

3

Efficient Sorting

23 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.~~

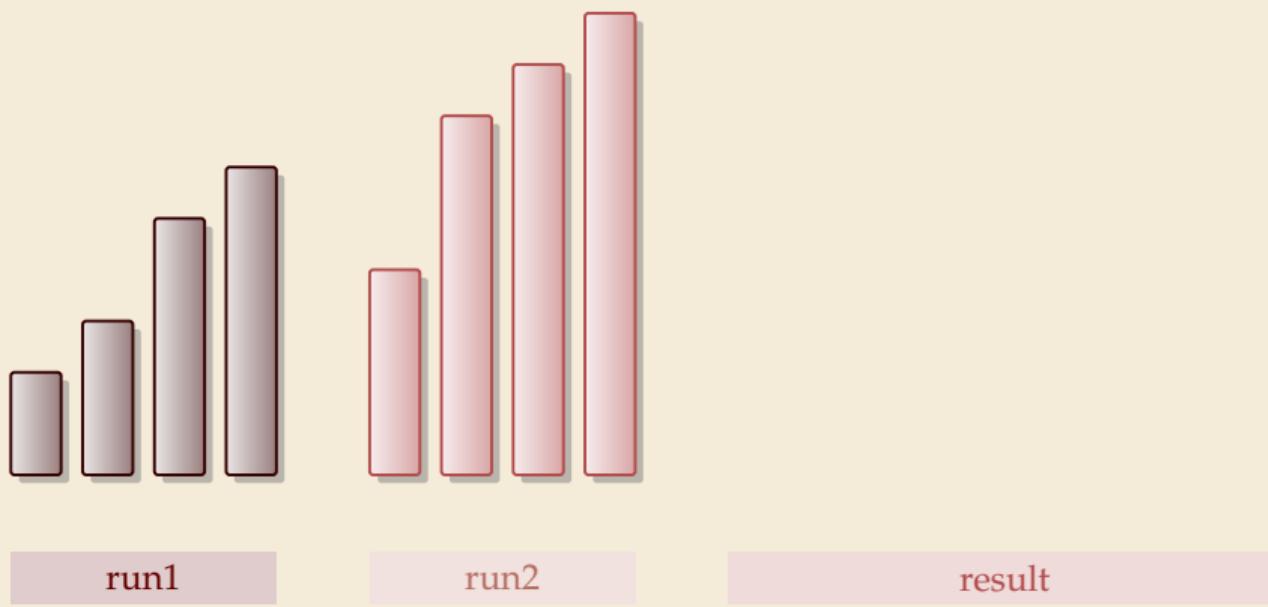
sli.do/comp526

Click on "Polls" tab

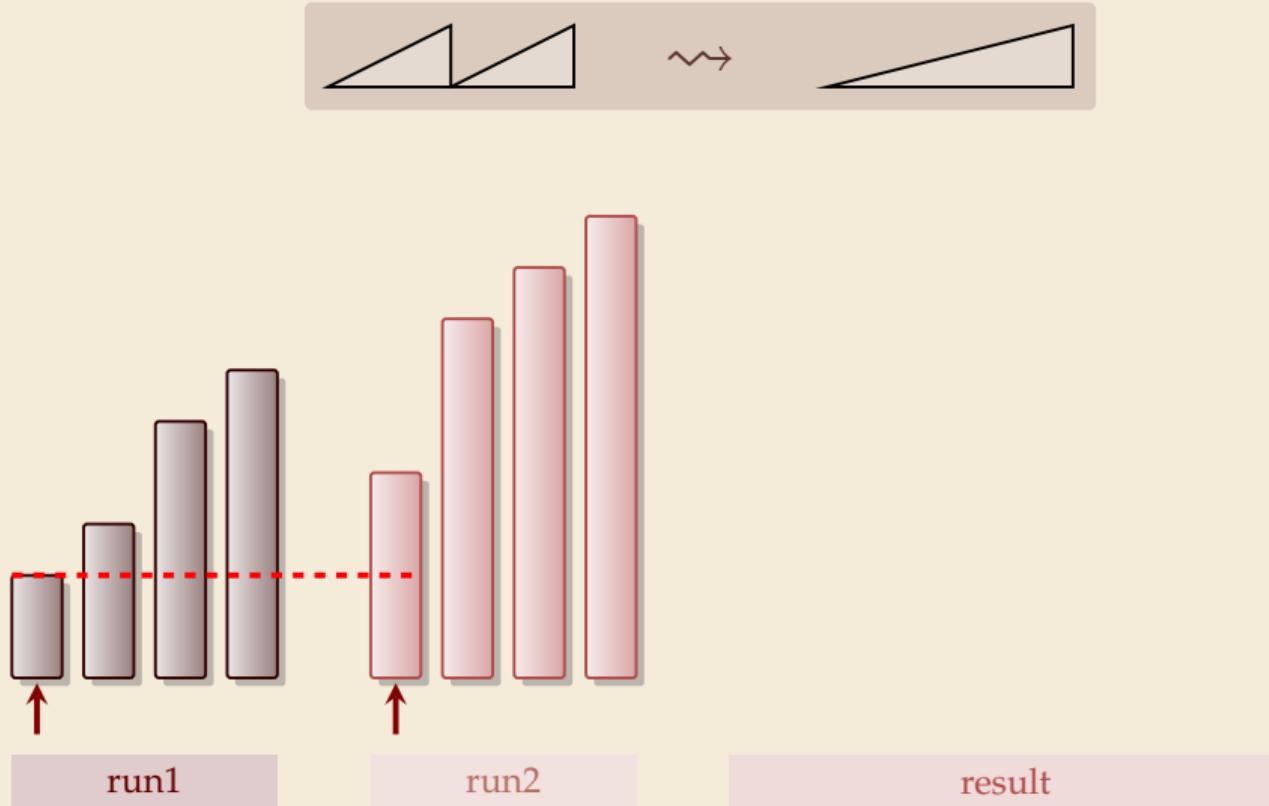
Merging sorted lists



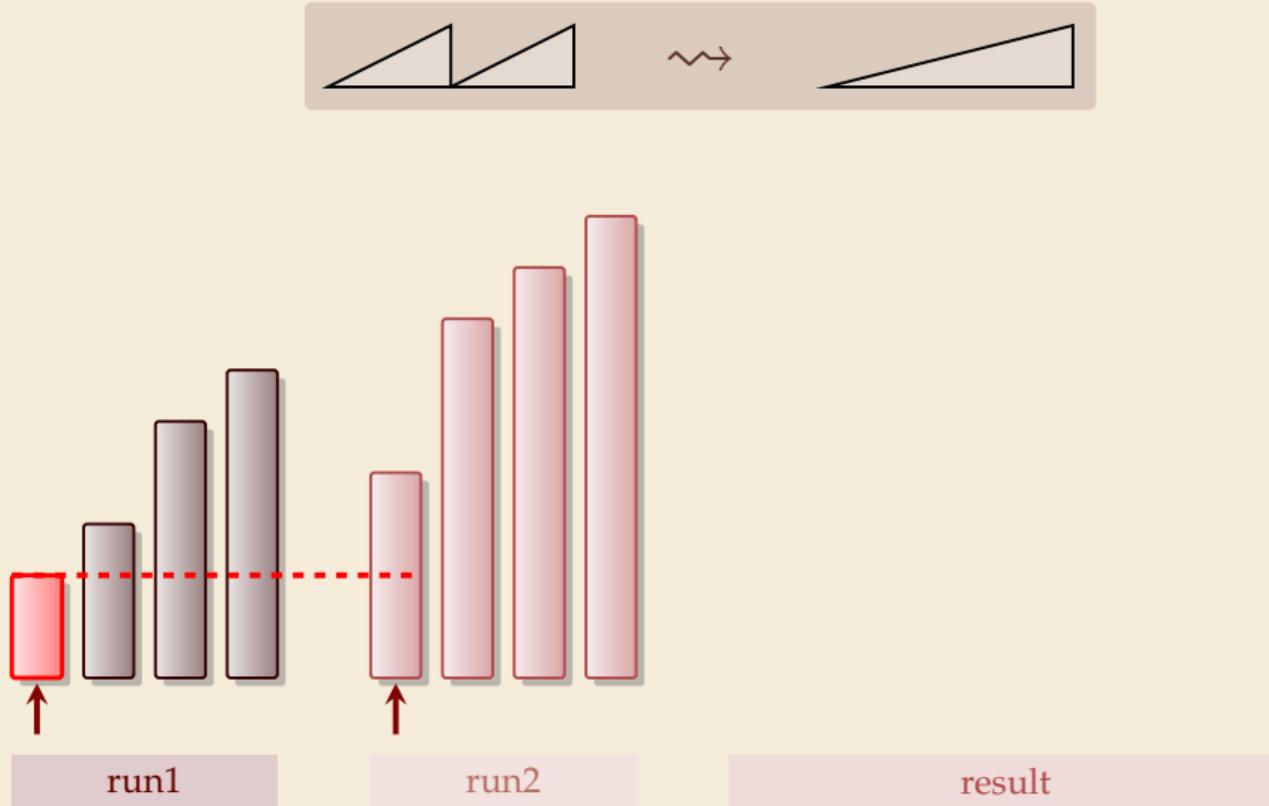
Merging sorted lists



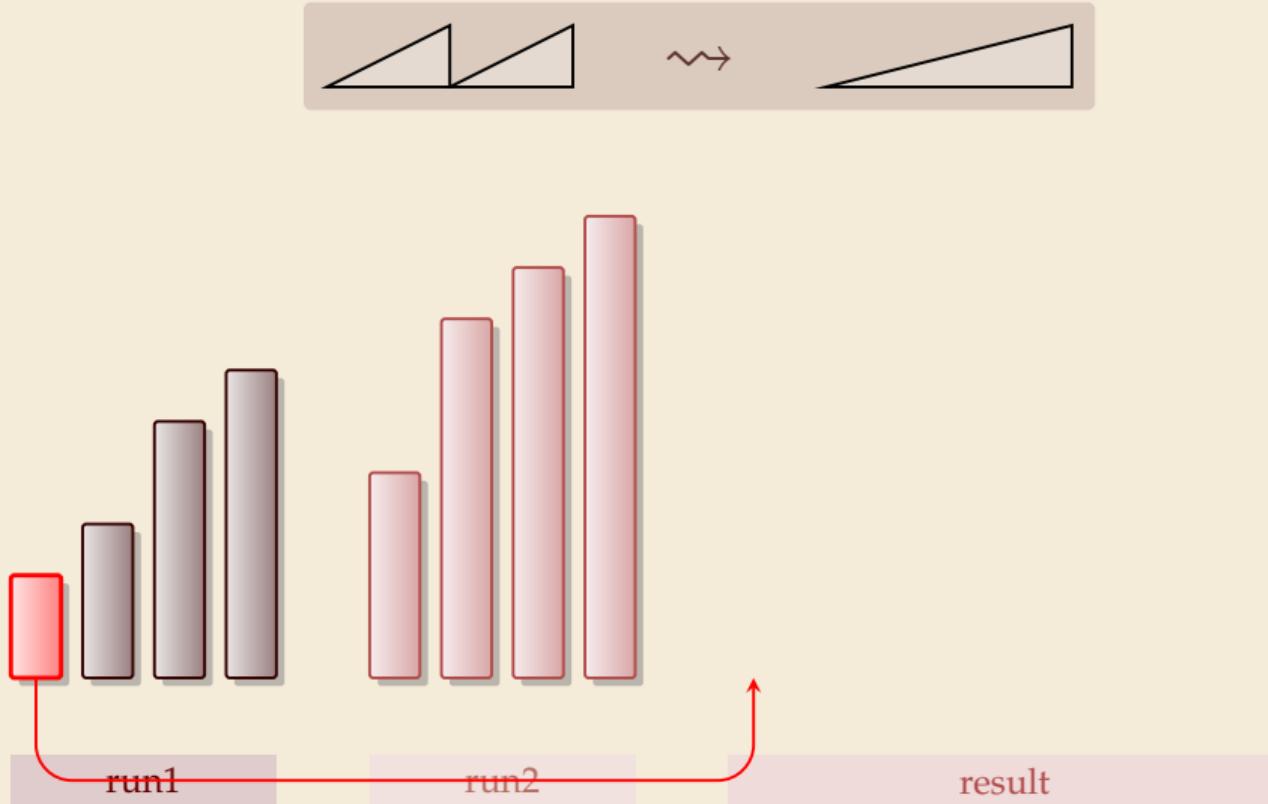
Merging sorted lists



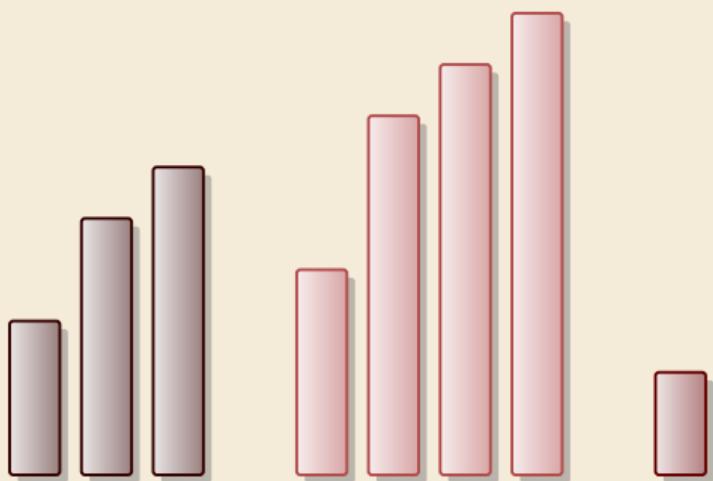
Merging sorted lists



Merging sorted lists



