Prüfungsprotokoll

Prüfung:	Vordiplom Informatik Teilprüfung Praktische Informatik (Info I, IIa, IIIb)
Prüfer:	Prof. Hesse
Termin:	9. Oktober 2001, 15:00 Uhr
Note:	1,0

Die Prüfung:

In der Praktischen Informatik ist der zentrale Begriff der Algorithmus. Was ist der Unterschied zwischen einem Programm und einem Algorithmus?
(Bin nicht gleich darauf gekommen:) Ein Algorithmus ist unabhängig von der Programmiersprache und von der Ausführungsmaschine. Ein Programm dagegen ist in einer bestimmten Programmiersprache geschrieben und läuft nur auf einem bestimmten Rechner.
Was bedeuten die Begriffe "deterministisch" und "determiniert"?
Ein Algorithmus ist deterministisch, wenn es für jede Eingabe einen vorher bestimmten Ablauf gibt. Ein Algorithmus ist determiniert, wenn für jede Eingabe das Resultat eindeutig bestimmt ist. Ein Beispiel: Die Nachrichtenübertragung im Internet ist nicht deterministisch, weil kein Mensch weiß, auf welchem Weg die Nachricht zum Empfänger gelangt. Sie ist aber determiniert, weil man das Ergebnis kennt: die gesendete Nachricht kommt beim Empfänger an.
[Freut sich] Schönes Beispiel. Was ist ein imperativer Algorithmus?
Das Charakteristikum für imperative Algorithmen ist, daß der Zustand von Behältern, also Variablen durch Wertzuweisungen verändert wird. Das bedeutet, daß der Zustand aller Behälter vom Zeitpunkt abhängt. Diese Zustandsabhängigkeit der Behälter macht imperative Algorithmen so schwer verifizierbar auf Korrektheit und Terminierung. Diese Schwierigkeit wird noch dadurch verstärkt, daß die Hauptkontrollstruktur bei imperativen Algorithmen die Schleife ist. Es ist oftmals schwierig festzustellen, ob eine Schleife überhaupt terminiert. Beispiel: Ulam-Funktion.
Naja, Ulam ist kein so gutes Beispiel...
Ja, Sie haben Recht. Man kann die Funktion auch rekursiv formulieren...
Wenn imperative Algorithmen so schwer zu verifizieren sind, warum programmiert man dann imperativ?
Man muß nicht imperativ programmieren. Es gibt da noch die deklarativen Sprachen. "declarare" heißt etwas erklären. Ein Programm in einer deklarativen Sprache ist also eher eine Problembeschreibung und keine Folge von Anweisungen. Die funktionalen Sprachen und die Logik-Sprachen sind deklarative Sprachen. Bei funktionalen Sprachen ist ein Programm eine Sammlung von Funktionsdefinitionen. Bei Logik-Sprachen besteht ein Programm aus expliziten und impliziten Axiomen, also aus Fakten und Regeln. Die Programmausführung ist hier eine Anfrage an das System - eine Behauptung. Das System versucht eine geeignete Variablenbelegung zu finden, so daß sich die Behauptung aus den Axiomen schließen läßt.

