Praktikum Algorithmische Anwendungen WS 2006/07
Sortieren in linearer Laufzeit

Team A blau
Martin Herfurth 11043831
Markus Wagner 11043447

5. Februar 2007

1 Untere Schranke für Vergleichsbasierte Algorithmen

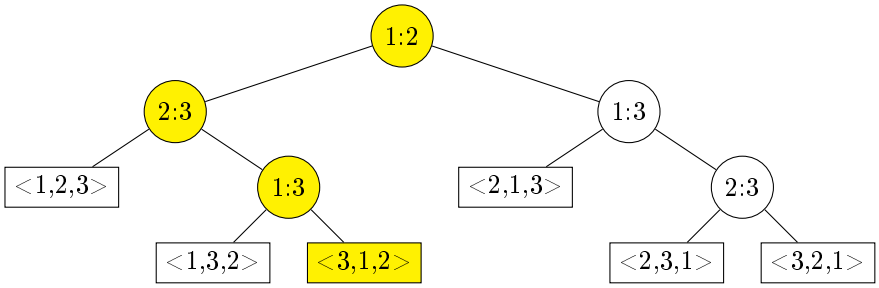
Die betrachteten Sortieralgorithmen CountingSort, RadixSort und BucketSort sind in der Lage, eine Menge in linearer Zeit zu sortieren. Dafür benötigen sie allerdings einige Voraussetzungen, die hier nicht weiter erläutert werden.

Ein vergleichsbasierter Sortieralgorithmus schafft diese Aufgabe im worst-case (MergeSort, HeapSort) bzw. im average-case (quicksort) bestenfalls in $O(n \cdot \log n)$. Dies gilt es zu beweisen.

1.1 Entscheidungsbaum

Sei T Entscheidungsbaum für ein einen vergleichsbasierten Sortieralgorithmus A mit Eingabegröße n. In T werden in jedem Knoten die Vergleiche gespeichert, im linken Sohn liegt dann der nächste Vergleich für den Fall, dass der linke Wert kleiner oder gleich dem rechten ist und im rechten Sohn für den umgekehrten Fall. Die Blätter tragen die Permutationen, die nötig sind, um die Eingabemenge zu sortieren. T ist ein voller Binärbaum.

Beispiel für einen Entscheidungsbaum für InsertSort mit $n = 3$ und der Eingabemenge $(6, 8, 5)$:



1.2 Anzahl Vergleiche

Jede Ausführung von A auf einer Eingabemenge gehört zu einem Pfad auf dem Entscheidungsbaum T. Das heisst, jede Sortierung gehört zu einem Blatt des Entscheidungsbaumes. Da die Blätter die Permutationen der Eingabemenge repräsentieren, gibt es mindestens $n!$ Blätter.

Eine untere Schranke für die Tiefe von T is gleichzeitig eine unterer Schranke für die Laufzeit jedes vergleichsbasierten Sortieralgorithmus im worst-case.

Es genügt nun zu zeigen, dass für die Tiefe d des Baumes T mit $l \geq n!$ Blättern gilt: $d = \Omega(n \cdot \log n)$.

Wir wissen: $n! \leq l \leq 2^d$

Mit:

$$\frac{n^n}{n!} < \sum_{i=0}^{\infty} \frac{n^i}{i!} = e^n \Rightarrow n! > \left(\frac{n}{e}\right)^n$$

$$\Rightarrow \log n! > n \cdot \log n - n \cdot \log e \geq \frac{1}{2} \cdot n \log n$$

$$\text{für } \log n \geq 2 \cdot \log e$$

gilt:

$$d \geq \log n! = \Omega(n \cdot \log n) \qquad q.e.d.$$

2 Radixsort

Radixsort ist ein einfaches und intuitives Sortierverfahren. Wie bei allen hier betrachteten Sortierverfahren setzte es eine gewisse Kenntnis über die Menge der möglichen Schlüssel voraus. Um mit Radixsort in linearer Zeit sortieren zu können muss man das Universum der möglichen Schlüssel kennen, es muss endlich sein und sich in Bereiche einteilen lassen. Sortiert werden hier natürliche Zahlen, die von denen die Basis verwendet wird. Daher auch der Name Radix (Wurzel, Basis).
Das Verfahren besteht aus mehreren Partitionierungs- und Sammelphasen. In der Partitionierungsphasen wird jeweils nach einer Stelle der Zahlen sortiert und danach wird in der zugehörigen Sammelphase ein neues Feld erzeugt, in dem einfach die partitionierten Werte aufgereiht werden. Dies wird für jede Stelle wiederholt. Nach der letzten Sammelphase ist das komplette feld sortiert.
Der Algorithmus arbeitet ortslos, das bedeutet, dass für das sortierte Ausgabefeld ein extra Feld benötigt wird. Eine weitere Eigenschaft ist die Stabilität des Verfahrens. Eine vorher vorhandene Sortierung wird bei gleichen Werten in der Partitionierungsphase erhalten. Dies ist auch nötig, da sonst bei jedem Schritt der vorangegangene Schritt wieder unwirksam würde.

