das korrekte Ergebnis liefert.

  Algorithmen, die beide Eigenschaften erfüllen, heißen **total korrekt**.

Induktion zum Beweis der Korrektheit

Betrachten wir erneut das Verfahren zur Berechnung der Zweierpotenz 2^n.

```
erg := 1;                  // 1
while n > 0 do             // 2
    erg := 2 * erg;        // 3
    n := n - 1;            // 4
    skip;                  // 5
end                        // 6
print erg;                 // 7
```

Im folgenden wollen wir **Induktion** als eine Möglichkeit betrachten, die Korrektheit dieses Algorithmus für alle Eingaben $n \geq 0$ zu beweisen.

Prinzip des Vorgehens

Um die Korrektheit zu beweisen, betrachten wir Zeile 5.

1. Sofern Zeile 5 das erste Mal erreicht ist, gilt `erg = 2 =` 2^1.
2. Beim `i+1`-ten erreichen von Zeile 5 hingegen ist `erg = 2 * 2^i =` 2^{i+1}.
3. Deshalb ist, sofern die Schleife `n` Mal durchlaufen ist, der Wert von `erg` am Ende genau 2^n.
4. Wird die Schleife überhaupt nicht durchlaufen, d.h. ist `n = 0`, so gilt `erg = 1 =` 2^0.
5. Sobald Zeile 7 erreicht wird, hat `erg` somit den Wert von 2^n.

Der Induktionsbeweis

Prinzip eines Induktionsbeweises:

- Ziel eines Induktionsbeweises ist es, eine Aussage (die **Induktionsannahme**) für eine Menge von Fällen zu zeigen.
- Dabei startet man mit einem besonderen Fall (dem **Induktionsanfang**), für den man die Aussage beweist.
- Danach zeigt man im **Induktionsschluss**, dass die Behauptung auch für alle weiteren Fälle gilt, sofern sie nur für den besonderen Fall gilt.

In unserem Beispiel:

- Die Induktionsannahme ist, dass `erg` bei Erreichen von Zeile 5 im `i`-ten Durchlauf immer den Wert 2^i hat.
- Schritt 1 ist der Induktionsanfang, da der Anfangsfall bewiesen wird.
- Im Induktionsschluss (Schritt 2) schließen wir dann vom Anfangsfall auf alle weiteren Fälle.

Anwendung der Induktionstechnik

Induktionsannahme: Mit dem Erreichen der fünften Zeile im `i`-ten Durchlauf hat `erg` den Wert 2^i.

Induktionsanfang: Ist `i = 1`, d.h. wird Zeile 5 zum ersten Mal erreicht, hat `erg` den Wert $2^i = 2$.

Induktionsschluss: Angenommen die Induktionsannahme ist wahr für irgendein `i`. Angenommen wir erreichen Zeile 5 zum `i+1`-ten Mal. In Zeile 3 wird `erg` dann der Wert `2 * 2^i = 2^(i+1)` zugewiesen. Die Behauptung gilt damit auch für `i+1`.

Demnach gilt die Behauptung für alle `n > 0`, denn wenn die Schleife nach `n` Durchläufen verlassen wird, hat `erg` den Wert 2^n.

Für den Fall `n = 0` hat `erg` den Wert $1 = 2^0$.

Daraus folgt die **partielle Korrektheit** des Algorithmus.

Nachweis der totalen Korrektheit

- Um die totale Korrektheit zu beweisen, müssen wir noch nachweisen, dass der Algorithmus für alle Eingaben `n >= 0` anhält.
- Da in jedem Durchlauf `n` in Zeile 4 um eins verringert wird, wird die Schleife genau `n` Mal durchlaufen.
- Falls `n = 0`, wird die Schleife nicht durchlaufen.
- Insgesamt terminiert der Algorithmus also nach `n` Schleifendurchläufen.
- Damit ist gezeigt, dass der Algorithmus total korrekt ist, da er für alle zulässigen Eingaben `n >= 0` terminiert und auch partiell korrekt ist.

Beweis der Korrektheit des Algorithmus von Euklid zur Berechnung des GgT

Um den größten gemeinsamen Teiler von zwei positiven ganzen Zahlen x und y zu berechnen, verfahre wie folgt:

```
while y != 0 do    // 1
    r := x % y;    // 2
    x := y;        // 3
    y := r;        // 4
    skip;          // 5
end                // 6
print x;           // 7
```

Im folgenden wollen wir nun die totale Korrektheit dieses Algorithmus beweisen.

Die Induktionsbehauptung

Der Schlüssel für diesen Beweis liegt in der Tatsache, dass der größte gemeinsame Teiler der aktuellen Werte von `x` und `y` stets gleich `g` ist, wobei `g` dem gesuchten `GgT` entspricht.

