Algorithmen & Datenstrukturen

Suchen und Sortieren

Wolfgang Auer

Suchen

FHV

Das Suchen von Elementen ist eine der häufigsten Aufgaben in Anwendungen

- Abhängig von der Datenstruktur und der Information über die bereits vorhandene Ordnung der Daten kommen unterschiedliche Verfahren zum Einsatz
  - z.B.: Lineares Suchen, Binäres Suchen, Pattern-Matching (Suche in Zeichenketten)

Lineare Suche

- Gegeben ist ein Feld von Werten `values[]`. Es sind keine zusätzlichen Angaben über die zu untersuchende Datenmenge vorhanden
- Lösungsidee:
  Feld wird schrittweise durchlaufen bis das Element gefunden oder das Ende des Feldes erreicht wird ⇒ *Lineares Suchen*

```
/* n: Number of elements
   x: element to be searched for */
i = 0;
while ( (i < n) && (values[i] != x)) {
   i++;
}

if (i < n) {
   /* x was found */
}
```

Im schlechtesten Fall müssen alle *n* Elemente untersucht werden

Binäre Suche (1)

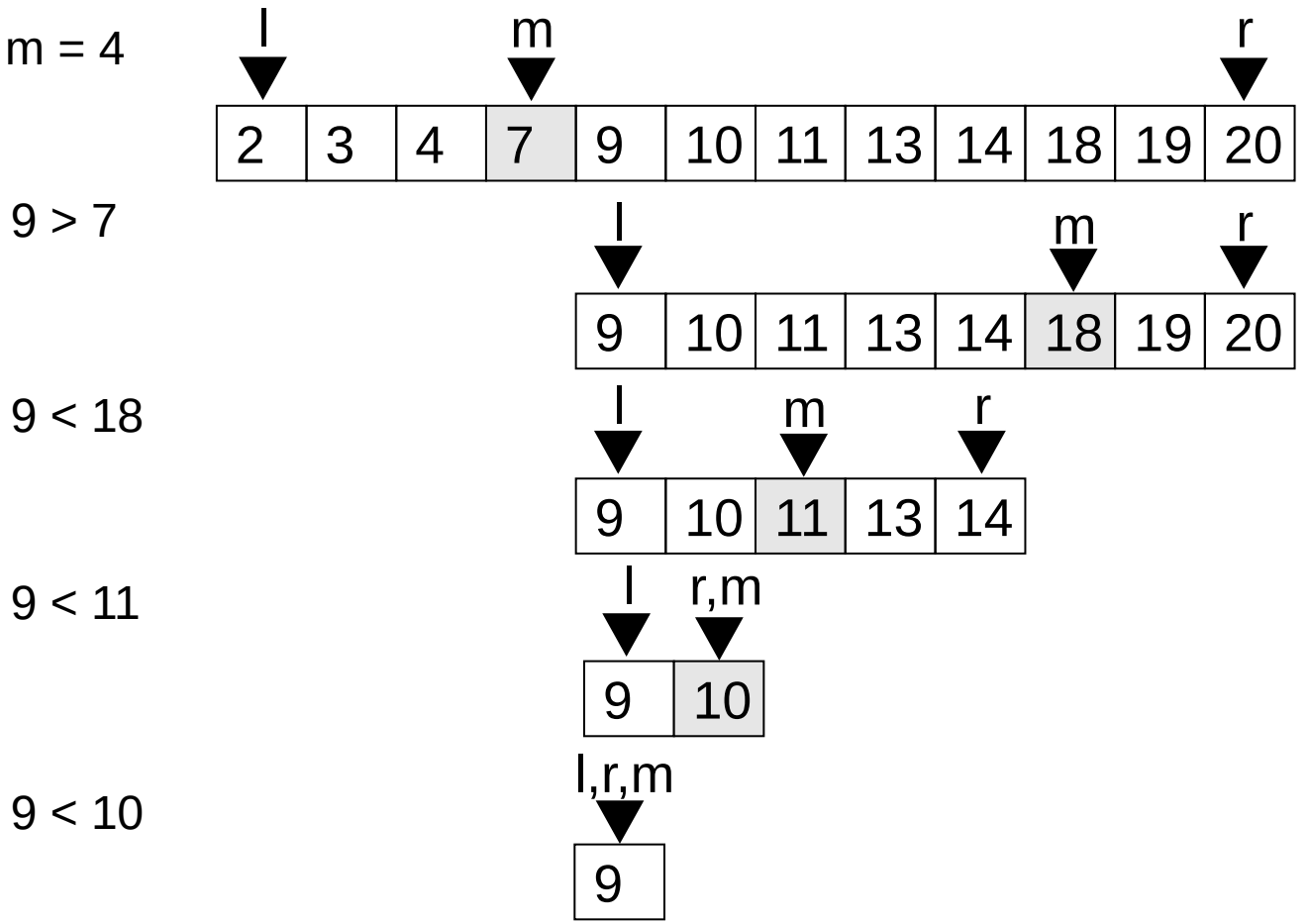
FHV

- Verbesserung der linearen Suche kann erzielt werden, wenn man in einer sortierten Datenmenge sucht ⇒ Ausnutzen der *Ordnung* der Daten
- Idee:
  - Wähle ein zufälliges Element `values[m]`
