DHBW Lörrach
Duale Hochschule Baden-Württemberg

Algorithmen und Komplexität
TIF 21A/B

Dr. Bruno Becker

4. Sortieren

Sortieren

- **Eigenschaften von Sortierverfahren**
- Elementare Sortierverfahren
- Mergesort
- Quicksort

Sortieren - Problemstellung

- **Input:** *N* Datensätze (items) mit Schlüsseln $a_i.key$.
- **Output:** Datensätze so anordnen, so dass ihre Schlüssel aufsteigend sortiert sind, d.h. $a_1.key \leq a_2.key \leq \ldots \leq a_n.key$

- **Totalordnung:** Elemente der Folge sind paarweise vergleichbar
  $a_i.key\ (<, =, >)\ a_j.key$ für alle i,j
  - z.B. in Java „Interface Comparable“

- Im folgenden gehen wir i.a. von *wahlfreiem Zugriff (array) aus* und verwenden a_i statt $a_i.key$

Sortieren - Eigenschaften von Sortierverfahren

Kosten: Folge von *N* Elementen

- **Schlüssel-Vergleiche** *C(N) (Comparison)*
- **Element-Bewegungen** *M(N) (Movements)*

- Sortierverfahren heißt ***stabil,*** wenn Reihenfolge der *items* für gleiche *keys* erhalten bleibt
  - Z.B. Kunden mit gleichem Nachnamen und unterschiedlichem Vornamen
- Sortierverfahren heißt ***in-place,*** wenn es nur konstant zusätzlichen Speicher benötigt

Sortieren

- Eigenschaften von Sortierverfahren
- **Elementare Sortierverfahren**
- Mergesort
- Quicksort

Sortieren durch Auswahl (Selection Sort)

Idee: Finde Minimum in Folge,
vertausche das Element mit dem Anfang, rücke eine Position nach rechts,
Finde wieder Minimum und vertausche das Element mit neuem Anfang … bis Folge sortiert

```
Public void  SelectionSort(int[] a) {
  for (int i=0; i < a.length, i++) {
     int min = i;  // search Minimum of a[i],..a[n-1]
     for (int j= i+1; j < a.length; j++)
            if (a[j] < a [min] ) min = j;
     swap (a, i, min); // swap a[i] with a [min]
    }
}
```

Selection Sort - Beispiel

12	7	5	14	3	11
3	7	5	14	12	11
3	5	7	14	12	11
3	5	7	14	12	11
3	5	7	11	12	14
3	5	7	11	12	14
3	5	7	11	12	14

Selection Sort - Analyse

1. **Anzahl Vergleichsoperationen *C(N)***
   Anzahl Schlüsselvergleiche hängt nicht von der Input-Folge ab
   → Best Case = Average Case = Worst Case
   $\sum_{i=1}^{N-1}(N-i)$ = ***N*(N-1)/2*** → ***Aufwand O(N^2)*** **(nicht gut…)**

2. **Anzahl Bewegungen *M(N)***
   ***…**hängen auch nicht von Input-Folge ab*
   N-1 Swap-Operationen → ***Aufwand O(N)*** **(sehr gut, sogar optimal)**
3. **In-place?: *Ja* → *Kein zusätzlicher Speicher***
4. **Stabil?: *Nein* - *wieso nicht stabil?***

Gegenbeispiel: 10_1 10_2 5 3 → 3 5 10_2 10_1

Sortieren durch Einfügen (Insertion Sort)

Idee: In Iteration *i* fügt man a_i an der richtigen Stelle in die bereits sortierte Folge $a_1, \ldots a_{i-1}$ ein.
In Iteration *i* vertausche a_i mit jedem größeren Element links davon

```
Public void  InsertionSort(int[] a) {
  for (int i=1; i < a.length, i++) {
       for (int j= i; j > 0; j--)
           if (a[j] < a [j-1] )
               swap (a, j, j-1); // swap a[j] with a [j-1]
           else break;
   }
}
```

Insertion Sort - Beispiel

	12	7	5	14	3	11
i=1	7	12	5	14	3	11
i=2	5	7	12	14	3	11
i=3	5	7	12	14	3	11
i=4	3	5	7	12	14	11
i=5	3	5	7	11	12	14

