RUB

# Vorlesung Datenstrukturen

## Sortieren und Auswählen

Maike Buchin

9., und 11.5.2017

# Motivation

häufig müssen Daten sortiert werden, z.B.

- zum Suchen
- zum Extrahieren von Teilmengen
- zum Löschen von doppelten Einträgen
- als Vorverarbeitung

# Sortieren

RUB

**Gegeben:** eine Folge $s = \langle e_1, \ldots, e_n \rangle$ von $n$ Einträgen.
Jeder Eintrag $e \in S$ hat einen Schlüssel key$(e) \in Key =: k_i$.
Die Schlüssel stammen aus einem geordneten Universum,
d.h. auf ihnen ist eine lineare Ordnung definiert.
Wir übertragen die Ordnung auf die Einträge, d.h. $e \leq e'$
bedeutet key$(e) \leq$ key$(e')$.
**Gesucht:** Permutation $s' = \langle e'_1, \ldots, e'_n \rangle$ von $s$ mit
$e'_1 \leq \ldots \leq e'_n$.

**Beachte:** Anordnung von Einträgen mit gleicher Ordnung ist beliebig.

Für einen Datentyp kommen versch Vergleichsrelationen in Frage.
Häufig haben Datentypen eine natürliche Ordnungsrelation.

- Zahlen: natürliche Anordnung
- Buchstaben: alphabetisch
- Tupel, strings, folgen: lexikographisch

# Übersicht

RUB

- einfache Sortierverfahren
- Mergesort
- untere Schranke
- Quicksort
- Auswahlproblem
- schneller Sortieren

# 5.1 Einfache Sortierverfahren

RUB

**Auswahlsortieren** (engl. Selection Sort)
*wiederhole:* wähle den kleinsten Eintrag der Eingabefolge, lösche ihn dort, und füge ihn an die Ausgabefolge an.

starte mit leerer Ausgangsfolge und wiederhole bis Eingabefolge leer ist.

**selectionSort**$(a[1 \ldots n])$
for $j = 1$ to $n - 1$ do
  $m = j$
  for $i = j + 1$ to $n$ do
    if $(a[i] < a[m])$ then $m = i$
  if $(m \neq j)$ then swap $(a[j], a[m])$

**Beispiel:**

$$\langle\rangle, \langle 4,7,1,1\rangle \rightsquigarrow \langle 1\rangle, \langle 4,7,1\rangle \rightsquigarrow \langle 1,1\rangle, \langle 4,7\rangle \rightsquigarrow \langle 1,1,4\rangle, \langle 7\rangle \rightsquigarrow \langle 1,1,4,7\rangle, \langle\rangle$$

# 5.1 Einfache Sortierverfahren

RUB

**Auswahlsortieren** (engl. Selection Sort)
*wiederhole:* wähle den kleinsten Eintrag der Eingabefolge, lösche ihn dort, und füge ihn an die Ausgabefolge an.

starte mit leerer Ausgangsfolge und wiederhole bis Eingabefolge leer ist.

**selectionSort**$(a[1 \ldots n])$
for $j = 1$ to $n - 1$ do
  $m = j$
  for $i = j + 1$ to $n$ do
    if $(a[i] < a[m])$ then $m = i$
  if $(m \neq j)$ then swap $(a[j], a[m])$

**Laufzeit:** Anzahl der Vergleiche $=$
$\sum_{j=1}^{n-1}(n-j) = \sum_{i=1}^{n-1} i = n(n-1)/2 = O(n^2)$

**Korrektheit:** folgt mit Schleifeninvariante: die Ausgabefolge enthält nach dem $j-$ten Durchlauf die $j$ kleinsten Elemente sortiert

# 5.1 Einfache Sortierverfahren

**Einfügesortieren** (engl. Insertion Sort)
*wiederhole:* entnehme ersten Eintrag der Eingabefolge, und füge ihn an richtiger Stelle in der Ausgabefolge ein.

starte mit leerer Ausgangsfolge und wiederhole bis Eingabefolge leer ist.

