Weitere Beispiele Effizienter Algorithmen

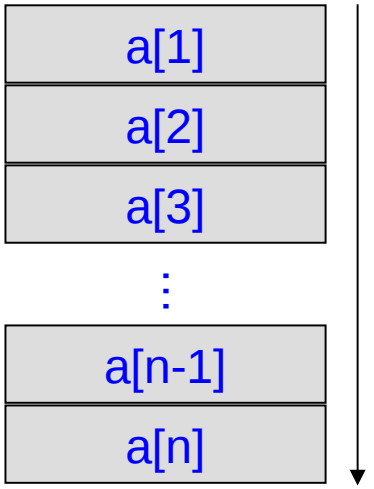
Sequentielle Suche

Gegeben: Array a[1..n]

Suche in a nach Element x

Ohne weitere Zusatzinformationen:

Sequentielle Suche

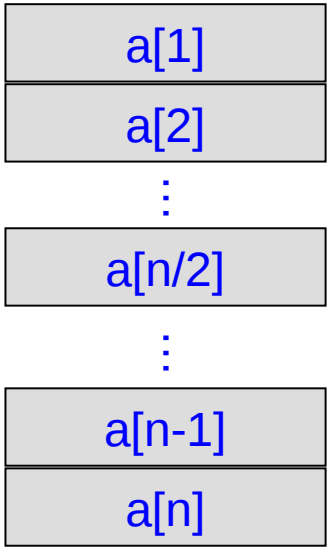


Laufzeit:

n Schritte im worst-case

Binäre Suche

Angenommen: Array a ist sortiert



Vergleiche x mit a[n/2]

$x \leq a[n/2] \Rightarrow$ x liegt in a[1..n/2]

$x \geq a[n/2] \Rightarrow$ x liegt in a[n/2..n]

Binäre Suche

Rekursive Programmierung:

```
function binarySearch (a,x,l,r)
  if (r = l)
    if (x = a[l])
      return l
    else
      return not found
  mid = floor((l+r)/2)
  if (x ≤ a[mid])
    binarySearch(a, x, l, mid)
  else
    binarySearch(a, x, mid+1, r)
```

Laufzeit Binäre Suche

Satz: Für die maximale Anzahl Vergleiche für die binäre Suche in $a[1..n]$ gilt

$$B_n = B_{\lceil n/2 \rceil} + 1 \text{ für } n \geq 2, \text{ und } B_1 = 1$$

Die Lösung dieser Rekursion lautet

$$B_n = \lceil \log_2 n \rceil + 1$$

Beweis:

- Beide Hälften von $a[1..n]$ sind höchstens $\lceil n/2 \rceil$ gross
- Es gilt die Monotonie: $B_m \leq B_n \; \forall m \leq n$
- Induktiv gilt $B_{2k} = k + 1 = \lceil \log_2(2^k) \rceil + 1$, und zusammen mit der Monotonie folgt die Behauptung.

