

7

Sortieren

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

Outline

7 Sortieren

- 7.1 Was bedeutet sortiert?
- 7.2 Primitive Sortieralgorithmen
- 7.3 Analyse der primitiven Verfahren
- 7.4 Mergesort
- 7.5 Mergesort Optimierungen
- 7.6 Quicksort
- 7.7 Quicksort Analyse & Randomisierte Algorithmen
- 7.8 Untere Schranke
- 7.9 Priority Queues & Heapsort
- 7.10 System sorts

7.1 Was bedeutet sortiert?

Sortieren

Gegeben: Array $A[0..n)$ von n Objekten

Ziel: Array so permutieren, dass Objekte aufsteigend sortiert sind.

Sortieren

Gegeben: Array $A[0..n)$ von n Objekten

Ziel: Array so permutieren, dass Objekte aufsteigend sortiert sind.

Beispiele:

- ▶ Hochschulen in Deutschland nach Gründungsjahr

https://de.wikipedia.org/wiki/Liste_der_Hochschulen_in_Deutschland

Sortieren

Gegeben: Array $A[0..n)$ von n Objekten

Ziel: Array so permutieren, dass Objekte aufsteigend sortiert sind.

Beispiele:

- ▶ Hochschulen in Deutschland nach Gründungsjahr
https://de.wikipedia.org/wiki/Liste_der_Hochschulen_in_Deutschland
- ▶ Züge nach Abfahrtszeit
<https://www.bahnhof.de/marburg-lahn/abfahrt>
- ▶ Index in Büchern
- ▶ Arbeitsgruppen des FB12, alphabetisch nach Nachnamen der Professor:innen
<https://www.uni-marburg.de/de/fb12/arbeitsgruppen>
- ▶ ...

Sortieren

Gegeben: Array $A[0..n)$ von n Objekten, Sortierkriterium

Ziel: Array so permutieren, dass Objekte aufsteigend sortiert sind.

Beispiele:

- ▶ Hochschulen in Deutschland nach Gründungsjahr
https://de.wikipedia.org/wiki/Liste_der_Hochschulen_in_Deutschland
- ▶ Züge nach Abfahrtszeit
<https://www.bahnhof.de/marburg-lahn/abfahrt>
- ▶ Index in Büchern
- ▶ Arbeitsgruppen des FB12, alphabetisch nach Nachnamen der Professor:innen
<https://www.uni-marburg.de/de/fb12/arbeitsgruppen>
- ▶ ...

Typischer Fall: verschiedene Sortierkriterien denkbar/möglich

↪ muss mit angegeben werden

Sample sort client 1

Goal. Sort **any** type of data.

Ex 1. Sort random real numbers in ascending order.

 seems artificial (stay tuned for an application)

```
public class Experiment
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Double[] a = new Double[N];
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
        Insertion.sort(a);
        for (int i = 0; i < N; i++)
            StdOut.println(a[i]);
    }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Sample sort client 2

Goal. Sort **any** type of data.

Ex 2. Sort strings in alphabetical order.

```
public class StringSorter
{
    public static void main(String[] args)
    {
        String[] a = StdIn.readAllStrings();
        Insertion.sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.println(a[i]);
    }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes
```

```
% java StringSorter < words3.txt
all bad bed bug dad ... yes yet zoo
[suppressing newlines]
```

Sample sort client 3

Goal. Sort **any** type of data.

Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;

public class FileSorter
{
    public static void main(String[] args)
    {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOut.println(files[i].getName());
    }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

Totale Ordnung

Damit Sortieren wohldefiniert ist, benötigen wir ein sinnvolles Sortierkriterium.

Informell: Alle Objekte sind vergleichbar und Vergleiche ergeben keine Widersprüche.

Totale Ordnung

Damit Sortieren wohldefiniert ist, benötigen wir ein sinnvolles Sortierkriterium.

Informell: Alle Objekte sind vergleichbar und Vergleiche ergeben keine Widersprüche.

Formal: Für eine Menge U (Universum) ist eine Relation $\leq \subset U \times U$ eine *totale Ordnung*, falls folgende Bedingungen erfüllt sind

$$(x, y) \in \leq \quad x \leq y$$

1. **Reflexivität:** $\forall x \in U : x \leq x$
2. **Antisymmetrie:** $\forall x, y \in U : x \leq y \wedge y \leq x \implies x = y$
3. **Transitivität:** $\forall x, y, z \in U : x \leq y \wedge y \leq z \implies x \leq z$
4. **Totalität:** $\forall x, y \in U : x \leq y \vee y \leq x$

Totale Ordnung

Damit Sortieren wohldefiniert ist, benötigen wir ein sinnvolles Sortierkriterium.

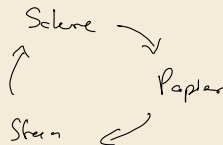
Informell: Alle Objekte sind vergleichbar und Vergleiche ergeben keine Widersprüche.

Formal: Für eine Menge U (Universum) ist eine Relation $\leq \subset U \times U$ eine *totale Ordnung*, falls folgende Bedingungen erfüllt sind

1. **Reflexivität:** $\forall x \in U : x \leq x$
2. **Antisymmetrie:** $\forall x, y \in U : x \leq y \wedge y \leq x \implies x = y$
3. **Transitivität:** $\forall x, y, z \in U : x \leq y \wedge y \leq z \implies x \leq z$
4. **Totalität:** $\forall x, y \in U : x \leq y \vee y \leq x$

Anti-Beispiele:

- ▶ Schere-Stein-Papier Gewinne ⚡ Transitivität
- ▶ $n \mid m$ (Teilbarkeit, „ n teilt m “) ⚡ Totalität



Clicker Question

Haben Sie schon einmal (beim Programmieren) eine Sortiermethode aufgerufen und explizit die gewünschte totale Ordnung mit angegeben? Falls Ja, in welcher Sprache?



- A** Nee, bisher nicht.
- B** Java
- C** Python
- D** C
- E** C++
- F** C#
- G** Javascript
- H** Perl
- I** sonstige



→ sli.do/cs210

Callbacks

Goal. Sort **any** type of data (for which sorting is well defined).

Q. How can `sort()` know how to compare data of type `Double`, `String`, and `java.io.File` without any information about the type of an item's key?

Callback = reference to executable code.

- Client passes array of objects to `sort()` function.
- The `sort()` function calls object's `compareTo()` method as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: class-type functors.
- C#: delegates.
- Python, Perl, ML, Javascript: first-class functions.

Callbacks: roadmap

client

```
public class StringSorter
{
    public static void main(String[] args)
    {
        String[] a = StdIn.readAllStrings();
        Insertion.sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.println(a[i]);
    }
}
```

data-type implementation

```
public class String
implements Comparable<String>
{
    ...
    public int compareTo(String b)
    {
        ...
        return -1;
        ...
        return +1;
        ...
        return 0;
    }
}
```

Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
    public int compareTo(Item that);
}
```

key point: no dependence
on String data type

sort implementation

```
public static void sort(Comparable[] a)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0; j--)
            if (a[j].compareTo(a[j-1]) < 0)
                exch(a, j, j-1);
            else break;
}
```

Comparable API

Implement `compareTo()` so that `v.compareTo(w)`

- Defines a total order.
- Returns a negative integer, zero, or positive integer if v is less than, equal to, or greater than w , respectively.
- Throws an exception if incompatible types (or either is `null`).



less than (return -1)



equal to (return 0)



greater than (return +1)

$v.compareTo(w)$

$v - w \geq 0$

Built-in comparable types. Integer, Double, String, Date, File, ...

User-defined comparable types. Implement the Comparable interface.

Implementing the Comparable interface

Date data type. Simplified version of java.util.Date.

```
public class Date implements Comparable<Date>
{
    private final int month, day, year;

    public Date(int m, int d, int y)
    {
        month = m;
        day   = d;
        year  = y;
    }

    public int compareTo(Date that)
    {
        if (this.year < that.year ) return -1;
        if (this.year > that.year ) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day   < that.day   ) return -1;
        if (this.day   > that.day   ) return +1;
        return 0;
    }
}
```

only compare dates
to other dates

```
x.compareTo(y) {
    return x - y;
}
```

Comparable vs. Comparator

Für die Eingangsbeispiele wäre Comparable nicht bequem!

- ▶ Möchten oft die **gleichen** Objekte nach unterschiedlichen Kriterien sortieren
- ▶ Comparable erlaubt ja aber nur **eine** Implementierung von compareTo!

Comparable vs. Comparator

Für die Eingangsbeispiele wäre Comparable nicht bequem!

- ▶ Möchten oft die **gleichen** Objekte nach unterschiedlichen Kriterien sortieren
- ▶ Comparable erlaubt ja aber nur **eine** Implementierung von compareTo!

↪ Java erlaubt alternativ, den Sortiermethoden ein **Comparator**-Objekt zu übergeben

```
1 public interface Comparator<T> {  
2     int compare(T v, T w);  
3 }
```

Gleiche Semantik:

$$c.compare(v, w) \leq 0 \equiv v \leq w \iff v - w \leq 0$$

7.2 Primitive Sortieralgorithmen

Sortieralgorithmen

Obwohl das Ergebnis eindeutig ist, gibt es *vielen* Möglichkeiten für Sortieralgorithmen.

Wir beginnen mit den naheliegendsten und ältesten Ideen.

(siehe Sedgewick & Wayne für mehr Details für diesen Abschnitt)

Sortieralgorithmen

Obwohl das Ergebnis eindeutig ist, gibt es *viele* Möglichkeiten für Sortieralgorithmen.

Wir beginnen mit den naheliegendsten und ältesten Ideen.

(siehe Sedgewick & Wayne für mehr Details für diesen Abschnitt)

Diese entstammen oft auch manuellen Verfahren zum Sortieren physischer Objekte

↪ Typisches Phänomen: Verfügbarkeit und Effizienz von Zugriffsoperationen bestimmt welcher Sortieralgorithmus schneller ist

Sortieralgorithmen

Obwohl das Ergebnis eindeutig ist, gibt es *vielen* Möglichkeiten für Sortieralgorithmen.

Wir beginnen mit den naheliegendsten und ältesten Ideen.

(siehe Sedgewick & Wayne für mehr Details für diesen Abschnitt)

Diese entstammen oft auch manuellen Verfahren zum Sortieren physischer Objekte

↪ Typisches Phänomen: Verfügbarkeit und Effizienz von Zugriffsoperationen bestimmt welcher Sortieralgorithmus schneller ist

Verwenden hauptsächlich folgende Elementaroperationen

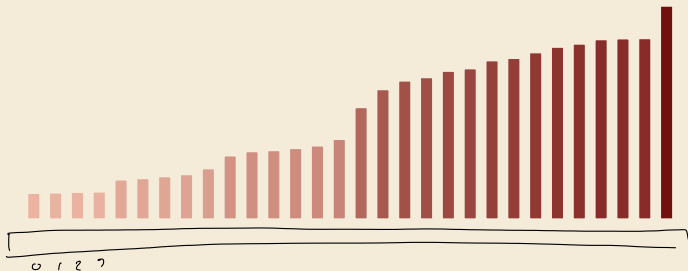
- ▶ $\text{less}(v, w)$ um zu testen ob $v < w$ (mittels Comparator)
- ▶ $\text{swap}(i, j)$ tauscht $A[i]$ und $A[j]$

```
1 static <Elem> boolean less(final Elem v, final Elem w, final Comparator<Elem> c) {  
2     return c.compare(v, w) < 0;  
3 }  
4  
5 static <Elem> void swap(final Elem[] A, final int i, final int j) {  
6     Elem tmp = A[i]; A[i] = A[j]; A[j] = tmp;  
7 }
```

$v < w$?

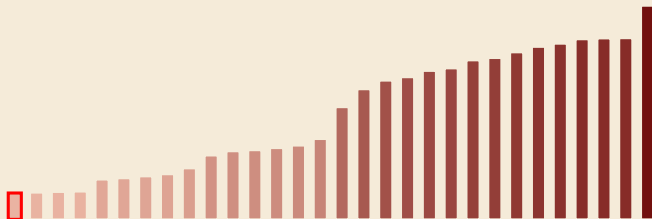
Selectionsort

- 💡 **Idee:** In einer sortierten Liste ist das **Minimum** an erster Stelle.
Finde iterative das Minimum und entferne es.



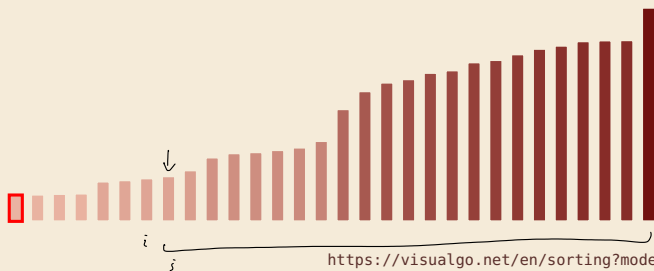
Selectionsort

- 💡 **Idee:** In einer sortierten Liste ist das **Minimum** an erster Stelle.
Finde iterative das Minimum und entferne es.



Selectionsort

- 💡 **Idee:** In einer sortierten Liste ist das **Minimum** an erster Stelle.
Finde iterative das Minimum und entferne es.



</> **Code:**

```
1 static <Elem> void selectionSort(Elem[] A, Comparator<Elem> c) {
2     int n = A.length;
3     for (int i = 0; i < n; ++i) {
4         Elem min = A[i]; int minPos = i;
5         for (int j = i+1; j < n; ++j)
6             if (less(A[j], min, c)) {
7                 min = A[j]; minPos = j;
8             }
9         swap(A, i, minPos);
10    }
11 }
```

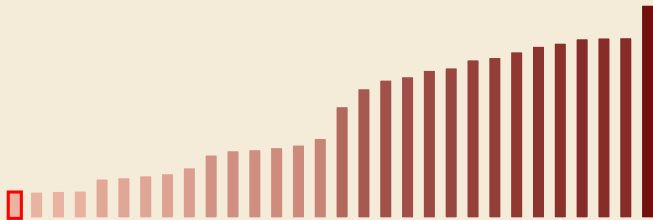
$A[0..i)$ schon sortiert

$i \rightarrow i+1$

$\leftarrow A[\text{minPos}] = \min A[i..n)$

Selectionsort

- 💡 **Idee:** In einer sortierten Liste ist das **Minimum** an erster Stelle.
Finde iterative das Minimum und entferne es.



</> **Code:**

<https://visualgo.net/en/sorting?mode=Selection>

```
1 static <Elem> void selectionSort(Elem[] A, Comparator<Elem> c) {
2     int n = A.length;
3     for (int i = 0; i < n; ++i) {
4         Elem min = A[i]; int minPos = i;
5         for (int j = i+1; j < n; ++j)
6             if (less(A[j], min, c)) {
7                 min = A[j]; minPos = j;
8             }
9         swap(A, i, minPos);
10    }
11 }
```

Vergleiche
Swap

1 }
~ n - i
1 }
$$\sim \sum_{i=0}^{n-1} n-i \sim \frac{n^2}{2}$$

$$\sum_{i=0}^{n-1} 1 \sim n$$

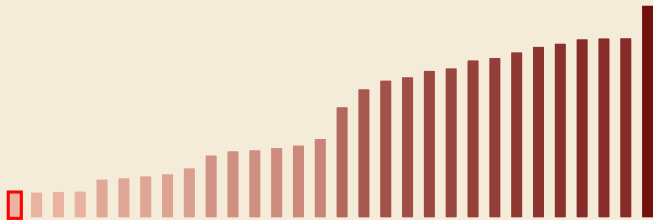
🎯 **Korrektheit:**

Invariante: $A[0..i]$ enthält die i kleinsten Elemente in sortierter Reihenfolge

🏔️ **Analyse:**

Selectionsort

- 💡 **Idee:** In einer sortierten Liste ist das **Minimum** an erster Stelle.
Finde iterative das Minimum und entferne es.



</> **Code:**

<https://visualgo.net/en/sorting?mode=Selection>

```
1 static <Elem> void selectionSort(Elem[] A, Comparator<Elem> c) {
2     int n = A.length;
3     for (int i = 0; i < n; ++i) {
4         Elem min = A[i]; int minPos = i;
5         for (int j = i+1; j < n; ++j)
6             if (less(A[j], min, c) ) {
7                 min = A[j]; minPos = j;
8             }
9         swap(A, i, minPos);
10    }
11 }
```

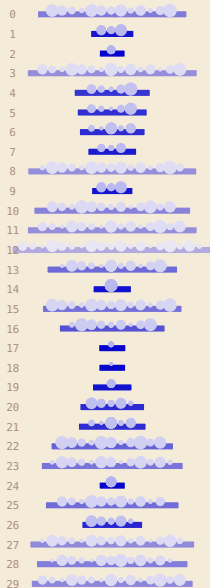
🎯 **Korrektheit:**

Invariante: $A[0..i]$ enthält die i kleinsten Elemente in sortierter Reihenfolge

🏔️ **Analyse:**

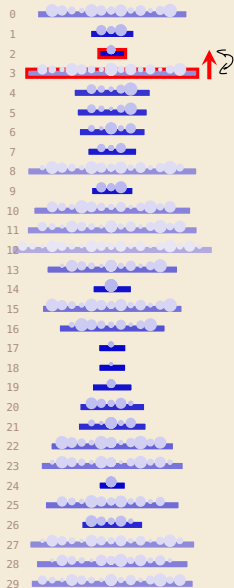
Anzahl Vergleiche $\sim \frac{1}{2}n^2$

Bubblesort



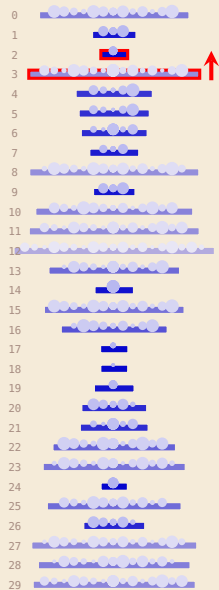
💡 **Idee:** In paarweisen Vergleichen steigt das ^{in viskosem Gel} leichtere Element wie eine Luftblase nach oben. Verfahrene in Iterationen über die Eingabe.