Insertion Sort - Analyse

1. **Vergleichsoperationen *C(N)***
   Best Case (Folge sortiert) C(N) = N-1; Worst Case C(N)= $\sum_{i=2}^{N} i \sim O(N^2)$
   Average Case (jeweils die Hälfte der dem i-ten Element vorangehenden Elemente sind größer als das i-te Element) $O(N^2)$
2. **Bewegungen *M(N)***
   Best Case M(N) = 0 Swaps, Worst Case M(N) = $O(N^2)$, Average Case = $O(N^2)$,
3. **In-place?: *Ja***
4. **Stabil: *Ja***

Übungsblatt: Bubblesort – Algorithmus und Eigenschaften

Sortieren

- Eigenschaften von Sortierverfahren
- Elementare Sortierverfahren
- **Mergesort**
- Quicksort

Sortieren durch Verschmelzen (Mergesort)

Idee: Verschmelze zwei sortierte Teillisten

Beispiel:

3,4,7,9 und 1,2,5,6,8 => 1,2,3,4,5,6,7,8,9

Zum Verschmelzen von zwei sortierten Folgen der Länge n und m benötigt man

- Mindestens *Min(n,m)* Schlüsselvergleiche, und höchstens n+m-1 Vergleiche: $O(N)$

Idee: Divide & conquer

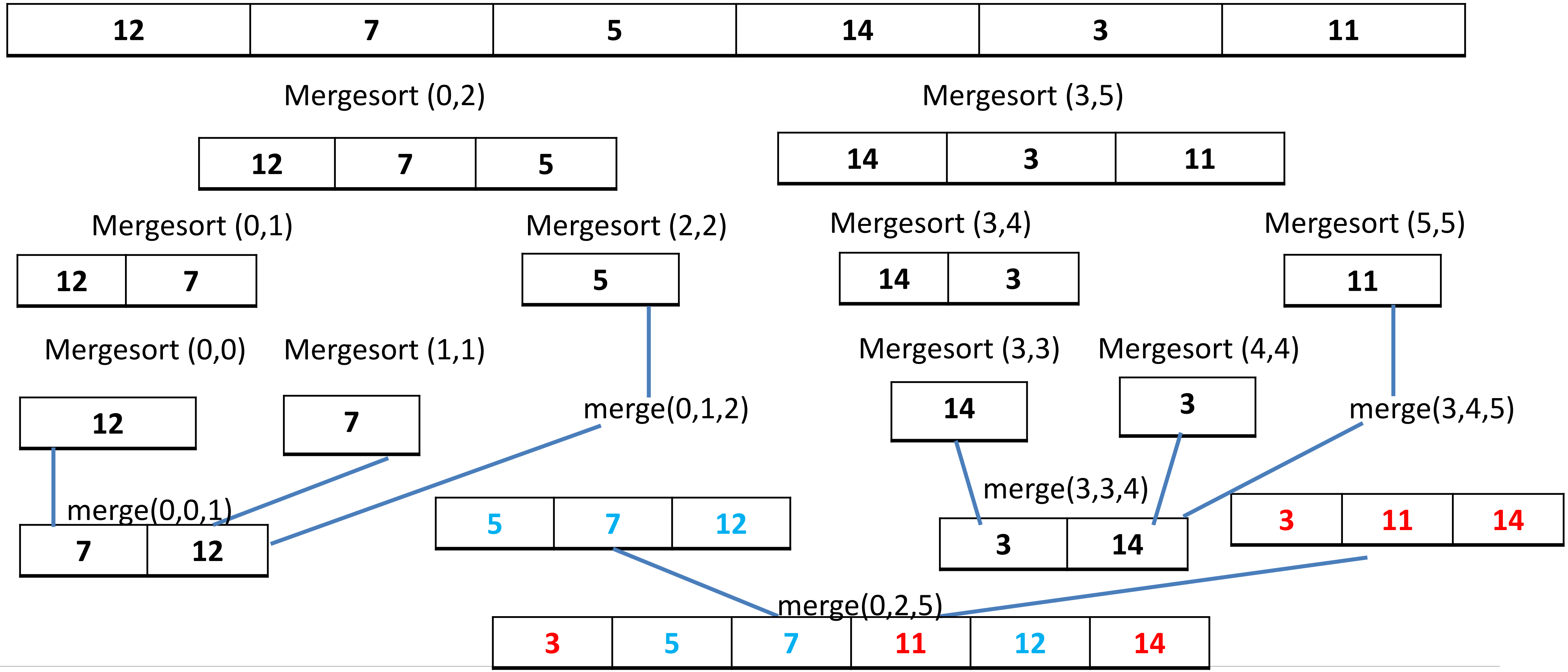
- Divide: Teile Folge F in zwei gleich große Teilfolgen F_1 und F_2
- Conquer: Mergesort(F_1); Mergesort (F_2);
- Merge: Bilde Resultatfolge durch Verschmelzen von F_1 und F_2

