

3

Efficient Sorting

24 February 2021

Sebastian Wild

Outline

3 Efficient Sorting

3.1 Mergesort

3.2 Quicksort

3.3 Comparison-Based Lower Bound

3.4 Integer Sorting

3.5 Parallel computation

3.6 Parallel primitives

3.7 Parallel sorting

Why study sorting?

- ▶ fundamental problem of computer science that is still not solved
- ▶ building brick of many more advanced algorithms
 - ▶ for preprocessing
 - ▶ as subroutine
- ▶ playground of manageable complexity to practice algorithmic techniques

Algorithm with optimal #comparisons in worst case?



Here:

- ▶ “classic” fast sorting method
- ▶ **parallel** sorting

Part I

The Basics

Rules of the game

- ▶ Given:

- ▶ array $A[0..n - 1]$ of n objects
- ▶ a total order relation \leq among $A[0], \dots, A[n - 1]$
(a comparison function) *Comparator (Java)*

$x.compareTo(y) \leq 0$
↑
 $x \leq y$

- ▶ Goal: rearrange (=permute) elements within A ,
so that A is *sorted*, i. e., $A[0] \leq A[1] \leq \dots \leq A[n - 1]$

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

Clicker Question



What is the complexity of sorting? Type your answer, e.g., as
“Theta(sqrt(n))”

sli.do/comp526

Click on “Polls” tab

3.1 Mergesort

Clicker Question

How does mergesort work?



- A** Split elements around median, then recurse on small / large elements.
- B** Recurse on left / right half, then combine sorted halves.
- C** Grow sorted part on left, repeatedly add next element to sorted range.
- D** Repeatedly choose 2 elements and swap them if they are out of order.
- E** Don't know.

sli.do/comp526

Click on “Polls” tab

Clicker Question

How does mergesort work?



- A** ~~Split elements around median, then recurse on small / large elements.~~
- B** Recurse on left / right half, then combine sorted halves. ✓
- C** ~~Grow sorted part on left, repeatedly add next element to sorted range.~~
- D** ~~Repeatedly choose 2 elements and swap them if they are out of order.~~
- E** ~~Don't know.~~

