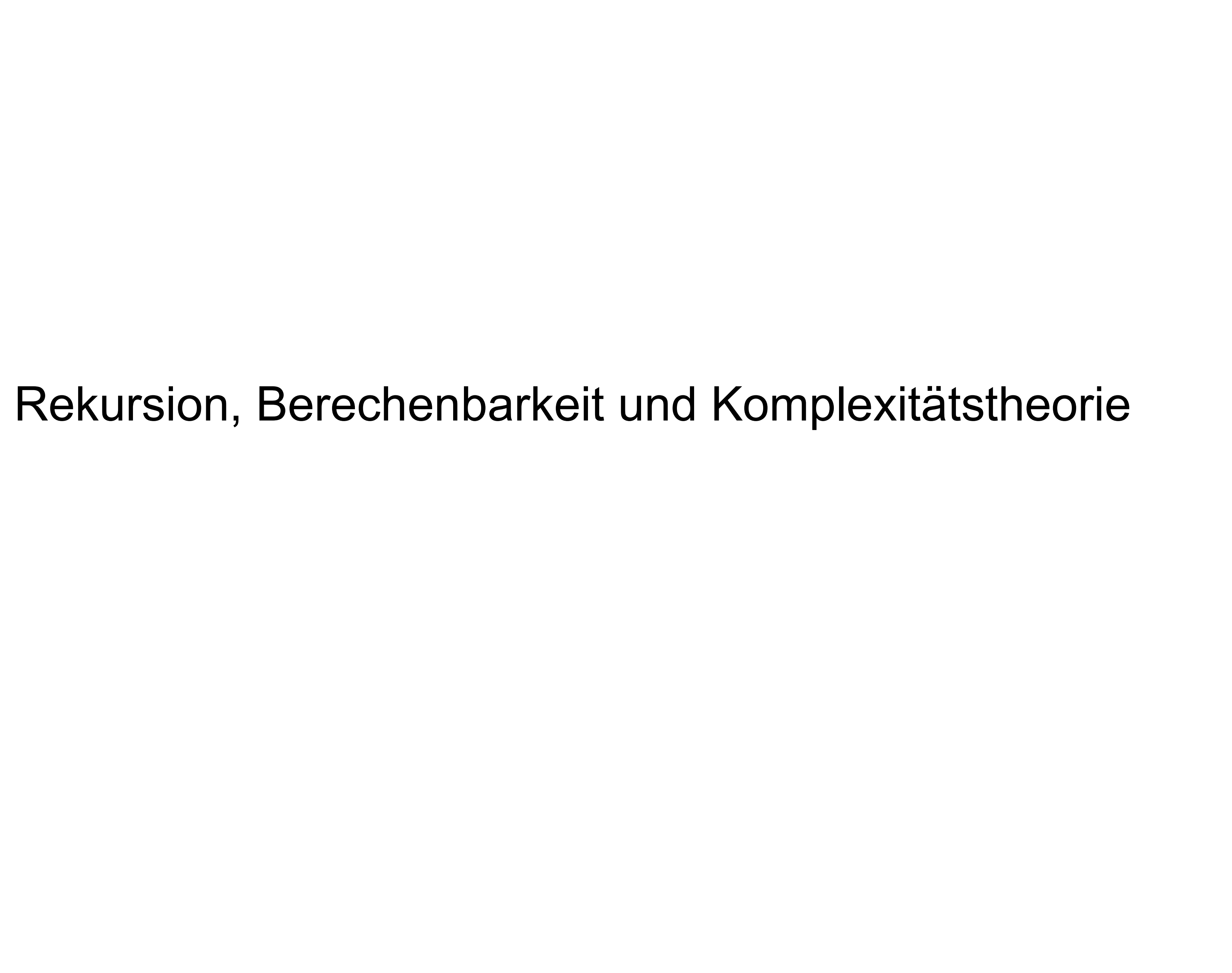

Rekursion, Berechenbarkeit und Komplexitätstheorie

Teil V: Rekursion, Komplexitätstheorie

1. Einführung
2. Beispiele
3. μ-Rekursion
4. Berechenbarkeit
5. Komplexitätstheorie

1. Einführung

- Einleitung – Rekursion

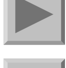
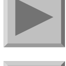
Einleitung – Rekursion

f: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$ $\qquad\qquad$ $\mathbb{N}_0 = \{ 0, 1, \ldots \}$

- f(n+1) berechnet sich aus einigen oder allen Werten f(n), f(n – 1), …,f(0) (der sog. Anweisungsteil der Funktion enthält mindestens einen Aufruf von sich selbst),
- Anfangswert f(0) und ggf. weitere Werte werden explizit vergeben,
- Rekursion ist die Übertragung des Rekursionsprinzips aus der Mathematik in die Informatik.

2. Beispiele

- Summe
- Produkt
- Potenz
- Fibonacci
- Natürliche Zahlen
- Schneeflocke
- Ulam Problem
- Türme von Hanoi
- Zusammenfassung

Summe

$$f(n) = \begin{cases} 0 & \text{, falls } n = 0 \quad \text{(Rekursionsanfang)} \\ f(n-1) + n & \text{, falls } n \geq 1 \quad \text{(Rekursionsschritt)} \end{cases}$$

$$\begin{aligned} f(0) &= 0 \\ f(1) &= f(0) + 1 = 0 + 1 = 1 \\ f(2) &= f(1) + 2 = 1 + 2 = 3 \\ f(3) &= f(2) + 3 = 3 + 3 = 6 \\ &\vdots \end{aligned}$$

geschlossene Form $f(n) = 0 + \ldots + n = \frac{n(n+1)}{2}$, d. h.

f gibt die Summe aller Zahlen von 0 bis n an.

Produkt

$$f(n) = \begin{cases} 1 & \text{, falls } n = 0 \\ f(n-1) \cdot n & \text{, falls } n \geq 1 \end{cases}$$

$$\begin{aligned}
f(0) &= 1 \\
f(1) &= f(0) \cdot 1 = 1 \cdot 1 = 1 \\
f(2) &= f(1) \cdot 2 = 1 \cdot 2 = 3 \\
f(3) &= f(2) \cdot 3 = 2 \cdot 3 = 6 \\
f(4) &= f(3) \cdot 4 = 6 \cdot 4 = 24 \\
&\vdots
\end{aligned}$$

„geschlossene“ Form $f(n) = 1 \cdot \ldots \cdot n = n!$, d. h.

f gibt das Produkt der Zahlen von 1 bis n an. (Fakultät)

Potenz

$$f(n) = \begin{cases} 1 & \text{, falls } n = 0 \\ 2 \cdot f(n-1) & \text{, falls } n \geq 1 \end{cases}$$

$$f(0) = 1$$

$$f(1) = 2 \cdot f(0) = 2 \cdot 1 = 2$$

$$f(2) = 2 \cdot f(1) = 2 \cdot 2 = 4$$

$$f(3) = 2 \cdot f(2) = 2 \cdot 4 = 8$$

$$\vdots$$

geschlossene Form $f(n) = 2^n$, d. h.

f gibt die Potenzen von 2 an.

Fibonacci

$$f(n) = \begin{cases} 0 & \text{, falls } n = 0 \\ 1 & \text{, falls } n = 1 \\ f(n-1) + f(n-2) & \text{, falls } n > 1 \end{cases}$$

$$f(2) = f(1) + f(0) = 1 + 0 = 1$$
$$f(3) = f(2) + f(1) = 1 + 1 = 2$$
$$f(4) = f(3) + f(2) = 2 + 1 = 3$$
$$f(5) = f(4) + f(3) = 3 + 2 = 5$$
$$f(6) = f(5) + f(4) = 5 + 3 = 8$$
$$\vdots$$

geschlossene Form $f(n) = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$, d. h.

f gibt die sog. Fibonacci Zahlen an.

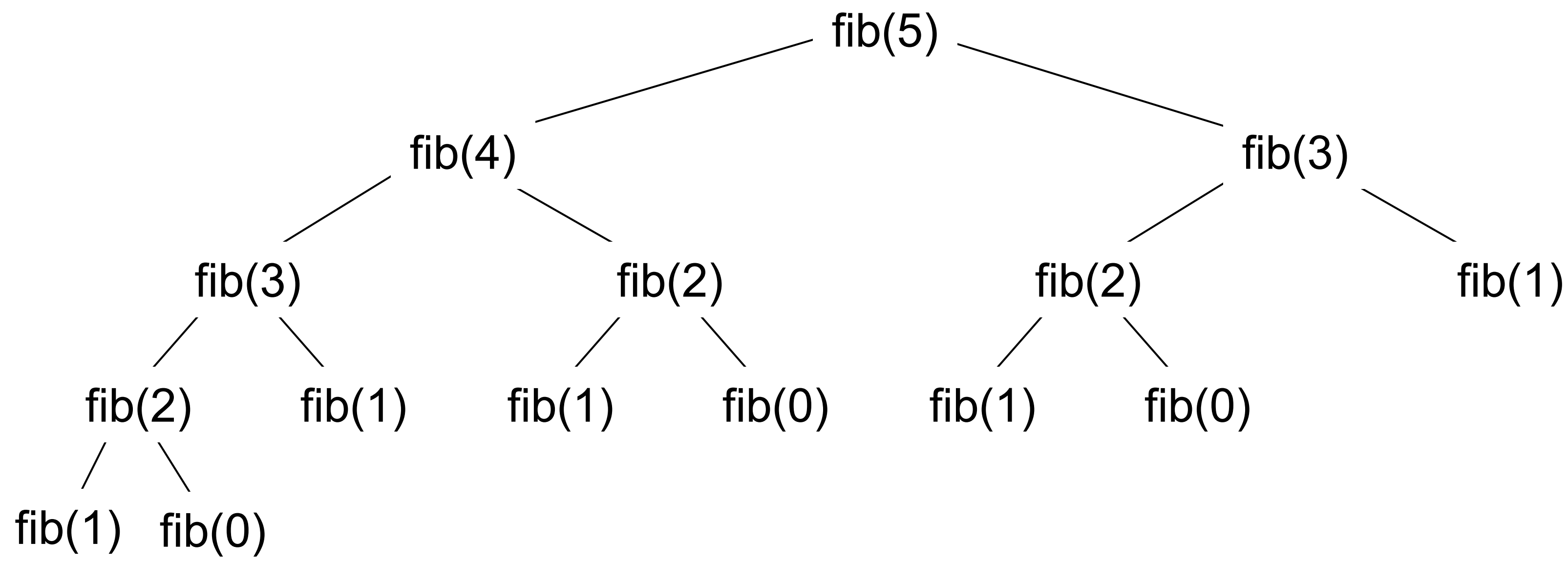
Fibonacci (1)

