4 Efficient Sorting

3 November 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

version `2025-11-04 14:01`

Learning Outcomes

Unit 4: ***Efficient Sorting***

1. Know principles and implementation of *mergesort* and *quicksort*.
2. Know properties and *performance characteristics* of mergesort and quicksort.
3. Know the comparison model and understand the corresponding *lower bound*.
4. Understand *counting sort* and how it circumvents the comparison lower bound.
5. Know ways how to exploit *presorted* inputs.

Outline

4 Efficient Sorting

Why study sorting?

- fundamental problem of computer science that is still not solved

Algorithm with optimal #comparisons in worst case?

- building brick of many more advanced algorithms
  - for preprocessing
  - as subroutine
- playground of manageable complexity
  to practice algorithmic techniques

Here:

- "classic" fast sorting method
- exploit **partially sorted** inputs
- **parallel** sorting

later

Part I

The Basics

Rules of the game

- **Given:**
  - array $A[0..n) = A[0..n-1]$ of n objects
  - a total order relation $\leq$ among $A[0], \ldots, A[n-1]$
    (a comparison function)
    Python: elements support <= operator (`__le__()`)
    Java: `Comparable` class (`x.compareTo(y) <= 0`)
- **Goal:** rearrange (i. e., permute) elements within A,
  so that A is *sorted*, i. e., $A[0] \leq A[1] \leq \cdots \leq A[n-1]$

- for now: A stored in main memory (*internal sorting*)
  single processor (*sequential sorting*)

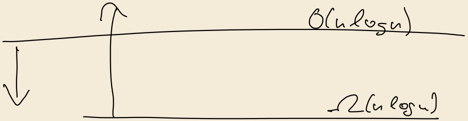
Clicker Question

$\Theta(n \log n)$

What is the complexity of sorting? Type you answer, e. g., as "`Theta(sqrt(n))`"

