Proseminar
Effiziente Algorithmen

Kapitel 4: Sortieren, Selektieren und Suchen

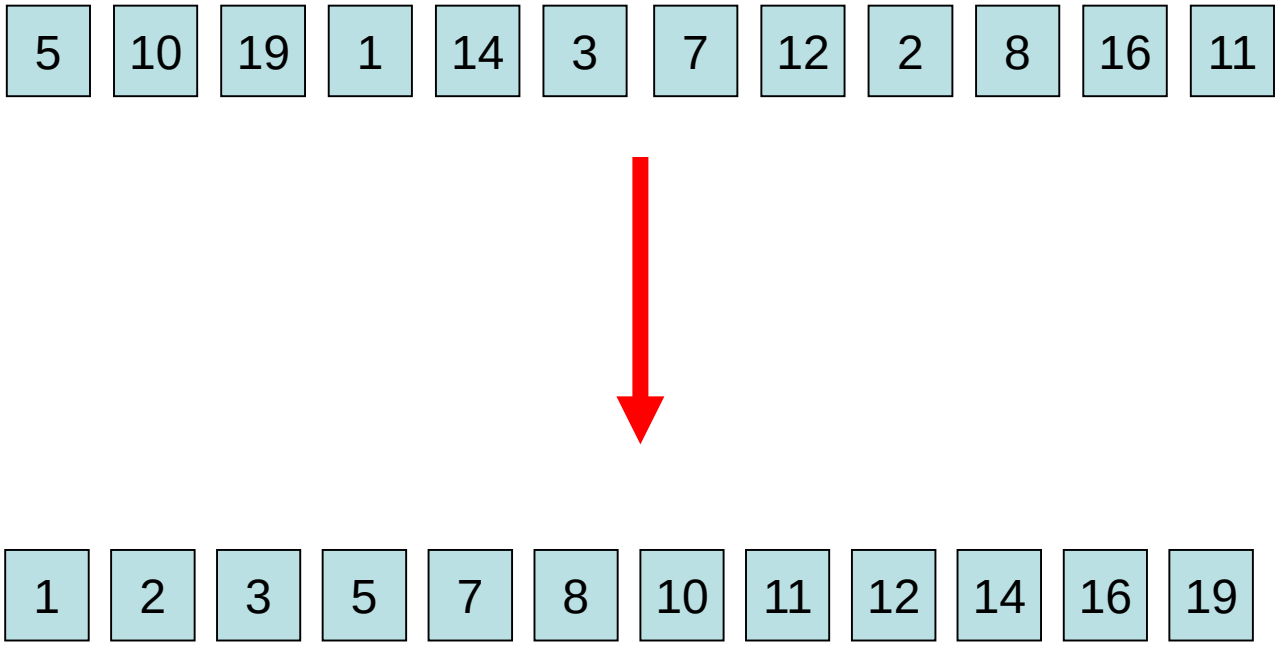
Prof. Dr. Christian Scheideler

WS 2017

Übersicht

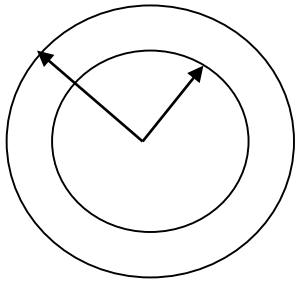
- Sortieren
- Selektieren
- Suchen

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.)

