Sortieren

Sortieren

- Aufgabe (in der Praxis): Ordnen von Dateien mit Datensätzen, die Schlüssel enthalten.
- Umordnung der Datensätze, dass eine klar definierte Ordnung der Schlüssel entsteht.
- Viele Anwendungen setzen - aus unterschiedlichen Gründen - sortierte Datenmengen voraus:
  - Kontoauszüge nach Kontonummer.
  - Prüfungsnoten nach Matrikelnummer.
  - Buchausleihe nach Namen.
  - Das Suchen wird schneller.

Sortieren : Grundbegriffe

- Interne Verfahren
  - Beim Sortieren von Hauptspeicherstrukturen (z.B. Felder).
  - Voraussetzung: die zu sortierende Folge passt in den Hauptspeicher.
- Externe Verfahren
  - Wenn der Hauptspeicher nicht ausreicht.
  - Es wird auf einem externen Speichermedium sortiert: Festplatte, Magnetband.
- Stabilität eines Sortierverfahrens
  - Sind 2 Schlüssel gleich, dann wird die relative Reihenfolge beibehalten.

Sortieren : Stabilität - Beispiel

Name	Alter
Abel, Günther	65
Krämer, Willy	52
Stein, Erwin	48
Urschel, Karin	24
Winter, Gerd	52

Sortierkriterium

Sortierkriterium

Name	Alter
Urschel, Karin	24
Stein, Erwin	48
Krämer, Willy	52
Winter, Gerd	52
Abel, Günther	65

Die Reihenfolge bleibt unverändert.

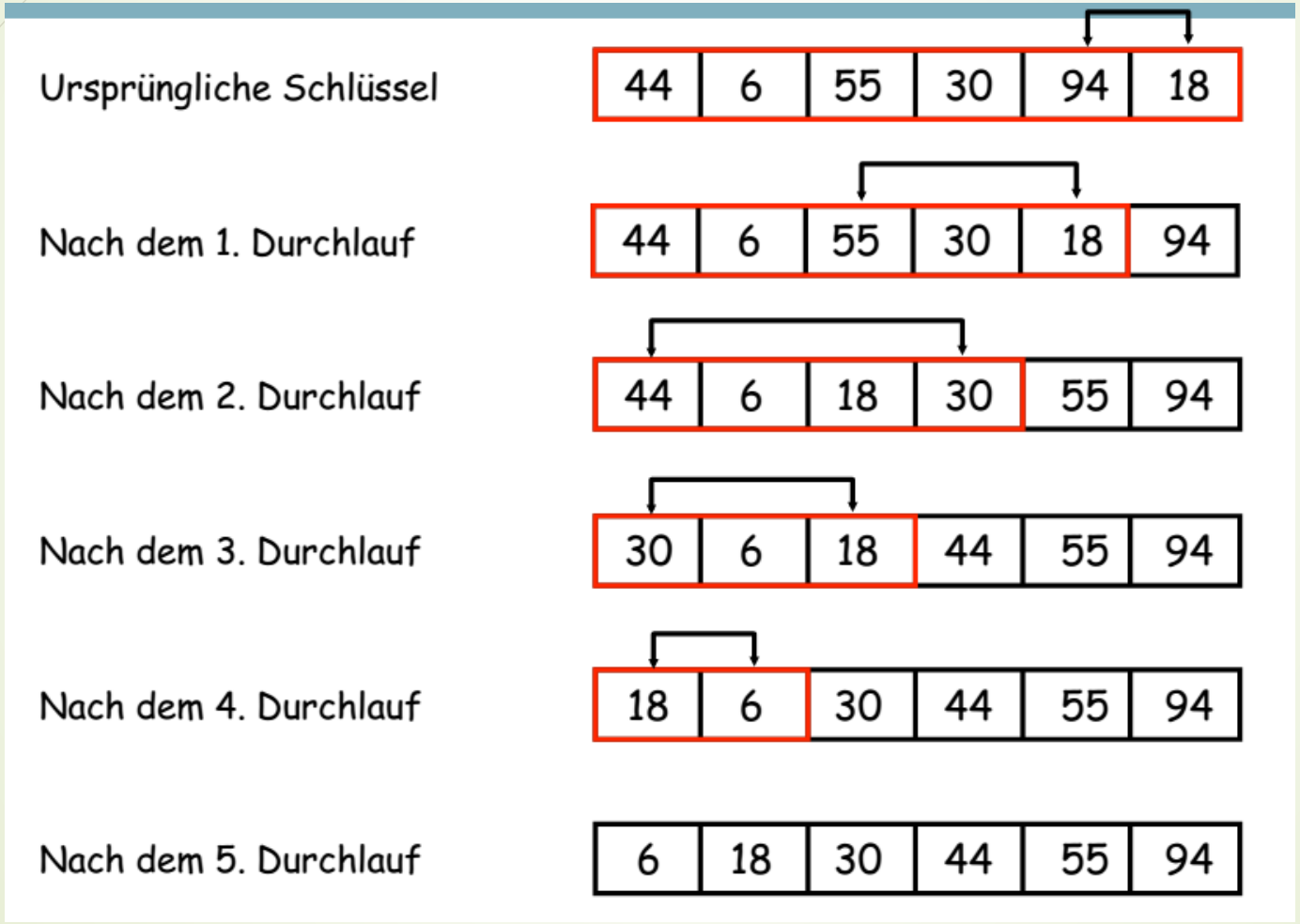
Sortieren durch Selektion – SelectionSort

