Grundlagen Theoretischer Informatik 3

SoSe 2010 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Grundlagen Theoretischer Informatik 3 Gesamtübersicht

- Organisatorisches; Einführung
- Algorithmenanalyse: Komplexität und Korrektheit
- Randomisierte Algorithmen und ihre Analyse
- Zur Behandlung NP-schwerer Probleme: Ausblicke

Organisatorisches

Vorlesungen FR 8.30-10.00 im HS 13

Übungsbetrieb in Form von einer Übungsgruppe
BEGINN: in der zweiten Semesterwoche
FR 10-12, HS 13

Dozentensprechstunde DO 13-14 in meinem Büro H 410 (4. Stock)

Mitarbeitersprechstunde (Stefan Gulan) DO 13-14 H 413

Tutorensprechstunde MO 13.30-14.30 & MI 14-15 H 407/H412

Benotung

Die Note ergibt sich ausschließlich aus der gezeigten Leistung bei der Abschluss-prüfung.

Die Abschlussprüfung erfolgt schriftlich am 20.7.2010.

Zulassungskriterien

Um an der Abschlussprüfung teilnehmen zu dürfen, wird vorausgesetzt:

(1) Mindestens zweimaliges erfolgreiches Vorrechnen von Hausaufgaben sowie
(2) 40% der Hausaufgabenpunkte (Zweier-/ Dreiergruppenabgabe möglich)

Abgabe von Hausaufgaben

Diese sollte in Gruppen zu 2-3 Personen erfolgen.

Abgabeschluss ist MI 14 Uhr, im mit GTI 3 beschrifteten Kasten im 4. Stock vor dem Sekretariat von Prof. Näher.
Verspätete Abgaben gelten als nicht abgegeben und werden dementsprechend mit 0 Punkten bewertet

In der darauffolgenden Übung (i.d.R. am Freitag in derselben Woche) werden die korrigierten Lösungen zurückgegeben. Wir nutzen die Zeit (i.d.R. zwei Tage) auch dazu, uns die Namen derjenigen herauszusuchen, die gewisse Aufgaben dann vorzurechnen haben.
Wer die Aufgaben auf Aufforderung **nicht erläuternd vorrechnen** kann, bekommt die durch Hausaufgabenabgabe erzielten Punkte für das gesamte Aufgabenblatt abgezogen.
Selbstverständlich gilt das Vorrechnen dann nicht als “erfolgreich” im Sinne der vorigen Folie.
Sollte ein Vorrechnen nicht erfolgreich sein, wird i.d.R. ein weiterer Studierender Gelegenheit zum Vorrechnen bekommen.

Die Lösungen sind handschriftlich anzufertigen; weder Schreibmaschinen- noch Computerausdrucke werden akzeptiert, erst recht keine Kopien.

Probleme ? Fragen ?

Klären Sie bitte Schwierigkeiten mit Vorlesungen oder Übungen möglichst umgehend in den zur Verfügung gestellten Sprechzeiten.

In der Tutorensprechstunde stehen Ihnen (alternierend) Studenten höherer Semester zu Rückfragen bereit.

Wir helfen Ihnen gerne!
. . . wir sind aber keine Hellseher, die Ihnen Ihre Schwierigkeiten an der Nasenspitze ansehen. . .

Algorithmenanalyse: Modelle

Erinnerung: Berechenbarkeitsmodelle aus GTI 2

— Turingmaschinen modellieren “menschliche Rechner”
— WHILE-Programme stellen ein einfaches Modell für eine “Hochsprache” dar, in der nur “strukturiertes Programmieren” erlaubt ist (keine wilden Sprünge)

Wollen wir die Komplexität von Algorithmen analysieren, so bieten sich WHILE-Programme an, aber wie “genau” wären solcherlei Aussagen?

$\rightsquigarrow$ Wir benötigen ein “realistischeres” Rechnermodell. Wir zeigen heute: Auch sog. *Registermaschinen* sind berechnungsuniversell.

