Gedächtnisprotokoll Diplomprüfung Theoretische Informatik

Prüfling: Michael Holtmann (Michael.Holtmann@rwth-aachen.de)
Prüfer: Prof. em. Dr. Klaus Indermark
Beisitz und Protokollführung: Volker Stolz
Datum: 9.11.2005, 9:00 Uhr
Geprüfte Fächer:

1. Compilerbau (Vorlesung, Sommersemester 2005)
2. Logikprogrammierung (Vorlesung, Wintersemester 2003/2004)
3. Effiziente Algorithmen (Buch „Algorithmics for Hard Problems“, zweite korrigierte Auflage)

Dauer: ca. 45 Minuten
Note: 1.0

1.) Compilerbau
KI: Wenn Sie an eine Lex-Spezifikation und reguläre Definitionen denken. Da können wir ja im i-ten Ausdruck nur die Ausdrücke mit Index 1 bis $i-1$ verwenden. Dann kann man das ja entschachteln...was bedeutet das denn für die Ausdrucksstärke?
Ich: Dass man den Bereich der kontextfreien Sprachen nicht verlässt. Wenn wir Rekursion hätten, würden wir in den Bereich der EBNF kommen.
KI: Wie könnten wir das denn entschachteln? Ich denke an so etwas wie eine „linke Seite“ und einen „Pfeil“ statt dem Gleichheitszeichen...
Ich: Das können wir dann durch eine kontextfreie Grammatik darstellen.
KI: Geht das so einfach oder muss man da was beachten?
Ich: (weiß nicht weiter)...Wir dürfen nicht den Sternoperator auf der rechten Regelseite verwenden, sonst verlassen wir die kontextfreien Sprachen.
KI: Hmmm...wenn wir jetzt eine rechte Seite mit Stern haben...können wir das dann trotzdem mit einer kontextfreien Grammatik darstellen?
Ich: (überlege ziemlich lang rum und komme nicht so recht weiter. Nach einiger Hilfe vom Prof dämmert es mir dann.) Achja, das können wir mit einer Rechtsrekursion machen. $A \to \alpha^\star$ wird übersetzt in $A \to \alpha A|\epsilon$.
KI: Sehen Sie...und geht das denn einfach so oder müssen wir da was Spezielles beachten? (Hinterher stellt sich raus, dass es diese Frage war, auf die Professor Indermark schon zu Anfang hinauswollte.)

Ich: (weiß schon wieder nicht weiter und behaupte dann sogar noch, dass wir den Bereich der kontextfreien Sprachen doch verlassen, wenn wir den Sternoperator zulassen.)
KI: (Daraufhin stellt der Prof das dann richtig.) Nein! Wir bleiben im kontextfreien Bereich. Aber bei Einführung der Rechtsrekursion müssen wir das mit einem neuen Nichtterminal von den anderen A-Regeln trennen. Sonst „vermischen" wir Regelalternativen von A, was natürlich nicht geht, wenn wir $A \to \alpha^\star$ anwenden. (Also: Den Stern in $A \to \alpha^\star|\beta$ können wir nicht einfach durch Einführung einer Rechtsrekursion eliminieren. $A \to \alpha A|\epsilon|\beta$ ist also falsch, weil wir dann auch z.B. $\alpha^{10}\beta$ von A ableiten können. Richtig wäre: $A \to A'|\beta, A' \to \alpha A'|\epsilon$.)
KI: Gut. Machen wir mal weiter. Was berechnet denn eigentlich der Backtrack-Automat?
Ich: Der berechnet eine „first longest match"-Analyse.
KI: Was ist denn die Komplexität und warum? Haben Sie das Beispiel aus der Vorlesung noch im Kopf?
Ich: Die Methode hat im worst case quadratische Komplexität. Das liegt daran, dass der Automat u.U. bis zum Wortende läuft, dann merkt, dass kein längerer match existiert und wieder zurück zum ersten gefundenen match läuft. (Ich gebe das Beispiel aus der Vorlesung an: $\alpha_1 = abc$, $\alpha_2 = (abc)^\star d$.) Für das Wort $w = (abc)^m$ mit beliebigem m erkennt der Automat erst ganz am Ende, dass das für α_2 erforderliche d nicht mehr kommt. Dann läuft er wieder ganz zurück.
KI: Es geht aber auch in Linearzeit, indem man sich merkt, was man bei der Suche nach einem längeren match so alles gesehen hat. Machen wir jetzt mal syntaktische Analyse. Welche Verfahren haben wir denn da kennengelernt?
Ich: Das waren zwei Verfahren, zum einen Die Top-Down-Analyse, die eine Linksanalyse berechnet und die Bottom-Up-Analyse, die eine gespiegelte Rechtsanalyse berechnet.
KI: In welcher Zeit kann man denn kontextfreie Grammatiken erkennen?
Ich: (denke zu kompliziert)...also, LL(1) kann man in Linearzeit erkennen...
KI: Ganz allgemein...ich denke an die Grundvorlesung über Automatentheorie.
Ich: Ach ja, der CYK-Algorithmus. Der läuft in $O(n^3)$. Man läuft über die Indices der Buchstaben des Wortes.
KI: Gibt es denn Probleme bei mehrdeutigen Grammatiken? Wann heißt eine Grammatik überhaupt mehrdeutig?
Ich: Wenn sie nicht eindeutig ist und eindeutig ist sie, wenn zu jedem Wort aus der Sprache genau ein Ableitungsbaum existiert, was auch gleichbedeutend mit der Existenz genau einer Linksanalyse bzw. Rechtsanalyse ist.
KI: Kann man denn mehrdeutige Grammatiken trotzdem behandeln?

