Grundlegende Sortieralgorithmen

Prof. Dr. Christian Böhm

in Zusammenarbeit mit
Gefei Zhang
http://www.dbs.ifi.lmu.de/Lehre/NFInfoSW

Ziele

- Grundlegende Sortieralgorithmen auf Reihungen kennen lernen

Klassifizierung

- 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)

Sortieren in Java

Man kann Sortierverfahren in einem imperativem oder einem objekt-orientierten Stil programmieren.

- **Imperativ** wählt man eine Klassenmethode und übergibt die zu sortierende Reihung als formalen Parameter, daraus resultieren folgende Methodenköpfe:

```
static void sort (int[] a)
```

bzw.

```
static void sort (Object[] a)
```

Sortieren in Java

- **Objekt-orientiert** steht die zu sortierende Reihung in einem Attribut und man übergibt (nur) das aktuelle Objekt; man erhält folgendes Schema für eine Sortierklasse:

```
class SortIntOO =
{    int[] arr;

    /* Gibt die aktuelle Reihung zurück
    **/
    int[] getArray () {  return arr;   }

    /* Vertauscht die Elemente der Reihung mit den Indizes i und j
    **/
    void tausche(int i, int j)
    {    int hilf = arr[i]; arr[i] = arr[j]; arr[j] = hilf;
    }
    /*Sortiert die Reihung in aufsteigender Reihenfolge
    **/
    void sort(){...}
}
```

Drei typische Sortieralgorithmen

- Direktes Aussuchen,
- Bubblesort,
- Quicksort.

Sortierproblem

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

Eine Folge $v = v_1 \ldots v_n$ heißt **geordnet**, falls $v_i <= v_{i+1}$ für i = 1, ... , n-1.

Sortierproblem: Gegeben sei eine Folge $v = v_1 \ldots v_n$ in A.

Ordne v so um, dass eine geordnete (oder invers geordnete) Folge entsteht.

Genauer:

Gesucht wird eine **Permutation**

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

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

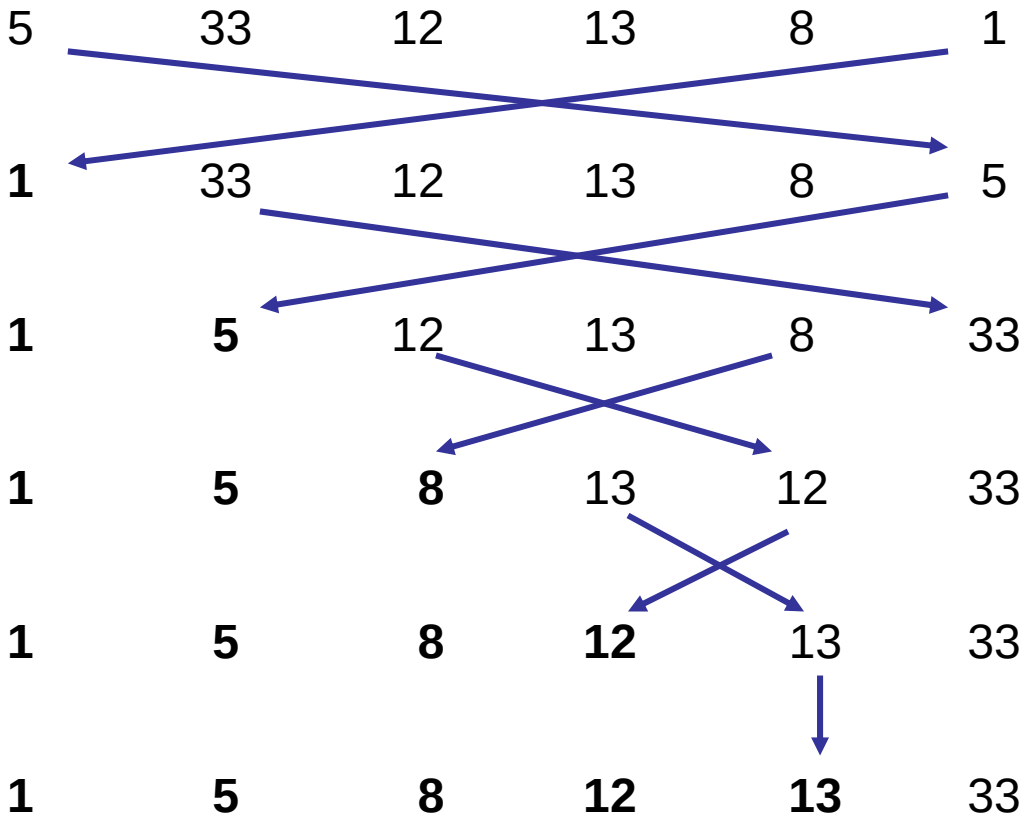
Direktes Aussuchen („Select Sort“)

