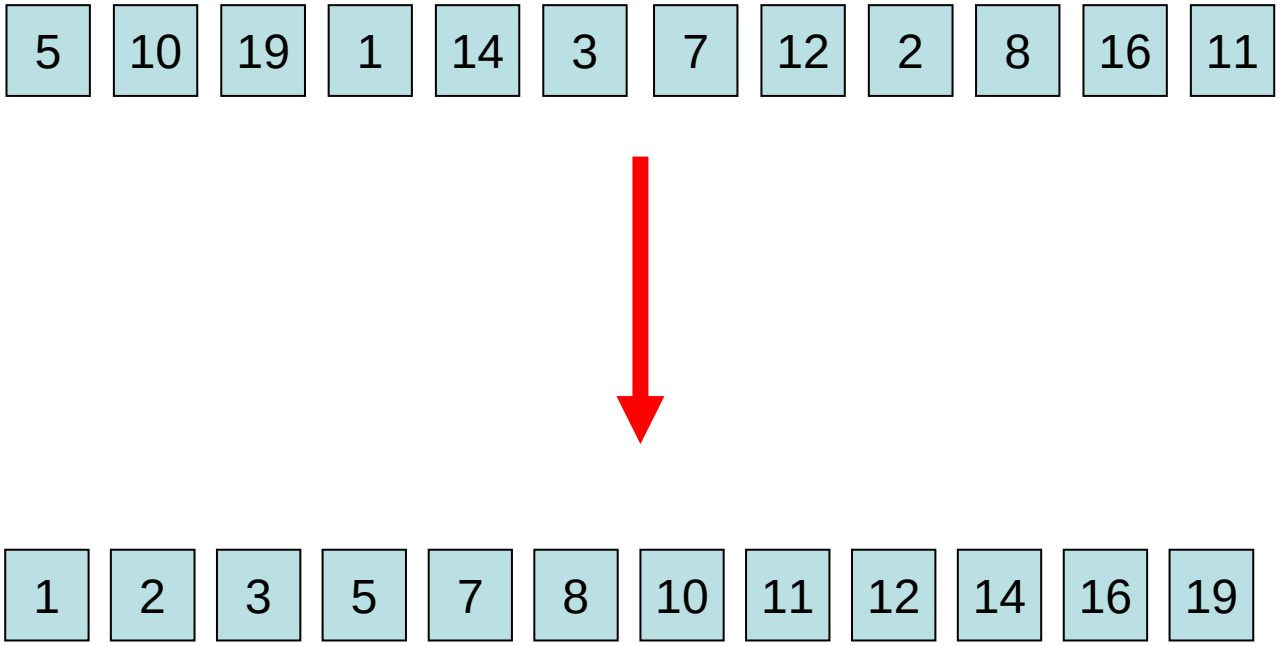
# Effiziente Algorithmen und Datenstrukturen I

# Kapitel 5: Sortieren und Selektieren

Christian Scheideler

WS 2008

# Sortierproblem



# Sortierproblem

- Eingabe: Sequenz $s = \langle e_1,\ldots,e_n \rangle$ mit Ordnung $\leqslant$ auf den Schlüsseln $key(e_i)$

  (Beispiel: | 5 | 10 | 19 | 1 | 14 | 3 | )

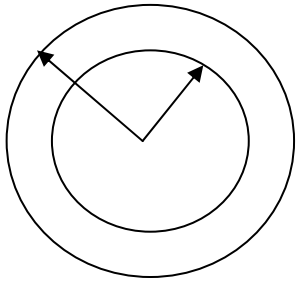
- Ausgabe: Sequenz $s' = \langle e'_1,\ldots,e'_n \rangle$, so dass $key(e'_i) \leqslant key(e'_{i+1})$ für alle $1 \leqslant i < n$

  und $s'$ eine Permutation von $s$ ist

  (Beispiel: | 1 | 3 | 5 | 10 | 14 | 19 | )

# 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.)

# Einfache Sortierverfahren

Selection Sort: nimm wiederholt das kleinste Element aus der Eingabesequenz, lösche es, und hänge es an das Ende der Ausgabesequenz an.

Beispiel: <4,7,1,1> | <> $\rightarrow$ <4,7,1> | <1>

$\rightarrow$ <4,7> | <1,1> $\rightarrow$ <7> | <1,1,4>

$\rightarrow$ <> | <1,1,4,7>

# Einfache Sortierverfahren

Selection Sort: nimm wiederholt das kleinste Element aus der Eingabesequenz, lösche es, und hänge es an das Ende der Ausgabesequenz an.

Zeitaufwand (worst case):

- Minimumsuche in Feld der Größe i: $\Theta(i)$
- Gesamtzeit: $\sum_{i=1..n} \Theta(i) = \Theta(n^2)$

# Selection Sort

```
Procedure SelectionSort(a: Array [1..n] of Element)
   for i:=1 to n-1 do
      // bewege min{a[i],…,a[n]} nach a[i]
      for j:=i+1 to n do
         if a[i]>a[j] then a[i] ↔ a[j]
```

Vorteil: sehr einfach zu implementieren, also gut für kurze Sequenzen

Nachteil: kann für lange Sequenzen lange dauern

# Einfache Sortierverfahren

Insertion Sort: nimm Element für Element aus der Eingabesequenz und füge es in der richtigen Position der Ausgabe-sequenz ein

Beispiel: <4,7,1,1> | <> $\rightarrow$ <7,1,1> | <4>

$\rightarrow$ <1,1> | <4,7> $\rightarrow$ <1> | <1,4,7>

$\rightarrow$ <> | <1,1,4,7>

# Einfache Sortierverfahren

Insertion Sort: nimm Element für Element aus der Eingabesequenz und füge es in der richtigen Position der Ausgabe-sequenz ein

Zeitaufwand (worst case):

- Einfügung an richtiger Stelle beim i-ten Element: $O(i)$ (wegen Verschiebungen)
- Gesamtaufwand: $\sum_{i=1..n} O(i) = O(n^2)$

# Insertion Sort

```
Procedure InsertionSort(a: Array [1..n] of Element)
   for i:=2 to n do
      // bewege a[i] zum richtigen Platz
      for j:=i-1 downto 1 do
         if a[j]>a[j+1] then a[j] ↔ a[j+1]
```

Vorteil: sehr einfach zu implementieren, also gut für kurze Sequenzen

Nachteil: kann für lange Sequenzen lange dauern

# Einfache Sortierverfahren

## Selection Sort:

- Mit binärem Heap zur Minimumbestimmung worst-case Laufzeit $O(n \log n)$ erreichbar $\rightarrow$ Heapsort

## Insertion Sort:

- Mit besserer Einfügestrategie (lass Lücken wie bei gewichtsbalancierten Bäumen) worst-case Laufzeit $O(n \log^2 n)$ erreichbar $\rightarrow$ Librarysort

