Algorithmisch lösbare und unlösbare Probleme

Empfehlungen für eine Unterrichtseinheit zum Wahlthema

“Theoretische Informatik”

in der Jahrgangsstufe 13

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

Inhalt

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

1.Vorwort

Die Unterrichtseinheit wurde von mir ursprünglich schon 1987 für das Thema “Algorithmen und Datenstrukturen” konzipiert, und so angelegt, daß die Erarbeitung theoretischer Aussagen der Informatik in der unmittelbaren Arbeit am Computer beim selbständigen Algorithmieren und Programmieren erfolgen soll. Ich bin der Meinung, daß sie inhaltlich trotz eines Wechsels vom prozeduralen zum objektorientierten Programmierparadigma auch für den neuen Rahmenplan als Einstieg in das Wahlthema *Theoretische Informatik* im 2. Halbjahr der Jahrgangsstufe 13 sehr gut geeignet, wobei dann aber durchaus auch fertige Programme genutzt und folglich die Anteile am Problemlösen durch **selbständiges** Algorithmieren und Programmieren gekürzt werden könnten (und sollten).

Die Unterrichtseinheit ist zunächst nur auf Fragen der Komplexität von Algorithmen und der damit verbundenen Leistungsfähigkeit und Leistungsgrenzen des Computers eingeschränkt und ist im Leistungskurs durch die Themenbereiche *Präzisierung des Algorithmusbegriffs* und *Formale Sprachen und abstrakte Automaten* zu ergänzen.

2. Fachwissenschaftliche Orientierung

Die Wissenschaft Informatik ruht auf mehreren Säulen, die in einer langen Geschichte insbesondere aus der Mathematik (theoretische Basis) und der Technik (materiell-energetische Basis) gewachsen sind.

Aus der Mathematik stammt auch der Begriff des Algorithmus, der vermutlich auf den Namen des im 9. Jahrhundert lebenden arabischen Gelehrten ***Al Chwarizmi*** zurückgeführt wird, der in einer seiner Arbeiten über mathematische und astronomische Probleme das aus Indien stammende Dezimalsystem und das Rechnen in diesem System erläuterte. Das Wesen eines Algorithmus zu ergründen, beschäftigte ganze Generationen von Mathematikern. Inspiriert durch die von den Arabern eingeführten algebraischen Methoden entwickelte D. Lullus (etwa 1253 - 1315) die großartige Idee des allgemeinen Verfahrens ("Ars magma") zum Auffinden aller Wahrheiten, die ihren ersten Höhepunkt bei G. W. Leibniz (1646-1716) fand. Er erkannte, daß bei allgemeinen Verfahren zwischen Entscheidungsverfahren (Ars iudicandi) und Erzeugungsverfahren (Ars invenienci) zu unterscheiden ist und nahm den heutigen Begriff der informationsverarbeitenden Maschine voraus, indem er darauf verwies, daß man ein allgemeines Verfahren auf einer Maschine realisieren können muß. Die von Leibniz vertretene Auffassung, daß alle algebraischen Probleme mit einem allgemeinen Verfahren

gelöst werden können, dominierte noch fast 300 Jahre, denn um diesen Irrtum aufzudecken war die Formalisierungs- und Interpretationstechnik der mathematischen Logik und Metamathematik erforderlich, die bis dato nicht existierte. Der Nachweis, daß es Probleme gibt, die nicht algorithmisch lösbar sind, erforderte zudem einen präzisierten Algorithmusbegriff, der erst Mitte der 30er Jahre dieses Jahrhunderts durch die Arbeiten von K. Gödel, A. M. Turing und A. Church bereitgestellt wurde.
Die Entwicklung und Untersuchung von Algorithmen zur Lösung vielfältiger Probleme gehört heute zu den wichtigsten Aufgaben der Informatik. Trotz ungeahnter Möglichkeiten der Hardware-Entwicklung stößt man durch den notwendigen hohen (manchmal nicht mehr zu bewältigenden) Zeit- und/oder Speicherplatzbedarf häufig auf Barrieren. Theoretische Untersuchungen in den entsprechenden Teilgebieten der Komplexitätstheorie haben hier entscheidend geholfen, haben die Lösung vieler praktischer Problem erst möglich gemacht und werden auch in Zukunft noch einen wesentlichen Einfluß haben. Zur Komplexitätstheorie gehört auch die Theorie der NP-Vollständigkeit, die erst in den 70er Jahren diese Jahrhunderts entstand und inzwischen eine stürmische Entwicklung genommen hat. Sie beschäftigt sich u. a. mit dem sogenannten "P-NP-Problem", das wohl gegenwärtig das am stärksten diskutierte offene Problem der Informatik ist.

2. Ziele der Unterrichtsreihe

Gemäß Wahlthema *Theoretische Informatik* im Rahmenplan Informatik für die gymnasiale Oberstufe (Jahrgangstufe 13/2) sollen die Schüler

- grundsätzliche Möglichkeiten und Grenzen des Problemlösens mit Informatiksystemen kennen
- informatische Probleme hinsichtlich ihrer algorithmischen Lösbarkeit in folgende Klassen einordnen:
  1. Klasse der Probleme, die in Polynomialzeit lösbar sind,
  2. Klasse der algorithmisch lösbaren Probleme, für die beim gegenwärtigen Stand der Theoretischen Informatik kein polynomialer Lösungsalgorithmus bekannt ist
  3. Klasse der nachweislich nicht algorithmisch lösbaren Probleme
- die Notwendigkeit der Präzisierung des anschaulichen Algorithmusbegriffs erkennen, wenn die prinzipielle algorithmische Unlösbarkeit von Problemen nachgewiesen werden soll.
- das Halteproblem als ein algorithmisch unlösbares Problem kennen
- die algorithmische Unlösbarkeit des Halteproblems beweisen

3. Verlaufsübersicht

Die Unterrichtseinheit besteht aus zwei Sequenzen, die hintereinander zu realisieren sind. Die Probleme innerhalb der einzelnen Sequenzen bauen logisch aufeinander auf und sollten in der angegebenen Reihenfolge bearbeitet werden. Der angegebene Stundenumfang hängt sehr stark von den Fähigkeiten der Schüler sowie von den Erfahrungen der Lehrer ab und stellt lediglich eine Orientierung dar. Die Zeitaussagen beziehen sich zudem auf die ursprüngliche Intension, daß die Erarbeitung theoretischer Aussagen der Informatik in der unmittelbaren Arbeit am Computer beim selbständigen Algorithmieren und Programmieren erfolgen soll. Bei Vorgabe fertige Programme oder Teilprogramme ändern sich die Zeitaussagen natürlich (siehe Vorwort)

Schematische Verlaufsübersicht

Sequenz 1:	**Sequenz 2:**
Effiziente Algorithmen	Algorithmisch schwere und unlösbare Probleme
1.-6. Stunde:	**1.-2. Stunde**
- Ripplesort	- Rundreiseproblem
- Bubblesort	
	3.-6. Stunde
7.-8. Stunde	- Rucksackproblem
- Mergesort	
	7.-10. Stunde
9.-12. Stunde	- Halteproblem
- Tanzpaarung (exponentieller A.)	- Zusammenfassung
13.-14. Stunde	
- Tanzpaarung (polynomialer A.)	
- Zusammenfassung	

4. Allgemeine methodische Hinweise für das Problemlösen durch selbständiges Programmieren

Beim Problemlösen durch selbständiges Programmieren orientiert sich das methodische Vorgehen am

Software-Lebenszyklus und ist als Einheit von Lehrer- und Schülertätigkeit in folgenden Phasen zu gestalten:

Orientierung auf das Problem →

Problembegegnung
- Formulieren des Problems
- Motivieren zum Lösen des Problems durch eigenständiges Programmieren **b**
- Orientieren auf das angestrebte Unterrichtsziel **e**