Ich: Ja, dann muss man aber Präzedenz-Regeln für die Operatoren haben, z.B. $*$ vor $+$.
KI: Gut. Kommen wir mal zur Bottom-Up-Analyse. Nehmen wir mal an, wir haben die $LR(0)$-Mengen berechnet und es treten Konflikte auf, die wir auch mit $SLR(1)$ nicht beseitigen können, also versuchen wir es mit $LR(1)$. Definieren Sie mir doch erstmal die $LR(1)$-Auskünfte.
Ich: (Definition mit lookahead-Symbol a hingeschrieben, ϵ-Fall nur mündlich)
KI: Was muss denn gelten, damit eine Grammatik $LR(1)$ ist?
Ich: Es dürfen keine Shift/Red.-Konflikte und keine Red./Red.-Konflikte auftreten.
KI: Sagen Sie mal nicht „keine Konflikte". Formulieren Sie es positiv. Was muss gelten?
Ich: Die lookahead-Symbole in der zweiten Komponente zu verschiedenen Reduce-Auskünften müssen verschieden sein. . . und diese müssen wiederum verschieden sein zu den Shift-Elementen hinter dem Punkt im Kern der Shift-Auskünfte. Bei den Reduce-Auskünften ist also das Symbol in der zweiten Komponente wichtig, bei den Shift-Auskünften nur das, hinter dem Punkt im Kern der Auskunft.
KI: Richtig. Aber wieso brauchen wir es dann überhaupt?
Ich: Wenn wir Auskünfte für verlängerte Keller berechnen, wandert der Punkt nach rechts und wir erhalten irgendwann eine Reduce-Auskunft, wenn der Punkt am Ende angekommen ist. Dann ist das Symbol wieder wichtig.
KI: Gehen wir mal zur semantischen Analyse. Da können ja Zirukularitäten auftreten. Wann nennen wir denn eine Attributgrammatik zirkulär?
Ich: Eine Attributgrammatik heißt zirkulär, wenn es einen Ableitungsbaum gibt, in dessen zugehörigem Abhängigkeitsgraphen eine aktuelle Attributvariable von sich selbst abhängt.
KI: Ja aber. . . es gibt doch unendlich viele Ableitungsbäume. Kann man das denn trotzdem entscheiden?
Ich: Ja kann man, denn die unendlich vielen Ableitungsbäume setzen sich nur aus endlich vielen Teilbäumen zusammen. Zirkularitäten treten also schon in beschränkter Tiefe auf.
KI: Wie kann man denn die Zirkularitäten feststellen?
Ich: Man berechnet zunächst die Unterhalb-Abhängigkeiten. Das sind dann Abhängigkeiten synthetischer von inheriten Attributen. Dazu muss man noch eine sogenannte Deckel-Regel finden. Da stecken die Unterhalb-Abhängigkeiten in den Nichtterminalen der Unterseite und die Deckel sind dann Abhängigkeiten inheriter von synthetischen Attributen, wodurch dann Zyklen entstehen.
KI: Bei der induktiven Berechnung betrachten wir ja die einzelnen Ablei-