Induktionsbehauptung: Die **Induktionsbehauptung** ist, dass immer, wenn Zeile 5 erreicht wird, der `GgT(x, y)` dem GgT der ursprünglichen Argumente entspricht.

Der Induktionsanfang

- In jedem Schritt ersetzen wir `x` durch den Wert von `y` und `y` durch den Wert von `x % y`.
- Um den **Induktionsanfang** zu beweisen, müssen wir zeigen, dass der GgT von x und y dem GgT von y und x `mod` y entspricht, wobei `mod` der Modulo-Operator ist.
- Hierzu beweisen wir zwei Aussagen:
  1. Jeder Teiler von `x` und `y` ist auch Teiler von y und `x mod y`.
  2. Jeder Teiler von `y` und `x mod y` ist auch Teiler von x.
- Dabei verwenden wir die Notation `z | y` um auszudrücken, dass `z` ein Teiler von `y` ist.

Schritt 1

- Wir müssen zeigen, dass `z | x` und `z | y` die Aussage `z | (x mod y)` impliziert.
- Aus `z | x` und `z | y` folgt, dass es Zahlen `a` und `b` gibt mit `x = az` und `y = bz`.
- Nach Definition der Division mit Rest `mod` folgt, dass es ein c gibt mit `x mod y = x − cy`, d.h. `x = x mod y + cy`.
- Einsetzen von `x = az` und `y = bz` ergibt: `az = cbz + x mod y`.
- Also folgt `(a − cb)z = (x mod y)`, d.h. `z | (x mod y)`.

Schritt 2

- Nehmen wir umgekehrt an, dass `z` ein Teiler von `y` und `x mod y` ist.
- Weil `x = cy + (x mod y)` und `z|cy` und `z|(x mod y)` gilt automatisch, dass `z|x` .

Da nun jeder Teiler von `x` und `y` ein Teiler von `y` und `x mod y` ist und umgekehrt, muss der GgT innerhalb der `while`-Schleife erhalten bleiben.

D.h. nach dem ersten Erreichen von Zeile 5, haben `x` und `y` den selben GgT wie die Eingabewerte.

Der Induktionsschluss

- Oben haben wir gezeigt, dass `x` und y den selben GgT haben wie `y` und `x mod y`.
- Da in jeder Runde `x` den Wert von `y` und `y` den Wert von `x % y` bekommt, ist der GgT im (`i+1`)-ten Durchlauf der selbe wie im i-ten Durchlauf.
- Wenn die Schleife abbricht, ist `y == 0`. Daher muss beim vorangegangenen Schleifendurchlauf `y` ein Teiler von `x` gewesen sein. Diesen vorangegangenen Wert von `y`, der dem GgT der Ausgangswerte entspricht, hat jetzt `x`.

Sofern das Verfahren terminiert, wird am Ende der GgT der Eingabewerte ausgegeben.

Der Algorithmus von Euklid ist somit **partiell korrekt**.

Totale Korrektheit des Algorithmus von Euklid

- Um die totale Korrektheit des Algorithmus von Euklid zu zeigen, müssen wir nachweisen, dass die `while`-Schleife terminiert, d.h. dass am Ende die Bedingung `y==0` erreicht wird.
- In jeder Runde wird `y` auf den Wert von `r = x % y` gesetzt.
- Aufgrund der Definition des Rests wissen wir aber, dass dieser niemals negativ ist und auch immer kleiner als der Wert von `y` ist.
- Damit wird `y` in jeder Runde um wenigstens `1` kleiner.
- Da `y` niemals negativ werden kann, ist die Terminierung somit gesichert.
- Der **Algorithmus von Euklid ist total korrekt**.

Verbleibende Probleme

- Im vorangegangenen Teil haben wir gesehen, dass eine **Vielzahl von essentiellen Problemen der Informatik** sich der **Lösung durch Algorithmen prinzipiell entziehen**.
- Auf der anderen Seite haben wir gezeigt, dass **es** häufig **gelingt, für bestimmte Algorithmen den Beweis der totalen Korrektheit zumindest manuell zu führen**.
- **In manchen Fällen ist allerdings der Beweis der Korrektheit schwer zu führen**.
- Dies kann man so interpretieren, dass hier „routinierte Plackerei“ in Form von Algorithmen nicht ausreicht sondern eher „ein **hohes Maß an Kreativität**“ erforderlich ist.

