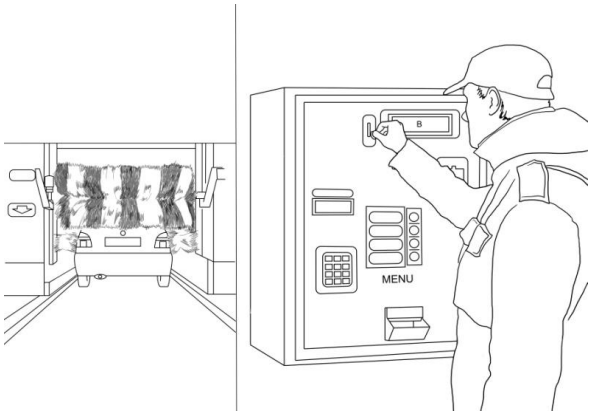
Vortrag am STIU14, Aarau, 8. Januar 2014

Armin P. Barth (KS Baden und MINT-Lernzentrum der ETH Zürich)

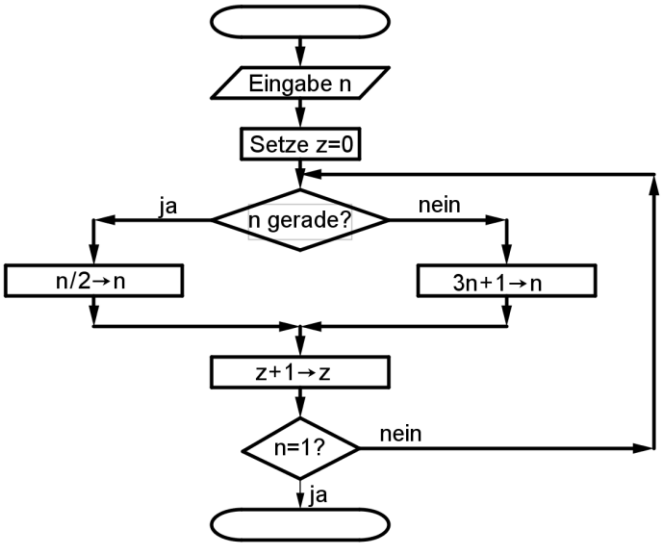
Armin.barth@ifv.gess.ethz.ch
www.armin-p-barth.ch

1	Algorithmen soweit das Auge reicht: • Spracherkennung (z.B. Geo Fluent, eine IBM-Tochter, hat eine Simultanübersetzungssoftware entwickelt, die geschriebene Sprache live übersetzen kann.) • Hochkomplexe Geschäftsverträge werden zunehmend von einer Software auf formale Fehler überprüft. • Unbemannte Drohnen • Robotergesteuerte Autos von Toyota und Audi • Watson, der Nachfolger von Deep Blue, holt bei „Jeopardy!“ dreimal mehr Punkte als die besten menschlichen Herausforderer • In den USA ist der grösste Teil der Produktionssteigerung in der Industrie auf Roboter zurückzuführen. • Trading-Algorithmen tätigen automatisch Kaufentscheidungen (Börsen-Flash-Crash am 6. Mai 2010)
2	Big Data **„Algorithmen wissen mehr über dich als Freunde und Familie.“** (Dirk Helbling, Physiker und Soziologe) • Statistiken des Bundes • Geodaten auf Smartphones • Aufzeichnungen von Kameras und Sensoren im öffentlichen Raum • Tweets, Blogs, Facebookeinträge,… • Links, die wir im Internet anklicken • Bei Amazon ermittelt ein Algorithmus, wofür ich mich interessiere • Usw.
3	Was ist an Schulen zu tun? (I) „Die Qualität der verwendeten Algorithmen und die Schlussfolgerungen, die daraus gezogen werden, sollten stetig hinterfragt werden.“ „Das Grundproblem ist, dass ein normaler Mensch zumindest eine Sprache beherrscht. Aber nur die wenigsten wissen, wie ein Algorithmus funktioniert. Und deshalb verstehen auch die wenigsten den Prozess, wie Algorithmen die Welt formen, in der wir uns bewegen.“ (Dirk Helbling, Physiker und Soziologe)
4	Was ist an Schulen zu tun? (II)

	„Im Zentrum sollten Algorithmen stehen, also exakte Beschreibungen eines Rechenablaufs, und Programme, Information und Daten. Die Schülerinnen und Schüler sollten kleine Programme schreiben, zum Beispiel, um eine Liste zu sortieren. Sie sollten für einfache Probleme selbständig Algorithmen entwickeln , entscheiden, welche Daten nötig sind, dann die Algorithmen in eine Programmiersprache übersetzen und auf einem Computer zum Laufen bringen." J. Kohlas, J. Schmid, C.A. Zehnder, „informatik@gymnasium", Verlag Neue Zürcher Zeitung (2013)
5	Was ist an Schulen zu tun? (III) „Computational thinking – problem analysis, algorithmic thinking, algorithmic expression, abstraction, modelling, stepwise fault isolation – in central to an increasingly broad array of fields. (…) Those who can practice computational thinking, and who can wield the power of computer science effectively, will be in the position to make greater contributions in all fields than those who can't. Indeed, the 2013 Nobel Prize in Chemistry was for computer models, and the official press release said: *Today the computer is just as important a tool for chemists as the test tube.*" David Patterson, professor of computer science at Berkeley
6	Was ist an Schulen zu tun? (IV) „Jedes Kind sollte programmieren können, denn beim Programmieren werden Fähigkeiten geschult, die in keinem anderen Fach in diesem Mass erlernt werden können. (…) Die Kinder bekommen eine Aufgabe, sie müssen diese verstehen, selbständig einen Lösungsweg entdecken und ihn dann in einer eindeutigen Sprache so formulieren, dass auch der Computer ihn versteht und ausführen kann. (…) Die Schule muss die Kinder befähigen, selber einem Problem auf den Grund zu kommen, Erfahrungen zu sammeln, die zu Entdeckungen führen, und dazu zu verstehen, wie man die Technik steuert." „Vielleicht werden die Laptops bald so billig, dass alle einen mitbringen, aber für den Informatikunterricht genügt ein mobiles Arbeitszimmer pro Schule. Informatik kann man sogar ganz ohne Computer vermitteln." Prof. Juraj Hromkovic, ETH Zürich
7	Was ist an Schulen zu tun? (V) Aus J. Hromkovic, „Homo informaticus" Schweizer Monat 1010, Vertiefen/Dossier, Oktober 2013 „…"
8	Auszüge aus meinem Buch „Algorithmik für Einsteiger", Springer-Verlag, Wiesbaden (2013): **Algorithmus *Waschstraße*** 1. Wähle Menü aus. 2. Warte, bis der zu bezahlende Betrag *B* angezeigt wird. 3. Falls Abbruch erwünscht, drücke die Rücksetz-Taste und beginne wieder bei 1.

	4. Wiederhole nun Einwurf einer Münze bis für die Summe S der Münzwerte gilt: $S \geq B$. 5. Drücke die Start-Taste. 6. Falls $S > B$ ist, entnehme das Wechselgeld. **End** 
9	Heron für Sqrt(2): $x = \frac{2}{x}$ Algorithmisch: **Algorithmus *Heron*** (a, x_0) Wiederhole Berechne $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ für $n = 0,1,2,...$ bis $\left\lvert a - x_{n+1}^2 \right\rvert < 0.000000001$ Schreibe: „Die Wurzel von“, a, „ist “, x_{n+1} **end** Dialektisch: mit um 2 nach unten verschobener Normalparabel
10	Algorithmen, soweit das Auge reicht… Neuerdings findet man Algorithmen in Bereichen, in denen man sie nicht erwarten würde. Vier Studenten der Northwestern University in Illinois haben einen Algorithmus namens Stats Monkey entwickelt, der automatisch Sportnachrichten erzeugen kann, indem er online verfügbare Informationen über Spielort, Spieler und Spielverlauf sowie vorgefertigte Textbausteine zu einem publizierbaren Artikel zusammenschustert. Das liest sich dann etwa so: „BOSTON – Things looked bleak for the Angels when they trailed by two runs in the ninth inning, but Los Angeles recovered thanks to a key single from Vladimir Guerrero to pull out a 7-6 victory over the Boston Red Sox at Fenway Park on Sunday.“ In der Musik gibt es Algorithmen, die den Komponisten unterstützen, indem sie automatisch Basslinien einfügen oder Stücke gleich selber komponieren. Der Algorithmus AARON nützt künstliche Intelligenz, um ständig neue Gemälde als Bildschirmschoner auf den Computerbildschirm zu zaubern. Und der Algorithmus *Cybernetic Poet* von Raymond Kurzweil verfasst selbständig Gedichte wie zum Beispiel dieses:

	You broke my soul the juice of eternity, the spirit of my lips. Gute Programmiererinnen und Programmierer sind heute sehr gefragt. Im Jahr 2004 ließ die Firma Google in Cambridge, Massachusetts, und im Silicon Valley riesige Plakate aufhängen, die nur den folgenden Text enthielten: $\left\{ \begin{matrix} \text{first 10-digit prime found} \\ \text{in consecutive digits of e} \end{matrix} \right\}.\text{com}$
11	„Definitionen“: “Bei der Lösung des Problems [mit einem Algorithmus] wird eine Regel angewendet, deren Vorschrift so klar ist, dass jedermann zu jeder Zeit dieses Problem auf die gleiche Art und Weise löst.” “Informally speaking, an algorithm is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called *procedures* or *recipes*.” “Algorithms are general step-by-step procedures for solving problems.” Endlichkeit, Determiniertheit, Sequenzanweisungen, Schleifen, Verzweigungen
12	Zum Nachdenken! Wie könnte wohl ein Algorithmus aussehen, der eine beliebige natürliche Zahl als Input akzeptiert und dann automatisch entscheidet, ob diese Zahl prim ist oder nicht? Welche Schritte müssen dazu ausgeführt werden? Und sind diese Schritte wirklich alle streng determiniert? Testen Sie Ihren Algorithmus mit den Zahlen 1, 2, 3, 10, 31, 2809, 57330. Wie reagiert er jeweils? Bemühen Sie sich dabei, nichts hinzuzudenken, was nicht im Algorithmus steht. Wie viele und welche Schritte müssen mindestens ausgeführt werden, damit der Entscheid des Algorithmus bei jeder beliebigen Zahl ganz sicher korrekt ist? Sie könnten diese Schrittzahl zum Beispiel abhängig von der Größe der Inputzahl ausdrücken. Wenn der Computer, auf dem der Algorithmus läuft, pro Sekunde eine Milliarde Operationen ausführen kann, wie lange dauert dann der Primtest bei einer 4-stelligen Zahl? Und bei einer 50-stelligen? Sie haben vor sich einen Bogen Papier, auf dem die ersten 1000 natürlichen Zahlen der Reihe nach notiert sind. Sie beginnen bei 2 und streichen alle Vielfachen von 2 (außer 2 selbst) weg. Dann gehen Sie zu 3 und streichen alle Vielfachen von 3 (außer 3 selbst) weg. Dann gehen Sie immer zur nächsten noch nicht gestrichenen Zahl und streichen alle Vielfachen der Zahl (außer die Zahl selbst) weg. Wann genau wird dieser Algorithmus terminieren? Und was wird er bis dahin geleistet haben?
13	Der aus der Bagdader Schule stammende Abu Abdallah Muhammed ibn Musa al-Chwarizmi al-Magusi, künftig kurz Al-Chwarizmi genannt, verfasste mehrere Mathematikbücher über Arithmetik und Algebra, deren große Bedeutung für die Nachwelt vor allem darin liegt, dass in ihnen

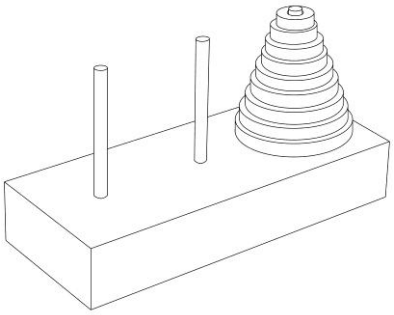
	das aus Indien stammende dezimale Stellenwertsystem und die darauf beruhenden Rechenoperationen detailliert beschrieben werden. Originale von al-Chwarizmi sind uns keine erhalten geblieben, doch liegt in der Bibliothek von Cambridge eine Handschrift einer lateinischen Übersetzung eines seiner Werke. Sie beginnt mit „Dixit Algoritmi: laudes deo rectori nostro atque defensori dicamus dignas…“ Eine andere Handschrift beginnt mit „Incipit liber Algorithmi…“ Und von dem um 1240 in Paris unterrichtenden Mönch Alexander de Villa Dei stammen die Hexameter „Hinc incipit algorismus. Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0 9 8 7 6 5 4 3 2 1“
14	Collatz-Problem: Flussdiagramm und Pseudocode aufstellen! 
15	Ideen für „Algorithmen auf dem Laufsteg“: Monte-Carlo-Pi Zum Nachdenken! Welchen Flächeninhalt hat das Quadrat? Welchen Flächeninhalt hat der Viertelkreis? Und durch welchen Term lässt sich folglich das Verhältnis dieser beiden Flächeninhalte ausdrücken? Wenn Sie 1000 Dartpfeile zufällig (also ohne Bemühen, eine bestimmte Stelle zu treffen) auf die quadratische „Scheibe“ schießen und davon 761 innerhalb des Viertelkreises landen und folglich 239 im Rest des Quadrates, wie groß ist dann das obige Flächenverhältnis ungefähr?

	Weshalb kann aber auf diesem Weg niemals der exakte Wert dieses Verhältnisses gefunden werden? Wie lässt sich aus dem bisher Gesagten ein Algorithmus ableiten, der die Zahl Pi näherungsweise bestimmt? Und welche Rolle spielt der Zufall in diesem Algorithmus? Wie könnte ein Flussdiagramm dieses Algorithmus aussehen? Und wie ein Pseudocode-Programm?

16	Ideen für „Algorithmen auf dem Laufsteg“: Monte-Carlo-Primzahl-Test (Solovay, Strassen, 1970er)
	
	Satz („kleiner Fermat“):
	$$n \text{ prim} \quad \Rightarrow \quad a^{n-1} = 1 \,(\mathrm{mod}\, n) \;\; \forall a \in \mathbb{Z}_n \setminus \{0\}$$
	
	Algorithmus *Monte-Carlo-Primtest*
	Var n, a, k, i // Integer-Variablen: n für zu testende Zahl, a für Zufallszahl aus $\mathbb{Z}_n$
	k für die Größe der Stichprobe, i für den Schleifenzähler
	Input „Zu testende Zahl?“, n
	Input „Größe der Stichprobe?“, k
	For i = 1 to k do
	Bestimme eine Zufallszahl $a \in \mathbb{Z}_n \setminus \{0\}$
	Berechne $a^{n-1} \bmod n$
	If $a^{n-1} \bmod n \neq 1$ then
	Print „Zahl ist sicher nicht prim.“
	Goto End
	Endif
	Endfor
	Print „Zahl ist wahrscheinlich prim.“
	End
	
	Wir groß ist die Irrtums-Wahrscheinlichkeit? Solovay und Strassen haben Folgendes nachweisen können: Bei einer einzigen Zufallszahl beträgt die Wahrscheinlichkeit, dass der Fermat-Test versagt, höchstens 0.5. Solovay und Strassen haben weiter zeigen können, dass die Wahrscheinlichkeit dafür, dass beide Tests (Fermat und Lagrange) versagen, weniger als 0.25 beträgt. Das mag enttäuschend klingen, aber nur schon bei 50 Zufallszahlen wird der Algorithmus sich praktisch nie mehr irren, denn
	$$0.25^{50} \approx 7.9 \cdot 10^{-31}.$$

17	Ideen für „Algorithmen auf dem Laufsteg“: Euklidischer Algorithmus Zum Nachdenken! Wie genau läuft der Euklidische Algorithmus ab, wenn man ihn mit den Zahlen 1802 und 1054 füttert? Wie viele Schritte sind nötig, und welches ist der ggT der beiden Zahlen? Können Sie zwei zweistellige Zahlen finden, bei denen der Euklidische Algorithmus neun

	Schritte benötigt bis zur Terminierung? Können Sie zwei zweistellige Zahlen finden, bei denen der Euklidische Algorithmus zehn Schritte benötigt? Wenn Sie eine Art Casting-Show mit Paaren von natürlichen Zahlen durchführen und ein Paar desto besser abschneidet, je mehr Schritte der Euklidische Algorithmus benötigt, was für Paare haben dann gute Siegeschancen? Wie fällt Ihre Antwort aus, wenn die Paare in verschiedenen Ligen mitspielen können: in der Liga der zweistelligen Zahlen, in der Liga der dreistelligen Zahlen und so weiter? Und was ist dann speziell an den einzelnen Schritten? Satz: Anzahl Divisionen im Euklidischen Algorithmus $\geq n \quad \Rightarrow \quad a_0 \geq F_{n+2}$ und $a_1 \geq F_{n+1}$ (Leicht mit vollständiger Induktion zu beweisen.) Satz von Lamé (1844): Die Anzahl benötigter Divisionen im Euklidischen Algorithmus ist stets $\leq 5 \cdot (\text{Stellenzahl von } a_1)$.

