Grundlagen der Algorithmen und Datenstrukturen

Kapitel 5

Christian Scheideler + Helmut Seidl

SS 2009

Wörterbuch

S: Menge von Elementen

Jedes Element e identifiziert über key(e).

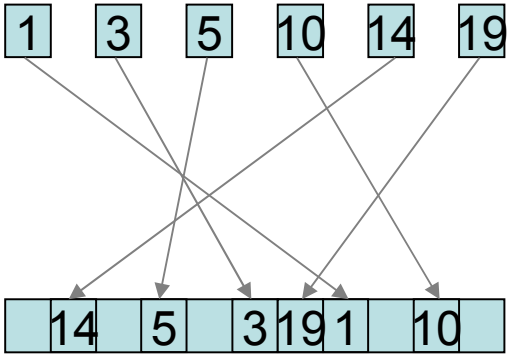
Operationen:

- S.insert(Element e): $S = S \cup \{e\}$;
- S.remove(Key k): $S = S \setminus \{e\}$; wobei e das Element ist mit key(e)=k
- S.find(Key k): Falls es ein $e \in S$ gibt mit key(e)==k, dann gib e aus, sonst gib null aus

Statisches Wörterbuch

Lösungsmöglichkeiten:

- Perfektes Hashing
  - Vorteil: Suche in konstanter Zeit
  - Nachteil: keine Ordnung auf Elementen, d.h. Bereichsanfragen (alle Namen, die mit A anfangen) teuer



- Speicherung der Daten in sortiertem Feld

1	3	5	10	14	19

  - Vorteil: Bereichsanfragen möglich
  - Nachteil: Suche teurer (logarithmische Zeit)

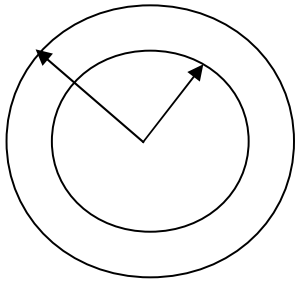
Sortierproblem

- Eingabe: Sequenz $s = \langle e_0,\ldots,e_{n-1}\rangle$ mit Ordnung $\leq$ auf den Schlüsseln $key(e_i)$
  (Beispiel: 5 | 10 | 19 | 1 | 14 | 3)

Ausgabe: Sequenz $s' = \langle e'_0,\ldots,e'_{n-1}\rangle$, so dass $key(e_i) \leq key(e_{i+1})$ für alle $0 \leq i < n-1$ 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

```
void selectionSort(Element[] a) {
    for (int i=0; i<a.length; i++)
        // bewege min{a[i],…,a[n]} nach a[i]
        for (int j=i+1; j<n;j++)
            if (a[i]>a[j]) 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

```
void insertionSort(Element[] a) {
    for (int i=1; i<n; i++)
        // bewege a[i] zum richtigen Platz
        for(int j=i-1; j>=0;j--)
            if (a[j]>a[j+1]) 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:

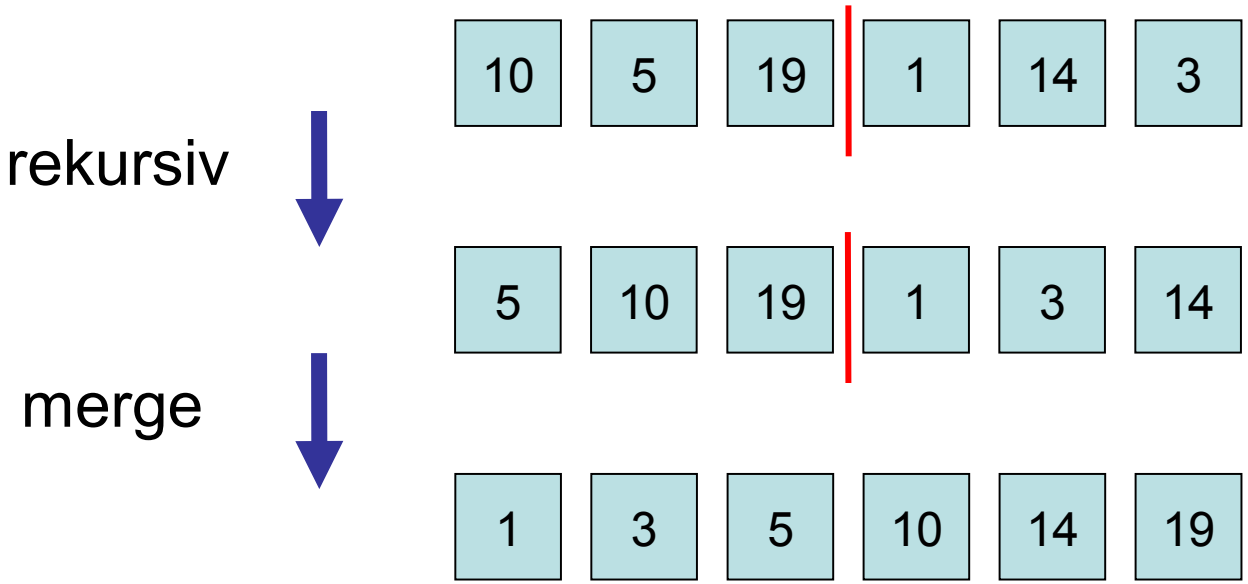
- Mit besserer Minimumstrategie worst-case Laufzeit O(n log n) erreichbar (mehr dazu in Kapitel 6)

Insertion Sort:

- Mit besserer Einfügestrategie ebenfalls worst-case Laufzeit O(n log n) erreichbar

Mergesort

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



Mergesort

```
static Element[] mergeSort (Element [] a) {
// a: zu sortierendes Feld, b: sortiertes Feld
    return  sort(a,0,a.length-1);
}           // fertig

Element[] void sort(Element[] a, int l, int r) {
    If (l>r) return new Element[0];
    if (l==r) {
       Element[] b = new Element[1];
       b[0] = a[l]; return b;
    }
    int t = (l+r)/2;
    Element[] b0 = sort(a,l,t);
    Element[] b1 = sort(a,t+1,r);
    return merge(b0,b1);
}
```

Mergesort

Theorem 5.2: 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
  $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$
- aus 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?

