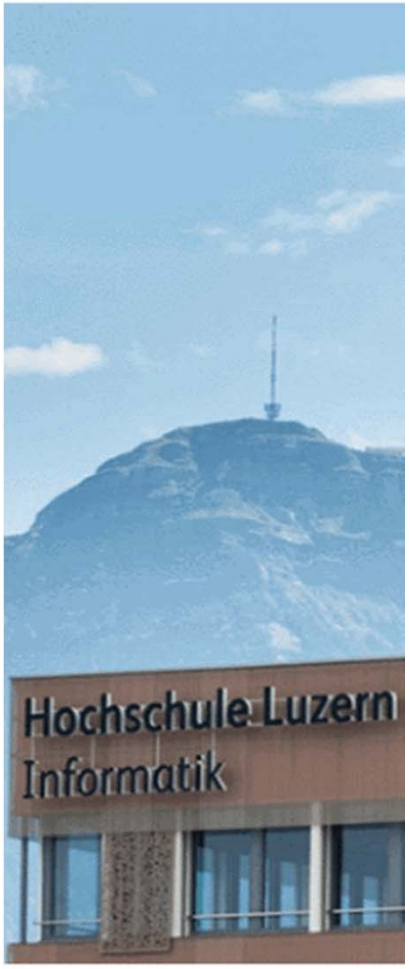
Algorithmen & Datenstrukturen

# Einfache Sortieralgorithmen

Hansjörg Diethelm

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
INFORMATIK



# Inhalt

- Direktes Einfügen (Insertion Sort)
- Direktes Auswählen (Selection Sort)
- Direktes Austauschen (Bubble Sort)
- Shellsort

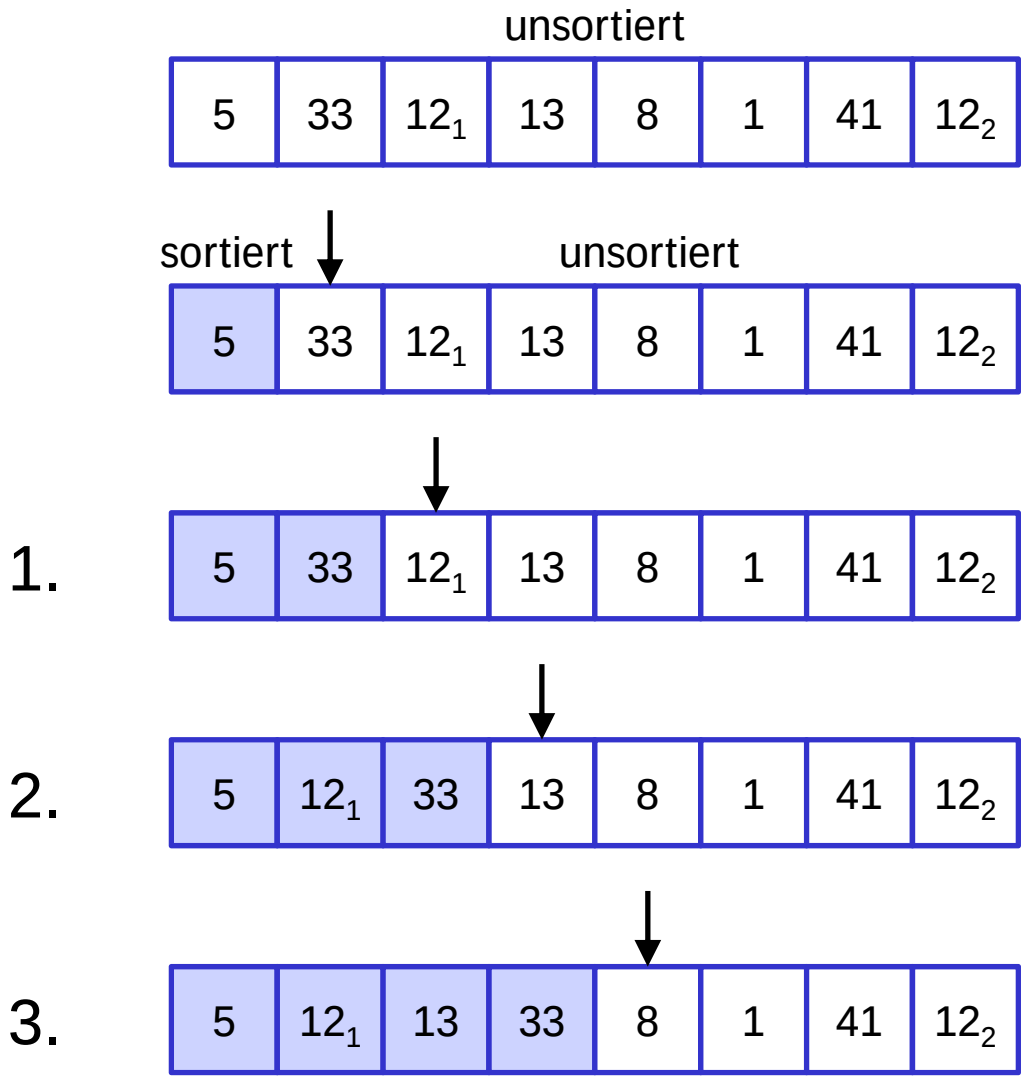
## Lernziele

Sie …

- können einfache Sortieralgorithmen für kleine Datenmengen exemplarisch auf Papier durchspielen.
- kennen die wesentlichen Eigenschaften der einfachen Sortieralgorithmen, vgl. Stabilität, Komplexität, Eigenheiten.
- können einfache Sortieralgorithmen bezüglich ihrer Komplexität analysieren.
- können einfache Sortieralgorithmen implementieren.

