Informatik I

Prof. Dr. Christian Pape

Kapitel 12

Sortieralgorithmen

Inhalt

- Sortieren
  - Sortieren durch direktes Einfügen
  - Mergesort
  - Sortieren durch direkte Auswahl
  - Bubblesort
    - Variante: Shakersort
  - Quicksort

Sortieren

- Für Vorverarbeitung einer Vielzahl von Algorithmen
  - Binäre Suche
  - Grafisch geometrische Algorithmen (Bildverarbeitung)
  - Sortierte Ausgabe auf dem Bildschirm
    - Immobilien nach Preis sortiert
    - Aktuellsten Nachrichten zuerst

Sortieren

- Gegeben:

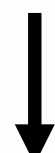
  Eine Folge von n Objekten

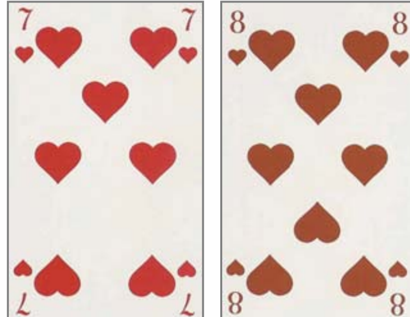
- Gesucht:

  Neue aufsteigend sortierte Folge aller n Objekte

Unsortierte Folge	1, 5, 8, 6, 3, 2, 9 ,7, 6, 8, 4, 2
Sortierte Folge	1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9

Sortieren



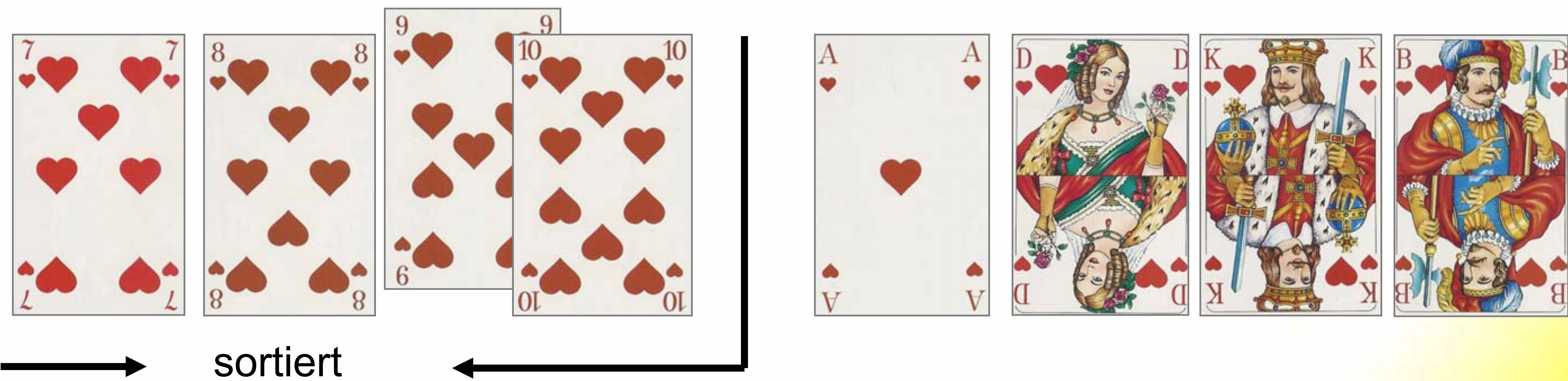


Sortieren durch direktes Einfügen

sortiert

Nächste Karte betrachten

Nächste Karte wird im sortierten Bereich an richtige Stelle direkt eingefügt



Sortieren durch direktes Einfügen

Sortieren durch direktes Einfügen

Sortieren durch direktes Einfügen

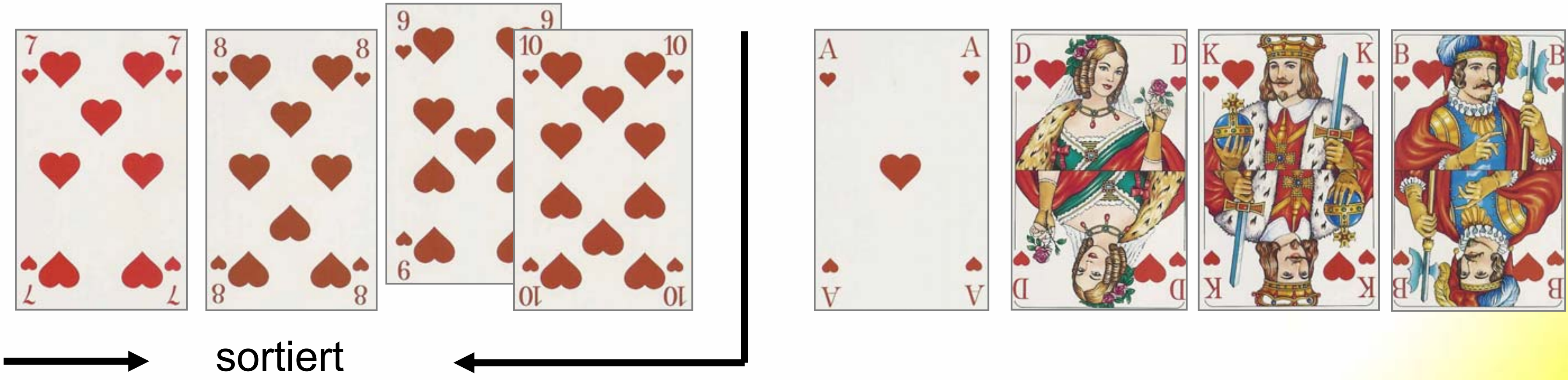
Sortieren durch direktes Einfügen

Sortieren durch direktes Einfügen

sortiert

Nächste Karte betrachten

Nächste Karte wird im sortierten Bereich an richtige Stelle direkt eingefügt



Sortieren durch direktes Einfügen

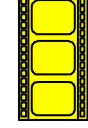
sortiert

Nächste Karte betrachten

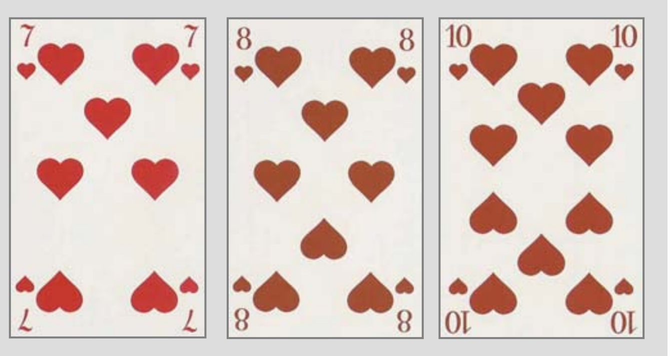
Nächste Karte wird im sortierten Bereich an richtige Stelle direkt eingefügt

```
void direktesEinfuegen(int a[]) {
  for (int i=1; i<a.length; i++) {
    int t = a[i];     // naechste Karte
    for (int j = i; j > 0 && a[j-1] > t; j--) {
      a[j] = a[j-1]; // Karte t eine Karte nach
      a[j-1] = t;    // links einfuegen
    }
  }
}
```

i=1: im ersten Durchlauf passiert nichts

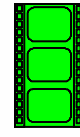


Sortieren durch direktes Einfügen



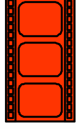
nach 3 Durchläufen

```
void direktesEinfuegen(int a[]) {
  for (int i=1; i<a.length; i++) {
    int t = a[i]; // naechste Karte
    for (int j = i; j > 0 && a[j-1] > t; j--) {
      a[j] = a[j-1]; // Karte t eine Karte nach
      a[j-1] = t;    // links einfuegen
    }
  }
}
```

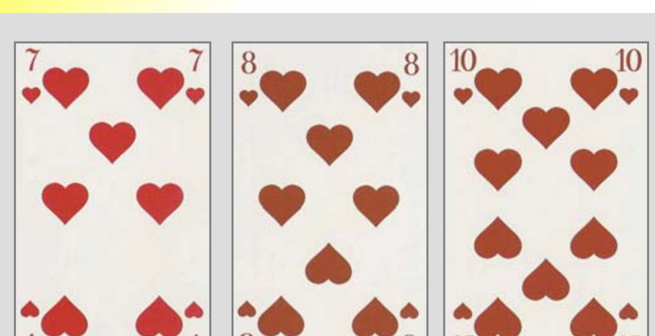


Sortieren durch direktes Einfügen

```
void direktesEinfuegen(int a[]) {
  for (int i=1; i<a.length; i++) {
    int t = a[i]; // naechste Karte
    for (int j = i; j > 0 && a[j-1] > t; j--) {
      a[j] = a[j-1]; // Karte t eine Karte nach
      a[j-1] = t;    // links einfuegen
    }
  }
}
```



Sortieren durch direktes Einfügen



```
void direktesEinfuegen(int a[]) {
   for (int i=1; i<a.length; i++) {
      int t = a[i]; // naechste Karte
      for (int j = i; j > 0 && a[j-1] > t; j--) {
        a[j] = a[j-1]; // Karte t eine Karte nach
        a[j-1] = t;    // links einfuegen
      }
   }
}
```

Sortieren durch direktes Einfügen

```
void direktesEinfuegen(int a[]) {
  for (int i=1; i<a.length; i++) {
    int t = a[i]; // naechste Karte
    for (int j = i; j > 0 && a[j-1] > t; j--) {
      a[j] = a[j-1]; // Karte t eine Position
      a[j-1] = t;    // links einfuegen
    }
  }
}
```

5	7	5	6	1	8	2

	5	7	5	6	1	8	2
	5	7	5	6	1	8	2
	5	7	5	6	1	8	2
	5	5	7	6	1	8	2
	5	5	6	7	1	8	2
1 wurde 4 mal getauscht	1	5	5	6	7	8	2
	1	5	5	6	7	8	2
2 wurde 5 mal getauscht	1	2	5	5	6	7	8

Sortieren durch direktes Einfügen

```
void direktesEinfuegen(int a[]) {
  for (int i=1; i<a.length; i++) {
    int t = a[i]; // naechste Karte
    for (int j = i; j > 0 && a[j-1] > t; j--) {
      a[j] = a[j-1]; // Karte t eine Karte nach
      a[j-1] = t;    // links einfuegen
    }
  }
}
```

- Aufwand innere for-Schleife für ein i

  Günstigster Fall (a schon sortiert)

  – kein mal durchlaufen, da immer a[j-1] <= t, also konstant

  – $T(n) = O(n)$

  Schlechtester Fall (a umgekehrt sortiert)

  – i -1 mal durchlaufen

  – $T_{wc}(n) = 1 + 2 + 3 + \ldots + (n-1) = O(???)$

Sortieren durch direktes Einfügen

- Behauptung

  $T(n) = 1 + 2 + 3 + \ldots (n-2) + (n-1) = n \cdot (n-1) / 2$
- Beweis mit vollständiger Induktion über n
  - Induktionsanfang: $n = 1$ (nur eine Zahle)

    $T(n) = 1 = 1 \cdot (1 + 1) / 2 = 1$
  - Induktionsschritt von n+1 auf n

    $T(n+1) = 0 + 1 + \ldots + (n-1) + n = T(n) + n$

    $= n \cdot (n-1) / 2 + n$ (Induktionsvoraussetzung für n)

    $= (n \cdot (n-1) + 2n) / 2$

    $= (n^2 - n + 2n) / 2$

    $= (n^2 + n) / 2$

    $= ((n+1) \cdot n) / 2$ (Behauptung für n+1)
- $O(n \cdot (n-1) / 2) = O(½ n^2 + ½ n) = O(½ n^2) = O(n^2)$

