

Proseminar

Effiziente Algorithmen

Kapitel 4: Sortieren

Prof. Dr. Christian Scheideler

WS 2016

Übersicht

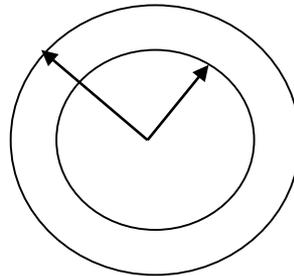
- Sortieren
- Selektieren
- Suchen

Sortierproblem



Ordnungen

- Ordnung auf **Zahlen**: klar
- Ordnung auf **Vektoren**: z.B. Länge des Vektors



- Ordnung auf **Namen**: lexikographische Ordnung (erst alle Namen, die mit A beginnen, dann B, usw.)

Heapsort

Procedure Heapsort(a : Array $[1..n]$ of
Element)

$H := \text{buildHeap}(a)$; // Zeit $O(n)$

for $i := 1$ to n do

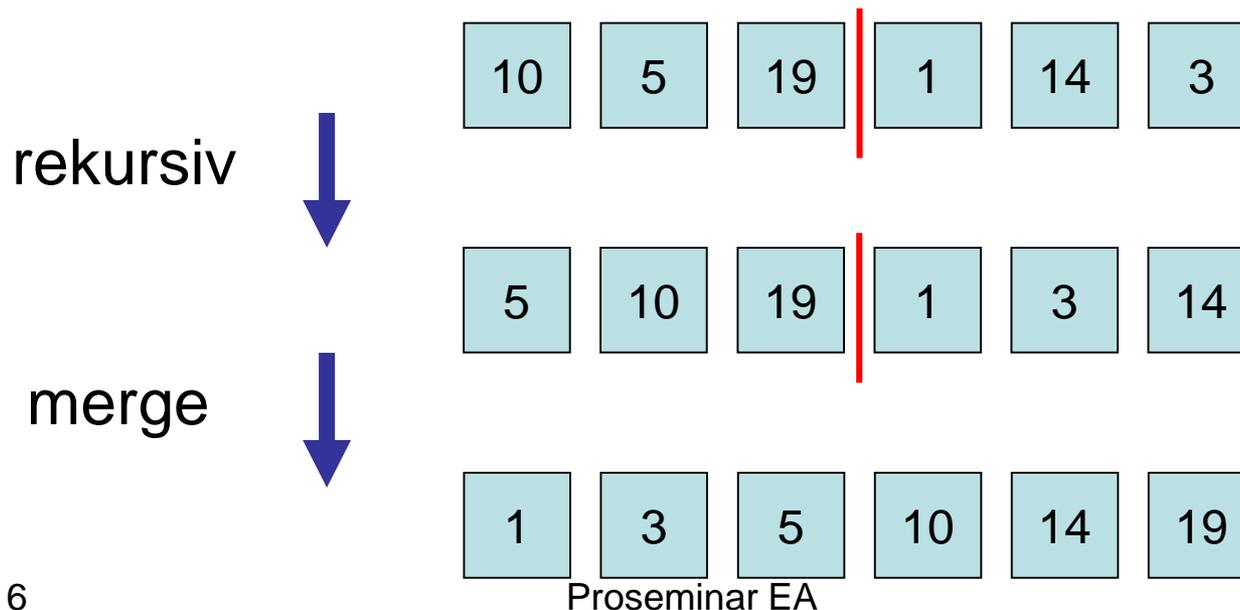
// speichere Minimum in H in $a[i]$

$a[i] := \text{deleteMin}(H)$ // Zeit $O(\log n)$

Also insgesamt Laufzeit $O(n \log n)$.