Merging sorted lists

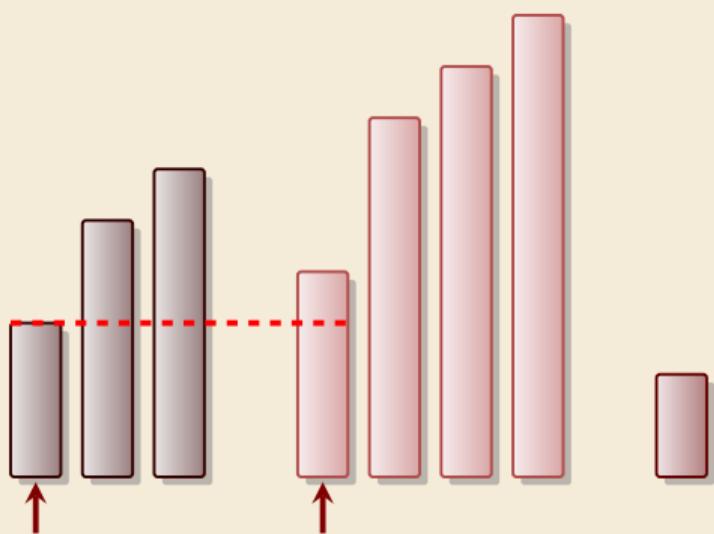


run1

run2

result

Merging sorted lists

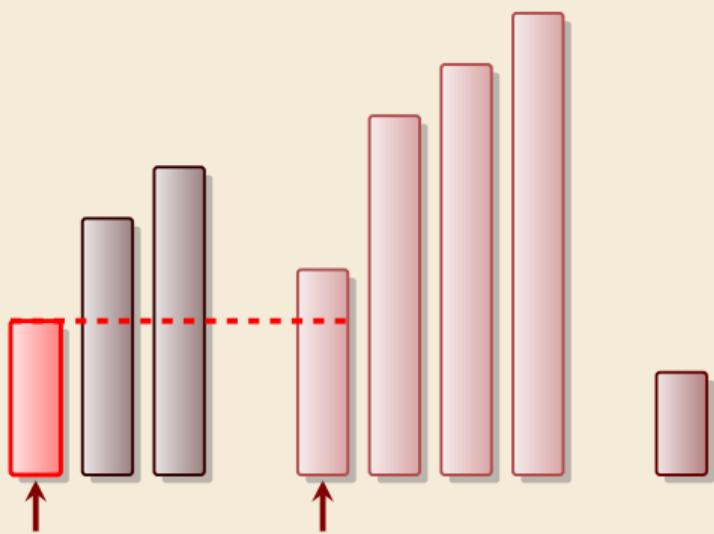


run1

run2

result

Merging sorted lists

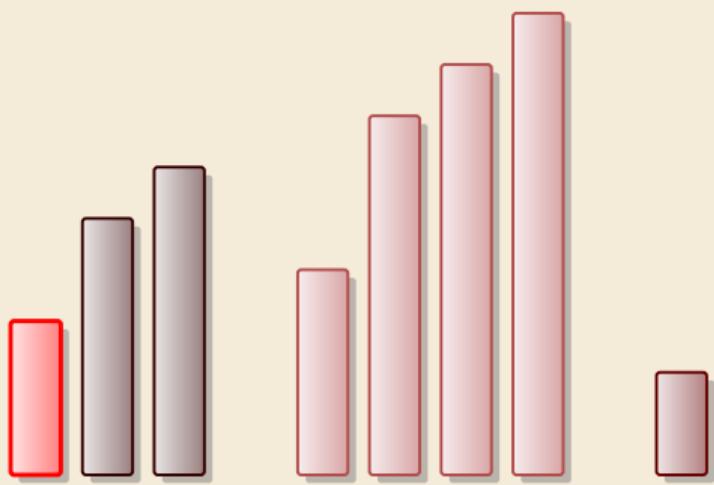


run1

run2

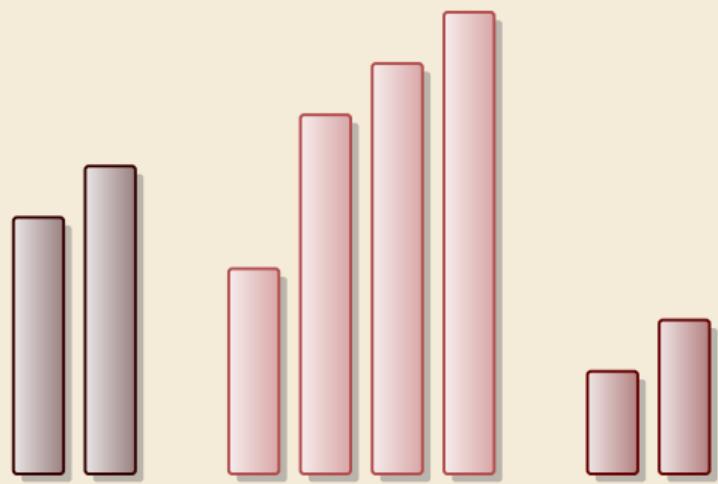
result

Merging sorted lists

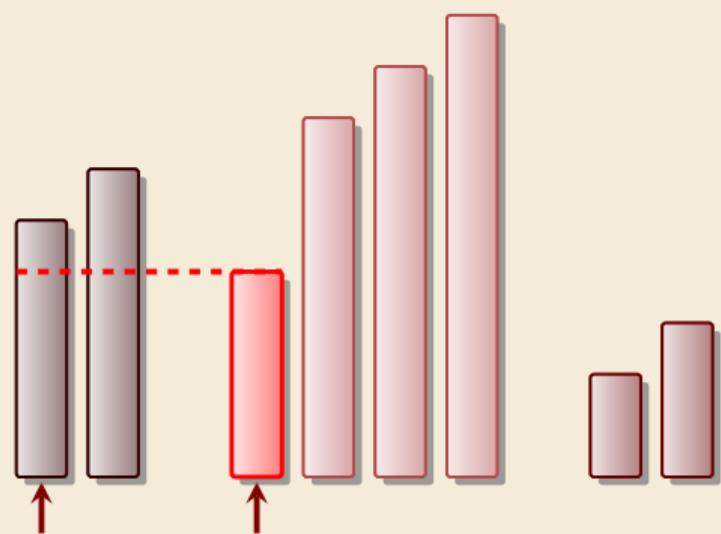


run1 run2 result

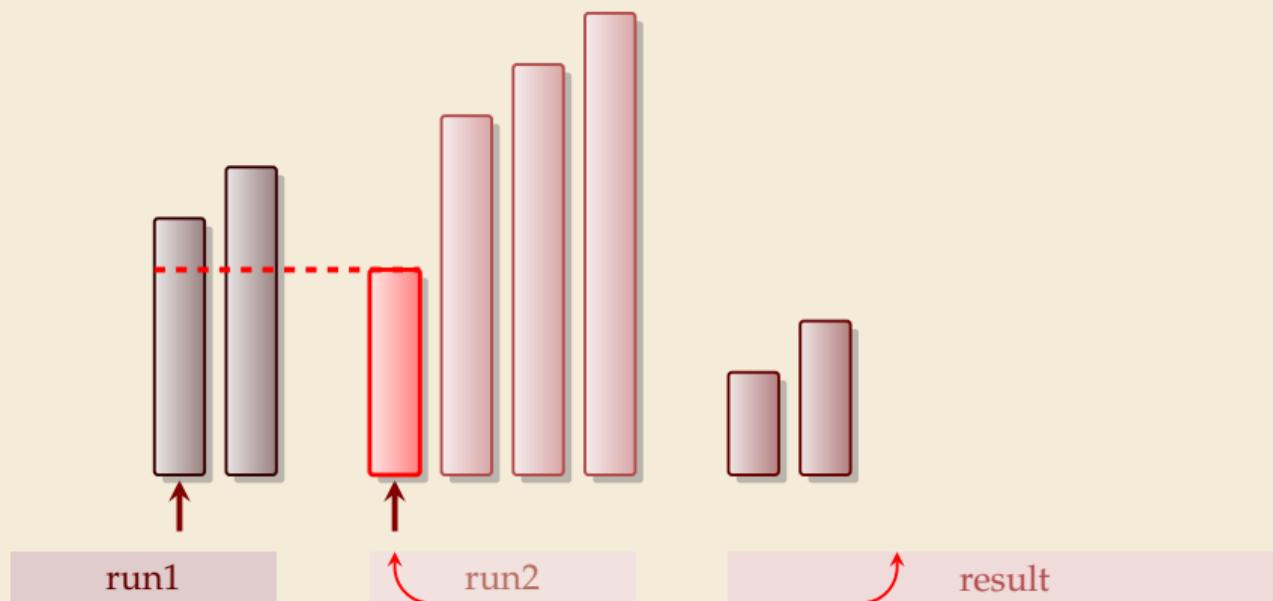
Merging sorted lists



