Algorithmen auf Reihungen: Suchen und Sortieren

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

http://www.pst.ifi.lmu.de/lehre/SS06/info2/

Ziele

- Wissen über Prozeduren und deren Spezifikation vertiefen
- Grundlegende Such- und Sortieralgorithmen auf Reihungen kennen lernen

Prozeduren (statische Methoden)

Prozeduren dienen zur **Abstraktion von Algorithmen**.

- Durch **Parametrisierung** wird von der Identität der Daten abstrahiert: Eingabedaten können an die speziellen aktuellen (Parameter-) Werte angepasst werden.
- Durch **Spezifikation** des (Ein-/Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert: Vorteile sind
  - **Örtliche Eingrenzung (Locality):** Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden ohne die Implementierungen anderer Abstraktionen kennen zu müssen.
  - **Änderbarkeit (Modifiability):** Jede Abstraktion kann reimplementiert werden ohne dass andere Abstraktionen geändert werden müssen.

In Java werden **Prozeduren** durch **statische Methoden** realisiert.

Achtung: Statische Methoden sind keine „echten“ Prozeduren sondern eine **Implementierung von Prozeduren** in einer OO-Sprache!

Suche nach einem Element in einer geordneten Reihung

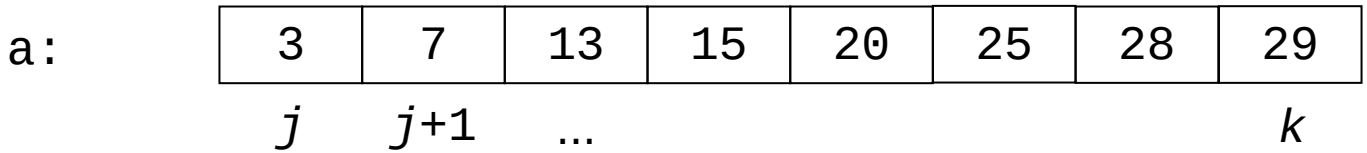
```
/**
*  determines whether the specified element is an element of the
*  specified ordered array.
*  @param elem: the element, a: the array.
*  pre: a is totally ordered in ascending order, i.e.
*       for each non-negative i < a.length-1 : a[i] <= a[i+1] .
*  @return the boolean value true if the specified element elem is an
*  element of the specified array; otherwise return false.
*  post: result == true if, and only, if
*        there exists i < a.length: elem == a[i].
*/
public static boolean searchBinary(int elem, int[] a)
```

Spezifikation von Prozeduren

- Die **Spezifikation einer Prozedur** besteht aus:
  - **Prozedurkopf**
    bestehend aus Name, Parameterliste, Rückgabetyp und Sichtbarkeit
  - **(Kommentar mit) Verhaltensbeschreibung**
    bestehend aus
    - **allgemeiner Beschreibung des Verhaltens**
    - **möglicherweise Angaben zur Implementierung und Komplexität**
    - **Vor- und Nachbedingungen**
    - **Beschreibung der Parameter**
    - **Beschreibung des Ergebnisses**
    - **Beschreibung des Verhaltens bei Ausnahmen (später)**

Implementierung: Binäre Suche eines Elements *e* in einer geordneten Reihung

Sei *a* ein geordnete Reihung mit den Grenzen *j* und *k*, d.h. $a[i] \leq a[i+1]$ für $i=j,...,k$; also z.B.:



Algorithmus:

Um den Wert *e* in *a* zu suchen, teilt man die Reihung in der Mitte und vergleicht *e* mit dem Element in der Mitte:

- Ist $e < a[mid]$, so sucht man weiter im linken Teil $a[j],..., a[mid-1]$.
- Ist $e = a[mid]$, hat man das Element gefunden.
- Ist $e > a[mid]$, so sucht man weiter im rechten Teil $a[mid+1],..., a[k]$.