    - Ist `values[m] == x`, kann die Suche beendet werden
    - Ist `values[m] < x`, können alle Elemente mit Index kleiner oder gleich `m` ausgeschlossen werden
    - Ist `values[m] > x`, können alle Element mit Index größer oder gleich `m` ausgeschlossen werden
  - ⇒ Es werden schrittweise Elemente oberhalb oder unterhalb der gewählten Stichprobe eliminiert. Aus diesem Grund wird dieses Verfahren als *Binäre Suche* bezeichnet

Binäre Suche (2)

Prinzipielles Vorgehen

gesucht ist `9`, `m` wird zufällig im Bereich `l..r` gewählt

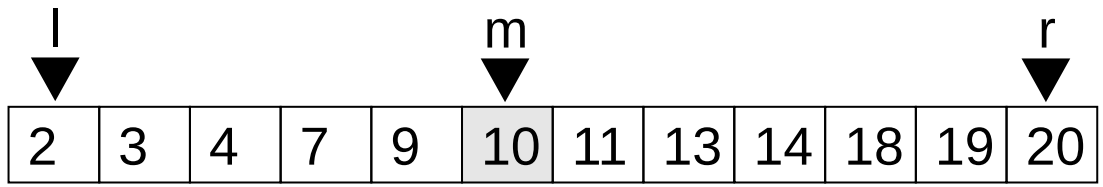


Binäre Suche (3)

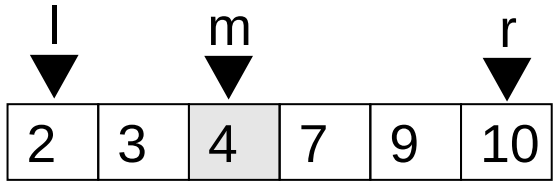
Verbesserung

gesucht ist `9`, `m` halbiert zu untersuchende Datenmenge

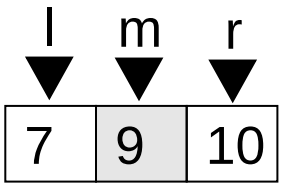
m = (r+l) / 2



9 < 10



9 > 4



$\Rightarrow$ maximale Anzahle der benötigten Vergleich: $\lceil ld\ n \rceil$

Binäre Suche (4)

FHV

```
l = 0;
r = nrOfValues;
m = 0;
found = false;

while ((l < r) && (!found)) {

     m = (l + r) / 2;

     if (values[m] == x) {
          found = true;
     } else if ( values[m] < x) {
          l = m + 1;
     } else {
          r = m;
     }
}
```