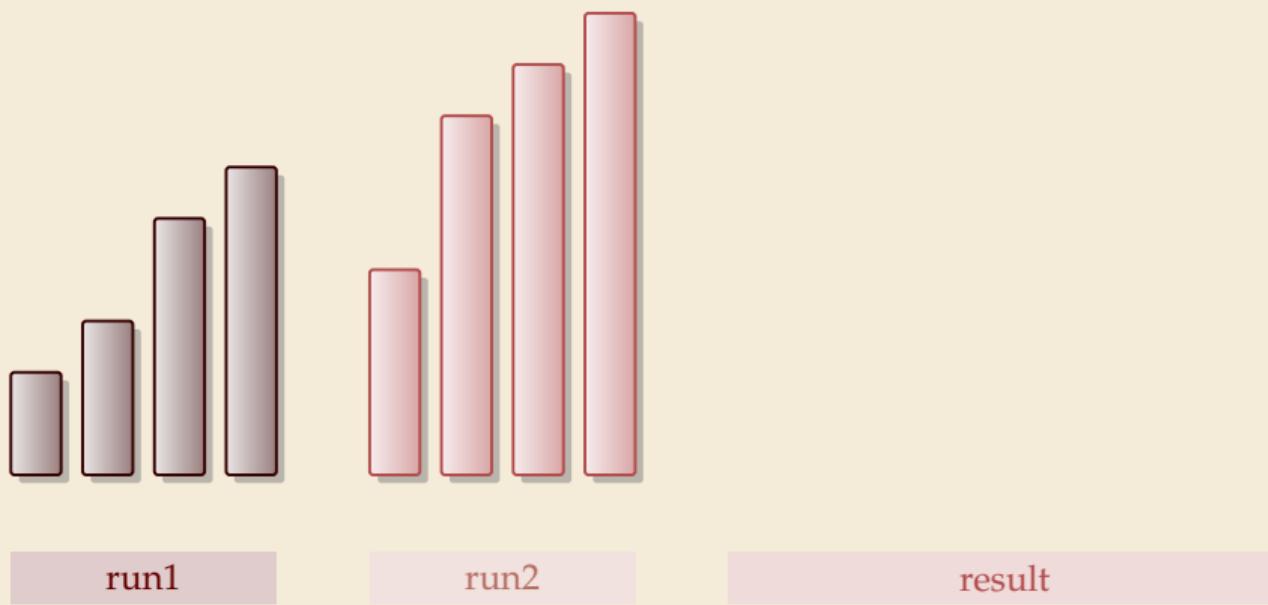
sli.do/comp526

Click on "Polls" tab

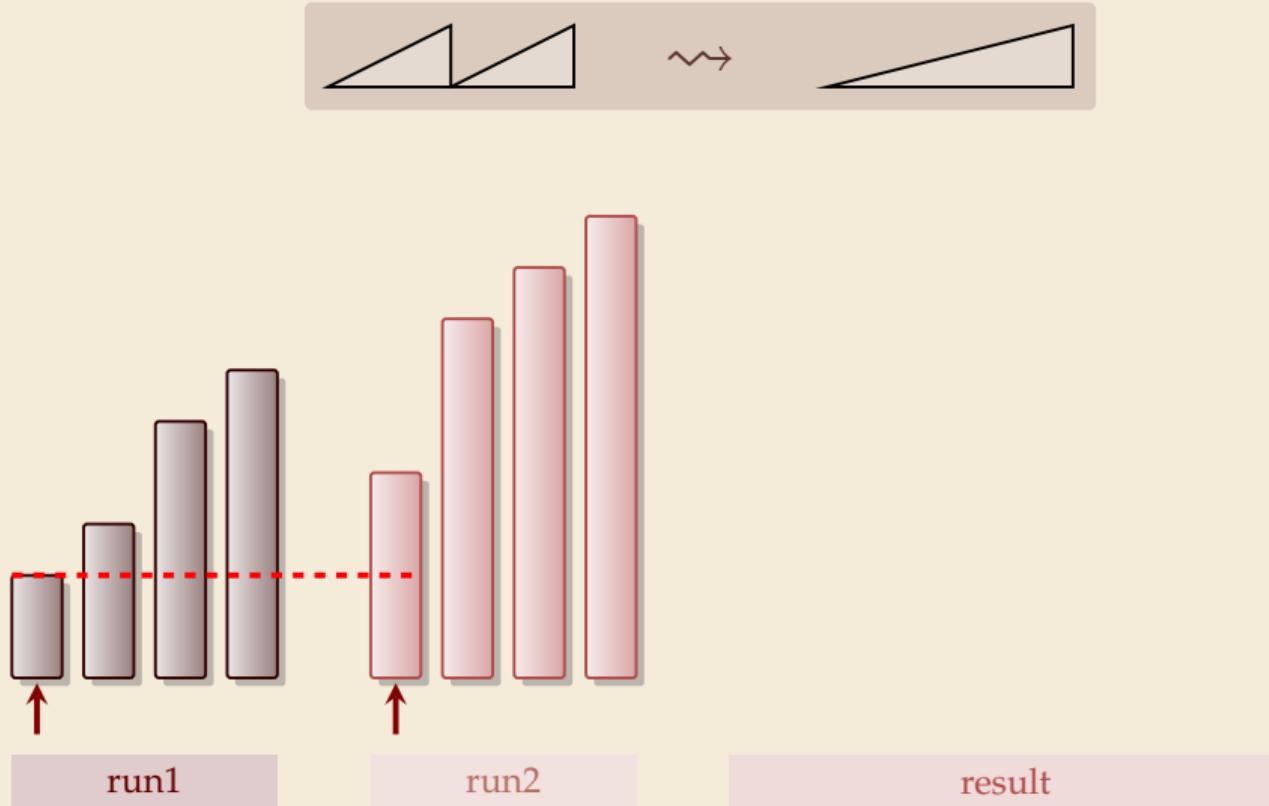
Merging sorted lists



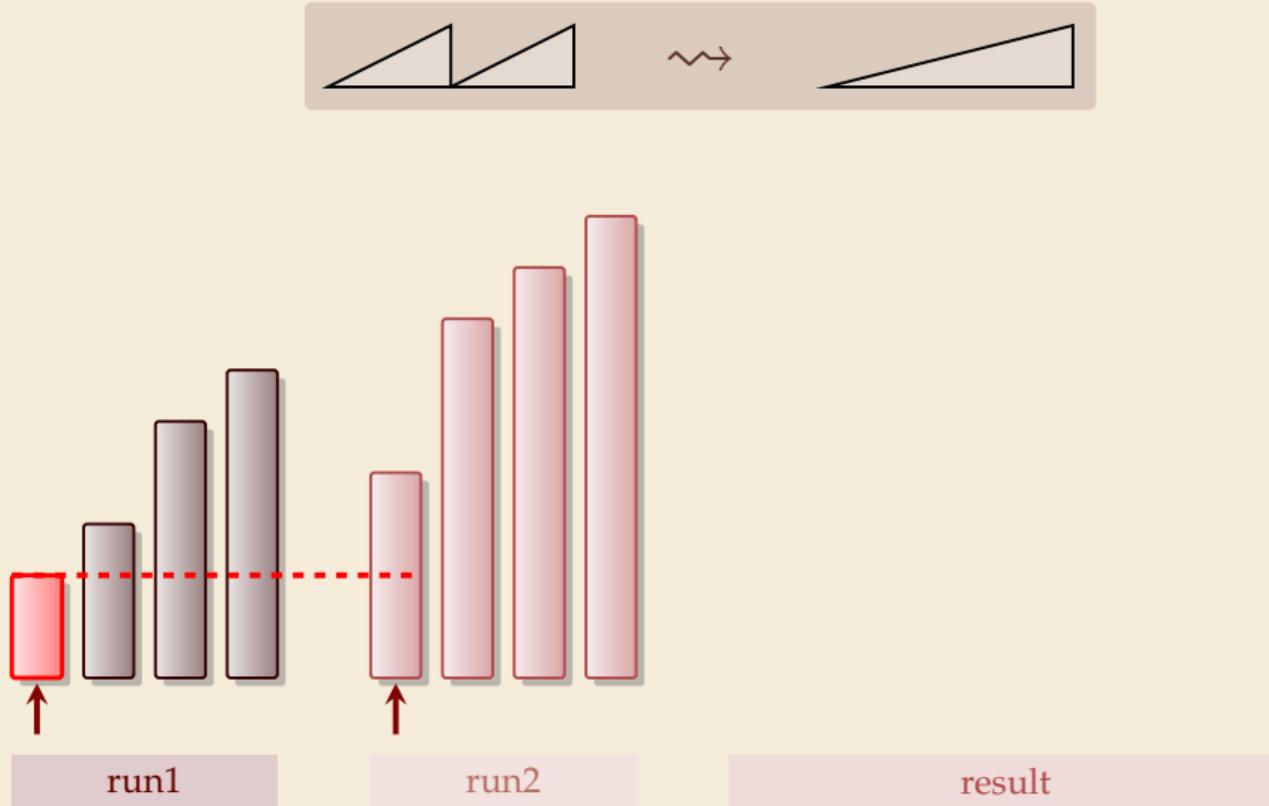
Merging sorted lists



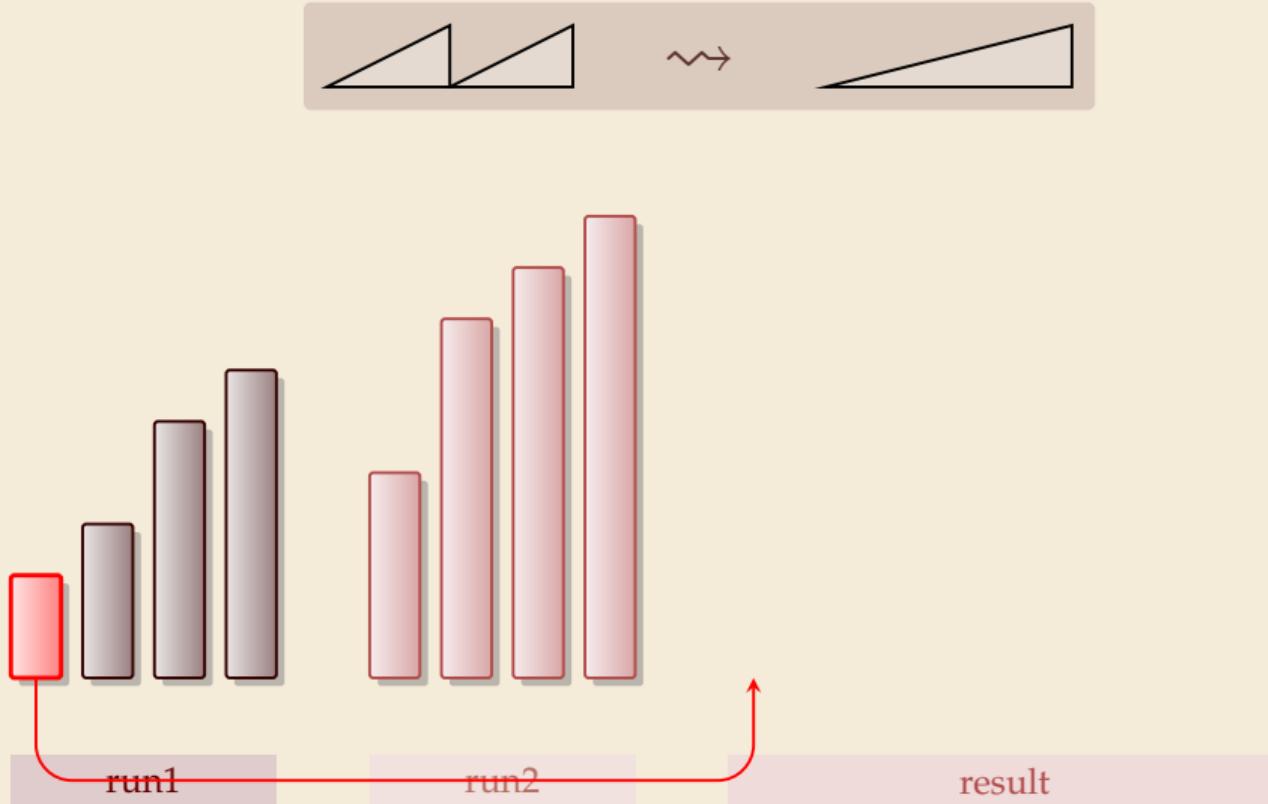
Merging sorted lists



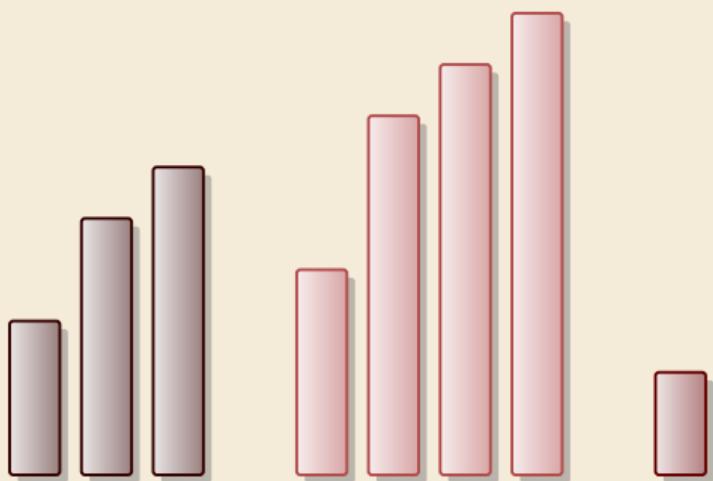
Merging sorted lists



Merging sorted lists



Merging sorted lists

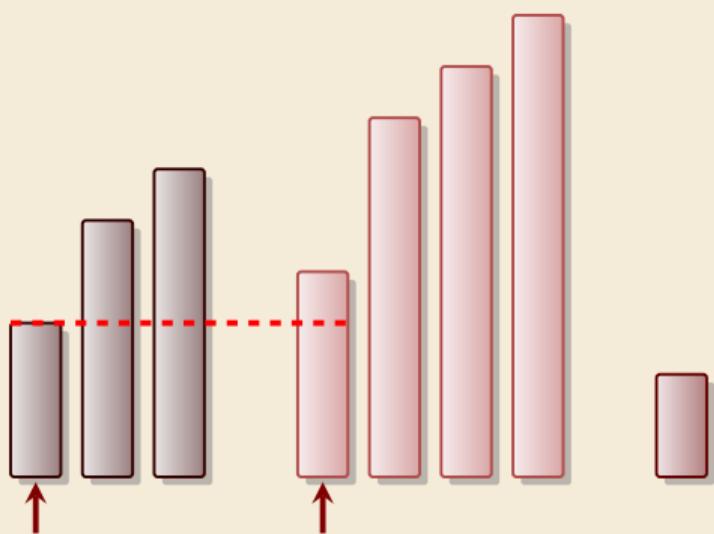


run1

run2

result

Merging sorted lists

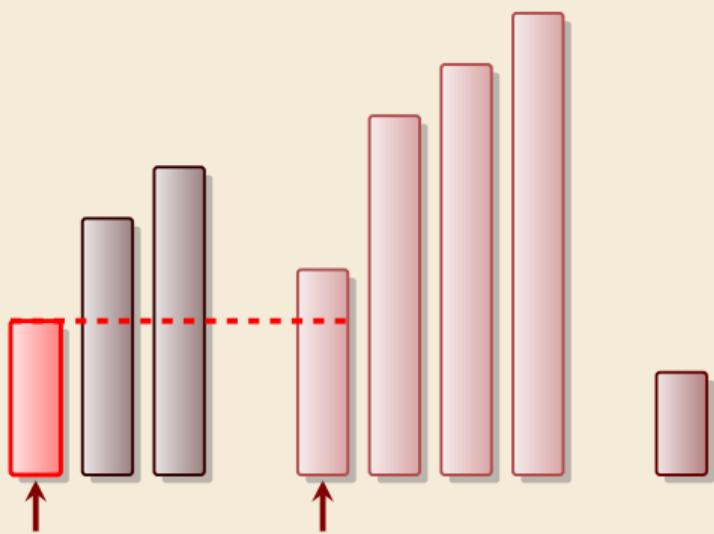


run1

run2

result

Merging sorted lists

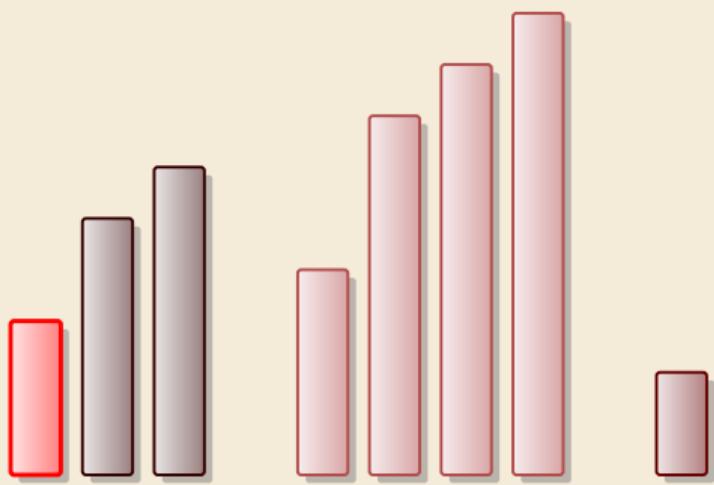


run1

run2

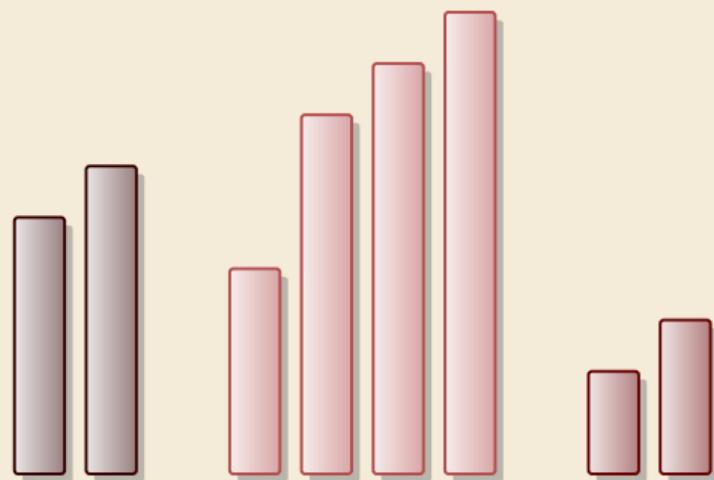
result

Merging sorted lists



run1 run2 result

Merging sorted lists

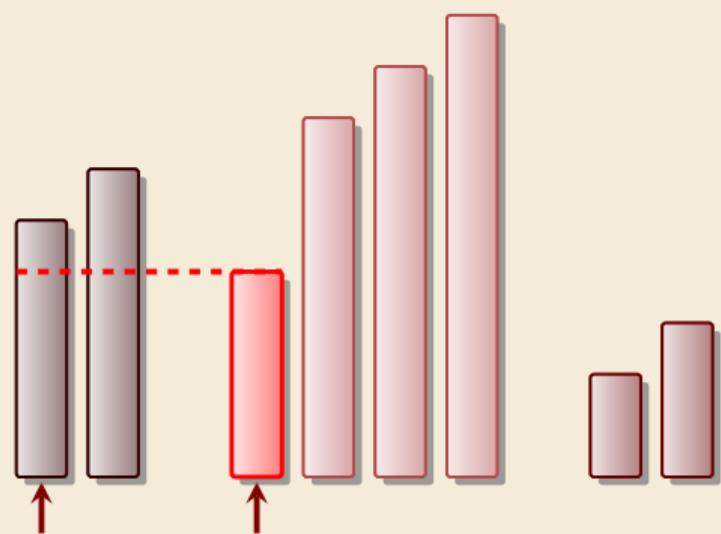


run1

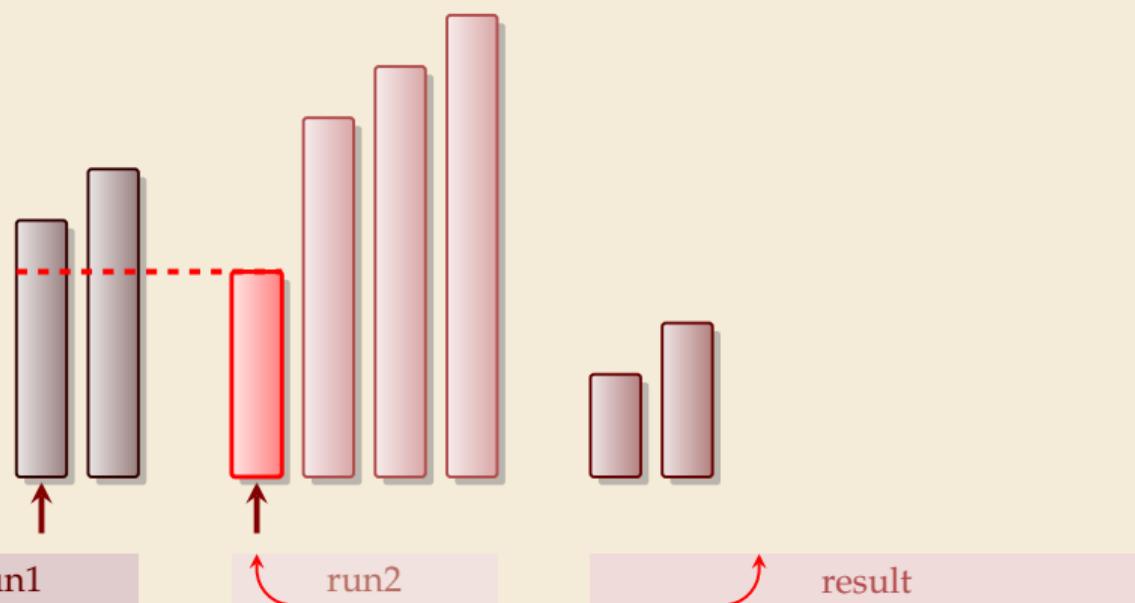
run2

result

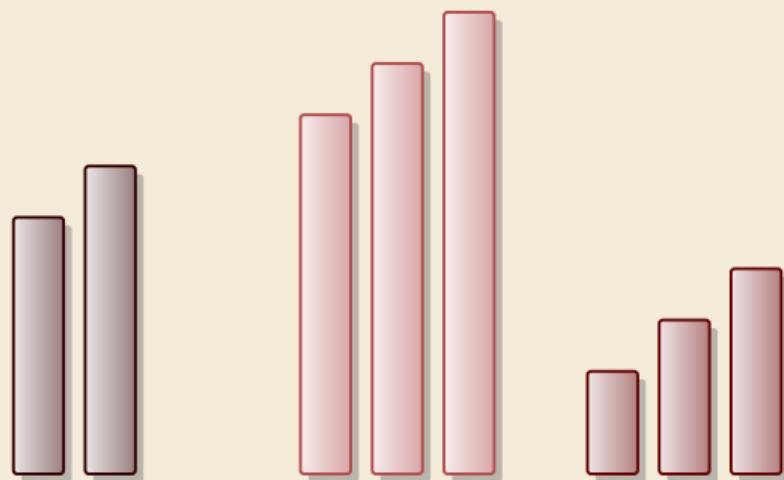
Merging sorted lists



Merging sorted lists



Merging sorted lists

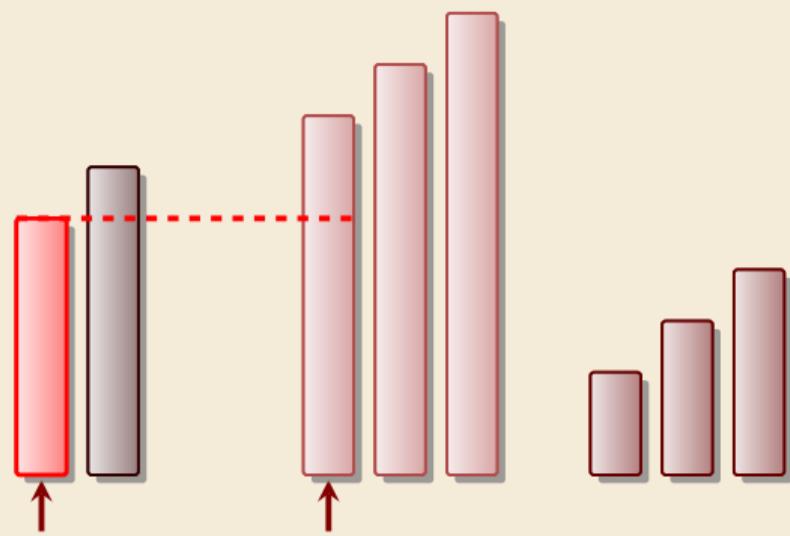


run1

run2

result

Merging sorted lists

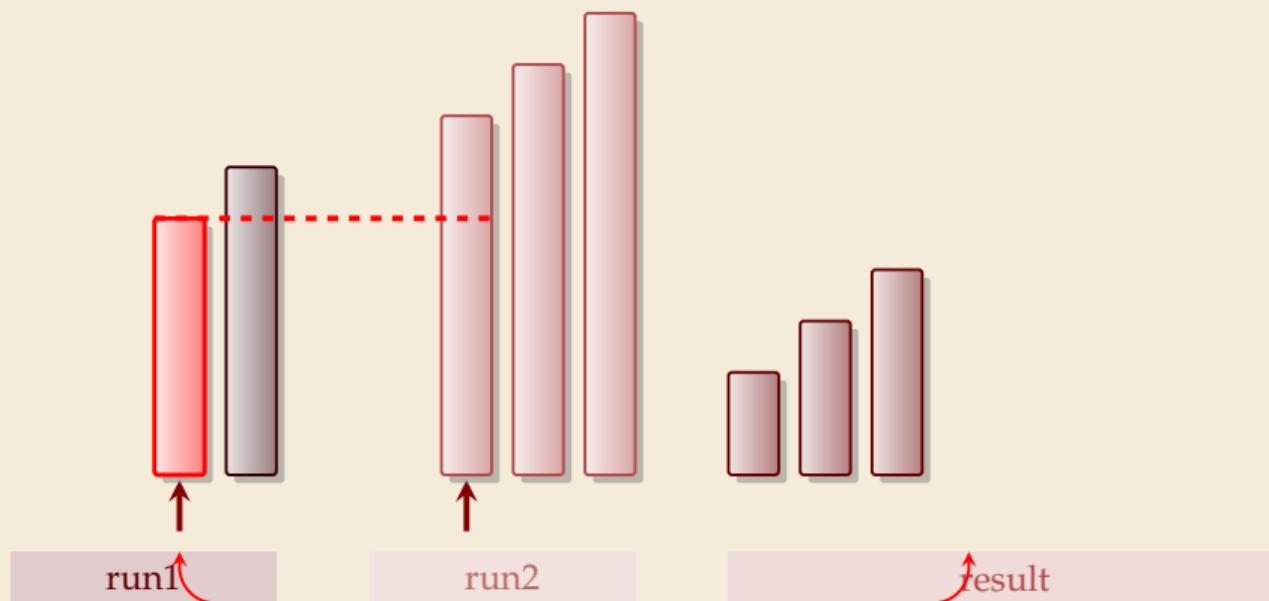


run1

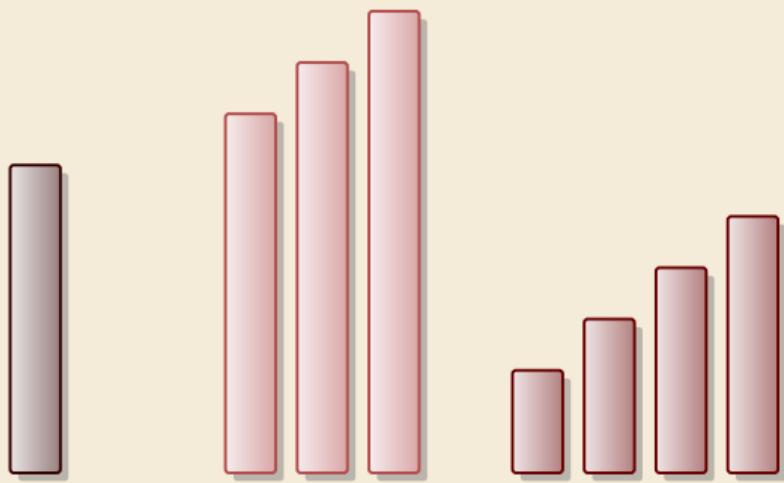
run2

result

Merging sorted lists



Merging sorted lists

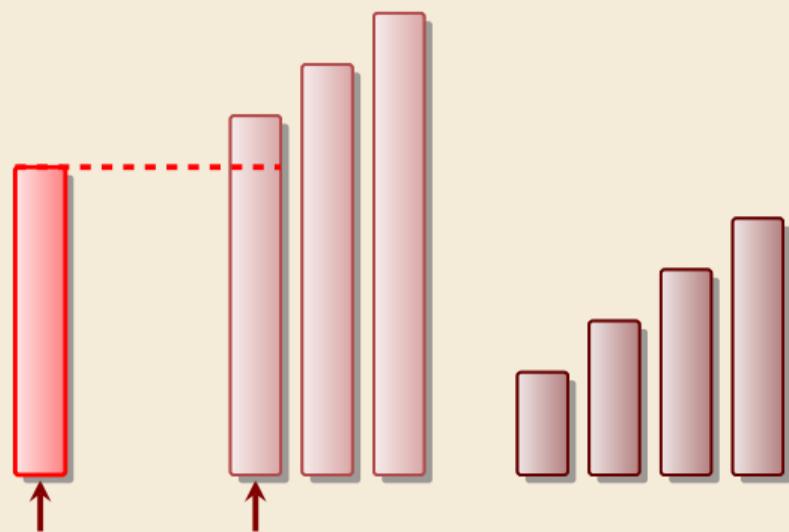


run1

run2

result

Merging sorted lists

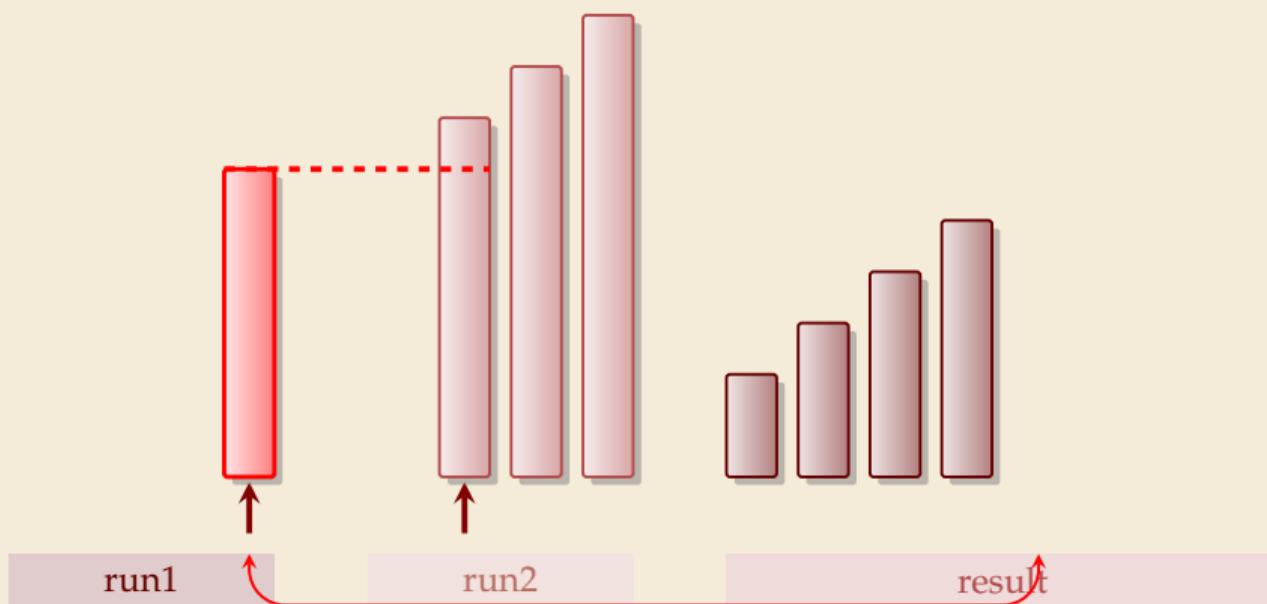


run1

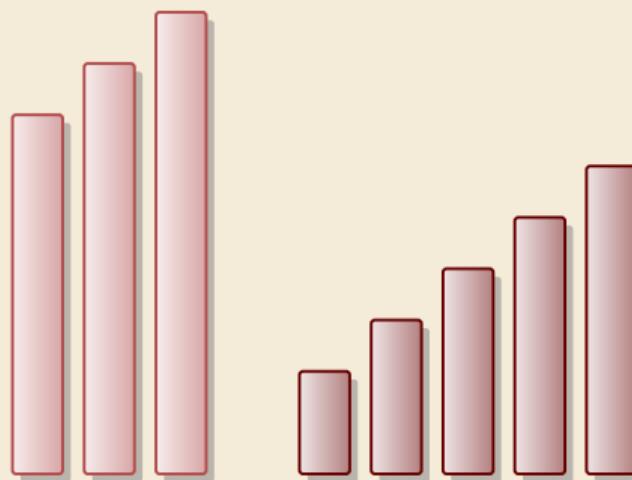
run2

result

Merging sorted lists



Merging sorted lists

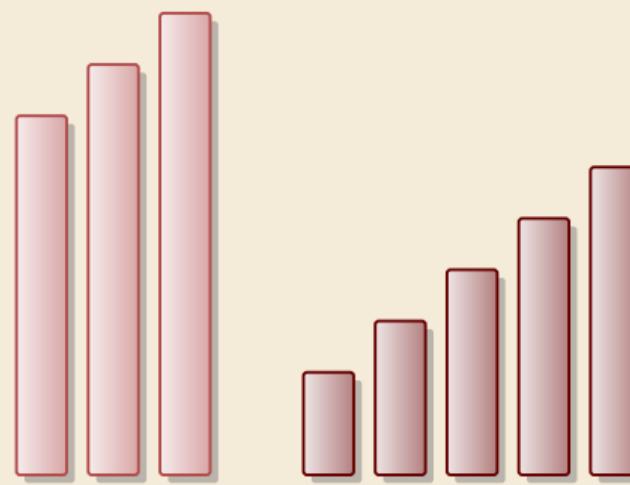


run1

run2

result

Merging sorted lists



run1

run2

result

Merging sorted lists

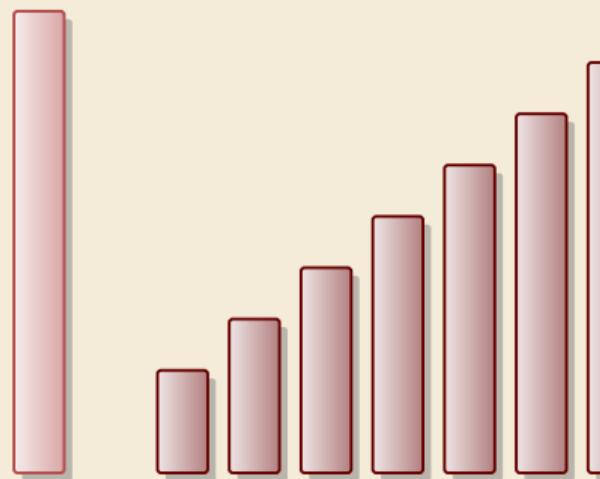


run1

run2

result

Merging sorted lists

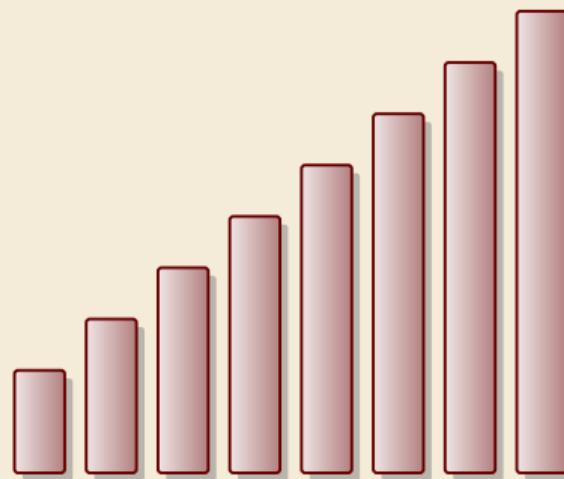


run1

run2

result

Merging sorted lists



run1

run2

result

Clicker Question

What is the worst-case running time of mergesort?



