Höhere Algorithmik
Mitschrift

Max Wisniewski

WS 2011/2012
30. Oktober 2011

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziel

Bisher haben wir einfache solcher Algorithmen betrachten (ALP1, ALP3, etc.). In dieser Vorlesung werden wir uns nun mit komplexeren Problemen beschäftigen. Wir wollen die Probleme unter folgenden Aspekten betrachten:

- Entwurf von Algorithmen
- Analyse dieser Algorithmen
- Bewertung dieser Algorithmen

1.2 Algorithmus

Def.: Ein Algorithmus ist ein endlich beschriebenes, effektives Verfahren, das eine Eingabe in eine Ausgabe überführt.

Zu Begin Betrachten wir ein einfaches Problem.

Kapitel 2

Repräsentant einer Menge

Gegeben sei folgendes statistisches Problem:
Es seien n Zahlen / Datensätze gegeben, wobei $n \ll 0$ gilt.
Gesucht ist ein Repräsentativer Wert für diese Menge.

2.1 Naiver Ansatz

Idee Wir verwenden den Durchschnitt / Mittelwert.
Die Laufzeit ist einfach, da wir nur einmal über alle Datensätze müssen. Setzen wir dabei eine konstante Zeit für Addition und Division vorraus, ist die Laufzeit $O(n)$.

Problem: Der Mittelwert ist Anfällig für Außreißer und daher nicht sehr aussagekräftig. Sind beispielsweise $n-1$ Werte zwischen 0 und 10 und ein $n$ter liegt bei 10.000.000 so wird das ganze Ergebnis zu diesem Wert hin verfälscht.

Dieser Repräsentant ist leicht zu berechnen, aber nicht sehr schön.
Betrachten wir daher einen anderen Ansatz.

2.2 K - SELECT

Def.: Ein Element s einer total geordneten Menge S hat den Rang k
$:\Leftrightarrow$ es gibt genau $(k-1)$ Elemente in S, die kleiner sind als s.

Man schreibt dafür $rg(s)$.

Def.: Sei S total geordnet mit $n = |\,S\,|$ und $s \in S$.

$$s \text{ heißt Median} :\Leftrightarrow\ rg(s) = \left\lceil \frac{n+1}{2} \right\rceil .$$

2.2.1 Das Problem

Gegeben Sei S, $|S| = n$ paarweise verschiedene Zahlen.
Nun wollen wir den Median s von S möglichst effizient finden.

2.2.2 Algorithmus I in $\Theta(n \cdot \log n)$

Was die Laufzeit schon nahe legt, bedienen wir uns hier eines Sortieralgorithmuses.

1. Soritere S. z. B. mit Heap - Sort .
   Benötigt $\Theta(n \cdot \log n)$ Schritte.
2. Gib das Element an der Stelle $\lceil \frac{n+1}{2} \rceil$ aus.
   Benötigt $\Theta(1)$ Schritte.

Laufzeit: $T(n) = \Theta(n \cdot \log n) + \Theta(1) = \Theta(n \cdot \log n)$.

Da für (vergleichsbasiertes) Sortieren jede Lösung mit $\Omega(n \cdot \log n)$ beschränkt ist, kann eine Lösung für das Medianproblem die Sortierung verwendet nicht schneller sein. Bleibt zu untersuchen, ob der Median ähnlich schwer ist, oder ob es einen Algorithmus gibt, der das Problem schneller lösen kann.

2.2.3 SELECT in O (n)

Angenommen es existiert eine Funktion SPLITTER(S), welche uns ein Element $q \in S$ liefert, so dass gilt:

$$rg(q) \geq \left\lfloor \frac{1}{4}\, n \right\rfloor \quad \wedge \quad rg(q) \leq \left\lceil \frac{3}{4}\, n \right\rceil .$$

Lemma: Angenommen wir können SPLITTER ohne weitere Kosten benutzen. Dann können wir den Median in $O(n)$ Zeit berechnen.

Beweis: Um diese Aussage zu beweisen lösen wir das allgemeinere Problem

$$\text{SELECT}(k, S)$$

finde Element mit Rang k. Dieses Problem wird Äuswahlproblem"genannt.

Idee: Nehme SPLITTER als PIVOT Element und teile die Menge der Daten daran auf.