- Vorgehensweise kann auch beim Sortieren von Spielkarten angewandt werden.
- Sortieren eines Arrays (= Stapel):
  - Suche das größte Element.
  - Füge dieses Element am Ende des Stapels ein.
  - Dies wird in jedem Schritt mit einem jeweils um 1 verkleinerten Bereich des Stapels ausgeführt, solange bis der Stapel die Länge 1 hat. Somit sammeln sich am Ende des Stapels die bereits sortierten Elemente.

SelectionSort: Algorithmus

```
SelectionSort (F)

Eingabe:    zu sortierende Folge F der Länge n

p := n;

while p > 1 do

    g := Index des größten Elementes aus F im Bereich 1 – p;

    Vertausche Werte von F[p] und F[g];

    p := p-1;

od
```

SelectionSort: Beispiel



SelectionSort: Programm 1/ 2

```
static void SelectionSort (int [] array) {
  int marker = array.length - 1;
  while (marker > 0) {
        // Suche größtes Element
    int max = 0;   // ... das ist zunächst das erste Element
    for (int i = 1; i <= marker; i++) // Suche in Restfolge
      if (array [i] > array [max]) // größeres Element gef.
        max = i;
    swap (array, marker, max);
      // Tausche array[marker] mit dem gefundenen Element
    marker--;
  }
}
```

SelectionSort: Programm 2/2

```
static void swap (int [] array, int idx1, int idx2)
// Hilfsmethode zum Vertauschen zweier Feldelemente
{
    int tmp = array[idx1];
    array[idx1] = array[idx2];
    array[idx2] = tmp;
 }
```

SelectionSort: Eigenschaften

- Man beginnt mit dem letzten Element als aktuellem Element und geht bei jedem Durchlauf um 1 nach links. Bei jedem Durchlauf wird das aktuelle Element mit dem größten Element links davon vertauscht.
- Die Anzahl der Vergleiche ist also unabhängig vom Aussehen der Eingabe-reihenfolge, egal ob die Folge schon sortiert ist oder in umgekehrter Reihenfolge sortiert ist. Für die Anzahl der Vergleiche ergibt sich:

  $$(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = n\,(n-1)/2 \approx n^2/2$$

- Vertauscht wird dann, wenn ein größeres Element gefunden wird. Ist die Folge schon sortiert (günstigster Fall), dann gibt es keine Vertauschungen, steht das kleinste Element hinten und ist der Rest sortiert (schlechtester Fall) ist sie = n - 1. Im Mittel deshalb = n/2.
- Das ergibt in der Summe für Vergleiche und Vertauschungen im Mittel:

  $$\approx n^2/2 + n/2$$

- Das Verfahren ist instabil, wegen der Vertauschungen, dadurch werden relative Reihenfolgen nicht beibehalten.