18	Ideen für „Algorithmen auf dem Laufsteg“: Türme von Hanoi (Rekursion)
	
	 
	
	Algorithmus Hanoi(n, s, z, h)
	
	Var n, s, z, h // Vier Integer-Variablen für Turmhöhe, Startnadel, Zielnadel und Hilfsnadel
	If n = 0, then
	Goto end // Abbruch des Algorithmus, sobald Turmhöhe 0 erreicht ist
	Else
	Hanoi(n-1, s, h, z)
	Print „Scheibe von“, s, „nach“ z
	Hanoi(n-1, h, z, s)
	Endif
	end
	
	Aufwandanalyse…

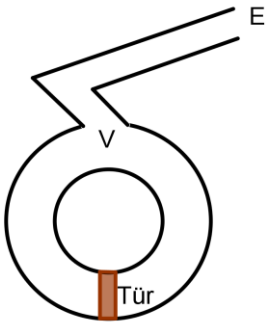
19	Ideen für „Algorithmen auf dem Laufsteg“: Numerische Integration
20	Ideen für „Algorithmen auf dem Laufsteg“ Sortieren Sortieren durch direktes Einfügen…

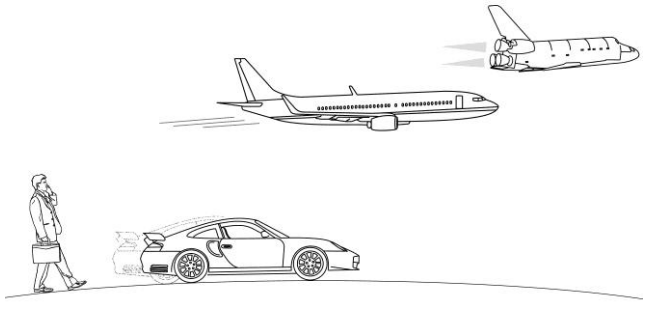
<table>
<tr><td></td><td> 
Sortieren durch direktes Auswählen…
 
Bubble Sort… usw.</td></tr>
<tr><td>21</td><td>Ideen für „Algorithmen auf dem Laufsteg“: Kryptographie

Caesar-Code, Maria Stuarts Hinrichtung, Vigenère, Enigma, usw.

Idee von Diffie & Hellman (1976)
Umsetzung durch Rivest, Shamir, Adleman (1978) → RSA
(Restklassen-Arithmetik, Euklidischer Algorithmus, Lemma von Bézout, Eulersche Phi-Funktion, Verallgemeinerung des „kleinen Fermat“,…)</td></tr>
<tr><td>22</td><td>Ideen für „Algorithmen auf dem Laufsteg“: Dijkstra – So schnell wie möglich von A nach B

Die Hauptkritik an Dijkstras Algorithmus lautet so: Es macht oft wenig Sinn, den gesamten Graphen abzusuchen; sinnvoller wäre es, das Suchgebiet möglichst einzuschränken auf den im Hinblick auf die Zielsetzung relevanten Teil. Der klassische *A*-Search-Algorithmus* tut genau dies. Er benutzt geschickte Abschätzungen der Distanzen zum Zielknoten, um damit die Auswahl der Knoten zu steuern, die besucht werden sollen. Im Jahr 2005 hatten Andrew V. Goldberg und Chris Harrelson eine glänzende Idee, wie das Suchgebiet noch mehr eingeschränkt werden kann. Ihre sogenannten *ALT-Algorithmen* (A* plus Landmarks plus Triangle Inequality) wählen zuerst einige Landmarks $L_1, L_2, L_3, \ldots$ aus und berechnen und speichern dann die kürzesten Distanzen sämtlicher Knoten zu diesen Landmarks. Es bezeichne $d_i(v)$ eine solche im Voraus berechnete kürzeste Distanz des Knotens v zum Landmark L_i. Wenn nun jemand den kürzesten Weg von Knoten v zu Knoten w wissen will, so ist aufgrund der Dreiecksungleichung klar, dass</td></tr>
</table>

	$$d(v,w) \geq d_i(v) - d_i(w)$$ sein muss. Indem man das Maximum all dieser unteren Schranken über alle Landmarks wählt, findet man eine gute untere Schranke für die gesuchte Distanz und kann diese benutzen bei der eigentlichen Berechnung des kürzesten Weges.
23	Ideen für „Algorithmen auf dem Laufsteg“: Zero-Knowledge-Protokolle Quisquater, J.-J., Guillou, L., Annick, M., Berson, T. (1989): “Höhlengleichnis”:  Zum Nachdenken! Sei $a \in \mathbb{Z}_n$. Eine Zahl $b \in \mathbb{Z}_n$ heißt Quadratwurzel von a, falls $b^2 = a$ in $\mathbb{Z}_n$ gilt. Versuchen Sie herauszufinden, wie viele Quadratwurzeln eine Zahl in $\mathbb{Z}_n$ haben kann, wenn n eine Primzahl ist. Untersuchen Sie vielleicht konkrete Beispiele wie etwa $n = 3$ oder $n = 5$. Stellen Sie anschließend eine allgemeine Vermutung auf, und versuchen Sie, diese zu beweisen. Ist $n = p \cdot q$ mit verschiedenen ungeraden Primzahlen p und q – und um eine solche Zahl geht es ja bei Fiat-Shamir –, so ist die Frage nach den Quadratwurzeln schon etwas schwieriger zu beantworten. Untersuchen Sie das Beispiel $\mathbb{Z}_{21} = \mathbb{Z}_{3\cdot 7}$, und stellen Sie anschließend eine Vermutung auf. (Falls Sie den *Chinesischen Restsatz* kennen, lässt sich die Vermutung relativ leicht beweisen.) Sei nun $b_1 \in \mathbb{Z}_n^\times$ und $a := b_1^2 \bmod n$, noch immer mit $n = p \cdot q$. Angenommen, es existiert tatsächlich ein effizienter Algorithmus, etwa der Algorithmus *sqrt*, zur Bestimmung einer Quadratwurzel, so könnte man jetzt *sqrt* auf die Zahl a anwenden und erhielte dann eine Quadratwurzel b_2. Da es, wie oben vermutet, genau vier Quadratwurzeln der Zahl a gibt, ist mit einer Wahrscheinlichkeit von 0.5 $b_2 \notin \{b_1, n - b_1\}$. Mit anderen Worten: Nach durchschnittlich zwei Wiederholungen der Prozedur *sqrt* hätte man eine Quadratwurzel mit dieser Eigenschaft gewonnen. Für eine solche gälte nun weiter: $$(b_1 + b_2)(b_1 - b_2) = b_1^2 - b_2^2 = a - a = 0 \pmod n$$ Können Sie herausfinden, wie nun mit Hilfe dieser Tatsache eine effiziente Faktorisierung von n durchgeführt werden könnte? Diese Überlegungen zeigen, dass das Berechnen einer Quadratwurzel gleich schwierig wie das Faktorisieren ist.
24	Effizienz von Algorithmen: Zum Nachdenken! Versuchen Sie einmal mehr, algorithmisch zu denken. Sie sind ein fleischgewordener Algorithmus, dem als Input eine bestimmte Zahl x übergeben wird und dessen Aufgabe nun darin

	besteht, die dreizehnte Potenz dieser Zahl zu berechnen. Dazu steht Ihnen aber nur die Multiplikation zur Verfügung. Wie gehen Sie dabei vor? Wie viele und welche Multiplikationsschritte benötigen Sie? Und wie viele Speicherplätze zur Speicherung von Zwischenresultaten? Was ist die minimale Anzahl Multiplikationen, die Sie verwenden müssen? Und wie können Sie ganz sicher sein, dass es mit weniger Multiplikationen nicht geht? Zur Berechnung von x^n aus dem Input x bei ausschließlicher Verwendung von Multiplikationsschritten sind mindestens $ceiling(\log_2(n))$ Operationen nötig.
25	Effizienz von Algorithmen: Zum Nachdenken! Diesmal bittet Sie jemand, ein Polynom auszuwerten. Als Input erhalten Sie die Koeffizienten $a_0, a_1, ..., a_n$ sowie die Zahl r, und Ihre Aufgabe besteht nun darin, das Polynom $$a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + ... + a_2 \cdot x^2 + a_1 \cdot x + a_0$$ an der Stelle r auszuwerten. Dafür dürfen Sie lediglich Multiplikationen und Additionen benutzen. Wie viele und welche Schritte benötigen Sie, um diese Aufgabe zu lösen? Horner, dann Ostrowski 1954
26	Effizienz von Algorithmen: Zum Nachdenken! Führen Sie sich erneut den Algorithmus Bubble-Sort zu Gemüte. Was würden Sie als einzelnen Schritt dieses Algorithmus bezeichnen? Was ist die Inputgröße? Und wie viele Schritte abhängig von der Inputgröße benötigt Bubble-Sort folglich bis zur Terminierung?
27	Effizienz von Algorithmen: Geschwindigkeitsklassen → Idee der O-Notation (Landau) Wir schreiben $S(n) = O(f(n))$ (lies: „Gross O von f von n“, eigentlich aber Gross Omikron), genau dann wenn die Werte der Funktion $S(n)$ schließlich, das heißt ab einer bestimmten Stelle n_0, stets $\leq c \cdot f(n)$ sind für eine Konstante c. Formal ausgedrückt: $$S(n) = O(f(n)) \quad \Leftrightarrow \quad \exists n_0, c: S(n) \leq c \cdot f(n) \ \forall n \geq n_0$$ 

28 Effizienz von Algorithmen:

Das ist von erheblicher praktischer Relevanz. Während zum Beispiel der klassische Algorithmus zur Berechnung einer Determinante zur katastrophalen Aufwandklasse $O(n!)$ gehört, gehört der Gauß-Jordan-Algorithmus für Determinantenberechnung zur Aufwandklasse $O(n^3)$. Mit letzterem lässt sich folglich eine 20×20-Determinante in deutlich weniger als einer Sekunde rechnen, während ersterer viele Jahre benötigen würde. Und während der in der Signaltechnik so wichtige Algorithmus DFT (Discrete Fourier-Transformation) eine für viele Anwendungen zu lange Laufzeit von $O(n^2)$ aufweist, benötigt der im Jahr 1965 von J. W. Cooley und J. W. Tuckey entwickelte Algorithmus FFT (Fast Fourier-Transformation) lediglich $O(n \cdot \log(n))$ Schritte.