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/comp526

Click on “Polls” tab

Clicker Question

What is the worst-case running time of mergesort?



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/comp526

Click on “Polls” tab

Mergesort

```
1 procedure mergesort(A[l..r])
2     n := r - l + 1
3     if n ≥ 1 return
4         m := l + ⌊  $\frac{n}{2}$  ⌋
5         mergesort(A[l..m - 1])
6         mergesort(A[m..r])
7         merge(A[l..m - 1], A[m..r], buf)
8         copy buf to A[l..r]
```

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

Mergesort

```
1 procedure mergesort(A[l..r])
2     n := r - l + 1
3     if n ≥ 1 return
4         m := l + ⌊ n / 2 ⌋
5         mergesort(A[l..m - 1])
6         mergesort(A[m..r])
7         merge(A[l..m - 1], A[m..r], buf)
8         copy buf to A[l..r]
```

- ▶ recursive procedure; *divide & conquer*
- ▶ merging needs
 - ▶ temporary storage 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)

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

same for best and worst case!

Simplification $n = 2^k$ $k \in \mathbb{N}$ $k = \log n$ telescoping

$$C(2^k) = \begin{cases} 0 & k \leq 0 \\ 2 \cdot C(2^{k-1}) + 2 \cdot 2^k & k \geq 1 \end{cases}$$
$$= \underbrace{2 \cdot 2^k + 2^2 \cdot 2^{k-1} + 2^3 \cdot 2^{k-2} + \dots + 2^k \cdot 2^1}_{k \text{ sumands}} = \frac{2k \cdot 2^k}{2^{k+1}}$$
$$C(n) = 2n \lg(n) = \Theta(n \log n)$$

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

3.2 Quicksort

Clicker Question

How does quicksort work?



- A** split elements around median, then recurse on small / large elements.
- B** recurse on left / right half, then combine sorted halves.
- C** grow sorted part on left, repeatedly add next element to sorted range.
- D** repeatedly choose 2 elements and swap them if they are out of order.
- E** Don't know.

sli.do/comp526

Click on “Polls” tab

Clicker Question

How does quicksort work?



- A** split elements around median, then recurse on small / large elements. ✓
- B** ~~recurse on left / right half, then combine sorted halves.~~
- C** ~~grow sorted part on left, repeatedly add next element to sorted range.~~
- D** ~~repeatedly choose 2 elements and swap them if they are out of order.~~
- E** ~~Don't know.~~