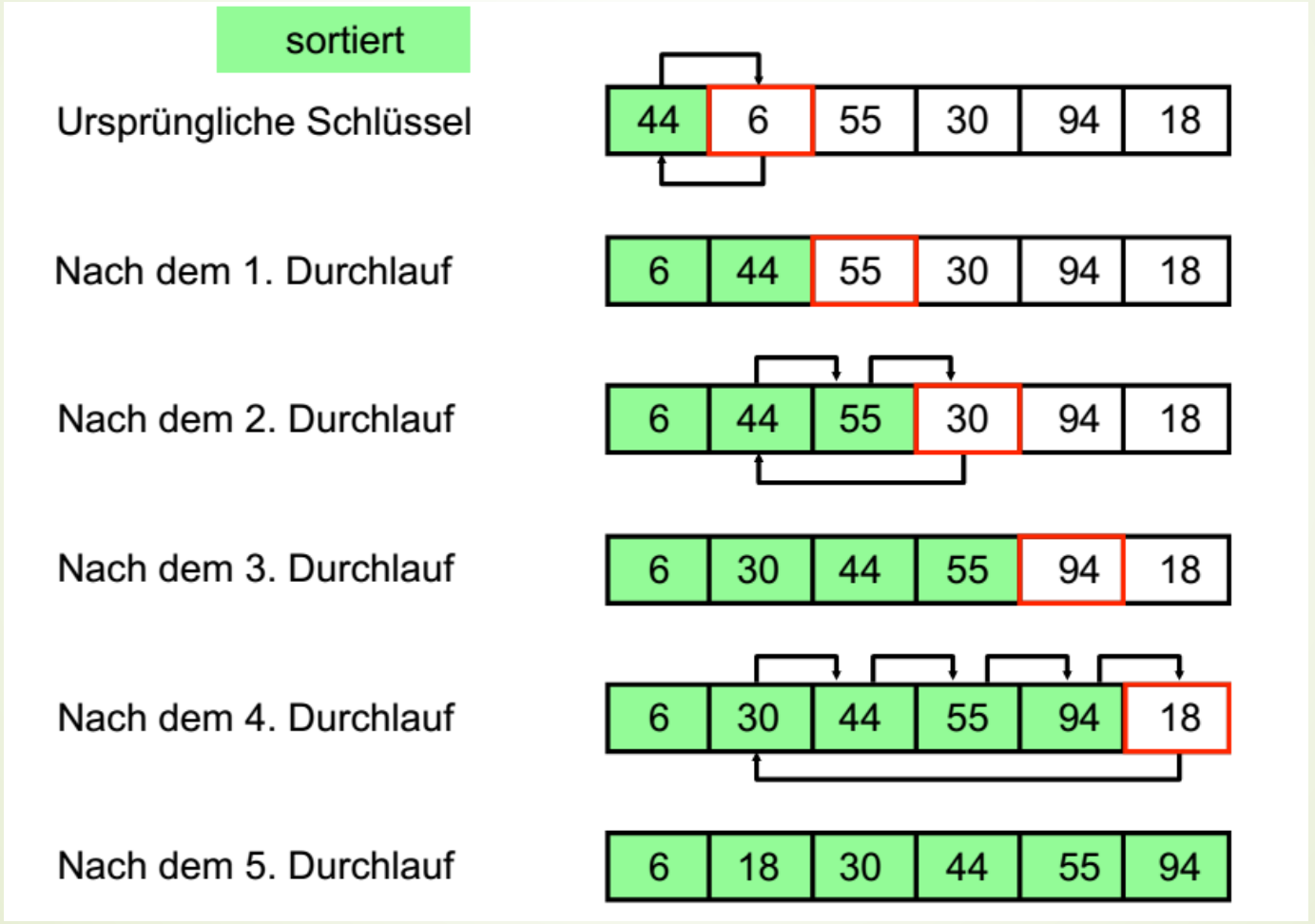
Sortieren durch Einfügen – InsertionSort

- Typische menschliche Vorgehensweise – etwa beim Sortieren von Spielkarten:
- 1. Beginne mit der ersten Karte einen neuen Stapel.
- 2. Nimm jeweils die nächste Karte des Originalstapels und füge diese an der richtigen Stelle in den neuen Stapel ein.
- Sortieren eines Arrays bzw. einer Folge (= Stapel):
  - Das erste Element ist bereits sortiert.
  - Der Zielstapel hat bis jetzt die Länge 1.
  - Beginnend ab dem 2. Element wird an der passenden Stelle in den Zielstapel eingefügt.
  - Muss ein Element an einer Stelle dazwischen geschoben werden, dann werden die rechts davon liegenden Elemente jeweils um eine Position nach rechts geschoben.