- **Idee:** Wiederholtes Durchsuchen des unsortierten Teils des Feldes nach dem **kleinsten Element**, welches dann **mit dem ersten Element** 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 („Select Sort“)



Selectsort 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;
}
```

Selectsort in Java (objekt-orientiert)

```
public class SelSortOO
{ private int[] arr;
  public SelSortOO (double[] a) {    arr = a;}
  public double[] getArray()    {    return arr; }
  public void tausche (int i, int j)  { ...    }
  public void sort ()
  {     for (int outer = 0; outer < arr.length; outer++)
        {       int min = outer;
                /*kleinstes Element suchen, Index nach min*/
                for (int inner = outer + 1; inner < arr.length; inner++)
                        if (arr[inner] < arr[min]) min=inner;
                tausche(outer, min);
        }
  }
```

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:**

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

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

- **Anzahl der Vertauschungen:**

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

- **Zeitkomplexität:**

$$T^{w}_{selectSort}(n) = T^{ave}_{selectSort}(n) \in O(n^2)$$

Sortieren durch Vertauschen („Bubble Sort“)

- **Idee:** Vertausche benachbarte Elemente, wenn sie nicht wie gewünscht geordnet sind.

  In jedem Durchlauf des Feldes steigt das relativ größte Element wie eine "`Blase"' (bubble) im Wasser auf, was dem Verfahren seinen Namen gegeben hat.

Beispiel: „Bubble Sort“

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

Nach erster Runde ist das größte Element außen

Nach zwei Runden sind die beiden größten Elemente geordnet

Bubble Sort in Java

```
public static  void sort(double[] a)
{ for (int outer = a.length-1; outer>0; outer--)
  {     for (int inner=0; inner<outer; inner++)
        {       if (a[inner]>a[inner+1]) tausche(a,inner,inner+1);
        }
  }
}
```

Aufwandsabschätzung

- **Anzahl der Vergleichsoperationen** (wie bei selectSort):

$$C_{min}(n) = C_{max}(n) = C_{ave}(n) = \sum_{i=1}^{n-1} i \;\; = (n/2)\,(n-1) \in O(n^2)$$

- **Anzahl der Vertauschungen**:

Schlechtester Fall:

(wenn die Elemente in der falschen Reihenfolge, d.h. absteigend geordnet sind):

$$V_{max}(n) = \sum_{i=1}^{n-1}(n-i) = \; n(n-1) - (n/2)(n-1) = (n/2)\,(n-1) \in O(n^2)$$

Durchschnittlicher Fall:

$$V_{ave}(n) = \sum_{i=1}^{n-1} (n-i)/2 = (n/4)(n-1) \in O(n^2)$$

- **Zeitkomplexität:**

$$T^{w}_{bubbleSort}(n) = T^{ave}_{bubbleSort}(n) \in O(n^2)$$

„Quicksort (von C.A.R. Hoare)“

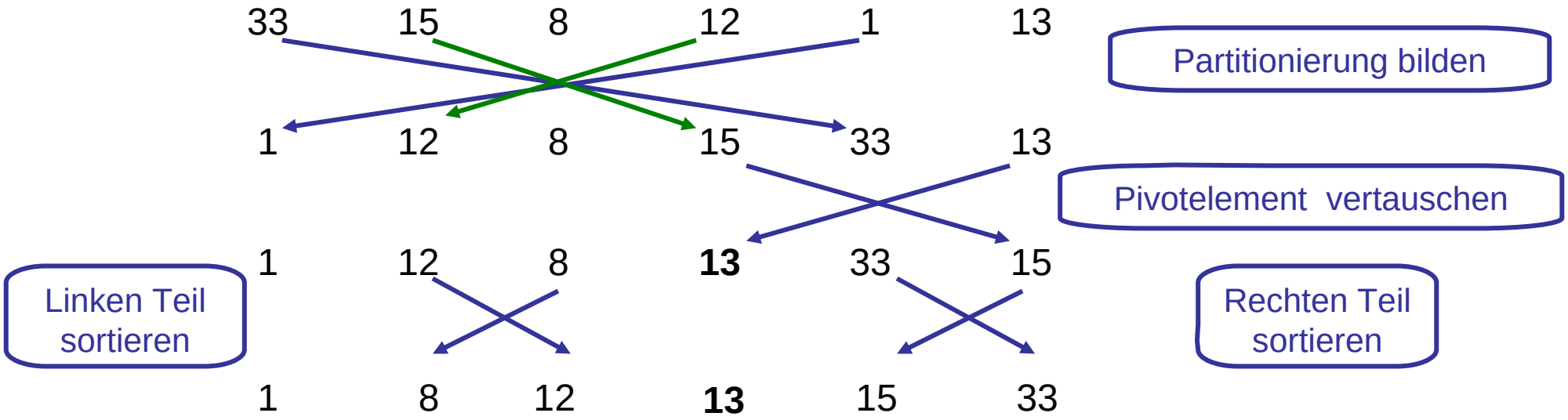
- Quicksort ist nach Heapsort der schnellste bekannte interne Sortieralgorithmus, da Austauschen am effizientesten ist, wenn es über große Distanzen erfolgt.

- **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 Pivotelement 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
    }
}
```

