Grenzen der Berechenbarkeit

Jedes mathematische Problem muß einer strengen Erledigung fähig sein; sei es, daß eine Antwort auf die gestellte Frage gelingt; sei es, daß die Unmöglichkeit einer Lösung und damit die Notwendigkeit des Mißlingens aller Lösungsversuche dargetan wird.

An anderer Stelle:

Diese Überzeugung von der Löslichkeit eines jeden mathematischen Problems ist uns ein kräftiger Ansporn während der Arbeit; wir hören in uns den steten Zuruf: Da ist das Problem, suche die Lösung. Du kannst sie durch reines Denken finden; denn in der Mathematik gibt es kein IGNORABIMUS!

David Hilbert
1862 - 1943

Hilbert stellte im Jahre 1900 auf dem Internationalen Mathematiker-Kongreß in Paris 23 Probleme vor, die zu diesem Zeitpunkt ungelöst waren; darunter

Hilberts zehntes Problem:

Gibt es ein Verfahren, das für eine beliebige diophantische Gleichung entscheidet, ob sie lösbar ist?

> ***Diophantische Gleichungen*** *sind Gleichungen der Form $f(x_1, x_2, ..., x_n) = 0$, wobei f ein Polynom in mehreren Variablen und mit ganzzahligen Koeffizienten ist und nur ganze Zahlen als Lösungen betrachtet werden. Ein bekanntes Beispiel ist die Gleichung $f(x,y,z) = 0$ mit $f(x,y,z) = x^2 + y^2 - z^2$, die mit dem Satz des Pythagoras zusammenhängt. Diophantische Gleichungen spielen in der Geschichte der Mathematik eine wichtige Rolle, und viele große Mathematiker haben sich intensiv mit diesen Fragen beschäftigt.*

Die Antwort (Nein!) wurde erst 1970 von dem russischen Mathematiker Juri W. Matijassewitsch (*1947) gefunden!

Um uns dem Problem der Entscheidbarkeit zu nähern, betrachten wir zunächst folgendes in Python codiertes **Beispiel** (*pumuckl.py*):

```
import os

i = int(input('i = '))
j = 0

while True:
    i = i // 2
    k = i + 1
    j += 1
    print('i =',i)
    print('k =',k)
    print()
    if k == i:  break

print('j =', j)

os.system('pause')
```

Anhand von Testdurchläufen läßt sich vermuten und mittels einer Analyse des Programmtextes auch absichern, daß das Programm für jede Eingabe einer natürlichen Zahl i in eine Endlosschleife gerät, somit nicht terminiert.

Frage: Gibt es ein allgemeines Verfahren, um für jeden Algorithmus a priori zu entscheiden, ob er für zulässige Eingabedaten terminiert?
Bevor wir uns dieser Problemstellung (**„Halteproblem"**) widmen, vertiefen wir den Begriff Entscheidbarkeit anhand einiger Beispiele.

Definition: Eine Teilmenge **M** der Menge der **N** der natürlichen Zahlen heißt entscheidbar genau dann, wenn es einen Algorithmus gibt, der für jede natürliche Zahl **n** entscheidet, ob sie Element der Menge **M** ist oder ob sie nicht Element der Menge **M** ist; falls **M** durch die Eigenschaft **E** gegeben ist, heißt die Eigenschaft **E** entscheidbar, falls **M** entscheidbar ist.

Bemerkung: Diese Definition beschreibt ein zweiseitiges Entscheidungsverfahren, da festgestellt wird, ob die Eigenschaft E zutrifft oder nicht zutrifft.

Primzahleigenschaft

Eine natürliche Zahl **n**, **n** ≥ **2**, ist eine Primzahl genau dann, wenn sie nur durch 1 und durch sich selbst ohne Rest teilbar ist (wenn also 1 und n die einzigen Teiler von n sind).

Menge der Primzahlen: **M** = {**n**∈ **N** | **n** ist Primzahl}

Ein Algorithmus, der eine Zahl **n** auf Primzahleigenschaft untersucht, testet, ob die Zahl **n** Teiler aus dem ganzzahligen Bereich [2, ... , n//2] hat. Falls ein Teiler gefunden wird, hat **n** nicht die Primzahleigenschaft, und der Algorithmus bricht ab. Falls ein Teiler nicht gefunden wird, hat **n** die Primzahleigenschaft, und der Algorithmus bricht nach n//2 − 1 Divisionsoperationen ab.

Die Eigenschaft, Primzahl zu sein, ist für jede natürliche Zahl **n** entscheidbar.