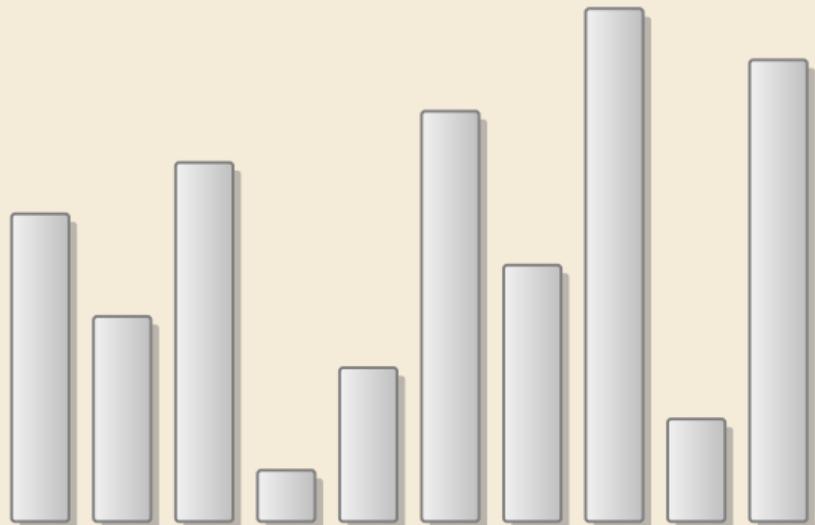
sli.do/comp526

Click on "Polls" tab

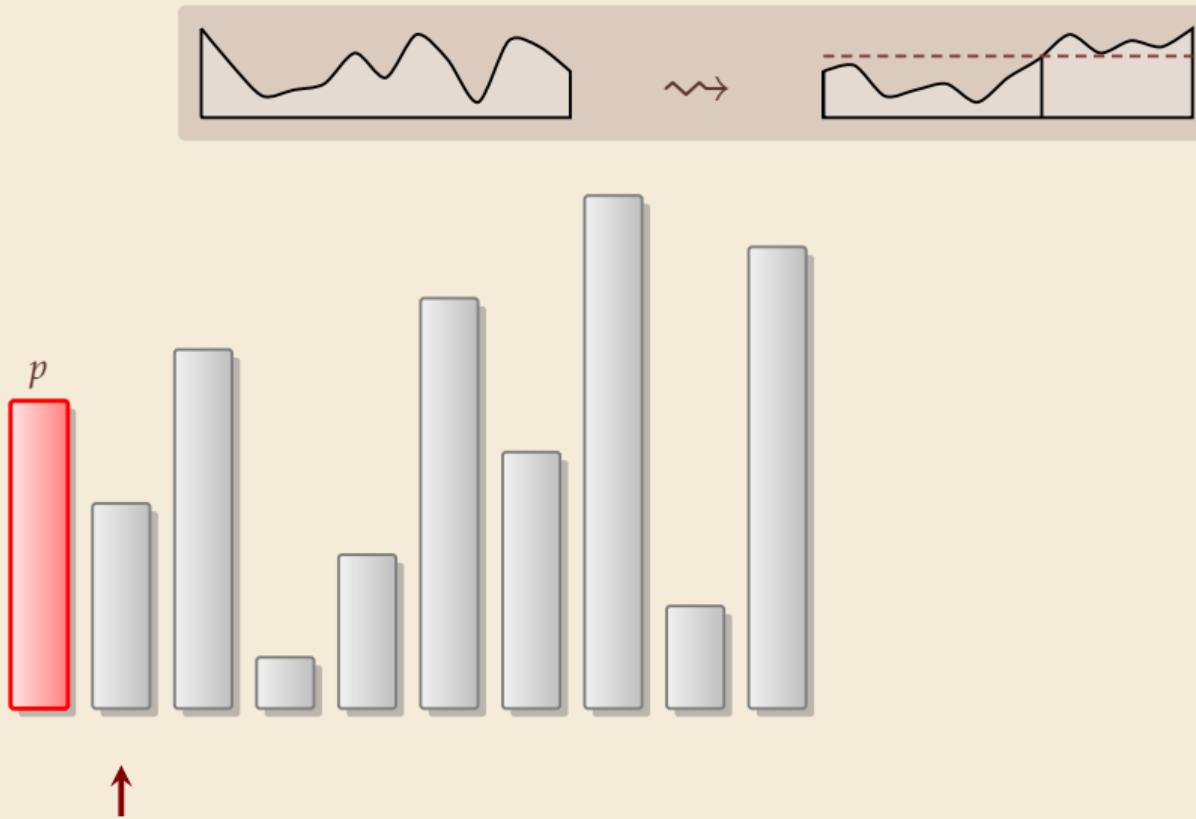
Partitioning around a pivot



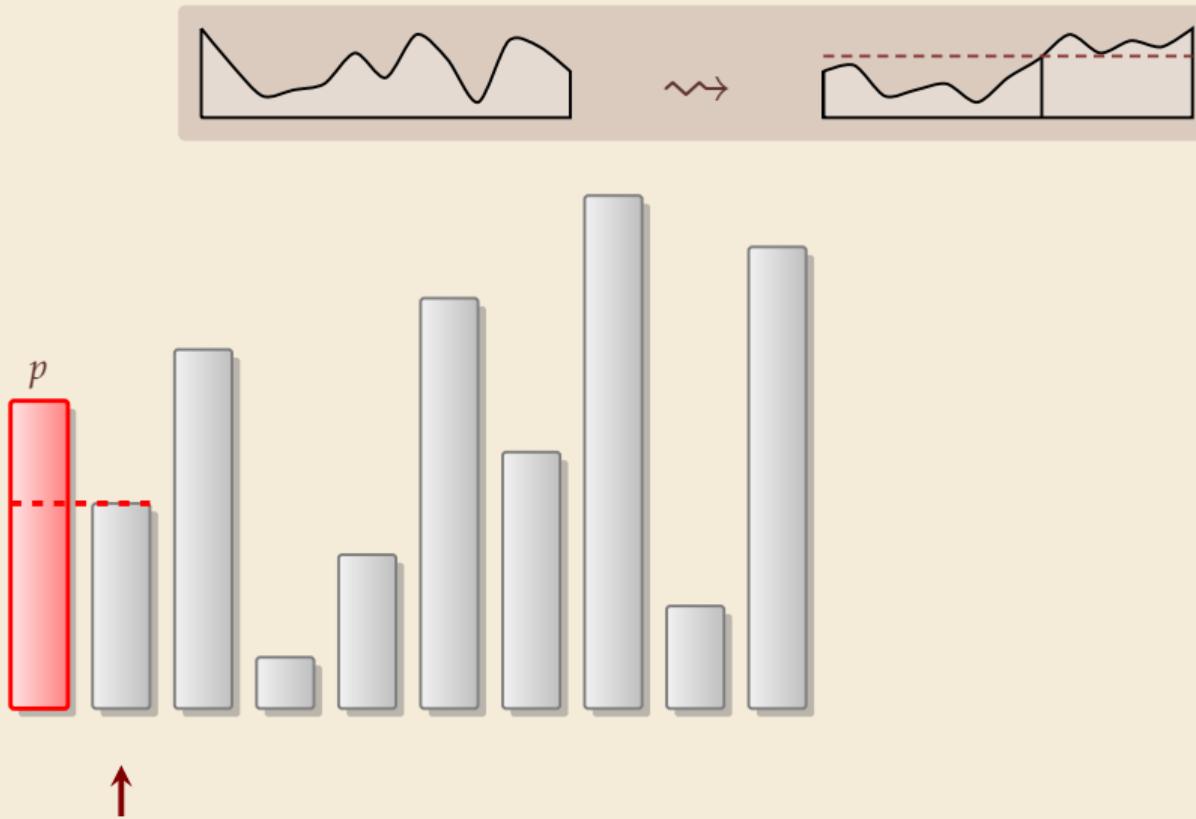
Partitioning around a pivot



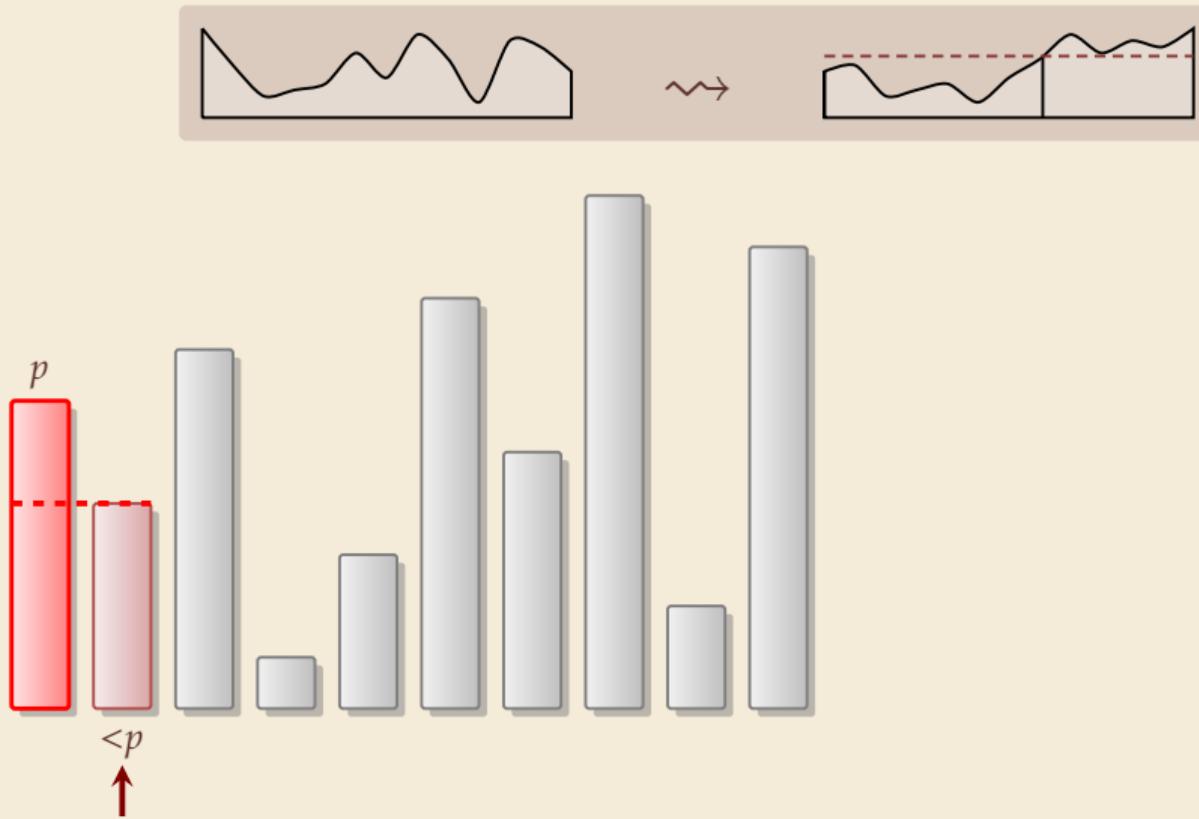
Partitioning around a pivot



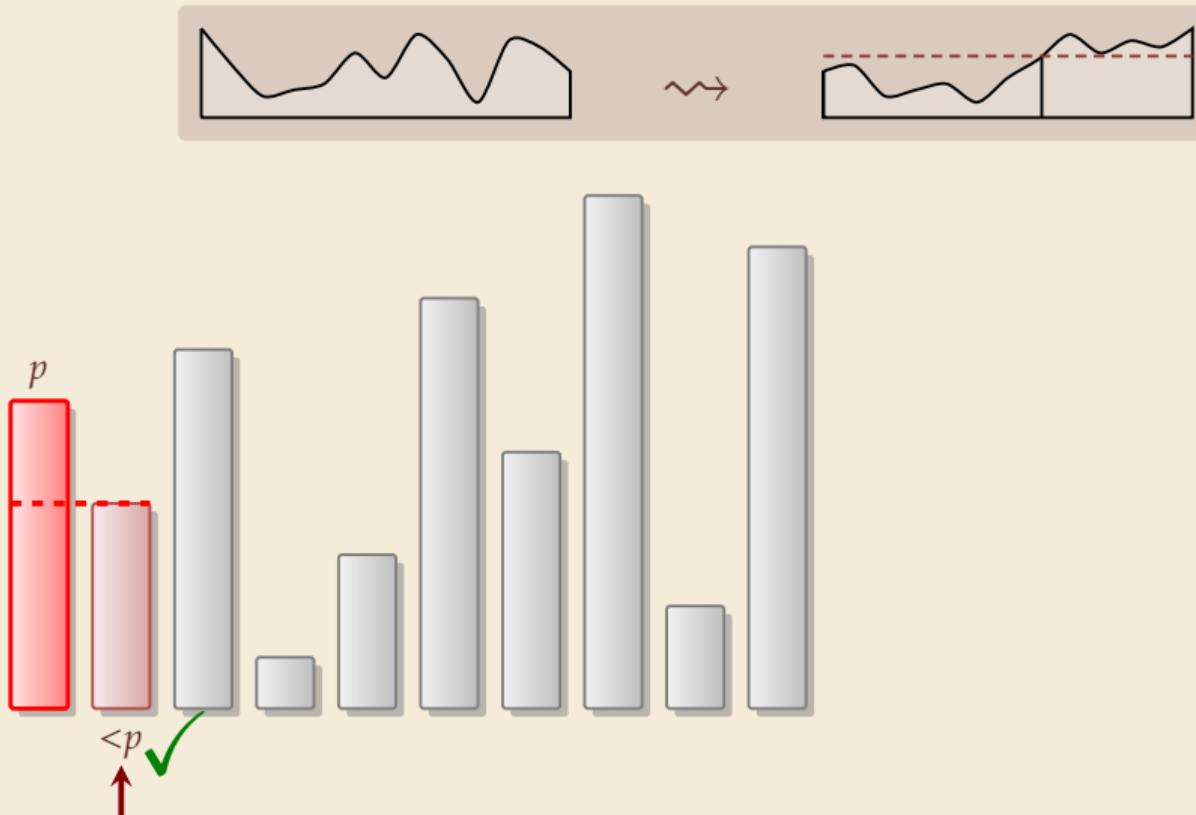
Partitioning around a pivot



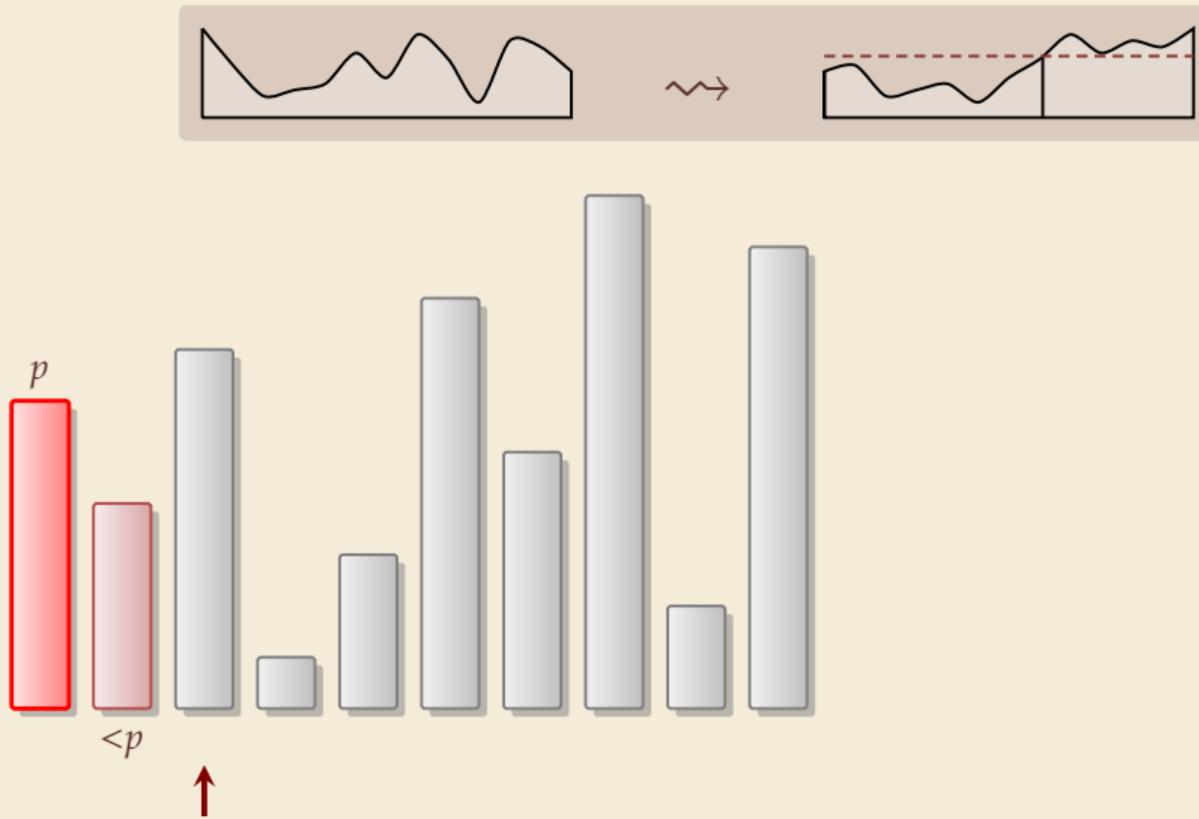
Partitioning around a pivot



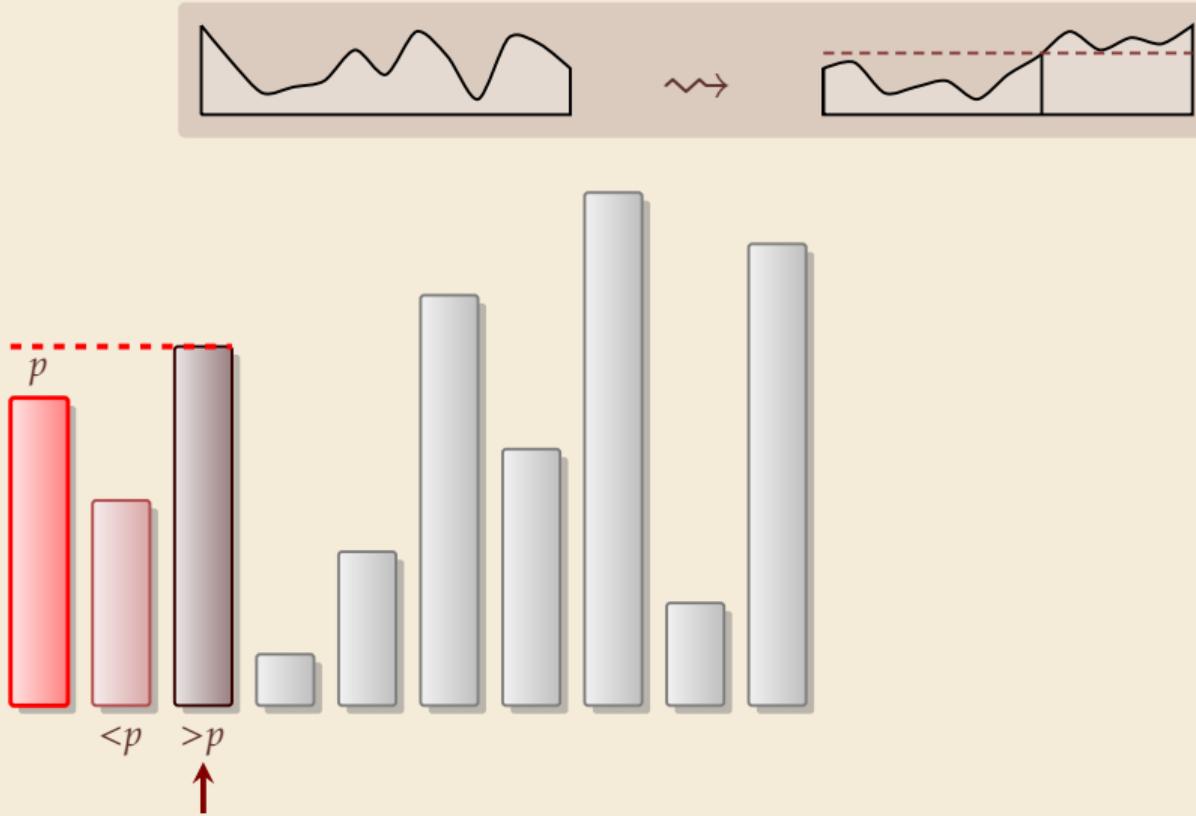
Partitioning around a pivot



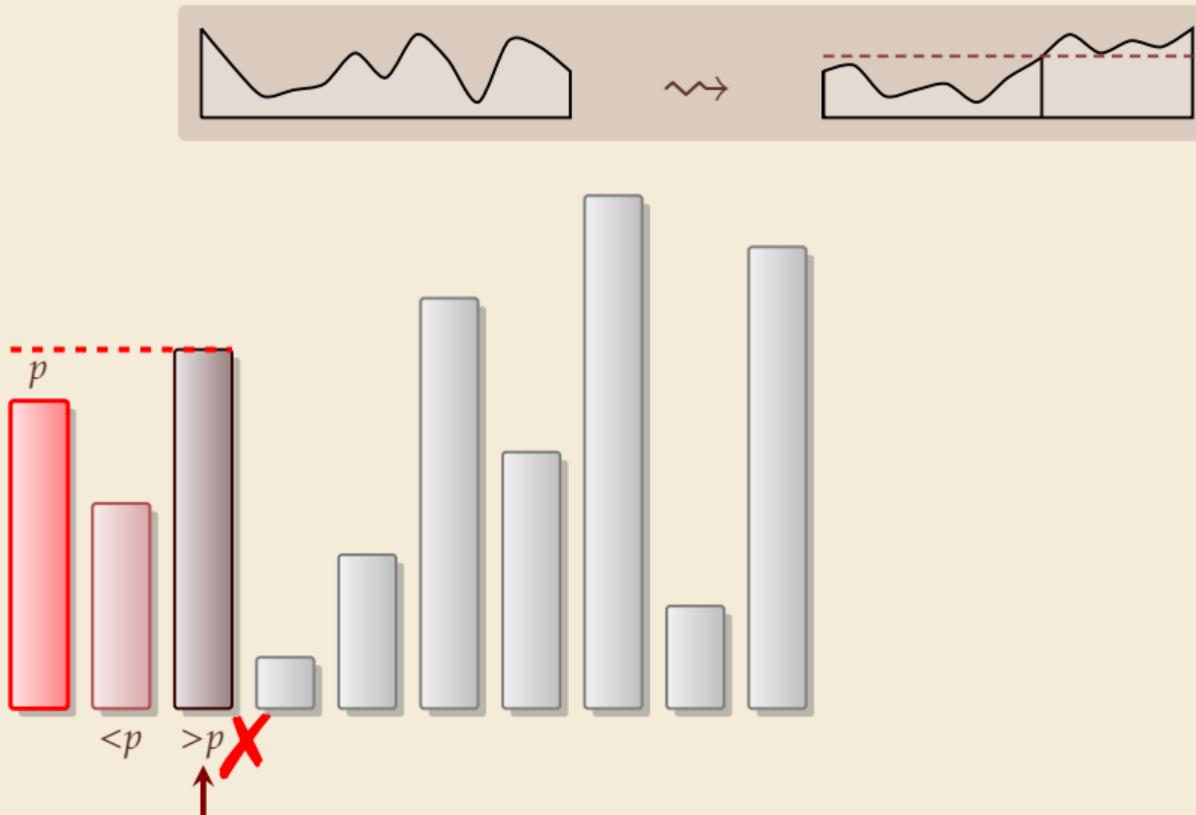
Partitioning around a pivot



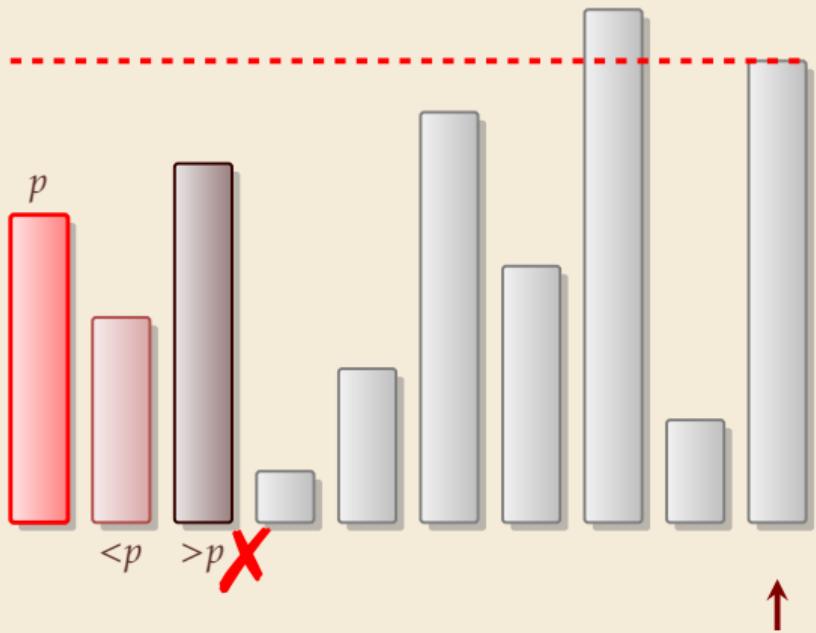
Partitioning around a pivot



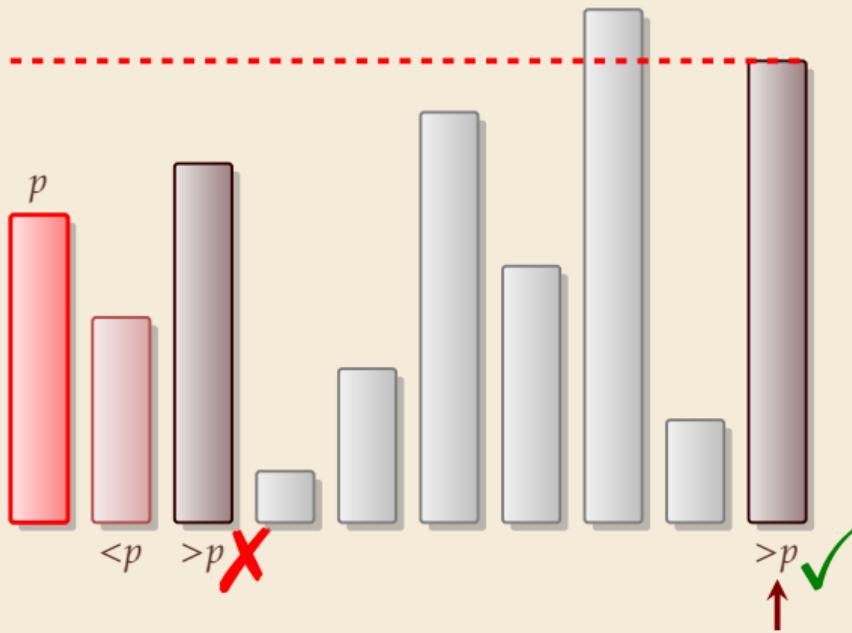
Partitioning around a pivot



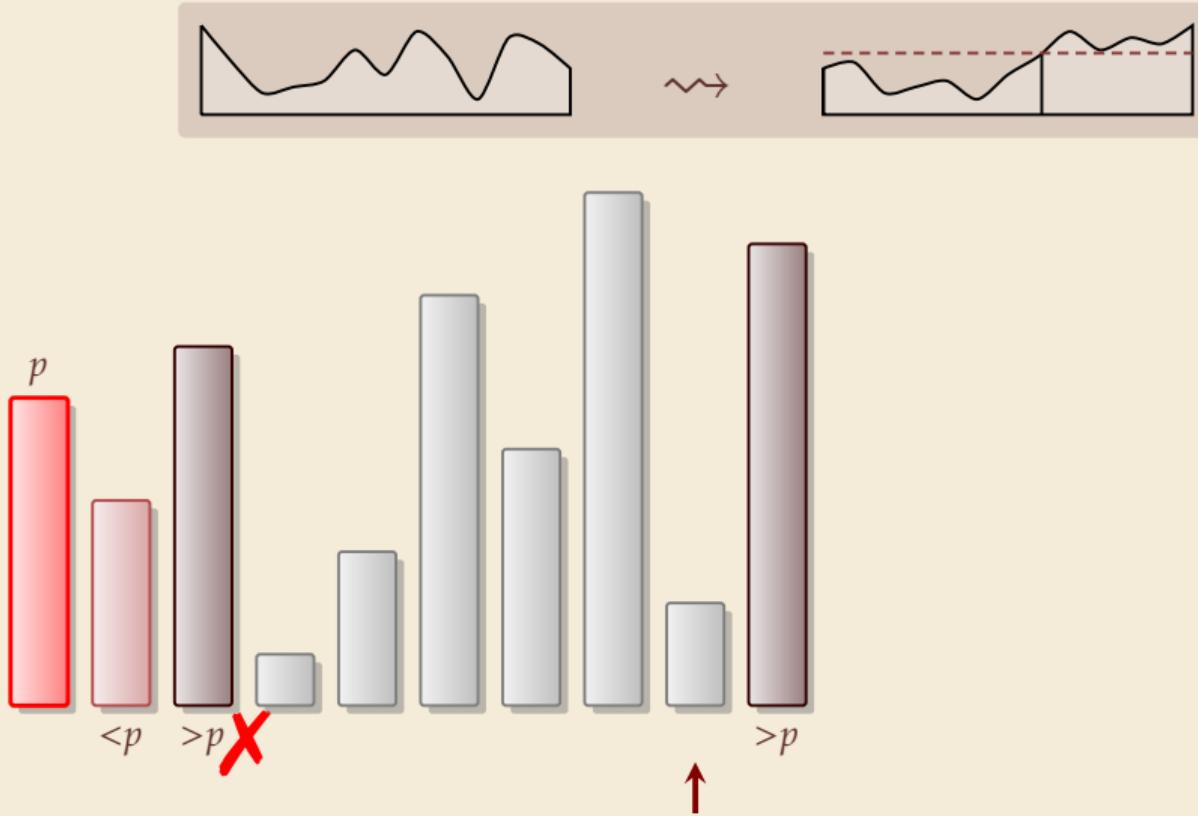
Partitioning around a pivot



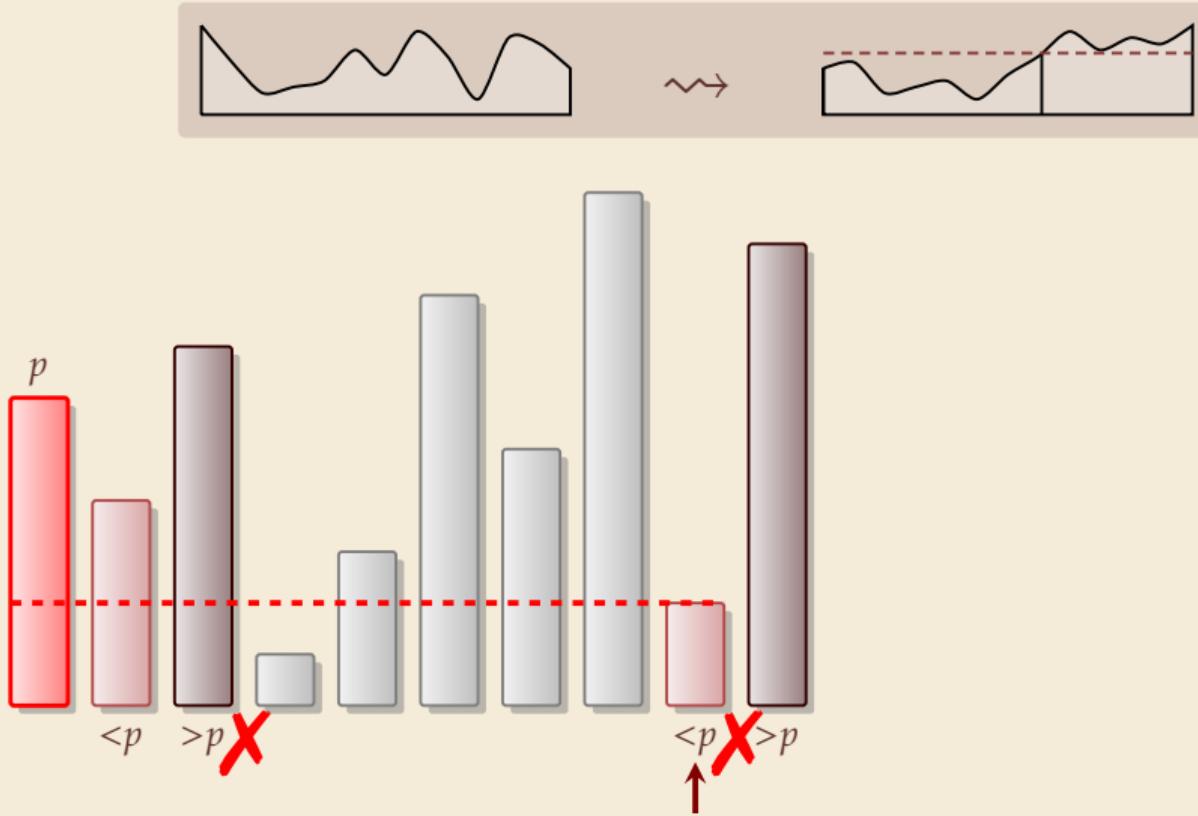
Partitioning around a pivot