# Direktes Einfügen (Insertion Sort)

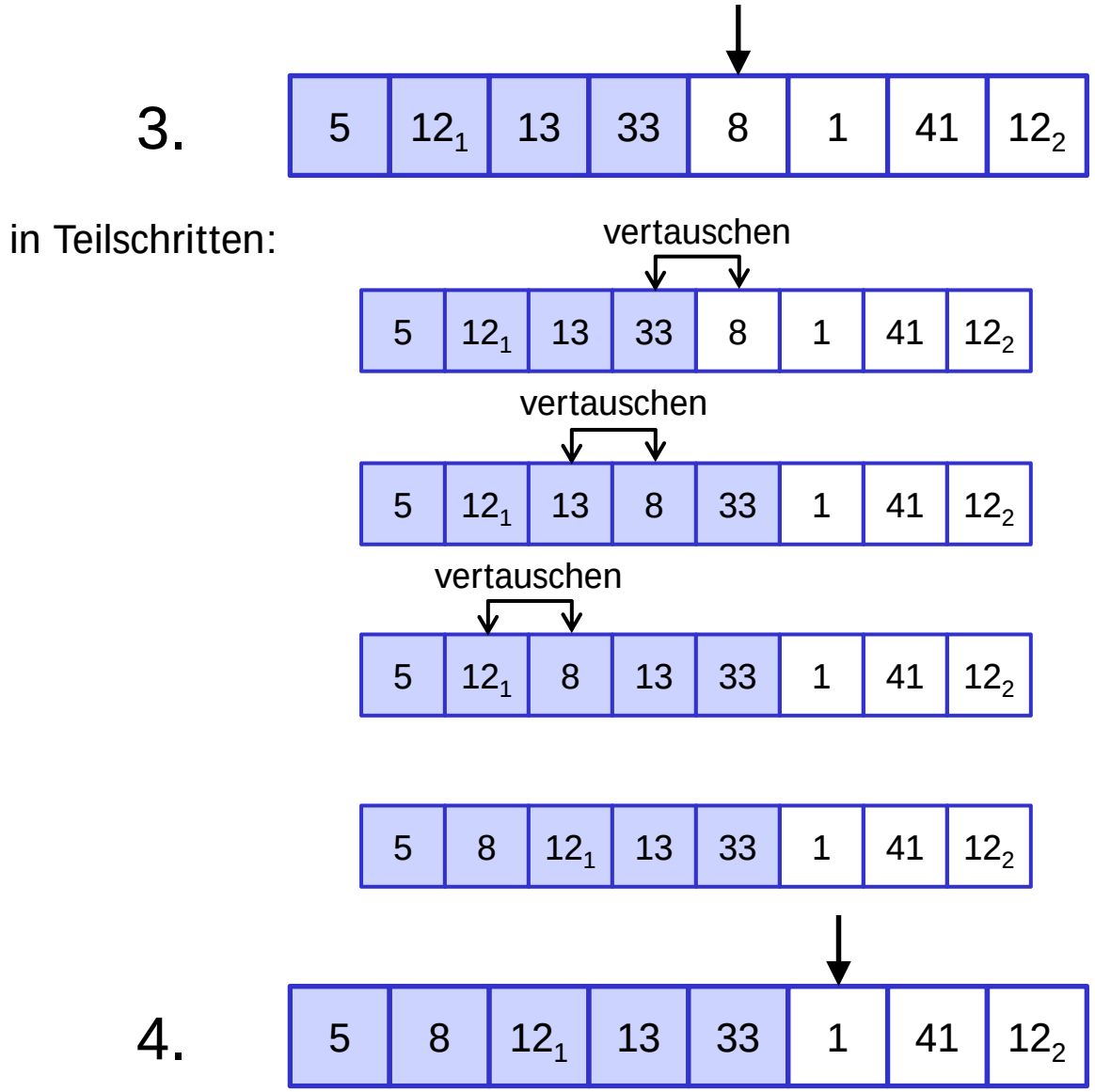
# Direktes Einfügen – Prinzip



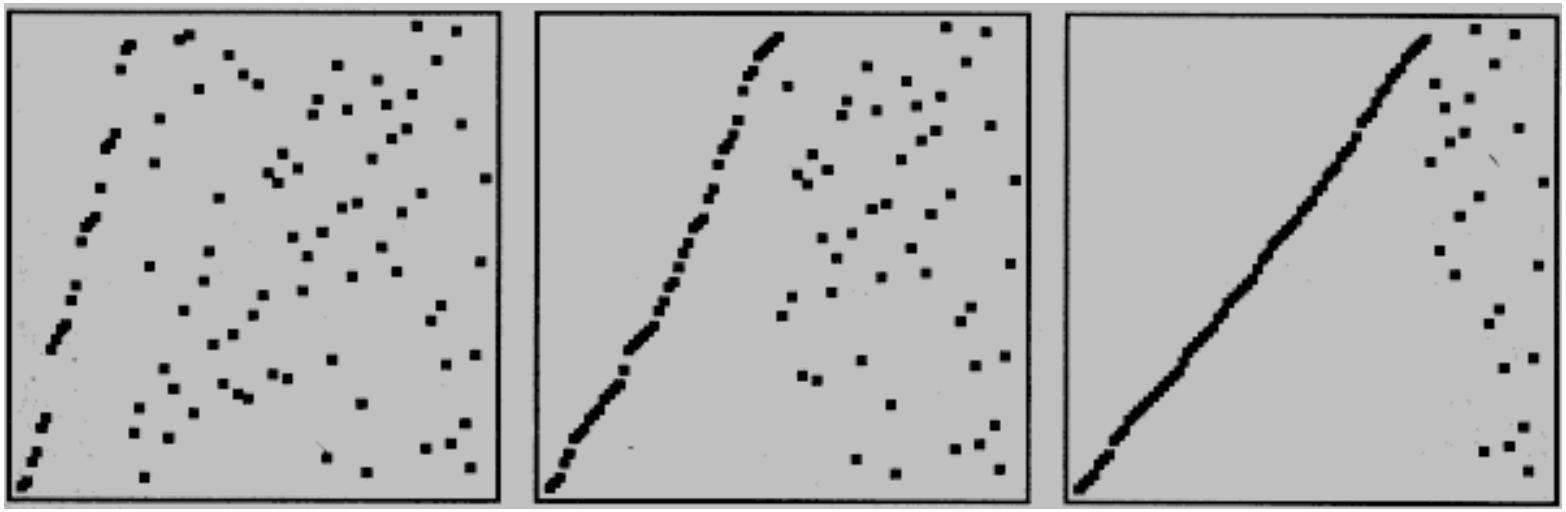
## Direktes Einfügen – Prinzip

- Das zu sortierende Array wird in einen unsortierten Teil (weiss) und einen sortierten Teil (blau) unterteilt.
- Das jeweils **erste Element im unsortierten Teil** wird gemäss seinem Schlüssel (d.h. gemäss Ordnung) **im sortierten Teil eingefügt**.
- Alle Elemente mit grösserem Schlüssel müssen somit im sortierten Teil nach rechts verschoben werden.
- Die Suche nach der Einfüge-Position wird typisch mit dem Verschieben kombiniert.
- Nach **n-1 mal Einfügen** ist das Array sortiert.

# Direktes Einfügen – Prinzip



# Direktes Einfügen – Prinzip



# Direktes Einfügen – `insertionSort()`

```
/**
 * Sortiert das int-Array aufsteigend.
 * @param a Zu sortierendes Array.
 */
public static void insertionSort(final int[] a) {
    int elt;
    int j;
    for (int i = 1; i < a.length; i++) {
        elt = a[i];           // next elt for insert
        j = i;                // a[0]..a[j-1] already sorted
        while ((j > 0) && (a[j - 1] > elt)) {
            a[j] = a[j - 1]; // shift right
            j--;              // go further left
        }
        a[j] = elt;           // insert elt
    }                         // a[0]...a[j] sorted
}
```

## Direktes Einfügen – Optimierung

- **Bei jedem Eintreten in die while-Schleife** sind **zwei Bedingungen** zu prüfen, was **relativ aufwendig** ist:
  - `(j > 0)` stellt sicher, dass spätestens am linken Ende des Arrays die Schleife beendet wird.
  - `(a[j-1] > elt)` dient zum Finden der Einfügestelle.

- Mit einem häufig verwendeten **Trick** kann auf die erste Bedingung verzichtet werden:
  - `a[0]` dient neu nur noch zum Speichern eines **Dummy-Elementes**. Das jeweils einzufügende Element `elt` wird dort vor dem Suchen der Einfügestelle gespeichert. Dies garantiert, dass spätestens am linken Ende `(a[j-1] > elt)` nicht mehr erfüllt ist und somit die Schleife terminiert.

# Direktes Einfügen – `insertionSort2()`

```
/**
 * Sortiert das int-Array aufsteigend, erst ab Index a[1].
 * @param a Zu sortierendes Array.
 */
public static void insertionSort2(final int[] a) {
    int elt;
    int j;
    for (int i = 2; i < a.length; i++) {
        elt = a[i];           // next elt for insert
        a[0] = elt;           // dummy-element
        j = i;                // a[1]..a[j-1] already sorted
        while (a[j - 1] > elt) {
            a[j] = a[j - 1]; // shift right
            j--;             // go further left
        }
        a[j] = elt;           // insert elt
    }                         // a[1]...a[j] sorted
}
```

## Direktes Einfügen – Analyse «Best Case»

- Das Array liegt **bereits sortiert** vor!
- Pro Durchlauf von i ist nur **1** Vergleich (**C**ompare) `(a[j-1] > elt)` erforderlich.

➔ $C_{\min} = \sum_{i=2}^{n} 1 = n - 1$

➔ Best Case: **O(n)**

## Direktes Einfügen – Analyse «Worst Case»

- Das Array liegt **umgekehrt sortiert** vor!
- Pro Durchlauf von i sind **1 + (i - 1) = i** Vergleiche `(a[j-1] > elt)` erforderlich.

➔ $C_{\max} = \sum_{i=2}^{n} i = \frac{n}{2} \cdot (n+1) - 1 = \frac{1}{2} \cdot (n^2 + n - 2)$

➔ Worst Case: **O(n²)**

vgl. «Gauss»: $\sum_{i=1}^{n} i = \frac{n}{2} \cdot (n+1)$

## Direktes Einfügen – Analyse «Average Case»

- Im Durchschnitt kommt das einzufügende Element **in die Mitte** des bereits sortierten Teils zu liegen.
- Pro Durchlauf von i sind somit **1 + (i - 1)/2 = (i + 1)/2** Vergleiche `(a[j-1] > elt)` erforderlich.

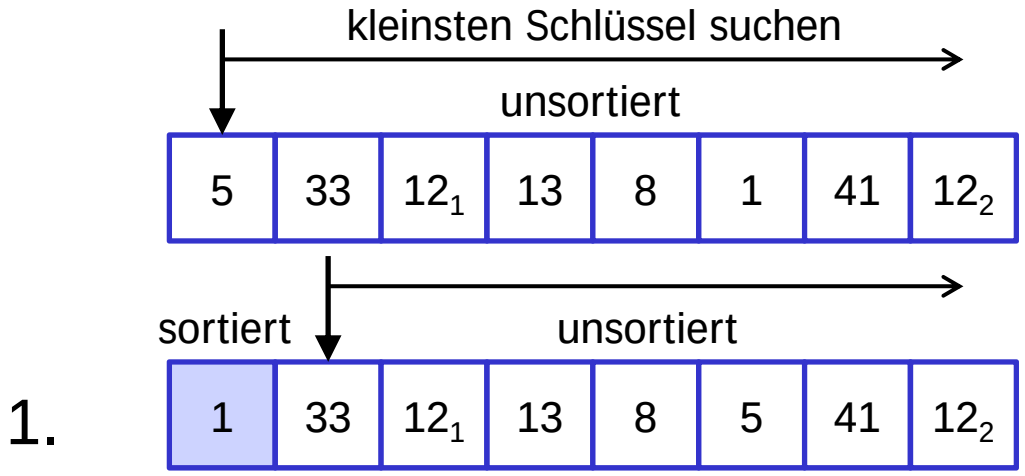
➔ $$C_{mittel} = \frac{1}{2} \cdot \sum_{i=2}^{n} (i+1) = \frac{1}{2} \cdot \left( (n-1) \cdot 1 + \sum_{i=2}^{n} i \right) =$$