Mergesort

Idee: zerlege Sortierproblem rekursiv in Teilprobleme, die separat sortiert werden und dann verschmolzen werden

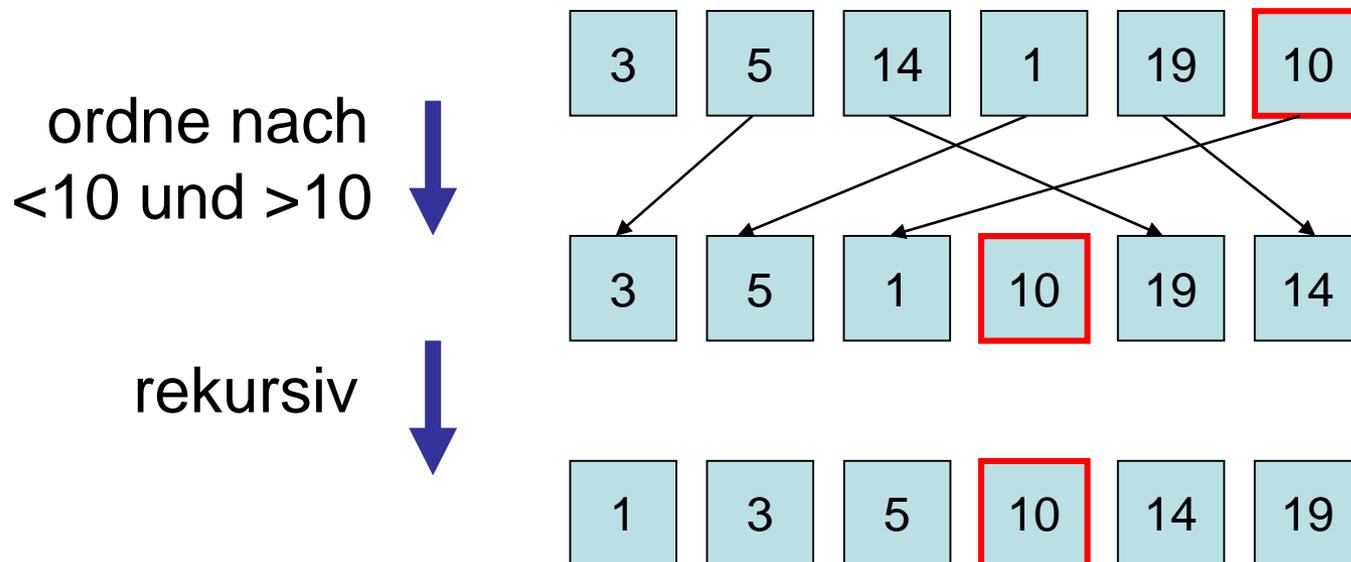


Mergesort

```
Procedure Mergesort(l,r: Integer)
  // a[l..r]: zu sortierendes Feld
  if l=r then return // fertig
  m:=  $\lfloor (r+l)/2 \rfloor$  // Mitte
  Mergesort(l,m)
  Mergesort(m+1,r)
  j:=l; k:=m+1
  for i:=1 to r-l+1 do // in Hilfsfeld b mergen
    if j>m then b[i]:=a[k]; k:=k+1
    else
      if k>r then b[i]:=a[j]; j:=j+1
      else
        if a[j]<a[k] then b[i]:=a[j]; j:=j+1
        else b[i]:=a[k]; k:=k+1
  for i:=1 to r-l+1 do a[l-1+i]:=b[i] // b zurückkopieren
```

Quicksort

Idee: ähnlich wie Mergesort, aber Aufspaltung in Teilfolgen nicht in Mitte sondern nach speziellem Pivotelement