Was ist die Grundlage dafür, daß das System dies zeigen kann?
Bei funktionalen Sprachen ist es das λ-Kalkül. Für die Logik-Sprachen hatten wir nichts im Skript stehen.
Ich meine etwas anderes.
(Nach einigem Überlegen:) Ahh, Sie meinen die Rekursion! Ok, Das System versucht mittels Unifikation und Backtracking eine richtige Variablenbelegung zu finden.
Genau, Unifikation, das ist der Begriff. Was passiert da genau?
Die Konstanten werden der Reihe nach an die Variablen gebunden und dann wird jedesmal geschaut, ob diese Belegung zu irgendwelchen Widersprüchen führt. Wenn ja, dann erfolgt Backtracking, d.h. die letzte Variablenbelegung wird durch eine andere ersetzt. Die gültigen Variablenbelegungen werden jedesmal ausgegeben. Dies geht so weiter, bis keine Variablenbelegungen mehr übrig sind.
Sagt Ihnen der Begriff "referentielle Transparenz" etwas?
Klar! Bei imperativen Programmiersprachen hat man das Problem, daß z.B. ein Ausdruck $f(2) = f(2)$ nicht immer *wahr* ergibt, weil die Funktion f irgendwelche Seiteneffekte haben könnte. $f(2)$ stellt also keinen Wert da, sondern einen aktuellen Zustand eines Behälters. Bei deklarativen Sprachen gibt es diese Seiteneffekte nicht. Ein Wert ist ein Platzhalter, wie in der Mathematik. D.h. wenn zwei Platzhalter den gleichen Wert haben, kann man sie jederzeit gegeneinander vertauschen. Dies ist das Substitutionsprinzip. Wenn wir nun einen großen Ausdruck betrachten, der aus mehreren Teilausdrücken besteht, dann ist es egal, in welcher Reihenfolge die einzelnen Teilausdrücke ausgewertet werden. Dies nennt man referentielle Transparenz.
Es kann aber trotzdem ein Fall auftreten, so daß auch bei deklarativen Sprachen die Korrektheit nicht auf der Hand liegt. Können Sie sich so etwas denken?
(Habe keine Idee...)
Wann kann es trotz der referentiellen Transparenz ein Problem geben?
Hat das etwas mit Lazy-Evaluation zu tun?
Genau. Was ist das?
(Verdammt, habe mich darauf nicht vorbereitet. Versuche mich an das Programmierpraktium in Haskell zu erinnern...) Wenn wir beispielsweise einen boolschen Ausdruck haben, in dem die Teilausdrücke mit *UND* verknüpft sind. Wenn das erste Konjunktionsglied als *unwahr* ausgewertet wird, müssen die restlichen Konjunktionsglieder nicht mehr ausgewertet werden.
Angenommen, das erste Konjunktionsglied wird mit wahr *ausgewertet. Wann kann der Gesamtausdruck trotzdem "falsch" sein?*
[...dummy-mode on...]
Was kann denn z.B. mit dem zweiten Teilausdruck sein?
(Mir dämmerts...) Ahhh, seine Auswertung kann in eine Endlosschleife laufen, also nicht terminieren.
Jaaa... [Erklärt mir noch die Einzelheiten, dann Themawechsel] Kommen wir zu Suchalgorithmen. Erzählen Sie was dazu.
Es gibt da z.B. die lineare Suche, die binäre Suche und die Interpolationssuche. Die lineare Suche funktioniert auf allen Behälter-Datentypen, also auf allen Datenstrukturen. Man entnimmt aus der Datenstruktur jeweils ein Element und guckt, ob es das gesuchte

