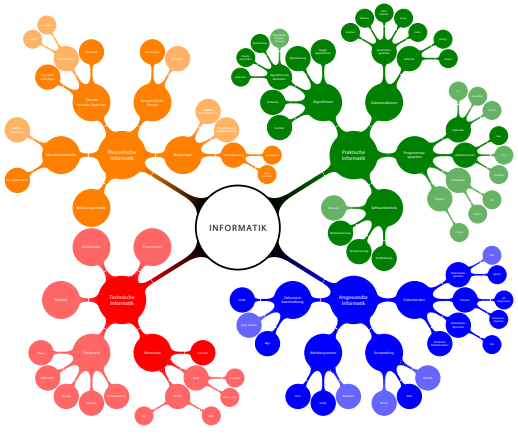
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 20

Schnelle Sortierverfahren und Rekursion

Ein Terabyte sortieren

Vorlesung Einführung in die Informatik 1 vom 14. Januar 2014 von Till Tantau



Lernziele von Kapitel 20

1. Merge-Sort verstehen und implementieren können
2. Quick-Sort verstehen und implementieren können

Gliederung von Kapitel 20

Wiederholung:
Definition des allgemeinen Sortierproblems.

Das allgemeine Sortierproblem

Eingabe Array von Objekten, die sich vergleichen lassen.

Ausgabe Eine Permutation der Objekte, so dass jedes Objekt in der neuen Reihenfolge kleiner oder gleich dem nachfolgenden ist.

Wiederholung: Wünschenswerte Eigenschaften von Sortieralgorithmen.

Folgende Eigenschaften sind bei Sortieralgorithmen besonders wünschenswert:

1. Ein Verfahren ist *stabil*, falls sich die Reihenfolge von gleichen Elementen nicht ändert.
2. Ein Verfahren ist *in-place*, falls es lediglich eine kleine Menge extra Speicher benötigt, falls es also keine Kopie des Arrays benötigt.
3. Das Verfahren sollte mit möglichst wenig Vergleichen, Vertauschungen und Verschiebungen auskommen.

Was haben wir bis jetzt erreicht?

- Wir kennen drei Sortierverfahren:
  1. Bubble-Sort,
  2. Selection-Sort und
  3. Insertion-Sort.
- Sie haben alle bei zufälligen Daten eine Laufzeit von $O(n^2)$.
- Das theoretische Minimum liegt aber bei $O(n \log n)$.

Brauchen wir schnellere Sortieralgorithmen?

Beispiel (Sortieren mit Bubble-Sort)

- Es soll ein Molekül mit 10.000 Atomen visualiert werden. Dazu muss (unter anderem) 30 mal pro Sekunde die Liste der Atome sortiert werden.
- Bei Bubble-Sort benötigt dies grob $10.000^2/2 = 50.000.000$ Vergleiche pro Sortierung.
- Bräuchte ein Vergleich nur einen Taktzyklus, so wäre eine 1.5GHz CPU *nur mit Sortieren beschäftigt*.

Beispiel (Schnelles Sortieren)

- Betrachten wir dasselbe Problem, nur wird nun ein Algorithmus verwendet, der $O(n \log n)$ Zeit benötigt.
- Das macht dann grob $10.000 \log_2 10.000 \approx 133.000$ Vergleiche pro Sortierung.
- Bei 30 Bildern pro Sekunde muss die CPU also grob 4.000.000 Vergleiche schaffen, wofür sie nur *wenige Millisekunden braucht*.

Der Merge-Sort-Algorithmus

Idee

- Wir gehen rekursiv vor: Erst sortieren wir die erste Hälfte, dann die zweite Hälfte.
- Nun müssen wir die beiden sortierten Hälften zu einem sortierten Array zusammenfügen.