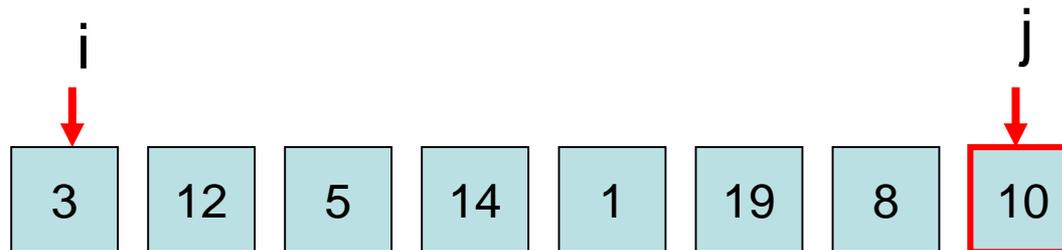


Quicksort

```
Procedure Quicksort(l,r: Integer)
// a[l..r]: zu sortierendes Feld
if r>l then
  v:=a[r]; i:=l-1; j:=r
  repeat // ordne Elemente in [l,r-1] nach Pivot v
    repeat i:=i+1 until a[i]>=v
    repeat j:=j-1 until a[j]<v or j=l
    if i<j then a[i] ↔ a[j]
  until j<=i
  a[i] ↔ a[r] // bringe Pivot an richtige Position
  Quicksort(l,i-1) // sortiere linke Teilfolge
  Quicksort(i+1,r) // sortiere rechte Teilfolge
```

Quicksort

Beispiel für einen Durchlauf mit Pivot 10:



Quicksort

Problem: im worst case kann Quicksort $\Theta(n^2)$ Laufzeit haben (wenn schon sortiert)

Lösungen:

- wähle **zufälliges** Pivotelement
(Laufzeit $O(n \log n)$ mit hoher W.keit)
- berechne Median (Element in Mitte)
→ dafür Selektionsalgorithmus (später)

Ist Kostenmodell fair?

Bisher haben wir angenommen: jeder Vergleich kostet eine Zeiteinheit.

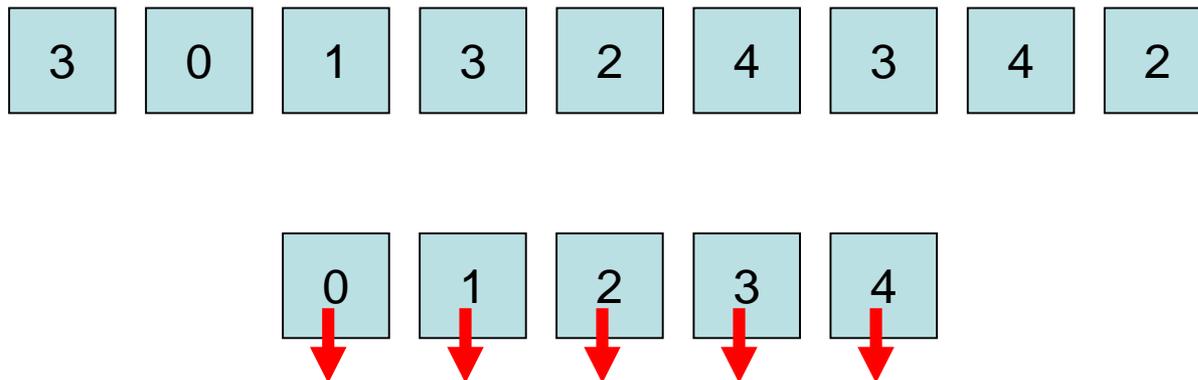
Besser: Bitmodell

Eingabe besteht aus n Bits. Vergleich zwischen zwei Zahlen aus k Bits kostet $O(k)$ Zeit.

- Sei n_i die Anzahl Bits, die zu Zahlen der Größe $[2^i, 2^{i+1})$ gehören.
- Sortierung dieser Zahlen kostet $O(i (n_i/i) \log (n_i/i)) = O(n_i \log n)$ Zeit
- Zeitaufwand insgesamt: $\sum_i O(n_i \log n) = O(n \log n)$

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

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



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

```
Procedure KSort(S: List of Element)
  b: Array [0..K-1] of List of Element
  foreach  $e \in S$  do
    // hänge e hinten an Liste b[key(e)] an
    b[key(e)].pushBack(e)
  // binde Listen zusammen zu einer Liste S
  S:=concatenate(b[0],...,b[K-1])
```

Laufzeit: $O(n+K)$ (auch für Bitkomplexität)

Problem: nur sinnvoll für $K=o(n \log n)$

Radixsort

Ideen:

- verwende K -adische Darstellung der Schlüssel
- Sortiere Ziffer für Ziffer gemäß K Sort
- Behalte Ordnung der Teillisten bei

Annahme: alle Zahlen $\leq K^d$

Radixsort

```
Procedure Radixsort(s: Sequence of Element)
  for i:=0 to d-1 do
    KSort(s,i) // sortiere gemäß  $key_i(x)$ 
               // mit  $key_i(x) = (key(x) \text{ div } K^i) \text{ mod } K$ 
```

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

Angenommen, alle Zahlen $< n^d$ für konstantes d .
In diesem Fall ($K=n$) Laufzeit $O(n)$.