- Die Anzahl der Funktionsaufrufe zur rekursiven Berechnung der Fibonacci Zahlen wächst stärker als die Grösse der Fibonacci Zahlen.

fib(0)	=	0,	1	Aufruf	fib(15)	=	610,	1973	Aufrufe
fib(1)	=	1,	1	Aufruf	fib(16)	=	987,	3193	Aufrufe
fib(2)	=	1,	3	Aufrufe	fib(17)	=	1597,	5167	Aufrufe
fib(3)	=	2,	5	Aufrufe	fib(18)	=	2584,	8361	Aufrufe
fib(4)	=	3,	9	Aufrufe	fib(19)	=	4181,	13529	Aufrufe
fib(5)	=	5,	15	Aufrufe	fib(20)	=	6765,	21891	Aufrufe
fib(6)	=	8,	25	Aufrufe	fib(21)	=	10946,	35421	Aufrufe
fib(7)	=	13,	41	Aufrufe	fib(22)	=	17711,	57313	Aufrufe
fib(8)	=	21,	67	Aufrufe	fib(23)	=	28657,	92735	Aufrufe
fib(9)	=	34,	109	Aufrufe	fib(24)	=	46368,	150049	Aufrufe
fib(10)	=	55,	177	Aufrufe	fib(25)	=	75025,	242785	Aufrufe
fib(11)	=	89,	287	Aufrufe	fib(26)	=	121393,	392835	Aufrufe
fib(12)	=	144,	465	Aufrufe	fib(27)	=	196418,	635621	Aufrufe
fib(13)	=	233,	753	Aufrufe	fib(28)	=	317811,	1028457	Aufrufe
fib(14)	=	377,	1219	Aufrufe	fib(29)	=	514229,	1664079	Aufrufe

Fibonacci (2)

- Die Anzahl der Funktionsaufrufe zur rekursiven Berechnung der Fibonacci Zahlen wächst stärker als die Grösse der Fibonacci Zahlen.



Funktionsaufrufe zur Berechnung von fib(5)

Hinweis: Man braucht eine enorme Buchführung und Speicherverwaltung, um die Größe unter rekursiven Aufrufen zu ermitteln.

Natürliche Zahlen

- Die natürlichen Zahlen selber können mittels Rekursion definiert werden
  - $1 \in \mathbb{N}$ (bzw. $0 \in \mathbb{N}_0$)
  - mit $n \in \mathbb{N}$ ist auch $n + 1 \in \mathbb{N}$
    (bzw. mit $n \in \mathbb{N}_0$ ist auch $n + 1 \in \mathbb{N}_0$)

„Schneeflocke“ oder „Koch-Kurve“

ersetzen durch

Ersetzen mehrfach
(rekursiv) wiederholen

„Schneeflocke“ oder „Koch-Kurve“ (1)

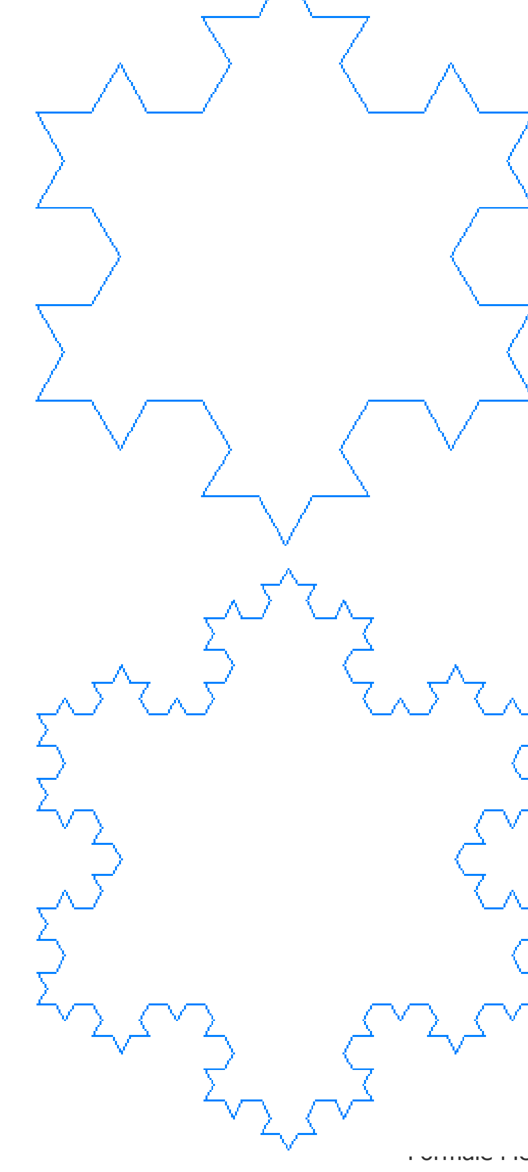
gleichseitiges Dreieck

1. Ersetzungsschritt

„Schneeflocke“ oder „Koch-Kurve“ (2)

2. Ersetzungsschritt

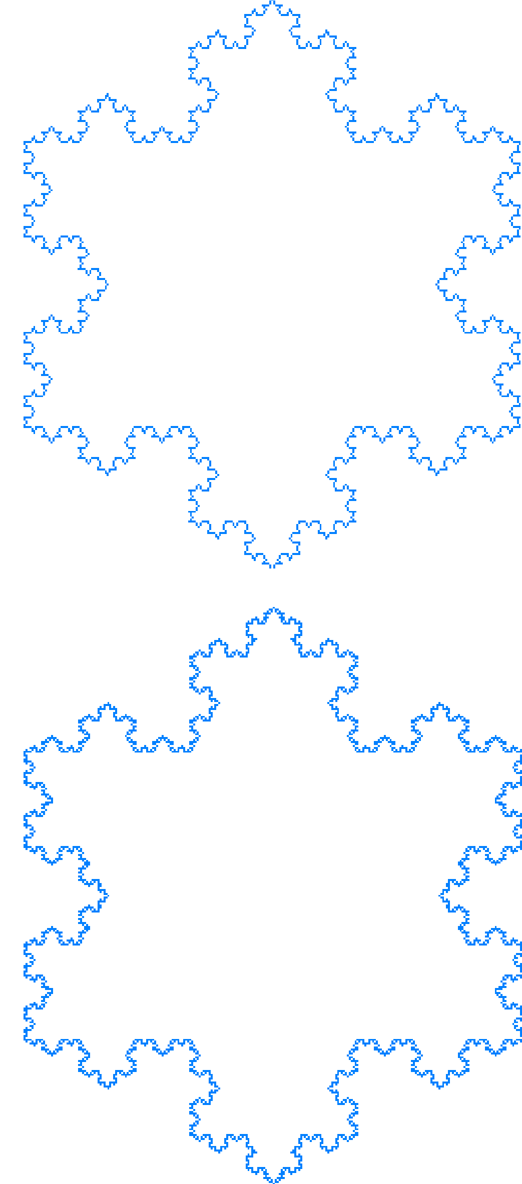
3. Ersetzungsschritt



„Schneeflocke“ oder „Koch-Kurve“ (3)

4. Ersetzungsschritt

5. Ersetzungsschritt



„Ulam Problem“

$$f(n+1) = \begin{cases} \tfrac{1}{2} \cdot f(n) & \text{, falls } f(n) \text{ gerade} \\ 3 \cdot f(n) + 1 & \text{, falls } f(n) \text{ ungerade} \end{cases} \qquad n = 1, 2, \ldots$$

f(0) beliebig, aber gegeben

es ist bis heute unklar, ob jeder Startwert f(0) auf den Zyklus 4-2-1-4 führt.

Beispiele:

$f(0) = 6 \Rightarrow$ f(1) = 3, f(2) = 10, f(3) = 5, f(4) = 16, f(5) = 8, f(6) = 4, f(7) = 2, f(8) = 1, f(9) = 4

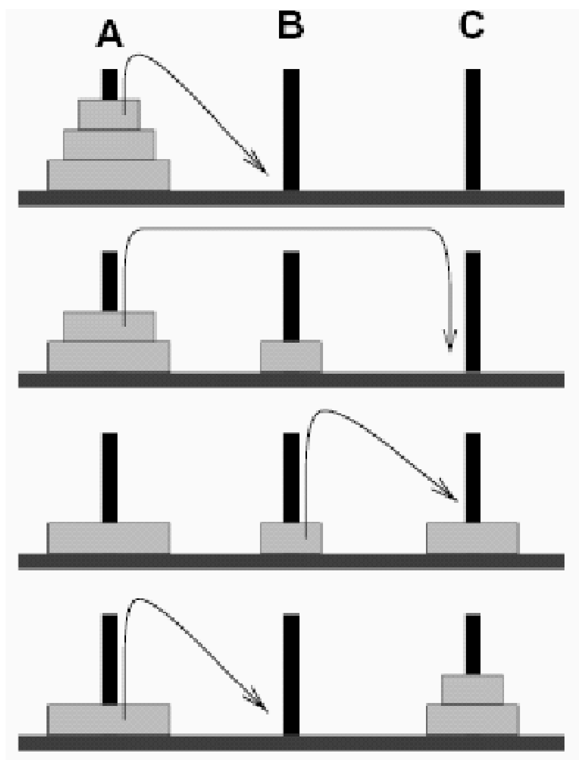
Zyklus

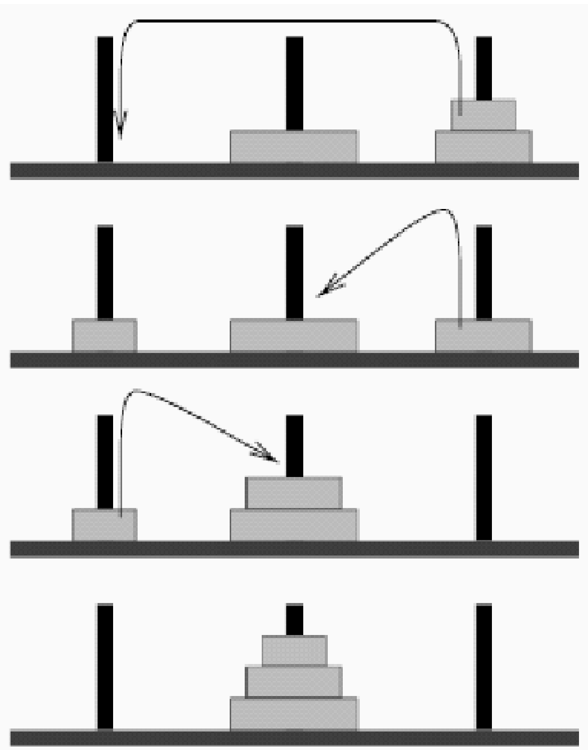
$f(0) = 7 \Rightarrow$ f(1) = 22, f(2) = 11, f(3) = 34, f(4) = 17, f(5) = 52, f(6) = 26, f(7) = 13, f(8) = 40, f(9) = 20, f(10) = 10, f(11) = 5, f(12) = 16, f(13) = 8, f(14) = 4, f(15) = 2, f(16) = 1, f(17) = 4

