Prüfungsprotokoll
INF 2 (DS & GTI)

20. Januar 2000

Prüfer: Paul Fischer
Datum: 20. Januar 2000
Zeit: 9.00 Uhr
Note: 1.3
Literatur:

1. Datenstrukturen: Skript „Datenstrukturen WS 1996/97“ von Ingo Wegener
2. GTI: „Theoretische Informatik“ von Ingo Wegener; „Kompendium Theoretische Informatik - eine Ideensammlung“ von Ingo Wegener

Prüfer: Womit möchtest du anfangen?
Ich: Entgegen allen Empfehlungen mit Datenstrukturen.

Prüfer: Dann fangen wir heute mal mit Sortieralgorithmen an. Welche sind dir denn so geläufig?
Ich: Ich kenne Insertion Sort, Quick Sort, Heap Sort, Merge Sort und Bucket Sort.

Prüfer: Dann erkläre mir doch mal Heap Sort.
Ich: Heap Sort arbeitet auf einem Array, welches als binärer Baum interpretiert wird.

Prüfer: Wie geht denn das?
Ich: Wenn ein Datum an Position i steht, stehen seine Kinder, sofern vorhanden, an Position $2i$ und $2i+1$. Ein Array heißt Heap, wenn für jeden Knoten die Daten seiner Kinder größer sind als das Datum des Knotens.

Prüfer: Gut. Jetzt kann ich ja nicht davon ausgehen, daß das Array, welches ich bekomme, schon ein Heap ist.
Ich: Heap Sort arbeitet in zwei Phasen: Der Heap Creation Phase und der Selection Phase. In der Heap Creation Phase wird aus dem Array ein Heap gemacht.

Für Blätter ist die Heap-Bedingung trivialerweise erfüllt. Die Hälfte aller Daten sind Blätter, also muß ich in der Reihenfolge $i = \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, ..., 1$ die Heap-Bedingung an den Knoten herstellen. Dies mache ich mit der Prozedur *reheap*.

Die Prozedur *reheap* bekommt als zwei Parameter i und m, wobei i der aktuell betrachtete Knoten ist und m die Größe des betrachteten Arrays. Für $i > \frac{m}{2}$ ist alles in Ordnung, da ich in diesem Fall an einem Blatt bin. Für $i = \frac{m}{2}$ ist ein Vergleich nötig, da der Knoten i genau ein Kind hat. Die Daten werden eventuell getauscht und alles ist in Ordnung. Für $i < \frac{m}{2}$ hat der betrachtete Knoten zwei Kinder, die ich miteinander vergleiche. Das Datum an Knoten i vergleiche ich mit dem kleineren der beiden Kinder und vertausche es eventuell. Dann rufe ich die Prozedur *reheap(j,m)* auf, wenn ich getauscht habe und j das kleinere Kind ist.

In der Selection Phase vertausche ich das Datum an der Wurzel mit dem letzten Datum im Array und betrachte nur noch das Array der Größe $m-1$, wenn das Array vorher Größe m hatte. An der Wurzel ist die Heap Bedingung eventuell verletzt, also lasse ich das Datum mit der Prozedur *reheap* einsinken.

Prüfer: Gut, dann machen wir mal eine Laufzeit-Analyse. In der Heap Creation Phase laufe ich höchstens n mal den kompletten Baum runter, ebenso in der Selection Phase. Der Baum ist höchstens $\log_2 n$ tief, also macht das insgesamt $2n \log_2 n$ Aufrufe der Prozedur *reheap*.
Ich: Nein. In der Heap Creation Phase wird *reheap* höchstens n mal aufgerufen. $\lfloor \frac{n}{2} \rfloor$ Knoten sind Wurzeln von Bäumen deren Tiefe **mindestens** 1 ist. $\lfloor \frac{n}{4} \rfloor$ sind Wurzeln von Bäumen deren Tiefe mindestens 2 beträgt. Allgemein sind $\lfloor \frac{n}{2^d} \rfloor$ Knoten Wurzeln von Bäumen, deren Tiefe mindestens d beträgt. Wenn ich diese Anzahlen bis zur Tiefe des Baumes aufsummiere, zähle ich also z.B. die Wurzel d mal und erhalte also die Gesamtsumme aller Tiefen. (Die Formel wollte er schon garnicht mehr sehen.)