2.1 Pseudocode

```
radixsort(field A)
1 B = new Array[10]
2 for 0 <= i < d do
3     for 0 <= j < A.length do
4         B[value(i, A[j])]=A[j]
5     end
6     A=B;
7 end
8 return A;
```

Die Funktion `value(i, x)` gibt die Ziffer der Zahl `x` an der Stelle `i` zurück. `d` ist konstant und bekannt (Anszahl der Ziffern, der größten Möglichen Zahl in `A`).

2.2 Beispiel

Gegeben is das folgende Feld mit 10 Elementen aus der Menge [0;999]:

129	944	318	322	450	111	397	745	58	922

Im ersten Schritt werde die nach der letzten Stelle in das Hilfsfeld eingefügt (Partitioniert) und dann von vorne nach hinten wieder aufgesammelt:

0	1	2	3	4	5	6	7	8	9
450	111	322 922		944	745		397	318 58	129

450	111	322	922	944	745	397	318	58	129

Im zweiten Schritt dann nach der vorletzten Stelle:

0	1	2	3	4	5	6	7	8	9
	111 318	322 922 129		944 745	450 58				397

111	318	322	922	129	944	745	450	58	397

Im dritten und letzten Schritt werden die Elemente dann nach der ersten Stelle sortiert:

0	1	2	3	4	5	6	7	8	9
	111 129		318 322 397	450	58		745		922 944

58	111	129	318	322	397	450	745	922	944

2.3 Performance

Der Algorithmus durchläuft für jede mögliche Ziffer (Zeilen 2-7) einmal das gesamte Eingabefeld (Zeilen 3-5). Das Einfügen jedes Wertes in die Partitionierungstabelle geschieht in konstanter Zeit, da hier direkt in die Stelle der Tabelle geschrieben wird.
Das heisst, die Laufzeit bei d Ziffern und einer Eingabegröße von n beträgt: $O(d \cdot n)$. Da d gegenüber n als konstant angesehen werden kann, gilt: $O(n)$.

3 Bucketsort

Bucketsort ist eine Variante von Radixsort. Hier werden leere Eimer (Buckets) initialisiert, in die dann die zu sortierende Folge einsortiert wird. Hierbei gibt es zwei Varianten, zum einen kann man für jeden möglichen Schlüssel im Universum einen Bucket erzeugen, dann hat man direkt nach einem Durchlaufen des Eingabefeldes, die Elemente sortiert. Alternativ kann man auch für verschiedene Bereiche des Universums Buckets erzeugen. Dann muss aber immer noch innerhalb eines Buckets sortiert werden. Dafür kann man dann rekursiv den Algorithmus erneut aufrufen, oder aber einen anderen Algorithmus verwenden, der innerhalb der Buckets sortiert. Im folgenden Betrachten wir die einfache Variante, also dass für jeden Schlüssel ein eigener Bucket existiert.
Somit wird das Eingabefeld einfach durchlaufen und die Werte werden an den enstprechenden Stellen im Hilfsfeld gespeichert. Falls die Schlüssel nicht natürliche Zahlen sind, so benötigt man noch eine bijektive Abbildung der Schlüssel auf die natürlichen Zahlen.

3.1 Pseudocode

```
bucketsort(field A)
1  B = new Array[m] //m ist die Größe des Universums
2  for 0 <= i < A.length do
3      B[A[i]]=A[i]
4  end
5  k = 0
6  for 0 <= j < B.length do
7      if B[j] != 0 do
8          A[k] = B[j]
9          k = k+1
10     end
11 end
```

3.2 Beispiel

Folgendes Eingabefeld soll sortiert werden:

10	11	109	205	77	344	129	76	4	129

Das Universum sind die natürlichen Zahlen im Bereich [0;999], also gibt es ein Hilfsarray mit 1000 Stellen. An die entsprechenden Positionen werden die Werte kopiert:

0	1	2	3	4	...	10	11	...	76	77	...	109	...	129	...	205	...	344	...	999
				4		10	11		76	77		109		129 129		205		344		

Danach werden die Werte einfach wieder aufgereiht:

4	10	11	76	77	109	129	129	205	344

3.3 Performance

Es gibt 2 Schleifen. Zunächst wird durch das Eingabefeld iteriert (Zeilen `2-4`) und die Werte werden in das Hilfsfeld gespeichert. Danach wird das Hilfsfeld in den Zeilen `6-10` ausgelesen und die gefundenen Werte werden in das Ausgabefeld kopiert. Die erste Schleife hat n Durchläufe, die zweite m. Damit ergibt sich eine Laufzeit von $O(n+m)$.