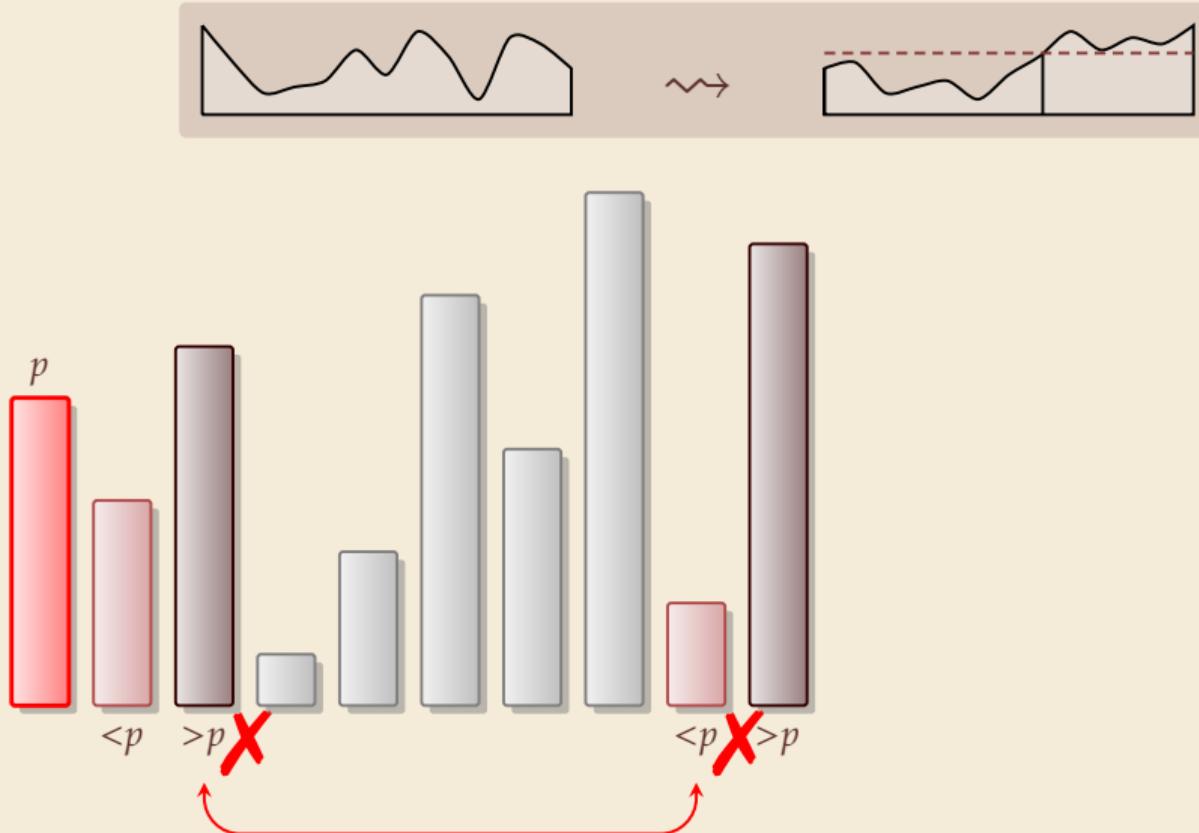
Partitioning around a pivot



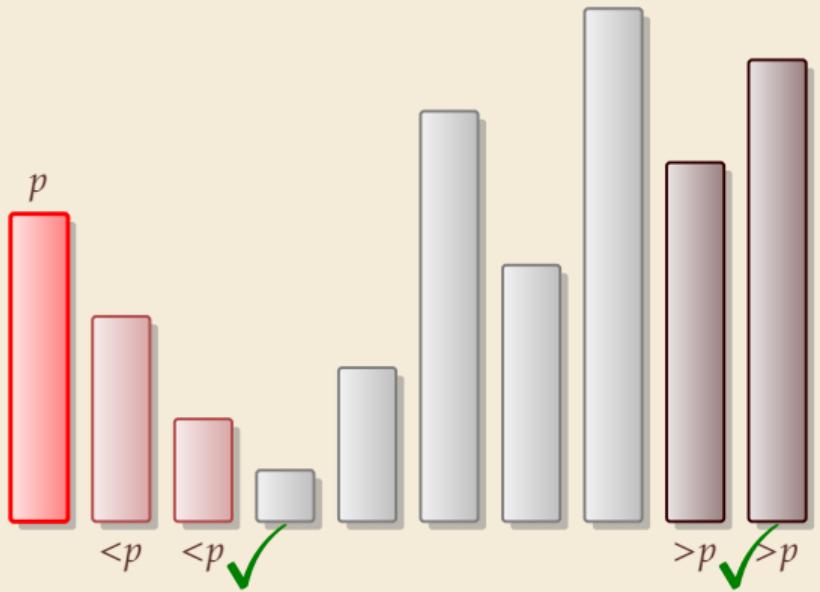
Partitioning around a pivot



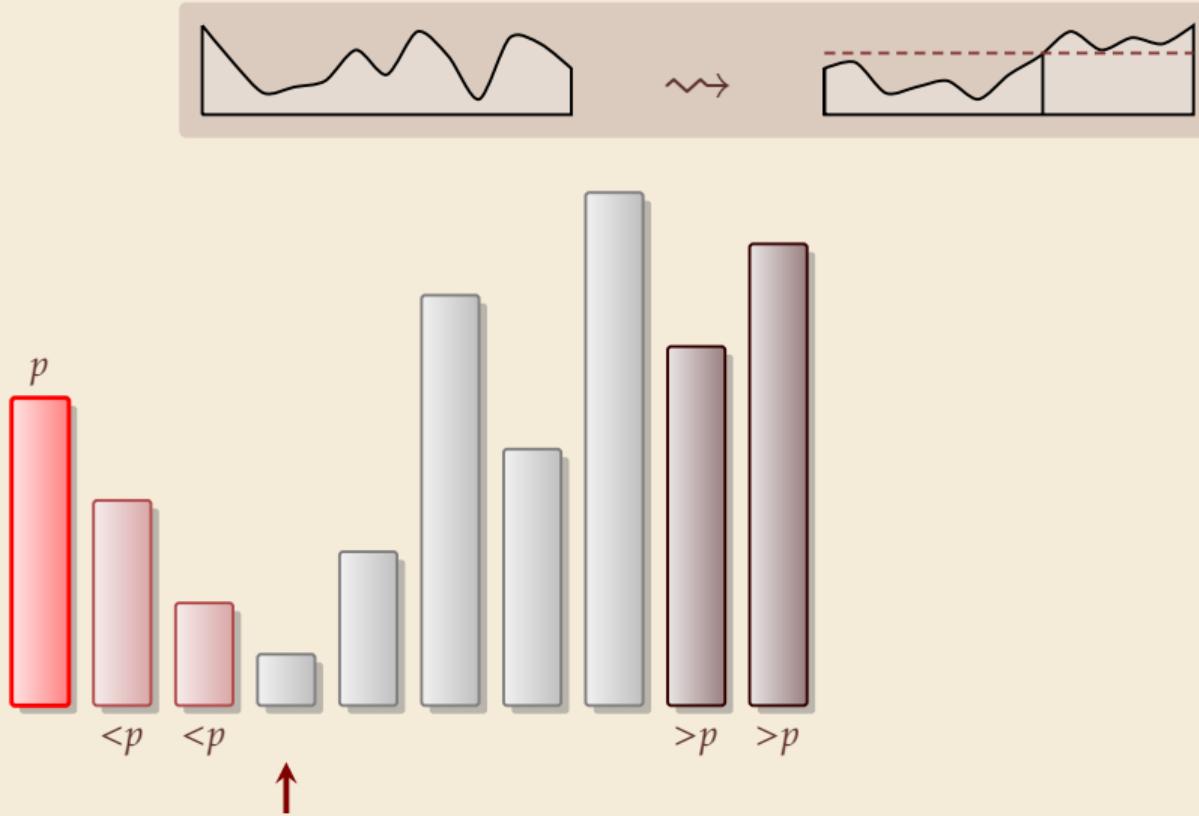
Partitioning around a pivot



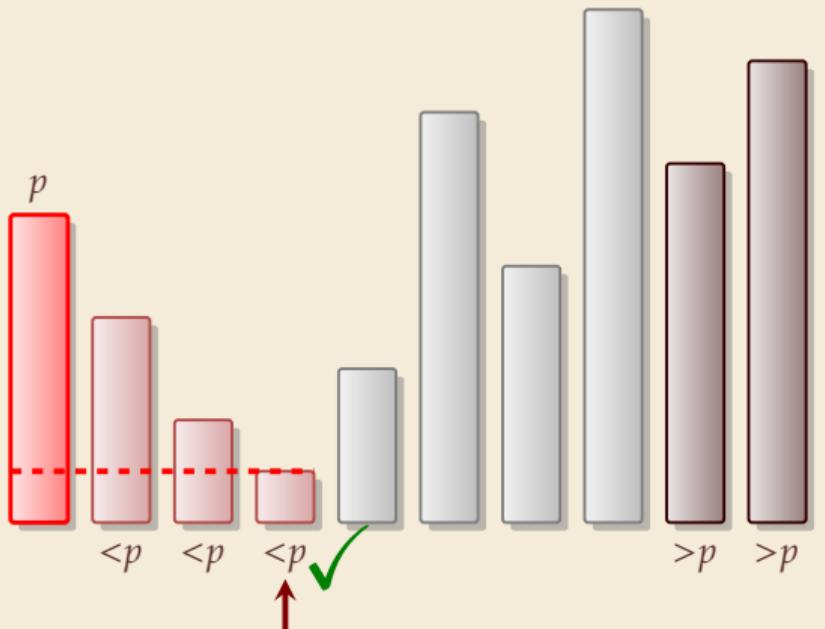
Partitioning around a pivot



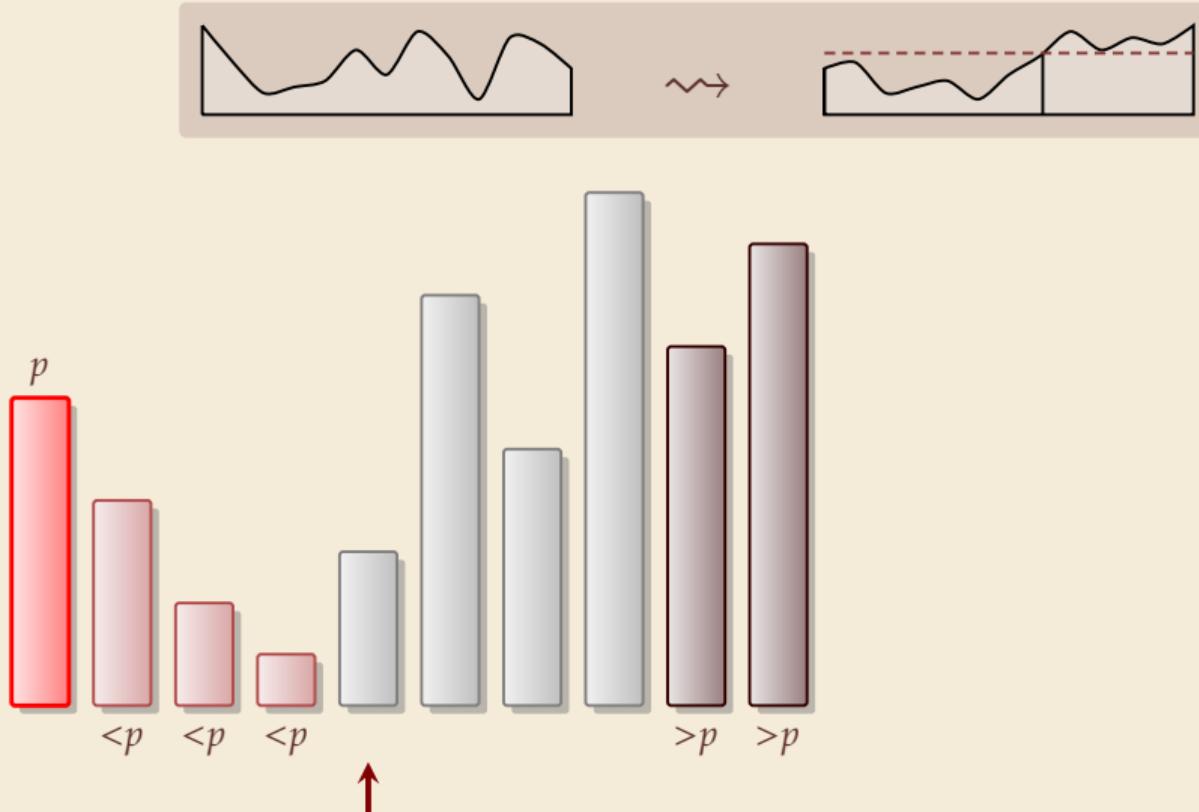
Partitioning around a pivot



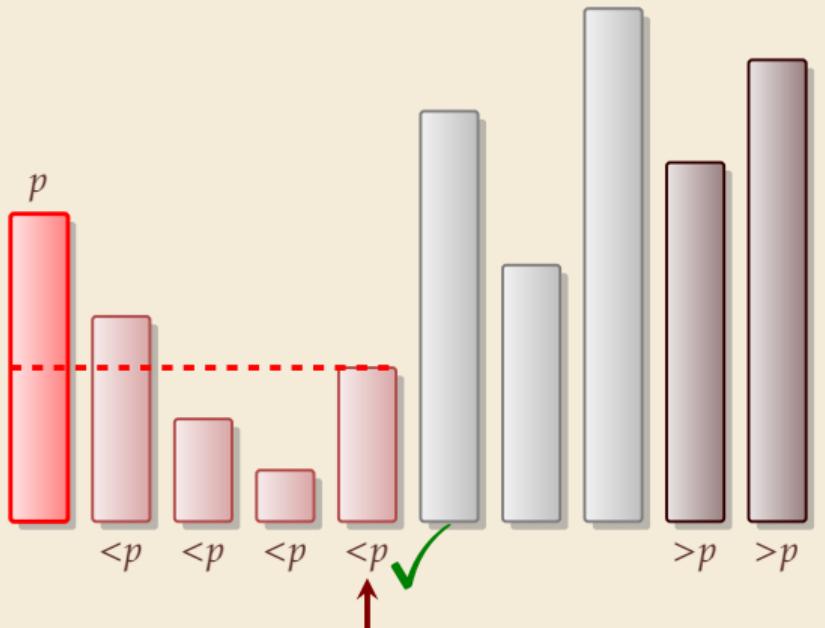
Partitioning around a pivot



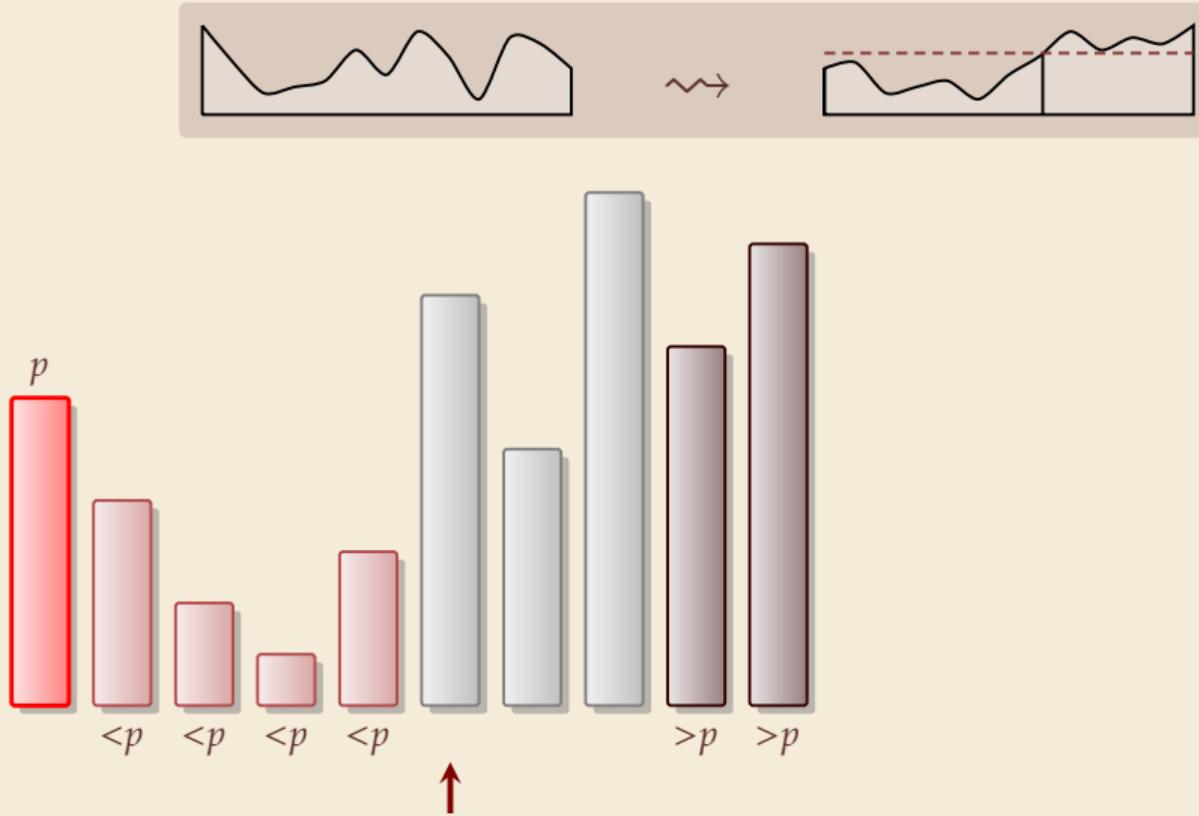
Partitioning around a pivot



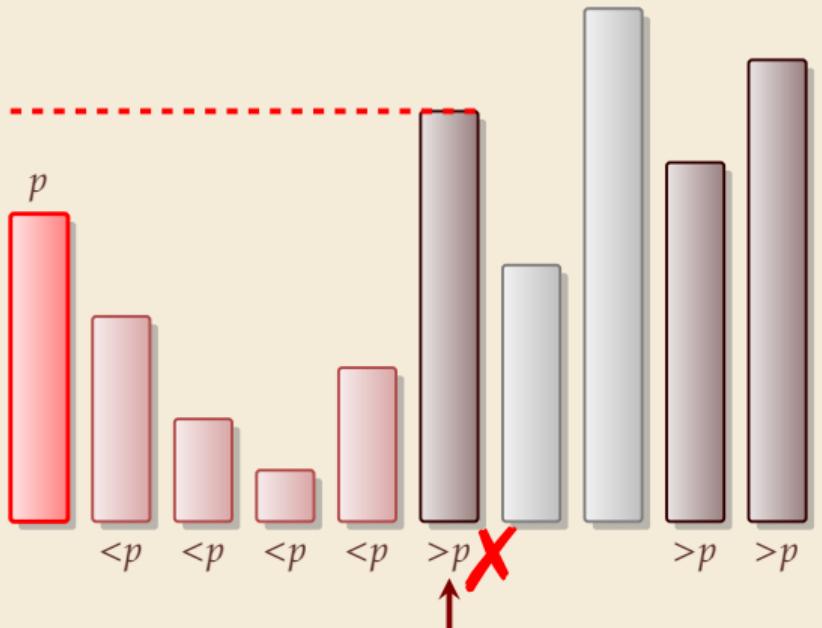
Partitioning around a pivot



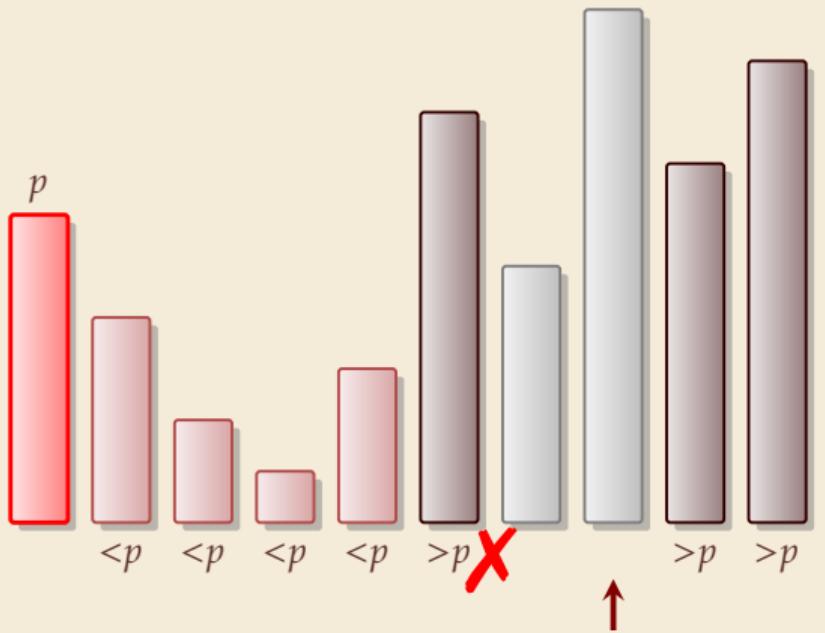
Partitioning around a pivot



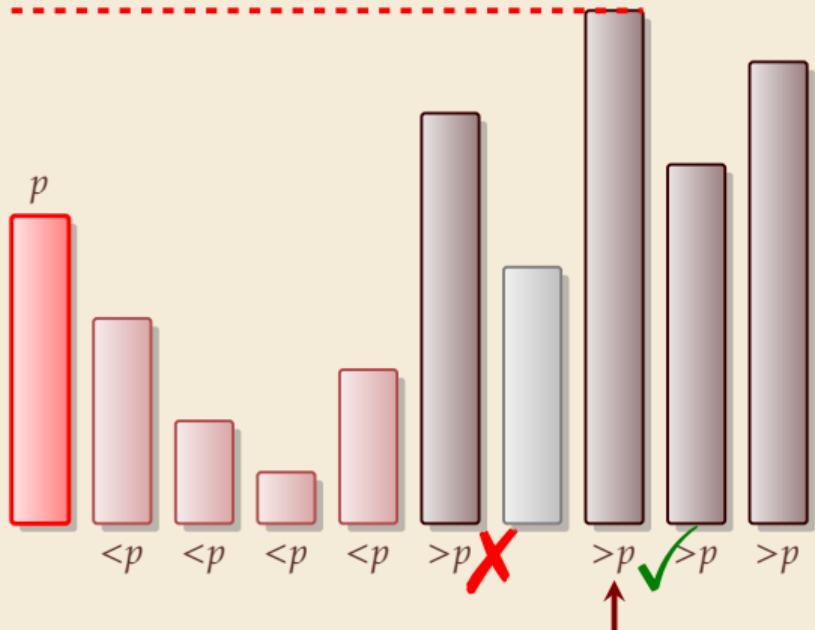
Partitioning around a pivot



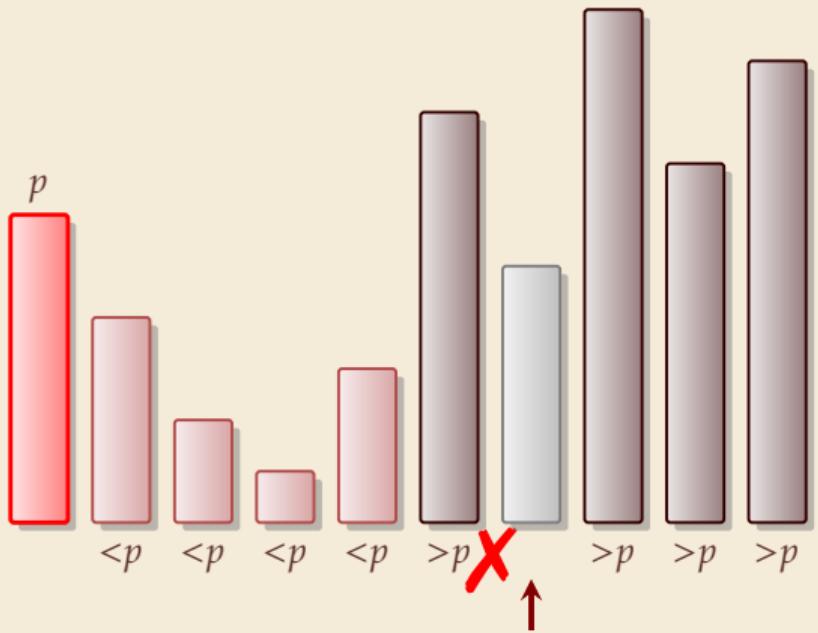
Partitioning around a pivot



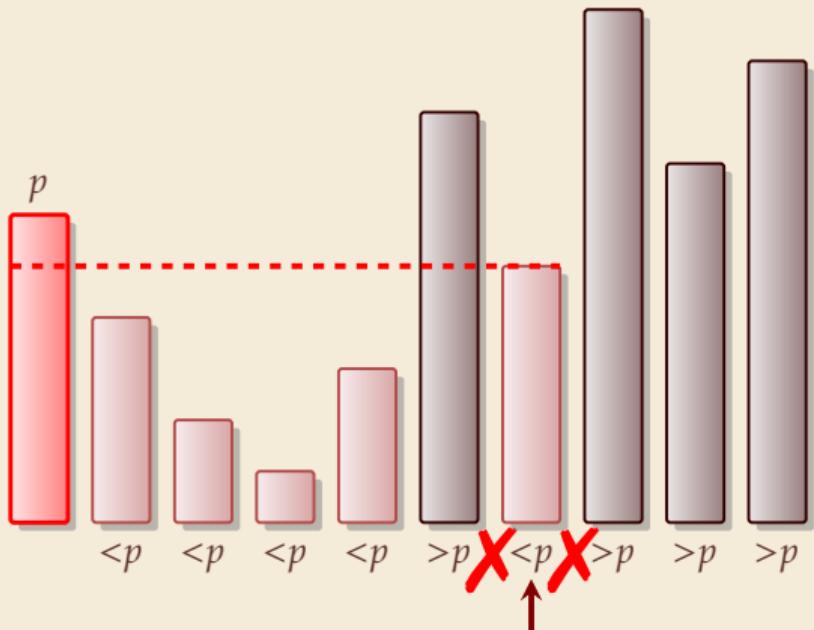
Partitioning around a pivot