**insertionSort**$(a[1 \ldots n])$
for $i = 2$ to $n$ do
  $e := a[i]$
  if $e < a[1]$ then
    for $j = i$ downto 2 do $a[j] := a[j-1]$
    $a[1] := e$
  else
    for $j = i$ downto 2 while $a[j-1] > e$ do $a[j] := a[j-1]$
    $a[j] := e$

**Beispiel:**

$$\langle\rangle, \langle 4,7,1,1\rangle \rightsquigarrow \langle 4\rangle, \langle 7,1,1\rangle \rightsquigarrow \langle 4,7\rangle, \langle 1,1\rangle \rightsquigarrow \langle 1,4,7\rangle, \langle 1\rangle \rightsquigarrow \langle 1,1,4,7\rangle, \langle\rangle$$

# 5.1 Einfache Sortierverfahren

RUB

**Einfügesortieren** (engl. Insertion Sort)
*wiederhole:* entnehme ersten Eintrag der Eingabefolge, und füge ihn an richtiger Stelle in der Ausgabefolge ein.

starte mit leerer Ausgangsfolge und wiederhole bis Eingabefolge leer ist.

**insertionSort**$(a[1 \ldots n])$
for $i = 2$ to $n$ do
  $e := a[i]$
  if $e < a[1]$ then
    for $j = i$ downto 2 do $a[j] := a[j-1]$
    $a[1] := e$
  else
    for $j = i$ downto 2 while $a[j-1] > e$ do $a[j] := a[j-1]$
    $a[j] := e$

**Laufzeit:** Anzahl der Vergleiche $= \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = O(n^2)$

**Korrektheit:** folgt mit Schleifeninvariante: die Ausgabefolge enthält nach dem $j-$ten Durchlauf die $j$ ersten Elemente der Eingabe sortiert

# 5.2 Mergesort

*Teile-und-Herrsche* Algorithmus

Die unsortierte Folge wird in zwei etwa gleich große Teile geteilt.
Die beiden Teile werden rekursiv sortiert.
Danach werden die sortierten Teilfolgen zusammengefügt.

Zwei sortierte Folgen lassen sich effizient zusammen fügen: der insgesamt kleinste Eintrag ist der kleinste von einer der Folgen. Verschiebe diesen in die Ausgabe und iteriere.

**mergeSort**$(\langle e_1, \ldots, e_n \rangle)$
if $n = 1$ then return $\langle e_1 \rangle$
else
  $S_1 =$ mergeSort$(\langle e_1, \ldots, e_{\lfloor \frac{n}{2} \rfloor} \rangle)$
  $S_2 =$ mergeSort$(\langle e_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, e_n \rangle)$
  return merge$(S_1, S_2)$

**merge**$(a, b)$
$C := \langle\rangle$
loop
  if a.isEmpty then return c.concat(b)
  if b.isEmpty then return c.concat(a)
  if (a.first $\leq$ b.first) then
c.moveToBack(a.first)
  else c.moveToBack(b.first)