tungsbäume getrennt. Was passiert denn sonst?
Ich: Da kriegen wir evtl. Zyklen aus Abhängigkeiten aus verschiedenen Ableitungsbäumen, die in Wirklichkeit gar nicht auftreten können. (Dass man damit den Sprung zum Polynomiellen schafft, habe ich nicht mehr erwähnt. Prof. Indermark merkte aber schon, dass ich das Thema drauf hatte und machte deswegen noch ein paar Bemerkungen seinerseits über die Arbeiten von Knuth und dass er seinen „Fehler“ ja bereits in seiner ersten Arbeit über dieses Thema berichtigt hat.)
KI: Gut. . .dann noch zur Codegenerierung. Da hatten wir ja eine Programmiersprache mit rekursiven Prozeduren mit und ohne Parameter behandelt. Bei der Variante ohne Paramter hatten wir viel in den $CALL$-Befehl reingesteckt. Aber können Sie mir erklären, wie das im Fall mit Parametern funktioniert? Wie sieht ein Prozedur-Aufruf da aus?
Ich: (male den Stack mit der Einteilung der Speicherzellen auf) Also, zunächst werden die aktuellen Parameter berechnet. Dann wird mit Hilfe des Indexregisters der statische Verweis berechnet. Danach wird die Rücksprung-Adresse eingetragen und zur aufgerufenen Prozedur gesprungen. Diese drei Aufgaben übernimmt die aufrufende Prozedur. Dann übernimmt die aufgerufene Prozedur. Sie speichert noch den alten Framepointer als Dynamic Link auf dem Stack und stellt abschließend noch den Platz für die lokalen Variablen bereit.
KI: Ja, genau. Sie haben es richtig gesagt. Der statische Verweis wird mit Hilfe des Indexregisters berechnet.
Ich: (schreibe auf: $R, < R + 2 >$) Es wird die statische Verweiskette entlang gelaufen.
KI: Gut. Dann gehen wir mal zur Logikprogrammierung. . .

2.) Logikprogrammierung
KI: In der Logikprogrammierung betrachtet man ja eine spezielle Logik. Welche ist das denn?
Ich: Das ist die Horn-Logik.
KI: Was ist denn genau die Horn-Logik?
Ich: Da betrachten wir eben nur Horn-Formeln. Eine Formel ist eine Horn-Formel, wenn sie in konjunktiver Normalform vorliegt und jede Klausel höchstens ein positives Literal enthält.
KI: Wie kann man so eine Formel als gewöhnliche FO-Formel schreiben?
Ich: Die Literale der einzelnen Klauseln werden durch $\vee$ verknüpft und die einzelnen Klauseln werden durch $\wedge$ verknüpft. Die Variablen werden allquantifiziert.
KI: Wenn wir jetzt eine beliebige FO-Formel haben, also in der Logik erster Stufe. . .können wir denn dann immer eine solche Formel erhalten?
Ich: Die Übersetzung in KNF geht immer, aber das mit dem nur einen po-