Auch sehr interessant sind Aufwandanalysen für Primtest-Algorithmen. Während die klassischen Algorithmen (inklusive Eratosthenes) exponentiell in der Länge der Inputzahl laufen, haben die drei indischen Mathematiker Agrawal, Kayal und Saxena im Jahr 2002 einen Algorithmus gefunden, welcher zur Klasse $O\left((\log(n))^{12}\right)$ gehört und somit in polynomialer Zeit läuft, was damals eine Sensation war.

29 Effizienz von Algorithmen:

	$n=20$	$n=40$	$n=60$	$n=100$	$n=500$
$n \cdot \log(n)$	0.00000003 s	0.00000007 s	0.0000001 s	0.0000002 s	0.000002 s
n^2	0.0000004 s	0.000002 s	0.000004 s	0.00001 s	0.0003 s
n^5	0.003 s	0.1 s	0.8 s	10 s	8.7 h
2^n	0.001 s	0.3 h	37 a	$4 \cdot 10^{13}$ a	10^{134} a
n^n	3325012683 a	$4 \cdot 10^{47}$ a	$1.5 \cdot 10^{90}$ a	$3 \cdot 10^{183}$ a	∞

30 Effizienz von Algorithmen: Multiplikation von Zahlen:

- Schulmethode: $O(n^2)$
- Karatsuba & Ofman 1962: $O(n^{1.58})$
- Schönhage & Strassen 1971: $O(n \cdot \log(n) \cdot \log(\log(n)))$

31 Effizienz von Algorithmen: Matrixmultplikation

Zum Nachdenken!

Der klassische Algorithmus für Matrixmultiplikation befolgt einfach die oben notierte Definition, das heißt, er berechnet der Reihe nach für jeden der n^2 Einträge der Produktmatrix das Skalarprodukt aus einer Zeile der ersten und einer Spalte der zweiten Matrix. Formulieren Sie hierfür einen Algorithmus in Pseudocode. Wie viele Multiplikationen und wie viele Additionen werden ausgeführt? Welchen Aufwand, notiert in der Landau-Symbolik, hat dieser Algorithmus folglich?

Angenommen, Sie dürften die besten Fachleute der Welt bitten, einen möglichst effizienten Algorithmus für Matrixmultiplikation zu liefern, welchen asymptotischen Aufwand wird dann keiner dieser Fachleute unterschreiten können? Und weshalb?

32 Effizienz von Algorithmen: Matrixmultiplikation

Datum	Entdecker	Exponent von n
1969	Strassen	2.807
Oktober 1978	Pan (**Fehler! Verweisquelle konnte nicht gefunden werden.**)	2.795
November 1978	Bini e.a.	2.78
Juni 1979	Schönhage	2.609
Oktober 1979	Pan	2.605
Oktober 1979	Schönhage	2.548
Oktober 1979	Pan und Winograd	2.522
März 1980	Pan	2.49
September 1986	Coppersmith und Winograd	2.376

33 Effizienz von Algorithmen: Beschleunigung des Sortierens

Z.B. Quicksort…

34 Effizienz von Algorithmen: Einführung in die Komplexitätstheorie

Berechnungsfolgen, (lineare, multiplikative,…) Komplexität.

Bsp.: Satz von Scholz (1937):

$$\lceil \log_2(n) \rceil \le L_m(x^n) \le 2 \cdot \log_2(n)$$

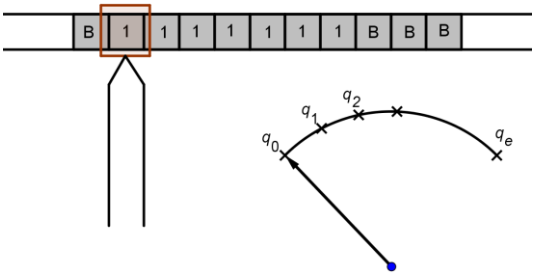
35 Von Llulls Ars Magna bis zum Entscheidungsproblem…

36 Turing-Maschinen

Eine *Turing-Maschine* (TM) besteht aus

- einem endlichen Alphabet A von Symbolen, die zur Ein- und Ausgabe von Daten dienen,
- einem endlichen Alphabet $\overline{A} \supset A$ von Symbolen, die zur Berechnung der Zwischenschritte dienen,

- einem Leerzeichen (Blank) $B \in A$,
- einem eindimensionalen (theoretisch unbegrenzten) Band mit Feldern, von denen jedes genau ein Symbol aus A enthalten kann,
- einem beweglichen Lese/Schreibkopf, der ein Feld aufs Mal lesen oder beschreiben kann und danach eine von drei Bewegungen (ein Feld nach links rücken, ein Feld nach rechts rücken oder stehenbleiben) ausführen kann,
- einer endlichen Menge Q von *Zuständen* (deren Bedeutung bald geklärt wird),
- zwei speziellen Zuständen q_0 und q_e (beide aus Q), die *Anfangszustand* und *Endzustand* heißen – und
- einer (partiellen) Funktion δ, die auch *Programm* heißt und die einigen Paaren (Zustand , Symbol) ein Tripel aus einem (neuen) Zustand, einem (neuen) Symbol und einem der Zeichen L (für links), R (für rechts), S (für Stop) zuordnet. Bei dem Programm handelt es sich also um eine partielle Funktion $\delta : Q \times A \to Q \times A \times \{L, R, S\}$.



37 Turing-Maschinen:

z.B. Zwei Zahlen (Folgen von Einsen) addieren…

$$(q_0,1) \mapsto (q_0,1,R)$$
$$(q_0,B) \mapsto (q_1,1,R)$$
$$(q_1,1) \mapsto (q_1,1,R)$$
$$(q_1,B) \mapsto (q_2,B,L)$$
$$(q_2,1) \mapsto (q_3,B,L)$$
$$(q_3,1) \mapsto (q_e,B,S)$$

Turing-Maschinen lösen mindestens zwei Arten von Problemen: Einerseits dienen sie dazu, *zahlentheoretische Funktionen* $f : \mathbb{N}^k \to \mathbb{N}$ zu berechnen. Das heißt, man schreibt einen Input $\underline{n} = (n_1, n_2, ..., n_k) \in \mathbb{N}^k$ aufs Band, und die Maschine berechnet dann $f(\underline{n})$ und terminiert mit einer Kette von $f(\underline{n})+1$ Einsen auf dem Band. Wir haben das ja soeben getan für die Funktion $f(n_1, n_2) = n_1 + n_2$. Einfach gesagt: Eine Turing-Maschine kann dazu benutzt werden, Funktionswerte zu berechnen.

Andererseits kann sie aber auch herangezogen werden, um Entscheidungsprobleme zu lösen.

Mit A^* bezeichnen wir die Menge aller Wörter, die sich mit Symbolen aus dem Alphabet A bilden lassen: Ein *Wort* ist einfach eine endliche Kette von Zeichen aus dem Alphabet. Unter einer *Sprache* verstehen wir eine Teilmenge $L \subset A^*$.

	Das Problem ist dann zu entscheiden, ob irgendein Wort $w \in A^*$ zu der gegebenen Sprache L gehört oder nicht. Eine TM, die dieses Problem löst, muss zusätzlich mit einem sogenannten akzeptierenden Zustand q_a ausgerüstet werden, in den sie übergeht, wenn sie w als zur Sprache L gehörig erkannt hat. Falls es eine TM gibt, die bei jedem Input nach endlich vielen Schritten anhält und nur genau bei den Elementen aus L in den akzeptierenden Zustand übergeht, so nennt man die Sprache *entscheidbar*.
38	Turing-Maschinen: Aufgaben a) testen, ob eine Inputzahl (Folge von Einsen) gerade ist oder nicht… b) durch lauter Einsen dargestellte Inputzahl in dezimale Notation umwandeln… c) Kopierprogramm… d) Euklidischer Algorithmus
39	Turing-Maschinen: Definition: Ein *Algorithmus* ist eine anhaltende Turing-Maschine.
40	Turing-Maschinen: These von Church-Turing (1936) Alles, was im intuitiven Sinne algorithmisch berechenbar ist, kann von einer Turing-Maschine berechnet werden.
41	Turing-Maschinen: Warum ist das eine These? Und warum ist sie plausibel? Zum einen hat man, seit Computer gebaut werden, niemals die Situation angetroffen, dass die eine Maschine grundsätzlich mehr kann als alle anderen. Nie hat bisher jemand eine Computerarchitektur entworfen, von der man annehmen könnte, dass dank ihr Probleme gelöst werden können, die bisher als algorithmisch unlösbar galten. Zum anderen hat man bisher keinen Algorithmus, kein intuitiv berechenbares Problem gefunden, zu dem nicht ein Turing-Maschinen-Programm entwickelt werden konnte. Allein diese erdrückende Menge positiver Befunde macht die Wahrscheinlichkeit, dass in der Zukunft ein Algorithmus entdeckt wird, zu dem bewiesenermaßen kein TM-Programm geschrieben werden kann, überaus klein. Und zu guter Letzt haben andere Mathematiker andere Definitionen von Algorithmus vorgeschlagen, so etwa Alonzo Church den λ-*Kalkül* (der später in der Programmiersprache LISP Verwendung finden sollte) oder Kurt Gödel und Stephen Kleene die (*partiell*) *rekursive Funktion* . Aber von allen vorgeschlagenen Definitionen konnte bewiesen werden, dass sie untereinander äquivalent sind. Church und Turing waren die ersten, die solche Äquivalenzbeweise leisten konnten. Das ist ein wirklich starkes Argument. Wenn mehrere glänzende Forscher unabhängig voneinander Wege suchen, den Algorithmus durch ein exaktes mathematisches Konzept zu erfassen, und wenn es sich herausstellt, dass all diese Konzepte genau dieselbe Klasse von Problemen lösen, dann bedeutet das, dass ein weitgehender Konsens darüber besteht, was man unter „intuitiver Berechenbarkeit“ verstehen soll und dass dieses Konzept gut und tief verstanden wird.
42	Turing-Maschinen: Fantastische Idee: Programme sind Daten!!! Also kann man einer Maschine ein Programm