Beispiel: Die Fermat ‘sche Vermutung

- Ein typisches Beispiel für einen schweren Beweis ist die Fermat ‘sche Vermutung, die besagt, dass es für $n>2$ keine positiven natürlichen Zahlen a, b und c mit $a^n+b^n=c^n$ gibt.
- Sie betrifft somit eine bestimmte Form der diophantischen Gleichungen mit

$$P(a,b,c) = a^n+b^n-c^n$$

  und wurde kürzlich, nach über 350 Jahren bestätigt.
- D.h. wir mussten sehr lange warten auf den Beweis, dass unser Algorithmus für die Suche nach einer Lösung für Diophantische Gleichungen mit $P(a,b,c) = a^n+b^n-c^n$ für beliebige $n>2$ nicht anhält.

Automatisierung von Beweisen

- Allerdings stellt die Informatik auch Techniken zur Verfügung, die in der Lage sind, zumindest in bestimmten Fällen den Beweis der totalen Korrektheit zu führen.
- Eine solche Technik für Korrektheitsbeweise, auf die wir hier allerdings nicht näher eingehen, ist das Hoare-Kalkül.
- Dabei stellt man logische Formeln über das Ein-/Ausgabeverhalten von Algorithmen auf.
- Dann kann man z.B. automatische Beweiser verwenden, d.h. Programme / Programmiersprachen (wie z.B. Prolog), die in der Lage sind, Beweise zu führen.