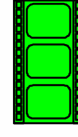
Fazit

- $O(n^2)$ schlecht für grosse Folgen
- Sortieren durch direktes Einfügen
  - Zu viele Tauschoperationen, die nicht immer sofort das einzufügende Element an seinen Platz bringt
  - n mal wird äussere Schleife durchlaufen und nur ein Element kommt an seinen Platz
- Geht es auch besser?
  - Sortieren durch Aufteilen in kleinere Probleme und Zusammenfügen der Teillösungen zu einem Ganzen

Inhalt

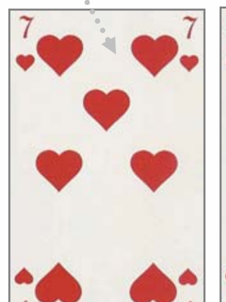
- Sortieren
  - Sortieren durch direktes Einfügen
  - Mergesort
  - Sortieren durch direkte Auswahl
  - Bubblesort
  - Quicksort

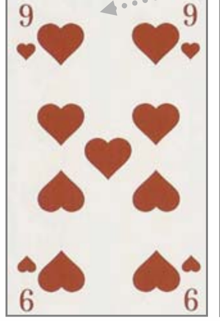
Mergesort

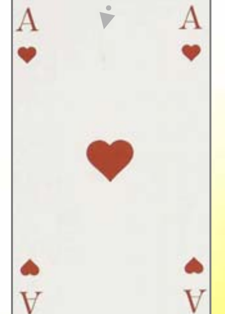


Feld in der Mitte teile und beide Teile getrennt sortieren

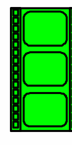
Zusammenfügen der beiden Teile nach dem „Reisverschlussprinzip“

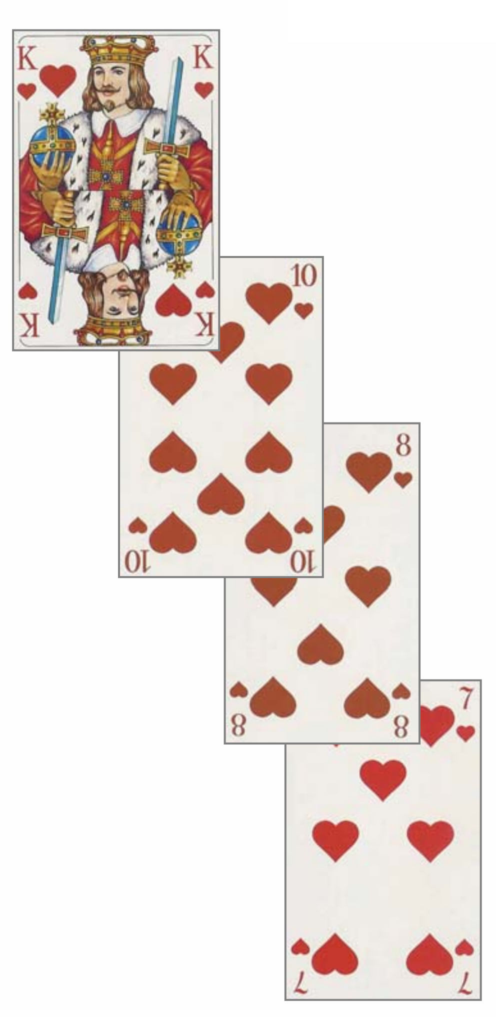


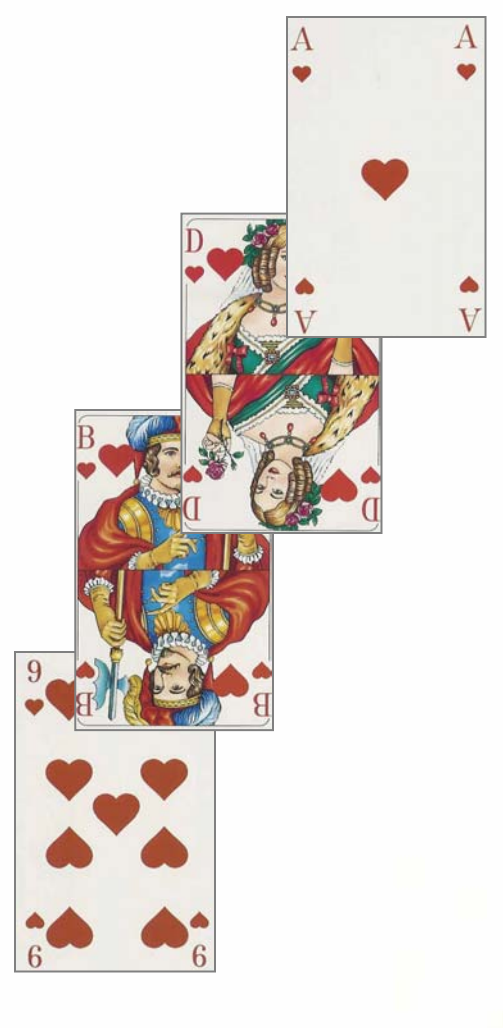




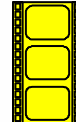
Reisverschlussprinzip



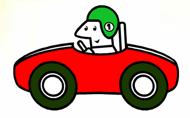
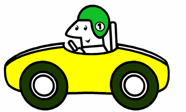
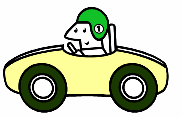
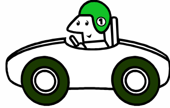




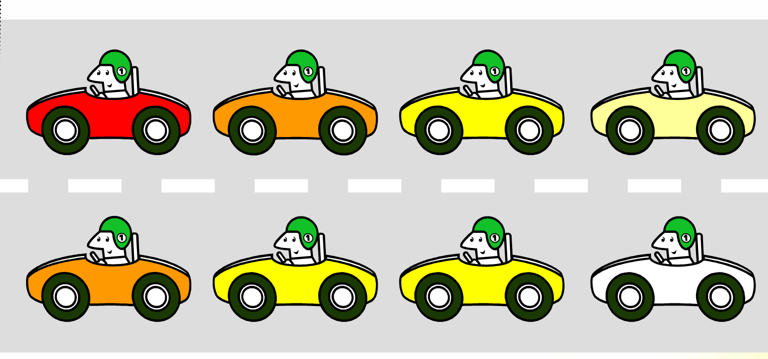
Reisverschlussprinzip / Autobahn



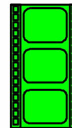
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

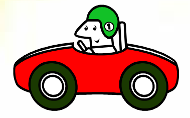
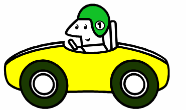
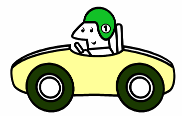
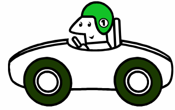
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



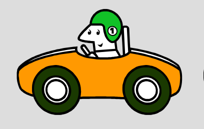
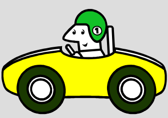
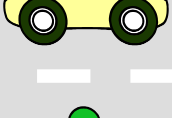
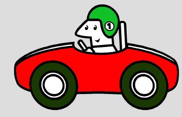
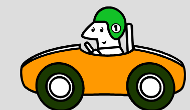
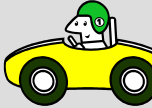
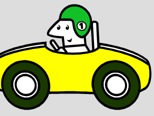
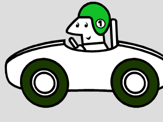
Reisverschlussprinzip / Autobahn



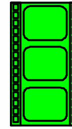
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

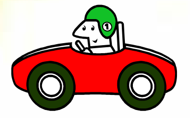
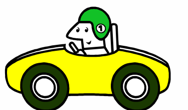
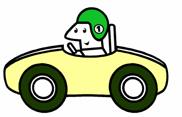
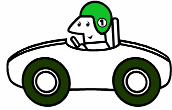
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten

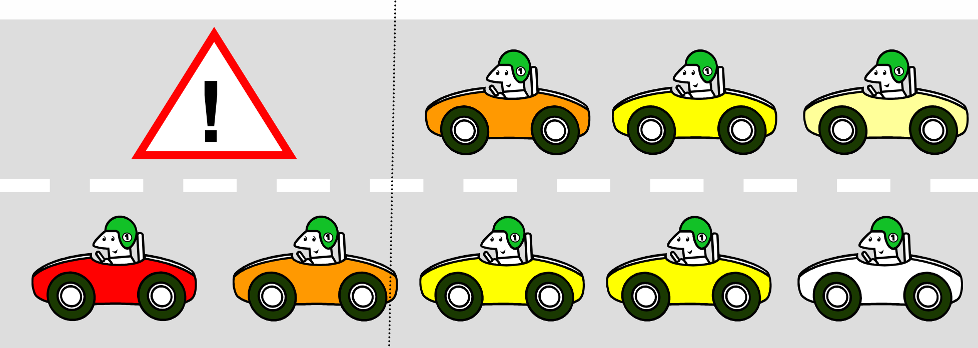
Reisverschlussprinzip / Autobahn



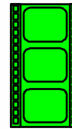
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

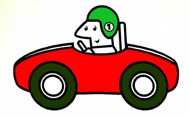
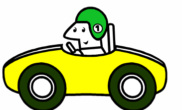
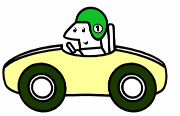
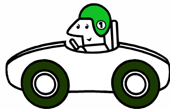
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



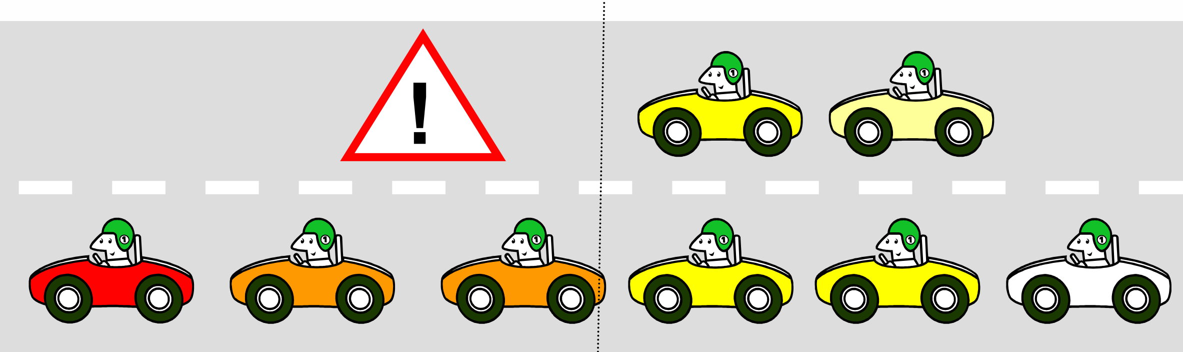
Reisverschlussprinzip / Autobahn



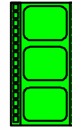
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

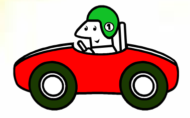
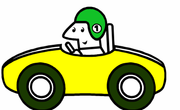
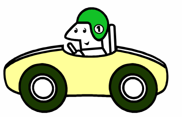
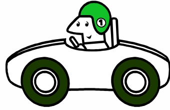
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