sitiven Literal geht nicht immer. Die Horn-Logik stellt also eine echte Einschränkung dar.
KI: Dann geht ja die Übersetzung einer beliebigen FO-Formel in Skolem-Normalform immer. Wie geht diese Übersetzung denn?
Ich: Also, zunächst müssen die inneren Quantoren rausgezogen werden. (Er hakt nach, wie das denn genau geht) Dabei muss darauf geachtet werden, dass Quantoren, die über freie Variablen gezogen werden, diese Variablen danach nicht binden...(ich schreibe folgenes auf: $x \wedge \exists xpx$)...hier zum Beispiel. Da muss das gebundene x erst umbenannt werden.
KI: Wie läuft denn nun die Skolemisierung ab?
Ich: Die Existenzquantoren werden eliminiert. Dazu werden für die entsprechenden existenzquantifizierten Variablen neue Funktionssymbole eingeführt. (ich schreibe auf: $\forall x_1 \ldots \forall x_k \exists x_{k+1}$) Hier ist natürlich klar. Die Instanziierung von x_{k+1} hängt von den Werten von $x_1, \ldots, x_k$ ab. Deswegen kann man den Wert von x_{k+1} als den Funktionswert einer k-stelligen Funktion mit den Argumenten $x_1, \ldots, x_k$ auffassen. Man führt also ein neues Funktionssymbol f ein und ersetzt im hinteren Teil der Formel alle Vorkommen von x_{k+1} durch $fx_1 \ldots x_k$. Dann werden noch die freien Variablen durch neue Konstantensymbole ersetzt.
KI: Wenn Sie an die Prozedurale Semantik denken. Wie läuft die denn ab?
Ich: Das ist im Grunde genommen einfach die Resolution...
KI: (unterbricht) Ok. Wenn Sie die Resolution schon ansprechen...können Sie die mal definieren?
Ich: (schreibe die Definition ausführlich wie im Skript auf)
KI: Ja, ok (hat ihm etwas zu lange gedauert). Wie geht jetzt die Prozedurale Semantik?
Ich: Da werden Rechenschritte gemacht. Wir haben Konfigurationen, die ineinander übergehen. Wir haben das Paar (G_1, s_1). Das geht über in das Paar $(G_2, s_2) \ldots$
KI: (unterbricht) Lassen Sie mal die Substitution weg. Betrachten wir mal nur das G. Wenn wir mal die lineare Resolution betrachten. Können Sie die mal defineren? Aber nicht wieder so wie eben, mündlich reicht.
Ich: (liefere die Definition mündlich)
KI: Richtig. Die erste Resolutionsklausel ist eindetuig festgelegt, aber die zweite kann auch aus der ursprünglichen Klauselmenge kommen. Dann gibt es ja auch noch die SLD-Resolution. Die Vollständigkeit der SLD-Resolution folgt für die Horn-Logik ja aus der Vollständigkeit der linearen Resolution. Was ist denn bei der SLD-Resolution anders?
Ich: Da beginnt die Resolution mit einer negativen Klausel.
KI: Und wie sieht der Resolvent dann aus?
Ich: Achso, ja...das ist wieder eine negative Klausel.

