KAPITEL 3

Rekursionstheorie: Algorithmen und Computer

1. Berechenbarkeit

Frage: Welche Funktionen sind berechenbar, welche Probleme effektiv entscheidbar?

1.1 Einschränkung: Wir beschäftigen uns vorerst nur mit *natürlichen Zahlen*, d.h. mit Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ (oder $\mathbb{N}^n$ nach $\mathbb{N}$), und Problemen der Form

"Hat die natürliche Zahl n die Eigenschaft P?", bzw.

"Hat $\vec{n} \in \mathbb{N}^n$ die Eigenschaft P?"

- Der Grund dafür ist, daß die Angelegenheit formal wesentlich einfacher wird.
- Es macht aber keinen Unterschied wenn wir statt natürlicher Zahlen rationale verwenden oder beliebig lange Zeichenketten (strings). Diese Objekte lassen sich ja leicht in natürliche Zahlen kodieren.
- Im Endeffekt reicht es natürlich auch, Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ zu betrachten, da sich $\mathbb{N}^n$ ja ebenfalls leicht kodieren läßt. Es stellt sich aber heraus daß sich die Theorie am leichtesten entwickeln bzw formulieren läßt, wenn man Funktionen von $\mathbb{N}^n$ nach $\mathbb{N}$ (für beliebige n) zugrundelegt (siehe z.B. die Definition der primitiven Rekursion. Die wäre etwas umständlicher wenn wir nur Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ verwenden würden).
- Eine Funktion $F : \mathbb{N}^n \to \mathbb{N}^m$ nennen wir berechenbar, wenn jede Komponente $\vec{x} \mapsto [F(\vec{x})]_i$ berechenbar ist. (Das ist äquivalent dazu daß $\vec{x} \mapsto \text{Code}(F(\vec{x}))$ berechenbar ist, wobei Code eine der üblichen Kodierungen von $\mathbb{N}^m$ nach $\mathbb{N}$ ist.)

Funktionen $f : \mathbb{R} \to \mathbb{R}$ können auch mit Methoden der Rekursionstheorie behandelt werden, das führt zum Gebiet der deskriptive Mengenlehre. Diese hat nützliche Anwendung z.B. in dynamischen Systemen und Ergodentheorie (Borel equivalence relations). In dieser Vorlesung können wir aber nicht darauf eingehen.

1.2 Beispiele für Algorithmen in der Mathematik

- *Euklidischer Algorithmus*, berechnet $\text{ggT} : \mathbb{N}^2 \to \mathbb{N}$.
  Bsp: $\text{ggT}(6, 9) = 3$.
  Das ist insofern ein problematisches Beispiel, als es ja einen trivialen Algorithmus gibt, der den ggT zweier Zahlen findet. (Welchen?) Die Pointe des Euklidischen Algorithmus ist, dass er den ggT von m und n *schneller* findet als der triviale (es werden höchstens in etwa $\log(\max(m, n))$ viele Schritte benötigt, im Gegensatz zu ca. $\max(m, n)$ beim trivialen).
- *Gauß Elimination* (oder Determinante) entscheidet effektiv die Frage:
  Ist $n \times n$-Matrix M (mit natürlichen Koeffizienten) in $\mathbb{Q}$ invertierbar?
  Bsp: $M = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ ist invertierbar, weil $\det(M) = 2 \neq 0$.

Dabei werden die $n \times n$-Matrizen z.B. als n^2-Tupel, d.h. Elemente von $\mathbb{N}^{n^2}$ aufgefaßt. Natürlich auch möglich: Matrizen mit Koeffizienten aus $\mathbb{Q}$ (z.B.: rationale Zahlen als Paare von natürlichen Zahlen kodieren).

DEFINITION 1.3 (der Berechenbarkeit). Eine Funktion $F : \mathbb{N} \to \mathbb{N}$ ist berechenbar, wenn es ein Computerprogramm gibt, das auf Input n den Output $F(n)$ liefert.

Um das zu präzisieren, müsste man natürlich den verwendeten Computer und die verwendete Programmiersprache genau angeben. Es stellt sich aber heraus: Der Begriff ist enorm robust, das heißt völlig unabhängig von der konkret gewählten Programmiersprache (vorausgesetzt natürlich diese Sprache ist einigermaßen sinnvoll).

Achtung: Gemeint ist natürlich ein idealisierter Computer (z.B. mit beliebig großem Speicher). Jeder reale/physikalische Computer kann natürlich nicht beliebig große Zahlen verarbeiten, könnte daher z.B. auch den Euklidischen Algorithmus nicht für alle Zahlen, sondern nur für entsprechend kleine berechnen.

Wir führen nun einige Computermodelle an, die alle zum selbsen Berechenbarkeitsbegriff führen. Die ersten sind die üblichen Programmiersprachen:

1.4 GLEICHER BERECHENBARKEITS-BEGRIFF FÜR:

- (prädikativ:) C, Fortran, Pascal, (objektor.:) C++, Java, Smalltalk, (Interpret.:) Basic, Perl, (funktional:) Lisp, (logisch:) Prolog, ...
- Sehr primitive Basic-artige Programmiersprache: URM
- (idealisiertes, 1-dim.) Papier, Bleistift, Radiergummi und einige primitive Anweisungen (Turingmaschine),
- μ-Rekursion (induktive Definition der berechenbaren Funktionen),
- in Robinson's Q repräsentierbare Funktionen. (Wird vielleicht noch später in der Vorlesung definiert.)

Ein Problem P ist also effektiv entscheidbar, wenn ein Computerprogramm die richtige Antwort auf das Problem geben kann.

1.5 WICHTIG:

- Das macht nur Sinn wenn P eine "freie Variable" hat, formaler:
- P ist Eigenschaft von (Tupeln von) natürlichen Zahlen.

Bsp: "Ist Matrix M invertierbar?" ist effektiv entscheidbar.
Es ergibt aber keinen Sinn zum Beispiel zu fragen:
"Ist Riemannsche Vermutung effektiv entscheidbar?"

Es gibt ja jedenfalls Computerprogramm das die R.V. richtig entscheidet, nämlich enweder "`Print Ja`" oder "`Print Nein`". Wir wissen halt nicht welches Programm das richtige ist.

Sehr wohl sinnvoll ist die Frage: "Ist R.V. in ZFC entscheidbar (d.h. beweisbar oder widerlegbar), oder nicht (wie ja z.B. CH)?"

2. Ein Computermodell: Die unbeschränkte Registermaschine

Wir stellen jetzt ein konkretes Computermodell vor, damit wir den Berechenbarkeitsbegriff formal definieren. Es ist allerdings in weiterer Folge nicht wirklich wichtig zu wissen, was eine URM genau ist — berechenbar heißt mit einem Computerprogramm berechenbar, dabei kann man an irgendeine Programmiersprache denken. Wenn man jedoch Aussagen wie "f ist nicht berechenbar" beweisen will, muß man sich natürlich vorerst auf ein bestimmtes Modell festlegen. Die URM hat da den Vorteil daß sie sehr einfach und daher leicht zu kodieren ist.

