Programmiertechnik++

Sortieren – Mergesort

Ralf Herbrich, Christoph Lippert
slides credit: Felix Naumann

Überblick

- Elementare Sortierverfahren (letzter Montag)
  - Selectionsort
  - Bubblesort
  - Insertionsort
  - Shellsort
- **Mergesort** (heute)
- Quicksort (Donnerstag)
- Countingsort

Mergesort Prinzip

Idee:

1. Teile die zu sortierenden Array a in zwei Hälften
   - L = a[0:mid-1]
   - R = a[mid:n-1]
2. Sortiere L und R jeweils mit mergesort
3. Führe die Ergebnisse in einen sortierten Array zusammen (merge)

Entwurfsmuster: Divide-and-Conquer

```
template <typename T>
T* mergesort(T* a, int n) {
    if(n < 2) return a; // Base case

    int mid = n / 2;

    // Split array into two halves
    T* L = new T[mid];
    T* R = new T[n - mid];

    for(int i = 0; i < mid; i++)
        L[i] = a[i];
    for(int i = mid; i < n; i++)
        R[i - mid] = a[i];

    // Recursively sort two halves
    L = mergesort(L, mid);
    R = mergesort(R, n - mid);

    // Merge sorted halves
    T* result = merge(L, mid, R, n - mid);

    // Cleanup
    delete[] L;
    delete[] R;

    return result;
}
```

Mergesort
Korrektheit

Induktionsannahme:

- Arrays der Länge $n/2$ können sortiert werden.

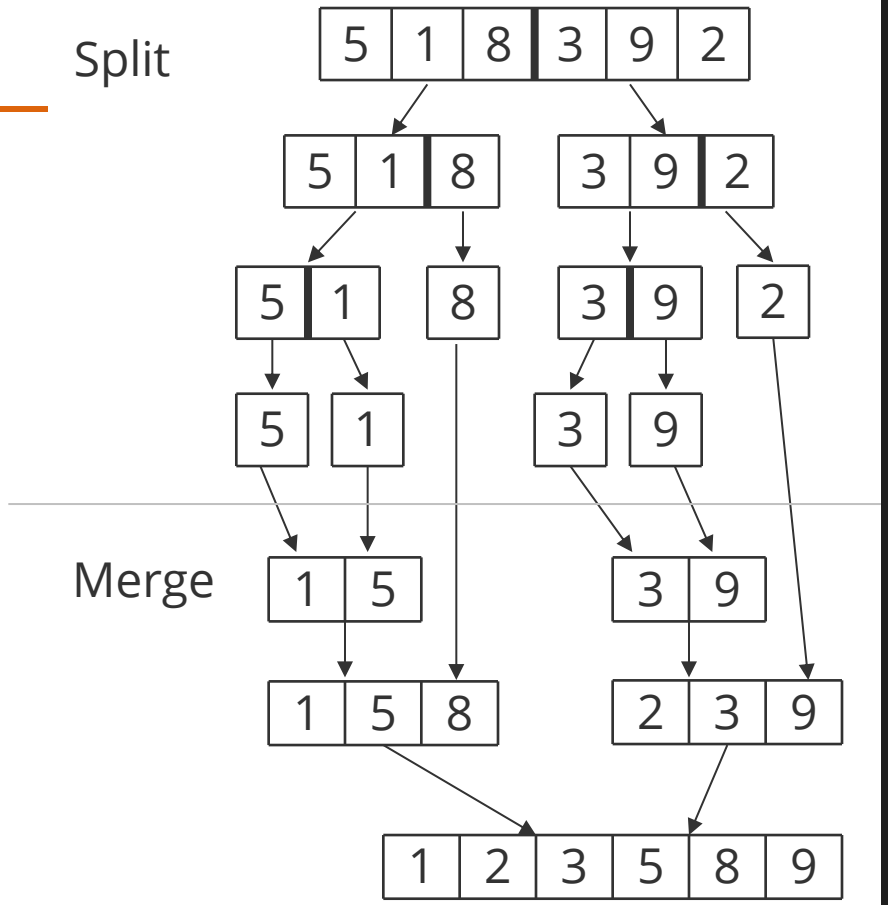
Induktionsanfang: $n = 1$: trivial

Induktionsschritt: $n/2 \rightsquigarrow n$:

- Teile das (Teil-)Array a der Länge n in zwei gleich große Hälften L und R
- Sortiere beide Teilarrays mit $n/2$ Elementen nach Induktionsannahme
- Füge die beiden sortierten Teilarrays zu einem Array zusammen (merge), indem jeweils das kleinste Element der beiden Teilarrays entfernt und in den zusammengefügten Array aufgenommen wird (erfordert n Schritte)

```
template <typename T>
T* mergesort(T* a, int n) {
    if(n < 2) return a; // Base case

    int mid = n / 2;

    // Split array into two halves
    T* L = new T[mid];
    T* R = new T[n - mid];

    for(int i = 0; i < mid; i++)
        L[i] = a[i];
    for(int i = mid; i < n; i++)
        R[i - mid] = a[i];

    // Recursively sort two halves
    L = mergesort(L, mid);
    R = mergesort(R, n - mid);

    // Merge sorted halves
    T* result = merge(L, mid, R, n - mid);

    // Cleanup
    delete[] L;
    delete[] R;

    return result;
}
```

Mergesort Beispiel



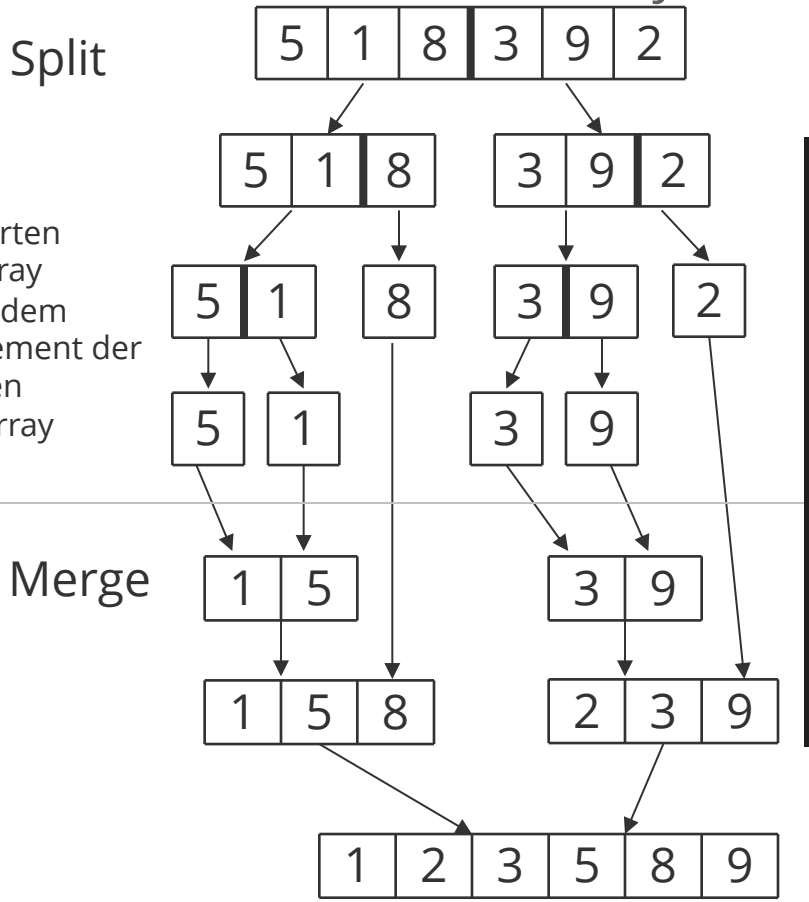

```cpp
template <typename T>
T* mergesort(T* a, int n) {
    if(n < 2) return a; // Base case

    int mid = n / 2;

    // Split array into two halves
    T* L = new T[mid];
    T* R = new T[n - mid];

    for(int i = 0; i < mid; i++)
        L[i] = a[i];
    for(int i = mid; i < n; i++)
        R[i - mid] = a[i];

    // Recursively sort two halves
    L = mergesort(L, mid);
    R = mergesort(R, n - mid);

    // Merge sorted halves
    T* result = merge(L, mid, R, n - mid);

    // Cleanup
    delete[] L;
    delete[] R;

    return result;
}
```