Von-Neumann-Architektur (siehe auch VL “Technische Informatik”; Quelle: Wikipedia)

Ein *Von-Neumann-Rechner* besteht aus:

- dem *Rechenwerk* (ALU: Arithmetic Logic Unit): dieser eigentliche Prozessor führt Rechenoperationen und logische Verknüpfungen durch;
- dem *Steuerwerk* (Control Unit), dieses interpretiert die Anweisungen eines Programms und steuert die Befehlsabfolge;
- dem *Speicherwerk* (Memory): es speichert sowohl Programme als auch Daten, welche für das Rechenwerk zugänglich sind;
- dem *Eingabe-/Ausgabewerk*, zuständig für I/O.

Prinzipien des gespeicherten Programms

— Befehle sind in einem RAM-Speicher (Random Access Memory: Speicher mit wahlfreiem Zugriff) mit linearem (1-dimensionalem) Adressraum abgelegt.

— Ein Befehls-Adressregister, genannt *Befehlszähler* (oder Programmzähler), zeigt auf den nächsten auszuführenden Befehl.

— Befehle können wie Daten geändert werden.

Prinzipien der sequentiellen Programm-Ausführung (*Von-Neumann-Zyklus*)

— Befehle werden aus einer Zelle des Speichers gelesen und dann ausgeführt.

— Normalerweise wird dann der Inhalt des Befehlszählers um Eins erhöht.

— Es gibt einen oder mehrere Sprung-Befehle, die den Inhalt des Befehlszählers um einen anderen Wert als $+1$ verändern.

— Es gibt einen oder mehrere Verzweigungs-Befehle, die in Abhängigkeit vom Wert eines Entscheidungs-Bit den Befehlszähler um Eins erhöhen oder einen Sprung-Befehl ausführen.

Registermaschinen (Cook / Reckhow 1973); RAM: Random Access Machines
Eine einfache Realisierung der von-Neumann-Architektur.

Wichtige Eigenschaften dieses Maschinenmodells:

* Speicher potentiell unendlicher Größe, realisiert als Register
* Register können natürliche ("manchmal" sogar reelle) Zahlen aufnehmen.
* Befehlssatz analog zu Assemblersprachen:
  — Lade-/Speicherbefehle,
  — arithmetische, logische etc. Operationen,
  — Sprungbefehle
* beschreibt nicht: Parallelrechner, Speicherhierarchie, ...

Bestandteile von Registermaschinen: genauer

(vgl. Kap. 1 aus: A. Asteroth, C. Baier: Theoretische Informatik, Pearson, 2003)

Befehlszähler $\mathtt{b}$, initiiert mit $\mathtt{b} = 1$ (erste Programmzeile)
Registerzellen $\mathtt{c}(\mathtt{i})$, $\mathtt{i} = 0, 1, \ldots$ zur Aufnahme natürlicher Zahlen
Akkumulator $\mathtt{c}(0)$: Hier wird gerechnet.
Programm: Folge von Befehlen.
Die verschiedenen Befehle werden im Folgenden erläutert.
Bei vielen Befehlen gibt es (drei) verschiedene Adressierungsarten.

Adressierungsarten am Beispiel des Ladebefehls

LOAD #12 Lade die Zahl 12 (in den Akkumulator $c(0)$) (*Konstante*)
LOAD 12 Lade den Inhalt von Register 12, also $c(12)$, nach $c(0)$ (*direkte Adressierung*)
LOAD $*12$ Lade den Inhalt von $c(12)$, also $c(c(12))$, nach $c(0)$ (*indirekte Adressierung*)

Allgemeine Form: LOAD x.
$v(x)$ bezeichne das "eigentliche Argument", also:
für $x = \#i$, $i \in \mathbb{N}$, gilt $v(x) = i$,
für $x = i$ gilt $v(x) = c(i)$, und
für $x = *i$ ist $v(x) = c(c(i))$.

Für den Speicherbefehl STORE ist der Operand $\#i$ unzulässig.