Hauptteil von Merge-Sort


```
static void mergeSort (int[] array)
{
  if (array.length > 1)
  {
    // Kopiere erste Hälfte von array nach first
    int[] first  = new int [array.length/2];
    for (int i = 0; i<first.length; i++) {
      first[i] = array[i];
    }

    // Kopiere Rest (zweite Hälfte) von array nach second
    int[] second = new int [array.length-first.length];
    for (int i = 0; i<second.length; i++) {
      second[i] = array[i+first.length];
    }

    // Sortiere first
    mergeSort(first);

    // Sortiere second
    mergeSort(second);

    // Verschmelze (merge) der sortierten Listen
    merge(first,second,array);
  }
}
```

5-Minuten-Aufgabe

Zur Übung

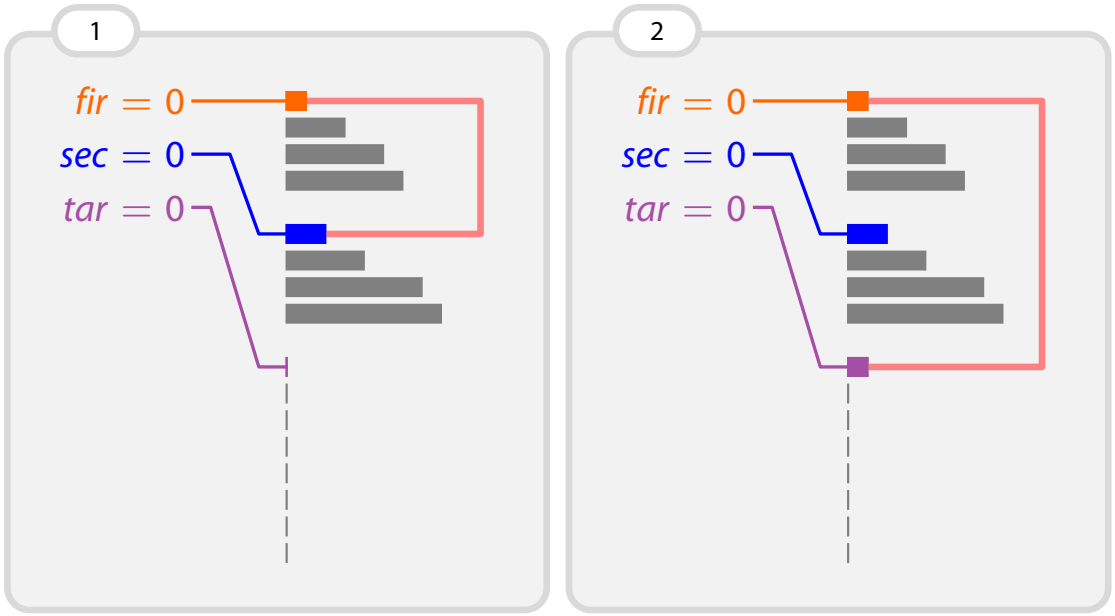
Es wird `mergeSort` aufgerufen mit dem Array `{3,9,5,0,11,2}`.
Geben Sie die genauen Inhalte der Arrays `array`, `first` und `second` zum Zeitpunkt jeder der Kommentarzeilen an.

Die `merge`-Methode.


```
static void merge(int[] first, int[] second, int[] array)
{
  int first_pos  = 0;
  int second_pos = 0;
  int target_pos = 0;

  while (   first_pos  < first.length
         || second_pos < second.length)
  {
    if (first_pos < first.length &&
        (second_pos >= second.length ||
         first[first_pos] <= second[second_pos]))
    { // Wähle Element aus first
      array[target_pos] = first[first_pos];
      target_pos++;
      first_pos++;
    }
    else
    { // Wähle Element aus second
      array[target_pos] = second[second_pos];
      target_pos++;
      second_pos++;
    }
  }
}
```

