

# Algorithms and Data Structures

## A4. Sorting II: Merge Sort

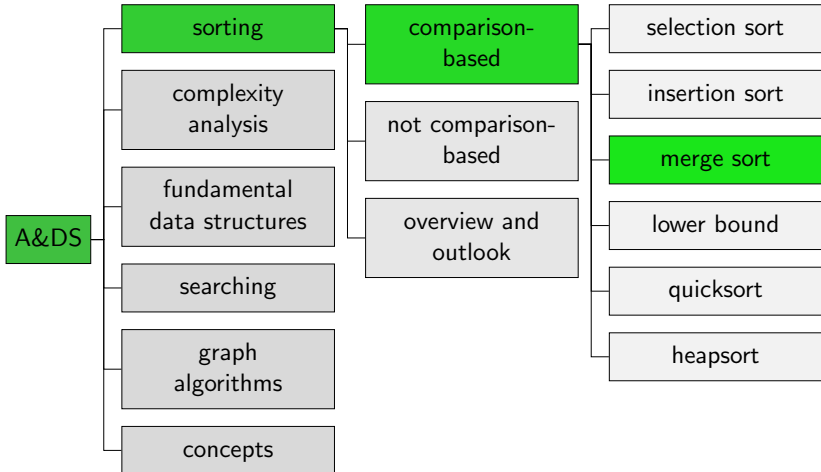
Gabriele Röger

University of Basel

February 29/March 6, 2024

# Merge Sort

# Content of the Course



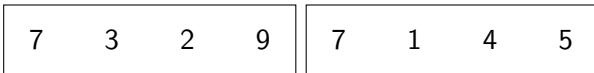
# Merge Sort: Idea

- **Observation:** two sorted sequences can easily be combined to a single sorted sequence.
- Empty sequences or sequences with a single element are sorted.
- **Idea** for longer sequences:
  - divide the input sequence into two roughly equally-sized ranges
  - recursive call for each of the two ranges
  - merge now sorted ranges into one
- **divide-and-conquer approach**

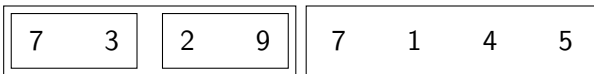
# Merge Sort: Illustration

7 3 2 9 7 1 4 5

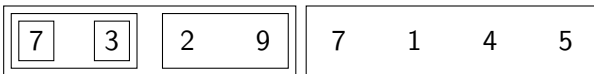
# Merge Sort: Illustration



# Merge Sort: Illustration

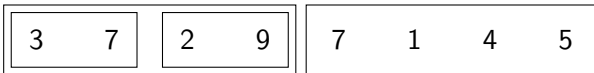


# Merge Sort: Illustration

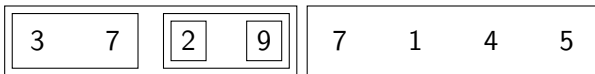




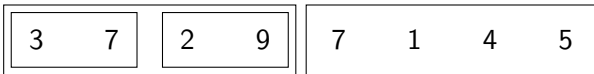
# Merge Sort: Illustration



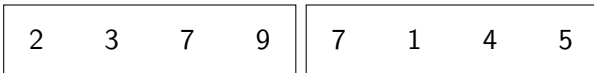
# Merge Sort: Illustration



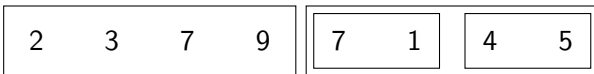
# Merge Sort: Illustration



# Merge Sort: Illustration



# Merge Sort: Illustration



# Merge Sort: Illustration



# Merge Sort: Illustration



# Merge Sort: Illustration

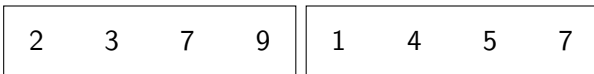




# Merge Sort: Illustration



# Merge Sort: Illustration



# Merge Sort: Illustration

1    2    3    4    5    7    7    9

# Merge Step

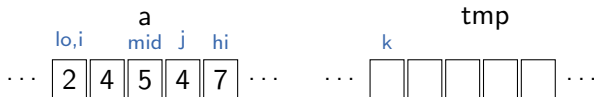
## Merging the Sorted Ranges

- indices  $lo \leq mid < hi$
- **prerequisite:** array[lo] to array[mid] and array[mid+1] to array[hi] already sorted
- **aim:** array[lo] to array[hi] sorted
- **idea:** process both ranges in parallel from front to end and collect the smaller element
- use additional storage for the collected entries

## Merge Step: Example

Array tmp has same size as input array.

