Programmiertechnik II

Ralf Herbrich

Sortieren: *Merge Sort*

Überblick

1. *Merge Sort*
2. *Bottom-Up Merge Sort*
3. Komplexität
4. Stabilität

Merge Sort und *Quick Sort*

Algorithmus	Best Case	Average Case	Worst Case
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Bubble Sort	n	n^2	n^2
Shell Sort (3x + 1)	$n \cdot \log_2(n)$	?	$n^{1.5}$
Merge Sort	$1/2 \cdot n \cdot \log_2(n)$	$n \cdot \log_2(n)$	$n \cdot \log_2(n)$
Quick Sort	$n \cdot \log_2(n)$	$2n \cdot \ln(n)$	$1/2 \cdot n^2$

- *Merge Sort* und *Quick Sort* sind kritische Komponenten in heutiger digitaler Infrastruktur
  - Praktisch die am meisten benutzten Sortierverfahren
  - *Quick Sort* als einer der Top-10 Algorithmen aller Zeiten ausgezeichnet

Überblick

1. ***Merge Sort***
2. *Bottom-Up Merge Sort*
3. Komplexität
4. Stabilität

Merge Sort

- **Grundidee**: Array in zwei Teile zerlegen, sortieren und Teile verschmelzen (*merge*)
  1. Zerlege ein gegebenes Array in zwei (fast) gleich große, nicht-überlappende Teilarrays
  2. Sortierte beide Teilarrays rekursiv
  3. Verschmelze (*merge*) beiden sortierten Teilarrays wieder

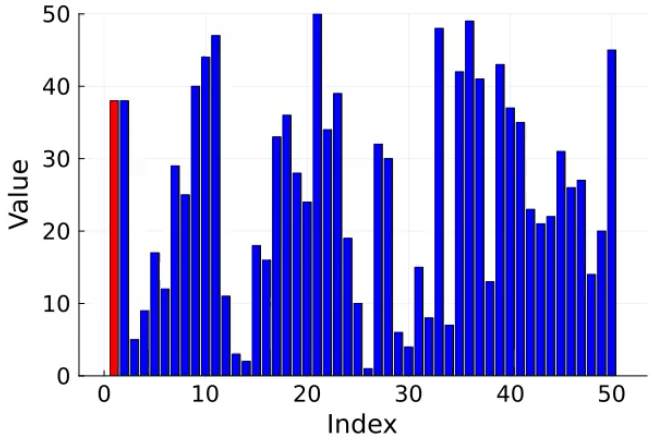
Eingabe	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Erste Hälfte Sortieren	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Zweite Hälfte Sortieren	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
Beide Hälften verschmelzen	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

John von Neumann
(1903 – 1957)

Merge Sort Algorithmus

```cpp
// Implements the recursive merge sort
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo) return;

    auto mid = lo + (hi - lo) / 2;
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
    return;
}
```

```cpp
// Implements merge
template <typename Value>
void merge(Value* a, Value* aux, const int lo, const int mid, const int hi) {
    for (auto k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    auto i = lo, j = mid+1;
    for (auto 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, i)) { a[k] = aux[j++]; }
        else                      { a[k] = aux[i++]; }
    }
    return;
}
```



Kopieren des Arrays

Verschmelzen durch paarweisen Vergleich des aktuellen Elements in der unteren und oberen Hälfte

Merge Sort: Ausführung

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

```
// Implements the recursive merge sort
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo) return;

    auto mid = lo + (hi - lo) / 2;
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
    return;
}
```

Merge Sort in der Praxis

1. ***Insertion Sort*** für kleine Arrays benutzen
   - *Merge Sort* hat zu viel Kopierkosten für kleine Arrays
   - Typischer *cutoff* bei Array der Länge 10

```
// Implements the recursive merge sort with optimizations
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo + CUTOFF - 1) insertion_sort(a, lo, hi);

    auto mid = lo + (hi - lo) / 2;
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    if (!less(a, mid+1, mid)) return;
    merge(a, aux, lo, mid, hi);
    return;
}
```