Binäre Suche eines Elements *e* in einer geordneten Reihung

```
public static boolean searchBinary(int e, int[] a) {

  int j = 0, k = a.length-1, mid;  //linke, rechte Grenze, Index der Mitte
  boolean found = false;
  while (j <= k & !found) { //solange a nicht leer und e nicht gefunden
      mid = (j + k) / 2;     //Berechnung der Mitte
      if (e < a[mid]){       //falls e kleiner als das mittlere Element
          k = mid - 1;       //setze rechte Grenze unterhalb der Mitte
    } else {                 //sonst
          if (e == a[mid]){ //falls e gleich mittlerem Element
            found = true;    // e ist in a gefunden
       } else {              //sonst
            j = mid + 1;} } } //setze linke Grenze oberhalb der Mitte

   return found;             //e gefunden genau dann, wenn found == true
}
```

Komplexität von Algorithmen

Der **Zeitbedarf** einer Prozedur wird in Abhängigkeit von der Größe der Eingabe berechnet; zur Vereinfachung wird nur die **Größenordnung** betrachtet (**O-Notation**, siehe Info 1).

- Beim **Zeitbedarf** zählen wir (der Einfachheit halber)
  - die **Anzahl der (zeitintensiven) Operationen**
    Z.B. bei der binären Suche in einer Reihung die Anzahl der notwendigen Vergleiche:
    Sei n die Länge der gegebenen Reihung a;
    `searchBinary(e, a)` benötigt $\log_2 n$ Vergleiche, d.h.
    Die Zeitkomplexität ist logarithmisch (von der Größenordnung $O(\log_2 n)$).

- **Speicherplatzbedarf** siehe später
  Der Speicherplatzbedarf von `searchBinary(e,a)` ist konstant (d.h. $O(1)$).

Suche in einer geordneten Reihung (Verbesserte Beschreibung)

```
/**
*  determines whether the specified element is an element of the
*  specified ordered array.
*  Uses binary search and has logarithmic time complexity.
*  @param elem: the element, a: the array.
*  pre: a is totally ordered in ascending order, i.e.
*       for each non-negative i < a.length-1 : a[i] <= a[i+1] .
*  @return the boolean value true if the specified element elem is an
*  element of the specified array; otherwise return false.
*  post: result == true if ,and only, if
*        there exists a non-negative i < a.length: elem == a[i].
*/
public static boolean searchBinary(int elem, int[] a)
```

Suche in einer geordneten Reihung

Mit Zusicherungen:

```
public static boolean searchBinary(int e, int[] a) {
  assert isOrdered(a) : „Vorbedingung“;
  int j = 0, k = a.length-1, mid;  //linke, rechte Grenze, Index der Mitte
  boolean found = false;
  while (j <= k & !found) {  //solange a nicht leer und e nicht gefunden
      mid = (j + k) / 2;     //Berechnung der Mitte
      if (e < a[mid]){       //falls e kleiner als das mittl. Element
          k = mid - 1;       //setze rechte Grenze unterhalb der Mitte
      } else { assert e != a[mid];  //sonst
          if (e == a[mid]){ //falls e gleich mittlerem Element
             found = true;   // e ist in a gefunden
       }else {assert e > a[mid];    //sonst
             j = mid + 1; }}} //setze linke Grenze oberhalb der Mitte
   assert found == isElem(e,a): „Nachbedingung: ...“;
   return found;             //e gefunden genau dann, wenn found == true
}
```

entspricht Suche in ungeordneter Reihung

Prüfung auf Totale Ordnung

wobei die Prozedur `isOrdered` folgendermaßen definiert ist:

```
/**
 * determines whether the specified array is ordered in ascending order.
 * @param a: the array.
 * @return the boolean value true iff the specified array is totally ordered;
 * otherwise return false.
 * post: result == true iff
 *    for each i < a.length-1 : a[i] <= a[i+1] .
*/
public static boolean isOrdered(int[] a) {
       for (int i =0; i< a.length-1;i++)
              if (!(a[i]<= a[i+1])) {return false;}
       return true; }
```

Bemerkung: Der Zeitbedarf von `isOrdered(a)` ist linear (Größenordnung O(n)), also höher als der Zeitbedarf zur Suche. Das Einschalten von Zusicherungen ist hier also nur zum Testen sinnvoll; die Gültigkeit der Vorbedingung sollte von der Anwendung sicher gestellt (und bewiesen) werden.

