

Algorithmen und Datenstrukturen

Sommersemester 2020

Stefan Näher

Universität Trier

naeher@uni-trier.de

Vorlesung 8

14. Mai 2020

Diskussion der Sortierverfahren

Wir haben bis jetzt zwei Verfahren kennengelernt, die eine Laufzeit von $\mathcal{O}(n \cdot \log(n))$ versprechen: Mergesort (ganz zu Anfang) und Heapsort.

Alle übrigen Verfahren (Sortieren durch Einfügen oder Auswahl, Bubblesort bzw. Quicksort) haben Laufzeit $\mathcal{O}(n^2)$ im schlimmsten Fall, wobei Quicksort heraussticht, da es im mittleren Fall selbst die oben genannten Verfahren schlägt.

Ist mit $\mathcal{O}(n \log(n))$ das “Ende der Fahnenstange” erreicht ?

Sicher müssen sich Sortierverfahren alle vorliegenden Daten anschauen dürfen, was sofort eine **triviale untere Schranke von $\Omega(n)$** liefert.

Wären aber Sortierverfahren denkbar, die z.B. in $\mathcal{O}(n \log(\log(n)))$ laufen ?

Was heißt eigentlich “Sortieren” ?

Ausgangslage: Folge A von paarweise verschiedenen “Gegenständen” von einem (abstrakten) Datentyp D , d.h.: $A : D[1..n]$.

D soll uns Vergleichsoperatoren $<$, $\leq$ und $=$ zur Verfügung stellen, mit dessen Hilfe wir Vergleiche der Art $A[i] < A[j]$ in $\Theta(1)$ Zeit durchführen können.

Auf der dem Datentyp zugrundeliegenden Menge soll $\leq$ eine lineare Ordnung (manchmal auch totale Ordnung genannt) darstellen, d.h. insbesondere, dass stets $x \leq y$ oder $y \leq x$ für zwei beliebige Elemente gilt.

Ziel: Finde Permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$, sodass

$$A[\pi(1)] < A[\pi(2)] < \dots < A[\pi(n)].$$

Frage: Wie viele Vergleiche muss ein Sortiervorgehen S mindestens im schlimmsten Falle durchführen, wenn der Ablauf von S für festes n nur von den Ergebnissen der durchgeführten Vergleiche abhängen darf ?

Exkurs: nochmal Binärbäume

Wir hatten bereits gesehen:

Lemma: Die minimale Höhe eines Binärbaumes mit m Knoten ist $h = \lceil \log_2(m + 1) \rceil - 1$.

Leicht per Induktion einzusehen ist:

Lemma: Ein Binärbaum mit b Blättern hat mindestens $2b - 1$ Knoten insgesamt.

Daraus ergibt sich:

Folgerung: Die minimale Höhe eines Binärbaumes mit b Blättern ist $h = \lceil \log_2(b) \rceil$.

Beweis: b Blätter $\leadsto$ mindestens $2b - 1$ Knoten; um eine kleinstmögliche Höhe zu erzielen, wird diese Anzahl auch angenommen, wodurch für die kleinstmögliche Höhe h folgt:

$$h = \lceil \log_2(2b) \rceil - 1 = \lceil \log_2(b) \rceil.$$

Untere Schranke — ein Beweis

Vergleichsabhängiger Algorithmenverlauf und Notwendigkeit, $n!$ viele Anordnungen herstellen zu müssen, liefert **binären Entscheidungsbaum** mit $n!$ vielen Blättern für jeden Sortieralgorithmus.

Die Anzahl der schlimmstenfalls notwendigen Vergleiche entspricht der Länge eines Pfades von der Wurzel zu irgendeinem Blatt, also der Höhe h_n des Entscheidungsbaumes.

Mit dem folgenden (leicht zu zeigenden) Lemma folgt die behauptete untere Schranke mit obiger Folgerung sofort, denn:

$$h_n = \Omega(n!) = \Omega(n \log n)$$

Lemma: $n! \geq (n/2)^{n/2}$. Genaueres würde die **Stirlingsche Formel** liefern.

Einschub: Eigenschaften von Sortialgorithmen

1. In-Place

Ein Algorithmus arbeitet **in-place** bzw. **in situ**, wenn er außer dem für die Speicherung der zu bearbeitenden Daten benötigten Speicher nur eine konstante, also von der zu bearbeitenden Datenmenge unabhängige, Menge von Speicher (explizit) benötigt (wobei der Rekursionskeller unberücksichtigt bleibt).

Der Algorithmus überschreibt die Eingabedaten mit den Ausgabedaten.

Hierbei ist auch von **Ortsfestigkeit** die Rede.

Aufgabe: Welche der in der Vorlesung und Übung behandelten Algorithmen sind in-place ?

Eigenschaften von Sortieralgorithmen

2. Stabilität

Ein **stabiles Sortierverfahren** ist ein Sortieralgorithmus, der die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt.

Wenn beispielsweise eine Liste alphabetisch sortierter Personendateien nach dem Geburtsdatum neu sortiert wird, dann bleiben unter einem stabilen Sortierverfahren alle Personen mit gleichem Geburtsdatum alphabetisch sortiert.

Aufgabe: Welche der in der Vorlesung und Übung behandelten Algorithmen sind stabil bzw. können so modifiziert werden, dass sie stabil werden ?

Sortieren in Linearzeit ?

Das sollte doch “eigentlich” nicht gehen. . .

Unsere Überlegungen waren aber mit verschiedenen Einschränkungen versehen, z.B.: es dürfen keine zwei gleichen “Schlüssel” auftreten, und das Sortieren muss durch ausschließliche Benutzung der Vergleichsoperatoren durchgeführt werden.