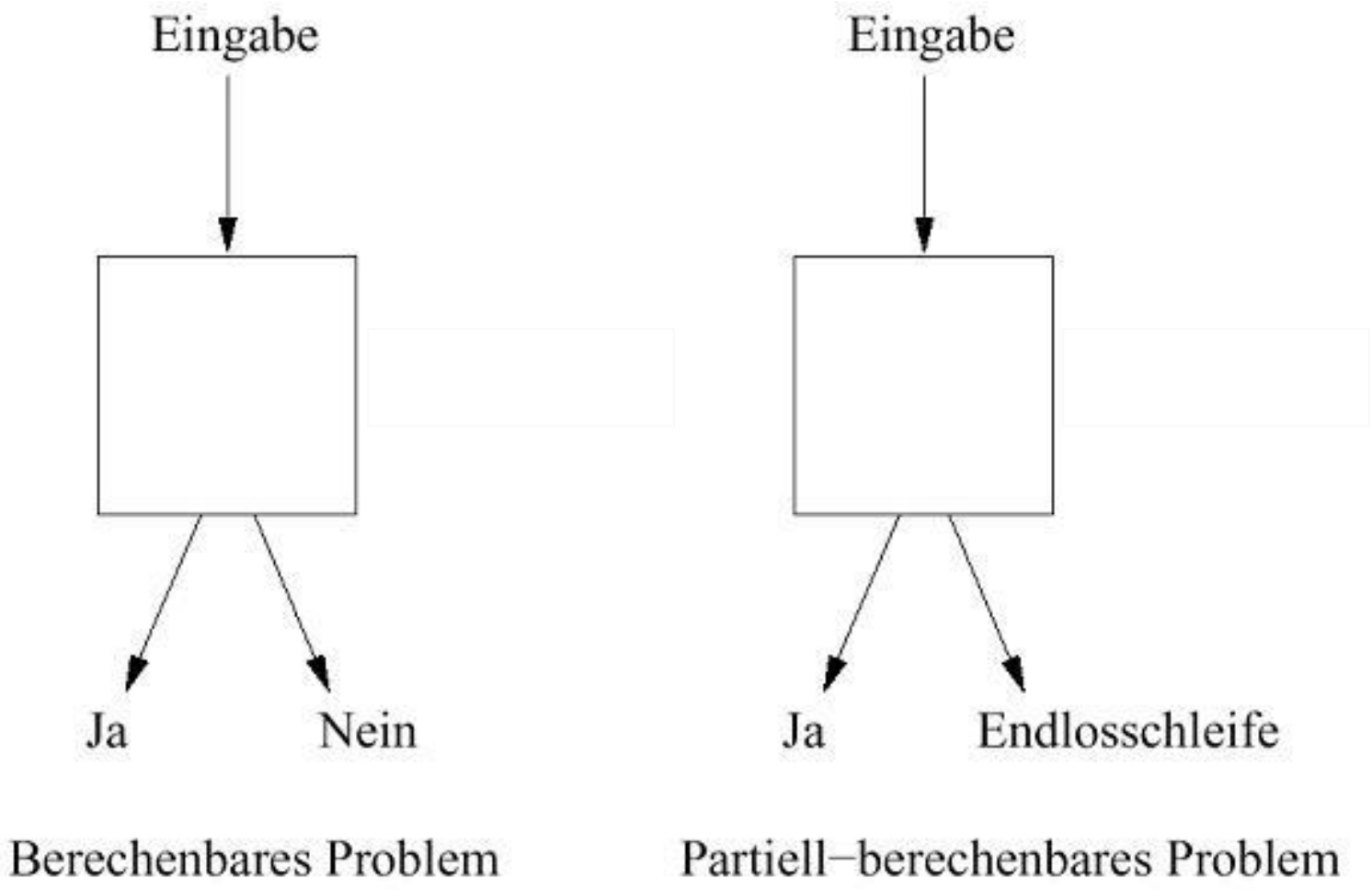
Partielle Berechenbarkeit

- Wie gerade geschildert, gibt es Techniken, die in der Lage sind, automatisch Beweise über Programmeigenschaften zu führen.
- D.h. man kann mitunter für einen speziellen Algorithmus mithilfe eines Programms nachweisen, dass er total korrekt ist.
- Wie aber stellt sich in diesem Zusammenhang die Unentscheidbarkeit dar?
- Oder anders ausgedrückt: Welche Eigenschaften haben Algorithmen, die versuchen, nichttriviale Eigenschaften von Programmen nachzuweisen bzw. unentscheidbare Probleme zu lösen?
- In diesem Kontext spielt der Begriff der **partiellen Berechenbarkeit** eine wichtige Rolle.

Grade der Berechenbarkeit

- Wir wissen, dass eine große Anzahl von Problemen nicht berechenbar oder unentscheidbar ist.
- Betrachten wir erneut das Halteproblem: Wir möchten für ein beliebiges Programm `P` wissen, ob es angesetzt auf die Eingabedaten `D` anhält oder nicht.
- Offensichtlich ist die Situation einfach, wenn `P` anhält.
- Lediglich wenn `P` nicht anhält, gibt es Schwierigkeiten, denn wir können nicht automatisch prüfen, ob `P` noch halten wird oder nicht.
- Das Halteproblem gilt daher als **partiell berechenbar**, denn es gibt ein Verfahren, das **`true`** ausgibt, wenn `P` angesetzt auf `D` hält.
- Lediglich, wenn `P` nicht hält, erzeugt dieses Verfahren keine Ausgabe.
- Da wir nur dann ein `true` erhalten, wenn `P` angesetzt auf `D` hält, heißt das Halteproblem **partiell berechenbar** oder **semi-entscheidbar**.

Unterschied zwischen berechenbaren und partiell berechenbaren Problemen



Nicht partiell-berechenbare Probleme

- Partielle Berechenbarkeit bedeutet, dass wir ein Programm schreiben können, das zumindest dann anhält, wenn das gegebene Problem eine bestimmte Eigenschaft hat.
- Lediglich im Fall, dass diese Eigenschaft nicht zutrifft, würde das Programm unendlich laufen.
- Allerdings gibt es auch Probleme, die nicht partiell berechenbar sind, d.h. für die wir, egal welcher Fall vorliegt, unendlich lange warten müssten.
- Beispielsweise müsste ein Algorithmus zum Nachweis der Äquivalenz von zwei Verfahren für alle möglichen Eingaben überprüfen, ob die Verfahren die gleichen Ausgaben liefern.
- Da er dazu jeweils die Programme auf die Eingaben anwenden muss, benötigt er selbst für die Fälle, in denen beide Programme halten, unendlich lange.

Zusammenfassung (1)

- Ein **Algorithmus ist ein allgemeines Verfahren zur Lösung einer Klasse von Einzelproblemen**.
- Algorithmen haben eine **endliche Länge** und sie sind präzise formuliert.
- Eine **typische Form von Algorithmen** sind **`while`-Programme**.
- Die Menge der **`while`-Programme** ist abzählbar.
- Da die **Menge der totalen Funktionen nicht abzählbar** ist, sind **nicht alle Probleme mit Algorithmen lösbar**.
- Ein Beispiel für ein solches Problem ist das **Halteproblem**.
- Wir haben gezeigt, dass das **Halteproblem unlösbar** ist, d.h. dass es keinen Algorithmus gibt, der für ein Programm `P` entscheiden kann, ob es, angesetzt auf die Eingabe `D`, anhält oder nicht.

Zusammenfassung (2)

- Andere Probleme, wie das **Äquivalenzproblem** oder das **Totalitätsproblem**, sind ebenso unentscheidbar.
- Der **Satz von Rice** besagt, dass **alle nichttrivialen Eigenschaften von Algorithmen unentscheidbar** sind.
- Dennoch kann man für viele Probleme **manuell Korrektheitsbeweise** führen, auch wenn dies mitunter schwierig ist.