Sortieren

FHV

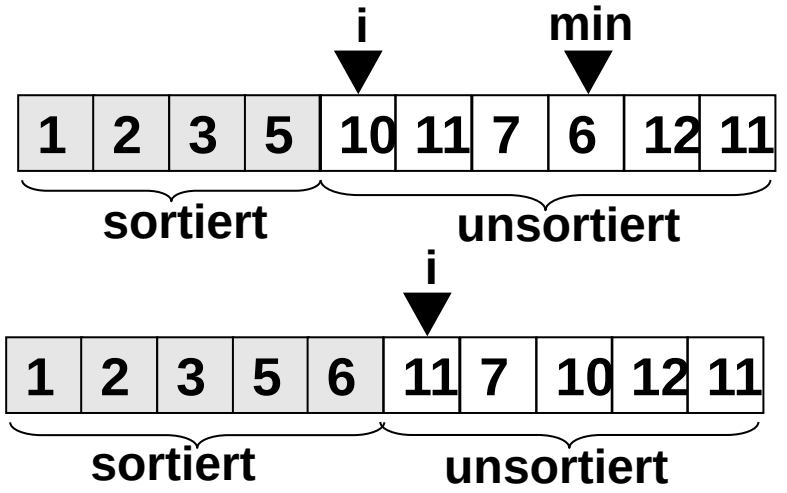
Sortieren ist das Anordnen einer Menge von Objekten in einer bestimmten Ordnung und dient zur Vereinfachung des späteren Suchens nach einem bestimmten Element

- Es werden *interne* und *externe* Verfahren unterschieden:
  - Interne Verfahren arbeiten mit Datenmengen, die sich direkt im Speicher befinden und den direkten Zugriff auf einzelne Elemente erlauben
    - Selection sort (Auswahlsortieren)
    - Insertion sort (Einfügesortieren)
    - Shell sort (Mehrfaches Einfügesortieren)
    - Bubble sort ("Bläschensortieren")
    - Quicksort
    - …
  - Externe Verfahren arbeiten mit Datenmengen auf externen Speichermedien, direkter Zugriff auf Elemente ist nicht möglich
    - Merge sort (Mischsortieren)
    - …

Selection sort (1)

Auswahlsortieren

- Geg: Feld von Ganzzahlen der Länge *n:* `int a[n]`
- Algorithmus:
  1. Setzte `i` auf den Anfang des Feldes
  2. **Wähle** ab `i` das kleinste Element a_{min}
  3. Tausche a_{min} mit a_i
  4. Setze `i = i + 1` und fahre bei Schritt 2 solange fort, bis das Ende des Feldes erreicht wird.



Selection sort (2)

Auswahlsortieren

```
/* n … number of values */
for (i = 0; i < n - 1; i++) {
     min = i;

     for (j = i + 1; j < n; j++) {
          if (values[j] < values[min]) {
               min = j;
          }
     } /* end for */

     /* swap values */
     temp = values[min];
     values[min] = values[i];
     values[i] = temp;
}
```

Anzahl der Vergleiche ≈ (n - 1) * n/2 ⇒ $O(n^2)$

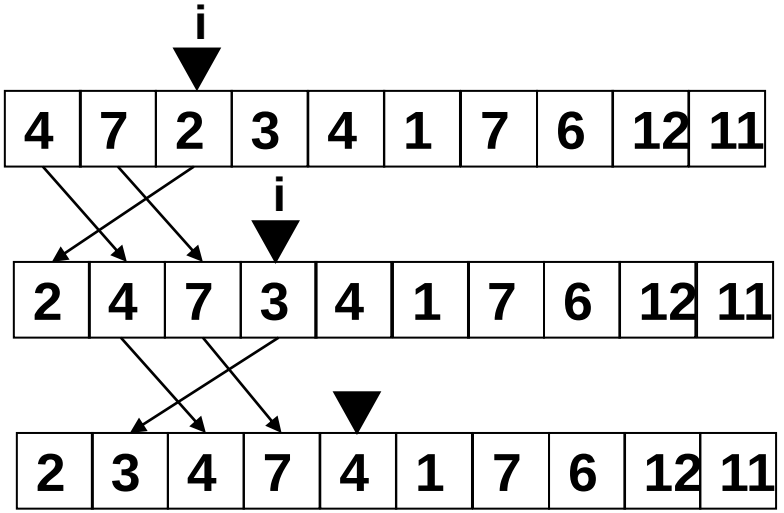
Anzahl der Vertauschungen ≈ (n - 1) ⇒ O(n)