2.1 UNBESCHRÄNKTE REGISTERMASCHINE (URM):

- Hardware: Unendlich viele Register (Variablen) $R_0, R_1, \ldots$, jede kann eine (beliebig große) natürliche Zahl speichern.
- Programmiersprache: Basic-artig, nummerierte Programm-Zeilen, jede enthält eines von:

addiere 1 zu R_i	$R_i := R_i + 1$,
subtrahiere 1 (wenn möglich, sonst 0)	$R_i := R_i - 1$,
eine Sprung-Anweisung	`goto` m,
teste auf = 0	`if` $R_i = 0$ `goto` m,
liefere R_0 als Output	`return`

Bevor wir URM-Programme formal definieren, ein einfaches Beispiel:

Beispiel 2.2 (Addition zweier Zahlen).

```
1  if R1 = 0 goto 5
2  R1 := R1 − 1
3  R0 := R0 + 1
4  goto 1
5  return
```

Die Berechnung läuft in diskreten Zeiteinheiten ab (z.B. 1 Berechnungsschritt pro Sekunde):

2.3 Berechnung bei Input $(3, 2)$

Zeit:	0	Zeile:	1	R_0 :	3	R_1 :	2	Input $(3, 2)$ ist in R_0 und R_1 gespeichert.
Zeit:	1	Zeile:	2	R_0 :	3	R_1 :	2	
Zeit:	2	Zeile:	3	R_0 :	3	R_1 :	1	
Zeit:	3	Zeile:	4	R_0 :	4	R_1 :	1	
Zeit:	4	Zeile:	1	R_0 :	4	R_1 :	1	
Zeit:	5	Zeile:	2	R_0 :	4	R_1 :	1	
Zeit:	6	Zeile:	3	R_0 :	4	R_1 :	0	
Zeit:	7	Zeile:	4	R_0 :	5	R_1 :	0	
Zeit:	8	Zeile:	1	R_0 :	5	R_1 :	0	
Zeit:	9	Zeile:	5	R_0 :	5	R_1 :	0	
Zeit:	S	Zeile:	S	R_0 :	5	R_1 :	0	Output 5 aus R_0.

Daher: Addition ist berechenbar (nach unserer Definition).

▸ Formale Definition der URM

Die Formale Definition trägt hier nicht unbedingt zum Verständnis bei, deutet aber an, wie wir später ein universelles Programm schreiben können:

Definition 2.4. Ein *URM-Programm* ist eine endliche Teilmenge P von $\mathbb{N}^4$ die folgendes erfüllt:

- Wenn $l_1 = (a, b, c, d) \in P$ und $l_2 = (a', b', c', d') \in P$, und $a = a'$, dann ist $l_1 = l_2$. (Die erste Komponente ist die eingeutige Zeilennummer.)
- Wenn $l = (a, b, c, d) \in P$ dann ist $b \in \{0, 1, 2, 3, 4\}$. (Wir interpretieren 0 als `return` , 1 als Addition, 2 als Subtraktion, 3 als `goto` und und 4 als Testfunktion.)
- Es gibt ein $l_0 = (1, b, c, d) \in P$ (Startzeile), und wenn $(a, b, c, d) \in P$ dann $a > 0$ (die Zeile 0 wird nicht verwendet).

In dieser formalen Schreibweise hat unser Beispielprogramm also z.B. Form
$P = \{(1, 4, 1, 5), (2, 2, 1, 0), (3, 1, 0, 0), (4, 3, 0, 1), (5, 0, 0, 0)\}$.
Dabei entspricht z.B. $(1, 4, 1, 4)$ der Zeile "1 `if` $R_1 = 0$ `goto` 4":

Der erste Eintrag im 4-Tupel, 1, ist die Zeilennummer, 4 steht für die Testfunktion, der dritte Eintrag, 0, zeigt an welches Register getestet wird, und der vierte sagt welche Programmzeile als nächstes ausgeführt wird, wenn der Test erfolgreich war.

DEFINITION 2.5. Sei P ein URM-Programm. Wir definieren B, die k-Berechnungsfolge von P auf Input $(x_0, \dots, x_{n-1})$ folgendermaßen:

- $m \coloneqq \min(\{d : (a, b, c, d) \in P\}) + 2$ (D.h. m ist um 2 größer als alle Nummern der in P verwendeten Register.)
- $B = (B^0, \dots, B^{k-1})$ ist eine Folge der Länge k.
- $B^t \in \mathbb{N}^m$, d.h. jedes B^t ist ein m-Tupel $(a^t, r_0^t, \dots, r_{m-2}^t)$ natürlicher Zahlen. (Das erste Element ist die aktive Zeilennumer, dann kommen die Registerinhalte zum Zeitpunk t.)
- $B^0 = (1, x_0, \dots, x_{n-1}, 0, \dots, 0)$. (Die Startzeile ist 1, und am Anfang wird der Input in die Register geschrieben.)
- Angenommen $B^t = (a, r_0, \dots, r_{m-2})$ und $t + 1 < k$. Entweder $a = 0$ oder es gibt genau ein $(a, b, c, d) \in P$. Wir definieren $B^{t+1} = (a', s_0, \dots, s_{m-2})$ folgendermaßen:
  - Wenn $b = 0$ oder $a = 0$ (d.h. das Programm terminiert mit Output r_0, oder das Programm hat schon in einem früheren Zeitpunk terminiert), dann ist $B^{t+1} = (0, r_0, \dots, r_{m-2})$.
  - Wenn $b = 3$, dann ist $(s_0, \dots, s_{m-2}) \coloneqq (r_0, \dots, r_{m-2})$ und $a' = c$. (wenn eine Zeile mit Nummer c existiert, sonst 0).
  - Wenn $b = 4$, dann ist $(s_0, \dots, s_{m-2}) \coloneqq (r_0, \dots, r_{m-2})$, und wenn $r_d = 0$, dann ist $a' = c$, ansonsten $a + 1$ (wenn eine Zeil mit dieser Nummer existiert, sonst 0).
  - Wenn $b = 1$ (oder 2), dann ist $s_d = r_d + 1$ (oder $\max(0, r_d - 1)$), für die $i \leq m - 2$ ungleich d gilt $s_i = r_i$. $a' = a + 1$ (wenn eine Zeile mit Nummer $a + 1$ existiert, sonst 0).

BEISPIEL 2.6. Wieder die Addition von 3 und 2:
Wir haben schon gesehen daß $P = \{(0, 4, 1, 4), (1, 2, 1, 0), (2, 1, 0, 0), (3, 3, 0, 0), (4, 0, 0, 0)\}$.
Im Programm kommen R_0 und R_1 vor, daher ist $m = 1 + 2 = 3$.
Erst berechnen wir die 3-Berechnungsfolge auf Input $(3, 2)$:
$B = (B_0, B_1, B_2)$, $B_0 = (1, 3, 2)$, $B_1 = (2, 3, 2)$, $B_0 = (3, 3, 1)$.
Die 12-Folge ist ((1, 3, 2),(2, 3, 2),(3, 3, 1),(4, 4, 1),(1, 4, 1),(2, 4, 1),(3, 4, 0),(4, 5, 0), (1, 5, 0),(5, 5, 0),(0, 5, 0),(0, 5, 0)).

Ein Programm P liefert also genau dann den Output y auf Input $\vec{x}$, wenn für ein (oder: jedes) hinreichend große t das letzte Tuple der t-Berechnungsfolge von P auf Input $\vec{x}$ die Form $(0, y, \dots)$ hat.

DEFINITION 2.7. Sei P ein URM-Programm. Mit $P^t(x_0, \dots, x_{n-1})$ bezeichnen wir den Zustand, in dem sich die URM befindet wenn man das Programm P auf Input $(x_0, \dots, x_{n-1})$ t Zeiteinheiten lang laufen läßt.
Formaler: $P^t(x_0, \dots, x_{n-1})$ ist das t-te Tupel der $t + 1$-Berechnungsfolge von P auf Input $(x_0, \dots, x_{n-1})$.

