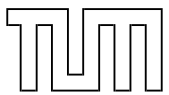
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010



Übersicht

MergeSort

Sortieren durch Verschmelzen

Zeitaufwand:

- $T(n)$: Laufzeit bei Feldgröße n
- $T(1) = \Theta(1)$
  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

$\Rightarrow$ $T(n) \in \mathcal{O}(n \log n)$
(folgt aus dem sogenannten Master-Theorem)

Untere Schranke

MergeSort hat Laufzeit $\mathcal{O}(n \log n)$ im worst case.

insertionSort kann so implementiert werden, dass es im best case nur lineare Laufzeit hat

Gibt es Sortierverfahren mit Laufzeit besser als $\mathcal{O}(n \log n)$ im worst case (z.B. $\mathcal{O}(n)$ oder $\mathcal{O}(n \log \log n)$?

$\Rightarrow$ nicht auf der Basis einfacher Schlüsselvergleiche

Entscheidungen: $x_i < x_j \quad \rightarrow \quad$ ja / nein

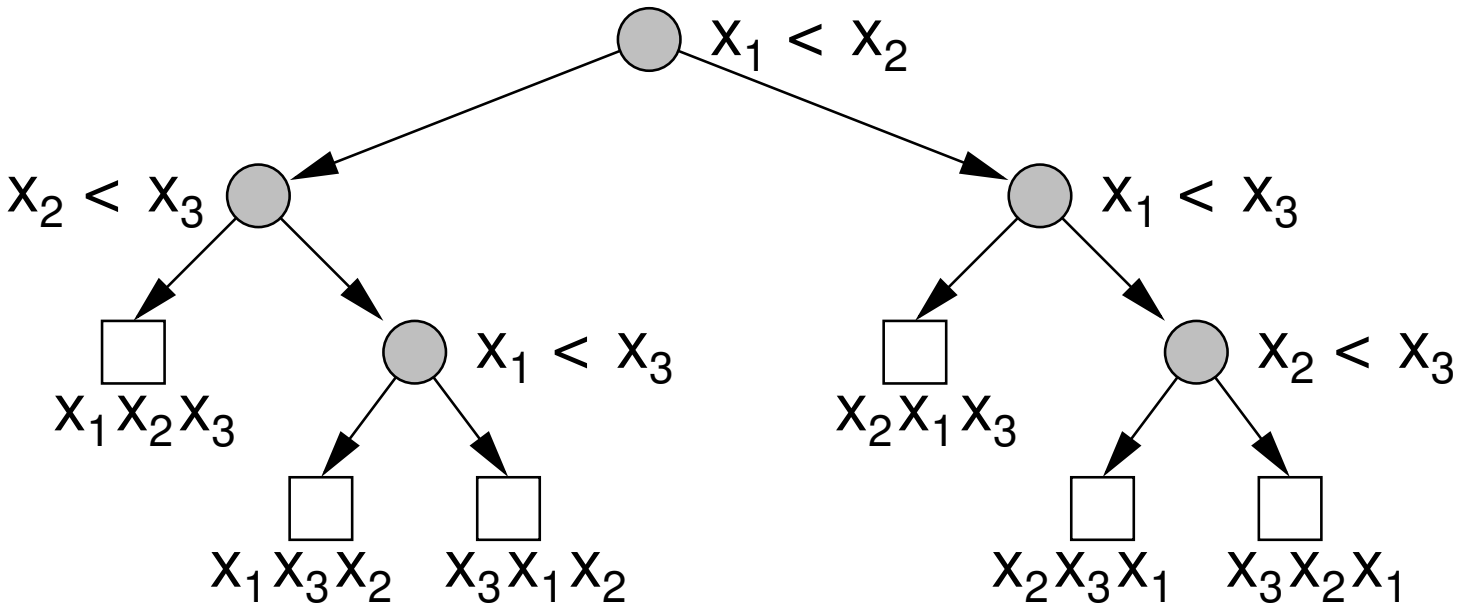
Satz

Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst case Zeit $\Omega(n \log n)$.