Beide Maßzahlen sind unabhängig vom Listeninhalt

Insertion sort (1)

Einfügesortieren

- Geg: Feld von Ganzzahlen der Länge `n`: `int a[n]`
- Algorithmus:
  1. Setzte `i = 1`
  2. **Füge** a_i am geeigneten Ort in $a_0..a_i$ **ein**.
  3. Setze i = i + 1 und fahre bei Schritt 2 solange fort, bis das Ende des Feldes erreicht wird.



Insertion sort (2)

Einfügesortieren

```
/* n … number of values */
for (i = 1; i < n; i++) {
     x = values[i];
     j = i;
     while (j > 0 && values[j - 1] > x) {
          values[j] = values[j-1];
          j--;
     }
     values[j] = x;
} /* end for */
```

Günstigster Fall: (korrekt sortierte Liste)

Anzahl der Vergleiche ≈ 2* (n - 1) ⇒ O(n)

Anzahl der Zuweisungen ≈ 2 * (n-1) ⇒ O(n)

Ungünstigster Fall: (umgekehrt sortierte Liste)

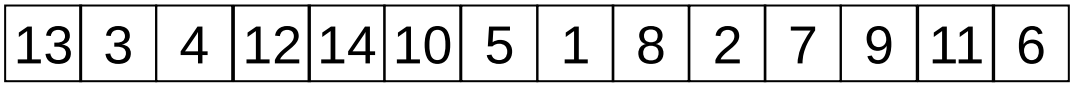
Anzahl der Vergleiche ≈ (n - 1) * n/2 ⇒ $O(n^2)$

Anzahl der Zuweisungen ≈ (n - 1) * n/2 ⇒ $O(n^2)$

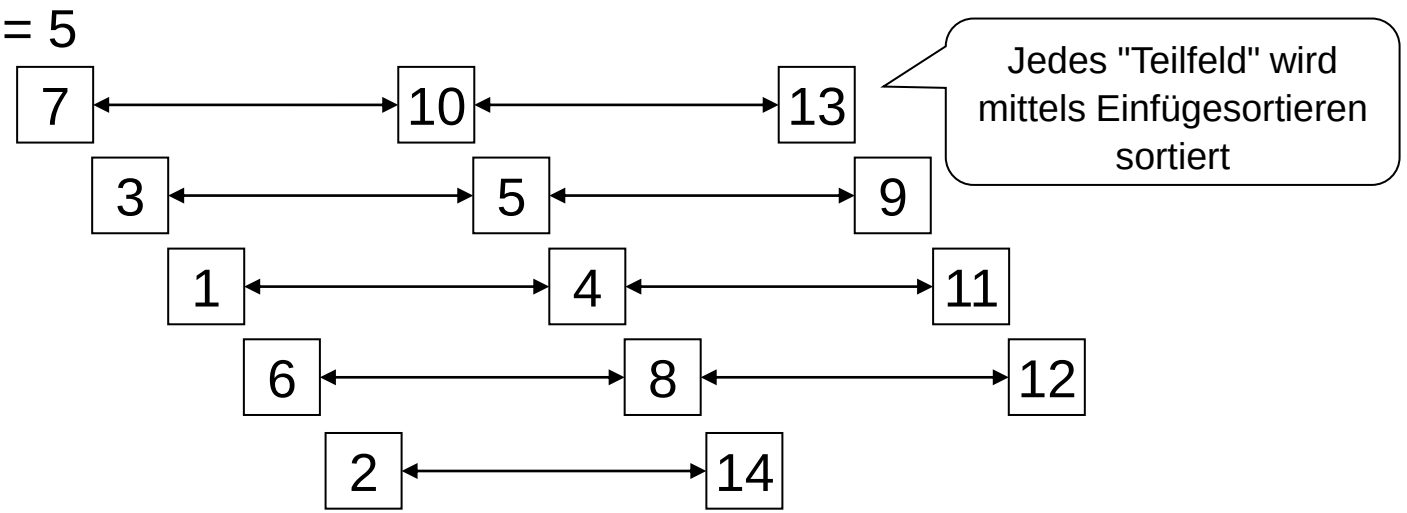
Shell sort (1)