2. **Kein Verschmelzen** wenn die Arrays schon sortiert sind
   - Überprüfe wenn größtes Element von unterem Array kleiner ist als kleinstes Element von oberem Array
   - Hilft bei teilweise sortierten Arrays

A B C D E F G **H** | **M** N O P Q R S T

A B C D E F G H M N O P Q R S T

Merge Sort in der Praxis II

3. **Kein Kopieren** zum temporären Array notwendig indem die Rolle der beiden Arrays in der Rekursion vertauscht wird

Merge Sort mit Kopieren

```
// Implements merge
template <typename Value>
void merge(Value* a, Value* aux, const int lo, const int mid, const int hi) {
    for (auto k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    auto i = lo, j = mid+1;
    for (auto 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, i)) { a[k] = aux[j++]; }
        else                      { a[k] = aux[i++]; }
    }
    return;
}
```

Merge Sort ohne Kopieren

```
// Implements merge from aux to a array
template <typename Value>
void merge(Value* a, Value* aux, const int lo, const int mid, const int hi) {
    auto i = lo, j = mid+1;
    for (auto 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, i)) { a[k] = aux[j++]; }
        else                      { a[k] = aux[i++]; }
    }
    return;
}
```

Verschmelzt von **`aux[]`** nach **`a[]`**

```
// Implements the recursive merge sort
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo) return;

    auto mid = lo + (hi - lo) / 2;
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
    return;
}
```

```
// Implements the recursive merge sort with optimizations
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo) return;

    auto mid = lo + (hi - lo) / 2;
    merge_sort(aux, a, lo, mid);
    merge_sort(aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
    return;
}
```

Sortiert **`aux[]`** mit **`a[]`** als Speicher

Verschmelzt von **`aux[]`** nach **`a[]`**

Überblick

1. *Merge Sort*
2. ***Bottom-Up Merge Sort***
3. Komplexität
4. Stabilität

Bottom-Up Merge Sort

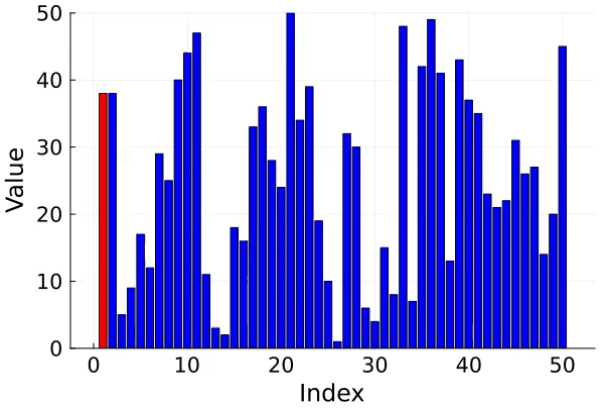
- **Idee**: Verschmelze Teilarrays der Größe 2^i
  - Trivial: $i = 0$
  - Wiederhole für $i = 1,2,3, \ldots$
- Algorithmus ist nicht rekursiv!

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

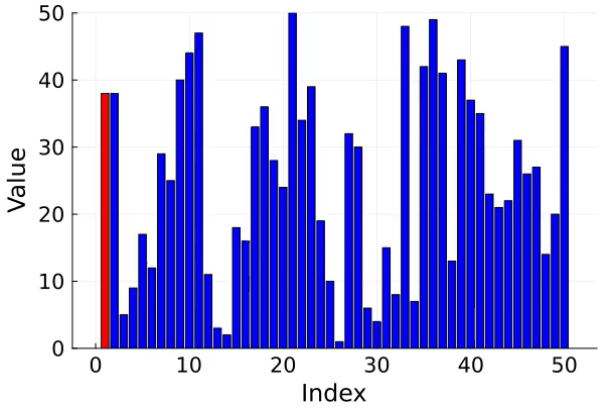
Bottom-Up Merge Sort Algorithmus

```cpp
// Implements the iterative merge sort
template <typename Value>
void bottom_up_merge_sort(Value* a, const int n) {
    Value* aux = new Value[n];
    for (auto sz = 1; sz < n; sz *= 2) {
        for (auto lo = 0; lo < n-sz; lo += sz+sz) {
            merge(a, aux, lo, lo+sz-1, std::min(lo+sz+sz-1, n-1));
        }
    }
    delete[] aux;
    return;
}
```