Problemanalyse **g**
- Klären des Sachverhalts an konkreten Beispielen
- Benennen der Ein- und Ausgabedaten **l**
- Zergliedern des Problems in Teilprobleme
- Suchen nach analogen Problemen, für die Lösungsstrategien bekannt sind **e**
- Festlegen von Bezeichnungen für Variable und Teilprobleme **i**

t

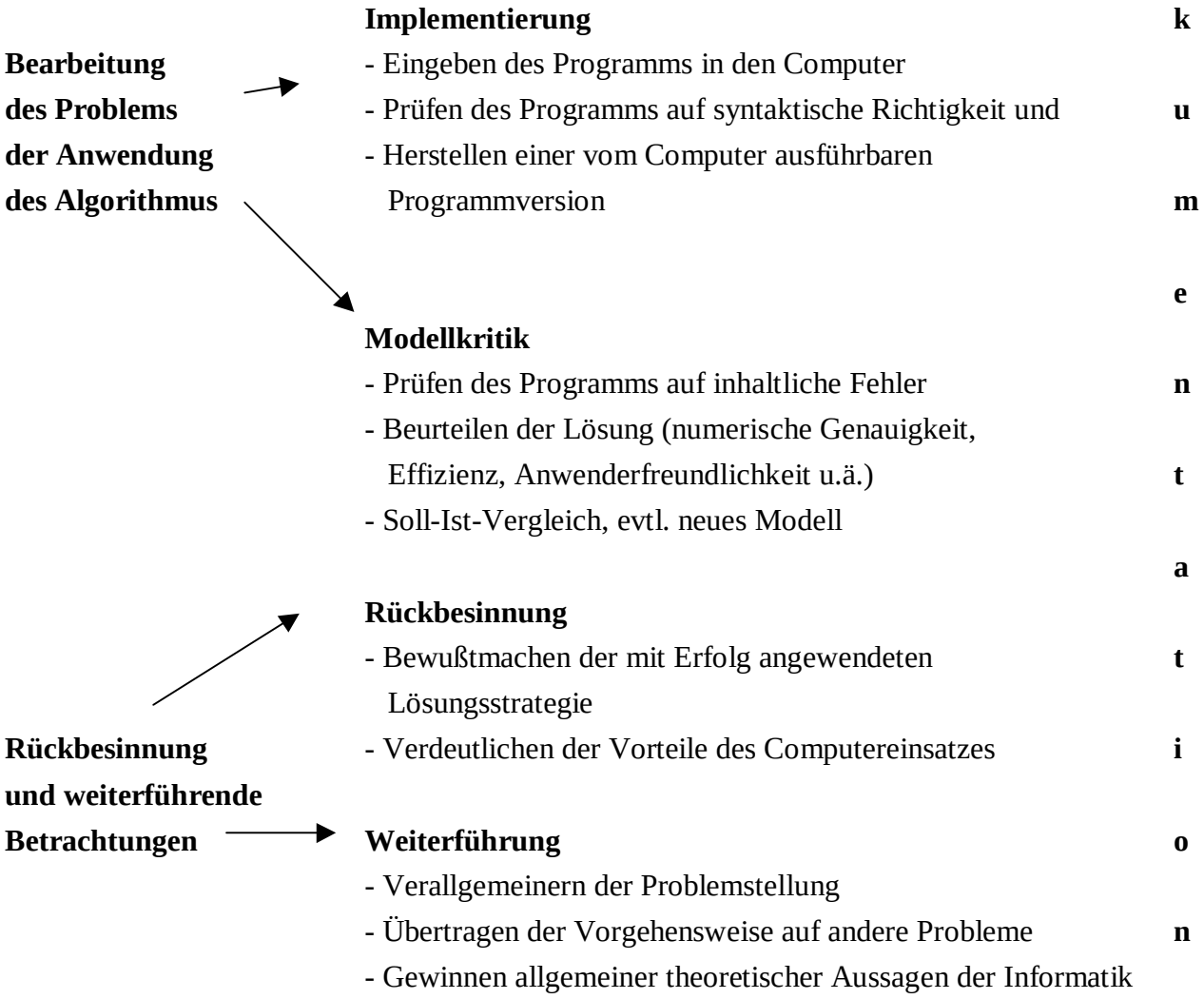
Bearbeitung des Problems des Findens eines Algorithmus

Modellbildung **e**
- Erarbeiten eines Daten- und Aktivitätenmodells
- Festlegen der Datentypen und -strukturen **n**
- Darstellen der Zusammenhänge zwischen Ein- und Ausgabedaten
- Rückgreifen auf bekannte Verfahren und Methoden **d**

Algorithmierung **e**
- Überführen des Modells in einen Algorithmus
- Darstellung des Algorithmus in einer genormten Form

D

Programmierung
- Codieren des Algorithmenentwurfs in die Programmiersprache **o**



Eine diesem Schema angepaßte methodische Vorgehensweise verlangt ein ausgewogenes Verhältnis zwischen unmittelbarer Arbeit am Computer und "computerferner" geistiger Vor- und Nacharbeit.
Der Schwerpunkt der Lehrertätigkeit liegt im gezielten Steuern wesentlicher Schülerhandlungen, um den gesamten Problemlöseprozeß ständig in Gang zu halten. Dabei hängen Umfang und Intensität der zu gebenden Impulse natürlich sehr stark von den konkreten Bedingungen in der Klasse und von der Komplexität des gestellten Problems ab, so daß insbesondere detaillierte, situations- und aufgabenspezifische Hinweise für die Schüler von Bedeutung sind.
Für die Organisation der Schülertätigkeiten ist es zunächst wichtig, jene Handlungen zu erkennen, deren Beherrschung für das Erreichen der Lernziele von grundlegender Bedeutung sind. Dazu gehören Handlungen, die auf

· das Präzisieren der Aufgabenstellung

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

· das Suchen eines Algorithmus
· das Anwenden eines Algorithmus
· das Bewerten der Arbeitsergebnisse

gerichtet sind.

Wesentlicher Bestandteil der Präzisierung der Aufgabenstellung ist das Festlegen der beteiligten Daten hinsichtlich Typ, Struktur und Bezeichnung, das Ausgliedern von Teilaufgaben und das Finden erster Lösungsideen. Insbesondere wird in diesem Arbeitsschritt das Ein- und Ausgabeverhalten des zu konstruierenden Algorithmus beschrieben.

Im Falle einer sehr allgemeinen Aufgabenstellung beinhaltet die Präzisierung zugleich auch eine Eingrenzung der Aufgabe auf einen angemessenen Umfang.

Folgende Handlungshinweise des Lehrers sind denkbare Orientierungshilfen für die Schüler:

- Lies die Aufgabe genau durch und gib das Wesentliche mit eigenen Worten wieder!
- Überlege welche Daten ein- bzw. auszugeben sind und welche Beziehungen zwischen den Daten bestehen!
- Überlege, welche Daten Variable sein sollten und welche konstant bleiben!
- Lege Bezeichnungen für die Variablen und Konstanten fest!
- Überlege, welcher Typ und welche Struktur für die einzelnen Daten geeignet ist!
- Zerlege die Aufgabe in überschaubare Teilaufgaben!
- Verschaffe Dir Klarheit über den Umfang der zu erfassenden und zu verarbeitenden Daten!

Der Lehrer sollte sich weitgehend zurücknehmen und das didaktische Prinzip der minimalen Hilfe beachten, d. h. darauf achten, daß das Aneignen neuer Lerninhalte nicht durch passive Rezeption geschieht, sondern durch aktive Beiträge der Schüler selbst.