Zyklus

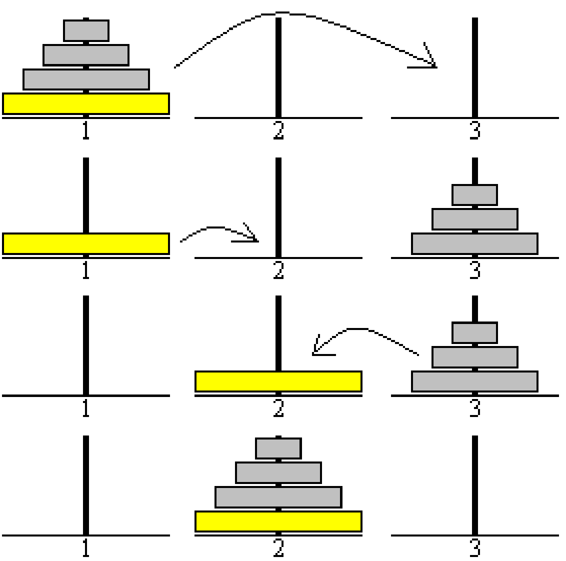
„Türme von Hanoi“

- n Scheiben mit unterschiedlichem Durchmesser sind auf einer Stange gestapelt. Die Scheiben sollen auf eine von zwei anderen Stangen umgestapelt werden, so dass stets kleine Scheiben oberhalb grösserer Scheiben liegen. Ablage ausserhalb der Stangen ist nicht erlaubt.





„Türme von Hanoi“ (1)



- Lösungsskizze:

  Wenn es eine Lösung für n – 1 Scheiben gibt, so auch eine für n Scheiben. Also werden die obersten n – 1 Scheiben in richtiger Schichtung in Parkposition gebracht, die unterste zur Zielstange bewegt, und die n – 1 Scheiben von der Park- in die Zielposition gebracht.

„Türme von Hanoi“ (2)

- Neben dem Lösungsansatz selber ist die Anzahl der Scheiben-bewegungen rekursiv.

$$f(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2 \cdot f(n-1) + 1 & \text{, falls } n > 1 \end{cases}$$

$$\begin{aligned}
f(2) &= 2 \cdot f(1) + 1 = 2 \cdot 1 + 1 = 3 \\
f(3) &= 2 \cdot f(2) + 1 = 2 \cdot 3 + 1 = 7 \\
f(4) &= 2 \cdot f(3) + 1 = 2 \cdot 7 + 1 = 15 \\
f(5) &= 2 \cdot f(4) + 1 = 2 \cdot 15 + 1 = 31 \\
&\vdots
\end{aligned}$$

geschlossene Form $f(n) = 2^n - 1$

Einfache Funktionen

- Rekursion erlaubt auch die Formulierung sehr einfacher Funktionen

$f(n) = f(n-1) \quad n \geq 1.$

$f(0)$ beliebig, aber gegeben.

$f(n) = f(n-1) = f(n-2) = \ldots = f(0),$

d. h. f ist Konstant (**konstante Funktion**).

$f(n) = f(n-1) + 1$

$f(0) = 7$

$f(1) = f(0) + 1 = 7 + 1 = 8$

$f(2) = f(1) + 1 = 8 + 1 = 9$

$\vdots$

geschlossene Form $f(n) = n + 7$, d. h. f ist **Addition mit einer Konstante**.

Einfache Funktionen (1)

- $f(n) = f(n-1) + 8$
  $f(0) = 0$

  $f(1) = f(0) + 8 = 0 + 8 = 8$
  $f(2) = f(1) + 8 = 8 + 8 = 16$
  $f(3) = f(2) + 8 = 16 + 8 = 24$
  ⋮

  geschlossene Form $f(n) = 8 \cdot n$, d. h. f ist die **Multiplikation mit einer Konstante**.

Zusammenfassung

- Die Werte der bisher angegebenen rekursiven Funktionen – mit Ausnahme der Fakultät – wachsen nicht schneller als exponentiell

- Beispiel:

  Fibonaccizahlen $f(n) < 2^n$

  $$\begin{aligned} f(0) &= 0 < 1 = 2^0 \\ f(1) &= 1 < 2 = 2^1 \\ f(n) &= f(n-1) + f(n-2) < 2^{n-1} + 2^{n-2} \\ &= 2^{n-2} \cdot (2 + 1) \\ &< 2^{n-2} \cdot 4 \\ &= 2^n \end{aligned}$$

  Fakultät $n! \approx 2^{n \cdot \log n}$

- Es gibt aber noch wesentlich stärkeres Wachstum $\rightarrow$ Ackermannfunktion

3. μ-Rekursion

- Ackermannfunktion
- Primitive Rekursion von Funktionen
- μ-Rekursion von Funktionen
- Berechenbarkeit

Ackermannfunktion

- Rekursion ist in bestimmten Situationen naheliegend.
- Mit Rekursion lassen sich Berechnungen formulieren, die ohne Rekursion kaum formulierbar sind und deren Wachstum kaum vorstellbar ist.
  $\rightarrow$ Ackermannfunktion

Ackermannfunktion (bzw. „verständliche“ Modifikation) (1)

$$a(0, m) = m + 1$$
$$a(n + 1, 0) = a(n, 1)$$
$$a(n + 1, m + 1) = a(n, a(n + 1, m))$$

verschachtelte Rekursion

Ackermannfunktion (bzw. „verständliche“ Modifikation) (2)

Einige Werte:

$$a(1, 0) = a(0, 1)$$

n = 0

$$= 1 + 1 = 2$$

$$a(1, 1) = a(0, a(1, 0))$$

n = m = 0

$$= a(1, 0) + 1$$

$$= 2 + 1 = 3$$

$$a(1, m) = m + 2$$

Ackermannfunktion (bzw. „verständliche“ Modifikation) (3)

n \ m	0	1	2	3	4	m
0	1	2	3	4	5	m + 1
1	2	3	4	5	6	m + 2
2	3	5	7	9	11	$2 \cdot m + 3$
3	5	13	29	61	125	$8 \cdot 2^m - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$	$a(3, 2^{65536} - 3)$	a(3, a(4, 3))	$\underbrace{2^{2^{\cdots^{2}}}}$ – 3 (m + 3 Terme)
5	65533	a(4, 65533)	a(4, a(5, 1))	a(4, a(5, 2))	a(4, a(5, 3))	
6	a(5, 1)	a(5, a(5, 1))	a(5, a(6, 1))	a(5, a(6, 2))	a(5, a(6, 3))	

- Lesart der Tabelle: $a(3, m) = 8 \cdot 2^m - 3$ etc.
  Der Wert a(4, 2) hat 19729 Dezimalstellen!

Primitive Rekursion von Funktionen

- Ausgangsfunktionen
  - Alle Konstanten (konstante Funktionen)
  - Nachfolgerfunktion f(x) = x + 1
  - … (mehrstellige Varianten)
- Erweiterungsprinzipien
  - Zu f(x) und g(x) wird die zusammengesetzte Funktion f(g(x)) betrachtet (Einsetzungsprinzip).
  - Zu f(x) wird die Funktion $g(n, x) = \underbrace{f(f(\ldots}_{n \text{ mal}}(x)\ldots)$ betrachtet (Rekursionsprinzip)
  - … (mehrstellige Varianten)
- Eine Funktion ist **primitiv rekursiv**, wenn sie eine der Ausgangsfunktionen ist oder aus den Ausgangsfunktionen durch endlich viele Anwendungen der Erweiterungsprinzipien gebildet werden kann.

μ-Rekursion von Funktionen

- Ausgangsfunktionen
  - Alle Konstanten (konstante Funktionen)
  - Nachfolgerfunktion $f(x) = x + 1$
  - … (mehrstellige Varianten)
- Erweiterungsprinzipien
  - Zu $f(x)$ und $g(x)$ wird die zusammengesetzte Funktion $f(g(x))$ betrachtet (Einsetzungsprinzip).
  - Zu $f(x)$ wird die Funktion $g(n, x) = \underbrace{f(f(\ldots(}_{n \text{ mal}}x)\ldots)$ betrachtet (Rekursionsprinzip)
  - Zu $f(n, x)$ wird $g(x) = \min \{ n \mid f(n, x) = 0 \}$ betrachtet (Minimumoperator → μ-Operator)
  - … (mehrstellige Varianten)
- Eine Funktion ist **μ-rekursiv**, wenn sie eine der Ausgangsfunktionen ist oder aus den Ausgangsfunktionen durch endlich viele Anwendungen der Erweiterungsoperationen gebildet werden kann.

μ-Rekursion von Funktionen (1)

- Die Gesamtheit aller μ-rekursiven Funktionen ist größer als die Gesamtheit aller primitiv rekursiven Funktionen
- Die Ackermannfunktion ist μ-rekursiv aber nicht primitiv rekursiv!
  (Beweis **sehr** schwierig)

Berechenbarkeit

- Berechenbarkeit im Sinne primitiver Rekursion und μ-Rekursion kann von der intuitiven Vorstellung des „Ausrechnens“ abweichen. (Pathologische Fälle).
  Z. B. ist die Funktion

$$f(x) = \begin{cases} 1, \text{ falls Goldbachvermutung richtig} \\ 0, \text{ falls Goldbachvermutung falsch} \end{cases}$$

  berechenbar.