Untere Schranke

Vergleichsbasiertes Sortieren

Entscheidungsbaum mit Entscheidungen an den Knoten:



Untere Schranke

Vergleichsbasiertes Sortieren

muss insbesondere auch funktionieren, wenn alle n Schlüssel verschieden sind

$\Rightarrow$ Annahme: alle verschieden

Wieviele verschiedene Ergebnisse gibt es?
$\Rightarrow$ alle Permutationen:

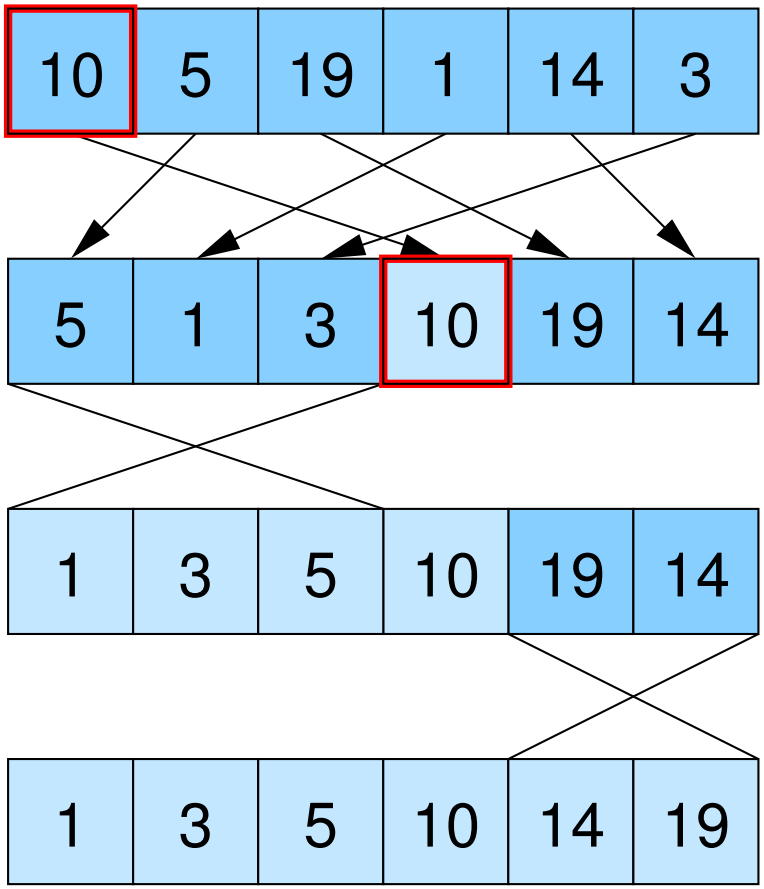
$$n! = \frac{n^n}{e^n}\sqrt{2\pi n}\left(1 + O\left(\frac{1}{n}\right)\right)$$

Binärbaum der Tiefe T hat höchstens 2^T Blätter
Oder: $T \geq \log_2(n!)$

$\Rightarrow T \geq n \log n - n \log e$

QuickSort

Idee: ähnlich wie bei MergeSort Aufspaltung in zwei Teilmengen, aber diesmal nicht einfach in der Mitte der Sequenz, sondern getrennt durch ein Pivotelement



QuickSort

```
void quickSort(int l, int r) {
  // a[l...r]: zu sortierendes Feld
  if (r > l) {
    v = a[r];  int i = l − 1;  int j = r;
    do { // spalte Elemente in a[l,...,r − 1] nach Pivot v
      do i++ while (a[i] < v);
      do j−− while (a[j] > v);
      if (i < j) swap(a[i], a[j]);
    } while (i < j);
    swap (a[i], a[r]);  // Pivot an richtige Stelle
    quickSort(l, i − 1);
    quickSort(i + 1, r);
}
```

QuickSort

Problem:

- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)

Lösungen:

- wähle zufälliges Pivotelement
  (Laufzeit $O(n \log n)$ mit hoher Wahrscheinlichkeit)
- berechne Median (mittleres Element)
  (mit Selektionsalgorithmus, spätere Vorlesung)

QuickSort

Laufzeit bei zufälligem Pivot-Element

- Zähle Anzahl Vergleiche (Rest macht nur konstanten Faktor aus)
- $\bar{C}(n)$: erwartete Anzahl Vergleiche bei n Elementen

Satz

$$\bar{C}(n) \leq 2n \ln n \leq 1.39 n \log n$$

QuickSort

Beweis.