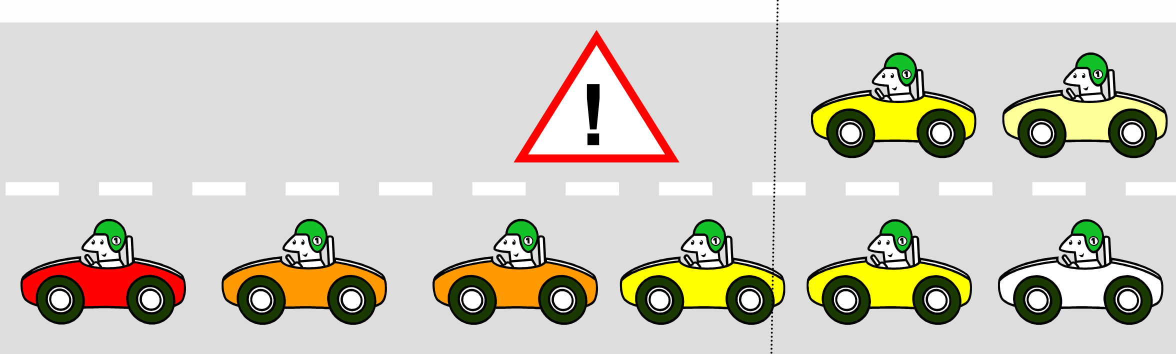
Reisverschlussprinzip / Autobahn



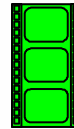
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

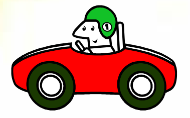
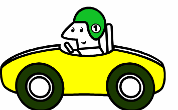
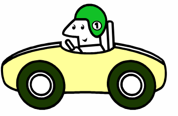
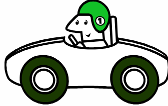
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



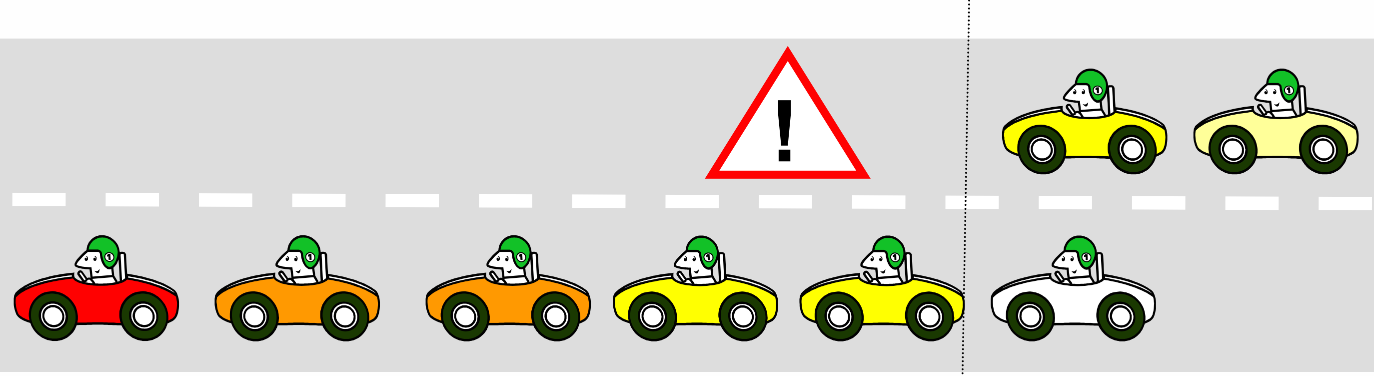
Reisverschlussprinzip / Autobahn



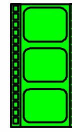
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

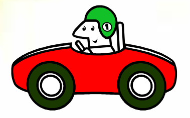
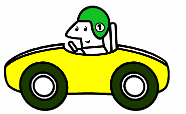
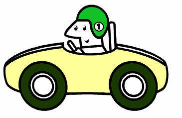
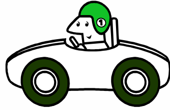
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



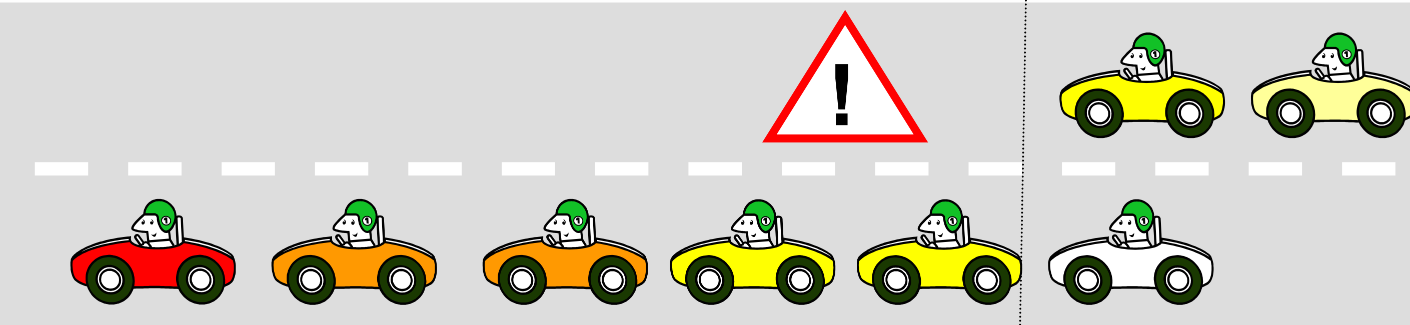
Reisverschlussprinzip / Autobahn



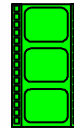
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

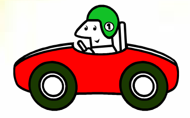
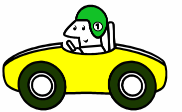
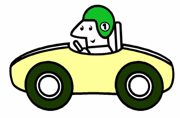
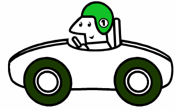
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



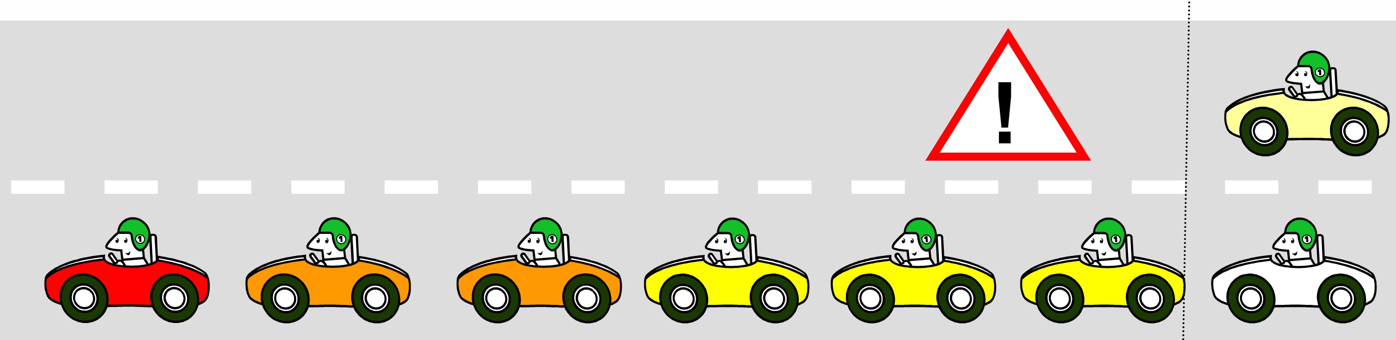
Reisverschlussprinzip / Autobahn



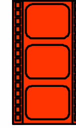
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

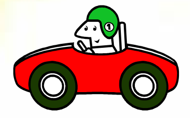
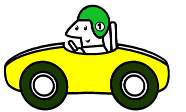
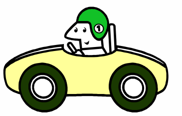
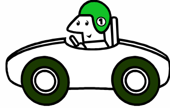
- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten



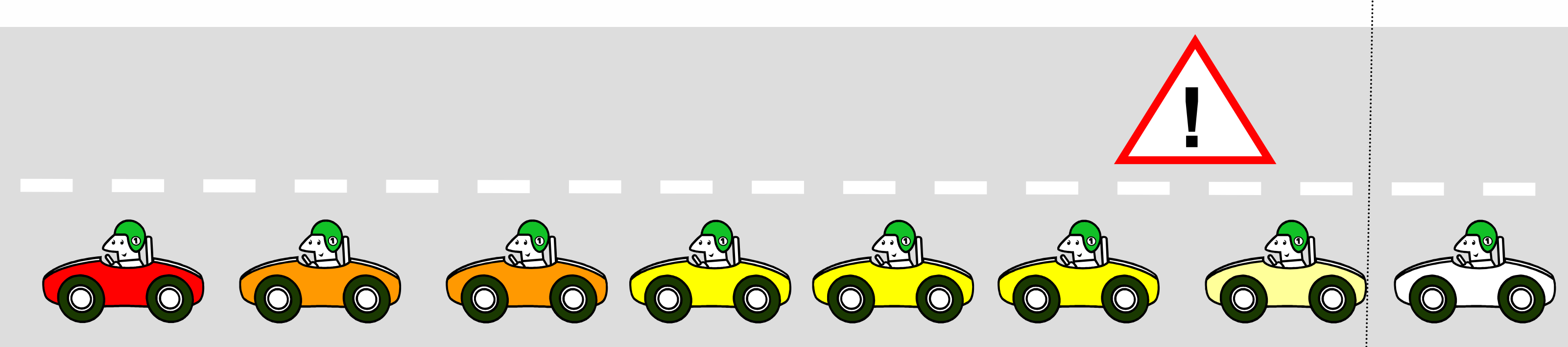
Reisverschlussprinzip / Autobahn



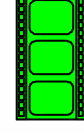
- Annahme
  - Auf beide Spuren sind schnellsten Autos vorne, langsamsten hinten

 ist schneller als ist schneller als  …  … 

- Reisverschlussprinzip bei Mergesort
  - Immer erst den schnelleren Wagen zuerst fahren lassen
- Danach sind schnellsten Autos vorne, langsamsten hinten

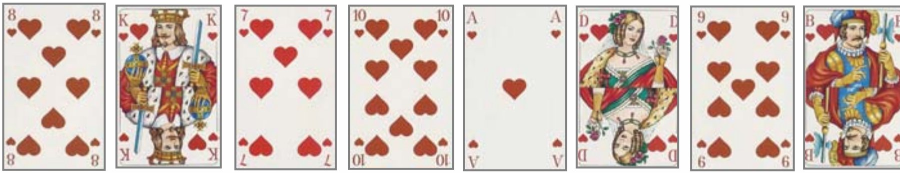


Mergesort

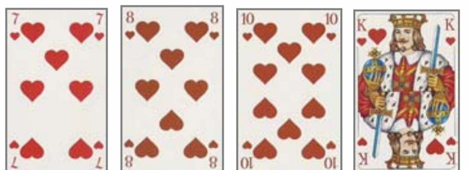


Teile und Beherrsche Prinzip

1. Problem in zwei gleich große Probleme **teilen**



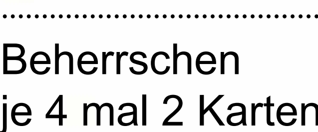
2. Probleme getrennt lösen (rekursiv)