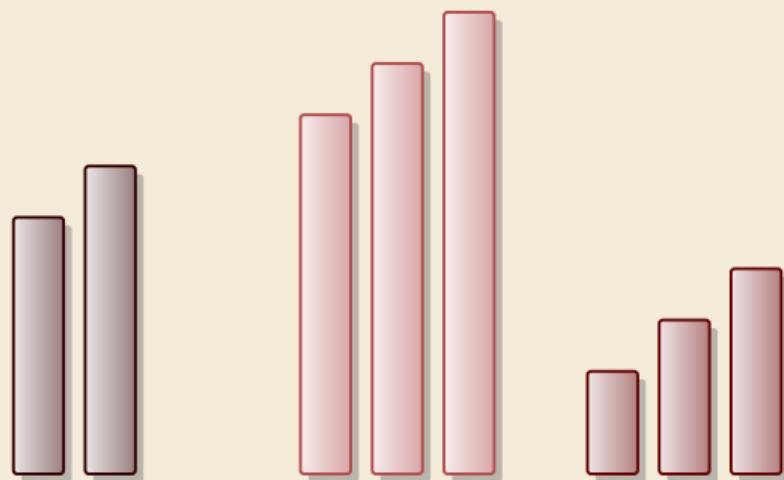
Merging sorted lists



Merging sorted lists



Merging sorted lists

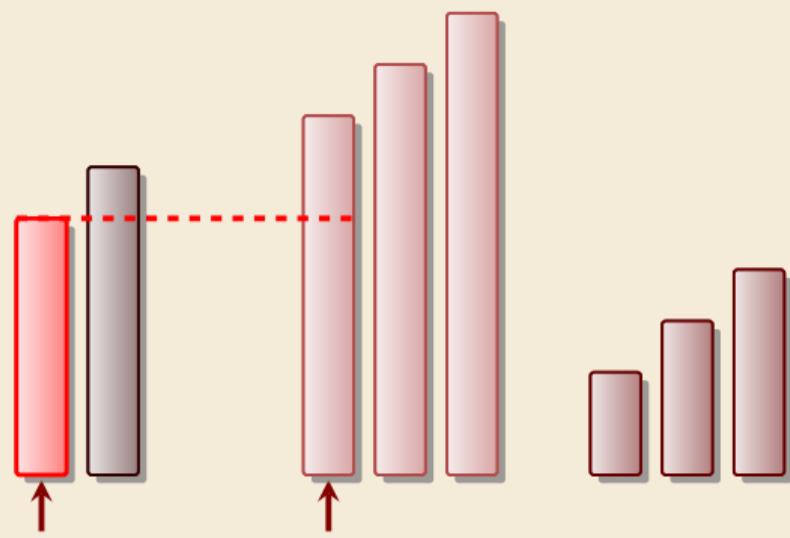


run1

run2

result

Merging sorted lists

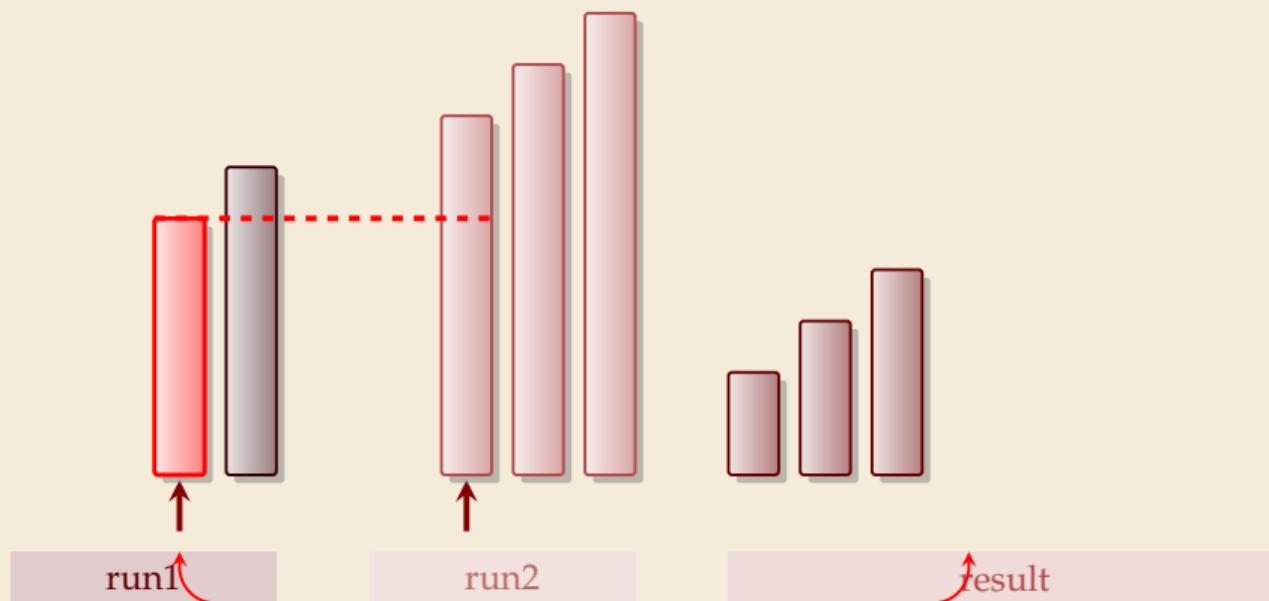


run1

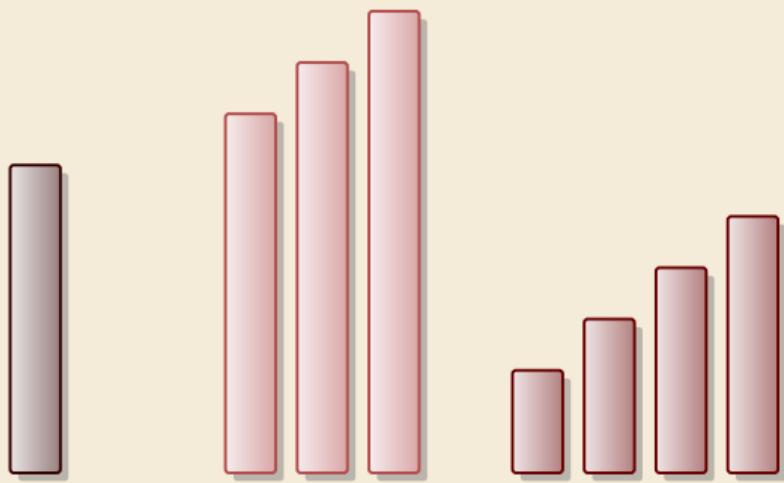
run2

result

Merging sorted lists



Merging sorted lists

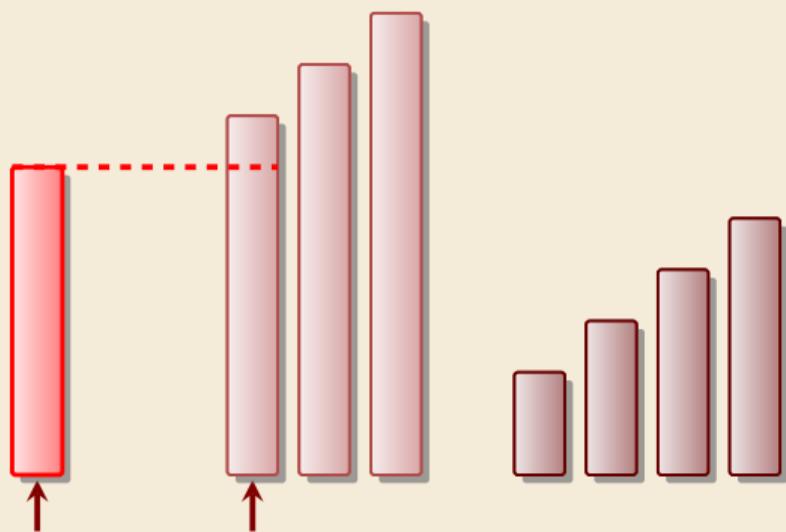


run1

run2

result

Merging sorted lists

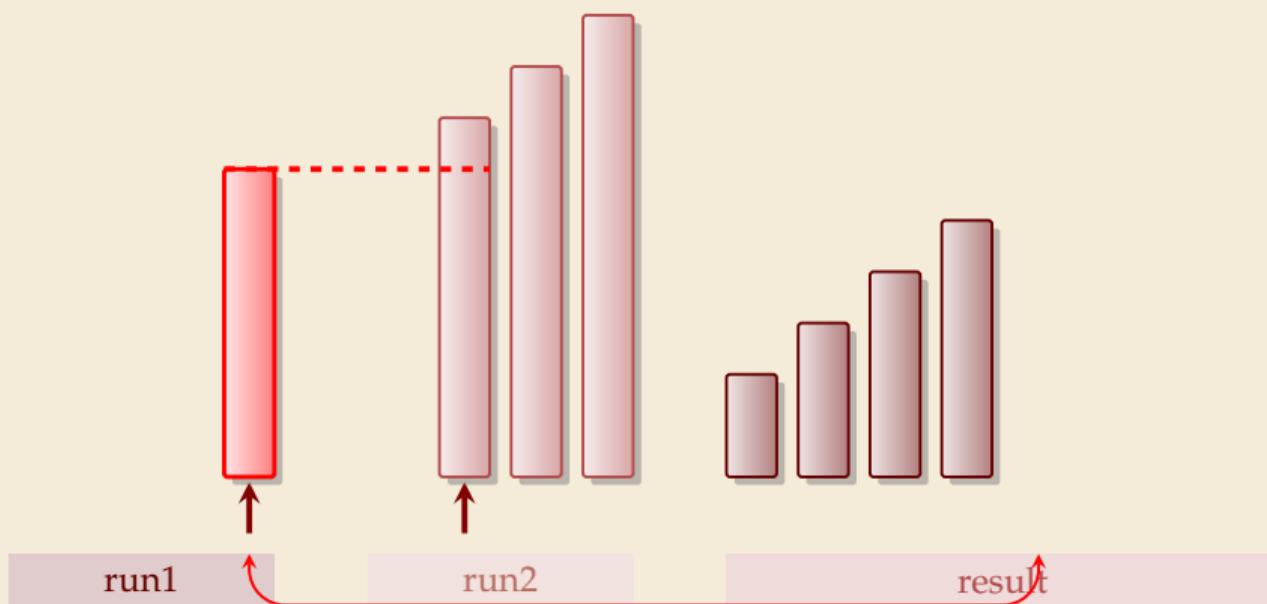