- Die bisher ungeklärte Goldbachvermutung besagt, dass sich jede gerade Zahl als Summe von genau zwei Primzahlen schreiben lässt, z. B.

$$20 = 17 + 3$$

Programmtechnischer Unterschied

- Primitive Rekursion

```
INPUT n, x
y = x
FOR i = 1 TO n
    y = f(y)
OUTPUT y
```

- μ-Rekursion

```
INPUT x
n = 0
WHILE f(n, x) ≠ 0
    n = n + 1
OUTPUT n
```

- Berechnung
  Rekursionsprinzip

  $g(n, x) = f(f(\ldots(x)\ldots))$

- Berechnung
  μ-Operator

  $g(x) = \min \{ n \mid f(n, x) = 0 \}$

- Zahl der Durchläufe durch eine Eingabegröße beschränkt.

- Zahl der Durchläufe **nicht** angebbar beschränkt

WHILE-Programme sind mächtiger als FOR-Programme!

Beweisbarkeit

Gödel:

- Die Arithmetik ist unvollständig, d. h. es gibt über den natürlichen Zahlen und deren Operatoren + und · wahre Aussagen, die nicht beweisbar sind.

Grund:

- Beweis heißt endlich viele Schritte (vgl. 4-Farben Theorem). Die Goldbachvermutung kann falsch oder richtig sein. Ist sie richtig, mag das an unendlich vielen Effekten liegen, die nicht auf endlich viele Fälle reduziert werden können.

4. Berechenbarkeit

- Berechenbarkeit
- Turingmaschine
- Turingmaschinen und Funktionen
- Konsequenzen bisheriger Fakten zur Berechenbarkeit
- Church‘sche These
- Post‘sche Korrespondenzproblem
- Game of Life

Berechenbarkeit

- „Axiomatischer“ Ansatz
  (Axiom = Grundannahme, die ohne Begründung akzeptiert wird)

  Man erklärt gewisse Funktionen für berechenbar und man erklärt gewisse Operationen, die die Berechenbarkeit erhalten
  (primitive Rekursion, μ-Rekusion)

- Maschinen Ansatz

  Man erklärt einen Typ von Rechenmaschinen und man erklärt als berechenbar das, was diese Maschinen „berechnen“ können.

  (**Turingmaschine**, unbegrenzte Registermaschine, Kellerautomat, …)

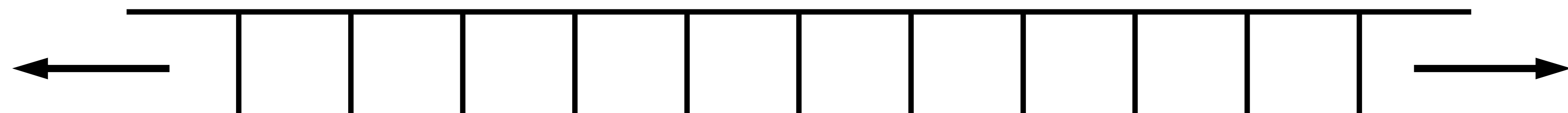
Turingmaschine

- Maschinen**modell** zur Untersuchung von Berechenbarkeit; das Modell hat keine besondere Eignung für praktische Anwendungen.

- Bemerkenswert:

  Modell entstand ca. 1936, also (knapp) vor den ersten „programmierbaren" Rechenmaschinen.

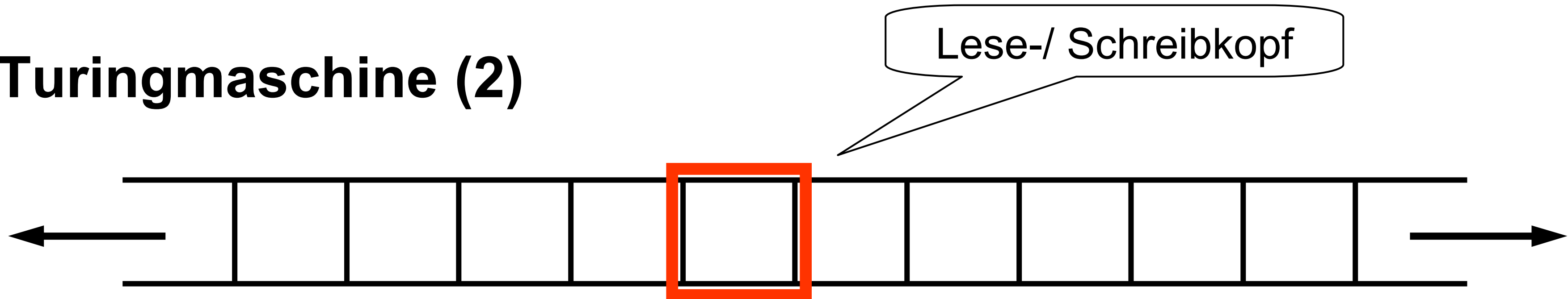
Turingmaschine (1)



Band von gleichartigen Zellen, beidseitig unendlich.

- Für das Band gibt es ein Bandalphabet, d.h. einen Vorrat von endlich vielen Zeichen, z. B. B = { a, b, …, z, 0, …, 9, ␣ }. Leerzeichen ␣ (oder □).
- Jede Zelle enthält genau ein Zeichen aus B.

Turingmaschine (2)



- Turingmaschine hat Lese- und Schreibkopf, der immer genau eine Bandzelle erkennt bzw. bearbeitet und (danach) sich oder das Band ggf. um eine Zelle nach rechts oder links bewegt.
  Der Lese-/Schreibkopf verfügt über eine endliche Zustandsmenge $Q = \{ q_0, q_1, \ldots, q_n, H \}$

  q_0 : Anfangszustand

  H : Haltezustand

Turingmaschine (3)

➢ Übergangsfunktionen:

in Abhängigkeit von

- aktuell gelesenen Zeichen und
- aktuellem Zustand

wird

- ggf. in die Zelle ein neues Zeichen geschrieben (und dabei ein evtl. vorhandenes gelöscht) und/oder
- das Band nicht bewegt, oder um 1 Zelle nach rechts oder links bewegt und/oder
- ein neuer Zustand angenommen

formal: $(b, q) \xrightarrow{\text{Ü}} \{ b', \{ N, R, L, H \}, q' \}$

Übergänge erfolgen so lange, bis der Haltezustand H erreicht ist.

Turingmaschine (4)

- Beispiel:

  B = { a, b, □ }   Q = { q_0, q_1 }

Eingabe	a	b	□
q_0	□Rq_1	□Rq_1	□Hq_1
q_1	□Rq_1	□Rq_1	□Hq_1

- Angesetzt auf nicht-leeres Zeichen löscht die Turingmaschine dies und geht nach rechts etc. bis sie auf erstes leeres Feld stösst. Dann halt.
- Nicht besonders sinnvoll, aber schon einfache „vernünftige“ Probleme erforden komplizierte Übergangsfunktionen (maschinennah statt problemnah).

Verdopplungsmaschine

- Am einfachsten lassen sich Turingmaschinen konstruieren, die Aufgaben über Zeichenfolgen ausführen

  z. B. „Verdoppele die Anzahl der 0 zwischen zwei begrenzenden 1‘en“

Eingabe	Ausgabe
1 0 1	1 0 0 1
1 0 0 1	1 0 0 0 0 1
1 0 0 0 1	1 0 0 0 0 0 0 1
⋮	⋮

„Verdoppelungsmaschine“

Arbeitsweise der Verdoppelungsmaschine

- Der Lese-/Schreibkopf wird auf einer Zelle mit nicht-leerem Symbol angesetzt. Von dort bewegt er sich zur linken 1 und streicht diese.

- Der Lese-/Schreibkopf geht 1 Zelle nach rechts .

Findet er dort eine 1, so gehe bis zur ersten leeren Zelle nach rechts,
setzt dort 1, stop

Findet er dort eine 0, so streicht er diese,
geht bis zur ersten leeren Zelle nach rechts,
setzt dort 0
geht um 1 Zelle nach rechts
setzt dort 0
geht bis zur ersten nicht leeren Zelle nach links

(nicht) terminieren oder „falsches“ Ergebnis

- Es kann passieren, dass eine Turingmaschine bei einer „falschen“ Eingabe

    1. niemals stehen bleibt (Turingmaschine hält nicht)

        z. B. „Gehe nach rechts bis zum ersten nicht-leeren Zeichen“, aber dort stehen nur leere Zeichen

oder

    2. mit falschem oder gar keinem Ergebnis stehen bleibt

        z. B. „Verdoppelungsmaschine“ angesetzt auf 011.
        Dies Verhalten ist auch von Funktionen her bekannt:

        z. B. $\sqrt{-2}$ = ?

Turingmaschinen und Funktionen

- Eine Turingmaschine **berechnet** eine Funktion im folgenden Sinn:

  Wird ein zulässiges Argument als Eingabe auf das Band der Turingmaschine geschrieben (mit dem Alphabet, das die Maschine versteht) und wird die Maschine auf die Eingabe angesetzt und in q_0 gestartet, so

  1) bleibt sie nach endlich vielen Schritten stehen und es
  2) steht auf dem Band der Funktionswert.

Turingmaschinen und Funktionen (1)

- Funktionen können binär sein, d. h. nur die Werte 0 und 1 annehmen.
  Dabei kann sein

$$f(x) = \begin{cases} 1 & \text{, falls x bestimmte Eigenschaft E besitzt} \\ 0 & \text{, sonst} \end{cases}$$

x: Eingabe der Turingmaschine (Zeichenkette)

- Beispiel:

  Eigenschaft E der Zeichenkette „ist Palindrom“, d. h. Zeichenkette, die vorwärts wie rückwärts gelesen werden kann

a b 1 2 3 2 1 b a a b c c b a	Palindrome
a b a b	kein Palindrom

Turingmaschinen und Funktionen (2)

- Es gibt Turingmaschinen, die die Funktion

$$f(x) = \begin{cases} 1 & \text{, x ist Palindrom} \\ 0 & \text{, x ist kein Palindrom} \end{cases}$$

  berechnen können.