3. Gelöste Teilprobleme zu Gesamtlösung zusammensetzen (**beherrschen**)

1. Teileschritt

2. Teileschritt
je 2mal



Beherrschen
je 4 mal 2 Karten

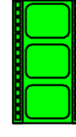
Beherrschen
je 2 mal 4 Karten

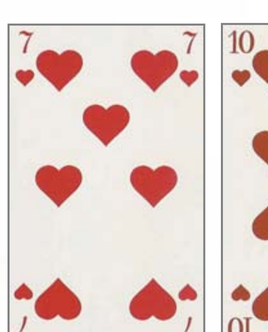
Beherrschen
1 mal 8 Karten

Rekursive Aufrufe

Rückkehr zum Aufruf

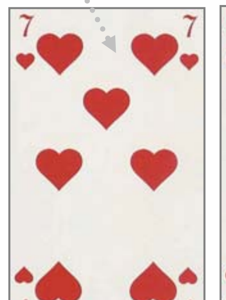
Mergesort

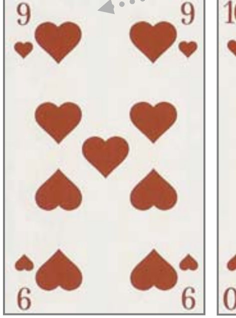


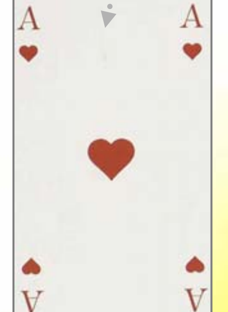


Feld in der Mitte teilen und beide Teile getrennt sortieren

Zusammenfügen der beiden Teile nach dem „Reisverschlussprinzip“







Mergesort

Teilen

links

rechts

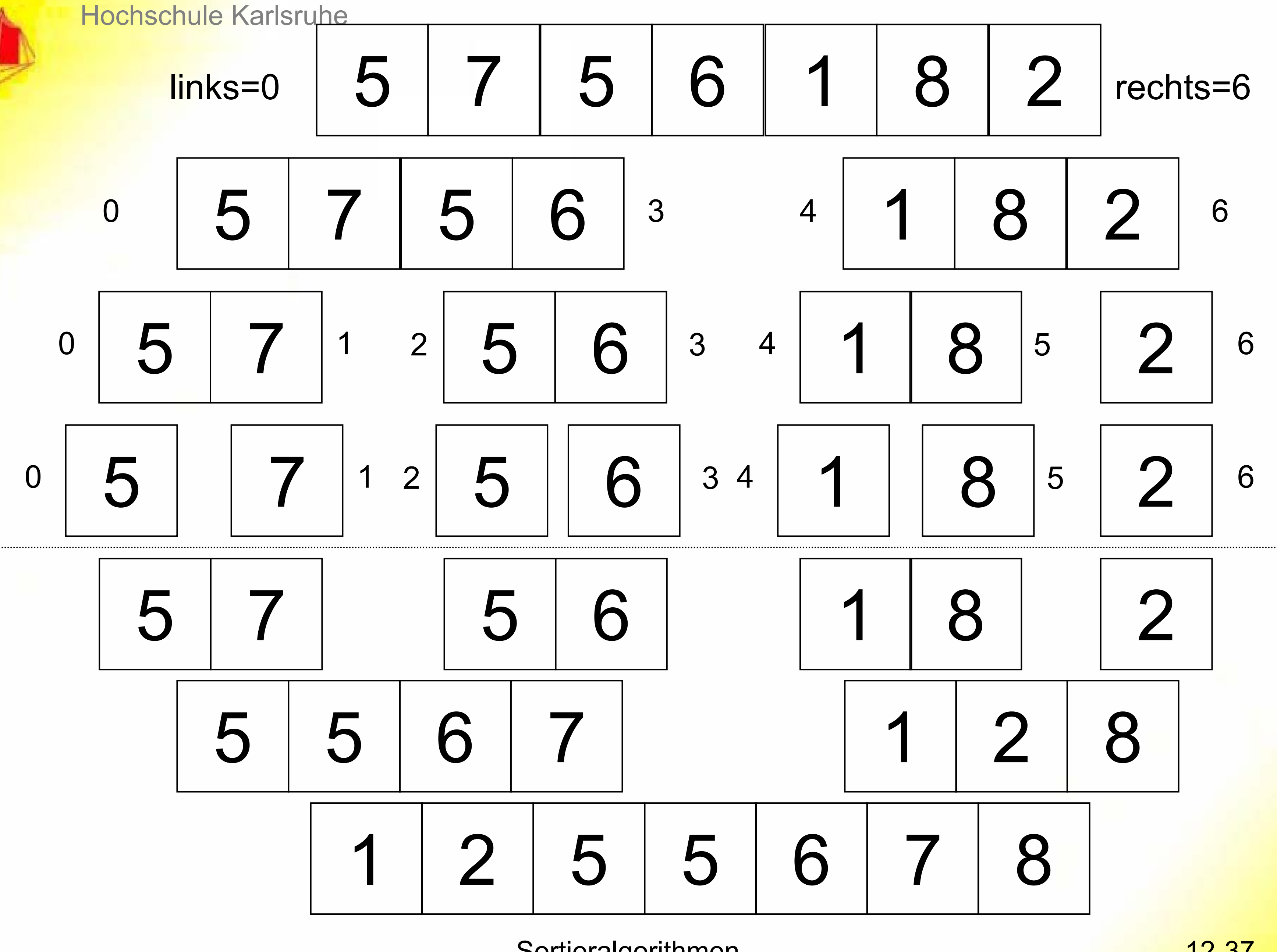
(links + rechts) / 2

```
void mergesort(int a[], int links, int rechts) {
  if (links < rechts) {
    int mitte = (links + rechts) / 2;
    mergesort(a, links, mitte );
    mergesort(a, mitte + 1, rechts);
    reisverschluss(a, links, mitte, rechts);
  }
}
```

Beherrschen

(Rechnerübung)

5	7	5	6	1	8	2


links=0
5 7 5 6 1 8 2
rechts=6
0
5 7 5 6
3
4
1 8 2
6
0
5 7
1
2
5 6
3
4
1 8
5
2
6
0
5
7
1
2
5
6
3
4
1
8
5
2
6
5 7
5 6
1 8
2
5 5 6 7
1 2 8
1 2 5 5 6 7 8

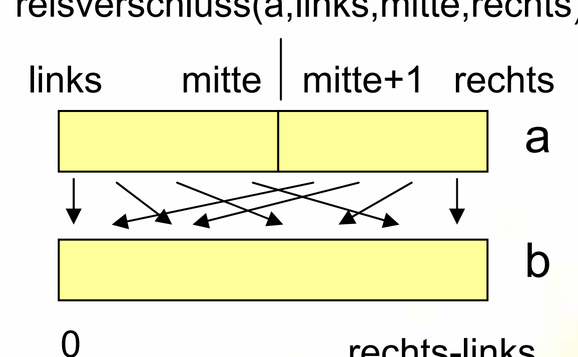
Mergesort

```
void mergesort(int a[], int links, int rechts) {
   if (links < rechts) {
      int mitte = (links + rechts) / 2;
      mergesort(a, links, mitte );
      mergesort(a, mitte + 1, rechts);
      reisverschluss(a, links, mitte, rechts);
   }
}
```

- Aufwand „Beherrschen“ bei n = rechts - links + 1
  - Beide Hälften von Anfang bis Ende durchgehen und nach Reisverschlussprinzip jeweils kleinste Element an nächster Stelle in einem neuen Feld b der Grösse n kopieren
  - Inhalt des neuen Feldes zurückkopieren a[rechts] bis a[links]
  - Insgesamt c·n Operationen nötig (c Konstante)
- Rekurrenzgleichung

  $T(1) = c$ (Rekursionsabbruch)

  $T(n) = 2\,T(n/2) + c \cdot n$



Mergesort

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 4T(n/4) + 2(cn/2) + cn \\
&= 8T(n/8) + 4(cn/4) + 2(cn/2) + cn \\
&= 8T(n/8) + cn + cn + cn \\
&= 2^3 T(n/2^3) + 3cn \\
&= 2^4 T(n/2^4) + 4cn \\
&= \ldots \\
&= 2^k T(n/2^k) + kcn \\
&= 2^k T(2^k/2^k) + kcn \qquad \text{Annahme } n = 2^k \\
&= 2^k T(1) + kcn \\
&= nc + (\log_2 n)cn = O(n \log_2 n)
\end{aligned}
$$

Vergleich

	Direktes Einfügen	Mergesort
Schlimmster Fall	$O(n^2)$ (Elemente in umgekehrter Sortierung gegeben)	$O(n \log_2 n)$
Bester Fall	$O(n)$ (Elemente von Anfang an sortiert)	$O(n \log_2 n)$
Mittlerer Fall	$O(n^2)$	$O(n \log_2 n)$

Java / Mergesort

```
import java.util.*;
…
String a[] = new String[3];

a[0] = "Müller";
a[1] = "Meier";
a[2]  = "Albrecht";

Collections.sort(list); // Mergesort Implementierung
for (int i = 0; i < a.length; i++) {
  System.out.println(a[i]);
}
```

Ausgabe:
Albrecht
Meier
Müller

Inhalt

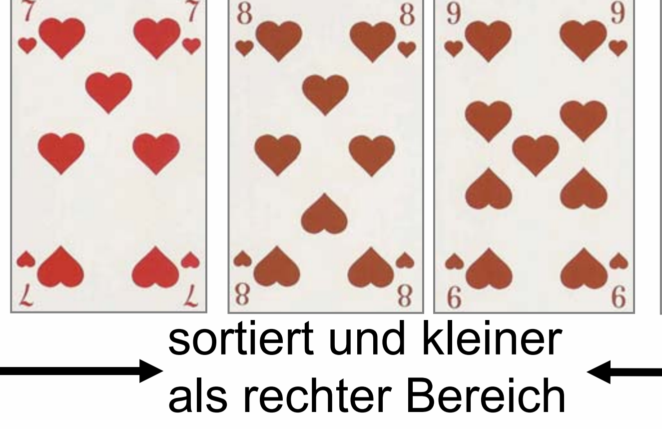
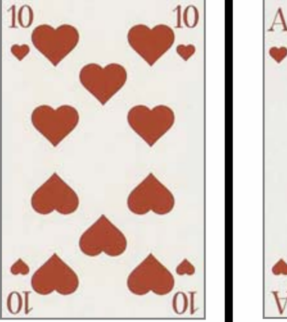
- Sortieren
  - Sortieren durch direktes Einfügen
  - Mergesort
  - Sortieren durch direkte Auswahl
  - Bubblesort
  - Quicksort

Sortieren durch direkte Auswahl

sortiert und kleiner
als rechter Bereich

Kleinste Karte aus rechtem
Bereich auswählen

Kleinste Karte wird mit Karte rechts vom sortierten Bereich (König) getauscht

sortiert und kleiner
als rechter Bereich

Sortieren durch direkte Auswahl

Sortieren durch direkte Auswahl

Sortieren durch direkte Auswahl

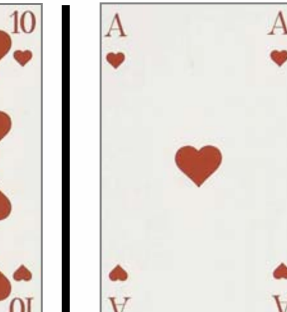
Sortieren durch direkte Auswahl

Sortieren durch direkte Auswahl

sortiert und kleiner
als rechter Bereich

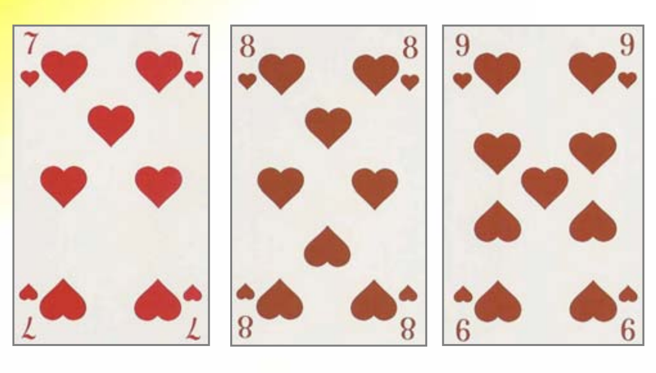
Kleinste Karte aus rechtem
Bereich auswählen

Kleinste Karte wird mit Karte rechts vom sortierten Bereich (König) getauscht



sortiert und kleiner
als rechter Bereich

Sortieren durch direkte Auswahl



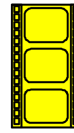
sortiert und kleiner
als rechter Bereich