run1

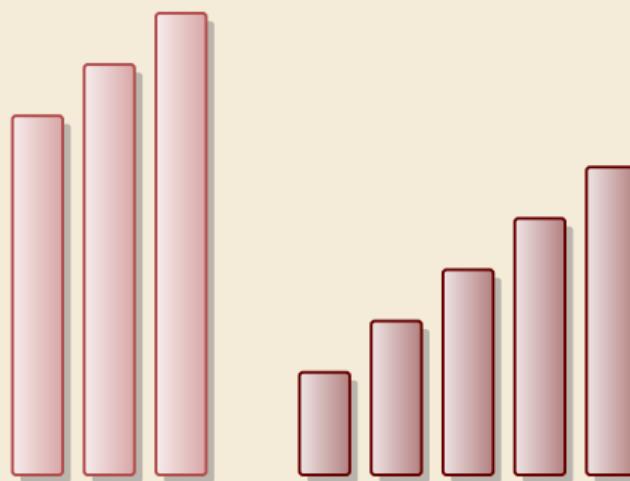
run2

result

Merging sorted lists



Merging sorted lists

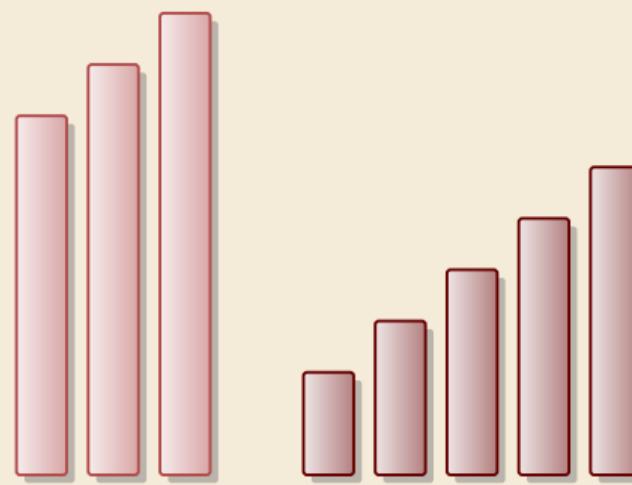


run1

run2

result

Merging sorted lists



run1

run2

result

Merging sorted lists

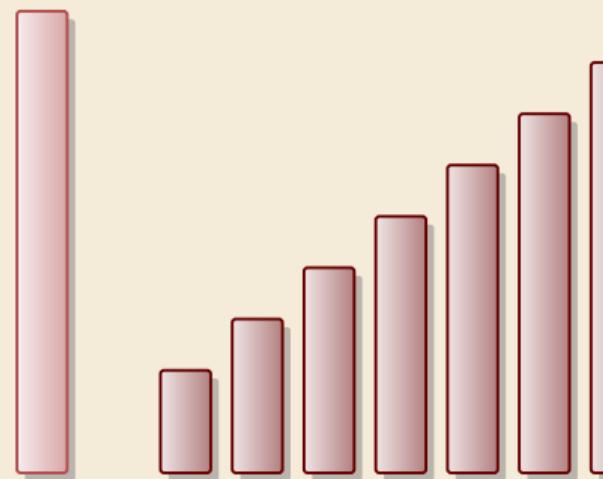


run1

run2

result

Merging sorted lists

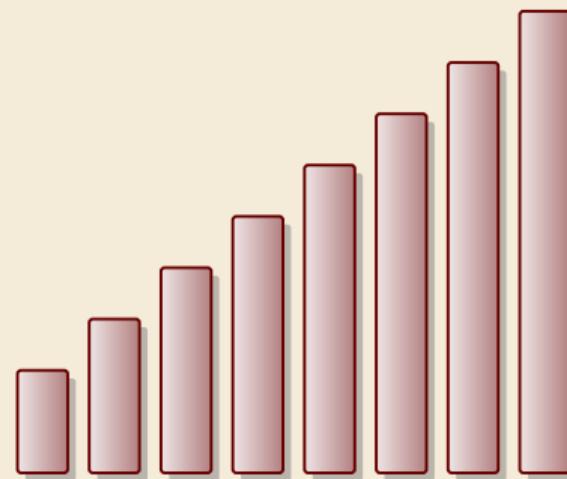


run1

run2

result

Merging sorted lists



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.~~

