Teile und Herrsche

Florian Gnepper Stephan Strehler

Hochschule Zittau/Görlitz

20.11.2014

Grundidee

Lösen eines großen Problems mittels Rekursion:

- zerlegen des Problems in viele Teilprobleme
- rekursives Lösen der Teilprobleme
- fertige Teillösungen wieder zu einem Ganzen zusammensetzen

Zielbestreben:

Einen effektiveren und damit schnelleren Algorithmus erzeugen, als jenen, der das Problem ohne Zerlegung löst.

Laufzeitbestimmung

Ausgangspunkt:

Gegeben ist ein Algorithmus A mit $T_A(n) \leq c \cdot n^2$. Er löst das Problem P in $\mathcal{O}\left(n^2\right)$.

Zerlegung des Problems:

Man nimmt einen Algorithmus B zur Lösung von P . Dieser löst 3 Teilprobleme von P mit der Größe von $\frac{n}{2}$ mittels A und fügt die Teillösungen mit einem Aufwand von $\mathcal{O}(n)$ zusammen.

Laufzeitbestimmung

Zusammensetzung der Formel:

$$T_B(n) = 3T_A\left(\lceil\frac{n}{2}\rceil\right) + d \cdot n$$

$$\leq 3 \cdot c \cdot \left(\frac{n+1}{2}\right) + d \cdot n$$

$$\leq \frac{3}{4}cn^2 + \left(\frac{3}{2}c + d\right)n + \frac{3}{4}c$$

Daraus folgt der kleinere Faktor im ersten Summanden ($\frac{3}{4}$ statt 1). Dieser ist allerdings zu vernachlässigen, da wenn n sehr groß ist ein großer Summand entsteht.

Laufzeitbestimmung

rekursiver Lösungsansatz:

Rekursive Lösung von P durch neuen Algorithmus C.
Die 3 Teilprobleme werden selbst wieder von C bearbeitet.
Nur Basisfälle ($n \leq n_0$) werden von A übernommen.
Aufwandsbeziehung:

$$T_C(n) = \begin{cases} T_A(n), & \text{wenn } n \leq n_0 \\ 3T_C\left(\frac{n}{2}\right), & \text{sonst} \end{cases}$$

Lösen der Gleichung durch Anwendung mathematischer Hilfsmittel:
$T_C(n) = \mathcal{O}\left(n^{1.59}\right)$

Laufzeitbestimmung

Schlussfolgerung:

- Effizienzsteigerung nur durch die Anwendung von Rekursion möglich.
- Rekursive Teilprobleme sind von gleicher Problemart, allerdings von kleinerer Problemgröße.
- Bei besonderen Problemen können auch mehrere Basisfälle auftreten, allerdings immer mindestens Einer.

Quicksort - Grundidee

1. Zerlege die Liste in 2 Teillisten
2. Nutze das dazu ein Vergleichselement (Pivot genannt)
3. Kleinere Elemente kommen in linke Liste, größere Elemente in rechte Liste
4. Rufe die Prozedur solange rekursiv auf, bis die neue Teilliste ein oder gar kein Element besitzt