Radixsort

Beispiel:



Ordnung nach Einerstelle:

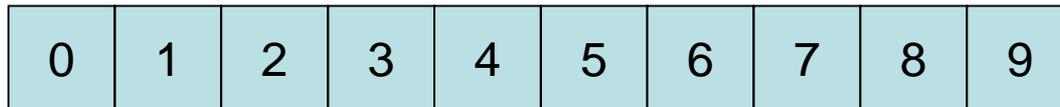


Radixsort

Ergebnis nach Einerstelle:



Ordnung nach Zehnerstelle:



Radixsort

Ergebnis nach Zehnerstelle:



Ordnung nach Hunderterstelle:



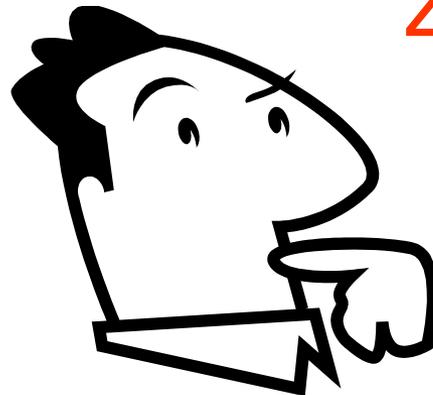
Radixsort

Ergebnis nach Hunderterstelle:



Sortiert!

Zauberei???



Radixsort

Korrektheit:

- Für jedes Paar x, y mit $\text{key}(x) < \text{key}(y)$ gilt: es existiert i mit $\text{key}_i(x) < \text{key}_i(y)$ und $\text{key}_j(x) = \text{key}_j(y)$ für alle $j > i$
- Schleifendurchlauf für i : $\text{pos}_S(x) < \text{pos}_S(y)$ ($\text{pos}_S(z)$: Position von z in Folge S)
- Schleifendurchlauf für $j > i$: Ordnung wird **beibehalten** wegen pushBack in KSort

Genaueres Kostenmodell

Annahmen:

- ein Wort besteht aus $\Theta(\log n)$ Bits
($\Omega(\log n)$ Bits wegen Adressierbarkeit)
- Operationen auf Wörtern kosten eine Zeiteinheit
- Alle n Zahlen der Eingabe bestehen aus W Worten, also Eingabegröße ist $W \cdot n$

Laufzeit von Radixsort:

$O(W \cdot n)$, was optimal ist.

Problem: was ist, wenn die Zahlen unterschiedliche Längen haben??

Genaueres Kostenmodell

Problem: was ist, wenn die Zahlen unterschiedliche Längen haben??

Neben Radixsort benötigen wir einen weiteren Sortieralgo: **Patriciasort.**

Grundlegender Ansatz: Patriciasort baut Patricia Trie der Zahlen auf.