Pseudocode:

```
SELECT( k , S )
  IF |S|  <  100 THEN
    RETURN BRUTFORCE( k , S )   // z. B. Algorithmus I
  q ← SPLITTER( S )
  S_< ← { s ∈ S | s < q }
  S_> ← { s ∈ S | s > q}
  IF | S_< | ≥ k THEN
    RETURN  SELECT( k , S_< )
  ELSE IF |S_<| = k − 1 THEN
    RETURN q
  ELSE
    RETURN SELECT( k − |S_<| − 1 , S_>)
```

Laufzeitanalyse:

Da $rg(q) \in \left[\left\lfloor \frac{1}{4}\, n \right\rfloor , \left\lceil \frac{3}{4}\, n \right\rceil\right]$ gilt $|S_<| , |S_>| \leq \frac{3}{4}\, n$.
Also gilt:

$$T(n) \leq \begin{cases} O(1) & , n < 100 \\ O(n) + T\left(\frac{3}{4}\, n\right) & , \text{sonst} \end{cases}$$

Behauptung:

$$T(n) \in O(n)$$

Beweis:

$$\begin{aligned} T(n) &\leq c \cdot n + T\left(\tfrac{3}{4}\, n\right) \\ &\leq c \cdot n + c\left(\tfrac{3}{4}\, n\right) + T\left((\tfrac{3}{4})^2\, n\right) \\ &\leq c \cdot n \sum_{i=0}^{\infty} \left(\tfrac{3}{4}^i\right) + O(1) \\ &\leq (4c) \cdot n + O(1) \\ &= O(n) \end{aligned}$$

□

2.2.4 Implementierung von SPLITTER

Damit k-SELECT die versprochene lineare Laufzeit erreicht, müssen wir uns als nächstes die Implementierung von Splitter ansehen. Da wir uns in jedem Schritt einen neuen Splitter besorgen, muss die Laufzeit sehr gering gehalten werden.

Randomisierte Lösung

Die erste Idee ist es, statt dem SPLITTER mit den gewünschten Eigenschaften einfach einen zufällig Gewählten zu nehmen. Wenn man diese Laufzeit berechnen wird man auch auf eine lineare kommen.

Um dieses Problem zu lösen werden wir später Randomisierte Algorithmen betrachten und wie man Laufzeiten aus Erwartungswerten bestimmt.

BFRPT - Algorithm

Der Algorithmus wurde nach seinen Entdeckern Blum[1], Floyd[2], Pratt, Rivest[3], Tarijan[4] benannt.

Grundlegend funktioniert der Algorithmus folgender Maßen:
Man wählt zufällig eine Stichprobe $S' \subseteq S$ mit $|S'| = \left\lfloor \frac{n}{5} \right\rfloor$, so dass der Median von S' ein guter Splitter von S ist. Bestimme rekursiv den Media von S'.

Wählen von S'

Die Idee ist S in 5er Gruppen zu unterteilen. Innerhalb dieser Gruppen können wir den Median in konstanter Zeit findet. Baue aus den Medianen der 5er Gruppen die Menge S' und nimm deren Median.

Lemma: Der Median von S' ist ein guter SPLITTER von S, wenn n groß genug ist.

Anschaung: HIER WIRD NOCH EIN BILD UND ERKLÄRUNG EINGEFÜGT!!

Beweis:

Wir wollen prüfen, ob der Median g, den wir finden, wirklich SPLITTER Eigenschaften besitzt. Das heißt wir wollen wissen, ob min. $\frac{1}{4}$ kleiner und $\frac{1}{4}$ größer ist.

Größer:
Es sind $\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil$ Elemente aus S' größer als g. Da alle Elemente aus S' Mediane ihrer 5er Gruppen sind, wissen wir, das in jeder dieser Gruppen 3 Elemente größer sind als$g$. Dies gilt für alle Gruppen, außer die Gruppevon g selber und die mögliche letzte Gruppe.
Dies führt zu $3 \cdot \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 3$

Kleiner:
Es gibt ebenso$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil$ Gruppen, deren Meridiane kleiner sind als g. In jeder

[1] Turing Award 1995
[2] Turing Award 1978
[3] Turing Award 2002
[4] Turing Award 1986

dieser Gruppe, wissen wir von 3 Elementen die kleiner sind, bis auf die Gruppe von g und die letzte Gruppe, die in diese Klasse fallen könnte.
Dies führt zu mindestens $3\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 3$ Elementen, die kleiner sind als q.