Quicksort - Bezug auf Teile und Herrsche

- Zerlege das Problem in Teilprobleme
  ✓Teillisten werden erzeugt

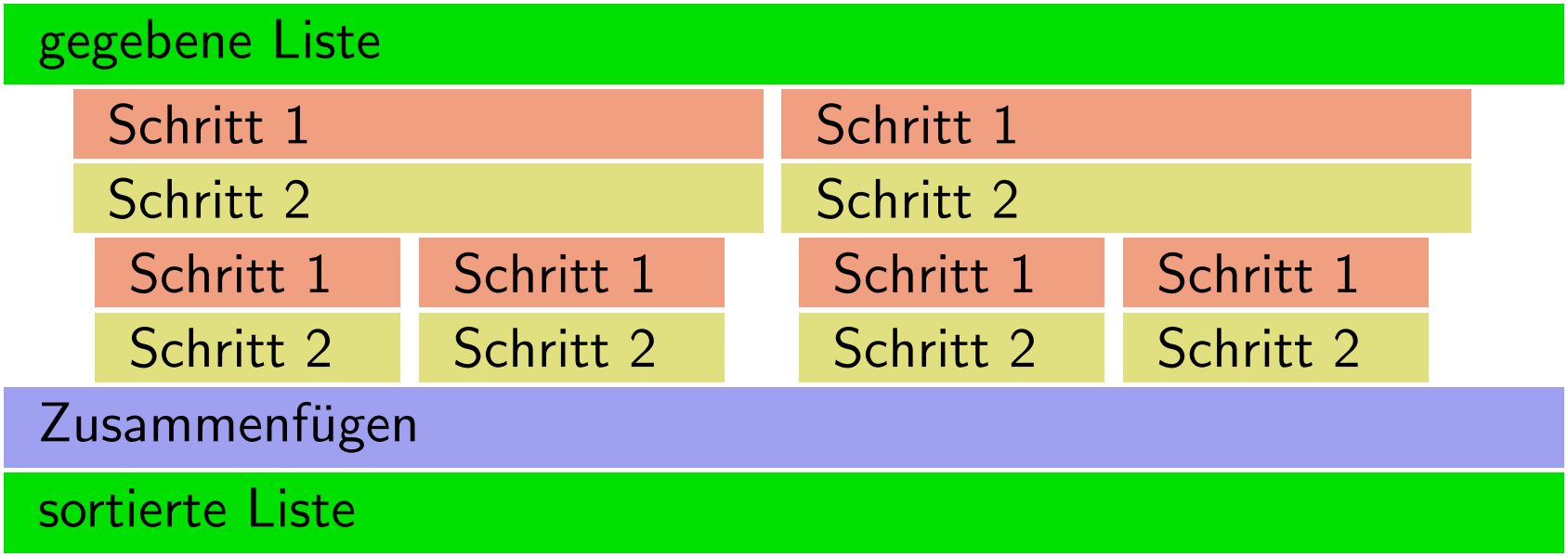
Quicksort - Bezug auf Teile und Herrsche

- Zerlege das Problem in Teilprobleme
  ✓Teillisten werden erzeugt
- Löse die Teilprobleme
  ✓Teillisten werden immer weiter zerlegt und sind am Ende gelöst

Quicksort - Bezug auf Teile und Herrsche

- Zerlege das Problem in Teilprobleme
  ✓Teillisten werden erzeugt
- Löse die Teilprobleme
  ✓Teillisten werden immer weiter zerlegt und sind am Ende gelöst
- Setze die Teillösungen wieder zusammen
  ✓Liste ist vollständig sortiert

Quicksort - Ablauf am Beispiel



Quicksort - Ablauf am Beispiel

gegebene Liste:	5	6	2	8	3	1	7	4
Schritt 1: Pivot wählen:	5	6	2	8	3	1	7	4

Quicksort - Ablauf am Beispiel

gegebene Liste:	5	6	2	8	3	1	7	4
Schritt 1: Pivot wählen:	5	6	2	8	3	1	7	4
Schritt 2: mit Pivot sortieren:	2	3	1	4	5	6	8	7

Quicksort - Ablauf am Beispiel

gegebene Liste:	5	6	2	8	3	1	7	4
Schritt 1: Pivot wählen:	5	6	2	8	3	1	7	4
Schritt 2: mit Pivot sortieren:	2	3	1	4	5	6	8	7
Schritt 1 für Teillisten:	2	3	1	4		6	8	7

Quicksort - Ablauf am Beispiel

gegebene Liste:	5	6	2	8	3	1	7	4
Schritt 1: Pivot wählen:	5	6	2	8	3	1	7	4
Schritt 2: mit Pivot sortieren:	2	3	1	4	5	6	8	7
Schritt 1 für Teillisten:	2	3	1	4		6	8	7
Schritt 2 für Teillisten:	1	2	3	4		6	8	7

Quicksort - Ablauf am Beispiel

gegebene Liste:	5	6	2	8	3	1	7	4
Schritt 1: Pivot wählen:	5	6	2	8	3	1	7	4
Schritt 2: mit Pivot sortieren:	2	3	1	4	5	6	8	7
Schritt 1 für Teillisten:	2	3	1	4		6	8	7
Schritt 2 für Teillisten:	1	2	3	4		6	8	7
Schritt 1 für Teillisten:	1		3	4			8	7
Schritt 2 für Teillisten:	1		3	4			7	8
Schritt 1 für Teillisten:				4			7	
Schritt 2 für Teillisten:				4			7	
sortierte Liste:	1	2	3	4	5	6	7	8

Quicksort - Pseudocode - Funktion quicksort

function quicksort (liste):
quicksort(liste, 0, liste.laenge-1) ;

funtion quicksort (liste, links, rechts):
if *rechts* > *links* **then**
    pivot= partition(liste, links, rechts);
    quicksort(liste,links,pivot-1);
    quicksort(liste, rechts, pivot+1);
end

Quicksort - Effizienz

best case

Best Case tritt dann auf, wenn

Quicksort - Effizienz

best case

Best Case tritt dann auf, wenn Teillisten gleich groß sind.