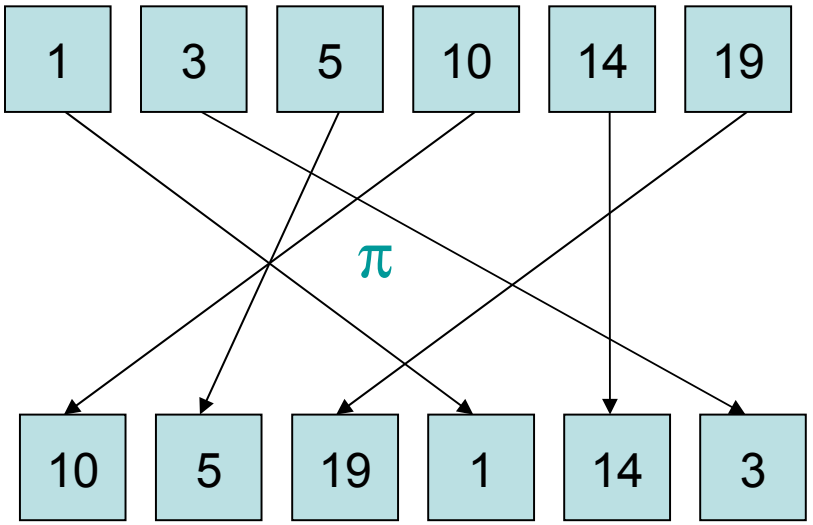
Antwort: $n! = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1$ viele

Untere Schranke

Permutation π der Eingabefolge:

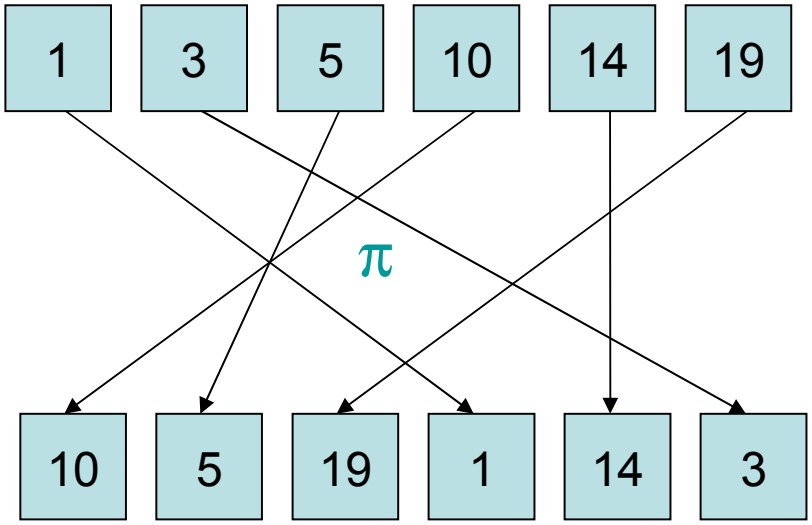
- Menge S:

- Eingabefolge:



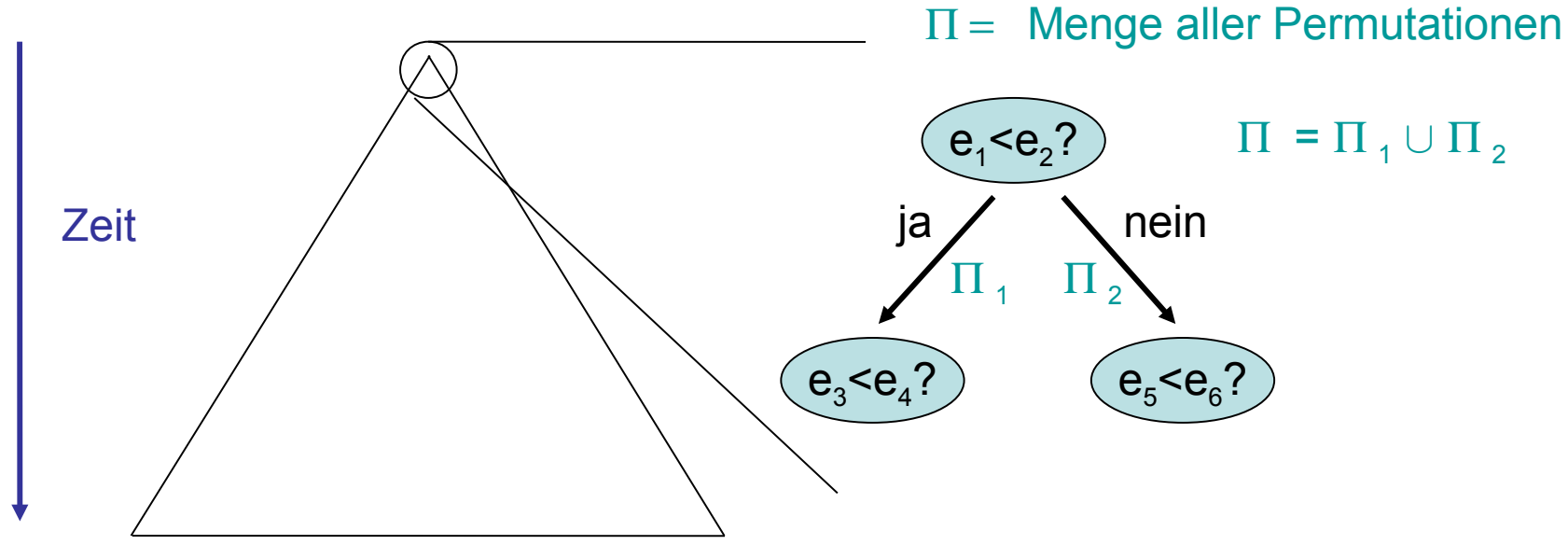
Untere Schranke

Wenn der Algorithmus sortieren kann, kann er auch die Permutation π ausgeben.