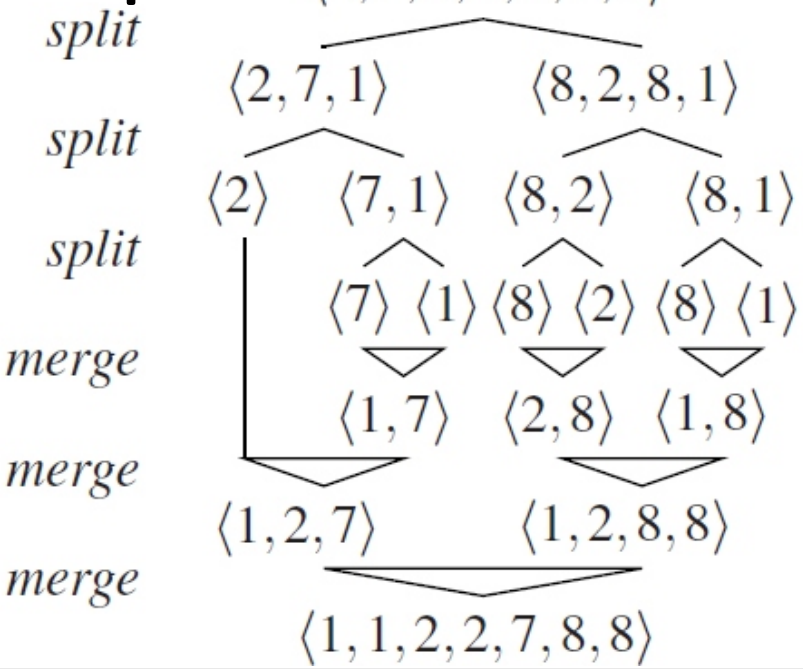
# 5.2 Mergesort

*Teile-und-Herrsche* Algorithmus

Die unsortierte Folge wird in zwei etwa gleich große Teile geteilt.
Die beiden Teile werden rekursiv sortiert.
Danach werden die sortierten Teilfolgen zusammengefügt.

Zwei sortierte Folgen lassen sich effizient zusammen fügen: der insgesamt kleinste Eintrag ist der kleinste von einer der Folgen. Verschiebe diesen in die Ausgabe und iteriere.

**Beispiel:** $\langle 2,7,1,8,2,8,1\rangle$




| $a$ | $b$ | $c$ | Operation |
|---|---|---|---|
| $\langle 1,2,7\rangle$ | $\langle 1,2,8,8\rangle$ | $\langle\rangle$ | move $a$ |
| $\langle 2,7\rangle$ | $\langle 1,2,8,8\rangle$ | $\langle 1\rangle$ | move $b$ |
| $\langle 2,7\rangle$ | $\langle 2,8,8\rangle$ | $\langle 1,1\rangle$ | move $a$ |
| $\langle 7\rangle$ | $\langle 2,8,8\rangle$ | $\langle 1,1,2\rangle$ | move $b$ |
| $\langle 7\rangle$ | $\langle 8,8\rangle$ | $\langle 1,1,2,2\rangle$ | move $a$ |
| $\langle\rangle$ | $\langle 8,8\rangle$ | $\langle 1,1,2,2,7\rangle$ | concat $b$ |
| $\langle\rangle$ | $\langle\rangle$ | $\langle 1,1,2,2,7,8,8\rangle$ | |

# 5.2 Mergesort

RUB

*Teile-und-Herrsche* Algorithmus

Die unsortierte Folge wird in zwei etwa gleich große Teile geteilt.
Die beiden Teile werden rekursiv sortiert.
Danach werden die sortierten Teilfolgen zusammengefügt.

Zwei sortierte Folgen lassen sich effizient zusammen fügen: der insgesamt kleinste Eintrag ist der kleinste von einer der Folgen. Verschiebe diesen in die Ausgabe und iteriere.

**Laufzeit:** Ein merge kostet $O(n)$, denn jede Iteration (bis auf die letzte) kostet 1 Vergleich. Für die Laufzeit von mergeSort bekommen wir die Rekurrenz:

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$$

$$\Rightarrow \quad T(n) = O(n \log n)$$

**Korrektheit:** folgt induktiv aus der Korrketheit des merge.
Korrektheit von merge folgt aus Invariante:
Folgen $a, b, c$ sind jeweils aufsteigend sortiert und alle Elemente in $c$ sind kleiner gleich allen in $a, b$.

# 5.3 Eine untere Schranke

RUB

Algorithmen liefern obere Schranken.
Wir wissen: Sortieren von $n$ Objekten in $O(n \log n)$.
Geht es besser, z.B. $O(n)$?