Suche in einer beliebigen Reihung

und wobei `isElem` folgendermaßen definiert ist:

```
/**
* determines whether the specified element is an element of the
* specified array.
* @param elem: the element, a: the array.
* @return the boolean value true if the specified element elem is an
* element of the specified array a; otherwise return false.
* post: result == true if ,and only, if
*       there exists i < a.length: elem == a[i].
*/
 public static boolean isElem(int elem, int[] a) {
       for (int x : a)
              if (x == elem) {return true;}
       return false;}
```

Bemerkung: Der Zeitbedarf von `isElem (elem, a)` ist wie bei `isordered` linear (Größenordnung O(n)), also höher als der Zeitbedarf zur Suche.

Systematische Übersetzung von Formeln in Zusicherungen

- **Boolesche (aussagenlogische) Formeln**
  lassen sich direkt als boolesche Ausdrücke schreiben.
- **Quantoren**:

Sei P eine boolesche Formel und B ihre Übersetzung in einen booleschen Ausdruck.

- **„Für jedes x in a: P(x)“** (Beispiel: `isOrdered`)
  wird übersetzt in eine Prozedur

```
static boolean allP(type[] a) {
   for (type x : a)
        if (!B(x)) {return false}
   return true;}
```

- **„Es gibt ein x in a: P(x)“** (Beispiel: `isElem`)
  wird übersetzt in eine Prozedur

```
static boolean existsP(type[] a) {
   for (type x : a)
        if (B(x)) {return true;}
   return false;}
```

Sortieren

- Das Sortieren dient dem schnelleren Wiederfinden von Informationen.
- Beispiele für sortierte Informationsmengen:

  Lexika, Tabellen, Adressbücher, Kataloge usw.

- Man unterscheidet
  - **internes Sortieren**: Die zu sortierende Folge befindet sich im Hauptspeicher (wahlfreier Zugriff) und
  - **externes Sortieren**: Die zu sortierende Folge ist auf externe Speichermedien wie Bänder oder Platten ausgelagert (i.a. sequentieller Zugriff)

Sortierproblem

Sei A ein Alphabet oder eine (nicht unbedingt endliche) geordnete Menge von Elementen.

Sortierproblem: Gegeben sei eine Folge $v = v_0 \ldots v_{n-1}$ in A.

Ordne v so um, dass eine **aufsteigend geordnete (oder absteigend geordnete) Folge** entsteht.

Genauer:

Gesucht wird eine **Permutation**

$$\pi : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\},$$

so dass die Folge $w = v_{\pi 0} \ldots v_{\pi(n-1)}$ geordnet ist

Sortieren einer Reihung

```
/**
*  sorts the specified array into ascending order.
*  @param a: the array.
*  post: the resulting array a is ordered and has the same elements
*  as the specified array a@pre .
*/
public static void sort(int[] a)
```