Beim Suchen eines für die Lösung der Aufgabe adäquaten Algorithmus sollten die Lehrer den Schwerpunkt der zu organisierenden Schülerhandlungen auf das bewußte Benutzen solcher heuristischer Verfahren richten, wie

- Zurückführen auf eine Aufgabe mit bekanntem Lösungsweg
- Vorwärtsarbeiten vom Gegebenen zum Gesuchten
- Rückwärtsarbeiten vom Gesuchten zum Gegebenen
- Anfertigen von Skizzen und Tabellen
- Untersuchen von Spezialfällen, Fallunterscheidungen

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

Beim Entwerfen eines Algorithmus haben sich die Hauptmethoden der strukturierten Programmierung
- modulares Arbeiten und
- schrittweise Verfeinerung
besonders gut bewährt. Beide Methoden ergänzen sich gegenseitig und erleichtern die systematische Bearbeitung umfangreicher Aufgaben spürbar.
Der Grundgedanke des modularen Arbeitens besteht darin, das Problem in überschaubare, relativ abgeschlossene Teilaufgaben zu zerlegen und die Lösungen der Teilaufgaben zum Abschluß zur Gesamtlösung zusammenzusetzen. Programmtechnisch wird das modulare Arbeiten über Prozeduren oder Funktionen realisiert, deren zeitliche und logische Abfolge durch das Hauptprogramm gesteuert wird.
Bei der schrittweisen Verfeinerung wird das Problem ebenfalls in einzelne Teilprobleme zerlegt, die ihrerseits dann weiter in Teilprobleme aufgespalten werden. Dieser Prozeß wird so lange fortgesetzt, bis das Problem in hinreichend kleine Teilprobleme zerlegt ist, die direkt in Anweisungen der verwendeten Programmiersprache codiert werden können.
Bewährt haben sich in dieser Phase detaillierte, aufgabenspezifische Steuerimpulse seitens des Lehrers.

Die wichtigsten Schülerhandlungen beim Anwenden des gefundenen Algorithmus sind
- das Codieren des Algorithmus in die verwendete Programmiersprache
- das Implementieren de Programms auf dem Computer
- das Testen des Programms auf seine Korrektheit
Im Ergebnis dieser Tätigkeiten entsteht ein lauffähiges Programm, das die Eingabedaten in korrekte, der Problemstellung entsprechende Ausgabedaten überführt.
Beim Codieren des Algorithmus müssen sich die Steuerimpulse des Lehrers insbesondere darauf konzentrieren, daß die Schüler die beim Entwerfen des Algorithmus festgelegte Daten- und Ablaufstruktur beibehalten. Wenn das nicht der Fall ist, dann erstellen die Schüler quasi einen neuen Algorithmus. Wichtig ist auch, daß ein übersichtliches, gut lesbares und somit leichter korrigierbares Programm entsteht.
Achtung!! Bis zum Implementieren des Programms auf dem Computer sollten die Schüler computerfern arbeiten, nach Möglichkeit sogar in einem anderen Raum oder an speziellen Schreibarbeitsplätzen. **Vor allem Jungen sind geneigt, ihre Ideen direkt nach der Methode "Trial and Error" am Computer zu implementieren.**
Beim Implementieren des Programms auf einem Computer geht es darum, daß die Schüler den Programmtext in den Computer eingeben und ein lauffähiges Programm erhalten. Zu beachten ist dabei, daß etwaige Fehler, die bereits im Programmentwurf vorhanden sein können oder erst beim

Editieren entstehen, vom Schüler auf Grund einer entsprechenden Fehlermeldung des Computers erkannt werden. Das Korrigieren derartiger Fehler kann z. B. durch folgende Hinweise vom Lehrer gesteuert werden.

- Interpretiere die Fehlermeldung!
- Überlege, wie ein Fehler dieser Art entstehen kann!

Nach der Beseitigung der syntaktischen Fehler liegt ein lauffähiges Programm vor, das vom Schüler anschließend aber noch auf seine Korrektheit getestet werden muß. Erfahrungsgemäß ist die Bedeutung dieses Handlungsschritts bei den Schülern meist nicht hinreichend im Bewußtsein verankert, was vielleicht auch dadurch begünstigt wird, daß inhaltliche Fehler im Gegensatz zu syntaktischen Fehlern vom Computer nicht angezeigt werden.

Die einzige zur Zeit praktikable Methode zum inhaltlichen Überprüfen eines Programms besteht darin, durch geschickt ausgewählte Stichproben die Wahrscheinlichkeit für das Auftreten von Fehlern möglichst klein zu halten. Dabei muß aber klargestellt werden, daß das Prüfen eines Programms durch empirisches Testen höchstens das Vorhandensein von Fehlern aufzeigen kann, jedoch niemals ein Beweis für die Korrektheit ist. Denn: ***Eine allgemeine Aussage - und um eine solche handelt es sich hier - kann zwar durch ein einziges Gegenbeispiel widerlegt, aber nicht durch noch so viele bestätigende Beispiele bewiesen werden.***

Die Steuerimpulse des Lehrers sollten sich in dieser Phase auf die Unterstützung der Schüler bei der sorgfältigen Auswahl von Testdaten und, falls erforderlich, bei der Fehlersuche konzentrieren.

- Wähle die Testdaten so, daß jede Anweisung des Programms mindestens einmal ausgeführt wird und alle möglichen Wege im Programm zu durchlaufen sind!
- Beachte Spezialfälle (Division durch 0) und Fallunterscheidungen!
- Suche den Fehler gegebenenfalls in einem Trockentest!
- Nutze den Debugger, um Dir die Belegungen einzelner Variablen anzeigen zu lassen!
- Schalte den TRACE-Modus ein und kontrolliere den Zeilendurchlauf während der Programmabarbeitung!
- Protokolliere die Testergebnisse!

Wenn Fehler auftreten, deren Ursache nicht auf der Ebenen der Codierung des Algorithmus liegen, dann ist dem Schüler zu raten, zum Arbeitsschritt "Algorithmieren" zurückzukehren.

Liegt im Ergebnis dieser Handlungen ein korrektes Programm vor, sollte der Schüler abschließend überlegen, ob das zugrundeliegende Modelle (Modellkritik) oder das Programm selbst noch weiter verbessert werden kann. Die Steuerung entsprechender Schülerhandlungen kann darin bestehen, daß

der Lehrer die Schüler frühzeitig auf die Beachtung folgender Gütekriterien für Programme orientiert:
Ein **gutes Programm** ist

- **korrekt**, d. h., es tut genau das, was es soll; nicht weniger, aber auch nicht mehr.
- **effizient**, d. h. es läuft schnell ab und/oder es benötigt wenig Speicherplatz. Beide Forderungen sind i. a. nicht zugleich erfüllbar.
- **zuverlässig**, d. h. es vermag Fehler, die durch falsche Eingabedaten oder durch falsche Anwendungen verursacht werden, selbst zu verhindern.
- **wartungsfreundlich**, d. h. es ist leicht zu ändern und leicht zu korrigieren - und zwar nicht nur vom Programmautor selbst, sondern auch von anderen Nutzern.
- **benutzerfreundlich**, d. h. der Benutzer kann das Programm ohne Konsultation des Programmautors verwenden, und die Arbeit mit ihm ist einfach.

5. Spezielle fachwisenschaftliche Grundlagen und didaktisch-methodische Hinweise zu den Sequenzen

5. 1. Sequenz I - Effiziente Algorithmen

Fachwissenschaftliche Grundlagen

Der Begriff der Zeitkomplexität eines Algorithmus wurde 1960 von Rabin eingeführt. Sie ist eine arithmetische Funktion, die den Aufwand an Rechenzeit in Abhängigkeit vom Umfang des Problems (Anzahl der Eingabedaten, Ordnung der Matrix, Grad des Polynoms o.ä.) ausdrückt. Man unterscheidet dabei zwischen dem Aufwand im Mittel (average case) und dem Aufwand im schlechtesten Fall (worst case). Interessiert man sich für die Laufzeit eines Algorithmus zur Lösung eines Problems vom Umfang n im Mittel bzw. im schlechtesten Fall, dann betrachtet man die Laufzeit für sämtliche Beispiele des Problems vom Umfang n und bestimmt davon den Mittelwert bzw. das Maximum. Dabei wird nicht unterschieden, ob es sich um eine Anweisung oder einen Test handelt. Natürlich wäre es wünschenswert, das Verhalten eines Algorithmus vollständig zu kennen und somit die Funktion des Rechenzeitaufwandes in Abhängigkeit vom Umfang des Problems exakt aufschreiben zu können, aber in der Praxis kann diese Aufgabe nur mit großen Schwierigkeiten gelöst werden. Oft muß man sich damit begnügen, den Zeitaufwand größenordnungsmäßig abzuschätzen und nur das asymptotische Algorithmusverhalten zu kennen. Um solche Größenordnungen von Funktionen auszudrücken, hat sich die sogenannte *Groß-Oh*-Notation bewährt:

Ein Algorithmus mit einem Rechenzeitaufwand A(n) hat die Zeitkomplexität O(g(n)), sofern es eine

Konstante $c > 0$ gibt, so daß $A(n) \leq c \cdot g(n)$ für alle $n \in N$ gilt.

