4.4 Grenzen der Berechenbarkeit

LK

> LK
>
> Der ganze Abschnitt gehört zum LK-Themenfeld *Berechenbarkeit und ihre Grenzen*. Für den Grundkurs ist er nicht verbindlich — er beantwortet aber die Frage, die sich nach jedem Laufzeitvergleich aufdrängt: Gibt es Probleme, für die gar kein Verfahren existiert?

Lange hielt man jedes mathematische Problem für lösbar, wenn man sich nur genug anstrengt. David Hilbert (1862–1943) formulierte diese Überzeugung so: Jedes bestimmte mathematische Problem müsse einer strengen Erledigung fähig sein — entweder durch eine Antwort oder durch den Nachweis, dass eine Lösung unmöglich ist.

Im 20. Jahrhundert stellte sich heraus: Das stimmt nicht. Es gibt Probleme, die sich computergerecht aufschreiben lassen und trotzdem von keinem Algorithmus gelöst werden können. Und zwar nicht, weil die Rechner zu langsam wären — sondern grundsätzlich.

Definition. Eine Menge M natürlicher Zahlen heißt **entscheidbar**, wenn es einen Algorithmus gibt, der für jede natürliche Zahl nach endlich vielen Schritten feststellt, ob sie zu M gehört oder nicht. Man nennt ihn ein **zweiseitiges Entscheidungsverfahren**.

Sie heißt **partiell entscheidbar**, wenn es einen Algorithmus gibt, der nur die Zugehörigkeit feststellt: Gehört die Zahl dazu, hält er mit „ja" an; gehört sie nicht dazu, läuft er möglicherweise ewig. Das ist ein **einseitiges Entscheidungsverfahren**.

4.4.1 Das Goldbach-Problem: entscheidbar

Die Goldbachsche Vermutung (1742) lautet: Jede gerade Zahl ≥ 4 lässt sich als Summe zweier Primzahlen schreiben.

$$4 = 2 + 2 \qquad 6 = 3 + 3 \qquad 8 = 5 + 3 \qquad 10 = 7 + 3 \qquad 12 = 7 + 5$$

Bis heute ist sie weder bewiesen noch widerlegt. Für die Informatik stellen wir eine schwächere Frage: Gibt es einen Algorithmus, der für eine gegebene gerade Zahl $n \geq 4$ entscheidet, ob sie die Goldbach-Eigenschaft hat?

Den gibt es — und er ist einfach:

Python

```python
def prim(zahl):
    if zahl < 2:
        return False
    i = 2
    while i * i <= zahl:
        if zahl % i == 0:
            return False
        i += 1
    return True


def goldbach(n):
    for k in range(2, n // 2 + 1):
        if prim(k) and prim(n - k):
            return True
    return False
```

Object Pascal

```
function Prim(zahl: integer): boolean;
var i: integer;
begin
  Result := zahl >= 2;
  i := 2;
  while i * i <= zahl do
  begin
    if zahl mod i = 0 then
    begin
      Result := false;
      exit;
    end;
    i := i + 1;
  end;
end;

function Goldbach(n: integer): boolean;
var k: integer;
begin
  Result := false;
  for k := 2 to n div 2 do
    if Prim(k) and Prim(n - k) then
    begin
      Result := true;
      exit;
    end;
end;
```

Die Schleife läuft höchstens bis $n/2$, endet also garantiert. Damit ist die Goldbach-Eigenschaft entscheidbar. Beachten Sie den Unterschied: *Für jede einzelne Zahl* ist die Frage beantwortbar. Ob die Vermutung *für alle* Zahlen gilt, beantwortet der Algorithmus nicht — dafür müsste er unendlich viele Zahlen prüfen.