(a) $O(n \log n)$ algorithm solving the problem

(b) lower bound for problem $\Omega(n \log n)$

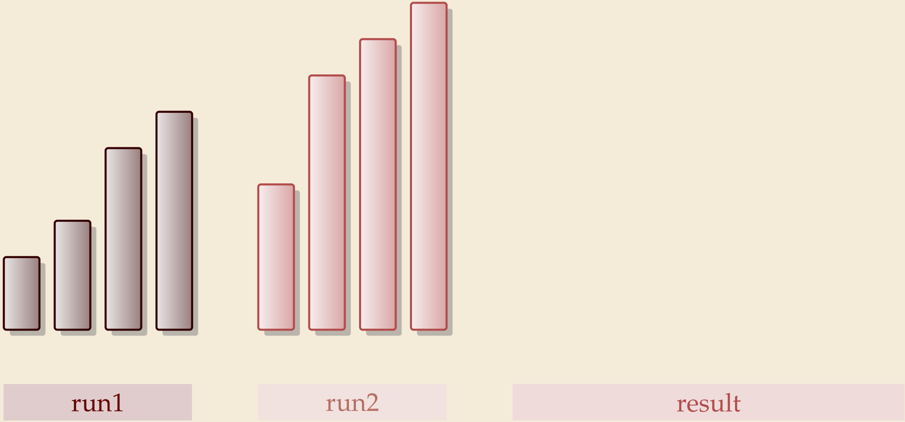


→ sli.do/cs566

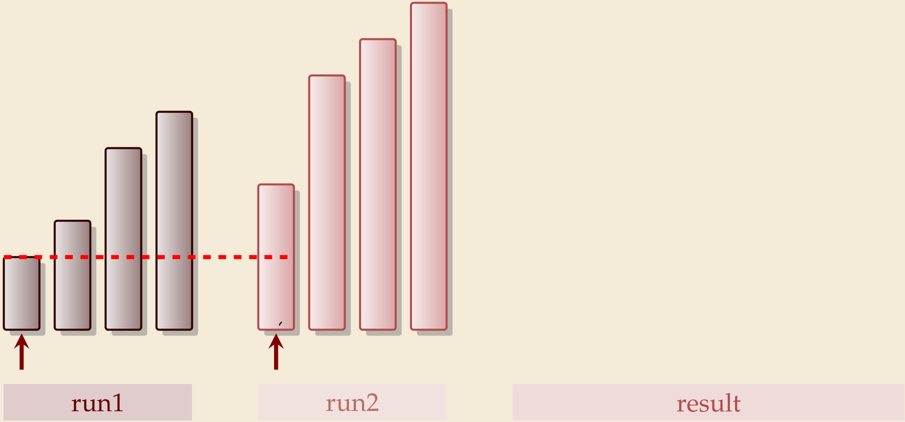
4.1 Mergesort

Merging sorted lists

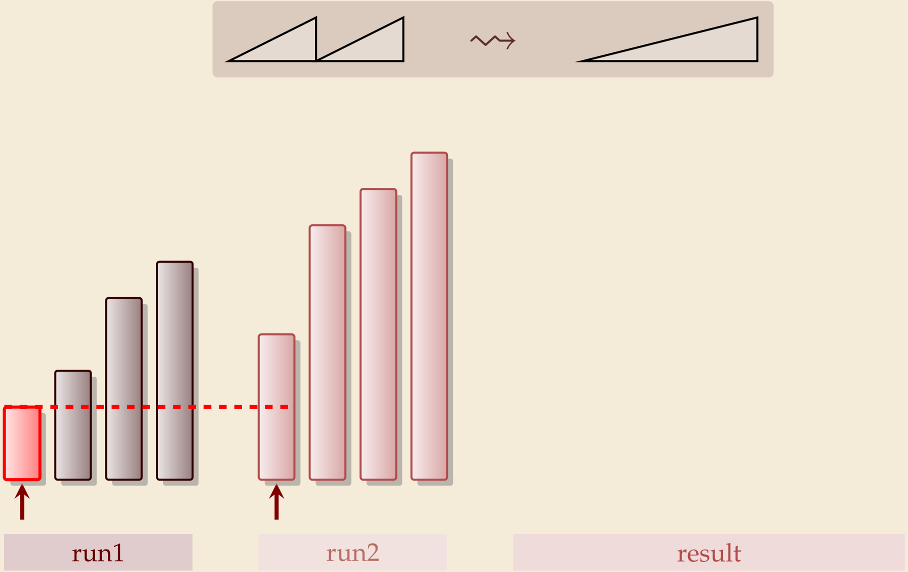
Merging sorted lists



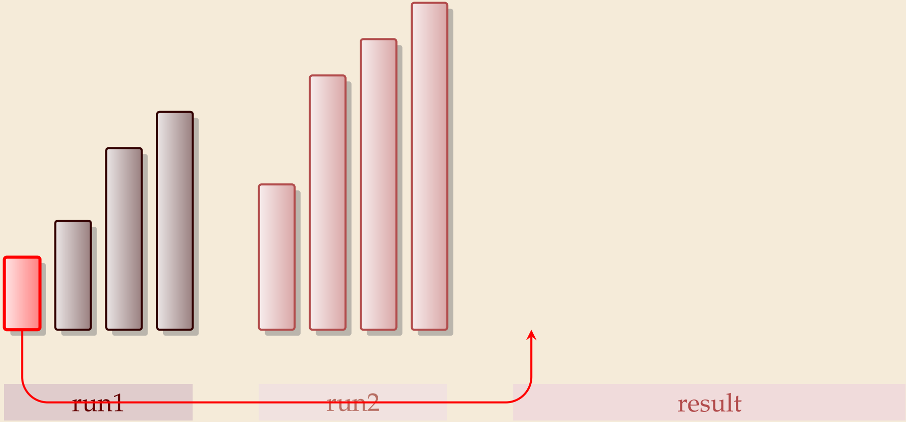
Merging sorted lists



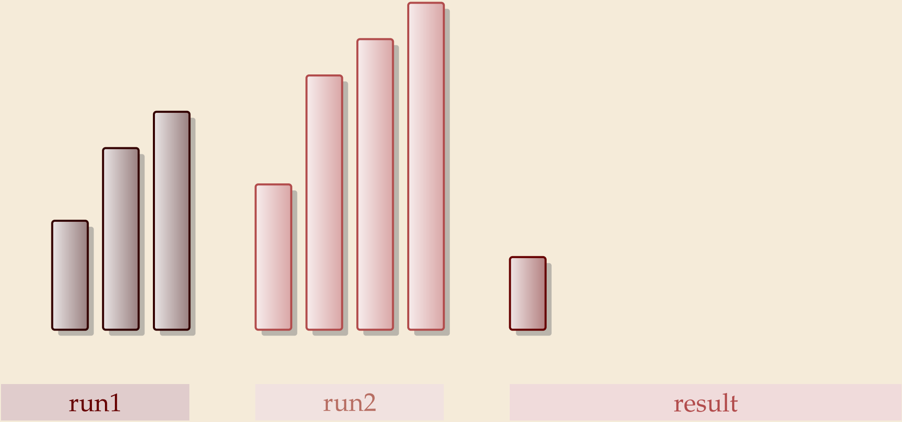
Merging sorted lists



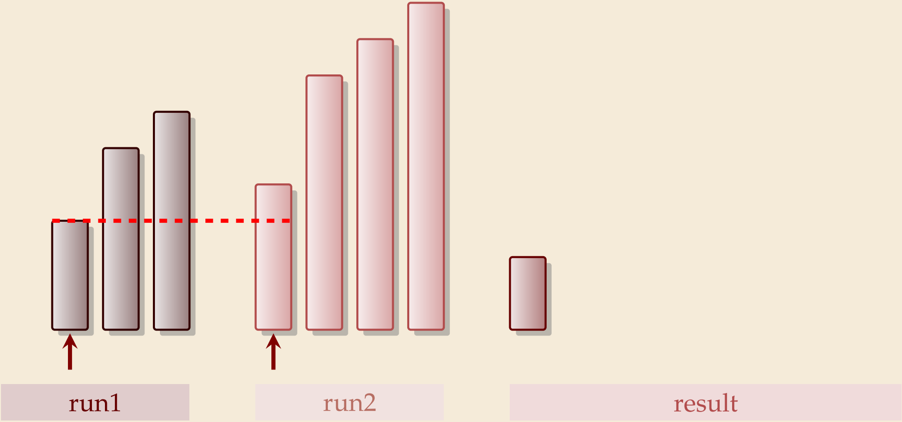
Merging sorted lists



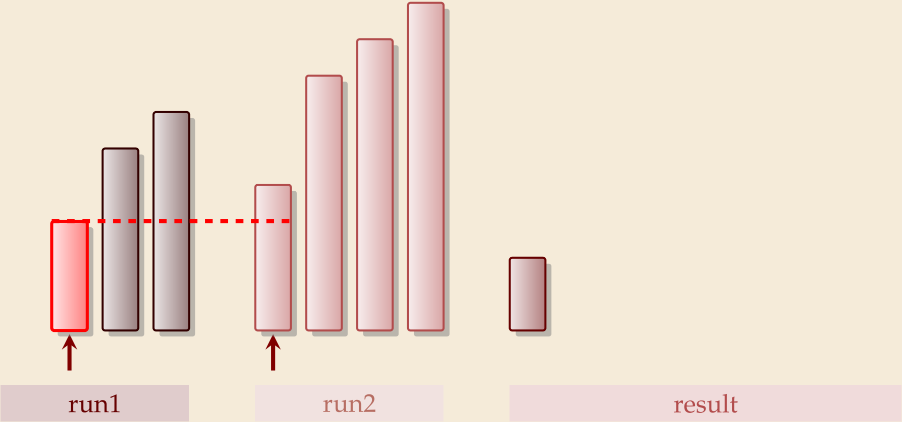
Merging sorted lists



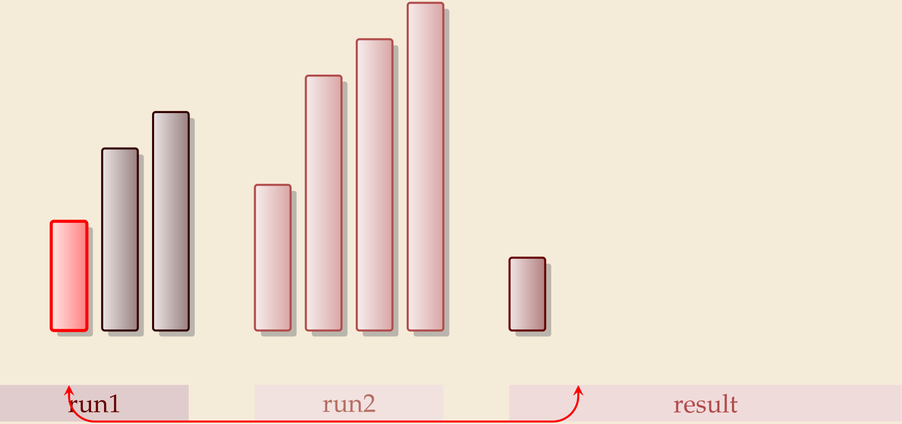
Merging sorted lists



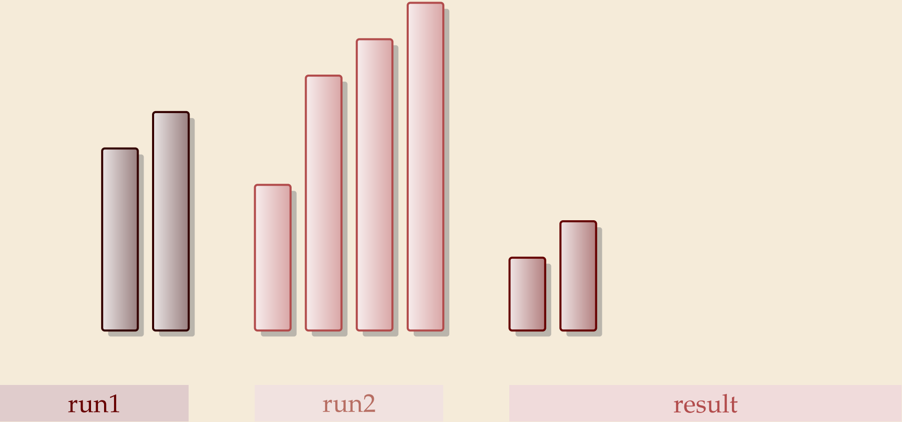
Merging sorted lists



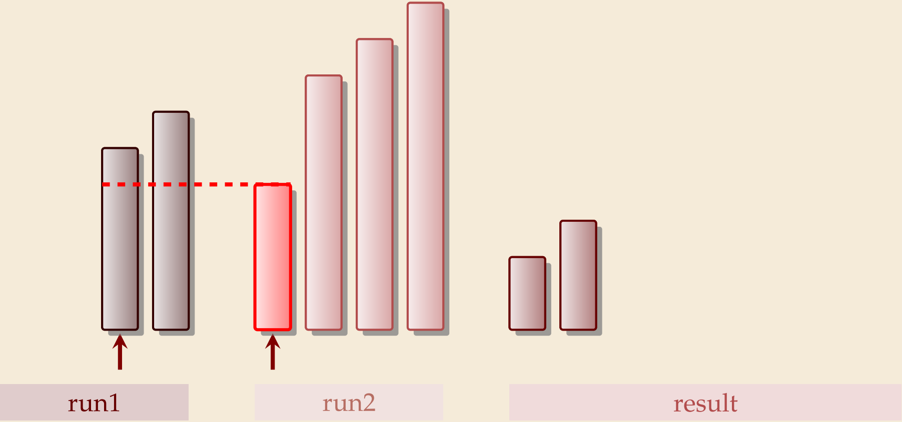
Merging sorted lists



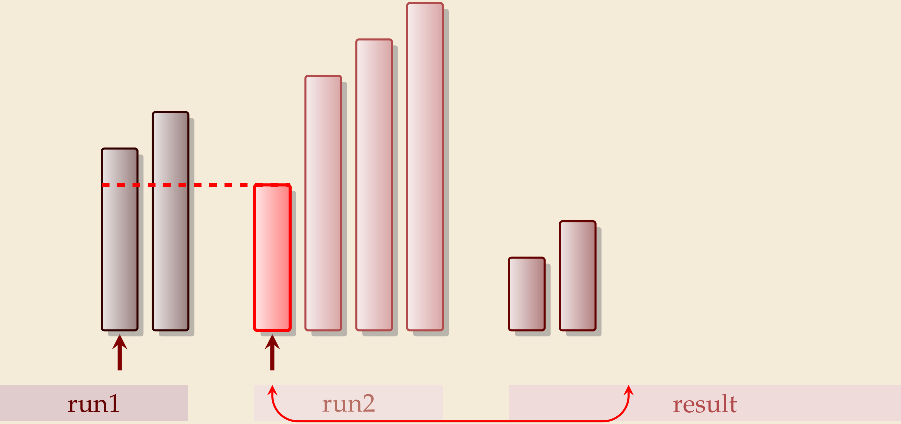
Merging sorted lists



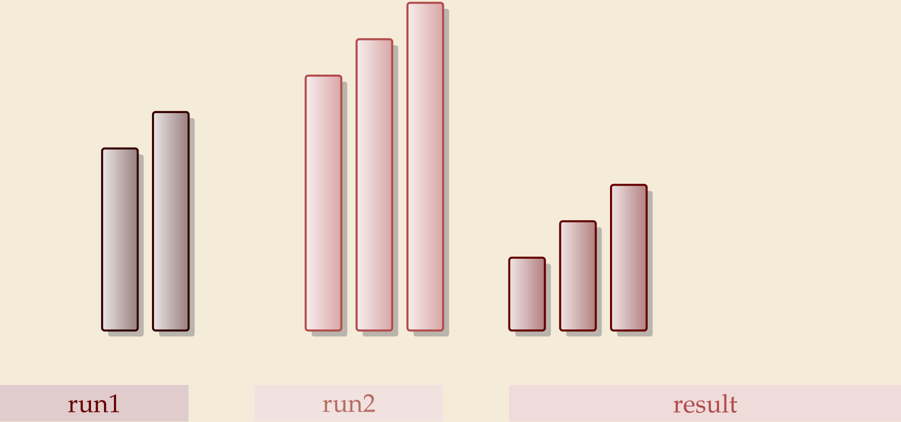
Merging sorted lists



Merging sorted lists



Merging sorted lists



Merging sorted lists

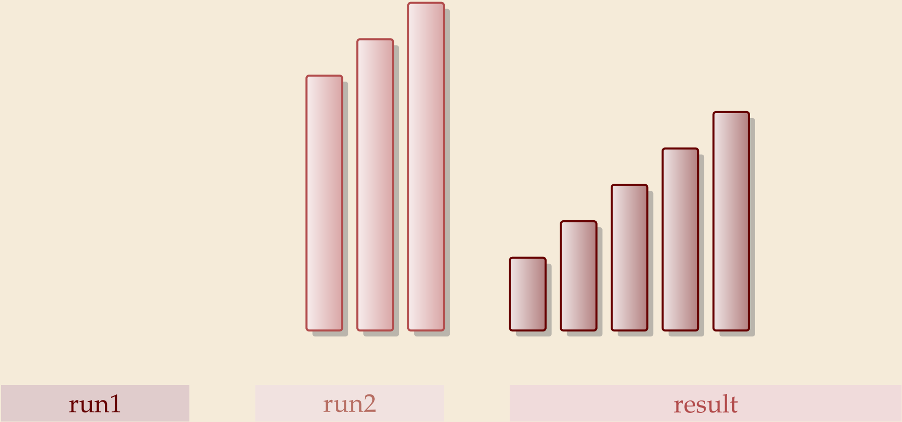
Merging sorted lists

Merging sorted lists

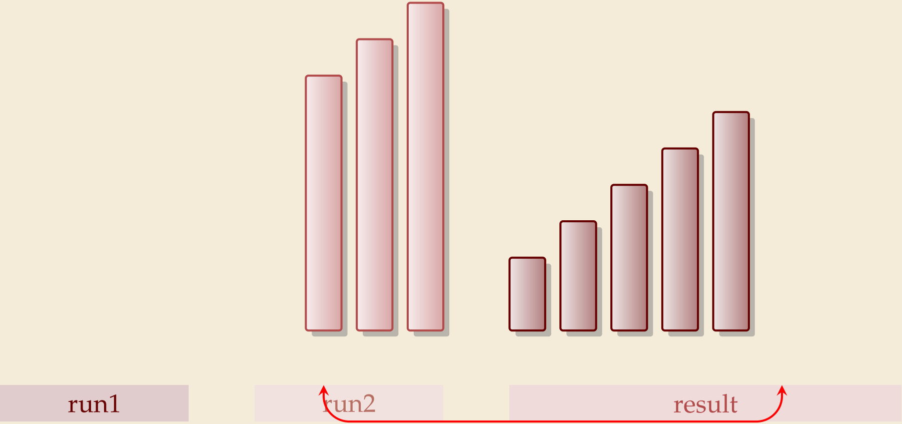
Merging sorted lists

Merging sorted lists

Merging sorted lists

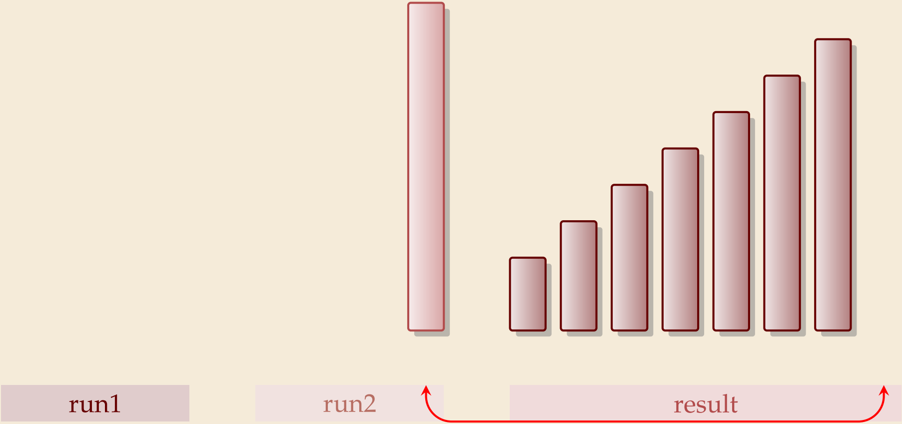


Merging sorted lists

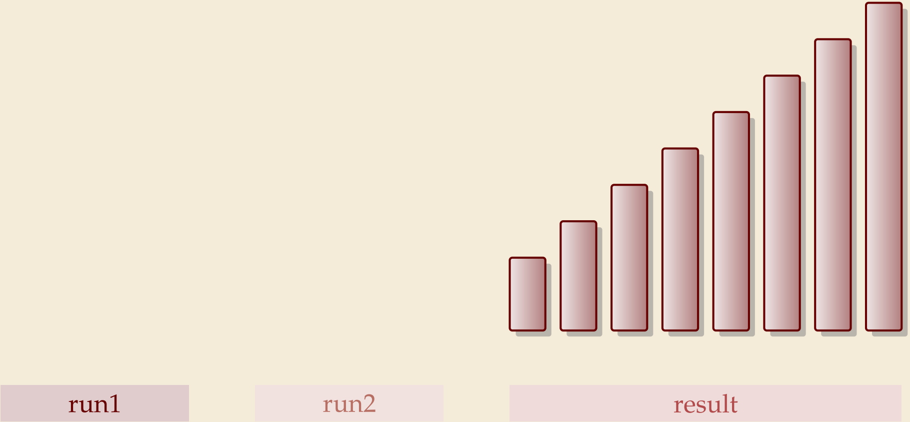


Merging sorted lists

Merging sorted lists



Merging sorted lists



Clicker Question

What is the worst-case running time of mergesort?