Die weitaus häufigsten und wichtigsten Funktionen g(n) sind n^k mit geeignetem ganzzahligen $k > 0$, log n und n log n sowie die Exponentialfunktionen 2^n, 3^n,... und n!.

Praktisch bedeutet das: Wenn die Abschätzung des Rechenzeitaufwandes A(n) z. B. ein Polynom $a_k n^k + a_{k-1} n^{k-1} + ... a_1 n + a_0$ mit $a_k \neq 0$ ist, dann hat der zugehörige Algorithmus die Zeitkomplexität $O(n^k)$. Für die Abschätzung des Rechenzeitaufwandes ist also lediglich von Interesse, daß die Gesamtzahl der benötigten Rechenoperationen ein Vielfaches von n^k nicht übersteigt. Auf konstante Faktoren kommt es dabei nicht an.

Kann die Zeitkomplexität eines Algorithmus nach oben mit einem geeigneten $k > 0$ durch n^k abgeschätzt werden, dann nennt man den Algorithmus polynomial. Ansonsten spricht man von exponentiellen Algorithmen.

Ein Problem wird polynomial genannt, wenn es einen polynomialen Algorithmus zur Lösung des Problems gibt. In der Praxis hat sich gezeigt, daß es für die meisten polynomialen Probleme Algorithmen gibt, deren Rechenzeitaufwand von der Größenordnung $O(n^3)$ oder kleiner ist.

Gegenwärtig ist man übereinstimmend der Meinung, daß höchstens polynomiale Algorithmen praktische Bedeutung haben. Algorithmen mit exponentiellem Rechenzeitaufwand sind für die praktische Anwendung nur bedingt geeignet, da die benötigte Rechenzeit schon bei relativ kleinen Problemumfängen extrem hoch ist, wovon man sich durch einen Blick auf den Werteverlauf einfacher Exponentialfunktionen sofort überzeugen kann:

n	10	100	1000
n^3	10^3	10^6	10^9
2^n	1024	$1{,}27 \cdot 10^{30}$	$1{,}05 \cdot 10^{301}$

Ziele der Sequenz I

Die Schüler

- kennen den Begriff der Zeitkomplexität als Gütemaß für einen Algorithmus.
- kennen den Sortieralgorithmus Ripplesort und können ihn programmieren,

- können die Zeitkomplexität von Ripplesort experimentell und rechnerisch bestimmen,
- verstehen den Sortieralgorithmus Mergesort und können ihn auf konkrete Zahlenbeispiele anwenden,
- kennen die Zeitkomplexität von Mergesort und können den Unterschied zwischen Algorithmen mit der Zeitkomplexität $O(n^2)$ und $O(n \cdot \log n)$ werten,
- erkennen, daß Algorithmen mit exponentiellem Rechenzeitaufwand für die praktische Anwendung nur bedingt geeignet sind, da die benötigte Rechenzeit schon bei relativ kleinen Problemumfängen extrem hoch ist, so daß das Lösen von Aufgaben mittels exponentieller Algorithmen selbst bei Verwendung modernster Superrechner nur theoretischer Natur bleibt.

Didaktische-methodische Empfehlungen

Für den Einstieg in die Unterrichtseinheit empfehle ich Sortieralgorithmen, deren große praktische Bedeutung den Schülern bekannt ist, denn mit sortierten Mengen kommen sie ständig in Berührung (Telefonbücher, Kontonummern, Duden, Datenbanken, Dateimanager u.v.a.m.).

In Gruppenarbeit sollten die Schüler zunächst den Algorithmus *Ripplesort* erarbeiten. Um das dabei erforderliche Vergleichen und Vertauschen zweier Zahlen stärker bewußt zu machen, empfehle ich die Vorgabe der zu sortierenden Zahlen (n = 5) in nichtunterscheidbaren, verschlossenen Behältern (Dosen, Überraschungseier o. a.). Die Schüler wenden den erarbeiteten Algorithmus selbständig auf weitere Zahlenfolgen an und erstellen anschließend ein zugehöriges PASCAL-Programm mit Zufallszahleneingabe und Stoppuhr. Mit diesem Programm führen sie dann eine Laufzeitanalyse durch und erkennen den funktionalen Zusammenhang zwischen der benötigten Rechenzeit und der Länge der Zahlenfolge. Als Hausaufgabe bietet sich an, daß die Schüler die Laufzeitanalyse - wenn möglich - am eigenen Computer noch einmal durchführen. Dazu ist das Programm als exe-File auf Diskette abzuspeichern. Die Schüler, die keinen Computer zu Hause haben, könnten die Aufgabe erhalten, für den Algorithmus *Ripplesort* die Anzahl der notwendigen Vergleiche zunächst für den konkreten Fall n=5 und dann für den schlechtesten Fall (worst case) einer n-elementigen Zahlenfolge zu bestimmen. Die Ergebnisse sollten in der nächsten Unterrichtsstunde vorgestellt und ausgewertet werden. Im Experiment auf verschiedenen Hardware-Plattformen wird deutlich, daß die Zeitkomplexität eine Eigenschaft des Algorithmus ist, die von der Hardware unabhängig ist. Erst danach sollte der Lehrer den Begriff der Zeitkomplexität definieren und die Zeitkomplexität für R*ipplesort* bestimmen lassen ($O(n^2)$).

Der Algorithmus *Bubblesort* muß in der Regel vom Lehrer zur Verfügung gestellt werden. Denkbar ist

das aber auch das der Algorithmus und seine Umsetzung in einem Pascal- Programm von einem Schüler erläutert wird. Alle Schüler führen mit diesem Sortierprogramm ebenfalls einen Laufzeittest durch und erkennen experimentell und rechnerisch die Zeitkomplexität $O(n^2)$.

Anschließend sollte die Frage beantwortet werden, wieviel Zeit ein fiktiver Computer, der 10^6 Operationen pro Sekunde ausführen kann, zum Sortieren der Personenkennzahlen der etwa 60 Mill. Wähler der BRD benötigen würde (Lösung: 114 Jahre). Diese eindrucksvolle Zahl zeigt, daß beide Sortieralgorithmen für derartig große Problemumfänge nicht geeignet sind, so daß die Frage naheliegt, ob es noch wesentlich bessere Sortieralgorithmen gibt. Der Lehrer (alternativ wieder ein leistungsstarker Schüler) stellt dazu den Sortieralgorithmus "Mergesort" (merge (engl.) - verschmelzen) vor, der im besten wie im schlechtesten Fall die Zeitkomplexität O(n·log n) besitzt. Die grundlegende Idee besteht darin, daß die Zahlenfolge der Länge n zunächst in n Zahlenfolgen der Länge 1 zulegt wird und dann wiederholt von links nach rechts zwei benachbarte (sortierte) Zahlenfolgen F_1 und F_2 zu einer sortierten Zahlenfolge F verschmolzen werden. Das Sortieren geschieht mit Hilfe von Positionszeigern in jeder Teilfolge, die die Folgen durchwandern: Man beginnt mit beiden Zeigern am jeweils linken Ende der Folgen F_1 und F_2 und bewegt in einem Schritt denjenigen der beiden Zeiger um eine Position nach rechts (in der betreffenden Teilfolge), der auf die kleinere Zahl zeigt. Eine Zahl wird immer dann in die Folge F aufgenommen, wenn der Zeiger, der weiterwandert, bisher auf diese Zahl gezeigt hat. Sobald eine der beiden Teilfolgen erschöpft ist, übernimmt man den Rest der anderen Folge in die Resultatfolge F.