Zusammensetzen:
Es gilt $3\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 3 \geq 3 \cdot \frac{1}{2} \cdot \frac{1}{5}n - 3 = \frac{3}{10}n - 3$.
Für einen guten SPLITTER, muss die Anzahl der außerhalb liegenden Elemente (sowohl größer als auch kleiner) größer als $\frac{1}{4}n$ sein.

$$\begin{aligned} \Rightarrow \frac{3}{10}n - 3 \geq \frac{1}{4} \quad &\Leftrightarrow \quad \left(\frac{3}{10} - \frac{1}{4}\right) n \geq 3 \\ &\Leftrightarrow \quad n \geq 60 \end{aligned}$$

Wir sehen hier, dass wir mit dem Verfahren garantiert einen guten SPLITTER finden, wenn wir mehr als 60 Elemente haben. Mit diesem Problem haben wir uns aber schon im Algorithmus beschäftigt. Dort haben wir gesagt, dass wir bei Listen bestimmter Größe das ganze Problem mit BRUTEFORCE lösen wollen. Damit werden die Listen in denen wir einen SPLITTER suchen immer garantiert 60 Elemente besitzen.

Nachdem wir nun wissen, dass wir mit diesem Verfahren einen SPLITTER erhalten, müssen wir prüfen, ob dieses Verfahren den Algorithmus asymptotisch langsamer macht oder ob wir bei einer lineraten Laufzeit bleiben.

Laufzeit K-SELECT mit BFRPT Algorithmus

Betrachten wir nocheinmal, wie unser Algorithmus nun nach dem Einsetzen des SPLITTER Codes aussieht.

```
SELECT (S, K)
  IF |S| < 100 THEN
    RETURN BRUTEFORCE(S , K)
  Unterteile S in 5er Gruppen
  S'← {Median jeder 5er Gruppe}
  q ← SELECT( S', ⌈(|S'|+1)/2⌉)
  S_< ← {s ∈ S | s < q}
  S_> ← {s ∈ S | s > q}
  IF |S_<| ≥ k THEN
    RETURN SELECT(S_<, K)
  ELSE IF |S_<| = K − 1 THEN
    RETURN q
  ELSE
    RETURN SELECT(S_>, K − |S_<| − 1)
```

Nun können wir aus dem Programm die Anzahl der Vergleiche ablesen:

$$T(n) \leq \begin{cases} O(1) & , n < 100 \\ O(n) + T\left(\left\lceil\frac{n}{5}\right\rceil\right) + T\left(\frac{3}{4}n\right) & , \text{sonst} \end{cases}$$

Behauptung: $T(n) = O(n)$

Beweis: Induktion über n mit $\exists \alpha > 0 \ : \ T(n) \leq \alpha \cdot n$

I.A.: $n < 100$ klar, da $T(n)$ konstant.

I.S.: $n \to n + 1$

$$\begin{array}{rcl} T(n) & \underset{\text{I.A.}}{\leq} & c \cdot n + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{3}{4}n\right) \\ & \leq & c \cdot n + \alpha \left\lceil \frac{n}{5} \right\rceil + \alpha \left(\frac{3}{4}n\right) \\ & \leq & c \cdot n + \alpha \left(\frac{n}{5} + 1\right) \alpha \left(\frac{3}{4}n\right) \\ & = & n \left(c + \frac{19}{20}\right) \alpha + \alpha \\ & \leq & \alpha \cdot n \end{array}$$

Den letzten Schritt darf jeder für sich selbst nachvollziehen.
So sehen wir, dass die Laufzeit immer noch lineare ist.

□

2.3 Bemerkung

Was wollten wir an diesem Beispiel sehen?
Was wir erreicht haben, ist ein Algorithmus, der kurz, elegant und optimal ist. Um diesen zu gewinne mussten wir nicht-triviale Struckturen benutzen, auf die man nicht mehr so leicht kommt, wie auf Quicksort oder Mergesort.
Die Laufzeit des Algorithmus war nicht sofort offensichtlich und brauchte eine Analyse samt Beweis.