A	$\Theta(1)$	**G**	$\Theta(n \log n)$
B	$\Theta(\log n)$	**H**	$\Theta(n \log^2 n)$
C	$\Theta(\log \log n)$	**I**	$\Theta(n^{1+\epsilon})$
D	$\Theta(\sqrt{n})$	**J**	$\Theta(n^2)$
E	$\Theta(n)$	**K**	$\Theta(n^3)$
F	$\Theta(n \log \log n)$	**L**	$\Theta(2^n)$

→ sli.do/cs566

Clicker Question

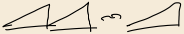
What is the worst-case running time of mergesort?

- **A** $\cancel{\Theta(1)}$
- **B** $\cancel{\Theta(\log n)}$
- **C** $\cancel{\Theta(\log \log n)}$
- **D** $\cancel{\Theta(\sqrt{n})}$
- **E** $\cancel{\Theta(n)}$
- **F** $\cancel{\Theta(n \log \log n)}$
- **G** $\Theta(n \log n)$ ✓
- **H** $\cancel{\Theta(n \log^2 n)}$
- **I** $\cancel{\Theta(n^{1+\epsilon})}$
- **J** $\cancel{\Theta(n^2)}$
- **K** $\cancel{\Theta(n^3)}$
- **L** $\cancel{\Theta(2^n)}$

→ sli.do/cs566

Mergesort

```
procedure mergesort(A[l..r]):
    n := r − l
    if n ≤ 1 return
    m := l + ⌊n/2⌋
    mergesort(A[l..m))
    mergesort(A[m..r))
    merge(A[l..m), A[m..r), buf)
    copy buf to A[l..r)
```



- recursive procedure
- merging needs
  - temporary storage *buf* for result (of same size as merged runs)
  - to read and write each element twice (once for merging, once for copying back)

Mergesort

```
procedure mergesort(A[l..r]):
    n := r − l
    if n ≤ 1 return
    m := l + ⌊n/2⌋
    mergesort(A[l..m))
    mergesort(A[m..r))
    merge(A[l..m), A[m..r), buf)
    copy buf to A[l..r)
```

- recursive procedure
- merging needs
  - temporary storage *buf* for result (of same size as merged runs)
  - to read and write each element twice (once for merging, once for copying back)

Analysis: count *"element visits"* (read and/or write)

$$C(n) \;=\; \begin{cases} 0 & n \le 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2n & n \ge 2 \end{cases}$$

same for best and worst case!

Simplification $\boxed{n = 2^k}$

$$C(2^k) \;=\; \begin{cases} 0 & k \le 0 \\ 2 \cdot C(2^{k-1}) + 2 \cdot 2^k & k \ge 1 \end{cases} \;=\; 2 \cdot 2^k + 2^2 \cdot 2^{k-1} + 2^3 \cdot 2^{k-2} + \cdots + 2^k \cdot 2^1 \;=\; 2k \cdot 2^k$$

$$C(n) \;=\; 2n \lg(n) \;=\; \Theta(n \log n)$$

(arbitrary n: $C(n) \le C(\text{next larger power of 2}) \le 4n\lg(n) + 2n = \Theta(n \log n)$)

Mergesort

```
procedure mergesort(A[l..r]):
    n := r − l
    if n ≤ 1 return
    m := l + ⌊n/2⌋
    mergesort(A[l..m))
    mergesort(A[m..r))
    merge(A[l..m), A[m..r), buf)
    copy buf to A[l..r)
```

- recursive procedure
- merging needs
  - temporary storage *buf* for result (of same size as merged runs)
  - to read and write each element twice (once for merging, once for copying back)

Analysis: count *"element visits"* (read and/or write)

$$C(n) \;=\; \begin{cases} 0 & n \le 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2n & n \ge 2 \end{cases}$$

same for best and worst case!

(precisely(!) solvable *without* assumption $n = 2^k$:
$C(n) = 2n\lg(n) + \big(2 - \{\lg(n)\} - 2^{1-\{\lg(n)\}}\big)2n$
with $\{x\} := x - \lfloor x \rfloor$)

Simplification $\boxed{n = 2^k}$

$$C(2^k) \;=\; \begin{cases} 0 & k \le 0 \\ 2 \cdot C(2^{k-1}) + 2 \cdot 2^k & k \ge 1 \end{cases} \;=\; 2 \cdot 2^k + 2^2 \cdot 2^{k-1} + 2^3 \cdot 2^{k-2} + \cdots + 2^k \cdot 2^1 \;=\; 2k \cdot 2^k$$

$$C(n) \;=\; 2n\lg(n) \;=\; \Theta(n \log n)$$

(arbitrary n: $C(n) \le C(\text{next larger power of 2}) \le \underline{4n\lg(n) + 2n} = \Theta(n \log n)$)

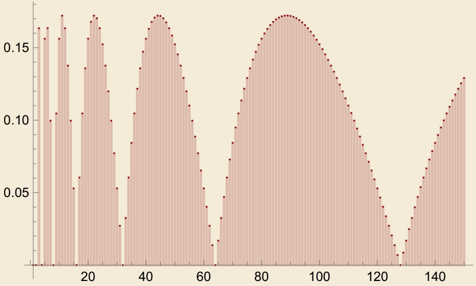
Linear Term of $C(n)$

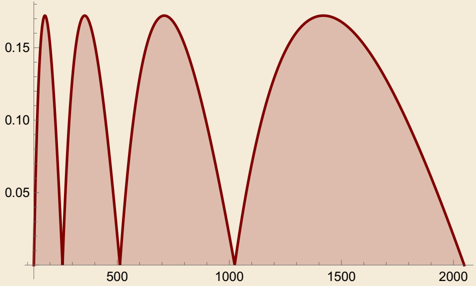
$\notin$ exam

Recall:

$$C(n) \;=\; 2n\lg(n) + \underbrace{\left(2 - \{\lg(n)\} - 2^{1-\{\lg(n)\}}\right)}2n \qquad \text{with } \{x\} := x - \lfloor x \rfloor$$

Plot of $2\left(2 - \{\lg(n)\} - 2^{1-\{\lg(n)\}}\right)$





$\rightsquigarrow$ Can prove: $\boxed{C(n) \;\le\; 2n\lg n + 0.172n}$

Mergesort – Discussion

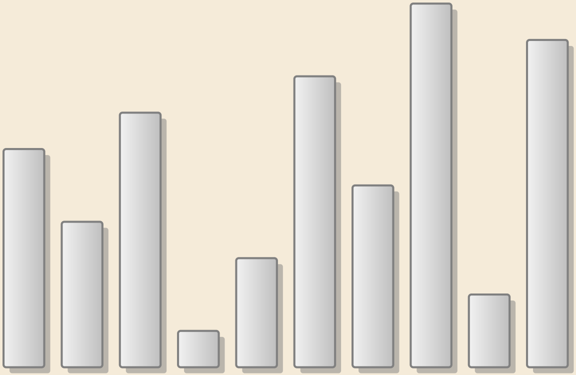
- optimal time complexity of $\Theta(n \log n)$ in the worst case
- *stable* sorting method   i. e., retains relative order of equal-key items
- memory access is sequential (scans over arrays)
- requires $\Theta(n)$ extra space

  there are in-place merging methods,
  but they are substantially more complicated
  and not (widely) used