Folgender in Python codierte Algorithmus entscheidet nach Eingabe einer natürlichen Zahl n, welche Zahlen aus der Menge {2, . . . , n} die Primzahleigenschaft haben:

```
# Primzahltest
# Nach Eingabe einer natuerlichen Zahl n, n>1, entscheidet dieser
# Algorithmus, welche Zahlen aus der Menge {2, 3, . . ,n} Primzahlen sind.

while True:
  try:    n = int(input('n = '))
  except:
    print('Gib eine natuerliche Zahl n mit n>1 ein!')
    continue
  if n <= 1:
    print('Gib eine natuerliche Zahl n mit n>1 ein!')
    continue
  break

def prim(x):
  if x == 2: return True
  i = 2
  while i <= x//2:
      if x % i == 0: return False
      i += 1
  return True

for m in range(2,n+1):
  if prim(m): print(m,' ist Primzahl')
  else:       print(m,' ist keine Primzahl')
```

Hinweis: $\mathbf{2}^{\mathbf{136279841}} - \mathbf{1}$, eine Zahl mit 41 024 320 Stellen, ist die z. Zt. größte bekannte Primzahl (November 2024).

Die Goldbachsche Vermutung:

„Jede gerade natürliche Zahl, die größer als 2 ist, läßt sich als Summe zweier Primzahlen schreiben."

Wir Definieren die Goldbach-Eigenschaft wie folgt:
Eine Zahl **n** $\in$ **N** hat die Goldbacheigenschaft **G** genau dann, wenn es Primzahlen **p** und **q** gibt mit **n = p + q**.

M = {**n**$\in$ **N** | **n** hat die Eigenschaft **G**}

Die Menge **M** ist entscheidbar, denn es läßt sich für jede natürliche Zahl feststellen, ob sie die Goldbacheigenschaft hat oder nicht hat.

Die Goldbachsche Vermutung kann man auch folgendermaßen formulieren:

„Jede gerade natürliche Zahl n mit n>2 hat die Goldbacheigenschaft."

Der Algorithmus „Goldbach-Test" in Python:

```
# Goldbachsche Vermutung
# Nach Eingabe einer geraden natuerlichen Zahl n, n>2,
# entscheidet dieser Algorithmus, ob n sich als Summe
# zweier Primzahlen darstellen laesst.

def prim(x):
  if x == 2: return True
  i = 2
  while i <= x//2:
      if x % i == 0: return False
      i += 1
  return True

while True:

  while True:
    try: n = int(input('n = '))
    except:
      print('Gib eine gerade natuerliche Zahl n mit n>2 ein!')
      continue
    if n <= 2 or n % 2 != 0:
      print('Gib eine gerade natuerliche Zahl n mit n>2 ein!')
      continue
    break

  k = 1
  while True:
    k += 1
    if prim(k) and prim(n-k):
      print(n,'hat Goldbacheigenschaft mit')
      print(n,'=',k,'+',n-k)
    if k == n//2 or (prim(k) and prim(n-k)): break

  print()
  print('weiter? <y> <n>')
  ans = input()
  if ans != 'y': break
```

Dieser Algorithmus greift auf die Boolesche Funktion **prim** zurück, die nach Eingabe einer beliebigen natürlichen Zahl **x** entscheidet, ob **x** Primzahl ist oder nicht ist.

Da die Entscheidung über die Goldbacheigenschaft einer geraden natürlichen Zahl **n** an die Entscheidung über die Primzahleigenschaft der Summanden **k** und **n-k**, in die **n** zerlegt wird, anknüpft, ist die Goldbacheigenschaft für jede gerade natürliche Zahl **n** entscheidbar. Dagegen ist bis heute nicht entschieden, ob jede gerade natürliche Zahl n ≥ 4 die Goldbacheigenschaft hat, sich somit als Summe zweier Primzahlen darstellen läßt.

Programmieraufgabe: Der oben in Python codierte Algorithmus bricht ab, sobald eine Zerlegung der geraden Zahl n in Primsummanden gefunden wird, und gibt die Zerlegung an. Erweitere den Quellcode so, daß alle möglichen Zerlegungen der Zahl n in Primsummanden ausgegeben werden.

Wundersame Zahlen (Collatz-Problem)

Gegeben ist eine natürliche Zahl **n**; wie bilden eine Zahlenfolge **$\{a_i\}$** nach folgender Vorschrift (**Collatz-Folge**):

(1) $\mathbf{a_1 = n}$; n ist also Startwert
(2) $\mathbf{a_{i+1} = 3 \cdot a_i + 1}$, falls a_i ungerade ist ($i \geq 1$)
(3) $\mathbf{a_{i+1} = a_i \ DIV \ 2}$, falls a_i gerade ist ($i \geq 1$)
(4) Die Folge **$\{a_i\}$** bricht ab, sobald der Wert 1 erreicht wird.

