Höhere Algorithmik
Mitschrift

Max Wisniewski

WS 2011/2012
8. November 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:

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

□

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 berechnet, wird man auch auf eine lineare Laufzeit 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 folgendermaßen:
Man wählt zufällig eine Stichprobe $S' \subseteq S$ mit $|S'| = \left|\frac{n}{5}\right|$, so dass der Median von S' ein guter Splitter von S ist. Bestimme rekursiv den Median von S'.

Wählen von S'

Die Idee ist S in 5er Gruppen zu unterteilen. Innerhalb dieser Gruppen können wir den Median in konstanter Zeit finden. 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.

Anschauung: 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.: $n \to n+1$

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

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

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

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:

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

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.

Kapitel 4

Grundlegende Techniken

4.1 Lösen von Rekursionsgleichungen

4.1.1 Raten und Induktion

Diesen Ansatz ist immer dann zu wählen, wenn einem die Laufzeit des Algorithmus sofort klar ist.

Das ganze führt natürlich nur zum Ziel, wenn die Induktion aufgeht. Man kann also prinzipiell ziemlich lange raten, bis man auf ein vernünftiges Ergebnis kommt.

4.1.2 Auflösen und Muster erkennen

Beim Auflösen, führen wir die Rekursion Schritt für Schritt aus. Sobald wir ein Muster beim Auflösen erkennen, können wir es bis auf Ebene k hinschreiben und dann analysieren, für welches k der Anker erreicht wird. Diesen kann man dann in die Gleichung einsetzen und wird dadurch die endgülitge Laufzeit erhalten.

Beispiel: Mergesort
Annahme: $n = 2^k$.

$$T(n) = \begin{cases} 0 & , n = 1 \\ 2T\left(\frac{n}{2}\right) + n - 1, \text{sonst} \end{cases}$$

$$\begin{aligned}
T(n) &= 2T\left(\tfrac{n}{2}\right) + n - 1 \\
&= 2 \cdot \left(2T\left(\tfrac{n}{2^2}\right) + \tfrac{n}{2} - 1\right) + n - 1 \\
&= 2^2 T\left(\tfrac{n}{2^2}\right) 2n - 2^1 - 2^0 \\
&= 2^3 T\left(\tfrac{n}{2^3}\right) 3n - 2^2 - 2^1 - 2^0 \\
& \text{Nach k Schritten} \\
&= 2^k T\left(\tfrac{n}{2^k}\right) + kn - \sum_{i=0}^{k-1} 2^i \\
& k = \log n \text{ (Beim Anker)} \\
&= 2^{\log n} \cdot T(1) + n \cdot \log n - \tfrac{2^{\log n} - 1}{2 - 1} \\
&= n \log n - n + 1
\end{aligned}$$

4.1.3 Rekursionbaum analysieren

Die Rekursionsbaummethode läuft ähnlich zur Einsetzmethode. Hier werden die einzelnen Schritte der Rekursion aber als Baum dargestellt und danach ausgewertet.
Beispiel:

$$T(n) \leq \begin{cases} 0 & , n = 1 \\ 3 \cdot T(\frac{n}{2}) + cn & , \text{sonst} \end{cases}$$

Graph is tbd

4.1.4 Master Theorem

Satz: Sei $a \geq 1$, $n > 1$, $c > 0$

$$f : \mathbb{N} \to \mathbb{R}^+ \quad , \quad t : \mathbb{N} \to \mathbb{R}^+$$

Sei

$$T(n) = \begin{cases} t(n) & , n < c \\ a \cdot T(\frac{n}{b}) + f(n) & , \text{sonst} \end{cases}$$

eine Rekursion.
Dann gilt

(i) Wenn

$$\exists \varepsilon > 0 \ : \ f(n) = O\left(n^{\log_b a - \varepsilon}\right),$$

dann ist

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

(ii) Wenn

$$f(n) = \Theta\left(n^{\log_b a}\right),$$

dann ist

$$T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right)$$

(iii) Wenn

$$\exists \varepsilon > 0 \ : \ f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \wedge \exists d < 1 \ : \ a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n),$$

dann ist

$$T(n) = \Theta\left(f(n)\right)$$

Beispiele:

(1) Karatsuba (mit Optimierung)
$T(n) \leq 3 \cdot T(\frac{n}{2}) + O(n)$
Das Mastertheorem geht mit $a = 3,\ b = 2,\ f(n) = cn,\ \varepsilon = 0.25$
$cn \leq n^{1.5-\varepsilon} = n^{1.25} = O(n^{\log_2 3\ -\varepsilon}$

$$\Rightarrow T(n) = \Theta\left(n^{\log_2 3}\right)$$

(2) Karatsuba (ohne Optimierung)
$T(n) = 4 \cdot T(\frac{n}{2}) + O(n)$
Mastertheorem geht mit $a = 4,\ b = 2,\ f(n) = cn,\ 0 < \varepsilon \leq 1$
$f(n) = O(n^{2-\varepsilon})$

$$\Rightarrow T(n) = \Theta(n^2)$$

(3) Mergesort
$T(n) = 2 \cdot T(\frac{n}{2}) + n - 1$
Das Mastertheorem geht, mit $a = 2, b = 2, f(n) = n - 1$
$f(n) = n - 1 = \Theta(n^{\log_2 2}) = \Theta(n)$

$$\Rightarrow \Theta(n \cdot \log n)$$

(4) Binäre Suche
$T(n) = T(\frac{n}{2}) + c$
Mastertheorem geht, mit $a = 1, b = 2, f(n) = c$
Das Mastertheorem geht, mit $a = 1, b = 2, f(n) = c$
$c = \Theta(n^{\log_2 1}) = \Theta(n^0) = \Theta(1)$

$$\Rightarrow T(n) = \Theta(\log n)$$

Beweisidee: Lösen wir den Rekursionsbaum der allgemeinen Gleichung auf, so erhalten wir die folgende Form:
Wir in der nächsten Ebene pro Knoten a neue Knoten und teilen unser n durch b. Die Höhe des Baumes ist damit $\log_b n$.

$$\Rightarrow T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

Nun können wir daraus mehrere Fälle gewinnen:
Falls auf jeder Ebene die selbe Zahl steht, gewinnen Fall 2 aus dem Satz.
Falls die Kosten pro Ebene immer kleiner werden, erhalten wir Fall 3.
Und falls die Kosten immer mehr werden, erhalten wir den ersten Fall1.

Grenzfälle: Wo geht das Mastertheorem nicht?
Bei mehr als einem verscheidenen Rekurisonsaufruf, z.B. $T(n) = 2 \cdot T(\frac{n}{2}) + T(\frac{n}{3}) + O(n)$

Wenn keiner der Fälle greift, z.B. $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n \cdot \log n)$

4.2 Divide & Conquer

Def.: Unterteile ein großes Problem in kleinere Probleme. Löse diese einzeln und setzte die Teilergebnisse zusammen.

Bsp.: Mergesort, Quicksort

Um Divide&Conquer Algorithmen zu Analysieren muss man häufig Rekursionsgleichungen analysieren.

4.2.1 Beispiel: Multiplizieren von Zahlen

Gegeben sind zwei Zahlen $a, b \in \mathbb{N}$
Gescuht ist $a \cdot b$

Schulmethode

Beispiel: Sei $a = 1234$ und $b = 512$

$$\begin{array}{cccccccc}
1 & 2 & 3 & 4 & \cdot & 5 & 1 & 2 \\
\hline
 & & & & 2 & 4 & 6 & 8 \\
 & & & 1 & 2 & 3 & 4 & \\
 & & 6 & 1 & 7 & 0 & & \\
\hline
 & & 6 & 3 & 1 & 8 & 0 & 8
\end{array}$$

Sei $n :=$ Anzahl der Ziffern (ist eine Zahl kürzer als die andere, füllen wir sie am Anfang mit 0en auf).
$\Rightarrow n^2$ Multiplikationen und n^2 Additionen $\Rightarrow \Theta(n^2)$

Dies gilt für jedes Zahlensystem, da sich die Anzahl der Ziffern mit den Systemen immer weiter verändern.

Divide&Conquer beim Multiplizieren

Wir teilen a, b in 2 kleinere Zahlen (weniger Ziffern) und lösen das ganze rekursiv. Wir nehmen jetzt einmal an, dass wir uns im Binärsystem bewegen.

$$a = a_h \cdot 2^{\lceil \frac{n}{2} \rceil} + a_l$$
$$b = b_h \cdot 2^{\lceil \frac{n}{2} \rceil} + b_l$$

$$\Rightarrow \begin{array}{rcl} a \cdot b & = & \left(a_h \cdot 2^{\lceil \frac{n}{2} \rceil} + a_l\right)\left(b_h \cdot 2^{\lceil \frac{n}{2} \rceil} + b_l\right) \\ & = & a_h b_h \cdot 2^{2\lceil \frac{n}{2} \rceil} + a_h b_l \cdot 2^{\lceil \frac{n}{2} \rceil} + a_l b_h \cdot 2^{\lceil \frac{n}{2} \rceil} + a_l b_l \end{array}$$

Das führt zu:

- 2 Multiplikationen mit $\frac{n}{2}$ Bits
- 2 x Bitshifting
- 3 Additionen

Anker bei 1,2,4 Bits (egal so lange eine Konstante gewählt wird).