Bubblesort



💡 **Idee:** In paarweisen Vergleichen steigt das leichtere Element wie eine Luftblase nach oben. *in viskosem Gel*
Verfahre in Iterationen über die Eingabe.

Bubblesort

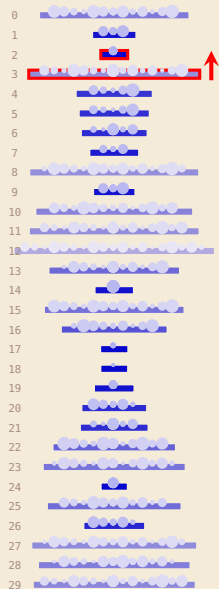


💡 **Idee:** In paarweisen Vergleichen steigt das ^{in viskosem Gel} leichtere Element wie eine Luftblase nach oben. Verfahrene in Iterationen über die Eingabe.

</> **Code:** <https://visualgo.net/en/sorting?mode=Bubble> (gespiegelt)

```
1 static <Elem> void bubbleSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 0; i <= n-2; ++i)  
4         for (int j = n-1; j >= i+1; --j)  
5             if (less(A[j],A[j-1],c)) swap(A,j,j-1);  
6 }
```

Bubblesort



💡 **Idee:** In paarweisen Vergleichen steigt das leichtere Element wie eine Luftblase nach oben. ↙ in viskosem Gel
Verfahre in Iterationen über die Eingabe.

</> **Code:** <https://visualgo.net/en/sorting?mode=Bubble> (gespiegelt)

```
1 static <Elem> void bubbleSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 0; i <= n-2; ++i)  
4         for (int j = n-1; j >= i+1; --j)  
5             if (less(A[j],A[j-1],c)) swap(A,j,j-1);  
6 }
```

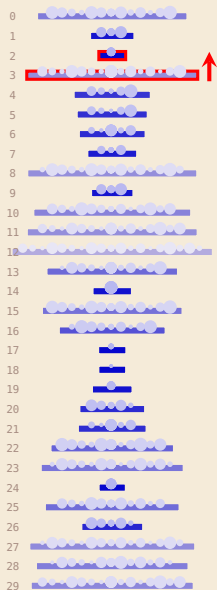
🎯 **Korrektheit:**

Mit jeder Vertauschung näher an sortiert.

Nach jeder Iteration der äußeren Schleife erreicht nächstes Minimum die Oberfläche und kann von Betrachtung ausgeschlossen werden.

🏠 **Analyse:**

Bubblesort



💡 **Idee:** In paarweisen Vergleichen steigt das leichtere Element wie eine Luftblase nach oben. ↙ in viskosem Gel
Verfahre in Iterationen über die Eingabe.

</> **Code:** <https://visualgo.net/en/sorting?mode=Bubble> (gespiegelt)

```
1 static <Elem> void bubbleSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 0; i <= n-2; ++i)  
4         for (int j = n-1; j >= i+1; --j)  
5             if (less(A[j],A[j-1],c)) swap(A,j,j-1);  
6 }
```

$$\left. \begin{array}{l} \text{1 1} \\ \sim n-i \end{array} \right\} \sum_{i=0}^{n-1} n-i \sim \frac{1}{2}n^2$$

🎯 **Korrektheit:**

Mit jeder Vertauschung näher an sortiert.

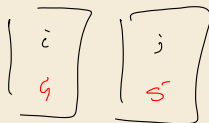
Nach jeder Iteration der äußeren Schleife erreicht nächstes Minimum die Oberfläche und kann von Betrachtung ausgeschlossen werden.

🏠 **Analyse:**

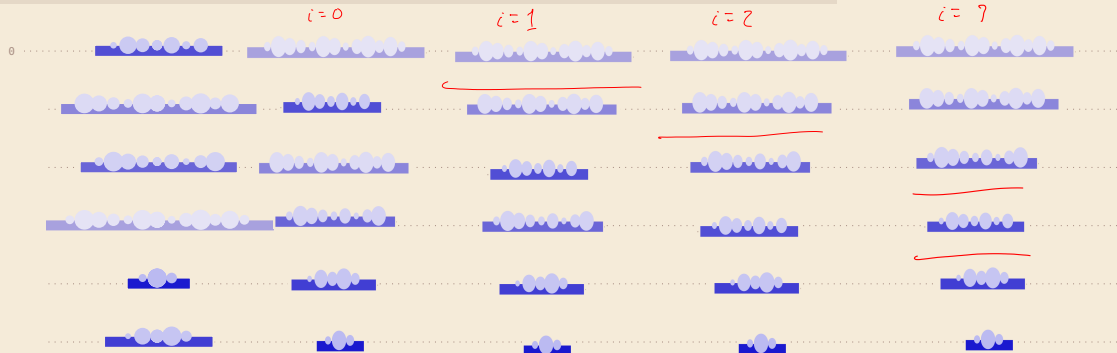
Anzahl Vergleiche $\sim \frac{1}{2}n^2$

Bubblesort – Beispiel

$n=6$



```
1 static <Elem> void bubbleSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 0; i <= n-2; ++i)  
4         for (int j = n-1; j >= i+1; --j)  
5             if (less(A[j], A[j-1], c)) swap(A, j, j-1);  
6 }
```



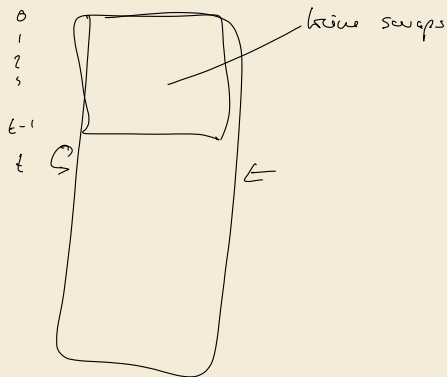
Verbesserter Bubblesort

Obiger Code ist oftmals verschwenderisch / blind!

Verbesserter Bubblesort

Obiger Code ist oftmals verschwenderisch / blind!

Beobachtung: Falls in voriger Iteration die letzte Vertauschung zwischen $A[t]$ und $A[t - 1]$ stattfand, enthält $A[0..t)$ die kleinsten t Elements in sortierter Reihenfolge.



Verbesserter Bubblesort

Obiger Code ist oftmals verschwenderisch / blind!

Beobachtung: Falls in voriger Iteration die letzte Vertauschung zwischen $A[t]$ und $A[t - 1]$ stattfand, enthält $A[0..t)$ die kleinsten t Elements in sortierter Reihenfolge.

```
1 static <Elem> void bubbleSortImproved(Elem[] A, Comparator<Elem> c) {
2     int n = A.length; int t = 0;
3     while (t != n) {
4         int i = t; t = n;
5         for (int j = n-1; j >= i+1; --j)
6             if (less(A[j], A[j-1], c)) {
7                 swap(A, j, j-1);
8                 t = j;
9             }
10    }
11 }
```

☉ **Korrektheit:** Invariante (Z.3): $A[0..t)$ enthält die t kleinsten Elemente in sortierter Reihenfolge

🏠 **Analyse:** später 😊

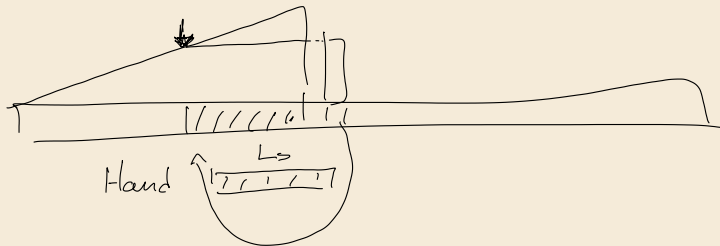
<https://www.wild-inter.net/teaching/ads/bubblesort>

Insertionsort

💡 **Idee:** Kartenspielermethode: Füge Elemente sukzessive in sortierte „Hand“ ein.

Hand = Präfix des Arrays

Einfügen = nach links Vertauschen, bis größer



Insertionsort

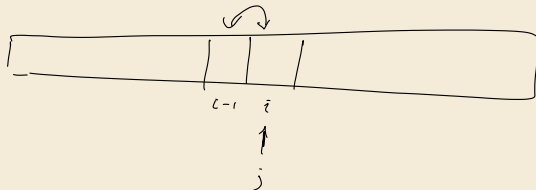
- 💡 **Idee:** Kartenspielermethode: Füge Elemente suzessive in sortierte „Hand“ ein.
Hand = Präfix des Arrays
Einfügen = nach links Vertauschen, bis größer

</> Code:

```
1 static <Elem> void insertionSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 1; i < n; ++i)  
4         for (int j = i; j > 0 && less(A[j], A[j-1], c); --j)  
5             swap(A, j, j-1);  
6 }
```

i = "Größe der Hand"
A[0..i) sortiert

<https://visualgo.net/en/sorting?mode=Insertion>



Insertionsort

- 💡 **Idee:** Kartenspielermethode: Füge Elemente sukzessive in sortierte „Hand“ ein.
Hand = Präfix des Arrays
Einfügen = nach links Vertauschen, bis größer

</> Code:

```
1 static <Elem> void insertionSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 1; i < n; ++i)  
4         for (int j = i; j > 0 && less(A[j], A[j-1], c); --j)  
5             swap(A, j, j-1);  
6 }
```

<https://visualgo.net/en/sorting?mode=Insertion>

© Korrektheit:

Invariante: $A[0..i]$ sortiert (in Z.3)

🏔 Analyse:

Insertionsort

💡 **Idee:** Kartenspielermethode: Füge Elemente sukzessive in sortierte „Hand“ ein.

Hand = Präfix des Arrays

Einfügen = nach links Vertauschen, bis größer

</> **Code:**

```
1 static <Elem> void insertionSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 1; i < n; ++i)  
4         for (int j = i; j > 0 && less(A[j], A[j-1], c); --j)  
5             swap(A, j, j-1);  
6 }
```

<https://visualgo.net/en/sorting?mode=Insertion>

© **Korrektheit:**

Invariante: $A[0..i]$ sortiert (in Z.3)

🏔 **Analyse:**

Anzahl Vergleiche $\leq \frac{1}{2}n^2 + O(n)$

Anzahl Swaps $\leq \frac{1}{2}n^2 + O(n)$

(nach jedem Vergleich außer letztem in Z.4)
(+1)

Insertionsort

💡 **Idee:** Kartenspielermethode: Füge Elemente suzessive in sortierte „Hand“ ein.

Hand = Präfix des Arrays

Einfügen = nach links Vertauschen, bis größer

</> **Code:**

```
1 static <Elem> void insertionSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 1; i < n; ++i)  
4         for (int j = i; j > 0 && less(A[j], A[j-1], c); --j)  
5             swap(A, j, j-1);  
6 }
```

<https://visualgo.net/en/sorting?mode=Insertion>

© **Korrektheit:**

Invariante: $A[0..i]$ sortiert (in Z.3)

🏔 **Analyse:**

Anzahl Vergleiche $\leq \frac{1}{2}n^2 + O(n)$

Anzahl Swaps $\leq \frac{1}{2}n^2 + O(n)$

(nach jedem Vergleich außer letztem in Z.4)

Optimierung: Einfügeposition per binärer Suche finden! \rightsquigarrow *Binary Insertionsort*

Da wir aber trotzdem per Swaps Elemente bewegen müssen, hilft das nur für Vergleiche ...

Insertionsort

- 💡 **Idee:** Kartenspielermethode: Füge Elemente sukzessive in sortierte „Hand“ ein.
Hand = Präfix des Arrays
Einfügen = nach links Vertauschen, bis größer

</> Code:

```
1 static <Elem> void insertionSort(Elem[] A, Comparator<Elem> c) {  
2     int n = A.length;  
3     for (int i = 1; i < n; ++i)  
4         for (int j = i; j > 0 && less(A[j], A[j-1], c); --j)  
5             swap(A, j, j-1);  
6 }
```

<https://visualgo.net/en/sorting?mode=Insertion>

© Korrektheit:

Invariante: $A[0..i]$ sortiert (in Z.3)

🏔 Analyse:

Anzahl Vergleiche $\leq \frac{1}{2}n^2 + O(n)$

Anzahl Swaps $\leq \frac{1}{2}n^2 + O(n)$

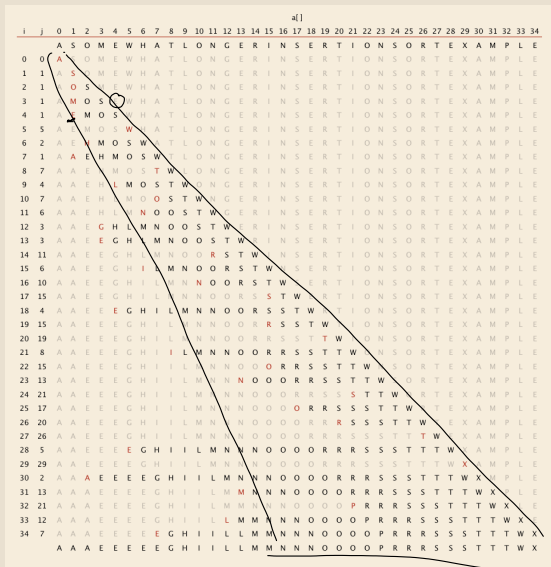
(nach jedem Vergleich außer letztem in Z.4)

- Optimierung:** Einfügeposition per binärer Suche finden! \rightsquigarrow *Binary Insertionsort*
Da wir aber trotzdem per Swaps Elemente bewegen müssen, hilft das nur für Vergleiche ...

Sowohl Bubblesort als auch Insertionsort sind datenabhängig. Wie gut sind sie?

<https://www.wild-inter.net/teaching/ads/insertionsort>

Insertion sort: trace



7.3 Analyse der primitiven Verfahren

Clicker Question

Von welchem der Algorithmen bisher versprechen Sie sich die beste Performance im Average-Case?



- A** Selectionsort
- B** Bubblesort
- C** Insertionsort
- D** keine Ahnung



→ sli.do/cs210

Datenabhängigkeit

Sowohl Bubblesort als auch Insertionsort sind datenabhängig. Wie gut sind sie?

Datenabhängigkeit

Sowohl Bubblesort als auch Insertionsort sind datenabhängig. Wie gut sind sie?

Guter Anfang: Visualisieren und experimentieren!

Hoffnung: Bekommen dadurch eine Idee für eine Hypothese

Permutationen und Inversionen

Für die Analyse benötigen wir etwas Notation

- ▶ Wir schreiben $[n] = [1..n] = \underline{\{1, 2, \dots, n\}}$

Permutationen und Inversionen

Für die Analyse benötigen wir etwas Notation

- ▶ Wir schreiben $[n] = [1..n] = \{1, 2, \dots, n\}$
- ▶ Eine *Permutation* ist eine bijektive Abbildung $\pi : [n] \rightarrow [n]$
wir identifizieren die Funktion π mit der Sequenz ihrer Bilder: $\pi(1), \dots, \pi(n)$

Beispiel: Die Permutation $1, 2, 3, 4, 5$ ist die Identitätsfunktion auf $[5]$

Permutationen und Inversionen

Für die Analyse benötigen wir etwas Notation

- ▶ Wir schreiben $[n] = [1..n] = \{1, 2, \dots, n\}$
- ▶ Eine *Permutation* ist eine bijektive Abbildung $\pi : [n] \rightarrow [n]$
wir identifizieren die Funktion π mit der Sequenz ihrer Bilder: $\pi(1), \dots, \pi(n)$

Beispiel: Die Permutation $1, 2, 3, 4, 5$ ist die Identitätsfunktion auf $[5]$

- ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ ist $(i, j) \in [n]^2$ mit einer Inversion, wenn $i < j$, aber $\pi(i) > \pi(j)$ gilt.

$2, 1, 4, 3$ hat genau 2 Inversionen: $(1, 2)$ und $(3, 4)$

x		1	2	3	4
$\pi(x)$		2	1	4	3

Permutation und Algebra

Permutationen sind ein gut untersuchtes Gebiet . . .

- ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^{-1} mit $\pi^{-1}(i) = j$ gdw. $\pi(j) = i$ die *inverse Permutation* zu π

Permutation und Algebra

Permutationen sind ein gut untersuchtes Gebiet . . .

- ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^{-1} mit $\pi^{-1}(i) = j$ gdw. $\pi(j) = i$ die *inverse Permutation* zu π
- ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^R mit $\pi^R = \pi(n), \pi(n-1), \dots, \pi(2), \pi(1)$ die reverse Permutation zu π

Permutation und Algebra

Permutationen sind ein gut untersuchtes Gebiet . . .

- ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^{-1} mit $\pi^{-1}(i) = j$ gdw. $\pi(j) = i$ die *inverse Permutation* zu π
 - ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^R mit $\pi^R = \pi(n), \pi(n-1), \dots, \pi(2), \pi(1)$ die *reverse Permutation* zu π
- ↪ **Algebra:** Die Menge aller Permutationen von $[n]$ mit der Operation $\pi \circ \pi'$ (Funktionskomposition) bildet eine Gruppe, die symmetrische Gruppe \mathcal{S}_n , mit neutralem Element $\text{id}_n = 1, 2, \dots, n$.

Permutation und Algebra

Permutationen sind ein gut untersuchtes Gebiet ...