Mit solchen Algorithmen werden wir uns im folgenden in der Vorlesung beschäftigen. Ein jeder hat mindestens einen kleinen Kniff dabei.
Zu Bemerken bleibt, dass der oben genannte und analysierte Algorithmus zwar linear läuft, die Konstanten sind allerdings so hoch, das bis zu einer Listengröße von 2^{25} Quicksort und anschließendes nehmen des k-ten Elements schneller ist.

Kapitel 3

Definition und Modell

In diesem Abschnitt werden das Berechnungsmodell, dass wir zu Grunde legen eingeführt und erklärt, sowie grundlegende Definition erneut eingeführt. Alldies sollte schon aus vorangegangenen Veranstaltungen bekannt sein, wird aber zu Klarheit nocheinmal eingeführt.

3.1 Berechnungmodell

Bei der Analyse von Algorithmen zählen wir *elementare Schritte*. Um diese Schritte näher zu beschreiben müssen wir zunächst das Berechnungsmodell einführen.
Da wir nicht für jeden neuen Computer, der entwickelt wird, ein neues Berechnungsmodell einführen wollen. ist das Berechungsmodell ein *abstraktes, mathematisches Modell* von Rechner, um Begriffe wie Berechenbarkeit, Algorithmus, Laufzeit, Speicherplatz, etc. zu definieren.

Beispiele: Turingmaschiene, μ-Rekursion, GameOfLife, Lambda Kalkül und viele mehr.

Wir werden im folgenden die RAM benutzen.

3.1.1 RAM

Die RAM (Random Access Maschine) ist eine Registermaschiene, die einen klassischen von-Neumann Rechner simuliert. Sie besteht aus zwei Komponenten.

Register: Eine Registermaschiene hat *unendlich* viele Register $[R_0|R_1|R_2|...]\,, mit \quad R_i \in \mathbb{N} \forall i \in \mathbb{N}$

Programm: Ein Programm ist eine *endliche* Folge von Befehlen. Die Befehle sehen, wie folgt aus:

- $A := B$ op C, wobei A, B, C Register, indireckt addressiert oder eine Zahl sein kann und op $\in \{+, -, */\}$ [1].
- $A := B$
- GOTO L (label)
- GGZ B, L
  springt zu L (label), wenn B größer als 0 ist.
- GLZ B, L
  springt zu L (label), wenn B kleiner als 0 ist.
- GZ B,L
  springt zu L (label), wenn B gleich 0 ist
- HALT
  Beendet das Programm

Variante: Als kleine Variation zu dieser RAM benötigen wir für den späteren Stoff die *Probabilistische RAM*. Diese besitzt eine zusätzliche Operation:

- RAND B
  erzeugt eine zufällige Zahl $[0, B)$

3.1.2 Imperatives Programmieren

Für das imperative Programmieren ist der Zustand am wichtigesten.

$$Z := (IP, R_0, R_1, R_2, ...)$$

wobei IP der Befehlszähler (Programmcounter) is.
Jeder Befehl hat einen *Effekt*, der den Zustand ändert. Darüber können wir wiederum, wie in GTI gelernt, den Folgezustand

$$Z_i = (IP, R_0, R_1, R_2, ...) \models (IP', R_0', R_1', R_2, ...) = Z_i + 1$$

und darüber den transitiven Abschluss:

$$Z_0 \overset{*}{\models} Z_n \Leftrightarrow \exists Z_1, ..., Z_{n-1} \ : \ Z_0 \models Z_1 \models ... \models Z_{n-1} \models Z_n$$

3.1.3 Funktionsberechnung

Def.: Ein Programm *berechnet* eine Funktion

$$f \ : \ \mathbb{N}* \to \mathbb{N}*$$

falls gilt:
Bei Eingabe $a_0, a_1, a_2, ..., a_{n-1}$ in die Register $R_0, R_1, R_2, ..., R_{n-1}$. Läuft das Programm bis HALT, steht danach die Ausgabe $b_0, b_1, b_2, ..., b_{m-1} = f(a_1, a_2, ..., a_{n-1})$ in den Registern $R_0, R_1, R_2, ...R_{m-1}$.

[1] / ist die ganzzahlige Division, da auf $\mathbb{N}$ nur diese definiert ist

3.1.4 Church - Turing - These

> Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.