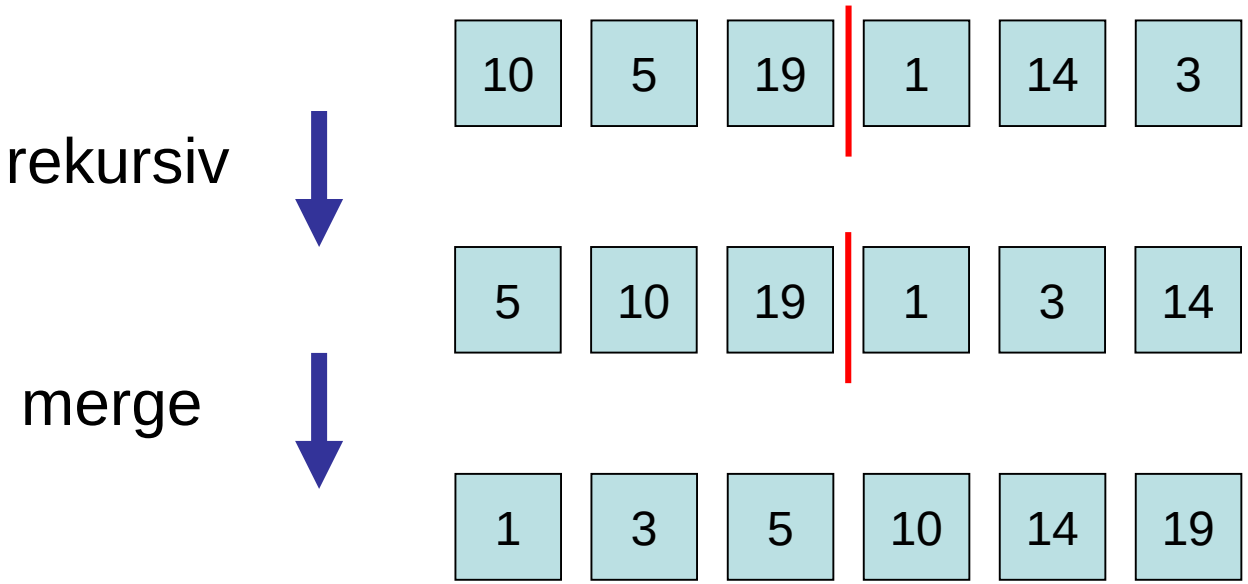
Heapsort

```
Procedure Heapsort(a: Array [1..n] of
   Element)
   H:=buildHeap(a);        // Zeit O(n)
   for i:=1 to n do
      // speichere Minimum in H in a[i]
      a[i]:=deleteMin(H)  // Zeit O(log n)
```

Also insgesamt Laufzeit O(n log n).

Mergesort

Idee: zerlege Sortierproblem rekursiv in Teil-probleme, die separat sortiert werden und dann verschmolzen werden



Mergesort

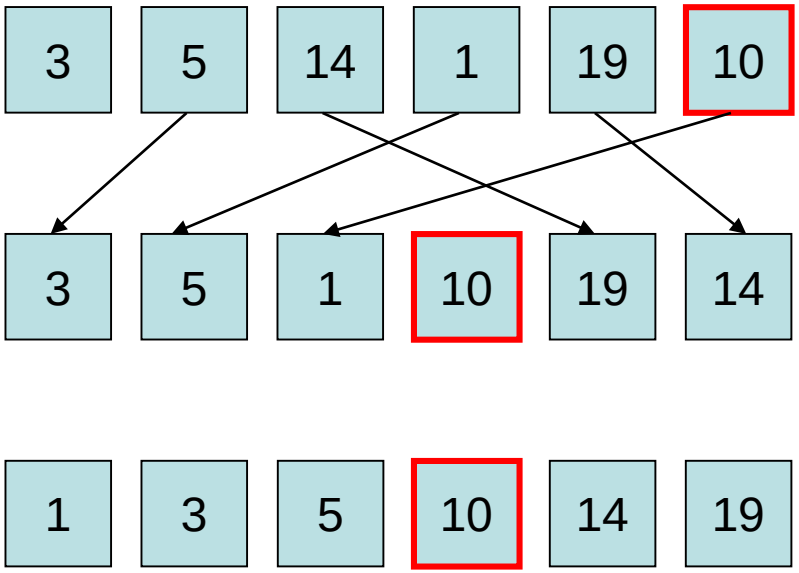
```
Procedure Mergesort(l,r: Integer)
    // a[l..r]: zu sortierendes Feld
    if l=r then return      // fertig
    m:= ⌊(r+l)/2⌋           // Mitte
    Mergesort(l,m)
    Mergesort(m+1,r)
    j:=l; k:=m+1
    for i:=1 to r-l+1 do   // in Hilfsfeld b mergen
        if j>m then b[i]:=a[k]; k:=k+1
  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

ordne nach <10 und >10

rekursiv

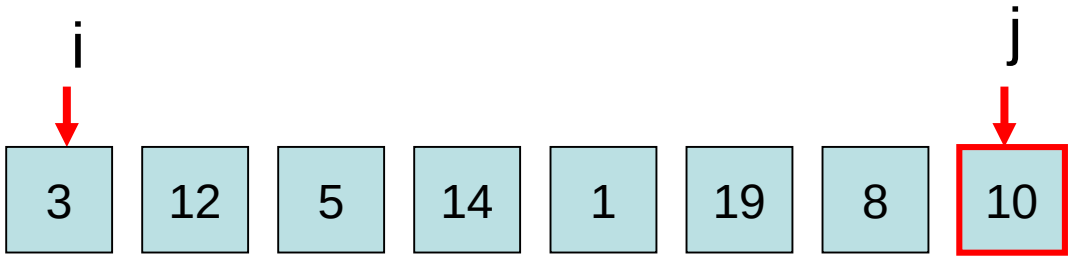


Quicksort

```
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)
  $\rightarrow$ dafür Selektionsalgorithmus (später)

Ist Kostenmodell fair?