Anzahl Vergleiche bei Binärer Suche

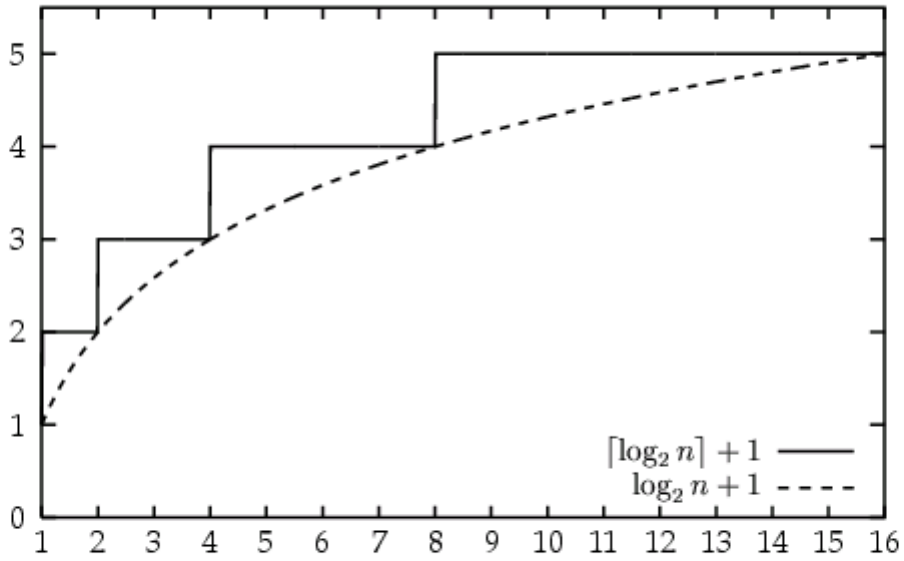


Abbildung 2.1: Anzahl Vergleiche $B_n = \lceil \log_2 n \rceil + 1$ bei einer binären Suche.

Für asymptotische Analyse:

Laufzeit von Binärer Suche ist O(log n)

2.2 Sortieralgorithmen

Gegeben: Zahlen $a_1,\ldots,a_n$;

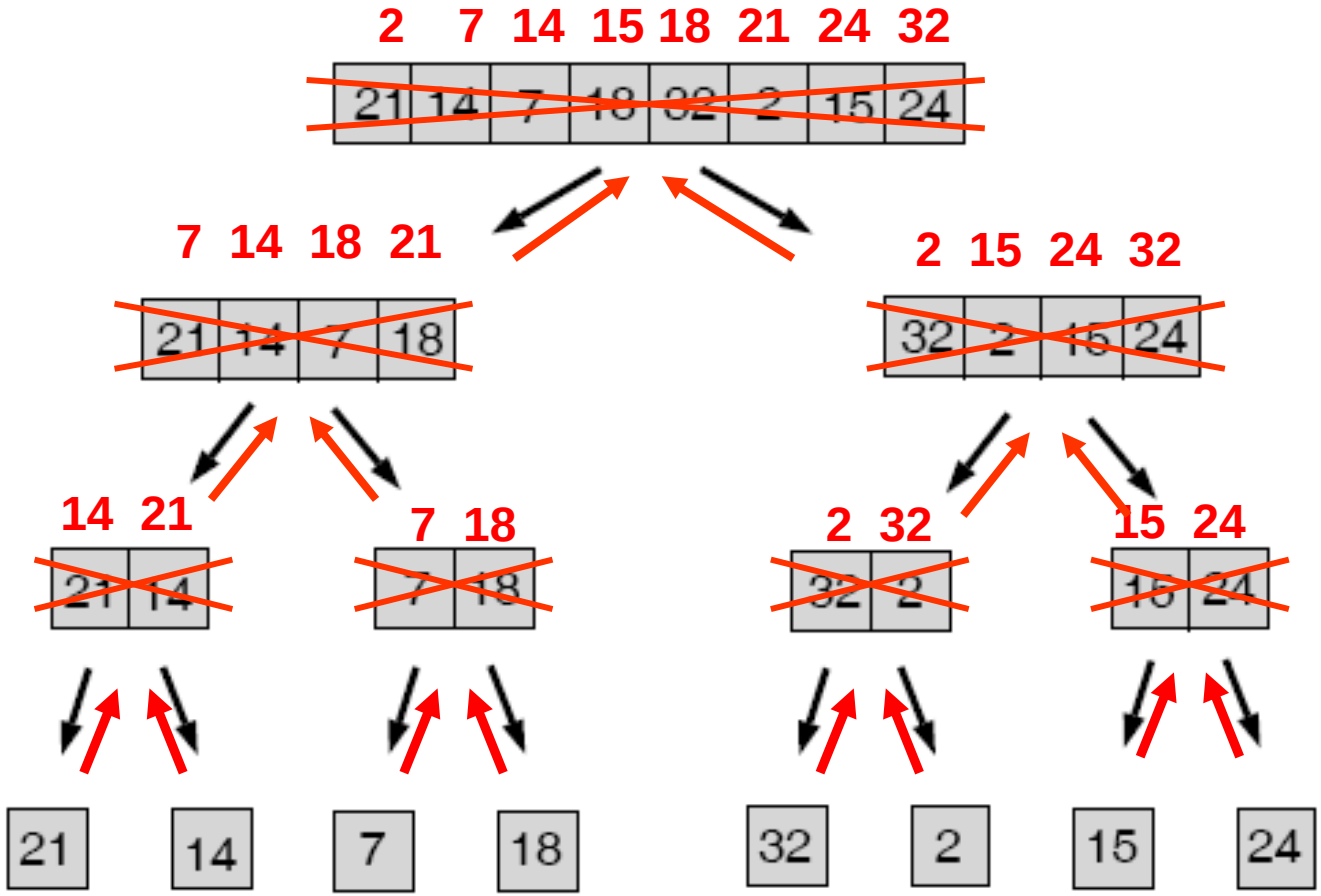
Aufgabe: Sortiere sie!