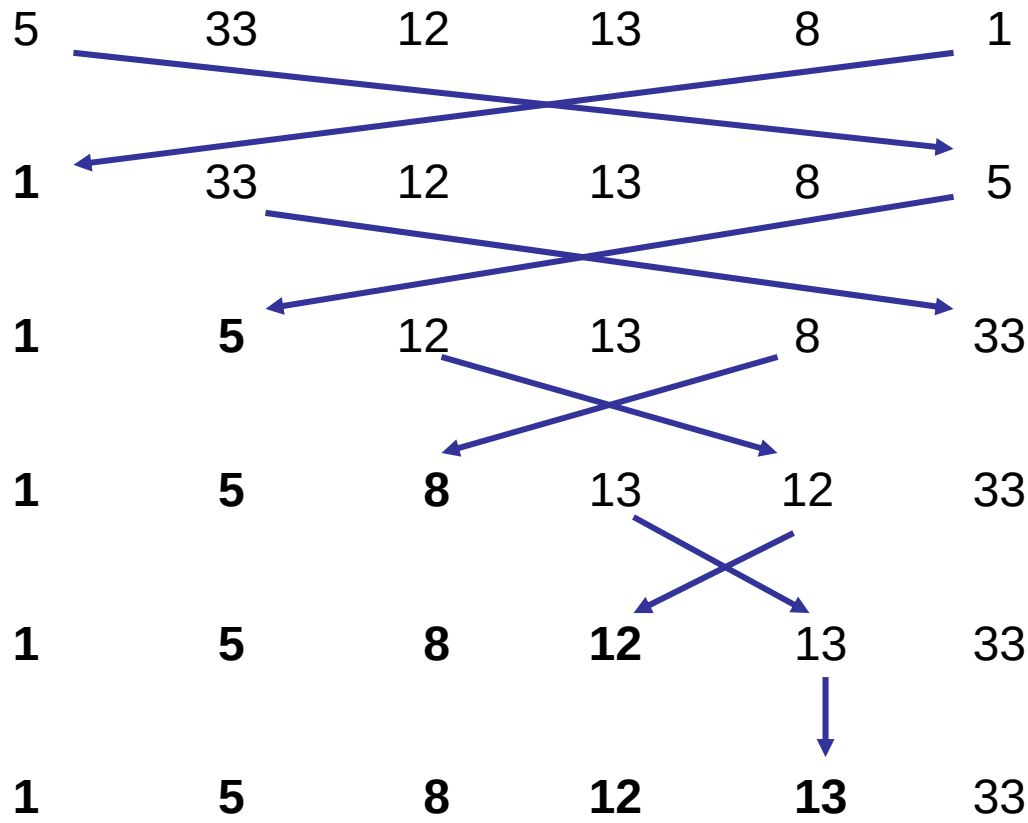
Direktes Aussuchen („Selection Sort“)

- **Idee:** Wiederholtes Durchsuchen des unsortierten Teils des Feldes nach dem **kleinsten Element**, welches dann **mit dem ersten Element** des noch unsortierten Teils **vertauscht** wird.

  Die Länge des zu sortierenden Feldes verringert sich so bei jedem Durchlauf um eins.

- **Bemerkung:** Ist bei einem Sortierschritt das nächste Element bereits am richtigen Platz, so müsste natürlich nicht vertauscht werden. Allerdings macht die zusätzliche Abfrage das Programm langsamer, so dass man diesen (i.a. relativ seltenen) Fall besser schematisch behandelt.

Beispiel: Direktes Aussuchen („Selection Sort“)



Selection Sort in Java (imperativ)

```
public static void sort (double a [])
{ for (int outer = 0; outer < a.length; outer++)
  {     int min = outer;
        /*kleinstes Element suchen, Index nach min*/
        for (int inner = outer + 1; inner < a.length; inner++)
              if (a[inner] < a[min]) min=inner;
        tausche(a, outer, min);
  }
}
```

Vertauschen zweier Elemente einer Reihung

```
public static  void tausche (double[] a, int i, int j)
{
     double hilf = a[i];
     a[i] = a[j];
     a[j] = hilf;
}
```

Aufwandsabschätzung

Da das restliche Feld stets bis zum Ende durchsucht wird, ist der Aufwand nur von der Länge n der Reihung, aber nicht vom Inhalt der Reihung abhängig.