Mergesort – Algorithmus

```
void Mergesort(int[] a, int left, int right) {
  if (left < right)  {
        int  middle = (left+right)/2; // In der Mitte aufteilen
        mergesort (a, left, middle);
        mergesort (a, middle +1; right));
         merge (a, left, middle, right); // Merge – Sortieren von 2 Teilfolgen
   }
}
```

- Die Funktion *merge* sortiert die beiden Teilfolgen, indem
  - beide Teilfolgen werden von links nach rechts durchlaufen und die aktuellen Elemente verglichen
  - es wird jeweils das kleinere Element in Hilfsfeld übernommen und für diese Teilfolge auf das nächste Element gewechselt
  - Wenn eine Folge zu Ende ist, wird der Rest der anderen Folge übernommen
  - Am Schluss wird Hilfsfeld in ursprüngliches Feld übertragen

Mergesort - Beispiel



Mergesort – Algorithmus und Analyse

Anzahl Vergleichsoperationen

- Das Verschmelzen von 2 Listen mit n_1 bzw. n_2 Elementen benötigt maximal $n_1 + n_2 - 1$ Vergleiche
- Der Aufruf von Mergesort mit N Elementen benötigt: ***C(N) = 2*C(N/2) + O(N)***

➔C(N) = O(N log N)   (Beweis über vollständige Induktion)

Anzahl Bewegungen *M(N):* Merge zweier Teilfolgen der Länge N kostet 2N Bewegungen ➔ **M(N) = O(N log N)**

- Aufwandsbetrachtungen gelten für Best-, Average und Wort Case
- **Man kann zeigen, dass Sortieren im Worst Case Θ (N log N) benötigt ➔ Mergesort optimal bezügliche Worst case-Verhalten**

- **In-place?: *Nein, benötigt linearen zusätzlichen Platz!***
- **Stabil?: *Ja***

Sortieren

- Elementare Sortierverfahren
- Eigenschaften von Sortierverfahren
- Mergesort
- **Quicksort**

Quicksort

Idee: Mergesort ohne Mergeschritt. Bekanntester und in der Praxis schnellste Sortier-Algorithmus (Hoare,1962)

Partitioniere Folge so, dass für ein *j:*

- Element a_j am richtigen Platz steht
- Kein größeres Element links von *j* steht
- Kein kleineres Element rechts von *j* steht
- Sortiere rekursiv linken und rechten Teil

```
Public static void  Quicksort(int[] a, int lo, int hi) {
        if (hi <= lo ) return;
        int j = partition (a,lo,hi); // partition
        Quicksort (a,lo,j-1);
        Quicksort (a,j+1,hi);
}
```

Quicksort

Partitionierung: Pivot-Element a_{lo}; Zwei **Zeiger** $i=lo+1; j=hi;$

Wiederhole, bis sich die Zeiger i und j überkreuzen:

- Bewege i nach rechts solange $a_i < a_{lo}$
- Bewege j nach links solange $a_j > a_{lo}$
- Vertausche a_i mit a_j