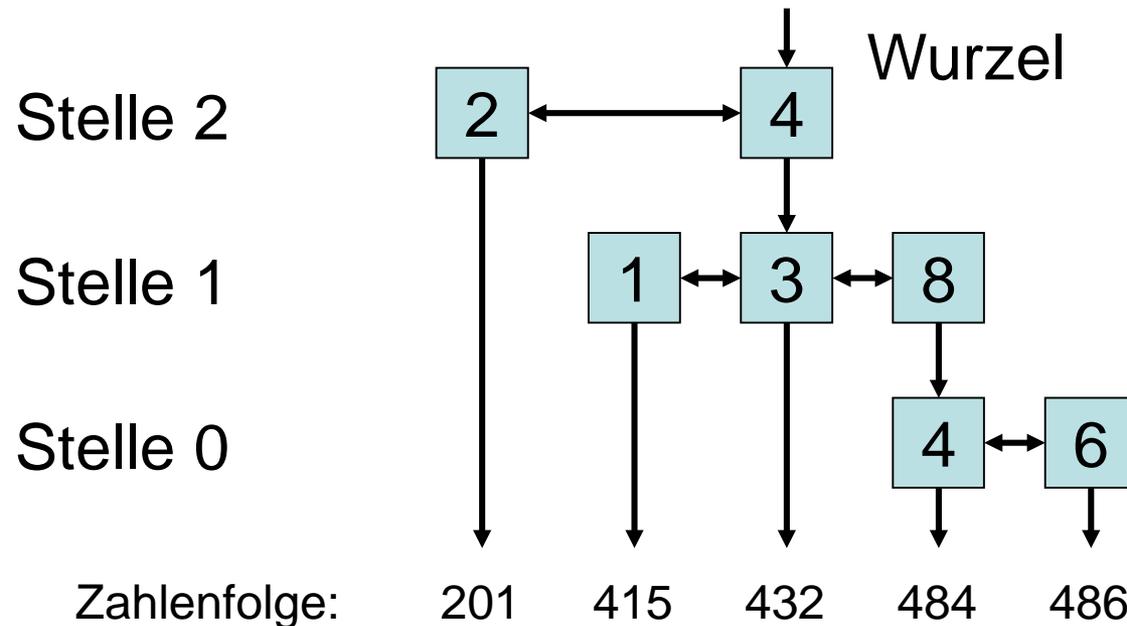
Patriciasort

Annahme: Ziffern aus $\{0, \dots, K-1\}$

Eingabe: n Zahlen mit Werten $< K^d$ (max d Ziffern)

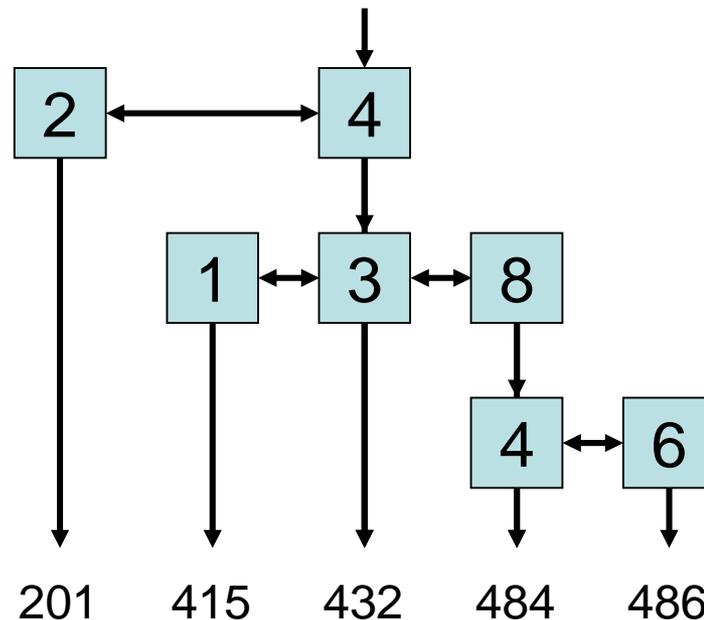
Beispiel: $K=10$, $d=3$

Ziel:



Patriciasort

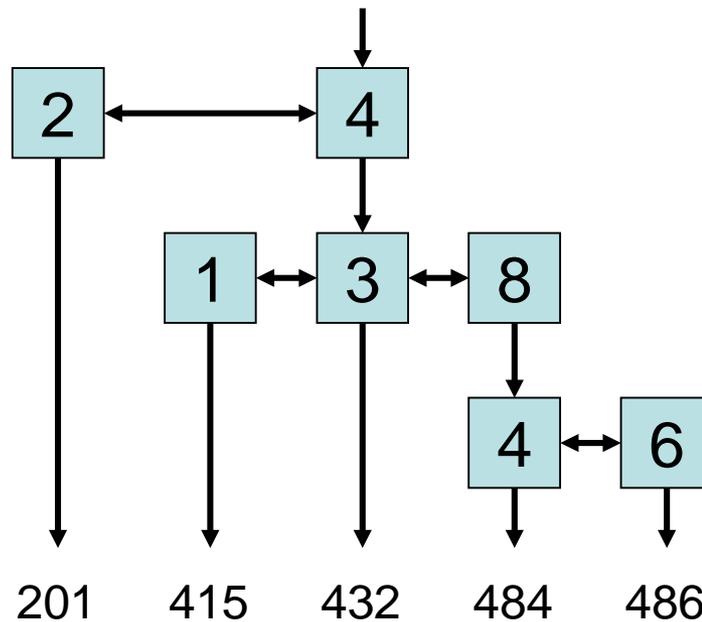
Aufbau für Eingabe (432,415,201,484,486):



