

# Was ist Informatik-Didaktik?

## Gedanken über die Fachkenntnisse des Informatiklehrers

J. Nievergelt  
Informatik, ETH, CH-8092 Zürich

**Zusammenfassung.** Die steigende Nachfrage nach Informatikunterricht verschiedener Art (an öffentlichen und Berufsschulen, innerbetriebliche Aus- und Weiterbildung etc) wirft die Frage der geeigneten Ausbildung der Lehrkräfte auf. Der Artikel beschreibt den Kern eines für Mathematik- und Informatik-Studenten konzipierten Ausbildungsgangs für zukünftige Informatik-Lehrkräfte. Algorithmen und Datenstrukturen als Einstieg und die historische Entwicklung fundamentaler Themen der Informatik stehen im Zentrum.

**Schlüsselwörter:** Lehrerausbildung, Fachdidaktik, Algorithmen, Datenstrukturen, Programmieren im Kleinen, historische Entwicklung, Grundthemen der Informatik.

**Summary.** The growing demand for computer science education in a variety of schools (public, private, continuing education, etc) raises the issue of how to train future teachers of computer science. This paper presents the core of a teacher training program designed for students of mathematics and computer science. It focuses on algorithms and data structures as an introduction, and on the historical development of fundamental ideas in computer science.

**Key words:** Teacher education, curriculum, algorithms, data structures, programming-in-the-small, historical development, fundamental ideas in computer science.

**Computing Reviews Classification:** A.1, E.1, F.1, K.2, K.3

### Inhalt

- 1 Gedanken eines Praktikers zum Thema "Didaktik"
- 2 Kernthemen eines Faches, Kenntnisstand des Lehrers
- 3 Algorithmen und Datenstrukturen als Einstieg in die Informatik
- 4 Rechnen mit geometrischen Objekten: Ein Beispiel zur Computergeometrie
- 5 Klassische Themen verständlich dargestellt
- 6 Einige klassische Beispiele
  - 6.1 Kann eine Maschine denken?
  - 6.2 Berechenbarkeit
  - 6.3 Selbstreproduzierende Maschinen
  - 6.4 Computer als Spieler
- 7 Was ist Informatik-Didaktik?

## 1. Gedanken eines Praktikers zum Thema "Didaktik"

"Didaktik: Die Kunst oder Wissenschaft des Unterrichtens."

Vor vier Jahren fiel mir unerwartet die Aufgabe zu, eine Vorlesung über "Informatik-Didaktik" für Mathematik- und Informatikstudenten in den oberen Semestern zu halten. Alles, was mit Lehren zu tun hat, hat mich immer fasziniert, und drei Jahrzehnte Erfahrung als Lehrer geben einem Zuversicht, dass man etwas vom eigenen Beruf versteht. Es sind aber doch zwei recht verschiedene Tätigkeiten kompetent auszuüben und darüber zu dozieren. Der Volksmund kennt allerlei Sp

diesen Unterschied hervorzuheben. Da gibt es z. B. den Tausendfüssler, der erklären wollte, in welcher Reihenfolge er seine Beine in Bewegung setzt und dabei bestürzt feststellen musste, dass er nicht mehr gehen konnte. Ganz bissig drückt es der amerikanische Spruch aus: "Those who can, do; those who can't, teach; those who can't teach, teach teachers". Im Weltbild dieses Spruches musste ich befürchten, von der zweiten Liga in die dritte abzustiegen.

Als technischer Wissenschaftler muss man noch eine zweite Hemmung überwinden, bevor man über ein so subjektives Thema wie "Die Kunst oder Wissenschaft des Unterrichtens" spricht. Wir vertreten gerne objektive, beweisbare Aussagen und sind stolz darauf, allfällige Zweifler mit unumstösslichen Argumenten zur "richtigen" Einsicht zu zwingen. Aber mit welchen zwingenden Argumenten überzeugen wir einen eigenwilligen Studenten davon, dass ein Beispiel wirklich vorbildlich ist, wenn er behauptet, er könne es nicht verstehen? Anders als dem Politiker oder Sozialwissenschaftler genügt dem Techniker eine Abstimmung oder statistisch signifikante Meinungsumfrage oft nicht; er sucht einen logischen Beweis für die Korrektheit seines Standpunkts. Da er diesen kristallklaren Beweis natürlich nicht findet, leidet sein Selbstvertrauen; der einfachste Ausweg ist dann oft, sich in die Welt der Fakten und Formeln zurückzuziehen, wo es richtige und falsche Ansichten gibt, nicht einfach "andere". Aber in Fragen der Ausbildung gibt es nur wenige richtige oder falsche Ansichten, dafür sehr viele "andere".

Die "Kunst des Unterrichtens" gehört also nicht zu den Themen, die sich mit unumstösslicher Objektivität, strenger Logik oder quantitativen Aussagen behandeln lassen. Aber wenn ein klares Bedürfnis ruft, nimmt man die Herausforderung an und löst die Aufgabe nach bestem Wissen und Gewissen, wenn auch unvollständig. In den Worten des berühmten Ökonomen John Maynard Keynes: "It is better to be roughly right than precisely wrong".

In diesem Sinne möchte ich die vorliegenden Ausführungen verstanden wissen. Das Bedürfnis nach einer Ausbildung, oder zumindest Wegweisung, für zukünftige Informatiklehrer ist wohl unbestritten. Mit Ausnahme der Hochschulstufe wird heute noch ein Grossteil der Informatikausbildung in öffentlichen Schulen und der Industrie durch Autodidakten erteilt. Das ist an sich nicht schlecht. Der Autodidakt hat seine Kenntnisse "im Schweisse seines Angesichts" erarbeitet - was er weiss, das beherrscht er. Oft ist er "der grosse Experte" für einen engen Themenkreis, z.B. ein bestimmtes System oder eine Programmiersprache. Aber ihm fehlen die Übersicht und die Perspektive. Er kann Einzelaspekte wirklichkeitsnah darstellen, aber er kann das Fachgebiet als ganzes nicht angemessen vertreten. Als Folge dieses unausgewogenen Kenntnisstands des Lehrers sieht der Schüler ein Zerrbild der Informatik: Einige Details von vorübergehender Bedeutung erscheinen trügerisch wichtig, während grundlegende Ideen unerwähnt und unerkannt bleiben. Diese Überlegungen führen zu folgender Festlegung des Hauptziels einer Lehrveranstaltung Informatik-Didaktik, das wir in Abschnitt 5 wieder aufnehmen werden:

**Der Informatiklehrer soll imstande sein, die grundlegenden Ideen der Informatik intelligent und auch für Laien verständlich zu besprechen.**

Nebst diesem Hauptziel treten natürlich weitere Anforderungen auf, z.B. bezüglich Grundfertigkeiten und Methodenwissen. Abschnitt 7 greift solche Nebenpunkte ganz am Schluss noch kurz auf. Der Schwerpunkt dieses Artikels soll aber beim Thema *grundlegende Ideen der Informatik* liegen.

## **2. Kernthemen eines Faches, Kenntnisstand des Lehrers**

Für etablierte Disziplinen, z. B. Mathematik und Physik, sind sowohl das Schulpensum wie auch das (wesentlich grössere) Kernwissen des Lehrers recht klar definiert, und beide Pflichtteile **stützen sich auf anerkannt klassische Themen**. So gehört die über zwei Jahrtausende alte euklidische Geometrie zum Mathematikpensum jedes Gymnasiums; vom Geometrielehrer erwarten wir ausserdem, dass er intelligent und verständlich über nicht-euklidische Geometrien sprechen kann, falls sich Gelegenheit dazu ergibt. Ähnlich gehört der Satz von der Erhaltung der Energie sicher ins Physikpensum jedes Gymnasiums, und vom Physiklehrer erwarten wir, dass er auch Fragen zum Zweiten Hauptsatz der Thermodynamik lehrreich beantworten kann.