Beispiel:

Vergleich	Teilfolge F_1	Teilfolge F_2	Resultatfolge
1 < 4	1 , 2 , 3 , 5 , 9 ↑	4 , 6 , 7 , 8 , 10 ↑	∅
2 < 4	1 ↑ , 2 , 3 , 5 , 9	4 , 6 , 7 , 8 , 10 ↑	**1**
3 < 4	1 , 2 ↑ , 3 , 5 , 9	4 , 6 , 7 , 8 , 10 ↑	1 , **2**
5 > 4	1 , 2 , 3 ↑ , 5 , 9	4 , 6 , 7 , 8 , 10 ↑	1 , 2 , **3**
5 < 6	1 , 2 , 3 ↑ , 5 , 9	4 , 6 , 7 , 8 , 10 ↑	1 , 2 , 3 , **4**
9 > 6	1 , 2 , 3 , 5 ↑ , 9	4 , 6 , 7 , 8 , 10 ↑	1 , 2 , 3 , 4 , **5**
9 > 7	1 , 2 , 3 , 5 ↑ , 9	4 , 6 , 7 , 8 , 10 ↑	1 , 2 , 3 , 4 , 5 , **6**
9 > 8	1 , 2 , 3 , 5 ↑ , 9	4 , 6 , 7 , 8 , 10 ↑	1 , 2 , 3 , 4 , 5 , 6 , **7**
9 < 10	1 , 2 , 3 , 5 ↑ , 9	4 , 6 , 7 , 8 , 10 ↑	1 , 2 , 3 , 4 , 5 , 6 , 7 , **8**

F_1 erschöpft			1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , **9**
F_2 erschöpft	**Stop!!**		1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9, **10**

Insgesamt waren beim Verschmelzen der beiden Teilfolgen also 9 Vergleiche erforderlich. Allgemein gilt: Beim Verschmelzen zweier Teilfolgen der Längen n_1 und n_2 sind mindestens $\min(n_1,n_2)$ und höchstens n_1+n_2-1 Vergleiche erforderlich., d. h. O(n) viele Vergleiche pro Verschmelzung. Da die Anzahl der Verschelzungen durch log n abgeschätzt werden kann, hat Mergesort folglich die Zeitkomplexität O(n log n).

Mergesort ist eines der ältesten und bestuntersuchten Verfahren zum Sortieren mit Hilfe von Computern. John von Neumann hat es bereits 1945 vorgeschlagen.

Die Schüler wenden diesen Algorithmus selbständig auf weitere Zahlenfolgen an und starten dann das Programm ***m_sort.pas*** von der Diskette zum Sortieren von 2000, 4000 und 6000 Zufallszahlen.

Bei dem Problem "**Tanzpaarung**" sollte zunächst geklärt werden, daß das Problem entscheidbar ist, d. h., daß es einen Algorithmus gibt, mit dessen Hilfe nach endlich vielen Schritten entschieden werden kann, ob es eine Tanzpaarung der gesuchten Art gibt oder nicht. Das leistet zum Beispiel der triviale Algorithmus, der darin besteht, alle möglichen Tanzpaarungen durchzumustern und stets zu prüfen, ob jeder mit einem ihm sympathischen Partner tanzt. Dieser Algorithmus kann schrittweise spielerisch erarbeitet werden: 2, 3 bzw. 4 Mädchen sitzen auf den Stühlen und werden von Jungen aufgefordert; wieviel verschiedene Möglichkeiten gibt es? Im Extremfall (worst case) müssen alle n! möglichen Tanzpaarungen gebildet werden. Die Schüler erkennen dabei, daß es genau n! verschiedene Tanzpaarungen gibt.

Dieser triviale Algorithmus sollte dann von den Schülern programmiert werden, wobei ich empfehle, den Schülern die relativ anspruchsvolle Prozedur zur schrittweisen Erzeugung aller möglichen Permutationen auf Diskette zur Verfügung zu stellen. Die anschließende Laufzeitanalyse wird folgenden funktionalen Zusammenhang, der der Fakultätsfunktion entspricht, verdeutlichen:

$A(n+1) \approx (n+1)*A(n)$

In einem Lehrervortrag wird dann der folgende Algorithmus vorgestellt (vgl. Walther/Nägler, 1987), der einfache graphentheoretische Grundlagen nutzt und die Zeitkomplexität $O(n^2)$ hat:

1. Bilde einen Graphen, der aus 2n+2 Knotenpunkten besteht, die mit Q, J_1,..., J_n, M_1,..., M_n und S bezeichnet werden, und in dem von Q zu jedem J_i (i = 1,...,n) und von jedem M_i (i = 1,...,n) zu S eine gerichtete Kante verläuft. Außerdem soll vom Knoten J_i eine gerichtete Kante zum Knoten M_j existieren, wenn $s_{ij} = 0$ ist (i,j = 1,...,n).
2. Falls der Knoten S von Q über einen gerichteten Weg erreichbar ist, dann gehe zu 6.
3. Bestimme die Anzahl Z der gerichteten Kanten, die von einem M-Knoten zu einem J-Knoten verlaufen.
4. Wenn Z = n ist, dann entspricht jede gerichtete Kante (M_i,J_k) einem Tanzpaar. ENDE
5. Wenn Z < n, dann existiert keine Tanzpaarung der gesuchten Art. ENDE
6. Drehe die Richtung aller Bögen des Weges von Q nach S um und gehe zu 2.

Die Schüler wenden den Algorithmus anschließend selbständig auf weitere Aufgaben an. Für das angestrebte Unterrichtsziel ist in erster Instanz entscheidend, daß die Schüler die grundlegende Idee des Lösungsweges kennenlernen. Deshalb ist es auch gerechtfertigt, daß die Schüler das zugehörige Programm nicht selbständig erarbeiten, sondern es vom Lehrer auf Diskette zur Verfügung gestellt bekommen (***polytanz.pas***). Die Sequenz sollte mit einer rechnerischen Gegenüberstellung von Zeitangaben für die Abarbeitung polynomialer und exponentieller Algorithmen abgeschlossen werden. Für die Schüler bleibt die Frage offen, ob zu jedem Problem ein polynomialer Algorithmus gefunden werden kann.
wird, einen polynomialen Lösungsalgorithmus zu finden.

5. 2. Sequenz II - Algorithmisch schwere und unlösbare Probleme

Fachwissenschaftliche Grundlagen

Viele Jahrhunderte, und zwar so lange wie es nur darum ging, für bestimmte Probleme geeignete Lösungsalgorithmen zu finden, gaben sich die Mathematiker mit dem folgenden anschaulichen Algorith-musbegriff zu frieden:
Ein Algorithmus ist ein schrittweise ablaufendes Verfahren für Objekte eines bestimmten Bereichs, das folgenden Bedingungen genügt:

- endliche Beschreibbarkeit
- Determiniertheit
- Universalität
- effektive Ausführbarkeit

Die Schwachstelle dieses anschaulichen Algorithmusbegriffs besteht darin, daß die geforderte effektive Ausführbarkeit der einzelnen Schritte des Verfahrens nicht exakt faßbar ist. Anschaulich ist jedem klar, was er darunter zu verstehen hat, aber genaugenommen hängt die effektive Ausführbarkeit von Prozessen immer vom jeweiligen Stand der Wissenschaft und Technik ab und hat somit stets relativen Charakter. Zum Beispiel hielten die Menschen vor 150 Jahren die effektive Ausführung eines direkten Telefongesprächs von Berlin nach Tokio bestimmt nicht für möglich.

Während vieler Jahrhunderte, in denen die Mathematiker sich darauf beschränkten, für bestimmte Aufgaben Algorithmen zu deren Lösung anzugeben, wurde die Notwendigkeit, den Algorithmusbegriff exakt zu definieren, nicht empfunden, da an der effektiven Ausführbarkeit der konkret angegebenen Verfahren nie ein Zweifel bestand.

Das änderte sich erst, als zunehmend Probleme auftraten, die sich einer algorithmischen Lösung hartnäckig widersetzten und Zweifel an der bis dato dominierenden Auffassung aufkommen ließen, daß jedes mathematische Problem algorithmisch lösbar ist. Beispiele dafür sind:

- *die Vermutung von Fermat (1601 - 1665)*: Es gibt keine natürliche Zahl $n \geq 3$, für die positive ganze Zahlen x, y und z mit $x^n + y^n = z^n$ existieren.
- *die Vermutung von Goldbach (1690 - 1764)*: Jede gerade Zahl > 4 ist als Summe von zwei Primzahlen darstellbar.
- *das Entscheidungsproblem für den Prädikatenkalkül der 1. Stufe*: Ist ein logischer Ausdruck des Prädikatenkalküls der 1. Stufe allgemeingültig oder nicht?
- *das 10 . Problem von Hilbert (1862 - 1943)*: Existiert ein Algorithmus zur Lösung von beliebigen diophantischen Gleichungen?