Untere Schranke

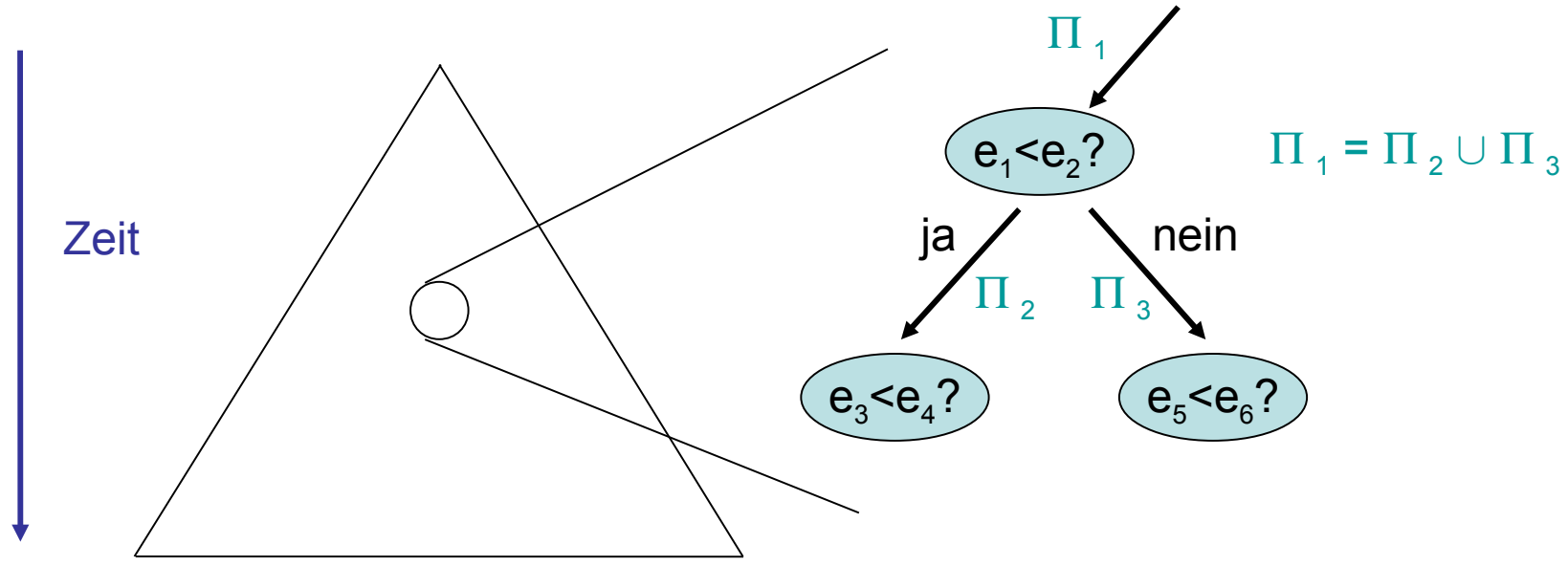
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:



Π_i : Permutationsmenge, die Bedingungen bis dahin erfüllt

Untere Schranke

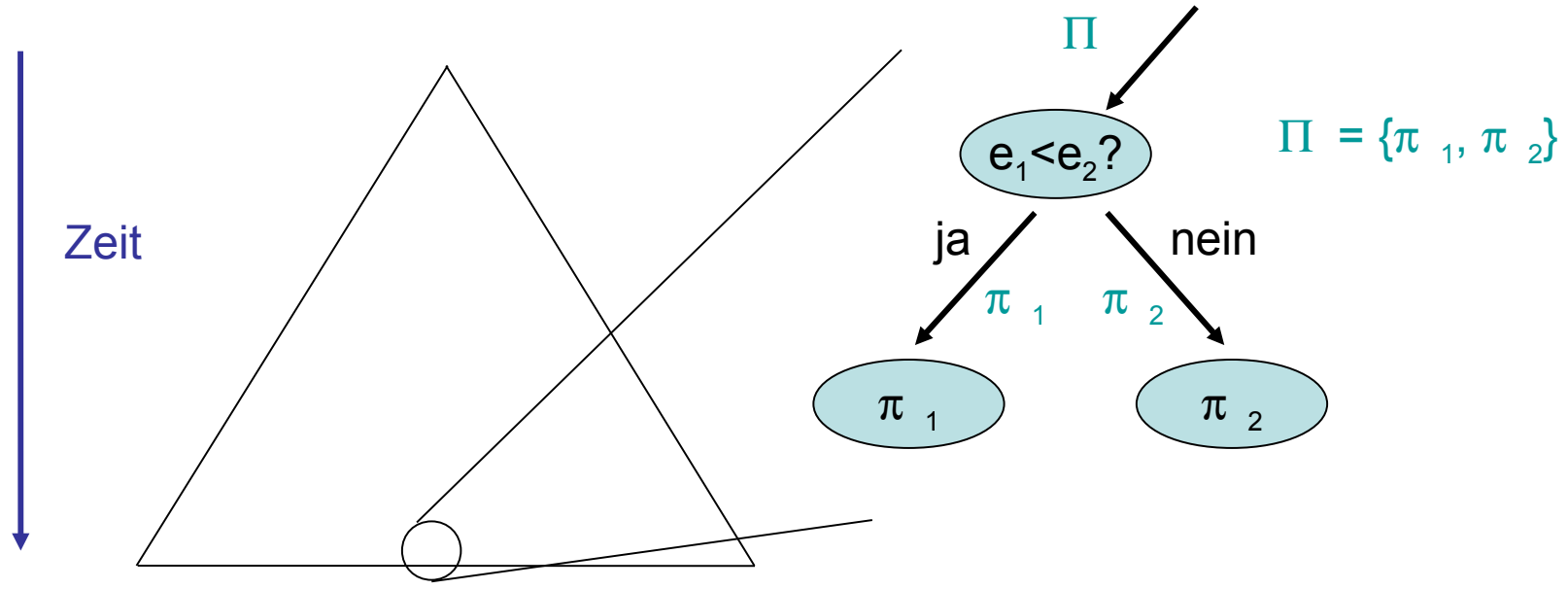
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:



Π_i : Permutationsmenge, die Bedingungen bis dahin erfüllt

Untere Schranke

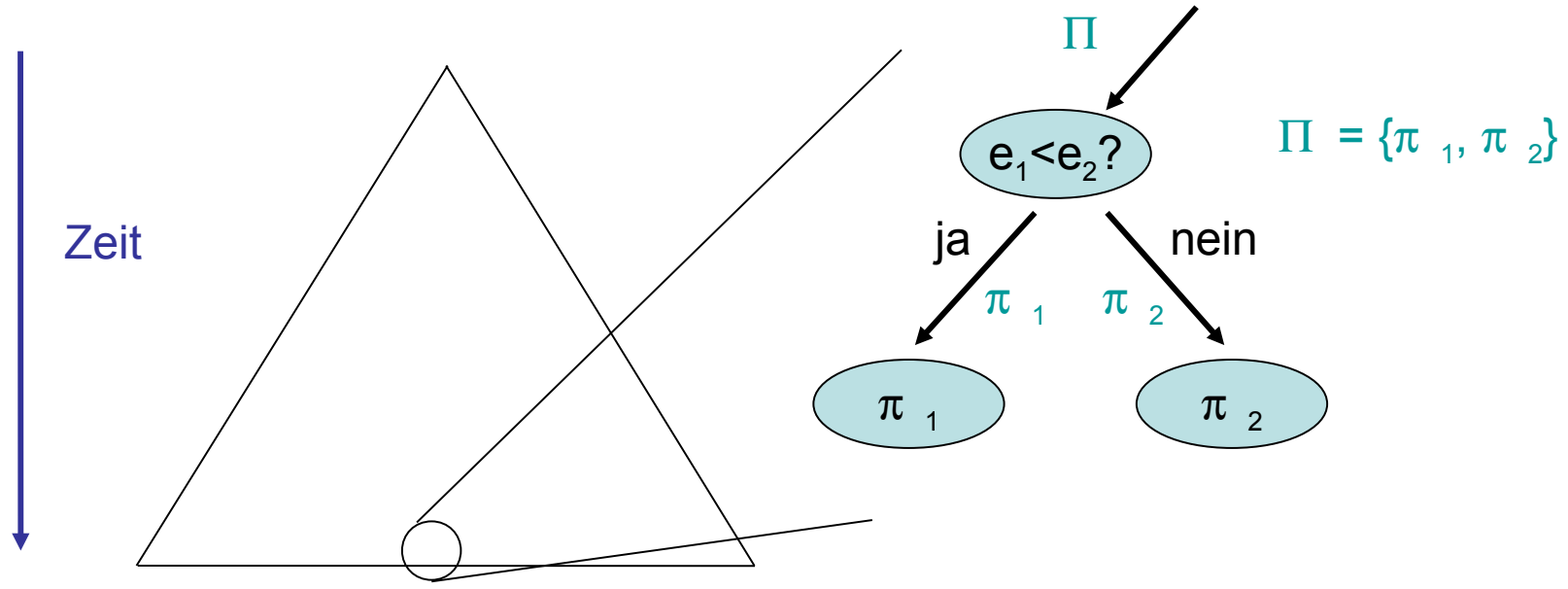
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:



π_i : eindeutige Permutation der Eingabefolge

Untere Schranke

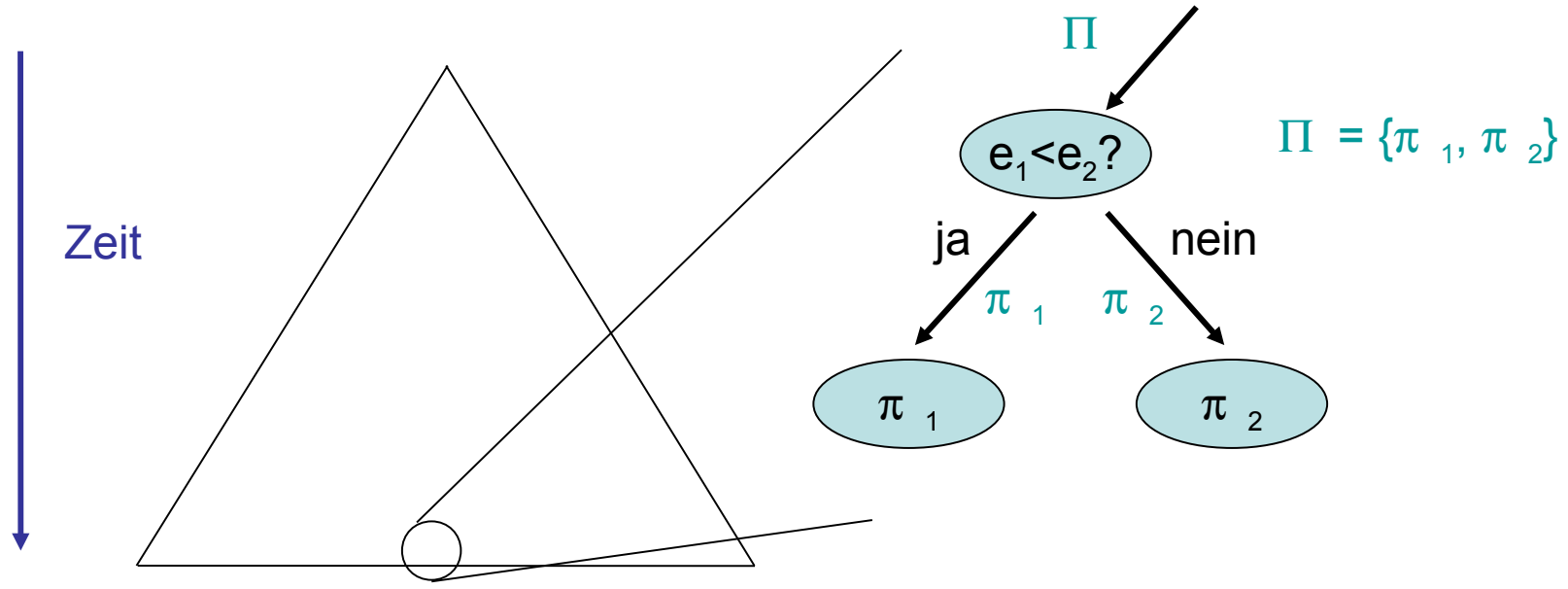
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:



Wieviele Blätter muss Entscheidungsbaum haben?