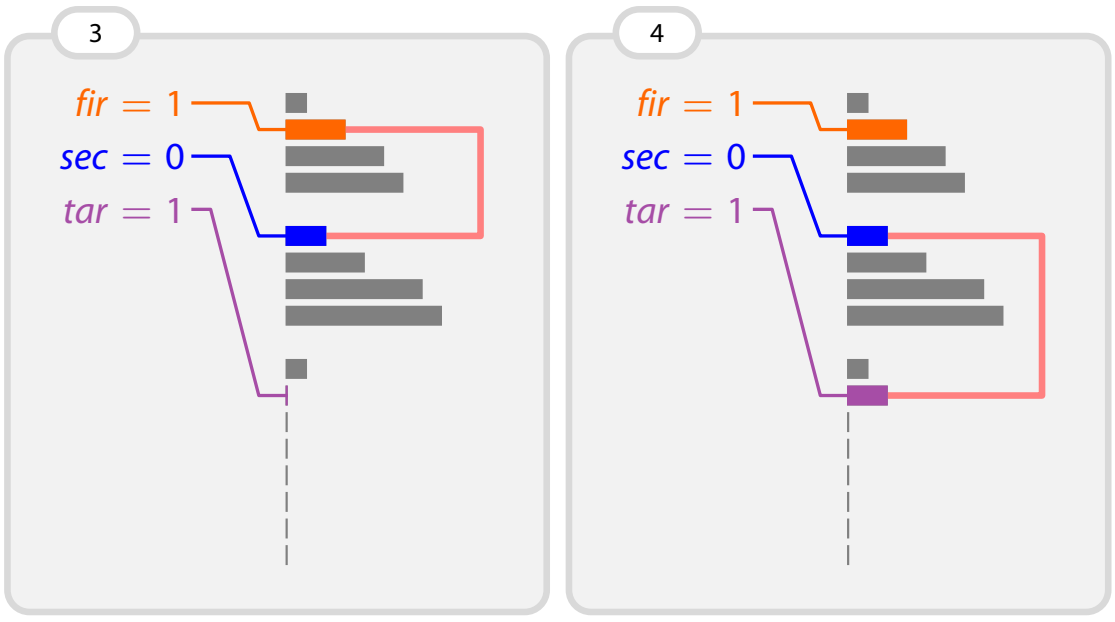
Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



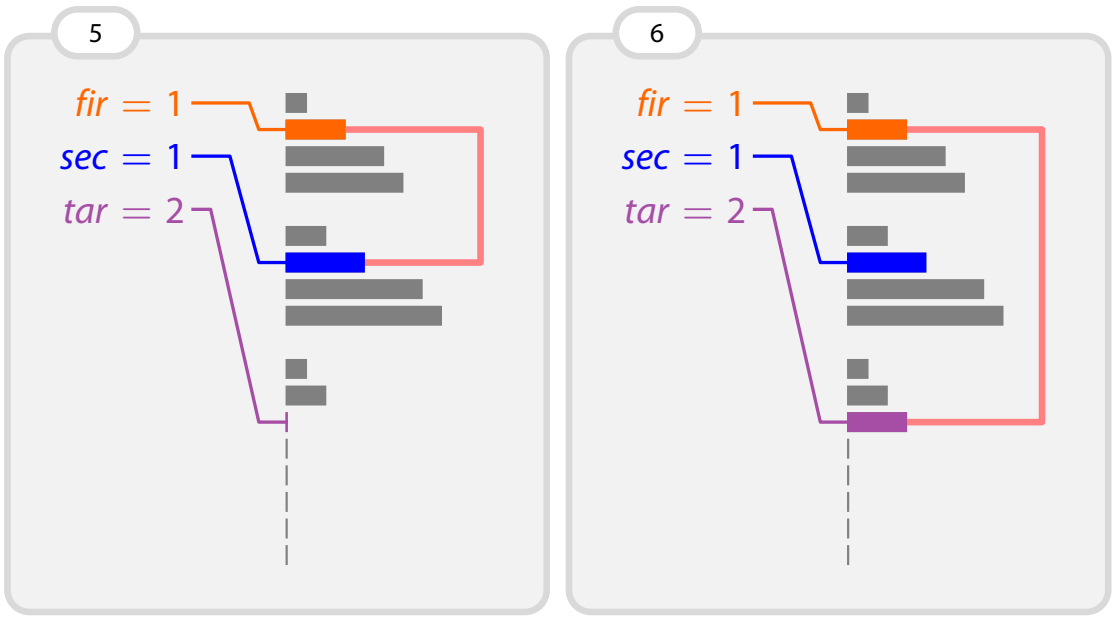
Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