Algorithmus: Zusammenführen von Arrays Merge

merge(L, nL, R, nR):

Füge die beiden sortierten Teilarrays zu einem Array zusammen (merge), indem jeweils das kleinste Element der beiden Teilarrays in den zusammengefügten Array aufgenommen wird

(*n=nL+nR* Schritte)




```
template <typename T>
T* merge(T* L, int leftCount, T* R, int rightCount) {
    T* result = new T[leftCount + rightCount];

    int leftIndex = 0, rightIndex = 0, resultIndex = 0;

    // Merge arrays
    while(leftIndex < leftCount && rightIndex < rightCount) {
        if(L[leftIndex] < R[rightIndex])
            result[resultIndex++] = L[leftIndex++];
        else
            result[resultIndex++] = R[rightIndex++];
    }

    // Copy remaining elements
    while(leftIndex < leftCount)
        result[resultIndex++] = L[leftIndex++];
    while(rightIndex < rightCount)
        result[resultIndex++] = R[rightIndex++];

    return result;
}
```

Mergesort Problem

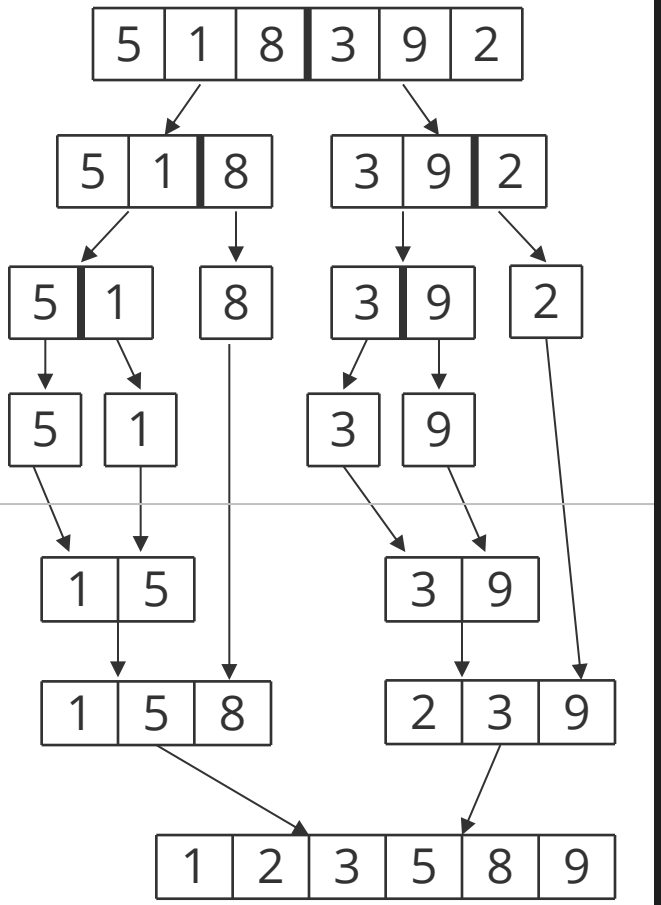
Problem:

- Splitting/merging in jeweils neues Array
  - Wegen Rekursion:

    Platz auf jeder Stufe
  - Platzbedarf: *n* log *n*

- Jetzt: Lösung mit Hilfsarray

  „Abstraktes In-Place-Mergen“




```
template <typename T>
T* mergesort(T* a, int n) {
    if(n < 2) return a; // Base case

    int mid = n / 2;

    // Split array into two halves
    T* L = new T[mid];
    T* R = new T[n - mid];

    for(int i = 0; i < mid; i++)
        L[i] = a[i];
    for(int i = mid; i < n; i++)
        R[i - mid] = a[i];

    // Recursively sort two halves
    L = mergesort(L, mid);
    R = mergesort(R, n - mid);

    // Merge sorted halves
    T* result = merge(L, mid, R, n - mid);

    // Cleanup
    delete[] L;
    delete[] R;

    return result;
}
```

Mergesort mit Hilfsarray

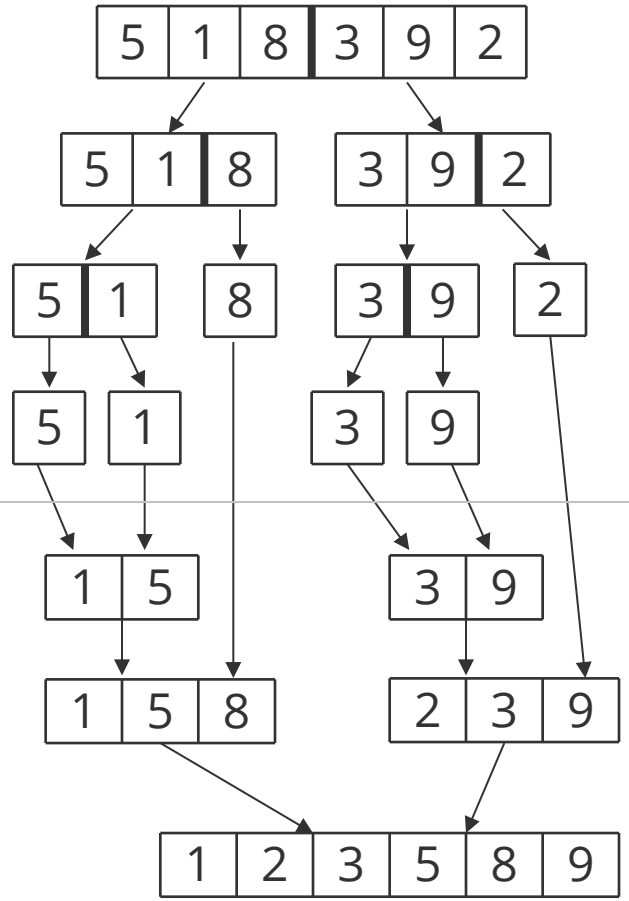
Problem:

- Splitting/merging in jeweils neues Array
  - Wegen Rekursion:

    Platz auf jeder Stufe
  - Platzbedarf: *n* log *n*

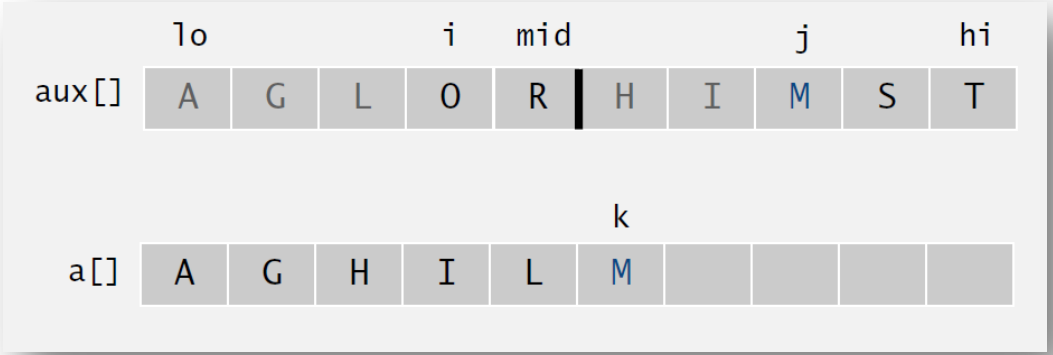
- Jetzt: Lösung mit Hilfsarray

  „Abstraktes In-Place-Mergen“
  - Platzbedarf: 2*n*



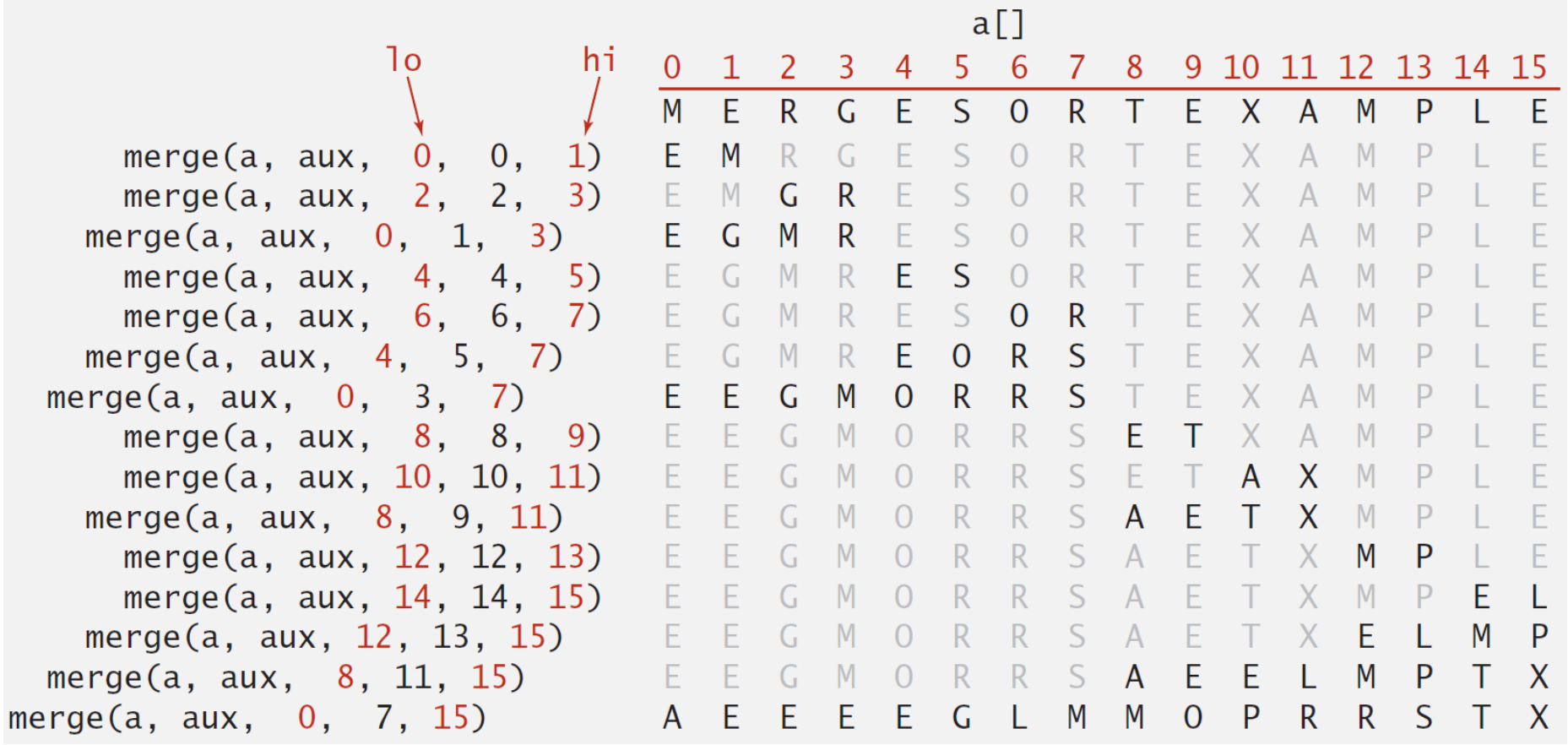

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Mergesort mit Hilfsarray
Beispiel merge




```java
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void sort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Mergesort Trace



Stabilität

- Ist Mergesort stabil?
  - D.h. relative Reihenfolge gleicher Schlüssel bleibt erhalten

Ja	Nein	Weiß nicht	Kommt drauf an
	X		

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void sort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Inputvarianten

- Sortierter Input
- Umgekehrt sortierter Input
- Unsortierter Input
- Viele gleiche Werte
- http://www.sorting-algorithms.com/merge-sort

Besonders günstig	Besonders ungünstig	Egal	Günstig
		Sortierter Input Umgekehrt sortierter Input Unsortierter Input Viele gleiche Werte	

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void sort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Aufwand

- Best case
- Average case
- Worst case

$O(n)$	$O(n \log n)$	$O(n^2)$	$> O(n^2)$
	Best case Average case Worst case		

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void sort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Analyse: MergeSort

- Array der Länge *n*, Anzahl der Vergleiche := V_n

$$V_n = 2V_{n/2} + n \qquad \text{für } n \geq 2 \text{ mit } V_1 = 0$$

2 Teilarrays sortieren

merge der Teilarrays

- Annahme zur Vereinfachung: $n = 2^N$ (*n* ist Zweierpotenz)

$$V_{2^N} = 2V_{2^{N-1}} + 2^N$$

$$\Leftrightarrow \quad \frac{V_{2^N}}{2^N} = \frac{2V_{2^{N-1}} + 2^N}{2^N} = \frac{2V_{2^{N-1}}}{2^N} + 1$$

$$= \frac{V_{2^{N-1}}}{2^{N-1}} + 1$$

von 2^N auf 2^{N-1} nur +1

  - Sukzessive $\frac{V_{2^{N-i}}}{2^{N-i}}$ ersetzen bis *i* = *N*:

$$\Rightarrow \frac{V_{2^N}}{2^N} = N \Leftrightarrow V_{2^N} = 2^N \cdot N$$

Da $n = 2^N$: $V_n = n \log n$

```cpp
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void sort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Analyse: MergeSort

- Laufzeitschätzung
  - Laptop: 10^8 Vergleiche / Sekunde
  - Supercomputer: 10^{12} Vergleiche / Sekunde

	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

- Schlussfolgerung: Gute Algorithmen sind besser als gute Computer!
- Parallele/verteilte Verarbeitung gut möglich (Divide & Conquer)

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void sort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Mergesort Verbesserung: (einfache) Parallelisierung

- rekursive Aufrufe von mergesort in einem eigenen Thread ausgeführt.
- Aufruf von std::async gibt ein std::future-Objekt zurück
- wait stellt sicher, dass der Thread seine Arbeit abgeschlossen hat, bevor wir fortfahren.

Problem:

- Starten von Threads generiert Overhead-Zeit.

mögliche Lösung:

- Schwellenwertgröße für *n* festlegen, bei der das Programm auf sequentielle Sortierung umsteigt.
- insbesondere für sehr große Arrays effizientere Paralleliserung möglich

```cpp
#include <future>

template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;

    auto f1 = std::async(std::launch::async, mergesort<T>, a, aux, lo, mid);
    auto f2 = std::async(std::launch::async, mergesort<T>, a, aux, mid + 1, hi);

    f1.wait();
    f2.wait();

    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

Mergesort Verbesserung: Umstieg auf Insertionsort

```
template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;

    // Use insertion sort for small subarrays
    const int THRESHOLD = 15;
    if (hi - lo + 1 <= THRESHOLD) {
        insertionSort(a, lo, hi);
        return;
    }

    int mid = lo + (hi - lo) / 2;
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Problem:

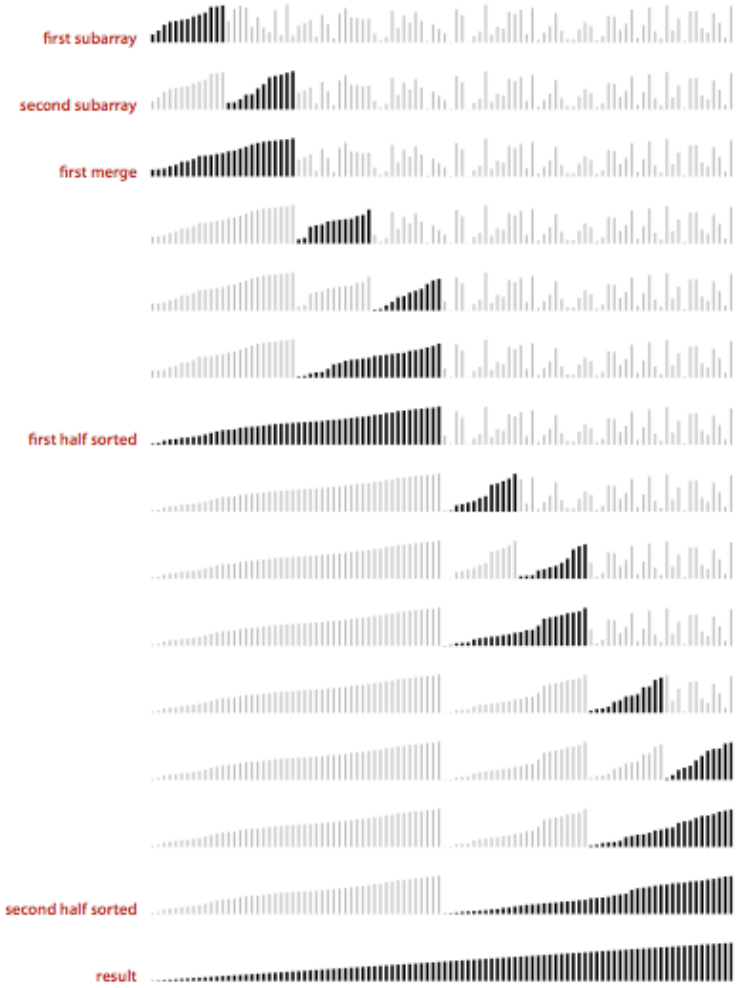
- Tiefe Rekursionsstufen von Mergesort haben hohen Overhead

Lösung:

- Auf Insertionsort oder andere in-place Verfahren ausweichen
  - 10% - 15% Verbesserung

Mergesort Verbesserung: Umstieg auf Insertionsort

```
template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;

    // Use insertion sort for small subarrays
    const int THRESHOLD = 15;
    if (hi - lo + 1 <= THRESHOLD) {
        insertionSort(a, lo, hi);
        return;
    }

    int mid = lo + (hi - lo) / 2;
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```



Visual trace of top-down mergesort for with cutoff for small subarrays

Mergesort Verbesserung: Abbruch bei sortiertem Array

- Prüfen ob bereits sortiert
  - Ist größtes Element links ≤ kleinstes Element rechts?

- Es werden immer noch alle rekursiven Aufrufe getätigt
  - Aber lineare Zeit für jedes schon sortierte Teilarray
    - kein Aufruf von merge für sortierte Arrays
    - => Best case: *O(n)*

```
template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;

    int mid = lo + (hi - lo) / 2;

    // Recurse on the two halves
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);

    // Skip merge if already in order
    if (!(a[mid] > a[mid + 1])) return;

    // Merge results
    merge(a, aux, lo, mid, hi);
}
```

Mergesort Verbesserung
Kopieren in Hilfsarray vermeiden

Idee:

Abwechselnd Hilfsarray und Originalarray verwenden

- Am Ende wird a sortiert sein, da Teilarrays im äußersten Aufruf in a gemerged werden.

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Merge a[lo...mid] with a[mid+1...hi] into aux[lo...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 (a[j] < a[i])       aux[k] = a[j++];
        else                        aux[k] = a[i++];
    }
}

template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    mergesort(aux, a, lo, mid);          // Sort left half into aux
    mergesort(aux, a, mid + 1, hi);      // Sort right half into aux
    merge(a, aux, lo, mid, hi);          // Merge results into a
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    for (int i = 0; i < length; i++) {
        aux[i] = a[i];                   // Copy data to aux array
    }
    mergesort(a, aux, 0, length - 1);
    delete[] aux;
}
```