Wenn Zeiger überkreuzt: Vertausche a_{lo} mit a_j

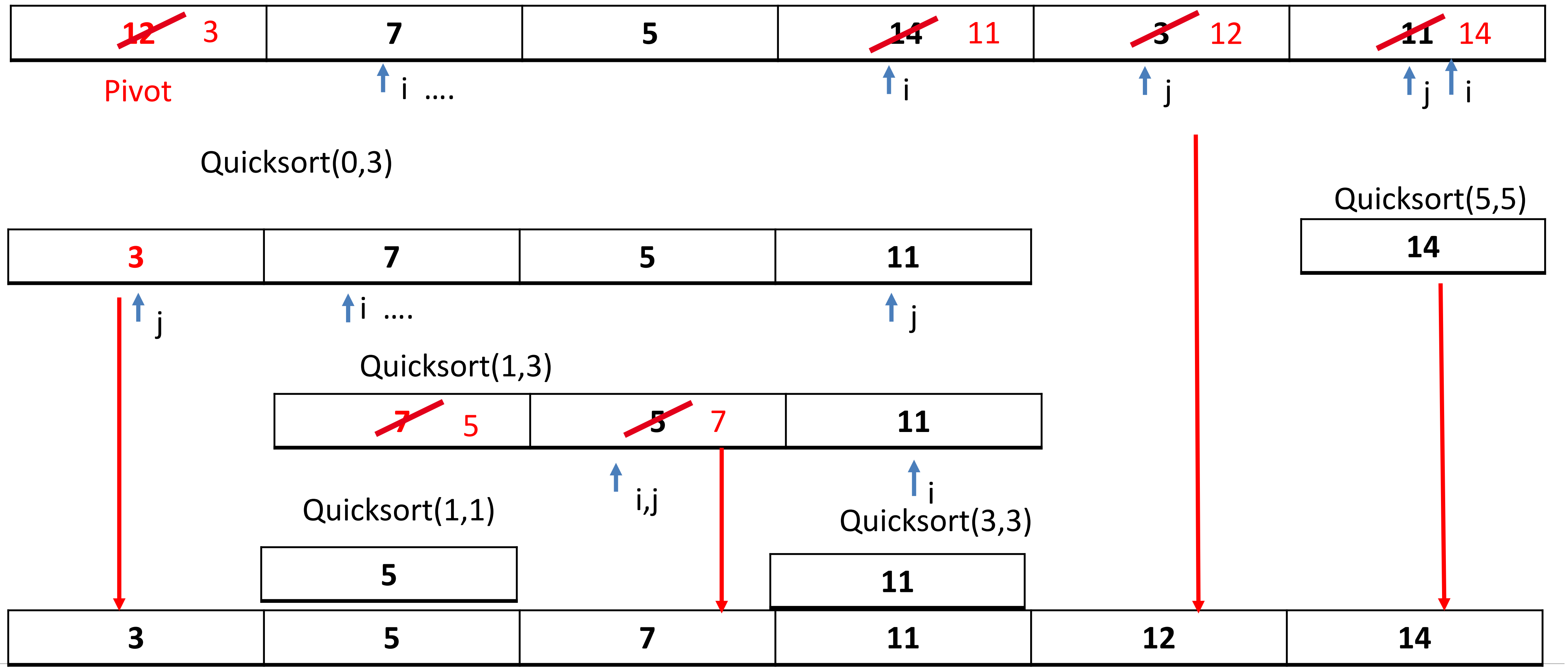
Beispiel:

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

...

E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Quicksort - Beispiel



Quicksort- Analyse

Quicksort verhält sich ähnlich wie Mergesort, sofern Partitionierung in zwei gleichgroße Teilfolgen. Das ist *fast* immer der Fall ➔ $O(N \log N)$ im Average Case

