Algorithms and Data Structures

A4. Sorting II: Merge Sort

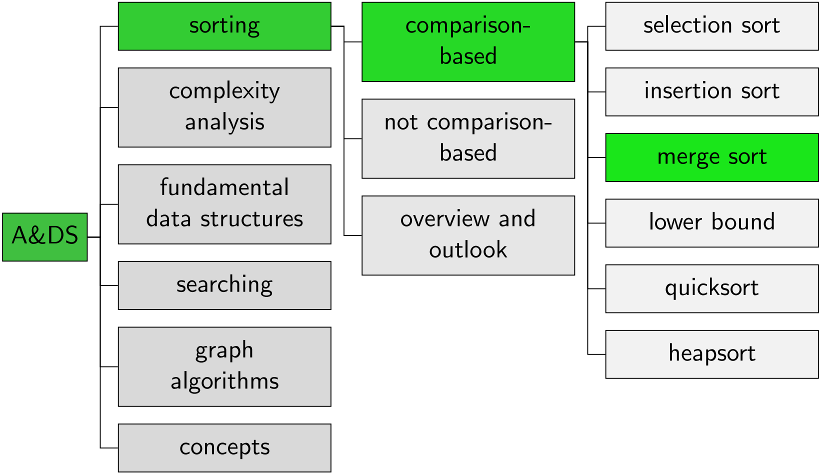
Gabriele Röger

University of Basel

February 29/March 6, 2024

A4.1 Merge Sort

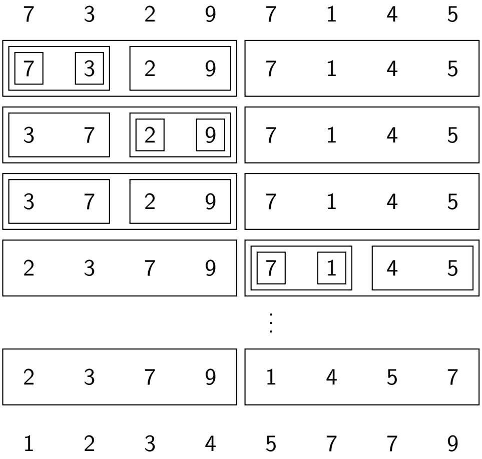
Content of the Course



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



(Detailed animation in screen version of slides)

A4.2 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] bis 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

	a							tmp						
	lo,i		mid	j	hi			k						
⋯	2	4	5	4	7	⋯	⋯						⋯	
	lo	i	mid	j	hi				k					a[i]≤a[j] ⇒ tmp[k] = a[i]
⋯	2	4	5	4	7	⋯	⋯	2					⋯	
	lo		mid,i	j	hi					k				a[i]≤a[j] ⇒ tmp[k] = a[i]
⋯	2	4	5	4	7	⋯	⋯	2	4				⋯	
	lo		mid,i		hi,j						k			a[j]<a[i] ⇒ tmp[k] = a[j]
⋯	2	4	5	4	7	⋯	⋯	2	4	4			⋯	
	lo		mid	i	hi,j							k		a[i]≤a[j] ⇒ tmp[k] = a[i]
⋯	2	4	5	4	7	⋯	⋯	2	4	4	5		⋯	
	lo		mid	i	hi									i>mid ⇒ tmp[k] = a[j]
⋯	2	4	5	4	7	⋯	⋯	2	4	4	5	7	⋯	

Merge Step: Algorithm

```
def merge(array, tmp, lo, mid, hi):
    i = lo
    j = mid + 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        if j > hi or (i <= mid and array[i] <= array[j]):
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j]
            j += 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        array[k] = tmp[k]
```

Also correct for lo $=$ mid $=$ hi

Jupyter Notebook

Jupyter notebook: `merge_sort.ipynb`

A4.3 Top-Down Merge Sort

Merge Sort: Algorithm

recursive top-down variant

```
def sort(array):
    tmp = [0] * len(array)  # [0,...,0] with same size as array
    sort_aux(array, tmp, 0, len(array) - 1)

def sort_aux(array, tmp, lo, hi):
    if hi <= lo:
        return
    mid = lo + (hi - lo) // 2
    # //: floor division
    sort_aux(array, tmp, lo, mid)
    sort_aux(array, tmp, mid + 1, hi)
    merge(array, tmp, lo, mid, hi)
```

Possible Improvements

- on short sequences, insertion sort faster than merge sort
  $\rightarrow$ 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
  $\rightarrow$ swap role of `array` and `tmp` in every recursive call

Merge Step: Correctness

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

Merge Sort: Correctness

`sort_aux`:

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

```
def sort(array):
    tmp = [0] * len(array)  # [0,...,0] with same size as array
    sort_aux(array, tmp, 0, len(array) - 1)

def sort_aux(array, tmp, lo, hi):
    if hi <= lo:
        return
    mid = lo + (hi - lo) // 2
    # //: floor division
    sort_aux(array, tmp, lo, mid)
    sort_aux(array, tmp, mid + 1, hi)
    merge(array, tmp, lo, mid, hi)
```

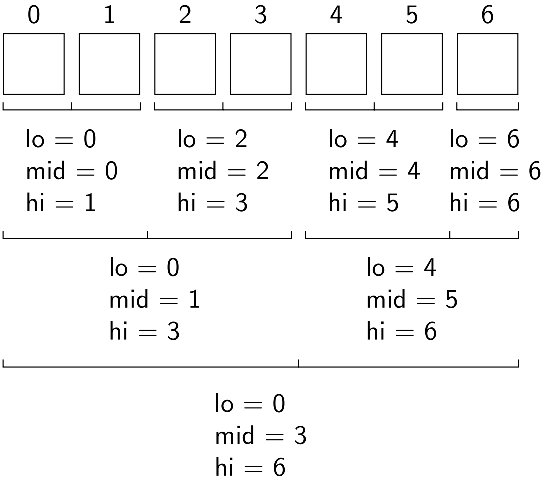
Which of the following properties does merge sort have? In-place? Adaptive? Stable?

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]`.

A4.4 Bottom-Up Merge Sort

Bottom-Up Variant



Bottom-Up Merge Sort: Algorithm

iterative bottom-up variant

```
def sort(array):
    n = len(array)
    tmp = [0] * n
    length = 1
    while length < n:
        lo = 0
        while lo < n - length:
            mid = lo + length - 1
            hi = min(lo + 2 * length - 1, n - 1)
            merge(array, tmp, lo, mid, hi)
            lo += 2 * length
        length *= 2
```

A4.5 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 to 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.