	eingeben und sie somit universell machen!!! $$\text{Code}(q_u, Z_v, Bew.) = 2^u \cdot 3^v \cdot 5^{\text{Code der Bewegung}}$$ Sind C_1, C_2, ..., C_s die Codes der einzelnen Konklusionen in der standardisierten Reihenfolge des TM-Programms, so wählen wir als Code des ganzen Programms die Zahl $$p_1^{C_1} \cdot p_2^{C_2} \cdot \ldots \cdot p_s^{C_s},$$ wobei p_i die aufeinanderfolgenden Primzahlen 2, 3, 5, 7, 11, … sind. Wir nennen diesen Code auch *Gödelnummer* (GN) der Turing-Maschine.
43	Unberechenbare Funktionen: Die Menge aller TM ist abzählbar. Angenommen, eine sortiere Liste aller totalen Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ wäre tatsächlich möglich und sie würde so beginnen: f_1: **1** 4 9 16 25 36 49 64 ... f_2: 4 **11** 18 25 32 39 46 53 ... f_3: 1 1 **1** 1 1 1 1 1 ... f_4: 1 2 3 **4** 5 6 7 8 ... f_5: 9 8 0 0 **13** 17 4 1 ... f_6: 5 6 4 7 3 **8** 2 9 ... f_7: 19 20 21 22 23 24 **25** 26 ... f_8: 7 7 2 10 0 3 5 **7** ... ... **...** Wir nehmen also an, er wäre eine Gesetzmässigkeit entdeckt worden, die wir hier allerdings nicht erklären (können), die *alle* Funktionen in eine eindeutige Reihenfolge bringt und die zu jeder beliebigen vorgelegten Funktion entscheiden kann, an welcher Stelle dieser Liste sie stehen muss. Genau so etwas ist für die Menge aller Turing-Maschinen ja möglich. Mit Cantor können wir nun aber zeigen, dass es mindestens eine Funktion geben muss, die in dieser Liste fehlt, nämlich die *Diagonalfunktion*: $$d : n \mapsto f_n(n) + 1$$
44	Die Funktion *busy beaver* $bb : \mathbb{N} \to \mathbb{N}$ ist so definiert: $bb(n)$ bezeichnet die maximale Anzahl zusammenhängender Einsen, die eine terminierende Turing-Maschine mit genau n Zuständen und zwei Zeichen auf ein leeres Band schreiben kann.

$bb(1)=1$

$bb(2)=4$

$bb(3)=6$

$bb(4)=13$

Danach sind die Funktionswerte nicht genau bekannt. 1989 haben Jürgen Buntrock und Heiner Marxen eine Turing-Maschine mit fünf Zuständen (und zwei Zeichen) entwickelt, die 4098 Einsen schrieb; daher weiss man, dass $bb(5) \geq 4098$ ist. Im Jahr 2010 hat Pavel Kropitz eine Turing-Maschine mit sechs Zuständen (und zwei Zeichen) entwickelt, mit der sich nachweisen lässt, dass $bb(6) > 3.514 \cdot 10^{18267}$ ist. Der Verlauf der Funktion ist wahrlich schwindelerregend. Zum Beispiel weiss man, dass

$$bb(12) \geq 6 \cdot 4096^{\left(4096^{4096^{\cdots^{\left(4096^4\right)}}}\right)}$$

ist, wobei die Zahl 4096 insgesamt 166mal erscheint. Das alles macht plausibel, dass kein Algorithmus gefunden werden kann, der die Werte dieser Funktion berechnen könnte.

Satz: Die Funktion bb ist nicht TM-berechenbar.

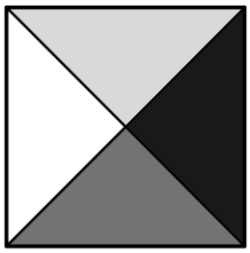
Wir geben hier einen einfachen Beweis an, der auf Charles B. Dunham zurückgeht. Dunham beweist aber eine Variante des obigen Satzes. Er benutzt statt der busy-beaver-Funktion die Funktion $D(n)$, die jeder natürlichen Zahl n die maximale Anzahl zusammenhängender Einsen zuordnet, die eine Turing-Maschine mit n Zuständen (und zwei Zeichen) schreiben kann, wenn sie auf ein Band mit einem Block von n Einsen angesetzt wird und anhalten soll. Im Unterschied zu bb trifft bei dieser Variante die Maschine also nicht ein leeres Band an, sondern eines mit einer Kette von Einsen, die gerade so viele Einsen enthält, wie die Maschine Zustände besitzt. Dann zeigt Dunham wie folgt, dass die Funktion D nicht TM-berechenbar sein kann. Als Folge kann dann natürlich auch bb nicht TM-berechenbar sein.

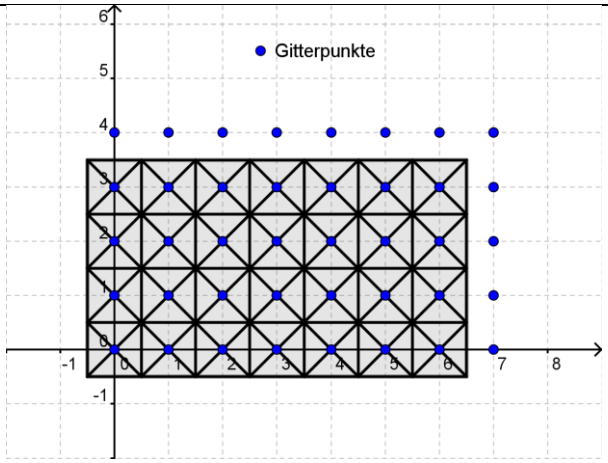
Angenommen, die Funktion D wäre TM-berechenbar. Dann müsste also eine Turing-Maschine mit zwei Zeichen, sagen wir die Maschine M, existieren, die jeden Input n in endlich vielen Schritten in den Output $D(n)$ umwandelt und dann anhält. Wenn das so wäre, dann liesse sich sicher auch eine andere Maschine M^*, ebenfalls mit zwei Zeichen, konstruieren, die jeden Input n in endlich vielen Schritten in den Output $D(n)+1$ umwandelt und dann anhält.

Angenommen, die Maschine M^* habe m^* Zustände. Dann wäre nach Definition dieser Maschine $M^*(m^*) = D(m^*)+1$. Andererseits wäre M^* eine anhaltende Turing-Maschine mit m^* Zuständen und zwei Zeichen, die auf eine Kette von m^* Einsen angesetzt wird. Sie spielt also in der Liga aller Maschinen dieses Typs mit und wird folglich höchstens so viele Einsen produzieren können wie die beste Maschine in dieser Liga. Daher ist auch $M^*(m^*) \leq D(m^*)$.

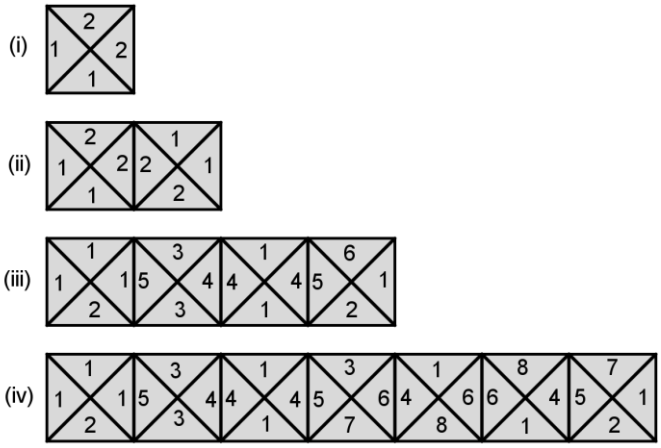
Zusammengesetzt erhalten wir $D(m^*)+1 \leq D(m^*)$, also einen Widersprich. Daher kann die Funktion D nicht TM-berechenbar sein.