ist. Dies erfordert keine besondere Struktur des Behälter-Datentyps. Die Suche hat aber linearen Aufwand. Einen besseren Aufwand bekommt man bei der binären Suche und der Interpolationssuche.
Wie funktioniert die Interpolationssuche?
Man benötigt dazu ein Array, in dem die Datensätze sortiert sind. Betrachten wir ein Array mit Namen. Wenn wir z.B. *Müller* suchen, dann können wir ungefähr abschätzen, wo sich der Datensatz im Array befindet, wenn wir die Anzahl der Elemente im Alphabet betrachten - hier 26 - und die Position des Buchstabens M im Alphabet. M hat ungefähr die Position 15, oder?
13.
Ok, nun kann man die 13 in Relation zu der Gesamtanzahl der Buchstaben im Alphabet setzen, also 13 durch 26, und wenn man diesen Quotienten mit der Arraylänge multipliziert, dann erhält man die ungefähre Position des Datensatzes *Müller* im Array. Die verschiedenen Namen müßten allerdings relativ gleichverteilt vorkommen, denn sonst wird die Schätzung sehr ungenau.
Was macht man dann?
Dann guckt man, ob das Element, was man gefunden hat, kleiner oder größer ist und sucht entweder links oder rechts davon weiter nach dem richtigen Element.
Genau, man sucht dann linear weiter.
Bleibt noch zu sagen, daß die binäre Suche und die Interpolationssuche effizienter sind als die lineare Suche - binäre Suche: $O(\log n)$. Aber man kann sie nur auf sortierte Arrays anwenden. Somit muß man sich überlegen, ob die effizientere Suche den Aufwand wettmacht, die Daten aus einer anderen Datenstruktur in ein Array zu kopieren und sie anschließend noch zu sortieren.
Fazit? Wann lohnt sich binäre oder Interpolationssuche?
Wenn viel in den Daten gesucht wird, lohnt sich der Aufwand für das Umkopieren und Sortieren. Übrigens wäre es auch gut, wenn das Sortieren nicht so aufwendig wäre. Womit wir bei der Bedeutung von Sortierverfahren wären...
[Grinst] Gut, kommen wir zum nächsten Thema. Wir haben eben die Interpolationssuche behandelt...
[Falle Prof. Hesse ins Wort] Hashing!
[Fängt an zu lachen] Ja, genau. Darauf wollte ich hinaus. Bitte, erzählen Sie.
Dem Hashing liegt eine ähnliche Idee zugrunde wie der Interpolationssuche. Es gibt da eine Hash-Funktion, die jedem Datensatz einen Sektor (engl. Bucket) zuordnet.
Wieviel Buckets sollte es geben?
Idealerweise mindestens so viele wie es Datensätze gibt.
Was passiert, wenn es weniger Buckets gibt als Datensätze?
Die Hash-Funktion ist dann eine surjektive Abbildung. Es kommt zu Kollisionen, d.h. es werden einige Elemente in bereits belegte Buckets abgebildet. Diese Elemente nennt man Überläufer.
Welches Verhältnis von Datensätzen zu Buckets, also welchen Belegungsfaktor hat man in einem solchen Fall?
Einen ziemlich großen.
Genau, und was sollte die Obergrenze für den Belegungsfaktor sein?

Er sollte maximal eins sein.
Richtig. Was ist denn ein guter Belegungsfaktor?
Im Skript stand, das sich ein Belegungsfaktor von 0.8 in der Praxis als gut herausgestellt hat.
Bei welchem Belegungsfaktor ist denn die Wahrscheinlicheit groß, daß es zu Kollisionen kommt?
[Ich gucke entsetzt]
Keine Angst, ich möchte nicht die exakte Formel zur Berechnung der Wahrscheinlichkeit hören, die weiß ich auch nicht auswendig.
Also im Skript stand, daß 0.8 ganz gut ist. D.h. da drüber und da drunter wird die Wahrscheinlichkeit wohl ungünster sein. Jedenfalls hängt die Wahrscheinlichkeit vom Belegungsfaktor ab. (Wir haben noch etwas über den Belegungsfaktor gesprochen und in diesem Zusammenhang hat Prof. Hesse das Geburtstagsproblem angesprochen. Er erzählte mir, daß die Wahrscheinlichkeit, daß zwei Leute in einem Raum am gleichen Tag Geburtstag haben, bereits ab einer überraschend kleinen Anzahl von Leuten auf über 50% ansteigt.)
Was kann man bei Kollisionen machen?
Es gibt da verschiedene Strategien: offenes und geschlossenes Hashing. Beim offenen Hashing befindet sich in den Sektoren eine Referenz auf eine verkettete Liste, in der die Überläufer gespeichert werden. Dabei gibt es wiederum zwei Möglichkeiten: direkte und separate Verkettung. Bei der separaten Verkettung wird jeweils ein Datensatz noch im Sektor gespeichert und alle Überläufer kommen in die verkettete Liste. Bei der direkten Verkettung befinden sich in den Sektoren nur Referenzen auf die verketteten Listen.
Und wie ist das beim geschlossenem Hashing?
Dort werden Überläufer in den noch freien Sektoren gespeichert. Dazu braucht man eine Sondierungsfolge. Das ist eine Permutation der Hash-Adressen, die zu jedem Sektor angibt, in welchem anderen Sektor der Überläufer gespeichert werden soll.
Was muß man beim geschlossenem Hashing beachten?
Man darf die Datensätze in den Sektoren nicht einfach löschen. Denn sonst würden Überläufer, also Datensätze, die nur deswegen in einem anderen Sektor gespeichert wurden, weil der eigentlich richtige Sektor schon belegt war, nicht mehr wiedergefunden.
Wie kann man das Problem lösen?
Man löscht Datensätze nicht einfach, sondern markiert sie nur als solches. Man kann sie erst dann löschen, wenn ein weiterer Datensatz in den Sektor abgebildet wird. Der alte ist dann sozusagen enlassen. (Stand nicht im Skript)
[Überlegt kurz] Ja richtig. Jedenfalls ist das Ganze etwas uneffizient, weil Speicherplatz verschwendet wird. Lassen Sie uns jetzt über Bäume sprechen. Bäume sind sehr wichtig in der Informatik. Wo kommen denn Bäume überall vor?
[Zähle auf...] Syntaxüberprüfung ... bei geklammerten und geschachtelten Ausdrücken ... Suchbäume ...
Suchbäume, das paßt gut zu unserem vorigen Thema. Was ist ein Suchbaum?
Das ist ein Binärbaum, dessen Knoten einen Schlüsselwert enthalten, auf dem eine Ordnungsrelation definiert ist. Es muß außerdem gelten, daß der Schlüsselwert des linken Nachfolgers lexikographisch kleiner-gleich des Schlüsselwertes der Wurzel ist, und der