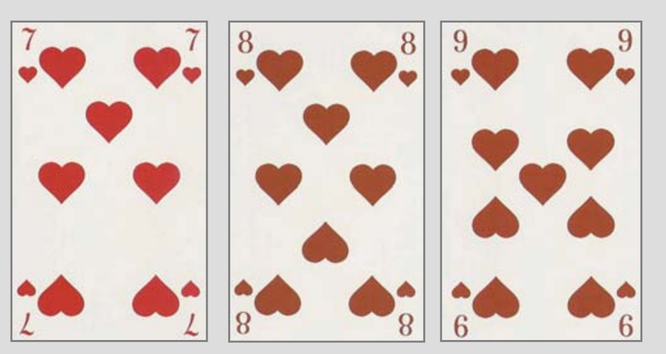
Kleinste Karte aus rechtem
Bereich auswählen

Kleinste Karte wird mit Karte rechts vom sortierten Bereich (König) getauscht

```
void direkteAuswahl(int a[]) {
  for (int i=0; i<a.length; i++) {
    // suche Index des kleinsten Elements
    int minIndex = minimum(a, i, a.length-1);
    vertauschen(a, minIndex, i);
  }
}
```

Sortieren durch direkte Auswahl

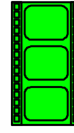


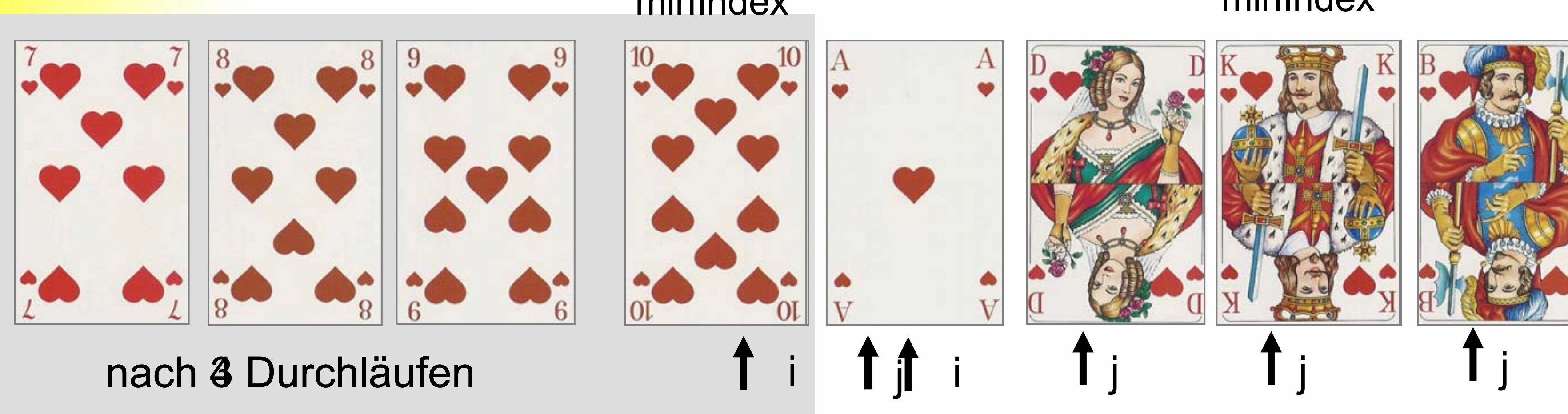


nach 3 Durchläufen

```
void direkteAuswahl(int a[]) {
  for (int i=0; i<a.length; i++) {
    int minIndex = i;
    for (int j=i+1; j<a.length; j++) {
      if (a[j] < a[minIndex]) {
        minIndex = j;
      }
    }
    int tmp = a[i]; a[i] = a[minIndex]; a[minIndex] = tmp;
  }
}
```

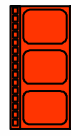
Sortieren durch direkte Auswahl

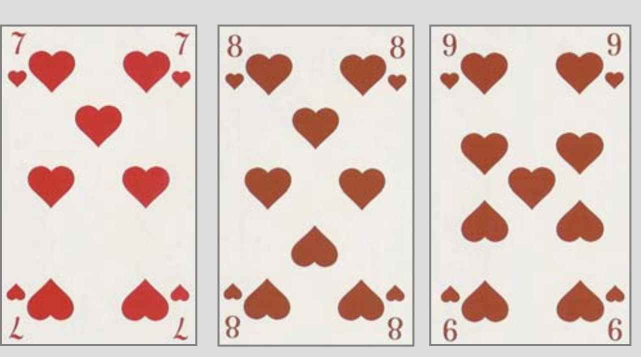




```
void direkteAuswahl(int a[]) {
  for (int i=0; i<a.length; i++) {
    int minIndex = i;
    for (int j=i+1; j<a.length; j++) {
      if (a[j] < a[minIndex]) {
        minIndex = j;
      }
    }
    int tmp = a[i]; a[i] = a[minIndex]; a[minIndex] = tmp;
  }
}
```

Sortieren durch direkte Auswahl



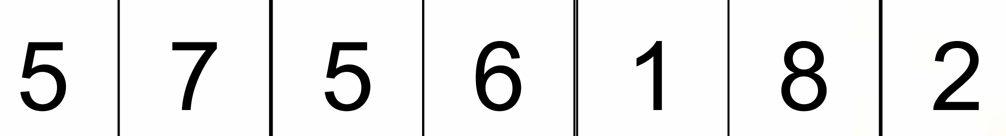


nach 4 Durchläufen

```
void direkteAuswahl(int a[]) {
  for (int i=0; i<a.length; i++) {
    int minIndex = i;
    for (int j = i + 1; j < a.length; j++) {
      if (a[j] < a[minIndex]) {
        minIndex = j;
      }
    }
    int tmp = a[i];
    a[i] = a[minIndex];
    a[minIndex] = tmp;
  }
}
```

Sortieren durch direkte Auswahl

```
void direkteAuswahl(int a[]) {
  for (int i=0; i<a.length; i++) {
    int minIndex = i;
    for (int j = i + 1; j < a.length; j++) {
      if (a[j] < a[minIndex]) {
        minIndex = j;
      }
    }
    int tmp = a[i];
    a[i] = a[minIndex];
    a[minIndex] = tmp;
  }
}
```



Kleinste Element

Bisher sortierte Folge

5	7	5	6	1	8	2
1	7	5	6	5	8	2
1	2	5	6	5	8	7
1	2	5	6	5	8	7
1	2	5	5	6	8	7
1	2	5	5	6	8	7
1	2	5	5	6	7	8
1	2	5	5	6	7	8

Letzter Schritt ist unnötig

Sortieren durch direkte Auswahl

unnötigen letzten Schritt auslassen

```
void direkteAuswahl(int a[]) {
  for (int i=0; i<a.length - 1; i++) {
    int minIndex = i;
    for (int j=i+1; j<a.length; j++) {
      if (a[j] < a[minIndex]) {
        minIndex = j;
      }
    }
    int tmp = a[i];
    a[i] = a[minIndex];
    a[minIndex] = tmp;
  }
}
```

$T(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$

$= n \cdot (n-1) / 2$

$= O(n^2)$

(siehe direktes Einfügen)

Anzahl Iterationen inneres-for für i=0

Anzahl Iterationen inneres-for für i=1

Vergleich

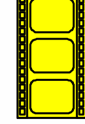
	Direkte Auswahl	Direktes Einfügen	Mergesort
Schlimmster Fall	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$
Bester Fall	$O(n^2)$	$O(n)$	$O(n \log_2 n)$
Mittlerer Fall	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$

Vergleich

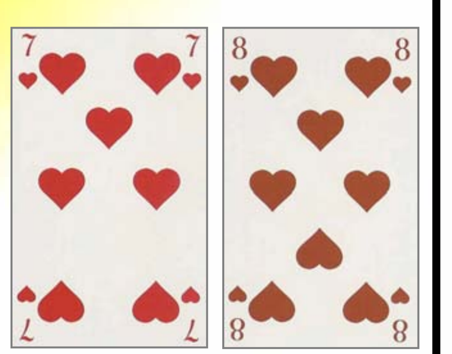
Direkte Auswahl	Direktes Einfügen
• Immer $O(n^2)$ • Viele Vergleichsoperationen	• Viele - im schlimmsten Fall $O(n^2)$- teuere Vertauschoperationen
•$O(n)$ Vertauscheoperationen	• $O(n)$ bei fast sortierten Folgen

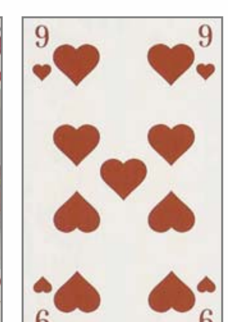
Inhalt

- Sortieren
  - Sortieren durch direktes Einfügen
  - Mergesort
  - Sortieren durch direkte Auswahl
  - Bubblesort
  - Quicksort



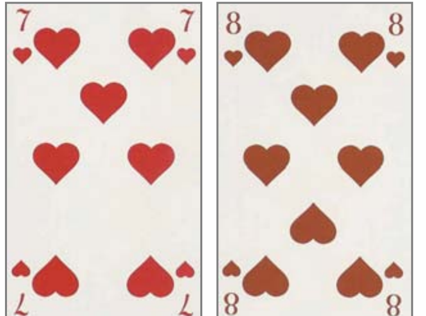
Bubblesort

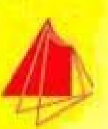


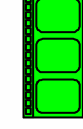


Sortiert und
kleiner als rechts

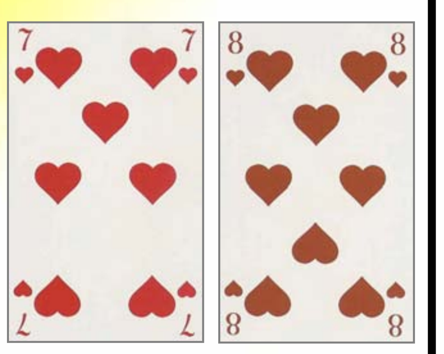
Gehe von rechts bis nach links (einschl. Bauer) und
vertausche dabei die Karten, wenn linke größer als die rechte ist





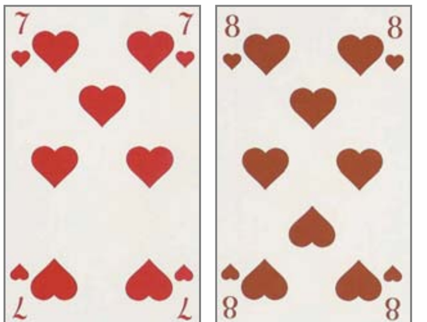


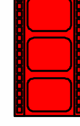
Bubblesort



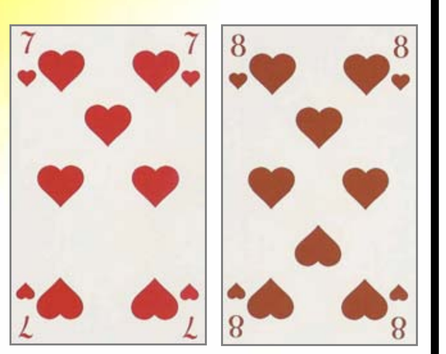
Sortiert und
kleiner als rechts

Gehe von rechts bis nach links (einschl. Bauer) und
vertausche dabei die Karten, wenn linke größer als die rechte ist



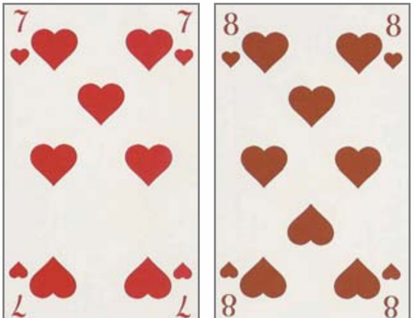


Bubblesort



Sortiert und
kleiner als rechts

Gehe von rechts bis nach links (einschl. Bauer) und
vertausche dabei die Karten, wenn linke größer als die rechte ist



Sortiert und kleiner als rechts

Bubblesort

Bubblesort

Bubblesort

Ende des ersten Durchlaufs
Kleinste Element ist garantiert ganz nach links gewandert.