KI: Und woher kommen dann die Klauseln für die Resolution?
Ich: Die eine ist ja der vorherige Resolvent also auch negativ, aber die andere ist definit.
KI: Richtig. Es wird also insbesondere nicht auf Resolventen vor dem letzten zurückgegriffen. Und bei der Prozeduralen Semantik...wie sieht da die Resolution aus?
Ich: Das ist eine binäre Resolution. Ein Literal der Zielklausel wird mit einer Programmklausel resolviert.
KI: Richtig. Und wie sieht dann der SLD-Baum aus?
Ich: (Definition mündlich geliefert)
KI: Kanonisch?
Ich: Ja, das bedeutet, dass mit dem ersten Literal der Zielklausel resolviert wird.
KI: Wenn Sie jetzt daran denken, wie so ein Baum aussieht, z.B. in Datalog.
Ich: Da kann es unendliche Pfade geben. Die kann man in Datalog aber erkennen.
KI: Wie ist das denn genau?
Ich: Also, es gibt nur endlich viele Konstanten und keine ein- oder mehrstelligen Funktionen. Deswegen gibt es im SLD-Baum nur endlich viele Teilziele. Dann müssen sich nach endlich vielen Schritten Teilziele oder Folgen von Teilzielen wiederholen.
KI: Ja, ich habe das in der Vorlesung immer so behandelt. Aber wie verändert sich denn die Anzahl der Literale?
Ich: Naja, wenn wir mit einem Fakt resolvieren, wird die Anzahl der Literale um eins kleiner, sonst bleiben es gleich viele oder werden evtl. sogar mehr als vorher.
KI: (Daraufhin sagt er noch einiges dazu, wie das genauer aussieht. Ich hatte mir dazu auch ein paar Gedanken gemacht und versuchte es einfach mal...)
Ich: Ja, also geht das nicht so: Wenn man irgendwann eine Folge von Teilzielen hat und man hinterher auf einem Pfad nochmal zu einem Zustand kommt, in dem diese Folge als Teilfolge mit noch etwas davor wieder auftritt...dann weiß man doch, dass der Pfad unendlich ist...
KI: (nimmt das nicht so recht an) Sie müssen das Kellerprinzip beachten. Darauf will ich hinaus. Das oberste Literal wird resolviert und vom Keller runtergenommen und andere Literale stattdessen draufgelegt. Gut...dann noch Folgendes. Wir hatten zum Ende der Vorlesung ja noch die Parserkonstruktion mit Hilfe von Prolog betrachtet. Wo kam das denn vor?
Ich: Bei den Definite Clause Grammars.
KI: Genau. Dazu wurden den Nichtterminalen einer kontextfreien Grammatik Prädikate zugeordnet. Wie haben wir das denn in Prolog umgesetzt?
Ich: Wir haben den Nichtterminalen Ableitungsprädikate zugeordnet. So ein

Prädikat ist erfüllt, wenn das Wort aus dem Nichtterminal, zu dem das Prädikat gehört, ableitbar ist. Das sind dann also erstmal einstellige Prädikate; Differenzlisten spielen da noch keine Rolle.
KI: Was ist denn da problematisch?
Ich: Naja, wenn wir so eine Regel nach Prolog übersetzen, dann können natürlich auf der rechten Seite mehrere Nichtterminale auftreten. Deswegen müssen wir dann unter Umständen mehrmals append aufrufen.
KI: Und das haben wir wie gelöst?
Ich: Wir haben zweistellige Prädikate eingeführt und uns Differenzlisten bedient.
KI: Beschreiben Sie mir mal genau, wann so ein Prädikat erfüllt ist.
Ich: (schreibe auf: $a(u, v)$) Das ist erfüllt, wenn v ein Suffix von u ist (schreibe auf: $u = wv$) und...(will es gerade sagen)
KI: Halt, da fehlt ja noch was.
Ich: (schreibe auf: $A \Rightarrow^\star w$)...ja, w muss aus A ableitbar sein.
KI: Das reicht dann. Kommen wir noch zu den Effizienten Algorithmen.

3.) Effiziente Algorithmen
KI: Nun...es gibt ja Probleme, bei denen es ziemlich schwierig ist, sie zu lösen. Die kann man nicht effizient lösen. Was heißt das überhaupt...„effizient“ lösen?
Ich: In Polynomzeit.
KI: Ja genau, in Polynomzeit. Wenn Sie an die besondere Klasse der NP-schweren Probleme denken...dazu gehören ja zum Beispiel SAT oder TSP. (So eröffnet er den Effiziente-Teil eigentlich immer und bei der Durchsicht früherer Prüfungsprotokolle war mir insbesondere diese Äußerung über SAT und TSP mehrmals aufgefallen. Dabei wirft er Entscheidungsprobleme und Optimierungsprobleme zusammen. Deswegen hatte ich mir für den Fall der Fälle einen Einwand überlegt...)
Ich: Ja, aber man sollte schon erwähnen, dass SAT ein Entscheidungsproblem ist und TSP ein Optimierungsproblem.
KI: Sehr richtig!
Ich: Die NP-Schwere von Optimierungsproblemen wird ja definiert über die NP-Schwere der zugehöreigen Schwellenwert-Sprache. (Das war sicher keine weltbewegende Sache und Professor Indermark ging auch nicht weiter darauf ein, was eine Schwellenwertsprache ist, aber da kann man schon zeigen, dass man den Stoff auch detailliert beherrscht; die Äußerung konnte also sicher nicht schaden.)
KI: Sagen Sie doch mal. Was sind denn P und NP?
Ich: P ist die Menge aller Sprachen, die von einer deterministischen Touringmaschine in polynomieller Zeit erkannt werden können. NP ist die Men-