$$\frac{1}{2} \cdot \left( n - 1 + \frac{n}{2} \cdot (n+1) - 1 \right) = \frac{1}{4} \cdot (n^2 + 3n - 4)$$

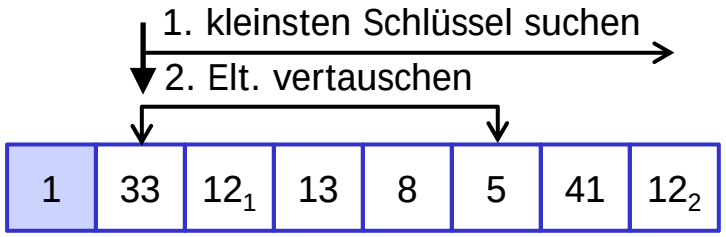
➔ Average Case: **O(n²)**

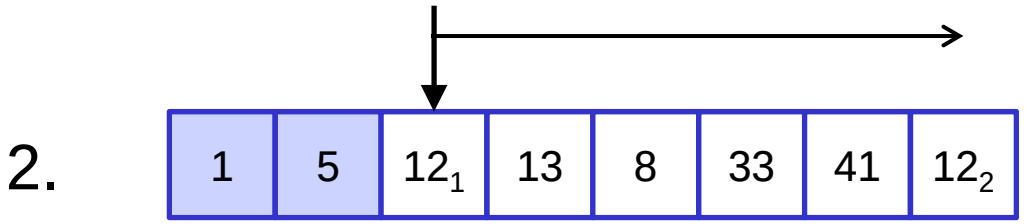
# Direktes Auswählen (Selection Sort)

# Direktes Auswählen – Prinzip




in Teilschritten:

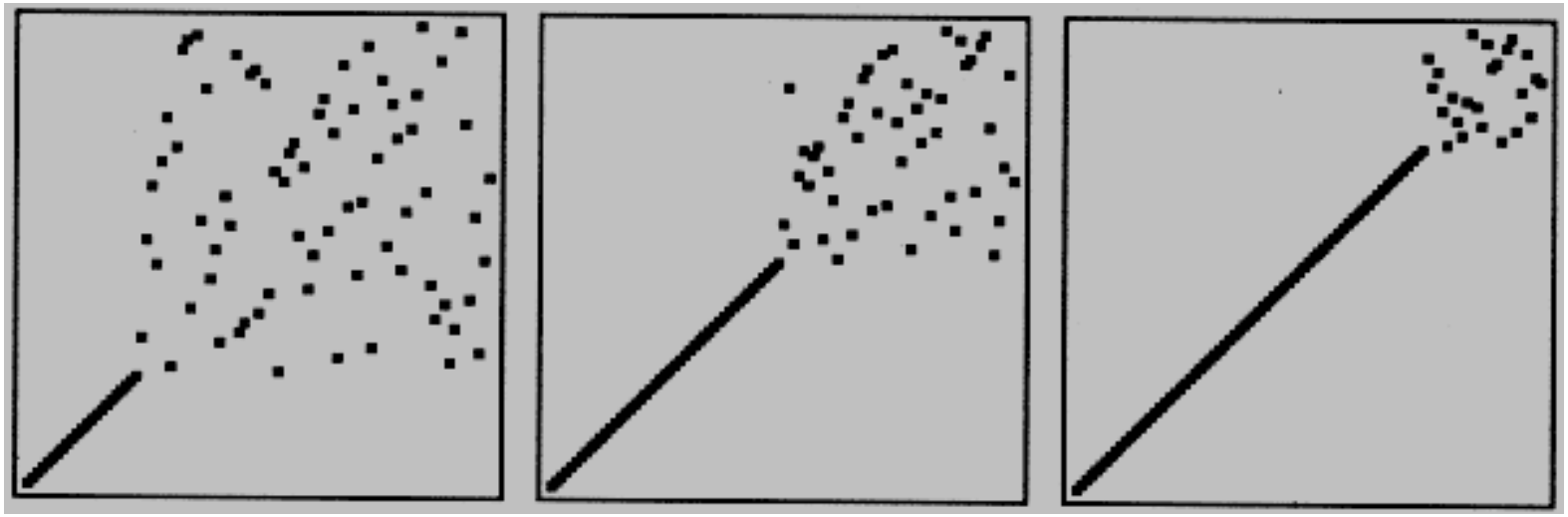


## Direktes Auswählen – Prinzip

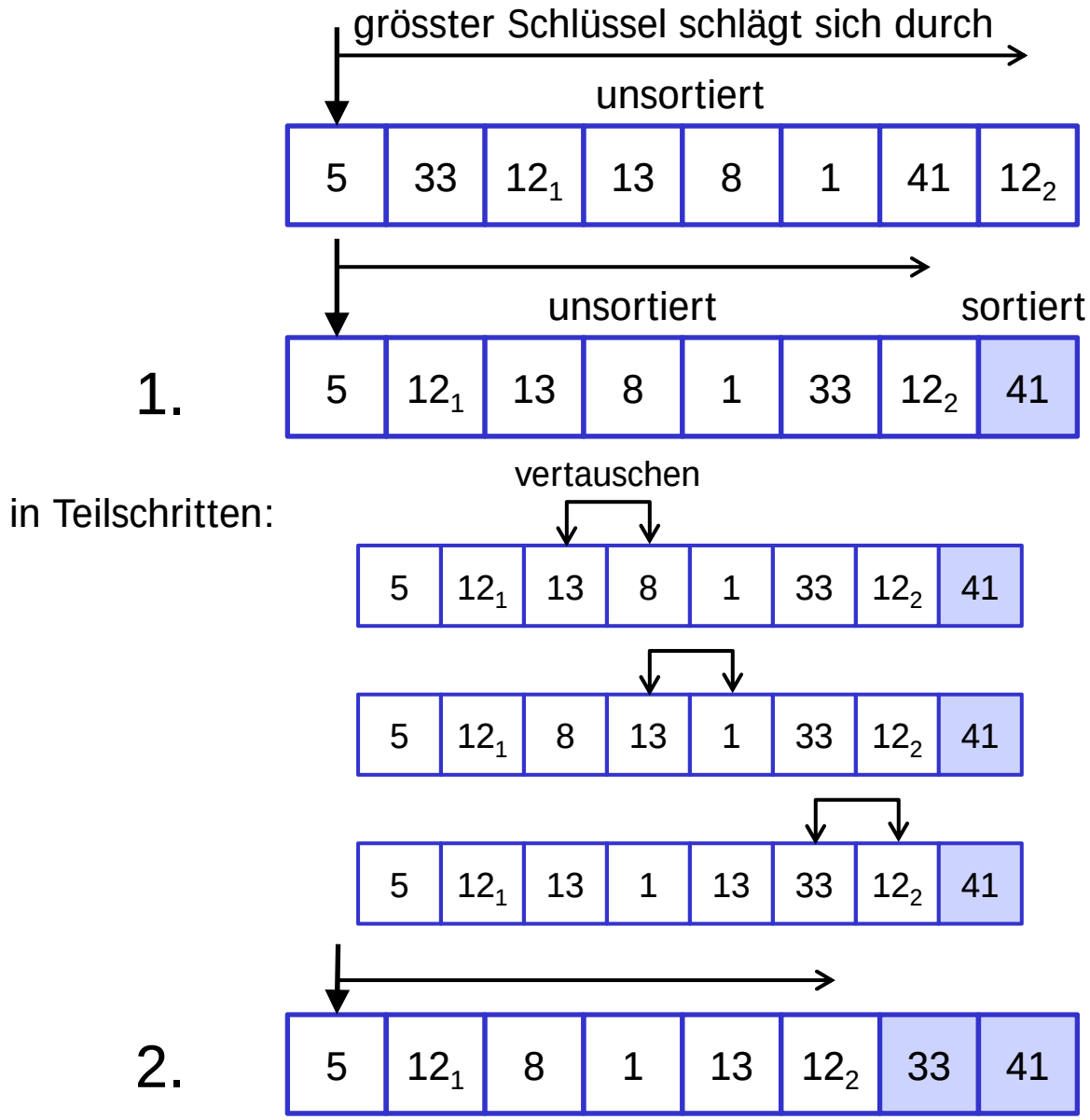
- Das zu sortierende Array wird in einen unsortierten Teil (weiss) und einen sortierten Teil (blau) unterteilt.
- **Im unsortierten Teil** des Arrays wird das **Element mit dem kleinsten Schlüssel gesucht**. Dieses vertauscht man dann mit dem ersten Element des unsortierten Teils.
- Nach **n-1 Durchläufen** ist das Array sortiert.
- Direktes Auswählen besitzt die Ordnung $\mathbf{O(n^2)}$.
- Weil zum Ermitteln des kleinsten Schlüssels immer der komplette, noch unsortierte Teil des Arrays durchlaufen werden muss, liegt direktes Auswählen nicht nur im schlechtesten Fall, sondern sogar in jedem Fall in dieser Komplexitätsklasse, d.h. im Best, Worst und Average Case.