Befehlssatz einer Registermaschine

Befehl	Wirkung
LOAD x	$c(0) \leftarrow \nu(x); b \leftarrow b + 1$
STORE i	$c(i) \leftarrow c(0); b \leftarrow b + 1$
STORE $*i$	IF $c(i) \geq 1$ THEN $c(c(i)) \leftarrow c(0); b \leftarrow b + 1$ ELSE $b \leftarrow \infty$ (Abbruch) ENDIF
ADD x	$c(0) \leftarrow c(0) + \nu(x); b \leftarrow b + 1$
SUB x	$c(0) \leftarrow \max\{0, c(0) - \nu(x)\}; b \leftarrow b + 1$
GOTO j	$b \leftarrow j$
JZERO j	IF $c(0) = 0$ THEN $b \leftarrow j$ ELSE $b \leftarrow b + 1$ ENDIF
END	$b \leftarrow \infty$

Häufig gibt es weitere Befehle wie MULT x oder DIV x, manchmal auch eingeschränktere Behlssätze wie lediglich Inkrement und Dekrement anstelle der Addition und Subtraktion.

Berechnung von Funktionen durch Registermaschinen

Die k Argumente $n_1, \ldots, n_k$ eines RAM-Programms seien anfangs in $c(1), \ldots, c(k)$ gespeichert.
Alle anderen Register enthalten anfangs Null.

Der Einfachheit halber wollen wir explizit das Ausgaberegister angeben.

Triviales Beispiel: Das Programm
1: ADD 1
2: ADD 2
3: END

berechnet $f(x, y) = x + y$ mit Ausgaberegister 0.

Dies liefert den Begriff der *RAM-Berechenbarkeit* von Funktionen.

Realisierung der Multiplikation $f(x, y) = x \cdot y$ mit Ausgaberegister 3

1: LOAD #0
2: STORE 3
3: LOAD 2
4: JZERO 12
5: LOAD 3
6: ADD 1
7: STORE 3
8: LOAD 2
9: SUB #1
10: STORE 2
11: GOTO 4
12: END

Gedanken zur Korrektheit:

Gilt unmittelbar vor Zeile 5:
$c(1) = i$, $c(2) = j$, $c(3) = k$,
so gilt unmittelbar vor Zeile 8:
$c(1) = i$, $c(2) = j$, $c(3) = k + i$.
Unmittelbar vor Zeile 11 gilt daher:
$c(1) = i$, $c(2) = j - 1$, $c(3) = k + i$.

Unmittelbar vor Zeile 4 ist:
$c(1) = x$, $c(2) = y$, $c(3) = 0$.
Daher gilt nach d Durchläufen der Zeilen 5 bis 11 unmittelbar vor Zeile 5 stets:
$c(1) = x$, $c(2) = y - d$ und $c(3) = d \cdot x$.
Dies sieht man formal durch Induktion über d.

Beim Abbruch der Schleife gilt (Test Zeile 4) $d = y$, also $c(3) = y \cdot x$.

Zur Äquivalenz der Berechenbarkeitsmodelle

Klar: Ein Mensch kann die Arbeit einer RAM mechanisch nachvollziehen, und daher (nach der Church-Turing-These) kann das auch eine Turingmaschine.

Eine formalere Behandlung dieser “Richtung” finden Sie im angegebenen Buchkapitel.

Die Berechnungs-Universalität von RAMs folgt am einfachsten durch Simulation von WHILE-Programmen.
Die Variablen in WHILE-Programmen entsprechen Registern der RAM.

Schleifensimulation 1: LOOP x_i DO P END; Q

In einem Extra-Register wird der Wert von x_i anfangs gespeichert und x_i-mal dekrementiert.
Etwas formaler: Ist $\rho(X)$ das X entsprechende RAM-Programm, so ist $\rho(\text{LOOP } x_i \text{ DO } P \text{ END; } Q)$ gegeben durch:
$L0$: LOAD i
$L0 + 1$: STORE ℓ
$L0 + 2$: JZERO L-END
L: $\rho(P)$
$L1$: LOAD ℓ
$L1 + 1$: SUB #1
$L2 + 2$: GOTO $L0 + 1$
L-END: $\rho(Q)$

Hier ist ℓ ein "frisches" Register, das vor der Simulation und danach (!) Null enthält. $L0$ etc. bezeichnen Sprung-Labels (siehe Assemblersprachen).