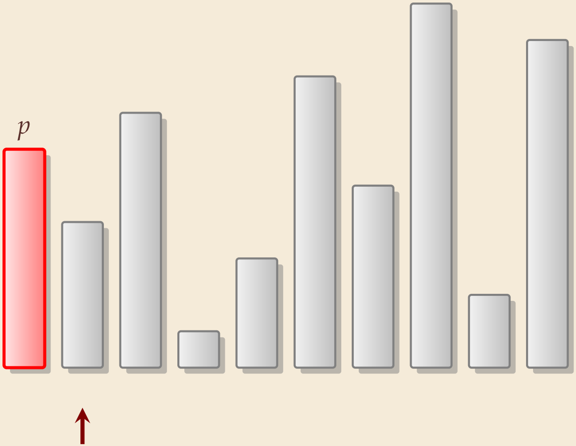
4.2 Quicksort

Partitioning around a pivot

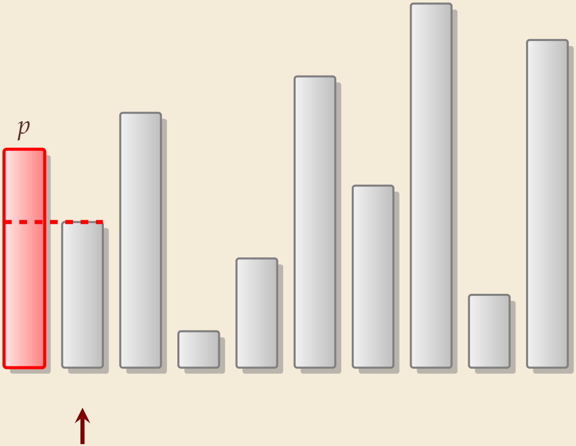
Partitioning around a pivot



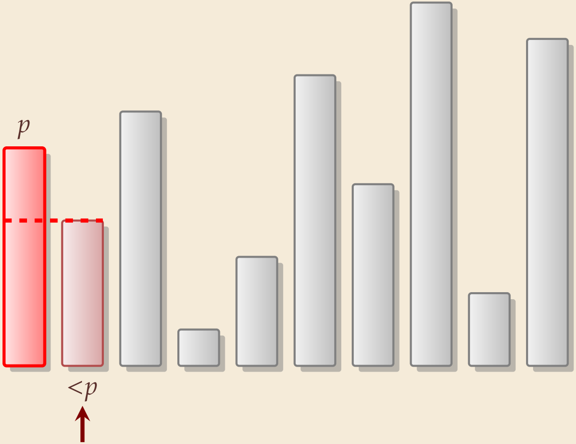
Partitioning around a pivot



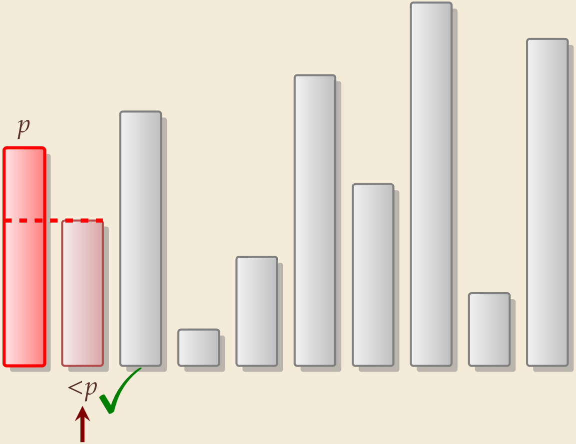
Partitioning around a pivot



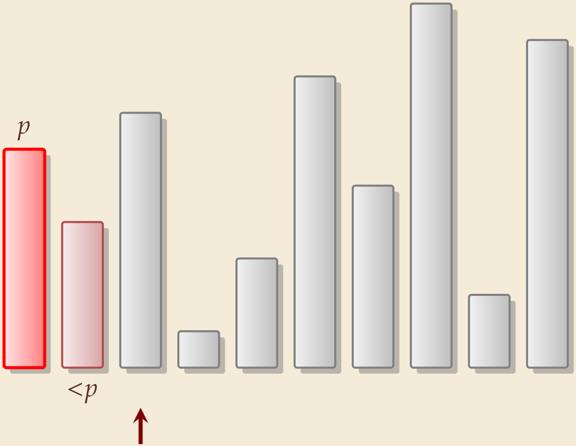
Partitioning around a pivot



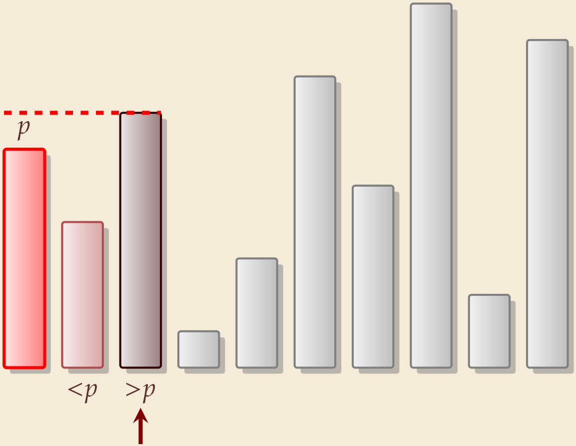
Partitioning around a pivot



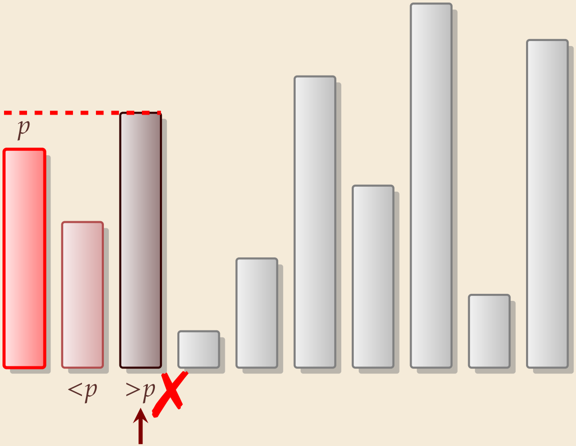
Partitioning around a pivot



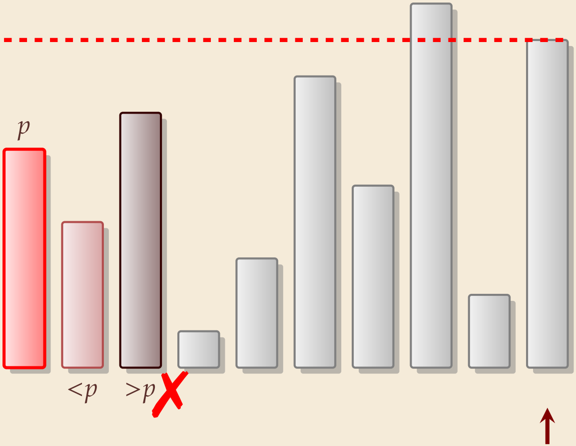
Partitioning around a pivot



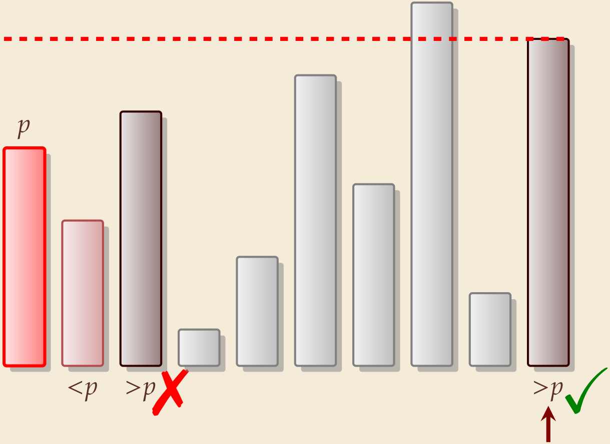
Partitioning around a pivot



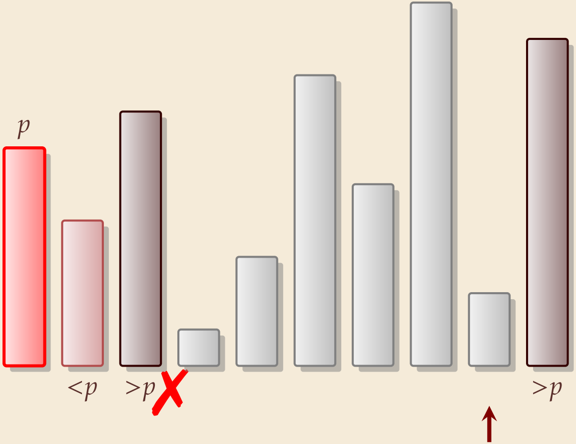
Partitioning around a pivot



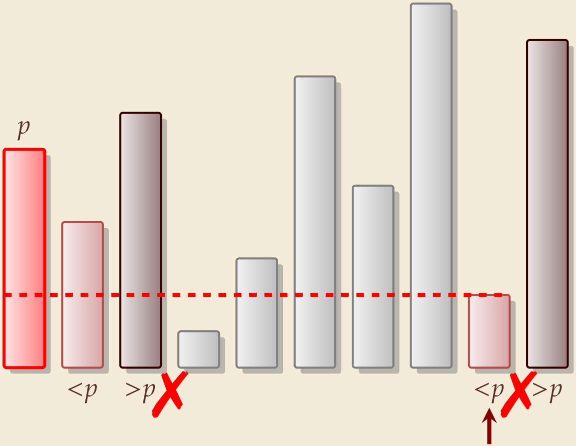
Partitioning around a pivot



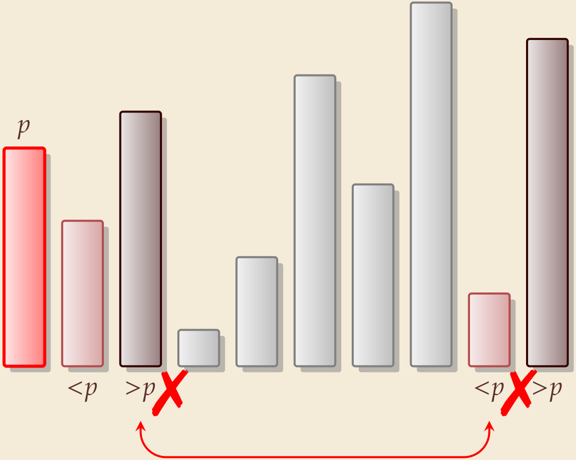
Partitioning around a pivot



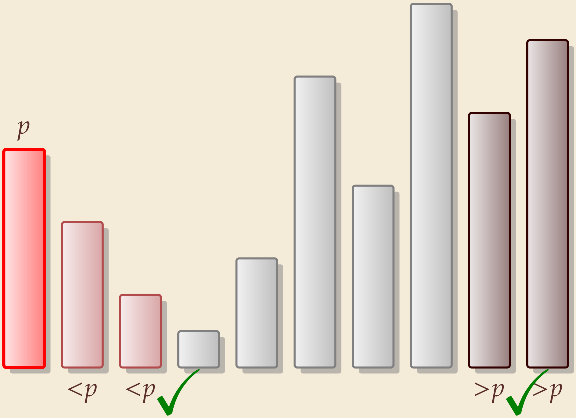
Partitioning around a pivot



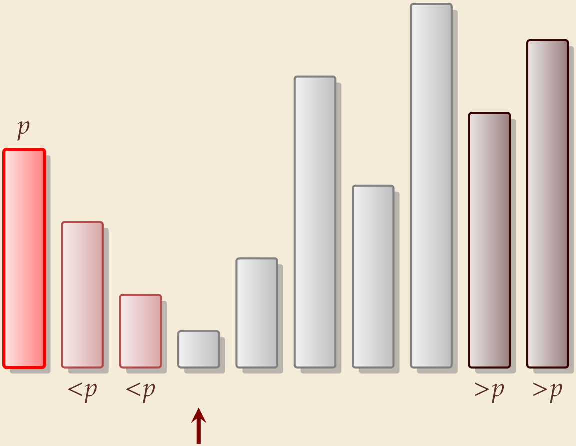
Partitioning around a pivot



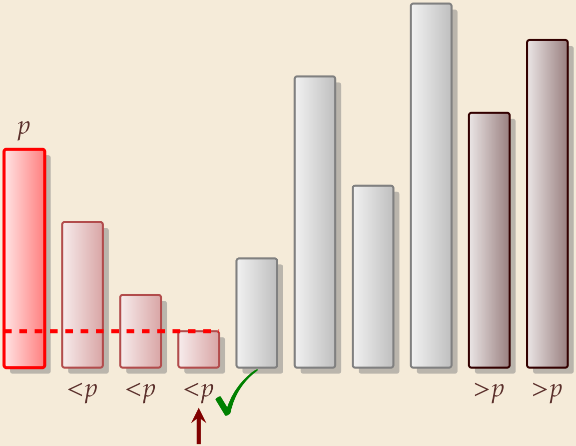
Partitioning around a pivot



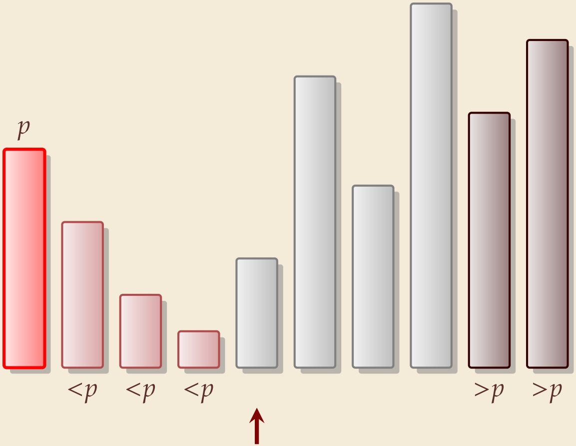
Partitioning around a pivot



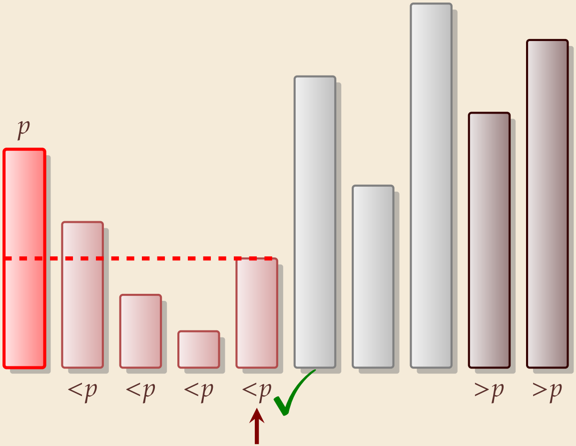
Partitioning around a pivot



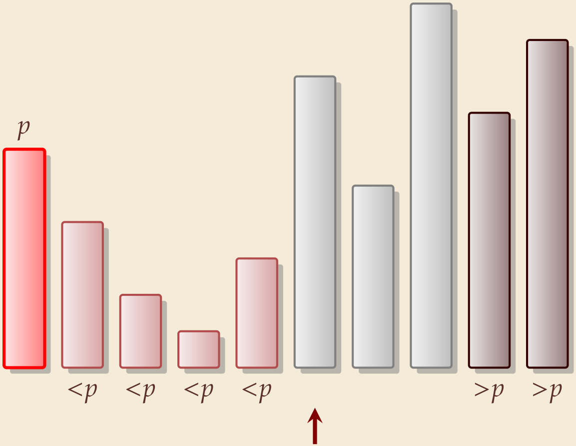
Partitioning around a pivot



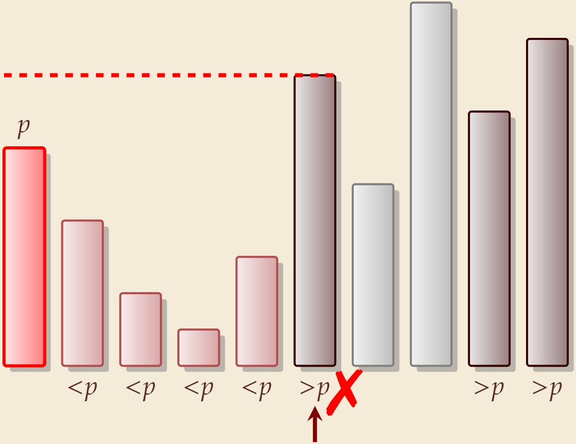
Partitioning around a pivot



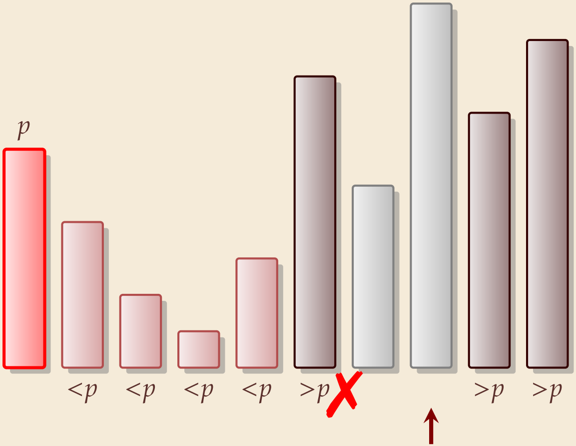
Partitioning around a pivot



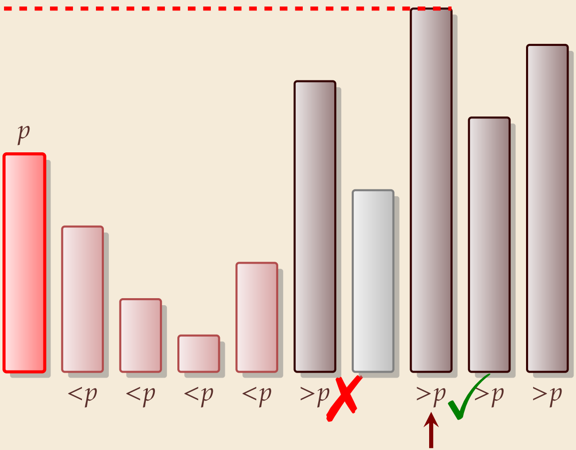
Partitioning around a pivot



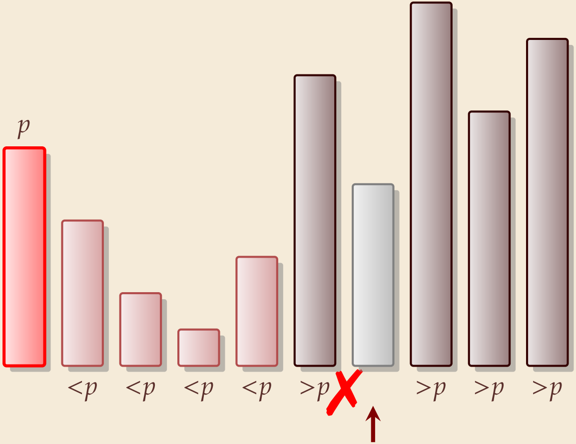
Partitioning around a pivot