Elementarfall: $T(0) = 0$

Rekursive Gleichung: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$

2 Teillisten sind halb so groß wie Gesamtliste und entstehen beim Durchlauf der Gesamtliste.

Anwenden der Mastermethode (Buch Seite 53):

Quicksort - Effizienz

best case

Best Case tritt dann auf, wenn Teillisten gleich groß sind.

Elementarfall: $T(0) = 0$

Rekursive Gleichung: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$

2 Teillisten sind halb so groß wie Gesamtliste und entstehen beim Durchlauf der Gesamtliste.

Anwenden der Mastermethode (Buch Seite 53): Fall 2 trifft zu, damit gilt:

resultierender Aufwand: $T(n) = \mathcal{O}(n \log_2 n)$

Quicksort - Effizienz

Begründung des Ergebnisses aus Mastermethode:

vorliegende Gleichung $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$

Allgemein: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$

Quicksort - Effizienz

Begründung des Ergebnisses aus Mastermethode:

vorliegende Gleichung $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$

Allgemein: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$

Matching führt zu: $a = 2$ und $b = 2$

Daraus folgt: $f(n) = n^{\log_2 2} = n^1 = n$

Quicksort - Effizienz

worst case

Was wäre der schlimmste Fall der auftreten kann?

Quicksort - Effizienz

worst case

Was wäre der schlimmste Fall der auftreten kann?

$\Rightarrow$ Wenn Liste vollständig sortiert ist!

Was macht Quicksort?

Quicksort - Effizienz

worst case

Was wäre der schlimmste Fall der auftreten kann?

$\Rightarrow$ Wenn Liste vollständig sortiert ist!

Was macht Quicksort?

$\Rightarrow$ Quicksort nimmt Vergleichselement (1) und erzeugt Teillisten

$\Rightarrow$ linke Liste: 0 Elemente $\Rightarrow$ rechte Liste $(n-1)$ Elemente

Qicksort - Effizienz

worst case

Elementarfall: $T(0) = 0$

Qicksort - Effizienz

worst case

Elementarfall: $T(0) = 0$

Rekursionsgleichung: $T(n) = T(0) + T(n-1) + c \cdot n$

Qicksort - Effizienz

worst case

Elementarfall: $T(0) = 0$

Rekursionsgleichung: $T(n) = T(0) + T(n-1) + c \cdot n$

Anwenden der Iterationsmethode:

Übung: Wenden Sie die Iterationsmethode auf die gegebene Gleichung an.

Qicksort - Effizienz

worst case

Anwenden der Iterationsmethode Lösung:

$$
\begin{aligned}
T(n-1) &= T(n-2) + c(n-1) \\
T(n-2) &= T(n-3) + c(n-2) \\
&\dots \\
T(n-k) &= T(0) + c(n-k)
\end{aligned}
$$

Quicksort - Effizienz

worst case

rekursiver Gedanke $\Rightarrow$ jede Zeile in darüber stehende Zeile einsetzen:

$$
\begin{aligned}
T(n) &= c \cdot \sum_{k=0}^{n-1} (n-k) \\
&= c \cdot \left(\sum_{k=0}^{n-1} n - \sum_{k=0}^{n-1} k \right) \\
&= c \cdot \left(n^2 - \frac{n-1}{2} ((n-1)+1) \right) \\
&= c \cdot \left(\frac{n^2}{2} + \frac{n}{2} \right) \\
T(n) &= \mathcal{O}\left(n^2\right)
\end{aligned}
$$

Quicksort - Effizienz

average case

Ausgangspunkt: Jede Teillistenlänge (0 - n) kann mit einer Wahrscheinlichkeit von $\frac{1}{n}$ auftreten:

Es gilt somit: $T(i) = T(n-i-1) = \frac{1}{n}\sum_{j=0}^{n-1} T(j)$ in

$$T(n) = \underbrace{T(i)}_{linke\,Teilliste} + \underbrace{T(n-i-1)}_{rechte\,Teilliste} + \underbrace{c \cdot n}_{Teillistenbildung}$$