Schleifensimulation 2: WHILE x_i DO P END; Q

Etwas formaler: Ist $\rho(X)$ das X entsprechende RAM-Programm, so ist $\rho(\text{WHILE } x_i \text{ DO } P \text{ END; } Q)$ gegeben durch:
$L0$: LOAD i
$L0 + 1$: JZERO L-END
L: $\rho(P)$
$L1$: GOTO $L0$
L-END: $\rho(Q)$

Hier ist ℓ ein "frisches" Register, das vor der Simulation und danach (!) Null enthält. $L0$ etc. bezeichnen Sprung-Labels (siehe Assemblersprachen).

Gedanken zum Compilerbau

WHILE-Programme sind ein einfacher Vertreter von “Hochsprachen”.
RAMs erscheinen sehr maschinennah (von-Neumann-Architektur!).

Trotzdem ist die Übersetzung von WHILE-Programmen in RAM-Programme recht einfach.
Dieser enthält dennoch wesentliche Gedanken aus dem Compilerbau.

Auch die Zeitkomplexität steigt nur unwesentlich an bei der Übersetzung.

Wie sollte man diese Komplexität aber messen?

Zeitmessung von Algorithmen (analog: Platz)

Technischer Fortschritt der Hardware $\leadsto$ Absolute Zeitmessung auf konkreten Rechnern führt nur bedingt zu vergleichbaren Aussagen.

Versuchen wir daher, "elementare Operationen" der Algorithmen zu zählen.

Beispiel: LOOP x_1 DO $x_2 \leftarrow x_2 + 1$ END.

Es werden x_1 Additionen (genauer: Inkremente) und ebensoviele Zuweisungen ausgeführt.

Unter Vernachlässigung weiterer (impliziter) Operationen, wie dem x_1-fachen Zähler-Dekrement (siehe Übersetzung in RAM-Programme) zählen wir hier $2x_1$ viele Elementaroperationen.

Hierbei blieb unberücksichtigt: Größe der beteiligten Zahlen.
Diese Art zu zählen führt zum *Einheitskostenmaß*.
Beim *logarithmischen Kostenmaß* gehen Zahlen mit ihrer Länge ein, es werden also letztlich die Bit-Operationen gezählt.

Wachstum von Funktionen — Motivation

Wichtige Ressourcen informatischen Denkens:

- Laufzeit (Rechenzeit) und
- Speicherbedarf (Speicherplatz, evtl. hierarchisiert)

Ziel mathematischer Modellierung: Unabhängigkeit von Stand der Technik
$\leadsto$ (maschinenabhängige) “Konstanten” sind “unwichtig”

Wesentliche Sichtweise der Theorie algorithmischer Probleme (*Komplexitäts-theorie*) und der Theorie konkreter algorithmischer Lösungen (*Algorithmik*)

$\mathcal{O}$-Notation

Definition (Paul Bachmann 1894, später von Edmund Landau popularisiert):
Es seien $f, g \in \mathbb{R}^{\mathbb{N}}$. $g(n) = \mathcal{O}(f(n))$ für $n \in \mathbb{N}$, falls $\exists M, n_0 \in \mathbb{N}$, so dass gilt:
$\forall n \geq n_0 : |g(n)| \leq |Mf(n)|$

Hinweis: Ungenauer (asymmetrischer) Gebrauch von "$=$"; genauer ist $\mathcal{O}(f)$ als Menge von Funktionen aufzufassen, dann müsste man $g \in \mathcal{O}(f)$ schreiben. Diese Sicht nehmen wir oft ein.

D.E. Knuth liest $\mathcal{O}$ als "Groß-Omikron"; weitere Schreibweisen:
$g \in \Omega(f)$ gdw. $f \in \mathcal{O}(g)$ (definiert wiederum Funktionenmenge)
$g \in \Theta(f)$ gdw. $g \in \mathcal{O}(f) \cap \Omega(f)$.