Bisher haben wir angenommen: jeder Vergleich kostet eine Zeiteinheit.

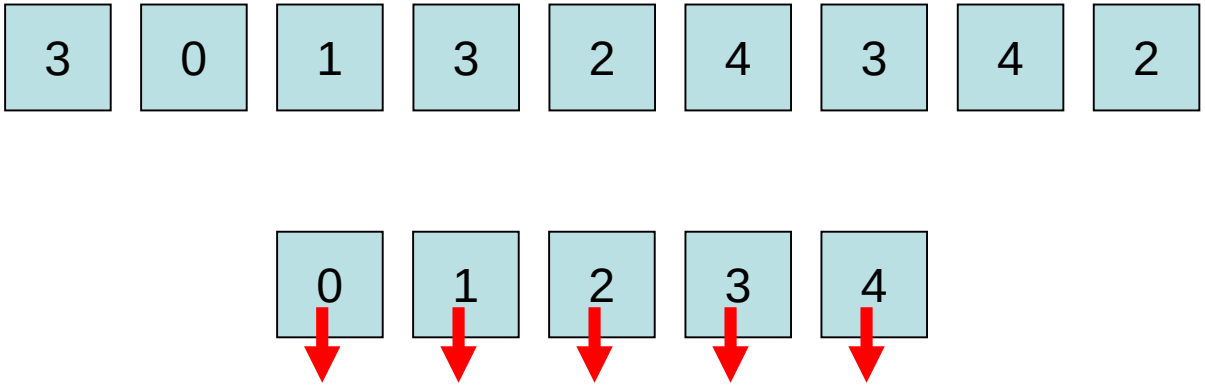
Besser: Bitmodell

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,…,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∈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 KSort
- 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) div K^i) 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:

12	203	3	74	24	17	112

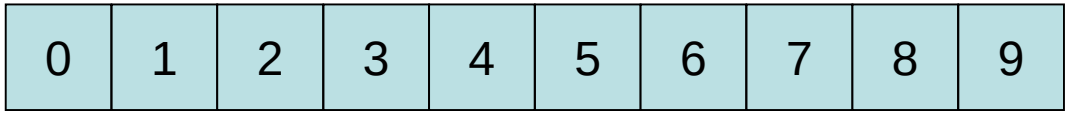
Ordnung nach Einerstelle:

0	1	2	3	4	5	6	7	8	9

Radixsort

Ergebnis nach Einerstelle:

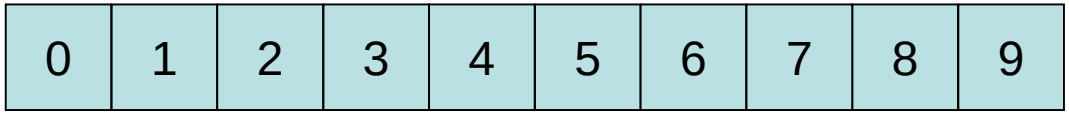
Ordnung nach Zehnerstelle:



Radixsort

Ergebnis nach Zehnerstelle:

Ordnung nach Hunderterstelle:



Radixsort

Ergebnis nach Hunderterstelle:

3	12	17	24	74	112	203

Sortiert!

Zauberei???

Radixsort

Korrektheit:

- Für jedes Paar x,y mit key(x)<key(y) gilt: es existiert i mit $key_i(x)<key_i(y)$ und $key_j(x)=key_j(y)$ für alle j>i
- Schleifendurchlauf für i: $pos_S(x)<pos_S(y)$ ($pos_S(z)$: 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.