Definition: Eine natürliche Zahl **n** heißt wundersam, wenn die gemäß vorstehenden Vorschriften gebildete Folge nach endlich vielen Schritten den Wert 1 erreicht.

Siehe auch: *http://de.wikipedia.org/wiki/Collatz-Problem*
http://de.wikipedia.org/wiki/Lothar_Collatz

Der Algorithmus zur Berechnung der Collatz-Folge liefert nur die Entscheidung, ob **n** zu der Menge

$$\mathbf{M} = \{n \in \mathbf{N} \mid n \text{ ist wundersam}\}$$

gehört; falls **n** allerdings nicht wundersam ist, terminiert der Algorithmus nicht, und eine Entscheidung wird nicht getroffen. Die Menge **M** ist daher partiell entscheidbar, es handelt sich hier um ein einseitiges Entscheidungsverfahren.

Definition: Eine Teilmenge **M** der Menge **N** der natürlichen Zahlen heißt partiell entscheidbar genau dann, wenn es einen Algorithmus gibt, der für jede natürliche Zahl **n** entscheidet, ob sie Element der Menge **M** ist; falls **M** durch die Eigenschaft **E** gegeben ist, heißt **E** partiell entscheidbar, falls **M** partiell entscheidbar ist.

Lothar Collatz formulierte 1937 folgende Vermutung:

Jede positive natürliche Zahl n ist wundersam.

oder:

Für jede positive natürliche Zahl n mündet die Collatz-Folge $\{a_i\}$ in den Zyklus 4, 2, 1.

Bis heute ist die Collatz-Vermutung weder bewiesen noch widerlegt.

Bisherige Resultate mittels Computerberechnungen:
Die Collatz-Vermutung trifft zu für

$n \leq 10^{20}$ (2017)
$n \leq 2^{68} \approx 2{,}95 \cdot 10^{20}$ (2020)

Ergänzende Links zum Collatz-Problem:

https://www.spiegel.de/wissenschaft/mensch/collatz-vermutung-deutscher-mathematiker-meldet-loesung-fuer-zahlenraetsel-a-766643.html

https://futurezone.at/science/mathematik-collatz-problem-vermutung-3n1-terence-tao/401862680

https://www.youtube.com/watch?v=Tx_sOO_G47k

Algorithmus zur Berechnung des Collatzfolge mit Ausgabe der Anzahl der Folgenglieder, bis die Folge zum ersten Mal den Wert 1 annimmt; als Datenstruktur für die Folge wird ein dynamisches Array gewählt, welches in Python als Liste realisiert wird:

```
# Collatz-Folge
# Nach Eingabe einer natuerlichen Zahl n, n>0, ermittelt
# dieser Algorithmus die Collatz-Folge mit Startwert n
# und berechnet deren Laenge.

while True:

  while True:
    try: n = int(input('n = '))
    except:
      print('Gib eine natuerliche Zahl n mit n>0 ein!')
      continue
    if n <= 0:
      print('Gib eine natuerliche Zahl n mit n>0 ein!')
      continue
    break

  a = list(range(1,2))
  i = 0
  a[i] = n

  while a[i] != 1:
    if a[i] % 2 != 0: a.append(3*a[i] + 1)
    else:             a.append(a[i]//2)
    i += 1

  print('Collatz-Folge mit Startwert n =', n,':')
  print(a)
  print('Laenge der Collatz-Folge:',len(a))

  print()
  print('weiter? <y> <n>')
  ans = input()
  if ans != 'y': break
```

Grenzen der Berechenbarkeit zeigen sich darin,

- daß der zeitliche Aufwand, um ein Problem zu lösen, zu stark anwächst (z. B. Ackermannfunktion, Travelling-Salesman-Problem: hyperexponentielles Wachstum; rekursive Berechnung der Fibonacci-Folge: exponentielles Wachstum),
- daß es Probleme gibt, die prinzipiell nicht entscheidbar sind, sich also einer algorithmischen Lösung entziehen.

Das Halteproblem

Vorbemerkung:
Jeder in einer Programmiersprache codierte Algorithmus (im Folgenden Programm genannt) wird als Quelltext **p** geschrieben, also in Form einer Zeichenkette (string); ebenso können wir die Eingabedaten für diesen Algorithmus als Zeichenkette **x** auffassen.

Der praktischen Informatik stellt sich folgende Frage:
Gibt es eine Prozedur oder Funktion, die als Eingabe den Quelltext eines beliebigen Programms p sowie dessen Eingabedaten x erhält und die entscheiden kann, ob das Programm p mit Eingabedaten x nach endlich vielen Schritten terminiert oder nicht terminiert?