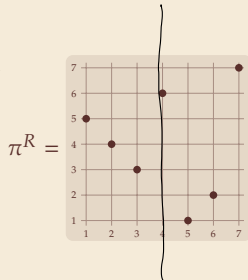
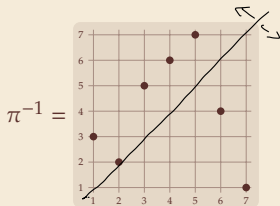
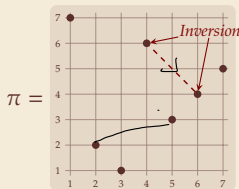
- ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^{-1} mit $\pi^{-1}(i) = j$ gdw. $\pi(j) = i$ die *inverse Permutation* zu π
 - ▶ Für eine Permutation $\pi : [n] \rightarrow [n]$ heißt die eindeutig bestimmte Permutation π^R mit $\pi^R = \pi(n), \pi(n-1), \dots, \pi(2), \pi(1)$ die *reverse Permutation* zu π
- ↪ **Algebra:** Die Menge aller Permutationen von $[n]$ mit der Operation $\pi \circ \pi'$ (Funktionskomposition) bildet eine *Gruppe*, die *symmetrische Gruppe* \mathcal{S}_n , mit neutralem Element $\text{id}_n = 1, 2, \dots, n$.

- ▶ Oft am einfachsten graphisch! (i auf x-Achse, $\pi(i)$ auf y-Achse)

$$\pi = [7, 2, 1, 6, 3, 4, 5]$$

$$\pi^{-1} = [3, 2, 5, 6, 7, 4, 1]$$

$$\pi^R = [5, 4, 3, 6, 1, 2, 7]$$



Zurück zum Sortieren

ohne Beschränkung der Allgemeinheit

für die Analyse

Wir sortieren Vergleichs-basiert \rightsquigarrow o. B. d. A. sortieren Permutation von $[n]$

► In Bubblesort und Insertionsort:

keine Duplikate!

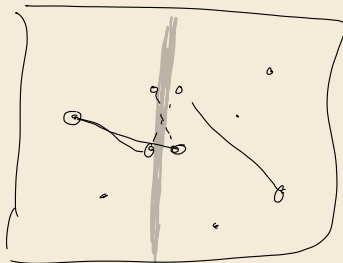
► jeder Swap vertauscht benachbarte Indices $j, j + 1$

← Definition von *primitiven* Sortierverfahren

► Vergleich davor stellt sicher: $A[j] > A[j + 1]$

\rightsquigarrow Beheben eine Inversion der Eingabe

$(j, j + 1)$



Zurück zum Sortieren

ohne Beschränkung der Allgemeinheit

Wir sortieren **Vergleichs-basiert** \rightsquigarrow o. B. d. A. sortieren Permutation von $[n]$

▶ In Bubblesort und Insertionsort:

keine Duplikate!

▶ jeder Swap vertauscht benachbarte Indices $j, j + 1$

← Definition von *primitiven* Sortierverfahren

▶ Vergleich davor stellt sicher: $A[j] > A[j + 1]$

\rightsquigarrow Beheben **eine Inversion** der Eingabe

▶ Aber auch **genau** eine Inversion! (benachbarte Indices!)

\rightsquigarrow *Bubblesort und Insertionsort verwenden genau so viele Swaps wie die Eingabe Inversionen hat.*

Zurück zum Sortieren

Wir sortieren **Vergleichs-basiert** \rightsquigarrow o. B. d. A. sortieren Permutation von $[n]$

ohne Beschränkung der Allgemeinheit

▶ In Bubblesort und Insertionsort:

keine Duplikate!

▶ jeder Swap vertauscht benachbarte Indices $j, j + 1$

← Definition von *primitiven* Sortierverfahren

▶ Vergleich davor stellt sicher: $A[j] > A[j + 1]$

\rightsquigarrow Beheben **eine Inversion** der Eingabe

▶ Aber auch **genau** eine Inversion! (benachbarte Indices!)

\rightsquigarrow *Bubblesort und Insertionsort verwenden genau so viele Swaps wie die Eingabe Inversionen hat.*

\rightsquigarrow Laufzeit $\geq \text{Inv}(A[0..n]) = \#\text{Inversionen in } A = |\{(i, j) : 0 \leq i < j < n : A[i] > A[j]\}|$

Zurück zum Sortieren

Wir sortieren **Vergleichs-basiert** \rightsquigarrow o. B. d. A. sortieren Permutation von $[n]$

ohne Beschränkung der Allgemeinheit

▶ In Bubblesort und Insertionsort:

keine Duplikate!

▶ jeder Swap vertauscht benachbarte Indices $j, j + 1$

← Definition von *primitiven* Sortierverfahren

▶ Vergleich davor stellt sicher: $A[j] > A[j + 1]$

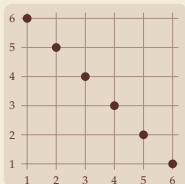
\rightsquigarrow Beheben **eine Inversion** der Eingabe

▶ Aber auch **genau** eine Inversion! (benachbarte Indices!)

\rightsquigarrow *Bubblesort und Insertionsort verwenden genau so viele Swaps wie die Eingabe Inversionen hat.*

\rightsquigarrow Laufzeit $\geq \text{Inv}(A[0..n]) = \#\text{Inversionen in } A = |\{(i, j) : 0 \leq i < j < n : A[i] > A[j]\}|$

Worst-Case:



Jedes Paar $i < j$ ist Inversion $\rightsquigarrow \text{Inv} = \binom{n}{2} \sim \frac{1}{2}n^2$ 😞

Average-Case-Analyse von Inversionen

Im Worst-Case ist die Anzahl Inversionen quadratisch

↪ Alle primitiven Sortierverfahren benötigen Laufzeit in $\Omega(n^2)$ im Worst-Case

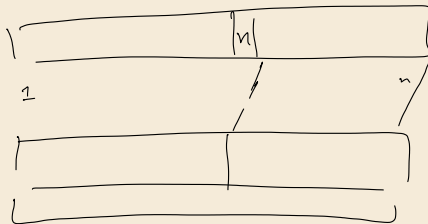
Wie sieht es im Average-Case aus?

$$\prod_{i=1}^n i$$

▶ **Random permutation model:** Alle $n!$ Permutation gleich wahrscheinlich

↪ $\mathbb{E}[\text{Inv}(\pi)] = ?$ (scheint knifflig?)

n Möglichkeiten für



Average-Case-Analyse von Inversionen

Im Worst-Case ist die Anzahl Inversionen quadratisch

↪ Alle primitiven Sortierverfahren benötigen Laufzeit in $\Omega(n^2)$ im Worst-Case

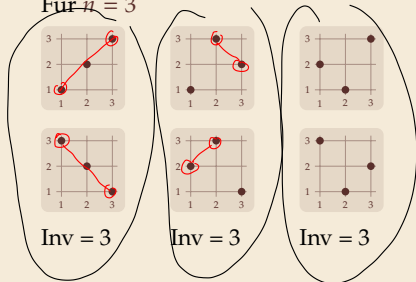
Wie sieht es im Average-Case aus?

► **Random permutation model:** Alle $n!$ Permutation gleich wahrscheinlich

↪ $\mathbb{E}[\text{Inv}(\pi)] = ?$ (scheint knifflig?)

► **Trick:** Betrachte Paar von Permutation π und π^R gemeinsam.

Für $n = 3$



Average-Case-Analyse von Inversionen

Im Worst-Case ist die Anzahl Inversionen quadratisch

↪ Alle primitiven Sortierverfahren benötigen Laufzeit in $\Omega(n^2)$ im Worst-Case

Wie sieht es im Average-Case aus?

► **Random permutation model:** Alle $n!$ Permutation gleich wahrscheinlich

↪ $\mathbb{E}[\text{Inv}(\pi)] = ?$ (scheint knifflig?)

► **Trick:** Betrachte Paar von Permutation π und π^R gemeinsam.

Für $n = 3$



Inv = 3

Inv = 3

Inv = 3

↪ $\text{Inv}(\pi) + \text{Inv}(\pi^R) = \binom{n}{2}$ gilt allgemein!

Im Mittel haben also beide $\binom{n}{2}/2$ Inversionen

$$\mathbb{E}[\text{Inv}(A[0..n])] = \binom{n}{2}/2 \sim \frac{1}{4}n^2$$

Clicker Question

Von welchem der Algorithmen bisher versprechen Sie sich die beste Performance im Average-Case?



- A** Selectionsort
- B** Bubblesort
- C** Insertionsort
- D** keine Ahnung



→ sli.do/cs210

Clicker Question



Von welchem der Algorithmen bisher versprechen Sie sich die beste Performance im Average-Case?

- A** ~~Selectionsort~~
- B** Bubblesort ✓
- C** Insertionsort ✓
- D** ~~keine Ahnung~~



→ sli.do/cs210

Primitive Sortierverfahren – Diskussion

- 👍 Einfache Algorithmen, kompakter Code
 - 👍 Kein zusätzlicher Speicher/Puffer nötig
 - 👎 $\Omega(n^2)$ im Worst-Case
 - 👎 Average-Case um Faktor 2 besser also Worst-Case ... aber trotzdem quadratisch
- ↪ Gute Wahl für sehr kleine Eingaben (minimaler Overhead)
- ⚡ nicht skalierbar

7.4 Mergesort

Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [this lecture]



...

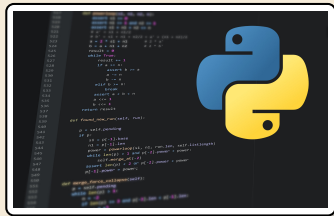
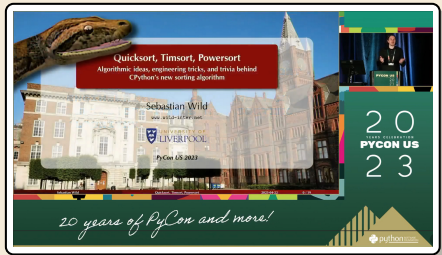
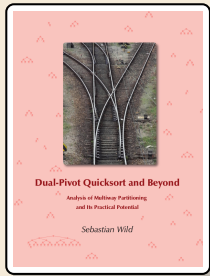
Quicksort. [next lecture]



...

— Werbung —

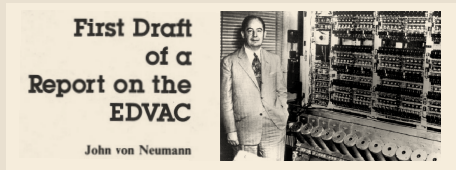
Sortieralgorithmen in Standardbibliotheken haben mich viel beschäftigt ...



Mergesort

Basic plan.

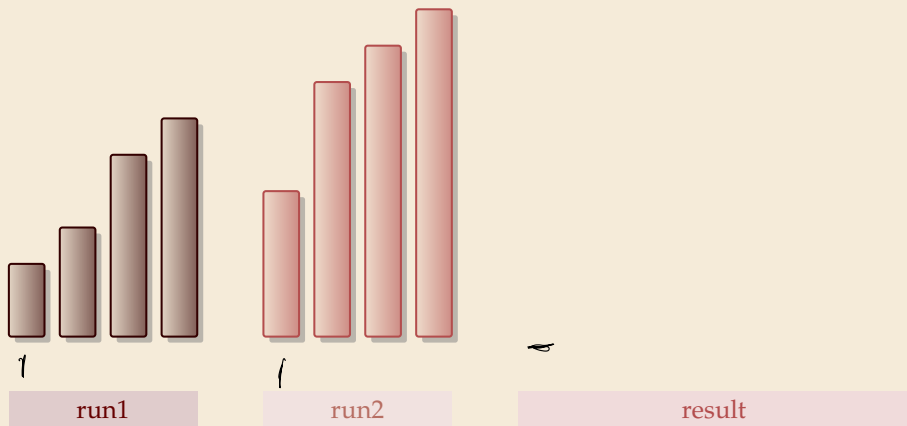
- Divide array into two halves.
- **Recursively** sort each half.
- Merge two halves.



Merging runs – Verschmelzen sortierter Teile



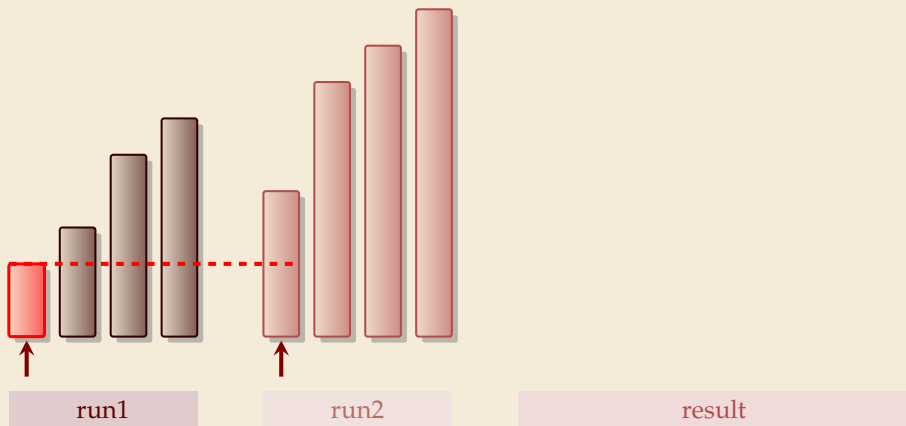
Merging runs – Verschmelzen sortierter Teile



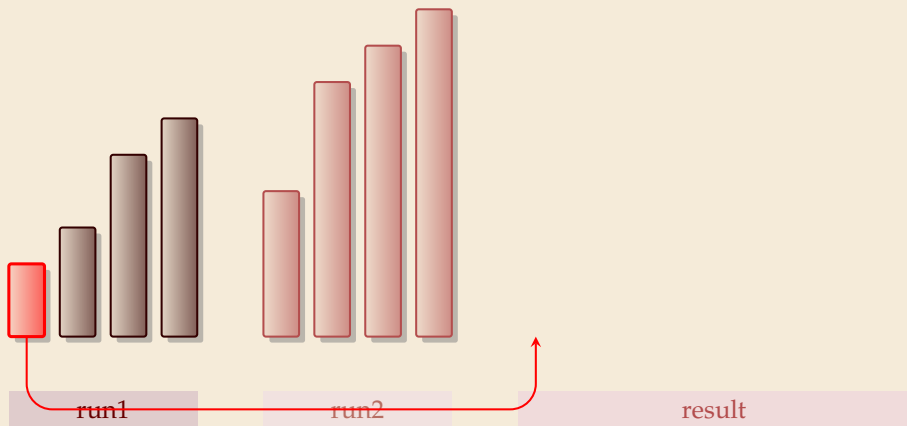
Merging runs – Verschmelzen sortierter Teile



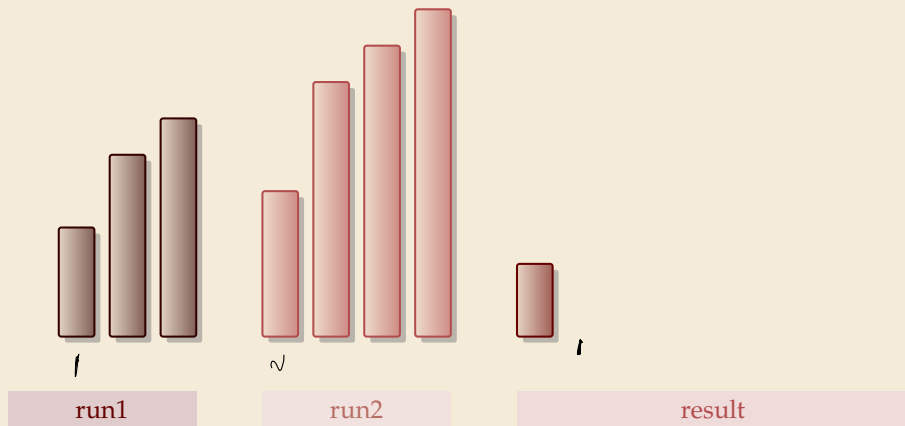
Merging runs – Verschmelzen sortierter Teile



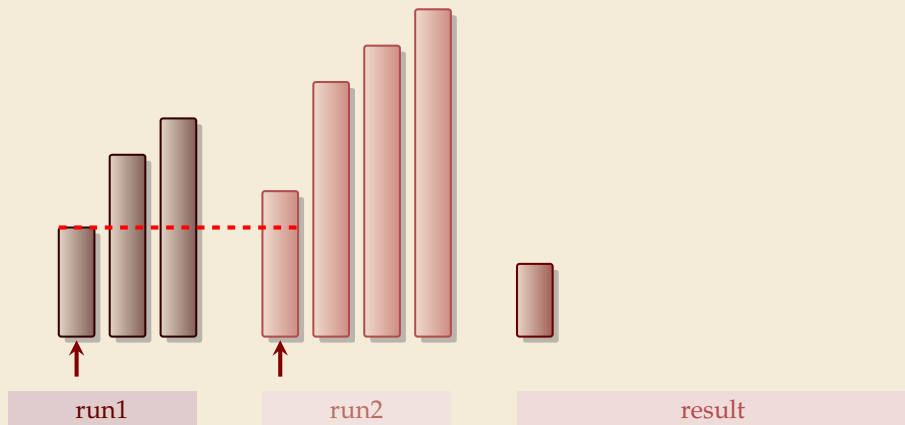
Merging runs – Verschmelzen sortierter Teile