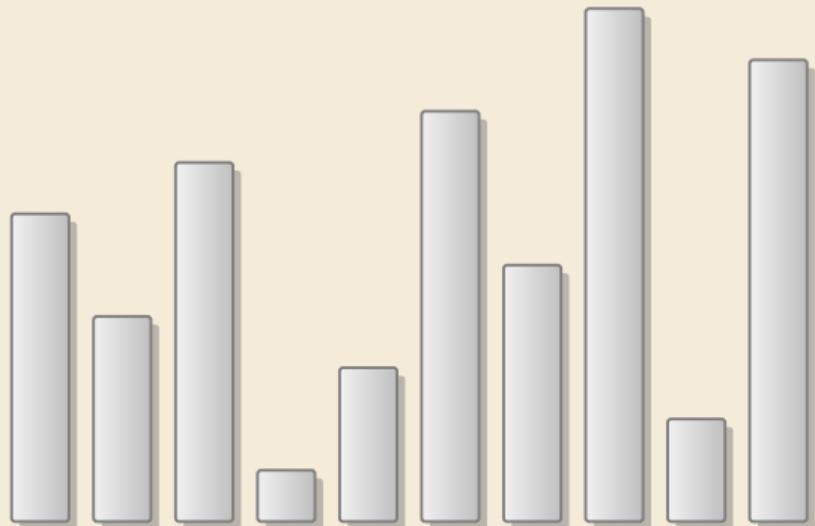
sli.do/comp526

Click on "Polls" tab

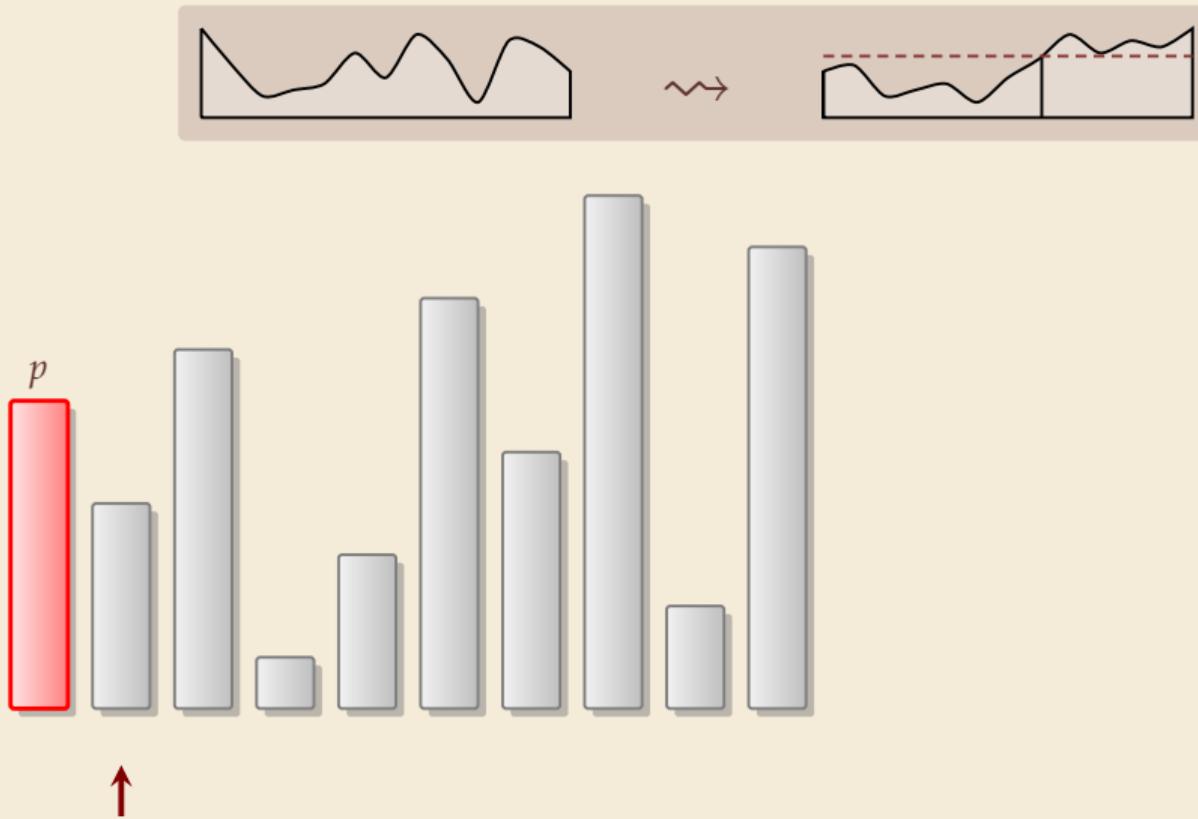
Partitioning around a pivot



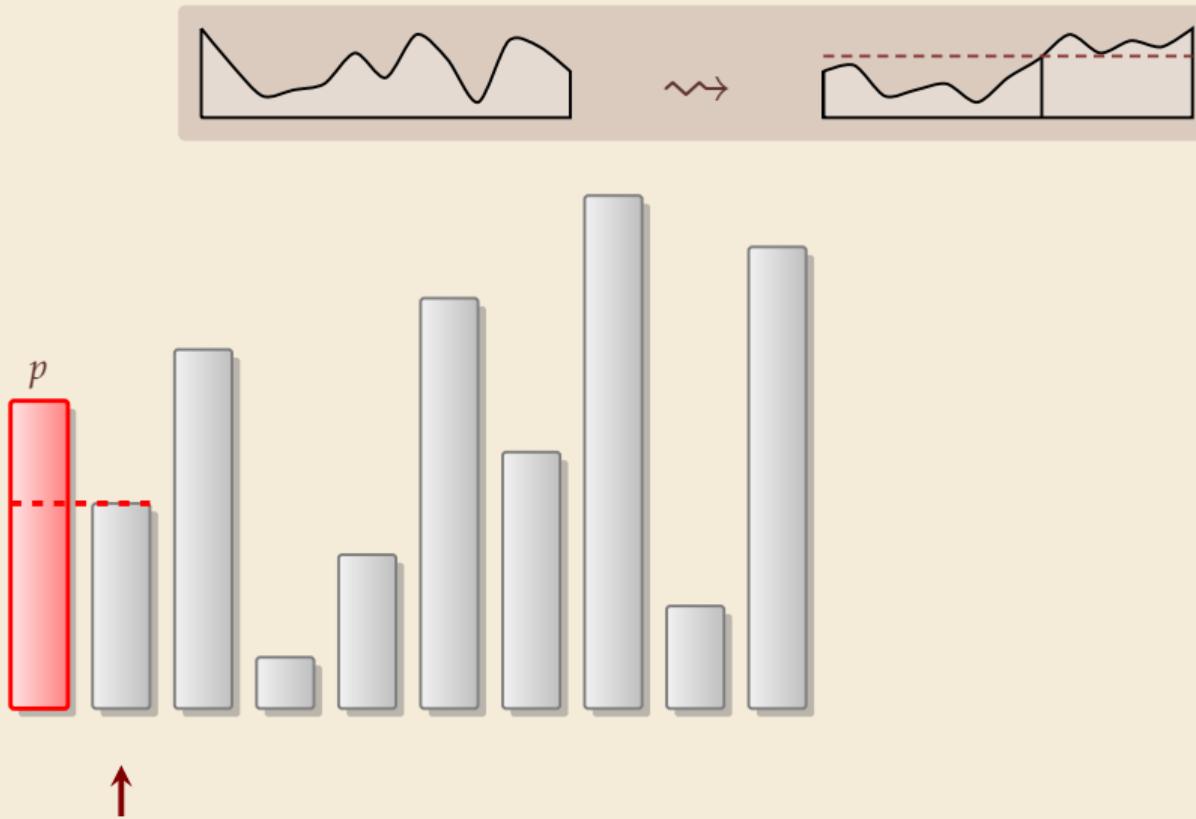
Partitioning around a pivot



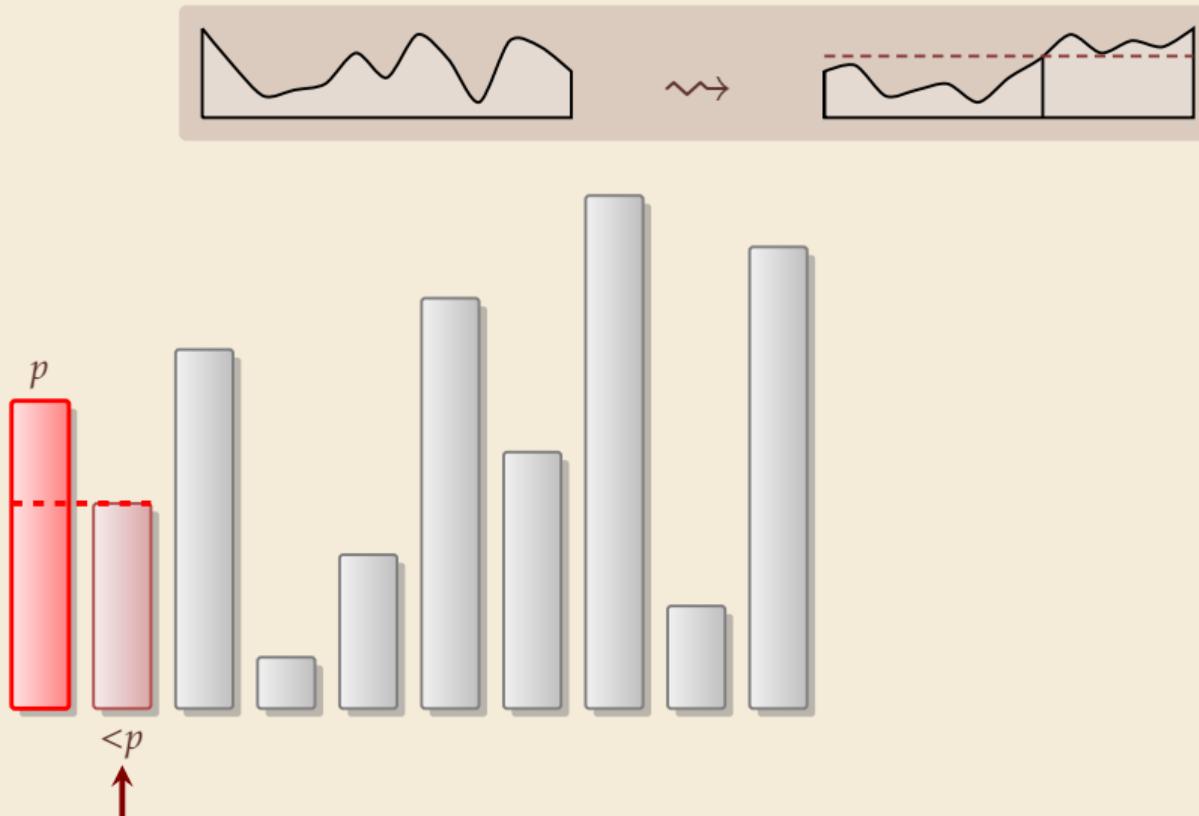
Partitioning around a pivot



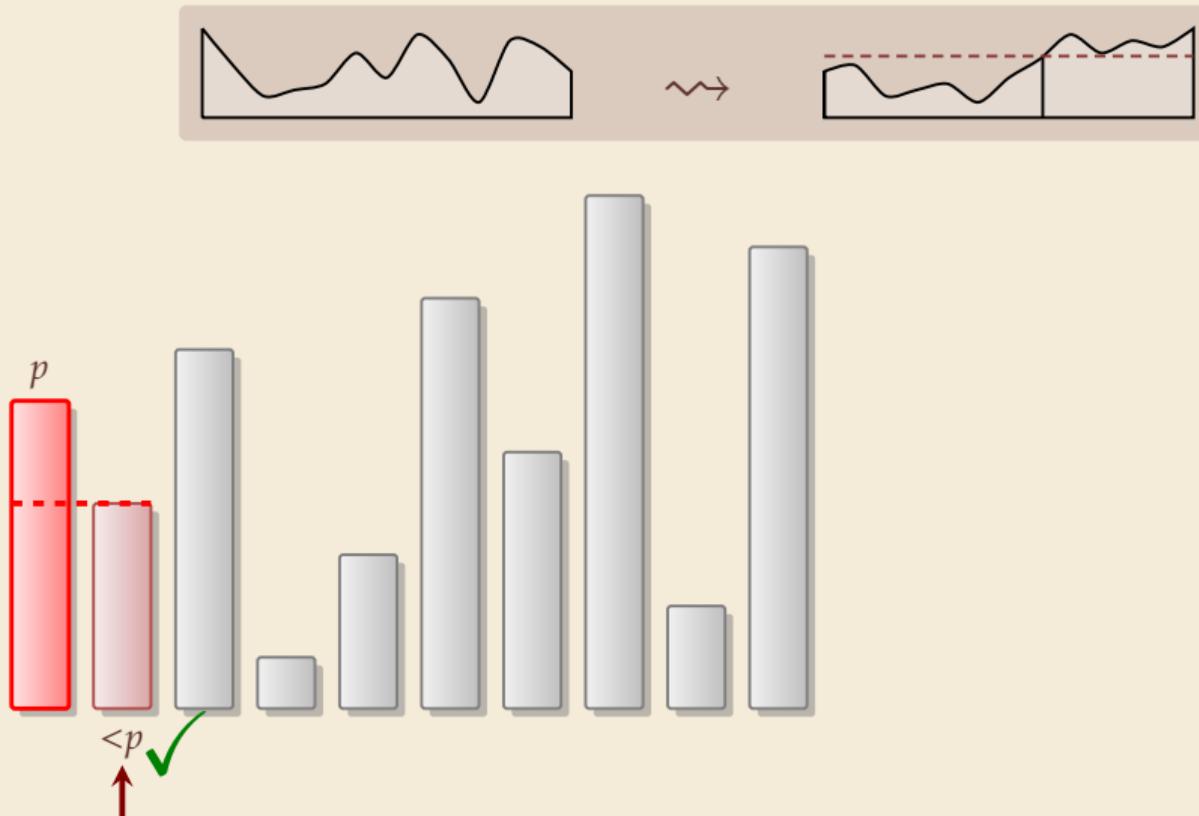
Partitioning around a pivot



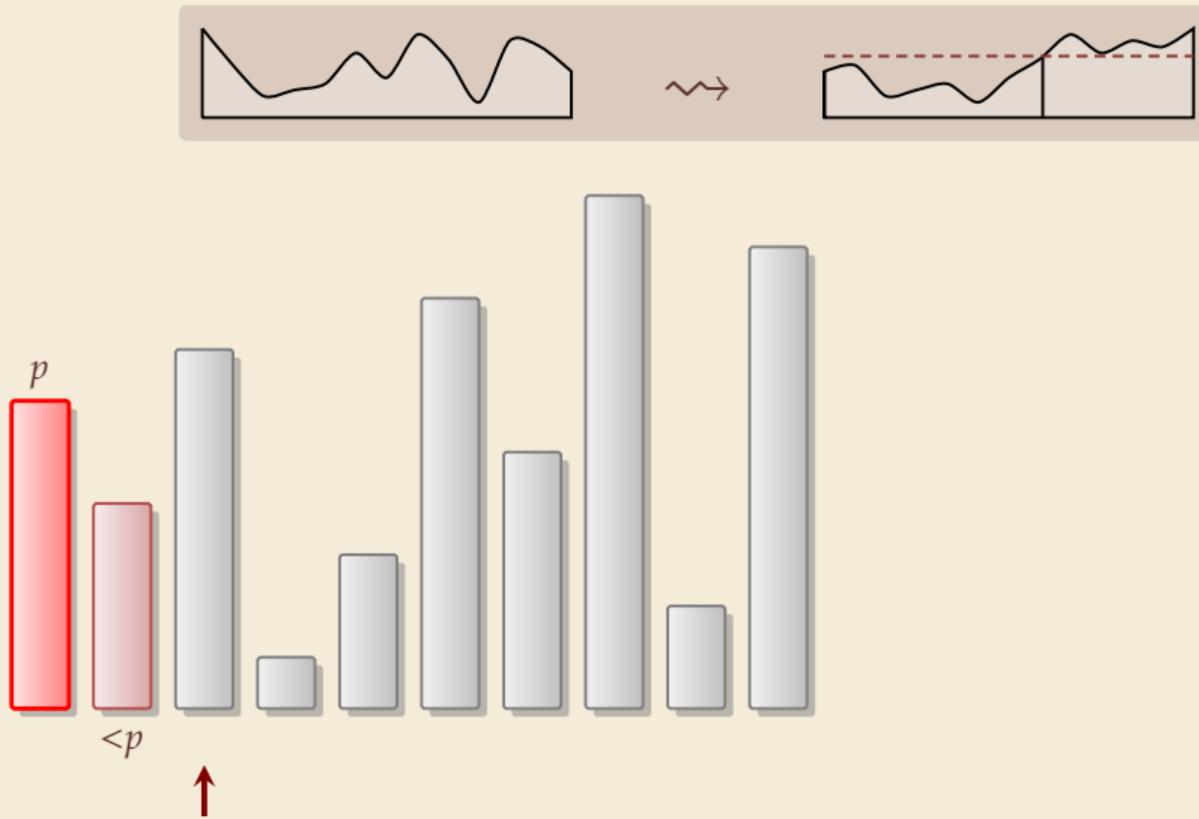
Partitioning around a pivot