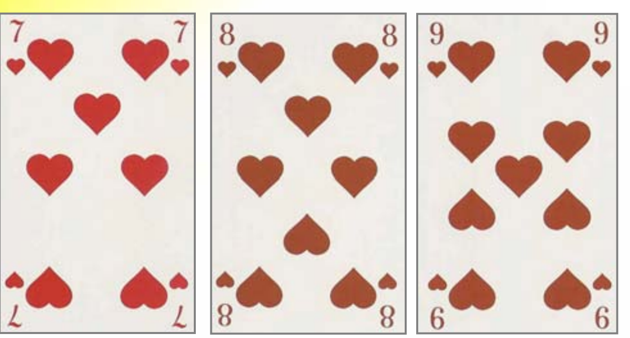
Bubblesort

Bubblesort

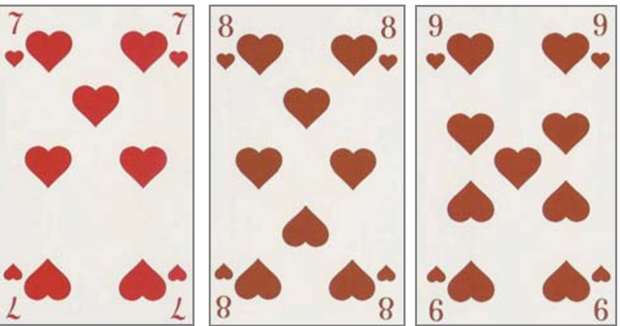
Ende des zweiten Durchlaufs
Zweit kleinste Element ist garantiert an seine Stelle (zweite) gewandert

Bubblesort

Bubblesort



Ende des dritten Durchlaufs
Dritt kleinste Element ist garantiert an seine Stelle (dritte) gewandert



Bubblesort

Ende des 4. Durchlauf

Ende des 5. Durchlauf

Ende des 6. Durchlauf, usw.

Bubblesort

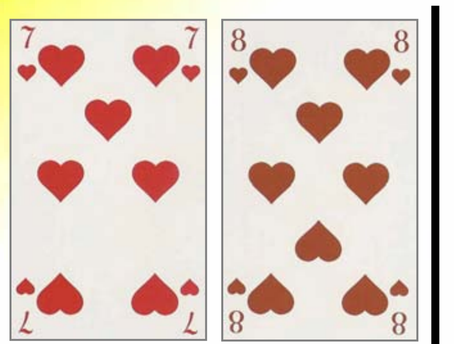
Sortiert und
kleiner als rechts

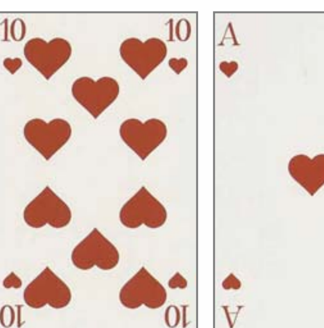
9 9 10

Gehe von rechts bis nach links (einschl. Bauer) und
vertausche dabei die Karten, wenn linke größer als die rechte ist

Sortiert und kleiner als rechts

Bubblesort





Sortiert und kleiner als rechts

Gehe von rechts bis nach links (einschl. Bauer)

vertausche dabei die Karten, wenn linke größer als rechte ist

```
void bubblesort(int a[]) {
  for (int i = 0; i < a.length; i++) {
    for (int j = a.length - 1; j > i; j--) {
      if (a[j-1] > a[j]) {
        int tmp = a[j-1];
        a[j-1] = a[j];
        a[j] = tmp;
      }
    }
  }
}
```

Bubblesort

```
void bubblesort(int a[]) {
  for (int i = 0; i < a.length; i++) {
    for (int j = a.length - 1; j > i; j--) {
      if (a[j-1] > a[j]) {
        int tmp = a[j-1];
        a[j-1] = a[j];
        a[j] = tmp;
      }
    }
  }
}
```

5	7	5	6	1	8	2

5	7	5	6	1	8	2
5	7	5	6	1	2	8
5	7	5	1	6	2	8
5	7	1	5	6	2	8
5	1	7	5	6	2	8
1	5	7	5	6	2	8

1	5	7	5	6	2	8
1	5	7	5	2	6	8
1	5	7	2	5	6	8
1	5	2	7	5	6	8
1	2	5	7	5	6	8

1	2	5	7	5	6	8
1	2	5	5	7	6	8

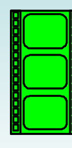
1	2	5	5	7	6	8

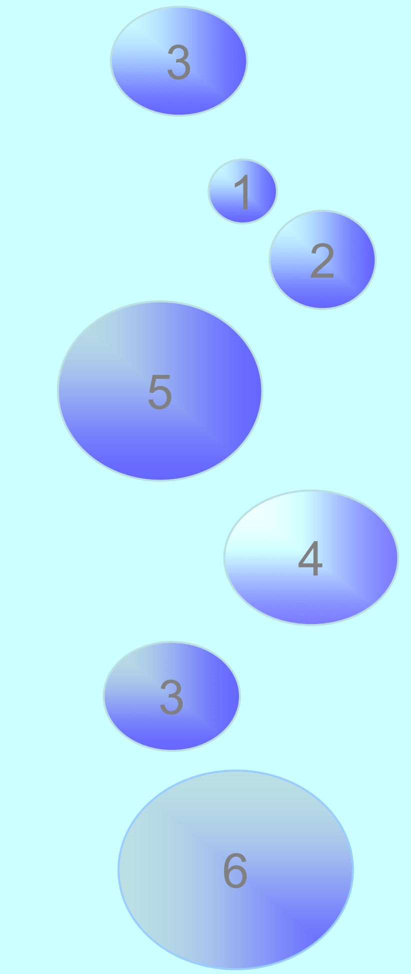
1	2	5	5	6	7	8

Letzte Schritt unnötig

1	2	5	5	6	7	8

Bubblesort





Bubblesort

```
void bubblesort(int a[]) {
  for (int i = 0; i < a.length - 1; i++) {
    for (int j = a.length - 1; j > i; j--) {
      if (a[j-1] > a[j]) {
        int tmp = a[j-1];
        a[j-1] = a[j];
        a[j] = tmp;
      }
    }
  }
}
```

$T(n)=(n-1) + (n-2) + \ldots + 2 + 1 = O(n^2)$ (siehe Direktes Einfügen)

Anzahl Iterationen inneres-for für i=0

Anzahl Iterationen inneres-for für i=1

Variante / Shakersort

sortiert und kleiner rechts

sortiert und größer links

1. Gehe von rechts bis nach links

vertausche dabei die Karten, wenn linke größer als rechte ist

sortiert und kleiner rechts

sortiert und größer links

2. Gehe von links bis nach rechts

vertausche dabei die Karten, wenn rechte kleiner als linke ist

sortiert und kleiner rechts

sortiert und größer links

Shakersort

$$T(n)=\overleftarrow{(n-1)} + \overrightarrow{(n-2)} + \overleftarrow{(n-3)} + \overrightarrow{(n-4)} + \ldots + 2 + 1 = O(n^2)$$

Aufwand gleich schlecht wie Bubblesort
und Implementierung komplizierter geworden
➔keine Vorteile, nur Nachteile

Vergleich

	Bubblesort	Direkte Auswahl	Direktes Einfügen	Mergesort
Schlimmster Fall	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$
Bester Fall	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n \log_2 n)$
Mittlerer Fall	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$

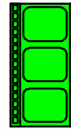
Vergleich

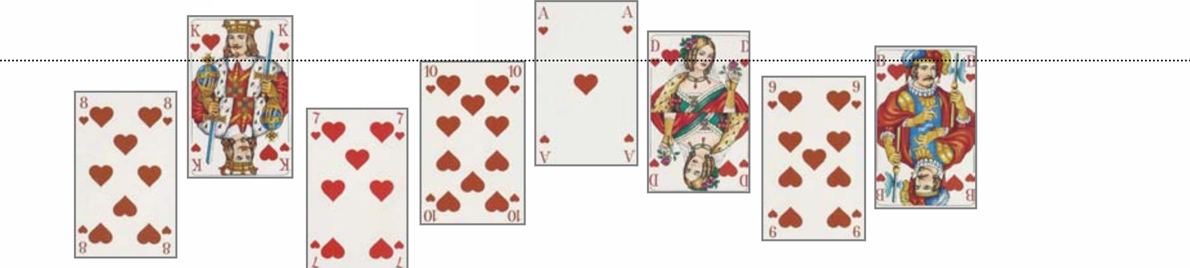
Bubblesort	Direkte Auswahl	Direktes Einfügen
• Im schlimmsten Fall $O(n^2)$ teuere Vertauschoperationen •Immer $O(n^2)$ Vergleichsoperationen	• Immer $O(n^2)$ viele Vergleichsoperationen	• Viele - im schlimmsten Fall $O(n^2)$- teuere Vertauschoperationen
	•O(n) Vertauscheoperationen im schlimmsten Fall	• O(n) bei (fast) sortierten Folgen

Inhalt

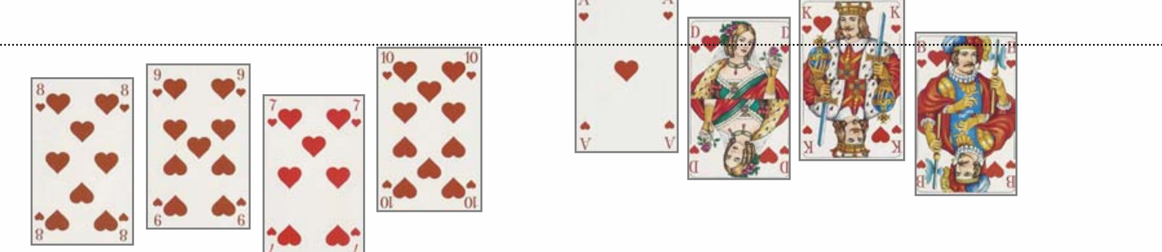
- Sortieren
  - Sortieren durch direktes Einfügen
  - Mergesort
  - Sortieren durch direkte Auswahl
  - Bubblesort
  - Quicksort

Quicksort

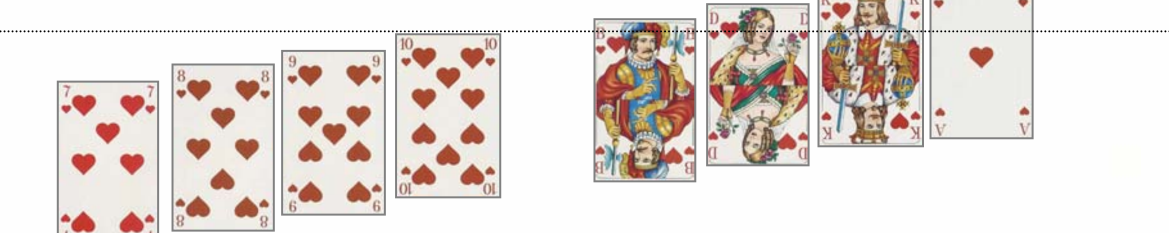




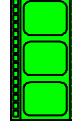
1. Problem in zwei Probleme teilen



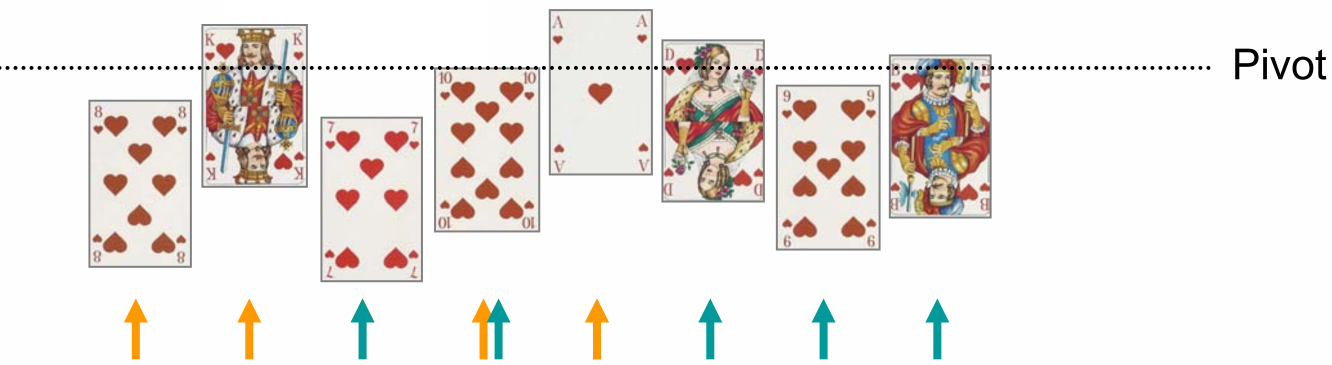
2. Probleme getrennt lösen (rekursiv)