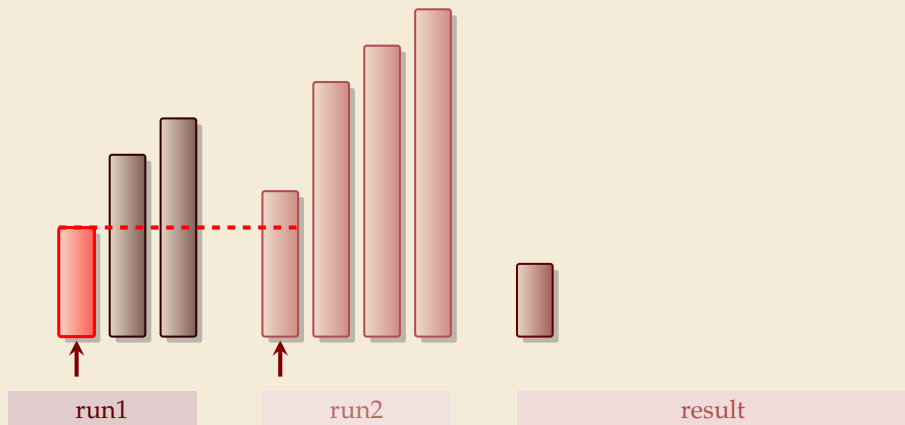
Merging runs – Verschmelzen sortierter Teile



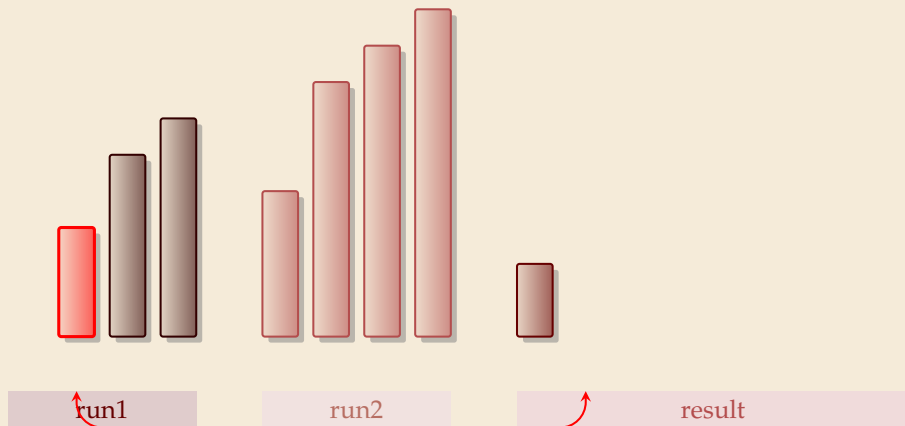
Merging runs – Verschmelzen sortierter Teile



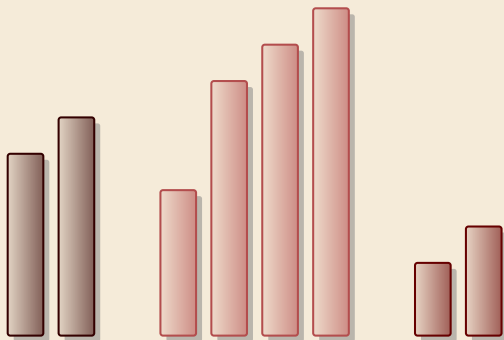
Merging runs – Verschmelzen sortierter Teile



Merging runs – Verschmelzen sortierter Teile



Merging runs – Verschmelzen sortierter Teile

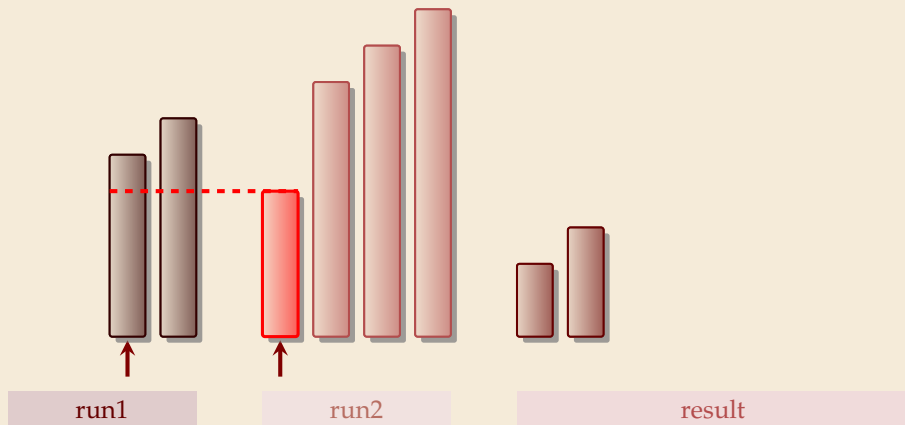


run1

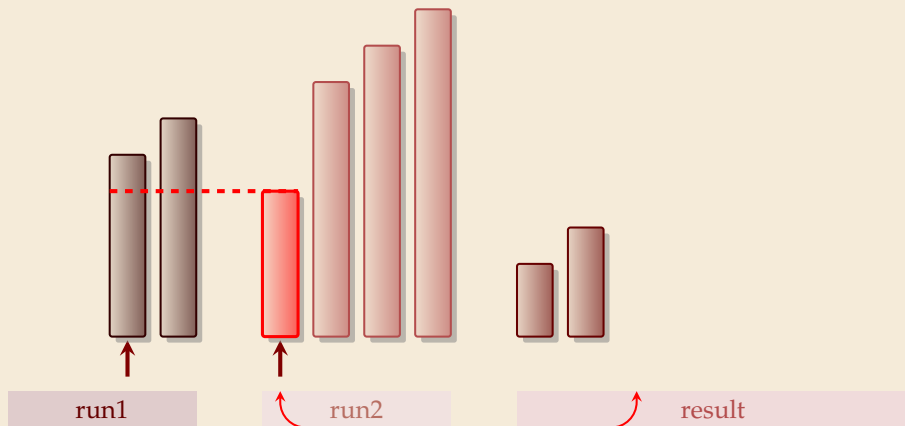
run2

result

Merging runs – Verschmelzen sortierter Teile



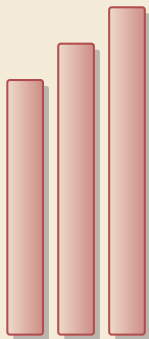
Merging runs – Verschmelzen sortierter Teile



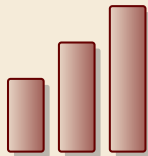
Merging runs – Verschmelzen sortierter Teile



run1

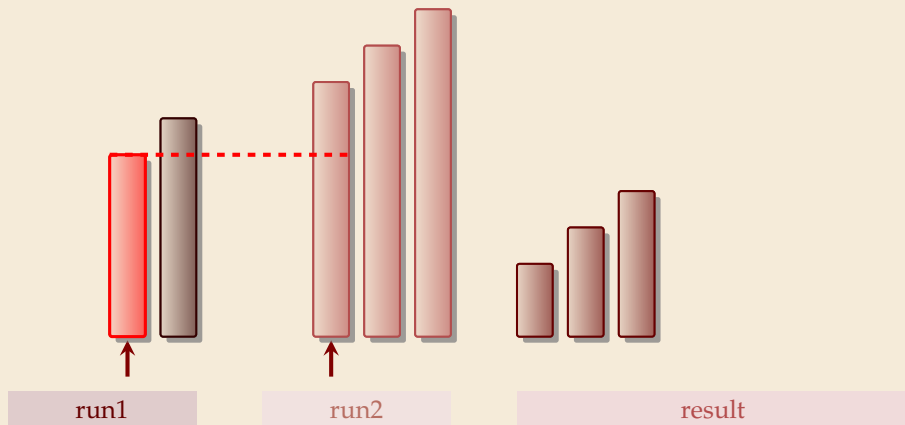


run2

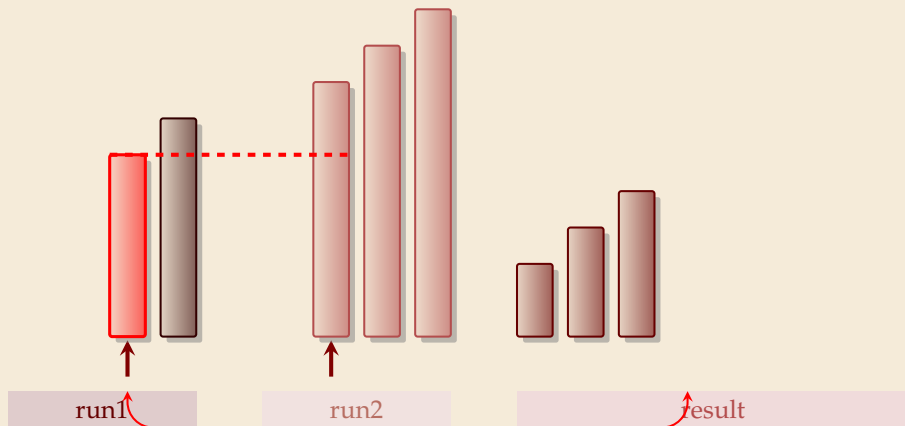


result

Merging runs – Verschmelzen sortierter Teile



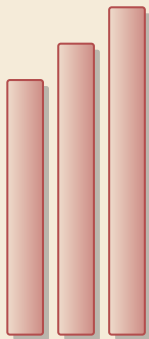
Merging runs – Verschmelzen sortierter Teile



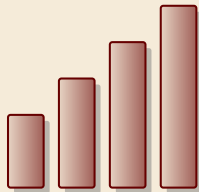
Merging runs – Verschmelzen sortierter Teile



run1

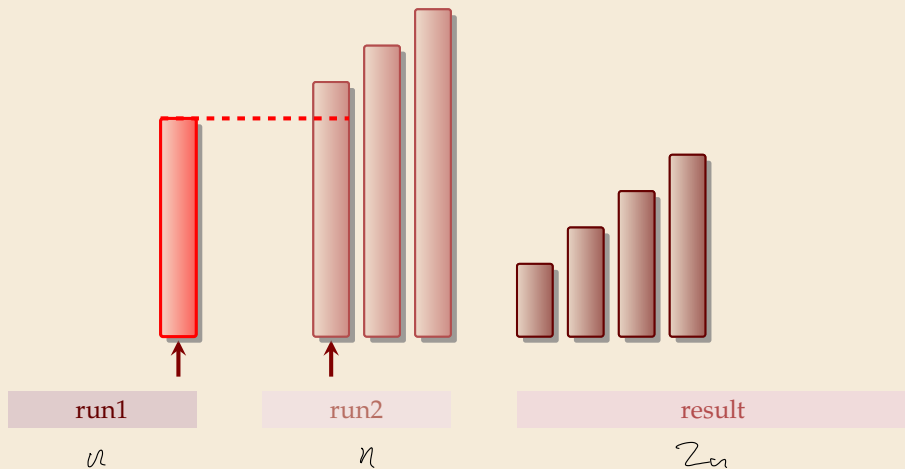


run2

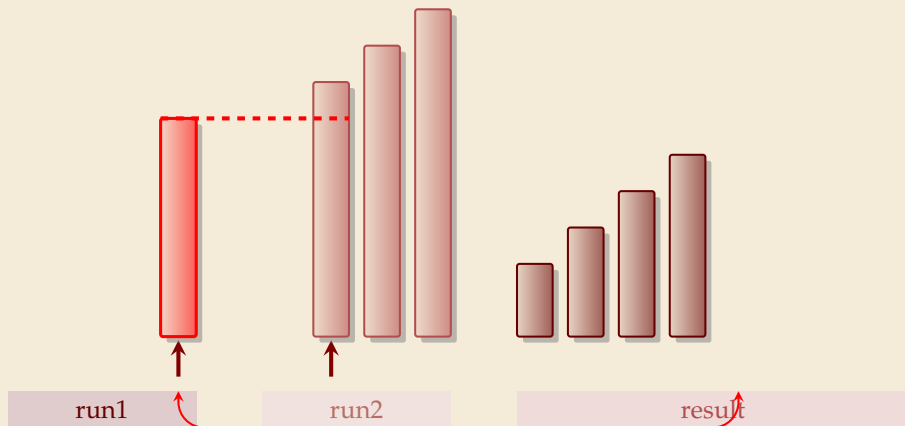


result

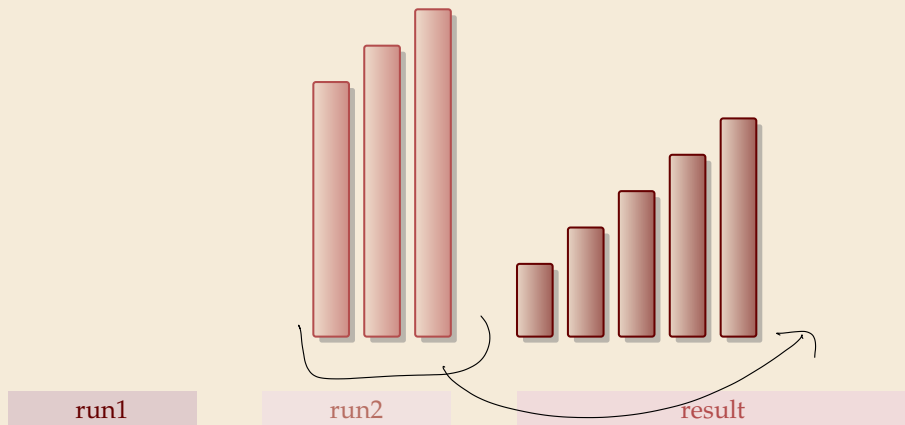
Merging runs – Verschmelzen sortierter Teile



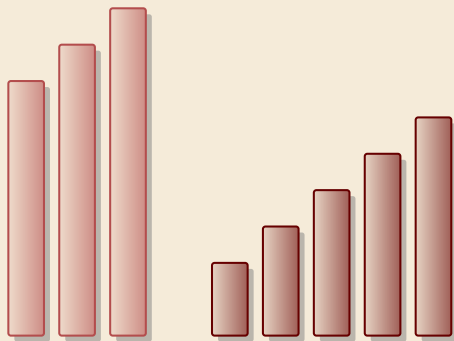
Merging runs – Verschmelzen sortierter Teile



Merging runs – Verschmelzen sortierter Teile



Merging runs – Verschmelzen sortierter Teile

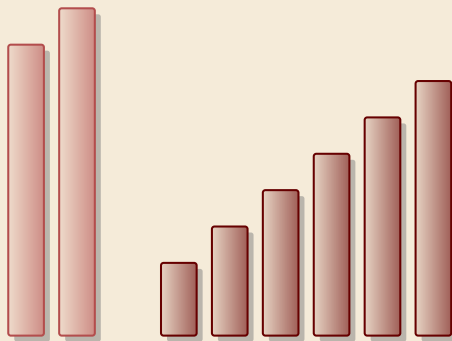


run1

run2

result

Merging runs – Verschmelzen sortierter Teile

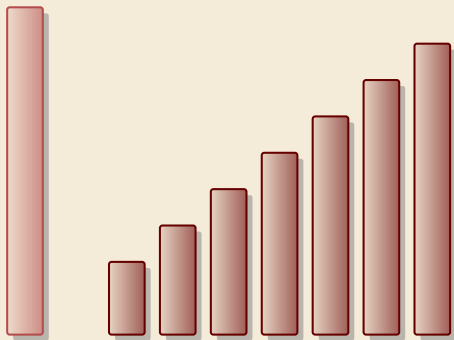


run1

run2

result

Merging runs – Verschmelzen sortierter Teile

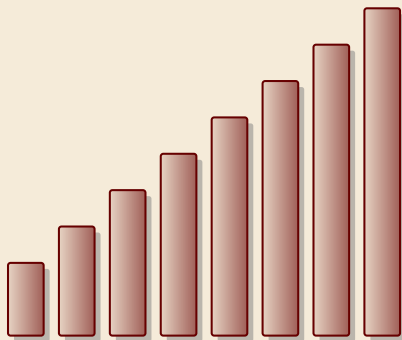


run1

run2

result

Merging runs – Verschmelzen sortierter Teile



run1

run2

result

Clicker Question



Welche Laufzeit sollte das Mergen von zwei Runs der Länge n haben?

A $O(1)$

B $O(\log n)$

C $O(\log^2 n)$

D $O(\sqrt{n})$

E $O(\sqrt[3]{n})$

F $O(n)$

G $O(n \log n)$

H $O(n^{1.5})$

I $O(n^2)$

J keine Ahnung



→ sli.do/cs210

Clicker Question

Welche Laufzeit sollte das Mergen von zwei Runs der Länge n haben?



A ~~$O(1)$~~

B ~~$O(\log n)$~~

C ~~$O(\log^2 n)$~~

D ~~$O(\sqrt{n})$~~

E ~~$O(\sqrt[3]{n})$~~

F $O(n)$ ✓

G ~~$O(n \log n)$~~

H ~~$O(n^{1.5})$~~

I ~~$O(n^2)$~~

J ~~keine Ahnung~~



→ sli.do/cs210

Merge – In-place API

Merge (aus Sicht des Benutzers):

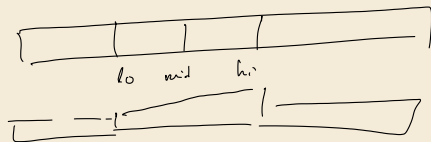
- ▶ **Eingabe:** Zwei sortierte Teilarrays
- ▶ **Ausgabe:** Sortiertes Array
- ▶ Bequem z.B. mit physischen beweglichen Objekten zu machen

Wie machen wir das aber mit einem Array $A[0..n)$ von Objekten?

Merge – In-place API

Merge (aus Sicht des Benutzers):

- ▶ **Eingabe:** Zwei sortierte Teilarrays
- ▶ **Ausgabe:** Sortiertes Array
- ▶ Bequem z.B. mit physischen beweglichen Objekten zu machen



Wie machen wir das aber mit einem Array $A[0..n)$ von Objekten?