Mehrfaches Einfügesortieren

- Idee von H.D. Shell war, den Weg des "Nach-Vorne-Wanderns" zu verkürzen
- Umsetzung der Lösungsidee durch Anpassung des Einfügesortierens, wobei die Schrittweise von `1` auf `m` verändert wird.

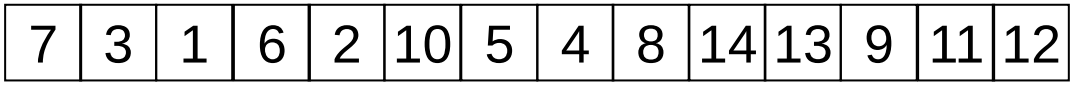


m = 5

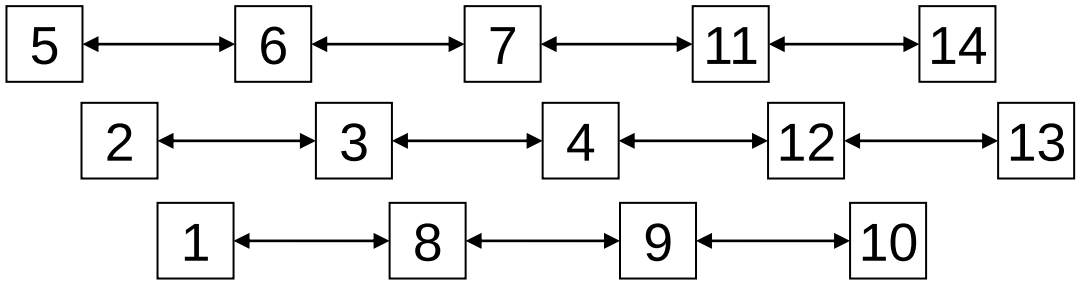


Shell sort (2)

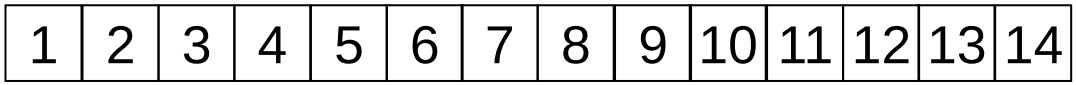
Mehrfaches Einfügesortieren



m = 3



m = 1



Shell sort (3)

Mehrfaches Einfügesortieren

```
void shellSort(int values[], int n) {
      int i = 0;
      int delta = n;
      do {
            delta = 1 + delta/3;
            for (i = 0; i < delta; i++) {
                  deltaInsertionSort(values, n, i, delta);
            }
      } while (delta > 1);
} /* end ShellSort */

void deltaInsertionSort(int values[], int n, int i, int delta) {
      int j = 0;
      int k = 0;
      int x = 0;

      j = i + delta;
      while(j < n) {
            x = values[j];
            k = j;
            while (k > 0 && values[k - delta] > x) {
                  values[k] = values[k-delta];
                  k = k - delta;
            };
            values[k] = x;

             j = j + delta;
      } /* end while */
} /* end DeltaInsertionSort */
```

Shell sort (4)