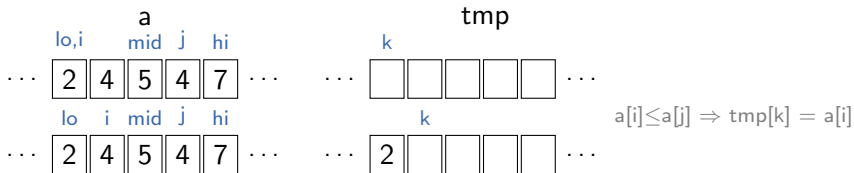
initialize:  $i := lo$ ,  $j := mid + 1$ ,  $k := lo$



## Merge Step: Example

Array tmp has same size as input array.

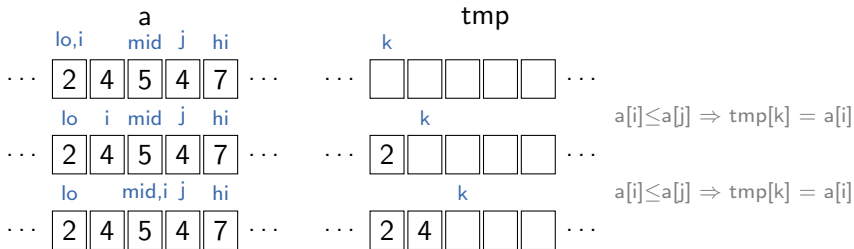
initialize:  $i := lo$ ,  $j := mid + 1$ ,  $k := lo$



## Merge Step: Example

Array tmp has same size as input array.

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

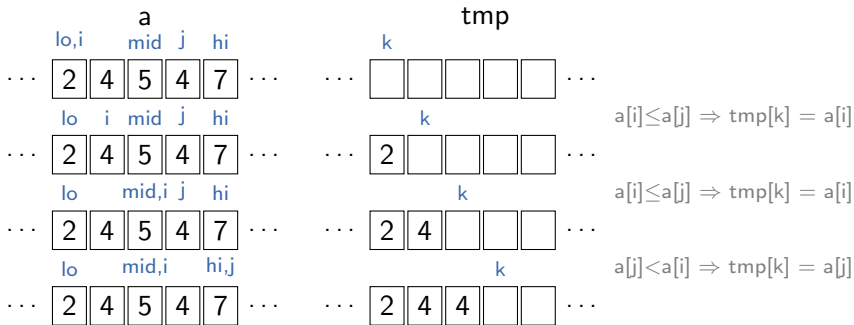




## Merge Step: Example

Array tmp has same size as input array.

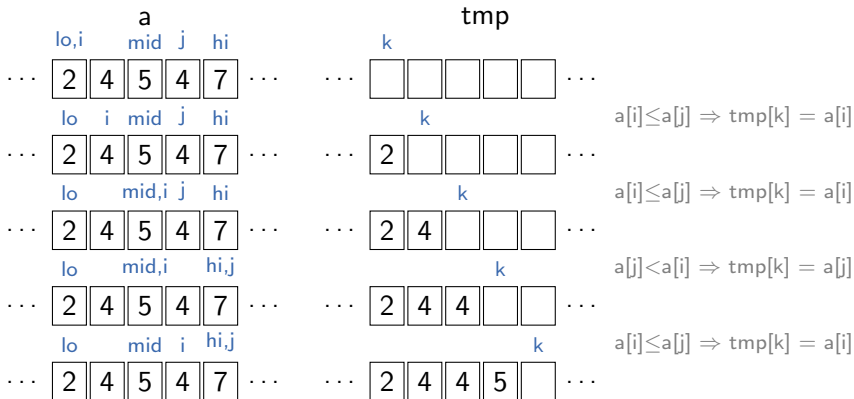
initialize:  $i := lo$ ,  $j := mid + 1$ ,  $k := lo$



## Merge Step: Example

Array tmp has same size as input array.

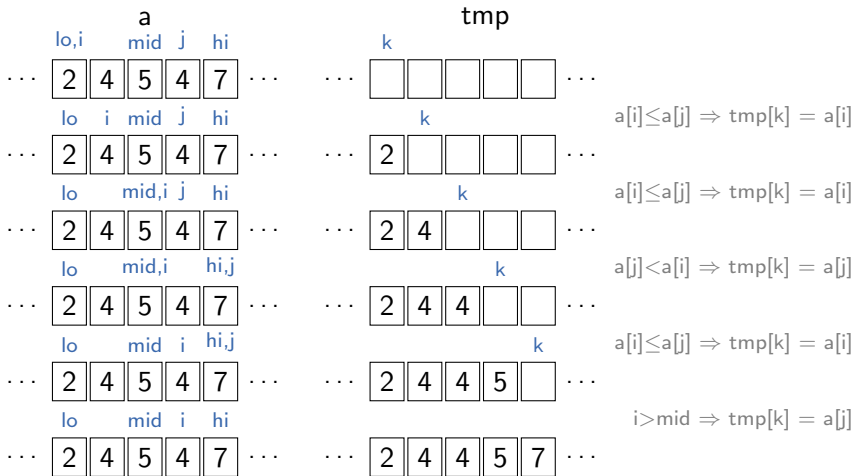
initialize:  $i := lo$ ,  $j := mid + 1$ ,  $k := lo$



## Merge Step: Example

Array tmp has same size as input array.

