Script generated by TTT

Title: Täubig: GAD (14.06.2012)

Date: Thu Jun 14 12:54:12 CEST 2012

Duration: 22:00 min

Pages: 12

QuickSort

Verbesserte Version ohne Check für Array-Grenzen

```
qSort(Element[] a, int l, int r) {
  while (r − l ≥ n0) {
    j = pickPivotPos(a, l, r);
    swap(a[l],a[j]);   p = a[l];
    int i = l;   int j = r;
    repeat {
      while (a[i] < p) do i++;
      while (a[j] > p) do j−−;
      if (i ≤ j) { swap(a[i], a[j]);   i++;   j−−; }
    } until (i > j);
    if (i < (l + r)/2) { qSort(a, l, j);   l = i; }
    else               { qSort(a, i, r);   r = j; }
  }
  insertionSort(a, l, r);
}
```


QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als 2/3 der aktuellen Feldgröße sind:

schlecht	gut	schlecht

$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

Erwartete Zeit bei n Elementen

- linearer Aufwand außerhalb der rekursiven Aufrufe: cn
- Pivot gut (Wsk. 1/3): Restaufwand $\le T(2n/3)$
- Pivot schlecht (Wsk. 2/3): Restaufwand $\le T(n-1) < T(n)$

QuickSelect

Beweis.

$$
\begin{aligned}
T(n) &\le cn + p\cdot T(n\cdot 2/3) + (1-p)\cdot T(n)\\
p\cdot T(n) &\le cn + p\cdot T(n\cdot 2/3)\\
T(n) &\le cn/p + T(n\cdot 2/3)\\
&\le cn/p + c\cdot(n\cdot 2/3)/p + T(n\cdot(2/3)^2)\\
&\ldots \text{ wiederholtes Einsetzen}\\
&\le (cn/p)(1 + 2/3 + 4/9 + 8/27 + \ldots)\\
&\le \frac{cn}{p}\cdot\sum_{i\ge 0}(2/3)^i\\
&\le \frac{cn}{1/3}\cdot\frac{1}{1-2/3} = 9cn \in O(n)
\end{aligned}
$$

Übersicht

6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

Sortieren schneller als $O(n \log n)$

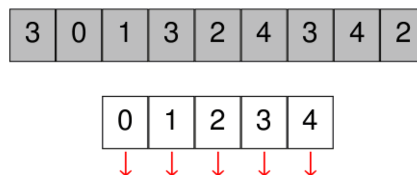
Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
- z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer bzw. das erste Zeichen. Nur bei Gleichheit des Anfangs kommt es auf weitere Ziffern / Zeichen an.
- Annahme: Elemente sind Zahlen im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)

Sortieren schneller als $O(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
- z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer bzw. das erste Zeichen. Nur bei Gleichheit des Anfangs kommt es auf weitere Ziffern / Zeichen an.
- Annahme: Elemente sind Zahlen im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)



Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n + K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n + K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

H. Täubig (TUM) GAD SS'12 264 / 637

Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n + K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n + K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

- verwende K-adische Darstellung der Schlüssel
- Annahme:
  Schlüssel sind Zahlen aus $\{0, \ldots, K^d - 1\}$
  repräsentiert durch d Ziffern aus $\{0, \ldots, K - 1\}$
- sortiere zunächst entsprechend der niedrigstwertigen Ziffer mit kSort und dann nacheinander für immer höherwertigere Stellen
- behalte Ordnung der Teillisten bei

RadixSort

```
radixSort(Sequence<Elem> s) {
  for (int i = 0; i < d; i++)
    kSort(s,i);   // sortiere gemäß key_i(x)
    // mit key_i(x) = (key(x)/K^i) mod K
}
```

Verfahren funktioniert, weil kSort stabil ist:
Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i - 1 \ldots 0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n + K))$ für n Schlüssel aus $\{0, \ldots, K^d - 1\}$

RadixSort

```
radixSort(Sequence<Elem> s) {
  for (int i = 0; i < d; i++)
    kSort(s,i);  // sortiere gemäß key_i(x)
    // mit key_i(x) = (key(x)/K^i) mod K
}
```

Verfahren funktioniert, weil kSort stabil ist:
Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1\dots 0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n+K))$ für n Schlüssel aus $\{0,\dots,K^d-1\}$