Einige Beispiele

$\mathcal{O}(1)$:	konstanter Aufwand, unabhängig von n
$\mathcal{O}(n)$:	linearer Aufwand (z.B. Einlesen von n Zahlen)
$\mathcal{O}(n \ln(n))$:	Aufwand guter Sortierverfahren (z.B. Quicksort(?!))
$\mathcal{O}(n^2)$:	quadratischer Aufwand
$\mathcal{O}(n^k)$:	polynomialer Aufwand (bei festem k)
$\mathcal{O}(2^n)$:	exponentieller Aufwand
$\mathcal{O}(n!)$:	Bestimmung aller Permutationen von n Elementen

Konkreter Vergleich

Annahme: 1 Schritt dauert 1 μs = 0.000001 s

$n =$	10	20	30	40	50	60
n	10 μs	20 μs	30 μs	40 μs	50 μs	60 μs
n^2	100 μs	400 μs	900 μs	1.6 ms	2.5 ms	3.6 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
2^n	1 ms	1 s	18 min	13 Tage	36 J	366 Jh
3^n	59 ms	58 min	6.5 J	3855 Jh	10^8 Jh	10^{13} Jh
$n!$	3.62 s	771 Jh	10^{16} Jh	10^{32} Jh	10^{49} Jh	10^{66} Jh

Anwendung in der Algorithmik: Laufzeitanalyse

Verschiedene Analysen sind von Interesse:

- bester Fall (best case)
- mittlerer Fall (average case)
- schlimmster Fall (worst case)

Für den eigentlich meist interessantesten mittleren Fall wären auch noch Annahmen über die zu erwartende Eingabeverteilung sinnvoll.
Meistens beschränkt man sich bei der Analyse auf den schlimmsten Fall.

Anwendung in der Algorithmik: Laufzeitanalyse von for-Schleifen (Beispiele)

Minimumsuche in einem Array der Länge n

```
min = a[0];
for (i = 1; i < n; i++)
       if(a[i] < min) min = a[i];
```

n "Schritte" für n Daten $\rightsquigarrow$ Laufzeit $\Theta(n)$

```
for (i = 0; i < k; i++)
   for (j = 0; j < k; j++)
      brett[i][j] = 0;
```

k^2 Schritte für k^2 Daten

$\rightsquigarrow$ linearer Algorithmus

```
for (i = 0, i < k; i++)
   for (j = 0; j < k; j++)
       if (a[i] == a[j]) treffer = true;
```

k^2 Schritte für k Daten

$\rightsquigarrow$ quadratischer Algorithmus

Anwendung in der Algorithmik: Laufzeitanalyse von for-Schleifen (Beispiele)

Minimumsuche in einem Array der Länge n

```
min = a[0];
for (i = 1; i < n; i++)
        if(a[i] < min) min = a[i];
```

n "Schritte" für n Daten $\leadsto$ Laufzeit $\Theta(n)$

Entsprechendes RAM-Programm ?

Wir nehmen an, das Feld sei in den Registern 1 bis n, $n \geq 1$, gespeichert.
(Wir können nicht bei Null anfangen aufgrund der besonderen Bedeutung vom nullten Register.)
m und ℓ seien "neue" Register.
m enthält am Ende das Minimum.

```
1:    LOAD 1
2:    STORE m
3:    LOAD #n
4:    JZERO 14
5:    STORE ℓ
6:    LOAD m
7:    SUB *ℓ
8:    JZERO 11
9:    LOAD *ℓ
10:   STORE m
11:   LOAD ℓ
12:   SUB #1
13:   GOTO 4
14:   END
```

Anwendung in der Algorithmik: Laufzeitanalyse von while-Schleifen (Beispiele)

Lineare Suche im Array

```
i = 0;
while (i < n) && (a[i] != x)
   i++;
```

Laufzeit:	bestenfalls	1 Schritt	$\Rightarrow$	$\Theta(1)$
	schlimmstenfalls	n Schritte	$\Rightarrow$	$\Theta(n)$
	im Mittel	$\frac{n}{2}$ Schritte	$\Rightarrow$	$\Theta(n)$

Annahme für den mittleren Fall: Es liegt (gleichwahrscheinliche) Permutation der Zahlen von 1 bis n vor. Dann ist die mittlere Anzahl

$$= \frac{1}{n} \sum_{i=1}^{n} i \approx \frac{n}{2}$$

Erinnerung: **Geometrische Verteilung**

Wir werfen wiederum wiederholt mit einer Münze, die mit Wahrscheinlichkeit p "Kopf" zeigt.