ge aller Sprachen, die von einer nichtdeterministischen Touringmaschine in polynomieller Zeit erkannt werden können. Wobei, wenn man da den Berechnungsbaum betrachtet, dann ist die Länge der kürzesten Berechnung die Zeit, die für ein Wort gebraucht wird.
KI: Und wie sieht das mit der Zeitkomplexität aus, wenn man eine Sprache aus NP deterministisch erkennen wollte?
Ich: Dann wird das exponentiell.
KI: Wieso?
Ich: Naja, wir können den Nichtdeterminismus deterministisch simulieren als Lauf durch den Berechnungsbaum mit Backtracking. Der Baum hat aber exponentiell viele Blätter.
KI: Richtig. Wenn wir jetzt zum Beispiel TSP betrachten. Beschreiben Sie mir das doch mal.
Ich: Gegeben ist ein vollständiger Graph und eine Kostenfunktion, die jeder Kante des Graphen Kosten zuordnet. Das TSP-Problem ist nun, einen Hamiltonschen Kreis zu finden, der minimale Kosten hat. Ein Hamiltonscher Kreis ist ein Kreis, der jeden Knoten genau einmal besucht.
KI: Richtig. Wie kann man denn das TSP-Problem lösen?
Ich: (Da ich mich sehr intensiv mit TSP beschäftigt hatte, sprach ich einige Lösungs-Ansätze an, obwohl ich schon ahnte, dass er auf Backtracking hinauswollte.) Ja, da haben wir ne ganze Menge kennengelernt. Zum einen haben wir da Approximationsalgorithmen behandelt, die polynomielle Zeit brauchen; zum Beispiel den Algorithmus von Christofides. Das ist eine $\frac{3}{2}$-Approximation. Aber das TSP kann man auch mit Backtracking lösen. (Vorsicht: hier habe ich glatt vergessen zu sagen, dass der Algorithmus von Christofides nur für $\triangle$-TSP funktioniert.) Man kann das dann auch noch zu Branch&Bound verbessern, um Zweige im Suchbaum abzuschneiden.
KI: Wie sieht denn Backtracking für TSP aus? Wie läuft das ab?
Ich: Naja, wir haben einen Baum, in dem man nach der Zugehörigkeit von Kanten zur Lösung verzweigt.
KI: Man braucht also eine Struktur in der Menge der Lösungen.
Ich: Ja, genau. (male einen Knoten mit zwei Kanten auf und schreibe an die eine Kante e_{12} und an die andere $\overline{e_{12}}$, um noch etwas genauer auf die Struktur des Baumes einzugehen) Bei TSP gibt es da noch eine Einschränkung. Wenn wir jetzt z.B. einen Graphen mit zehn Knoten haben und sowohl die Kante e_{12} als auch die Kante e_{23} gehört zur Lösung, dann schließt das die Zugehörigkeit der Kante e_{31} zur Lösung aus. Das heißt, die Menge der zulässigen Lösungen ist kleiner. Das ist ja bei SAT anders. Da ist jede Belegung der Variablen zunächst mal zulässig. Das heißt, der Lösungsbaum hat eine einfachere Struktur. (Wie schon oben erwähnt, habe ich an dieser Stelle in meinen Ausführungen öfter Dinge vorweggenommen, die mir in Prüfungspro-