Bemerkungen:

- Wir schätzen statt der Laufzeit meist die Anzahl der benötigten Vergleiche zwischen Elementen der Eingabe ab.

- *Wichtig:* Wir zählen hier wirklich nur die Anzahl Vergleiche zwischen Elementen der Eingabe (so genannte Schlüsselvergleiche) und nicht Vergleiche der Form „$i \leq n$“, die bspw. benötigt werden, um festzustellen, ob eine Laufvariable die vorgegebene Grösse eines Feldes überschreitet.

MergeSort - Beispiel



MergeSort - Analyse

C_n := Anzahl (Schlüssel-)Vergleiche, die MergeSort höchstens durchführt, wenn ein Feld der Grösse n sortiert wird

$n = 2^k$:

$$C_1 = 0$$

$$C_{2k} = 2 \cdot C_{2k-1} + 2 \cdot 2^{k-1} - 1$$

MergeSort – Analyse (2)

$n = 2^k$: $C_1 = 0$

$$C_{2k} = 2 \cdot C_{2k-1} + 2 \cdot 2^{k-1} - 1$$
$$\leq 2 \cdot C_{2k-1} + 2^k$$
$$\leq 2 \cdot (2 \cdot C_{2k-2} + 2^{k-1}) + 2^k$$
$$\leq \ldots \leq 2^k \cdot C_1 + k \cdot 2^k = k \cdot 2^k$$

n beliebig:

$$C_n \leq C_{2\lceil \log_2 n \rceil} \leq \lceil \log_2 n \rceil \cdot 2^{\lceil \log_2 n \rceil} \leq 2n \cdot (\log_2 n + 1)$$

MergeSort – Analyse (3)

Satz:

Um ein Feld der Grösse n zu sortieren genügen $2n \cdot (\log_2 n + 1)$ Vergleiche bzw. Laufzeit $O(n \cdot \log_2 n) = O(n \log(n))$.