Tatsächlich läßt sich in Einzelfällen (Beispiele: Potenzierungsalgorithmus, ägyptische Multiplikation, MergeSort) entscheiden, ob das jeweils gegebene Programm bei jeder Eingabe terminiert und korrekt im Sinne der Spezifikation arbeitet, indem man z. B. eine Schleifeninvariante verifiziert oder die mathematische Struktur des Algorithmus (MergeSort) analysiert.
Alan Turing zeigte bereits 1936, gründend auf sein Modell der Turingmaschine, daß es keinen Algorithmus gibt, der im allgemeinen das Halteproblem für beliebige Programme **p** und deren Eingabewerte **x** löst.

Link: http://de.wikipedia.org/wiki/Halteproblem
Turings Originalarbeit: https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

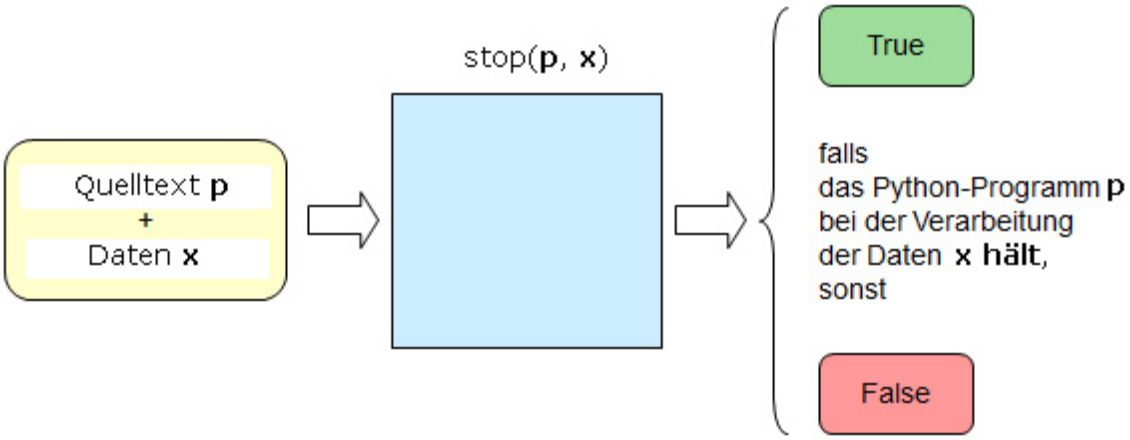
Behauptung: Es gibt kein allgemeines Entscheidungsverfahren, welches als Eingabe den Quelltext eines beliebigen Programms **p** sowie dessen Eingabedaten **x** hat und entscheidet, ob das Programm **p** mit Eingabedaten **x** nach endlich vielen Schritten terminiert oder nicht terminiert.

Der exakte Beweis gründet auf dem Modell der Turingmaschine; bei unserer Plausibilitätsbetrachtung lehnen wir uns an eine höhere Programmiersprache (hier: Python) an.

Wir führen den Beweis indirekt, indem wir unter der Annahme, daß es ein solches Entscheidungsverfahren gibt, einen Widerspruch zur Annahme herleiten.

Den Algorithmus, der die Entscheidung über die Terminierung eines Programms **p** mit Eingabedaten **x** liefert, formulieren wir als boolesche Funktion **„stop"**, welche als Eingabe das Programm **p** sowie dessen Eingabedaten **x** erhält; **„stop"** liefert den Wert **`True`**, falls **p** angewendet auf **x** terminiert, andernfalls den Wert **`False`**:

```
def stop(p, x):
   if <p terminiert angewandt auf x> == True: return True
   else:                                      return False
```



Das folgende Programm **„strange"** hat als Eingabe ein Programm **p** (formuliert als Quelltext **p**) und benutzt die boolesche Funktion **„stop"**.