Im obigen Beispiel wäre also $P^2(3, 2) = (1, 2, 1, 0)$ und $P^{12}(3, 2) = P^{2010}(3, 2) = (0, 5, 0)$.

Wir wiederholen nochmals einen wichtigen Punkt:

2.8 ENDLICHKEIT:

- Der Speicher ist nur *potentiell* unendlich (oder: beliebig groß). D.h. in einem Register kann eine beliebig große Zahl stehen. (→ Unterschied zu wirklichem Computer.)

- Programme sind (beliebig groß, aber) *endlich*! (Sonst wäre jede Funktion trivialerweise berechenbar.) (Wie?)

3. Primitiv rekursive (prim.r.) Funktionen

Eine nützliche Klasse berechenbarer Funktionen sind die primitiv rekursiven (prim.r.) Funktionen. Diese sind induktiv definiert: Gewisse Grundfunktionen sind prim.r., und die Zusammensetzung bzw. primitive Rekursion von prim.r. Funktionen ist wieder prim.r.
Jede prim.r. Funktion hat also eine endliche Definition bzw Konstruktion.

3.1 Grundfunktionen:

- *Projektionen:* $(x_0, \dots, x_i, \dots, x_{n-1}) \mapsto x_i$,
- *Konstante Nullfunktione:* $x \mapsto 0$,
- *Plus 1:* $x \mapsto x + 1$.

3.2 Zusammensetzung/Einsetzung: Wenn $f_1, \dots f_n : \mathbb{N}^m \to \mathbb{N}$ und $g : \mathbb{N}^n \to \mathbb{N}$ prim.r., dann ist $h : \mathbb{N}^m \to \mathbb{N}$, definiert durch $h(\vec{x}) = g(f_1(\vec{x}), \dots, f_n(\vec{x}))$, prim.r.

3.3 Primitive Rekursion: Wenn $g : \mathbb{N}^n \to \mathbb{N}$ und $h : \mathbb{N}^{n+1} \to \mathbb{N}$ prim.r., dann auch $f : \mathbb{N}^{n+1} \to \mathbb{N}$, definiert durch $f(n, \vec{p}) := \begin{cases} g(\vec{p}) & \text{wenn } n = 0, \\ h(f(n-1), \vec{p}) & \text{sonst.} \end{cases}$

Beispiel 3.4 (für prim.r. Funktionen:).

- Die Konstante Funktion 4: $f(n) = ((((0 + 1) + 1) + 1) + 1)$.
- Addition: $f(0, m) = m$, $f(n + 1, m) = f(n, m) + 1$.
- Charakteristische Funktion $\chi_{\mathbb{N}\setminus\{0\}}$: $f(0) = 0$, $f(n + 1) = 1$.
- Minus 1: $f(0) = 0$, $f(n + 1) = f(n) + \chi_{\mathbb{N}\setminus\{0\}}(n)$.
- Subtraktion: $f(0, m) = 0$, $f(n + 1, m) = f(n, m) - 1$.
- Multiplikation: $f(0, m) = 0$, $f(n + 1, m) = f(n, m) + m$.
- $(m, n) \mapsto m^n$: $f(0, m) = 1$, $f(n + 1, m) = f(n, m) \cdot m$.
- Charakteristische Funktion der Primzahlen (Übung),
- n wird auf die n-te Primzahl abgebildet (Übung).
- Code_1^n, eine Bijektion von $\mathbb{N}^n$ nach $\mathbb{N}$.

Wir können Code_1^n so wählen daß auch jede Komponentenfunktion der Umkehrfunktion, $x \mapsto [\text{Code}_1^n)^{-1}(x)]_i$, prim.r. ist.

Satz 3.5. *Alle prim.r. Funktionen sind berechenbar.*

3.6 Beweisskizze Das ist sehr leicht (wenn auch etwas umständlich) nachzurechnen. Man muß z.B. zeigen: Wenn P ein URM-Programm für f und Q eines für g ist, dann gibt es ein Programm das $x \mapsto g(f(x))$ berechnet. Das ist aber klar: man schreibt die Programme P und Q hintereinander, nummeriert die Zeilen von Q neu (dabei bekommt die alte Zeile 1 die Nummer l) und ändert alle "`return` " in P zu "`goto` l".

Nicht alle berechenbaren Funktionen sind prim.r. Grund:

3.7 Diagonalisierung Gegeben eine effektive Aufzählung berechenbarer Funktionen. Dann gibt es berechenbare Funktion nicht in diese Aufzählung.

Das wird etwas klarer werden nachdem wir Gödelnummern eingeführt haben, siehe 6.5.

3.8 Beispiel: Ackermann Funktion (berechenbar, aber nicht prim.r.)

$$A(m,n) := \begin{cases} n+1 & \text{wenn } m = 0, \\ A(m-1,1) & \text{wenn } m > 0 \text{ und } n = 0, \\ A(m-1, A(m, n-1)) & \text{sonst.} \end{cases}$$

Es ist also jede prim.r. Funktion berechenbar, aber nicht jedes Programm, das einer URM entspricht, ist prim.r. Es gilt aber, daß das Programmergebnis *bis zur Zeit t* (wobei t als Input gegeben wird) sehr wohl prim.r. ist. In der Notation von Definition 2.7:

Satz 3.9. *Wenn P ein URM-Programm ist, dann ist die Funktion $f : \mathbb{N}^{n+1} \to \mathbb{N}^m$, definiert durch $(t, x_0, \ldots, x_{n-1}) \mapsto P^t(x_0, \ldots, x_{n-1})$, prim.r.*

Damit ist gemeint daß jede Komponentenfunktion der Funktion f prim.r. ist, oder äquivalent daß die Funktion $g := \mathrm{Code}_1^k \circ f$ prim.r. ist.

3.10 Anstelle eines Beweises Das ist etwas mühsam, aber nicht sehr originell. Insbesondere muß man zeigen daß bestimmte Fallunterscheidungen und Kodierungen von Tupeln in natürliche Zahlen prim.r. sind. In der Literatur wird der Beweis oft nicht direkt geführt, sondern über dem Umweg z.B. der Turing-Maschine. (Das hat den Vorteil daß man so gleich mit beweist daß Turing Maschinen und URMs äquivalent sind.)
Wir werden in 5.4(7) eine stärkere Version dieses Satztes kennenlernen, nämlich eine "uniforme" für alle Programme (das Programm wird als Input-Parameter übergeben).

4. Partiell rekursive (p.r.) Funktionen: Endlosschleifen, μ-rekursive Funktionen

Ein ganz wesentlicher Punkt, den wir bis jetzt noch nicht explizit angesprochen haben, ist folgender: URM Programme müssen nicht bei jedem Input terminieren (d.h. die Berechnung beenden und einen Output liefern).

Man könnte meinen, daß es sich in solchen Fällen immer um "Programmierfehler" handelt. Man könnte sogar versuchen, eine Programmiersprache zu entwickeln, in der solche Endlosschleifen gar nicht erst auftreten können. Es stellt sich aber heraus, daß das nicht möglich ist. Es gibt (partielle) Funktionen, die man nur mit Programmen berechnen kann, die auf bestimmten Inputs nicht halten, siehe 6.6.