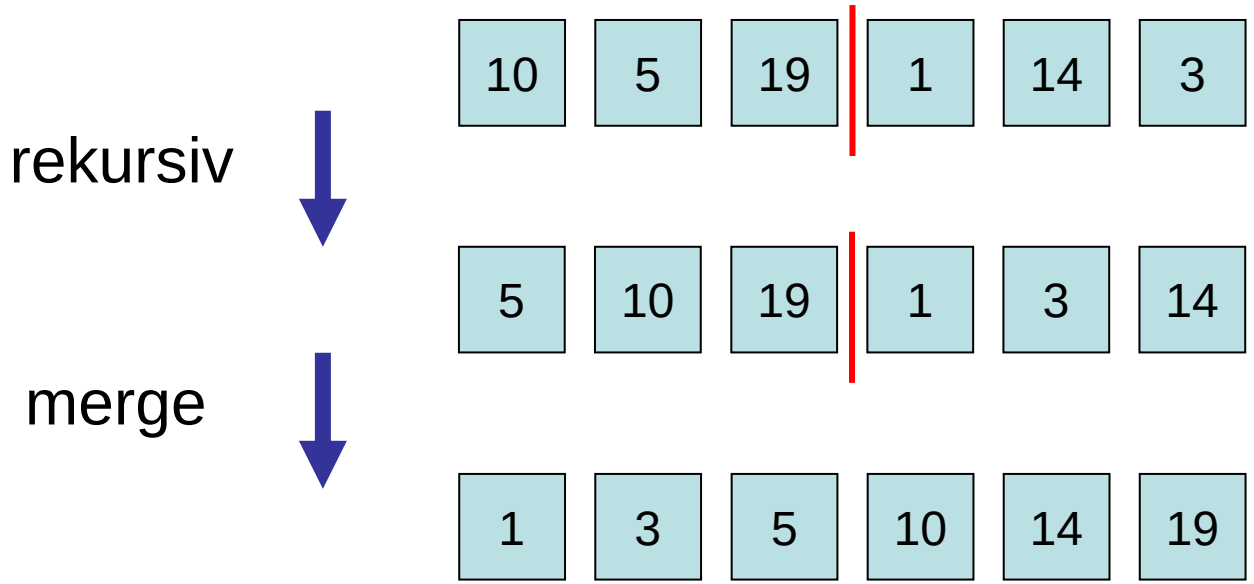
# Heapsort

```
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
```

# Mergesort

Theorem 5.1: Mergesort benötigt $O(n \log n)$ Zeit, um eine Folge der Länge $n$ zu sortieren.

Beweis:

- $T(n)$: Laufzeit bei Folgenlänge $n$
- $T(1) = \Theta(1)$ und
  $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$
- Übungsaufgabe: $T(n)=O(n \log n)$

# Untere Schranke

Ziel einer unteren Laufzeitschranke t(n):
Zeige, dass kein Algorithmus (aus einer gewissen Klasse) eine bessere Laufzeit als O(t(n)) im worst case haben kann.

Methodik: nutze strukturelle Eigenschaften des Problems

Beispiel: vergleichsbasiertes Sortieren

# Untere Schranke

Vergleichsbasiertes Sortieren: Wie oft muss jeder vergleichsbasierte Algo im worst case einen Vergleich $e<e'$ machen?

Frage: Gegeben sei beliebige Menge $S$ von $n$ Elementen. Wieviele Möglichkeiten gibt es, $S$ als Folge darzustellen?

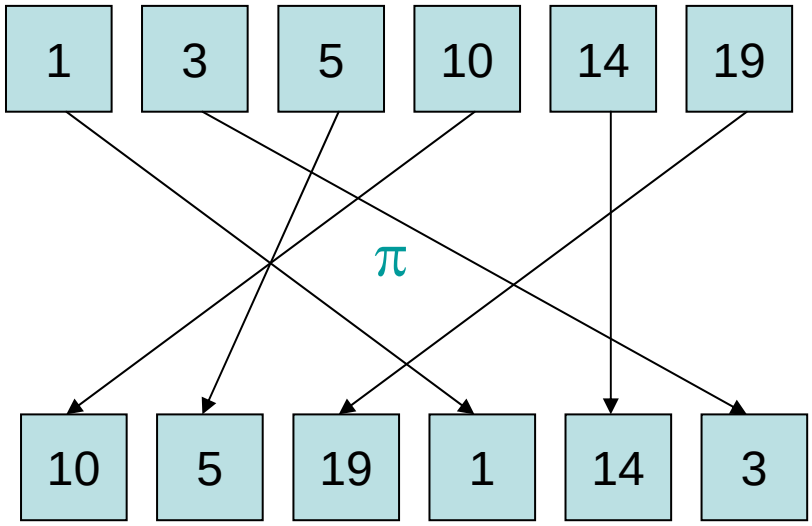
Antwort: $n! = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1$ viele

# Untere Schranke

## Permutation $\pi$ der Eingabefolge:

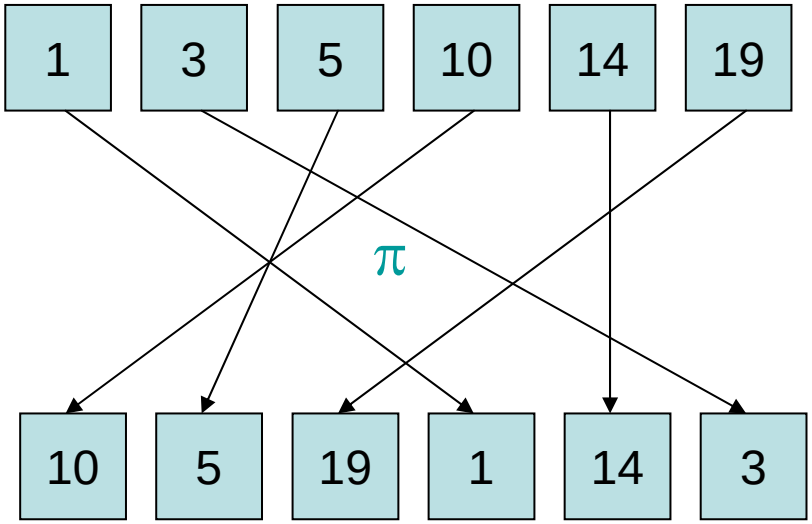
- Menge S:

- Eingabefolge:



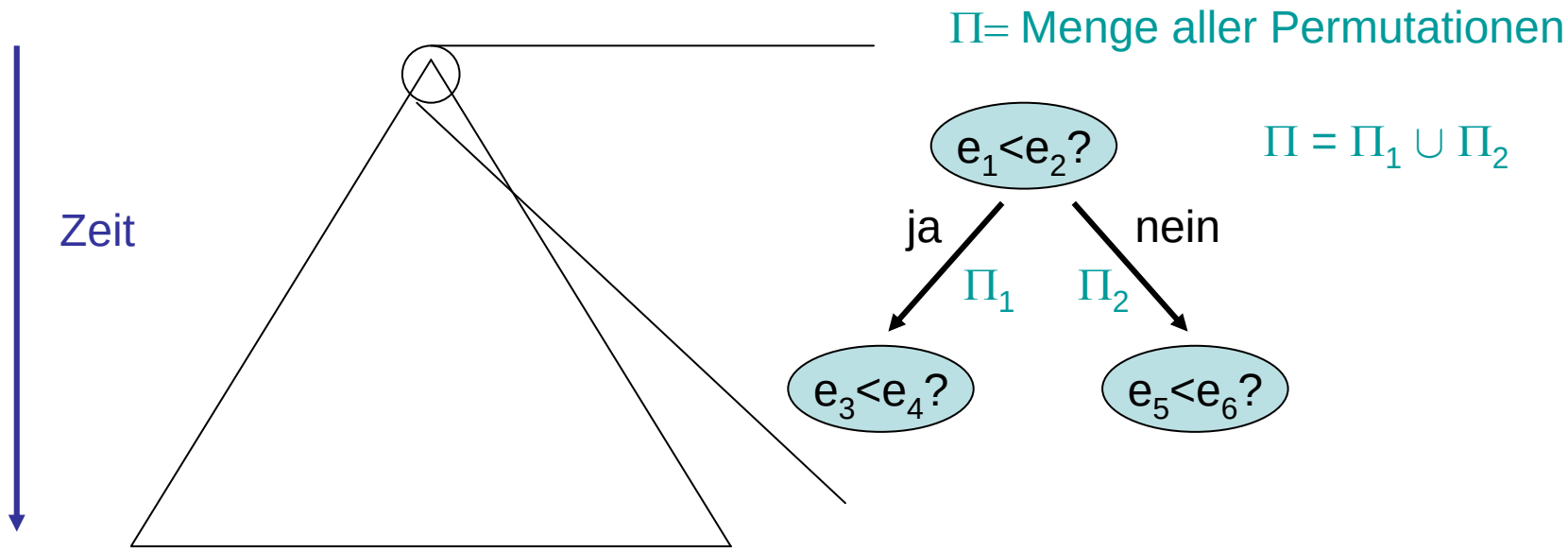
# Untere Schranke

Wenn der Algorithmus sortieren kann, kann er auch die Permutation $\pi$ ausgeben.



# Untere Schranke

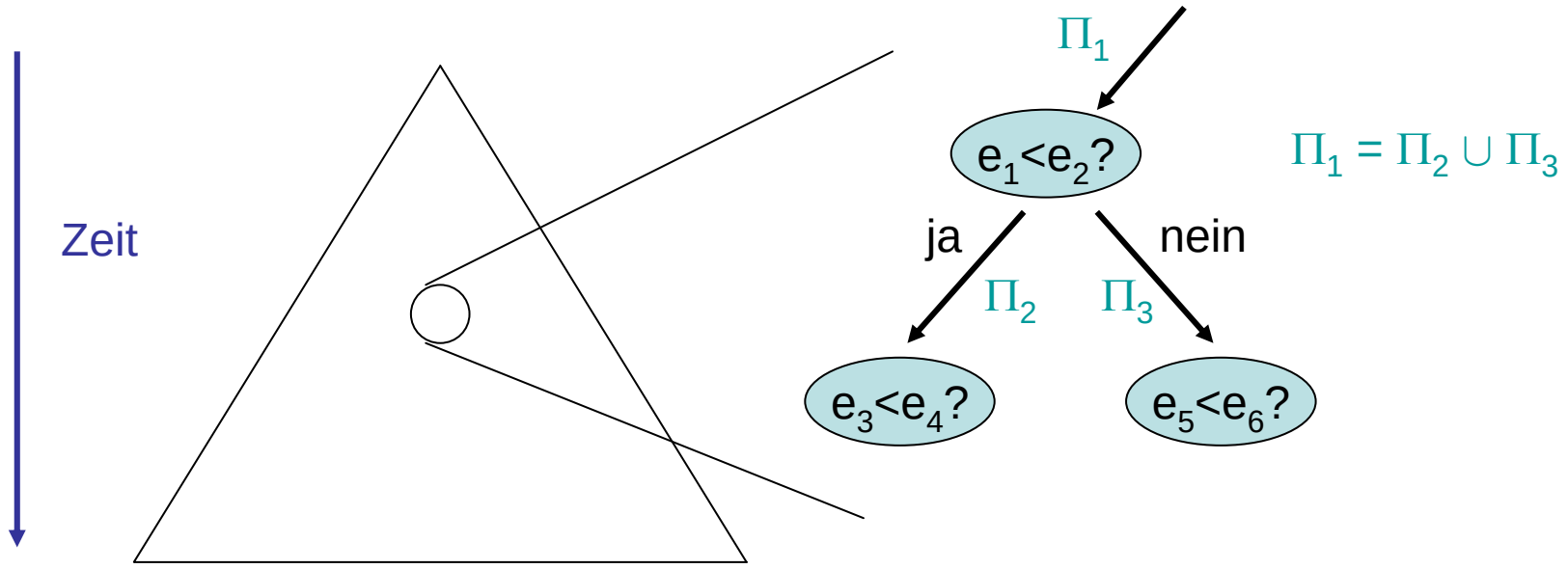
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:




$\Pi_j$ : Permutationsmenge, die Bedingungen bis dahin erfüllt

# Untere Schranke

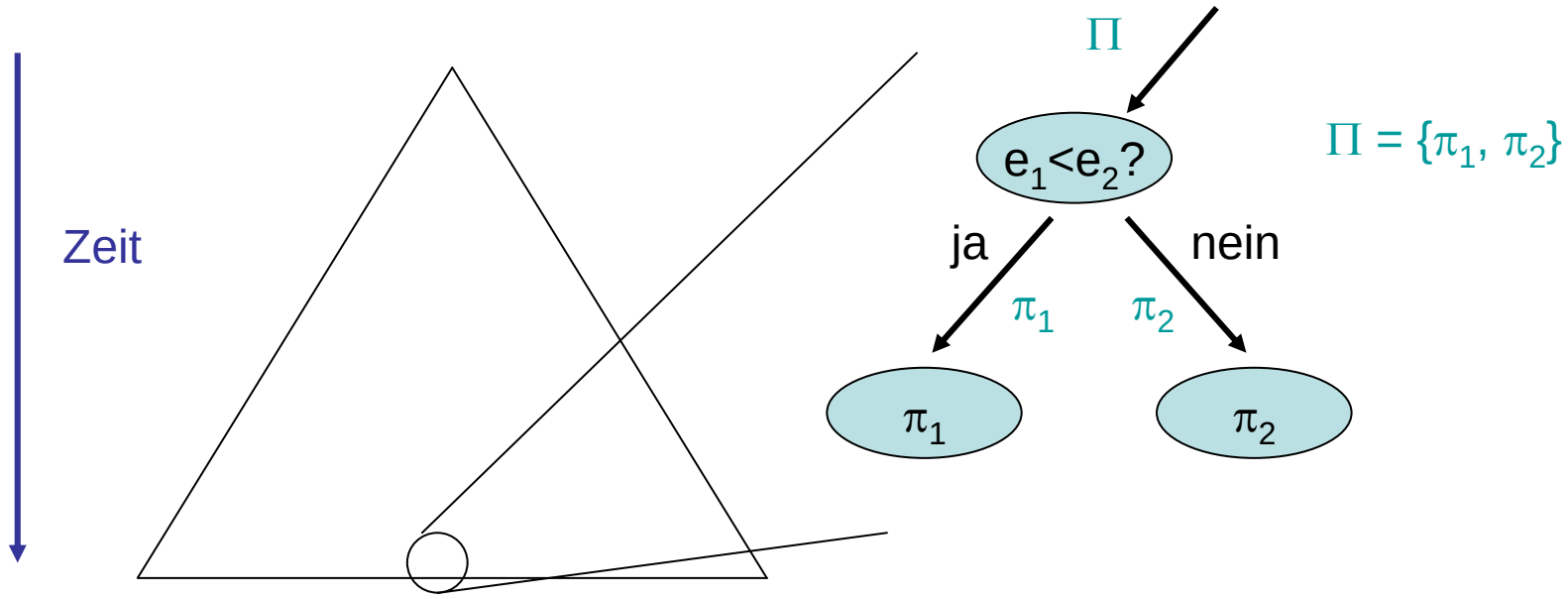
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:




$\Pi_j$ : Permutationsmenge, die Bedingungen bis dahin erfüllt

# Untere Schranke

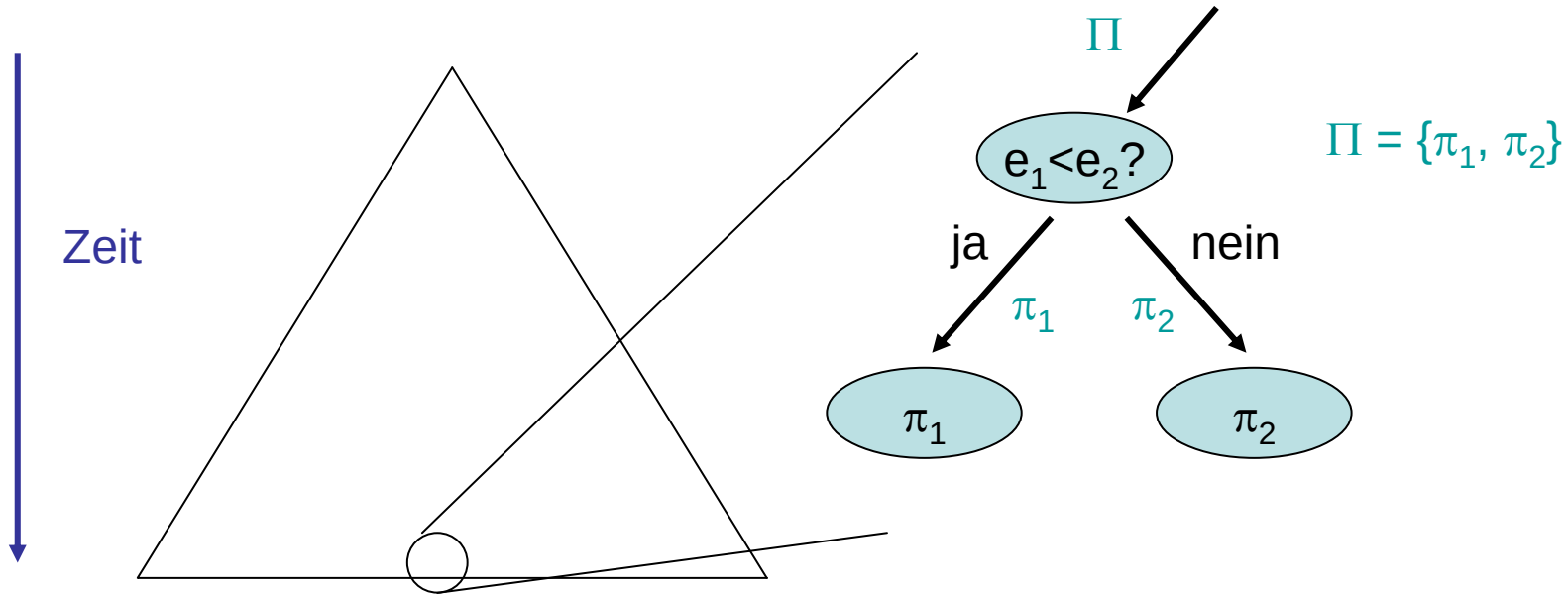
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:




$\pi_i$ : eindeutige Permutation der Eingabefolge

# Untere Schranke

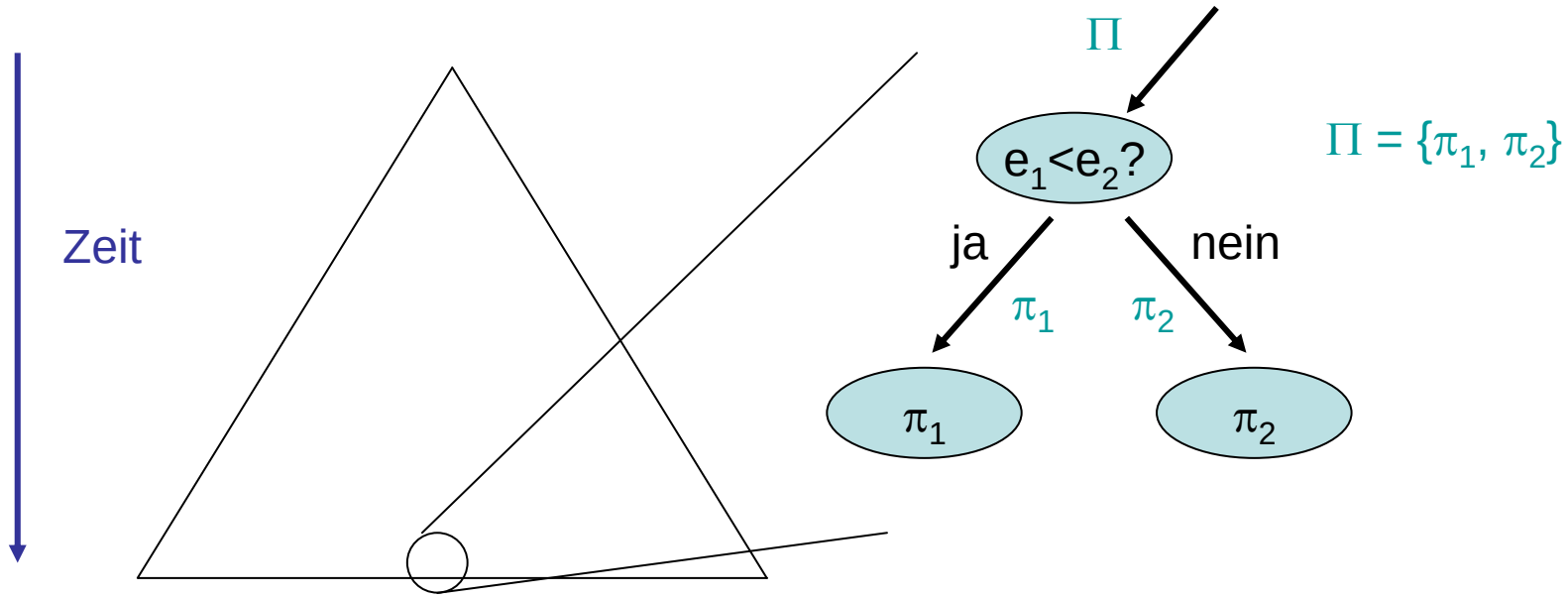
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:




Wieviele Blätter muss Entscheidungsbaum haben?