Untere Schranke

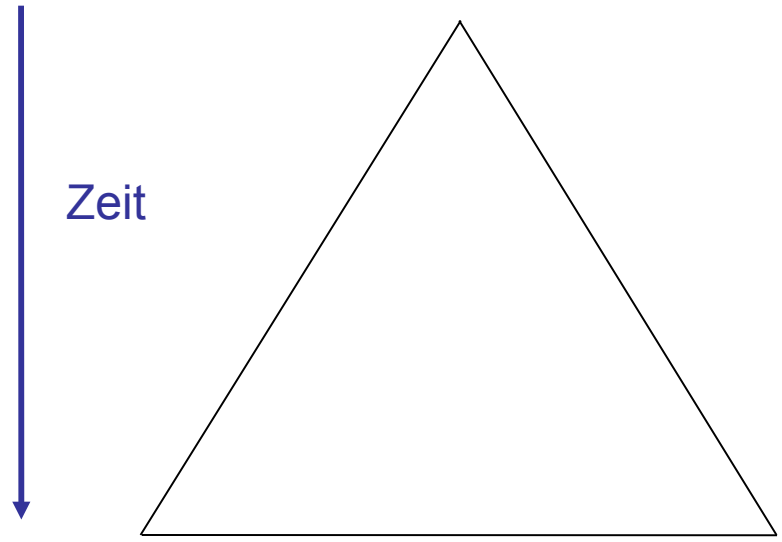
Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:



Mindestens n! viele Blätter!

Untere Schranke

Beliebiger vergleichsbasierter Algo als Entscheidungsbaum:



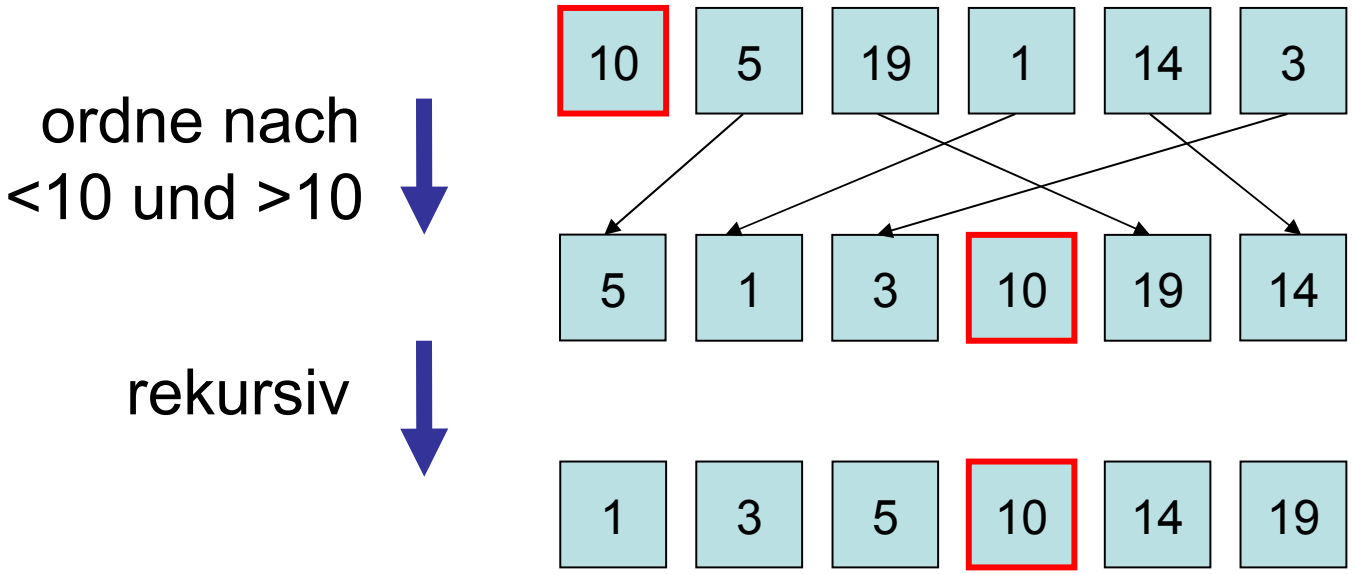
Baum der Tiefe T:
Höchstens 2^T Blätter

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

Jeder vegleichsbasierte Algo hat $\Omega(n \log n)$ Laufzeit

Quicksort

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



Quicksort

```
void quickSort(int l, int r) {
    // a[l..r]: zu sortierendes Feld
    if (r>l) {
        v=a[r]; int i=l-1; int j=r;
        do {     // ordne Elemente in [l,r-1] nach Pivot v
            do i=i+1; while (key(a[i])<key(v));
            do j=j-1; while (key(a[j])>key(v) && j!=l);
            if (i<j) a[i] ↔ a[j];
        } while (i<j);
        a[i] ↔ a[r];              // bringe Pivot an richtige Position
        quickSort(l,i-1);    // sortiere linke Teilfolge
        quickSort(i+1,r);    // sortiere rechte Teilfolge
}
```

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)

Quicksort

Laufzeit bei zufälligem Pivot-Element:

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

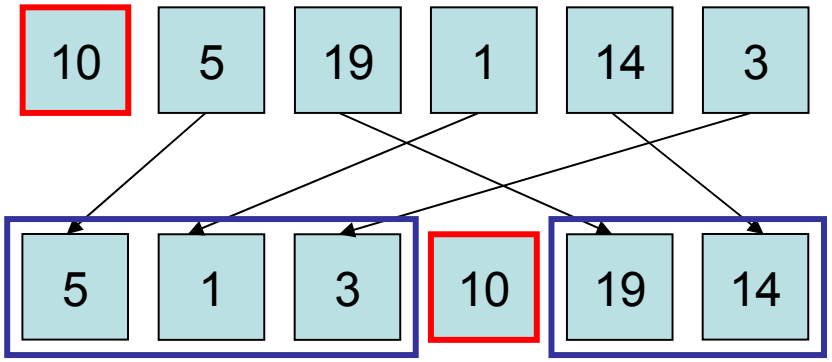
Theorem 5.6:

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

Beweis von Theorem 5.6

- $s=\langle e_0,\ldots,e_{n-1}\rangle$: sortierte Sequenz

ordne nach
<10 und >10



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

Beweis von Theorem 5.6

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

Beweis von Theorem 5.6

Lemma 5.7: $Pr[X_{i,j}] = 2/(j-i+1)$

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 5.6

$$C(n) = \sum_{i<j} \Pr[X_{i,j}=1] = \sum_{i<j} 2/(j-i+1)$$
$$= \sum_{i=0}^{n-1} \sum_{k=2}^{n-i} 2/k \leq \sum_{i=0}^{n-1} \sum_{k=2}^{n} 2/k$$
$$= 2n \sum_{k=2}^{n} 1/k \leq 2n \ln n$$

Erwartungsgemäß ist quickSort also sehr effizient (bestätigt Praxis).

Selektion

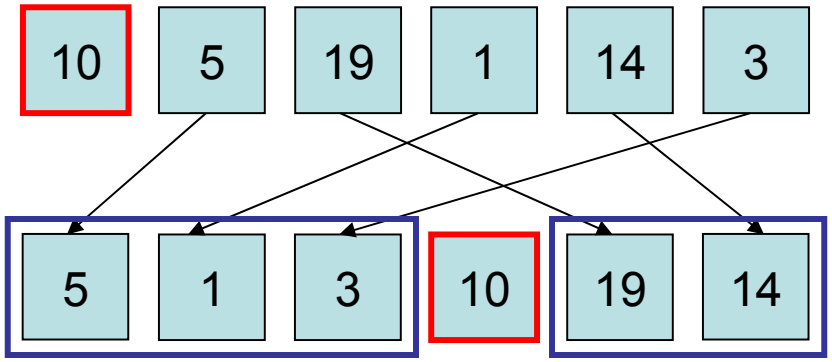
Problem: finde k-kleinstes Element in einer Folge von n Elementen

Lösung: sortiere Elemente, 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

```
Element quickSelect(int l, int r, int k) {
    // a[l..r]: Restfeld, k: k-kleinstes Element
    if(r==l) return a[l];
    int z = zufällige Position in {l,..,r}; a[z] ↔ a[r];
    Element v=a[r]; int i=l-1; int j=r;
    do {     // ordne Elemente in [l,r-1] nach Pivot v
        do i=i+1; while (key(a[i])<key(v));
        do j=j-1; while (key(a[j])>key(v) && j!=l);
        if (i<j) a[i] ↔ a[j];
    } (j>i);
    a[i] ↔ a[r];
    if (k<i) e = quickSelect(l,i-1,k);
    if (k>i) e = quickSelect(i+1,r,k);
    if (k==i) e=a[k];
    return e;
}
```

Quickselect

- C(n): erwartete Anzahl Vergleiche

Theorem 5.8: 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 $\leq C(2n/3)$

Pivot schlecht: Restaufwand $\leq C(n)$

$$C(n) \leq n + p\,C(2n/3) + (1-p)\,C(n)$$

$$\Leftrightarrow C(n) \leq n/p + C(2n/3)$$

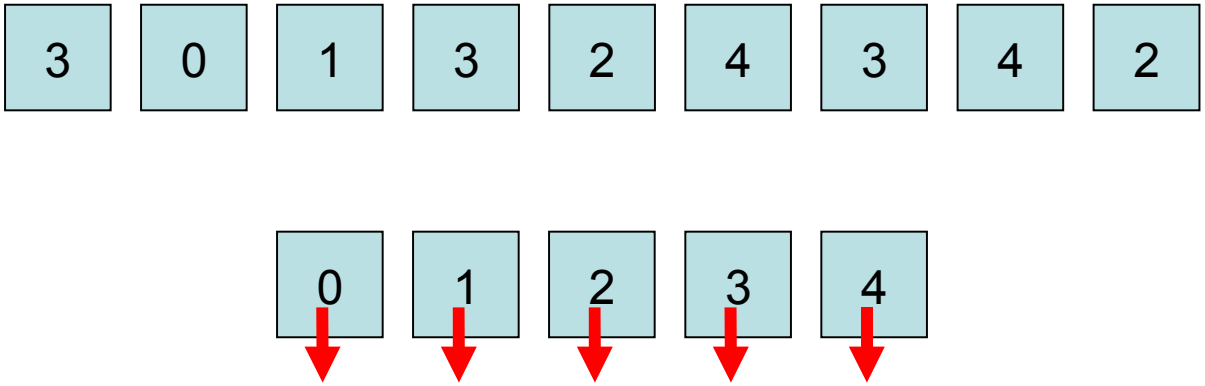
$$\leq 3n + C(2n/3) \leq 3(n+2n/3+4n/9+\ldots)$$

$$\leq 3n \sum_{i>=0} (2/3)^i$$

$$\leq 3n / (1-2/3) = 9n$$

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)

```
Sequence <Elem> kSort(Sequence <Elem> s) {
  Sequence <Elem>[] b =  new Sequence <Elem>[K];
  foreach (Element e∈s)
     b[key(e)].pushBack(e);
  return concatenate (b);
}
```

Laufzeit: O(n+K)

Problem: nur gut 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

Radixsort

```
Sequence<Elem> radixSort(Sequence<Elem> s) {
   Sequence<Elem> result = s;
   for (int i=0; i<d;i++)
       result = kSort(s,i);   // sortiere gemäß key_i(x)
                              // mit key_i(x) = (key(x) / k^i) % k
   return result;
}
```

Laufzeit $O(d(n+k))$

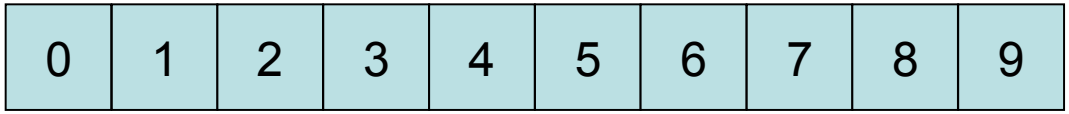
Falls maximale Zahlengröße $O(\log n)$, dann sind alle Zahlen $< n^d$ für konstantes d. In diesem Fall:

Laufzeit $O(n)$.