# Direktes Auswählen – Prinzip



# Direktes Austauschen (Bubble Sort)

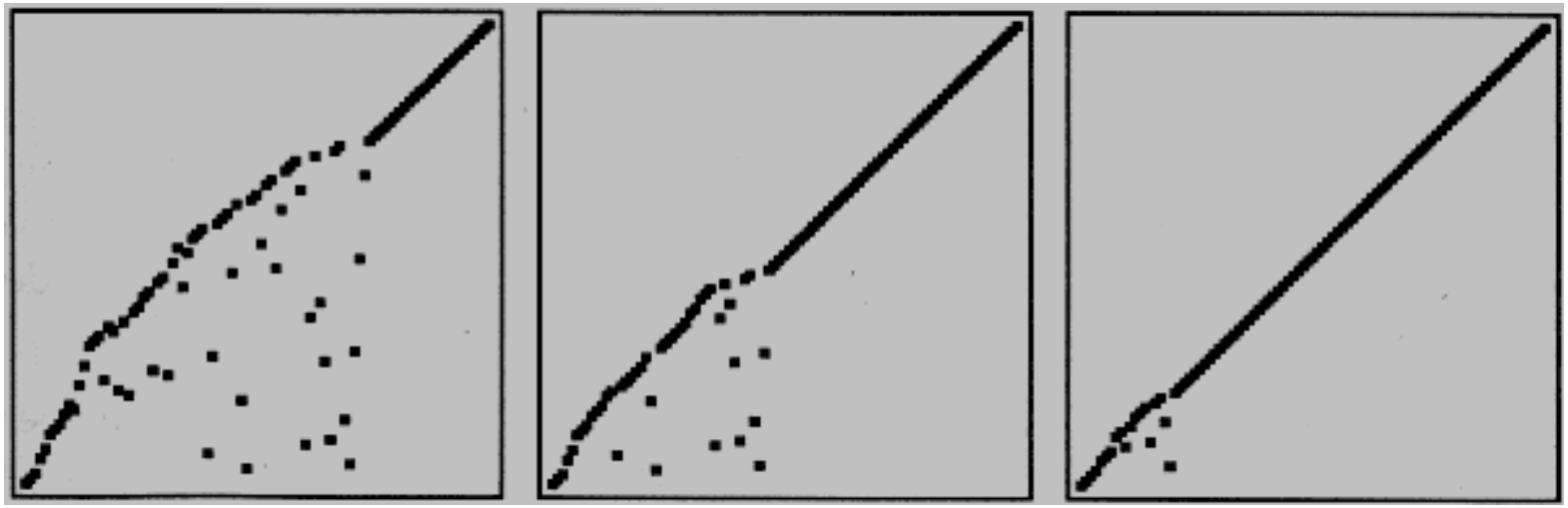
# Direktes Austauschen – Prinzip



## Direktes Austauschen – Prinzip

- Im Array wird wiederum zwischen einem unsortierten Teil (weiss) und einen sortierten Teil (blau) unterschieden.
- **Zwei benachbarte Elemente** werden einfach **ausgetauscht**, falls deren Schlüssel **nicht aufsteigend geordnet** sind.
- Während eines Durchlaufs schlägt sich das **Element mit dem grössten Schlüssel ganz nach rechts** durch, d.h. «blubbert» hoch.
- Nach **n-1 Durchläufen** ist das Array sortiert.
- Wie oben illustriert, besitzt direktes Austauschen die Ordnung **$O(n^2)$**. Der Algorithmus lässt sich einfach modifizieren, so dass er im «Best Case» mit O(n) arbeitet.
- Direktes Austauschen wird in der Praxis kaum eingesetzt.

# Direktes Austauschen – Prinzip

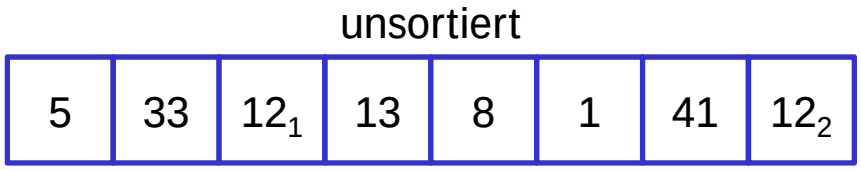


# Shellsort
# (Donald L. Shell, 1959)

## Shellsort – Prinzip

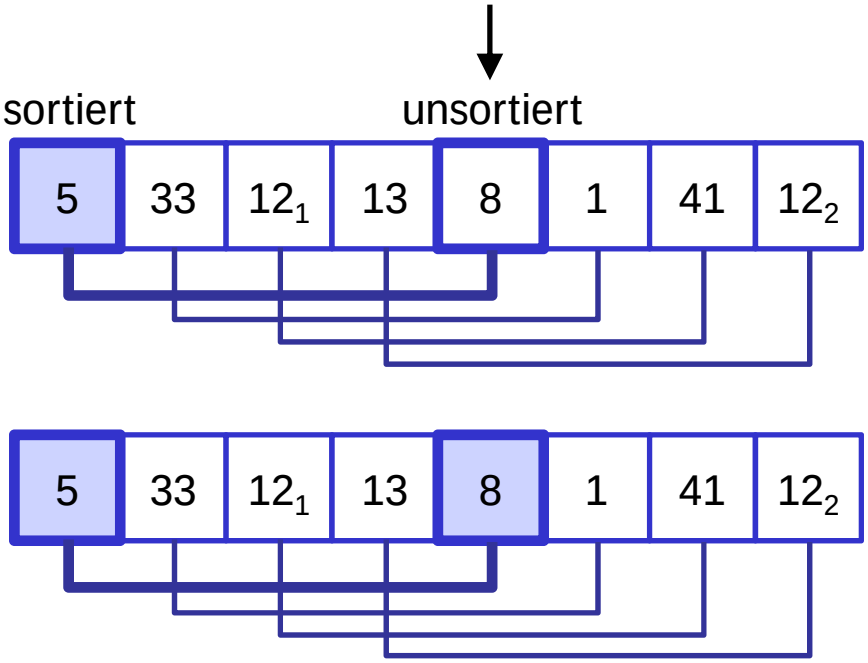
- Shellsort ist eine **Verfeinerung des direkten Einfügens**.
- Nachteilig beim direkten Einfügen war das ausschliesslich benachbarte «Sesselrücken»!
- Bei Shellsort erfolgt das **Sortieren in mehreren Stufen** von «grob» bis «fein».
- Bei der Grobsortierung erfolgt das Sortieren über «grosse Distanzen»; bei der Feinsortierung sind es – falls überhaupt noch notwendig – nur noch «kleine Distanzen».
- Das eigentliche Sortieren erfolgt aber wie beim direkten Einfügen.