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



## Merge Step: Algorithm

---

```
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]
```

---

## Merge Step: Algorithm

---

```
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]
```

---

Also correct for  $lo = mid = hi$

# Jupyter Notebook



Jupyter notebook: `merge_sort.ipynb`

# Questions



Questions?

# Top-Down Merge Sort



# Merge Sort: Algorithm

recursive top-down variant

---

```
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     # //: floor division
10    sort_aux(array, tmp, lo, mid)
11    sort_aux(array, tmp, mid + 1, hi)
12    merge(array, tmp, lo, mid, hi)
```

---

## Possible Improvements

- on short sequences, insertion sort faster than merge sort  
→ use insertion sort for small `hi - lo`

## Possible Improvements

- on short sequences, insertion sort faster than merge sort  
→ use insertion sort for small  $hi - lo$
- directly skip the merge step if positions  $lo$  to  $hi$  already sorted  

```
if array[mid] <= array[mid + 1]:  
    return
```

## Possible Improvements

- on short sequences, insertion sort faster than merge sort  
→ use insertion sort for small  $hi - lo$
- directly skip the merge step if positions  $lo$  to  $hi$  already sorted  

```
if array[mid] <= array[mid + 1]:  
    return
```
- copying `tmp` in merge takes time  
→ swap role of array and `tmp` in every recursive call

## Merge Step: Correctness

- **Invariant:** at the end of each iteration of the loop:
  - $\text{tmp}[k] \leq \text{array}[m]$  for all  $i \leq m \leq \text{mid}$ , and
  - $\text{tmp}[k] \leq \text{array}[n]$  for all  $j \leq n \leq \text{hi}$ .
- $\text{tmp}$  is written from left to right.
- After the last iteration of the loop it holds for all  $\text{lo} \leq r < s \leq \text{hi}$  that  $\text{tmp}[r] \leq \text{tmp}[s]$  (= range is sorted).

# Merge Sort: Correctness

`sort_aux`:

- Proof by induction over length  $hi - lo$   
(always 1 smaller than the number of cells in the range)
- Basis  $hi - lo = -1$ : empty range is sorted.
- Basis  $hi - lo = 0$ : range with a single element is sorted.

# Merge Sort: Correctness

`sort_aux`:

- Proof by induction over length  $hi - lo$   
(always 1 smaller than the number of cells in the range)
- Basis  $hi - lo = -1$ : empty range is sorted.
- Basis  $hi - lo = 0$ : range with a single element is sorted.
- Induction hypothesis: merge sort is correct for all  $hi - lo < m$
- Inductive step ( $m - 1 \rightarrow m$ ):

# Merge Sort: Correctness

sort\_aux:

- Proof by induction over length  $hi - lo$   
(always 1 smaller than the number of cells in the range)
- Basis  $hi - lo = -1$ : empty range is sorted.
- Basis  $hi - lo = 0$ : range with a single element is sorted.
- Induction hypothesis: merge sort is correct for all  $hi - lo < m$
- Inductive step ( $m - 1 \rightarrow m$ ):  
Merge sort makes two recursive calls with  $hi - lo \leq \lfloor m/2 \rfloor$ ,  
afterwards the input is sorted between  $lo$  and  $mid$  and  
between  $mid + 1$  and  $hi$ . (by ind. hyp.)



# Merge Sort: Correctness

sort\_aux:

- Proof by induction over length  $hi - lo$   
(always 1 smaller than the number of cells in the range)
- Basis  $hi - lo = -1$ : empty range is sorted.
- Basis  $hi - lo = 0$ : range with a single element is sorted.
- Induction hypothesis: merge sort is correct for all  $hi - lo < m$
- Inductive step ( $m - 1 \rightarrow m$ ):  
Merge sort makes two recursive calls with  $hi - lo \leq \lfloor m/2 \rfloor$ ,  
afterwards the input is sorted **between lo and mid** and  
**between mid + 1 and hi**. (by ind. hyp.)  
  
Since the merge step is correct, at the end the entire range  
**from lo to hi is sorted**.

# Merge Sort: Correctness

sort\_aux:

- Proof by induction over length  $hi - lo$   
(always 1 smaller than the number of cells in the range)
- Basis  $hi - lo = -1$ : empty range is sorted.
- Basis  $hi - lo = 0$ : range with a single element is sorted.
- Induction hypothesis: merge sort is correct for all  $hi - lo < m$
- Inductive step ( $m - 1 \rightarrow m$ ):  
Merge sort makes two recursive calls with  $hi - lo \leq \lfloor m/2 \rfloor$ , afterwards the input is sorted between  $lo$  and  $mid$  and between  $mid + 1$  and  $hi$ . (by ind. hyp.)  
Since the merge step is correct, at the end the entire range from  $lo$  to  $hi$  is sorted.