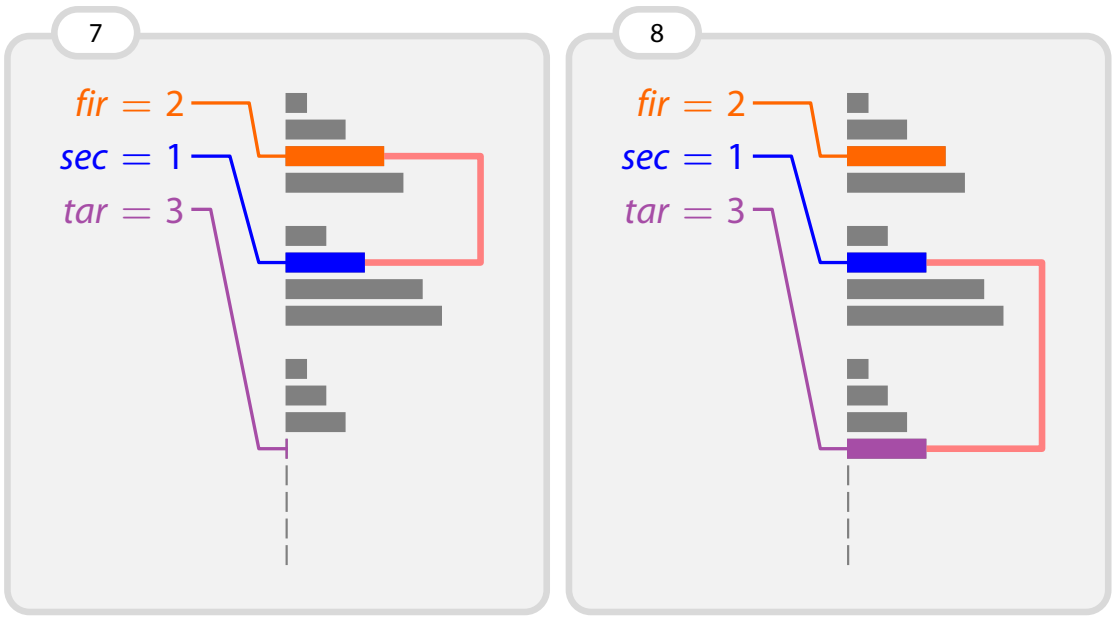
Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



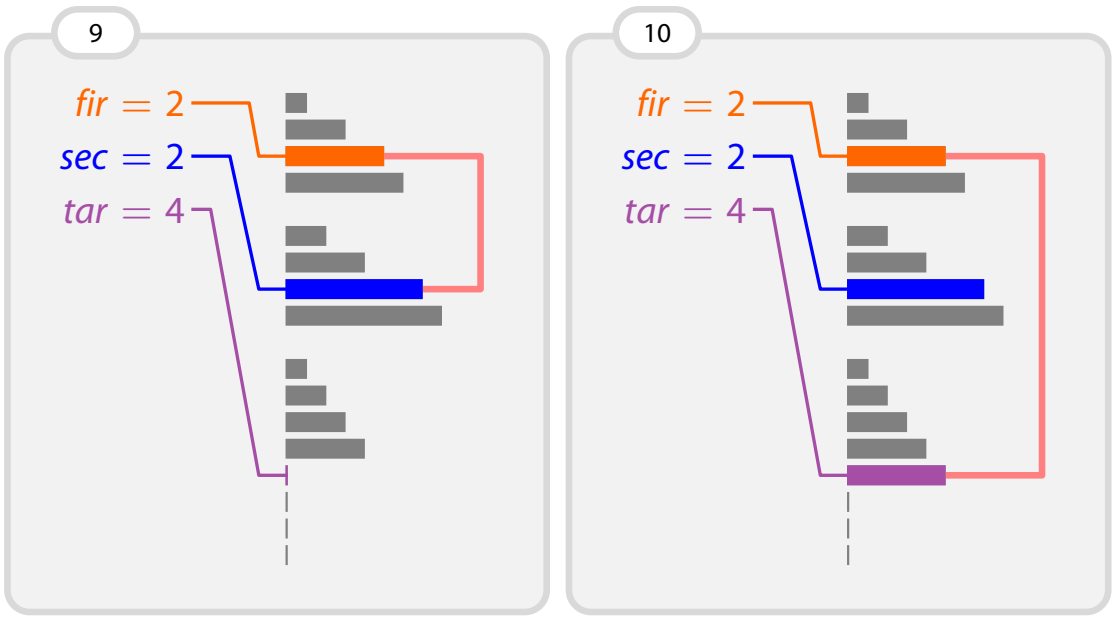
Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