Wie oft muss man werfen, bis das erst Mal "Kopf" erscheint ?

X: ZV, die die Anzahl der nötigen Würfe beschreibt.
Definitionsbereich von X: Menge der endlichen Folgen von Münzwürfen.
$X(e)$ ist dann der Index der ersten Stelle, die "Kopf" ist.
Beispiel: $X((\text{Zahl}, \text{Zahl}, \text{Zahl}, \text{Kopf}, \text{Zahl}, \text{Kopf})) = 4$.
Wertebereich von X: Menge der positiven ganzen Zahlen.

$P[X = k] = (1-p)^{k-1}p$
$P[X = \cdot]$ ist Wahrscheinlichkeitsdichte wegen geometrischer Reihe:
$\sum_{k=1}^{\infty} P[X = k] = \sum_{k=1}^{\infty}(1-p)^{k-1}p = p \cdot \frac{1}{1-(1-p)} = 1.$

$$E[X] = \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1}p = \frac{p}{1-p}\sum_{k=0}^{\infty} k(1-p)^k = \frac{p}{1-p} \cdot \frac{1-p}{p^2} = \frac{1}{p}$$

Anwendung in der Algorithmik: Laufzeitanalyse von while-Schleifen (Beispiele)

Lineare Suche in 0/1-Array (also $x, a[i] \in \{0, 1\}$)

```
i = 0;
while (i < n) && (a[i] != x)
   i++;
```

Bester und schlimmster Fall wie bisher.
Mittlerer Fall bei Annahme einer "Erfolgswahrscheinlichkeit" pro Stelle von 1/2 und der weiteren (in der Regel fälschlichen) Annahme der Unabhängigkeit aufeinanderfolgender "Experimente"
$\leadsto$ Erwartungswert der geometrischen Verteilung ist 2 (unabhängig von n)
$\leadsto \Theta(1)$ im mittlereren Fall
Gilt analog für beliebige "endliche Alphabetgrößen"

Amortisierte Analyse

Oft wird der schlechteste Fall eines komplizierteren Algorithmus abgeschätzt durch die schlechtesten Fälle der Teilschritte; dies vermeidet die *Aggregat-Methode*, eine Möglichkeit der *amortisierten Analyse*.

Beispiel: Wir wollen zählen (abschätzen), wie viele Bitwechsel beim Inkrementieren eines k-Bit-Zählers von 0 bis $(2^k - 1)$ entstehen. Eine einfache Analyse, ausgehend vom schlimmsten Fall eines einzelnen Inkrements, liefert $\mathcal{O}(k \cdot 2^k)$. Geht es besser ?

Betrachten wir die Anzahl der Bitwechsel bei einem Zähler mit 3 Bit:

Zähler	0000	0001	0010	0011	0100	0101	0110	0111	1000
Anzahl Bitwechsel	0	1	2	1	3	1	2	1	4

Beobachte: Das niedrigste Bit ändert sich bei jedem Inkrement, das nächst höhere bei jeder zweiten, das wiederum nächst höhere bei jeder vierten usw.
Damit ergibt sich bei n Inkrementen folgende Summe von Bitwechseln:

$$n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^k} \right\rfloor \leq n \sum_{i=0}^{k} \frac{1}{2^i}$$

Diese Summe können wir nach oben abschätzen:

$$n \sum_{i=0}^{k} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n$$

Ein einzelnes Inkrement kostet daher amortisiert 2 Bitwechsel, nicht etwa k wie zunächst abgeschätzt.