

Algorithmen und Datenstrukturen

Sommersemester 2020

Stefan Näher

Universität Trier

naeher@uni-trier.de

Vorlesung 3

29. April 2020

Informationen von Moritz Gobbert

- Donnerstags oder Freitags lade ich im Stud.IP das Übungsblatt hoch
- Abgabetermin ist immer Montag der jeweils übernächsten Woche (so, dass mehr als 7 Tage Bearbeitungszeit sind)
- Abzugeben sind die Blätter dann auch über Stud.IP (ich erstelle dort einen Ordner für die jeweils abzugebende Übung)
- Wir korrigieren die Blätter so schnell wie möglich und schicken die Korrekturen (per Mail) zurück. Wenn die Abgaben als PDF kommen werden wir die PDFs selbst mit Kommentaren versehen (die Funktion haben die meisten PDF-Reader ja), bei Bildern werden wir auf Textdateien zurückgreifen

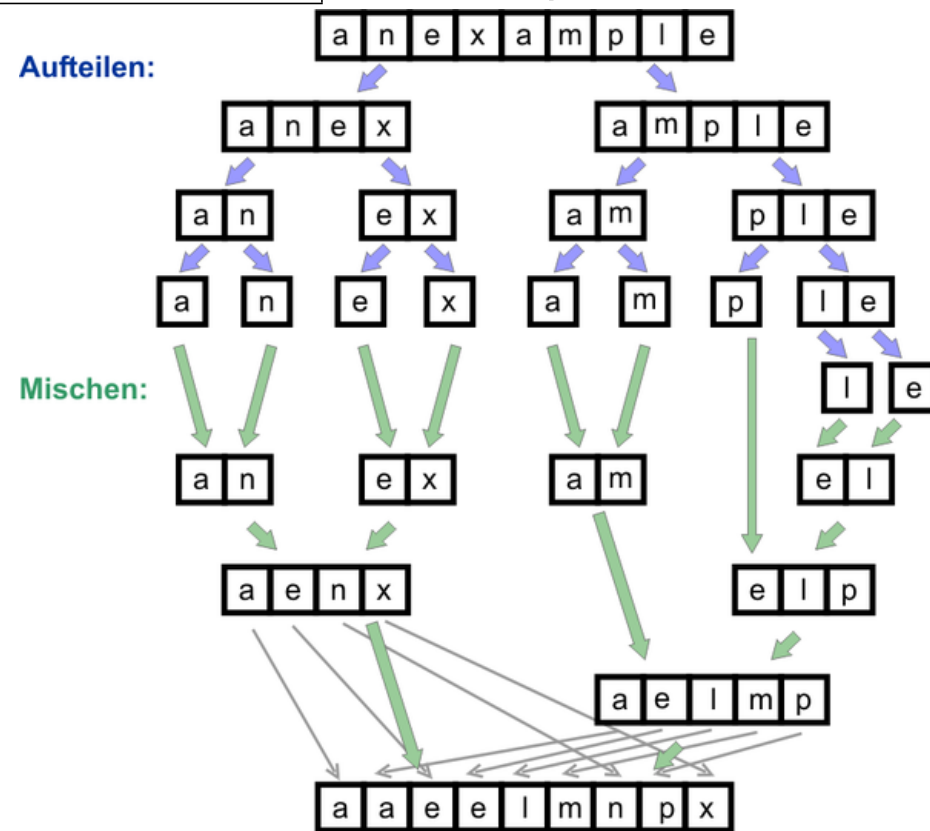
- Irgendwann zwischen offiziellem Abgabetermin und Verschicken der Korrekturen lade ich die Musterlösung zum Blatt hoch
- Die Sprechstunden finden Montags von 14-16, Mittwochs von 14-16 und Freitags von 10-12 statt. Die sind allerdings nur für Fragen zu den Übungsblättern (oder zu anderen inhaltlichen/organisatorischen Dingen) gedacht. Das Vorrechnen der Lösung entfällt, da ich die Musterlösung hochlade.
- Die Sprechstunden finden über BigBlueButton statt. (Ich glaube Norbert hat das schon alles eingerichtet.) Ich werde das dazugehörige Passwort noch per Mail rumschicken.

MERGESORT (Wiederholung)

1. MERGESORT(ℓ, r)
2. **if** $\ell \geq r$ **then return** ; //Verankerung: Feld der Länge ≤ 1 (nichts zu tun)
3. $m \leftarrow \lfloor (\ell + r)/2 \rfloor$
4. MERGESORT(ℓ, m);
5. MERGESORT($m + 1, r$);
6. **MERGE**(ℓ, m, r);

Die eigentliche Arbeit steckt in der Prozedur **MERGE**. Sie ‘mischt’ die beiden sortierten Teilfelder $A[\ell, m]$ und $A[m + 1, r]$ zu einem sortierten Feld $A[\ell, r]$.