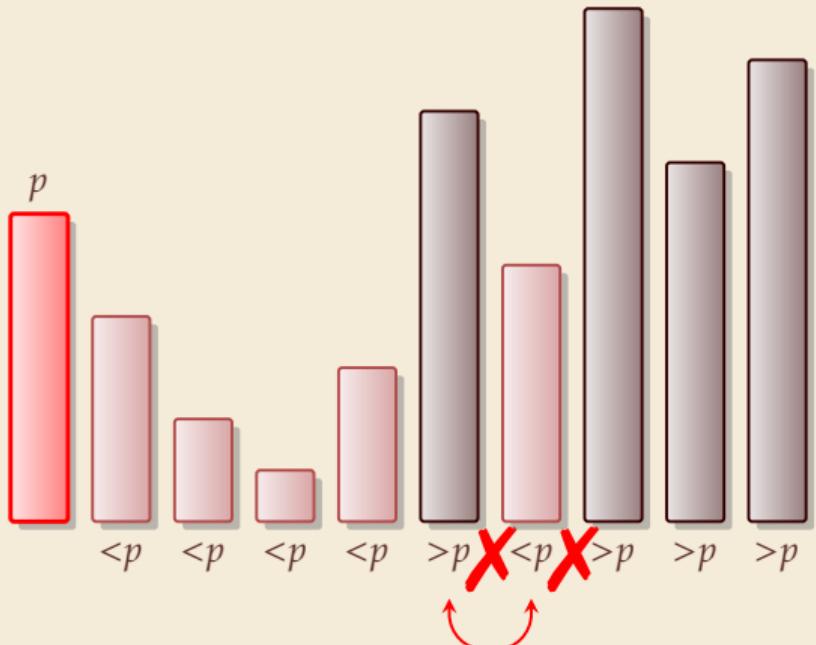
Partitioning around a pivot



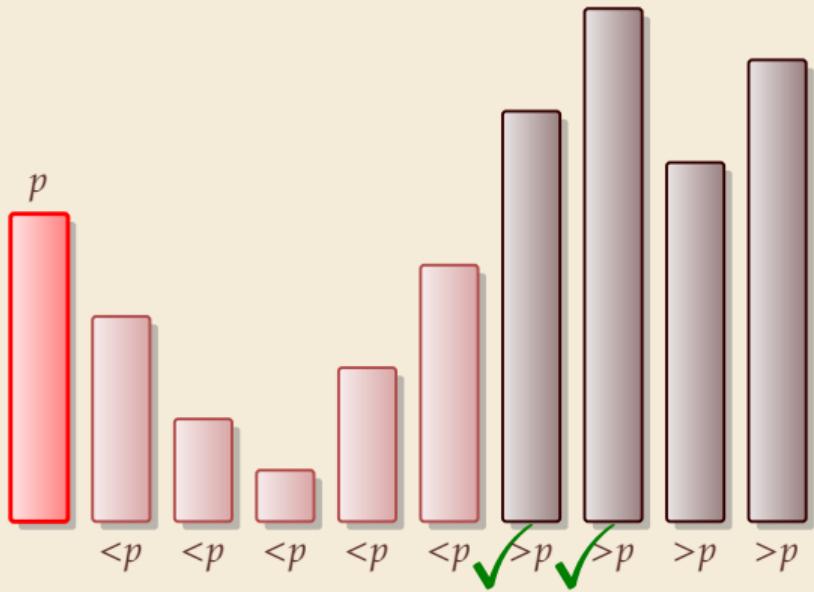
Partitioning around a pivot



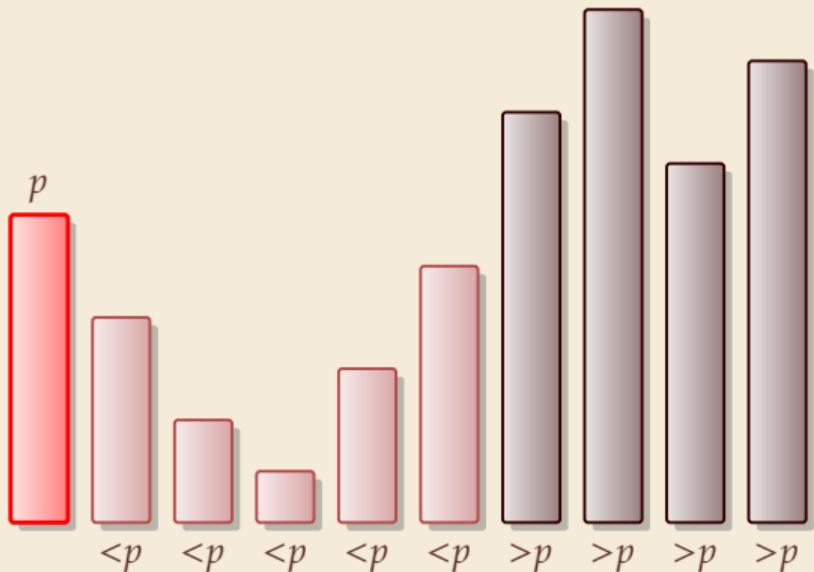
Partitioning around a pivot



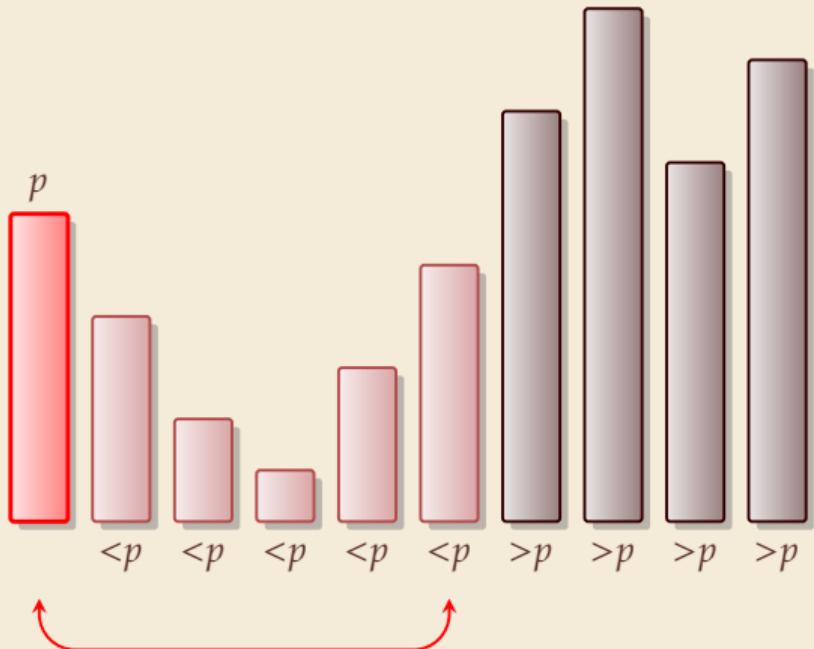
Partitioning around a pivot



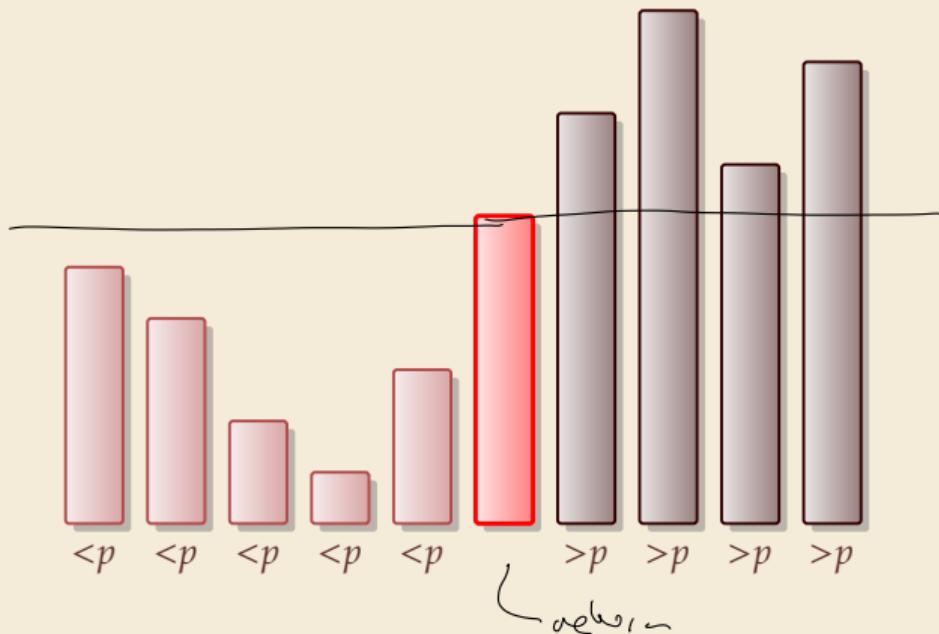
Partitioning around a pivot



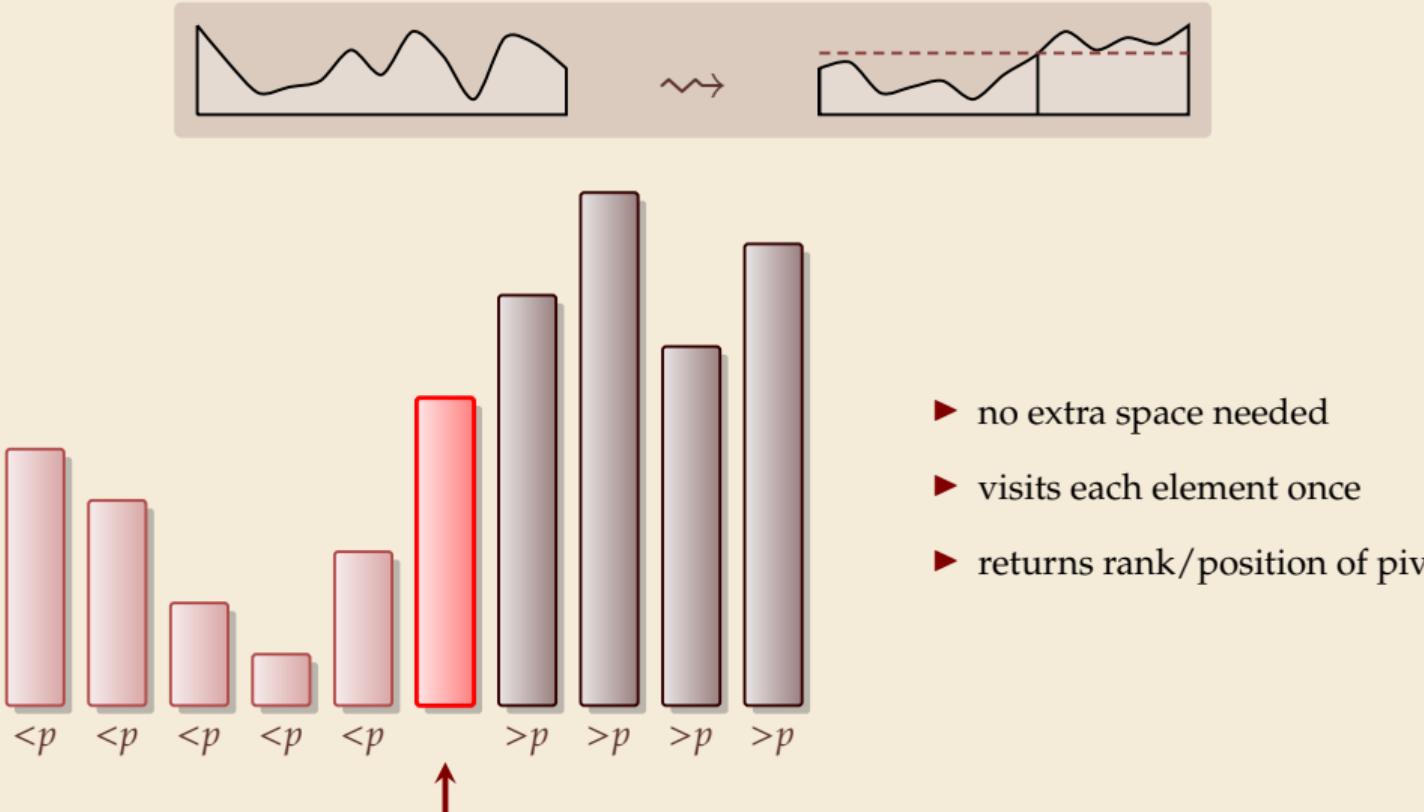
Partitioning around a pivot



Partitioning around a pivot



Partitioning around a pivot



Partitioning – Detailed code

Beware: details easy to get wrong; use this code!

```
1 procedure partition( $A, b$ )
2     // input: array  $A[0..n - 1]$ , position of pivot  $b \in [0..n - 1]$ 
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  $j \geq 1$  and  $A[j] > A[0]$ 
8         if  $i \geq j$  then break (goto 8)
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