tokollen aufgefallen waren)
KI: Wo Sie schon SAT ansprechen. Da gibt es ja auch noch das Optimierungsproblem $Max\text{-}SAT$. Definieren Sie mir das mal.
Ich: Da versucht man eine Belegung zu finden, die möglichst viele Klauseln der gegebenen Formel wahr macht.
KI: Richtig. Erklären Sie mir doch mal das Prinzip der Dynamischen Programmierung.
Ich: Also, die Dynamische Programmierung funktioniert Bottom-Up im Unterschied zu Divide&Conquer. Man beginnt mit der Berechnung der Lösungen zu den elementaren Problemen und setzt aus diesen dann Lösungen zu größeren Problemen zusammen. Das macht man so lange, bis man die Lösung zu dem ursprünglichen Problem hat. Alle Teillösungen, die man berechnet, werden dabei gespeichert, zum Beispiel in einer Tabelle. Das heißt, man muss jede Teillösung nur einmal berechnen und kann auf das gespeicherte Ergebnis, wenn notwendig, zurückgreifen. Das ist auch der wesentliche Vorteil gegenüber Divide&Conquer, wo man das gleiche Teilproblem evtl. sehr oft lösen muss, z.B. bei Fibonbacci.
KI: Dann gibt es ja auch noch das Rucksackproblem. Was ist das denn?
Ich: Also, da muss man zwei Varianten unterscheiden. Beim einfachen Rucksackproblem hat man n Objekte mit Kosten $w_1, \ldots, w_n$ und das Ziel ist es, Objekte mit möglichst hohen Gesamtkosten in den Rucksack reinzupacken, ohne dabei jedoch die Kapazität des Rucksacks zu übersteigen. Beim „normalen" Rucksackproblem hat man nochmal n Werte $c_1, \ldots, c_n$ extra. Wieder darf die Summe der w_i die Kapazität des Rucksacks nicht übersteigen. Das Ziel ist es dann aber, die Summe der zugehörigen c_i zu maximieren.
KI: Ok. Wie kann man das denn mit Dynamischer Programmierung lösen?
Ich: Naja, man lässt immer mehr Objekte zu und speichert alle möglichen Mengen, die die Kapazität des Rucksacks nicht übersteigen.
KI: (schaut mich an)...wie genau?
Ich: Ach,...Sie wollen die Tupel sehen?!?
KI: (grinst) Ja, genau. Die würde ich gerne sehen...
Ich: Kein Problem. (schreibe hin: $(0, 0, 0)$,$(c_1, w_1, 1)$). Ja, also das ist die Initialisierung; das leere Tupel und das, wo nur das erste Objekt drin ist. Dann lässt man schrittweise für jeden weiteren Index i das i-te Tupel zu und berechnet in einer While-Schleife alle möglichen neuen Tupel, das heißt man prüft, ob das hinzugenommene Tupel zur jeweiligen Lösung hinzugenommen werden kann, ohne die Kapazität des Rucksacks zu übersteigen. Halt...hier stimmt was nicht! (ich merke, das ich die Anfangstupel falsch hingeschrieben habe. Ich berichtige das und es ergeben sich $(0, 0, \emptyset)$ und $(c_1, w_1, \{1\})$)...das sind ja Indexmengen hier hinten. So, jetzt stimmt's...vor dem nächsten Schleifendurchlauf dünnt man die Menge der Tupel noch etwas aus und rechnet

nur mit der dünneren Menge weiter. Von den Tupeln, die dieselben Kosten der c_i haben, betrachtet man nur dasjenige Tupel weiter, welches die geringsten Kosten der w_i hat. (Dass wir diesen Algorithmus im Zusammenhang mit pseudo-polynomiellen Algorithmen kennengelernt haben und er eine Komplexität von $O(|I|^2 * Max - Int(I))$ hat, fällt mir zwar sofort ein, ich erwähne es aber nicht mehr.)
KI: Schauen wir doch nochmal auf Divide&Conquer und Fibonacci. Sie haben ja schon gesagt, dass das sehr schlecht dafür ist. Wenn wir uns die Komplexität genauer anschauen und mal das Master-Theorem betrachten. Wie sieht das denn da aus?
Ich: (UFFF!!!!)...also...ääh...(ich versuche erstmal, das Master-Theorem allgemein aufzuschreiben, was mir aber nicht so richtig gelingt. Nur folgendes: $T(n) := \dots \lceil \frac{n}{k} \rceil \dots$ an der Stelle fiel mir noch ein, dass hinten noch was addiert werden müsste, für das Zerlegen in Teilprobleme bzw. das Zusammensetzen der Teillösungen...und dann wäre ich gar nicht mehr so weit von der Lösung weggewesen...)
KI: (bemerkt meine Verunsicherung und fügt Folgendes hinzu) Wissen Sie, bei Fibonacci kann man das Master-Theorem gar nicht anwenden. Aber warum?
Ich: (rätsle einige Zeit rum)...sind Sie sicher, dass das im Buch war? (so eine Frage sollte man besser nicht stellen, glaube ich...meine Schuld. Auch nach längerem Hinundherüberlegen komme ich nicht drauf.)
KI: Doch doch, das ist im Buch...(wartet noch etwas und verrät die Lösung dann) Ja, also, um das Master-Theorem anzuwenden braucht man eine besondere Zerlegung des Problems in Teilprobleme. Die liegt aber bei Fibonacci nicht vor. (So oder so ähnlich hat er es gesagt. Ich kann mich leider nicht mehr an seinen genauen Wortlaut erinnern; aber ich denke es ist klar, was gemeint ist.) Gut, Herr Holtmann. Ich denke, damit sind wir dann durch. Warten Sie doch bitte eine Minute draußen...

