4. Grenzen der Berechenbarkeit

4.2 Grenzen der Berechenbarkeit

Der Begriff **Berechenbarkeit** in der Informatik:

Eine Funktion nennt man **berechenbar**, wenn es einen Algorithmus gibt, der diese Funktion realisiert.
Das heißt, dass der Algorithmus für jede Eingabe aus der Definitonsmenge der Funktion terminiert und den korrekten Funktionswert berechnet.

Beispiel:

Die Quadratfunktion ist berechenbar.
Ein einfacher Algorithmus in Pseudocode lautet:

```
Eingabe n
Ausgabe n*n
```

Auch die Wurzelfunktion ist berechenbar.

Mögliche Algorithmen in Java:

```
public double wurzelJava(double r){
        return Math.sqrt(r);
}

public double wurzelHeron(double r, double genauigkeit){
    double erg = r;
    do {
        erg = (erg+ r / erg) / 2;
    }
    while (erg*erg - r > genauigkeit);

    return erg;
    }
```

Die beiden Algorithmen ergeben zwar nur Näherungswerte, deren Genauigkeit jedoch theoretisch beliebig verbessert werden kann.

Algorithmus für die Berechnung der Wurzelfunktion mit Berücksichtigung der Definitionsmenge:

```
public double wurzelJava2(double r){
    if (r >= 0){
        return Math.sqrt(r);
    }

    else{
        while(true){
            //tue nichts!
        }
    }
}
```

Der Algorithmus terminiert für negative Eingabewerte aufgrund der Endlosschleife nicht.
Dennoch berechnet er die Wurzelfunktion, da er für jeden Eingabewert aus dem Definitionsbereich den korrekten Funktionswert berechnet.
Anstelle der Endlosschleife wird man in der Praxis natürlich sinnvoller eine Fehlermeldung ausgeben.

Der Begriff der Berechenbarkeit lässt sich auch auf **nichtmathematische Funktionen** übertragen:

Beispiel:

```
public String rueckwaerts(String s){
     if(s.length()==1){
         return s;
     }
     else{
       return s.charAt(s.length()-1)+rueckwaerts(s.substring(0,s.length()-1));
     }
}
```

Der Algorithmus berechnet eine Funktion, die das eingegebene Wort rückwärts schreibt.
$f: wort \mapsto wort\ r\ddot{u}ckw\ddot{a}rts\ geschrieben$

Der Begriff der Berechenbarkeit kann ganz wesentlich von der intuitiven Vorstellung des „Berechnens“ abweichen.

Beispiel:

$$f(x) = \begin{cases} 1, wenn\ die\ Goldbachsche\ Vermutung\ richtig\ ist \\ 0, wenn\ die\ Goldbachsche\ Vermutung\ falsch\ ist \end{cases} \qquad D_f = \mathbb{R}$$

Die bislang noch nicht bewiesene oder widerlegte Goldbachsche Vermutung besagt dabei, dass sich jede gerade natürliche Zahl $n \geq 4$ als Summe zweier Primzahlen schreiben lässt.

Beispiel:
$12 = 5 + 7$ (einzige Möglichkeit)
$2020 = 3 + 2017 = 17 + 2003 = 23 + 1997 = \cdots$ (verschiedene Möglichkeiten)

Für ungerade Zahlen gibt es auch solche Zerlegungen, aber eben nicht für jede ungerade Zahl:
$7 = 2 + 5$
11: keine Zerlegung möglich

Die obige Funktion ist berechenbar. Der zugehörige Algorithmus würde für alle $x \in \mathbb{R}$ entweder 1 oder 0 ausgeben. Dazu müsste man jedoch wissen, ob die Goldbachsche Vermutung richtig ist oder nicht.