Einbauzeit pro Zahl: $O(\#Zahlen + \#Ziffern \text{ pro Zahl}) = O(n+d)$

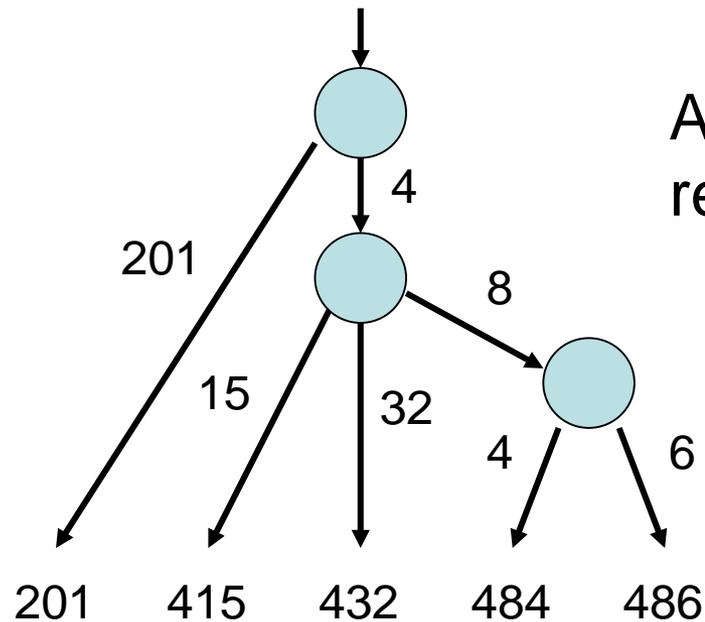
Patriciasort

Ist wie Patricia Trie, nur mit Listen



Patriciasort

Idealer Patricia Trie ohne Listen:



Aber Listenstruktur
reicht für Effizienz.

Patriciasort

Gegeben: n Zahlen $a_1, \dots, a_n < K^d$.

Patriciasort:

for $i:=1$ to n do

füge Zahl a_i in Patricia Trie mit Listen
ein // Zeit $O(i+d)$

durchlaufe Patricia Trie und gib sortierte
Folge aus

Laufzeit: $O(\sum_i (i+d)) = O(n(n+d))$

Sortierung beliebiger Zahlen

Eingabe: Folge M von Zahlen, die zusammen n Worte der Größe $\Theta(\log n)$ umfassen

LinearSort(M):

- Teile Zahlen in Mengen M_i ein, wobei M_i aus Zahlen der Länge in $[2^i, 2^{i+1})$ besteht und n_i Zahlen umfasse
- Für jede Menge M_i , betrachte 3 Fälle:
 1. $n_i \leq 2^i$: Patriciasort(M_i)
 2. $n_i > 2^i$ und $n_i > n^{1/3}$: Radixsort(M_i)
 3. $n_i > 2^i$ und $n_i \leq n^{1/3}$: Patriciasort(M_i)
- Hänge sortierte Folgen hintereinander

Sortierung beliebiger Zahlen

Theorem 5.5: LinearSort hat Laufzeit $O(n)$.

Beweis:

- Fall 1: Laufzeit: $O(n_i(n_i+2^i)) = O(n_i 2^i)$
- Fall 2: Laufzeit: $O(2^i(n_i+n^{1/3})) = O(n_i 2^i)$
- Fall 3: Laufzeit: $O(n_i(n_i+2^i)) = O(n_i^2) = O(n^{2/3})$
- Gesamtlaufzeit:
 $O(\sum_i (n_i 2^i + n^{2/3})) = O(n)$.