# Untere Schranke

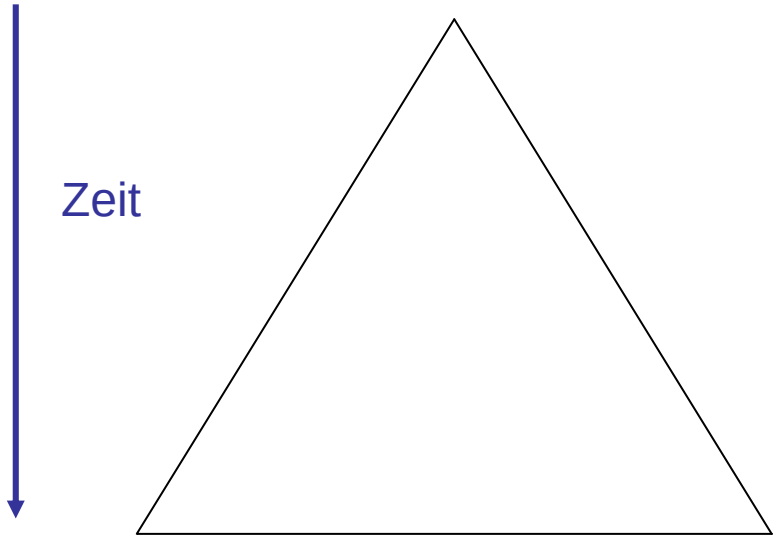
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:




Mindestens n! viele Blätter!

# Untere Schranke

Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:




Baum der Tiefe $T$:
Höchstens $2^T$ Blätter

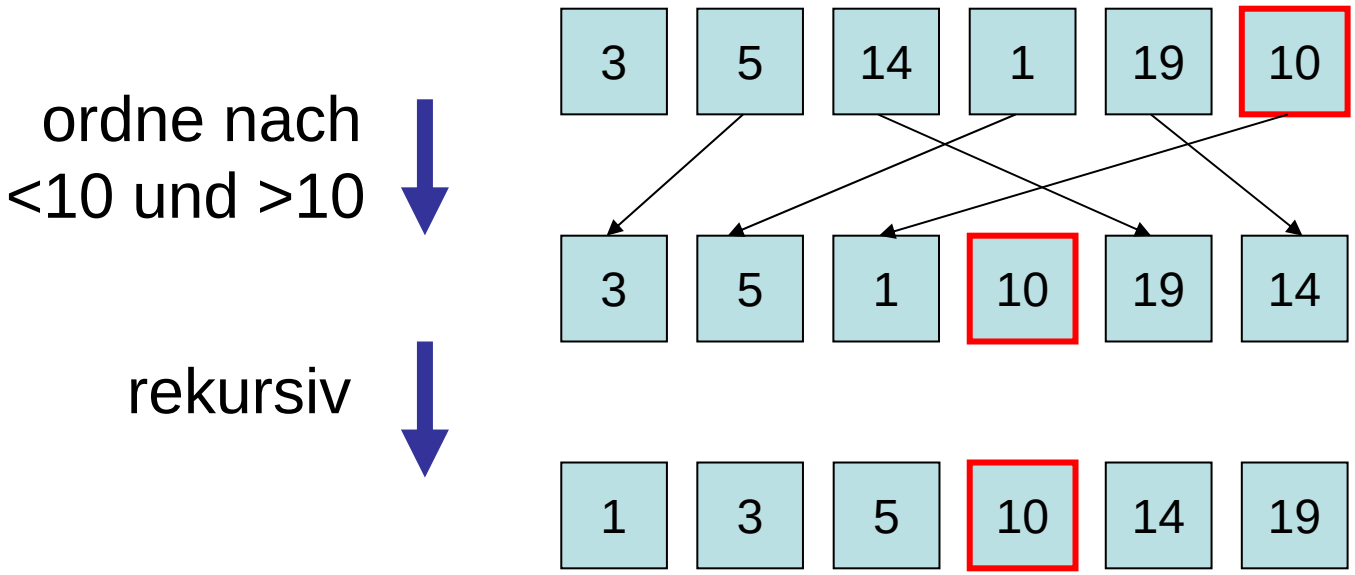
$2^T >= n! \Leftrightarrow$
$T >= \log(n!) = \Theta(n \log n)$

# Untere Schranke

Theorem 5.2: Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Schritte, um eine Folge von $n$ Elementen zu sortieren.

# Quicksort

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

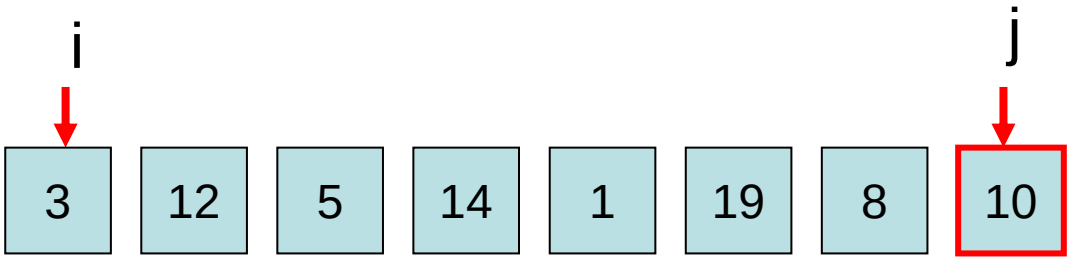


# Quicksort

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

# Quicksort

Laufzeit bei zufälligem Pivot-Element:

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

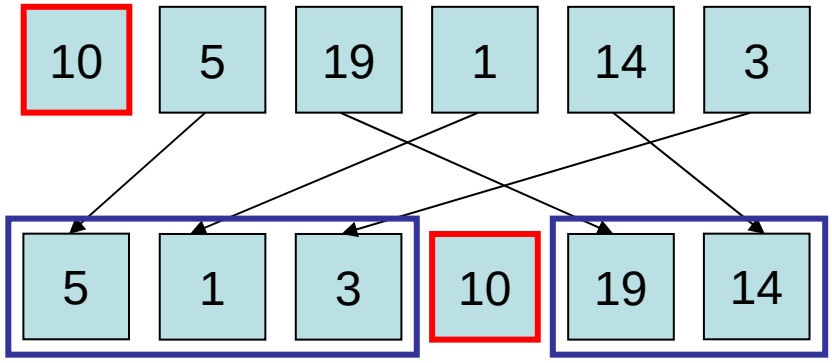
Theorem 5.3:

$$C(n) \leqslant 2n \ln n \leqslant 1.4\, n \log n$$

# Beweis von Theorem 5.3

- $s=\langle e_1,\ldots,e_n\rangle$: sortierte Sequenz

ordne nach
<10 und >10




- Quicksort: nur Vergleiche mit Pivotelement, Pivotelement nicht im rekursivem Aufruf
- $e_i$ und $e_j$ werden $\leqslant 1$-mal verglichen und nur, wenn $e_i$ oder $e_j$ Pivotelement ist

# Beweis von Theorem 5.3

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

$$C(n) = E[\sum_{i<j} X_{i,j}] = \sum_{i<j} E[X_{i,j}]$$
$$= \sum_{i<j} Pr[X_{i,j} = 1]$$

Lemma 5.4: $Pr[X_{i,j}] = 2/(j-i+1)$

# Beweis von Theorem 5.3

Lemma 5.4: $\Pr[X_{i,j}] = 2/(j-i+1)$

Beweis:

- $M=\{e_i,\ldots,e_j\}$
- Irgendwann wird Element aus $M$ als Pivot ausgewählt (bis dahin bleibt $M$ zusammen)
- $e_i$ und $e_j$ werden nur verglichen, wenn $e_i$ oder $e_j$ als Pivot ausgewählt werden
- $\Pr[e_i$ oder $e_j$ in $M$ ausgewählt$]=2/|M|$

# Beweis von Theorem 5.3

$$C(n) = \sum_{i<j} \Pr[X_{i,j}=1] = \sum_{i<j} 2/(j-i+1)$$
$$= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} 2/k \leqslant \sum_{i=1}^{n} \sum_{k=2}^{n} 2/k$$
$$= 2n \sum_{k=2}^{n} 1/k \leqslant 2n \ln n$$

Erwartungsgemäß ist Quicksort also sehr effizient (bestätigt Praxis).

# Ist Kostenmodell fair?

Bisher haben wir angenommen: jeder Vergleich kostet eine Zeiteinheit.

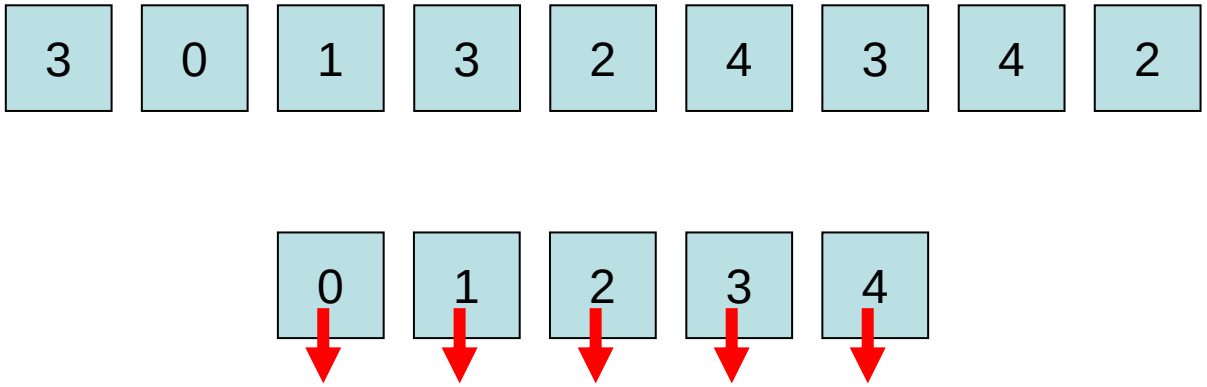
Besser: Bitmodell

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: Sequence of Element)
   b: Array [0..K-1] of Sequence 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 $\leqslant K^d$

# Radixsort

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

Laufzeit: O(d(n+K))

Falls maximale Zahlengröße O(log n), dann alle Zahlen $\leqslant n^d$ für konstantes d.

In diesem Fall (K=n) Laufzeit O(n).

# Radixsort

Beispiel:

| 12 | 203 | 3 | 74 | 24 | 17 | 112 |
|---|---|---|---|---|---|---|

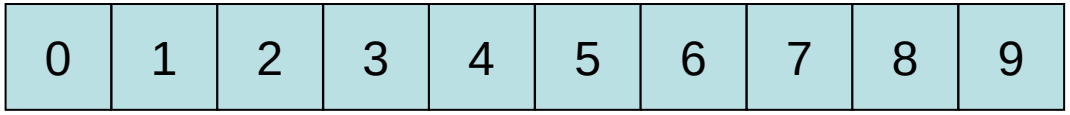
Ordnung nach Einerstelle:

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

# Radixsort

Ergebnis nach Einerstelle:

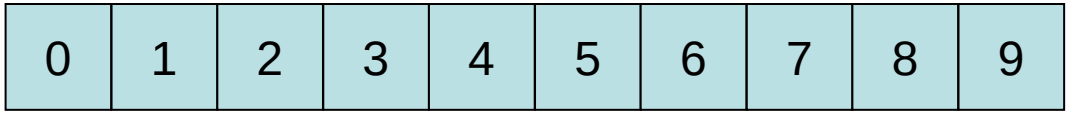

Ordnung nach Zehnerstelle:



# 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.

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.