4.4.2 Wundersame Zahlen: partiell entscheidbar

Die **Collatz-Funktion** (L. Collatz, um 1930) ist definiert durch

$$f(n) = \begin{cases} n/2 & \text{falls } n \text{ gerade} \\ 3n+1 & \text{falls } n \text{ ungerade} \end{cases}$$

Betrachtet werden dabei nur natürliche Zahlen $n \geq 1$. Eine solche Zahl heißt **wundersam**, wenn wiederholtes Anwenden von f irgendwann bei 1 ankommt. Für $n = 5$ ergibt sich die Folge

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Die 5 ist also wundersam. Der naheliegende Algorithmus:

Python

```python
def wundersam(n):
    while n > 1:
        n = n // 2 if n % 2 == 0 else 3 * n + 1
    return True
```

Object Pascal

```
function Wundersam(n: integer): boolean;
begin
  while n > 1 do
    if n mod 2 = 0 then
      n := n div 2
    else
      n := 3 * n + 1;
  Result := true;
end;
```

Ist n wundersam, hält er nach endlich vielen Schritten und antwortet „ja". Ist n es nicht, läuft er ewig — und wir erfahren nie, ob er noch anhält oder nicht. Er ist also nur ein einseitiges Entscheidungsverfahren.

::: stolperstein Der Rumpf prüft `n > 1` und meldet alles andere ungeprüft als wundersam — für $n = 0$ oder $n = -7$ antwortet er also sofort „ja", obwohl diese Zahlen die 1 nie erreichen. Das Programm setzt den Definitionsbereich $n \geq 1$ stillschweigend voraus. Wer es ernst meint, fängt den Fall ab: In Python `if n < 1: raise ValueError(...)`, in Pascal eine Prüfung vor der Schleife. Dieselbe Lücke steckt in sehr vielen Programmen, die „für die sinnvollen Eingaben" geschrieben wurden. ::: Ein zweiseitiges ist bis heute nicht bekannt; Collatz vermutete, dass jede Zahl wundersam ist, bewiesen ist auch das nicht.

4.4.3 Game of Life: nachweislich unentscheidbar

Im Spiel des Lebens (J. Conway) lebt jede Zelle eines Gitters oder ist tot. Eine tote Zelle wird lebendig, wenn genau drei ihrer acht Nachbarn leben. Eine lebende Zelle stirbt, wenn mindestens vier Nachbarn leben (Übervölkerung) oder höchstens einer (Vereinsamung).

Gesucht ist ein Algorithmus, der für eine beliebige Anfangsgeneration entscheidet, ob sie irgendwann ausstirbt. Man kann die Generationen einfach nachrechnen: Stirbt die Population, ist die Antwort da. Stirbt sie nicht, rechnet der Algorithmus ewig. Wieder nur ein einseitiges Verfahren — mit einem Unterschied zur Collatz-Frage: Für das Game of Life ist *bewiesen*, dass es kein zweiseitiges Verfahren geben kann.

4.4.4 Das Halteproblem

Der wichtigste Satz dieses Abschnitts geht auf Alan Turing (1936) zurück — auf dieselbe Arbeit, die die Turingmaschine aus Abschnitt 1.2 einführte. Turing brauchte die Maschine genau dafür: um überhaupt sagen zu können, was „es gibt kein Verfahren" bedeutet.

Der Satz lässt sich für jede Programmiersprache formulieren:

Merke: Satz von Turing (Halteproblem)

Es gibt kein Programm, das für ein beliebiges Programm P und eine beliebige Eingabe x entscheidet, ob P angewendet auf x nach endlich vielen Schritten anhält. Die Halte-Eigenschaft ist unentscheidbar und nur partiell entscheidbar.