Spezielle Sortiervverfahren

1. COUNTING SORT (Sortieren durch Zählen)
2. BUCKET SORT (Sortieren durch Fachverteilung)

Counting Sort oder Sortieren durch Zählen

Eingabe: Feld $A[1..n]$ von Zahlen aus $\{0 \dots k - 1\}$

Ausgabe: Feld $B[1..n]$ aufsteigend sortiert

Hilfsfeld: $COUNT[0..k-1]$

Counting Sort besteht aus 4 Schritten (for-Schleifen)

1. Initialisierung

for $j = 0$ **to** $k - 1$ **do** $COUNT[j] \leftarrow 0$; **od**

2. Counting

for $i = 1$ **to** n **do** $COUNT[A[i]]++$; **od**

Nach Schritt 2 gilt

$COUNT[x] = \text{Anzahl aller Elemente } y \in A \text{ mit } y = x$

3. Aufaddieren

FEHLER im ursprünglichen Skript: **for** $j = 1$ **to** $k - 1$ **do** $A[j] \leftarrow A[j] + A[j - 1]$; **od**
for $j = 1$ **to** $k - 1$ **do** $\text{COUNT}[j] \leftarrow \text{COUNT}[j] + \text{COUNT}[j - 1]$; **od**

Nach Schritt 3 gilt

$\text{COUNT}[x]$ = Anzahl aller Elemente $y \in A$ mit $y \leq x$ d.h.

$\text{COUNT}[x]$ = max. Position an der x im Ausgabefeld B stehen kann.

4. Ausgabe (Verteilen)

for $i = n$ **downto** 1 **do**
 $x \leftarrow A[i]$;
 $p \leftarrow \text{COUNT}[x]$;
 $A[p] = x$;
 $\text{COUNT}[x] \leftarrow p - 1$;
od

Eigenschaften von Counting Sort

CountingSort ist **stabil**,
wenn die letzte for-Schleife rückwärts über das Eingabefeld A läuft.

Frage: Kann man auch vorwärts laufen - was muss man ändern ?

Laufzeitanalyse

4 for-Schleifen mit jeweils konstantem Rumpf:

2 Schleifen über das COUNT-Feld: $\mathcal{O}(k)$

2 Schleifen über das Eingabe-Feld A : $\mathcal{O}(n)$

Damit ist die Gesamtlaufzeit $\mathcal{O}(n + k) = \mathcal{O}(n)$ für $k = \mathcal{O}(n)$.

Ein Beispiel

Bucket Sort oder Sortieren durch Fachverteilung

Ein Feld $A[1..n]$ von Zahlen aus $\{0 \dots k - 1\}$ soll aufsteigend sortiert werden.

Allgemeiner:

A ist Feld von Daten mit jeweils einem Schlüssel aus $[0 \dots k - 1]$

Beispiel: Daten von Studierenden mit Schlüssel = Matrikelnummer

Hilfsfeld

$B[0..k - 1]$ von k **Buckets**, genauer

$B[j]$ = Kopf einer einfach verketteten Liste ($0 \leq j \leq k - 1$).

Bucket Sort ist **stabil** und besteht aus 3 Schritten (for-Schleifen)

1. **Initialisierung** (stelle k leere Buckets bereit)

for $j = 0$ **to** $k - 1$ **do** $B[j] \leftarrow \text{null}$; **od**

2. **Verteilen**

for $i = 1$ **to** n **do**

$x \leftarrow A[i]$;

$B[x].\text{append}(x)$; //bzw. $B[x.\text{key}].\text{append}(x)$;

od

3. **Aufsammeln**

$i \leftarrow 1$;

for $j = 0$ **to** $k - 1$ **do**

forall $x \in B[j]$ **do** $A[i++] \leftarrow x$; **od**

od

Laufzeit: $\mathcal{O}(n + k)$

Anwendung stabiler Sortialgorithmen

Sortieren nach mehreren Kriterien

Beispiel:

Sortiere die Studierenden der Uni Trier nach ihrem Alter in Jahren (1. Kriterium), Studierende mit gleichem Alter sollen nach ihrer Matrikelnummer (2. Kriterium) sortiert werden.

Lösung:

Sortiere zunächst nach Matrikelnummer und dann nach Alter, d.h. zunächst nach dem weniger wichtigen dann nach dem wichtigsten Kriterium. Für den zweiten Schritt sollte ein stabiles Sortiervorgehen verwendet werden, damit bei gleichem Alter die Ordnung gemäß der Matrikelnummer erhalten bleibt.

Allgemeiner: Sortieren von Zeichenketten (Strings)

Radix Sort: Sortieren von Strings

Es handelt sich um eine iterierte Variante von Bucketsort.

z.B. kann man bei natürlichen Zahlen als “Fächer” die möglichen Werte der Buchstaben ansehen. Sortiert man nun zunächst mit Bucketsort gemäß dem letzten (also unwichtigsten) Buchstaben, dann nach dem zweitletzten usf. bis zum ersten Buchstaben, erhält man am Ende eine vollständig alphabetisch geordnete Liste.

Formaler unterscheidet man zwischen sich abwechselnden **Partitionsphasen** und **Sammelphasen**.

Beispiel: Sortieren der Strings (Zahlen) 124, 523, 483, 128, 923, 584.

0	1	2	3	4	5	6	7	8	9
			523	124				128	
			483	584					
			923						

--> 523, 483, 923, 124, 584, 128

0	1	2	3	4	5	6	7	8	9
		523						483	
		923						584	
		124							
		128							

--> 523, 923, 124, 128, 483, 584

0	1	2	3	4	5	6	7	8	9
	124			483	523				923
	128				584				

--> 124, 128, 483, 523, 584, 923 OK!