

Algorithmen und Datenstrukturen

A4. Sortieren II: Mergesort

Marcel Lüthi and Gabriele Röger

Universität Basel

20. Februar 2020

Algorithmen und Datenstrukturen

20. Februar 2020 — A4. Sortieren II: Mergesort

A4.1 Mergesort

A4.2 Merge-Schritt

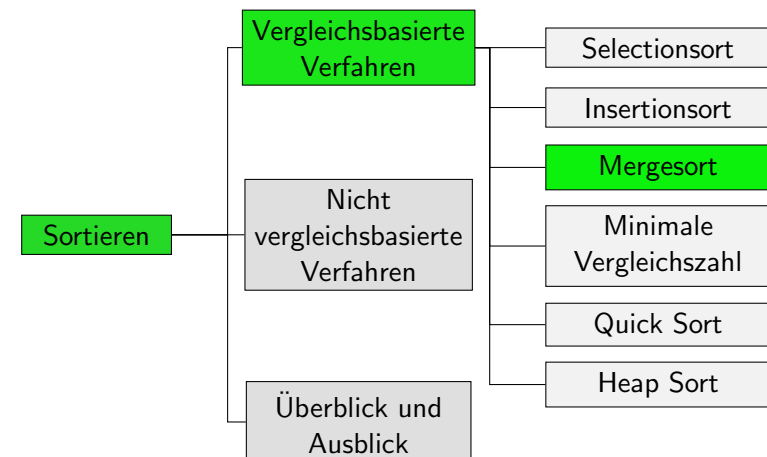
A4.3 Top-Down-Mergesort

A4.4 Bottom-Up-Mergesort

A4.5 Zusammenfassung

A4.1 Mergesort

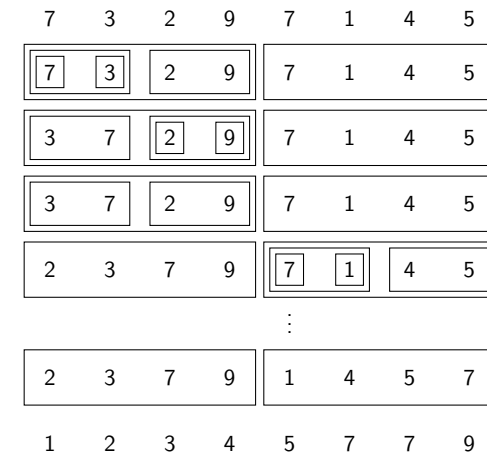
Sortierverfahren



Mergesort: Idee

- ▶ **Beobachtung:** zwei bereits sortierte Sequenzen lassen sich leicht zu einer sortierten Sequenz vereinen.
- ▶ Sequenzen mit einem oder keinem Element sind sortiert.
- ▶ **Idee** für längere Sequenzen:
 - ▶ Teile Eingabesequenz in zwei etwa gleich grosse Teilbereiche
 - ▶ Rekursiver Aufruf für beide Teilmengen
 - ▶ Füge nun sortierte Teilbereiche zusammen.
- ▶ **Teile-und-Herrsche-Ansatz** (divide and conquer)

Mergesort: Illustration



(Detaillierte Animation in Bildschirm-Version der Folien)

A4.2 Merge-Schritt

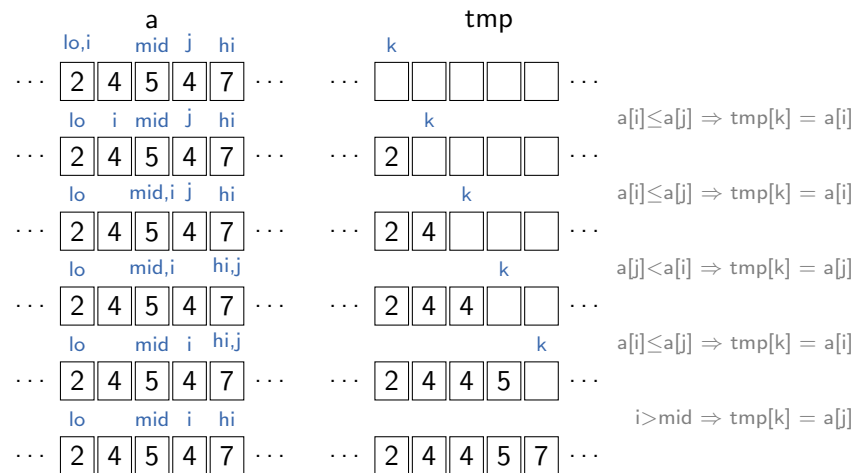
Verbinden der Teillösungen

- ▶ Indizes $lo \leq mid < hi$
- ▶ **Annahme:** array[lo] bis array[mid] und array[mid+1] bis array[hi] sind bereits sortiert
- ▶ **Ziel:** array[lo] bis array[hi] ist sortiert
- ▶ **Idee:** gehe parallel von vorne nach hinten durch beide Teilbereiche und sammle das jeweils kleinere Element auf
- ▶ Verwendet zusätzlichen Speicher für aufgesammelte Werte

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.