Merge Sort



Bottom-Up Merge Sort



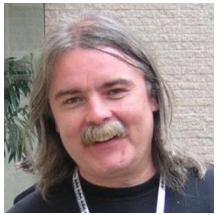
Natural Merge Sort

- **Idee**: Anstatt die Teilarrays in Größe 2^i zu wählen, kann man auch die natürliche Ordnung in der Ursprungslösung ausnutzen!

Eingabe	1	5	10	16	3	4	23	9	13	2	7	8	12	14	17	31
Natürliche Teilarrays	1	5	10	16 |	3	4	23 |	9	13 |	2	7	8	12	14	17	31
Erstes Verschmelzen	1	3	4	5	10	16	23 |	9	13 |	2	7	8	12	14	17	31
Zweites Verschmelzen	1	3	4	5	10	16	23 |	2	7	8	9	13	12	14	17	31
Finales Verschmelzen	1	2	3	4	5	7	8	9	10	12	13	14	16	17	23	31

  - Weniger Verschmelzungen aber Mehraufwand bei Finden der natürlichen Teilarrays
- **Timsort (2002)**: Kombination aus
  - *Natural Merge Sort*
  - Binärem *Insertion Sort* für den ersten Durchlauf
  - Optimierungen beim Zwischenspeicher für das Verschmelzen
- Timsort hat in der Praxis oft lineare Komplexität
  - Wird in Python, Java 7, Rust, GNU Octave, Android, … benutzt

Hasso Plattner Institut
Digital Engineering · Universität Potsdam



Tim Peters

Überblick

1. *Merge Sort*
2. *Bottom-Up Merge Sort*
3. **Komplexität**
4. Stabilität

Merge Sort: Anzahl Vergleiche

- **Satz**: *Merge Sort* benutzt höchstens $n \cdot \log_2(n)$ Vergleiche, um ein Array der Länge n zu sortieren.
  - **Beweisskizze**: Die Anzahl der Vergleiche $C(n)$ treten alle in **`merge`** auf und es gilt

$$C(n) \leq C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$
$$C(1) = 0$$

```
// Implements merge
template <typename Value>
void merge(Value* a, Value* aux, const int lo, const int mid, const int hi) {
    for (auto k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    auto i = lo, j = mid+1;
    for (auto 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, i)) { a[k] = aux[j++]; }
        else                      { a[k] = aux[i++]; }
    }
    return;
}
```

Zur Vereinfachung nehmen wir an, dass n durch zwei teilbar ist. Dann bleibt zu zeigen dass $C(n) = 2 \cdot C(n/2) + n$ durch $C(n) = n \cdot \log_2(n)$ erfüllt ist. Wir benutzen Induktion über k für $n = 2^k$.

Induktionsanfang ($k = 0$): $C(2^0) = C(1) = 1 \cdot \log_2(1) = 0$

Induktionsannahme (k): $C(2^k) = 2^k \cdot \log_2(2^k)$

Induktion ($k+1$):

$$C(2^{k+1}) = 2 \cdot C(2^k) + 2^{k+1}$$
$$= 2 \cdot (2^k \cdot \log_2(2^k)) + 2^{k+1}$$
$$= 2^{k+1} \cdot (\log_2(2^k) + 1)$$
$$= 2^{k+1} \cdot \log_2(2^{k+1})$$