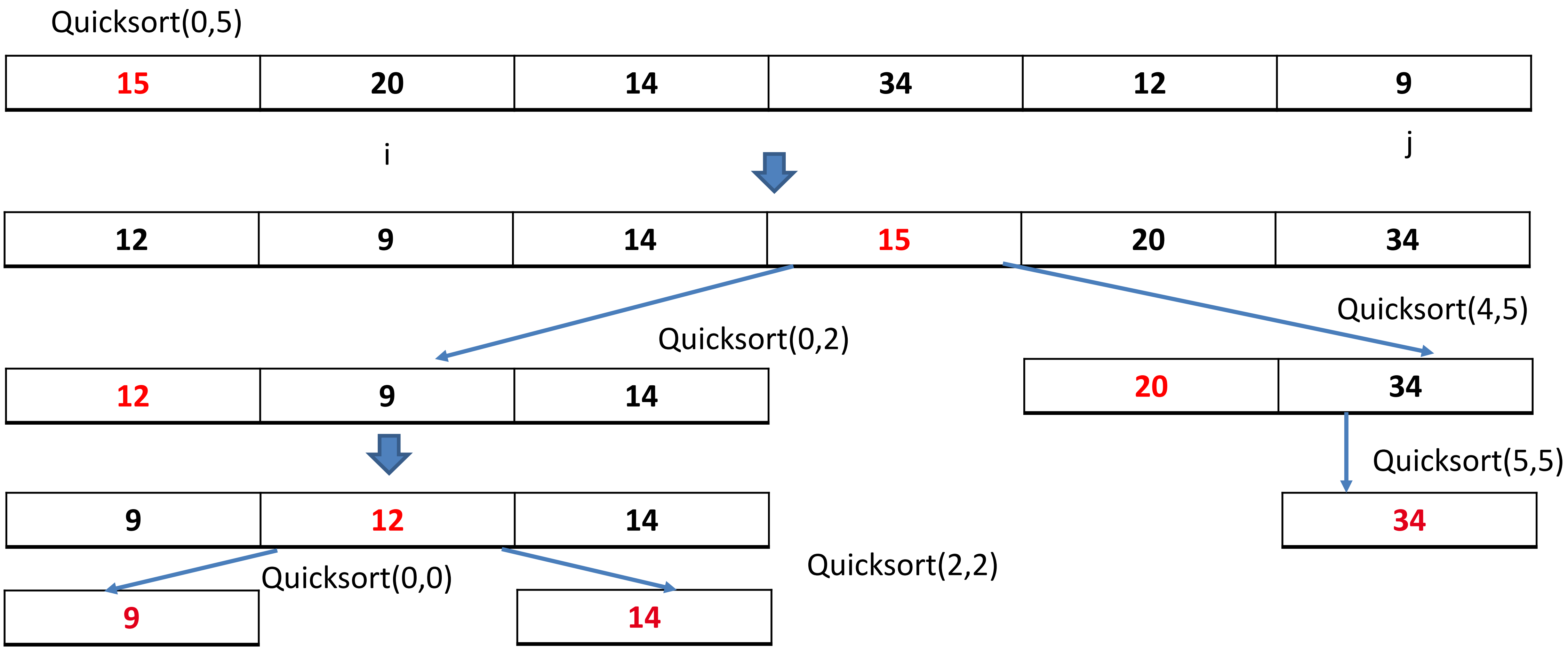
Ungünstig ist, wenn eine Teilfolge leer ist (Wann?) ➔ $O(N^2)$ im Worst Case

Praxis:

- Ca 40% mehr Vergleiche als Mergesort, aber weniger Bewegungen
- Schnellstes Sortierverfahren
- Varianten: Zufälliges Pivot-Element oder Zufällige Mischung der Folge vor dem Sortieren
  ➔ Weniger Abhängigkeit vom Input
- Variante: Quicksort bis zur Mindestanzahl Folge, darunter Insertion Sort

In-place?: Nicht ganz, log_n zusätzlicher Platz
Stabil: Nein

Quicksort -Übung



Zusammenfassung

- Elementare Sortierverfahren *Selection, Insertion, Bubble Sort* sind nur für kleinere Datenmengen geeignet, da im Mittel $O(N^2)$ Aufwand
- Sortieren benötigt im Worst Case Θ (N log N) –Aufwand
- Mergesort ist hinsichtlich Worst Case optimal, benötigt aber linearen zusätzlichen Platz
- Quicksort im WorstCase $O(N^2)$, im Durchschnitt und in der Praxis das schnellste bekannte Sortierverfahren
- Später in der Vorlesung folgt noch *Heapsort,* Worst Case optimal und *in-place.*