In-place Merging

- spart Platz (kein aux),
- erhöht Anzahl Operationen (verschieben)

Funktionsweise:

- Zwei Cursor bewegen sich nach rechts
- Falls rechtes Element größer: Erhöhe nur linken Cursor
- Falls rechtes Element klein: Füge Element an richtige Stelle ein
  - Durch Verschieben aller Elemente zwischen den beiden Cursors

```cpp
template <typename T>
void inplaceMerge(T a[], int low, int mid, int high) {
    int start = low;
    int end = mid + 1;

    // Wenn das direkte linke Element kleiner oder gleich
    // ist als das rechte direkte Element, dann sind die Elemente sortiert
    if (a[mid] <= a[end]) {
        return;
    }

    // Zwei Zeiger zum Traversieren der beiden Hälften
    while (start <= mid && end <= high) {

        // Wenn das Element am Anfang der linken Seite kleiner
        // ist als das am Anfang der rechten Seite, inkrementiere den Zeiger
        if (a[start] <= a[end]) {
            start++;
        } else {
            T value = a[end];
            int index = end;

            // Verschieben aller Elemente um eins nach rechts, bis das
            // rechte Element an der richtigen Stelle ist
            while (index != start) {
                a[index] = a[index - 1];
                index--;
            }
            a[start] = value;

            // Update der Indizes für weiteres Durchgehen
            start++;
            mid++;
            end++;
        }
    }
}
```

Iterative Mergesort Variante: Bottom-up Mergesort

	a[i]															
	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
sz = 1																
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																
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
sz = 4																
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
sz = 8																
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

```
template <typename T>
void merge(T* a, T* aux, int lo, int mid, int hi) {
    // Copy data to aux array
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // Merge back to a[]
    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 (aux[j] < aux[i])   a[k] = aux[j++];
        else                        a[k] = aux[i++];
    }
}

template <typename T>
void mergesort(T* a, int length) {
    T* aux = new T[length];
    for (int sz = 1; sz < length; sz = sz+sz) { // sz: subarray size
        for (int lo = 0; lo < length - sz; lo += sz + sz) {
            // Merge subarrays a[lo..lo+sz-1] & a[lo+sz..lo+sz+sz-1]
            int mid = lo + sz - 1;
            int hi = std::min(lo + sz + sz - 1, length - 1);
            merge(a, aux, lo, mid, hi);
        }
    }
    delete[] aux;
}
```

Mergesort Variante: „Natural Mergesort“

- Prüfe auf sortierte Teilarrays („runs“) in einem ersten Durchlauf
- merge iterativ (Bottom-up) alle runs, bis keine mehr da sind.
- Benötigt arrays variabler Größe
  - std::vector
  - std::vector<T> aux(a + low, a + high + 1); kopiert den Teilarray
  - T muss „Copy Assignable“ und „Move Assignable“ sein
- Best case O(n)

```
template <typename T>
void merge(T* a, int low, int mid, int high) {
    int i = low, j = mid + 1;
    std::vector<T> aux(a + low, a + high + 1);

    for (int k = low; k <= high; k++) {
        if (i > mid)                a[k] = aux[j++ - low];
        else if (j > high)          a[k] = aux[i++ - low];
        else if (aux[j - low] < aux[i - low])  a[k] = aux[j++ - low];
        else                        a[k] = aux[i++ - low];
    }
}

template <typename T>
void naturalMergesort(T* a, int length) {
    int low = 0, mid, high;

    while (low < length - 1) {
        while (low < length - 1 && a[low] <= a[low + 1]) low++;
        mid = low;

        if (mid < length - 1) {
            while (low < length - 1 && a[low] > a[low + 1]) low++;
            high = low;

            merge(a, low - high, mid, high);
        }

        low++;
    }
}
```

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!