- $s' = \langle e'_1, \ldots, e'_n \rangle$: sortierte Sequenz
- nur Vergleiche mit Pivotelement
- Pivotelement ist nicht in den rekursiven Aufrufen enthalten

$\Rightarrow$ e_i und e_j werden höchstens einmal verglichen und zwar dann, wenn e_i oder e_j Pivotelement ist

QuickSort

Beweis.

- Zufallsvariable $X_{ij} \in \{0, 1\}$
- $X_{ij} = 1 \quad \Leftrightarrow \quad e_i$ und e_j werden verglichen

$$\begin{aligned} \bar{C}(n) &= \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right] \\ &= \sum_{i<j} \Pr\left[X_{ij} = 1\right] \end{aligned}$$

QuickSort

Lemma

$$\Pr[X_{ij}] = 2/(j - i + 1)$$

Beweis.

- Sei $M = \{e_i, \ldots, e_j\}$
- Irgendwann wird ein Element aus M als Pivot ausgewählt.
- Bis dahin bleibt M immer zusammen.
- e_i und e_j werden genau dann *direkt* verglichen, wenn eines der beiden als Pivot ausgewählt wird
- Wahrscheinlichkeit:

$$\Pr[e_i \text{ oder } e_j \text{ aus } M \text{ ausgewählt}] = \frac{2}{|M|} = \frac{2}{j - i + 1}$$

□

QuickSort

Beweis.

$$
\begin{aligned}
\bar{C} &= \sum_{i<j} \Pr\left[X_{ij} = 1\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\leq 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{1}{k} = 2(n-1) \sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n - 1) \\
&\leq 2(n-1)(1 + \ln n - 1) \leq 2n \ln n
\end{aligned}
$$

□

Rang-Selektion

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

Lösung:

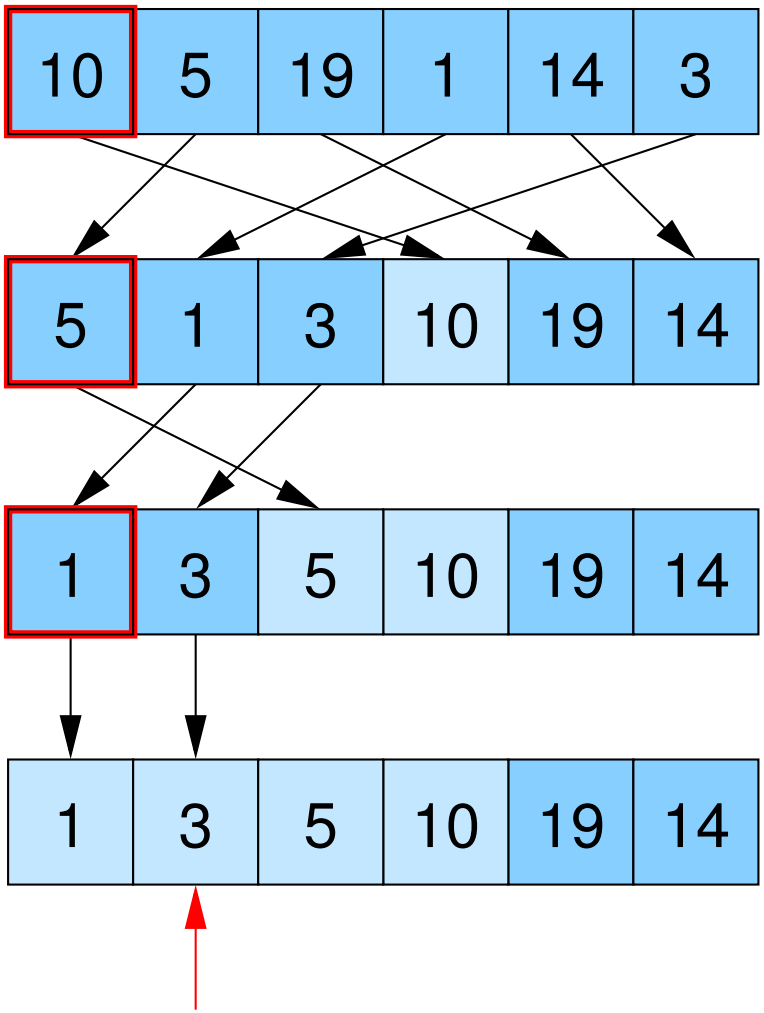
Sortiere Elemente
Gib k-tes Element aus

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

Geht das auch schneller?

QuickSelect

Ansatz: ähnlich zu QuickSort, aber nur eine Seite betrachten



QuickSelect

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

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

$\bar{C}(n)$: erwartete Anzahl Vergleiche

Satz

Die erwartete Laufzeit von QuickSelect ist linear: $\bar{C}(n) \in O(n)$.

Beweis.

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

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

QuickSelect

Beweis.

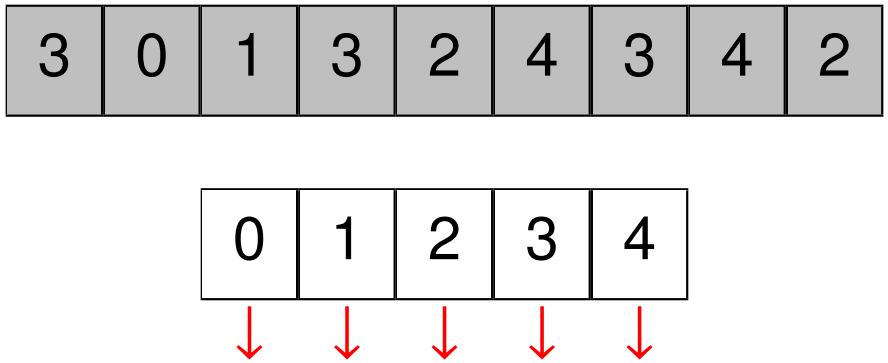
- Pivot gut: Restaufwand $\leq \bar{C}(2n/3)$
- Pivot schlecht: Restaufwand $\leq \bar{C}(n)$

$$
\begin{aligned}
\bar{C}(n) &\leq n + p \cdot \bar{C}(2n/3) + (1-p) \cdot \bar{C}(n) \\
p \cdot \bar{C}(n) &\leq n + p \cdot \bar{C}(2n/3) \\
\bar{C}(n) &\leq n/p + \bar{C}(2n/3) \\
\bar{C}(n) &\leq 3n + \bar{C}(2n/3) \\
&\leq 3(n + 2n/3 + 4n/9 + 8n/27 + \ldots) \\
&\leq 3n \sum_{i \geq 0} (2/3)^i \\
&\leq 3n/(1-2/3) = 9n
\end{aligned}
$$

□

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

- Annahme: Elemente im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Listenzeigern



Sortieren schneller als $\mathcal{O}(n \log n)$

```
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: $\mathcal{O}(n + K)$

Problem: nur gut für $K \in o(n \log n)$

RadixSort

- verwende K-adische Darstellung der Schlüssel
- sortiere Ziffer für Ziffer gemäß kSort
- 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: $\mathcal{O}(d(n+K))$ für n Schlüssel aus $\{0,\ldots,K^d-1\}$

Falls maximale Zahlengröße $\mathcal{O}(\log n)$, dann sind alle Zahlen $< n^d$ für konstantes d.
In diesem Fall Laufzeit $\mathcal{O}(n)$

RadixSort

Beispiel