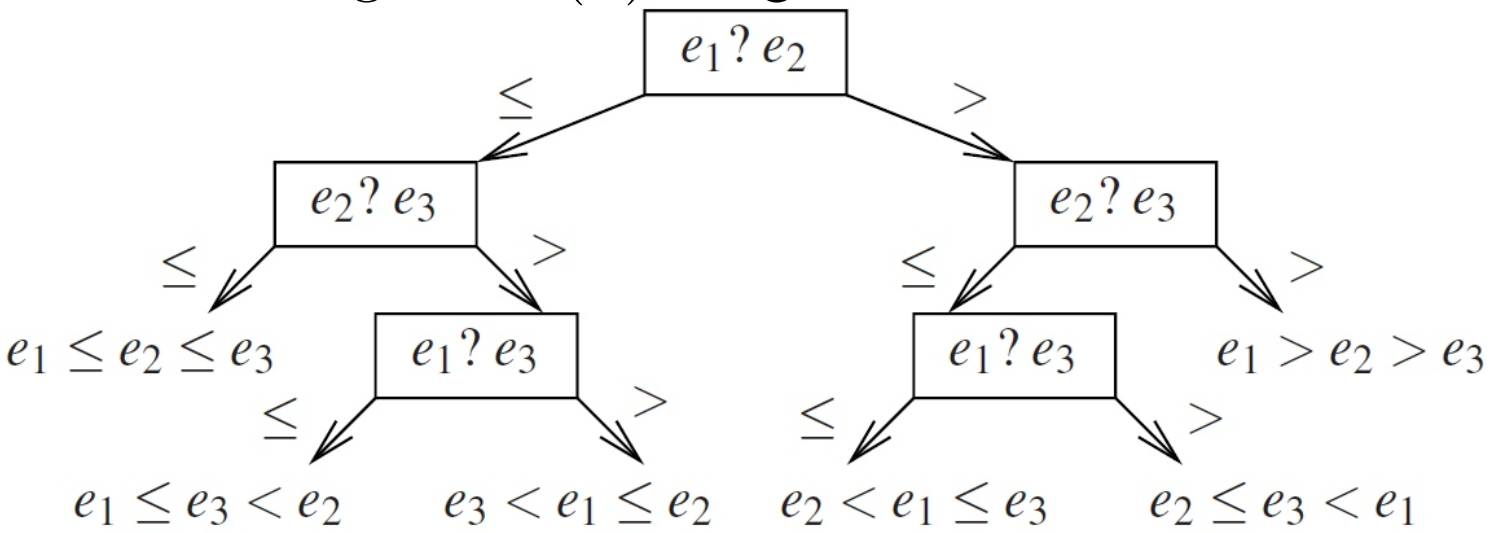
- nein, für vergleichsbasierte Algorithmen
- ja, für nicht vergleichsbasierte Algorithmen

# 5.3 Eine untere Schranke

Ein *vergleichsbasierter Algorithmus* darf nur vergleichen, verschieben und kopieren. Am Ende liefert er eine sortierte Permutation der Eingabe.
Deterministische vergleichsbasierte Algorithmen kann man als *Vergleichsbäume* darstellen.
Innere Knoten – Vergleiche, Blätter – Ausgabe, Wege – Berechnungen

Unter der Annahme, dass alle Schlüssel verschieden sind,
ist der Vergleichsbaum ein binärer Baum mit $\geq n!$ Blätter
$\Rightarrow$ Höhe des Baumes $\geq \log n! \geq \log((\frac{n}{e})^n) = n(\log n - \log e)$

