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**.

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**.

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: Verarbeitungsvorschiften, 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 Verarbeitungsvorschiften 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, Verarbeitungsvorschiften eine Rolle.

Determinismus und Determiniertheit

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

Determiniertheit: Verarbeitungsvorschiften sind *determiniert*, wenn sie bei gleichen Parametern und Startwert stets das gleiche Resultat liefern. Das trifft zum Beispiel nicht für randomisierte Verarbeitungsvorschiften zu, bei denen das Ergebnis zu einem gewissen Grad auf Zufall beruht.

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

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.
- Es existieren kompakte Sprachen, mit denen Algorithmen spezifiziert werden können, z.B. **`while`-Programme**.
- **`while`-Programme** sind ähnlich zu Java Programmen eingeschränkt auf: while, if-then-else, Vergleiche, Wertzuweisungen, allerdings erlauben sie Integer-Variablen mit beliebig vielen Stellen

Was können Computer berechnen?

- Programme sind die Umsetzung von Algorithmen in einer Programmiersprache sind.
- 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 Programm durch einen endlichen Text beschrieben sein muss**.

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

Hinweis: Entsprechendes gilt für alle Arten von Programmen. 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 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 $f : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$ 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 **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

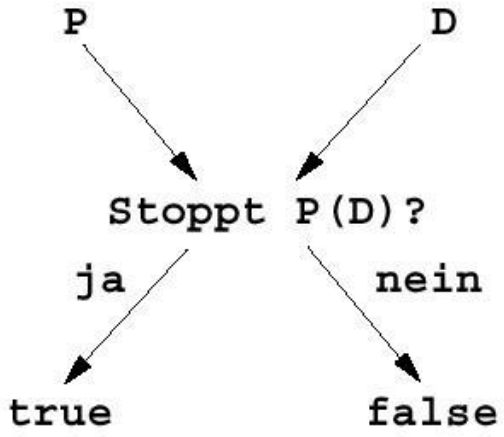
- Eines der größten Probleme bei Computerinstallationen auf der ganzen Welt sind Programmfehler (und insbesondere Laufzeitfehler).
- Mögliche Folgen solcher Fehler (wie z.B. der Jahr-2000-Fehler) sind
  - falsche Ergebnisse (Rentenbescheide für Säuglinge),
  - unvorhergesehene Programmabbrüche (Ausnahmefehler) oder gar
  - Endlosschleifen.
- Es wäre natürlich schön, wenn man mithilfe von Computerprogrammen prüfen könnte, 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, wenn ein Programm für eine gegebene Eingabe hält, und `false` sonst, ist **nicht berechenbar**.

Beweis der Unentscheidbarkeit des Halteproblems

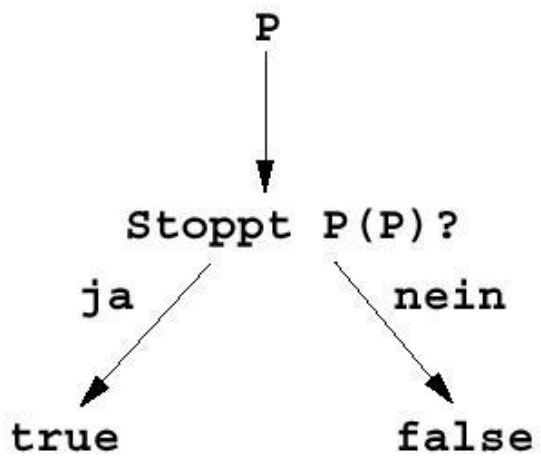
- Nehmen wir für einen Moment an, wir hätten ein Programm `Stopp-Tester` zur Lösung des Halteproblems.
- `Stopp-Tester` hat offensichtlich zwei Eingaben: Das zu überprüfende Programm `P` sowie die Eingabe `D` für P.
- `Stopp-Tester(P, D)` soll nun den Wert `true` ausgeben, wenn `P`, ausgeführt mit den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