Radixsort

Beispiel:

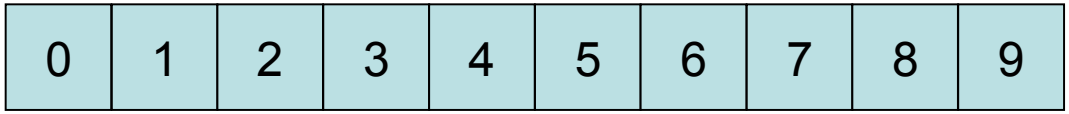
Ordnung nach Einerstelle:



Radixsort

Ergebnis nach Einerstelle:

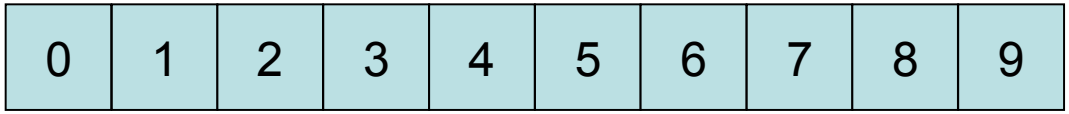
Ordnung nach Zehnerstelle:



Radixsort

Ergebnis nach Zehnerstelle:

Ordnung nach Hunderterstelle:



Radixsort

Ergebnis nach Hunderterstelle:

3	12	17	24	74	112	203

Sortiert!

Zauberei???

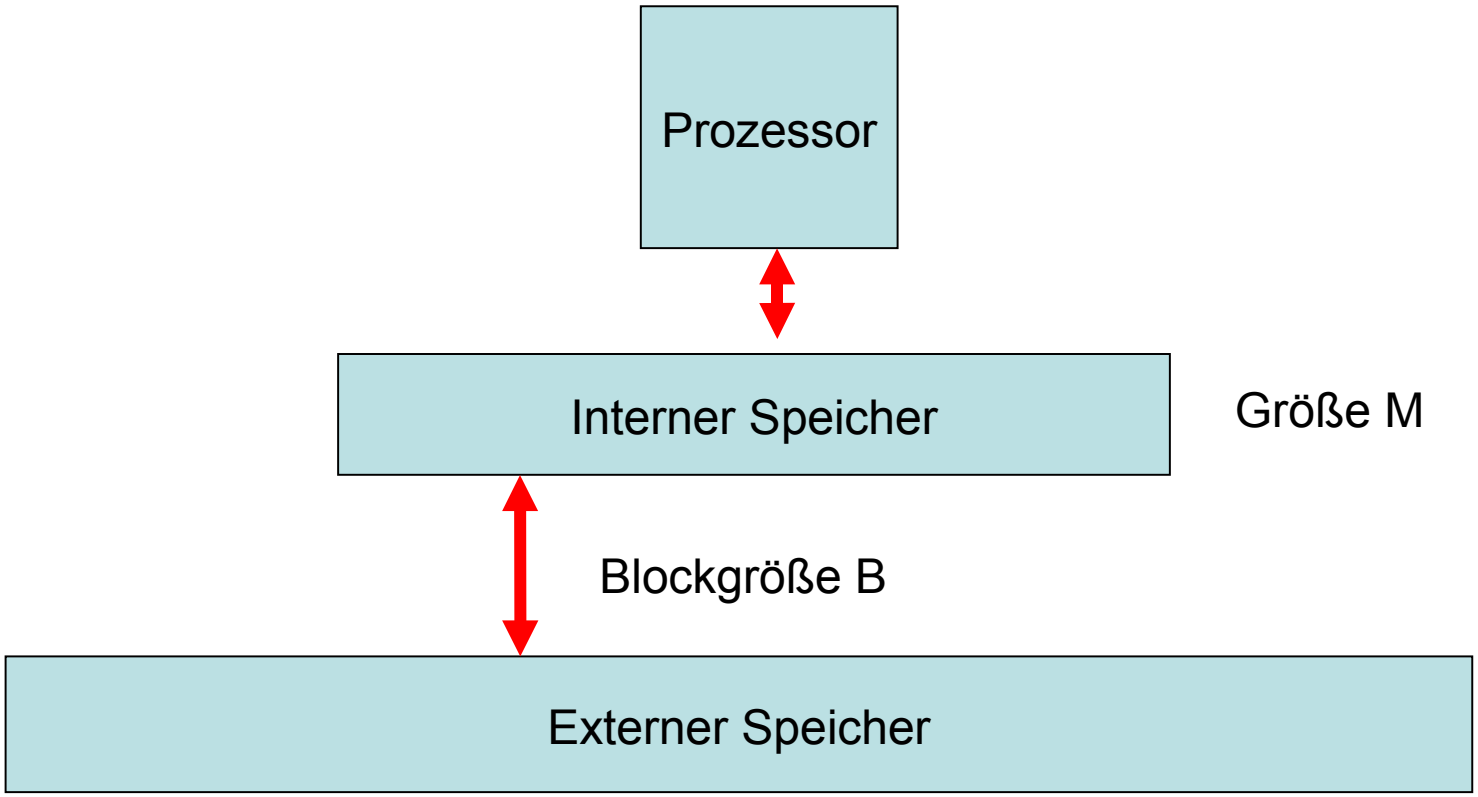
Radixsort

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

Externes Sortieren

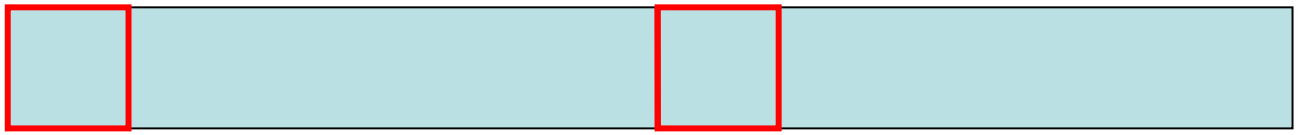
Heutige Computer:



Externes Sortieren

Problem: Minimiere Blocktransfers zwischen internem und externem Speicher

Lösung: verwende Mergesort

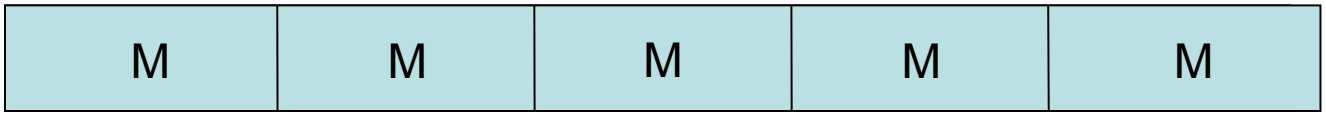


O(n/B) Blocktransfers

Externes Sortieren

Externes Mergesort (einfach):

- sortiere Folge in Abschnitten der Größe M (komplett intern möglich)



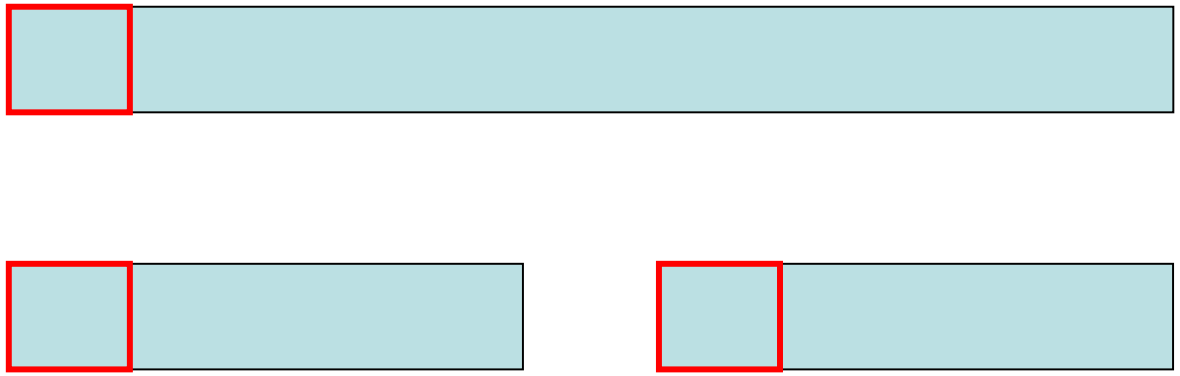
- Merge von jeweils zwei Teilfolgen bis zur Gesamtfolge (≤ 1+log(n/M) Durchläufe)

Anzahl Blocktransfers: O(2n/B (1+log(n/M)))

Externes Sortieren

Externes Mergesort (einfach):

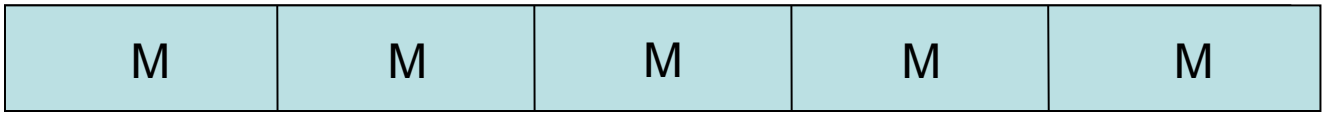
Mergen zweier Teilfolgen mittels 3 Blöcken im internen Speicher (rote Kästen):



Externes Sortieren

Externes Mergesort (verbessert):

- sortiere Folge in Abschnitten der Größe M (komplett intern möglich)



- Merge von jeweils M/B-1 Teilfolgen bis zur Gesamtfolge ($\leq 1+\log_{M/B}(n/M)$ Durchläufe)

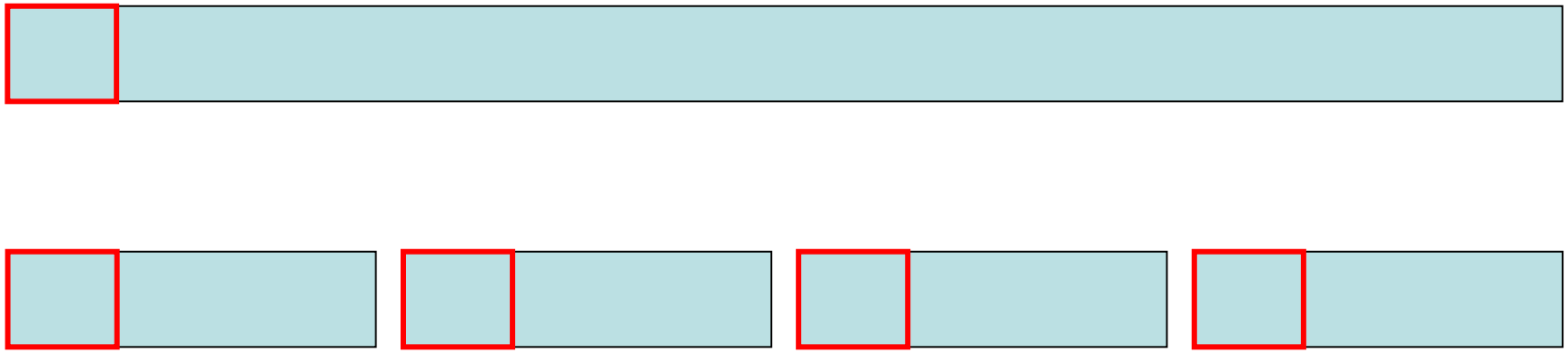
Anzahl Blocktransf.: $O(2n/B\ (1+\log_{M/B}(n/M)))$

Externes Sortieren

Externes Mergesort (verbessert):

Mergen von M/B-1 Teilfolgen mittels M/B

Blöcken im internen Speicher (rote Kästen):



Nächstes Kapitel

Priority Queues

(Damit ist Selection Sort viel besser!)