RUB

Vorlesung Datenstrukturen

Sortieren fortgesetzt

Maike Buchin

16.5.2017

5.6 Brechen der Unteren Schranke

RUB

- nur Vergleichen erlauben keine Algorithmen mit Laufzeit schneller als $O(n \log n)$
- stattdessen: Struktur der Eingabe ausnutzen.

5.6 Brechen der Unteren Schranke

RUB

KSort [simple bucket sort]

Annahme: Schlüssel sind kleine natürliche Zahlen in $0 \ldots k-1$.

Verwende Array $b[0 \ldots k-1]$ von anfangs leeren Behältern.
Durchlaufe die Eingabe und tue Schlüssel k in Behälter $b[k]$.
Füge anschliessend alle Behälter der Reihe nach zusammen.

kSort(S)

$b = \langle\langle\rangle, \ldots, \langle\rangle\rangle$: array of k lists
for each $e \in S$ do $b[key(e)].pushBack(e)$
return $concatenation\ of\ b[0], \ldots, b[k-1]$

Beispiel: $s = \langle(3,a),(1,b),(2,c),(3,d),\quad k = 4$

$b = \langle\langle\rangle, \langle(1,b)\rangle, \langle(2,c)\rangle, \langle(3,a),(3,d)\rangle\rangle$

Ausgabe $= \langle(1,b),(2,c),(3,a),(3,d)\rangle$

5.6 Brechen der Unteren Schranke

RUB

KSort [simple bucket sort]

Annahme: Schlüssel sind kleine natürliche Zahlen in $0 \ldots k-1$.

Verwende Array $b[0 \ldots k-1]$ von anfangs leeren Behältern.
Durchlaufe die Eingabe und tue Schlüssel k in Behälter $b[k]$.
Füge anschliessend alle Behälter der Reihe nach zusammen.

kSort(S)
$b = \langle\langle\rangle, \ldots, \langle\rangle\rangle$: array of k lists
for each $e \in S$ do $b[key(e)].pushBack(e)$
return $concatenation\ of\ b[0], \ldots, b[k-1]$

stabil: Einträge mit gleichem Schlüssel behalten Reihenfolge

Laufzeit: $O(n+k)$

Korrektheit: klar

Radixsort

verwende KSort als Baustein um größere Schlüssel zu sortieren.

Annahme: Schlüssel haben (bis zu) d Ziffern in $0 \ldots K-1$.
Ansatz: wende KSort auf alle d Ziffern an.

LSD-Radixsort: sortiere Ziffern vom kleinsten zum größten

Procedure *LSDRadixSort*$(s : Sequence$ **of** $Element)$
  for $i := 0$ **to** $d-1$ **do**
    Definiere $key(x)$ als $(x$ **div** $K^i)$ **mod** K
    $KSort(s)$
    invariant s ist bezüglich der Ziffern $i..0$ sortiert

Beispiel:

017	111	007	007
042	911	111	017
666	042	911	042
007 →	666 →	017 →	111
111	017	042	666
911	007	666	911
999	999	999	999

Radixsort

RUB

verwende KSort als Baustein um größere Schlüssel zu sortieren.

Annahme: Schlüssel haben (bis zu) d Ziffern in $0 \ldots K-1$.
Ansatz: wende KSort auf alle d Ziffern an.

LSD-Radixsort: sortiere Ziffern vom kleinsten zum größten

Procedure *LSDRadixSort*$(s : Sequence$ **of** $Element)$
  for $i := 0$ **to** $d-1$ **do**
    Definiere $key(x)$ als $(x$ **div** $K^i)$ **mod** K
    $KSort(s)$
    invariant s ist bezüglich der Ziffern $i..0$ sortiert

Korrektheit: folgt aus der Stabilität von KSort:
Sei $a < b$ mit $a, b \in S$ Sei j der größte Index auf dem sie sich unterscheiden. Dann wird für $i = j$ a vor b platziert, und ihre Reihenfolge danach (wegen Stabilität) nicht mehr verändert.

Laufzeit: $O(d(n+k))$

Radixsort

RUB

verwende KSort als Baustein um größere Schlüssel zu sortieren.

Annahme: Schlüssel haben (bis zu) d Ziffern in $0 \ldots K-1$.
Ansatz: wende KSort auf alle d Ziffern an.

MSD-Radixsort: sortiere Eingabe nach größter Ziffer in Behälter und dann die Behälter mit langsamen Sortieralgorithmus.
Es kann natürlich passieren, dass ein Behälter sehr voll, und der Algorithmus dann sehr langsam ist.