Beispiel 4.1. Das folgende Programm hält genau bei Input 0:

```
1  if R_0 = 0 goto 3
2  goto 2
3  return
```

Definition 4.2.
- F ist *partielle Funktion* von $\mathbb{N}^n$ nach $\mathbb{N}$, wenn $F : A \to \mathbb{N}$ für ein $A \subseteq \mathbb{N}^n$. $A := \mathrm{dom}(F)$.
- F *total* wenn $\mathrm{dom}(F) = \mathbb{N}^n$. (D.h. wenn klassisch $F : \mathbb{N}^n \to \mathbb{N}$.)
- F ist *partiell rekursiv (p.r.)*, wenn es ein URM Programm gibt daß auf Input $\vec{x}$ genau dann einen Output liefert wenn $\vec{x} \in \mathrm{dom}(F)$ und in diesem Fall wird der Output $F(\vec{x})$ geliefert.
- F is *total rekursiv (t.r.)* wenn p.r. und total (das haben wir bisher berechenbar genannt).

Beispiel 4.3. Das folgende F ist p.r.:

$$F(n) := \begin{cases} 0 & \text{wenn n=0,} \\ \text{undefiniert} & \text{sonst.} \end{cases} \qquad \mathrm{dom}(F) = \{0\}.$$

Jede prim.r. Funktion ist total (und daher t.r.): Die primitiv rekursiven Grundfunktionen sind total, und die primitiv rekursiven Operationen erzeugen aus totalen Funktionen immer totale Funktionen.

Um nicht-totale Funktionen zu bekommen, führen wir die μ-Operation ein. Die Klasse der μ-rekursiven Funktionen ist wieder induktiv definiert:

DEFINITION 4.4 (der μ-rekursive (paritellen) Funktionen).

- Jede prim.r. Funktion ist μ-r.
- Sei $f : \mathbb{N}^{n+1} \to \mathbb{N}$ prim.r. Dann ist μf ebenfalls μ-r., wobei $(\mu f)(\vec{x}) = \mu_m(f(m, \vec{x})) := \min\{m : \ f(m, \vec{x}) = 0\}$.
- Die Zusammensetzung von μ-r. Funktionen ist wieder μ-r.

Man kann die μ-rekursiven Funktionen auch als abgeschlossen unter μ-Rekursionen definieren, wobei man dann allerdings die Definition von μf für nicht totale Funktionen leicht modifizieren muß:
$(\mu f)(\vec{x}) = \mu_m(f(m, \vec{x})) := \min\{m : \ f(m, \vec{x}) = 0$ und $f(l, \vec{x})$ ist definiert für alle $l < m\}$.

SATZ 4.5. *f ist p.r. $\leftrightarrow$ f ist μ-r.*

4.6 BEWEISSKIZZE: Eine Richtung ist wieder recht einfach: Wenn P ein URM Programm ist, kann man ein URM Programm Q schreiben daß P zuerst auf Input 0 ausführt, dann auf 1 etc, solange bis P den Output 0 liefert. Dann wird der Input für P als Output von Q ausgegeben.
Die andere Richtung verwendet 3.9: Sei P ein URM-Programm. $P^t(\vec{x})$ ist der Zustand der URM zum Zeitpunkt t (wenn P auf Input $\vec{x}$ berechnet wird). $(t, \vec{x}) \mapsto P^t(\vec{x})$ ist prim.r. Um den Output von P zu berechnen suchen wir das kleinste t mit der folgenden Eigenschaft: die erste Koordinate des Tupels $P^t(\vec{x})$ ist 0. Das ist genau eine Anwendung eines μ-Operators. Nennen wir diese μ-r. Funktion $\mathrm{HT}_P(\vec{x})$ (für halting time). Dann ist der Output von P genau die zweite Komponente des Tupels $P^{\mathrm{HT}_P(\vec{x})}(\vec{x})$. Also ist die Funktion die durch P berechnet wird μ-rekursiv.

Die Begriffe μ-rekursiv und URM-berechenbar stimmen also überein. Wie schon erwähnt hat sich gezeigt, daß alle "natürlichen" Berechenbarkeits-Definitionen zum selben Begriff führen. Diese Erkenntnis, daß es einen natürlichen und universellen Begriff der Berechenbarkeit gibt, wird oft als These von Church bezeichnet:

4.7 CHURCH'SCHE THESE Genau die URM-berechenbaren Funktionen sind (im "natürlichen Sinn") berechenbar.

5. Das universelle Programm (Interpreter), Gödel-Nummerierungen

Es gibt ein URM-Programm U, das eine URM *simuliert*.

Dieses Programm U verwendet als Input:

- ein URM-Programm P, und
- zusätzlichen Input x (für P).

Als Output wird geliefert:

- Der Output von P auf Input x.

Wenn P auf x nicht terminiert, dann terminiert U nicht auf (P, x).

Nicht neues für Informatiker: Man kann in perl einen perl-Interpreter schreiben, in C einen C-Interpreter, man kann einen ganzen Computer als Computerprogramm simulieren etc.

Das konkrete URM Programm U wäre natürlich furchtbar lang und uninteressant. (U wird daher üblicherweise auch nie aufgeschrieben, sondern es wird nur seine Existenz bewiesen.)

Um Programme als Input verwenden zu können, müssen wir sie als Zahlen *kodieren*!

5.1 Codierung/Gödel-Nummerierung: Wir ordnen jedem URM-Programm "auf vernünftige Weise" eine (eindeutige) Nummer zu.

Wir haben also eine injektive Abbildung von den Programmen nach $\mathbb{N}$. Wir werden und in weiterer Folge eher für die Umkehrung dieser Abbildung interessieren: Einem $e \in \mathbb{N}$ ordnen wir das Programmen mit Nummer e zu.

Beispiel 5.2.
- Schreibe Programm als ASCII-Text. Das gibt Folge von Bytes (1 Byte=Zahl zwischen 0 und 255), d.h. Zahl zur Basis 256.
- "effizienter": Die Formale Definition eines Programms ist ja eine endliche Menge von 4-Tuplen. Wir können ein 4-Tupel $\vec{v} = (a, b, c, d)$ als $c(\vec{v}) = 2^a \cdot 3^b \cdot 3^c \cdot 3^d$ kodieren, und eine endliche Menge $\{\vec{v}_1, \ldots, \vec{v}_l\}$ als $\prod p_i^{c(\vec{v}_i)}$. Dabei bezeichnet p_i die i-te Primzahl.
- bijektiv: Generiere automatisch der Reihe nach alle Programme und zähle sie ab.

Die ersten beiden Beispiele bilden die Programme nicht surjektiv auf $\mathbb{N}$ ab, aber wir können zumindest leicht (d.h. effektiv) entscheiden, ob eine Nummer e der Code eines Programms ist. Falls nicht, ordnen wir der Nummer e ein beliebiges fixes Programm zu, z.B. das Programm "1 `goto` 1" (welches nie hält, d.h. die leere partielle Funktion berechnet).

So bekommen wir eine *surjektive* Abbildung von $\mathbb{N}$ in Programme.

Definition 5.3 (Gödelnummer). φ_e^n ist die n-stellige p.r. Funktion, die dem URM-Programm mit Nummer e entspricht. $\varphi_e \coloneqq \varphi_e^1$.

D.h.: $\varphi_e^n(\vec{x}) = y$ gdw das Programm Nr. e auf Input $\vec{x}$ den Output y liefert.

Nach Definition gilt daher trivialerweise: $\varphi_e^0() = \varphi_e^1(0) = \varphi_e^2(0,0)$, $\varphi_e^1(x) = \varphi_e^2(x,0)$ etc.

Satz 5.4 (Grundsätzliche Eigenschaften der Gödelnummern:).