Merge (in Arrays, als Teil unseres Mergesort):

- ▶ **Gegeben:** zwei *benachbarte* sortierte Bereiche $A[lo..mid]$ und $A[mid..hi]$ in Array $A[0..n)$
- ▶ **Ziel:** $A[lo..hi]$ sortiert

Merge – In-place API

Merge (aus Sicht des Benutzers):

- ▶ **Eingabe:** Zwei sortierte Teilarrays
- ▶ **Ausgabe:** Sortiertes Array
- ▶ Bequem z.B. mit physischen beweglichen Objekten zu machen

Wie machen wir das aber mit einem Array $A[0..n)$ von Objekten?

Merge (in Arrays, als Teil unseres Mergesort):

- ▶ **Gegeben:** zwei *benachbarte* sortierte Bereiche $A[lo..mid]$ und $A[mid..hi]$ in Array $A[0..n)$
- ▶ **Ziel:** $A[lo..hi]$ sortiert
- ⚡ extrem knifflig ohne einen Pufferbereich, in den wir Objekte auslagern können!

↪ intern: (1) Verschiebe Elemente in Hilfsarray $aux[lo..hi]$ |
(2) Merge zurück von aux nach $A[lo..hi]$ }

Merging: Java implementation

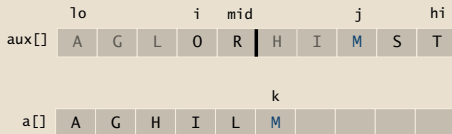
$A[0..n)$ $aux[0..n)$

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)           a[k] = aux[j++];
        else if (j > hi)       a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                   a[k] = aux[i++];
    }
}
```

copy

merge



Slide - Frage

$n = mid - lo + 1$

$m = hi - mid$

$n \leq m$

Clicker Question

Wie viele less-Aufrufe macht merge, wenn die beiden Runs Länge n bzw. m haben?



- A** genau n
- B** genau m
- C** genau $n + m$
- D** genau nm
- E** höchstens $n + m$
- F** höchstens nm
- G** mindestens $n + m$
- H** mindestens nm
- I** mindestens $\min\{n, m\}$



→ sli.do/cs210

Clicker Question

Wie viele less-Aufrufe macht merge, wenn die beiden Runs Länge n bzw. m haben?



- A** ~~genau n~~
- B** ~~genau m~~
- C** ~~genau $n + m$~~
- D** ~~genau $n \cdot m$~~
- E** höchstens $n + m$ ✓
- F** ~~höchstens $n \cdot m$~~
- G** ~~mindestens $n + m$~~
- H** ~~mindestens $n \cdot m$~~
- I** mindestens $\min\{n, m\}$ ✓



→ sli.do/cs210

Mergesort: Java implementation

```
public class Merge
{
    private static void merge(...)
    { /* as before */ }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
    {
        if (hi <= lo) return; // 1 Element
        int mid = lo + (hi - lo) / 2; // (hi+lo) / 2
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a)
    {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```



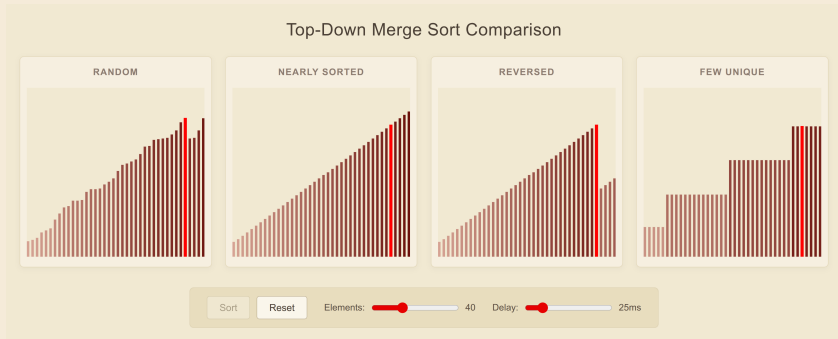
Mergesort: trace

	lo	hi	a[]														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
merge(a, aux, 0, 0, 1)	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E	
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E	
merge(a, aux, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E	
merge(a, aux, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E	
merge(a, aux, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E	
merge(a, aux, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E	
merge(a, aux, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L	
merge(a, aux, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P	
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X	

result after recursive call

Datenabhängigkeit?

Tatsächlich hängt die Laufzeit von Mergesort (fast) nicht von den Daten ab!



<https://www.wild-inter.net/teaching/ads/mergesort>

Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N .

Pf sketch. The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

↑ left half ↑ right half ↑ merge (worst case)

We solve the recurrence when N is a power of 2: ← result holds for all N
(analysis cleaner in this case)

$$D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

Wir erinnern uns: Master-Theorem

Laufzeit von Mergesort

$$T(n) = \begin{cases} 0 & n \leq 1 \\ 2 \cdot T(n/2) + n & n \geq 2 \end{cases}$$

Für $n = 2^k, k \in \mathbb{N}_0$

$$c = \log_2(2) = 1$$

$$\begin{array}{ccc} n^c & \text{vs.} & n \\ = & & \\ n & & \end{array}$$

$$f(n) = \Theta(n^c)$$

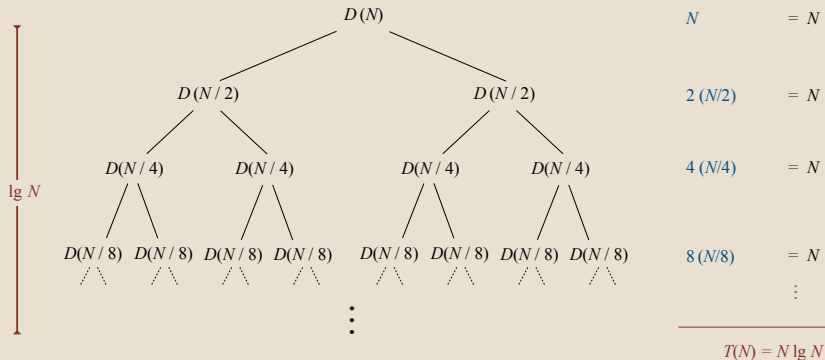
$$\Rightarrow C(n) = \Theta(n \log n)$$

MT-(b)

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by induction

Proposition. If $D(N)$ satisfies $D(N) = 2 D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 2. [assuming N is a power of 2]

- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N \lg N$.
- Goal: show that $D(2N) = (2N) \lg (2N)$.

$$D(2N) = 2 D(N) + 2N$$

given

$$= 2 N \lg N + 2N$$

inductive hypothesis

$$= 2 N (\lg (2N) - 1) + 2N$$

algebra

$$= 2 N \lg (2N)$$

QED

Mergesort: number of array accesses

Proposition. Mergesort uses $\leq 6N \lg N$ array accesses to sort an array of length N .

Pf sketch. The number of array accesses $A(N)$ satisfies the recurrence:

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \text{ for } N > 1, \text{ with } A(1) = 0.$$

↑

Key point. Any algorithm with the following structure takes $N \log N$ time:

```
public static void linearithmic(int N)
{
    if (N == 0) return;
    linearithmic(N/2); ← solve two problems
    linearithmic(N/2); ← of half the size
    linear(N); ← do a linear amount of work
}
```

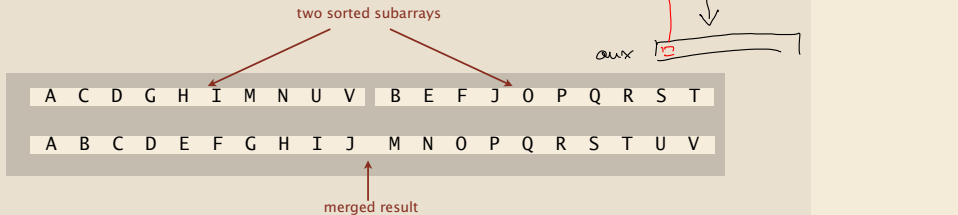
Notable examples. FFT, hidden-line removal, Kendall-tau distance, ...

7.5 Mergesort Optimierungen

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N .

Pf. The array `aux[]` needs to be of length N for the last merge.



Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

$O(N)$ typischerweise die Größe

Challenge 1 (not hard). Use `aux[]` array of length $\sim \frac{1}{2} N$ instead of N .

Challenge 2 (very hard). In-place merge. [Kronrod 1969] \notin exam

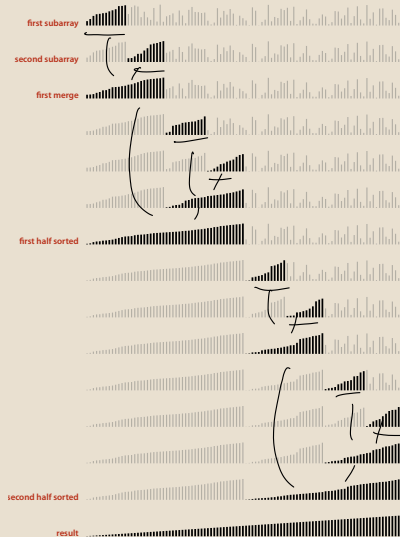
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

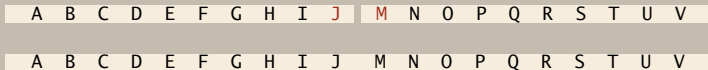
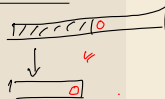
Mergesort with cutoff to insertion sort: visualization



Mergesort: practical improvements

Stop if already sorted.

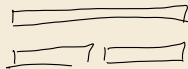
- Is largest item in first half \leq smallest item in second half?
- Helps for partially-ordered arrays.



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```

Mergesort: practical improvements

Pingpong Mergesort



Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)      aux[k] = a[j++];
        else if (j > hi)  aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else              aux[k] = a[i++];
    }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort (aux, a, lo, mid);
    sort (aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

← merge from a[] to aux[]

↑ assumes aux[] is initialize to a[] once, before recursive calls

↑ switch roles of aux[] and a[]

Iterativer Mergesort

Bisher: Mergesort als rekursive Methode

- ▶ meistens effizienteste Methode
- ▶ manchmal unbequem, dass man Platz für call stack braucht

Iterativer Mergesort

Bisher: Mergesort als rekursive Methode

- ▶ meistens effizienteste Methode
- ▶ manchmal unbequem, dass man Platz für call stack braucht
- ▶ manchmal hilfreich, Sortieren zu können, ohne n zu kennen!

Iterativer Mergesort

Bisher: Mergesort als rekursive Methode

- ▶ meistens effizienteste Methode
- ▶ manchmal unbequem, dass man Platz für call stack braucht
- ▶ manchmal hilfreich, Sortieren zu können, ohne n zu kennen!

Mergesort kann auch ohne Rekursion auskommen!

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8,

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz = 1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 0, 1, 3)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 8, 11, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X
sz = 8	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom-up mergesort: Java implementation

```
public class MergeBU
{
    private static void merge(...)
    { /* as before */ }

    public static void sort(Comparable[] a)
    {
        int N = a.length;
        Comparable[] aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo += sz+sz)
                merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
```

but about 10% slower than recursive,
top-down mergesort on typical systems

Bottom line. Simple and non-recursive version of mergesort.

Stability

A typical application. First, sort by name; then sort by section.

```
Selection.sort(a, new Student.ByName());
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Selection.sort(a, new Student.BySection());
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

@#%&@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.

Stability

Q. Which sorts are stable?

A. Need to check algorithm (and implementation).

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

no longer sorted by time

still sorted by time

Stability: mergesort

Proposition. Mergesort is **stable**.

```
public class Merge
{
    private static void merge(...)
    { /* as before */ }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a)
    { /* as before */ }
}
```

Pf. Suffices to verify that merge operation is stable.

Stability: mergesort

Proposition. Merge operation is **stable**.