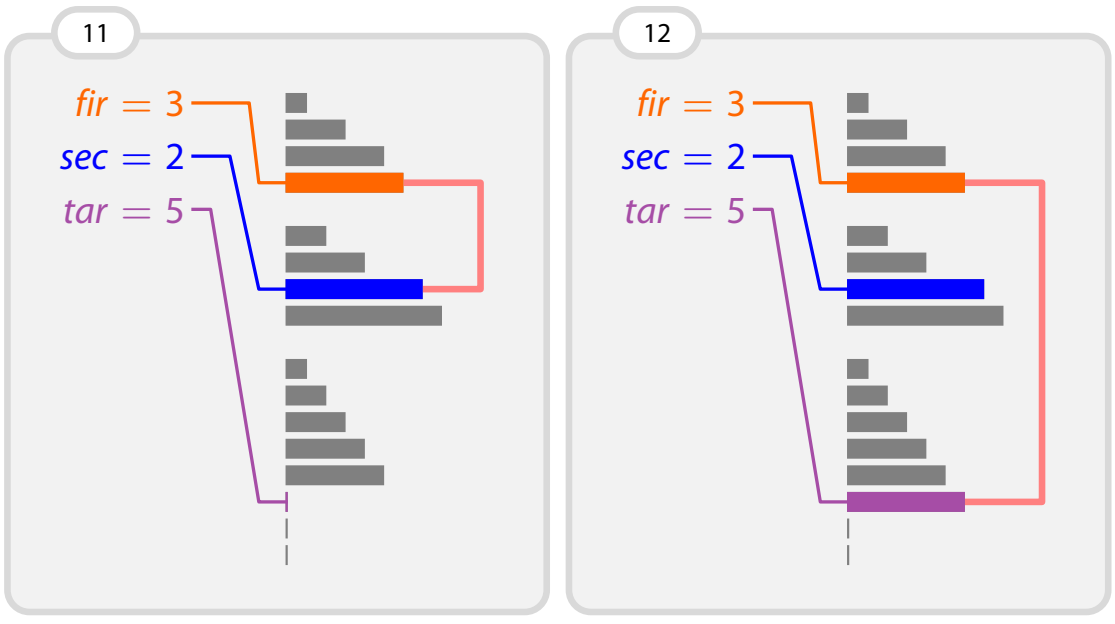
Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



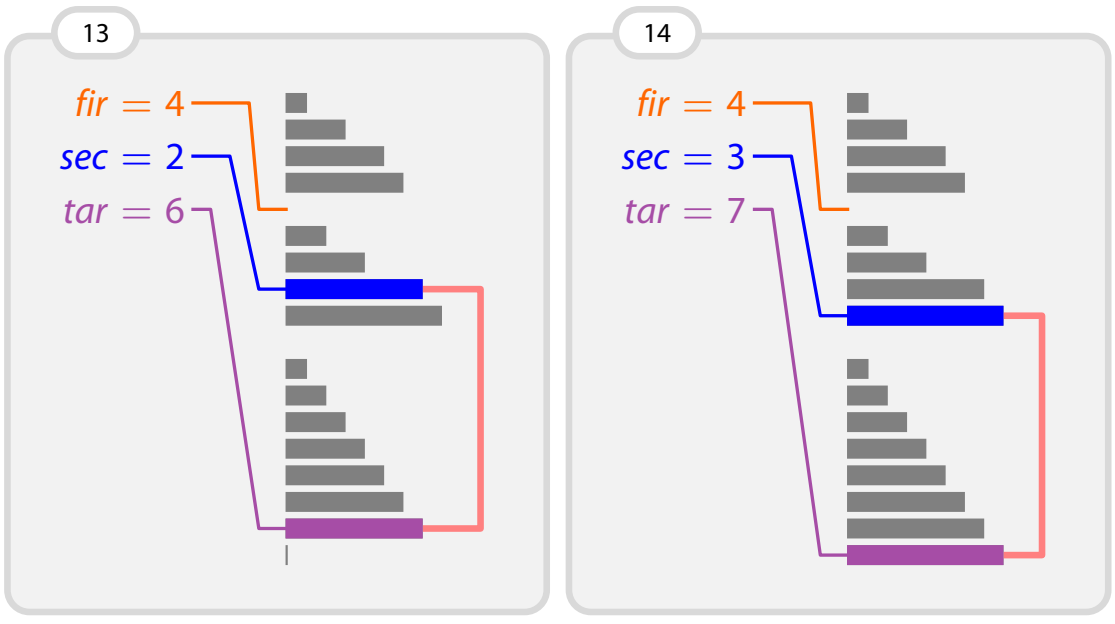
Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