Zusammensetzen:

$$T(n) = \frac{2}{n}\sum_{j=0}^{n-1} T(j) + c \cdot n$$

Quicksort - Effizienz

average case

Beseitigung der Summe:

$$nT(n) = 2\sum_{j=0}^{n-1} T(j) + c \cdot n^2$$

n durch (n-1) ersetzen:

$$(n-1)\,T(n-1) = 2\sum_{j=0}^{n-2} T(j) + c \cdot (n-1)^2$$

$$nT(n) - (n-1)\,T(n-1) = 2T(n-1) + 2cn - c$$

$\Rightarrow$ - c entfällt wenn n $\rightarrow \infty$

Quicksort - Effizienz

average case

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{n-1}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\dots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

$$\frac{T(n)}{n+1} = \underbrace{\frac{T(1)}{2}}_{=0, da\, T(1)=0} + 2c \sum_{i=3}^{n+1} \frac{1}{i}$$

Quicksort - Effizienz

average case

Benutzung der bekannten Schranken $\frac{\lfloor \log_2 n \rfloor + 1}{2} < H_n \leq \lfloor \log_2 n \rfloor + 1$ für harmonischen Zahlen $H_n = \sum_{k=1}^{n} \frac{1}{k}$ ergibt sich:

$$\frac{T(n)}{n+1} = \mathcal{O}(\log_2 n) \Rightarrow T(n) = \mathcal{O}(n \log_2 n)$$

Interne Suchverfahren

Forderungen:

Bei einem internen Suchverfahren darf kein zusätzlicher Speicher verwendet werden.

Interne Suchverfahren

Forderungen:

Bei einem internen Suchverfahren darf kein zusätzlicher Speicher verwendet werden.

Problem:

Durch das erstellen von 2 neuen Teillisten wird zusätzlicher Speicher benötigt.

Interne Suchverfahren

Forderungen:

Bei einem internen Suchverfahren darf kein zusätzlicher Speicher verwendet werden.

Problem:

Durch das erstellen von 2 neuen Teillisten wird zusätzlicher Speicher benötigt.

Lösung:

Verwendung von einem Vektor anstatt eines Arrays.

Verfahren mit einem Vektor e

- Pivot p = erstes Vektor-Element

Verfahren mit einem Vektor e

- Pivot p = erstes Vektor-Element
- Linker Zeiger l steht auf zweiten Element
- Rechter Zeiger r steht auf letztem Element

Verfahren mit einem Vektor e

- Pivot p = erstes Vektor-Element
- Linker Zeiger l steht auf zweiten Element
- Rechter Zeiger r steht auf letztem Element
- l wird so lange nach rechts verschoben bis Element unter $l > p$
- r wird so lange nach links verschoben bis Element unter $r < p$

Verfahren mit einem Vektor e

- Pivot p = erstes Vektor-Element
- Linker Zeiger l steht auf zweiten Element
- Rechter Zeiger r steht auf letztem Element
- l wird so lange nach rechts verschoben bis Element unter $l > p$
- r wird so lange nach links verschoben bis Element unter $r < p$
- Wenn danach $l \leq r$, dann werden $e[l]$ und $e[r]$ miteinander vertauscht und Zeigerwanderung fortgesetzt. Sonst wird $e[l]$ und $e[r]$ miteinander vertauscht und Zeigerwanderung eingestellt.

Verfahren mit einem Vektor e

- Pivot p = erstes Vektor-Element
- Linker Zeiger l steht auf zweiten Element
- Rechter Zeiger r steht auf letztem Element
- l wird so lange nach rechts verschoben bis Element unter $l > p$
- r wird so lange nach links verschoben bis Element unter $r < p$
- Wenn danach $l \leq r$, dann werden $e[l]$ und $e[r]$ miteinander vertauscht und Zeigerwanderung fortgesetzt. Sonst wird $e[l]$ und $e[r]$ miteinander vertauscht und Zeigerwanderung eingestellt.
- Ergebnis: Das Listenelement $e[0]$ (Pivotelement) steht nun an richtiger Stelle.

Mergesort - Grundidee

1. Zerlege die zu sortierende Liste in 2 gleich große Teillisten l1 und l2. Dabei sortiere nichts vor. Die Länge der Teillisten beträgt entweder $l = \frac{n}{2}$ oder $l = \frac{n}{2} + 1$ im ganzzahligen Bereich.
2. Sortiere die Teillisten rekursiv. Dabei gelten die Elementarfälle von Quicksort.
3. Füge die sortierten Teillisten nach dem Reißverschlussprinzip wieder zusammen:
   Wenn l1[0] $<$ l2[0], dann füge das l1[0] in die Liste ein und entferne es aus Warteschlange. Andere Elemente von l1 rücken nach.
   Bemerkung: Gleiches Prinzip gilt auch für l2[0], wenn es das Kleinere ist.