Laufzeit:

$$T(n) \leq \begin{cases} O(1) & ,n = O(1) \\ 4T(\frac{n}{2}) + O(n) & ,sonst \end{cases}$$

Diese Formel wird an dieser Stelle nicht aufgelöst, da das Ergbnis mit $\Theta(n^2)$ nicht besser geworden ist.

Optimierung:

Unser Problem ist, das wir 4 rekursive Aufrufe haben. Doch durch einen genialen Einfall, können wir das ganze auf 3 reduzieren.

Betrachten wir einmal die Form von eben, aber wir streichen die Verschiebung durch 2^k.

$$\begin{array}{rrcl} & (a_h + a_l)(b_h + b_l) & = & a_h b_h + a_h b_l + a_l b_j + a_l b_l \\ \Leftrightarrow & a_h b_l + a_l b_j & = & (a_h + a_l)(b_h + b_l) - a_h b_h - a_l b_l \end{array}$$

Hier sieht man, dass wir mit 3 Gleichungen auf unsere 4 Terme kommen können. Das bedeutet für unsere Rekursion:

$$T(n) \leq \begin{cases} O(1) & ,n \leq 3 \\ 4T(\frac{n}{2}) + O(n) & ,sonst \end{cases}$$

Lösen der Rekursionsgleichung

$$\begin{array}{rcl} T(n) & \leq & 3 \cdot T\left(\frac{n}{2}\right) + cn \\ & \leq & 3 \cdot \left(3 \cdot T\left(\frac{n}{2^2}\right) + c \cdot \frac{n}{2}\right) + cn \\ & & \dots \\ & \leq & 3^k \cdot T\left(\frac{n}{2^k}\right) + cn \cdot \left(\sum\limits_{i=0}^{\log k} \frac{3^i}{2^i}\right) \\ & & \text{Nach } k = \log n \text{ Schritten haben wir den Anker erreicht} \\ & = & 3^{\log n} \cdot O(1) + cn \cdot \left(\sum\limits_{i=0}^{\log k} \frac{3^i}{2^i}\right) \\ & = & c \cdot \frac{\left(\frac{3}{2}\right)^{\log n + 1} - 1}{\frac{3}{2} - 1} \\ & = & 2cn \cdot \frac{3}{2} \cdot \left(\frac{3}{2}\right)^{\log n} \\ & = & 3cn^{\log 3} \end{array}$$

Die Laufzeit ist also $T(n) \in O(n^{\log 3})$, wobei $n^{\log 3} \approx n^{1.58}$ ist, was weniger ist als n^2.

4.2.2 Beispiel: Closest Pair Problem

Bei diesem Problem haben wir n Punkte in der Ebene gegeben und wollen das Paar von Punkten finden, so dass der Abstand dieser Punkte das Minimum aller Abstände der menge ist.

Naiver Ansatz

Wir berechnen alle Abstände und nehmen davon das minimum.
Alle Abstände zu berechnen dauert $\left(\binom{n}{2}\right)$, das minimum einer Menge kann man in $n-1$ herraus finden. $\Rightarrow \Theta(n^2)$

Closest-Pair mit Divide & Conquer

Wir teilen die Punktmenge in 2 gleichgroße Hälften indem wir nach x Koordinate sortieren. Wir nehmen q als Median dieser Menge und definieren:

$$P_L = \{p \in P \mid p_x \leq q_x\} \quad , \quad P_R = \{p \in P \mid p_x > q_x\}$$

Finde in diesen Mengen rekrusiv das engste Paar.
$\delta_L = CP(P_L)$, $\delta_R = CP(P_R)$
$\delta = \min\{\delta_L \ , \ \delta_R\}$

Nun muss dieses δ aber nicht der kleinste Abstand sein, da das engeste Paar genau auf die beiden Teile verteilt sein könnte.

Beobachtung 1: Wen es ein engstes Paar p, q mit $p \in P_L$, $q \in P_R$ gibt, so liegt es in einem 2δ Streifen um die Mittellinie q_x.

Beobachtung 2: Wenn es ein engeres Paar gitb, so muss es zusätzlich im Rechteck $(\delta, 2\delta)$ zentriert um die Mittelachse liegen.

Beobachtung 3: Der Abstand zwischen 2 Punkten wird üblicherweise mit $\sqrt{\|\|\|p,q\|\|\|}$ berechent. Aber auf unserer RAM können wir die Wurzel leider nicht einfach so berechnen. Aber da wir nur die Abstände vergleichen wollen, können wir auch die Quadrate der Abstände vergleichen.