Wie anders ist doch die heutige Lage im Informatikunterricht! Von der Vielfalt der in verschiedenen Schulen behandelten Themen kann sich jeder überzeugen, der mit Informatiklehrern spricht. Ein Beispiel aus der Übersicht [NF 90] 'Informatik und Computernutzung im schweizerischen Bildungswesen: Bestandsaufnahme

1989' möge dies belegen. Von über zwei Dutzend aufgeführten Themen erreichen nur zwei eine Verbreitung in mindestens 70% der befragten Mittelschulen, nämlich 'Textverarbeitung' und 'Programmiersprachen'. Einige andere Themen, z.B. 'Geschichte / Entwicklung der Informatik', 'Bedeutung der Informatik' und 'Auswirkungen von Computeranwendungen', sind in etwa 50% der befragten Mittelschulen ein Unterrichtsgegenstand.

Von den beiden Spitzenreitern ist 'Textverarbeitung' eine nützliche Fertigkeit (die der Schüler selbst erlernen kann), aber sicher kein klassisches Thema im Zentrum des Lehrstoffes. 'Programmiersprachen' kann auf Hochschulstufe als klassisches Thema vorgestellt werden; nämlich für Hörer, denen verschiedene Programmiersprachen schon geläufig sind, für die man grundsätzliche Gemeinsamkeiten hervorheben kann. Auf Gymnasialstufe hingegen ist damit wohl einfach eine Einführung in die Programmiersprache der jeweiligen Schule gemeint: Pascal 45%, Basic 43%, Logo 6%, gemäss [NF 90].

**Fazit: An den Schulen besteht heute noch keine Einigkeit darüber, welche der anerkannt zentralen Themen zum Kern jeder Einführung in die Informatik gehören!**

Um Missverständnissen vorzubeugen sei gleich festgehalten: Dieser Artikel betrifft die Wissenschaft Informatik, "computer science", und nicht den Bereich der alltäglichen Informatikanwendungen, "computer literacy". 'Informatikunterricht' soll also eine Einführung in die Wissenschaft bezeichnen; nicht das Vermitteln von Anwenderfertigkeiten (z.B. Datenbankabfrage, Einsatz eines bestimmten Textverarbeitungssystems), und nicht den Einsatz des Computers im Unterricht (z.B. als Multimedia-Gerät). Diese Abgrenzung richtet sich nicht gegen die Computer-Fertigkeit - diese ist durchaus erwünscht. Sie soll nur unfruchtbare Diskussionen vermeiden, die aufgrund verschiedener Interpretationen der Informatik als Unterrichtsgegenstand entstehen.

Soweit zum Schulstoff, betrachten wir jetzt die Kenntnisse des Lehrers. Welches Kernwissen sollte ein Informatiklehrer meistern, damit er diesen Stoff auf verschiedenen Stufen dem Wissensstand der Schüler anpassen kann? Der Gymnasiallehrer in etablierten Disziplinen hat ein Hochschulstudium in seinem Fach absolviert, was zweierlei garantiert: Erstens gehen seine Kenntnisse in Bezug auf Breite und Tiefe weit über das hinaus, was er seinen Schülern erklärt; zweitens darf ein recht grosses, gemeinsames Kernwissen von jedem Fachlehrer vorausgesetzt werden. Der heutige Informatiklehrer hat hingegen nur in seltenen Fällen ein Informatikstudium absolviert, und keine der beiden Voraussetzungen darf a priori als gegeben angenommen werden: Hin und wieder übersteigen des Lehrers Kenntnisse über ein Informatikthema kaum die seiner Schüler, und er stösst oft auf Kernthemen der Informatik, die er bestenfalls dem Namen nach kennt.

Die fehlenden Informatikkenntnisse lassen sich nicht in den wenigen Stunden einer Vorlesung über Informatik-Didaktik nachholen. Der Missstand wird erst behoben sein, wenn ein abgeschlossenes Hochschulstudium in Informatik Vorbedingung für das Erteilen von Informatikunterricht sein wird - also in der nächsten Generation. Als Übergangslösung schlage ich zwei Informatik-Kernthemen vor, die sich jeder Informatiklehrer durch persönliche Weiterbildung erarbeiten und die er auch in jeder Einführung in die Informatik einsetzen kann:

- **Algorithmen und Datenstrukturen:** Ein wohldefiniertes, stofflich begrenztes, begrifflich klares, praktisch wichtiges Fachgebiet, das sich als Einstieg in die Informatik eignet.
- **Ausgewählte klassische Themen,** deren historische Entwicklung seit der Frühzeit der Informatik sich allgemeinverständlich erklären lässt. Diese Themen eignen sich als Ergänzungsstoff, den der Lehrer nach Bedarf in seinen Unterricht einfließen lassen kann.

Diese beiden Themenkreise stelle ich ins Zentrum meiner Lehrveranstaltung über Informatik-Didaktik, und der Rest dieses Artikels ist ihrer Darstellung gewidmet.

### 3. Algorithmen und Datenstrukturen als Einstieg in die Informatik

Welche Themen eignen sich zur Einführung in die Informatik, und welche nicht? Wir dürfen keinen eindeutig "besten" Inhalt eines Einführungskurses erwarten. Aber von den zahlreichen Themenkreisen, die heute als Einführung in die Informatik auftreten, sind einige besser geeignet als andere, und einige oft vorgeschlagene Themen haben wenig mit Informatik zu tun. Um die Spreu vom Weizen zu scheiden, müssen wir nach dem Kern der Informatik als Beruf fragen.

Informatik als Ingenieurdisziplin beschäftigt sich primär mit dem Entwurf Computer-gestützter Systeme und ihrer Realisierung in Hardware und Software. Entwurf und Implementierung von Systemen ist daher der Kern der Berufsausbildung eines Informatik-Ingenieurs. Aber das Erstellen von Systemen, das "Programmieren im Grossen", eignet sich nicht für eine Einführung in die Informatik. Das Ziel wäre zu hoch gesteckt, man könnte damit in kurzer Zeit keine abgeschlossene Lernerfahrung erreichen. Dass Informatik nicht analytisch vorgeht wie die Naturwissenschaften, sondern synthetisch wie die Ingenieurwissenschaften, lässt uns aber am Prinzip festhalten, dass **Entwurf und Implementierung im Zentrum jeder Einführung in die Informatik stehen sollten**. Was kann der Schüler im Rahmen einer Einführung selbst entwerfen und implementieren? Quantitativ nicht viel, aber unter geschickter Anleitung qualitativ sehr hochstehende, elegante kleine Programme!

Das "Programmieren im Kleinen" eignet sich als Einstieg in die Informatik ganz besonders, wenn man Perfektion bis ins Detail anstrebt. Hier stehen algorithmische Aspekte im Vordergrund: Entwurf eines Algorithmus, Korrektheitsbeweis, zu verwendende Datenstrukturen, Analyse des Zeit- und Speicherbedarfs. Die Programme sind kurz - oft weniger als eine Seite lang - können aber nur über einen mathematischen Gedankengang verstanden werden. Dadurch knüpft das Programmieren im Kleinen auf natürliche Weise an die Mathematik an, was im Mittelschulunterricht eine gegenseitige Befruchtung ermöglicht.