Mergesort - Bezug auf Teile und Herrsche

- Zerlege das Problem in Teilprobleme
  ✓Teillisten werden erzeugt

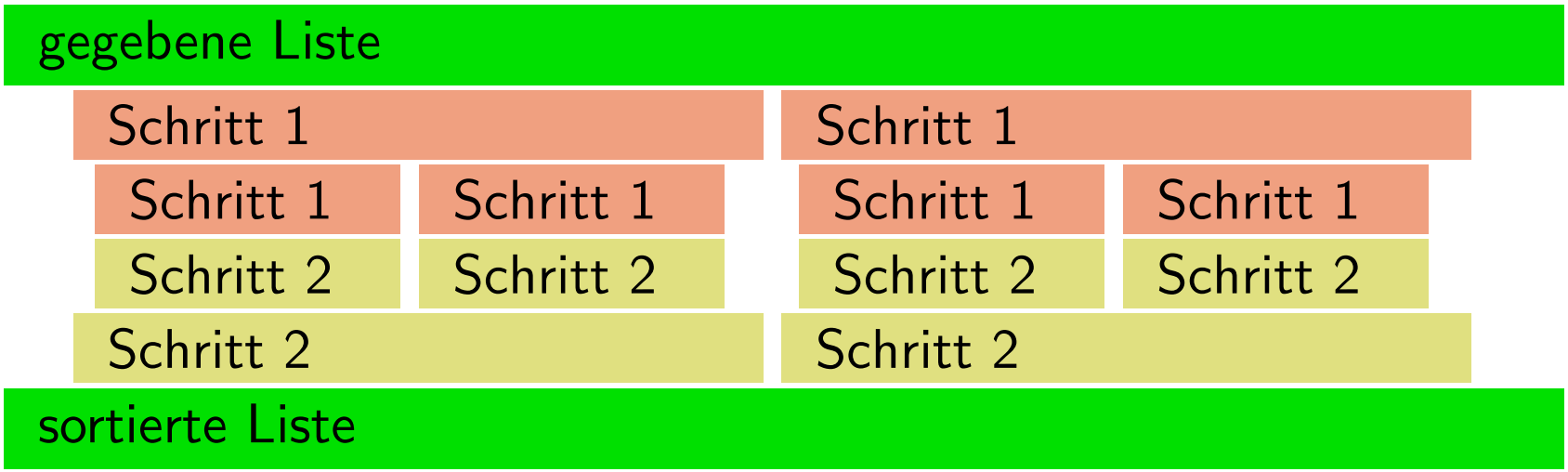
Mergesort - Bezug auf Teile und Herrsche

- Zerlege das Problem in Teilprobleme
  ✓Teillisten werden erzeugt
- Löse die Teilprobleme
  ✓Teillisten werden immer weiter zerlegt und sind am Ende gelöst

Mergesort - Bezug auf Teile und Herrsche

- Zerlege das Problem in Teilprobleme
  ✓Teillisten werden erzeugt
- Löse die Teilprobleme
  ✓Teillisten werden immer weiter zerlegt und sind am Ende gelöst
- Setze die Teillösungen wieder zusammen
  ✓Liste ist vollständig sortiert

Quicksort - Ablauf am Beispiel



Mergesort - Ablauf am Beispiel

Beispiel:	5	6	2	8	3	1	7	4
Schritt 1: Listen teilen:	5	6	2	8	3	1	7	4

Mergesort - Ablauf am Beispiel

Beispiel:	5	6	2	8	3	1	7	4
Schritt 1: Listen teilen:	5	6	2	8	3	1	7	4
Schritt 1:	5	6	2	8	3	1	7	4

Mergesort - Ablauf am Beispiel

Beispiel:						5	6	2	8	3	1	7	4		
Schritt 1: Listen teilen:				5	6	2	8			3	1	7	4		
Schritt 1:			5	6		2	8			3	1		7	4	
Schritt 1:		5		6		2		8		3		1		7	4

Mergesort - Ablauf am Beispiel

Beispiel:							5	6	2	8	3	1	7	4	
Schritt 1: Listen teilen:				5	6	2	8		3	1	7	4			
Schritt 1:			5	6		2	8		3	1		7	4		
Schritt 1:		5		6		2		8		3		1		7	4
Schritt 2: Reißverschluss:		5		6		2		8		1		3		4	7

Mergesort - Ablauf am Beispiel

Beispiel:							5	6	2	8	3	1	7	4	
Schritt 1: Listen teilen:				5	6	2	8		3	1	7	4			
Schritt 1:			5	6		2	8		3	1		7	4		
Schritt 1:		5		6		2		8		3		1		7	4
Schritt 2: Reißverschluss:		5		6		2		8		1		3		4	7
Schritt 2:				2	5	6	8		1	3	4	8			