Prüfer: Gibt es auch Fälle, in denen man lieber Heap Sort statt Quick Sort benutzt, oder anders gesagt, welche Vorteile hat Heap Sort gegenüber Quick Sort?
Ich: (Ich dachte erst, er wollte auf den Kruskal hinaus, wo man die Kanten lieber mit Heap Sort sortieren kann, da man dann unter Umständen nicht alle Kanten komplett sortieren muß, dann stieß er mich jedoch mit der Nase auf die worst case-Laufzeiten.) Achja. Quick Sort hat einen worst case von $O(n^2)$, wogegen der Standard Heap Sort einen worst case von $O(n \log_2 n)$ hat.

Prüfer: Gut. Kommen wir zu etwas anderem. Angenommen ich habe einen Graph $G = (V, E)$ und eine Funktion $c : E \to \mathbb{R}^+$, die jeder Kante Kosten zuweist. Wie finde ich die günstigsten Wege zwischen allen Knoten?
Ich: Hier benutzt man das Prinzip der dynamischen Programmierung, welches man meist benutzt, wenn man eigentlich top-down, also z.B. mit einen Divide-And-Conquer Algorithmus, vorgehen könnte, nicht jedoch weiß, wo man das Problem teilen muß. Alle Zerlegungen auszuprobieren verbietet sich von selbst, da man dann exponentielle Kosten bekommt. Soll ich das mit den exponentiellen Kosten eben zeigen?

Prüfer: Nein. Zeige mal, wie der Algorithmus funktioniert.
Ich: (Ich habe einen Graphen gezeichnet und dann gezeigt, daß wenn ich einen optimalen Weg zwischen zwei Knoten i und j habe, dann auch die Teilwege dieses Weges optimal sein müssen, da man sonst einen billigeren Weg konstruieren könnte.) Die Bellmann'sche Optimalitätsgleichung hierfür lautet:

$$w_{ij}^{l+1} = \min\{w_{ij}^{l}, w_{i,l+1}^{l} + w_{l+1,j}^{l}\}$$

Entweder man benutzt einen Knoten als Zwischenknoten oder nicht. Zur Berechnung eines Matrixeintrags brauchen wir 2 Operationen (Addition und Vergleich). Also für eine Matrix insgesamt $2n^2$ Operationen. Da wir n verschiedene Matrizen berechnen, kommen wir auf insgesamt $2n^2 \cdot n = O(n^3)$.

Prüfer: Du hast ja gerade den Kruskal erwähnt. Das ist ja ein Greedy Algorithmus und die sind ja nicht immer optimal. Kennst du Beispiele für Greedy Algorithmen, die nicht immer optimale Lösungen berechnen?
Ich: Ja, das Münzwechselproblem und das Rucksack-Problem (KP).

Prüfer: Zeige mir mal das Münzwechselproblem.
Ich: (Ich habe erklärt, wie der Algorithmus arbeitet.) Als Beispiel nehme ich jetzt:

$$\begin{aligned} &n_3 = 1, \\ &n_2 \geq 3 \text{ beliebig }, \\ &n_1 = 2n_2 + 1 \end{aligned}$$

Ich möchte nun den Betrag $w = 3n_2$ wechseln. Offensichtlich würden drei Münzen vom Typ n_2 reichen. Der Algorithmus wählt jedoch „gierig“ zuerst eine Münze vom Typ n_1. Danach bleibt ein Betrag von $n_2 - 1$ übrig, wir wählen also keine Münze vom Typ n_2 und müssen den Rest durch $n_2 - 1$ Münzen vom Typ n_3 rausgeben. Man kann diese Anzahl also über die Variable n_2 beliebig verschlechtern.