Angenommene Eigenschaft von $C(n)$

Induktionsannahme

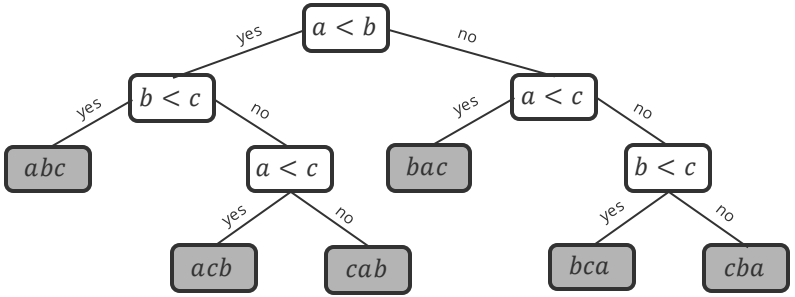
```
// Implements the recursive merge sort
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo) return;

    auto mid = lo + (hi - lo) / 2;
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
    return;
}
```

Merge Sort: Minimale Anzahl Vergleiche

- **Frage**: Kann die Anzahl der Vergleiche im schlechtesten Fall kleiner sein als $n \cdot \log_2(n)$?
  - **Antwort**: Nein!
- **Beweisskizze**: Entscheidungsbaum mit paarweisen Vergleichen, um Sortierung zu bestimmen
  1. Jeder Blattknoten ist eine Sortierung des Arrays
  2. Ein perfekt balancierter Baum der Tiefe n (d.h. n Vergleiche) hat 2^n Blattknoten
  3. Ein Array der Länge n hat $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$ Sortierungen
  4. Daher ist die minimale Tiefe des Baumes

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n)$$

$$\sim n \cdot \log_2(n)$$



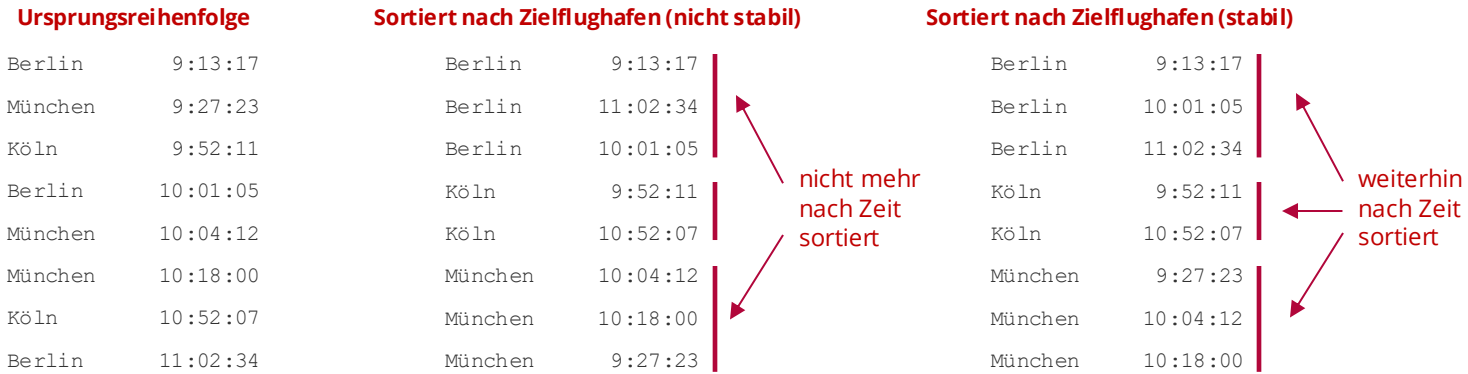
- **Aber**: *Merge Sort* ist nicht optimal im Speicherplatz!

Überblick

1. *Merge Sort*
2. *Bottom-Up Merge Sort*
3. Komplexität
4. **Stabilität**

Stabilität

- Was passiert, wenn ein Array mehrere gleiche Schlüssel enthält?
  - **Beispiel**: Abflugzeiten an einem Flughafen



- **Stabilität**: Ein Sortieralgorithmus ist stabil, wenn die Ursprungsreihenfolge für alle Elemente mit dem gleichen Schlüssel beibehalten werden.

Stabilität: *Insertion Sort*

- **Satz**. *Insertion Sort* ist stabil.
- **Beweis**: Gleiche Schlüssel werden nie vertauscht!