Übersicht

- Sortieren
- Selektieren
- Suchen

Selektion

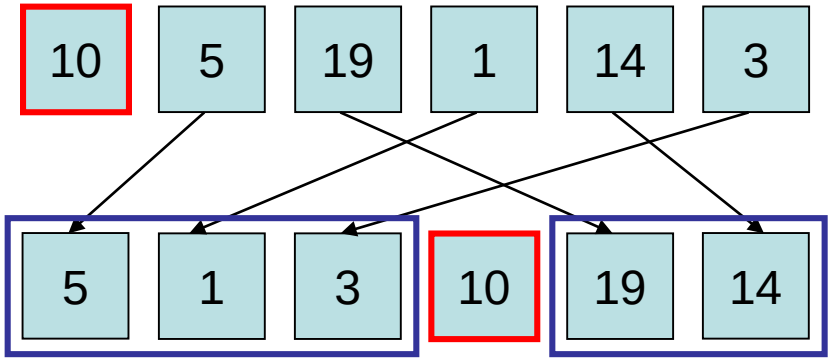
Problem: finde k-kleinstes Element in einer Folge von n Elementen

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

Geht das auch schneller??

Selektion

Ansatz: verfahre 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[Pivot ist gut]

- p=1/3

BFPRT-Algorithmus

Gibt es auch einen deterministischen Selek-tionsalgorithmus 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 $S=\{a_1,\ldots,a_n\}$.
- 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. s:=BFPRT(S',$|S'|/2$) // berechnet Median der Mediane
5. $S_1:=\{ x \in S \mid x<s\}$; $S_2:=\{x \in S \mid x>s\}$
6. Falls $k \leq |S_1|$, dann return BFPRT(S_1,k)
7. Falls $k>|S_1|+1$, dann return BFPRT(S_2,$k-|S_1|-1$)
8. return m

BFPRT-Algorithmus

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

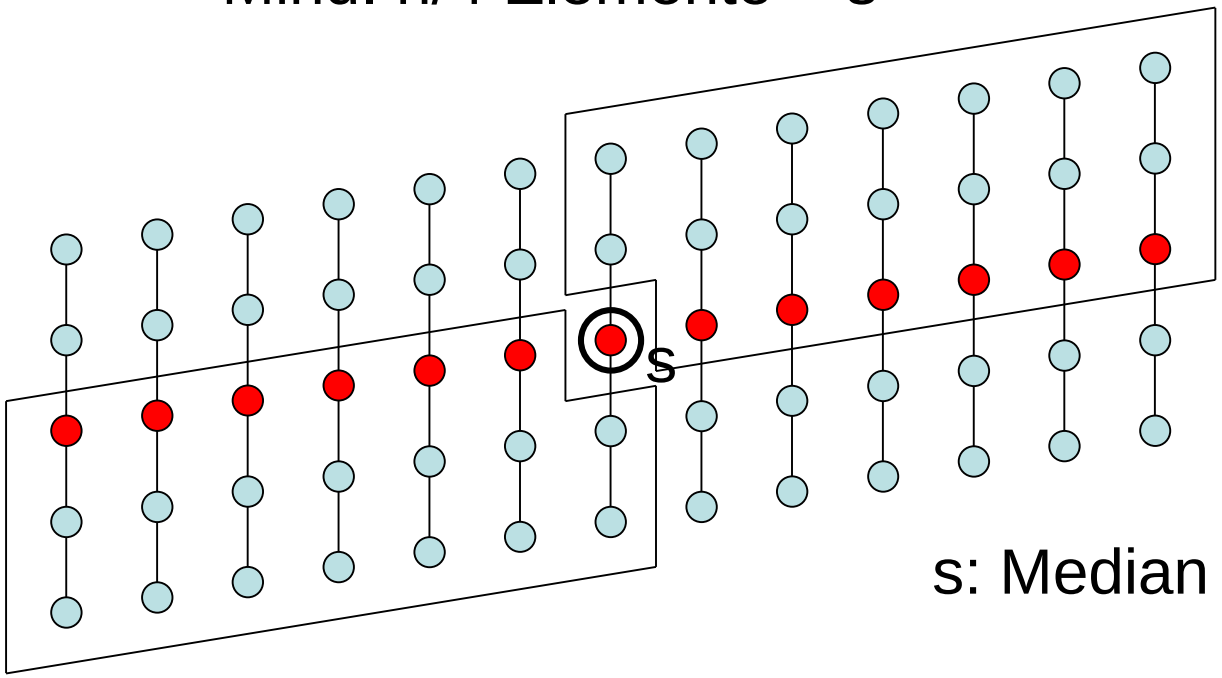
- Schritte 1-3: $O(n)$
- Schritt 4: $T(\lceil n/m \rceil)$
- Schritt 5: $O(n)$
- Schritt 6 bzw. 7: ???

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
- 10152: ShellSort
- 10041: Vito‘s family
- 10037: Bridge
- 10026: Shoemaker‘s Problem

Hausaufgabe:

- 855: Lunch in Grid City