InsertionSort- Algorithmus

```
InsertionSort (F)
Eingabe:    zu sortierende Folge F der Länge n
for i := 2 to n do   // Element an Position i wird an der passenden Stelle eingefügt
 m := F[i];  // merke einzufügendes Element
 j := i;
 while j > 1 do // laufe von rechts nach links
    if F[j-1] >= m then   // verschiebe F[j-1] eine Position nach rechts
            F [j] := F[j-1];
            j := j-1;
    else
            Verlasse innere Schleife (= while-Schleife) // weil Einfügeposition gefunden
    fi;
 od;  //  while-Schleife
 F [j] :=m; // gemerktes Element an der richtigen Position einfügen
od;  // for-Schleife
```

InsertionSort- Beispiel



InsertionSort- implementierung

```
static void insertionSort (int [] array) {
  for (int i = 1; i < array.length; i++) {
    int j = i;
    int m = array[i]; // Marker-Feld
    while (j > 0 && array [j-1] > m) {
      // Verschiebe alle größeren Elemente nach hinten
      array [j] = array [j-1];
      j--;
    }
    // Setze m auf das freie Feld
    array[j] = m;
  }
}
```

InsertionSort- Eigenschaften(1)

- Man beginnt mit dem 2. Element als aktuellem Element und prüft, wo es links davon eingefügt werden muss. Man fährt dann mit dem 3. Element fort. Solange, bis man beim letzten Element angekommen ist. Muss ein Element eingefügt werden, dann werden alle anderen ab der Einfügeposition bis zur Position des aktuellen Elements um eine Position nach rechts verschoben.
- Die Anzahl der Vergleiche ist abhängig von der Art der Eingabe-reihenfolge: ist die Folge in umgekehrter Reihenfolge sortiert (der **schlechteste Fall** bzgl. der Vergleiche), dann werden alle Elemente miteinander verglichen:

$$1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = n\,(n-1)/2 \approx n^2/2$$

- Sind die Elemente schon sortiert (der **günstigste Fall** bzgl. der Vergleiche), dann ist man nach dem jeweils ersten Vergleich schon fertig:

$$1 + 1 + \ldots + 1 = n - 1$$

- In der Summe:

$$(n-1) + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = n\,(n+1)/2 - 1 \approx n^2/2$$

- Im Mittel: $\approx n^2/4$

InsertionSort- Eigenschaften(2)

- Für die Verschiebungen verhält es sich ganz ähnlich:
- Sind die Elemente schon sortiert (günstigster Fall) , dann ist keine Verschiebung notwendig.
- Sind die Elemente in umgekehrter Reihenfolge (schlechtester Fall) sortiert, dann sind

  $1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = n \cdot (n-1)/2 \approx n^2/2$ Verschiebungen notwendig

- Im Mittel: $\approx n^2/4$.
- Das ergibt in der Summe für Vergleiche und Verschiebungen im Mittel:

  $$\approx 2 \cdot n^2/4$$

- Das Verfahren ist stabil, weil bei den Verschiebungen relative Reihenfolgen nicht verändert werden.

Sortieren durch Vertauschen – BubbleSort

- Man vergleicht paarweise immer 2 benachbarte Elemente miteinander.
- Entsprechen diese Elemente nicht der Sortierreihenfolge, dann vertauscht man sie.
- Elemente, die größer sind als ihr Nachfolger überholen diese und steigen zum Ende der Folge hin.
- Man stellt sich die sortierende Folge vertikal angeordnet vor, dann kann man sich verschieden große, aufsteigende (Luft-) Blasen (bubbles) vorstellen. Wie in einer Flüssigkeit sortieren sich diese alleine, da die größeren Blasen die kleineren überholen.