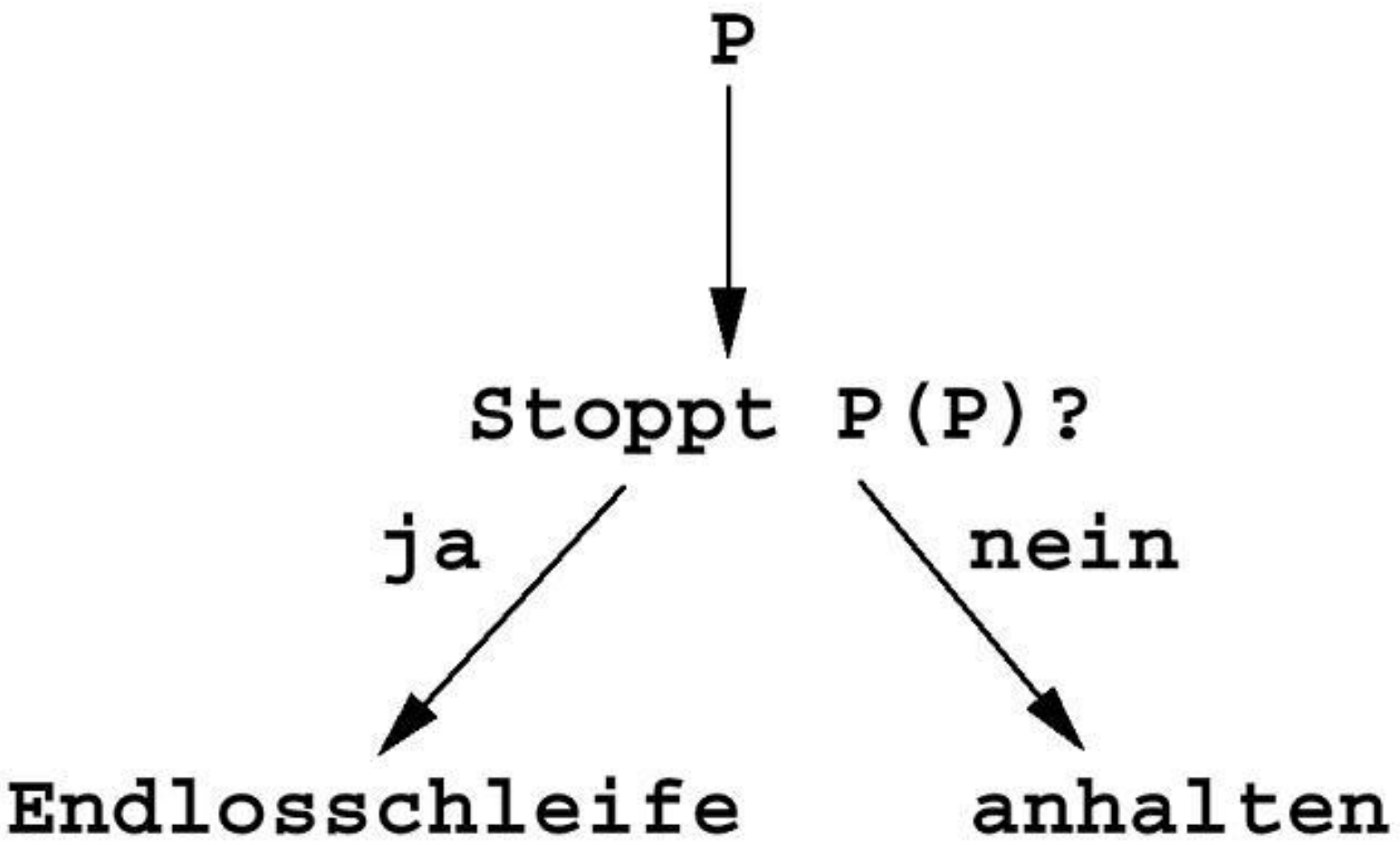
Das Programm `Spaßig`

- 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 anhält**.
- **Hält `P`** hingegen **nicht**, **hält** dafür **`Spaßig`**.