Merge sort: calls sort\_aux for the entire range of the input, thus at the end the entire input has been sorted.

## Merge Sort: Properties (Slido)

---

```
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     # //: floor division
10    sort_aux(array, tmp, lo, mid)
11    sort_aux(array, tmp, mid + 1, hi)
12    merge(array, tmp, lo, mid, hi)
```

---

Which of the following properties does merge sort have? In-place? Adaptive? Stable?



# Merge Sort: Properties

- **not in-place**: uses non-constant storage for tmp and call stack
- **running time**: not adaptive  
(except with merge-skipping improvement)  
precise analysis: later chapter
- **stable**: merge prefers array[i] if array[i] equals array[j].

# Questions

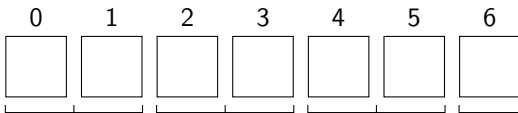


Questions?

# Bottom-Up Merge Sort

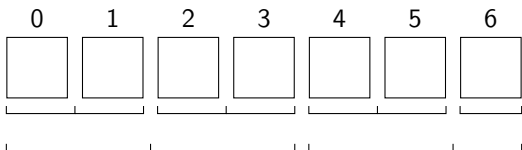


# Bottom-Up Variant

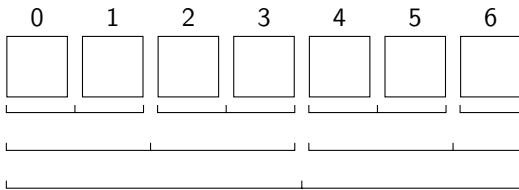




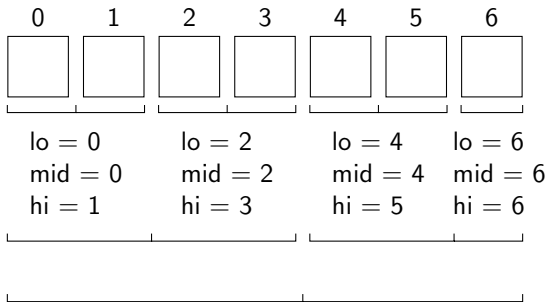
# Bottom-Up Variant



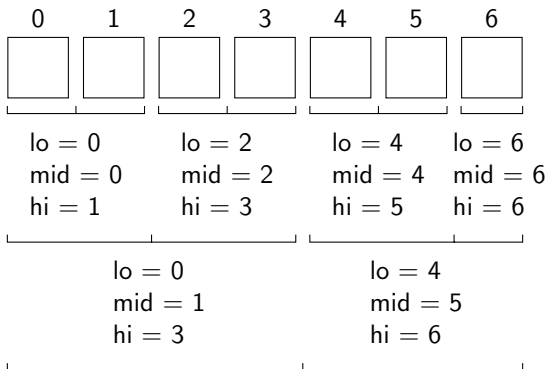
# Bottom-Up Variant



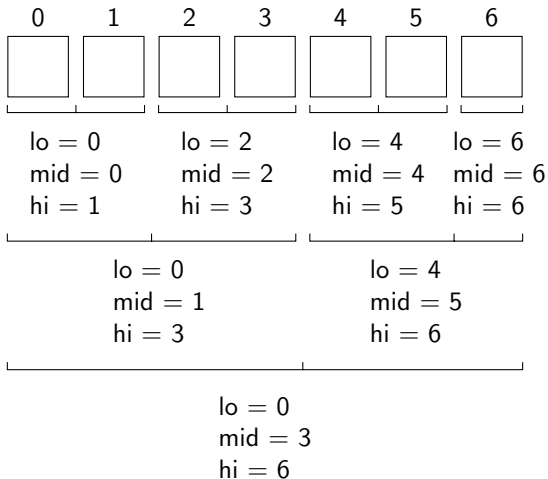
# Bottom-Up Variant



# Bottom-Up Variant



# Bottom-Up Variant



# Bottom-Up Merge Sort: Algorithm

iterative bottom-up variant

---

```
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
```

---

# Questions



Questions?

# Summary



# Summary

- Merge sort is a **divide-and-conquer** algorithm, which divides the input sequence into two roughly equally-sized ranges.
- The **merge step** combines two already sorted ranges.
- Merge sort is **stable**, but does **not work in-place**.
- The **top-down variant** is a **recursive** algorithm.
- The **bottom-up variant** is an **iterative** algorithm.