Sortieren durch Mischen: Ein Beispiel



Die Arbeitsweise von MERGE [hier Skizze einfügen]

Übungsaufgabe: Ausformulierung des Algorithmus im PseudoCode.

Laufzeitabschätzung

Es sei $T(n)$ eine obere Schranke für die Laufzeit von **MERGESORT** auf Feldern der Größe n . Auf Feldern der Größe 1 braucht der Algorithmus offenbar konstante Zeit, d.h., $T(1) = c_0$.

Wenn wir annehmen, dass **MERGE** Laufzeit $O(n)$ hat, dann gilt für $n > 1$

$$T(n) \leq c \cdot n + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) \text{ für eine geeignete Konstante } c \geq 1.$$

Sind wir nur am asymptotischen Wachstum von T interessiert, können wir $c = 1$ und $c_0 = 1$ annehmen. Außerdem können wir $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ jeweils durch $n/2$ ersetzen - **warum ?**

Wie löst man aber eine **Rekursionsgleichung** der Form

$$T(n) \leq c \cdot n + 2 \cdot T(n/2)$$

Analyse von Rekursionen

Ziel: Überführen in eine geschlossene, nichtrekursive Form

Substitutionsmethode:

Anhand der Rekursionsgleichung wird eine Lösung geraten und durch Induktion bestätigt (konstruktive Induktion).

Iterationsmethode:

Wird die Funktion auf der rechten Seite bis zur Verankerung iteriert, d.h. immer wieder durch die Rekursionsgleichung ersetzt.

Rekursive Algorithmen

Divide and Conquer: Teile und herrsche

Viele Algorithmen zerlegen das zu lösende Problem einer Größe in 2 oder mehrere Teilprobleme mit einer geringeren Größe.

Die Teilprobleme werden (rekursiv) berechnet und die Lösung aus den Ergebnissen zusammengesetzt. Die Laufzeit für eine Eingabe der Länge n setzt sich also zusammen aus der Laufzeit für die kleineren Eingaben plus der Laufzeit die benötigt, wird die Eingabe zu zerlegen und die Lösung zusammenzufügen.

DIVIDE_AND_CONQUER(x)

if $|x| \leq c$ **then**

// Verankerung

berechne Lösung L für x direkt

else

// Teile

zerlege x in zwei (gleichgroße) Teile x_1 und x_2

// Herrsche

$L_1 = \text{DIVIDE_AND_CONQUER}(x_1)$

$L_2 = \text{DIVIDE_AND_CONQUER}(x_2)$

// Zusammenfügen

berechne Lösung L für x aus Teilösungen L_1 und L_2

fi

Beispiel: MERGESORT

Analyse Divide and Conquer (MERGESORT)

Sei der Aufwand für Teilen und Zusammenfügen $c \cdot n$ für eine Konstante $c > 0$ (d.h. $\mathcal{O}(n)$).

Dann gilt für die Gesamtlaufzeit für ein Problem der Größe n
 $T(n) \leq 2 \cdot T(n/2) + c \cdot n$

Wir verwenden die *Substitutionmethode*.

Behauptung: $\exists$ Konstante $c_1 > 0$ mit $T(n) \leq c_1 \cdot n \log n$ d.h. $T(n) = \mathcal{O}(n \log n)$

Beweis durch Substitution (Induktion)

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + c \cdot n \\ &\leq 2 \cdot c_1 \cdot n/2 \cdot \log(n/2) + c \cdot n && \text{Induktionsannahme} \\ &= c_1 \cdot n \cdot (\log n - 1) + c \cdot n \\ &= c_1 \cdot n \log n - c_1 \cdot n + c \cdot n \\ &= c_1 \cdot n \log n + (c - c_1) \cdot n \\ &\leq c_1 \cdot n \log n \quad \text{für } c_1 \geq c \end{aligned}$$

Die Behauptung gilt also für alle $c_1 \geq c$.

Iterationsmethode

Beispiel: $T(n) = n + 3T(n/4)$

Iteriertes Einsetzen und Ersetzen der Summe ergibt:

$$\begin{aligned} T(n) &= n + 3f(n/4) = n + 3(n/4 + 3f(n/16)) \\ &= n + 3n/4 + 9(n/16 + 3f(n/64)) \leq n + 3n/4 + 9n/16 + \dots \\ &\leq n \cdot \sum_{i \geq 0}^{\infty} (3/4)^i \end{aligned}$$

Geometrische Reihe

$$\sum_{i \geq 0}^{\infty} x^i = \frac{1}{1-x} \quad \text{für } x < 1$$

Für $\alpha = 3/4$ folgt

$$T(n) \leq n \cdot \frac{1}{1 - 3/4} = 4n = \mathcal{O}(n)$$

Die Verankerung $T(1) = c$ wirkt sich nur auf die in der $\mathcal{O}$ -Notation verborgenen Konstanten aus.