```
1 procedure quicksort( $A[l..r]$ )
2   if  $l \geq r$  then return
3    $b := \text{choosePivot}(A[l..r])$ 
4    $j := \text{partition}(A[l..r], b)$ 
5   quicksort( $A[l..j - 1]$ )
6   quicksort( $A[j + 1..r]$ )
```

- ▶ recursive procedure; *divide & conquer*
- ▶ 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/comp526

Click on “Polls” tab

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/comp526

Click on “Polls” tab

Quicksort & Binary Search Trees

Quicksort

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

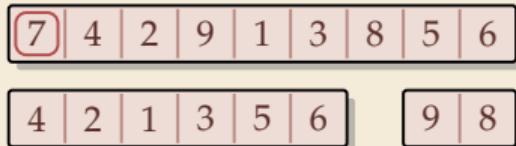
Quicksort & Binary Search Trees

Quicksort

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

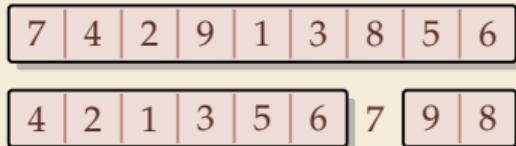
Quicksort & Binary Search Trees

Quicksort



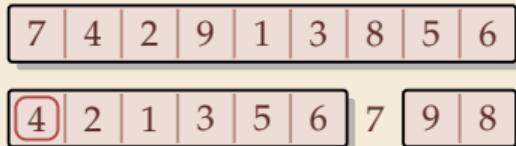
Quicksort & Binary Search Trees

Quicksort



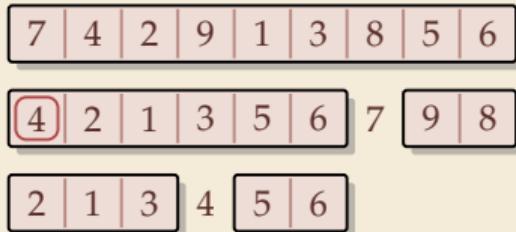
Quicksort & Binary Search Trees

Quicksort



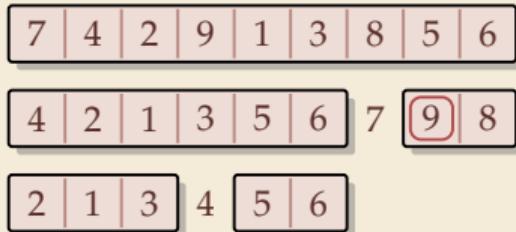
Quicksort & Binary Search Trees

Quicksort



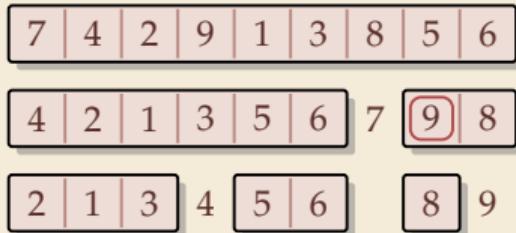
Quicksort & Binary Search Trees

Quicksort



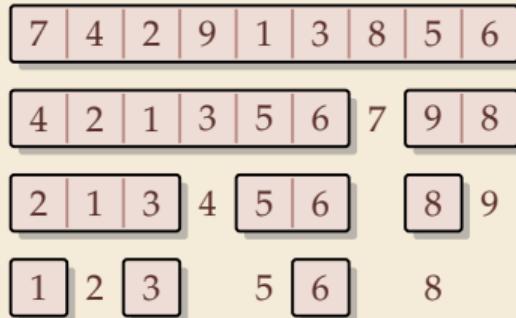
Quicksort & Binary Search Trees

Quicksort



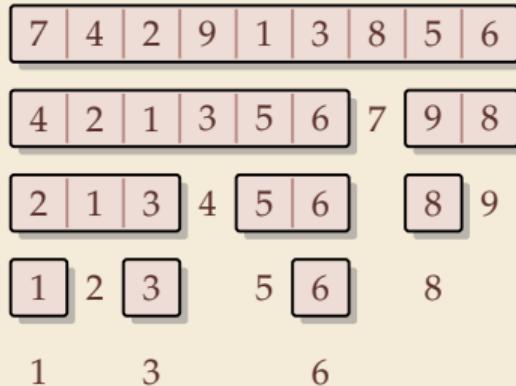
Quicksort & Binary Search Trees

Quicksort



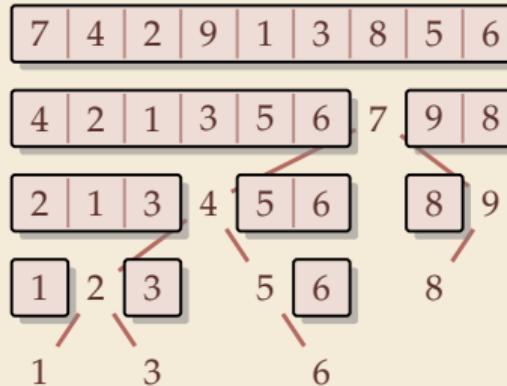
Quicksort & Binary Search Trees

Quicksort



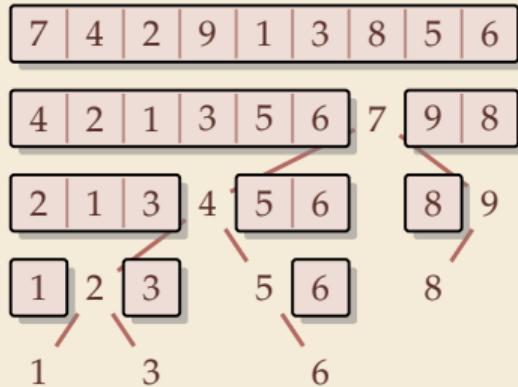
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort



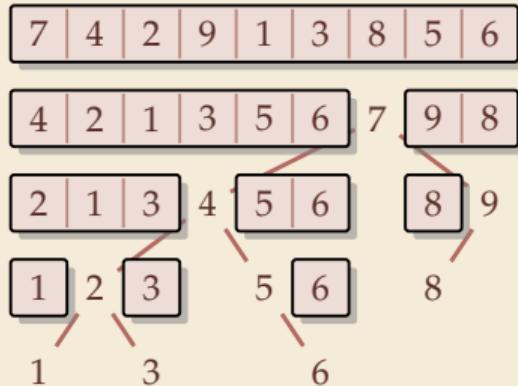
Binary Search Tree (BST)

7 4 2 9 1 3 8 5 6

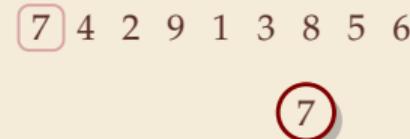
7 4 2 9 1 3 8 5 6

Quicksort & Binary Search Trees

Quicksort

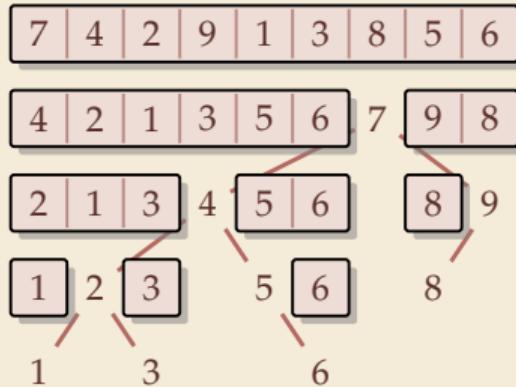


Binary Search Tree (BST)

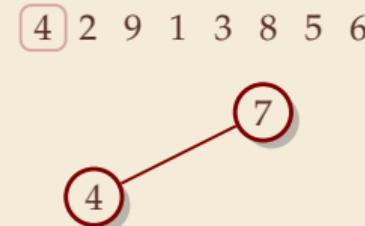


Quicksort & Binary Search Trees

Quicksort

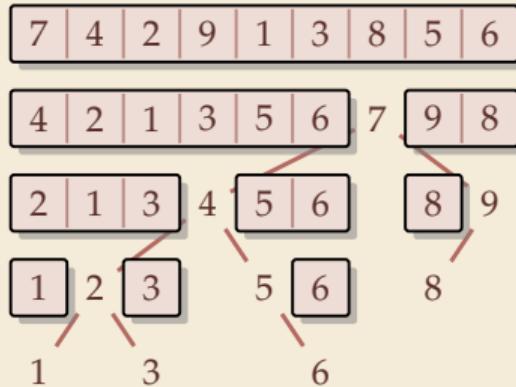


Binary Search Tree (BST)

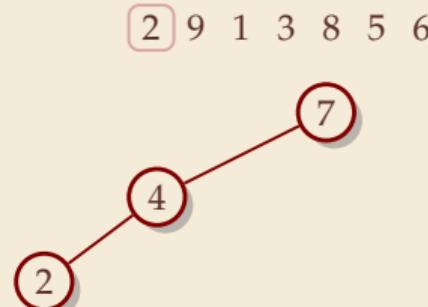


Quicksort & Binary Search Trees

Quicksort

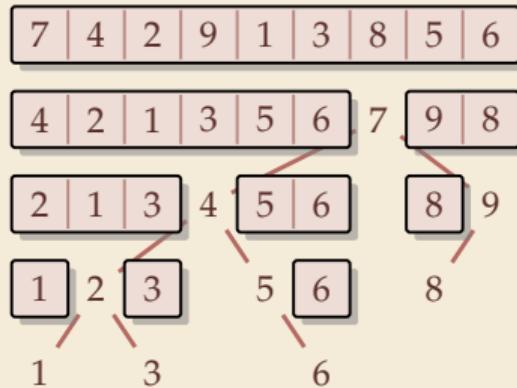


Binary Search Tree (BST)

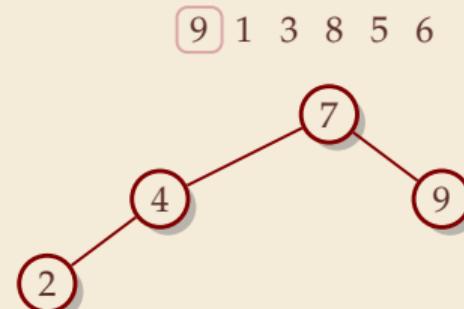


Quicksort & Binary Search Trees

Quicksort

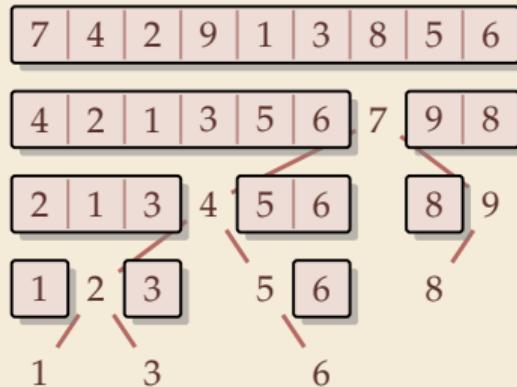


Binary Search Tree (BST)

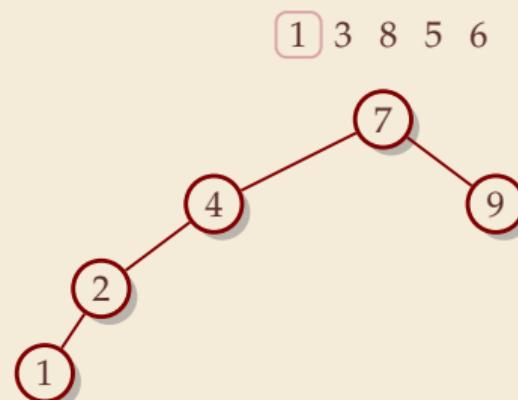


Quicksort & Binary Search Trees

Quicksort

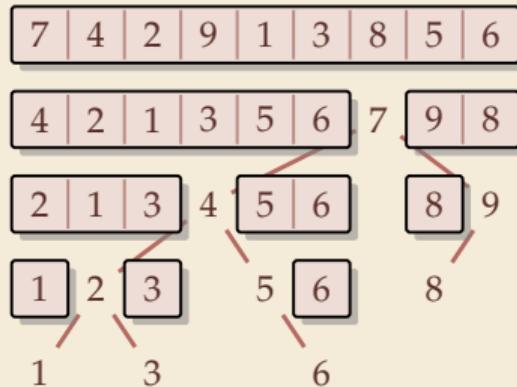


Binary Search Tree (BST)

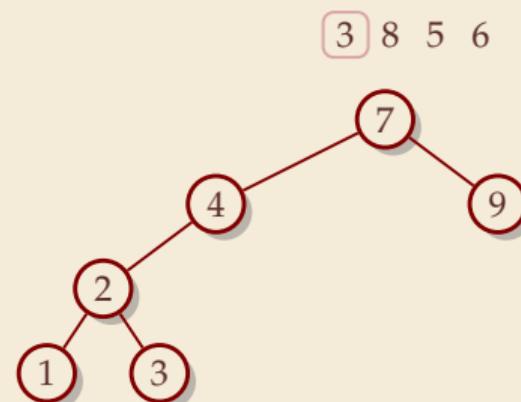


Quicksort & Binary Search Trees

Quicksort

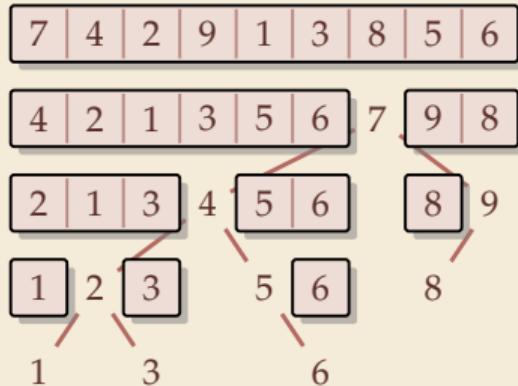


Binary Search Tree (BST)

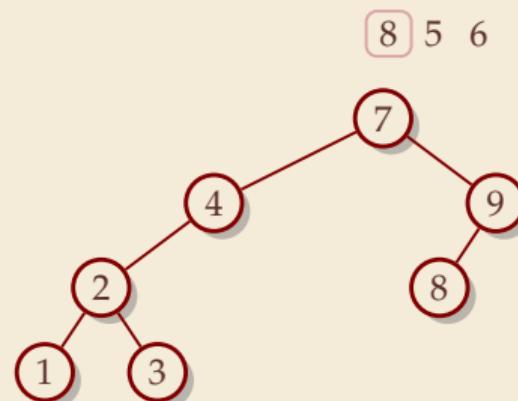


Quicksort & Binary Search Trees

Quicksort

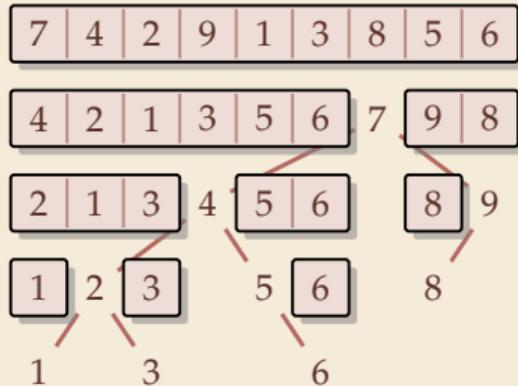


Binary Search Tree (BST)

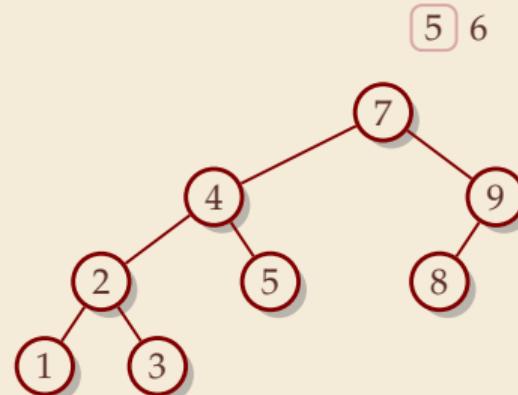


Quicksort & Binary Search Trees

Quicksort

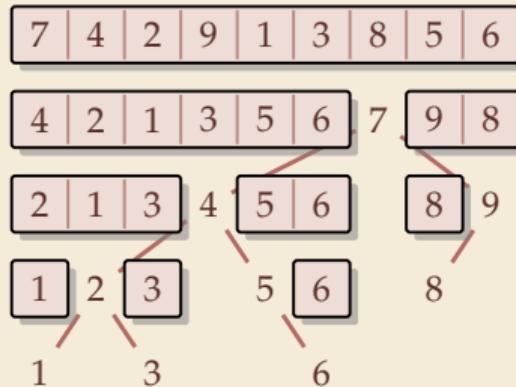


Binary Search Tree (BST)

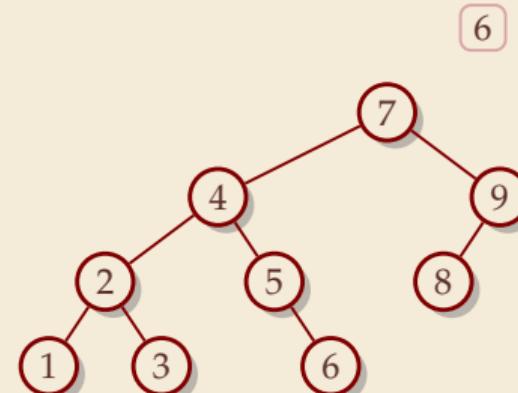


Quicksort & Binary Search Trees

Quicksort

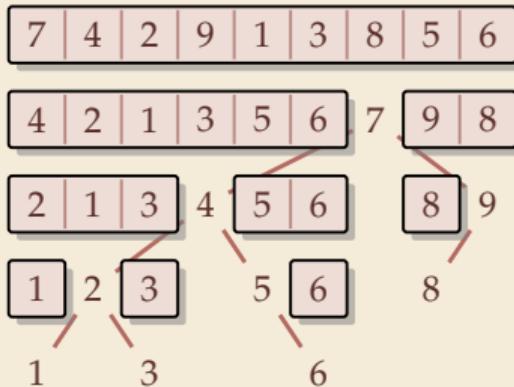


Binary Search Tree (BST)

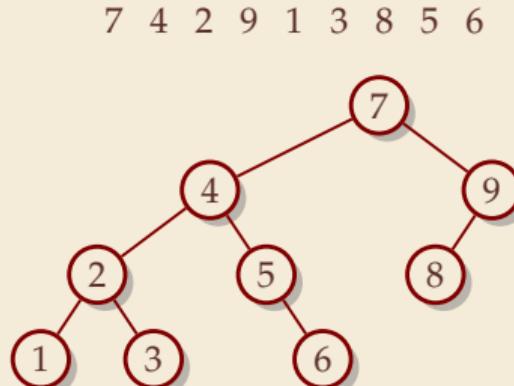


Quicksort & Binary Search Trees

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

Quicksort – Worst Case

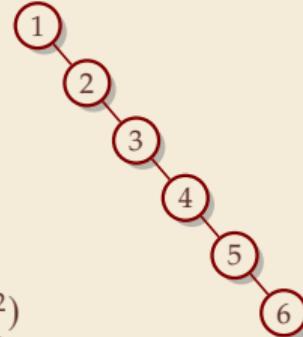
- ▶ Problem: BSTs can degenerate

- ▶ Cost to search for k is $k - 1$

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

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

terribly slow!



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

- ▶ $C(n)$ = element visits (as for mergesort)

↝ quicksort needs $\sim \underline{2 \ln(2) \cdot n \lg n} \approx 1.39n \lg n$ in expectation

Mergesort $2n \lg n$

- ▶ 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

Quicksort – Discussion

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

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

3.3 Comparison-Based Lower Bound

Lower Bounds

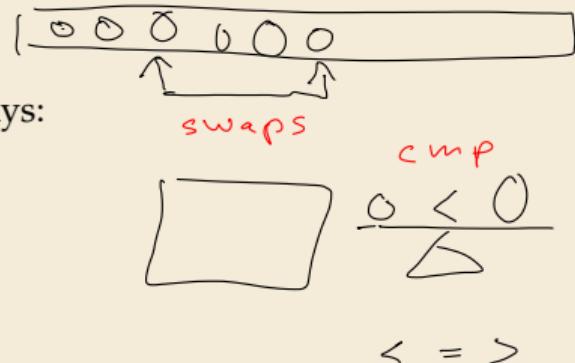
- ▶ **Lower bound:** mathematical proof that *no algorithm* can do better.
 - ▶ very powerful concept: bulletproof *impossibility* result
≈ *conservation of energy* in physics
 - ▶ **(unique?) feature of computer science:**
for many problems, solutions are known that (asymptotically) achieve the lower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

Lower Bounds

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



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 these operations.
- ▶ This makes very few assumptions on the kind of objects we are sorting.

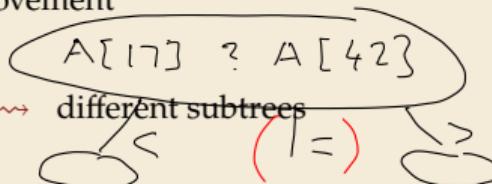
That's good!
Keeps algorithms general!
- ▶ Mergesort and Quicksort work in the comparison model.

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 these operations.
- ▶ This makes very few assumptions on the kind of objects we are sorting.
 - That's good!
Keeps algorithms general!
- ▶ Mergesort and Quicksort work in the comparison model.

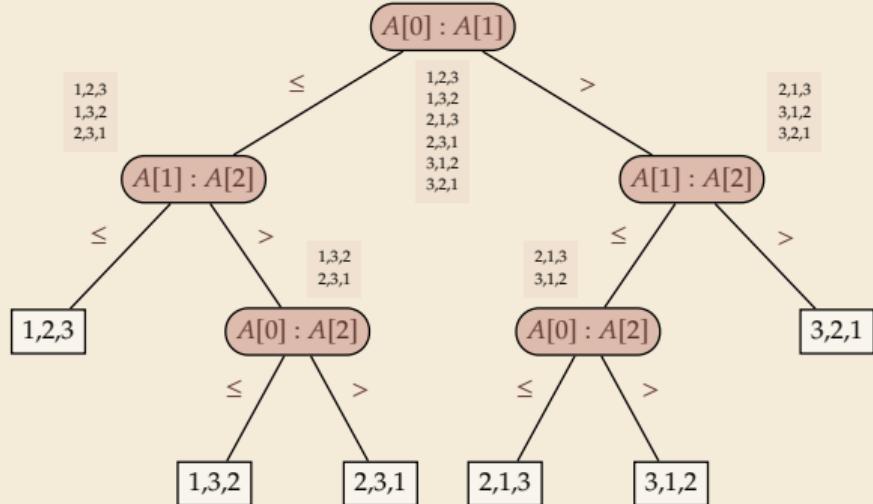
~~ Every comparison-based sorting algorithm corresponds to a decision tree.

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



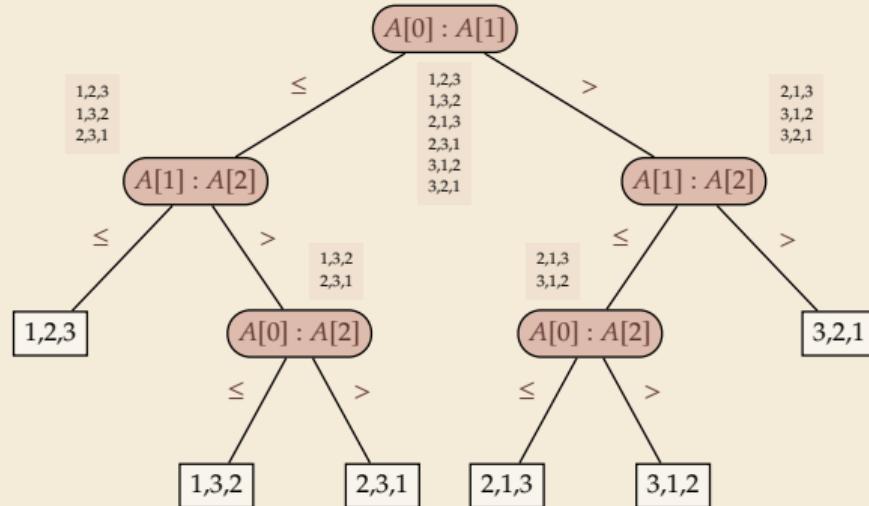
Comparison Lower Bound

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



Comparison Lower Bound

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

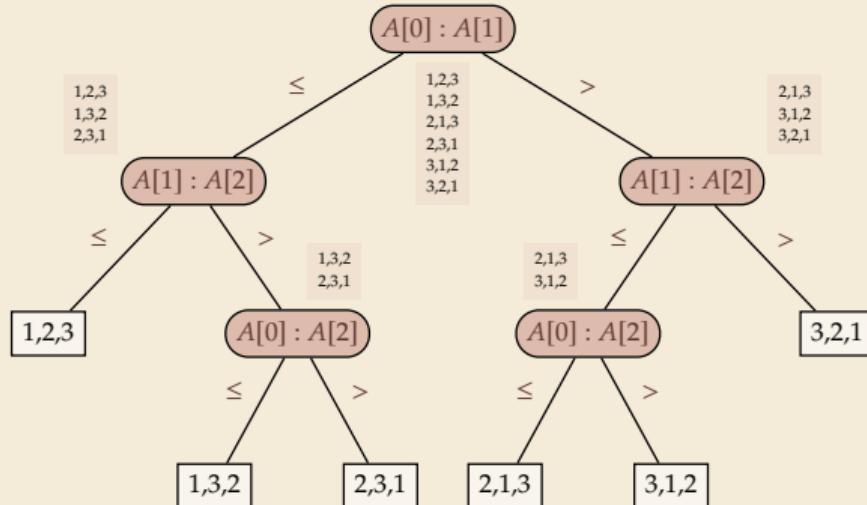


1 comparison tree $\hat{=}$ 1 algorithm
and 1 input size

- ▶ Execution = follow a path in comparison tree.
 - ~~ height of comparison tree = worst-case # comparisons
 - ▶ comparison trees are *binary* trees
 - ~~ ℓ leaves \rightsquigarrow height $\geq \lceil \lg(\ell) \rceil$
↳ heaps
 - ▶ comparison trees for sorting method must have $\geq \underline{n!}$ leaves
 - ~~ height $\geq \lg(n!) \sim n \lg n$
more precisely: $\lg(n!) = n \lg n - \lg(e)n + O(\log n)$
- any cmp-based sorting algorithm needs
 $\geq \lg(n!) \sim n \lg n$ cmps in the worst case

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 \rightsquigarrow height $\geq \lceil \lg(\ell) \rceil$
- ▶ comparison trees for sorting method must have $\geq n!$ leaves
- ~~ height $\geq \lg(n!) \sim n \lg n$

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

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

$$\approx 1.4427$$

Clicker Question



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

A Yes

B No

sli.do/comp526

Click on “Polls” tab

Clicker Question



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

A Yes ✓

B No

sli.do/comp526

Click on “Polls” tab

3.4 Integer Sorting

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*!
 - ~~ 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!*
 - ~~ 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)
 - ~~ we are **not** working in the comparison model
 - ~~ *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!*
 - ~~ 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)
 - ~~ we are **not** working in the comparison model
 - ~~ *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 $\underbrace{[0..U)} = \{0, \dots, U - 1\}$ typically $U = \underbrace{2^b} \rightsquigarrow \underline{b\text{-bit binary numbers}}$

encoded in binary

Counting sort

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

Counting sort