45 Es bezeichne n die Gödelnummer einer Turing-Maschine M und x einen beliebigen (im Alphabet dieser Maschine geschriebenen) Input. Unter dem *Halteproblem* (HP) versteht man

	folgendes Problem: Kann man eine Turing-Maschine angeben, die für jedes Paar (n, x) entscheidet, ob M mit Input x nach endlich vielen Schritten zu einer Stop-Anweisung gelangt oder nicht? In anderen Worten: Ist die Sprache $$L := \{(n, x) \text{ / TM mit GN } n \text{ und Input } x \text{ hält an}\}$$ TM-entscheidbar? Satz von Turing (1936): Das Halteproblem ist algorithmisch unentscheidbar.
46	Zum Nachdenken! Jemand schlägt vor, das Halteproblem folgendermaßen algorithmisch zu lösen: Zuerst werden die Zahlen n (Gödelnummer eines Programms) und x (Input für dieses Programm) eingelesen. Dann wird n dechiffriert, bis alle Befehle des Programms vorliegen. Dann simuliert der Algorithmus das Programm mit dem Input x, und falls das Programm anhält, gibt der Algorithmus σ aus, sonst τ. Können Sie argumentieren, weshalb das eben keine algorithmische Lösung des Halteproblems ist? Woran liegt das genau? Und: Wiederholen Sie die zentrale Beweisidee anhand der folgenden Variante: Wiederum bezeichne n die Gödelnummer einer Turing-Maschine M. Können Sie beweisen, dass kein Algorithmus existieren kann, der bei Input n Output 1 liefert, falls M bei Input n 0 liefert, und der bei Input n Output 0 liefert, falls M bei Input n 1 liefert? Was würde passieren, wenn man dieser Maschine ihre eigene Gödelnummer eingeben würde?
47	Methode der Reduktion: Eine *Reduktionsfunktion, die eine Sprache B auf das Halteproblem reduziert*, ist eine Funktion r, die 1. TM-berechenbar ist, 2. jedem Paar (n, x) genau ein y zuordnet und 3. die Eigenschaft $(n, x) \in HP \Leftrightarrow y = r(n, x) \in B$ hat. Freilich gibt es auch Reduktionsfunktionen für andere Probleme; diese werden analog definiert. Ein Problem B heißt *auf das Halteproblem reduzierbar*, in Zeichen $HP \leq B$, wenn eine Reduktionsfunktion r existiert, die B auf HP reduziert.
48	Bsp.: Translationsproblem…
49	Das Fliesenproblem von Hao Wang: 



Kann mit der folgenden Fliesenserie der erste Quadrant vollständig gefliest werden?



Wir formulieren das Problem noch etwas präziser: Gegeben ist jeweils eine endliche Menge F von Fliesen der oben beschriebenen Art und zudem die Eck-Fliese c (corner). Jede Fliese ist ein Quadrupel (o,r,u,l) aus vier Symbolen, die die Farben im oberen (o), rechten (r), unteren (u) und linken (l) Sektor codieren. Das Problem ist gelöst, wenn wir eine Funktion $f:\mathbb{N}\times\mathbb{N}\to F$ angeben können, die jedem Gitterpunkt des unbegrenzten ersten Quadranten einen Fliesentyp zuordnet, und wenn diese Funktion zudem die Farb-Bedingung erfüllt: Für alle $i,j\geq 0$ muss

$$f(i,j)_2 = f(i+1,j)_4 \text{ und } f(i,j)_1 = f(i,j+1)_3$$

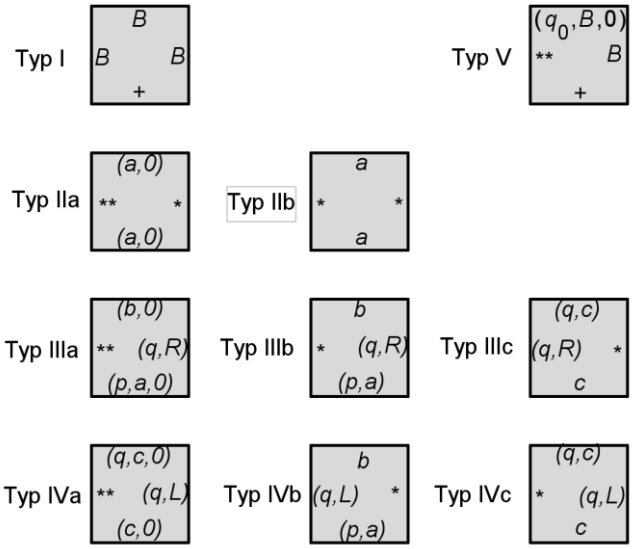
gelten. Dass c die Eck-Fliese sein muss, notieren wir einfach in der Form $f(0,0)=c$.

Die Sprache
$FP := \{(F,c) \,/\, \exists f:\mathbb{N}\times\mathbb{N}\to F \text{ mit } f(0,0)=c \text{ und erfüllter Farb-Bedingung}\}$ nennen wir *Fliesenproblem*.

Satz: FP ist algorithmisch unentscheidbar.

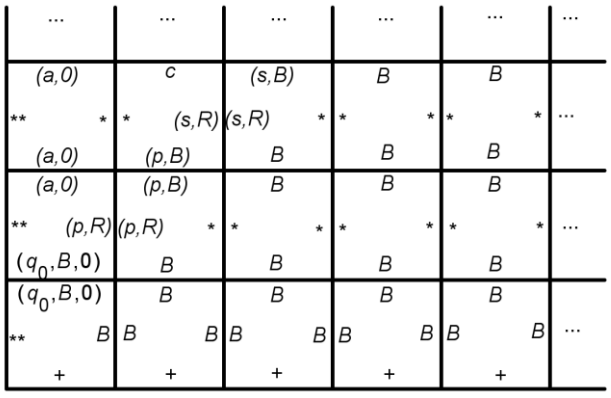
50 Beweis der Unentscheidbarkeit des Fliesenproblems durch Reduktion auf HP_{var}:

Sei M eine beliebige Turing-Maschine, die im Startzustand q_0 das Blank ganz links sieht. (Wir denken uns das Band also links begrenzt.) Ferner gibt es eine endliche Menge Q von Zuständen und ein endliches Alphabet A von Zeichen. Wir ordnen dieser Maschine die folgende Fliesenserie zu (vgl. Abb.):



- Es soll eine unbegrenzte Anzahl Fliesen vom Typ I geben. Dazu sind offenbar die „Farben“ B und + nötig.
- Zu jedem Zeichen $a \in A^*$ soll es eine unbegrenzte Anzahl Fliesen vom Typ IIa und IIb geben. Dazu benötigen wir als weitere Farben $*$, $**$, a und $(a,0)$. Die Fliesen vom Typ IIa werden dann den linken Rand des Quadranten bilden.
- Zu jedem Befehl der Art $(p,a) \mapsto (q,b,R)$ und jedem Zeichen $c \in A^*$ soll es eine unbegrenzte Anzahl Fliesen von den Typen IIIa, IIIb und IIIc geben, wozu wiederum weitere Farben eingeführt werden müssen.
- Zu jedem Befehl der Art $(p,a) \mapsto (q,b,L)$ und jedem Zeichen $c \in A^*$ soll es eine unbegrenzte Anzahl Fliesen von den Typen IVa, IVb und IVc geben, wozu wiederum weitere Farben eingeführt werden müssen.
- Schliesslich soll es noch eine Fliese vom Typ V geben, die Eck-Fliese.

Draus ergibt sich folgende Fliesung:



Und so weiter. Wir können den Quadranten also genau dann vollständig fliesen, wenn die zugrunde liegende Turing-Maschine nie zu einer Stop-Anweisung gelangt.

51 Weitere algorithmisch unlösbare Probleme:

Postsches Korrespondenzproblem

52 Weitere algorithmisch unlösbare Probleme_

„There are no safe virus tests“
(W. F. Dowling, Drexel University, Philadelphia, 1989)

Beweis:

Definition: Ein Computerprogramm heisst *Virus*, gewenn es, nachdem es gestartet wird, das Betriebssystem (BS) ändert.

Definition: Ein Computerprogramm *P*, gestartet mit Input *x* in der Umgebung des BS, *verbreitet ein Virus*, gewenn es das BS ändert; sonst heisst es *sicher*.

Abmachung: Es existiert mindestens ein Virus, nennen wir es *V*.

Angenommen, $\exists$ sichere Virus-Test-Machine *VTM*, die zu jedem Paar (Programm *P*, Input *x*) entscheiden kann, ob *P* sicher ist mit *x* oder nicht:

$$(P,x) \mapsto \boxed{VTM} \mapsto \begin{cases} \text{"}sicher\text{"}, \text{ falls } P \text{ sicher mit } x \\ \text{"}nicht\ sicher\text{"}, \text{ sonst} \end{cases}$$

$\Rightarrow \exists$ auch Maschine *DEVIL*:

$$P \mapsto \boxed{DEVIL} \mapsto \begin{cases} \text{"}I\ love\ you\text{"}, \text{ falls } VTM(P, GN(P)) = \text{"}nicht\ sicher\text{"} \\ \text{verändert BS mit Hilfe von } V, \text{ sonst} \end{cases}$$

Ist nun *DEVIL* sicher mit Input *GN(DEVIL)* oder nicht?