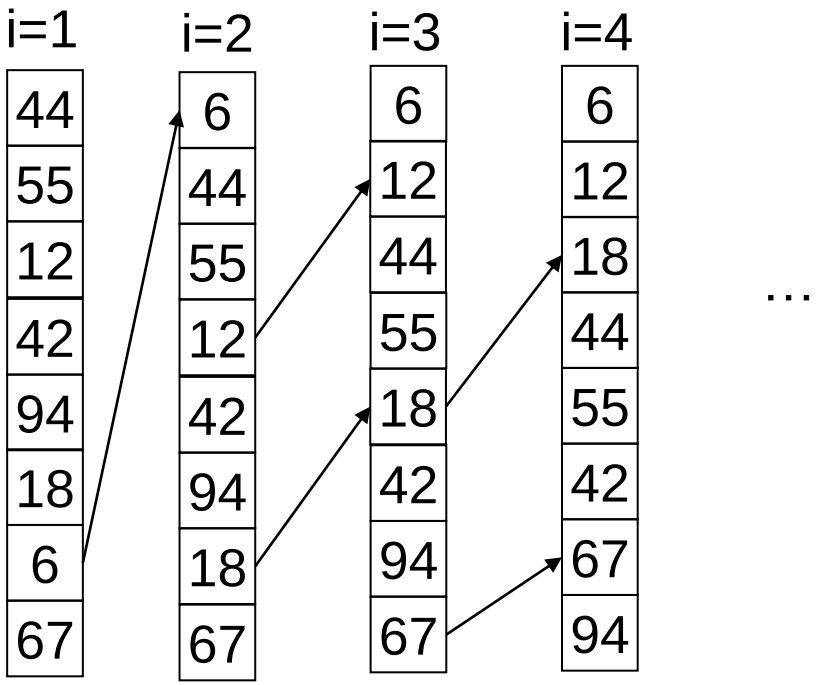
Mehrfaches Einfügesortieren

- Komplexitätsanalyse bis heute nicht abgeschlossen
  Laufzeit ≈ $O(n^{1.5})$
- Prinzipiell sehr gutes Verhalten, das durch die Wahl von `m` unwesentlich beeinflusst werden kann.
- Wahl von `m`
  - Jede absteigende Zahlenfolge
  - Knuth schlägt Fibonacci-Folge

Bubble sort (1)

Bläschensortieren

- *Bubble sort* beruht auf der Vorstellung, dass kleine Elemente ihrem "Gewicht" entsprechend wie Blasen in einer Flüssigkeit nach oben (d.h. nach vorne) steigen.



Bubble sort (2)

Bläschensortieren

```
for (i = 1; i < n; i++) {
     for (j = n - 1; j >= i; j--) {
          if (values[j - 1] > values[j]) {
               int x = values[j - 1];
               values[j - 1] = values[j];
               values[j] = x;
          }
     }
}
```

***Bubble sort* ist einfach, aber sehr** ineffizient**!**

Günstigster Fall: (korrekt sortierte Liste)

Anzahl der Vergleiche ≈ 2* (n - 1) ⇒ O(n)

Anzahl der Zuweisungen ≈ 2 * (n-1) ⇒ O(n)

Ungünstigster Fall: (umgekehrt sortierte Liste)

Anzahl der Vergleiche ≈ (n - 1) * n/2 ⇒ $O(n^2)$

Anzahl der Zuweisungen ≈ (n - 1) * n/2 ⇒ $O(n^2)$

Durchschnittlicher Fall: $O(n^2)$!!!!