Algorithmen und Datenstrukturen sind als Kern des Informatikunterrichts durch viele ausgezeichnete Lehrbücher allgemein zugänglich. Die enzyklopädischen Pionierleistungen von Knuth [Kn 68ff] oder umfassende moderne Darstellungen wie [OW 90] eignen sich vorwiegend für die Weiterbildung des Lehrers; [NH 93] bietet eine Auswahl anschaulicher Beispiele für den Einführungsunterricht. Da Algorithmen und Datenstrukturen umfassend in Lehrbüchern dargestellt sind und die Studenten in meiner Lehrveranstaltung über Informatik-Didaktik eine Grundvorlesung über Algorithmen und Datenstrukturen bereits hinter sich haben, genügt es, ihnen bekannte Themen kurz und bündig nochmals aufzugreifen und dabei das Schwergewicht auf die didaktisch geschickte Darstellung zu legen.

Als Beispiel für die Behandlung von Algorithmen und Datenstrukturen möchte ich kurz einen noch zu wenig bekannten Zweig der Algorithmik erwähnen, der sich meiner Ansicht nach hervorragend für Informatik- und Mathematik-Unterricht auf Gymnasialstufe eignet - die algorithmische Geometrie. Diese stützt sich weitgehend auf intuitiv leicht zugängliche Grundbegriffe von Geometrie, Algorithmik, Datenstrukturen und Programmierung - eine für die Grundausbildung in Informatik ideale Kombination von mathematisch klaren, praktisch wichtigen Themen. Man kann den Standpunkt vertreten, dass sich die Geometrie von allen Themenkreisen der Mittelschulmathematik am besten eignet, den fruchtbaren Einsatz des Computers zu illustrieren. Folgenden Gründe sprechen für die algorithmische Geometrie als Bereicherung des Mathematik- und Informatik-Unterrichts:

- Aufgaben und Lösungen können klar, einfach und anschaulich dargestellt werden.
- Der Schwierigkeitsgrad der angestrebten Lösungen reicht oft von "trivial" bis "unmöglich", der Lehrer kann anspruchsvolle Probleme für jede Stufe auswählen.
- Geometrische Algorithmen eignen sich ausgezeichnet zur "Animation", d.h. zur graphischen Darstellung ihres Ablaufs, indem man laufend zeigt, wie die Daten schrittweise verarbeitet werden.
- Information wird aus Bildern schneller aufgenommen als z. B. aus Zahlentabellen. Also fallen Fehler in animierten geometrischen Algorithmen schneller auf als in anderen Programmen, was Verständnis und Testen wesentlich erleichtert.

Es ist in Fachkreisen wohlbekannt, dass geometrische Datenverarbeitung in manchen Punkten technisch anspruchsvoll ist. Einige Beispiele: Höhere Datenstrukturen treten auf, deren Programmierung Fachkenntnisse verlangt; es ist schwierig, einfach anmutende primitive Operationen in Gegenwart von Rundungsfehlern robust zu gestalten; die Behandlung entarteter Konfigurationen (geometrische Objekte nicht in allgemeiner Lage) ist schwierig. Kann ein Schüler, sicher noch kein Experte der Programmierkunst, sich da zurecht finden? Ja, sofern ihm eine geeignete Softwareumgebung zur Verfügung steht.

Die XYZ GeoBench [NS 91] ist ein Software-Baukasten mit vielen nützlichen Komponenten und Skelettprogrammen, die als erweiterbare Musterlösungen dienen. Man programmiert durch Zusammensetzen von Bausteinen, die der Schüler nicht selbst erstellen, aber einsetzen kann. Damit ist ein Weg aufgezeigt, wie neue Ideen und Methoden der Informatik eine dreitausend Jahre alte Wissenschaft verändern und neu beleben können.

## 4 Rechnen mit geometrischen Objekten: Ein Beispiel zur Computergeometrie

Das "Problem des nächsten Paares" (closest pair) veranschaulicht obige Überlegungen: Gegeben  $n$  Punkte in der Ebene, finde man die kleinste Entfernung zweier Punkte und ein zugehöriges Punktepaar. Die Aufgabe scheint so einfach zu sein, dass sie im traditionellen Geometrieunterricht nie auftaucht: Man misst alle  $n \cdot (n-1) / 2$  paarweisen Abstände und nimmt einen kleinsten unter diesen. Die unten vorgestellte Lösung ist jedoch gar nicht trivial - der kurz angedeutete Gedankengang, wie auch die späteren Beispiele in Abschnitt 6, mögen als zu schwierig für die Schulstufe erscheinen. Dazu zwei Kommentare:

- Die scheinbare Schwierigkeit der Lösung hat sicher auch damit zu tun, dass die hier vorgestellten Themen ungewohnt sind. Alles Neue ist zunächst einmal schwierig, bis die entsprechenden Begriffe nicht nur rational verstanden, sondern auch im Unterbewusstsein aufgenommen sind.
- Die hier gewählte Darstellung richtet sich nicht an Schüler, sondern an Informatik- und Mathematik-Studenten. Als zukünftige Lehrer werden sie bestimmen, ob und wie dieser Stoff auf Gymnasialstufe umgesetzt wird.