3. Teilprobleme zusammensetzen (Beherrschen)

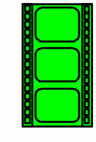


Quicksort



1. Wähle ein Element (Pivot-Element), Beispielsweise 10
2. Gehe wie folgt von beiden Enden zum mittleren Element:
   - Suche sequentiell von links erste Element >= dem Pivot-Element
   - Suche sequentiell von rechts erste Element <= dem Pivot-Element
   - Falls sich Zeiger noch nicht überkreuzt haben:
     tausche beide Elemente aus und bewege Zeiger aufeinander zu
3. Wiederhole Schritt 2 solange, bis Zeiger sich überkreuzt haben

Wiederhole dieses Verfahren (rekursiv) für alle Elemente links bis zum rechten Zeiger und für alle Elemente rechts bis zum linken Zeiger



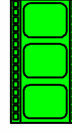
Quicksort

Rekursionstiefe

0

1

2

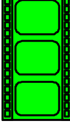


Quicksort

0

1

2



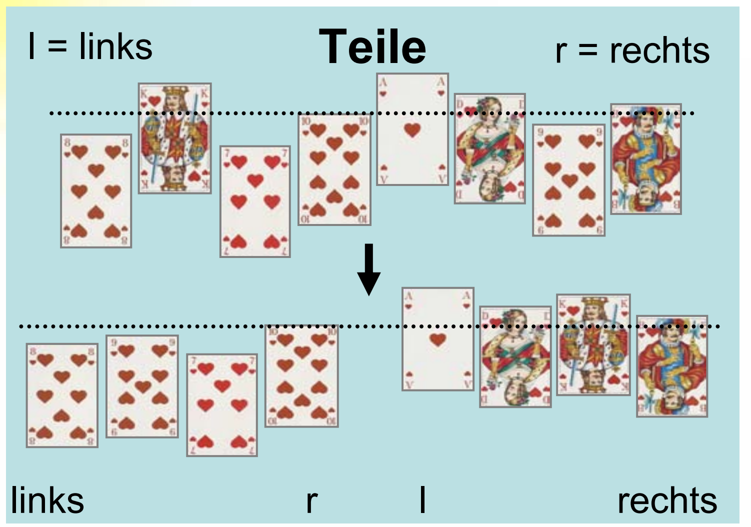
Quicksort

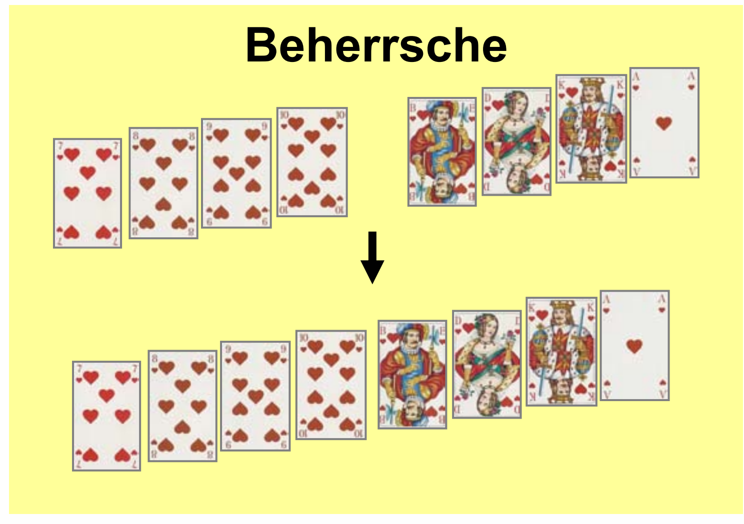
0

1

2

Quicksort






```
void quicksort(int a[], int links, int rechts) {
      if (links < rechts) {
        int l = links;
        int r = rechts;
        // Problem partitionieren in zwei Teilprobleme
        quicksort(a, links, r);
        quicksort(a, l, rechts);
        // Beherrschen: nichts tun
      }
    }
```

Quicksort

1. Wähle ein Element (Pivot-Element), Beispielsweise 10
2. Gehe wie folgt von beiden Enden zum mittleren Element:
   - Suche sequentiell von links erste Element >= dem Pivot-Element
   - Suche sequentiell von rechts erste Element <= dem Pivot-Element
   - Falls sich Zeiger noch nicht überkreuzt haben:
     tausche beide Elemente aus und bewege Zeiger aufeinander zu
3. Wiederhole Schritt 2 solange, bis Zeiger sich überkreuzt haben

```
void quicksort(int a[], int links, int rechts) {
  if (links < rechts) {
    int l = links;
    int r = rechts;
    int pivot = a[ ( links + rechts) / 2 ];
    while (l < r) {
      while ( a[l] < pivot) l++;
      while ( a[r] > pivot) r--;
      if (l <= r) {
        vertauschen(a, l, r); l++; r--;
      }
    }
    quicksort(a, links, r);
    quicksort(a, l, rechts);
  }
}
```

„irgend ein Element wählen“

Quicksort

```
void quicksort(int a[], int links, int rechts) {
  if (links < rechts) {
    int l = links;
    int r = rechts;
    int pivot = a[ ( links + rechts) / 2 ];
    while (l < r) {
      while ( a[l] < pivot) l++;
      while ( a[r] > pivot) r--;
      if (l <= r) {
        vertauschen(a, l++, r--);
      }
    }
    quicksort(a, links, r);
    quicksort(a, l, rechts);
  }
}
```

5	7	5	6	1	8	2

Quicksort

links = 0 | pivot | rechts = 6

quicksort(a, 0, a.length-1)

zu vertauschende Elemente

5	7	5	6	1	8	2
5	7	5	6	1	8	2
5	2	5	6	1	8	7
5	2	5	1	6	8	7
			r	l		

quicksort(a, 0, 3)

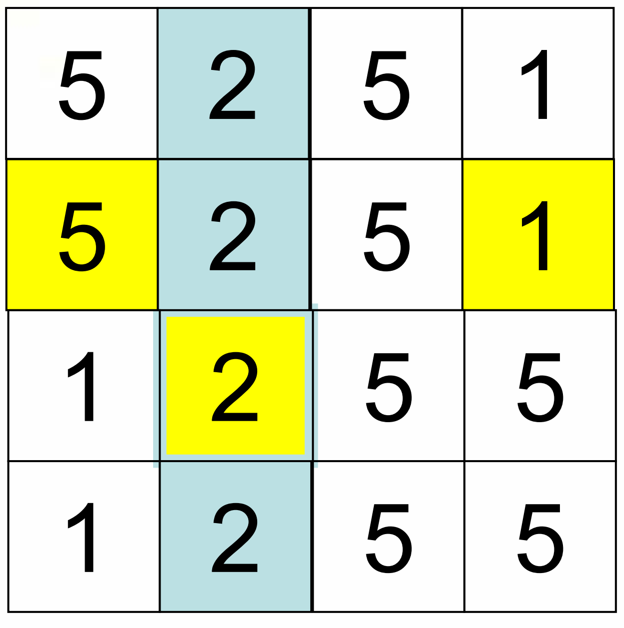
5	2	5	1

quicksort(a, 4, 6)

6	8	7

Quicksort

quicksort(a, 0, 3)



r l

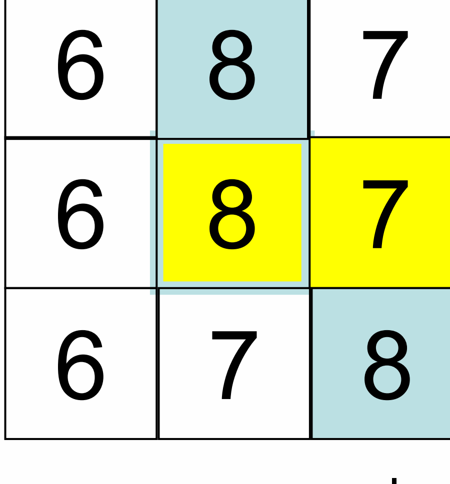
quicksort(a, 0, 0)

1

quicksort(a, 2, 3)

5	5

quicksort(a, 4, 6)



quicksort(a, 4, 5)

6	7

quicksort(a, 6, 6)

8

Quicksort

quicksort(a, 0, 0)	quicksort(a, 2, 3)		quicksort(a, 4, 5)		quicksort(a, 6, 6)
1	5	5	6	7	8
1	5	5	6	7	8
1	5	5	6	7	8

r l   l r   l r   r l

quicksort(a, -1, 0)

quicksort(a, 0, 1)

quicksort(a, 1, 1)

quicksort(a, 2, 2)

quicksort(a, 4, 3)

quicksort(a, 5, 5)

quicksort(a, 5, 6)

quicksort(a, 6, 7)

links > rechts: Rekursion wird abgebrochen

Quicksort

- Zubewegen von l und r muss beim Pivot-Element stoppen
- a[l] <= pivot statt a[l] < pivot nicht korrekt

```
void quicksort(int a[], int links, int rechts) {
  if (links < rechts) {
    int l = links;
    int r = rechts;
    int pivot = a[ ( links + rechts) / 2 ];
    while (l < r) {
      while ( a[l] <= pivot) l++;
      while ( a[r] >= pivot) r--;
      if (l <= r) {
          vertauschen(a, l++, r--);
      }
    }
    quicksort(a, links, r);
    quicksort(a, l, rechts);
  }
}
```

pivot = 2

1	2	1

l

l wandert bis nach rechts+1 und verursacht IndexOutOfBoundException

Quicksort

```
void quicksort(int a[], int links, int rechts) {
  if (links < rechts) {
    int l = links;
    int r = rechts;
    int pivot = a[ ( links + rechts) / 2 ];
    while (l < r) {
      while ( a[l] < pivot) l++;
      while ( a[r] > pivot) r--;
      vertauschen(a, l++, r--);
     }
    quicksort(a, links, r);
    quicksort(a, l, rechts);
  }
}
```

Bedingung (l <= r) muss überprüft werden

pivot = 2

1	2	1	2

Partitionierung bis vor Vertauschung

1	2	1 (r)	2 (l)

vertauschen

1	2 (r)	2	1

Linke Teilproblem

l Rechte Teilproblem leer

nicht sortiert +
nicht rekursiv weiter behandelt

Quicksort

```
void quicksort(int a[], int links, int rechts) {
  if (links < rechts) {
    int l = links;
    int r = rechts;
    int pivot = a[ ( links + rechts) / 2 ];
    while (l < r) {
      while ( a[l] < pivot) l++;
      while ( a[r] > pivot) r--;
      if (l <= r) {
        vertauschen(a, l++, r--);
      }
    }
    quicksort(a, links, r);
    quicksort(a, l, rechts);
  }
}
```

O(n) Schritte

Zeitaufwand im **günstigsten** Fall:

- `pivot` ist mittleres Element und nach Partitionieren in der Mitte
- Teilprobleme sind etwa gleich gross

$$T(n) = 2\,T(n/2) + n \cdot c$$
$$= O(n \log_2 n)$$

Quicksort

Zeitaufwand im **schlechtesten** Fall:

- `pivot` ist kleinstes (größtes) Element und nach partitionieren ganz links (rechts)
- Ein Teilproblem der Größe 1 und eines der Größe n-1

n

n-1

n-2

n-3

1

$$
\begin{aligned}
T_{wc}(n) &= T_{wc}(1) + T_{wc}(n-1) + n \cdot c \\
&= c + T_{wc}(n-1) + n \cdot c \\
&= T_{wc}(n-1) + c + n \cdot c \\
&= T_{wc}(n-2) + c + c + (n-1) \cdot n \cdot c \\
&= \dots \\
&= T_{wc}(0) + n \cdot c + (1+2+\dots+(n-1)+n) \cdot c \\
&= c + n \cdot c + (1 \cdot 2+\dots+(n-1)) \cdot n \cdot c \\
&= O(n^2)
\end{aligned}
$$

Quicksort

Beim **schlechtesten** Fall ist die Auswahl des pivot Elements entscheidend:

- Median (mittleres Element) wählen
- Bei einer *sortierten* Folge von Zahlen ist der **Median** der Wert, der in der Mitte liegt

1, 2, 3, 7, 8, 8, 8, 10, 11, 56, 87, 99, 1923

Bestimmen des Median in einer Folge n *unsortierter* Zahlen?