Eine daraus folgende Implikation ist, dass es kein Rechnermodell geben kann, dass mehr als die bisherigen Modelle berechnen kann. Für uns ist (ohne das wir es nocheinmal speziell gezeigt haben) RAM-berechenbarkeit genau intuitiv-berechenbra (Turingberechenbar).

3.2 Laufzeit und Speicherplatz

Im folgenden sei P_f RAM - Programm, das f berechnet und x Eingabe.

Def.:

T_{P_f} : Gesammtkosten der Arbeitsschritte bis HALT bei x erreicht wird.

S_{P_f} : Gesammter Platzbedarf bis HALT bei x erreicht wird.

3.2.1 Kostenmaße

Was bedeuten diese Definitionen konkret für uns?

Einheitskostenmaß (EKM)

Jeder Schritt hat die Kosten 1.

$T_{P_f}(x) =$ #Schritte, die bei Eingabe x ausgeführt werden.

$S_{P_f}(x) =$ #*verschiedene* Register auf die das Programm zugreift.

Vorteil

[-]

- einfach
- einigermaßen Realistisch

bfseries Nachteil

[-]

- unrealistisch bei großen Zahlen

Logarithmisches Kostenmaß (LKM)

Die Kosten eines Befehles sind die gesammte Anzahl der manipulierten Bits. Ist auch auf andere basen übertragbar.

$T_{P_f}(x)$ = Muss für die einzelnen Befehle eigen definiert werden.
z.B. $R_0 = R_1 + R_2 : \lfloor \log |R_1| + \log |R_2| \rfloor$, wobei noch mehr denkbar ist (z.B. R_0 reinrechnen)

$S_{P_f}(x) = \max \{$Bits, die zu einem Zeitpunkt benutzt werden$\}$

Vorteil

[-]

- realistisch, auch bei großen Zahlen

bfseries Nachteil

[-]

- schwer damit zu rechnen

Pragmatische Entscheidung

Wann wollen wir nun welches Modell anwenden?
Die Vor- und Nachteile geben uns schon einen guten Indikator, wann sich was anbieten. Folgende Überlegung erweist sich oft als sinnvoll:

EKM Kombinatorischer Algorithmus
(z.B. Suchen, Sortieren, Zeichenketten, Graphen)

LKM Zahlentheoretische Algorithmen
(z.B. Primzahlen (-test, -findung), Rechenoperationen, etc.)

Vorsicht Im EKM sind einige schmutzige Tricks möglich. (z.B. Primzahlfindun in linearer Zeit)

3.3 Verallgemeinerung

Bisher haben wir für die Laufzeit nur jeweils eine feste Eingabe betrachtet. Im folgenden wollen wir Eingaben in Größen (zusammenhängenden Gruppen) zusammen fassen.
Wie verhält sich der Algorithmus bei einer Eingabegröße?

3.3.1 Beispiel

Diese Eingabegrößen sind uns in früheren Analysen schon untergekommen.

1. Die Anzahl der Elemente einer Liste/Menge. Wird oft beim sortieren verwendet.

2. Anzahl der Knoten und Kanten eines Graphen. Wird bei vielen Graphenalgorithmen benutzt.

3. Anzahl der Stellen einer Zahl. Kann man für Primzahltests nehmen.

3.3.2 Worst-Case-Analyse

Mit der Worst-Case-Analyse soll die schlimmstmögliche Laufzeit und Speicherplatzbedarf für eine Eingabegröße errechnet werden:

$$T_{\text{wc}}(n) = \max\{T(x) \mid |x| = n\}$$
$$S_{\text{wc}}(n) = \max\{S(x) \mid |x| = n\}$$

In der Analyse kann man sich das Leben oft leichter machen, wenn man sich die Eingabe für den schlimmsten Fall vorher überlegt und nicht erst alle durchprobiert.

3.3.3 Amortisiert Analyse

Bei der amortisierten Analyse will man die durchschnittlichen Kosten für eine oder mehrere Operationen bestimmen. Uns sind diese Analysen beispielsweise schon beim einfügen in eine Arraylist begegnet.

Da wir uns in der Vorlesung noch nicht spiezieller damit beschäftigt haben, wird dieser Punkt zu einer späteren Zeit ergänzt werden.