Hauptsatz der Laufzeitfunktionen (Master Theorem)

Allgemeine Form rekursiver Laufzeitfunktionen

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

Hierbei sind $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ bezeichnet eine von $T(n)$ unabhängige und nicht negative Funktion.

Interpretation:

a : Anzahl der Unterprobleme in der Rekursion

n/b : Größe der Unterprobleme

$f(n)$: Kosten von Teilen und Zusammenfügen

Im Beispiel für die Iterationsmethode ($T(n) = n + 3T(n/4)$)

$a = 3$, $b = 4$ und $f(n) = n$.

Datentypen

Alle gängigen Programmiersprachen bieten Typen wie “ganze Zahlen”, “reelle Zahlen”, “boolean” an.

Auf diesen Typen gibt es Operationen wie “Addition”, “logisches UND”,

Auf der Mathematik- bzw. Algorithmik-Ebene werden wir uns (meist) im Folgenden der üblichen mathematischen Notation bedienen und schreiben $x : \mathbb{Z}$, wenn wir ausdrücken wollen, dass die Variable x ganze Zahlen enthalten kann.

Dabei / dadurch abstrahieren wir (meist) von Überlaufproblemen etc.

Elementare Datenstrukturen

bauen auf (*nicht notwendigerweise elementaren*) *Datentypen* auf, z.B. kann ein **Feld** (engl.: **Array**) A von 20 reellen Zahlen durch $A : \mathbb{R}[1..20]$ eingeführt (**dekla-riert**) werden (womit auch der Indexbereich klar ist).

Als einzige Operation gestattet ein Feld den **Zugriff** (lesend und schreibend) auf einzelne Elemente durch **Indizierung**.

Beispiel: $A[8]$ greift auf das achte Element von A zu.

Vereinfachend gestatten wir auch den Zugriff auf ganze Bereiche des Feldes, z.B. durch $A[3..5]$.

In unserer Vorstellung liegen die Elemente eines Feldes nacheinander im Speicher des Rechners.

Man beachte die Ähnlichkeit mit dem “Speichermodell” einer RAM.

Einfache Datenstrukturen

Keller (Stack)

Schlangen (Queue)

Listen

Ein **statischer Keller** wird beschrieben durch:

Maximalgröße n ;

Datentyp D der Kellerelemente (hier meistens Integers)

Feld $A[1..n]$

Index des obersten Kellerelements: **top** $\in \{0, \dots, n\}$

Hierbei $\text{top} = 0$ gdw. Keller ist leer.

Kellerspeicher arbeiten nach dem LIFO-Prinzip: Last-In / First-Out.

Keller-Operationen

S.clear()	$\text{top} \leftarrow 0;$
S.empty()	return (top = 0);
S.push(x)	$\text{top} \leftarrow x + 1; A[\text{top}] \leftarrow x;$
S.pop()	$x \leftarrow A[\text{top}]; \text{top} \leftarrow \text{top} - 1; \text{return } x$
S.top()	return A[top];

Achtung: Überlauf / Unterlauf (Overflow / Underflow) bislang unbeachtet.
Dafür könnte man (Test-)Operationen bzw. Exceptions einführen.

Bemerkungen:

- (1) Alle Operationen brauchen Zeit $\mathcal{O}(1)$
- (2) top zeigt immer auf das oberste Kellerelement, sofern vorhanden.
- (3) Dynamische Keller später $\leadsto$ Listen.

Keller-Operationen und ihre Eigenschaften

1. Nach $S.\text{clear}()$ ist $S.\text{empty}()$ stets wahr.
2. Tritt kein Überlauf auf, so gilt: $S.\text{top}(S.\text{push}(x)) = x$.
3. $x \leftarrow S.\text{top}()$ ist äquivalent zu: $x \leftarrow S.\text{pop}(); S.\text{push}(x)$.
4. Ist der Keller nicht leer, so gilt: $S.\text{push}(S.\text{pop}())$ verändert den Keller nicht.

Beobachte: Auf die “Implementierung” des Kellers als Feld wird hier nicht mehr Bezug genommen. $\leadsto$ Alternativer (abstrakterer, mathematischerer) Ansatz über *abstrakte Datentypen / Algebren*

Ein **(statische) Schlange oder Queue** wird beschrieben durch:

Maximalgröße n ;

Datentyp D der Queue-Elemente (hier meistens Integers)

Feld $A[1..n + 1]$

Index des ersten und letzten Schlangen-Elements: **first** und **last**

Achtung:

Das Feld A muss zyklisch betrachtet werden $\leadsto$ Übung.

Schlangen arbeiten nach dem FIFO-Prinzip: First-In / First-Out.

Schlangen-Operationen

S.clear() S.empty() S.append(x) S.pop() S.top()	
---	--