Prüfer: Du hast ja das Prinzip der dynamischen Programmierung genannt. Hat das auch in GTI eine Anwendung? Was könnte ich wohl meinen? (Grinst.)
Ich: Den Cocke-Younger-Kasami Algorithmus, der das Wortproblem für kontextfreie Sprachen in polynomieller Zeit löst. Genauer gesagt in Zeit $O(|P|n^3)$. Angenommen wir haben ein Wort $w = w_1...w_n$ und wollen wissen, ob eine gegebene kontextfreie Grammatik G dieses Wort erzeugen kann.

Dazu berechnen wir jetzt die Menge der Variablen V_{1n}, aus denen man das Wort $w_1...w_n$ ableiten kann. Ist $S \in V_{1n}$ kann man das Wort also ableiten. Allgemein berechnen wir nach wachsendem $l = j - i$ die Menge der Variablen V_{ij}, die sich zum Wort $w_i...w_j$ ableiten lassen. Achja, die Grammatik ist in Chomsky-Normalform gegeben. Dann läßt sich die Menge der Variablen V_{ii} einfach berechnen. Es sind alle Variablen A, für die es die Ableitung $a \to w_i$ gibt. Für $j - i \geq 1$ gilt jetzt folgendes: $A \in V_{ij}$, wenn es eine Ableitung $A \to BC$ und ein $k \in \{i, ..., j - i\}$ gibt, so daß $B \in V_{ik}$ (also $B \to w_i...w_k$) und $C \in V_{k+1,j}$ gilt. Ich kann das Wort höchstens auf n Arten trennen und für jede Trennung muß ich mir $|P|$ Variablen anschauen. Das gibt dann für **ein** V_{ij} eine worst case Laufzeit von $O(|P|n)$. Es gibt nur $\binom{n}{2} + n = O(n^2)$ verschiedene V_{ij}. Also macht das insgesamt $O(|P|n^3)$.

Prüfer: Wie sieht es denn mit dem Wortproblem für reguläre Sprachen und rekursiv aufzählbare Sprachen aus?
Ich: Für reguläre Sprachen gibt es einen DFA, auf den ich das Wort ansetzen kann. Für rekursiv aufzählbare Spachen ist das Wortproblem die Sprache $U = \{\langle M \rangle w | M \text{ akzeptiert } w\}$, die nur rekursiv aufzählbar ist.

Prüfer: Kannst du das für U zeigen?
Ich: Ja, dies macht man über die Konstruktion der Diagonalsprache D. Man zeigt, daß D nicht rekursiv ist und daher auch nicht $\overline{D}$. Denn wenn $\overline{D}$ rekursiv wäre, so wäre auch D rekursiv. Jedoch ist $\overline{D}$ rekursiv aufzählbar. Und da U die Verallgemeinerung von $\overline{D}$ ist, ist auch U nicht rekursiv, da sonst $\overline{D}$ auch rekursiv wäre.

Prüfer: Gut. Es gibt ja diese Chomsky-Hierarchie. Kannst du zeigen, daß die echt ist? Also Chomsky-2 von Chomsky-0 abgrenzen, sowie Chomsky-3 von Chomsky-2.
Ich: Die Abgrenzung der Sprachklasse Chomsky-3 von Chomsky-2 kann man mit folgender Sprache zeigen:

$$L = \{0^i 1^i | i \geq 0\}$$

Diese Sprache ist nicht regulär, jedoch kontextfrei. Für die andere Abgrenzung reicht folgende Sprache:

$$L = \{0^i 1^i 2^i | i \geq 1\}$$

Diese Sprache ist nicht kontextfrei, jedoch rekursiv aufzählbar (sogar rekursiv).

Prüfer: Das muß man jetzt ja auch irgendwie beweisen. Wie mache ich das?
Ich: Dazu benutzt man das Pumping Lemma für reguläre Sprachen und das Pumping Lemma für Chomsky-2.