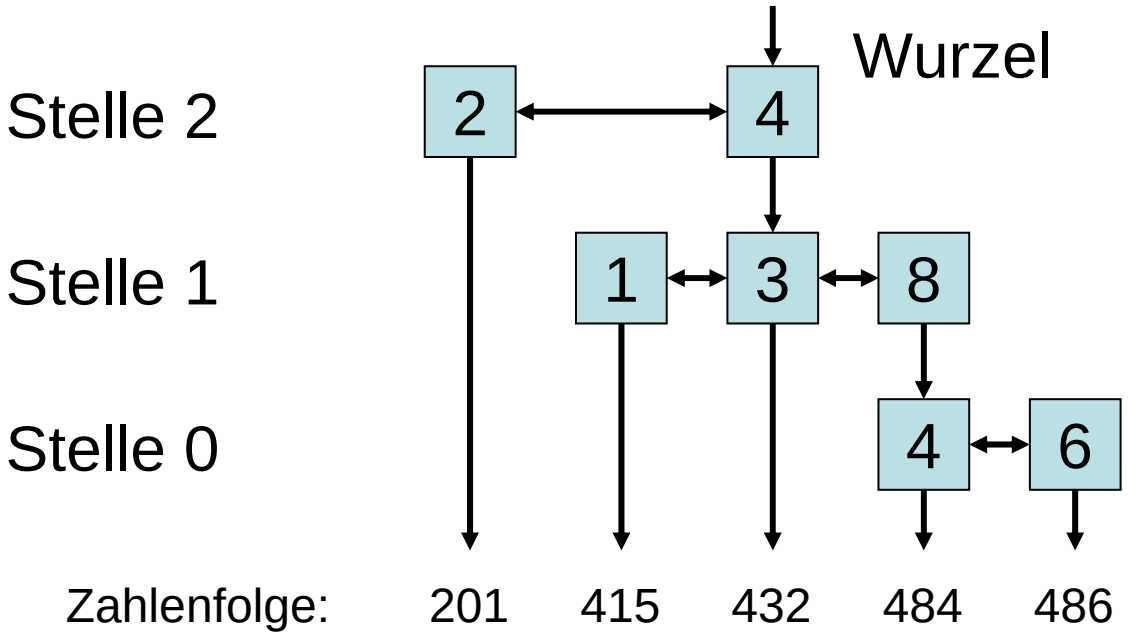
# Patriciasort

Annahme: Ziffern aus {0,…,K-1}

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

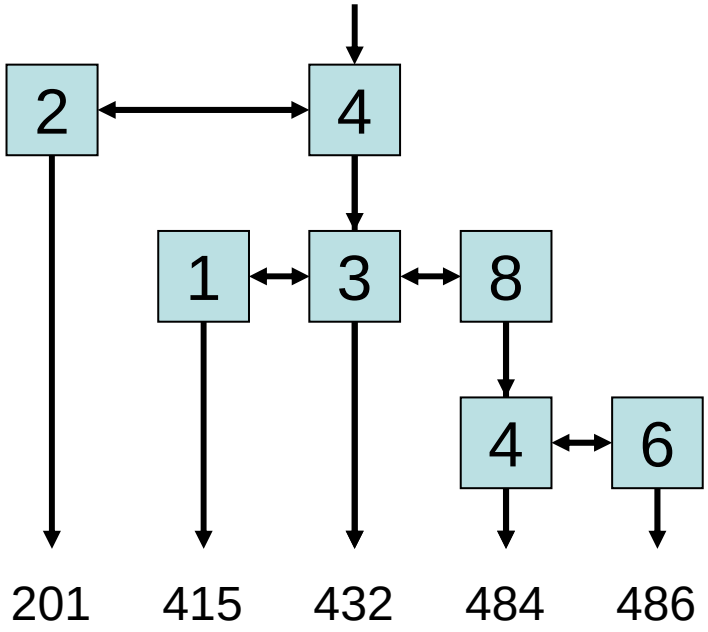
Beispiel: K=10, d=3

Ziel:



# Patriciasort

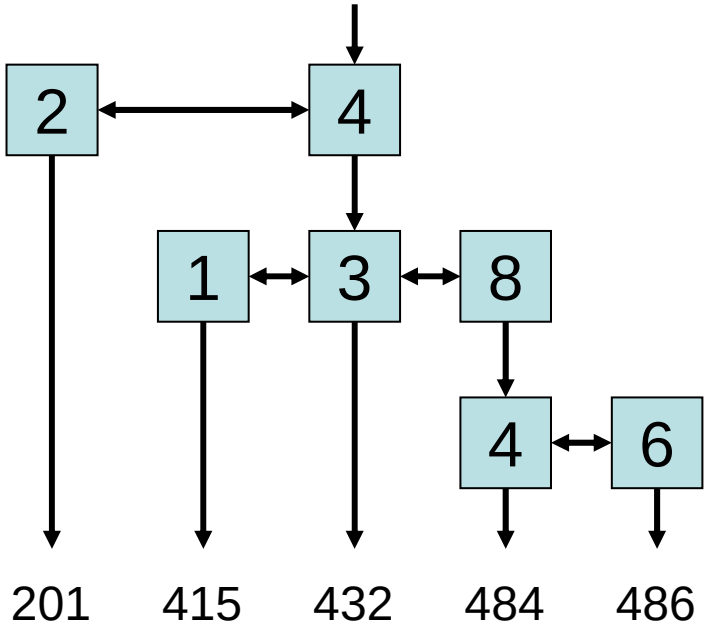
Aufbau für Eingabe (432,415,201,484,486):




Einbauzeit pro Zahl: O(#Zahlen + #Ziffern pro Zahl) = O(n+d)

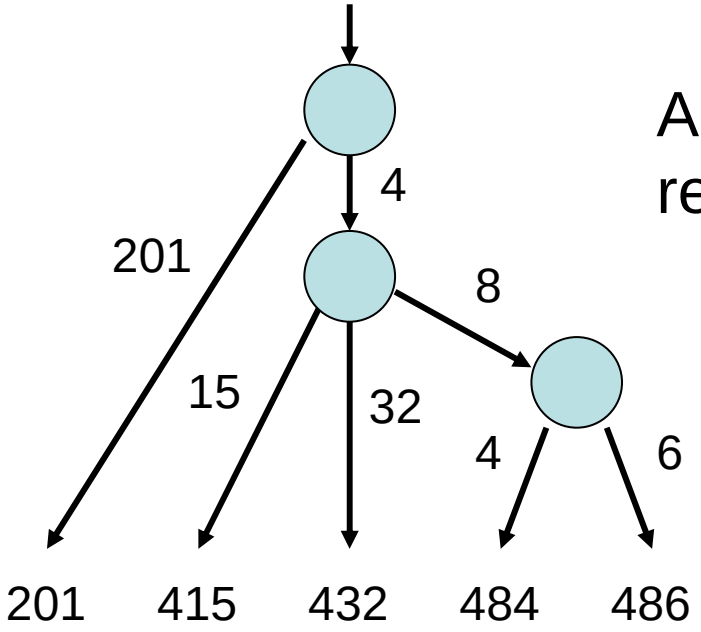
# Patriciasort

Ist wie Patricia Trie, nur mit Listen



# Patriciasort

Idealer Patricia Trie ohne Listen:




Aber Listenstruktur reicht für Effizienz.