Schlüsselwert der Wurzel kleiner als der des rechten Nachfolgers ist. Wenn man auf diesem Baum eine Inorder-Traversierung durchführt, erhält man die Elemente in aufsteigender Reihenfolge.
Ist ein Suchbaum balanciert?
Nein, das ist nicht die Voraussetzung für einen Suchbaum. Es wäre aber gut, wenn er balanciert wäre, denn sonst hätten wir nicht diesen Suchaufwand von nur $O(\log n)$. Angenommen, es würde eine sortierte Folge von Datensätzen vorliegen, die wir der Reihe nach in den Suchbaum einfügen würden. Es ergäbe sich eine lineare Liste. Der Suchaufwand wäre linear. Der Suchbaum würde seinen Namen nicht verdienen. Daher muß man beim Einfügen zuerst das mittlere Element aus der Folge, dann das jeweils mittlere Element aus der rechten und linken Teilfolge einfügen, und so weiter...
Das ist schön. Sie haben mir meine Fragen vorweggenommen. [Lachen zur Beisitzerin] Kann man den Suchbaum nachträglich balancieren?
Ja, sicher! Es gibt da z.B. die AVL-Bäume. Die hatten wir zwar nicht im Skript und ich weiß nicht, wie sie genau funktionieren, aber ich habe mal mit einem Template für AVL-Bäume unter C++ gearbeitet. Die sind toll: Man stopft einfach Datensätze rein und der AVL-Baum sorgt selbst dafür, daß er balanciert bleibt.
Ja, und der Begriff dafür heißt "Rotation". Gut. Kommen wir zu anderen Bäumen. Was ist ein Heap?
Das ist ein vollständiger Baum, dessen Knoten wie beim Suchbaum Schlüsselwerte mit einer Ordnungsrelation enthalten. Bei Heaps gilt aber, daß die Wurzel immer größergleich als seine Nachfolger ist. Wenn man einen Heap von einem beliebigen Knoten aus nach oben Richtung Wurzel traversiert, erhält man die Elemente in aufsteigender Reihenfolge. Oh, es handelt sich hierbei natürlich auch um einen Binärbaum.
Ist der Heap balanciert?
Auf jeden Fall, es ist doch ein vollständiger Baum! Ein balancierter Baum hat die Eigenschaft, daß seine Höhe kleiner-gleich $\lfloor \lg_2 n \rfloor + 1$ ist. Dies ist bei einem vollständigen Baum gegeben.
Anwendung von Heaps?
Zum Beispiel Heap-Sort...
Gut gut, sprechen wir nun über was anderes...
Och, keine Priority-Queues mehr? [Grinsen]
Es gibt den Begriff "Heap" noch in einem anderen Zusammenhang...
Ja, bei dynamischer Speicherverwaltung.
Hat das irgendetwas mit den Heaps von vorhin gemeinsam?
Nein, gar nichts. Hier handelt es sich um einen Speicherbereich - eine Halde - den das Betriebssystem vorrätig hält, wenn zur Ausführungszeit eines Programms neuer Speicherplatz für dynamische Datenstrukturen wie z.B. Listen angefordert wird.
Wie geschieht das in einem Programm? In Java oder Pascal?
(Hier hatte ich ein Brett vorm Kopf. Schließlich:) Mit der Anweisung *new*.
Genau. Was steht in Konkurrenz zum Heap?
Der Stack.
Genau. Was passiert denn mit dem Stack während der Ausführung eines Programms?

