Script **generated by TTT**

Title: Täubig: GAD (12.06.2012)

Date: Tue Jun 12 15:16:41 CEST 2012

Duration: 45:33 min

Pages: 13

QuickSort

Verbesserte Version ohne Check für Array-Grenzen

qSort(Element[ ] a, int l, int r) {
  while ($r - l \geq n_0$) {
    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 ($i \leq j$) { 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);
}

Übersicht

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

Sortieren QuickSort

QuickSort

Verbesserte Version ohne Check für Array-Grenzen

qSort(Element[ ] a, int l, int r) {
  while ($r - l \geq n_0$) {
    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 ($i \leq j$) { 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);
}

H. Täubig (TUM) GAD SS'12 252 / 637

Übersicht

6 Sortieren

Rang-Selektion

- Bestimmung des kleinsten und größten Elements ist mit einem einzigen Scan über das Array in Linearzeit möglich
- Aber wie ist das beim k-kleinsten Element, z.B. beim $\lfloor n/2 \rfloor$-kleinsten Element (Median)?

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

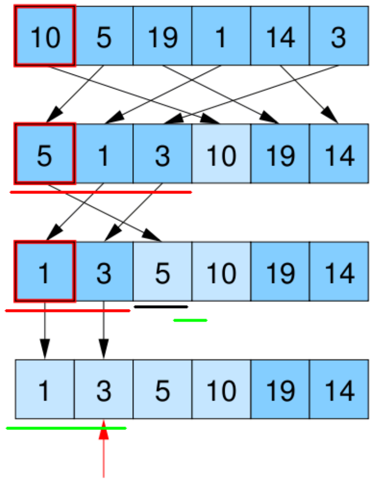
Naive Lösung: Sortieren und k-tes Element ausgeben

$\Rightarrow$ Zeit $O(n \log n)$

Geht das auch schneller?

QuickSelect

Ansatz: ähnlich zu QuickSort, aber nur eine Seite betrachten



QuickSelect

Methode analog zu QuickSort

Element quickSelect(Element[ ] a, int l, int r, int k) {
  // $a[l \ldots r]$: Restfeld, k: Rang des gesuchten Elements
  if ($r == l$) return $a[l]$;
  int z = zufällige Position in $\{l, \ldots, r\}$; swap($a[z]$, $a[r]$);
  Element $v = a[r]$; int $i = l - 1$; int $j = r$;
  do { // spalte Elemente in $a[l, \ldots, r-1]$ nach Pivot v
    do $i{+}{+}$ while ($a[i] < v$);
    do $j{-}{-}$ while ($a[j] > v$ && $j \neq l$);
    if ($i < j$) swap($a[i]$, $a[j]$);
  } while ($i < j$);
  swap ($a[i]$, $a[r]$); // Pivot an richtige Stelle
  if ($k < i$) return quickSelect(a, l, $i - 1$, k);
  if ($k > i$) return quickSelect(a, $i + 1$, r, k);
  else return $a[k]$; // $k == i$
}

QuickSelect
Alternative Methode

```
Element select(Element[] s, int k) {
    assert(|s| ≥ k);
    Wähle p ∈ s zufällig (gleichverteilt);

    Element[] a := {e ∈ s : e < p};
    if (|a| ≥ k)
        return select(a,k);

    Element[] b := {e ∈ s : e = p};
    if (|a| + |b| ≥ k)
        return p;

    Element[] c := {e ∈ s : e > p};
    return select(c,k − |a| − |b|);
}
```


QuickSelect
Alternative Methode

Beispiel

s	k	a b c
$\langle 3,1,4,1,5,9,2,6,5,3,5,8,9\rangle$	7	$\langle 1,1\rangle\ \langle 2\rangle\ \langle 3,4,5,9,6,5,3,5,8,9\rangle$
$\langle 3,4,5,9,6,5,3,5,8,9\rangle$	4	$\langle 3,4,5,5,3,5\rangle\ \langle 6\rangle\ \langle 9,8,9\rangle$
$\langle 3,4,5,5,3,5\rangle$	4	$\langle 3,4,3\rangle\ \langle 5,5,5\rangle\ \langle\rangle$

In der sortierten Sequenz würde also an 7. Stelle das Element 5 stehen.

Hier wurde das mittlere Element als Pivot verwendet.

QuickSelect

teilt das Feld jeweils in 3 Teile:

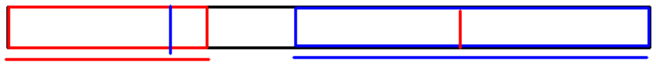
- a Elemente kleiner als das Pivot
- b Elemente gleich dem Pivot
- c Elemente größer als das Pivot

$T(n)$: erwartete Laufzeit bei n Elementen

Satz

Die erwartete Laufzeit von QuickSelect ist linear: $T(n) \in O(n)$.

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 $\leq T(2n/3)$
- Pivot schlecht (Wsk. 2/3): Restaufwand $\leq T(n-1) < T(n)$

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.

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

□