Insbesondere ist zulässig, daß **„strange"** seinen eigenen Quelltext **p** als Eingabe erhält:

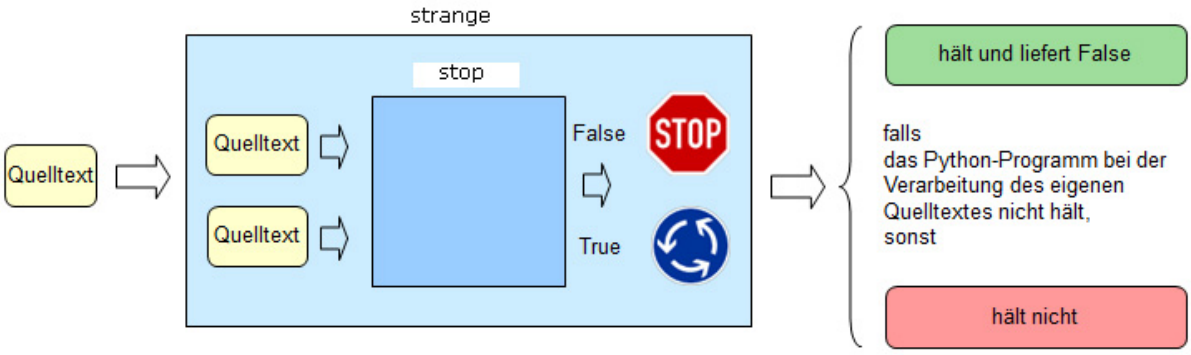
```
# strange

def stop(p, x):
   if <p terminiert angewandt auf x> == True: return True
   else:                                      return False

# Eingabe des Quelltextes p
p = input()

if stop(p, p):
   while True: pass

print('strange terminiert')
```



Beachte:
Falls **„stop"** den Wert `True` annimmt, gerät **„strange"** in eine Endlosschleife (in Python ist `pass` eine leere Anweisung, bei der nichts geschieht), falls **„stop"** den Wert `False` erhält, terminiert **„strange"** und gibt den Text „strange terminiert" aus.

Da **„strange"** als allgemeines Verfahren jeden Quelltext **p** akzeptiert, ist es insbesondere zulässig, daß **„strange"** seinen eigenen Quelltext **p** als Eingabedaten erhält, **„strange"** somit auf sich selbst angewendet wird.

Unter der Voraussetzung, daß es eine Boolesche Funktion `stop(p, x)` gibt, die entscheidet, ob das Programm **p** mit Eingabedaten **x** terminiert oder nicht terminiert, untersuchen wir folgenden Fälle:

1. Fall:

Annahme:
„strange" terminiert, falls **„strange"** als Eingabe seinen eigenen Quelltext **p** erhält.

⇒ Die Funktion `stop(p, p)` nimmt den Wahrheitswert `True` an.
⇒ **„strange"** gerät in die Endlosschleife `while True: pass`
⇒ **„strange"** terminiert nicht, im Widerspruch zu der Annahme, daß **„strange"** nach Eingabe seines eigenen Quelltexts anhält.

2. Fall:

Annahme:
„strange" terminiert nicht, falls **„strange"** als Eingabe seinen eigenen Quelltext **p** erhält.

⇒ Die Funktion `stop(p, p)` nimmt den Wahrheitswert `False` an.
⇒ **„strange"** terminiert mit Ausgabe des Text-Strings „strange terminiert", im Widerspruch zu der Annahme, daß **„strange"** nach Eingabe seines eigenen Quelltexts nicht anhält.

Da sich in beiden Fällen ein Widerspruch zur Annahme ergibt, kann es eine Boolesche Funktion `stop(p, x)`, die entscheidet, ob **p** mit Eingabedaten **x** terminiert oder nicht terminiert, nicht geben.

Anhang:
Algorithmus **strange** in Python

```
def strange(p):

    # Annahme: es gibt eine Funktion stop, die den Booleschen Wert
    # True liefert, falls Programm p mit Eingabedaten x haelt.
    # Andernfalls liefert stop den Booleschen Wert False.
    def stop(p, x):
        ans = input('Annahme: strange terminiert <y>es <n>o  ')
        if ans == 'y': return True
        else:
           if ans == 'n': return False

    if stop(p, p):
        while True: pass

    else: return False

# Eingabe des Quelltexts p von strange
p = '''
def strange(p):

    # Annahme: es gibt eine Funktion stop, die den Booleschen Wert
    # True liefert, falls Programm p mit Eingabedaten x haelt.
    # Andernfalls liefert stop den Booleschen Wert False.
    def stop(p, x):
        ans = input('Annahme: strange terminiert <y> <n> ')
        if ans == 'y': return True
        else:
           if ans == 'n': return False

    if stop(p, p):
        while True: pass

    else: return False

# Eingabe des Quelltexts p von strange
p = '''

'''
if strange(p) == False:
    print('strange, angewandt auf den eigenen Quelltext, terminiert.')
else:
    print('strange, angewandt auf den eigenen Quelltext, terminiert nicht.')
'''

if strange(p) == False:
    print('strange, angewandt auf den eigenen Quelltext, terminiert.')
else:
    print('strange, angewandt auf den eigenen Quelltext, terminiert nicht.')
```