Während für die beiden zuletzt genannten Probleme bewiesen werden konnte, daß sie nicht entscheidbar sind (Church, 1936, bzw. Matijasevic, 1970), konnten die Vermutungen von Fermat und Goldbach bis heute weder bewiesen noch widerlegt werden.

Um auf derartige Probleme eine negative Antwort der Form “Es gibt nachweislich keinen Algorithmus zur Lösung des Problems!” geben zu können, bedurfte es offensichtlich einer im mathematischen Sinne exakten Definition des Algorithmusbegriffs.

Die ersten Arbeiten dazu erschienen in den 30er Jahren unseres Jahrhunderts. Interessant ist in diesem

Zusammenhang, daß die einzelnen Forscher (Gödel, Church, Kleene, Post, Turing u.a.) von verschiedenen technischen und logischen Gesichtspunkten aus an das Problem herangingen, so daß zunächst sehr unterschiedliche Definitionen entstanden, die sich später aber alle als äquivalent erwiesen. Diese Tatsache hat erkenntnistheoretische Bedeutung und bezeugt, daß die erarbeiteten Definitionen sehr gut das Wesen des anschaulichen Algorithmus erfassen.

Die ersten in dieser Sequenz zu bearbeitenden Probleme gehören zur Klasse der Probleme, für die beim gegenwärtigen Stand der Mathematik kein polynomialer Lösungsalgorithmus bekannt ist und vermutlich auch nicht existiert. Sie sind praktisch unlösbar, da die benötigte Rechenzeit schon bei relativ kleinen Problemumfängen extrem hoch ist. Unglücklicherweise ist diese Klasse sehr umfangreich und täglich kommen neue Probleme hinzu. Um diese Klasse genauer zu charakterisieren, ist es erforderlich, den Begriff des nichtdeterministischen Algorithmus einzuführen. Er ist eine Erweiterung des intuitiven Algorithmusbegriffs in dem Sinne, daß auf die Forderung verzichtet wird, daß der nächstfolgende Schritt bei der Abarbeitung eines Algorithmus stets eindeutig bestimmt ist. Man nimmt statt dessen an, daß man den nächsten Schritt zu jedem Zeitpunkt aus einer endlichen Menge von möglichen Schritten auswählen (raten) kann . Die Abarbeitung eines nichtdeterministischen Algorithmus erfolgt in zwei Phasen. In der ersten Phase, der Ratephase, wird ein Lösungskandidat geraten und in der zweiten Phase wird getestet, ob der Lösungskandidat tatsächlich eine Lösung darstellt. Die Nichtdeterminiertheit liegt dabei lediglich in der ersten Phase, während die zweite Phase völlig deterministisch verläuft.
Betrachten wir dazu das sogenannte Rundreiseproblem, daß als Entscheidungsproblem wie folgt formuliert werden kann:
Es seien k und n natürliche Zahlen und D= (d_{ij}) eine Matrix vom Typ (n,n). d_{ij} sei dabei die Entfernung von der Stadt **i** zur Stadt **j**. Ein Handelsreisender soll, ausgehend von der 1.Stadt, n-1 andere Städte genau einmal durchreisen und am Ende wieder in die Ausgangsstadt zurückkehren. Gibt es eine Rundreise derart, daß die Summe der zurückgelegten Entfernungen kleiner oder gleich k ist.
Um eine Lösung zu bestimmen, können Sie wie folgt vorgehen: Sie raten zunächst nichtdeterministisch irgendeine Rundreise. Anschließend benutzen Sie einen polynomialen Algorithmus zur Bestimmung der Summe der Teilstrecken dieser Rundreise und zum Test, ob diese Summe kleiner oder gleich k ist.

Alle Probleme, die mit nichtdeterministischen Algorithmen in polynomialer Zeit gelöst werden können, fasse ich in einer Klasse zusammen, die ich mit **NP** bezeichne. Die Klasse aller Probleme, die mit

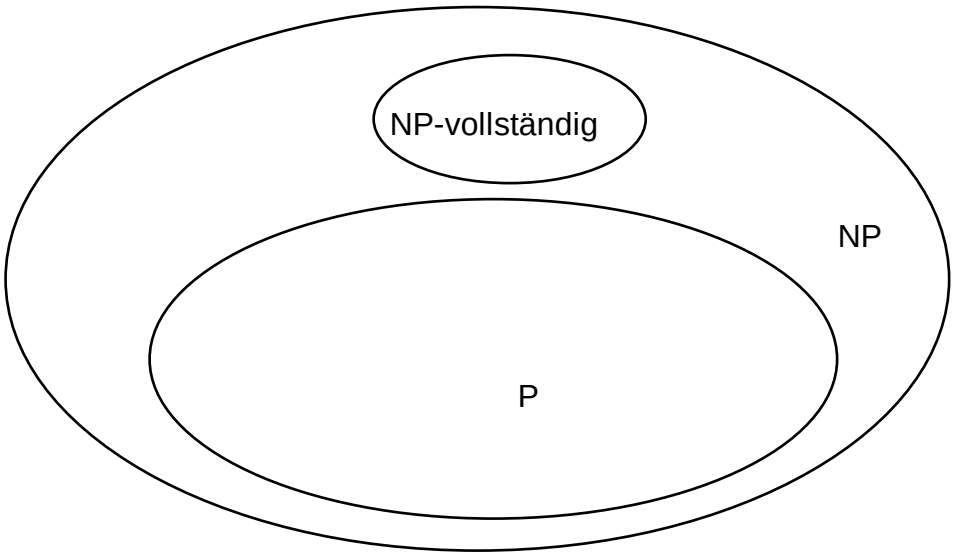
deterministischen Algorithmen in polynomialer Zeit gelöst werden können, bezeichne ich mit **P**. Dann gilt offensichtlich die Aussage

$$\mathbf{P} \subseteq \mathbf{NP},$$

denn jeder deterministische Algorithmus ist ein Spezialfall eines nichtdeterministischen. Ein noch immer offenes Problem der Informatik ist die Frage, ob diese Inklusion echt ist. Das ist der Inhalt des sogenannten **P**-**NP**-Problems.

Da gegenwärtig weltweit vermutet wird, daß **P** eine echte Teilmenge von **NP** ist, wird nach Problemen gesucht, die in NP, aber nicht in P liegen. Diese Suche führte zur Klasse der sogenannten **NP**-vollständigen Probleme:

Ein Problem heißt **NP-vollständig**, wenn es erstens zu Klasse **NP** gehört und zweitens vollständig in folgendem Sinne ist: Findet man einen deterministischen polynomialen Algorithmus für das Problem, so kann man aus diesem einen polynomialen Algorithmus für jedes Problem aus **NP** ableiten und dann wäre **P**=**NP**.



Die **NP**-vollständigen Probleme sind sozusagen die "schwierigsten" Probleme. Wenn $\mathbf{P} \neq \mathbf{NP}$ ist, dann kann man kein **NP**-vollständiges Problem mit einem deterministischen Algorithmus in polynomialer Zeit lösen.

Eine Lösung des **P**-**NP**-Problems hätte weitreichende Konsequenzen: Wäre nämlich $\mathbf{P} = \mathbf{NP}$, dann würde mit Sicherheit auch für die Probleme ein polynomialer Lösungsalgorithmus existieren, , für die bis heute nur exponentielle bekannt sind.

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

Sowohl das Rundreiseproblem als auch das Rucksackproblem gehören nachweislich zur Klasse der **NP**-vollständigen Probleme. Die exakten Beweise dieser Aussage sind sehr umfangreich und können im Rahmen dieser Arbeit nicht nachvollzogen werden. Interessierten Lehrerinnnen und Lehrern empfehle ich das Buch von M. R. Garey und D. S. Johnson, das als Standardwerk für die Theorie der NP-Vollständigkeit gilt.