Prüfer: Kannst du mal für eine der beiden Sprachen das Pumping Lemma erklären bzw. es beweisen.
Ich: Dann nehme ich das Pumping Lemma für reguläre Sprachen.

Prüfer: Soll mir recht sein.
Ich: Das Pumping Lemma für reguläre Sprachen lautet:

$$\begin{aligned}&\exists N \in \mathbb{N}:\\ &\forall z \in L, |z| \geq N:\\ &\exists \text{ Zerlegung } z = uvw, |uv| \leq N, |v| \geq 1:\\ &\forall i \geq 0 : uv^i w \in L\end{aligned}$$

Wir wählen $N = |Q|$. Wenn wir nun vom Startzustand aus ein Wort lesen, welches mindestens N lang ist, werden auch mindestens noch N andere Zustände erreicht. Insgesamt macht dies mindestens $|Q|+1$ benutzte Zustände. Damit wird mindestens ein Zustand also doppelt durchlaufen. Sei q^* dieser Zustand. Dann ist u das Wort, welches wir gelesen haben, wenn der Automat zum ersten Mal q^* erreicht, v das Wort nachdem wir den Zustand q^* zum zweiten Mal erreichen und w der Rest. Da der Automat sich nicht merken kann, was er vorher gelesen hat, können wir v also noch beliebig oft lesen oder einfach weglassen.

Prüfer: Gut, kannst du die Sprache $L = \{0^i 1^j | i > j\}$ einordnen?
Ich: Die Sprache ist nicht regulär. (Ich habe erst versucht, mit dem Wort $0^{2N}1^N$ die Nichtregularität der Sprache nachzuweisen. Dabei wollte ich das normale Pumping Lemma benutzen, habe dann jedoch noch bemerkt, daß ich meinem Gegner (im Pumping Lemma-Spiel, nicht Paul... :)) damit ja nur unter die Arme greife. Habe mich korrigiert und gesagt, daß man das verallgemeinerte Pumping Lemma benutzen muß, damit man die Einsen aufpumpt.)

Prüfer: Gut, kannst du uns einen Moment alleine lassen.

Fazit: Ich kann mich den vielen Protokollschreibern vor mir nur anschließen. Die Atmosphäre der Prüfung ist wirklich ruhig. Am Anfang war ich zwar aufgeregt, nachher habe ich jedoch kaum noch bemerkt, daß ich mich in einer Prüfung befand. Wenn man den Stoff beherrscht, ist es fast schon so etwas wie ein kleines, ruhiges Fachgespräch, in dem auch mal gelacht wird.

Was ebenfalls sehr positiv war, ist die Tatsache, daß man nicht sofort gesagt bekommt, wenn man etwas falsch macht. So hat Paul mir erst nach der Prüfung gesagt, daß ich den Beweis der Diagonalsprache D wohl ziemlich vergeigt habe, bzw. anscheinend garnicht richtig verstanden habe. Hätte er es mir während der Prüfung gesagt, wäre ich wohl auch nur unsicherer geworden. Der Prüfer möchte also auch, daß man besteht.

Das soll jedoch nicht heißen, daß die Prüfung ein Spaziergang ist. Man muß schon zeigen, daß man etwas kann und daß man etwas getan hat. Ich habe zum Beispiel den Datenstrukturen-Stoff sowie den GTI-Stoff noch einmal selber zusammengefaßt, was ich nur empfehlen kann. Eine average case Analyse von Quick Sort wird auf einmal viel einfacher, wenn man sie mal in eigenen „Worten“ hingeschrieben (und natürlich auch verstanden) hat. Außerdem ist es eine gute Gelegenheit sich in LaTeXeinzuarbeiten.

Wer zu faul ist, zum selber schreiben (obwohl es sich wirklich lohnt!) und trotzdem Interesse an den beiden Zusammenfassungen hat, kann mir gerne eine Mail schreiben. Meine e-Mail-Adresse lautet:

`gregor01@marvin.cs.uni-dortmund.de`