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E_1	X	A	M	P	L	E_2
1	0	O	S	R	T	E_1	X	A	M	P	L	E_2
2	1	O	R	S	T	E_1	X	A	M	P	L	E_2
3	3	O	R	S	T	E_1	X	A	M	P	L	E_2
4	0	E_1	O	R	S	T	X	A	M	P	L	E_2
5	5	E_1	O	R	S	T	X	A	M	P	L	E_2
6	0	A	E_1	O	R	S	T	X	M	P	L	E_2
7	2	A	E_1	M	O	R	S	T	X	P	L	E_2
8	4	A	E_1	M	O	P	R	S	T	X	L	E_2
9	2	A	E_1	L	M	O	P	R	S	T	X	E_2
10	2	A	E_1	E_2	L	M	O	P	R	S	T	X

```cpp
// Implements insertion sort
template <typename Value>
void insertion_sort(Value* a, const int n) {
    for (auto i = 1; i < n; i++) {
        for (auto j = i; j > 0 && less(a, j, j - 1); j--) {
            swap(a, j, j - 1);
        }
    }
    return;
}
```

Stabilität: *Selection Sort*

- **Satz**. *Selection Sort* ist nicht stabil.
- **Beweis**: Durch Vertauschen über weite „Distanzen“ kann die Reihenfolge innerhalb einer Gruppe sich ändern.

i	min	0	1	2
		B_1	B_2	A
0	2	A	B_2	B_1
1	1	A	B_2	B_1

```
// Implements selection sort
template <typename Value>
void selection_sort(Value* a, const int n) {
    for (auto i = 0; i < n; i++) {
        auto min = i;
        for (auto j = i+1; j < n; j++) {
            if (less(a, j, min)) {
                min = j;
            }
        }
        swap(a, i, min);
    }
    return;
}
```

Stabilität: *Shell Sort*

- **Satz**. *Shell Sort* ist nicht stabil.
- **Beweis**: Durch Vertauschen über weite „Distanzen“ kann die Reihenfolge innerhalb einer Gruppe sich ändern.

h	0	1	2	3	4
	B_1	B_2	B_3	B_4	A
4	A	B_2	B_3	B_4	B_1
1	A	B_2	B_3	B_4	B_1

```
// Implements shell sort
template <typename Value>
void shell_sort(Value* a, const int n) {

    // 3x+1 increment sequence:  1, 4, 13, 40, 121, 364, 1093, ...
    int h = 1;
    while (h < n/3) {
        h = 3*h + 1;
    }

    while (h >= 1) {
        // h-sort the array
        for (auto i = h; i < n; i++) {
            for (auto j = i; j >= h && less(a, j, j-h); j -= h) {
                swap(a, j, j-h);
            }
        }
        h /= 3;
    }
    return;
}
```

Stabilität: *Merge Sort*

- **Satz**. *Merge Sort* ist stabil.
- **Beweis**: Es reicht, zu überprüfen dass die **`merge`** Funktion stabil ist

  Die entscheidende Einsicht ist, dass beim Vergleich immer vom linken Teilarray kopiert wird!

```
// Implements merge
template <typename Value>
void merge(Value* a, Value* aux, const int lo, const int mid, const int hi) {
    for (auto k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    auto i = lo, j = mid+1;
    for (auto 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, i)) { a[k] = aux[j++]; }
        else                      { a[k] = aux[i++]; }
    }
    return;
}
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A_1	A_2	A_3	B	D	E	F_1	F_2	A_4	A_5	C	F_3	F_4	F_5	G	H
`merge(a, aux, 0, 7, 15)`	A_1	A_2	A_3	A_4	A_5	B	C	D	E	F_1	F_2	F_3	F_4	F_5	G	H

```
// Implements the recursive merge sort
template <typename Value>
void merge_sort(Value* a, Value* aux, const int lo, const int hi) {
    if (hi <= lo) return;

    auto mid = lo + (hi - lo) / 2;
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
    return;
}
```

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!