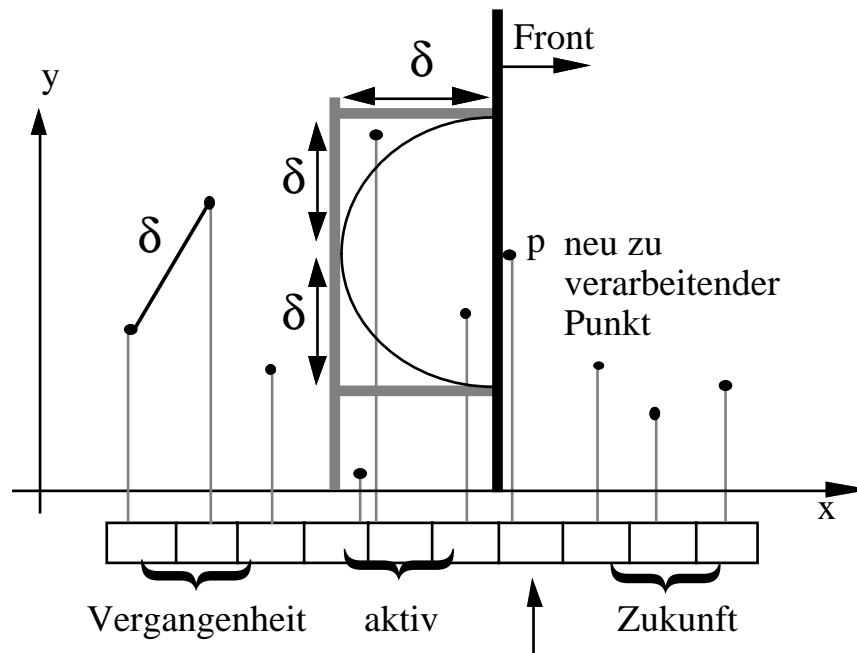
Das "Problem des nächsten Paares" wird mathematisch interessant sobald die Effizienz des Verfahrens zur Sprache kommt: Gibt es einen effizienteren Algorithmus als den erwähnten trivialen  $O(n^2)$ -Algorithmus? Zur Algorithmik gehören Begriffsbildungen, Fragen und Ergebnisse, die der Lehrer nicht unbedingt aufbringen, jedoch sicher kennen muss, wenn er dieses einfache Problem der algorithmischen Geometrie in abgekürzter Form kompetent erklären will. Erstens muss ein "Rechenmodell" festgelegt werden: Dies sei eine sequentielle "random-access-Maschine"; eine theoretische Konstruktion, die heutige Rechner so gut nachbildet, dass man sie jetzt nicht erklären muss. Es sei nur hervorgehoben, dass wir die Abstandsmessung zweier Punkte als Elementaroperation dieser Maschine betrachten, deren Ausführung eine Zeiteinheit beansprucht. Zweitens schimmert die hinter den Kulissen versteckte Asymptotik durch, die Idee, dass wir benötigte Zeit oder geleistete Arbeit nicht in Sekunden oder anderen genauen Einheiten messen, sondern nur ganz grob in Abhängigkeit von der Grösse  $n$  der Eingabedaten. Drittens taucht die Frage nach unteren Schranken auf, welchen Zeitaufwand man vernünftigerweise anvisieren sollte. Da ist die Idee naheliegend, dass jeder korrekte Algorithmus sicher alle  $n$  Punkte betrachten muss, also einen Zeitaufwand verlangt, der mindestens linear in  $n$  wächst. Wenn das Problem des Sortierens genügend bekannt ist, kann man hier auch als Ergebnis der Komplexitätstheorie erwähnen, dass  $\Omega(n \log n)$  eine untere Schranke sowohl für das Sortieren wie auch für das Problem des nächsten Paares ist. Es besteht also die Hoffnung auf einen  $O(n \log n)$ -Algorithmus, und es ist angebracht, den praktischen Unterschied zwischen  $n \log_2 n$  und hervorzuheben. Die folgende Tabelle illustriert diese Wachstumsraten anhand der Beispiele  $n = 1024 = 2^{10}$  und  $n = 1'048'576 = 2^{20}$ . Besonders eindrücklich wird der Unterschied, wenn wir diese Zahlen in Laufzeiten umsetzen. Je nach Algorithmus ist der Zeitaufwand für das Durchlaufen der innersten Schleife eines solchen Programms sehr verschieden, aber oft liegt er in der Grössenordnung einer Millisekunde (ms). Unter dieser Annahme ergeben sich die in den beiden rechten Spalten angegebenen Laufzeiten:

$n$	$n \log_2 n$	$n^2$	$n \log_2 n$ ms	$n^2$ ms
$2^{10} \sim 10^3$	$10 \times 2^{10} \sim 10^4$	$2^{20} \sim 10^6$	$\sim 10$ s	$\sim 20'000$ s $\sim 6$ h
$2^{20} \sim 10^6$	$20 \times 2^{20} \sim 2 \times 10^7$	$2^{40} \sim 10^{12}$	$\sim 20'000$ s $\sim 6$ h	$\sim 10^9$ s $\sim 30$ Jahre

Der Unterschied zwischen einem  $n \log n$  Algorithmus und einem  $n^2$  Algorithmus läuft bei  $n = 1000$  also darauf hinaus, ob interaktive Verarbeitung möglich ist oder nicht, und bei  $n = 1'000'000$ , ob die Rechnung praktikabel ist oder nicht. Wie weit und tief der Lehrer auf solche allgemeinen Betrachtungen über Algorithmik eingeht, sei der konkreten Einzelsituation überlassen. Notwendig sind sie nicht, um die folgende überraschende Einsicht zu vermitteln. Wir müssen nicht alle  $n \cdot (n-1) / 2$  paarweisen Abstände berechnen, sondern ..

**Satz:** Das Problem des nächsten Paares wird mit maximal  $4n$  paarweisen Abstandsberechnungen gelöst.

Der einfachste Weg zu dieser Einsicht führt über eine wichtige Klasse geometrischer Algorithmen, die wir am Beispiel des "nächsten Paares" anhand der folgenden Abbildung zu erklären versuchen. Die "sweep-Algorithmen" überstreichen die in der Ebene eingebetteten Daten von "links nach rechts" mittels einer vertikalen Front. Die Invariante dieser inkrementellen Algorithmen sagt aus, dass man die Lösung "hinter der Front", also für die bereits untersuchten Daten, kennt. In unserem Fall heisst das, dass wir ein nächstgelegenes Paar unter den Punkten links der Front kennen, und dieses habe Abstand  $\delta$ .



Ein Schritt des Algorithmus verarbeitet einen neuen Punkt  $p$ , nämlich den nächstgelegenen rechts der Front. Ergibt dieser neue Punkt  $p$  einen kleineren Abstand als den bisher kleinsten,  $\delta$ ? Um dies zu beantworten muss  $p$  keineswegs mit allen Punkten links der Front verglichen werden, sondern nur mit denen in einem Halbkreis mit dem Radius  $\delta$ . Mit Hilfe der Invariante "links der Front sind alle Punkte mindestens  $\delta$  voneinander entfernt" sieht man nun sofort, dass nur ganz wenige Punkte im Halbkreis um  $p$  mit Radius  $\delta$  liegen können.

Da keine dynamischen Datenstrukturen bekannt sind, die genau diese Halbkreisabfrage mit der erwünschten Effizienz beantworten, wird sie durch die gezeichnete rechtecksförmige Bereichsabfrage implementiert. Es zeigt sich, dass jeder der  $n$  Punkte explizite mit höchstens 4 anderen verglichen werden muss - aber welchen 4? Für die Verwaltung der Daten werden Datenstrukturen (eine einfache Warteschlange, im Bild als Array angedeutet, und eine Tabelle, die der vertikalen Front entspricht) derart eingesetzt, dass wir immer schnell auf die gerade benötigten Punkte zugreifen können. Der Aufwand  $O(n \log n)$  für diese Buchführung, das "Gewusst wo", übersteigt den linearen Aufwand für die geometrischen Operationen (Abstandsberechnungen), was für viele Algorithmen typisch ist. [NS 88] enthält eine ausführlicher Darstellung dieses "closest-pair-Algorithmus", samt Programm.

## 5. Klassische Themen verständlich dargestellt

Neben Algorithmen und Datenstrukturen als Kern jeder Einführung in die Informatik habe ich eine zweite Anforderung an die Kenntnisse des Lehrers genannt: Eine Sammlung klassischer Themen, die sich in allgemein verständlicher Form erklären lassen und die der Lehrer nach Bedarf in seinen Unterricht einfließen lassen kann. Wo und wie finden wir grundlegende Ideen der Informatik, und welche Gedankengänge und Darstellungsarten lassen sich allgemeinverständlich erklären? Die Antwort suchen wir in der Geschichte: grundlegende Ideen tauchen oft im frühen Entwicklungsstadium einer Disziplin auf und werden dann sukzessive verbessert, bis sie ihre "definitive" Form angenommen haben. So wie die Ideen entstanden und verfeinert wurden, so kann der Laie sie oft auch am klarsten verstehen. Unser Ziel ist also, klassische Themen der Informatik durch ihre historische Entwicklung allgemein zugänglich zu machen.