Beispielablauf einer Verschmelzung

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



Wie schnell ist Merge-Sort?

Beitrag der ersten Rekursionsstufe

- Zunächst werden n Zahlen in $n/2$ und nochmal $n/2$ Zahlen aufgeteilt.
- Das dauert $n/2 + n/2 = n$ Schritte.
- Am Ende werden die Arrays verschmolzen.
- Das dauert nochmal $n/2 + n/2 = n$ Schritte.

Wie schnell ist Merge-Sort?

Beitrag der zweiten Rekursionsstufe

- Beide Arrays der Größe $n/2$ werden in zwei Arrays der Größe $n/4$ aufgeteilt.
- Das dauert $2(n/4 + n/4) = n$ Schritte.
- Am Ende werden je zwei Arrays wieder verscholzen.
- Das dauert $2(n/4 + n/4) = n$ Schritte.

Wie schnell ist Merge-Sort?

Beitrag der dritten Rekursionsstufe

- Die vier Arrays der Größe $n/4$ werden in je zwei Arrays der Größe $n/8$ aufgeteilt.
- Das dauert $4(n/8 + n/8) = n$ Schritte.
- Am Ende werden je zwei Arrays wieder verscholzen.
- Das dauert $4(n/8 + n/8) = n$ Schritte.

Die Geschwindigkeit von Merge-Sort.

- Auf jeder Rekursionsstufe werden $2n$ Schritte gemacht.
- Es gibt $\log_2 n$ Rekursionsstufen.
- Das macht $2n \log_2 n = O(n \log n)$ Schritte insgesamt.

Vor- und Nachteile von Merge-Sort

Vorteile

+ Garantierte Laufzeit von $O(n \log n)$.
+ Stabil.
+ Gut auch für Listen statt Arrays einzusetzen.

Nachteile

– Nicht in-place (braucht doppelten Speicher).

Der Quick-Sort-Algorithmus

Idee

- Wir gehen wieder rekursiv vor.
- Diesmal wählen wir zunächst ein Element aus, das wir *Pivot-Element* nennen,
  es ist idealerweise der Median.
- Dann sorgen wir durch Vertauschungen dafür, dass folgendes gilt:
  - Links stehen alle Elemente, die kleiner als das Pivot-Element sind.
  - Rechts stehen alle Elemente, die größer als das Pivot-Element sind.
- Dann sortieren wir rekursiv die linken und die rechten Elemente.

Offene Fragen zum Quick-Sort-Algorithmus

1. Was ist das Pivot-Element?
2. Wie »sorgt man dafür«, dass die Eigenschaften erfüllt sind?
3. Was sind die linke und rechte Seite?

Was ist das Pivot-Element?

- Das Pivot-Element ist *idealerweise* der *Median* der Liste.
- Leider kennt man den Median am Anfang noch nicht.
- Deshalb gibt es verschiedene Strategien, das Pivot-Element zu wählen:
  - Man nimmt das erste Element (sehr dumm).
  - Man nimmt das mittlere Element (sehr klug).
  - Man nimmt ein zufälliges Elment (nicht schlecht).
  - Man führt eine so genannte Pivot-Suche durch (wer's mag).

Wie »sorgt man dafür«, dass die kleinen Element links sind?