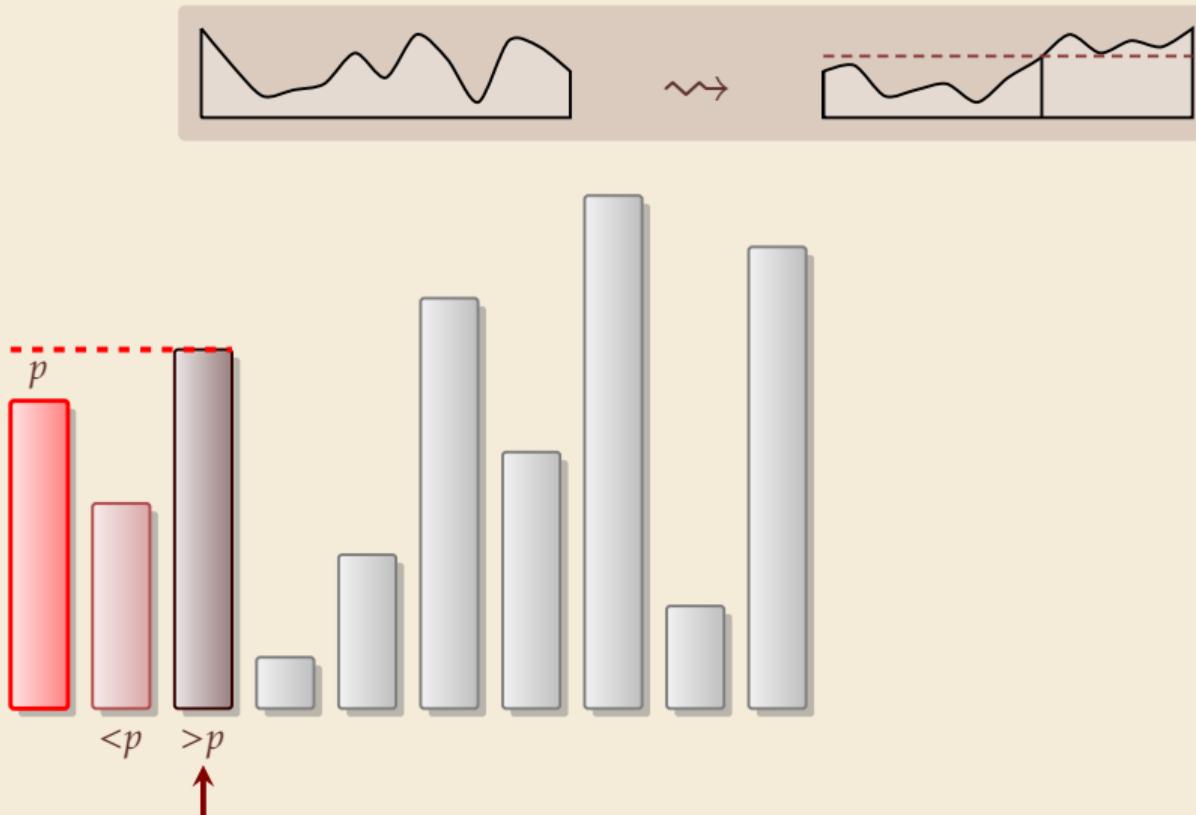
Partitioning around a pivot



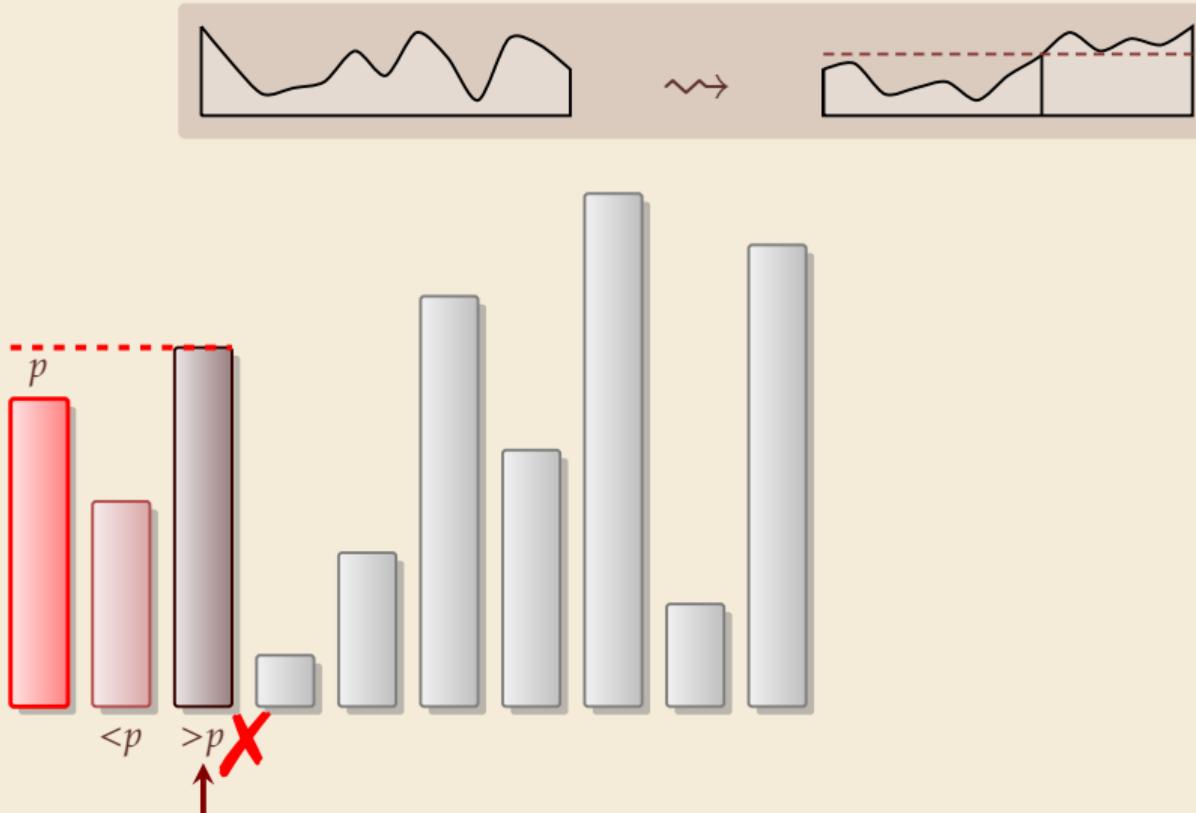
Partitioning around a pivot



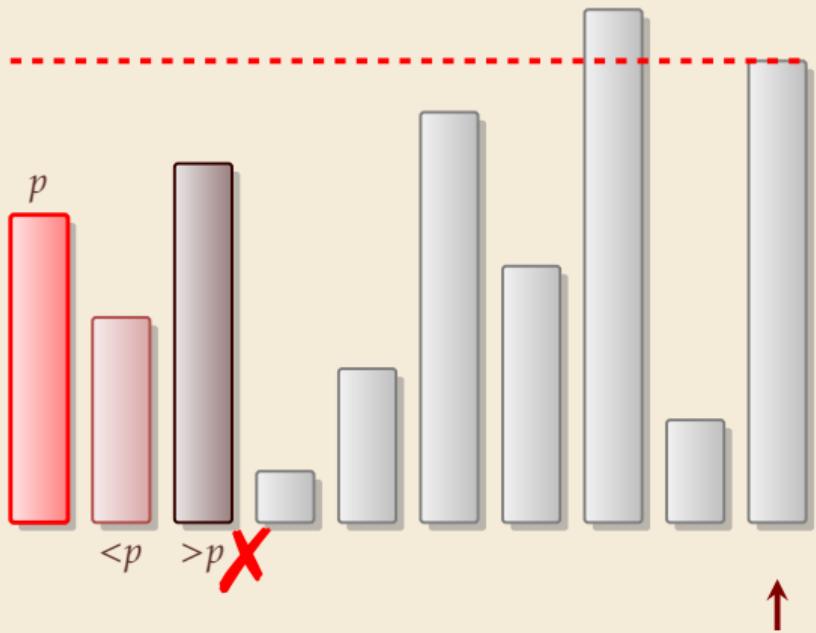
Partitioning around a pivot



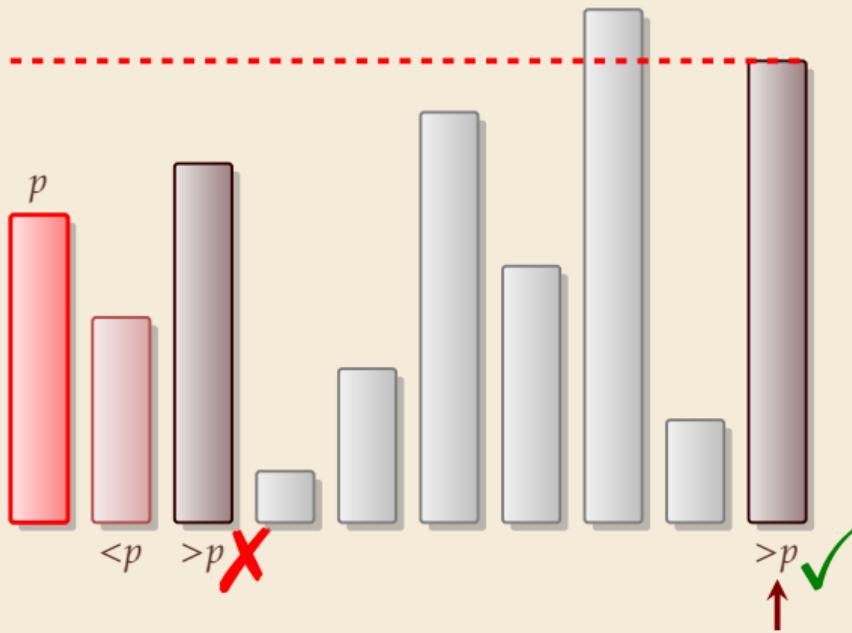
Partitioning around a pivot



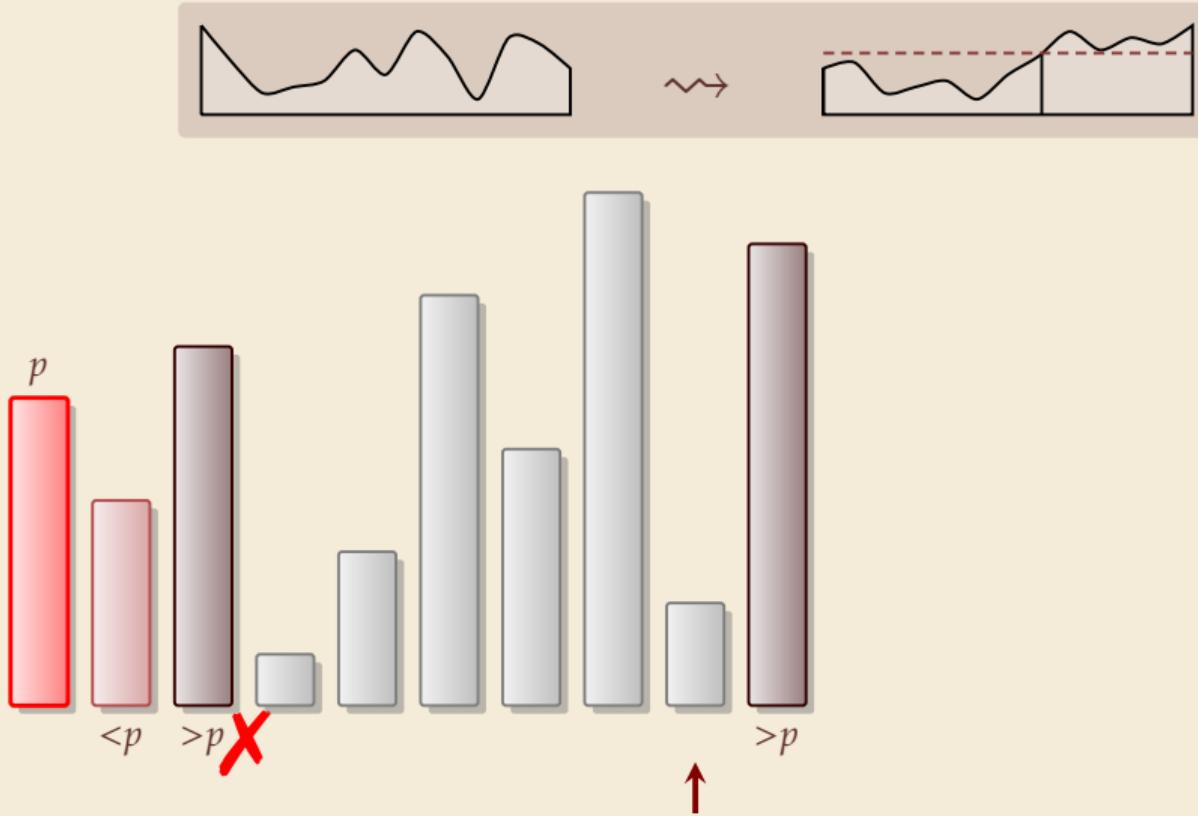
Partitioning around a pivot



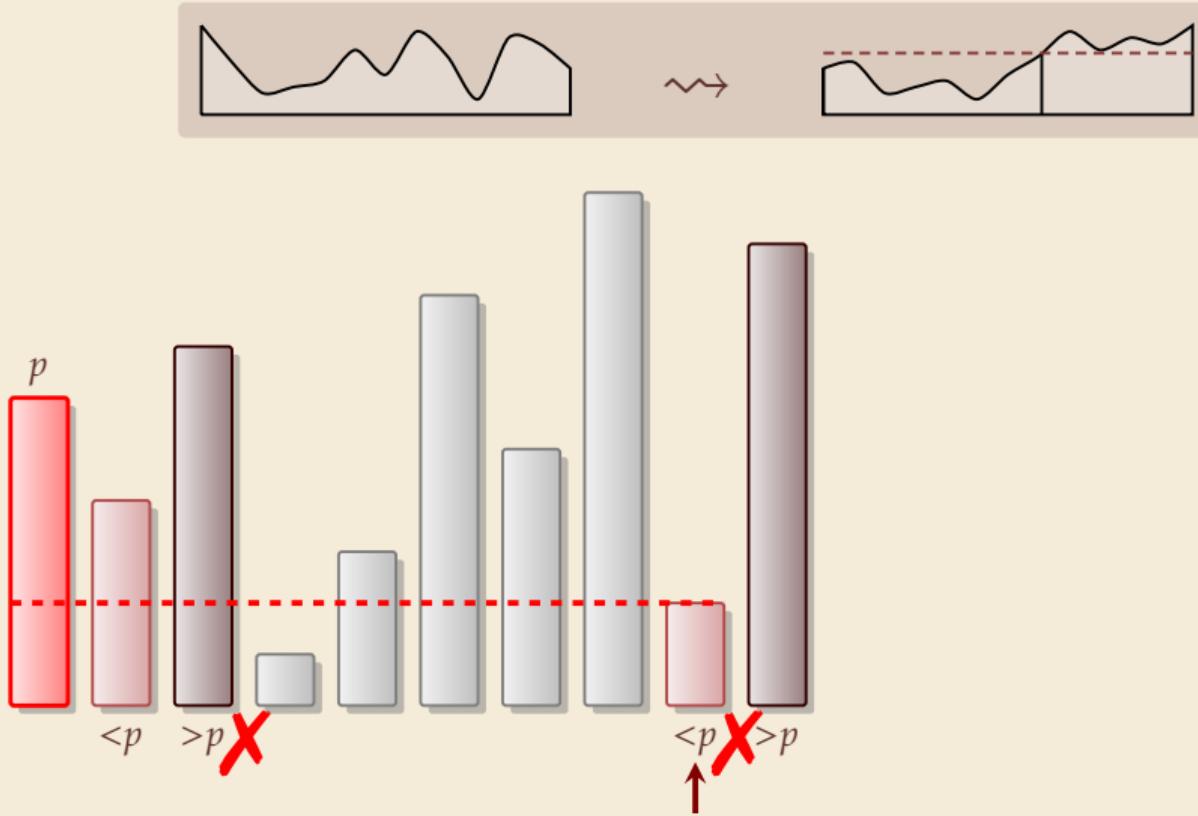
Partitioning around a pivot



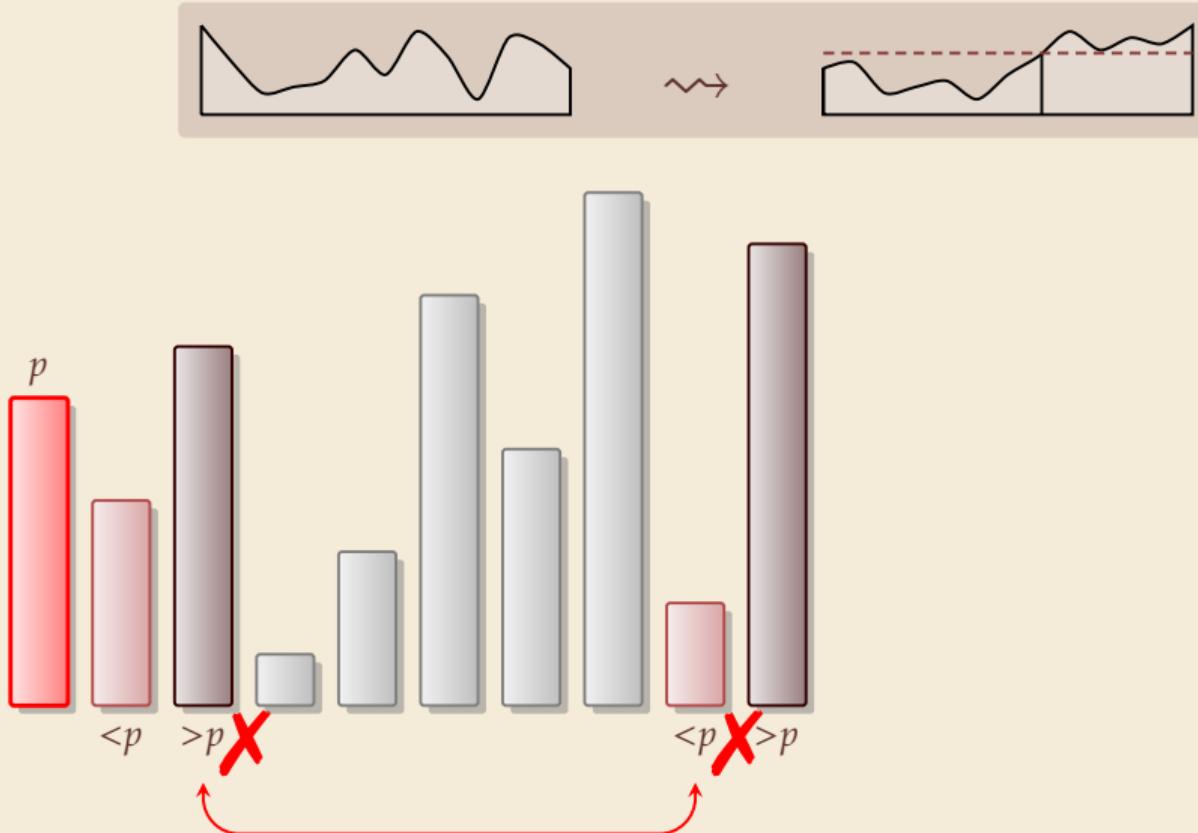