Ziele der Sequenz

Die Schüler

- wissen, daß es eine große Klasse von Problemen gibt, für die beim gegenwärtigen Stand der Mathematik kein polynomialer Algorithmus bekannt ist und vermutlich auch nicht existieren wird,
- kennen das Rundreiseproblem und das Rucksackproblem als typische Vertreter dieser Aufgabenklasse,
- kennen für beide Probleme triviale Lösungsalgorithmen auf der Grundlage der Methode der vollständigen Durchmusterung und können deren exponentiellen Rechenzeitaufwand rechnerisch und experimentell abschätzen,
- kennen das Halteproblem für PASCAL-Programme und verstehen den exakten Beweis der algorithmischen Unlösbarkeit dieses Problems.
- erkennen die Notwendigkeit der Präzisierung des Algorithmusbegriffs, wenn die prinzipielle algorithmische Unlösbarkeit von Problemen nachgewiesen werden soll,

Didaktisch-methodische Hinweise

Die Schüler lernen zum Einstieg an einem konkreten Beispiel das Rundreiseproblem kennen. Erfahrungsgemäß ist zu erwarten, daß die meisten Schüler bei der Lösungssuche nach dem sogenannten greedy-Prinzip (greedy (engl.) - gierig) vorgehen: Unter den noch nicht besuchten Städten wird stets die ausgesucht, die vom gegenwärtigen Standort den geringsten Abstand hat. Der Lehrer hat bei der Auswahl des konkreten Beispiels darauf zu achten, daß diese Vorgehensweise nicht zu einer optimalen Lösung führt. In einem Lehrer-Schüler-Gespräch wird die Ähnlichkeit zum Zuordnungsproblem "Tanzpaarung" herausgearbeitet und geklärt, daß die Lösung des Problems zumindest wieder mit dem trivialen Algorithmus möglich ist, der im Durchmustern aller möglichen Fälle besteht. Die Schüler erkennen dessen Zeitkomplexität O(n!) und schätzen rechnerisch den Zeitaufwand für einen fiktiven Computer ab.

Auf das Programmieren des Lösungsalgorithmus würde ich verzichten und den Schülern ein

zugehöriges Programm zur Verfügung stellen.

Das Rucksackproblem sollten die Schüler ebenfalls an einem konkreten Beispiel kennenlernen. Auch hier ist zu vermuten, daß die Schüler bei der Suche nach einer optimalen Lösung wieder nach dem greedy-Prinzip vorgehen, d. h., sie packen stets den Gegenstand ein, der unter denen, die noch eingepackt sind, maximalen Wert hat.

Ein exakter Lösungsalgorithmus könnte auch hier darin bestehen, alle Auswahlmöglichkeiten durchzumustern und unter denen, deren Gesamtmasse die Tragfähigkeit des Rucksackes nicht übersteigt, eine Auswahl mit maximalem Gesamtwert zu ermitteln.

Modellierung

Für die systematische Erzeugung der Auswahlmöglichkeiten gibt es sehr verschiedene Varianten, wobei sich der Weg über Dualzahlvektoren der Länge n als sehr günstig erwies, d. h. konkret, ich erzeuge alle Vektoren $(x_1,..., x_n)$ der Länge n mit $x_i \in \{0, 1\}$ und interpretiere die Vektoren wie folgt:

$x_i = 1$, wenn der Gegenstand G_i eingepackt wird.

$x_i = 0$, wenn der Gegenstand G_i nicht eingepackt wird.

Da es offensichtlich 2^n verschiedene derartige Vektoren der Länge n gibt, die als Dualzahlen der Dezimalzahlen von 0 bis 2^n-1 interpretiert werden können, bietet sich für die Erzeugung der Auswahlvektoren die Umwandlung der Dezimalzahlen von 0 bis 2^n-1 in die zugehörigen Dualzahlvektoren von (0,...,0) bis (1,...,1) mit Hilfe des Divisions-Algorithmus an, der den Schülern zu diesem Zeitpunkt schon bekannt sein sollte.

Beispiel: Umwandlung der Dezimalzahl 124_{10} in die zugehörige Dualzahl der Länge 8

1. Schritt: 124 : 2 = 62 Rest **0**
2. Schritt: 62 : 2 = 31 Rest **0**
3. Schritt: 31 : 2 = 15 Rest **1**
4. Schritt: 15 : 2 = 7 Rest **1**
5. Schritt: 7 : 2 = 3 Rest **1**
6. Schritt: 3 : 2 = 1 Rest **1**
7. Schritt: 1 : 2 = 0 Rest **1**
8. Schritt: 1 : 2 = 0 Rest **0** Stop!!

Die Reste von unten nach oben ergeben die zugehörige Dualzahl 01111100_2

Probe: $01111100_2 = 0*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 64+32+16+8+4=124$

Die Schüler können nach diesen Vorbetrachtungen das Rucksackproblem wie folgt formulieren:
Bestimme die $x_i \in \{0, 1\}$ (i=1,2,...,n) so, daß $\Sigma x_i \cdot m_i \leq T$ und $\Sigma x_i \cdot w_i$ = max!
Nach der Programmierung des trivialen Lösungsalgorithmus und der anschließenden Laufzeitanalyse sollte die Sequenz vom Lehrer mit dem Hinweis abgeschlossen werden, daß beide Probleme zu den sogenannten "algorithmisch schweren" Problemen gehören, für die beim gegenwärtigen Stand der Theoretischen Informatik kein polynomialer Algorithmus existiert und vermutlich auch in Zukunft nicht existieren wird. Wichtig ist auch der Hinweis, daß die "algorithmisch schweren" Probleme aber den Vorteil haben, daß sie zumindest theoretisch lösbar sind.

Für den Einstieg in das Haltproblem eignet sich sehr gut die Diskussion folgender Situation:
Sie geben ein Programm in den Computer ein, starten es und stellen fest, daß das Programm in der beobachteten Zeit nicht stoppt. Warum stoppt es nicht? Welche Fälle sind denkbar?
Erwartete Antwort: Entweder das Programm benötigt soviel Zeit (siehe behandelte Beispiele) oder das Programm durchläuft – bedingt durch einen nichterkannten Programmierfehler - eine Schleife, aus der es nicht mehr herauskommt (Endlosschleife)
Fazit: *Wünschenswert wäre ein Pascal-Programm, mit dessen Hilfe man schon vorab entscheiden kann, ob das Programm stets stoppt oder nicht.*
Ein solches Problem gibt es aber nachweislich nicht, wie der folgende indirekte Beweis zeigt.

Zum Beweis des Halteproblems

Zur Vorbereitung des Beweises sind bei den Schülern folgende Einsichten zu arbeiten:
- Jedes PASCAL-Programm kann als Zeichenkette - das ist dann eine sogenannte Niederschrift eines Algorithmus - dargestellt werden und als solches dann auch als Eingabe in PASCAL-Programmen verwendet werden.
- PASCAL-Programme sind folglich auf PASCAL-Programme (z. B. in Form einer Niederschriften) anwendbar, und speziell kann jedes PASCAL-Programm auf sich selbst (d. h. auf eine eigene Niederschrift) angewendet werden.

Anstelle des eigentlichen Halteproblems kann dann zunächst folgendes Problem betrachtet werden:
Gibt es ein PASCAL-Programm, das von jedem beliebigen PASCAL-Programm, das auf sich selbst angewendet wird, entscheidet, ob es nach endlich vielen Schritten stoppt oder nicht (**Selbstanwend-**

barkeitsproblem)? Programme, die bei Anwendung auf sich selbst nach endlich vielen Schritten stoppen, nenne ich selbststoppend.

Das Selbstanwendbarkeitsproblem ist offensichtlich ein Spezialfall des Halteproblems, und folglich gilt, daß aus seiner algorithmischen Unlösbarkeit die Unlösbarkeit des Halteproblems folgen würde.

Der Beweis wird indirekt geführt (vgl. Bitsch/Rompel, 1983).