BubbleSort- 1. Algorithmus

```
BubbleSort (F)
Eingabe:    zu sortierende Folge F der Länge n
for j := 1 to n do
 for i := 1 to n-1 do
  if F[i]  > F[i+1] then
    Vertausche Werte von F[i] und F[i+1];
  fi;
 od;
od;
```

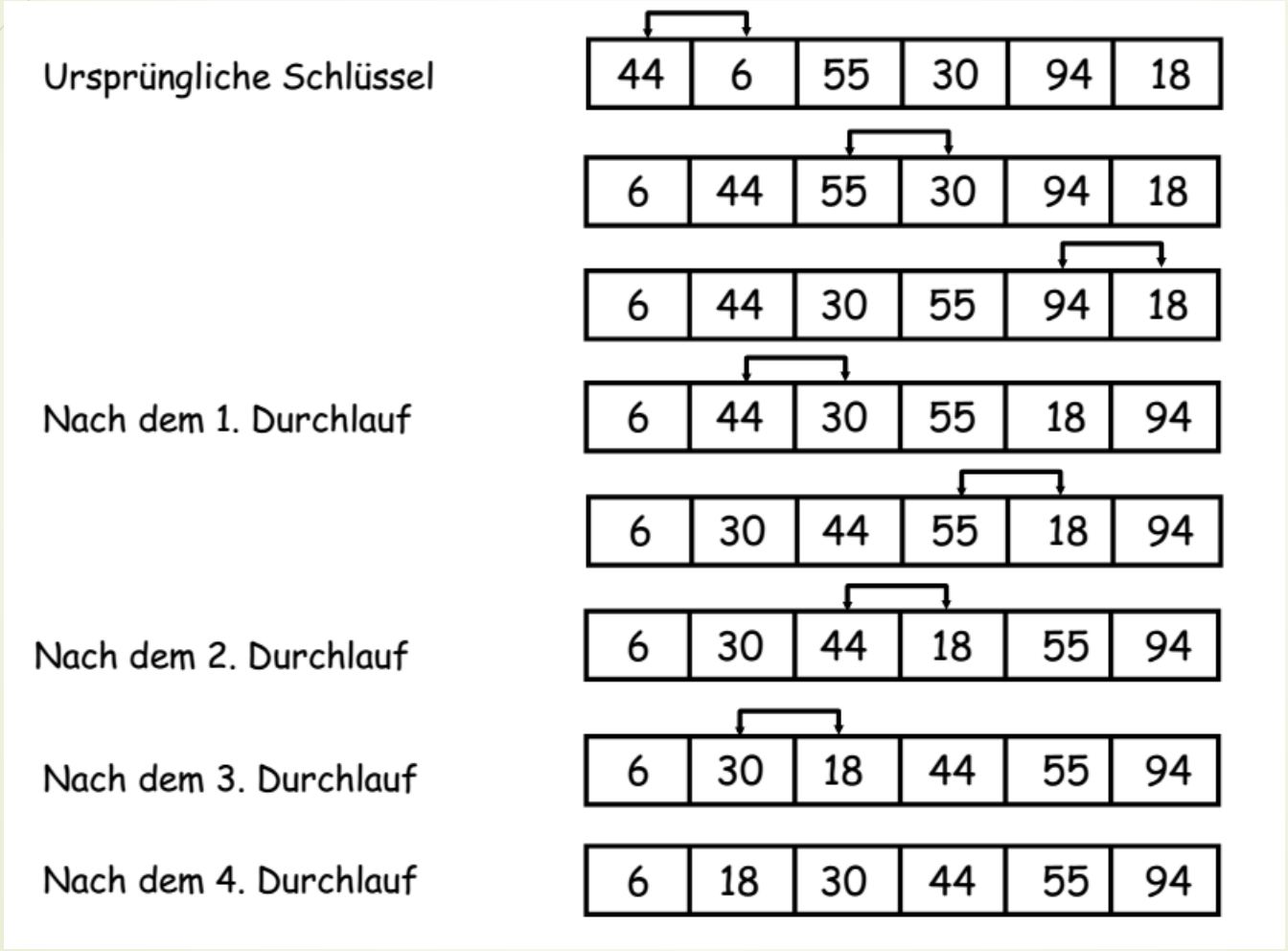
BubbleSort- 2. Algorithmus

```
BubbleSort (F)
Eingabe:    zu sortierende Folge F der Länge n
do
  for i := 1 to n -1 do
    if F[i]  > F[i+1] then
      Vertausche Werte von F[i] und F[i+1];
    fi;
  od
while Vertauschungen statt gefunden haben
```