```
private static void merge(...)
{
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)           a[k] = aux[j++];
        else if (j > hi)      a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                  a[k] = aux[i++];
    }
}
```

0	1	2	3	4	5	6	7	8	9	10
A ₁	A ₂	A ₃	B	D	A ₄	A ₅	C	E	F	G

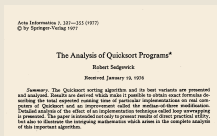
Pf. Takes from left subarray if equal keys.

7.6 Quicksort

A brief history of quicksort

Tony Hoare.

- Developed quicksort in 1960 to translate Russian into English.
- Later implemented it in Algol 60, using recursion.



Tony Hoare
(Turing Award, 1980)

Bob Sedgwick.

- Refined and popularized quicksort in the 1970s.
- Analyzed many versions of quicksort.



Bob Sedgwick
(longtime COS 226 instructor)

Quicksort – 60th Anniversary Workshop in 2021

The workshop will include a special event marking the 60th anniversary of Tony Hoare's invention of Quicksort.

Given the current restrictions on movement, due to the ongoing pandemic, the INI intends to run this workshop virtually across 5 days. All talks and discussion sessions will be available virtually.



Quicksort – 60th Anniversary Workshop in 2021

The workshop will include a special event marking the 60th anniversary of Tony Hoare's invention of Quicksort.

Given the current restrictions on movement, due to the ongoing pandemic, the INI intends to run this workshop virtually across 5 days. All talks and discussion sessions will be available virtually.

MONDAY 7TH JUNE 2021

16:00 to 16:30	Douglas McIlroy Dartmouth College Chair: Cliff Jones Folklore of Computational Complexity
16:30 to 17:00	Robert Sedgewick Princeton University Chair: Cliff Jones What Do We Know About Quicksort?
17:00 to 17:30	Break
17:30 to 18:00	Sebastian Wild University of Liverpool Chair: Cliff Jones Quicksorts of the 21st century
18:00 to 18:30	Tony Hoare Isaac Newton Institute Chair: Cliff Jones Tony Hoare Answers Questions



Quicksort – 60th Anniversary Workshop in 2021

The workshop will include a special event marking the 60th anniversary of Tony Hoare's invention of Quicksort.

Given the current restrictions on movement, due to the ongoing pandemic, the INI intends to run this workshop virtually across 5 days. All talks and discussion sessions will be available virtually.

MONDAY 7TH JUNE 2021

16:00 to 16:30 **Douglas McIlroy Dartmouth College**
Chair: Cliff Jones
Folklore of Computational Complexity

16:30 to 17:00 **Robert Sedgewick Princeton University**
Chair: Cliff Jones
What Do We Know About Quicksort?

17:00 to 17:30 Break

17:30 to 18:00 **Sebastian Wild University of Liverpool**
Chair: Cliff Jones
Quicksorts of the 21st century

18:00 to 18:30 **Tony Hoare Isaac Newton Institute**
Chair: Cliff Jones
Tony Hoare Answers Questions

What do we know about Quicksort? a brief summary of facts learned since the 1960s

Robert Sedgewick
Princeton University



Quicksort, Folklore, and Trust

Doug McIlroy
VSOW04

Quicksorts of the 21st century

Sebastian Wild

based on joint work with Yuhua Fan, Gonzalo Martorell, Markus Nebel, Ralph Henninger, and Avrim Sridhar



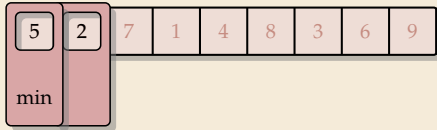
Isaac Newton Institute – Verified Software Workshop

Talk Recordings: <https://tiny.cc/quicksort-60>

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

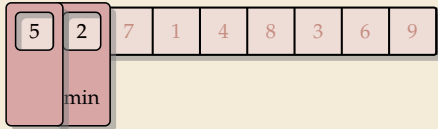


?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

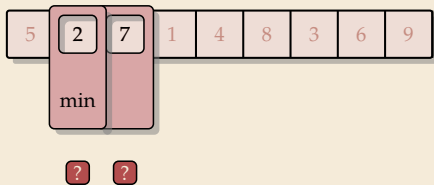


?

Irgendwas von Selectionsort zu lernen?

Selectionsort

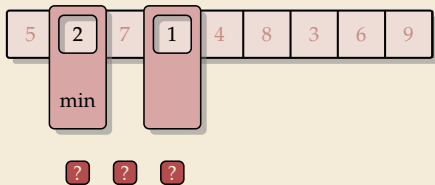
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

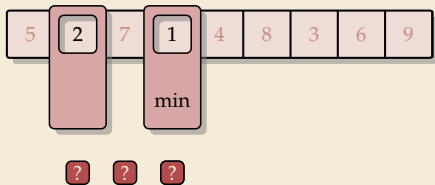
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

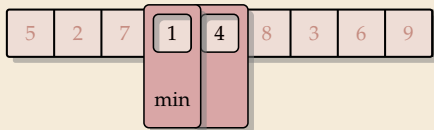
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

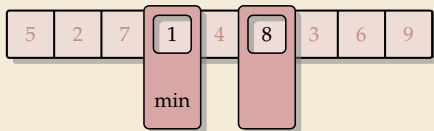
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

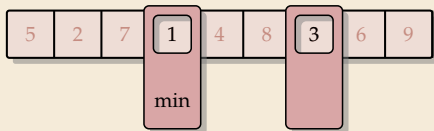


? ? ? ? ?

Irgendwas von Selectionsort zu lernen?

Selectionsort

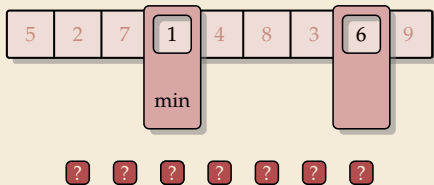
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

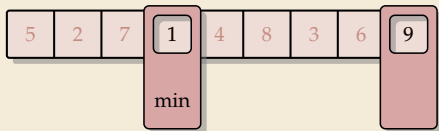
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

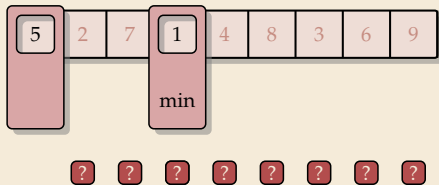
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

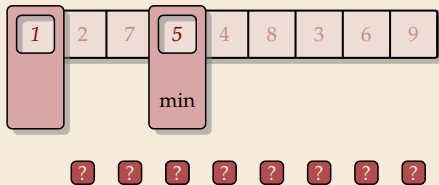
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

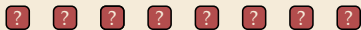
1	2	7	5	4	8	3	6	9
---	---	---	---	---	---	---	---	---



Irgendwas von Selectionsort zu lernen?

Selectionsort

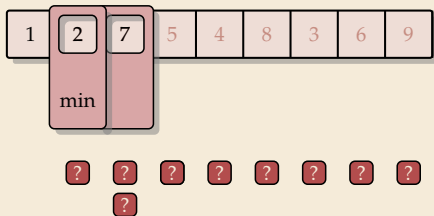
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

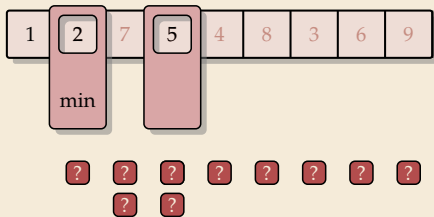
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

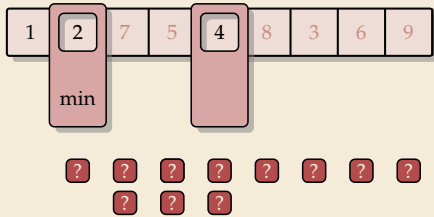
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

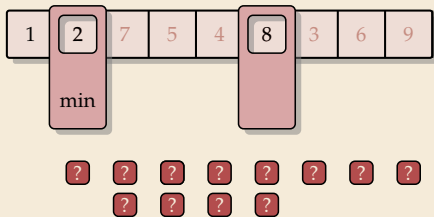
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

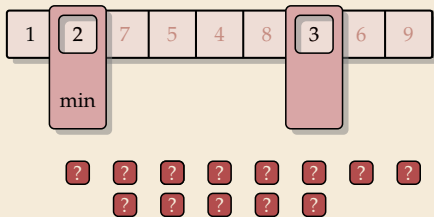
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

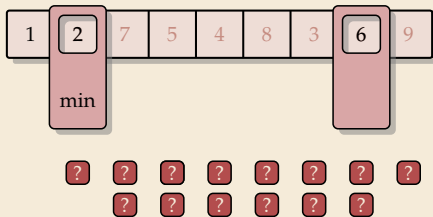
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

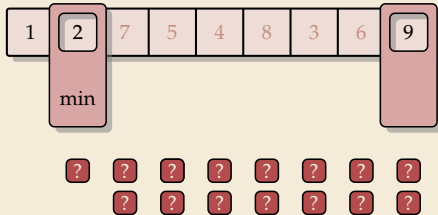
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

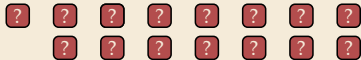
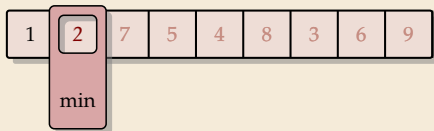
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

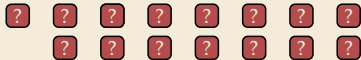


Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	7	5	4	8	3	6	9
---	---	---	---	---	---	---	---	---



Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	5	4	8	7	6	9
---	---	---	---	---	---	---	---	---



Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	8	7	6	9
---	---	---	---	---	---	---	---	---

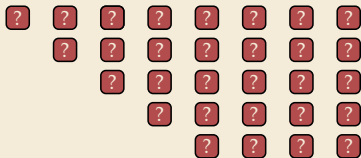


Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	8	7	6	9
---	---	---	---	---	---	---	---	---

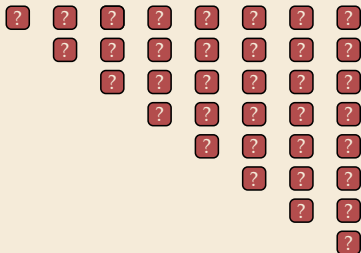


Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

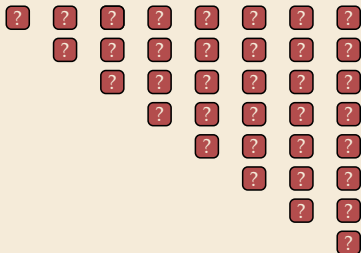


Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



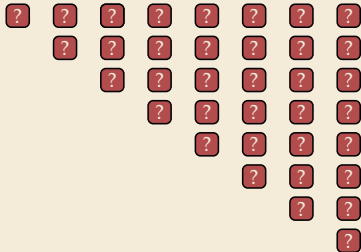
n - 1 Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

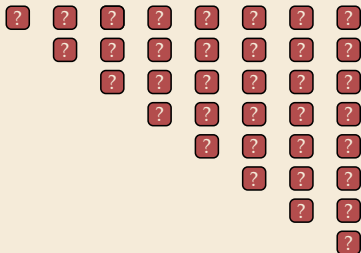
5	2	7	1	4	8	3	6	9
---	---	---	---	---	---	---	---	---

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

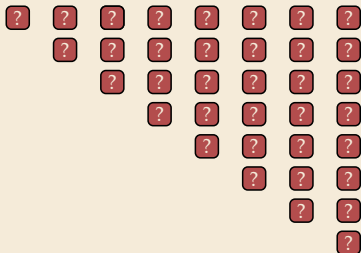
5	2	7	1	4	8	3	6	9
---	---	---	---	---	---	---	---	---

$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

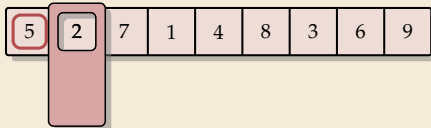
Finde Minimum und tausche nach links.



$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Rang finden

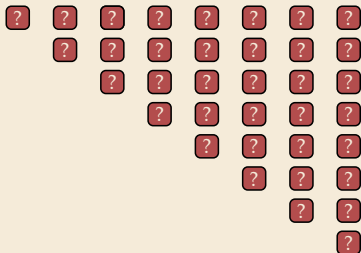
Zähle, wie viele Elements kleiner als *Pivot*.



Irgendwas von Selectionsort zu lernen?

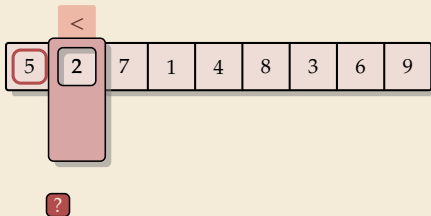
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

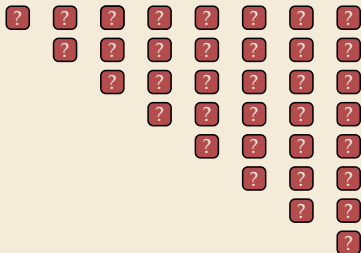


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

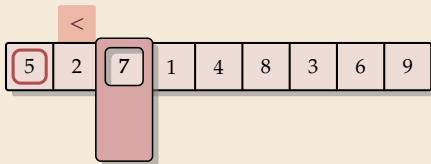
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

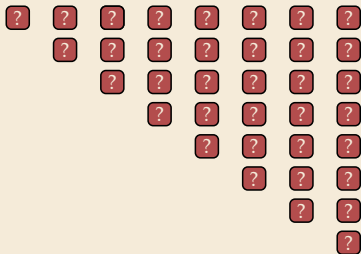


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

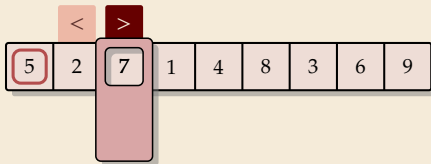
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

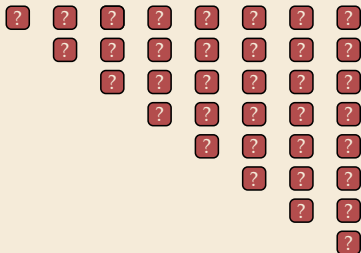


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

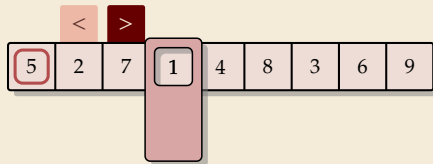
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

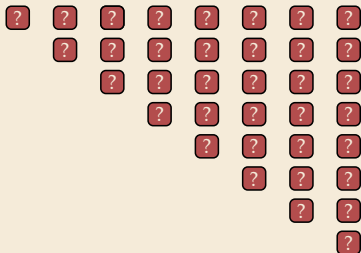


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

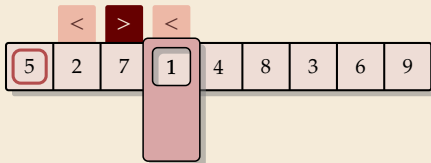
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

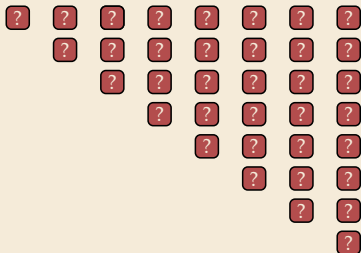


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

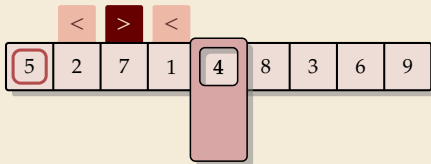
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

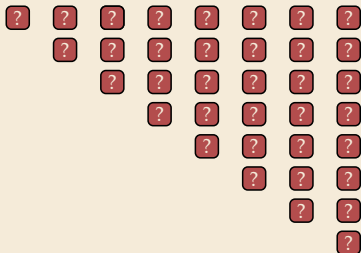


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

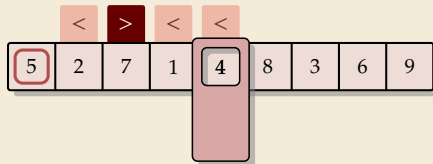
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

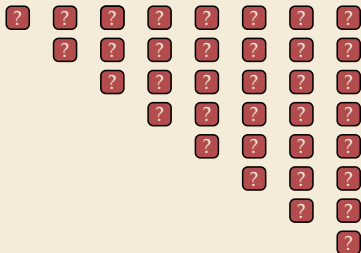


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

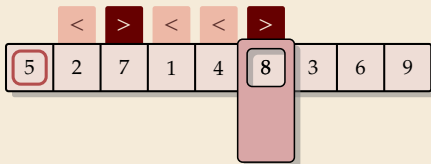
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

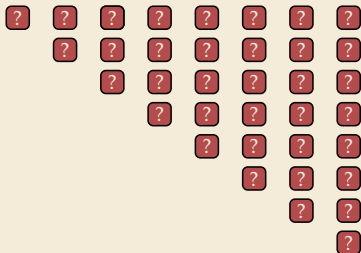


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

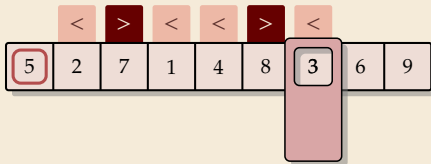
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

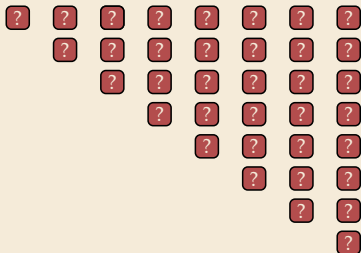


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

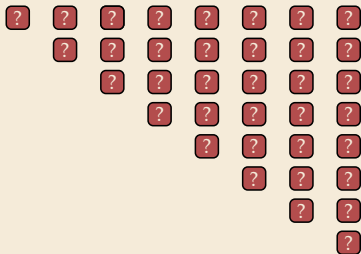


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

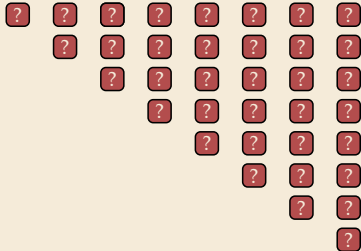


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

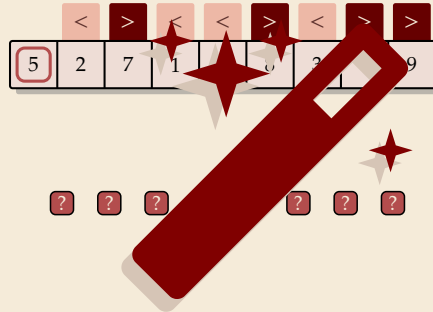
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

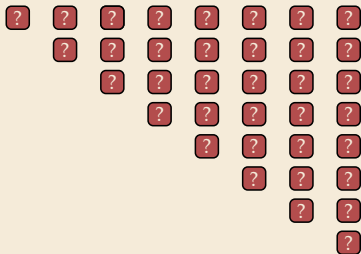


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

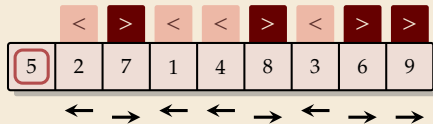
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.



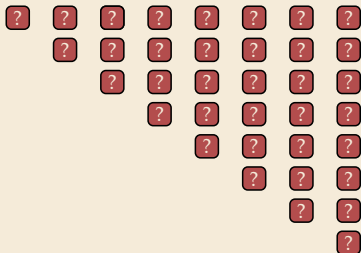
$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

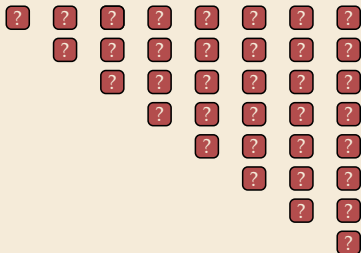


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.



endgültiger Platz erreicht!

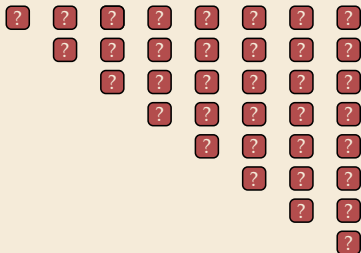


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

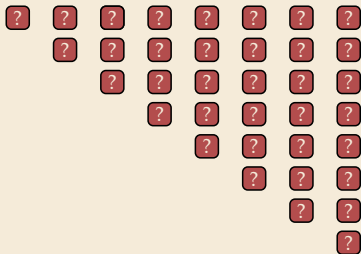


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

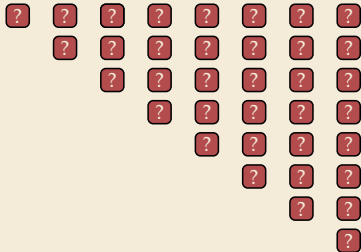


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.



$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Rang finden

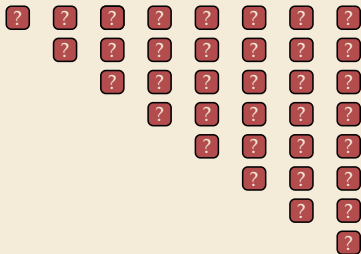
Zähle, wie viele Elements kleiner als *Pivot*.



Irgendwas von Selectionsort zu lernen?

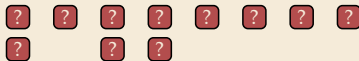
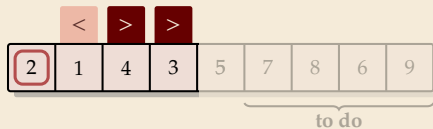
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

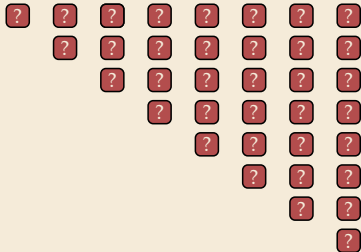


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

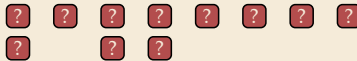
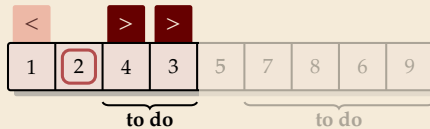
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

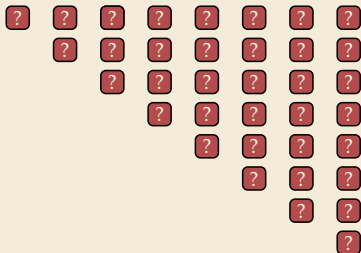


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

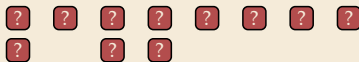
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

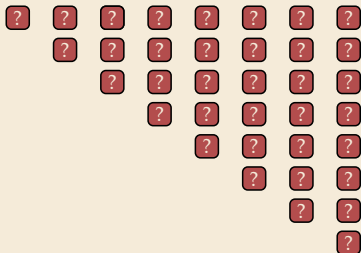


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

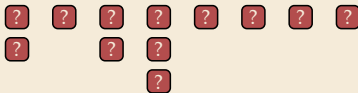
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

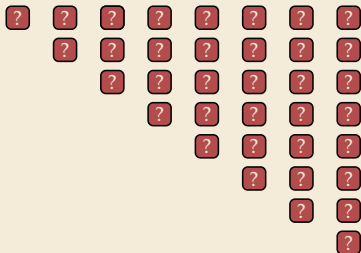


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

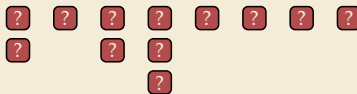
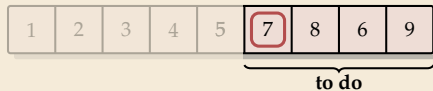
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

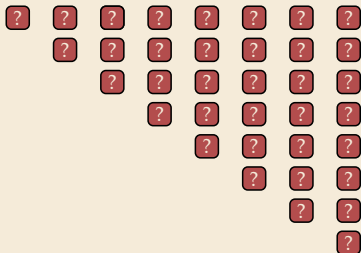


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

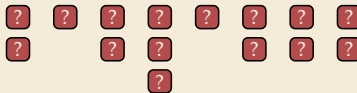
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

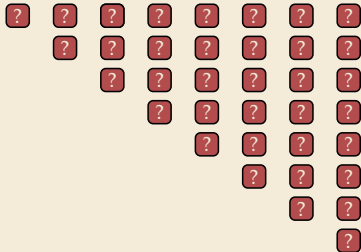


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

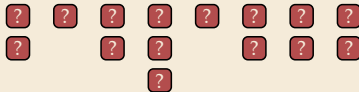
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

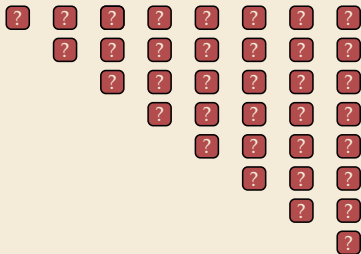


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

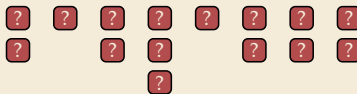
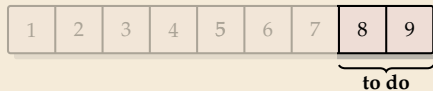
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

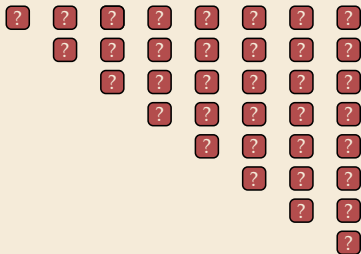


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

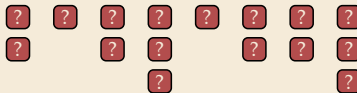
Selectionsort

Finde Minimum und tausche nach links.



Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

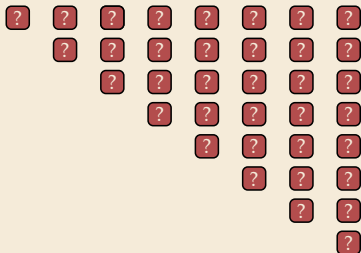


$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Irgendwas von Selectionsort zu lernen?

Selectionsort

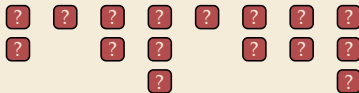
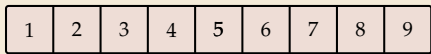
Finde Minimum und tausche nach links.



$n - 1$ Vergleiche um 1 Element loszuwerden ...
können wir damit nicht mehr machen?

Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.



$n - 1$ Vergleiche um 1 Rang zu bestimmen ...
viel mehr Fortschritt!

Quicksort

Mit einer **rekursiven** Methode ist Verwaltung der to-do-Teile einfach!