  D. h. Turingmaschine kann Test- oder Entscheidungsproblem lösen.

Turingmaschinen und Funktionen (3)

- Bei komplizierteren Eigenschaften als „Palindrom“ kann es sein, dass eine Turingmaschine die Eigenschaft bestätigt,

  wenn sie tatsächlich bei einer Eingabe vorliegt,
  („zertifiziert“)

  aber zu keinem Ergebnis führt,
  wenn die Eigenschaft bei einer anderen Eingabe nicht vorliegt.

Turingmaschinen und Funktionen (4)

- Turingmaschinen sind so mächtig wie μ-Rekursion, d. h. zu jeder μ-rekursiven Funktion gibt es eine Turingmaschine, die diese berechnet.

  UND

  Jede Turingmaschine kann als μ-rekursive Funktion aufgefasst werden.

  (Nachweis SEHR schwierig)

Konsequenzen bisheriger Fakten zur Berechenbarkeit

- Turingmaschinen:
  endliche Beschreibungen, die nach endlich vielen Berechnungsschritten (angesetzt auf vernünftige Eingaben) zum Ergebnis führen.

- μ-rekursive Funktionen:
  endliche Beschreibungen von Funktionen und Operationen darauf, die nach endlich vielen Funktionsanwendungen zum Wert führen.

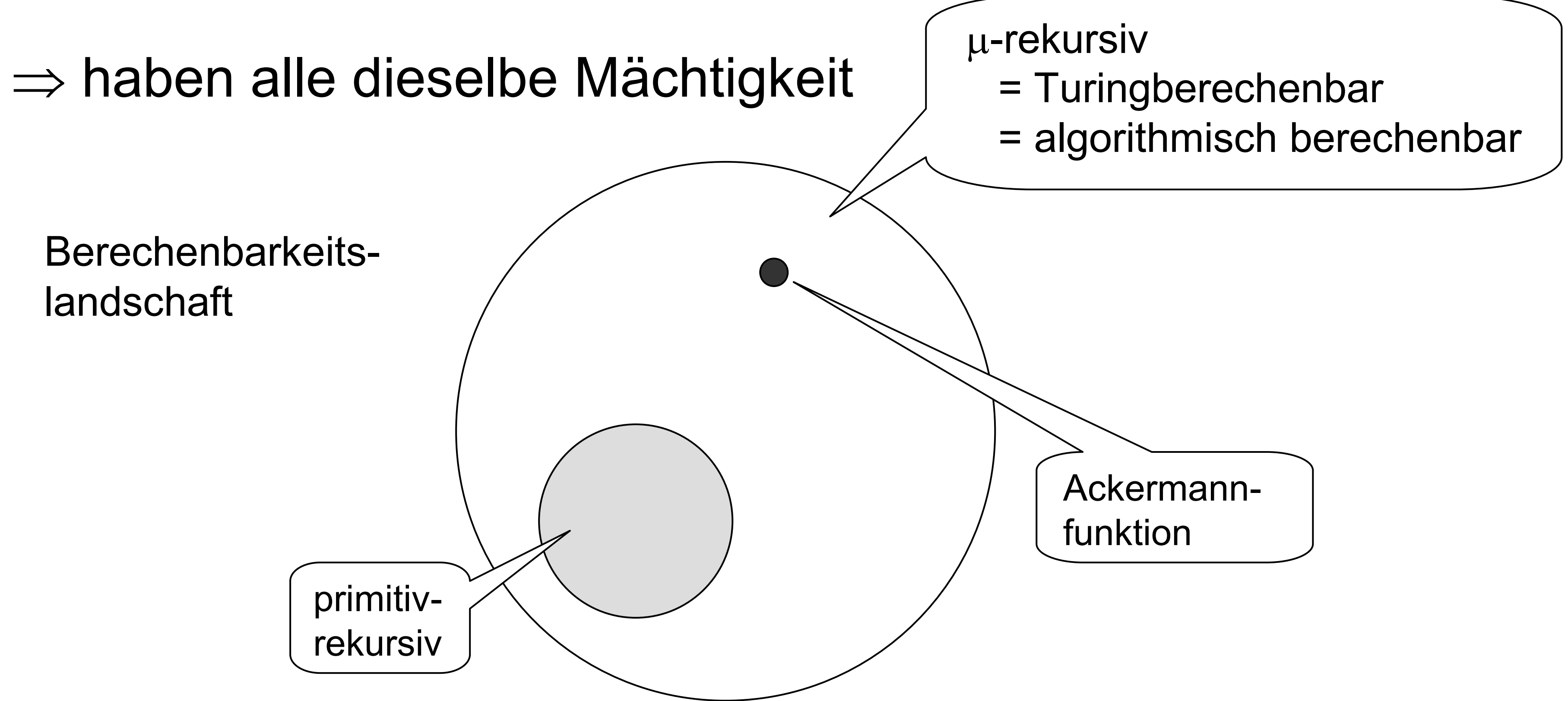
$\Rightarrow$ Definition Algorithmus:
Endliche Beschreibung von Berechnungen, die nach endlich vielen Schritten zum Ergebnis führen.

(Die Beschreibung von Algorithmen erfolgt dabei typischerweise in sog. formalen Sprachen, die von Turingmaschinen und rekursiven Funktionen abweichen.)

Konsequenzen bisheriger Fakten zur Berechenbarkeit (1)

- Turingmaschinen
- μ-Rekursion
- Algorithmen (die WHILE Bedingungen zulassen)

⇒ haben alle dieselbe Mächtigkeit



Church'sche These

- Jeder sinnvolle Berechenbarkeitsbegriff („effective" computability) ist äquivalent zur Turingberechenbarkeit bzw. μ–Rekursionen bzw. zu Algorithmen (mit WHILE Bedingung).

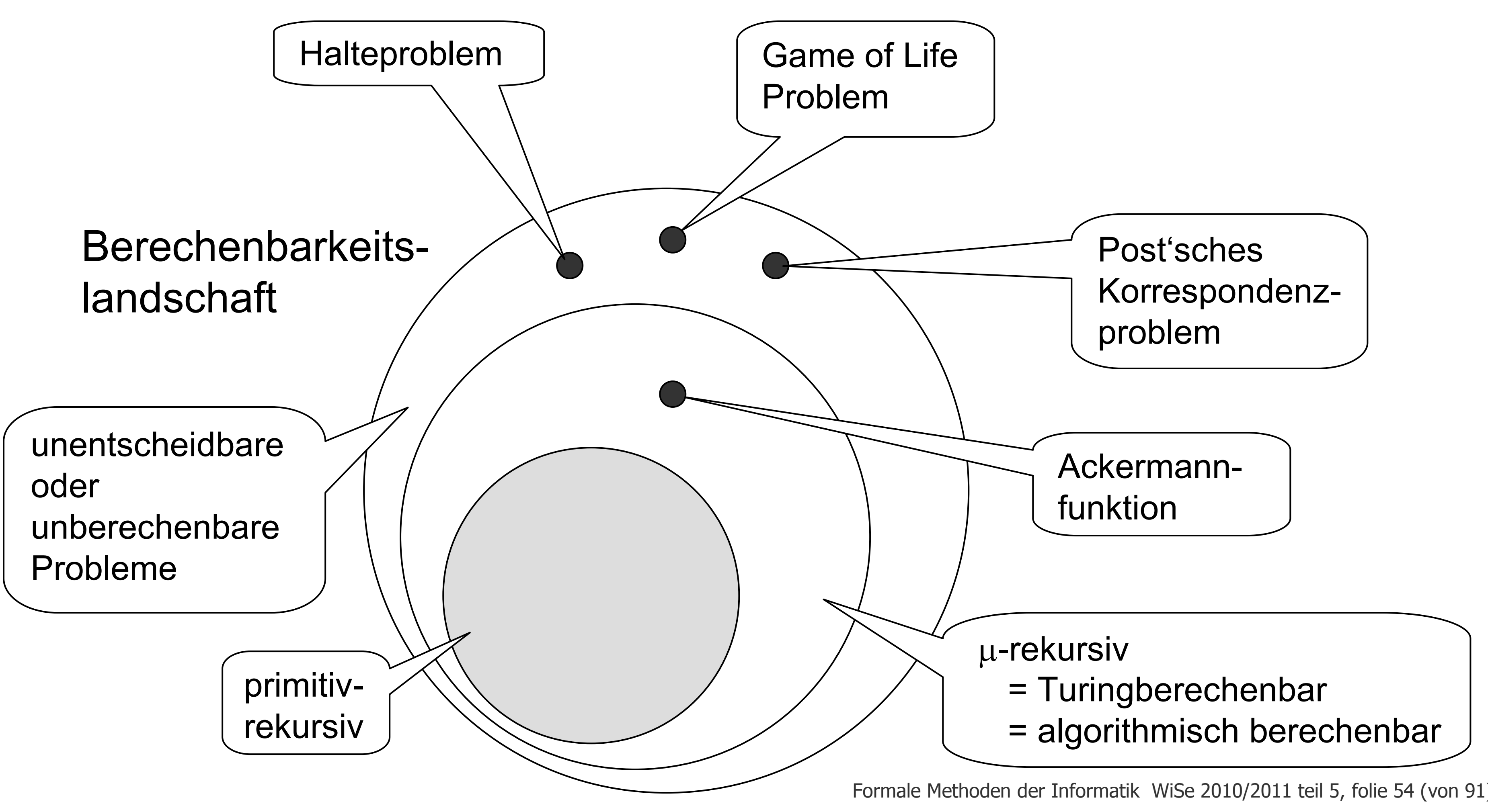
- Die Church'sche These bezieht sich u. a. auf Turingmaschinen selber.
  Egal ob mehrere Bänder und entsprechend viele Lese- / Schreibköpfe aber gemeinsame Übergangsfunktion oder Band einseitig beschränkt:

  Bereich der Turingberechenbarkeit bleibt erhalten.

Gibt es nicht-berechenbare Probleme?

Nicht-Berechenbarkeit

- Es gibt also noch „schwierigere“ Probleme als die Berechnung der Ackermannfunktion.