Annahme: uniform verteilte Schlüssel aus $[0, 1)$

Procedure *uniformSort*$(s : Sequence$ **of** $Element)$
$n := |s|$
$b = \langle\langle\rangle, \ldots, \langle\rangle\rangle : Array\ [0..n-1]$ **of** $Sequence$ **of** $Element$
foreach $e \in s$ **do** $b[\lfloor key(e) \cdot n \rfloor].pushBack(e)$
for $i := 0$ **to** $n-1$ **do** sortiere $b[i]$ in Zeit $O(|b[i]| \log |b[i]|)$
$s :=$ *Konkatenation von* $b[0], \ldots, b[n-1]$

Radixsort

RUB

verwende KSort als Baustein um größere Schlüssel zu sortieren.

Annahme: Schlüssel haben (bis zu) d Ziffern in $0 \dots K-1$.
Ansatz: wende KSort auf alle d Ziffern an.

MSD-Radixsort: sortiere Eingabe nach größter Ziffer in Behälter und dann die Behälter mit langsamen Sortieralgorithmus.
Es kann natürlich passieren, dass ein Behälter sehr voll, und der Algorithmus dann sehr langsam ist.

Annahme: uniform verteilte Schlüssel aus $[0, 1)$

Beispiel: $s = \langle 0.8, 0.4, 0.7, 0.6, 0.3 \rangle$

$$b = \langle \langle\rangle, \langle (0.3) \rangle, \langle (0.4) \rangle, \langle (0.7), (0.6), \langle (0.8) \rangle \rangle$$

$$\text{Ausgabe} = \langle 0.3, 0.4, 0.6, 0.7, 0.8 \rangle$$

Radixsort

Satz 5.9 Wenn n Schlüssel unabhängig und uniform aus $[0,1)$ gewählt werden, dann werden sie von uniformSort in Zeit $O(n)$ im mittleren und Zeit $O(n \log n)$ im schlechtesten Fall sortiert.

Beweis:

$E[T] = \Theta(n) + E[\sum_i T_i] = \Theta(n) + \sum_i E[T_i] = \Theta(n) + nE[T_0]$

wobei T_i die Zeit zu Sortieren des i-ten Behälter, $0 \leq i < n$.

B.z.z. $E[T_0] = O(1)$.
Wir zeigen dies sogar für einen quadratischen Sortieralgorithmus.

Sei $B_0 = |b[0]|$. Dann gilt $E[T_0] = O(E[B_0^2])$.

$P[B_0 = k] = \binom{n}{k}(\frac{1}{n})^k(1-\frac{1}{n})^{n-k} \leq \frac{n^k}{k!}\frac{1}{n^k} = \frac{1}{k!} \leq (\frac{e}{k})^k$

$\Rightarrow E[B_0^2] = \sum_{k \leq n} k^2 P[B_0 = k] \leq \sum_{k \leq n} k^2 (\frac{e}{k})^k$

$\leq \sum_{k \leq 5} k^2 (\frac{e}{k})^k + e^2 \sum_{k \geq 6} (\frac{e}{k})^{k-2} = O(1)$

CountingSort [CLRS 8.2, Alternative zu KSort]

RUB

Annahme: Schlüssel sind kleine natürliche Zahlen in $0 \ldots k$.

Gegeben Array $A[1 \ldots n]$ von Schlüsseln, um Position von $A[i]$ zu bestimmen zähle wieviele Elemente in A kleiner sind als $A[i]$

Achtung: Einträge können gleiche Werte haben!
Lösung: Zähle $<$ in $A[1 \ldots n]$ und $\leq$ in $A[1 \ldots i]$

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

CountingSort [CLRS 8.2, Alternative zu KSort]

RUB

Annahme: Schlüssel sind kleine natürliche Zahlen in $0 \ldots k$.

Gegeben Array $A[1 \ldots n]$ von Schlüsseln, um Position von $A[i]$ zu bestimmen zähle wieviele Elemente in A kleiner sind als $A[i]$

Achtung: Einträge können gleiche Werte haben!
Lösung: Zähle $<$ in $A[1 \ldots n]$ und $\leq$ in $A[1 \ldots i]$

Laufzeit: $\Theta(n)$

Korrektheit: für den letzten Loop verwende Loopinvariante:
Inv(m): for $m \leq j \leq n$: $B[$position of $A[j]]$ enthält $A[j]$
und for $0 \leq i \leq k$: $C[i] =$ Anzahl Elemente $< A[i]$ in $A[1 \ldots n]$
$+$ Anzahl Elemente $= A[i]$ in $A[1 \ldots m-1]$