Partitioning around a pivot



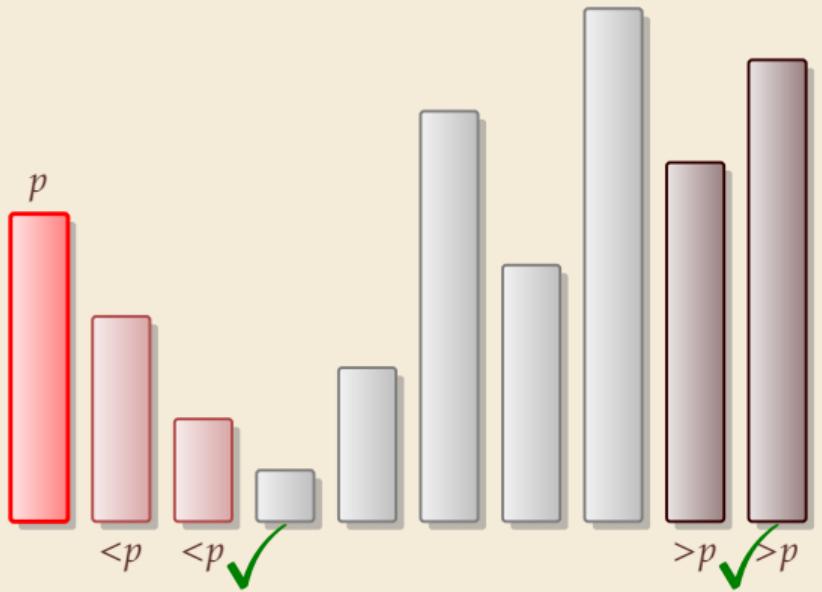
Partitioning around a pivot



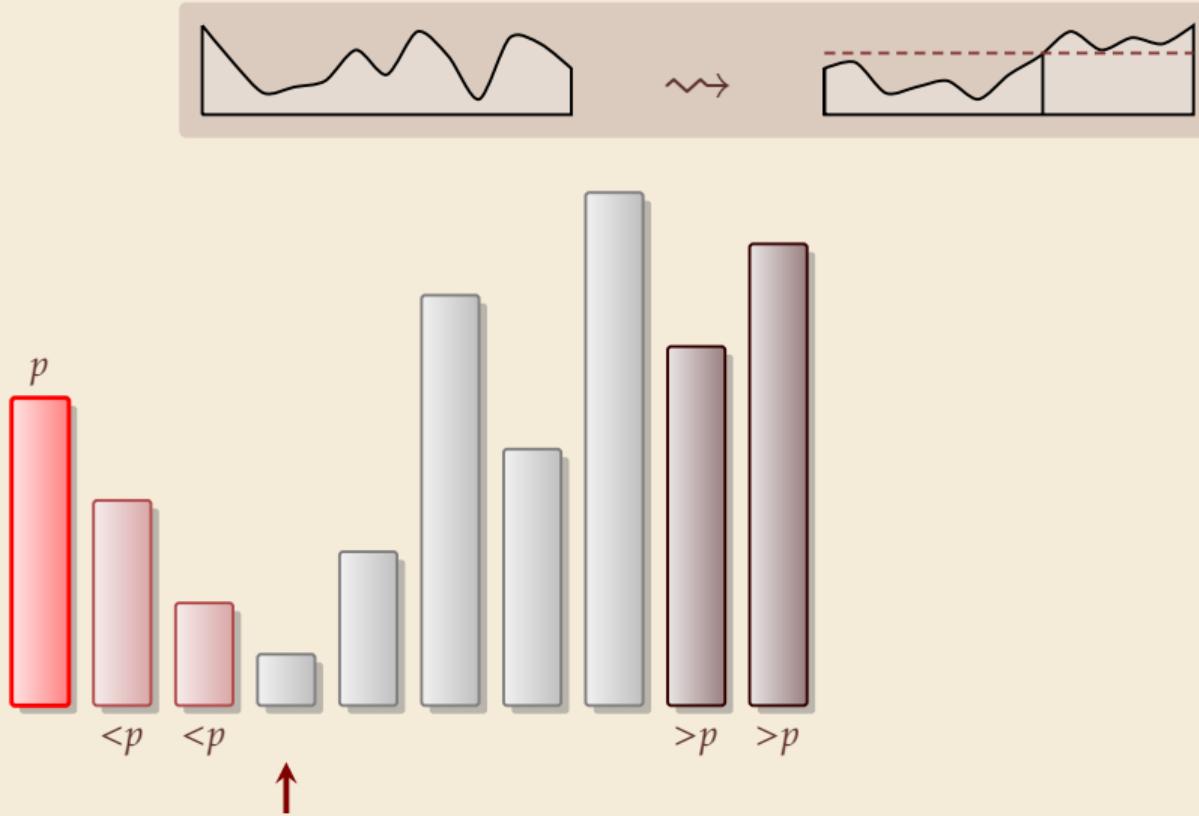
Partitioning around a pivot



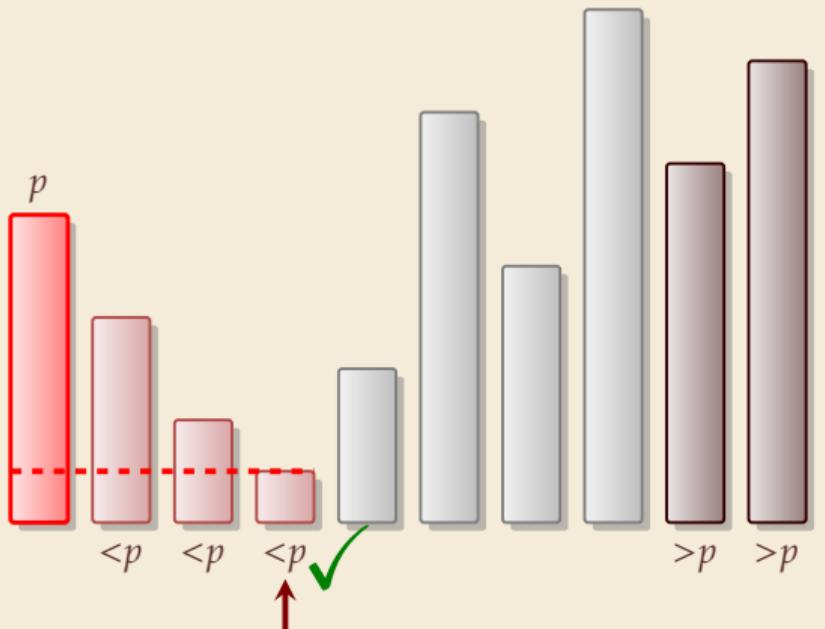
Partitioning around a pivot



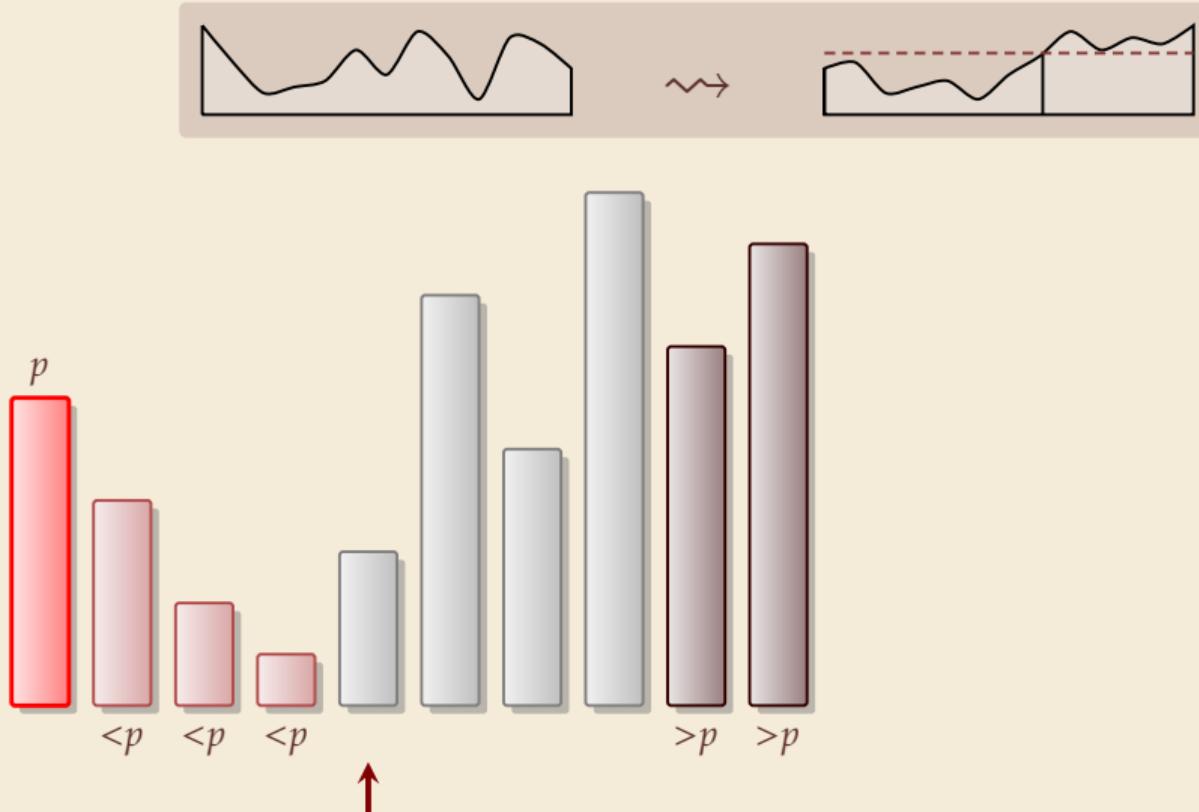
Partitioning around a pivot



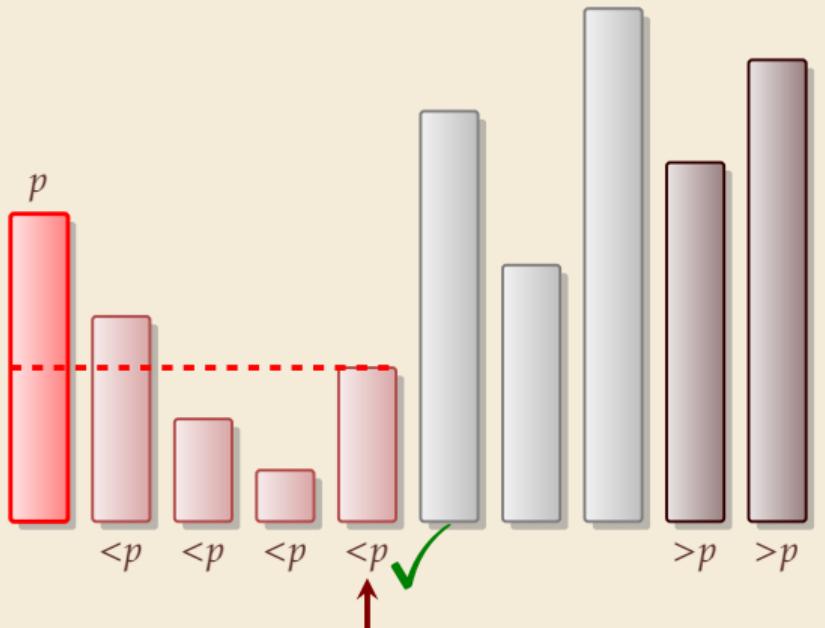
Partitioning around a pivot



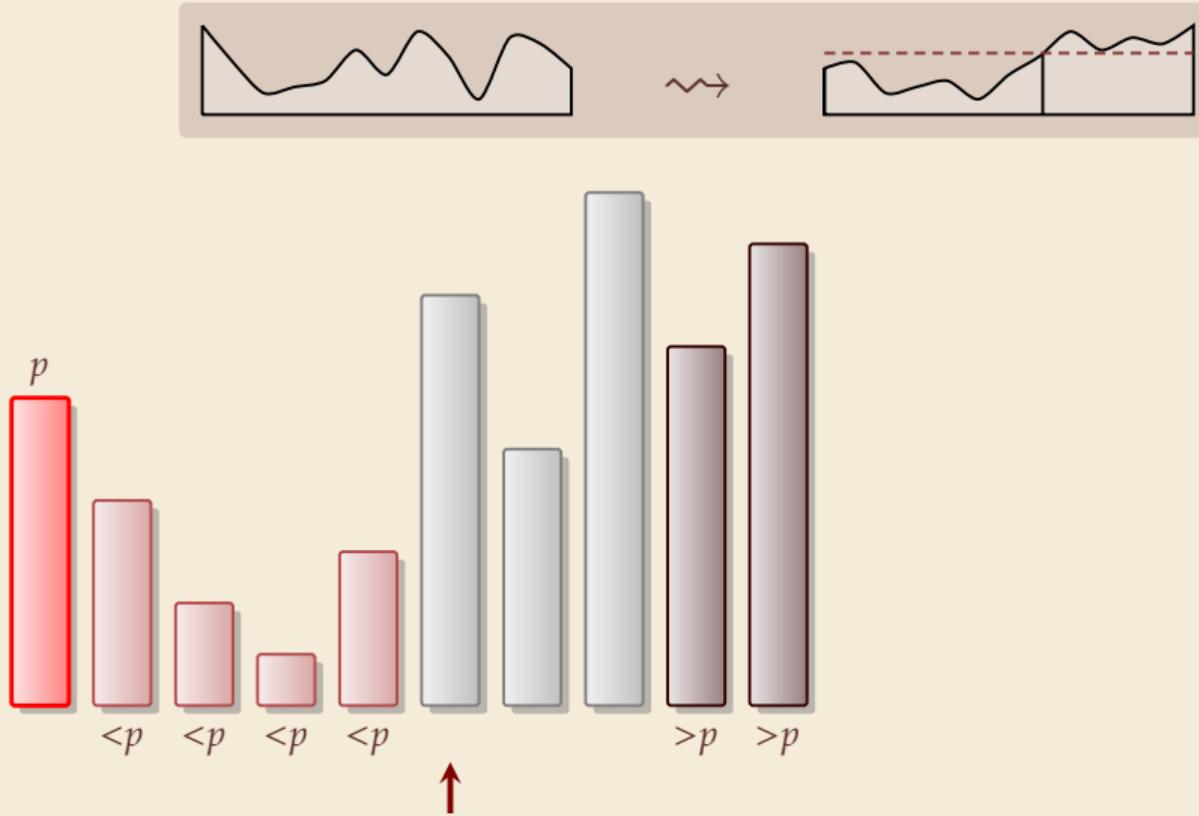