Javaprogramm zum Testen der Goldbachschen Vermutung:

```
public class GoldbachVermutung{

    public void zerlege(int n){
        Primzahlsieb pz = new Primzahlsieb(n);
        pz.primzahlenBerechnen();

        int summand1, summand2;

        for(int i = 2; i<=n/2; i++){
            if(pz.sieb[i]){
               summand1 = i;
               summand2 = n - summand1;
               if (pz.isPrim(summand2)){
                   System.out.println(n + "  =  " + summand1 +"  +  "+ summand2);
               }
            }
        }
    }
}
```

Die Klasse Primzahlsieb stellt dabei Methoden zum Primzahltest zur Verfügung. Quelltext auf der nächsten Folie:

Javaprogramm zum Erstellen einer Liste von Primzahlen und zur Überprüfung, ob eine Zahl eine Primzahl ist:

```
public class Primzahlsieb {
    boolean[] sieb;
     // Erzeugt ein Primzahlsieb, mit dem alle Primzahlen von 1 bis n berechnet werden können.
    public Primzahlsieb(int n) {
        sieb = new boolean[n + 1];
    }

    //Berechnet alle Primzahlen von 1 bis zu einer Obergrenze.
    public void primzahlenBerechnen() {
        for (int p = 2; p < sieb.length; p++) {
            sieb[p] = true;
        }

        for (int p = 2; p < sieb.length; p++) {
            if (sieb[p]) {
                for (int i = 2 * p; i < sieb.length; i += p) {
                    sieb[i] = false;
                }
            }
        }
    }

    //Gibt alle Primzahlen von 1 bis n aus.
    public void primzahlenAusgeben() {
        primzahlenBerechnen();
        for (int p = 1; p < sieb.length; p++) {
            if (sieb[p]) {
                System.out.println(p);
            }
        }
    }

    //Gibt genau dann true zurück, wenn p eine Primzahl ist.
    //Bevor diese Methode aufgerufen wird, muss primzahlenBerechnen einmal aufgerufen werden.
    public boolean isPrim(int p) {
        return sieb[p];
    }
}
```

Es gibt auch Funktionen, für die nicht bekannt ist, ob sie berechenbar sind.

Beispiel: Collatz-Problem

Bei der Collatz-Folge startet man mit einer natürlichen Zahl $n > 0$. Ist n gerade, so ist $n/2$ die nächste Zahl der Folge. Ist n ungerade, ist $3n + 1$ die nächste Zahl. Diese Vorgehensweise wird mit der erhaltenen Zahl wiederholt. Erhält man als Ergebnis 1, endet die Folge.

Beispiel für $n = 7$: $22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$

Die folgende Funktion terminiert nur, falls die zugehörige Collatz-Folge bei 1 endet:

$$f(n) = \begin{cases} 1, & wenn\ n = 1 \\ f(n/2), & wenn\ n\ gerade\ ist \\ f(3n+1), & wenn\ n\ ungerade\ ist \end{cases} \qquad D_f = \mathbb{N}$$

Man vermutet, dass für jeden Startwert $n \in \mathbb{N}$ die Folge bei 1 endet und die obige Funktion terminiert. Auch diese Vermutung ist aber bisher nicht bewiesen oder widerlegt.
Deshalb ist noch nicht bekannt, ob die obige Funktion berechenbar ist.

Javaprogramm zum Erzeugen von Collatz-Folgen:

```
public class Collatz{
    //Gibt die Collatz-Folge mit den Startwert n aus:
    public void collatzAusgeben(long n){
        System.out.println(n);
        while (n>1){
            if( n % 2 == 0){
                System.out.println(n/2);
                n=n/2;
            }
            else{
               System.out.println(3*n+1);
               n=3*n+1;
            }
        }
        System.out.println("Ende");
    }

    //Gibt 1 aus, wenn die Collatz-Folge terminiert
    public long collatzBerechnen(long n){
        if( n == 1){
            return 1;
        }
        else{
            if( n % 2 == 0){
                return collatzBerechnen(n/2);
            }
            else {
                return collatzBerechnen(3*n+1);
            }
        }
    }
}
```

Es geht aber noch deutlich problematischer:

Beispiel: Ackermannfunktion

$$a(n,m) = \begin{cases} m+1, & wenn\ n = 0 \\ a(n-1,1), & wenn\ m = 0 \\ a\big(n-1, a(n,m-1)\big), & sonst \end{cases} \qquad D_f = \mathbb{N}\times\mathbb{N}$$

Die Werte der Ackermannfunktion sind nur für Sonderfälle und wenige Eingabewerte bekannt.