- **Anzahl der Vergleichsoperationen:**

$$C_{max}(n) = C_{ave}(n) = \sum_{i=1}^{n-1} (n-i)$$

$$= n(n-1) - (n/2)(n-1) = (n/2)\,(n-1)\ \varepsilon\ O(n^2)$$

- **Anzahl der Vertauschungen:**

$$V_{max}(n) = V_{ave}(n) = \sum_{i=1}^{n-1} 1 = n-1\ \varepsilon\ O(n)$$

- **Zeitkomplexität:**

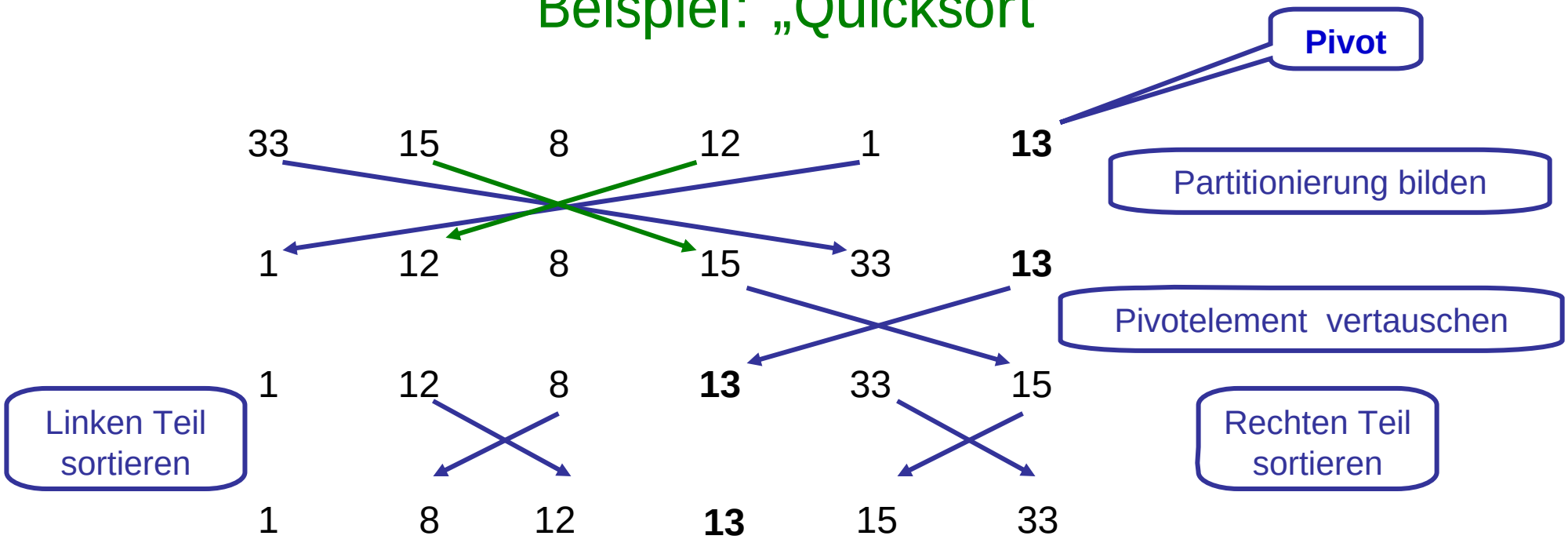
$O(n^2)$

„Quicksort (von C.A.R. Hoare)“

- Quicksort ist nach Heapsort der schnellste bekannte interne Sortieralgorithmus, da Austauschen am Effizientesten ist, wenn es über große Distanzen erfolgt.
- **Idee:**
  - Man wählt aus dem Array irgendein Element als **Pivot** (z.B. das am weitesten rechts stehende Element) aus und läuft von der linken und der rechten Grenze der Reihung solange nach innen, bis ein nicht kleineres (auf der linken Seite) und ein nicht größeres (auf der rechten Seite) Element gefunden sind. Diese beiden werden dann vertauscht.

    Man wiederholt das Ganze solange, bis sich die beiden Indizes getroffen haben. Das Pivot-Element wird dann in die Mitte getauscht (mit dem Element, das auf dem Platz des linken Index steht).
  - Jetzt hat man zwei Teilfelder, wobei das eine nur Elemente kleiner oder gleich dem Grenzwert und das andere nur Elemente größer oder gleich dem Grenzwert enthält. Diese beiden Teilfelder werden entsprechend dem oben beschriebenen Algorithmus **rekursiv** weiter sortiert, bis nur noch einelementige und damit sortierte Teilfelder vorhanden sind.