Mergesort - Ablauf am Beispiel

Beispiel:							5	6	2	8	3	1	7	4	
Schritt 1:	Listen teilen:			5	6	2	8		3	1	7	4			
Schritt 1:			5	6		2	8		3	1		7	4		
Schritt 1:		5		6		2		8		3		1		7	4
Schritt 2:	Reißverschluss:		5	6		2	8		1	3		4	7		
Schritt 2:				2	5	6	8		1	3	4	8			
Schritt 2:					1	2	3	4	5	6	7	8			

Mergesort - Ablauf am Beispiel

Beispiel:							5	6	2	8	3	1	7	4	
Schritt 1: Listen teilen:				5	6	2	8		3	1	7	4			
Schritt 1:			5	6		2	8		3	1		7	4		
Schritt 1:		5		6		2		8		3		1		7	4
Schritt 2: Reißverschluss:		5	6		2	8		1	3		4	7			
Schritt 2:			2	5	6	8		1	3	4	8				
Schritt 2:				1	2	3	4	5	6	7	8				
sortierte Liste:				1	2	3	4	5	6	7	8				

Mergesort - Pseudocode - Funktion mergesort

```
funtion mergesort ( liste ):
if liste.laenge ≤1 then
    return liste;
else
    linkeListe = liste[0] bis liste[laenge/2];
    rechteListe = liste[laenge-(laenge/2)] bis liste[laenge-1];
    linkeListe =mergesort(linkeListe);
    rechteListe=mergesort(rechteListe);
    return merge(linkeListe, rechteListe);
end
```

Mergesort - Laufzeiten

Elementarfälle:

$$T(0) = 0$$
$$T(1) = 0$$

Rekursionsgleichung:

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$

Durch Rückführung auf Elementarfälle [1]:

$T(n) = \mathcal{O}(n \log_2 n)$

Dies gilt für den best-, worst- und average case.

[1] Es wird angenommen, dass n eine Zweierpotenz ist

Vergleich von Qicksort und Mergesort

Verfahren:

- beide Verfahren arbeiten rekursiv
- beide Verfahren haben gleiche Elementarfälle
- beide Verfahren sind interne Sortierverfahren

Effizienz:

case	Quicksort	Mergesort
best	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$
average	$\mathcal{O}(n \log_2 n)$	$\mathcal{O}(n \log_2 n)$
worst	$\Omega(n^2)$	$\Omega(n \log_2 n)$

Suchen - Grundsätzliches

Grundsätzliches:

- neben Sortieren auch Suchen sehr wichtig
- Ziel: aus Liste ein Element herausfiltern
- Liste kann sortiert oder auch unsortiert sein

Suchen - Grundsätzliches

Grundsätzliches:

- neben Sortieren auch Suchen sehr wichtig
- Ziel: aus Liste ein Element herausfiltern
- Liste kann sortiert oder auch unsortiert sein

Frage:

Was hat die Sortierungsgrad der Liste für Konsequenzen auf das Suchen von einzelnen Elementen?

Binäres Suchen - Herleitung der Idee

Algorithmus bei unsortierter Liste:

- Liste wird so lange durchlaufen bis...
  - ... Element gefunden wurde.
  - ... Liste vollständig durchlaufen wurde.
- Damit ergbibt sich die Effizienz (worst case): $\mathcal{O}(n)$

Algorithmus bei sortierter Liste:

- Liste wird so lange durchlaufen bis...
  - ... Element gefunden wurde.
  - ... zu vergleichendes Element größer gesuchtes Element ist.
- Damit ergibt sich die Effizienz (worst case): $\mathcal{O}(n)$. Allerdings kürzt sich die Average Time.

Grundidee

Gegeben:

- sortierte Liste
- Überlegungen über Teile und Herrsche

Überlegungen:

- Zerlegung der sortierten Liste in 2 Teile,
- Vergleichen des gesuchten Elements mit dem ersten Element der rechten Liste
  - Wenn größer, Element ist vlt. Teil der rechten Liste
  - Wenn kleiner, Element ist vlt. Teil der linken Liste
  - Wenn gleich, dann Element gefunden
- rekursiver Aufruf der Liste, bei der Chance besteht Element zu besitzen