Zum Beispiel gilt:

$a(0,m) = m+1$
$ack(1,m) = m+2$ $\quad ack(2,m) = 2m+3$ $\quad ack(3,m) = 2^{m+3}-3$
$ack(4,0) = 13$ $\quad ack(4,1) = 65533$ $\quad ack(4,2) = 2^{65536}-3 \approx 2\cdot 10^{19728}$

Die Ackermannfunktion ist berechenbar.

Hintergrund: Man vermutete Anfang des 20. Jahrhunderts, dass jede berechenbare Funktion primitiv-rekursiv sei, d.h. sich aus einfachen Grundfunktionen zusammensetzt.
Wilhelm Ackermann konnte nachweisen, dass die von ihm konstruierte Funktion berechenbar, aber nicht primitiv-rekursiv ist.

Javaprogramm zum Berechnen der Ackermannfunktion:

```
public long ackermannBerechnen(long n, long m){
        if( n == 0){
            //System.out.println("n = " + n + "  m = " +m);
            return m+1;
        }

        else{
            if( m == 0){
                //System.out.println("n = " + n + "  m = " +m);
                return ackermannBerechnen(n-1,1);
            }
            else {
                System.out.println("n = " + n + "  m = " +m);
                return ackermannBerechnen(n-1, ackermannBerechnen(n, m-1));
            }
        }
    }
}
```

Gibt es Funktionen, die nicht berechenbar sind?

Ein analoges Beispiel aus der Geometrie ist die einfache Fragestellung, ob man einen Winkel mit Zirkel und Lineal dreiteilen kann. Im Gegensatz zur Fragestellung ist die Begründung der Antwort (hier: "nein") sehr schwierig.

Ähnlich verhält es sich auch mit dem **Halteproblem**:

Gibt es einen Algorithmus, der entscheidet, ob ein anderer Algorithmus mit einer bestimmten Eingabe terminiert?

Um das Problem konkreter in Zusammenhang mit dem Begriff der Berechenbarkeit zu bringen, verwendet man das Konzept der Turingmaschinen.
Diese geben nach der sogenannten Church-Turing-These den intuitiven Begriff der Berechenbarkeit wieder.

Der Berechenbarkeitsbegriff ist auf Funktionen zugeschnitten und hängt in der theoretischen Informatik mit dem Begriff der **Entscheidbarkeit** zusammen.

Eine Teilmenge A einer Menge (z.B. einer Sprache) heißt **entscheidbar**, wenn die zugehörige **charakteristische Funktion c berechenbar** ist.
Die charakteristische Funktion c hat nur zwei Funktionswerte, nämlich
$c(w) = 1$, wenn $w \in A$ und $c(w) = 0$, wenn $w \notin A$

Die wichtigste Fragestellung bezüglich des Halteproblems ist nun, ob es entscheidbar ist, d.h. ob eine Turingmaschine existiert die für jedes Paar aus kodierter Turingmaschine und Eingabe berechnen kann, ob die kodierte Maschine auf dieser Eingabe anhält.

Alan Turing konnte zeigen, dass das Halteproblem ein **unentscheidbares Problem** ist.

Die zugehörige charakteristische Funktion des Halteproblems ist also nicht berechenbar.

Eine verständliche Formulierung des Beweises (kein Pflichtstoff) findest du im Buch auf den Seiten 142 -143 oder auch im Ordner „Beweis_Halteproblem“

Ein weiteres Beispiel für eine nicht berechenbare Funktion ist die Fleißige-Biber-Funktion

Ausblick: Das P-NP-Problem (Ordner „pnp“)