Situation der Nicht-Berechenbarkeit und Geometrie

- In Berechenbarkeit und Geometrie gibt es Probleme, die einfach zu beschreiben aber unlösbar sind.

- In Geometrie

  Zweiteilung eines Winkels mit Zirkel und Lineal möglich
  Dreiteilung eines Winkels mit Zirkel und Lineal unmöglich!

  (Nachweis SEHR schwierig)

Halteproblem für Turingmaschinen (bzw. Algorithmen)

- **Eingabe:**

  Beliebige Turingmaschine (bzw. Algorithmus)
  Eingabe für Turingmaschine

- **Frage:**

  Hält die Turingmaschine (bzw. Algorithmus) irgendwann?

Halteproblem für Turingmaschinen (bzw. Algorithmen) (1)

- Beispiel:

```
INPUT x            (natürliche Zahl)
WHILE x ≠ 1
  x = x – 2
OUTPUT x
```

- Offensichtlich hält diese „Turingmaschine“ (bzw. Algorithmus) für $x \in \{ 1, 3, 5, \dots \}$.
  Aber kein Anhalten bei $x \in \{ 2, 4, 6, \dots \}$.

- (Praktisch wird angehalten, wenn der Bereich der darstellbaren Zahlen nach unten verlassen wird. Dies spielt bei Maschinen**modellen** aber keine Rolle.)

Pragmatisches Vorgehen zum Halteproblem

➢ Lasse Maschine (bzw. Algorithmus) laufen und gib Zeitobergrenze vor.

Falls Maschine (bzw. Algorithmus) anhält ✓

Falls Maschine (bzw. Algorithmus) nicht anhält, so könnte sein

- nie anhalten
- anhalten nach höherer Zeitobergrenze

$\Rightarrow$ Halteproblem ist unentscheidbar, d. h. es gibt keinen Algorithmus, der für **alle** Algorithmen und **alle** Eingaben das Problem klärt.

Post'sche Korrespondenzproblem

- Paare zu Zeichenfolgen

$$(x_1,y_1), \ldots, (x_n,y_n)$$

- Kann man durch Hintereinanderscheiben von Zeichenfolgen x_i (Wiederholungen erlaubt!) eine Zeichenfolge erzeugen, die gleich ist der Zeichenfolge, der „Korrespondenten" y_i ?

Post‘sche Korrespondenzproblem (1)

- Beispiel:

	1	2	3	4	5
x_i :	abb	a	bab	baba	aba
y_i :	bbab	aa	ab	aa	a

2, 1, 1, 4, 1, 5 ist eine Korrespondenz, d. h.

$$x_2\, x_1\, x_1\, x_4\, x_1\, x_5 = y_2\, y_1\, y_1\, y_4\, y_1\, y_5$$

$$\underbrace{a}_{x_2}\,\underbrace{abb}_{x_1}\,\underbrace{abb}_{x_1}\,\underbrace{baba}_{x_4}\,\underbrace{abb}_{x_1}\,\underbrace{aba}_{x_5} = \underbrace{a\,a}_{y_2}\underbrace{bb\,ab}_{y_1}\underbrace{b\,bab}_{y_1}\underbrace{a\,a}_{y_4}\underbrace{bb\,ab}_{y_1}\underbrace{a}_{y_5}$$

Post‘sche Korrespondenzproblem (2)

- Beispiel:
  unlösbarer Fall des Post‘schen Korrespondenzproblems

	1	2	3	4	5
x_i :	bb	a	bab	baba	aba
y_i :	bab	aa	ab	aa	a

- Unlösbarkeit durch spezielle Einzelüberlegung für diese „Eingabe“ ermittelt.

Post‘sche Korrespondenzproblem (3)

- Es gibt keinen Algorithmus, der für beliebige Eingaben zum Post‘schen Korrespondenzproblem
  - eine Korrespondenz angibt, falls es eine gibt und
  - angibt, dass es keine Korrespondenz gibt, falls es tatsächlich keine gibt.

Game of Life

- „Game of Life“ ist unentscheidbar

  4 x 4 Spielfeld und grössere Spielfelder mit quadratischen, gleichartigen Zellen (ähnlich Band einer Turingmaschine)

- Zelle ist mit Zeichen „lebendig“ oder „tot“ belegt. Zellenbelegung ändern sich zyklisch („von Generation zu Generation“)

- Jede Zelle hat bis zu acht Nachbarn

Game of Life (1)

- Regeln:

  1) Lebende Zellen mit nur einem oder keinem lebenden Nachbarn stirbt in nächster Generation („Vereinsamung“)

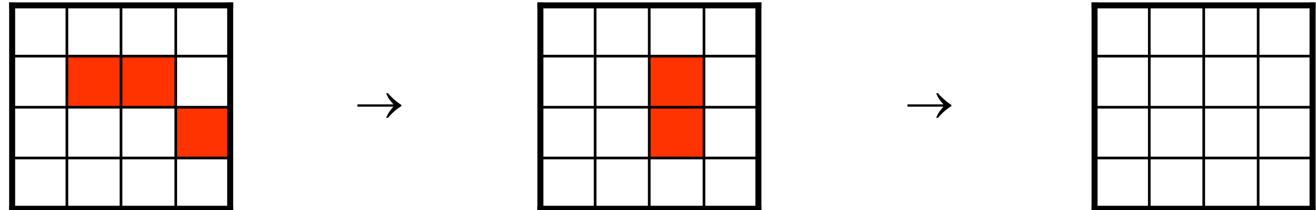
  2) Lebende Zelle mit 2 oder 3 lebenden Nachbarn bleibt lebend

  3) Lebende Zelle mit 4 oder mehr lebenden Nachbarn stirbt aus („Überbevölkerung“)

  4) Tote Zelle mit 3 lebenden Nachbarn wird neu geboren

Game of Life (2)

- Beispiel für Spielablauf:



- Berechungsproblem:

  Können zwei beliebig gegebene Konfigurationen ineinander überführt werden gemäss der Spielregeln?

  Einzelfallentscheidungen möglich, aber es gibt keinen Algorithmus, der dies für beliebige Konfigurationspaare feststellen kann. („Game of Life" ist unentscheidbar).

- (siehe auch: http://www.bitstorm.org/gameoflife)

5. Komplexitätstheorie

- Einführung
- Entscheidungs- vs. Suchprobleme
- Nichtdeterminismus
- Wie wird Rechenzeit gemessen?
- Komplexitätsklasse P
- Polynomielle Rechenzeit
- Vergleich von Algorithmen
- Das Erfüllbarkeitsproblem/satisfiability problem (SAT)
- NP-vollständige Probleme

Was ist Komplexitätstheorie?

Ziel der Komplexitätstheorie

Bestimmung des Ressourcenbedarfs für die algorithmische Lösung von Problemen.

Welche Ressourcen?

- Rechenzeit T
- Speicherplatz S
- Produkt TS
- Parallele Rechenzeit (Parallelität hat keinen Einfluss auf Berechenbarkeit an sich, verändert aber den Zeitbedarf)
- etc.

Was ist ein Problem?

Ein Problem wird beschrieben durch

- das Eingabeformat
- Angabe der zulässigen Ausgaben.

Üblich

- Eingabe: endliche Folge von Zeichen aus einem endlichen Alphabet (also z.B. keine irrationalen Zahlen)
- Es genügt, *eine* korrekte Ausgabe zu berechnen, egal wie viele es gibt.

Entscheidungs- vs. Suchprobleme

Partition – Entscheidungsvariante

Eingabe: Natürliche Zahlen $a_1,\ldots,a_n$.

Frage: Gibt es $I \subseteq \{1,\ldots,n\}$ mit $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

Partition – Suchproblem

Eingabe: Natürliche Zahlen $a_1,\ldots,a_n$.

Aufgabe: Finde eine Menge $I \subseteq \{1,\ldots,n\}$ mit

$$\sum_{i \in I} a_i = \sum_{i \notin I} a_i$$

Entscheidungs- vs. Suchprobleme (1)

Kürzestes Wegeproblem – Entscheidungsvariante

Eingabe: Gerichteter Graph mit positiver Kantenbewertung, Startknoten, Zielknoten, Testgrösse M.

Frage: Gibt es einen (kürzesten) Weg vom Start- zum Zielknoten mit Länge ≤ M?

Kürzestes Wegeproblem – Suchvariante

Eingabe: Gerichteter Graph mit positiver Kantenbewertung, Startknoten, Zielknoten.

Frage: Was ist ein (kürzester) Weg vom Start- zum Zielknoten mit Länge ≤ M?

(Oft kann man aus einer Lösung des Entscheidungsproblems eine Lösung des Suchproblems ableiten).

Entscheidungs- vs. Suchprobleme (2)

Hinweis:

Hat man einen polynomiellen bzw. schnellen Algorithmus (s. u.) um zu entscheiden, ob es einen Weg vom Start- zum Zielknoten mit Länge $\leq M$ gibt

dann folgt:

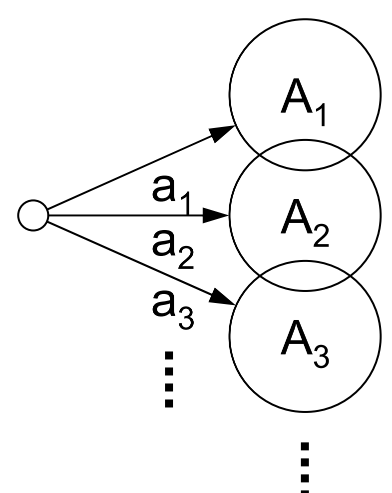
Es gibt auch einen polynomiellen bzw. schnellen Algorithmus, um festzustellen, ob es einen Kürzesten derartigen Weg gibt.
Man kann z. B. einen Besuch (0, $\sum$ aller Kantenlängen) für jede denkbare Länge M' schnell (mit Hilfe von Intervallteilung) entscheiden, ob es einen Weg vom Start zum Zielknoten mit Länge $\leq M'$ gibt oder nicht.

Entscheidungs- vs. Suchprobleme (3)

Hinweis:

Hat man einen polynomiellen bzw. schnellen Algorithmus um festzustellen, ob es einen kürzesten Weg vom Start- zum Zielknoten mit Länge ≤ M gibt, dann kann man einen solchen Weg auch in polynomieller Zeit bzw. schnell finden.