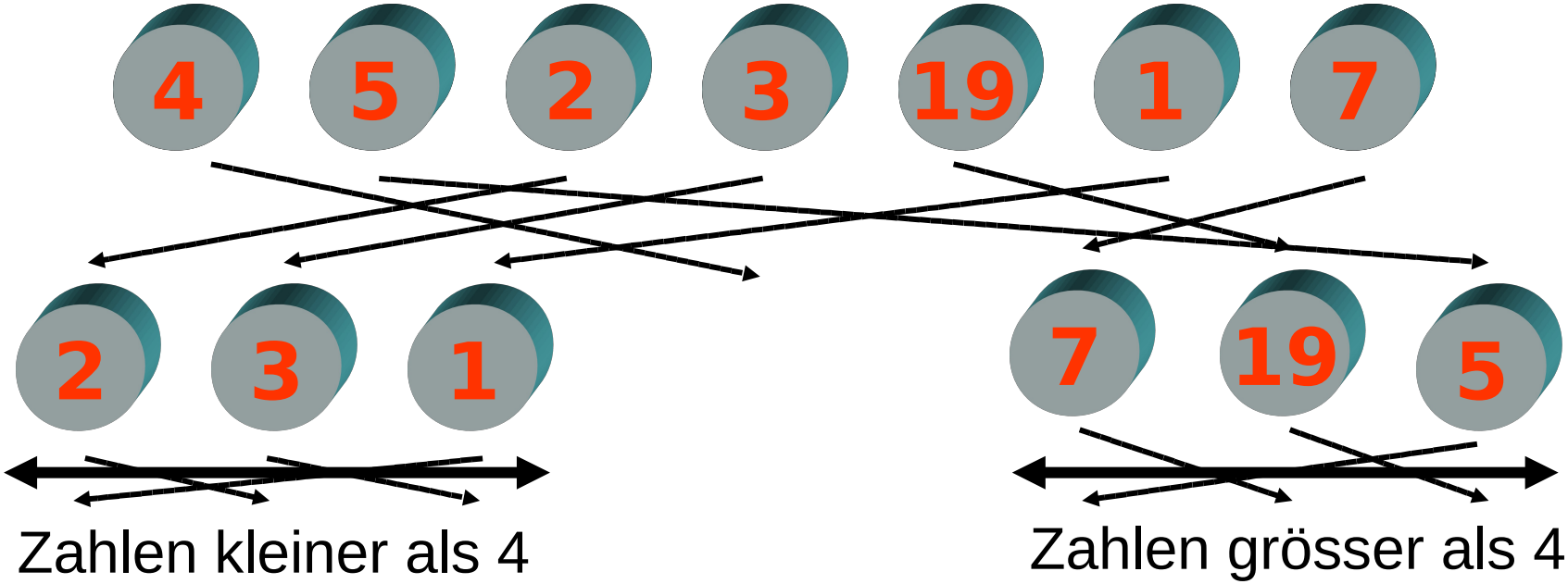
Bemerkung:

Genauere Analyse zeigt, dass sogar

$$C_n \leq n \cdot \lfloor \log_2 n \rfloor + 2\,n$$

gilt.

QuickSort - Beispiel



- Wähle erstes Element und teile übrige Elemente in „kleiner“ und „grösser“
- Wende analoges Verfahren auf die

- Test auf einem handelsüblichen Rechner: 500.000 Zahlen werden in 0.4 sec sortiert

QuickSort

- Wir testen unseren Algorithmus an einer realen Aufgabe.
- Dazu bitten wir einen Mathematiker  uns 500.000 Zahlen in einer beliebigen, von ihm/ihr gewählten Reihenfolge zu liefern.
- M. liefert uns die Zahlen