Volumenbetrachtung: Wir wollen n Punkte auf eine Fläche F setzten mit $\forall p, q \ : d(p,q) \geq 1$.
Um es leichter Abzuschätzen erlauben wir $d(p,q) \geq 0.5$ können wir das ganze mit disjunkten Kreisen beschreiben. Jedem Punkt gehört von einem solchen Kreis ein Viertel. $\Rightarrow$ Die Menge der Punkte pro Rechtekc ist beschänkt.

Aus diesen Betrachtungen können wir folgendes schließen:
Alle Punkte in P_L haben Mindestabstand δ.
Alle Punkte in P_R haben Mindestabstand δ.

Wie viele Punkte aus P_L können in dem Rechteck R leigen? Das Rechteck hat die Maße (δ, δ) und aus unserer Volumenbetrachtung folgt, dass jedem Punkt $\frac{1}{4}(\frac{\delta}{2})^2 \cdot \pi = \frac{3}{16}\delta^2$ Fläche gehört.
Das heißt, dass in der Fläche R_L maximal $\lfloor \frac{16}{3} \rfloor = 5$.

Die selbe Überlegung können wir für R_R anstellen.

$\Rightarrow$ Wir müssen pro Rechteck nur den Abstand zu 9 Nachfolgern berechnen. Dann nehmen wir immer das globale Minimum von δ.

Die Rechtecke können wir optimal Durchmustern, indem wir die Punkte im 2δ Bereich nehmen (geht leicht, da wir eine sortierte Liste haben) und diese nun auch aufsteigend nach y - Koordinate sortieren und jeweils 10 Punkte vergleichen.

Optimierung: Wir müssten nur in jedem Rekursionsschritt 2 mal Sortieren. Dies ist unnötig, da wir die Liste 2 mal Vorsortieren können und dann in jedem Schritt maximal lineare Zeit benötigen um die Listen für den nächsten Schritt zu gewinnen.
Im Falle der x-Kooridnaten brauchen wir bloß die Listen zu splitten. Im Falle der y-Koordinaten müssen wir einen umgekehrten Mergealgorithmus anwenden (der bekannte aus Mergesort).

Laufzeit:
Vorverarbeitung: $O(n) \cdot \log n$
Im Algorithmus: $O(n) = 2 \cdot T(\frac{n}{2}) = O(n \cdot \log n)$

4.3 Dynamisches Programmieren

Das dynamische Programmieren wurde schon einmal ein ALP III eingeführt. Im weiteren Sinne handelt es sich um eine Rekursion, in der schon einmal berechnete Rekursionswerte gecached werden oder alle möglichen Ergebnisse von klein nach groß vorberechnet werden.

4.3.1 Einführungsbeispiel: Einkaufsproblem

Wir haben ein Guthaben von $1,60$ und wollen aus den folgenden Waren einkaufen, so dass der Wert dieser Waren sich für uns maximiert.

Anzahl	Name	Preis	Wert
1x	Apfel	$0,40$€	3
1x	Brötchen	$0,60$€	9
1x	Buttermilch	$1,00$€	5
1x	Gummibärchen	$0,80$€	10
1x	Bifi	$0,60$€	6

Naiver Algorithmus:

Wir berechnen alle Teilmengen, deren Gewicht kleiner als 1,50 € ist. Dann suchen wir uns das Minimum der Werte herraus. Im Worstcase beduetet das für uns, dass wir alle Teilmengen berechnen müssen, was eine Laufzeit von 2^n zur Folge hat. Danach suchen wir uns das Minimum herraus. Für das Minimum müssen wir im schlimmsten Fall n Zahlen auf addieren.

Als Laufzeit erhalten wir insgesammt $T(n) = 2^n$ eine exponentielle Laufzeit.

Rekursiver Ansatz: Schränke das Problem auf Teilprobleme ein und setzte die teillösungen in Beziehung.

Wir definieren k_i als die Kosten von der Ware i und w_i als Wert der Ware i.
Als Rekursion nehmen wir:

$$\begin{array}{rrcl} \forall p & E[0,p] & = & 0 \\ \forall n & E[n,0] & = & 0 \\ & E[n,p] & = & \max\{E[n-1,p], E[n-1,p-k_n]\} \end{array}$$

Diese Gleichung beduetet für uns, dass wir uns in jedem Schritt entscheiden können, ob wir einen Artikel nehmen oder nicht. Nehmen wir ihn, müssen wir von unseren vorhandenen Geld die Kosten der Ware abziehen. Von diesen beiden Ergebnissen wählen wir einfach das Maximum um die Beste Möglichkeit zu erhalten.