Pseudocode:

```
function binaeresSuchen (liste, element):
mitte = liste.laenge/2;    linkeListe = liste[0 bis mitte];
rechteListe= liste[mitte+1 bis liste.laenge-1];
if element=rechteListe[0] then
    Ausgabe:„Element in Liste vorhanden";
else
    if element <rechteListe[0] then
        if linkeListe.laenge >1 then
            binaeresSuchen(linkeListe, element);
        else
            Ausgabe: „Element nicht in Liste vorhanden!";
        end
    else
        if rechteListe.laenge >1 then
            binaeresSuchen(rechteListe, element);
        else
            Ausgabe: „Element nicht in Liste vorhanden!";
        end
    end
end
```

Binäres Suchen - Effizienzbetrachtung

- Mit jedem Schritt wird die Größe der Liste halbiert.
  ✓Besonders wenn Startlänge der Liste Zweierpotenz ist
- Damit Übereinstimmung Anzahl rekursiver Aufrufe mit Anzahl der Halbierungen ($\log_2 n$)
- Jeder Vergleich (pro Schritt) benötigt einen Aufwand von $\mathcal{O}(c)$
- Gesamtaufwand für mittleren und schlechtesten Fall: $T(n) = \mathcal{O}(\log_2 n)$
- Best-Case tritt dann auf, wenn beim ersten Durchlauf Element gefunden wird: $T(n) = \mathcal{O}(c)$. Dies entspricht einem konstanten Aufwand.

Vergleich binäres Suchen mit sequenzielles Suchen

sequenzielle Suche

- Liste G ist aufsteigend sortiert
- Sucht solange bis E gefunden, oder zu vergleichendes Element aus Liste $<$ E
- Best Case: E ist erstes Element: $T(n) = \mathcal{O}(1)$
- Worst Case: E nicht vorhanden: $T(n) = \mathcal{O}(n)$
- Average Case: Wahrscheinlichkeit auf E $= \frac{1}{n}$ so ergibt sich:

$$T(n) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n}{2n}(n+1) = \mathcal{O}(n)$$

✓Schlussfolgerung: binäre Suche ist besser als sequenzielle Suche

Multiplikation großer Zahlen - Vorstellung

Zielsetzung:

- Möglichst schnelleres Verfahren zur Multiplikation entwickeln
- Idee Multiplikation (quadratischer Aufwand) durch Addition und Verschiebung (linearer Aufwand) ersetzen
- Entwickelt von Karatsuba und Ofman 1960

Bezug auf Teile und Herrsche:

- Vorgaben werden in Teilprobleme zerlegt
- ... Teilprobleme werden gelöst
- ... und zu einer Lösung wieder zusammengesetzt.

Grundsätzliche Gedanken:

Voraussetzungen:

- Beide Zahlen (x,y) haben die gleiche Anzahl an Ziffern n
- n muss gerade sein
- Beide Zahlen werden in einen High- und in einen Low-Teil unterteilt
  - $x \rightarrow x_1, x_0$
  - $y \rightarrow y_1, y_0$
- Die Basis b stammt üblicherweise aus dem dezimalen Zahlensytem

Gleichungen:

$$x = x_1 b^{\frac{n}{2}} + x_0$$
$$y = y_1 b^{\frac{n}{2}} + y_0$$

Multiplikation großer Zahlen - fertige Formel

Formel

$$x \cdot y = (x_1 \cdot y_1)\, b^n + (x_0 \cdot y_1 + y_0 \cdot x_1)\, b^{\frac{n}{2}} + x_0 \cdot y_0$$

Verfahren

- Multiplikation kann in 4 gleichartige Multiplikationen zweier $\frac{n}{2}$ großer Zahlen plus 3 Additionen von zwei Zahlen, welche nicht größer als n-stellig sind, zerlegt werden
- Aufwand für Addition: $\Theta(n)$
- Gesamtaufwand: $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$
  $\Rightarrow$ Anwendung der Meistermethode $T(n) = \mathcal{O}\left(n^2\right)$

Multiplikation großer Zahlen - Verbesserung

Vergleich der Effizienz:

Durch Teile und Herrsche hat man nur gleiche Effizienz erreicht wie bei Schulmethode. $\Rightarrow$ inaktzeptabel.