```
1 static <Elem> void quicksort(Elem[] A, Comparator<Elem> c) {
2     quicksort(A, 0, A.length, c);
3 }
4
5 /** Sortiert A[l..r] */
6 static <Elem> void quicksort(Elem[] A, int l, int r,
7                             Comparator<Elem> c) {
8     if (r - l <= 1) return; // n ≤ 1 -> schon sortiert
9     // 1. unsortiert
10    int p = partition(A, l, r, c); // Details unten
11    // 2. partitioniert, A[p] an endgültiger Position!
12    quicksort(A, l, p, c);
13    // 3. A[l..p] sortiert
14    quicksort(A, p+1, r, c);
15    // 4. alles sortiert
16 }
```

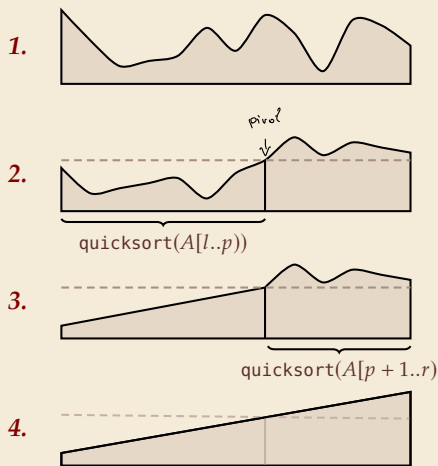
Quicksort

Mit einer **rekursiven** Methode ist Verwaltung der to-do-Teile einfach!

```
1 static <Elem> void quicksort(Elem[] A, Comparator<Elem> c) {
2     quicksort(A, 0, A.length, c);
3 }
4
5 /** Sortiert A[l..r) */
6 static <Elem> void quicksort(Elem[] A, int l, int r,
7     Comparator<Elem> c) {
8     if (r - l <= 1) return; // n ≤ 1 -> schon sortiert
9     // 1. unsortiert
10    int p = partition(A, l, r, c); // Details unten
11    // 2. partitioniert, A[p] an endgültiger Position!
12    quicksort(A, l, p, c);
13    // 3. A[l..p] sortiert
14    quicksort(A, p+1, r, c);
15    // 4. alles sortiert
16 }
```

“Writing code for partition wasn’t too difficult, but when it came to keeping a record of the ranges awaiting treatment, I couldn’t manage it. [...] [When I read about recursion], I realized that I had the solution to my coding problem: If I use recursion, everything would look very simple.”

Tony Hoare, https://youtu.be/WZqsjdbW_Es&t=348s



Partitionieren – Einfache Version

Erste Version: Wir separieren Elemente die größer bzw. kleiner als ein „Pivot-Element“ sind

```
1 @SuppressWarnings("unchecked")           A[l..r)
2 public static <Elem> int partition(Elem[] A, int l, int r,
3                                   Comparator<Elem> c) {
4     Elem[] smaller = (Elem[]) new Object[r-l];
5     Elem[] larger  = (Elem[]) new Object[r-l];
6     Elem pivot = A[l]; // fixe Pivotwahl ⚡
7     int s = 0, g = 0;
8     for (int i = l; i < r; ++i) {
9         if (less(A[i], pivot, c))
10            smaller[s++] = A[i];
11        else larger [g++] = A[i];
12    }
13    System.arraycopy(smaller, 0, A, l, s);
14    System.arraycopy(larger, 0, A, l+s, g);
15    return l + s;
16 }
```

► Verwenden Zusatzspeicher, um Elemente zwischenzulagern

↪ Müssen Sie am Ende zurückkopieren

△ Pivot bei `large` mit `drin`
aber immer in `large[0]`

Partitionieren – Einfache Version

Erste Version: Wir separieren Elemente die größer bzw. kleiner als ein „Pivot-Element“ sind

```
1 @SuppressWarnings("unchecked")
2 public static <Elem> int partition(Elem[] A, int l, int r,
3                                 Comparator<Elem> c) {
4     Elem[] smaller = (Elem[]) new Object[r-l];
5     Elem[] larger  = (Elem[]) new Object[r-l];
6     Elem pivot = A[l]; // fixe Pivotwahl ⚡
7     int s = 0, g = 0;
8     for (int i = l; i < r; ++i) {
9         if (less(A[i], pivot, c))
10            smaller[s++] = A[i];
11        else larger [g++] = A[i];
12    }
13    System.arraycopy(smaller, 0, A, l, s);
14    System.arraycopy(larger, 0, A, l+s, g);
15    return l + s;
16 }
```

► Verwenden Zusatzspeicher, um Elemente zwischenzulagern

↪ Müssen Sie am Ende zurückkopieren

► Rückgabewert ist Position des Pivots
↪ brauchen wir für die Rekursion!

Partitionieren – Einfache Version

Erste Version: Wir separieren Elemente die größer bzw. kleiner als ein „Pivot-Element“ sind

```
1 @SuppressWarnings("unchecked")
2 public static <Elem> int partition(Elem[] A, int l, int r,
3                                 Comparator<Elem> c) {
4     Elem[] smaller = (Elem[]) new Object[r-l];
5     Elem[] larger  = (Elem[]) new Object[r-l];
6     Elem pivot = A[l]; // fixe Pivotwahl ⚡
7     int s = 0, g = 0;
8     for (int i = l; i < r; ++i) {
9         if (less(A[i], pivot, c))
10            smaller[s++] = A[i];
11        else larger [g++] = A[i];
12    }
13    System.arraycopy(smaller, 0, A, l,  s);
14    System.arraycopy(larger,  0, A, l+s, g);
15    return l + s;
16 }
```

▶ Verwenden Zusatzspeicher, um Elemente zwischenzulagern

↪ Müssen Sie am Ende zurückkopieren

▶ Rückgabewert ist Position des Pivots

↪ brauchen wir für die Rekursion!

▶ *Pivot wie hier zu wählen, ist eine ganz schlechte Idee ... mehr dazu später!*

Partitionieren – Einfache Version

Erste Version: Wir separieren Elemente die größer bzw. kleiner als ein „Pivot-Element“ sind

```
1 @SuppressWarnings("unchecked")
2 public static <Elem> int partition(Elem[] A, int l, int r,
3                                 Comparator<Elem> c) {
4     Elem[] smaller = (Elem[]) new Object[r-l];
5     Elem[] larger = (Elem[]) new Object[r-l];
6     Elem pivot = A[l]; // fixe Pivotwahl ⚡
7     int s = 0, g = 0;
8     for (int i = l; i < r; ++i) {
9         if (less(A[i], pivot, c))
10            smaller[s++] = A[i];
11        else larger [g++] = A[i];
12    }
13    System.arraycopy(smaller, 0, A, l, s);
14    System.arraycopy(larger, 0, A, l+s, g);
15    return l + s;
16 }
```

- ▶ Verwenden Zusatzspeicher, um Elemente zwischenzulagern
↪ Müssen Sie am Ende zurückkopieren
- ▶ Rückgabewert ist Position des Pivots
↪ brauchen wir für die Rekursion!
- ▶ *Pivot wie hier zu wählen, ist eine ganz schlechte Idee ... mehr dazu später!*
- ▶ Java: @SuppressWarnings und cast wegen generic arrays ... 😬

- ▶ Java: System.arraycopy schneller zum Kopieren großer Arrays als eine Schleife