# Patriciasort

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

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. $n_i \leqslant 2^i$: Patriciasort($M_i$)
  2. $n_i > 2^i$ und $n_i > n^{1/3}$: Radixsort($M_i$)
  3. $n_i > 2^i$ und $n_i \leqslant n^{1/3}$: 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:

$$O(\sum_i (n_i\, 2^i + n^{2/3})) = O(n).$$

# Selektion

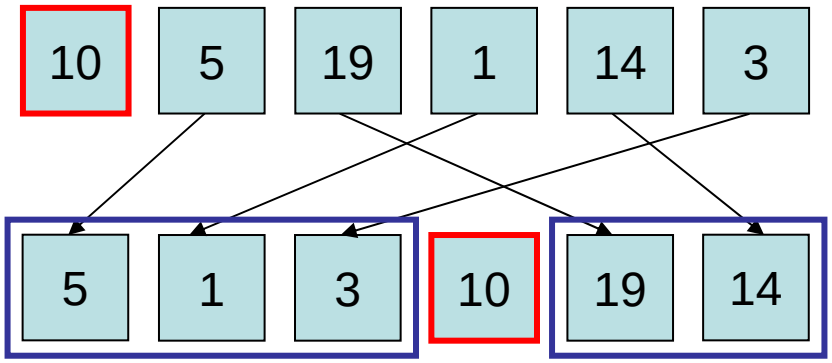
Problem: finde $k$-kleinstes Element in einer Folge von $n$ Elementen

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
```

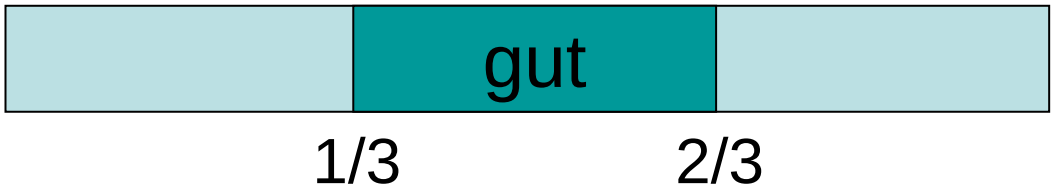
# Quickselect

- C(n): erwartete Anzahl Vergleiche

Theorem 5.6: C(n)=O(n)

Beweis:

- Pivot ist gut: keine der Teilfolgen länger als 2n/3
- Sei p=Pr[Pivot ist gut]




- p=1/3

# Quickselect

- Pivot gut: Restaufwand $\leqslant C(2n/3)$
- Pivot schlecht: Restaufwand $\leqslant C(n)$

$$C(n) \leqslant n + p \cdot C(2n/3) + (1-p) \cdot C(n)$$
$$\Leftrightarrow C(n) \leqslant n/p + C(2n/3)$$
$$\leqslant 3n + C(2n/3) \leqslant 3(n+2n/3+4n/9+\ldots)$$
$$\leqslant 3n \sum_{i>=0} (2/3)^i$$
$$\leqslant 3n / (1-2/3) = 9n$$

# 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 ($5 \leqslant m \leqslant 21$).
- 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. $m$:=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 $k \leqslant |S_1|$, 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

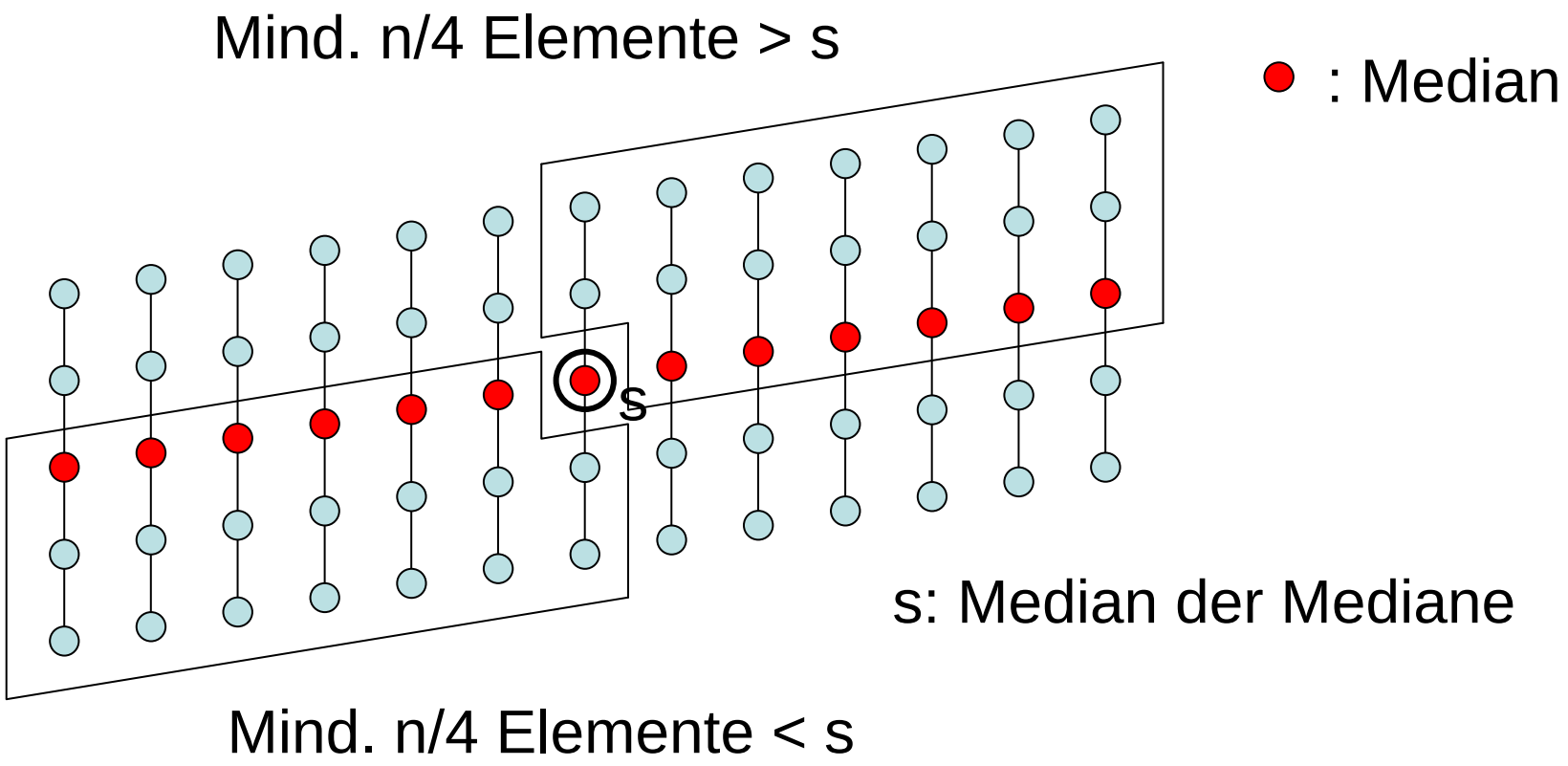
Laufzeit $T(n)$ des BFPRT-Algorithmus:

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

Lemma 5.7: Schritt 6/7 ruft BFPRT mit maximal $\lfloor (3/4)n \rfloor$ Elementen auf

# BFPRT-Algorithmus

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



# BFPRT-Algorithmus

Laufzeit für $m=5$:

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

für eine Konstante $c$.

Theorem 5.8: $T(n) \leqslant d \cdot n$ für eine Konstante $d$.

Beweis: Übung.

# Nächstes Kapitel

Verschiedenes:

- Union-Find Datenstrukturen
- Buddy-Datenstrukturen für Speicher- und Bandbreitenallokation