# Shellsort – Prinzip (1)



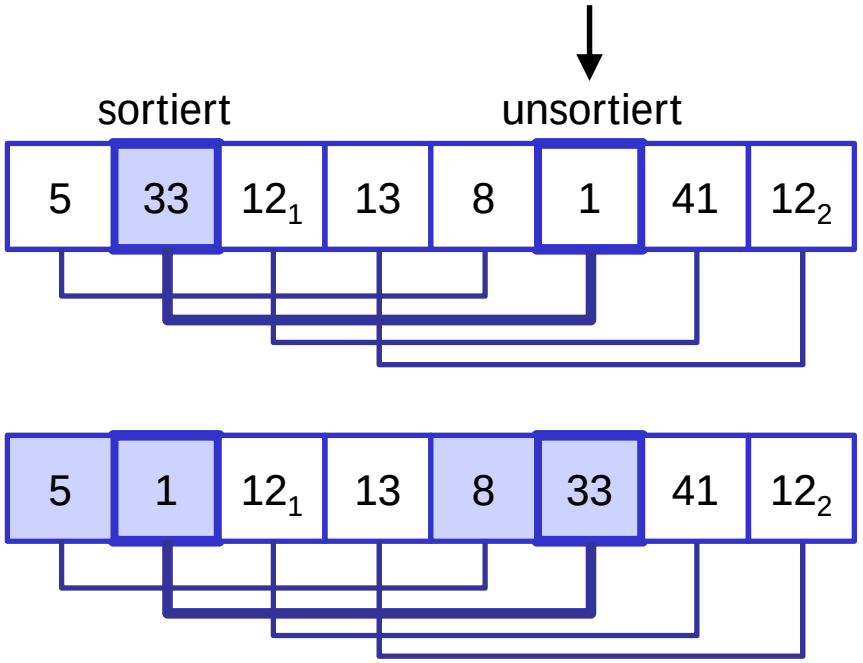

1. Vier Grobsortierungen über Schrittweite 4:

a)



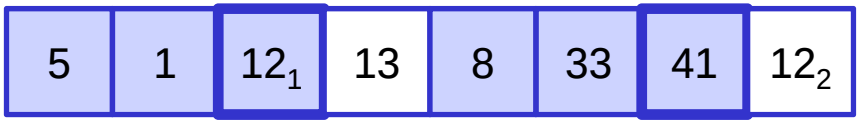
# Shellsort – Prinzip (2)

b)

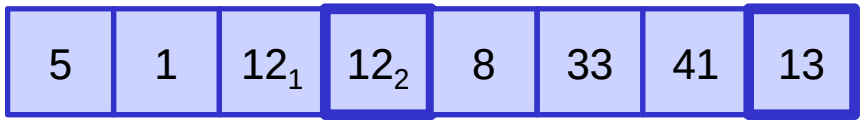



analog:

c)



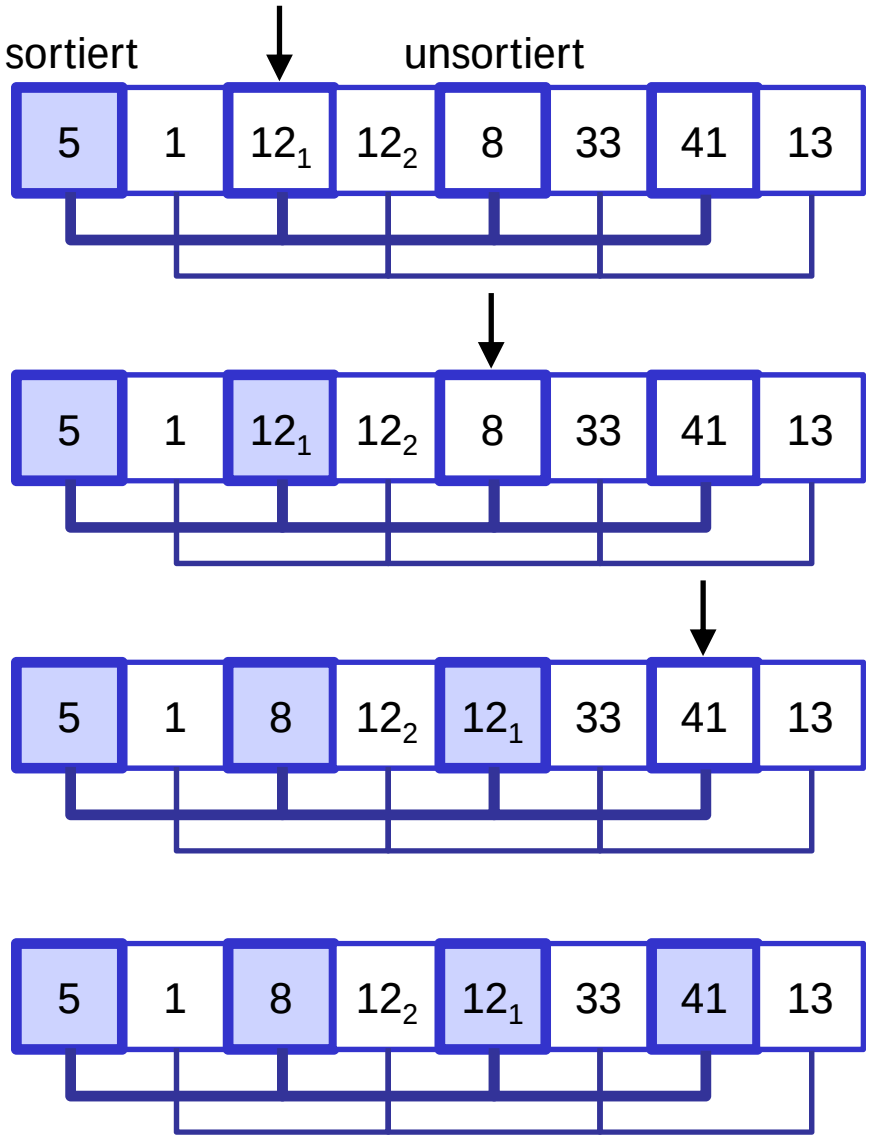

d)



# Shellsort – Prinzip (3)

2. Zwei Grobsortierungen über Schrittweite 2:

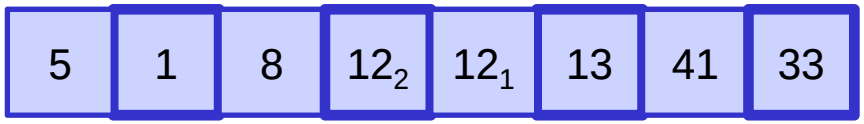
a)