Er wächst ständig auf und ab. Immer wenn ein Unterprogramm aufgerufen wird, dann werden die Parameter, die Rücksprungadresse und der ganze Schmodder auf dem Stack abgelegt. Dies ist besonders bei der Abarbeitung rekursiver Funktionen der Fall. Dann geht richtig die Post ab! [Alle grinsen] (Wir haben dann noch weiter über rekursive Funktionen gesprochen und warum der Stack besondes bei nicht-rumpfrekursiven Funktionen unverzichtbar ist. Es ging um die Ergebnisübergabe zwischen Rekursionsstufen, die rekursive Fakultätsberechnung, Entrekursivierung, akkumulierende Parameter. Kann mich aber nicht mehr an den genauen Wortlaut erinnern...)

Sprechen wir zum Schluß noch über Graphen...

Schön!

[Grinst] Was ist ein Graph?

Eine Menge von Punkten (Knoten, Ecken, Vertices. Prof. Hesse gefiel der Begriff "Knoten" am besten) und eine Menge von Kanten. Die Kanten sind hierbei eine zweistellige Relation auf die Menge der Knoten.

Wie kann man Graphen darstellen?

Da gibt es einige Möglichkeiten: z.B. als Liste von Listen, Array von Listen. Die Darstellung als Matrix ist zwar nicht besonders ökonomisch, aber dafür kann man bei dieser Darstellung die Hilfsmittel aus der Linearen Algebra anwenden.

Wie heißt die Matrix?

Die Matrix? Es gibt da mehrere, wie z.B. die Inzidenz-Matrix. Sie meinen aber sicher die Adjazenz-Matrix, denn nur diese kommt im Skript vor.

[Freut sich; glücklicher Blick zur Beisitzerin] Warum ist die Adjazenz-Matrix von Bedeutung?

Wie gesagt, man kann da ganz tolle Sachen mit machen, z.B. sie k mal mit sich selbst multiplizieren - dann erhält man alle Kantenzüge der Länge k zwischen zwei Knoten (hatte ich noch aus der Diskreten Mathematik drauf). Aber Sie wollen sicher auf Warshall's Algorithmus hinaus. [Grinst] Warshall's Algorithmus berechnet die transitive Hülle für einen Graphen. Die transitive Hülle einer zweistelligen Relation R, also der Kanten des Graphen, ist die kleinste Relation für die gilt: sie ist transitiv und R ist eine Teilmenge von ihr. Bildlich kann man sich das so vorstellen: [Zeichne einen einfachen Graphen] Es werden alle Verbindungen, nein: Pfade bzw. Wege zwischen zwei Knoten in die Matrix eingetragen. Wenn es also einen Weg zwischen zwei Knoten gibt, wird dort in der Matrix - im Falle eines ungewichteten Graphen - ein *true* stehen. Warshall's Algorithmus iteriert über die Anzahl der Zwischenknoten, die auf einem Weg zwischen zwei Knoten liegen dürfen. Deswegen darf man auch die zwei äußeren Schleifen nicht miteinander vertauschen.

Richtig. Welchen Aufwand hat der Algorithmus?

$O(n^3)$, also kubisch.

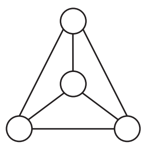
Genau, weil es drei Schleifen gibt. Es gibt da noch einen anderen Algorithmus, der die kürzesten Wege berechnet.

Ja, Dijkstra. Aber der berechnet nur die kürzesten Wege für einen einzigen Knoten!

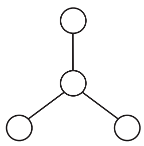
Nein, den meine ich nicht.