  1, 2, 3, 4, 5, … , 499.999, 500.000
- QuickSort rechnet … und rechnet …

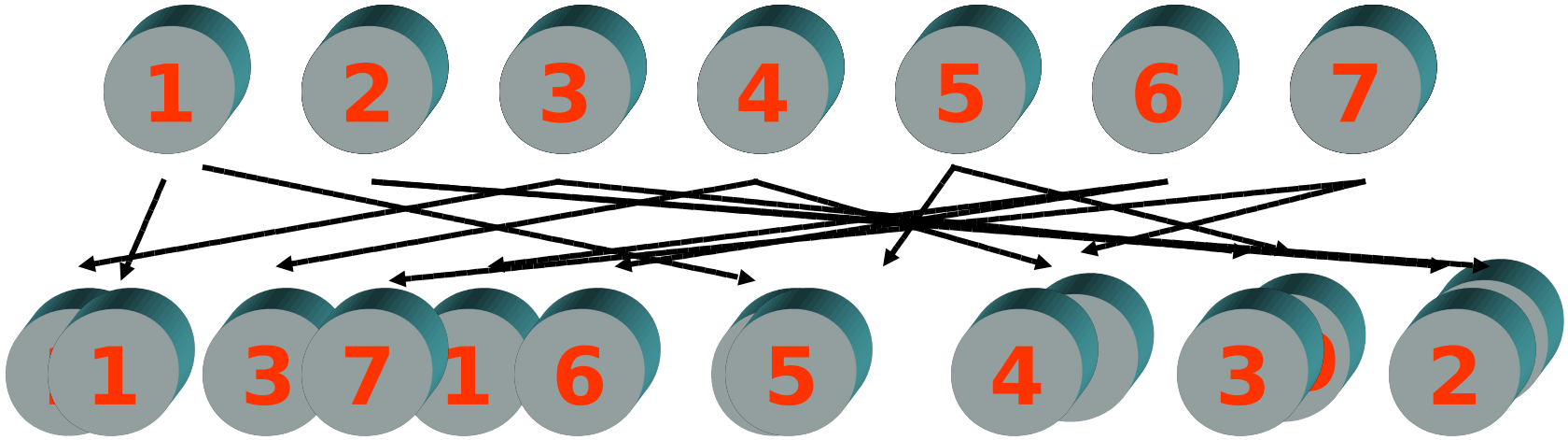
  und rechnet … und rechnet …

  und rechnet … und rechnet …

  und stürzt ab.

Idee des QuickSort-Algorithmus:

Die zu sortierenden Elemente werden in jedem Schritt in zwei etwa gleich große Mengen aufgeteilt: **für Testeingabe gilt aber**



QuickSort - Analyse

C_n := Anzahl (Schlüssel-)Vergleiche, die QuickSort höchstens durchführt, wenn ein Feld der Grösse n sortiert wird

$$C_n = \max_{0 \le k < n} (n-1 + C_k + C_{n-k-1})$$

Und somit:

$$C_n \ge n-1 + C_{n-1} \ge (n-1) + (n-2) + C_{n-2}$$
$$\ge \ldots \ge (n-1) + (n-2) + \ldots + 1 + C_1$$
$$= \tfrac{1}{2}\, n(n-1)$$

QuickSort – Analyse (2)

Satz:

Um ein Feld der Grösse n zu sortieren benötigt QuickSort im schlimmsten Fall $\frac{1}{2}\, n(n-1)$ Vergleiche bzw. Laufzeit $\Theta(n^2)$.

Bemerkung:

- Trotzdem gilt QuickSort als ein (in der Praxis und leicht abgewandelt) als sehr effizientes Sortierverfahren.
- Mathematisch exakt kann man dies begründen indem man die erwartete Laufzeit betrachtet, wobei der Erwartungswert über alle n! Reihenfolgen der Zahlen 1, …,n gebildet wird.

  Man erhält: erwartete Laufzeit = $O(n \log n)$

Sortieren – Untere Schranken

Satz:

Jeder Algorithmus, der n Elemente mit Hilfe von „if“-Abfragen sortiert, benötigt mindestens $\Theta(n \log n)$ Vergleiche.

Beweisidee:

Gegenspieler-Beweis …

BucketSort - Beispiel

2 1 3 4 1 4 2 3 2 1

1 1 1 2 2 2 3 3 4 4

BucketSort

Algorithmus 2.3 BucketSort

Eingabe: Feld $a[1..n]$ mit Elementen aus $\mathcal{U} = \{0, 1, \ldots, N-1\}$.
Ausgabe: Feld $a[1..n]$, wobei die Einträge aufsteigend sortiert sind.

```
for i = 0 to N − 1 do b[i] = 0;
for i = 1 to n do b[a[i]] = b[a[i]] + 1;
k := 0;
for i = 0 to N − 1 do
    for j = 1 to b[i] do
        k:= k+1; a[k] = i;
```

Satz:

Eine Folge aus n Zahlen aus dem Bereich U = {0,1,…,N-1} lässt sich mit BucketSort in Zeit O(n+N) sortieren.