Die traditionelle Unterscheidung zwischen Bildung einerseits, und der Aneignung von Fertigkeiten andererseits, mag als Wegweiser dienen, der die grundlegenden Ideen einer Disziplin aufzeigt. Diese Unterscheidung ist im wesentlichen eine Frage des Zeithorizonts: Bildung befasst sich mit Gedanken von bleibendem Wert, Fertigkeiten sind oft von vorübergehendem Nutzen. Der Unterschied lässt sich an zwei Arten verdeutlichen, wie ein Lehrer Programmieren unterrichten kann. Er kann einerseits eine abstrakte Einsicht betonen, deren Gültigkeit alle heutigen Programmiersprachen überleben wird: Nämlich die, dass ein Programm eine statische Spezifikation eines zeitlichen Ablaufes ist, in einer beliebig vorgegebenen Notation

geschrieben. Im anderen Extrem kann der Lehrer auf der Syntax einer konkreten Sprache herumreiten, was dem Schüler möglicherweise sofort zu einer Anstellung verhilft - auf die Gefahr hin, dass er zeitlebens als 'BASIC Programmierer' (oder COBOL- oder APL- oder ..) eingestuft wird, statt als *Programmierer*, und sich in einer neuen Umgebung nicht entfalten kann.

Schulen müssen natürlich auch kurzlebige Fertigkeiten vermitteln, aber der Schwerpunkt liegt doch bei der Bildung von langfristigem Wert. Wie können wir Ideen von langfristigem Wert von den Modetrends unterscheiden? Am zuverlässigsten durch einen Rückblick in die Geschichte. Im grossen Ganzen haben Ideen, die unsere Vorgänger beeindruckten, eine grössere Chance, unsere Nachfolger zu beeindrucken, als unsere letzten Neuigkeiten. Klassische Themen sind alte Ideen, die relevant und frisch geblieben sind. Auch in der jungen Disziplin Informatik haben wir immerhin ein halbes Jahrhundert Erfahrung mit automatischem Rechnen, und wir schöpfen aus dem Gedankengut einiger Jahrhunderte über prinzipielle Fragen des Rechnens (z.B. Algorithmik, Logik), dessen Relevanz heute grösser ist als zur Zeit der Pioniere. Suchen wir also klassische Themen der Informatik, indem wir die Wurzeln unseres Faches durchleuchten; und fragen wir danach, wie wir dieses Gedankengut allgemein zugänglich darstellen können - keine leichte Aufgabe.

Klassische Themen vereinigen oft zwei scheinbar widersprüchliche Eigenschaften: Sie sind sowohl einfach wie auch schwierig, elementar wie auch fortgeschritten. Der Experte sieht klassische Themen als einfach und fundamental, weil sie sich nur auf grundlegende Prinzipien stützen - sie liegen nahe der Basis einer begrifflichen Hierarchie, nicht bei der Spitze. Der Anfänger aber empfindet klassische Themen als schwierig und fortgeschritten, weil ihre Interpretation Reife und Abstraktionsvermögen verlangt.

Wie reift des Schülers Abstraktionsvermögen, wie werden Abstraktionen zu operationellen Richtlinien? Die Erfahrung zeigt klar, dass wir abstrakte Begriffe um so besser verstehen und nutzen, je mehr konkrete Beispiele dieser Abstraktion wir gründlich kennen. Erst dann wird ein abstrakter Begriff zu einer "mächtigen Idee" (Seymour Paperts Bezeichnung), wenn er viele konkrete Erfahrungen treffend zusammenfasst. Also muss eine Einführung in klassische Themen auf ausgezeichneten Beispielen aufbauen, die gleichzeitig zwei Zwecke erfüllen: Sie sind an sich von Interesse (elegant, wichtig, etc.) und veranschaulichen den abstrakten Begriff, den wir erklären wollen. Hat ein Schüler treffende Beispiele gut verstanden, dann bleibt die intuitive Einsicht in die dahinterliegende Abstraktion noch erhalten, nachdem er längst Definitionen, Sätze und Beweise vergessen hat.

Also liegt der Schlüssel zu einer Einführung in dieses schwierige Thema in einer Sammlung hervorragender Beispiele. Solche lassen sich nicht aus dem Busch trommeln - man muss sie jahrelang sammeln und immer wieder sorgfältig überprüfen und verbessern. Als Beispiel diene Dewdney's 'Turing Omnibus - 61 Excursions in Computer Science' [De 89] ist eine Sammlung von Artikeln aus der Zeitschrift *Scientific American*, die auch einem Laien Einblick in breite Teile der Informatik erlaubt.

## 6. Einige klassische Beispiele

Die folgenden Themen habe ich oft als Beispiele für grundlegende Ideen in einer Einführung in die Informatik eingesetzt. Sie sollen zeigen, dass klassische Themen auch allgemeinverständlich auf lehrreiche Art besprochen werden können und sich sogar für Programmierübungen eignen. Aus Platzgründen sind sie nicht so detailliert ausgeführt, wie es für Schüler notwendig wäre, sondern nur soweit, dass der Leser, der diese Grundideen bereits kennt, der Meta-Diskussion folgen kann: Welche Rolle können diese Beispiele im Unterricht spielen? Die folgenden einschränkende Bemerkungen sollen Missverständnisse verhüten:

- Unsere Beispiele sind nicht als Empfehlung gemeint, dass gerade diese grundlegende Ideen zu unterrichten. Ein Lehrer braucht genügend Spielraum, um die Wahl des Unterrichtstoffes den Gegebenheiten seiner Umgebung anzupassen. Ich empfehle aber jedem Informatiklehrer, die Gedankengänge aus den folgenden Beispielen zu kennen, nebst anderen.
- Der Einwand, etliche grundlegende Ideen der Informatik liessen sich nicht auf einfach verständliche Art darstellen, ist nicht tragisch - dann soll sich der Lehrer einen Vorrat von anderen wichtigen Ideen aufbauen, die er allgemeinverständlich erklären kann. Davon sollte es genug geben, denn wer gar keine grundlegenden Ideen seines Faches verständlich machen kann, hat sie selbst nicht verstanden.
- Ein Potpourri von unzusammenhängenden Grundideen, auch wenn jede treffend vorgestellt wird, empfehle ich keineswegs als Einführung in die Informatik. Ein Kurs braucht eine thematische Einheit und ist somit immer "unausgewogen". Aber der *Kenntnisstand des Lehrers* soll ausgewogen sein, damit er bei

Gelegenheit fruchtbare Querverbindungen aufbauen kann.

## 6.1 Kann eine Maschine denken?

Turings Diskussion [Tu 50] dieser emotional belasteten, grundlegenden Frage vom philosophischen, ethischen und technischen Standpunkt aus ist sicherlich ein klassisches Thema der Informatik. Jeder Informatiklehrer sollte bereit sein, diese Frage klärend zu besprechen, denn es ist oft die erste substantielle Frage, die ein Laie über Computer stellen kann.