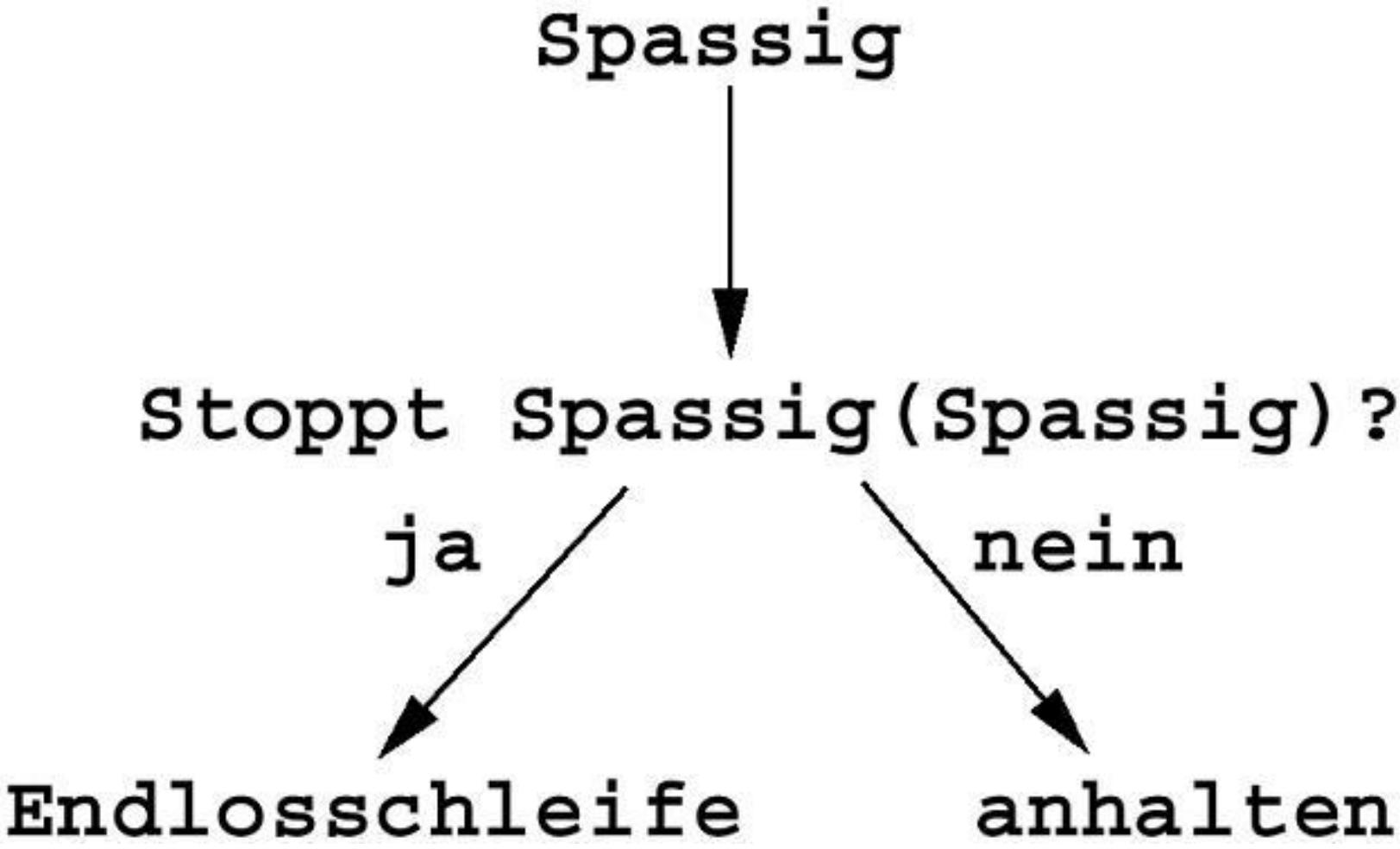
Arbeitsweise des Programms `Spaßig`



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 so genannten diophantischen 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.
- Es gibt viele weitere, nicht-berechenbare Probleme.
- 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 Äquivalenzproblem und die Reduktion

- 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.
- Um die Unentscheidbarkeit des Äquivalenzproblem zu beweisen, **reduzieren wir das Halteproblem auf das Äquivalenzproblem**.
- Idee: wir zeigen, dass wir das Halteproblem lösen könnten, wenn wir das gerade betrachtete Problem lösen könnten.