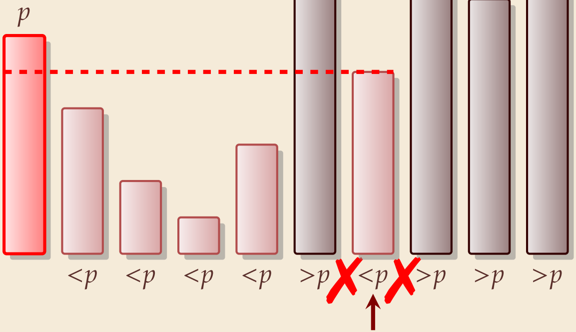
Partitioning around a pivot



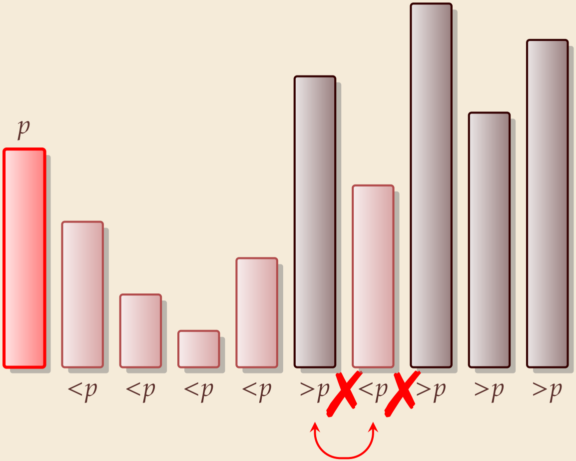
Partitioning around a pivot



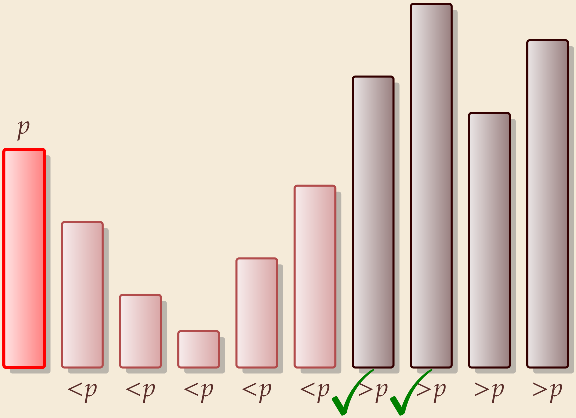
Partitioning around a pivot



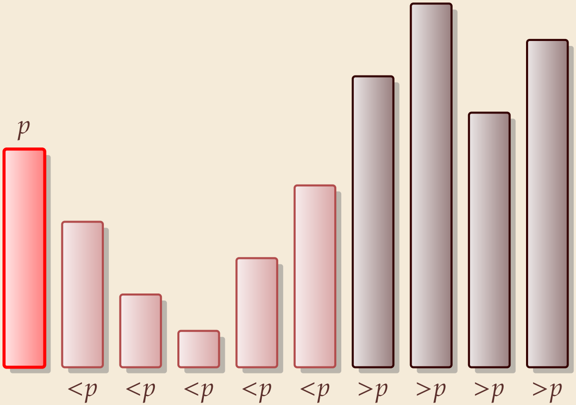
Partitioning around a pivot



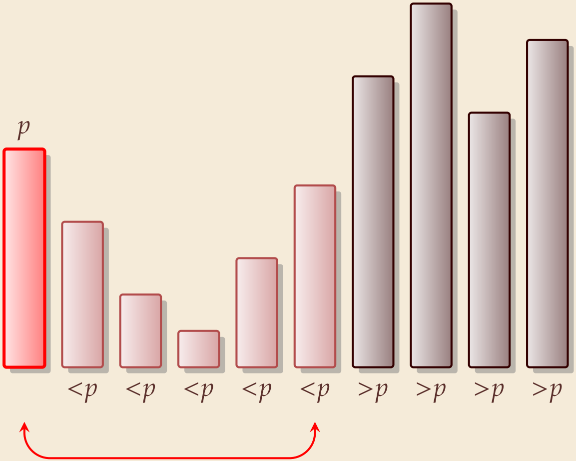
Partitioning around a pivot



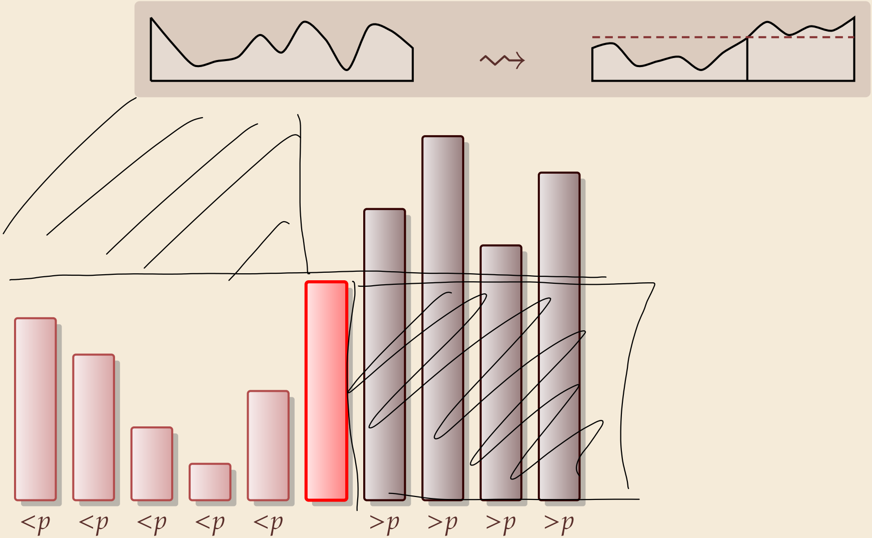
Partitioning around a pivot



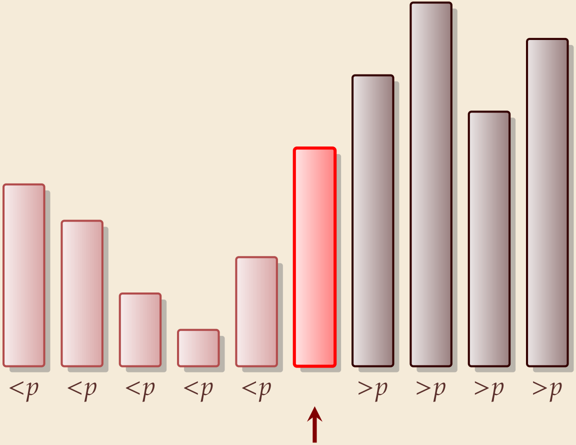
Partitioning around a pivot



Partitioning around a pivot



Partitioning around a pivot



- no extra space needed
- visits each element once
- returns rank/position of pivot

Partitioning – Detailed code

Beware: details easy to get wrong; use this code! (if you ever have to)

```
1  procedure partition(A, b):
2      // input: array A[0..n), position of pivot b ∈ [0..n)
3      swap(A[0], A[b])
4      i := 0,   j := n
5      while true do
6          do i := i + 1 while i < n and A[i] < A[0]
7          do j := j − 1 while j ≥ 1 and A[j] > A[0]
8          if i ≥ j then break     (goto 11)
9          else swap(A[i], A[j])
10     end while
11     swap(A[0], A[j])
12     return j
```

Loop invariant (5–10): A

p	$\leq p$	?	$\geq p$
	i	j	

Quicksort

```
procedure quicksort(A[l..r]):
    if r − ℓ ≤ 1 then return
    b := choosePivot(A[l..r])
    j := partition(A[l..r], b)
    quicksort(A[l..j))
    quicksort(A[j + 1..r))
```

- recursive procedure
- choice of pivot can be
  - fixed position $\rightsquigarrow$ dangerous!
  - random
  - more sophisticated, e. g., median of 3

Clicker Question

What is the worst-case running time of quicksort?