Die Informatik bietet den 'Turing Test' an als objektives Kriterium zur Beantwortung der verwandten Frage, wie weit ein Computer intelligente menschliche Aktivität nachahmen kann. Wenn der Lehrer unser Prinzip akzeptiert, dass grundlegende Ideen durch konkrete Beispiele erläutert werden sollen, dann zeigt er seinen Schülern einige interessante Fälle des Turing Tests. Das Programm ELIZA [We 66] ist ein klassisches Spielzeug, das menschliche Konversation nachahmen soll und dies kurzfristig auch überraschend erfolgreich tut - bis man ihr auf die Schliche kommt. ELIZA transformiert nämlich einfach die vom Benutzer eingetippten Sätze nach oberflächlichen Regeln, um "ohne etwas vom Inhalt zu verstehen" eine plausible Erwiderung zu konstruieren. Die Regel: "Ich bin x" -> "Seit wann bist du x?", z. B., funktioniert für viele Werte von x recht gut, und ruft für x = "50 Jahre alt" oft Erheiterung hervor. Ein einfaches ELIZA-Programm zu schreiben ist eine ausgezeichnete Programmierübung, die mit einer Programmiersprache für die Verarbeitung von Zeichenketten nicht allzu aufwendig ist. Der Schüler erlebt dabei durch eigene Arbeit den Zusammenhang zwischen der Komplexität des Mechanismus und der "Qualität" der erzeugten Antworten. Er versteht, dass ELIZA nur ganz minimale Kenntnisse der Satzstruktur einer Sprache haben muss, um amüsante Phrasen zu basteln, und erhält dadurch Einsicht in die schwierige Frage, was es bedeuten könnte, "ein Computer versteht etwas über einen kleinen, wohlbegrenzten Wissensbereich". Diese persönliche Programmier-Erfahrung ist mehr wert als jede allgemeine Diskussion sogenannter "Expertensysteme", bei denen der Aussenstehende oft nicht beurteilen kann, ob ein Programm wirklich Expertenwissen vorweist oder nicht.

## 6.2 Berechenbarkeit

Im Gegensatz zu dem Begriff "denkende Maschine", der zu philosophischen, subjektiven und oft vagen Diskussionen verleiten kann, hat sich der Begriff "berechenbar" oder "entscheidbar" im Laufe dieses Jahrhunderts zu einem Eckpfeiler der theoretischen Informatik entwickelt, der mit mathematischer Schärfe definiert ist. Die völlig unterschiedliche Entwicklung der beiden zunächst intuitiven Begriffe "denken" einerseits, und "rechnen oder entscheiden" andererseits, ist für Fachleute heute so offensichtlich, dass man leicht vergisst, dass dieser Unterschied noch im vergangenen Jahrhundert keineswegs klar war. Als Beispiel möge der Titel 'The laws of thought' des 1854 erschienenen Buches des englischen Mathematikers George Boole dienen, der einer der Begründer der symbolischen Logik war. Heute bezeichnen wir den Inhalt als Boole'sche Algebra oder Aussagenlogik und sehen darin keine tiefe Beziehung zum menschlichen Denken. Die Metamorphose des anfänglich intuitiven Begriffs "berechenbar" zu einem mathematischen Begriff ist tatsächlich eine der grossen intellektuellen Leistungen unseres Jahrhunderts.

Wenn auch in der heutigen Informatik der Stellenwert grundlegender Begriffe wie "denken" und "rechnen oder entscheiden" völlig anders ist als noch vor einem Jahrhundert, dann ist es für den Lehrer doch eine Herausforderung, die Entwicklung dieser Begriffe zu erklären - sogar in einer Einführung in die Informatik, wenn nur wenige Stunden dafür zur Verfügung stehen. Kann dies gelingen? Berechenbarkeit ist ja bekanntlich ein technisch schwieriges Fach, das beträchtliche mathematische Reife voraussetzt. Ich glaube aber, wichtige Einsichten in den Themenkreis Berechenbarkeit können auch ohne grosse mathematische Vorkenntnisse gewonnen werden, wenn die Ideen anhand guter Beispiele eingeführt werden. Verschieden Rechenmodelle wie Turing-Maschinen, Produktionen oder künstlich einfache Programmiersprachen eignen sich für eine informelle Einführung in das Thema Berechenbarkeit. Minskys Buch [Mi 67] ist eine didaktisch hervorragende Einführung mit vielen inhaltsreichen Beispielen. Ein elegantes Beispiel, das ich oft in einer Einführung in die Informatik zu erklären versuche, hat C. Strachey in einem Leserbrief im Computer Journal (Vol 7, No 4, p. 313, 1965) formuliert:

To the Editor,  
The Computer Journal

### An impossible program

Sir,

A well-known piece of folklore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it runs. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (... in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.

Suppose  $T[R]$  is a Boolean function taking a routine (or program)  $R$  with no formal or free variables as its argument and that for all  $R$ ,  $T[R] = \text{True}$  if  $R$  terminates if run and that  $T[R] = \text{False}$  if  $R$  does not terminate. Consider the routine  $P$  defined as follows

```
rec routine P
  §L: if T[P] go to L
  Return §
```

If  $T[P] = \text{True}$  the routine  $P$  will loop, and it will only terminate if  $T[P] = \text{False}$ . In each case  $T[P]$  has exactly the wrong value, and this contradiction shows that the function  $T$  cannot exist.

Yours faithfully, C. Strachey.

## 6.3 Selbstreproduzierende Maschinen

Ähnlich wie bei "denkenden Maschinen" hat die Frage, ob Maschinen sich reproduzieren können, in den frühen Jahren der Computer zahlreiche Spekulationen hervorgerufen, z.B. die einer Flotte schwimmender Roboter, die Rohstoffe aus dem Wasser schöpfen, diese zu nützlichen Produkten verarbeiten, am Hafen abliefern und zusätzlich zu all diesen technischen Wundern sich auch noch selbst reproduzieren - so dass es genügt, einen zu bauen (oder vielleicht zwei?) um jede gewünschte Menge zu erzeugen.

John von Neumann erkannte eine logische Schlüsselfrage im Wunschdenken über selbstreproduzierende Maschinen. Alle Erfahrung mit Maschinen, z. B. Werkzeugmaschinen, die andere Maschinen produzieren, zeigt, dass der Produzent komplexer ist als das Produkt. Der Produzent braucht ja erstens eine vollständige Beschreibung des Produkts und zweitens allerlei Baugeräte wie Sensoren, Greifer etc. Die logische Komplexität einer Beschreibung ist der Komplexität des Produktes gleichzusetzen. Also ist es unumgänglich, so lautet der Pseudobeweis, dass komplexe Maschinen notwendig sind, um einfache zu bauen; und dass deshalb echte Selbstreproduktion (Kinder identisch wie die Eltern) unmöglich ist.

Diese Argumentation enthält aber einen Fehlschluss, den von Neumann mittels einer lehrreichen Konstruktion eines fiktiven Roboters aufzeigte. Dieser fährt in einem Warenhaus herum, das Kopien derjenigen Teile enthält, aus denen der Roboter gebaut ist - z.B. Räder, Batterien etc. Der Kern der Konstruktion ist ein universeller Konstruktionsroboter (hier ist eine Analogie zur universellen Turing-Maschine angebracht!) der jedes beliebige Produkt bauen kann, für das er eine Beschreibung (Konstruktionsanleitung) erhält. Neben diesem anschaulichen Gedankenexperiment untersuchte von Neumann in einem 400-seitigen Buch [vN 66] das Problem der Selbstreproduktion auch im Modell der zellularen Automaten.

Das Thema Selbstreproduktion eignet sich hervorragend für den Einsatz von Computern im Unterricht. Ein Programm für das Spiel 'Life' ist lehrreich, sowohl beim Schreiben als auch beim Experimentieren. Ebenso die knifflige Aufgabe, ein Programm zu schreiben, das sich exakt selbst ausdrückt. In vielen Programmiersprachen ist dies recht schwierig, aber in anderen ist es trivial, wie das Beispiel '10 LIST' in BASIC zeigt. Dieser Schwierigkeitsunterschied wirft einige verblüffende Fragen auf: Ist Selbstreproduktion ein triviales oder ein überraschendes Phänomen? Ist eine Reihe aufrecht stehender Dominosteine, die durch umwerfen des ersten einer nach dem andern umfallen, ein Beispiel von Selbstreproduktion?