Quicksort (1)

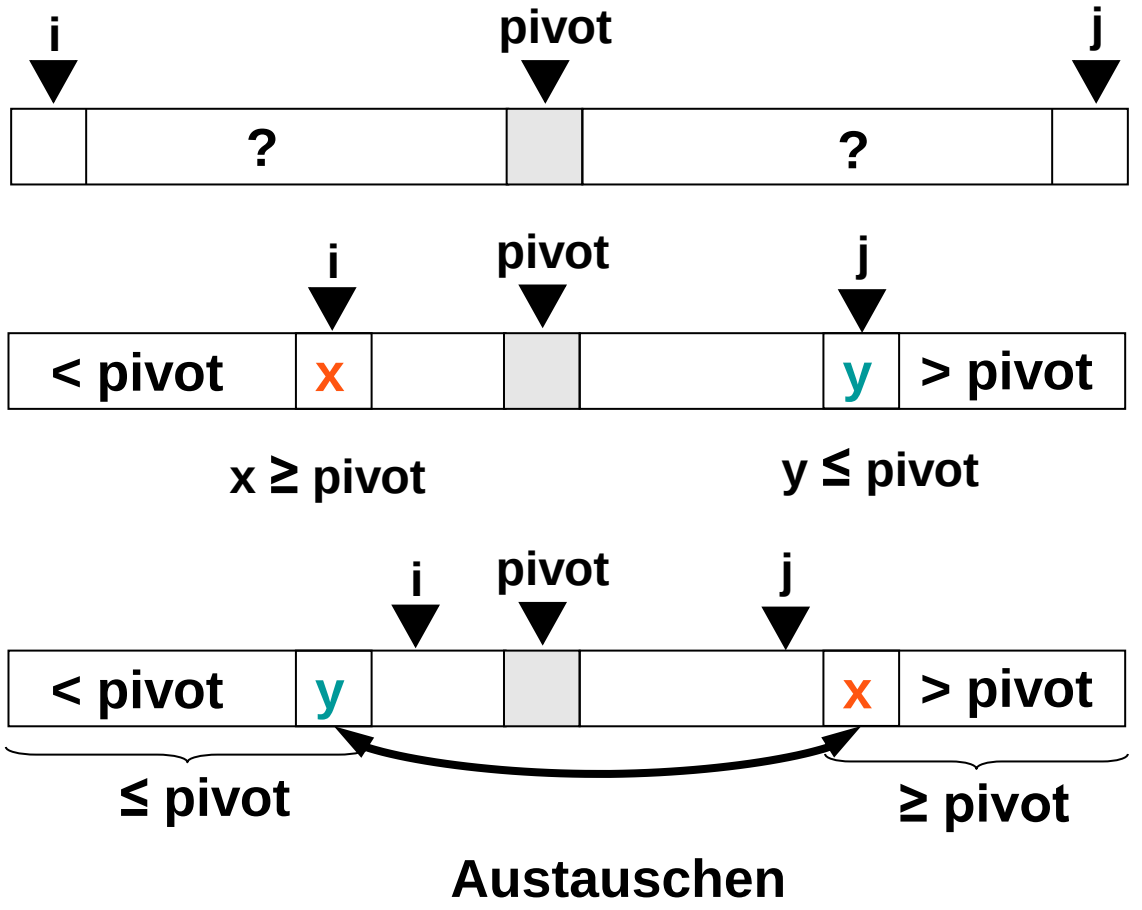
Sortieren durch Zerlegen

- *Quicksort* wurde von C.A.R. Hoare erfunden und ist einer der leistungsfähigsten Sortieralgorithmen.
- *Quicksort* beruht auf dem "Teile-und-Herrsche-Prinzip" (Devide and conquer) d.h. das Gesamtproblem wird in kleinere, einfachere Teilprobleme zerlegt. Die Gesamtlösung ergibt sich aus der Kombination der Teillösungen.
- Algorithmus:
  1. Wähle aus dem Feld ein "willkürliches" Element `pivot`.
  2. Zerlege das Feld in zwei Teilfelder, wobei im einen Teilfeld nur Werte <= `pivot` und im anderen Werte >= `pivot` enthalten sind.
  3. Wiederhole die Schritte rekursive für jedes Teilfeld bis nur noch Felder der Länge 1 betrachten werden müssen.