(1) *$e \mapsto \varphi_e^n$ bildet $\mathbb{N}$ surjektiv auf Menge der p.r. Funktionen $\mathbb{N}^n \to \mathbb{N}$ ab.*
(2) *$(e, x) \mapsto \varphi_e^1(x)$ ist eine 2-stellige p.r. Funktion, d.h. $\exists u\, \forall e\, \varphi_u^2(e, x) = \varphi_e^1(x)$*
Allgemeiner:
(3) *Enumeration Thm: $\forall n\, \exists u\, \forall e\, \varphi_u^{n+1}(e, \vec{x}) = \varphi_e^n(\vec{x})$.*
(4) *Input kann effektiv ins Programm kodiert werden: Es gibt prim.r. Funktion S s.t. $\varphi_e^1(x) = \varphi_{S(e,x)}^0$.* *Allgemeiner:*
(5) *S_m^n Thm: Für n, m gibt es prim.r. Funktion S_m^n s.t. $\forall e\, \varphi_e^{n+m}(\vec{x}, \vec{y}) = \varphi_{S_m^n(e,\vec{x})}^m(\vec{y})$.*
(6) *Die Funktion $(e, t, \vec{x}) \mapsto P_e^t(x_0, \ldots, x_{n-1})$ ist prim.r.*
(7) *Normalformesatz: Es gibt für alle n prim.r. Funktionen $R0^n : \mathbb{N} \to \mathbb{N}$ und $HT^n : \mathbb{N}^{n+2} \to \mathbb{N}$ s.d. $\varphi_e^n(\vec{x}) = R0^n(\mu_t HT^n(e, t, \vec{x}))$.*

Der 2. Punkt besagt eben nichts anderes, als daß es ein universelles Programm gibt.

5.5 Anstelle eines Beweises Wir haben φ so definiert daß (1) erfüllt ist.
(4) folgt aus (5).
(5) ist recht einfach: S_m^n entspricht folgendem Programm: Als Input gegeben ist der Programmcode e und $(x_0, \ldots, x_{n-1})$, als Output generiere ein Code des folgenden Programms:

- Schreibe den Inhalt der Register $(R_0, \ldots, R_{m-1})$ in die Register $(R_n, \ldots, R_{n+m-1})$,
- schreibe $(x_0, \ldots, x_{n-1})$ in $(R_0, \ldots, R_{n-1})$,

- der Rest des Programms ist einfach dasselbe Programm wie e, nur mit neu numerierten Zeilen.

Hier wird das e-te Programm nicht simuliert, es werden nur einfache syntaktische Modifikationen am Programm-Code vorgenommen.
(2) folgt aus (3), und (3) folgt aus (7) und (1). (Warum?)
(6) zu beweisen ist recht mühsam. Es handelt sich dabei um die uniforme Version von 3.9. Mit (6) beweist man (7) genauso wie in 4.5.

6. Unberechenbare Funktionen

Verschiedene Arten, unberechenbare Funktionen zu finden/konstruieren:

6.1 KARDINALITÄT: Es gibt nur abzählbar viele Programme, aber überabzählbar viele Funktionen $\mathbb{N} \to \mathbb{N}$. Daher gibt es unberechenbare Funktionen.

6.2 KLASSISCHE DIAGONALISIERUNG:
$g(n) := \begin{cases} \varphi_n(n) + 1 & \text{falls } \varphi_n(n) \text{ definiert,} \\ 0 & \text{sonst} \end{cases}$ ist nicht rekursiv.

(Beweis: Sonst wäre $g = \varphi_e$, d.h. $g(e) = \varphi_e(e)$.)

g mod 2 ist eine nicht rekursive 0-1-Folge.

6.3 HALTEPROBLEM Die Frage "ist $\varphi_n(n)$ definiert" ist nicht effektiv entscheidbar.

(Beweis: Sonst wäre das obige g wegen 5.4(2) berechenbar.)

6.4 DIAGONALISIERUNG ÜBER GRÖSSE (BUSY BEAVER): $g(n) := \max\{\varphi_e(0) : e \le n,\ \varphi_e(0) \text{ definiert}\}$.

Eine analoge Funktion, die URM-Programme direkt verwendet:
Betrachte alle URM-Programme mit maximal n Zeichen (im wesentlichen endlich viele). Sei $h(n)$ der größte Output eines solchen Programms bei Input 0. Die Funktion $n \mapsto h(n)$ ist nicht rekursiv.

Diese Funktionen wachsen stärker als alle rekursiven Funktionen, d.h. für jedes p.r. f gibt es ein n s.d. $g(m) > f(m)$ für alle $m > n$ in $\mathrm{dom}(f)$.

(Ohne Beweis.)

6.5 TOTALE, EFFEKTIVE FUNKTIONENKLASSE Sei f t.r. so daß $\varphi_{f(e)}$ total ist für jedes e. Dann ist g, definiert durch $g(n) := \varphi_{f(n)}(n) + 1$, eine t.r. Funktion und ungleich jedem $\varphi_{f(e)}$.

Das zeigt daß es für jede effektiv aufzählbare Klasse totaler Funktionen eine totale berechenbare Funktion gibt die nicht in der Klasse liegt. Es ist auch leicht zu zeigen daß die Klasse der prim.r. Funktionen so eine Klasse ist: Es gibt ein t.r. f s.d. g prim.r. genau dann wenn $g = \varphi_{f(n)}$ für eine $n \in \mathbb{N}$. (Achtung: Es ist aber nicht möglich, ein rekursives f zu finden so daß φ_e prim.r. genau dann wenn $e = f(n)$ für eine $n \in \mathbb{N}$!)

6.6 EINE P.R. FUNKTION DIE KEINE T.R. ERWEITERUNG HAT Die p.r. Funktion $g : e \mapsto \mu_t \mathrm{HT}^1(e, t, e)$ aus dem Normalformesatz 5.4(7) hat keine t.r. Erweiterung, d.h. es gibt kein f t.r. die g fortsetzt.

(Beweis: g gibt im wesentlichen den Zeitpunkt an, zu dem das Programm e auf input e terminiert. Gäbe es eine totale Funktion f die g fortsetzt (oder auch nur überall größer wäre als g), dann könnte man entscheiden, ob $\varphi_e(e)$ terminiert, weil man die Berechnung nur bis zur oberen Schranke $f(e)$ durchführen muß.)

7. Rekursive und rekursiv aufzählbare Mengen

Das Halteproblem $H = \{e : \varphi_e(0) \text{ definiert}\}$ ist nicht entscheidbar (oder: die charakteristische Funktion von H ist nicht berechenbar.)

Man kann also nicht entscheiden *ob* ein Programm hält.

Aber wenn ein Programm hält, dann kann man natürlich feststellen *daß* das Programm hält.

Man sagt: H ist rekursiv aufzählbar (r.e.), d.h. man kann ein Programm angeben das alle (auf Input 0) haltenden Programm-Nummern ausgibt / auflistet.
Man kann sich dabei z.B. ein Programm vorstellen das als Output eine unendliche Liste mit den Codes aller haltenden Programme generiert. (N.B.: diese Liste kann nicht monoton wachsend sein). Unendliche Listen als Outputs haben wir aber nicht vorgesehen, daher verwenden wir das äquivalente Konzept: Das Programm gibt auf Input n das n-te Element der Liste als Output aus. Als offizielle Definition von rekursiv aufzählbar verwenden wir daher:

Definition 7.1.
- $A \subseteq \mathbb{N}$ ist *rekursiv aufzählbar* (r.e.), wenn $A = \varphi_e''\mathbb{N}$ für ein e.
- Im Unterschied dazu: $A \subseteq \mathbb{N}$ ist *rekursiv*, wenn die charakteristische Funktion von A berechenbar (t.r.) ist.