- Man tauscht zunächst das Pivot-Element ans Ende, damit es nicht stört.
- Man hält in einer Variable `end_of_left_side` das rechte Ende der linken Seite.
- Man läuft nun über alle Elemente des Arrays.
- Immer, wenn man ein Element findet, das kleiner als das Pivotelement ist, tauscht man es an das Ende der linken Seite, die dann um dieses Element größer.

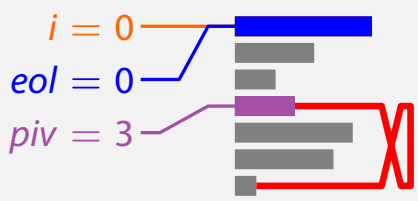
Das Quick-Sort-Programm.


```
static void quickSort (int[] array, int start, int end)
{
  if (start < end) {
    int pivot = (start + end)/2;
    int end_of_left_side = start;

    swap (array, pivot, end);
    pivot = end;
    for (int i = start; i < end; i++) {
      if (array[i] < array[pivot])
      {
        swap(array, end_of_left_side, i);
        end_of_left_side++;
      }
    }
    swap (array, end_of_left_side, pivot);

    quickSort(array,start,end_of_left_side-1);
    quickSort(array,end_of_left_side+1,end);
  }
}
```

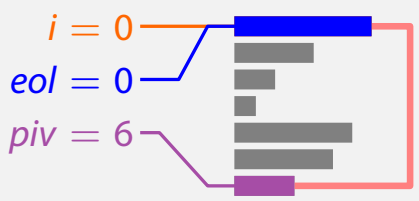
Beispielablauf des Quicksorts vor der Rekursion

Umsortierung von `array == {7,4,2,3,6,5,1}` mit `start == 0` und `end == 6`.

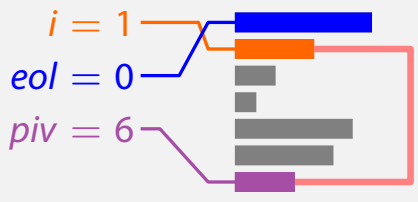
1



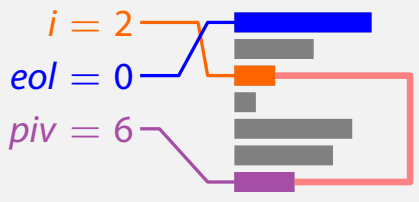
2



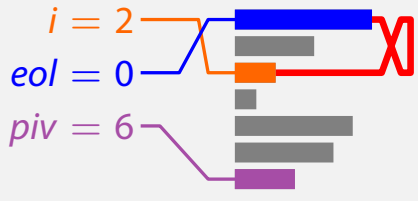
3



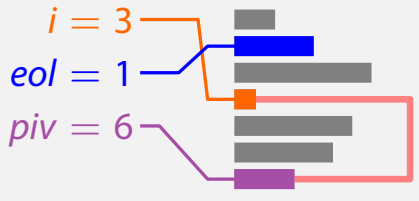
4



5



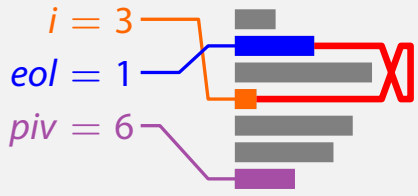
6



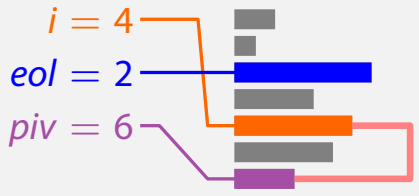
Beispielablauf des Quicksorts vor der Rekursion

Umsortierung von `array == {7,4,2,3,6,5,1}` mit `start == 0` und `end == 6`.

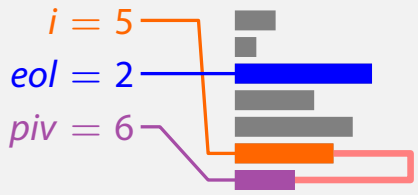
7



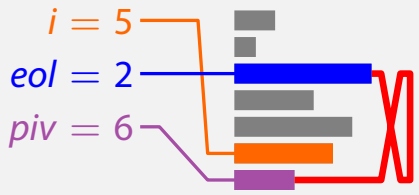
8



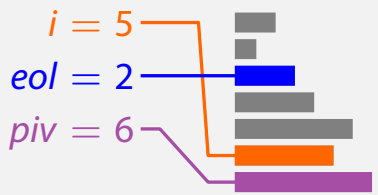
9



10



11



Die Geschwindigkeit von Quick-Sort.