```
1 procedure countingSort(A[0..n - 1])
2     // A contains integers in range [0..U).
3     | C[0..U - 1] := new integer array, initialized to 0
4     // Count occurrences
5     | for i := 0, ..., n - 1
6     |   C[A[i]] := C[A[i]] + 1
7     |   i := 0 // Produce sorted list
8     } for k := 0, ..., U - 1
9     |   for j := 1, ..., C[k]
10    |     A[i] := k; i := i + 1
```

$$\sum_k C[k] = n$$

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

used in `Arrays.sort(byte[])`

- ▶ count how often each possible value occurs (1)
- ▶ produce sorted result directly from counts
- ▶ circumvents lower bound by using integers as array index / pointer offset



Integer Sorting – State of the art

- ▶ $O(n)$ time sorting also possible for numbers in range $\underline{U = O(n^c)}$ for constant c .
 - ▶ radix sort with radix 2^w

▶ Algorithm theory

(Unit 1 : $\omega = \Theta(\log n)$)

- ▶ 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!

notin exam

Integer Sorting – State of the art

- ▶ $O(n)$ time sorting also possible for numbers in range $U = O(n^c)$ for constant c .
 - ▶ *radix sort* with radix 2^w
- ▶ **Algorithm theory**
 - ▶ 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 time proportional to n .
- B in $O(n)$ time.
- C in $O(n \log n)$ time.
- D in constant time.
- E some time, but not possible to say from given information.

sli.do/comp526

Click on “Polls” tab

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 time proportional to n .~~ OK
- B in $O(n)$ time. ✓ $\Theta(n + \cup) = \Theta(n)$
counting sort
- C in $O(n \log n)$ time. ✓ (mergesort)
- D ~~in constant time.~~ $n = 10$
- E some time, but not possible to say from given information. ✓ \cup need $\Theta(\cup)$ space

Part II

Sorting with many processors

3.5 Parallel computation

Clicker Question



Have you ever written a concurrent program (explicit threads, job pools library, or using a framework for distributed computing)?

- A Yes
- B No
- C Concur... what?

sli.do/comp526

Click on “Polls” tab

Clicker Question



Have you ever written a concurrent program (explicit threads, job pools library, or using a framework for distributed computing)?

- A Yes
- B No
- C ~~Concur... what?~~

sli.do/comp526

Click on “Polls” tab

Types of parallel computation

£££ can't buy you more time . . . but more computers!

~~~ Challenge: Algorithms for parallel computation.

# Types of parallel computation

£££ can't buy you more time . . . but more computers!

~~~ Challenge: Algorithms for *parallel* computation.

There are two main forms of parallelism:

1. shared-memory parallel computer \leftarrow focus of today
 - ▶ p *processing elements* (PEs, processors) working in parallel
 - ▶ single big memory, **accessible from every PE**
 - ▶ communication via shared memory
 - ▶ think: a big server, 128 CPU cores, terabyte of main memory
2. **distributed computing**
 - ▶ p PEs working in parallel
 - ▶ each PE has **private** memory
 - ▶ communication by sending **messages** via a network
 - ▶ think: a cluster of individual machines

PRAM – Parallel RAM

- ▶ extension of the RAM model (recall Unit 1)
- ▶ the p PEs are identified by ids $0, \dots, p - 1$
 - ▶ like w (the word size), p is a parameter of the model that can grow with n
 - ▶ $p = \Theta(n)$ is not unusual maaany processors!
- ▶ the PEs all independently run ~~a~~^{the same} RAM-style program
(they can use their id there)
- ▶ each PE has its own registers, but **MEM** is shared among all PEs
- ▶ computation runs in synchronous steps:
in each time step, every PE executes one instruction

PRAM – Conflict management



Problem: What if several PEs simultaneously overwrite a memory cell?

- ▶ **EREW-PRAM** (exclusive read, exclusive write)
any parallel access to same memory cell is forbidden (crash if happens)
- ▶ **CREW-PRAM** (concurrent read, exclusive write)
parallel write access to same memory cell is forbidden, but reading is fine
- ▶ **CRCW-PRAM** (concurrent read, concurrent write)
concurrent access is allowed,
need a rule for write conflicts:
 - ▶ common CRCW-PRAM:
all concurrent writes to same cell must write same value
 - ▶ arbitrary CRCW-PRAM:
some unspecified concurrent write wins | closest to CPUs
 - ▶ (more exist ...)
 race conditions
- ▶ no single model is always adequate, but our default is CREW

PRAM – Execution costs

Cost metrics in PRAMs

- ▶ **space:** total amount of accessed memory
- ▶ **time:** number of steps till all PEs finish assuming sufficiently many PEs!
sometimes called *depth* or *span*
- ▶ **work:** total #instructions executed on **all** PEs

PRAM – Execution costs

Cost metrics in PRAMs

- ▶ **space:** total amount of accessed memory
- ▶ **time:** number of steps till all PEs finish assuming sufficiently many PEs!
sometimes called *depth* or *span*
- ▶ **work:** total #instructions executed on **all** PEs *sequential* : $\text{work} = \text{time}$

Holy grail of PRAM algorithms:

- ▶ minimal time (and space)
- ▶ work (asymptotically) no worse than running time of best sequential algorithm
~~ “*work-efficient*” algorithm: work in same Θ -class as best sequential

Clicker Question



Does every computational problem allow a work-efficient algorithm?

- A Yes
- B No

sli.do/comp526

Click on “Polls” tab

Clicker Question



Does every computational problem allow a work-efficient algorithm?

- A Yes ✓
- B No

sli.do/comp526

Click on “Polls” tab

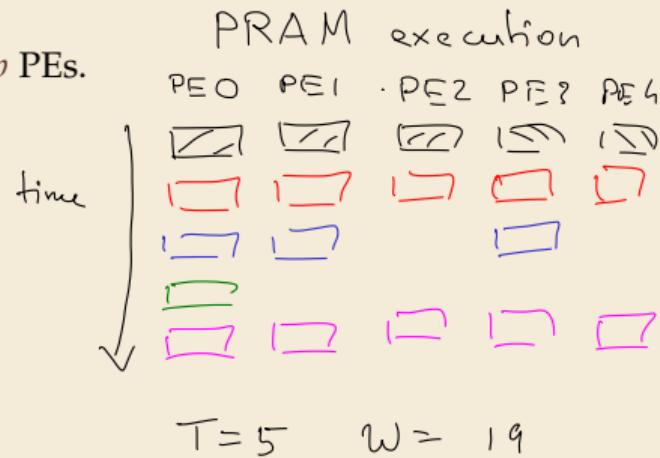
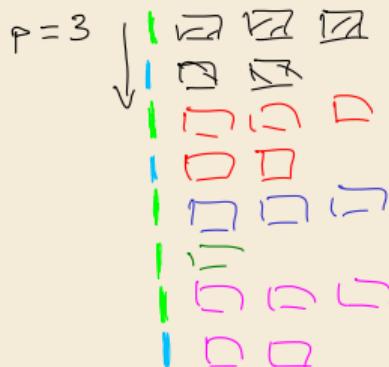
The number of processors

Hold on, my computer does not have $\Theta(n)$ processors! Why should I care for span and work!?

Theorem 3.1 (Brent's Theorem):

If an algorithm has span T and work W (for an arbitrarily large number of processors), it can be run on a PRAM with p PEs in time $O(T + \frac{W}{p})$ (and using $O(W)$ work). ◀

Proof: schedule parallel steps in round-robin fashion on the p PEs.



↔ span and work give guideline for *any* number of processors

PRAM

$$P = kP + 1$$

$$\square \cdots \square \quad P \quad \curvearrowleft \quad W = T \cdot P$$

$$k \times \begin{pmatrix} \square & \square & \square & P \\ \square & \square & \square & k-1 \end{pmatrix} \quad k = \frac{W}{P}$$