Dabei verwenden wir die Notation $f''A := \{f(x) : x \in A \cap \mathrm{dom}(f)\}$. "Ist Problem P entscheidbar (rekursiv)" ist äquivalent zu: ist die charakteristische Funktion von P berechenbar/p.r./t.r.
A ist rekursiv heißt: Es gibt ein Programm daß bei Input n entscheidet ob $n \in A$ oder nicht (und das Programm terminiert immer).

Rekursiv aufzählbar ist so etwas wie "halb rekursiv".

Satz 7.2. *Äquivalent sind ($A \subseteq \mathbb{N}$):*

- *A r.e., d.h. A ist Bild einer p.r. Funktion,*
- *$A = \emptyset$ oder A Bild einer t.r. Funktion,*
- *$A = \mathrm{dom}(\varphi_e)$ für ein e, d.h. A domain einer p.r. Funktion,*
- *$A = Pr(B)$ für ein $B \subseteq \mathbb{N} \times \mathbb{N}$ rekursiv.*

Dabei ist $\mathrm{Pr}(B) = \{n : \exists m : (n, m) \in B\}$, und $B \subseteq \mathbb{N} \times \mathbb{N}$ ist rekursiv wenn es die charakteristische Funktion ist.

Wenn man die Begriffe r.e. und rekursiv einmal verstanden hat, sind die folgenden Eigenschaften völlig klar:

- Wenn A und B r.e., dann auch $A \cup B$ und $A \cap B$.
- Wenn A r.e. und unendlich dann gibt es ein injektives t.r. f mit $f''\mathbb{N} = A$.
- A ist rekursiv genau dann wenn A r.e. und $\mathbb{N} \setminus A$ r.e.
- Wenn A r.e. dann ist $\mathbb{N} \setminus A$ i.a. nicht r.e.
- r.e. bleibt erhalten unter p.r. Bildern und Urbildern.
- Wenn A und B r.e., dann gibt es disjunkte r.e. $A' \subseteq A$ und $B' \subseteq B$ s.d. $A' \cup B' = A \cup B$.
- Es gibt X unendlich ohne unendliche r.e. Teilmenge.

Die Beweise von 7.2 und den obigen Eigenschaften sind alle sehr einfach und verwenden im wesentlichen nur 5.4(6). Der letzte Punkt ist wieder eine Diagonalisierung: Zähle alle unendlichen r.e. Mengen als $A_1, A_2, \ldots$ auf (das kann man nicht effektiv!), und wähle $X = \{x_i : i \in \mathbb{N}\}$ so daß im Intervall $[x_i + 1, x_{i+1} - 1]$ mindestens ein Element von A_i liegt.

Aber nicht alle elementare Eigenschaften von r.e. Mengen sind völlig klar. Die folgenden Sätze sind etwas schwieriger:

- Wenn A r.e. und unendlich, dann hat A eine rekursive unendliche Teilmenge.
- Wenn A, B r.e. und disjunkt, dann gibt es i.a. kein rekursives C s.d. $A \subseteq C$ und C disjunkt zu B.

8. Anwendungen: Algorithmisch unlösbare Probleme

Man kann zeigen, daß bestimmte Probleme nicht effektiv entscheidbar sind, indem man zeigt daß sie genauso schwer sind wie das Halteproblem.

Definition 8.1. A lässt sich auf B reduzieren ($A \leq_m B$), wenn es ein t.r. $f : \mathbb{N} \to \mathbb{N}$ gibt s.d. $n \in A$ gdw $f(n) \in B$.

Wenn also H (die Haltemenge) auf eine Menge A reduzierbar ist, dann kann A nicht rekursiv sein (sonst wäre H rekursiv).

Damit kann man die effektive Unlösbarkeit vieler Problem aus der Mathematik zeigen. Als Beispiele führen wir hier an:

8.2 nichtrekursive r.e. Mengen

- Ableitbarkeit in Prädikatenlogik.
- 10. Hilbert'sches Problem: Hat das ganzzahlige Polynom $f(X_1, \ldots, X_n)$ ganzzahlige Nullstellen?
- Wortproblem in Gruppen.

▸ Ableitbarkeit in der Prädikatenlogik

Im Kapitel über Prädikatenlogik werden wir sehen, daß es ein vollständiges Ableitungssystem für first order Sätze gibt. Es gibt also ein Computerprogramm, das alle wahren first order Sätze ableitet. Die Menge der wahren Sätze ist also r.e. Andrerseits ist es nicht *entscheidbar*, ob ein Satz gültig (oder äquivalent: ableitbar) ist.

Der Beweis ist nicht sehr schwer und verwendet die Tatsache, daß man eine Berechnung in first order Formeln kodieren kann, wenn man zumindest ein zweistelliges Prädikatensymbol zur Verfügung hat.

▸ Wortproblem in Gruppen

Sei G eine Gruppe. Wort: Formales Produkt (inklusive Inverses). $M \subseteq G$ erzeugt G, wenn jedes $a \in G$ ein Wort in M ist.

Beispiel 8.3. S_3 lässt sich als Dreh-Spiegelungen vom Dreieck schreiben, daher ist S_3 erzeugt durch Spiegelung s und Rotation r. Ein Wort in s, r ist z.B. $rsr^{-1}s^{-1}sr^{-1}srr^{-1}r^{-1}s^{-1}rr$.

Jede endlich erzeugte Gruppe kann durch die Menge der Erzeuger und eine Menge von Gleichungen charakterisiert werden (group presentation).

Beispiel 8.4. Die Gleichungen $ss = e$, $rrr = e$ und $srs = r^{-1}$ definieren S_3. Dann gilt z.B. $rs = sr^{-1}$ und $s = rrsr^{-1}$ etc.

Frage: Gibt es einen Algorithmus, der folgendes entscheidet: Gegeben endlich viele Erzeuger, endlich viele Gleichungen und ein Wort w. Ist $w = e$?

Auch hier ist die Menge der Wörter w so daß $w = e$ offenbar aufzählbar. (Wenn $w = e$, dann gibt es dafuer eine endliche Ableitung im Gleichungskalkül, und man kann alle Ableitungen einfach der Reihe nach aufzählen.) Aber auch hier kann man (durch Kodieren von Berechnungen in Gruppen) zeigen, daß das Problem nicht rekursiv ist.

Beachte daß es für bestimmte Arten von Gruppen natürlich schon Algorithmen gegen kann. (Trivialerweise z.B. für alle abelschen oder endlichen Gruppen.) Es kann nur keinen Algorithmus geben, der für *alle* endlich präsentierten Gruppen funktioniert.

► 10. Hilbertsches Problem

Sei $f(X_1, \ldots, X_n) \in \mathbb{Z}[X_1, \ldots, X_n]$ ein Polynom mit ganzzahligen Koeffizienten. Hat f eine ganzzahlige Nullstelle? D.h. gibt es $X_1, \ldots, X_n$ aus $\mathbb{Z}$ s.d. $f(X_1, \ldots, X_n) = 0$? Das ist also die Frage nach der Lösbarkeit allgemeiner Diophantischer Gleichungen, ein zentrales Problem der Zahlentheorie.

Für ein Polynom $f(X) \in \mathbb{Z}[X]$ in nur einer Variable ist diese Frage übrigens sehr leicht entscheidbar. Das allgemeine Problem ist offebar r.e.: Wenn es eine Nullstelle gibt, kann man sie durch systematisches Ausprobieren immer finden.