Vorgehen:



Für jede Startkante $a_i \in \{ a_1, \ldots, a_l \}$, prüft man, ob es in dem zugehörigen A_i einen kürzesten Weg mit Länge ≤ M – Länge(a_i) gibt.

Falls ja, kann man aus a_i und (induktiv) aus einem geeigneten Weg in A_i den gesuchten Weg zusammensetzen.

Offenbar gibt es mindestens ein a_j, für das die gefundene Bedingung erfüllt ist.

Entscheidungs- vs. Suchprobleme (4)

Im Weiteren werden fast immer Entscheidungsprobleme betrachtet

- Einfacher zu verstehen.
- Später: Ressourcenbedarf für beide Varianten meist gleich.
- Entscheidungsprobleme können auch als Menge der Eingaben beschrieben werden, für die die Antwort „Ja“ lautet, z. B.

$$\text{Partition} = \{ (a_1,\ldots,a_n) \mid \exists\, I: \sum_{i \in I} a_i = \sum_{i \notin I} a_i \}$$

Nichtdeterminismus

Nichtdeterministischer Algorithmus:

Algorithmus, der in jedem Schritt zwischen 2 Aktionen wählen darf.

Nichtdeterministische Turingmaschine (NTM):

Übergangsfunktion:

$$Ü: B \times Q \rightarrow (B, \{ N, R, L, H \}, Q)^2$$

Bei einer gewöhnlichen Turingmaschi-ne steht hier 1

Wichtig

Es gibt **keine Vorschrift**, welche Aktion gewählt wird.

Nichtdeterminismus (1) (gewöhnungsbedürftig)

Das Nichtdeterministische einer Turingmaschine kann man sich vereinfacht so vorstellen:

- Die Turingmaschine rät eine Lösung (Orakel!) und kann dann als gewöhnliche Turingmaschine überprüfen, ob die geratene Lösung die gewünschte Eigenschaft hat.

- Falls es eine Lösung mit gewünschter Eigenschaft gibt, so rät das Orakel eine, d. h. das Orakel rät richtig, wenn es geht.

Wie wird Rechenzeit gemessen?

Komplexitätsmaße sollten von der verwendeten Technologie unabhängig sein.

- Statt Rechenzeit wird immer die Anzahl der **Rechenschritte** gezählt.
- Wir verwenden ein einfaches Rechnermodell mit wenigen Befehlen (Turingmaschine).

  Rechenzeiten werden so **grob** gemessen, dass die Art der Operation keine Rolle spielt (Addition dauert genauso lange wie Division oder Potenzierung aufgrund dieser Annahme).

Komplexitätsklasse P

Definition:

Ein Problem L gehört zur Komplexitätsklasse P, wenn es einen deterministischen Algorithmus mit polynomieller Rechenzeit für L gibt.

Manchmal wird mit P auch nur die Menge der Entscheidungsprobleme mit Polynomialzeitalgorithmus bezeichnet.

Polynomielle Rechenzeit

Erweiterte Churchsche These:

Höchstens polynomieller Unterschied in Rechenzeiten verschiedener Rechnermodelle.

Beobachtung:

Einsetzen eines Polynoms in ein Polynom ist wieder Polynom.

$\rightarrow$ Polynomielle Rechenzeit ist unabhängig vom Rechnermodell.

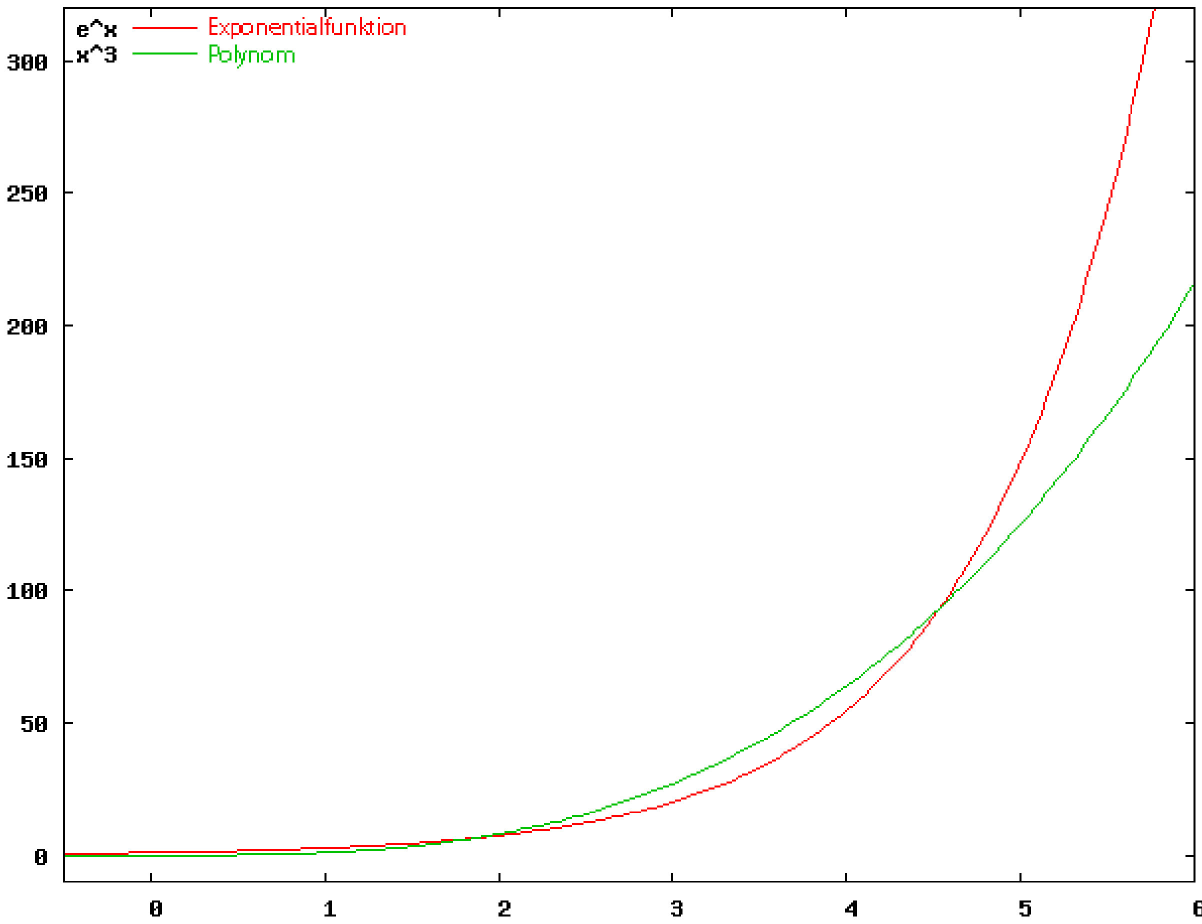
Sonderrolle polynomieller Rechenzeit

Gängige Interpretation in der theoretischen Informatik:

- Polynomielle Rechenzeit – **effizient**
  (z. B. n^2, n^{100})
- Superpolynomielle Rechenzeit – **nicht effizient**
  (z. B. $n^{\log n}$, $2^{0.01 \cdot n}$, 2^{n^2})

Ist eine starke Vereinfachung der Realität

Sonderrolle polynomieller Rechenzeit (1)



➢ Verläufe von Polynomen und Exponentialfunktionen sind generell unvergleichbar

Sonderrolle polynomieller Rechenzeit (2)

Warum ist es trotzdem sinnvoll,
polynomiale Rechenzeiten zu betrachten?

Überlege: Wie verändert sich die maximale Eingabelänge, wenn die erlaubte Anzahl von Rechenschritten verdoppelt wird?

Rechenzeit des Algorithmus sei genau n^d:

$$n_{neu}^d \leq 2 \cdot n_{alt}^d \quad \Rightarrow \quad \frac{n_{neu}}{n_{alt}} \leq 2^{1/d}$$

d. h.: Eingabelänge vergrößert sich um Faktor >1, falls d konstant ist, also Rechenzeit polynomiell ist; $n_{neu} = 2^{1/d} \cdot n_{alt}$ zulässig.

Vergleich von Algorithmen

Nur für große Einheiten n:

Algorithmus A ist asymptotisch mindestens so schnell wie Algorithmus B, wenn es eine von A und B möglicherweise abhängige Konstante m gibt, so dass

$t_A(n) \leq m \cdot t_B(n)$ Schreibweise: $t_A(n) \in O(t_B(n))$

Beachte:

- Die Algorithmen A und B können auch unvergleichbar sein.
- Es gibt nicht unbedingt einen optimalen Algorithmus

Vergleich von Algorithmen (1)

Hier: worst-case Sichtweise

Wir betrachten die schlechteste Rechenzeit für jede Eingabelänge.

Alternative:

Durchschnittliche Rechenzeit

Problem:

Unklar, für welche Wahrscheinlichkeitsverteilung man den Durchschnitt bilden soll.

Vergleich von Algorithmen (2)

P ist die Klasse aller Probleme, für die es einen Algorithmus A gibt mit

$$t_A(n) \in O(p(n))$$

wobei p(n) ein Polynom ist, das vom Algorithmus A abhängen kann und A auf einer Turingmaschine läuft.

P steht für „polynomial“

Vergleich von Algorithmen (3)

NP ist die Klasse aller Probleme, für die es einen Algorithmus A gibt mit

$$t_A(n) \in O(p(n))$$

wobei p(n) ein Polynom ist, das vom Algorithmus A abhängen kann und A auf einer nichtdeterministischen Turingmaschine läuft.