## 6.4 Computer als Spieler

Zurück zur Frage 'Kann eine Maschine denken?'. Sie führte uns zum Turing-Test, der eine unscharfe allgemeine Frage in viele konkrete Einzelfragen auflöst, von denen jede eine objektive Antwort zulässt. Eine konkrete Version des Turing Tests befasst sich damit, wie gut ein Computer menschliches Verhalten bei einer intellektuellen Tätigkeit nachahmen kann. Strategische Spiele wie Schach, die nach allgemeiner Ansicht

Intelligenz verlangen, liefern besonders attraktive Turing-Tests. Ein solches Spiel ist eine 'Mikrowelt' geeigneter Komplexität: Einerseits genügend einfach, dass die Regeln vollständig formalisiert sind und man plausible Heuristiken relativ leicht implementieren kann; andererseits genügend komplex, dass das resultierende Verhalten kaum voraussehbar ist, sondern sich erst durch Experimente mit oft überraschendem Ausgang zeigt. Spielprogramme eignen sich für Turing-Tests auch deswegen gut, weil es für viele Spiele recht scharfe Massstäbe gibt, um Spielqualität zu messen.

Shannon [Sh 50] und Turing [Tu 53] beschrieben einfache Rezepte, um Schachprogramme zu schreiben. Diese stützten sich auf einige Grundkonzepte der von Neumannschen Spieltheorie [vN 28] und auf grossen Optimismus hinsichtlich der Existenz nützlicher 'statischer Bewertungsfunktionen'. Diese einfachen Formeln sollen messen, wie gut oder schlecht eine Stellung ist, und sie stützen sich auf grobe heuristische Attribute wie Material, Mobilität, oder Bauernstruktur, welche einen kleinen Teil des Schachwissens erfassen sollen. Computer spielten Schach jahrzehntelang auf Anfängerstufe, aber stetiger Fortschritt seit den Siebzigerjahren hat die Spielstärke des z. Z. besten Schachcomputers Deep Thought [Hsu 90] bis auf Grossmeisterniveau angehoben. Deep Thought und andere Schachcomputer haben bewiesen, dass Shannons Ansatz erfolgreich ist, dass Kenntnis eines verschwindend kleinen Bruchteils menschlicher Schachtheorie genügt, um Meisterstärke zu erreichen; vorausgesetzt, man untersucht eine Million Stellungen pro Sekunde - wahrlich eine eindruckliche Bestätigung der Macht des Rechnens. [Ma 87], [Ko 90], [MS 90] enthalten Übersichten zur Entwicklung von Schachcomputern.

Wird ein Spielprogramm als Beispiel eines Turing-Tests untersucht, dann ist der Zusammenhang zwischen Implementierung und Spielverhalten die interessante Informatikfrage. Wiederum suchen wir nach einem Weg, wie wir im Rahmen einer wenig fortgeschrittenen Lehrveranstaltung mit verschiedenen Implementierungen eines Spielprogramms experimentieren können. Wir verfahren ähnlich wie bei der in Abschnitt 3 beschriebenen algorithmischen Geometrie, indem wir eine hochentwickelte Programmierumgebung zur Verfügung stellen, die sehr vieles enthält, was der Spielprogrammierer sonst mühsam selbst erarbeiten müsste, z.B. eine Mensch-Maschine Schnittstelle für Brettspiele, Spielbäume und heuristische Suchverfahren (Minimax-Evaluation eines Spielbaumes mittels alpha-beta-pruning, Transpositionstabellen etc.). Das Smart Game Board [Ki 90], [KCN 90] ist ein spielunabhängiges Programm skelett, das es Studenten ermöglicht, im Rahmen von Übungen interessante Spielprogramme zu schreiben.

## 7. Was ist Informatik-Didaktik?

Ein Blick in die Literatur zeigt, dass die Antwort für die Informatik heute anders ausfällt als für etablierte Disziplinen wie die Mathematik.

- Die erst im Entstehen begriffene Literatur zur Informatik-Didaktik (z.B. [Hu 88], [Ba 90], [CFG 90], [Mo 91]) tendiert dazu, die verschiedensten Themen aus dem breiten Spektrum der Informatik einzuführen, von technischen Aspekten wie Hardware und Software bis zu rechtlichen Fragen und gesellschaftlichen Konsequenzen. Allgemeinverständliche, thematisch breite Einführungen in die Informatik wie [Re 91] gehören sicher zur Allgemeinbildung, und der Informatiklehrer sollte sie kennen. Für den Unterricht aber birgt thematische Breite die Gefahr einer oberflächlichen Behandlung in sich, die nicht tiefer schürft als die öffentlichen Medien dies tun. Wir trauen dem verantwortungsbewussten (zukünftigen) Lehrer zu, dass er sich diese Bildung im Alltag selbst aneignet und suchen für den Unterricht schärfer fokussierte Themen.
- Repräsentative Bücher der Mathematik-Didaktik beschränken sich auf ein engeres Ziel, wie [Ri 85], [Scho 90], [St 90] aus der Reihe 'Lehrbücher und Monographien zur Didaktik der Mathematik' belegen. Zwei Merkmale fallen schon beim ersten Blättern auf: Die vielfältigen, konkret ausgearbeiteten Beispiele, und die dominante Rolle der geschichtlichen Entwicklung des Faches.
- Bisher nur selten wurde der Brückenschlag versucht, den in der Mathematik-Didaktik erprobten methodischen Ansatz auf die Informatik anzuwenden (z.B. [NFR 74], [CI 87], [Gr 88], [Schw 91]).

Dieser Artikel ist eine persönliche Antwort auf die im Titel aufgeworfene Frage. Diese Antwort, mit einer allgemeinen und einer fachspezifischen Komponente, sei nun zusammengefasst:

- Allgemein: Die Informatik-Didaktik sollte die Tradition der Mathematik-Didaktik übernehmen und die historische Entwicklung des Faches anhand treffend gewählter Beispiele illustrieren, die man auch ohne Kenntnis der allgemeinen Theorie verstehen kann. Also **nicht** versuchen, soweit wie möglich den heutigen Stand eines Faches inhaltlich vollständig und in aller Strenge zu charakterisieren. Letzteres mag ein Ziel der

Fachausbildung sein, aber nicht der Didaktik des Faches.

- Die fachspezifische Antwort gliedert sich in zwei Themen:
  - Algorithmen und Datenstrukturen ist der empfohlene Einstieg in die Informatik. Dieser praktisch wichtige Themenkreis ist begrifflich klar und ermöglicht es dem Schüler auf jeder Stufe, elegante Lösungen kleiner Probleme **vollständig** zu verstehen und selbst **kreativ zu entwerfen und zu implementieren** - das Wesentliche an der Informatik.
  - Der (zukünftige) Lehrer sollte sich eine Sammlung ausgewählter klassischer Themen erarbeiten, deren historische Entwicklung seit der Frühzeit der Informatik sich allgemeinverständlich erklären lässt, und die er nach Bedarf in seinen Unterricht einfließen lassen kann.

Neben dieser Darstellung des Hauptziels meiner Lehrveranstaltung Informatik-Didaktik möchte ich nur kurz erwähnen, wie Studenten mit methodischen Fragen, z.B. der Vielfalt von Unterrichtsmethoden, konfrontiert werden.