- **A** $\Theta(1)$
- **B** $\Theta(\log n)$
- **C** $\Theta(\log \log n)$
- **D** $\Theta(\sqrt{n})$
- **E** $\Theta(n)$
- **F** $\Theta(n \log \log n)$
- **G** $\Theta(n \log n)$
- **H** $\Theta(n \log^2 n)$
- **I** $\Theta(n^{1+\epsilon})$
- **J** $\Theta(n^2)$
- **K** $\Theta(n^3)$
- **L** $\Theta(2^n)$

→ sli.do/cs566

Clicker Question

What is the worst-case running time of quicksort?

- **A** $\cancel{\Theta(1)}$
- **B** $\cancel{\Theta(\log n)}$
- **C** $\cancel{\Theta(\log \log n)}$
- **D** $\cancel{\Theta(\sqrt{n})}$
- **E** $\cancel{\Theta(n)}$
- **F** $\cancel{\Theta(n \log \log n)}$
- **G** $\cancel{\Theta(n \log n)}$
- **H** $\cancel{\Theta(n \log^2 n)}$
- **I** $\cancel{\Theta(n^{1+\epsilon})}$
- **J** $\Theta(n^2)$ $\checkmark$
- **K** $\cancel{\Theta(n^3)}$
- **L** $\cancel{\Theta(2^n)}$

→ `sli.do/cs566`

Quicksort & Binary Search Trees

Quicksort

7	4	2	9	1	3	8	5	6

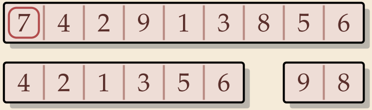
Quicksort & Binary Search Trees

Quicksort

7	4	2	9	1	3	8	5	6

Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

7	4	2	9	1	3	8	5	6

4	2	1	3	5	6	7	9	8

Quicksort & Binary Search Trees

Quicksort

7	4	2	9	1	3	8	5	6

4	2	1	3	5	6		9	8

7

Quicksort & Binary Search Trees

Quicksort

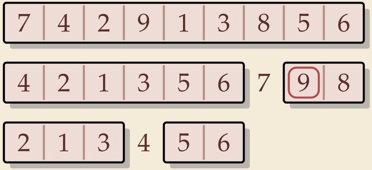
7 | 4 | 2 | 9 | 1 | 3 | 8 | 5 | 6

4 | 2 | 1 | 3 | 5 | 6   7   9 | 8

2 | 1 | 3   4   5 | 6

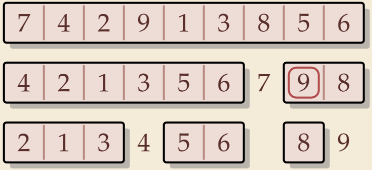
Quicksort & Binary Search Trees

Quicksort



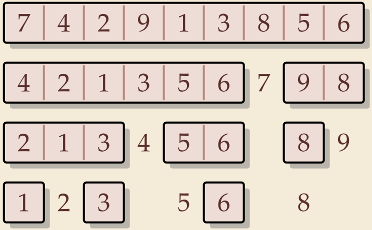
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

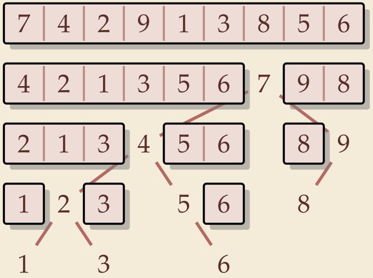


Quicksort & Binary Search Trees

Quicksort

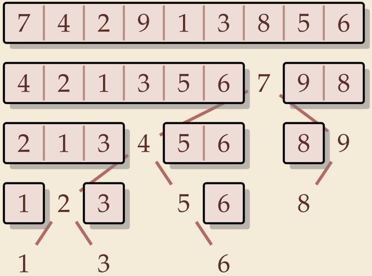
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

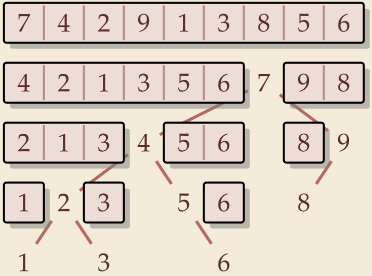


Binary Search Tree (BST)

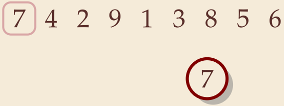
7 4 2 9 1 3 8 5 6

Quicksort & Binary Search Trees

Quicksort

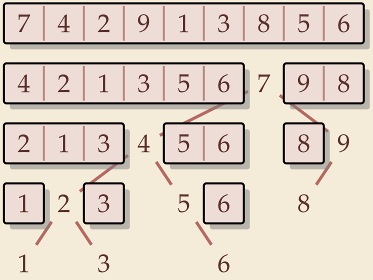


Binary Search Tree (BST)

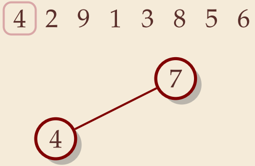


Quicksort & Binary Search Trees

Quicksort

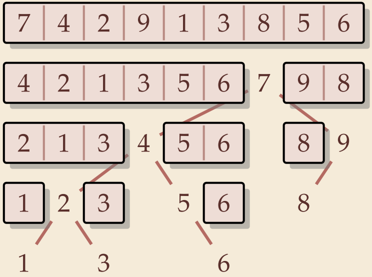


Binary Search Tree (BST)

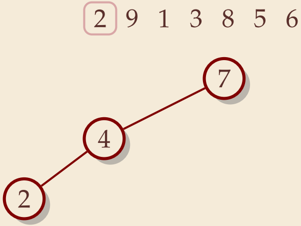


Quicksort & Binary Search Trees

Quicksort

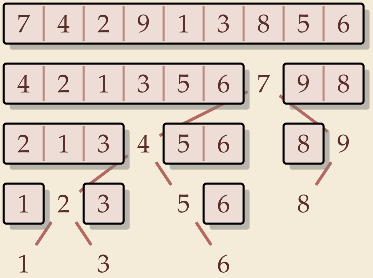


Binary Search Tree (BST)

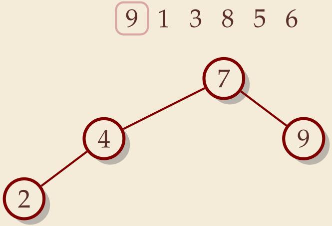


Quicksort & Binary Search Trees

Quicksort

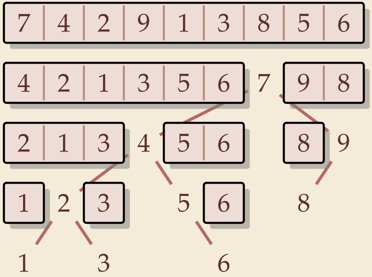


Binary Search Tree (BST)

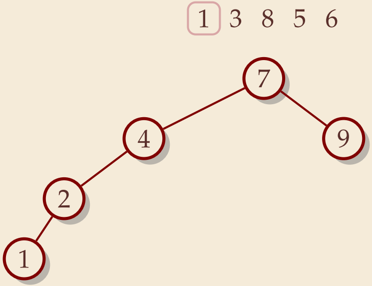


Quicksort & Binary Search Trees

Quicksort

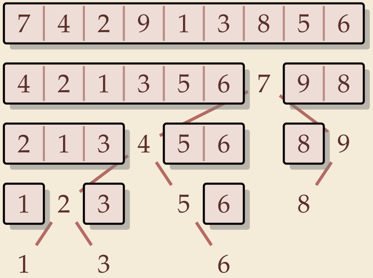


Binary Search Tree (BST)

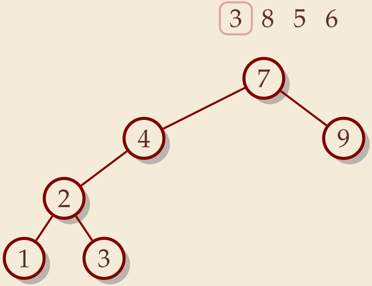


Quicksort & Binary Search Trees

Quicksort

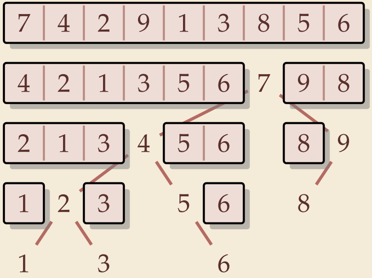


Binary Search Tree (BST)

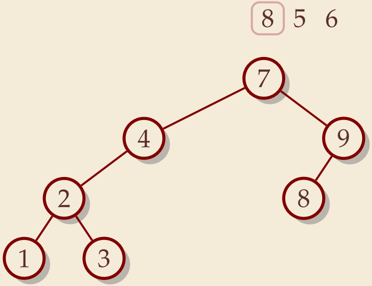


Quicksort & Binary Search Trees

Quicksort

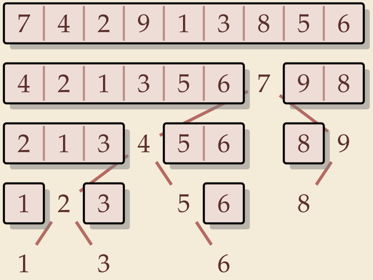


Binary Search Tree (BST)

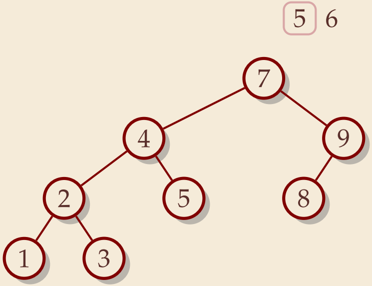


Quicksort & Binary Search Trees

Quicksort

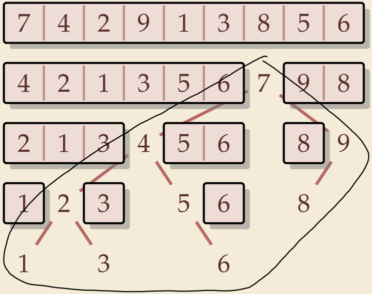


Binary Search Tree (BST)

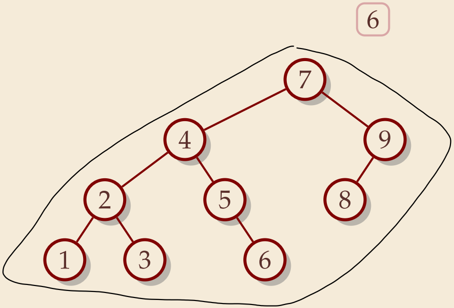


Quicksort & Binary Search Trees

Quicksort

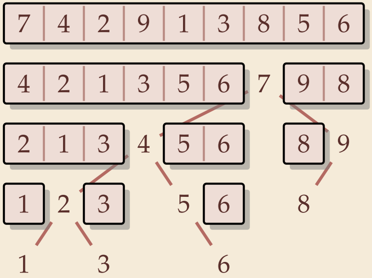


Binary Search Tree (BST)

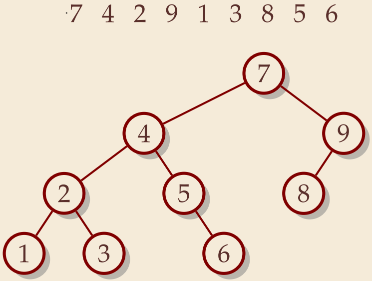


Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)



- recursion tree of quicksort $=$ binary search tree from successive insertion
- comparisons in quicksort $=$ comparisons to built BST
- comparisons in quicksort $\approx$ comparisons to search each element in BST

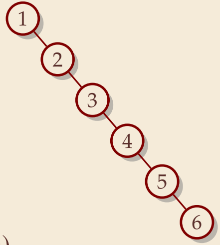
random perm.