```
1 /** Kopiert B[j..j+n) := A[i..i+n) */
2 static void System.arraycopy(Object[] A, int i, Object[] B, int j, int n) {
3     // direkter Aufruf von JVM an Betriebssystem, daher idR schneller
4 }
```

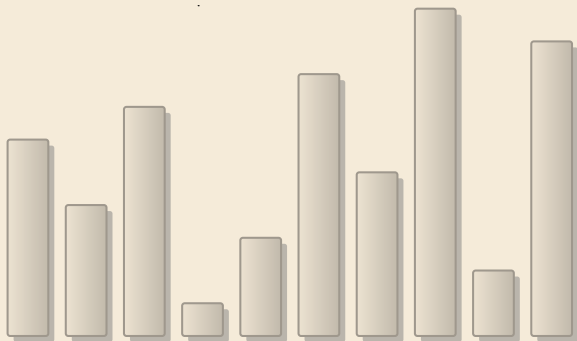
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



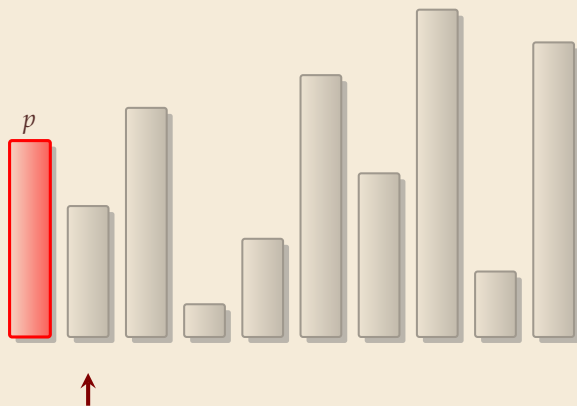
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



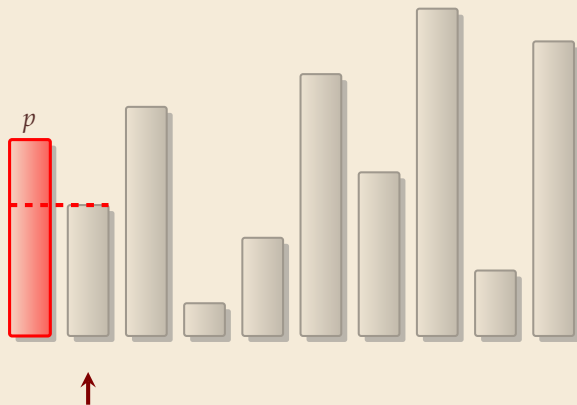
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



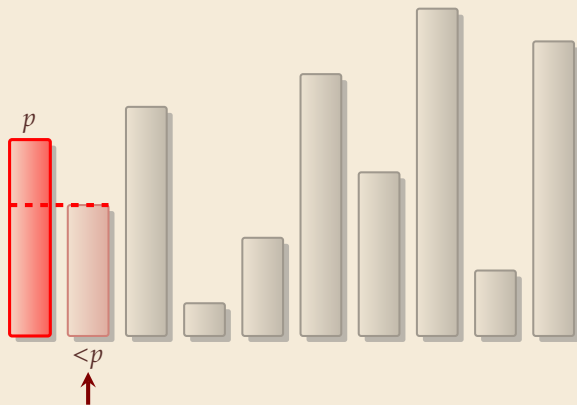
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



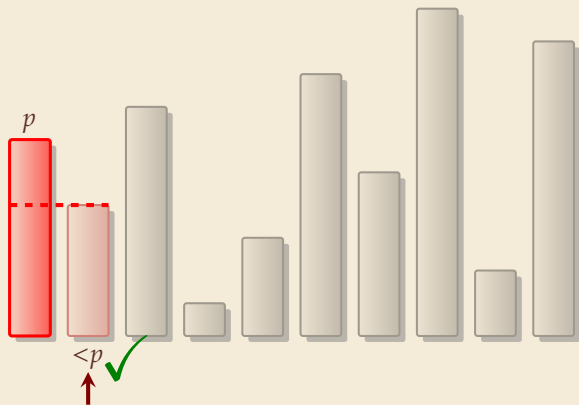
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



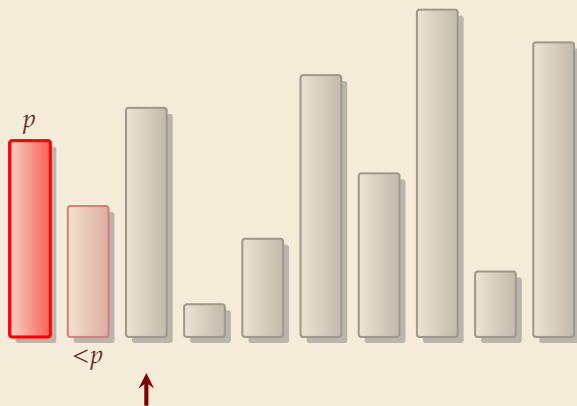
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



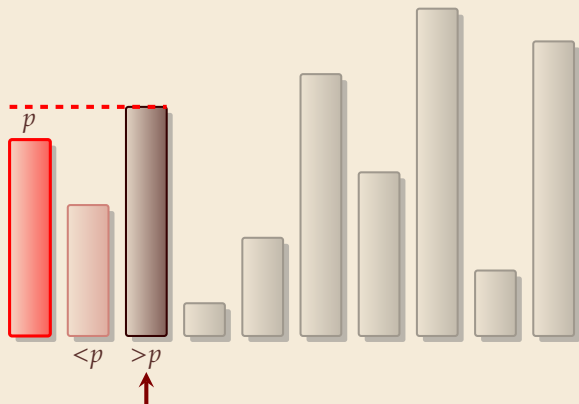
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



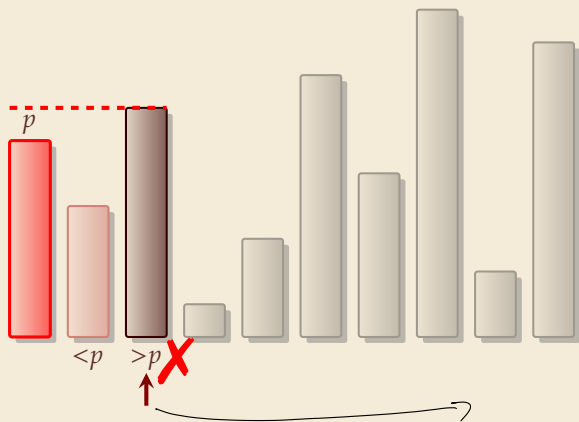
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



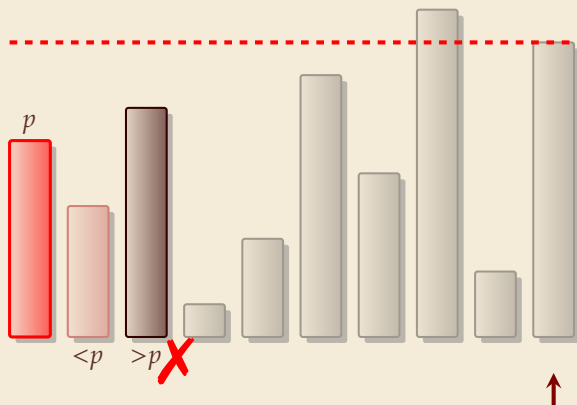
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



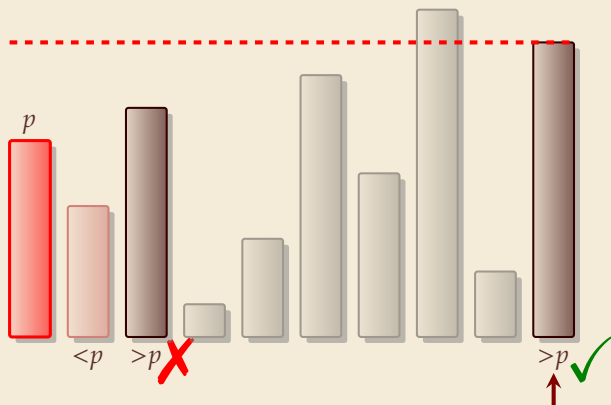
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



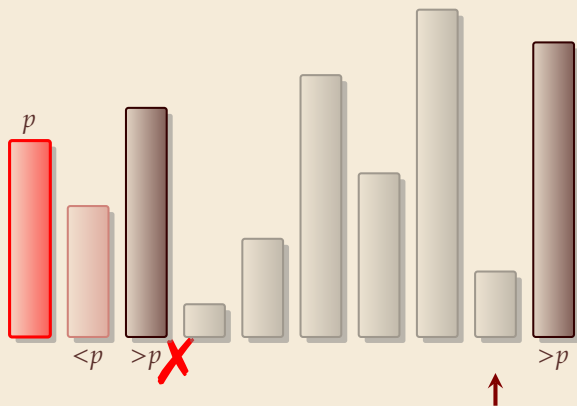
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



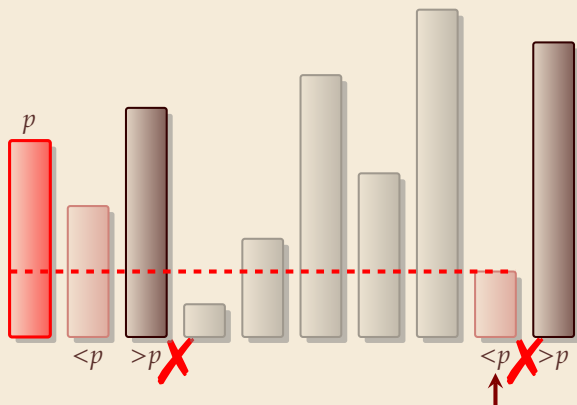
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



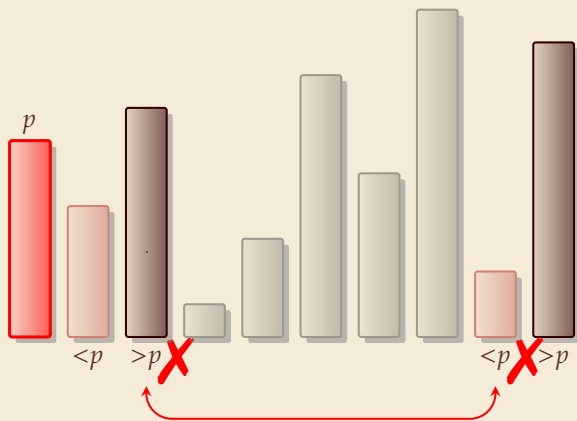
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



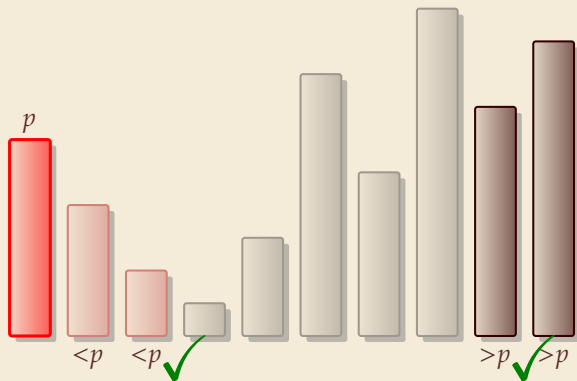
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



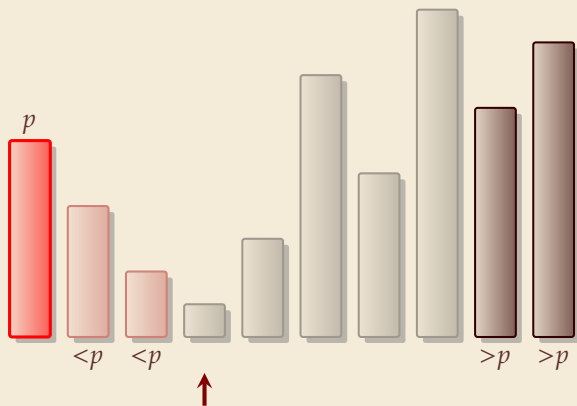
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



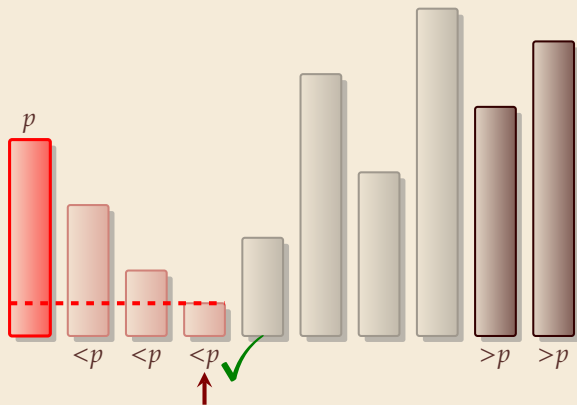
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



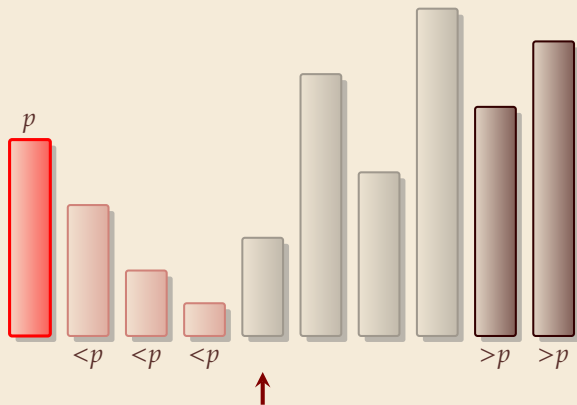
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



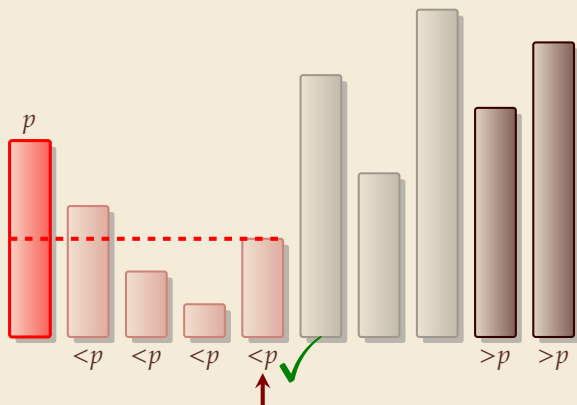
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



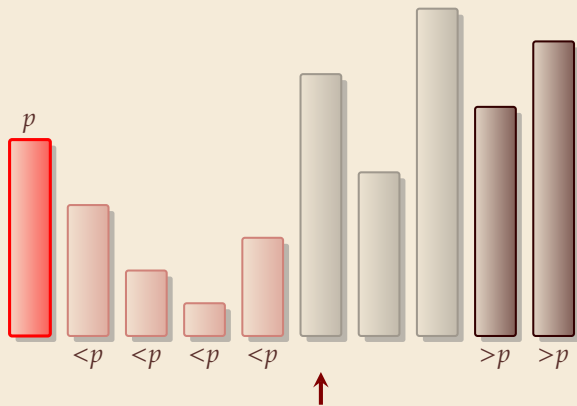
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



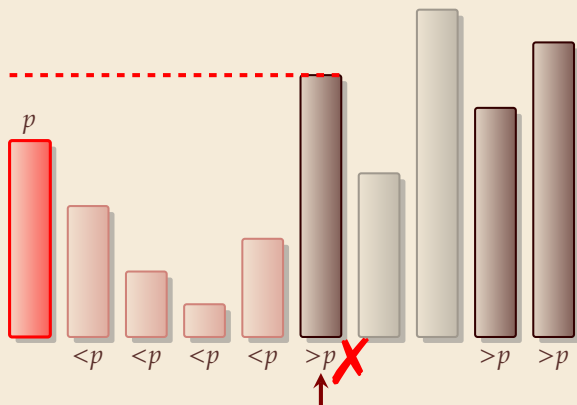
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



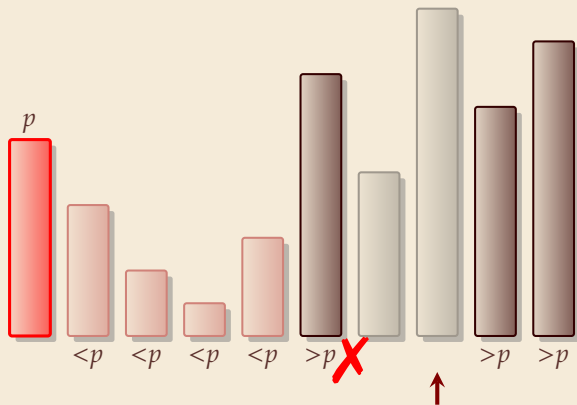
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



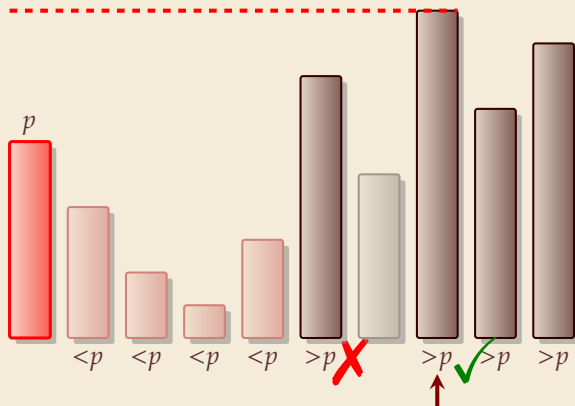
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



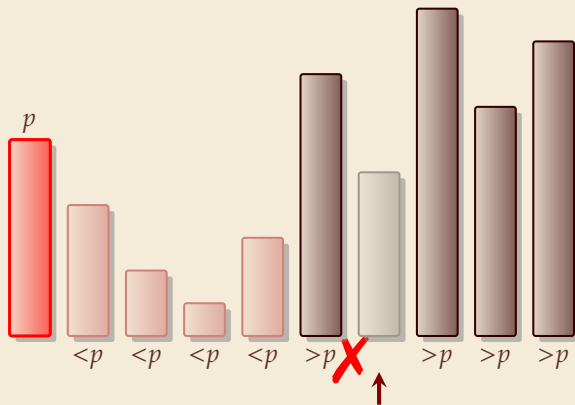
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



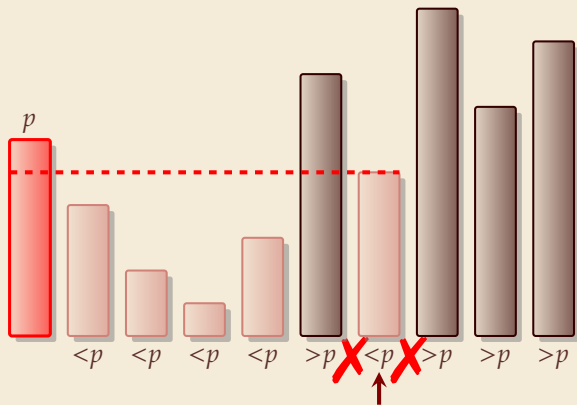
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



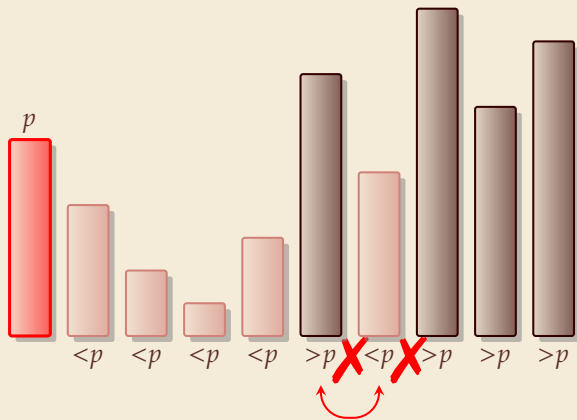
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



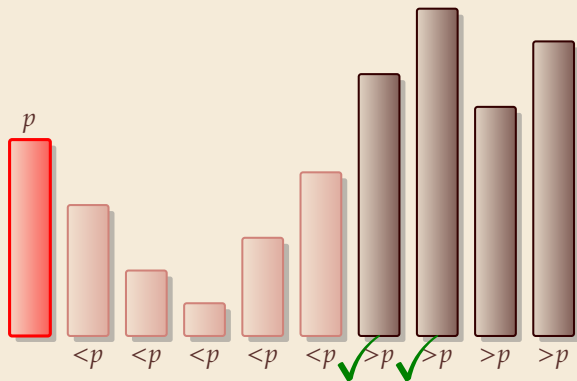
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



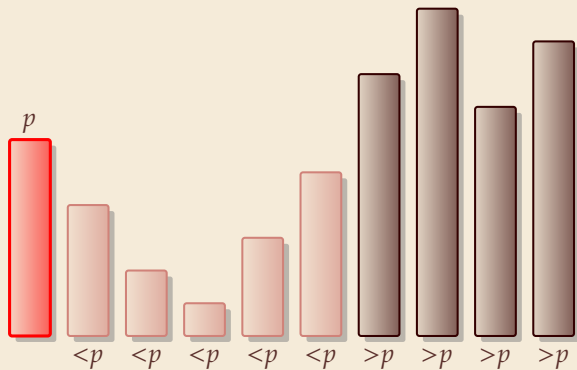
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



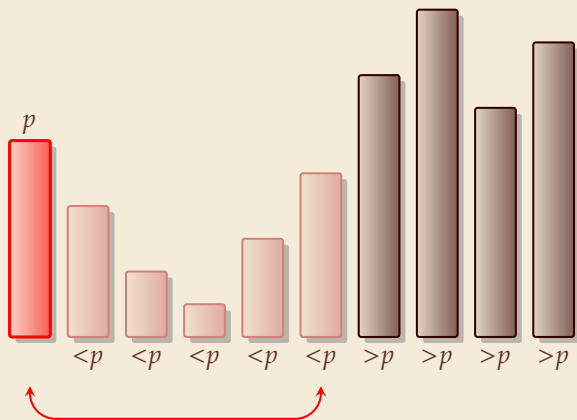
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



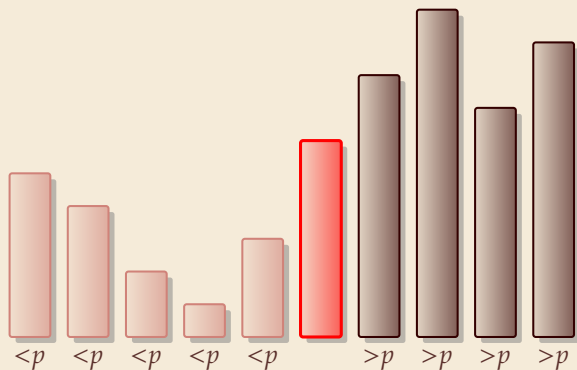
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



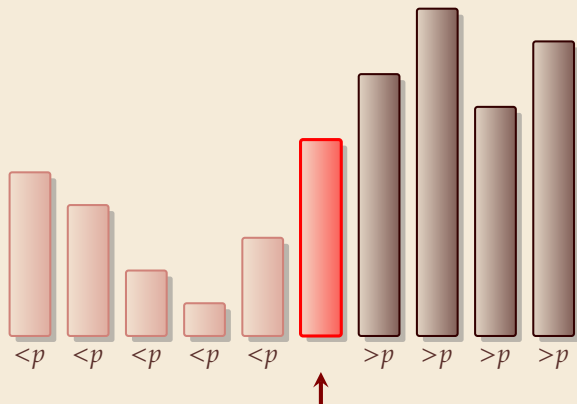
Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



Inplace Partition

Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!



- ▶ kein Zusatzspeicher!
- ▶ nur zwei Indizes + Pivot
- ▶ betrachten jedes Element genau einmal
- ▶ wo Indizes sich treffen gehört das Pivot hin

Hoare-Sedgewick Partitionierung

Idee mit kreuzenden Indizes von Tony Hoare, mit Verfeinerungen von Robert Sedgewick

Achtung: Voll subtiler Details! Verwenden Sie exakt diesen Code

(falls Sie das je brauchen)

```
1 static <Elem> int partition(Elem[] A, int l, int r,
2                             Comparator<Elem> c) {
3     swap(A, l, choosePivot(l, r));
4     int i = l, j = r;
5     while (true) {
6         do ++i; while (i < r && less(A[i], A[l], c));
7         do --j; while (j > l && less(A[l], A[j], c));
8         if (i >= j) break;
9         swap(A, i, j);
10    }
11    swap(A, l, j);
12    return j;
13 }
```

► Wahl des Pivots über einen Index
choosePivot(l , r) $\in [l..r)$

↪ Details dazu gleich

Subtilitäten (\notin exam)

- Beide do-while-Schleifen stoppen bei Duplikaten des Pivots
- Check $j > l$ könnte man weglassen (immer wahr)
- für paarweise verschiedene Elemente, verlassen äußere Schleife immer mit $i = j + 1$;
 $i = j$ aber auch möglich, wenn $A[i] = A[j] = A[l]$

Loop invariant (5–10):

$A[l..r)$