4 Countingsort

Countingsort verfolgt den Ansatz für jeden Wert i des Eingabefeldes A die Anzahl der Werte $j \in A$ zu finden, für die gilt: $j \leq i$. Dafür wird zunächst ein Hilfsfeld mit der Länge k des Universums erzeugt. In diesem Feld werden dann für alle Werte aus dem Eingabefeld die Positionen um 1 inkrementiert. Damit hat man für jeden Schlüssel die Häufigkeit im Eingabefeld. Nun addiert man von links beginnend die Werte auf: $i_n = \sum_{m=0}^{n-1} i_m$.
Nun wird das Eingabefeld von rechts durchlaufen und im Hilfsfeld überprüft, wiviele Werte kleiner oder gleich dem aktuellen Schlüssel sind. An diese Stelle wird der Wert dann im Ausgabefeld gespeichert und im Hilfsfeld der zugehörige Wert um 1 dekrementiert.

4.1 Pseudocode

```
countingsort(A)
1  for 0 < i <= k do
2      C[i] = 0
3  end
4  for 0 < j <= length(A) do
5      C[A[j]] = C[A[j]] + 1
6  end
7  for 2 <= i <= k do
8      C[i] = C[i] + C[i-1]
9  end
10 for length(A) >= j >= 1 do
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] - 1
13 end
14 return B
```

4.2 Beispiel

Es werden neben dem Eingabefeld A noch ein Hilfsfeld C (mit 0 initialisiert) und das Ausgabefeld B benötigt:

A:

3	6	3	4	1	2	7	10	11	12	13	11	11	4	3	9	18	3	7	1

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

B:

Dann werden die Felder in C nach und nach inkrementiert:

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

...

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	1	4	2	0	1	2	0	1	1	3	1	1	0	0	0	0	1

Danach werden die Werte aufaddiert:

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	3	4	2	0	1	2	0	1	1	3	1	1	0	0	0	0	1

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	3	7	2	0	1	2	0	1	1	3	1	1	0	0	0	0	1

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	3	7	9	0	1	2	0	1	1	3	1	1	0	0	0	0	1

...

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	3	7	9	9	10	12	12	13	14	17	18	19	19	19	19	19	20

Nun wird das Eingabefeld A von rechts ausgelesen und die Werte werden Anhand der in C vorhandenen Zahlen in die entsprechende Stelle in das Ausgabefeld geschrieben. Die Werte in C werden dann immer dekrementiert:

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	3	7	9	9	10	12	12	13	14	17	18	19	19	19	19	19	20

B:

	1																

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	3	7	9	9	10	11	12	13	14	17	18	19	19	19	19	19	20

B:

	1									7							

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	3	6	9	9	10	11	12	13	14	17	18	19	19	19	19	19	20

B:

	1						3				7						

...

C:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	2	3	7	9	9	10	12	12	13	14	17	18	19	19	19	19	19

B:

1	1	2	3	3	3	3	4	4	6	6	7	9	10	11	11	11	12	13	18

Fertig.

4.3 Performance

Die erste (Zeilen `1-3`) und dritte (Zeilen `7-9`) Schleife laufen jeweils k-mal ab. Die zweite (Zeilen `4-6`) und vierte (Zeilen `10-13`) Schleife jeweils n-mal. Somit ergibt sich eine Komplexität von $O(k+n)$.

5 Fazit

Die vorgestellten Algorithmen haben zwar lineare Laufzeit. Allerdings hängt ihre effektive Laufzeit immer von der Größe des Universums ab. Daher lohnen sich Ausführungen auf Mengen mit potenziell vielen veschiedenen Schlüsseln hier nicht.
Aber gerade, wenn die Anzahl der Schlüssel höher ist, als die Größe des Universums erreicht man sehr gute Ergebnisse. Hier gibt es dann viele Schlüssel, die doppelt vorkommen. Da eben auch alle vorgestellten Algorithmen stabil sind, geht eine Vorsortierung unter diesen gleichen Schlüsseln nicht verloren.

6 Quellen

1. http://www.wikipedia.de (Stand: 04.02.2007)
2. http://www.sortieralgorithmen.de (Stand: 04.02.2007)
3. Karl Heinz Niggl; 8. Vorlesung: Algorithmen und Datenstrukturen; TU Ilmenau
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein; Introduction to Algorithms