BucketSort - Verbesserungen

Satz:

Eine Folge aus n Zahlen aus dem Bereich $U = \{0,1,\ldots,N^k-1\}$ lässt sich mit BucketSort in Zeit $O(k \cdot (n+N))$ sortieren.

2.3 Medianbestimmung

Selektionsproblem:

Gegeben: Menge $S=\{a_1,\ldots,a_n\}$; $k \in N$

Aufgabe: Gebe das k-te kleinste Element aus.

Annahme: a_i paarweise verschieden

(allg. Fall geht ähnlich, aber in der Notation aufwendiger …)

Bemerkung:

Medianbestimmung: $k = \lceil n/2 \rceil$

1. Ansatz

- Sortiere die a_i`s.
- Gebe das k-te Element aus.

Korrektheit: ✓

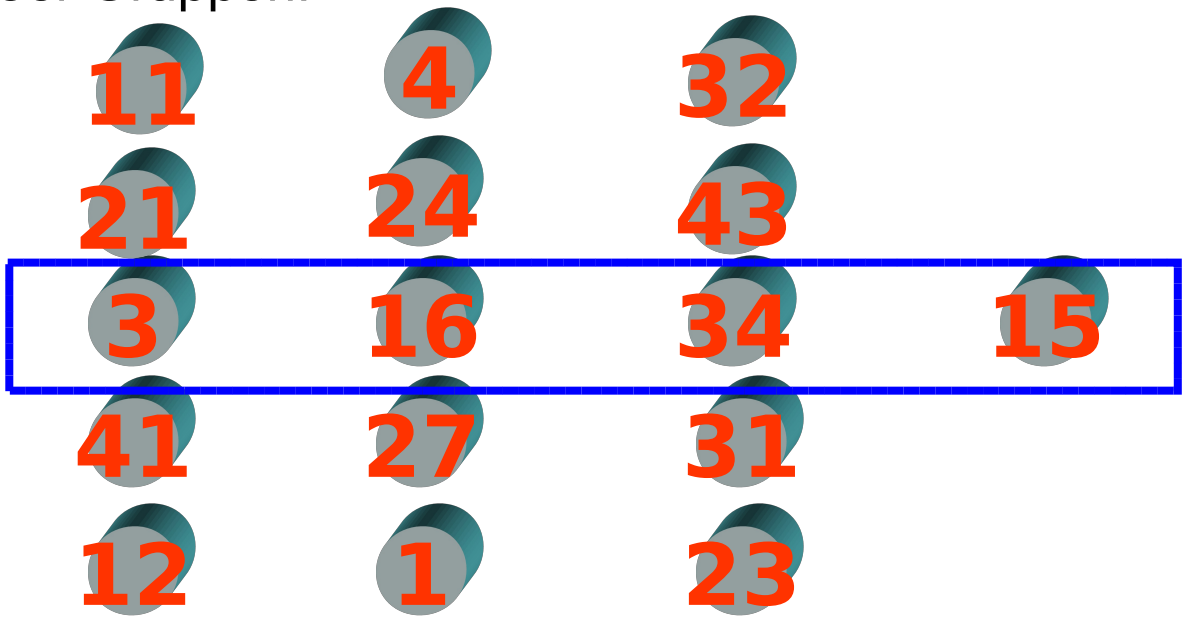
Laufzeit: O(n log n)

Ziel: O(n)

Select - Beispiel

11 21 3 41 12 4 24 16 27 1 32 43 34 31 23 15

5er Gruppen:



Stelle Median jeder Gruppe in die Mitte ….