**Satz 5.4** Jeder vergleichsbasierte Sortieralgorithmus benötigt im schlechtesten Fall $n \log n - O(n)$ Vergleiche.



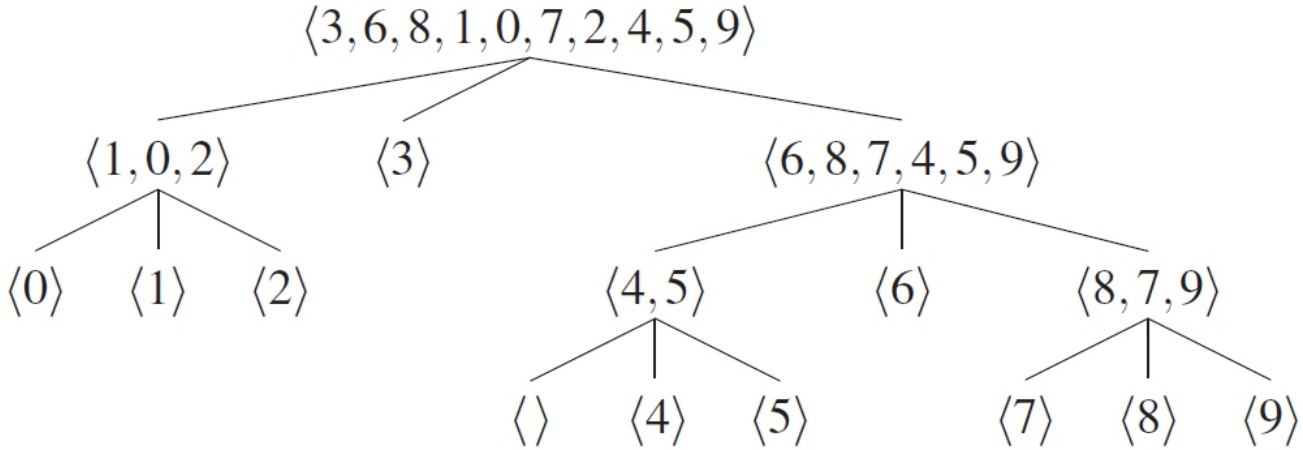
# 5.4 Quicksort

ebenfalls *Teile-und-Herrsche Algorithmus*
wähle Pivot-Element $p$, teile auf in $<, =, > p$, sortiere Teile rekursiv, und konkateniere diese.

**Problem:** Wie findet man einen mittleren Wert zum Aufteilen?
**Eine Lösung:** Pivotelement zufällig wählen.

**quickSort**(S)
if $|S| \leq 1$ then return $S$
choose $p \in S$ at random
$a := \langle e \in S : e < p \rangle$
$b := \langle e \in S : e = p \rangle$
$c := \langle e \in S : e > p \rangle$
$a' :=$ quickSort$(a)$
$c' :=$ quickSort$(c)$
return concat$(a', b, c')$

**Beispiel:**



# 5.4 Quicksort

RUB

ebenfalls *Teile-und-Herrsche Algorithmus*
wähle Pivot-Element $p$, teile auf in $<, =, > p$, sortiere Teile rekursiv, und konkateniere diese.

**Analyse:**

- im schlechtesten Fall:
  $C(n) \leq n-1+\max\{C(k)+C(k') : 0 \leq k \leq n-q, 0 \leq k' < n-k\}$
  $= \Theta(n^2)$
- im besten Fall:
  $C(n) \leq n - 1 + 2T(n/2) = \Theta(n \log n)$
- und im erwarteten Fall?

**Satz 5.6** Die erwartete Anzahl $C(n)$ von Vergleichen, die Quicksort auf Eingaben mit $n$ Einträgen ausführt, erfüllt $C(n) = O(n \log n)$.

**Beweis:** OBdA alle Einträge verschieden.
Sei $s = \langle e_1, \ldots, e_n \rangle$ die Eingabe und $s' = \langle e'_1, \ldots, e'_n \rangle$ die Ausgabe.
Zwei Einträge $e'_i, e'_j$ werden max. 1 mal verglichen, wenn einer von beiden Pivot, und beide noch in gleicher Teilfolge.
Definiere Indikator-Zufallsvariablen $X_{ij}$: werden $e'_i, e'_j$ verglichen?
Dann gilt

$$C(n) = \sum_i \sum_j X_{ij} = \sum_i \sum_j E[X_{ij}] = \sum_i \sum_j P[X_{ij} = 1]$$
$$= \sum_i \sum_j \frac{2}{j-i+1} = O(n \log n)$$

# In-Place Sortieren

RUB

Ein Sortieralgorithmus heißt *in-place*, wenn die Ausgabe direkt in das Eingabearray geschrieben wird, ohne weiteren Speicherplatz zu benutzen.