Es wird angenommen, daß ein Programm - ich nenne es "Stoptest" existiert, daß von jedem Programm entscheiden kann, ob es bei Anwendung auf sich nach endlich vielen Schritten stoppt (also selbststoppend ist) oder nicht. Dieses Programm hat o.B.d.A. eine boolesche Variable, die bei Abarbeitung des Programms “Stoptest” den Wert *TRUE* erhält, wenn festgestellt wird, daß das eingegebene Programm selbststoppend ist. Ansonsten erhält die Variable den Wert *FALSE*.

```
PROGRAM Stoptest;
...
VAR
...
stop : Boolean;
...
BEGIN
{in diesem Programmabschnitt erfolgt der Test, ob das eingegebene Programm selbststoppend ist (dann
stop := TRUE) oder nicht (dann stop := FALSE)}
IF stop THEN writeln ('Das Programm ist selbststoppend.')
          ELSE writeln (`Das Programm ist nicht selbststoppend. ')
END.
```

Aus diesem Programm "Stoptest" wird dann ein Programm "seltsam" konstruiert.

```
PROGRAM seltsam;
...
VAR
...
stop : Boolean;
...
BEGIN
{in diesem Programmabschnitt erfolgt der Test, ob das eingegebene Programm selbststoppend ist (dann
stop := TRUE) oder nicht (dann stop := FALSE)}
  IF stop THEN BEGIN writeln ('Das Programm ist selbststoppend.'); WHILE TRUE DO ; END
          ELSE writeln (`Das Programm ist nicht selbststoppend. ')
END.
```

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

Beide Programme “seltsam” und “Stoptest” sind nahezu identisch. Der Unterschied besteht nur darin, daß im Programm "seltsam" hinter die positive Ausgabe *'Das Programm ist selbststoppend.'* zusätzlich eine Endlosschleife *WHILE TRUE DO;* eingebaut wurde.
Wie jedes andere Programm muß auch das Programm "Seltsam" entweder selbststoppend sein oder nicht.
1. Angenommen, das Programm "Seltsam" ist selbststoppend. Startet man es dann mit seinem eigenen Programm als Eingabe, so wird zunächst der “Selbstsstoptest” durchgeführt und die boolesche Variable erhält den Wert *TRUE*. Das bedingt, daß die positive Antwort " Das Programm ist selbststoppend." ausgegeben wird, der Computer anschließend aber in die Endlosschleife gerät und im Widerspruch zur Annahme kein Stop erfolgt.
2. Angenommen, daß Programm "Seltsam" ist nicht selbststoppend. Auch in diesem Falle würde das Programm mit seinem eigenen Programmtext als Eingabe zunächst den “Selbststoptest” abarbeiten, wobei die boolesche Variable nun aber den Wert FALSE erhält. Das aber bedingt die Ausgabe "Das Programm ist nicht selbststoppend." und den anschließenden Stop des Programms, was wiederum im Widerspruch zur Annahme steht.
Das Programm "Seltsam" trägt also völlig zu Recht seinen Namen, da es weder die Eigenschaft "selbststoppend" noch deren logische Negation besitzt. Da das aber nach den Gesetzen der Logik nicht möglich ist, kann ein solches Programm "Seltsam" nicht existieren. Dann kann es aber auch kein PASCAL-Programm "Stoptest" geben, da "Seltsam" aus dem Programm "Stoptest" konstruiert werden kann.

Zum Abschluß der Sequenz sollten die Ergebnisse der gesamten Unterrichtsreihe noch einmal zusammen gefaßt werden. Dabei ist insbesondere die Frage zu klären, warum eine Präzisierung des bisher benutzten Algorithmusbegriffs erforderlich ist. Die Diskussion könnte eingeleitet werden mit einen Schülervortrag, in dem die charakteristischen Eigenschaften eines Algorithmus zusammenfassend erläutert werden. In dem anschließenden Unterrichtsgespräch ist herauszuarbeiten, warum dieser anschauliche Algorithmusbegriff im streng mathematischen Sinne nicht exakt ist und welche Konsequenzen diese Tatsache für den Nachweis der prinzipiellen algorithmischen Unlösbarkeit von Problemen hat.

Diesen Ausführungen schließt sich im Wahlthema *Theoretische Informatik* des Informatik-

Leistungskurses ein 2. Teil "*Formale Sprachen und abstrakte Automaten*" an.

Laut Rahmenplan sollen die Schüler dabei

- Aufbau und Funktionsweise eines Computers durch theoretische Maschinenmodelle (TURINGmaschinen, Registermaschinen) beschreiben
- Abstrakte Automatenmodelle entwerfen
- Begriffe wie Alphabet, Zeichen, Wort, Satz, Grammatik, Sprache, Syntax und Semantik kennen und sie auf die natürliche Sprache und formale Sprachen anwenden
- Gemeinsamkeiten und Unterschiede von natürlichen und formalen Sprachen vertiefen
- Formale Sprachen mittels abstrakter Automaten und über Grammatiken der Chomsky-Hierarchie beschreiben
- die CHURCHsche Hypothese kennen

Literatur

Bitsch,G.; Rompel, H.: Informatik für Gymnasien. Kiehl-Verlag Ludwigshafen 1983

Breier, N.: Theoretische Aspekte der Informatik - eine Unterrichtsreihe zur Vermittlung theoretischen Grundwissens der Informatik im Rahmen des fakultativen Informatikunterichts in der Abiturstufe. In: Preprint-Reihe "Mathematik", Bd. 26, Ernst-Moritz-Arndt-Universität Greifswald 1990

Breier, N.: Grenzen des Computereinsatzes. -In: LOG IN 10 (1990) H.1, S. 42-50 (Teil 1); LOG IN 10 (1990) H.3, S. 37-42 (Teil 2)

Breier, N.: Berechenbarkeit und Entscheidbarkeit. - In: LOG IN 11 (1991) H. 3, S.29-35

Engel, A.: Elementarmathematik vom algorithmischen Standpunkt. Ernst Klett Verlag Stuttgart 1977

Garey, M.R.; Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman Publ. Co., San Francisco 1979

Kerner, I. O.: Was jedermann über Informatik wissen sollte. - In: LOG IN 9 (1989), H. 6, S. 12-14 und LOG IN 10 (1990), H. 1, S. 8-12

Malcev, A.I.: Algorithmen und rekursive Funktionen. Akademie-Verlag Berlin 1974

Rabin, M.O.: Degree of difficulty of computing a function and a partial ordering of recursive sets.-In: Tech. Rep. No. 2, Hebrew u., Jerusalem, Israel, 1960

Walther, H.; Nägler, G.: Graphen, Algorithmen, Programme. VEB Fachbuchverlag Leipzig 1987

PD Dr. Norbert Breier, Universität Greifswald, Institut für Mathematik und Informatik

Materialübersicht

Sequenz 1: Effiziente Algorithmen

M 1	(Tx)	Ripplesort
M 2	(Pg)	r_sort.pas
M 3	(Tx)	Laufzeitanalyse für das Programm ***r_sort.pas***
M 4	(Tx)	Bubblesort
M 5	(Pg)	b_sort.pas
M 6	(Tx)	Laufzeitanalyse für das Programm ***b_sort.pas***
M 7	(Tx)	Mergesort
M 8	(Pg)	m_sort.pas
M 9	(Tx)	Tanzpaarung
M 10	(Pg)	permutiere.pas Prozedur zur Erzeugung aller Permutationen
M 11	(Pg)	expotanz.pas Programm mit exponentiellem Rechenzeitaufwand
M 12	(Tx)	Laufzeitanalyse für das Programm ***expotanz.pas***
M 13	(Tx)	polynomialer Algorithmus für das Problem "Tanzpaarung"
M 14	(Pg)	polytanz.pas Programm mit polynomialem Rechenzeitaufwand
M 15	(Tx)	Vergleich zwischen polynomialer und exponentieller Rechenzeit

Sequenz 2: Algorithmisch schwere und unlösbare Probleme

M 16	(Tx)	Rundreiseproblem
M 17	(Tx)	Rucksackproblem
M 18	(Pg)	rucksack.pas
M 19	(Tx)	Laufzeitanalyse für das Programm ***rucksack.pas***
M 20	(Tx)	Halteproblem
M 21	(Tx)	Anmerkungen zur Präzisierung des Algorithmusbegriffs