Bestimme Median … rekursiv

→ 16

k=2

~~k=9~~

Partitioniere Elemente: ⟨ < 16 ⟩ , 16 , ⟨ > 16 ⟩

~~11 3 12 4 1 15~~ 16 21 41 24 27 32 43 34 31 23

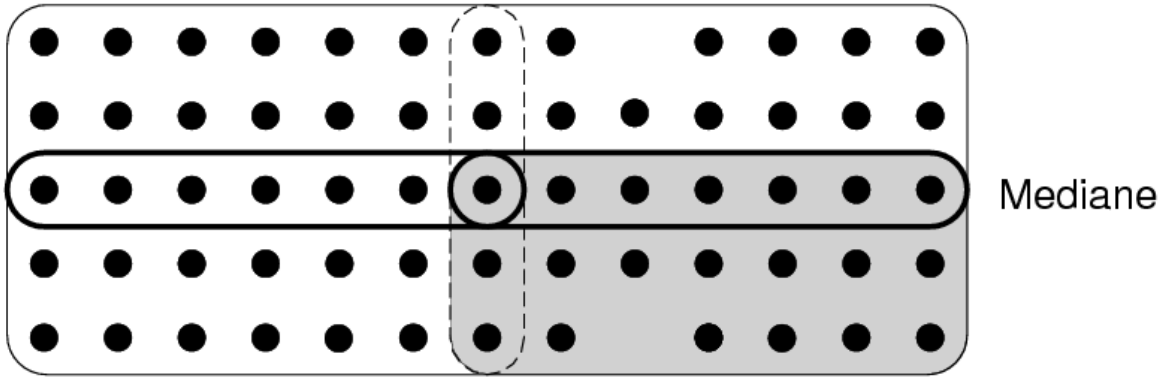
Select

Algorithmus 2.4 SELECT

Eingabe: Menge $S = \{a_1, \ldots, a_n\}$, Index k mit $1 \leq k \leq n$.
Ausgabe: Das k-te kleinste Element in S.

func SELECT(S, k)
if $(n \leq 15)$ **then**
    Sortiere die Menge S mit MERGESORT und gebe das k-te kleinste Element aus;
else
    Teile Elemente aus S in $\lceil \frac{n}{5} \rceil$ Gruppen auf, $\lfloor \frac{n}{5} \rfloor$ davon mit jeweils genau 5 Elementen;
    Bestimme den Median jeder Gruppe;
    Bestimme rekursiv den Median dieser Mediane, nenne ihn p;
    Partitioniere $S \setminus \{p\}$:
        $S_1 := \{s \in S \mid s < p\}$,
        $S_2 := \{s \in S \mid s > p\}$,
    if $|S_1| = k - 1$ **then**
        return p
    elseif $k \leq |S_1|$ **then**
        SELECT(S_1, k)
    else
        SELECT$(S_2, k - 1 - |S_1|)$;

Anschaulich



- 5-er Gruppen und ihre Mediane (im Bild aufsteigend angeordnet)
- In der Mitte: Der Median der Mediane (rekursiv bestimmt)
- Bestimme den Rang m des Median der Mediane
- Vergleiche diesen mit dem Rang k des gesuchten Elements
- Dann: mind. 3/10 aller Elemente können ausgeschlossen werden
- Warum? Angenommen $m \geq k$. Dann haben die Elemente im grau schraffierten Bereich ebenfalls Rang grösser k.
- Korrektheit: Aus Definition des Algorithmus klar… ☺

Laufzeit

Satz

Select benötigt für die Bestimmung des k-kleinsten Elements einer n-elementigen Menge (n ≥ 3), höchstens 22(n – 2) Vergleiche.

Beweis

S_n := max. Anzahl Vergleiche bei einer n-elementigen Menge.

$$S_n \leq 6 \cdot \lceil \frac{n}{5} \rceil + S_{\lceil \frac{n}{5} \rceil} + n - 1 + S_{\lceil \frac{7n}{10} + 2 \rceil}$$

- Aus 5er-Gruppen den Median bestimmen: 6 Vergleiche pro Gruppe
- Algorithmus rekursiv zur Bestimmung des Median der Mediane aufrufen
- Partitionierung der Elemente nach dem Median der Mediane
- Algorithmus rekursiv auf Menge mit $\lfloor \frac{7n}{10} + 2 \rfloor$ Elementen anwenden