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

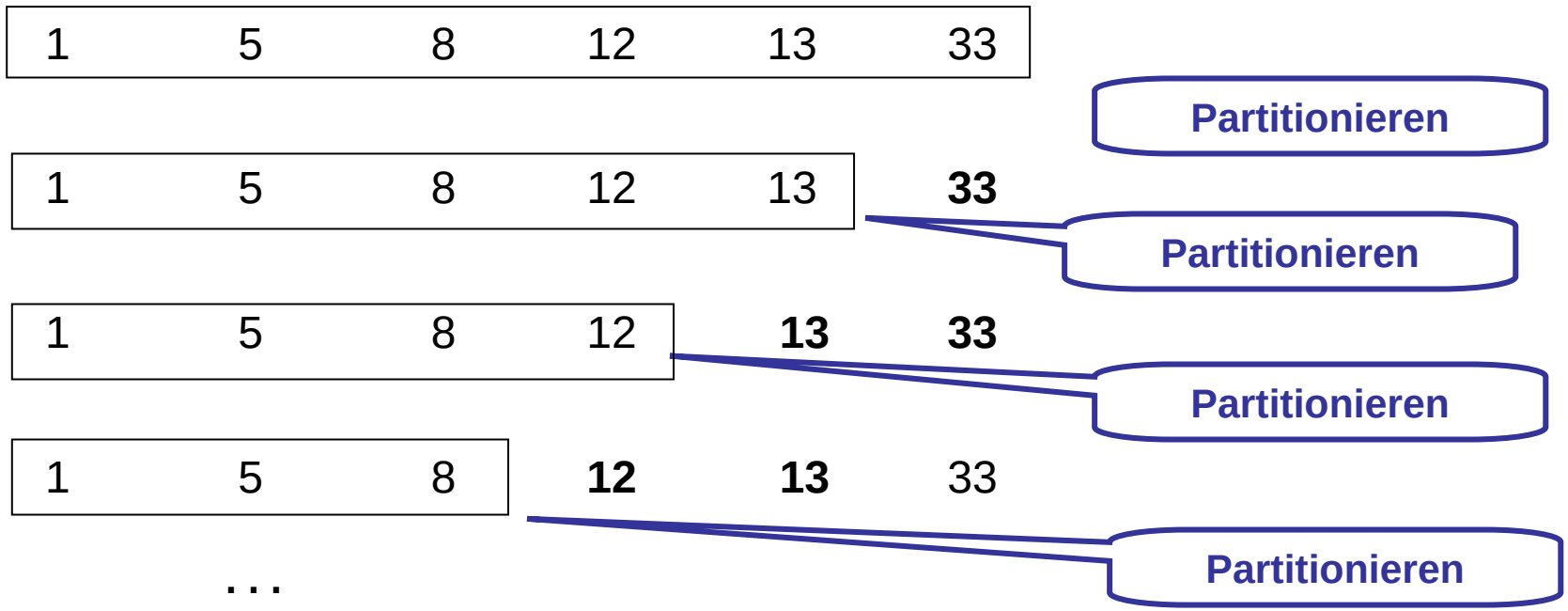
- Bei kurzen Reihungen sind die einfachen Sortieralgorithmen meist besser, deshalb kann man Teilreihungen mit weniger als 10 Elementen z.B. mit SelectSort sortieren.
- Eine bessere Pivotbestimmung ergibt sich aus dem mittleren Wert der drei folgenden Elemente der Reihung: linkes Element, rechtes Element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

Aufwandsabschätzung

O(n) Vergleiche pro Partitionierung **O(n) Partitionierungen**

Schlechtester Fall: O(n * n)

z.B. wenn die Reihung schon aufsteigend sortiert ist, gleiche Anzahl von Vergleichen wie bei SelectSort $\sum_{i=1}^{n-1} i = (n/2)\,(n-1) \in O(n^2)$:

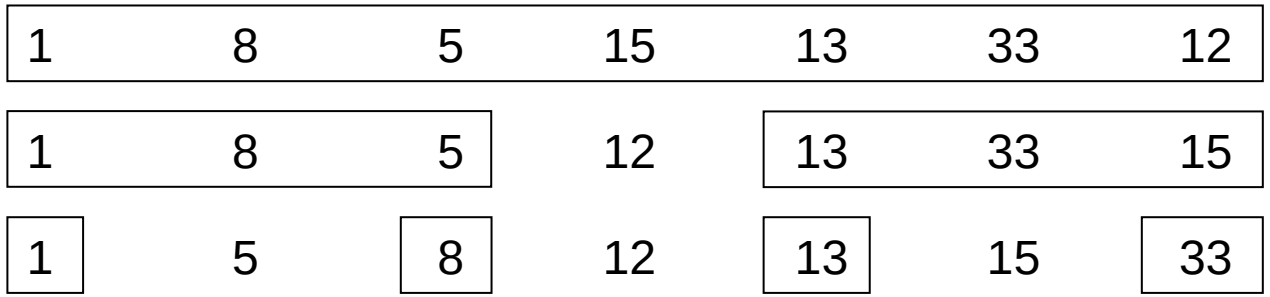


Aufwandsabschätzung

- **Bester Fall:** O(n * log n)

  Idee:

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



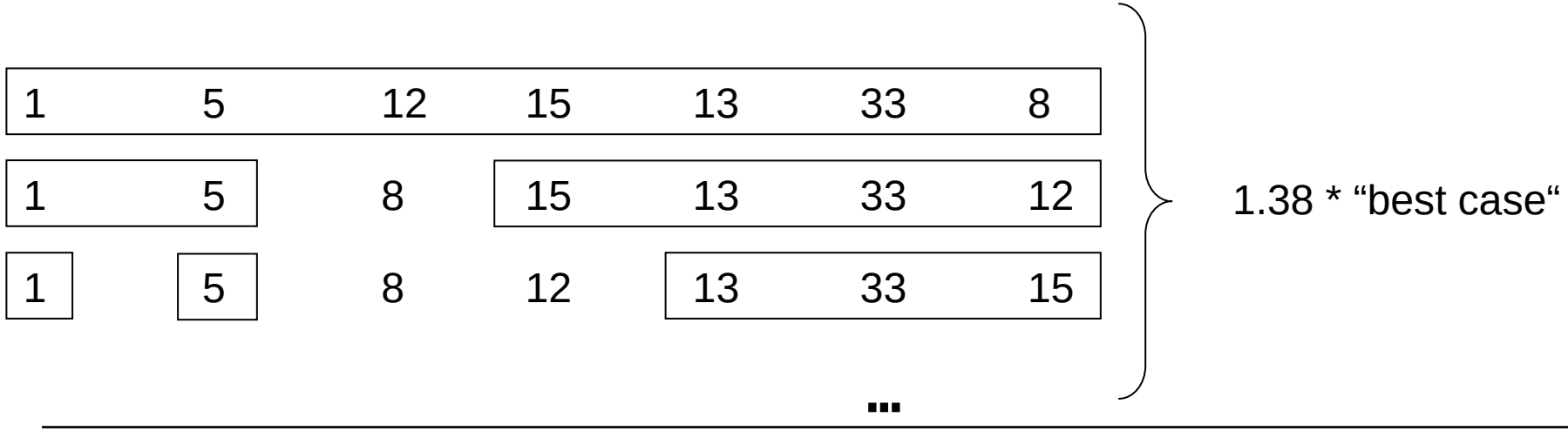
Anzahl der Partitionen verdoppelt sich von Zeile zu Zeile, ihre Größe halbiert sich (ungefähr). $\log_2(n)$: Anzahl der Verdopplungen um von 1 auf n zu kommen.

Aufwandsabschätzung

- **Durchschnittlicher Fall:** O(n * log n)

  Idee:

  - Jede Aufteilung gleich wahrscheinlich (1:n-1, 2:n-2, 3:n-3 … n-1:1),
  - Im Durchschnitt benötigt man etwas mehr Zeilen als im besten Fall.
  - Man kann zeigen, dass Anzahl der Zeilen = 2*ln(n) ist, das sind 38% mehr Zeilen



Zusammenfassung

- Sortieren durch Einfügen und Bubblesort sind Sortieralgorithmen mit quadratischer Zeit- und konstanter Speicherplatzkomplexität.
- Quicksort hat im Mittel die Zeitkomplexität O(n * log n). Im schlechtesten Fall ist es n^2.