- Falls *DEVIL* sicher mit *GN(DEVIL)*
  $\Rightarrow$ *DEVIL* schreibt *I love you*
  $\Rightarrow$ *VTM(DEVIL,GN(DEVIL))="nicht sicher"*
  $\Rightarrow$ *DEVIL* ist nicht sicher mit Input *GN(DEVIL)*. WIDERSPRUCH!

- Falls *DEVIL* nicht sicher mit *GN(DEVIL)*
  $\Rightarrow$ Entweder verändert *DEVIL* das BS mit *V* (dann ist *VTM(DEVIL;GN(DEVIL))="sicher"*, d.h. *DEVIL* ist sicher mit *GN(DEVIL)*) oder *DEVIL* schreibt „*I love you*“ (dann muss der Aufruf von *VTM* das Virus verursachen, d.h. *VTM* ist selber nicht sicher). In beiden Fällen resultiert ein WIDERSPRUCH!

□

Richardsons Taschenrechner-Sabotage

Sei E eine Menge von Termen über $\mathbb{R}$ mit höchstens einer Variablen x, z.B.

$$E = \left\{ \frac{3}{5},\ x,\ 0.2x^2,\ \sqrt{x},\ \frac{2}{x-1}, \ldots \right\}$$

Sei E^* die Menge der Funktionen, repräsentiert durch die Terme aus E. (Ist z.B. der Term $A = x^2$ in E, so ist die Funktion $A(x) = x^2$ in E^*.)

Annahmen über E^*:

- Die Funktion $id(x) = x$ ist enthalten.
- Die Funktionen $r(x) = \frac{m}{n}$ sind enthalten $\forall \frac{m}{n} \in \mathbb{Q}$.
- E^* ist abgeschlossen unter Addition, Subtraktion, Multiplikation und Komposition.

$A(x) \equiv B(x)$ soll bedeuten, dass beide Definitionsmengen übereinstimmen und zudem $\forall x \in D_A = D_B : A(x) = B(x)$.

Mögliche Prämissen für E^*:

(1) E^* enthält log(2), π, e^x und $\sin(x)$.

(2) E^* enthält die Betragsfunktion

(3) E^* enthält eine (totale) Funktion $\varphi(x)$, so dass für keine Funktion $f(x)$ in E^* gilt: $f'(x) = \varphi(x)$.

<u>Sätze von D. Richardson (1968):</u>

E^* erfüllt (1) $\Rightarrow$ Es kann algorithmisch NICHT entschieden werden, ob zu einer beliebigen Funktion $A(x)$ aus E^* ein $x \in \mathbb{R}$ existiert, so dass $A(x) < 0$ ist.

E^* erfüllt (1)&(2) $\Rightarrow$ Es kann algorithmisch NICHT entschieden werden, ob für eine beliebige Funktion $A(x) \in E^*$ gilt: $A(x) \equiv 0$

E^* erfüllt (1)&(2)&(3) $\Rightarrow$ Es kann algorithmisch NICHT entschieden werden, ob zu einer beliebigen Funktion $A(x)$ aus E^* eine Funktion $f(x)$ aus E^* existiert, so dass $f'(x) \equiv A(x)$.

- Wortproblem von Thue (1914)
- Hilbert Nr. 10 (1970)
- Usw.

53

NP-vollständige Probleme

Eine *nicht-deterministische* TM ist ein 3-Tupel (A, Q, δ), wobei

- Q eine endliche Menge von Zuständen
- I das Inputalphabet
- A das Bandalphabet mit $I \subseteq A$
- $B \in A \setminus I$ das Leerzeichen (Blank)
- q_0 der Startzustand
- q_a der akzeptierende Zustand
- $\boxed{\delta : Q \times A \to P(Q \times (A \times \{L, R, S\}))}$ die Programm-Funktion ist, die dem aktuellen Zu-

stand und dem aktuell gesehenen Zeichen den neuen Zustand und das neue Zeichen samt der Bewegungen zuordnet. (L=Links, R=Rechts, S=Stop)

Ein Funktionswert von δ ist aber ein Element der Potenzmenge von $Q\times\left(A\times\{L,R,S\}\right)$, d.h. es stehen der Maschine zu jedem Zeitpunkt mehrere mögliche Fortsetzungen offen! <u>*Ein*</u> möglicher Berechnungspfad der nicht-det. TM ergibt sich aus der Auswahl von jeweils einer der Aktionen aus $\delta\left(q,\left(a_1,...,a_k\right)\right)$.

P:= Menge aller Sprachen, die von einer deterministischen TM in polynomialer Zeit akzeptiert werden.

NP := Menge aller Sprachen, die von einer nicht-deterministischen TM in polynomialer Zeit akzeptiert werden. (NP = **N**on deterministic **p**olynomial)

Das *P-NP-Problem*:

$$\boxed{\text{Ist } P \subset NP \text{ oder } P = NP \text{ ?}}$$

Ein Problem *L* heisst *NP-vollständig*, gewenn

- $L \in NP$ und
- für jedes $L_1 \in NP$ $L_1 \leq_p L$ gilt.

(Das heisst, dass es eine polynomial-zeitbeschränkte deterministische TM gibt, die eine Funktion $f(x)$ berechnet, so dass $x \in L_1 \Leftrightarrow f(x) \in L$.)

<u>Folgerung:</u>
Kann von einem einzigen *NP-vollständigen* Problem gezeigt werden, dass es zu *P* gehört, dann ist automatisch *P=NP* !!!

<u>*NP-vollständige* Probleme:</u>

1) SAT – das erste als *NP-vollständig* erkannte Problem!
   (S. A. Cook, 1971)

2) *k*-Färbbarkeit von Graphen

3) Partition eines Graphen in Subgraphen mit Hamilton-Wegen

4) CLIQUE (Enthält ein Graph eine Clique der Grösse *k*?)

5) HAMILTONIAN CIRCUIT und H. PATH

6) SUBGRAPH ISOMORPHISM

7) DEGREE CONSTRAINT SPANNING TREE
   (Gibt es einen aufspannenden Baum, dessen Knoten alle Grad $\leq k$ haben?)

8) SHORTEST TOTAL PATH LENGTH SPANNING TREE

9) MAX CUT

(Gibt es eine Graphenpartition in 2 Subgraphen, so dass die Summe der Gewichte aller Kanten, die einen Subgraphen mit dem anderen verbinden, $\geq k$ ist?)

10) TSP

11) LONGEST CIRCUIT
(Gibt es einen circuit der Länge $\geq k$?)

12) MINIMUM EDGE-COST FLOW

13) RUCKSACK
(Menge von Waren, Gewichte, Wertzuordnung, geg. Höchstgewicht und Mindestwert. Gibt es Submenge von Waren, die Höchstgewicht und Mindestwert einhält?)

…und zahlreiche weitere Probleme aus den Gebieten:

- Mengen und Partitionen
- Datenspeicherung und Kompression
- Scheduling
- Algebra und Zahlentheorie
- Lösbarkeit von Gleichungen
- Spiele und Puzzles
- Logik
- Automatentheorie
- Programmoptimierung

54

Was vom Traum übrigblieb:

Der Traum von Leibniz kann in einzelnen engen mathematischen Theorien tatsächlich realisiert werden! Zwei dieser Theorien möchte ich im Folgenden kurz ansprechen. Im Anschluss daran werden wir uns noch mit einer interessanten Frage auseinandersetzen, die sich im Zusammenhang entscheidbarer Theorien eigentlich aufdrängt: Wie schnell ist eine entscheidbare Theorie entscheidbar? Genauer: Wie viele Schritte benötigt ein algorithmischer Test, ob eine bestimmte Formel wahr oder falsch ist, mindestens? Fragen dieser Art sind sehr neu, erst in den 60er-Jahren des letzten Jahrhunderts wurden die ersten mathematischen Sätze aus der sog. Komplexitätstheorie bewiesen, der Theorie eben, die nach der Komplexität, dem minimalen Aufwand von Algorithmen fragt.

Betrachten wir nun eine erste entscheidbare Theorie etwas genauer:

Es handelt sich um die **Theorie der dichten Ordnung (ThdO).** Diese mathematische Teildisziplin vereinigt in sich alle Sätze zum Thema „Ordnungsrelation über rationalen Zahlen", also alle Sätze, die eine Aussage über rationale Zahlen machen unter Verwendung der ordnenden Zeichen $=, \neq, <, \leq, >, \geq$.
Die THdO benutzt also die Zeichen $=, \neq, <, \leq, >, \geq$

ferner die logischen Zeichen $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, $\rightarrow$ samt den Hilfszeichen) (,

die rationalen Zahlen als Konstanten

die Buchstaben x, y, z, ... (auch indiziert) als Variabeln

die Axiome der Prädikatenlogik 1. Stufe

und die folgenden theoriespezifischen Axiome: (1) $\forall x\ (\neg (x<x))$

(2) $\forall x,y\ ((x<y)\ v\ (x=y)\ v\ (y<x))$

(3) $\forall x,y,z\ ((x<y)\ \wedge\ (y<z)\ \rightarrow\ (x<z))$

(4) $\forall x,y\ ((x<y)\ \rightarrow\ \exists z((x<z)\ \wedge\ (z<y))$

(5) $\forall x\ \exists y,z\ ((y<x)\ \wedge\ (x<z))$

(Das Axiom (4) bedeutet die Dichtheit; es gibt keine Löcher in dem durch das Zeichen „<“ definierten Ordnungsgefüge!)