BubbleSort- 3. Algorithmus

```
BubbleSort (F)
Eingabe:    zu sortierende Folge F der Länge n
for j := 1 to n do
 for i := 1 to n - j do
  if F[i]  > F[i+1] then
    Vertausche Werte von F[i] und F[i+1];
  fi;
 od;
od;
```

BubbleSort- Beispiel



BubbleSort: Programm (zu 2. Algorithmus)

```
static void BubbleSort (int [] array) {
   boolean swapped;
   do {
     swapped = false;
     for (int i = 0; i < array.length - 1; i++) {
       if (array [i] > array [i+1]) {
         swap (array, i, i+1); // tausche Elemente
         swapped = true;
       }
     }
   } while (swapped); // solange noch Vertauschung
 }
```

BubbleSort: Eigenschaften

- Bei jedem Durchlauf werden alle Elemente paarweise miteinander ver-glichen und ggf. miteinander vertauscht. Dabei wird das jeweils größte Element gefunden und durch Vertauschen mit seinem rechten Nachbarn an das Ende der Folge bewegt.
- Die Anzahl der Vergleiche und Vertauschungen ist abhängig von der An-ordnung der Eingabereihenfolge: ist die Folge in umgekehrter Reihenfolge sortiert (der **schlechteste Fall**), dann werden im i-ten Durchlauf n-i Elemente miteinander verglichen und vertauscht:

$$2 \cdot ((n-1) + (n-2) + (n-3) + \ldots + 2 + 1) = n\,(n-1) \approx n^2$$

- Ist die Folge schon sortiert (der **günstigste Fall**), ist nur ein Durchlauf notwendig:

  n-1 Vergleiche und keine Vertauschungen

- In der Summe:

$$(n-1) + 2\,((n-1) + (n-2) + (n-3) + \ldots + 2 + 1) = n\,(n+1) - 1 \approx n^2$$

- Im Mittel: $\approx n^2/2$
- Zwei benachbarte Elemente werden nur dann getauscht, wenn das erste Element größer ist, d.h. das Verfahren ist stabil.

Sortieren durch Mischen – MergeSort

- Externes Verfahren – wenn Dateien nicht in den Hauptspeicher passen.
- Sortieren in 2 Schritten:
  - Die Folge wird in Teile zerlegt, die jeweils in den Hauptspeicher passen und daher getrennt voneinander mit internen Verfahren sortiert werden können. Diese sortierten Teilfolgen werden wieder in Dateien ausgelagert.
  - Anschließend werden die Teilfolgen parallel einlesen und gemischt, indem jeweils das kleinste Element aller Teilfolgen gelesen und die neu Folge (d.h. wieder eine Datei) geschrieben wird.
- Der MergeSort ist in der Praxis eine Mischung aus internem und externem Sortieren. Für kleine Datenmengen kann man den MergeSort als rein internes Verfahren (mit einem Array) realisieren.

MergeSort: Algorithmus (grob)

- 1. Zerlege die Folge F in zwei Hälften F1 und F2 .
- 2. Mische F1 und F2 durch Kombination einzelner Elemente zu geordneten Paaren.
- 3. Bezeichne die gemischte Sequenz mit F und wiederhole die Schritte 1 und 2,
  wobei dieses Mal die geordneten Paare zu geordneten Quadrupeln zusammengefasst werden.
- 4. Wiederhole die voranstehenden Schritte durch Mischen der Quadrupel zu
  Octupeln und fahre damit solange fort, indem jedes Mal die Längen der gemischten Sequenzen verdoppelt werden, bis die ganze Sequenz geordnet ist.

MergeSort: Algorithmus

MergeSort (F) // sogenanntes reines oder direktes Mischen

Eingabe: zu sortierende Folge F der Länge n (n ist 2-er Potenz)

tl := 1; // aktuelle Lauflänge

while tl < n **do**

Schreibe von der Folge F jeweils einen Lauf alternierend auf F_1 und F_2, bis alle Läufe von F nach F_1 und F_2, kopiert sind.