# Shellsort – Prinzip (4)

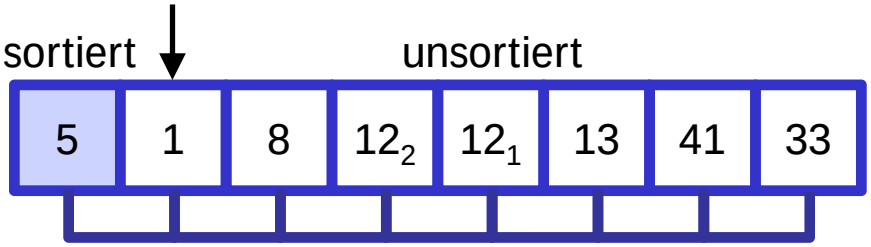
analog:

b)

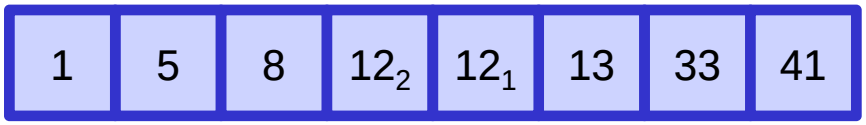



3. Eine **letzte Feinsortierung** über **Schrittweite 1**:

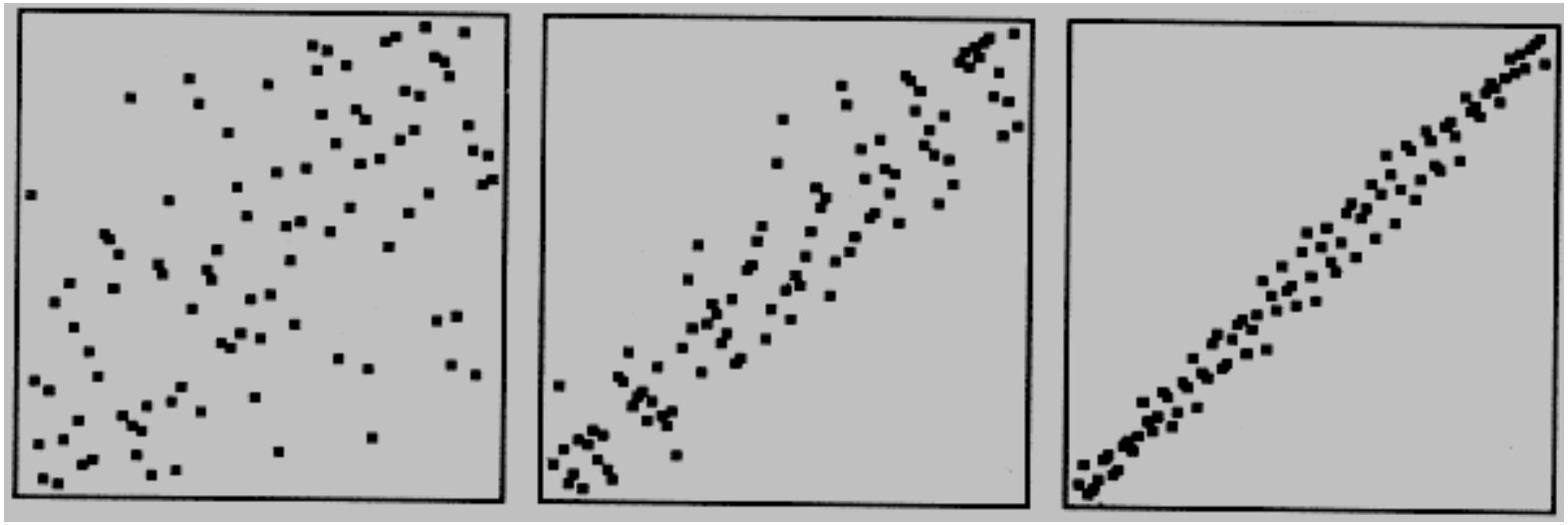
a)




fertig sortiert!:



# Shellsort – Prinzip



## Shellsort – Analyse

- Im allgemeinen Fall ist der Sortieraufwand von Shellsort wie beim direkten Einfügen ebenfalls von der Ordnung $O(n^2)$.
- Shell-Folge: 1, 2, 4, 8, …, $2^k$ bzw. «mal 2» wäre eine schlechte Wahl, vgl. «Inzucht»!
- Man kann allerdings **für gewisse Folgen von Schrittweiten** beweisen (extrem schwierig), dass damit der **Aufwand wesentlich kleiner** wird! Welches die optimalste Folge ist, kann bis heute nicht beantwortet werden.
- Knuth-Folge: 1, 4, 13, 40, …, $(3^k-1)/2$ bzw. «mal 3 + 1»
- **Hibbard-Folge**: 1, 3, 7, 15, …, $2^k-1$ bzw. «mal 2 + 1» ➔ **$O(n^{1.5})$**
- Shellsort schlägt damit eine **Brücke zwischen den einfachen und höheren Sortieralgorithmen**, vgl. $O(n^2)$ vs. $O(n \cdot \log n)$.

## Zusammenfassung

- Einfache Sortieralgorithmen besitzen typisch eine Zeitkomplexität von $O(n^2)$.
- In der Regel muss man aber zwischen Average Case, Worst Case und Best Case unterscheiden.
- Sortieralgorithmen, welche Datenelemente über grössere Distanzen verschieben, arbeiten typisch schneller.
- Sortieralgorithmen, welche nur benachbarte Datenelemente manipulieren, arbeiten typisch stabil.
- Bei den einfachen Sortieralgorithmen sind das direkte Einfügen und das direkte Auswählen zu bevorzugen.
- Mit einer Zeitkomplexität von $O(n^{1.5})$ schlägt Shellsort eine Brücke zwischen den einfachen und den höheren Sortieralgorithmen.

# Fragen?