- Jeder Student hält einen viertelstündigen Kurzvortrag über ein selbst gewähltes Thema, wobei er/sie auf Videoband aufgenommen wird. Beim anschließenden Abspielen des Bandes diskutiert man den Vortrag vom fachlichen und didaktischen Standpunkt aus wie auch die Präsentationstechnik. Die Studenten beteiligen sich dabei lebhaft, was meine Aufgabe wesentlich erleichtert, weil wichtige kritische Bemerkungen von den Kollegen aufgebracht werden, nicht immer nur vom Lehrer.
- Unter den Übungsaufgaben gibt es zwei grössere, bei denen Studenten in kleinen Gruppen Unterrichtshilfen konstruieren, nämlich:
  - Einen mechanischen "Computer" erfinden und bauen, d. h. ein Modell aus Papier, Holz, Draht, Seil, Gummibändern konstruieren, angetrieben durch rollende oder fallende Kugeln - was immer die gewünschte logische Funktion physisch realisieren kann. Addierwerke, Schaltkreise, Umwandlung zwischen Binär- und Dezimalnotation sind beliebte Ziele.
  - Ein Unterrichtsprogramm schreiben, das als Vorführ- oder Übungsprogramm eingesetzt wird, und die Fähigkeit des Computers zur graphischer Darstellung möglichst virtuos zeigt. Erklären, wo und wie dieses Unterrichtsprogramm eingesetzt werden kann.

Es hat viel Arbeit gekostet und mir viel Freude bereitet, die angegebenen Richtlinien und Zielvorstellungen im Detail auszuarbeiten und im Unterricht zu erproben. Ich hoffe, damit einen gangbaren Weg aufgezeigt zu haben, wie sich ein Lehrer mit dem Problem 'Informatik Didaktik' auseinandersetzen kann. Wer unser Konzept probiert, möge mir bitte seine Erfahrungen mitteilen - ich wünsche viel Glück und Freude.

**Bemerkung.** Ich danke dem Programmkomitee der GI Tagung "Computer und Schule" in Oldenburg, Oktober 1991 für die Einladung zu einem Vortrag, aus dem dieser Artikel entsprang; P. Rechenberg, W. Hartmann und dem mir unbekanntem Gutachter für ihre tiefeschürfenden Kommentare und Verbesserungen. Zur Sprachform sei hinzugefügt, dass Wörter wie "Lehrer", "Student" (an Stelle der Neuschöpfungen "LehrerIn", "StudentIn") ohne Bezug auf Geschlecht verwendet werden - für diese "geschlechtslose" Interpretation von Berufsbezeichnungen gibt es im Gebrauch vieler Sprachen nützliche Vorbilder.

## Literatur

- [Ba 90] R. Baumann: Didaktik der Informatik, Klett Schulbuchverlag, Stuttgart 1990.
- [Cl 87] V. Claus: Was sollte von Informatik in der Schule vermittelt werden?, Hauptvortrag auf der Tagung über Informatik-Grundbildung und Beruf, 2-8, Informatik-Fachberichte 129, 1987.
- [CFG 90] G. Cyranek, H. Forneck, H. Goorhuis (Hrsg.): Beiträge zur Didaktik der Informatik, Verlag Diesterweg, Sauerländer, 1990.
- [De 89] A.K. Dewdney: The Turing Omnibus, Computer Science Press, 1989.
- [Gr 88] K. D. Graf: Computer in der Schule 2. Beispiele für Mathematikunterricht und Informatikunterricht, Teubner, Stuttgart 1988.
- [Hsu 90] F. Hsu, T. Anantharaman, M. Campbell, A. Nowatzky: A grandmaster chess machine, Sci. American, Vol 263, No 4, 18-24, Oct 1990.

- [Hu 88] R. Hugelshofer (Hrsg.): Informatik: Anwendungen - Algorithmen - Computer - Gesellschaft, Verlag Diesterweg, Sauerländer, 1988.
- [KCN 90] A. Kierulf, K.H. Chen, J. Nievergelt: Smart Game Board and Go Explorer: A study in software and knowledge engineering. *Comm. ACM*, Vol 33, No2, 152-166, Feb 1990.
- [Ki 90] A. Kierulf: Smart Game Board: A Workbench for Game-Playing Programs, with Go and Othello as Case Studies, Diss. ETH Zurich, 1990
- [Kn 68ff] D. E. Knuth: The Art of Computer Programming, Vol 1, 2, 3, Addison-Wesley 1968, ..
- [Ko 90] D. Kopec: Advances in man-machine play, 9-32 in [MS 90].
- [Ma 87] T. A. Marsland: Computer chess methods, 159-171 in *Encyclopedia of AI* (ed. Shapiro), Wiley 1987.
- [Mo 91] E. Modrow: Zur Didaktik des Informatik-Unterrichts, F. Dümmler, Bonn 1991.
- [MS 90] T. A. Marsland, J. Schaeffer (eds.): *Computers, Chess, and Cognition*, Springer 1990.
- [Mi 67] M. Minsky: *Computation - Finite and infinite machines*, Prentice-Hall 1967.
- [NF 90] R. Niederer, K. Frey: *Informatik und Computernutzung im schweizerischen Bildungswesen: Bestandesaufnahme 1989*, ETH Zurich, 1990.
- [NFR 74] J. Nievergelt, J.C. Farrar and E.M. Reingold: *Computer Approaches to Mathematical Problems*. Prentice-Hall 1974.
- [NH93] J. Nievergelt, K. Hinrichs: *Algorithms and data structures, with applications to graphics and geometry*, Prentice-Hall 1993.
- [NS 88] J. Nievergelt, P. Schorn: Rechnen mit geometrischen Objekten: Beispiele zur Computergeometrie, 77-95 in [Gr 88].
- [NS 91] J. Nievergelt, P. Schorn, M. De Lorenzi, C. Ammann, A. Brüngger: XYZ: A project in experimental geometric computation, 171-186 in H. Bieri and H. Noltemeier (eds.): *Computational Geometry: Methods, Algorithms and Applications. Proc. CG'91, International Workshop on Computational Geometry*, Bern, March 1991, Springer LNCS, 1991.
- [OW 90] T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen*, BI Wissenschaftsverlag, 1990.
- [Re 91] P. Rechenberg: *Was ist Informatik?*, Hanser Verlag 1991.
- [Ri 85] W. Riemer: *Neue Ideen zur Stochastik*, BI Wissenschaftsverlag, 1990.
- [Scho 90] E. Scholz (Hrsg.): *Geschichte der Algebra*, BI Wissenschaftsverlag, 1990.
- [Schw 91] A. Schwill: *Didaktik der Informatik*, Vorlesungsskript, Univ. Oldenburg, 1991.
- [Sh 50] C.E. Shannon: Programming a computer for playing chess. *Philosophical Magazine* 41, 314, 256-275, 1950.
- [St 90] H. Struve: *Grundlagen einer Geometrie-Didaktik*, BI Wissenschaftsverlag, 1990.
- [Tu 50] A. M. Turing: Computing machinery and intelligence, *Mind*, 9, 433-460, 1950.
- [Tu 53] A.M. Turing: Digital computers applied to games. In 'Faster than Thought: A Symposium on Digital Computing Machines', (B. V. Bowden, ed.), Ch. 25, 286-310, Pitman, London, 1953.
- [vN 28] J. von Neumann: Zur Theorie der Gesellschaftsspiele, *Math. Annalen* 100 (1928) 295-320
- [vN 66] J. von Neumann: *Theory of Self-Reproducing Automata*, (A. W. Burks, ed.), Univ. of Illinois Press 1966.
- [We 66] J. Weizenbaum: ELIZA - A computer program for the study of natural language communication between man and machine, *Comm. ACM* 9, 36-45, 1966.