So in etwa ist meine Prüfung abgelaufen. An alle Details konnte ich mich nicht mehr erinnern, insbesondere nicht an die Reihenfolge. Die Prüfungsatmosphäre war sehr angenehm. Professor Indermark gibt gerne Hilfestellungen, wenn man nicht weiter weiß. Man sollte sich auf keinen Fall von kleinen Schwächen verunsichern lassen. Es ist nicht das Schlechteste, erstmal kurz über eine Frage nachzudenken, bevor man antwortet. Man sollte sich aber immer möglichst präzise ausdrücken. Der Anfang und das Ende meiner Prüfung waren ja nicht so optimal, aber man muss nicht alles wissen...auch nicht für ne 1,0. Das hat er nach der Prüfung auch zu mir gesagt. Das Gesamtverständnis ist entscheidend! Trotzdem sollte man wichtige Definitionen, Sätze, etc. auf jeden Fall auswendig parat haben.

Verwendete Materialien:

1. Compilerbau
   Die Vorlesung habe ich im Sommersemester 2005 gehört und meine Mitschrift mit den Ausführungen aus den Vorlesungsvideos, den Folien, den Übungsaufgaben und den Prüfungsprotokollen zu einem kommentierten Skript erweitert. Die Übungsvideos habe ich mir nur zu speziellen Aufgaben angesehen. Ich habe dann mit dem kompletten, kommentierten Skript gelernt (also ohne Zusammenfassungen), um auch die Details draufzukriegen.

2. Logikprogrammierung
   Die Vorlesung habe ich nie gehört. Das Skript zum Wintersemester 2003/2004 gibt es auf s-inf.de, aber Vorsicht...da fehlen Vorlesungen. Auch hier bin ich lernmäßig wie in Compilerbau vorgegangen (nur Vorlesungsvideos gibt's keine).

3. Effiziente Algorithmen
   Ich habe im Wintersemester 2004/2005 die Vorlesung bei Vöcking gehört, wo aber im Wesentlichen nur die Grundlagen identisch sind. Da meiner Meinung nach ein gutes Buch immer besser ist als eine Vorlesung und ich natürlich auch damit spekuliert habe, dass Professor Indermark das nicht ganz so intensiv prüft, habe ich mich entscheiden, die Hromkovic-Version prüfen zu lassen. Ich habe das komplette Buch gelesen, wobei ich nicht die korrigierte sondern die normale zweite Auflage habe. Die Inhalte sind aber wohl weitestgehend identisch. In Effiziente Algorithmen habe ich zuletzt von einer 9-seitigen Zusammenfassung gelernt. Ich habe bei meiner Prüfung zufällig mitbekommen, dass Professor Indermark diese Vorlesung in zukünftigen Prüfungen wohl nicht mehr mitprüft...die Information ist hier natürlich ohne Gewähr!

Ich hoffe, das Protokoll hilft Euch weiter. Viel Erfolg!!!!