- Auf jeder Rekursionsstufe werden n Schritte gemacht.
- Leider ist unklar, wie viele Rekursionsstufen es gibt:
  Dies hängt davon ab, wie nahe das Pivot-Element am Median liegt.
- Bei *sortierten*, *fast sortierten* und bei *umgekehrt sortierten* Daten ist das Pivot-Element der Median und
- bei *zufälligen* Daten ist das Pivot-Element nahe am Median.
- *Quick-Sort hat in aller Regel eine Laufzeit von* $O(n \log n)$.

Vor- und Nachteile von Quick-Sort

Vorteile

+ In der Regel um den Faktor 2 schneller als Merge-Sort.
+ In-place.
+ Liefert viele schöne Fragestellungen für Theoretiker.
+ In der Praxis oft der schnellste Algorithmus.

Nachteile

– Nicht stabil.
– Worst-case $O(n^2)$.

Kann man in linearer Zeit sortieren?

Satz

Jeder Sortieralgorithmus, der lediglich Vergleiche zum Sortieren nutzt, benötigt mindestens $n \log_2 \frac{n}{e}$ *Vergleiche, um n Werte zu sortieren.*

- Man kann also *nicht in linearer Zeit* sortieren.
- Sortieralgorithmen wie *Merge-Sort* sind also, bis auf einen kleinen Faktor, *optimal*.
- Wer *trotzdem schneller* werden will, muss einen Trick anwenden: Man vermeidet Vergleiche.

Der Distribution-Sort-Algorithmus

Idee

- Nehmen wir an, wir wissen, die Werte im Array liegen zwischen 0 und $k - 1$.
- Dann zählen wir für jeden dieser möglichen Werte, wie oft er vorkommt.
- Dann durchlaufen wir alle möglichen Werte und schreiben so viele Elemente wie nötig in das Array.

Das einfache Distribution-Sort-Programm


```
static void distributionSort (int[] array, int k)
{
  // Annahme: Alle Werte in array liegen zwischen 0 und k
       - 1.
  int[] values = new int[k];
  for (int i = 0; i<values.length; i++) {
    values[i] = 0;
  }

  for (int i = 0; i<array.length; i++) {
    values[array[i]] ++;
  }

  int pos = 0;
  for (int i = 0; i<values.length; i++) {
    for (int j = 0; j<values[i]; j++) {
      array[pos] = i;
      pos++;
    }
  }
}
```

Vor- und Nachteile von Distribution-Sort

Vorteile

+ Braucht nur $O(k + n)$ Schritte.
+ Sehr einfach aufgebaut.
+ Erweiterungen können stabil gemacht werden.
+ Wenn k bekannt und klein ist, ist ein Geschwindigkeitsvorteil vom Faktor 100 gegenüber sehr guten Standardalgorithmen möglich.

Nachteile

– k muss bekannt sein (kann umgangen werden).
– Es müssen Zahlen sortiert werden (kann umgangen werden).

Zusammenfassung

Merge-Sort

Algorithmus Teile den Array in zwei Hälften. Sortiere diese rekursiv. Verschmelze die sortierten Ergebnislisten.

Laufzeit $O(n \log n)$.

Vorteile Garantierte Laufzeit, stabil.

Nachteile Nicht in-place.

Quick-Sort

Algorithmus Wähle ein Pivot-Element. Tausche Elemente so, dass im linken Teil alle Elemente kleiner als das Pivot-Element sind, im rechten alle größer. Sortire rekursiv beide Teile.

Laufzeit $O(n \log n)$ im Schnitt, $O(n^2)$ im schlimmsten Fall.

Vorteile In der Praxis schneller als Merge-Sort, in-place.

Nachteile Sehr schlechte Laufzeit im schlimmsten Fall.