Verbesserung des Algorithmus:

Ersetzen von $x \cdot y$ durch

$$x_0 \cdot y_1 + y_0 \cdot x_1 = x_0 \cdot y_1 + y_0 \cdot x_1 - (x_0 - x1) \cdot (y_0 - y_1)$$

Ergebnis:

$$x \cdot y = x_1 \cdot y_1 b^n + (x_1 \cdot y_1 + x_0 \cdot y_0 - (x_0 - x_1) \cdot (y_0 - y_1)) \, b^{\frac{n}{2}} + x_0 \cdot y_0$$

Vereinfachen: $p = x_0 \cdot y_0$, $q = x_1 \cdot y_1$ Aufwand dafür jeweils: $T\left(\frac{n}{2}\right)$

Multiplikation großer Zahlen - Verbesserung

verbesserte Formel:

$$x \cdot y = qb^n + \left(q + p - (x_0 - x_1) \cdot (y_0 - y_1)\right) b^{\frac{n}{2}} + p$$

Hinzu kommen zwei Additionen von je zwei $\frac{n}{2}$-stelliger Zahlen sowie deren Multiplikation mit einem Aufwand von $\Theta(n)$.

Gesamtaufwand:

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

Multiplikation großer Zahlen - Verbesserung

verbesserte Formel:

$$x \cdot y = q b^n + \left(q + p - (x_0 - x_1) \cdot (y_0 - y_1)\right) b^{\frac{n}{2}} + p$$

Hinzu kommen zwei Additionen von je zwei $\frac{n}{2}$-stelliger Zahlen sowie deren Multiplikation mit einem Aufwand von $\Theta(n)$.

Gesamtaufwand:

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

Lösen der Rekursionsgleichung:

Übung: Wenden Sie die Meistermethode an!

Multiplikation großer Zahlen - Lösen der Rekursionsgleichung

Allgemein: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$
Anwendungen der Mastermethode: $a = 3$ und $b = 2$
Erste Fall wird angewendet. Damit ergibt sich:

$$T(n) = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.585}\right)$$

Schnelle Matrixmultiplikation - Problemaufnahme

Schulmethode:

Multiplikation zweiter Matrizen vom Typ (n, n) fordert Aufwand von $\mathcal{O}\left(n^3\right)$

Idee nach Volker Strassen:

1. Zerlegung der Matrizen in Kleinere
2. Lösen der Kleineren
3. Addition der kleineren Matrizen wieder zu einer Großen

Schnelle Matrixmultiplikation - Grundaufbau Matrix

Einfache Matrix (Teilmatrizen farbig):

$$X = \begin{pmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} \end{pmatrix}$$

Matrixmultiplikation (Teilmatrizen farbig):

$$C = A \cdot B = \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{pmatrix}$$

Schnelle Matrixmultiplikation - Teile und Herrsche

Entstehung von C:

$$C = A \cdot B = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Teilmatrizen in Gleichungsform:

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

Schnelle Matrixmultiplikation - Effizienz

Einzelrechnungen:

8 Matrizenmultiplikationen: $8 \cdot T\left(\frac{n}{2}\right)$
4 Matrizenadditionen: $\mathcal{O}\left(n^2\right)$

Gesamtaufwand:

$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}\left(n^2\right)$ ergibt $T(n) = \mathcal{O}\left(n^3\right)$
$\Rightarrow$ keine Verbesserung zu Schulmethode. Neuorganisation der Gesamtrechnung nötig.

Schnelle Matrixmultiplikation - Neuorganisation

7 Matrizen berechnen:

$$
\begin{aligned}
M_1 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \\
M_2 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\
M_3 &= (A_{1,2} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\
M_4 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
M_5 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\
M_6 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\
M_7 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1}
\end{aligned}
$$

Schnelle Matrixmultiplaktion - Neuorganisation

Teilmatrizen berechnen:

$$C_{1,1} = M_2 + M_6 - M_4 + M_1$$
$$C_{1,2} = M_4 + M_5$$
$$C_{2,1} = M_6 + M_7$$
$$C_{2,2} = M_2 - M_7 + M_5 + M_3$$

Gesamtaufwand:

7 Matrizenmultiplationen, 18 Additionen:
$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}\left(n^2\right)$
Gelöst: $T(n) = \mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}\left(n^{2.81}\right)$
$\Rightarrow$ Verbesserung der Effizienz.

Schnelle Matrixmultiplikation - Schlussfolgerung

- Bei sehr großen Matrizen ist der Strassen-Algorithmus der Schulmethode überlegen. Allerdings nicht, wenn viele Nullen in den Matrizen existieren. Außerdem leidet die Stabilität bei Gleitkommazahlen.
  Trotzdem ist der Algorithmus ein Meilenstein.
- Winograd und Coppersmith entiwckelten 1986 einen weiteren Algorithmus zur Berechnung von Matrizenprodukten. Dieser hat einen Aufwand von $\mathcal{O}\left(n^{2.376}\right)$, bringt allerdings weitere Nachteile in der praktischen Nutzung mit sich.