F := leere Folge; // Setze F auf die Ausgangsposition zurück – zum Überschreiben;

for 1. Lauf **to** letzter Lauf **do**

while Lauf der Länge tl in F_1 oder F_2 noch nicht abgearbeitet **do**

Entferne das kleinere Anfangselement aus F_1 bzw. F_2;

Füge dieses Element an F an;

od

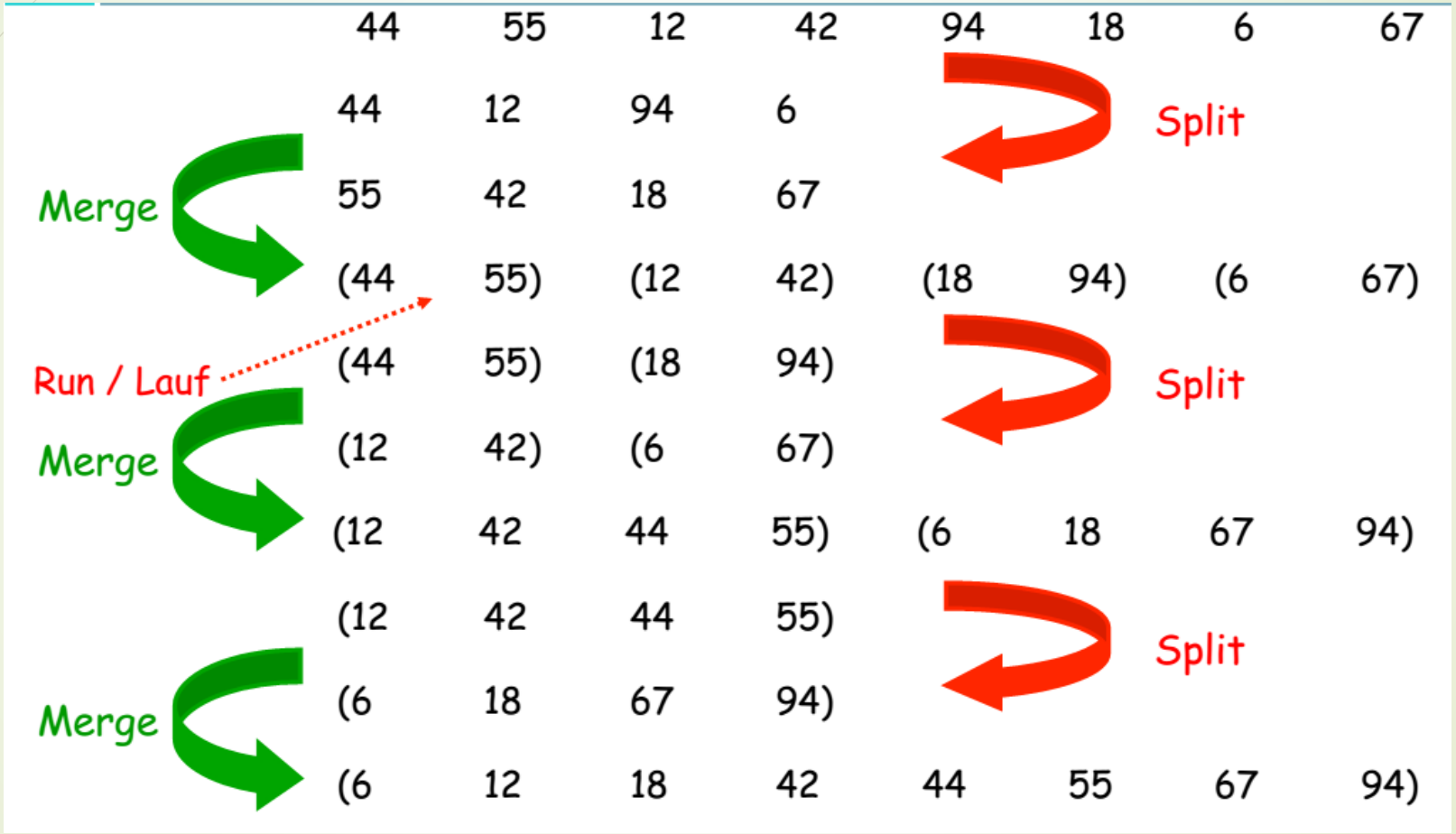
Füge den verbliebenen nichtleeren Rest des aktuellen Laufs von F_1 oder F_2 an F an;

od

tl := tl * 2; // verdopple Lauflänge für nächsten Durchlauf

od

MergeSort: Beispiel



MergeSort-Eigenschaften(1)