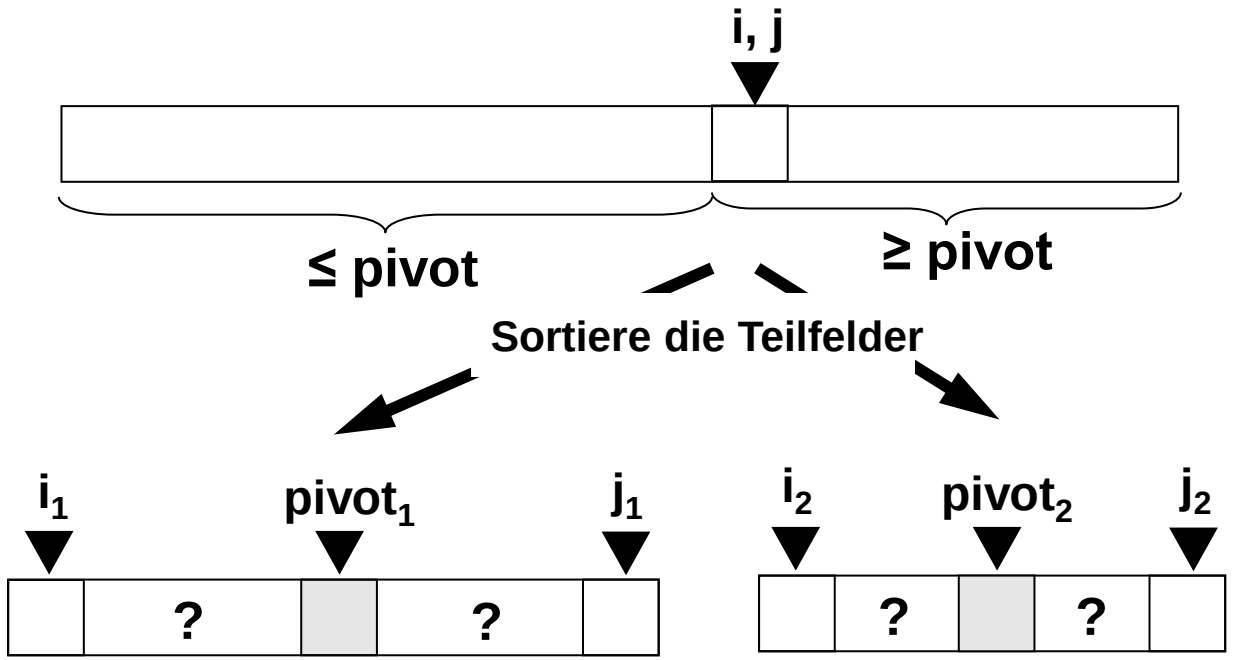
Quicksort (2)

Vorgehen

Bilde zwei Teilfelder



Quicksort (3)
Vorgehen



Quicksort (4)
Implementierung

```
void quickSort(int[] values, int m, int n) {
    if (m < n) {
        int i = m;
        int j = n;

        partition(values, ref i, ref j);

        quickSort(values, m, j);
        quickSort(values, i, n);
    }
}
```

Quicksort (5)
Implementierung

```
void Partition(int[] values, ref int i, ref int j) {
     int pivot = values[(i + j) / 2];

     while(i <= j) {
          /* from left to right */
          while(values[i] < pivot ) {
               i++;
          }

          /* from right to left */
          while ( values[j] > pivot ) {
               j--;
          }

          if ( i <= j) {
               /* swap values */
               int temp = values[i];
               values[i] = values[j];
               values[j] = temp;

               i++;
               j--;
          }
     }
}
```

Quicksort (6)
Bewertung

- günstigster Fall: das Feld wird jeweils halbiert. Damit hat der Baum der rekursiven Aufrufe eine minimale Höhe `ld(n)`. Auf jeder Ebene werden `n` Elemente untersucht:
  - Durchläufe: ld (n)
  - Vergleiche: n * ld (n)
- Ungünstigster Fall: das Feld ist bereits sortiert:
  - Durchläufe: n
  - Vergleiche: n * n
- Durchschnittlicher Fall: *1.4n * ld (n)*