1923, 2, 8, 87, 3, 56, 1, 8, 11, 10, 7, 99, 8

Aufwand im schlimmsten Fall: O(n) (ohne Beweis)
Aufwendige statistische Verfahren oder mit komplexen Datenstrukturen

$$T_{wc}(n) = 2\,T_{wc}(n/2) + n \cdot c = O(n \log_2 n)$$

Verfahren ist für die Praxis zu aufwendig und rentiert sich nicht.
(der konstante Faktor wird zu hoch im Vergleich zu, z.B., Mergesort)

Quicksort

Zeitaufwand im **mittleren** Fall (1)
- Elemente zufällig verteilt
- Im Durchschnitt zwei halb große Teilprobleme
  (ohne Beweis)

Zeitaufwand im **mittleren** Fall (2)
- Pivot zufällig wählen
- Im Durchschnitt zwei halb große Teilprobleme
  (ohne Beweis)

$$T(n) = 2\,T(n/2) + n \cdot c$$
$$= O(n \log_2 n)$$

- Quicksort benötigt im Gegensatz zu Mergesort keinen zusätzlichen Speicherplatz (außer für die Rekursion).
- Im mittleren Fall ist Quicksort etwa 2-3 mal so schnell wie Mergesort, da kein zusätzlicher Speicher und weniger Speicherbewegungen
- Mergesort besser für externe Speichermedien (Bandlaufwerke, Festplatten) und Folgen von Elementen ohne indizierten (d.h. direkten) Zugriff

Vergleich

	Bubblesort	Direkte Auswahl	Direktes Einfügen	Quicksort (1960, Hoare)	Mergesort
Schlimmster Fall	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$
Bester Fall	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Mittlerer Fall	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$

Vergleich

Andere Sortierverfahren
(siehe z.B. Niklaus Wirth, Algorithmen und Datenstrukturen)

	Shakersort (mod. Bubblesort)	**Shellsort (1959, Shell)**	**Heapsort**
Schlimmster Fall	$O(n^2)$	???	$O(n \log_2 n)$
Bester Fall	$O(n^2)$	vermutlich größer als $O(n \log_2 n)$	$O(n \log_2 n)$
Mittlerer Fall	$O(n^2)$	$O(n^{1,25})$ (experimentell gefundener Wert)	$O(n \log_2 n)$

Untere Schranke

- Untere Schranke *für ein Problem*
  - Zeitaufwand im schlechtesten Fall, den jeder Algorithmus *mindestens* benötigt, um das Problem zu lösen
- Untere Schranke für Sortieren
  - Kann es ein Sortierverfahren geben, das im schlechtesten Fall schneller als $O(n \log_2 n)$ ist?
  - Genauer: Kann es ein Sortierverfahren geben, das im schlechtesten Fall schneller als $O(n \log_2 n)$ ist unter Verwendung von *Vergleichsoperationen (<,>, usw.) und Ja/Nein Entscheidungen (if)*?
- Mindestaufwand O(n)
  - Zum Überprüfen, ob Zahlen sortiert sind

Untere Schranke

1, 2, 3
1, 3, 2
3, 1, 2
2, 1, 3
2, 3, 1
3, 2, 1

- Es seien *n* verschiedene Zahlen zu sortieren
- Jedes Sortierverfahren muss die *n*! Permutationen dieser *n* Zahlen *individuell* voneinander unterscheiden können
- Anzahl Schritte, um *n*! verschiedene Zahlen individuell zu unterscheiden ist $\log_2(n!)$ (Entscheidungsbaum, gleich)
- Es gilt: $n! >= (n/2)^{(n/2)}$ (gleich)
- $\log_2(n!) >= \log_2((n/2)^{n/2})$

  $= n/2 \cdot \log_2(n/2)$ ($\log a^b = b \cdot \log a$)

  $= n/2 \cdot \log_2(n) - n/2 \cdot log(2)$

  $= O(n \cdot \log_2 n)$

➢ Jedes Sortierverfahren hat Zeitaufwand von mindestens $O(n \cdot \log_2 n)$

Entscheidungsbaum

Anzahl Schritte **drei** *verschiedene* Zahlen a_1, a_2 , a_3 zu unterscheiden

Jeder Algorithmus muss Zahlen irgendwann vergleichen, im Minimum in jedem Schritt

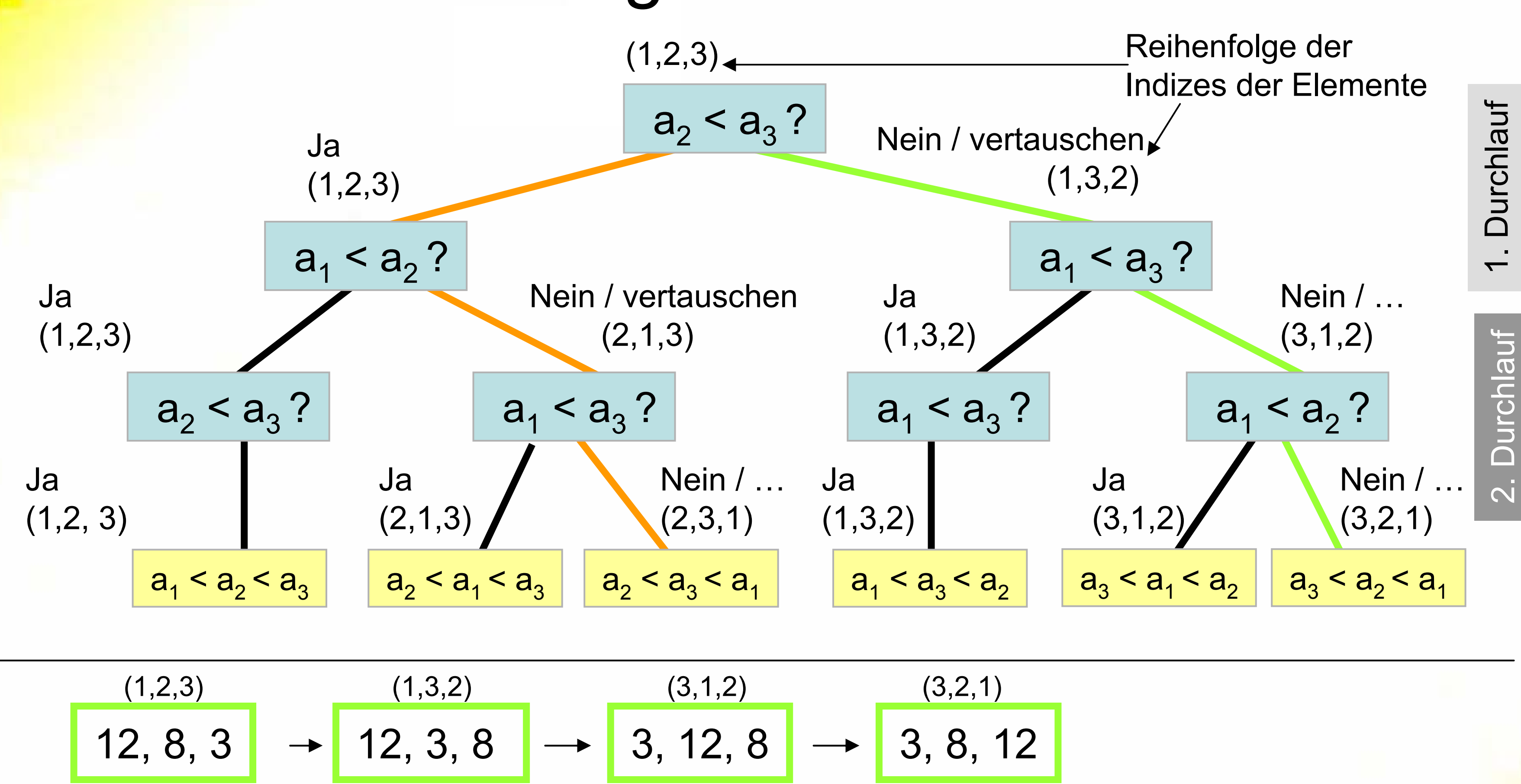
$a_1 < a_2$?

- Ja: $a_2 < a_3$?
  - Ja: $a_1 < a_2 < a_3$
  - Nein: $a_1 < a_3$?
    - Ja: $a_1 < a_3 < a_2$
    - Nein: $a_3 < a_1 < a_2$
- Nein: $a_1 < a_3$?
  - Ja: $a_2 < a_1 < a_3$
  - Nein: $a_1 < a_3$?
    - Ja: $a_2 < a_3 < a_1$
    - Nein: $a_3 < a_2 < a_1$

3! = 2·3=6 Blätter, jedes Blatt stellt eine individuelle Lösung dar.
Jedes Verfahren produziert *mindestens* einen Baum mit 6 Blätter.

Entscheidungsbaum / Bubblesort

Entscheidungsbaum / Bubblesort



1. Durchlauf v. Bubblesort

2. Durchlauf

11, 2, 7 → 11, 2, 7 → 2, 11, 7 → 2, 7, 11

(1,2,3) (1,2,3) (2,1,3) (2,3,1)

Untere Schranke

Anzahl Schritte, um *n* verschiedene Zahlen $a_1, a_2, \ldots, a_n$ zu unterscheiden

Kleinste mögliche Baum hat n! Blätter, da n! *individuelle* Lösungen

Tiefe des Baumes $\log_2(n!)$

Untere Schranke

$n! = \quad n\cdot (n-1)\cdot (n-2) \cdot\ldots\cdot (n+1)/2\cdot n/2\cdot\ldots\cdot 2\cdot 1$

$\quad >= n\cdot (n-1)\cdot (n-2) \cdot\ldots\cdot (n/2)$ (Produkt schon bei Hälfte abbrechen)

$\quad >= (n/2) \cdot(n/2) \cdot\ldots\cdot(n/2)$ ($n >= n/2, \ldots, (n/2+1) >= (n/2)$)

$\quad = \quad (n/2)^{(n/2)}$

Untere Schranke

- Sortierproblem hat untere Schranke $O(n \cdot \log_2 n)$
- Mergesort ist *optimal*, da es das Sortierproblem in jedem Fall mit Zeitaufwand $O(n \cdot \log_2 n)$ löst