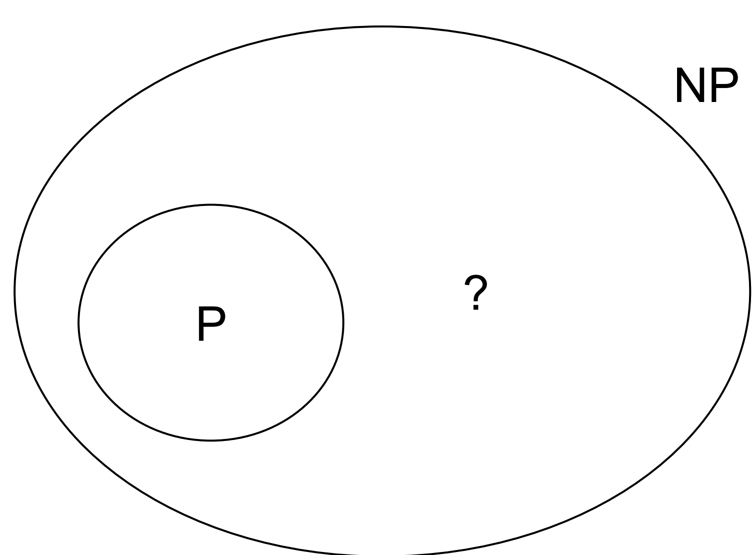
NP steht für „nichtdeterministisch" polynomial.

Vergleich von Algorithmen (4)

Gibt es ein Problem, das in NP liegt, aber nicht in P?
D. h. gewinnt man durch Nicht-Determinismus?

Antwort bis heute offen

Komplexitätslandschaft



Vergleich von Algorithmen (5)

Ausweg: NP-Vollständigkeit

Idee: Finde eine Klasse von „gleich schweren“ Problemen (die NP-vollständigen Probleme), so dass entweder

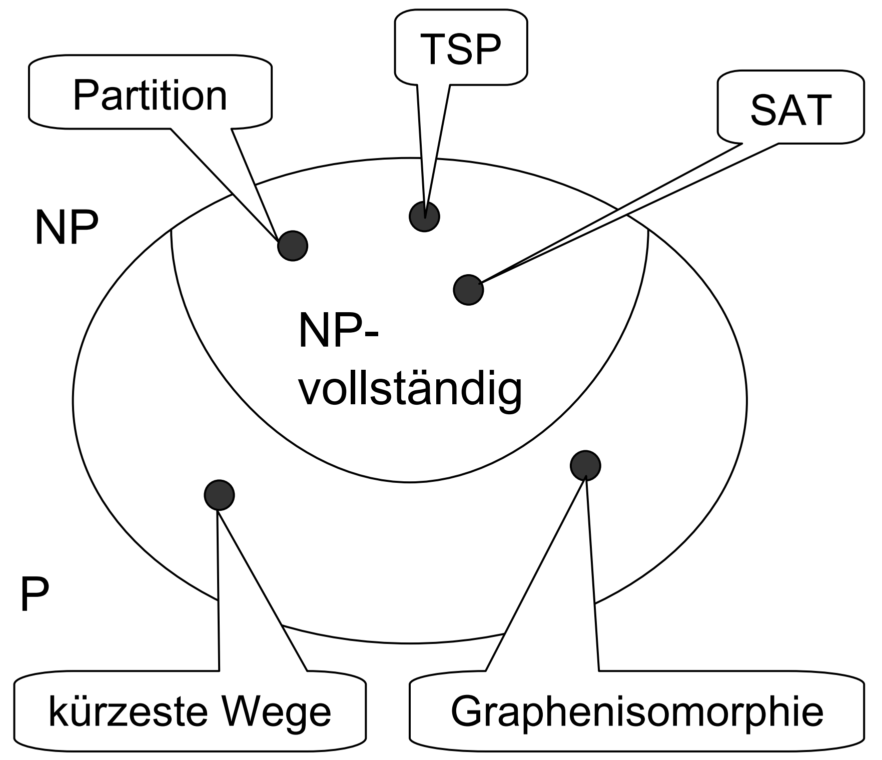
- P = NP: alle diese Probleme haben polynomielle Algorithmen;

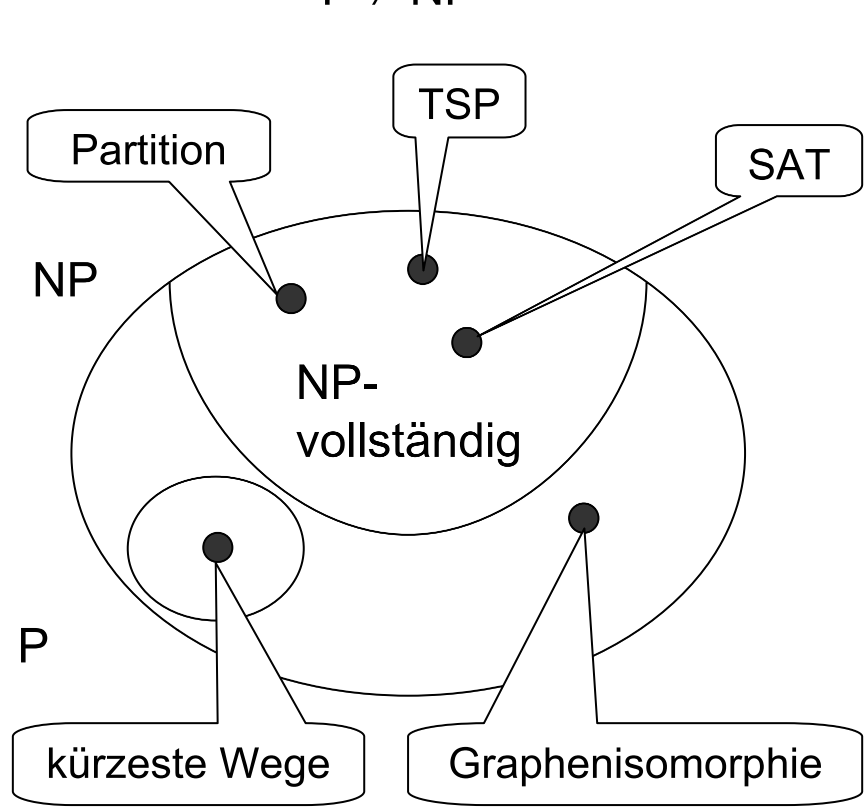
  oder

- P $\neq$ NP: alle diese Probleme haben keine polynomielle Algorithmen

Letzteres ist die Vermutung der „Experten“, da man für keines der mehreren Tausend bisher bekannte NP-vollständigen Probleme einen polynomiellen Algorithmus gefunden hat.

Komplexitätslandschaft





man kennt mehrere tausend (!) Probleme die NP-vollständig sind.

Das Erfüllbarkeitsproblem / satisfiability problem (SAT)

(Urproblem der NP-Vollständigkeit)

Gibt es eine Belegung logischer Variablen (binärer Variablen) x, y, z, w, … die Ausdrücke wie

$$(x \text{ oder } y \text{ oder } \neg z) \text{ und } (\neg x \text{ oder } w \text{ oder } z) \text{ und } \neg y$$

erfüllt, d. h. wahr (= 1) werden lässt?

hier:

x = 1		x = 1
y = 0	oder	y = 0
z = 0		z = 1
w = 0		w = 0

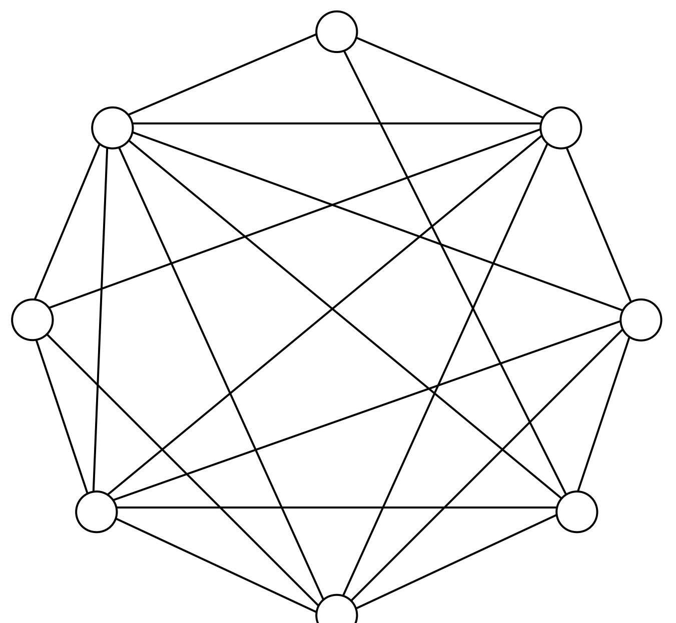
SAT wurde Anfang der 1970-er Jahre von Cook anhand der Definition als NP-vollständig bewiesen.

NP-vollständige Probleme

Eingabe: Ungerichteter Graph G = (V,E)

Aufgabe: Berechne eine Clique mit maximaler Knotenanzahl, also eine Menge V‘ $\subseteq$ V, so dass alle Knoten aus V‘ paarweise verbunden sind.

Beispiel:

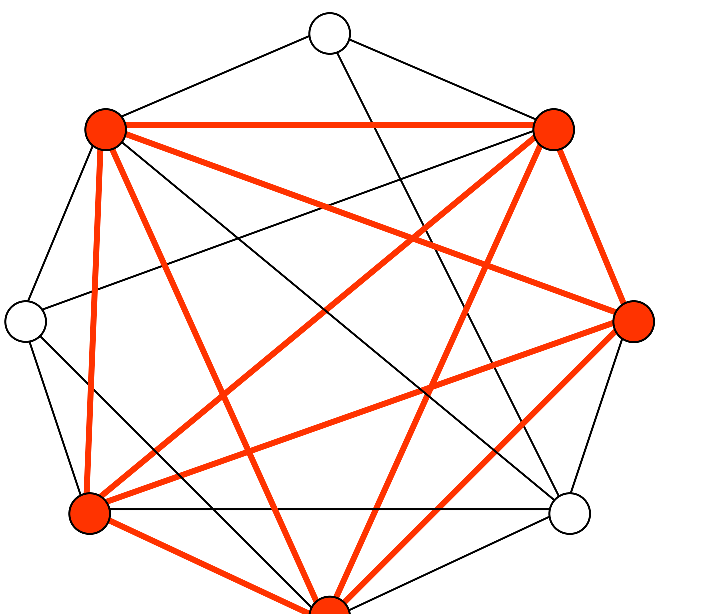


NP-vollständige Probleme (1)

Eingabe: Ungerichteter Graph G = (V,E)

Aufgabe: Berechne eine Clique mit maximaler Knotenanzahl, also eine Menge V‘ $\subseteq$ V, so dass alle Knoten aus V‘ paarweise verbunden sind.

Beispiel:



Diese 5er Clique ist maximal (eindeutig)