- Da eine Folge der Länge N in 2 Teilfolgen zerlegt wird und dann wieder zusammengefügt wird, nennt man das Verfahren auch 2-Wege-Mergesort.
- Beginnt man mit Teilfolgen der Länge 1 und fährt solange in 2-er-Potenzen fort, bis man Teilfolgen der Länge N/2 zu einer (Ergebnis-) Folge der Länge N zusammengefügt hat (wie im Algorithmus angegeben), dann nennt man das Verfahren reines 2-Wege-Mergesort (straight 2-way merge sort).
- Der reine 2-Wege Mergesort ist ineffizient, da er immer mit einelementigen Teilfolgen beginnt und nicht eine Vorsortierung nutzt und auf möglichst langen vorsortierten Teilfolgen aufbaut. Ein Verfahren, das dies umsetzt, nennt man natürlichen 2-Wege Mergesort.
- Weitere Mergesort-Varianten sind:
  ¤ Mehr-Wege-Mergesort
  ¤ Mehrphasen-Mergesort

MergeSort-Eigenschaften(2)

- Die Anzahl der Durchläufe ist (beim reinen MergeSort) konstant. Die Anzahl der Operationen pro Durchlauf auch.
- Beim reinen MergeSort wird die Lauflänge (beginnend mit 1) bei jedem Durchlauf verdoppelt. D.h. dass man bei n Elementen (n ist eine 2-er Potenz!) $\log_2 n$ Durchläufe erhält.
- Bei jedem Durchlauf werden alle n Elemente des Ausgangsbands gleichmäßig auf die beiden Hilfsbänder verteilt: $2 \cdot n / 2 = n$ Kopieroperationen. Da es log n Durchläufe gibt, ist die Gesamtzahl der Kopieroperationen beim Zerlegen = $n \cdot \log_2 n$.
- Beim Zusammenfügen der Läufe auf den Hilfsbändern werden pro Durchlauf (Lauflänge-1) · n / Lauflänge Vergleiche gemacht (beachte: die Lauflänge ist immer eine 2-er Potenz), um über die Reihenfolge auf dem Ausgangsband zu entscheiden. Für die Gesamtzahl SV der Vergleiche ergibt sich demnach:

$$SV = (1 \cdot n / 2) + (3 \cdot n / 4) + (7 \cdot n / 8) + \ldots + ((2^{\log n} - 1) \cdot n / 2^{\log n})$$

$$= (1 \cdot n / 2) + (3 \cdot n / 4) + (7 \cdot n / 8) + \ldots + ((n-1) \cdot n / n)$$

$$= (1/2 \cdot n) + (3/4 \cdot n) + (7/8 \cdot n) + \ldots + (n-1)$$

MergeSort-Eigenschaften(3)

- Jeder einzelne Summand ist kleiner als n, die Anzahl der Summanden ist log n. Deshalb gilt die für die Summe (SV) folgende Abschätzung:

  $SV < n \cdot \log n$

- Beim Zurückschreiben auf das Ausgangsband gibt es genau n Kopieroperationen. Da es log n Durchläufe gibt, ist die Gesamtzahl der Kopieroperationen beim Zurückschreiben = $n \cdot \log n$.

- Insgesamt gilt für den Gesamtaufwand (GA):

  GA <= Anzahl Durchläufe · (Kopieroperationen beim Aufteilen + Vergleichsoperationen + Kopieroperationen beim Mischen)

  $\approx \log n \cdot (n + n + n)$ $\boxed{\approx n \cdot \log n}$

- Die relative Ordnung der Elemente bleibt erhalten, da immer nur verschoben und nicht getauscht wird. Das Verfahren ist also stabil.

- Der natürliche MergeSort hat auf jeden Fall ein besseres Laufzeitver-halten, da er höchstens genauso viele Läufe hat wie der reine MergeSort.