- Selection und Insertion Sort sind in-place
- Merge Sort ist nicht in-place
- Quicksort kann in-place implementiert werden

# In-Place Sortieren

RUB

## Quicksort in-place auf einem Array:

```
Procedure qSort(a : Array of Element; ℓ,r : ℕ)          // Sortiere das Teilarray a[ℓ..r].
  while r − ℓ + 1 > n0 do                               // Benutze Teile-und-Herrsche.
    j := pickPivotPos(a,ℓ,r)                            // Wähle Pivotelement
    swap(a[ℓ],a[j])                                     // und schaffe es an die erste Stelle.
    p := a[ℓ]                                           // p ist jetzt das Pivotelement.
    i := ℓ; j := r
    repeat                                              // a: | ℓ   i→  ←j   r |
      while a[i] < p do i++                             // Überspringe Einträge,
      while a[j] > p do j--                             // die schon im richtigen Teilarray stehen.
      if i ≤ j then                                     // Wenn die Partitionierung noch nicht fertig ist,
        swap(a[i],a[j]); i++; j--                       // (*) vertausche falsch positionierte Einträge.
    until i > j                                         // Partitionierung ist fertig.
    if i < (ℓ+r)/2 then qSort(a,ℓ,j); ℓ := i            // Rekursiver Aufruf für kleineres
    else                qSort(a,i,r); r := j            // der beiden Teilarrays.
  endwhile
  insertionSort(a[ℓ..r])                                // schneller für kleine r − ℓ
```

# In-Place Sortieren

## **Beispiel** für qSort

| $i \rightarrow$ | | | | | | | | $\leftarrow j$ | $j$ |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 8 | 1 | 0 | 7 | 2 | 4 | 5 | 9 |
| 2 | 6 | 8 | 1 | 0 | 7 | 3 | 4 | 5 | 9 |
| 2 | 0 | 8 | 1 | 6 | 7 | 3 | 4 | 5 | 9 |
| 2 | 0 | 1 | 8 | 6 | 7 | 3 | 4 | 5 | 9 |
| | | $j$ | $i$ | | | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 8 | 1 | 0 | 7 | 2 | 4 | 5 | 9 |
| 2 | 0 | 1 | 8 | 6 | 7 | 3 | 4 | 5 | 9 |
| 1 | 0 | 2 | 5 | 6 | 7 | 3 | 4 | 8 | 9 |
| 0 | 1 | | 4 | 3 | 7 | 6 | 5 | 8 | 9 |
| | | | 3 | 4 | 5 | 6 | 7 | | |
| | | | | | 5 | 6 | | | |

# 5.5 Das Auswahlproblem

RUB

**Gegeben:** eine Folge $s = \langle e_1, \ldots, e_n \rangle$ von $n$ Einträgen.
**Gesucht:** $k$-kleinster Eintrag (Rang $k$) in einer sortierten Permutation $s' = \langle e'_1, \ldots, e'_n \rangle$ von $s$

**Z.B.:** $k = 1$: Minimum, $k = n$: Maximum, $k = \lceil n/2 \rceil$: Median

**Frage:** Minimum und Maximum können einfach in $O(n)$ Zeit gefunden werden. Geht das auch für beliebige $k$?

# Quickselect

wie Quicksort, aber nur in einer Teilfolge wird weiter gesucht

**select**(S,k)

assert $|S| \geq k$
choose $p \in S$ at random
$a := \langle e \in S : e < p \rangle$
if $|a| \geq k$ then return select(a,k)
$b := \langle e \in S : e = p \rangle$
if $|a| + |b| \geq k$ then return $p$
$c := \langle e \in S : e > p \rangle$
return select$(c, k - |a| - |b|)$