$= C_n \sim 2n \ln n$

$\approx 1.39\, n \lg n$

Quicksort – Worst Case

- Problem: BSTs can degenerate
- Cost to search for k is $k - 1$



$\rightsquigarrow$ Total cost $\displaystyle\sum_{k=1}^{n}(k-1) = \frac{n(n-1)}{2} \sim \frac{1}{2}n^2$

$\rightsquigarrow$ quicksort worst-case running time is in $\Theta(n^2)$

terribly slow!

average case $1.39\, n \lg n$

But, we can fix this:

Randomized quicksort:

- choose a *random pivot* in each step

$\rightsquigarrow$ same as randomly *shuffling* input before sorting

Randomized Quicksort – Analysis

- cost measure: element visits (as for mergesort)
- $C(n)$ = #element visits when sorting n randomly permuted elements
  = cost of searching every element in BST built from input

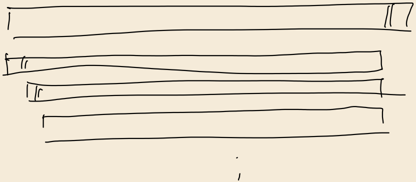
Randomized Quicksort – Analysis

- cost measure: element visits (as for mergesort)
- $C(n)$ = #element visits when sorting n randomly permuted elements
  = cost of searching every element in BST built from input

$\rightsquigarrow$ quicksort needs $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ *in expectation*
(see analysis of C_n in Unit 3!)

- also: very unlikely to be much worse:
  e. g., one can prove: $\Pr[\text{cost} > 10n \lg n] \;=\; O(n^{-2.5})$
  distribution of costs is "concentrated around mean"
- intuition: have to be *constantly* unlucky with pivot choice

$\notin$ exam



Quicksort – Discussion

mergesort quicksort

$2 n \lg n$ $1.39 n \lg n$ rand.

- fastest general-purpose method
- $\Theta(n \log n)$ average case #cmps = #element visits
- works *in-place* (no extra space required)
- memory access is sequential (scans over arrays)
- $\Theta(n^2)$ worst case (although extremely unlikely)
- not a *stable* sorting method

Open problem: Simple algorithm that is fast, stable and in-place.

4.3 Comparison-Based Lower Bound

Lower Bounds

- **Lower bound:** mathematical proof that *no algorithm* can do better.
  - very powerful concept: bulletproof *impossibility* result
    $\approx$ *conservation of energy* in physics
  - **(unique?) feature of computer science:**
    for many problems, solutions are known that (asymptotically) **achieve the lower bound**

$\rightsquigarrow$ can speak of "*optimal* algorithms"

Lower Bounds

- **Lower bound:** mathematical proof that *no algorithm* can do better.
  - very powerful concept: bulletproof *impossibility* result
    $\approx$ *conservation of energy* in physics
  - **(unique?) feature of computer science:**
    for many problems, solutions are known that (asymptotically) **achieve the lower bound**

  $\rightsquigarrow$ can speak of "*optimal* algorithms"

- To prove a statement about *all algorithms*, we must precisely define what that is!
- already know one option: the word-RAM model
- Here: use a simpler, more restricted model.

The Comparison Model

- In the *comparison model* data can only be accessed in two ways:
  - comparing two elements
  - moving elements around (e. g. copying, swapping)
  - Cost: number of comparisons.

The Comparison Model

expert note

cell probe model

- In the *comparison model* data can only be accessed in two ways:
  - comparing two elements
  - moving elements around (e. g. copying, swapping)
  - Cost: number of comparisons.

That's good!
Keeps algorithms general!

- This makes very few assumptions on the kind of objects we are sorting.
- Mergesort and Quicksort work in the comparison model.

The Comparison Model

- In the *comparison model* data can only be accessed in two ways:
  - comparing two elements
  - moving elements around (e. g. copying, swapping)
  - Cost: number of comparisons.

That's good!
Keeps algorithms general!

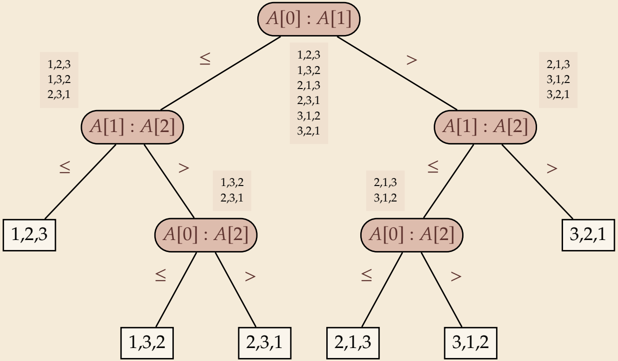
- This makes very few assumptions on the kind of objects we are sorting.
- Mergesort and Quicksort work in the comparison model.

$\rightsquigarrow$ Every comparison-based sorting algorithm corresponds to a ***decision tree***.

  - only model comparisons $\rightsquigarrow$ ignore data movement
  - nodes $=$ comparisons the algorithm does
  - child links $=$ outcomes of comparison
  - leaf $=$ unique initial input permutation compatible with comparison outcomes
  - next comparisons can depend on outcomes $\rightsquigarrow$ child subtrees can look different

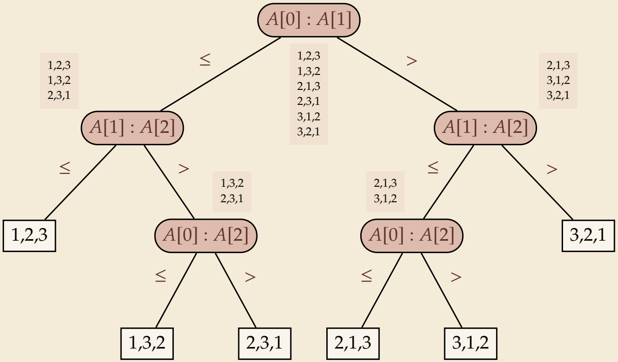
Comparison Lower Bound

Example: Comparison tree for a sorting method for $A[0..2]$:



Comparison Lower Bound

Example: Comparison tree for a sorting method for $A[0..2]$:



- ▶ Execution $=$ follow a path in comparison tree.
- $\rightsquigarrow$ height of comparison tree $=$ worst-case # comparisons
- ▶ comparison trees are *binary* trees
- $\rightsquigarrow$ ℓ leaves $\rightsquigarrow$ height $\geq \lceil \lg(\ell) \rceil$
- ▶ comparison trees for sorting method must have $\geq n!$ leaves
- $\rightsquigarrow$ $\boxed{\text{height} \geq \lg(n!)}$ $\sim n \lg n$

more precisely: $\lg(n!) = n \lg n - \lg(e)n + O(\log n)$

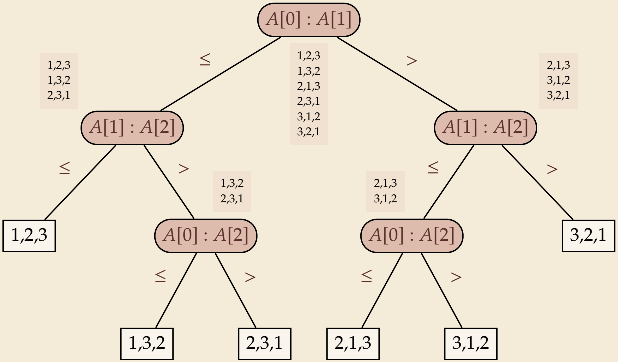
$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right).$$

$$\lg(n!) = \lg(\sqrt{2\pi n}) + n \cdot \lg\left(\frac{n}{e}\right) + O\left(\frac{1}{n}\right)$$

$$= n \lg n - n \cdot \lg e \pm O(\log n)$$

Comparison Lower Bound

Example: Comparison tree for a sorting method for $A[0..2]$:



- Execution = follow a path in comparison tree.
- ⇝ height of comparison tree = worst-case # comparisons
- comparison trees are *binary* trees
- ⇝ ℓ leaves ⇝ height $\ge \lceil \lg(\ell) \rceil$
- comparison trees for sorting method must have $\ge n!$ leaves
- ⇝ height $\ge \lg(n!) \sim n \lg n$

more precisely: $\lg(n!) = n \lg n - \lg(e) n + O(\log n)$

- Mergesort achieves $\sim n \lg n$ comparisons ⇝ asymptotically comparison-optimal!
- Open (theory) problem: Sorting algorithm with $n \lg n - \lg(e) n + o(n)$ comparisons?

≈ 1.4427

Clicker Question

Does the comparison-tree from the previous slide correspond to a worst-case optimal sorting method?

A Yes

B No

→ `sli.do/cs566`

Clicker Question

Does the comparison-tree from the previous slide correspond to a worst-case optimal sorting method?

A Yes ✓

B ~~No~~

→ sli.do/cs566

4.4 Integer Sorting

Clicker Question

Select all **correct formulations** of our **lower bound** from §4.3.

A Any sorting algorithm requires $O(n \log n)$ running time in the worst case.

B Every comparison-based sorting algorithm requires $\Omega(n \log n)$ running time in the worst case for sorting n elements.

C Every comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case for sorting n elements.

D Every sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case for sorting n elements.

E The complexity of sorting n elements in the comparison-model is $\Theta(n \log n)$.

F The complexity of sorting n elements in the comparison-model is $\Omega(n \log n)$.

→ sli.do/cs566

Clicker Question

Select all **correct formulations** of our **lower bound** from §4.3.

A ~~Any sorting algorithm requires $O(n \log n)$ running time in the worst case.~~

B Every comparison-based sorting algorithm requires $\Omega(n \log n)$ running time in the worst case for sorting n elements.

C Every comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case for sorting n elements.

D ~~Every sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case for sorting n elements.~~

E ~~The complexity of sorting n elements in the comparison model is $\Theta(n \log n)$.~~

F The complexity of sorting n elements in the comparison-model is $\Omega(n \log n)$.

→ sli.do/cs566

How to beat a lower bound

- Does the above lower bound mean, sorting always takes time $\Omega(n \log n)$?

How to beat a lower bound

- Does the above lower bound mean, sorting always takes time $\Omega(n \log n)$?

- **Not necessarily;** only in the *comparison model!*
  - $\rightsquigarrow$ Lower bounds show where to *change* the model!

How to beat a lower bound

- Does the above lower bound mean, sorting always takes time $\Omega(n \log n)$?

- **Not necessarily;** only in the *comparison model!*
  - $\rightsquigarrow$ Lower bounds show where to *change* the model!

- Here: sort n **integers**
  - can do *a lot* with integers: add them up, compute averages, ... (full power of word-RAM)
  - $\rightsquigarrow$ we are **not** working in the comparison model
  - $\rightsquigarrow$ *above lower bound does not apply!*

How to beat a lower bound

- Does the above lower bound mean, sorting always takes time $\Omega(n \log n)$?

- **Not necessarily;** only in the *comparison model!*
  $\rightsquigarrow$ Lower bounds show where to *change* the model!