(Ups, da war ich etwas vorlaut.) Sorry, natürlich, Sie meinen Floyd's Algorithmus - die Erweiterung von Warshall's Algorithmus. Der berechnet für gewichtete Graphen die

kürzesten Wege zwischen allen Knoten. (Mußte dann noch auf dem Blatt erklären, wie der Algorithmus genau arbeitet.)
Wie funktioniert denn Dijkstra?
Dijkstra's Algorithmus berechnet die kürzesten Wege von einem Knoten zu allen anderen Knoten. Es ist ein Greedy-Algorithmus, d.h. er findet in jedem Schritt die lokal optimalste Lösung. Er funktioniert so: [Verdeutliche am Graphen] Es wird in jedem Schritt eine Menge S um jeweils einen Knoten erweitert. In dieser Menge S befinden sich alle Knoten, deren minimaler Weg zum Ausgangsknoten bereits bekannt ist. In jedem Schritt kommt der Knoten hinzu, der den nächstkleinsten Abstand zum Ausgangsknoten hat. Man macht solange weiter, bis alle Knoten in S liegen. Am Ende erhält man einen aufspannenden Baum, der die kürzesten Wege zum Ausgangsknoten anzeigt.
Was ist das Traveling Salesman Problem?
Wenn man einen ungewichteten Graphen betrachtet, dann stellt sich die Frage, ob es einen Zyklus gibt, der durch alle Knoten führt. Dieser Zyklus heißt Hamilton-Zyklus. Wenn man einen gewichteten Graphen betrachtet, kann man diese Frage dahingehend erweitern, welcher Hamilton-Zyklus - sofern er existiert - der kürzeste ist. Das ist das Traveling Salesman Problem. Es ist bis heute kein effizienter Algorithmus bekannt, der das beantworten kann. Alle Algorithmen erzeugen einfach alle möglichen Zyklen und wählen dann den kürzesten aus. Der Aufwand ist exponentiell.
Welche Versionen vom Traveling Salesman Problem gibt es noch?
Man kann das Traveling Salesman Problem verallgemeinern, indem man z.B. fordert bzw. zuläßt, daß jeder Knoten k-mal besucht wird. Außerdem muß es nicht immer ein Zyklus sein, es kann auch gefragt sein, ob es einen Weg zwischen zwei Knoten gibt, der durch alle anderen Knoten führt.
Wann stellt sich die Frage nach der Existenz eines solchen Zyklus?
Wenn z.B. jeder Knoten genau einmal besucht werden soll.
Können Sie mir ein Beispiel dafür geben, daß dann nicht immer ein Zyklus existiert?
[Muß überlegen. Zeichne dazu einen Graphen...]



Sie haben schon zu viele Kanten eingezeichnet.
[Alles klar. Ich zeichne einen zweiten Graphen...]



Der ist richtig. Sie können jetzt kurz draußen warten.

Zur Prüfungsatmosphäre:

Die Prüfung war sehr angenehm, obwohl sie mir zwischendurch relativ lang vorkam. Dies mag aber auch an meinen teilweise langen Monologen gelegen haben. Prof. Hesse war sehr freundlich und ließ mich immer ausreden. An einigen Stellen mußte ich ziemlich lange überlegen. Prof. Hesse hat mir stets die Zeit dafür gegeben und nicht gleich das Thema gewechselt. Wenn ich absolut nicht auf die Antwort kam, habe ich ihn freundlich darauf hingewiesen („Nun lassen Sie mich doch nicht so lange zappeln!“). Dann gab’s auch Extra-Hilfestellung.

Die Stimmung war sehr gelöst. Ich hatte den Eindruck, daß die Prüfung allen Anwesenden Spaß gemacht hat. Prof. Hesse saß gößtenteils zurückgelehnt in seinem Sessel und hörte mir zu. Meine Monologe waren zum Teil wirklich lang.

Ich bin mir nicht sicher, ob ich in diesem Protokoll alles wiedergegeben habe. Irgendwie verwischen die Erklärungen, die ich mir während der Vorbereitungen selbst vorgesprochen habe mit meinen Antworten aus der Prüfung. So könnte es auch sein, daß ich etwas zu objektorientierten Sprachen erzählt habe, aber ich glaube eher nicht.

Insgesamt eine sehr schöne Prüfung.