Initialisierung: $i := lo$, $j := mid + 1$, $k := lo$



Verbinden der Teillösungen: Algorithmus

```

1 def merge(array, tmp, lo, mid, hi):
2     i = lo
3     j = mid + 1
4     for k in range(lo, hi + 1): # k = lo, ..., hi
5         if j > hi or (i <= mid and array[i] <= array[j]):
6             tmp[k] = array[i]
7             i += 1
8         else:
9             tmp[k] = array[j]
10            j += 1
11     for k in range(lo, hi + 1): # k = lo, ..., hi
12         array[k] = tmp[k]

```

Auch korrekt für $lo = mid = hi$

Jupyter-Notebook



Jupyter-Notebook: merge_sort.ipynb

A4.3 Top-Down-Mergesort

Mergesort: Algorithmus

rekursive Top-Down-Version

```

1 def sort(array):
2     tmp = [0] * len(array) # [0,...,0] with same size as array
3     sort_aux(array, tmp, 0, len(array) - 1)
4
5 def sort_aux(array, tmp, lo, hi):
6     if hi <= lo:
7         return
8     mid = lo + (hi - lo) // 2
9     # //: Division mit Abrunden
10    sort_aux(array, tmp, lo, mid)
11    sort_aux(array, tmp, mid + 1, hi)
12    merge(array, tmp, lo, mid, hi)

```

Mögliche Verbesserungen

- ▶ Auf kurzen Sequenzen ist Insertionsort schneller als Mergesort
→ verwende Insertionsort wenn $hi - lo$ klein
- ▶ Breche Merge-Schritt direkt ab, falls Positionen lo bis hi bereits vollständig sortiert


```

                if array[mid] <= array[mid + 1]:
                    return
            
```
- ▶ Kopieren von tmp-Ergebnis in merge kostet Zeit
→ tausche Rolle von array und tmp bei jedem rekursiven Aufruf

Merge-Schritt: Korrektheit

- ▶ **Invariante:** Am Ende jeder Schleifeniteration ist
 - ▶ $tmp[k] \leq array[m]$ für alle $i \leq m \leq mid$, und
 - ▶ $tmp[k] \leq array[n]$ für alle $j \leq n \leq hi$.
- ▶ tmp wird von vorne nach hinten beschrieben.
- ▶ Nach letzter Schleifeniteration gilt für alle $lo \leq r < s \leq hi$, dass $tmp[r] \leq tmp[s]$ (= Bereich ist sortiert).

Mergesort: Korrektheit

sort_aux:

- ▶ Induktionsbeweis über Bereichslänge $hi - lo$
- ▶ **Basis $hi - lo = -1$:** leerer Bereich ist sortiert.
- ▶ **Basis $hi - lo = 0$:** Bereich mit nur einem Element ist sortiert.
- ▶ **Induktionshypothese:** Mergesort ist korrekt für alle $hi - lo < m$
- ▶ **Induktionsschritt ($m - 1 \rightarrow m$):**
Mergesort macht zwei rekursive Aufrufe mit $hi - lo \leq m/2 + 1$, danach ist die Eingabe jeweils **zwischen lo und mid und zwischen $mid + 1$ und hi sortiert (lt. Ind.-hyp)**.
Wir wissen bereits, dass der Merge-Schritt korrekt ist, also ist am Ende der gesamte Bereich **zwischen lo und hi sortiert**.

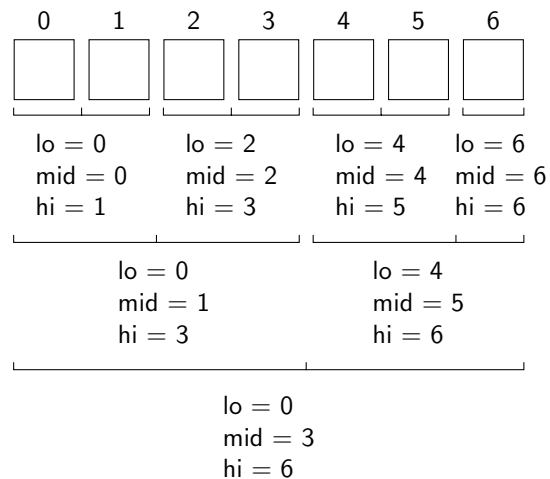
Mergesort: Ruft sort_aux für gesamten Bereich auf, daher ist am Ende die gesamte Eingabe sortiert.

Mergesort: Eigenschaften

- ▶ **nicht in-place**: verwendet zusätzlichen Speicherplatz für tmp und für Aufrufstapel (call stack)
- ▶ **Zeitbedarf**: nicht adaptiv (ausser mit Mergeabbruch-Verbesserung)
genauere Analyse: nächste Woche
- ▶ **stabil**: merge präferiert array[i], wenn array[i] gleich array[j].

A4.4 Bottom-Up-Mergesort

Bottom-Up-Version



Bottom-Up-Mergesort: Algorithm

iterative Bottom-Up-Version

```

1 def sort(array):
2     n = len(array)
3     tmp = [0] * n
4     length = 1
5     while length < n:
6         lo = 0
7         while lo < n - length:
8             mid = lo + length - 1
9             hi = min(lo + 2 * length - 1, n - 1)
10            merge(array, tmp, lo, mid, hi)
11            lo += 2 * length
12            length *= 2

```

A4.5 Zusammenfassung

Zusammenfassung

- ▶ Mergesort ist ein **Teile-und-Herrsche-Verfahren**, das den zu sortierenden Bereich in zwei etwa gleich grosse Bereiche teilt.
- ▶ Der **Merge-Schritt** führt zwei bereits sortierte Teilbereiche zusammen.
- ▶ Mergesort ist **stabil**, arbeitet aber **nicht in-place**.
- ▶ Die **Top-Down-Variante** ist ein **rekursives** Verfahren.
- ▶ Die **Bottom-Up-Variante** ist ein **iteratives** Verfahren.