Übersicht

- Sortieren
- **Selektieren**
- Suchen

Selektion

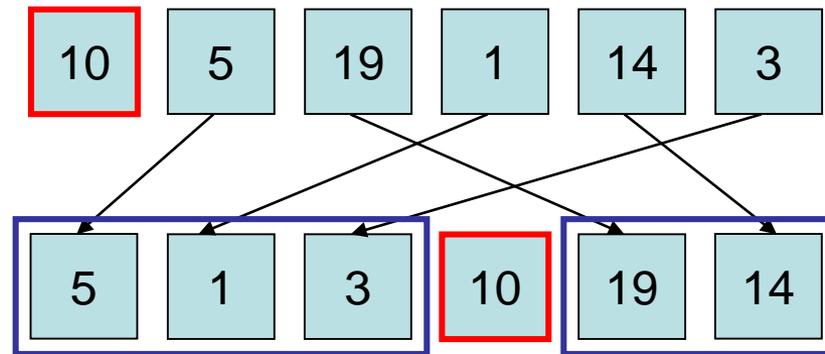
Problem: finde k -kleinstes Element in einer Folge von n Elementen

Lösung: sortiere Elemente (z.B. Mergesort), gib k -tes Element aus \rightarrow Zeit $O(n \log n)$

Geht das auch schneller??

Selektion

Ansatz: verfahren ähnlich zu Quicksort



- j : Position des Pivotelements
- $k < j$: mach mit linker Teilfolge weiter
- $k > j$: mach mit rechter Teilfolge weiter

Selektion

```
Function Quickselect(l,r,k: Integer): Element
// a[l..r]: Restfeld, k: k-kleinstes Element, l<=k<=r
if r=l then return a[l]
z:=zufällige Position in {l,...,r}; a[z] ↔ a[r]
v:=a[r]; i:=l-1; j:=r
repeat // ordne Elemente in [l,r-1] nach Pivot v
  repeat i:=i+1 until a[i]>=v
  repeat j:=j-1 until a[j]<v or j=l
  if i<j then a[i] ↔ a[j]
until j<=i
a[i] ↔ a[r]
if k<i then e:=Quickselect(l,i-1,k)
if k>i then e:=Quickselect(i+1,r,k)
if k=i then e:=a[k]
return e
```

Quickselect

- $C(n)$: erwartete Anzahl Vergleiche

Satz: $C(n) = O(n)$

Beweis:

- Pivot ist **gut**: keine der Teilfolgen länger als $2n/3$
- Sei $p = \Pr[\text{Pivot ist gut}]$



- $p = 1/3$

BFPRT-Algorithmus

Gibt es auch einen deterministischen Selektionsalgorithmus mit linearer Laufzeit?

Ja, den **BFPRT-Algorithmus** (benannt nach den Erfindern Blum, Floyd, Pratt, Rivest und Tarjan).

BFPRT-Algorithmus

- Sei m eine ungerade Zahl ($5 \leq m \leq 21$).
- Betrachte die Zahlenmenge $S = \{a_1, \dots, a_n\}$.
- Gesucht: k -kleinste Zahl in S

Algorithmus BFPRT(S, k):

1. Teile S in $\lceil n/m \rceil$ Blöcke auf, davon $\lfloor n/m \rfloor$ mit m Elementen
2. Sortiere jeden dieser Blöcke (z.B. mit Insertionsort)
3. $S' :=$ Menge der $\lceil n/m \rceil$ Mediane der Blöcke.
4. $m := \text{BFPRT}(S', |S'|/2)$ // berechnet Median der Mediane
5. $S_1 := \{x \in S \mid x < m\}$; $S_2 := \{x \in S \mid x > m\}$
6. Falls $k \leq |S_1|$, dann return $\text{BFPRT}(S_1, k)$
7. Falls $k > |S_1| + 1$, dann return $\text{BFPRT}(S_2, k - |S_1| - 1)$
8. return m

BFPRT-Algorithmus

Laufzeit $T(n)$ des BFPRT-Algorithmus:

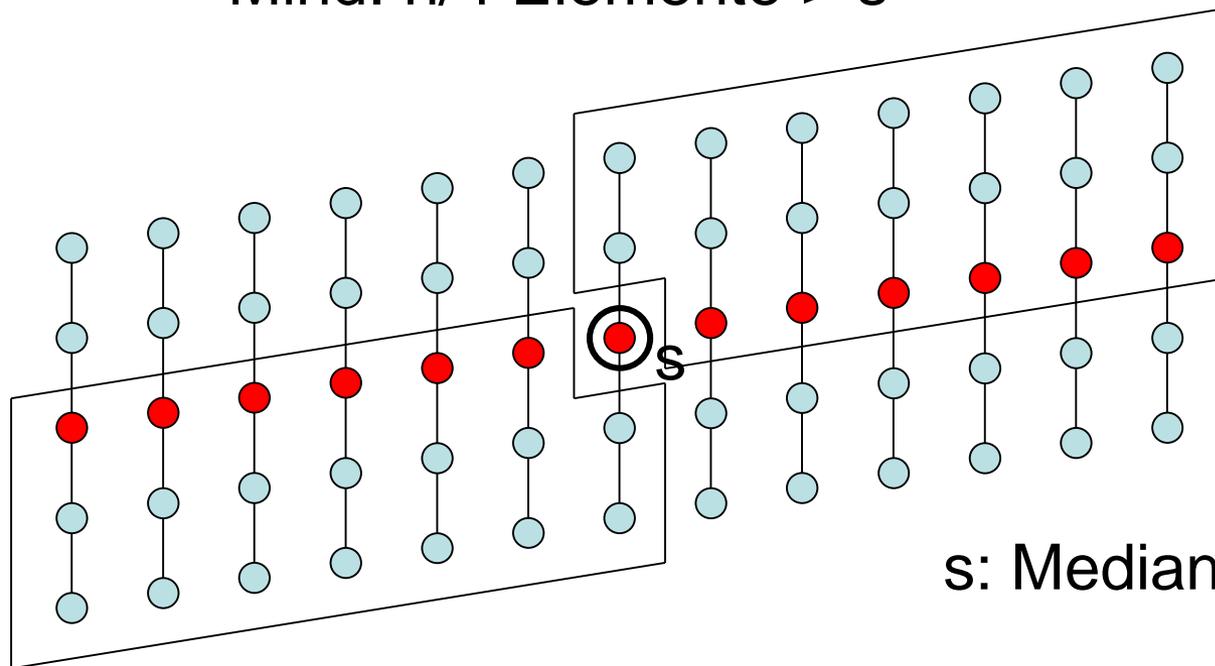
- Schritte 1-3: $O(n)$
- Schritt 4: $T(\lceil n/m \rceil)$
- Schritt 5: $O(n)$
- Schritt 6 bzw. 7: ???

Lemma: Schritt 6/7 ruft BFPRT mit maximal $\lfloor (3/4)n \rfloor$ Elementen auf

BFPRT-Algorithmus

Beweis: O.B.d.A. sei $m=5$

Mind. $n/4$ Elemente $> s$



● : Median

s: Median der Mediane

Mind. $n/4$ Elemente $< s$

BFPRT-Algorithmus

Laufzeit für $m=5$:

$$T(n) \leq T(\lfloor (3/4)n \rfloor) + T(\lceil n/5 \rceil) + c \cdot n$$

für eine Konstante c .

Satz: $T(n) \leq d \cdot n$ für eine Konstante d .

Beweis: Übung.

Übersicht

- Sortieren
- Selektieren
- Suchen

Suchen

- Balancierter Suchbaum
AVL-Baum, rot-schwarz-Baum, Splay
Baum (siehe Kapitel 2)
- Hashing (siehe Kapitel 2)

Probleme

- 10191: Longest Nap
- 612: DNA Sorting
- 10037: Bridge
- 10152: ShellSort
- 10041: Vito's family
- 10026: Shoemaker's Problem

Hausaufgabe:

- 855: Lunch in Grid City