Durch ein nicht triviales Argument in (elementarer) Zahlentheorie kann man zeigen, daß sich das Halteproblem in diese Frage übersetzen läßt: Man kann jedem URM Programm effektiv ein Polynom f (in 7 Variablen) zuordnen, so daß das Programm auf Input 0 hält genau dann wenn f eine ganzzahlige Nullstelle hat.

Dieser Beweis wurde 1970 von Matijasevic gefunden, aufbauend auf Arbeiten von Robinson, Davis und Putnam. Siehe z.B. den sehr schönen Artikel J. P. Jones und Y. V. Matijasevic, *Proof of recursive unsolvability of Hilbert's tenth problem*, Amer. Math. Monthly 98 (1991), no. 8, 689–709.

Das ist eine sehr starke Form des Gödelschen Unvollständigkeitssatzes: Es kann nicht einmal ein effektives Beweissystem geben, das die Unlösbarkeit jeder unlösbaren diophantischen Gleichung zeigt. (Andernfalls wäre das 10. Hilbertsche Problem rekursiv: Die in einem effektiven Beweissystem ableitbaren Sätze sind r.e., und die lösbaren diophantischen Gleichungen sind ebenfalls r.e., und wenn A und $\mathbb{N} \setminus A$ beide r.e. sind dann ist A r.e.)

9. Weiterführendes

Dieser Abschnitt ist kein Prüfungsstoff.

► Der Fixpunktsatz

9.1 Fixpunktsatz (Kleene) Für alle p.r. f gibt es e mit $\varphi_e = \varphi_{f(e)}$.
(Dabei wird, falls $f(e)$ nicht definiert ist, $\varphi_{f(e)}$ als die leere partielle Funktion definiert.)

Der Fixpunktsatz ist in der Rekursionstheorie von zentraler Bedeutung.

Eine einfache Folgerung ist, daß es immer Programme gibt die Ihren eigenen Code ausgeben: f sei die (totale) Funktion, die jedem x den Code eines Programms zuweist, das auf jedem beliebigen Input den Output x ausgibt. Der Fixpunktsatz liefert also ein Programm φ_e das den Output e liefert. Das gilt für jede beliebige Gödelnummerierung, solange sie nur die Eigenschaften in 5.4 erfüllt.

Eine andere schöne Folgerung des Fixpunktsatzes ist z.B. der Satz von Rice:

9.2 Satz von Rice Sei $\emptyset \neq C \subsetneq \{f : \mathbb{N} \to \mathbb{N} \text{ p.r.}\}$ eine nicht-triviale Klasse partiell rekursiver Funktionen. Dann ist $\{e : \varphi_e \in C\}$ nicht rekursiv.

Wir wissen ja schon, daß die Haltemenge $H := \{e : \varphi_e(0) \text{ definiert}\}$ nicht rekursiv (aber r.e.) ist. Der Satz von Rice sagt nun, daß keine nichttrivale Funktionenmenge eine rekursive Code-Menge hat. Man kann also weder effektiv entscheiden ob φ_e unendlichen Definitionsbereich hat, noch ob φ_e leeren Definitionsbereich hat, noch oder ob 0 im Wertebereich von φ_e liegt etc.

(Der Satz von Rice läßt sich übrigens nur auf externe Eigenschaften anwenden, d.h. auf Eigenschaften der partiellen Funktion, nicht Eigenschaften die die Gödelnummer betreffen. Der Satz von Rice beweist also z.B. nicht, daß es unentscheidbar ist ob $\varphi_e(0) = e$.)

▸ Komplexitätstheorie

Wir haben bisher gefragt: Welche Probleme sind entscheidbar?

Die Komplexitätstheorie beschäftig sich mit der Frage: Welche Probleme sind leicht, welche nur schwer zu lösen? Man beginnt sinnvollerweise mit der Frage: Wie misst man die Komplexität eines Programms/Algorithmus?

Dabei gibt es natürlich viele Varianten: Sei P ein Programm. Der Einfachkeit halber nehmen wir an daß P auf jedem Input hält. Üblicherweise denkt man sich den Input x als Binärzahl der Länge n gegeben, d.h. der Input x hat Größe $n \approx \ln(x)$.

- Üblicherweise wird der Zeitaufwand betrachtet: Wie lange dauert die Berechnung von $P(x)$, als Funktion von n? (Manchmal wird aber auch der Speicherplatzbedarf untersucht.)
- Man kann den worst case oder den average case untersuchen: Was ist die maximale Berechnungsdauer von $P(x)$ unter allen x mit Länge n, oder was ist die durchschnittliche?
- Bei average case kann man auch verschiedene Gewichte für die x mit Länge n wählen.

Beispiel 9.3.

- ggT: Der trivialer Algorithmus ist exponentiell (linear in Größe der Zahl x, d.h. exponentiell in Länge n der Zifferndarstellung).
  Der Euklidischer Algorithmus ist linear (in Länge der Zifferndarstellung).
- Dasselbe gilt für Primzahlentest: Der trivale Algorithmus, der testet ob x eine Primzahl ist, braucht $\sqrt{x} \approx 2^{n/2}$ Schritte. Erst unlängst wurde ein polynomieller Algorithmus gefunden. (Genauer: Die Riemannsche Vermutung vorausgesetzt gab es schon länger solche Algorithmen.)
- Es scheint aber schwieriger zu sein, einen Faktor einer nicht-Primzahl zu finden (der triviale Algorithmus ist wieder exponentiell). Das wird für Kryptographie verwendet: Man kann relativ leicht zwei Primzahlen p_1, p_2 finden, aber aus dem Produkt $p_1 \cdot p_2$ läßt sich nur schwer p_1 zurückgewinnen.

Wenn man zwei Algorithmen (oder Implementierungen auf verschiedenen Computermodellen) vergleicht, interessiert man sich in der Theorie nie für konstante Unterschiede (z.B.: P' 100x so schnell wie P), obwohl das in der Praxis natürlich relevant sein kann. Man interessiert sich nur um die "groß-O" Klasse:

Definition 9.4. $f(n) \in O(g(n))$ heißt: $(\exists c > 0) f(n) < c \cdot (g(n) + c)$.

In der Komplexitätstheorie gibt es zwar (im Unterschied zur Rekursionstheorie) schon Abhängigkeit vom Computermodell, aber viele Klassen sind immer noch erstaunlich robust (z.B. Funktionen die in polynomieller oder exponentieller Zeit berechnet werden können).

Verschiedene Programmiersprachen machen i.A. fast keinen Unterschied: C mag 1000x so schnell sein wie Basic, aber das ist eben nur ein linearer (und daher vernachlässigter) Unterschied.

Bisher haben wir von der Komplexität eines Algorithmus gesprochen. Daraus ergibt sich der Begriff der Komplexität einer t.r. Funktion f: f ist "schnell" zu berechnen, wenn es einen Algorithmus P gibt der f "schnell" berechnet.

Einen optimalen Algorithmus für f gibt definieren wir folgendermaßen:

DEFINITION 9.5. P ist ein optimaler Algorithmus für f, wenn P f berechnet und für alle anderen Q, die f berechnen, gilt: $K(P) \in O(K(Q))$.

Dabei ist K ein Komplexitätsmaß für, d.h. $K(P)$ bildet die Input-Länge auf den Berechnungsaufwand von P ab.

Viele Probleme haben optimale Algorithmen. Ein Beispiel für einen optimalen Sortieralgorithmus ist heapsort. (Dabei muß man aber natürlich erst mal K spezifizieren.)