Beweis durch Widerspruch. Nehmen wir an, es gäbe eine Funktion `haelt(P, x)`, die für den Quelltext `P` und die Eingabe `x` den Wert `True` liefert, wenn `P` bei Eingabe `x` anhält, und sonst `False`.

```python
def haelt(P, x):
    # Annahme: liefert True, falls P bei Eingabe x anhaelt, sonst False
    ...
```

```pascal
function Haelt(P, x: string): boolean;
begin
  { Annahme: liefert true, falls P bei Eingabe x anhaelt, sonst false }
end;
```

Mit dieser Funktion bauen wir ein zweites Programm:

```python
def seltsam(P):
    if haelt(P, P):
        while True:      # Endlosschleife
            pass
    return "fertig"
```

```pascal
procedure Seltsam(P: string);
begin
  if Haelt(P, P) then
    while true do ;      { Endlosschleife }
  writeln('fertig');
end;
```

`seltsam` tut also genau das Gegenteil dessen, was `haelt` vorhersagt: Hält `P` bei Eingabe des eigenen Quelltextes an, so läuft `seltsam` ewig weiter; hält `P` nicht an, so hält `seltsam` an.

Da `P` beliebig ist, dürfen wir `P = seltsam` einsetzen. Dann gilt:

- Ist `haelt(seltsam, seltsam) = True`, so gerät `seltsam` in die Endlosschleife — hält also **nicht** an. Widerspruch.
- Ist `haelt(seltsam, seltsam) = False`, so überspringt `seltsam` die Schleife und gibt „fertig" zurück — hält also **doch** an. Widerspruch.

`seltsam` hält bei Eingabe des eigenen Quelltextes genau dann an, wenn es nicht anhält. Beides ist unmöglich. Also war die Annahme falsch: Die Funktion `haelt` kann es nicht geben. ■

Merke

Der Beweis sagt nichts über fehlende Rechenleistung. Er zeigt, dass die Annahme selbst widersprüchlich ist. Kein schnellerer Rechner, keine bessere Programmiersprache und keine künstliche Intelligenz ändern daran etwas.

4.4.5 Was folgt daraus für die Praxis?

Das Halteproblem ist kein Kuriosum für Theoretiker, sondern die Ursache für eine Reihe sehr praktischer Grenzen:

- Kein Werkzeug kann für beliebige Programme zuverlässig entscheiden, ob sie sich aufhängen. Entwicklungsumgebungen warnen vor *erkennbaren* Endlosschleifen — mehr ist nicht möglich.
- Kein Virenscanner kann durch reine Programmanalyse für jedes Programm entscheiden, ob es sich schädlich verhält. Deshalb arbeiten Scanner mit Signaturen, Heuristiken und Beobachtung zur Laufzeit.
- Allgemein gilt (Satz von Rice): Jede nichttriviale Frage über das *Verhalten* eines beliebigen Programms ist unentscheidbar. Fragen über seinen *Text* — Zeilenzahl, benutzte Schlüsselwörter — sind dagegen problemlos beantwortbar.

Arbeitsauftrag 9

36. Bestimmen Sie die Collatz-Folge für $n = 7$ und $n = 15$. Wie viele Schritte brauchen Sie jeweils?
37. Programmieren Sie den Goldbach-Test und prüfen Sie alle geraden Zahlen von 4 bis 10 000.
38. Programmieren Sie einen Collatz-Rechner, der die Folge und ihre Länge ausgibt. Suchen Sie die Zahl unter 1000 mit der längsten Folge.
39. Erklären Sie mit eigenen Worten den Unterschied zwischen entscheidbar und partiell entscheidbar. Ordnen Sie Goldbach-Eigenschaft, Wundersamkeit und Halte-Eigenschaft zu.
40. Warum genügt es für den Beweis des Halteproblems nicht zu sagen, `haelt` sei „zu schwer zu programmieren"? Was leistet der Widerspruchsbeweis stattdessen?
41. Ein Mitschüler schlägt vor: „Man lässt das Programm einfach eine Stunde laufen. Läuft es dann noch, hält es nicht an." Nehmen Sie Stellung.
42. Erklären Sie in fünf Sätzen, warum Turing für diesen Beweis die Maschine aus Abschnitt 1.2 brauchte — warum also die Aussage „es gibt kein Verfahren" ohne einen festgelegten Verfahrensbegriff gar nicht prüfbar wäre.