Beispiel: „Quicksort“



Quicksort rekursiv


```
static  void sort(double[] a)
{   sort(a, 0, a.length-1);
}


static  void sort(double[] a, int left, int right)
{   if (right-left> 0)        //    nur sortieren, wenn Groesse >= 2
   {    double pivot = a[right];    // Element ganz rechts
        int partition = partitionIt(a, left, right, pivot);
        sort(a, left, partition-1);   // sortiere linke Seite
        sort(a, partition+1, right);  // sortiere rechte Seite
    }
}
```

Quicksort rekursiv


```
static  void sort(double[] a)
{   sort(a, 0, a.length-1);
}


static  void sort(double[] a, int left, int right)
{   if (right-left> 0)        //    nur sortieren, wenn Groesse >= 2
   {    double pivot = a[right];    // Element ganz rechts
        int partition = partitionIt(a, left, right, pivot);
        sort(a, left, partition-1);   // sortiere linke Seite
        sort(a, partition+1, right);  // sortiere rechte Seite
    }
}
```

Partitionierung der Reihung

```
public int partitionIt(double[] a, int left, int right, double pivot)
{   int leftPtr = left;
    int rightPtr = right-1;
    while(leftPtr<=rightPtr)
    {   while( a[leftPtr] < pivot )leftPtr++; // finde nichtkleineres Elem
        while(rightPtr >=0 && a[rightPtr] > pivot) rightPtr--;
                                                  // finde nichtgroesseres Elem
        if (leftPtr<=rightPtr)
        {   if(leftPtr<rightPtr)tausche(a, leftPtr,rightPtr);
            leftPtr++;rightPtr--;        }
    }
    tausche(a, leftPtr, right);        // setze pivot an richtige Stelle
    return leftPtr;                   // return Index des pivot Elements
}
```

Verbesserungen des Quicksort-Algorithmus

- Bei kurzen Reihungen sind die einfachen Sortieralgorithmen meist besser, deshalb kann man Teilreihungen mit weniger als 10 Elementen manuell sortieren.
- Eine bessere Pivotbestimmung ergibt sich aus dem mittleren Wert der drei folgenden Elemente der Reihung: linkes Element, rechtes Element und das Element in der Mitte der Reihung.

Aufwandsabschätzung

n Vergleiche pro Iteration

n Iterationen

Schlechtester Fall: O(n * n)

z.B. wenn die Reihung schon aufsteigend sortiert ist:

1 5 8 12 13 33

Partitionieren

1 5 8 12 13 **33**

Partitionieren

1 5 8 12 **13** **33**

Partitionieren

1 5 8 **12** **13** 33

Partitionieren

. . .

Aufwandsabschätzung

- **Durchschnittlicher Fall:** $O(n * \log_2 n)$

Idee:

- Jede Partitionierung benötigt n Vergleiche,
- Bei Teilung der Reihung in jeweils ungefähr gleich große Hälften benötigt man $\log_2 n$ Iterationen.

Zusammenfassung

- Quantoren über endlichen Mengen (Reihungen) können systematisch in boolesche Prozeduren übersetzt werden. Zusicherungen, die Quantoren enthalten, sollen wegen ihrer mindestens linearen Zeitkomplexität nur zum Testen und Beweisen verwendet werden.
- Spezifikationen von Prozeduren können (und sollen häufig auch) Angaben über die Komplexität des Algorithmus enthalten.
- Suchen in einer geordneter Reihung benötigt logarithmische Zeitkomplexität.
- Sortieren durch Einfügen ist ein Sortieralgorithmus mit quadratischer Zeit- und konstanter Speicherplatzkomplexität.
- Quicksort hat im Mittel die Zeitkomplexität $n * \log_2 n$. Im schlechtesten Fall ist es n^2.