Man kann aber auch zeigen:

9.6 SPEED UP THEOREM Für jedes Komplexitätsmaß (und jedes Computermodell) gibt es eine berechenbare totale Funktion, für die es keinen optimalen Algorithmus gibt.

► Kolmogorov-Komplexität

Ein anderer Begriff der Komplexität (nahe zur Informationstheorie):

9.7 KOLMOGOROV-KOMPLEXITÄT KK(n) ist die Größe des kleinsten URM-Programms, das bei Input 0 den Output n liefert.

- Die Funktion $n \to \mathrm{KK}(n)$ ist nicht rekursiv.
- Z.B. $\mathrm{KK}(10^{10^{10}})$ ist sehr klein.
- KK(n) ist höchstens (in etwa) $\log(n)$ (Basisdarstellung).
- Es gibt (viele) n mit $\mathrm{KK}(n) \approx \log(n)$ (einfache Abzählung).
- KK mißt gewissermaßen Informationsgehalt von n.

► Turing-Grade, Orakel-Berechnung

- Wir können URM' definieren: Wie URM, aber Halteproblem als zusätzliche Grundfunktion.
- Für URM' könen wir ganz analoge Rekursionstheorie entwickeln (eigenes Halteproblem H', Gödel-Nummerierung, Fixpunktsatz,...).
- Allgemeiner: Sei $B \subseteq \mathbb{N}$. Definiere URM^B: Verwende char. Fkt von B als Grundfunktion.
- $A \leq_T B$ (A ist Turing-reduzierbar auf A): A ist URM^B-rekursiv.
- $A =_T B$ heißt $A \leq_T B$ und $B \leq_T A$.
- Turing Grade: Faktorisiere $\mathfrak{P}(\mathbb{N})$ nach $=_T$.
- Turing Grade sind sehr interessante (und komplizierte) Halbordnung.
- Z.B.: Je zwie Grade haben kleinste obere Schranke, aber i.a. keine größte untere.

► Physikalische Computer

- Speicher realer Computer endlich.
- Daher: Reale Computer "schwächer" als idealisierte.
- Aber: Es gibt auch physikalische "Idealisierung".
- Z.B.: URM läßt sich physikalisch (idealisiert) realisieren.
- Denkbar: Nicht-URM-berechenbare Fkt physikalisch berechenbar.
- Z.B. Maschine die jeder Berechnungsschritt doppelt so schnell ausführt wie den vorigen. So eine Maschine könnte offenbar das Halteproblem lösen.
- Es ist unklar ob y.B. ein Quantencomputer nicht URM-berechenbare Funktionen berechnen könnte.
- Quantencomputer kann aber schnell faktorisieren (URM kann das vermutlich nicht, ist aber noch nicht bewiesen).

Beispiele für Prüfungsfragen

(Zur Erinnerung: p.r. heißt partiell rekursiv, t.r. total rekursiv, und r.e. rekursiv aufzählbar.)

Vorausgesetzt A ist eine unendliche Menge, welche der folgenden Aussagen ist äquivalent zu "A ist r.e."

- ☐ A ist Bild einer p.r. Funktion.
- ☐ A ist Definitionsbereich einer p.r. Funktion.
- ☐ A ist Bild einer t.r. Funktion.
- ☐ A ist Definitionsbereich einer t.r. Funktion.
- ☐ A ist Bild einer injektiven t.r. Funktion.
- ☐ A ist Bild einer streng monoton wachsenden t.r. Funktion.
- ☐ A und $\mathbb{N} \setminus A$ sind beide r.e.
- ☐ Die charakteristische Funktion von A ist p.r.
- ☐ Die charakteristische Funktion von A ist t.r.

Vorausgesetzt A ist eine unendliche Menge, welche der folgenden Aussagen ist äquivalent zu "A ist rekursiv"

- ☐ A ist Bild einer p.r. Funktion.
- ☐ A ist Definitionsbereich einer p.r. Funktion.
- ☐ A ist Bild einer t.r. Funktion.
- ☐ A ist Definitionsbereich einer t.r. Funktion.
- ☐ A ist Bild einer injektiven t.r. Funktion.
- ☐ A ist Bild einer streng monoton wachsenden t.r. Funktion.
- ☐ A und $\mathbb{N} \setminus A$ sind beide r.e.
- ☐ Die charakteristische Funktion von A ist p.r.
- ☐ Die charakteristische Funktion von A ist t.r.

Seien A, B r.e. und C, D rekursiv Welche der folgenden Aussagen ist richtig:

- ☐ $A \cup B$ ist r.e.
- ☐ $C \cup D$ ist r.e.
- ☐ $C \cup D$ ist rekursiv
- ☐ $A \cup C$ ist r.e.
- ☐ $A \cup C$ ist rekursiv
- ☐ $A \cap B$ ist r.e.
- ☐ $C \cap D$ ist r.e.
- ☐ $C \cap D$ ist rekursiv
- ☐ $A \cap C$ ist r.e.
- ☐ $A \cap C$ ist rekursiv
- ☐ $A \setminus B$ ist r.e.
- ☐ $C \setminus D$ ist r.e.
- ☐ $C \setminus D$ ist rekursiv
- ☐ $A \setminus C$ ist r.e.
- ☐ $A \setminus C$ ist rekursiv

P ist die Menge der Primzahlen. Welche der folgenden Mengen ist r.e.:

- ☐ P
- ☐ $\{n : \varphi_n(0) \text{ definiert}\}$
- ☐ $\{n : \varphi_n(n) \text{ definiert}\}$
- ☐ $\{n : \varphi_n(\varphi_n(n)) \text{ definiert}\}$
- ☐ $\{n : \text{dom}(\varphi_n) \text{ ist endlich}\}$
- ☐ $\{n : \text{dom}(\varphi_n) \text{ ist unendlich}\}$
- ☐ $\{n : \text{dom}(\varphi_n) \text{ ist nichtleer}\}$
- ☐ $\{n : \varphi_n \text{ ist eine totale Funktion}\}$
- ☐ $\{n : (\exists p \in P) : \varphi_n(p) \text{ definiert}\}$
- ☐ $\{n : (\exists p \in P) : \varphi_p(n) \text{ definiert}\}$
- ☐ $\{n : (\forall p \in P) : \varphi_n(p) \text{ definiert}\}$
- ☐ $\{n : (\forall p \in P) : \varphi_p(n) \text{ definiert}\}$
- ☐ $\{n : \varphi_n(n) = \varphi_{n+1}(\varphi_n(n))\}$

Seien A und B r.e. P ist die Menge der Primzahlen. Welche der folgenden Mengen ist r.e:

- ☐ $\{n : (\exists p \in P) : n + p \in A\}$
- ☐ $\{n : (\forall p \in P) : n + p \in A\}$
- ☐ $\{n : (\exists p \in P) : \varphi_n(p) \in A\}$

Eine partielle Funktion $f : \mathbb{N} \to \mathbb{N}$ heißt injektiv, wenn $f(x) \neq f(y)$ für alle $x \neq y \in \text{dom}(f)$. Für so ein f ist die Umkehrfunktion f^{-1} definiert (wieder eine partielle Funktion). Welche der folgenden Aussagen ist richtig:

- ☐ Wenn f p.r., dann f^-1 p.r.
- ☐ Wenn f t.r., dann f^-1 t.r.
- ☐ Wenn $f : \mathbb{N} \to \mathbb{N}$ p.r. und bijektiv, dann f^-1 t.r.