Eine typische Aussage dieser Theorie könnte z.B. so aussehen:

$\exists y\ \exists x\ (\ (\ (x\leq y)\ \wedge\ (0\leq x)\ \wedge\ (x\neq y)\ \wedge\ (x\neq 5)\ )\ v\ (\ (y\leq x)\ \wedge\ (x\leq 0)\ \wedge\ (x\neq y)\ \wedge (y\neq -3)\ )\ )$

Die ThdO ist entscheidbar, d.h. es gibt tatsächlich einen Algorithmus, der automatisch von jeder beliebigen in ThdO formulierbaren Aussage entscheidet, ob sie wahr oder falsch ist. Eine genaue und detaillierte Beschreibung dieses Algorithmus würde den Rahmen dieses Vortages sprengen. Ich beschränke mich daher auf ein paar erläuternde Bemerkungen:

Sei also eine beliebige Formel F aus der ThdO gegeben.

1.) Zuerst reduzieren wir die 6 Zeichen $=, \neq, <, \leq, >, \geq$ auf zwei, indem wir alle Zeichen allein durch die beiden Zeichen $\leq, \neq$ ausdrücken:
   a=b ersetzen wir durch .....
   a>b ersetzen wir durch .....
   a<b ersetzen wir durch
   a≥b ersetzen wir durch .....
2.) Dann wird F in die sog. pränexe Normalform übergeführt, d.h. in eine Form, in der alle Quantoren ($\forall$, $\exists$) am Anfang der Formel stehen. Diese Umformung ist ausserordentlich trickreich, aber stets möglich. F nimmt dann also die Gestalt
   $\forall\ \forall\ \forall\ \forall\ \forall\ \forall\ \forall\ \exists$. $\exists$ ..(-------- quantorenfreier Teil ----------) an.
3.) Nun schafft man alle Allquantoren weg, indem man überall $\forall x$ ersetzt durch $\neg\ \exists x\ \neg$.
4.) Dann wird der quantorenfreie Teil von F in die sog. disjunktive Normalform (DNF) übergeführt, d.h. in die Form
   ((.....) $\wedge$ (.....) $\wedge$ (.....) .$\wedge$.) v ((.....) $\wedge$ (.....) $\wedge$ (.....) .$\wedge$.) v (.....
   Nun ist der quantorenfreie Teil eine oder-Verbindung von Klammern, die jeweils und-Verbindungen sind.
   Unser obiges Beispiel ist bereits in dieser Form!
5.) Schliesslich wird von innen her jeder $\exists$ -Quantor eliminiert, indem man im quantorenfreien Teil gewisse Ersetzungen und Streichungen vornimmt, die auch wieder besonders trickreich, aber möglich sind. Dabei werden gleichzeitig alle Teile der Formel durch „T“ (für „true“) oder „F“ (für „false“) ersetzt.
6.) Am Schluss kann man aus den und- und oder-Verbindungen der Zeichen „T“ und „F“ den Wahrheitswert der ganzen Formel F ablesen.

Ich hoffe, Ihnen mit diesen vagen Erläuterungen eine wenigstens ungefähr Vorstellung gegeben zu haben von der Möglichkeit, eine beliebige Formel aus ThdO automatisch zu entscheiden. Bei Problemen, die sich in der ThdO formulieren lassen, können wir also in

der Tat, wie Leibniz das vorschlug, zurücklehnen und durch blosses Rechnen entscheiden, wer recht hat!

Eine weitere entscheidbare Theorie ist die sog. **Presburger-Arithmetik (PA)**. Hierin sind alle mathematischen Sätze zusammengefasst, die einzig von den natürlichen Zahlen, der Operation"+" und der Ordnungsrelation „<" handeln. Den Beweis der Entscheidbarkeit der PA finden Sie in Presburgers Originalarbeit:

M. Presburger: „Über die Vollständigkeit eines gewissen Systems der Arithmetik der ganzen Zahlen, in welchem die Addition als einzige Operation hervortritt"
Compte-Rendus du 1. Congrès des Mathématiciens des pays Slav., Warsaw, 1929, pp. 92-101.

Nun hatte ich weiter oben angedeutet, dass sich in der 60er-Jahren des letztem Jahrhunderts die Bemühungen der Mathematiker der Komplexität, also dem zeitlichen (und räumlichen!) Aufwand von Algorithmen, zuwandten. Was war geschehen?

Das Jahr 1968 war nicht nur auf der politischen Bühne ein Jahr der Revolution. Auch unter den Mathematikern machte sich eine Unzufriedenheit breit über das damals Erreichte. Man war unzufrieden über die klassischen wissenschaftlichen Paradigmen, das Paradigma etwa, ein Problem für definitiv gelöst zu halten, wenn schon eine Lösung bzw. ein Lösungsalgorithmus existiert. Fragen nach der Optimalität von Lösungen im Hinblick auf die Anzahl benötigter Operationen machten sich jetzt breit. Wie viele Operationen sind zur Lösung mindestens nötig? Wie viel Speicherplatz (auf einer Turing-Maschine) wird zur Lösung mindestens benötigt? Durch solche Fragestellungen entstand die sog. Komplexitätstheorie, die Theorie, die lange bestehende Algorithmen neu in Frage stellt, ihren Aufwand untersucht und neue, bessere, schnellere Algorithmen für längst bekannte Probleme sucht. Einer der Pioniere dieser Theorie war (und ist) Volker Strassen, gewissermassen der „Rudi Dutschke der Algorithmik". Seine Arbeit aus dem Jahre 1969 „Gaussian elimination is not optimal" gab den Anstoss zu vielen weiteren Sätzen dieser Art. (Es ging bei dieser Arbeit darum zu zeigen, dass die „Gausselimination", ein Algorithmus zur Bearbeitung von Matrizen, nicht optimal im Hinblick auf die Anzahl Operationen ist. Strassen konnte dasselbe Problem mit weniger Operationen(reelle Multiplikationen!) lösen und setzte damit die Frage in Umlauf, was denn die Zeitkomplexität, als die minimale Anzahl Operationen, der Gausselimination sei.) Strassen war sich damals der Bedeutung seiner Forschungen so wenig bewusst, dass er vom seinem Fachkollegen Cook regelrecht zur Publikation gedrängt werden musste.

Die Einfachheit und Natürlichkeit vieler älterer Algorithmen haben deren Optimalität lange als selbstverständlich erscheinen lassen. Daher kam es in den 60er-Jahren einer (mathematischen) Revolution gleich, die Frage nach der Optimalität überhaupt zu stellen.

Auch bei unseren beiden oben erwähnten entscheidbaren Theorien stellte sich bald die Frage nach ihrer Komplexität. Die in diesem Zusammenhang wichtigsten Resultate sind:

ThdO: Anzahl Operationen= $n^{O(n)}$

(Das im Exponenten verwendete sog Landau-Symbol „O" bedeutet, dass der Exponent $\leq C \cdot n$ ist für eine gewisse Konstante *C*.) Dabei ist n die Länge der Inputformel. Formeln der Länge $n = 10$ ziehen also 10^{10} Berechnungsschritte nach sich.

Anzahl Speicherzellen $= n \cdot lb(n)$

PA:
Derek C. Oppen von der University of Toronto analysierte 1973 einen auf D. C. Cooper (1972) zurückgehenden Algorithmus zur Entscheidung der PA. Die Zeit- und Raumschranken, die er dabei fand, sind allerdings obere Schranken, d.h. sie geben die maximale Anzahl Operationen und die maximale Anzahl Speicherzellen wieder. Im Jahre 1974 lieferten Michael J. Fischer und Michael O. Rabin vom MIT untere Schranken, mithin die eigentliche Komplexität, die minmale Anzahl Operationen bzw. Speicherzellen. Nach diesem Werk ist die PA „super-exponentiell“:

Es gibt Formeln der Länge n, deren Entscheidung mind. $2^{2^{c \cdot n}}$ Schritte und auch Speicherzellen benötigt.

Es ist klar, dass die Frage nach der Komplexität eines Algorithmus von enormer Bedeutung sein kann. Ein Algorithmus nützt uns nichts, wenn sein Aufwand so enorm ist, dass eine automatische Lösung Jahre dauert. Denken wir nur an das Problem der „Dreifärbbarkeit von Graphen“. Es gibt 3 mögliche Farben pro Knoten. Da es n Knoten sind, sind im Wesentlichen 3^n verschiedene Färbungen zu prüfen. Wenn unser Algorithmus nun einfach alle möglichen Färbungen prüft, dann haben wir ein Komplexitätsproblem! Angenommen nämlich, unser Computer könnte 1 Färbung pro Nanosekunde prüfen; dann würde er 3^n Nanosekunden arbeiten. Bei einem Graphen mit $n = 100$ Knoten wären das bereits 3^{100} Nanosekunden, was ungefähr $1.6 \cdot 10^{31}$ Jahre sind.