- Here: sort n **integers**
  - can do *a lot* with integers: add them up, compute averages, ... (full power of word-RAM)

  $\rightsquigarrow$ we are **not** working in the comparison model
  $\rightsquigarrow$ *above lower bound does not apply!*

  - but: a priori unclear how much arithmetic helps for sorting ...

Counting sort

- Important parameter: size/range of numbers
  - numbers in range $[0..U) = \{0, \ldots, U-1\}$ typically $U = 2^b$ $\rightsquigarrow$ b-bit binary numbers

Counting sort

- Important parameter: size/range of numbers
  - numbers in range $[0..U) = \{0, \ldots, U-1\}$ typically $U = 2^b$ $\rightsquigarrow$ b-bit binary numbers
- We can sort n integers in $\Theta(n+U)$ time and $\Theta(U)$ space when $\boxed{b \le w}$: $\rightsquigarrow$ $U \le 2^w$

word size

Counting sort

procedure countingSort($A[0..n)$):
 // *A contains integers in range $[0..U)$.*
 $C[0..U)$:= new integer array, initialized to 0
 // *Count occurrences*
 for $i := 0, \ldots, n-1$
  $C[A[i]] := C[A[i]] + 1$ indirect addressing
 $i := 0$ // *Produce sorted list*
 for $k := 0, \ldots U-1$
  for $j := 1, \ldots C[k]$
   $A[i] := k$; $i := i+1$

- *count* how often each *possible* value occurs
- produce sorted result directly from counts
- circumvents lower bound by using integers as array index / pointer offset

$\rightsquigarrow$ Can sort n integers in range $[0..U)$ with $\boxed{U = O(n)}$ in time and space $\Theta(n)$.

Larger Universes: Radix Sort

- *MSD Radix Sort:*
  - split numbers into base-R "digits"
  - Use counting sort on <u>m</u>ost <u>s</u>ignificant <u>d</u>igit
    (with variant of counting sort that moves full number)

  ⇝ integers sorted with respect to first digit
  - recurse on sublist for each digit value, using next digit for counting sort

⇝ After $\lfloor \log_R(U) \rfloor + 1$ levels of counting sort, fully sorted!

- For $R \le 2^w$, all counting sort calls on same level cost total of $O(n)$ time
  (requires care to avoid reinitialization cost of array C)

⇝ total time $O(n \log_R(U)) = O\left(n \frac{\log(U)}{\log(R)}\right)$

⇝ $O(n)$ time sorting possible for numbers in range $U = O(n^c)$ for constant c.

Integer Sorting – State of the art

Algorithm theory

- integer sorting on the w-bit word-RAM
- suppose $U = 2^w$, but w can be an arbitrary function of n
- how fast can we sort n such w-bit integers on a w-bit word-RAM?
  - for $w = O(\log n)$: linear time (*radix/counting sort*) // standard assumption
  - for $w = \Omega(\log^{2+\varepsilon} n)$: linear time (*signature sort*)
  - for w in between: can do $O(n\sqrt{\lg\lg n})$ (very complicated algorithm)
    don't know if that is best possible!

$\notin$ exam

Integer Sorting – State of the art

Algorithm theory

- integer sorting on the w-bit word-RAM
- suppose $U = 2^w$, but w can be an arbitrary function of n
- how fast can we sort n such w-bit integers on a w-bit word-RAM?
  - for $w = O(\log n)$: linear time (*radix/counting sort*)
  - for $w = \Omega(\log^{2+\varepsilon} n)$: linear time (*signature sort*)
  - for w in between: can do $O(n\sqrt{\lg \lg n})$ (very complicated algorithm)
    don't know if that is best possible!

* * *

... for the rest of this unit: back to the comparisons model!

Clicker Question

Which statements are correct? Select all that apply.

My computer has 64-bit words, so an `int` has 64 bits. Hence I can sort any `int[]` of length n ...

- **A** in constant time.
- **B** in $O(\log n)$ time.
- **C** in $O(n)$ time.
- **D** in $O(n \log n)$ time.
- **E** some time, but not possible to say from given information.

→ sli.do/cs566

Clicker Question

Which statements are correct? Select all that apply.

My computer has 64-bit words, so an `int` has 64 bits. Hence I can sort any `int[]` of length n . . .

- **A** ~~in constant time.~~
- **B** ~~in $O(\log n)$ time.~~
- **C** in $O(n)$ time. ✓
- **D** in $O(n \log n)$ time. ✓
- **E** some time, but not possible to say from given information. ✓

→ sli.do/cs566

Part II

Exploiting presortedness

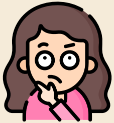
4.5 Adaptive Sorting

Adaptive sorting

- Comparison lower bound also holds for the *average case* $\rightsquigarrow$ $\lfloor \lg(n!) \rfloor$ cmps necessary
- Mergesort and Quicksort from above use $\sim n \lg n$ cmps even in best case

Adaptive sorting

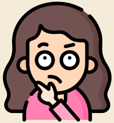
- Comparison lower bound also holds for the *average case* $\rightsquigarrow$ $\lfloor \lg(n!) \rfloor$ cmps necessary
- Mergesort and Quicksort from above use $\sim n \lg n$ cmps even in best case



Can we do better if the input is already "almost sorted"?

Adaptive sorting

- Comparison lower bound also holds for the *average case* $\rightsquigarrow$ $\lfloor \lg(n!) \rfloor$ cmps necessary
- Mergesort and Quicksort from above use $\sim n \lg n$ cmps even in best case



Can we do better if the input is already "almost sorted"?

Scenarios where this may arise naturally:

- Append new data as it arrives, regularly sort entire list (e. g., log files, database tables)
- Compute summary statistics of time series of measurements that change slowly over time (e. g., weather data)
- Merging locally sorted data from different servers (e. g., map-reduce frameworks)

$\rightsquigarrow$ Ideally, algorithms should *adapt* to input: *the more sorted the input, the faster the algorithm*

. . . but how to do that!?

Warmup: check for sorted inputs

- Any method could first check if input already completely in order!
  - 👍 Best case becomes $\Theta(n)$ with $n-1$ comparisons!
  - 👎 Usually $n-1$ extra comparisons and pass over data "wasted"
  - 👎 Only catches a single, extremely special case ...

Warmup: check for sorted inputs

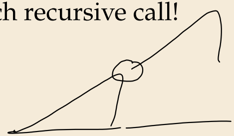
- Any method could first check if input already completely in order!
  - 👍 Best case becomes $\Theta(n)$ with $n-1$ comparisons!
  - 👎 Usually $n-1$ extra comparisons and pass over data "wasted"
  - 👎 Only catches a single, extremely special case ...

- For divide & conquer algorithms, could check in each recursive call!
  - 👍 Potentially exploits partial sortedness!
  - 👎 usually adds $\Omega(n \log n)$ extra comparisons

Warmup: check for sorted inputs

- Any method could first check if input already completely in order!
  - 👍 Best case becomes $\Theta(n)$ with $n - 1$ comparisons!
  - 👎 Usually $n - 1$ extra comparisons and pass over data "wasted"
  - 👎 Only catches a single, extremely special case . . .

- For divide & conquer algorithms, could check in each recursive call!
  - 👍 Potentially exploits partial sortedness!
  - 👎 usually adds $\Omega(n \log n)$ extra comparisons



For Mergesort, can instead check before merge with a **single** comparison

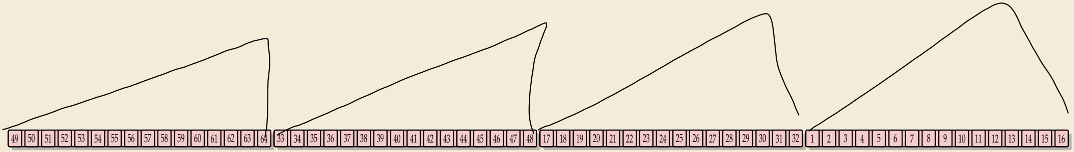
- If last element of first run $\le$ first element of second run, skip merge

How effective is this idea?

```
procedure mergesortCheck(A[l..r)):
    n := r − l
    if n ≤ 1 return
    m := l + ⌊n/2⌋
    mergesortCheck(A[l..m))
    mergesortCheck(A[m..r))
    if A[m − 1] > A[m]
        merge(A[l..m), A[m..r), buf)
        copy buf to A[l..r)
```

Mergesort with sorted check – Analysis

- Simplified cost measure: *merge cost* = size of output of merges
  ≈ number of comparisons
  ≈ number of memory transfers / cache misses
- Example input: $n = 64$ numbers in sorted *runs* of 16 numbers each:



Mergesort with sorted check – Analysis

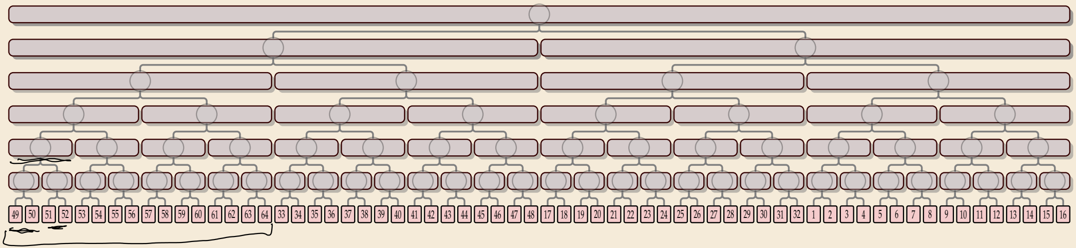
- Simplified cost measure: *merge cost* = size of output of merges
  $\approx$ number of comparisons
  $\approx$ number of memory transfers / cache misses
- Example input: $n = 64$ numbers in sorted *runs* of 16 numbers each:

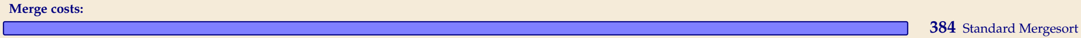
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Mergesort with sorted check – Analysis

- Simplified cost measure: *merge cost* $=$ size of output of merges
  $\approx$ number of comparisons
  $\approx$ number of memory transfers / cache misses

- Example input: $n = 64$ numbers in sorted *runs* of 16 numbers each:

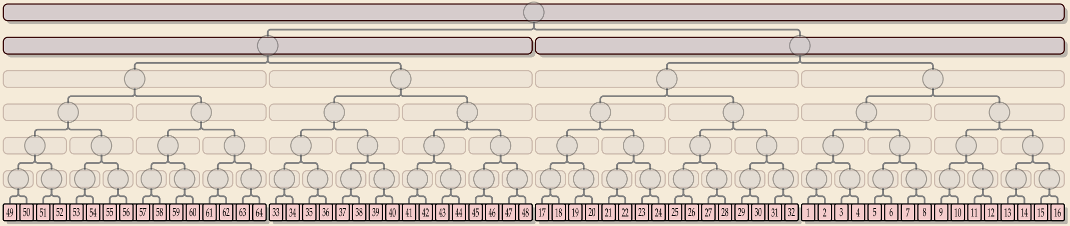


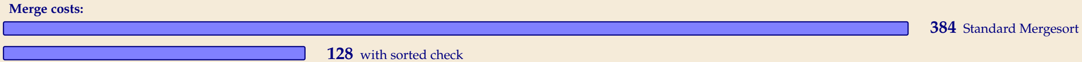


Mergesort with sorted check – Analysis

- Simplified cost measure: *merge cost* = size of output of merges
  ≈ number of comparisons
  ≈ number of memory transfers / cache misses

- Example input: $n = 64$ numbers in sorted *runs* of 16 numbers each:





Sorted check can help a lot!