Das Äquivalenzproblem und die Reduktion

- 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);
}
```

Das Äquivalenzproblem und 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.

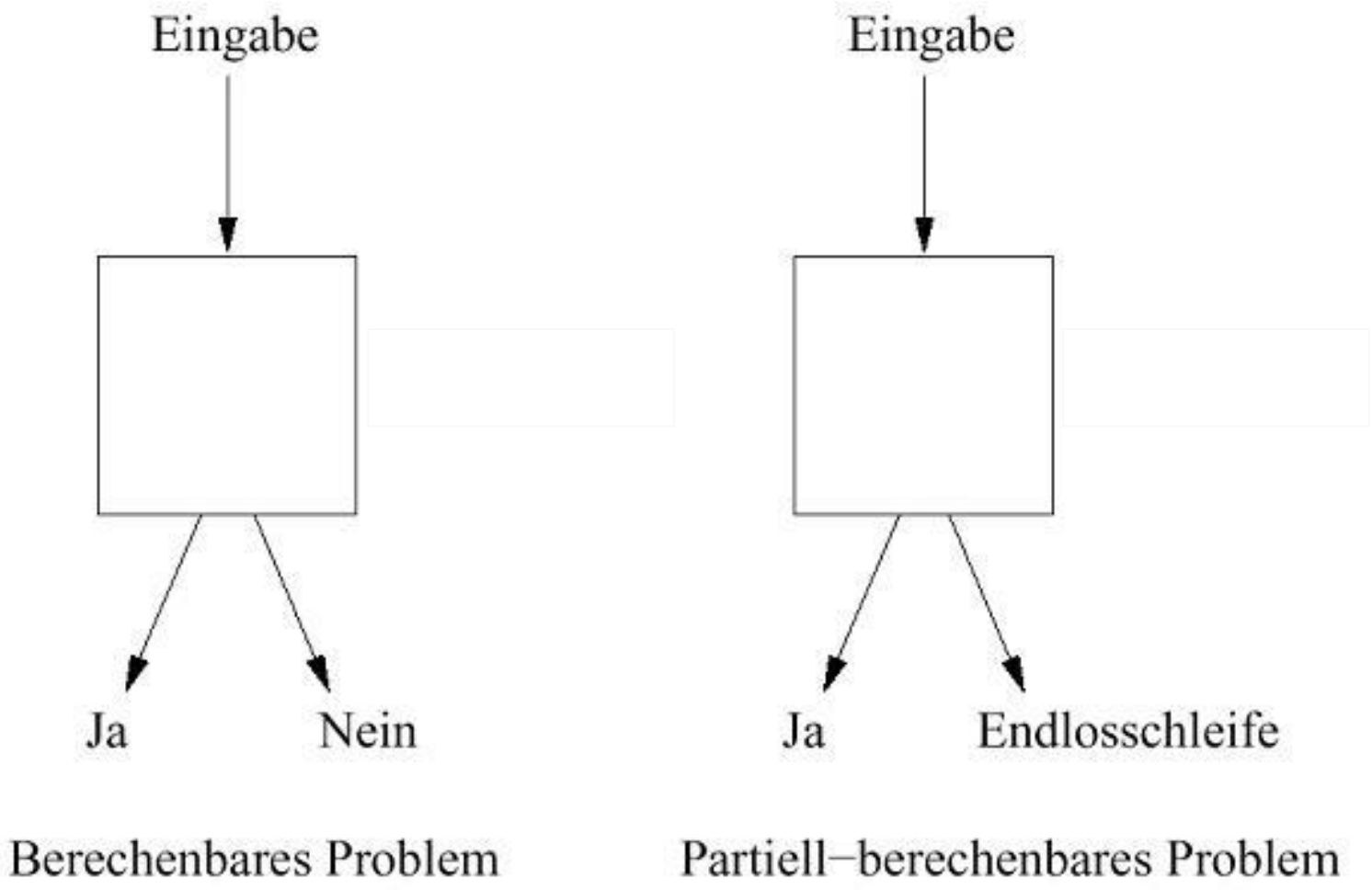
Rice's Theorem

- 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.

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.
- Die Menge der **(Java-) 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.