**Beispiel:**

| $s$ | $k$ | $p$ | $a$ | $b$ | $c$ |
|---|---|---|---|---|---|
| $\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8 \rangle$ | 6 | 2 | $\langle 1 \rangle$ | $\langle 2 \rangle$ | $\langle 3,4,5,9,6,5,3,5,8 \rangle$ |
| $\langle 3,4,5,9,\mathbf{6},5,3,5,8 \rangle$ | 4 | 6 | $\langle 3,4,5,5,3,4 \rangle$ | $\langle 6 \rangle$ | $\langle 9,8 \rangle$ |
| $\langle 3,4,\mathbf{5},5,3,5 \rangle$ | 4 | 5 | $\langle 3,4,3 \rangle$ | $\langle 5,5,5 \rangle$ | $\langle \rangle$ |

# Quickselect

RUB

wie Quicksort, aber nur in einer Teilfolge wird weiter gesucht

**select**(S,k)

assert $|S| \geq k$

choose $p \in S$ at random

$a := \langle e \in S : e < p \rangle$

if $|a| \geq k$ then return select(a,k)

$b := \langle e \in S : e = p \rangle$

if $|a| + |b| \geq k$ then return $p$

$c := \langle e \in S : e > p \rangle$

return select$(c, k - |a| - |b|)$

**Analyse:**

- im schlechtesten Fall wie quickSort $O(n^2)$
- im besten Fall $O(1)$
- und im erwarteten Fall?

# Quickselect

RUB

**Satz 5.6** Die erwartete Rechenzeit von Quickselect auf einer Eingabe mit $n$ Einträgen $O(n)$.

**Beweis:** Sei $T(n)$ das Maximum der erwarteten Rechenzeit.
Im "guten Fall" sind beide Folgen kürzer als $2/3n$.
Dann ist $\gamma := P[\text{guter Fall}] \geq 1/3$.

Und es gilt $$T(n) \leq cn + \gamma T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + (1-\gamma)T(n)$$

Auflösen ergibt

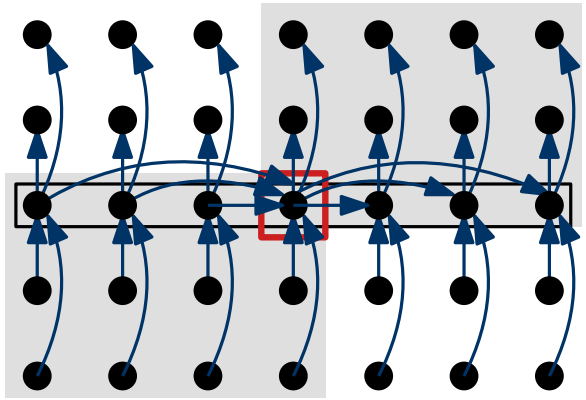
$$T(n) \leq \frac{cn}{\gamma} + T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) \leq 3cn + T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) \leq 3c\left(n + \frac{2n}{3} + \frac{4n}{9} + \ldots\right)$$
$$\leq 3cn \sum_{i \geq 0} \left(\frac{2}{3}\right)^i \leq 3cn \frac{1}{1-2/3} = 9cn\ .$$

# Deterministic Selection

**Idee:** finde ein gutes Pivotelement rekursiv.
**dselect(s,k):** wie select, aber verwende median-of-medians als Pivotelement

**choosePivot(s)**
partition S in $\lfloor n/5 \rfloor$ groups of 5 elements
medians: $=$ sequence of median in each group
return (dselect(medians,$\lceil \lfloor n/5 \rfloor /2 \rceil$))

**Laufzeit:** $T(n) \leq T(n/5) + T(3/4n) + O(n) = O(n)$



[CLRS Abschnitt 9.3]