Partitioning around a pivot



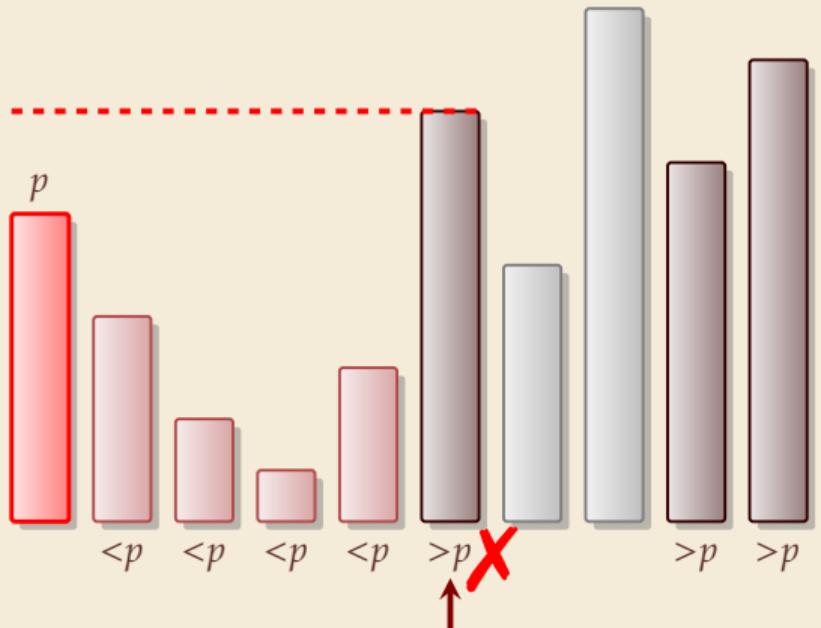
Partitioning around a pivot



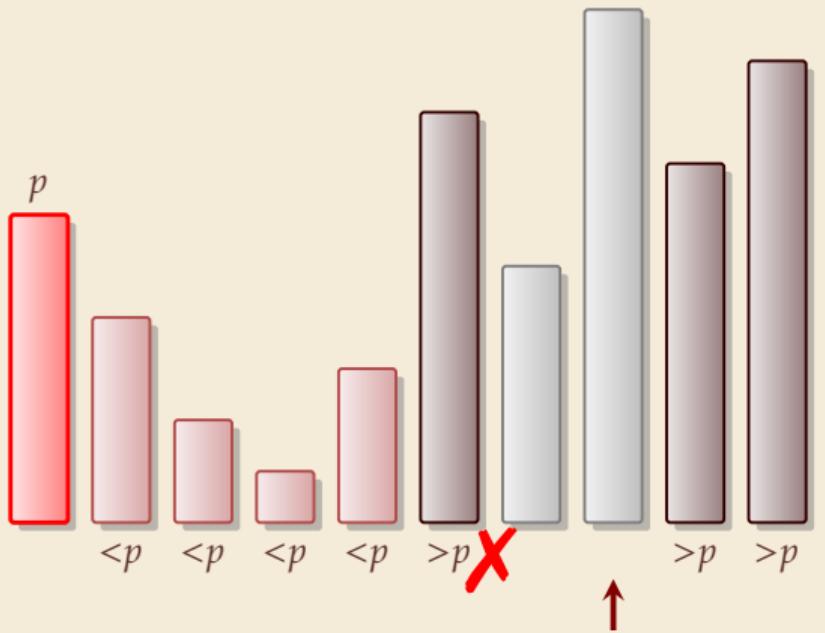
Partitioning around a pivot



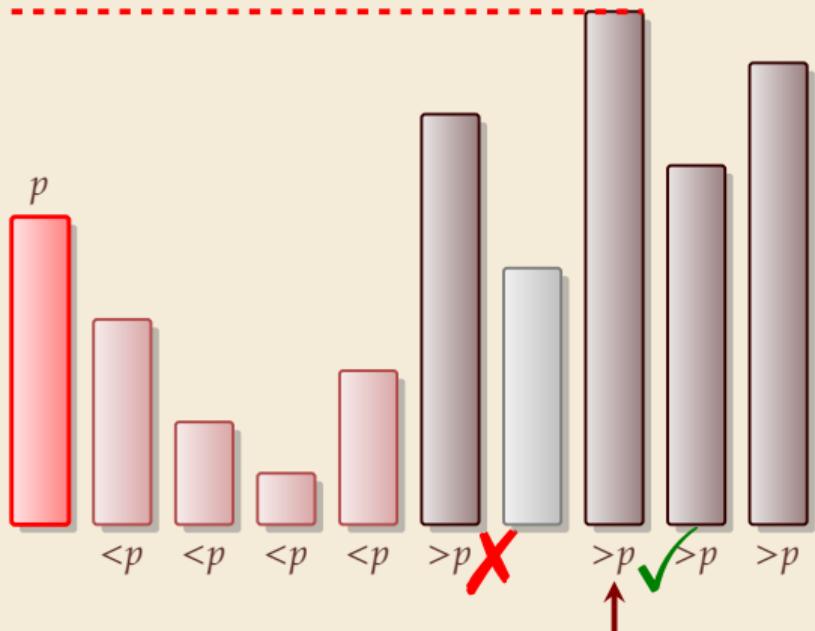
Partitioning around a pivot



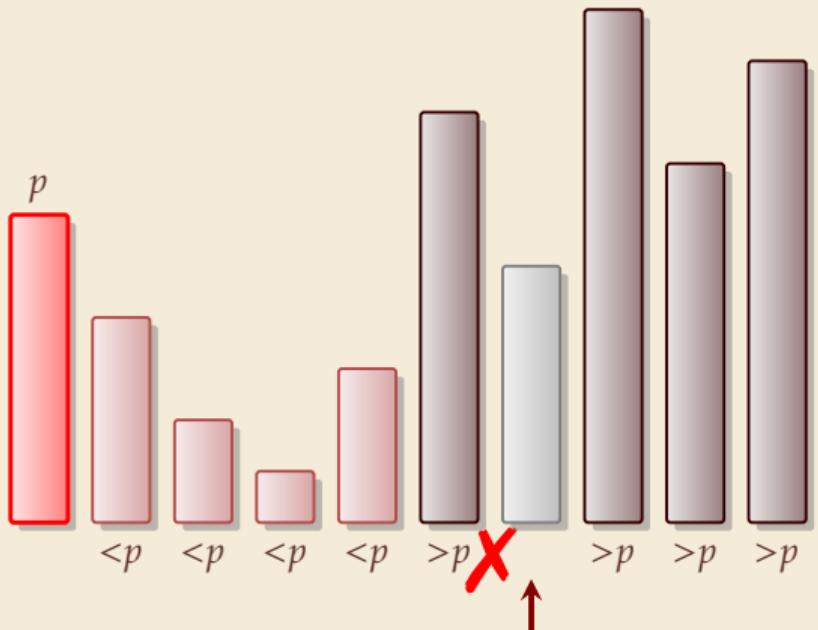
Partitioning around a pivot



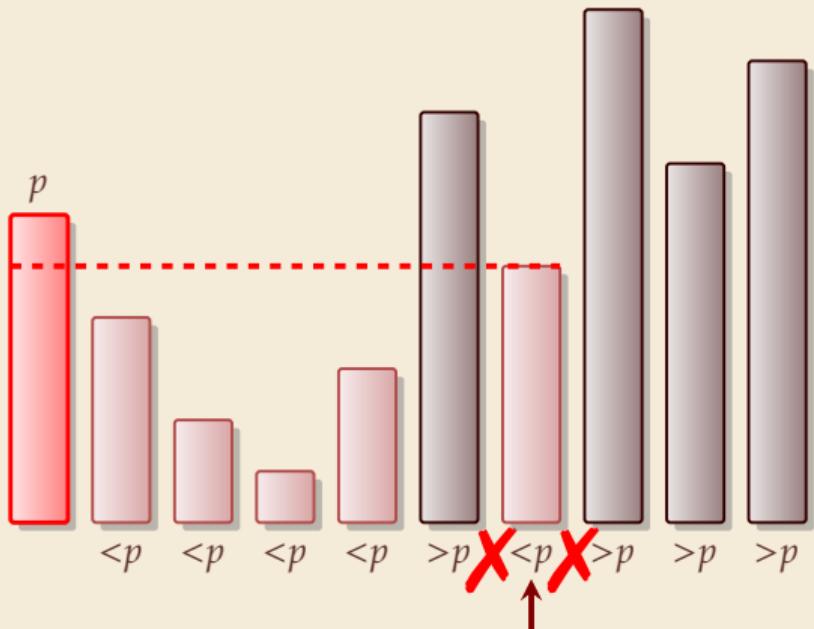
Partitioning around a pivot



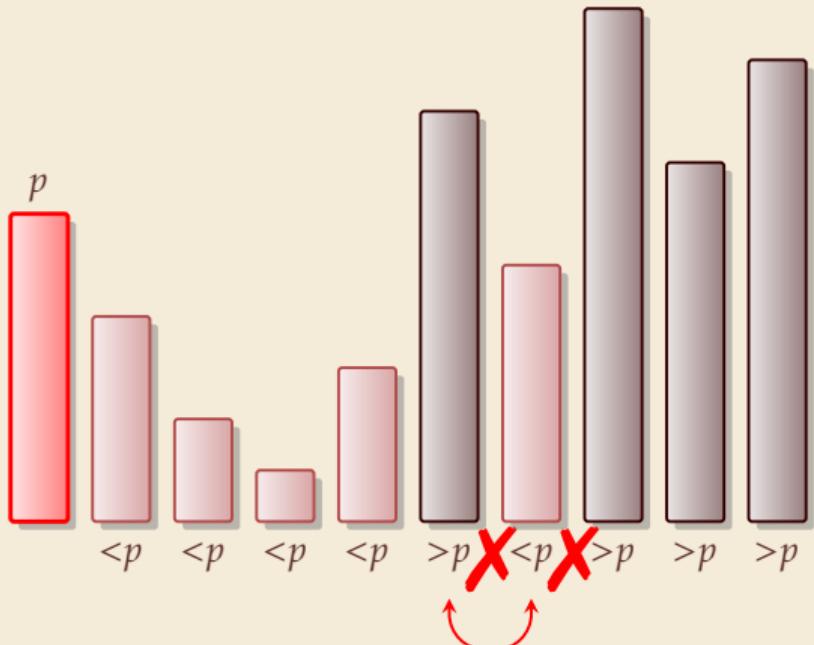
Partitioning around a pivot



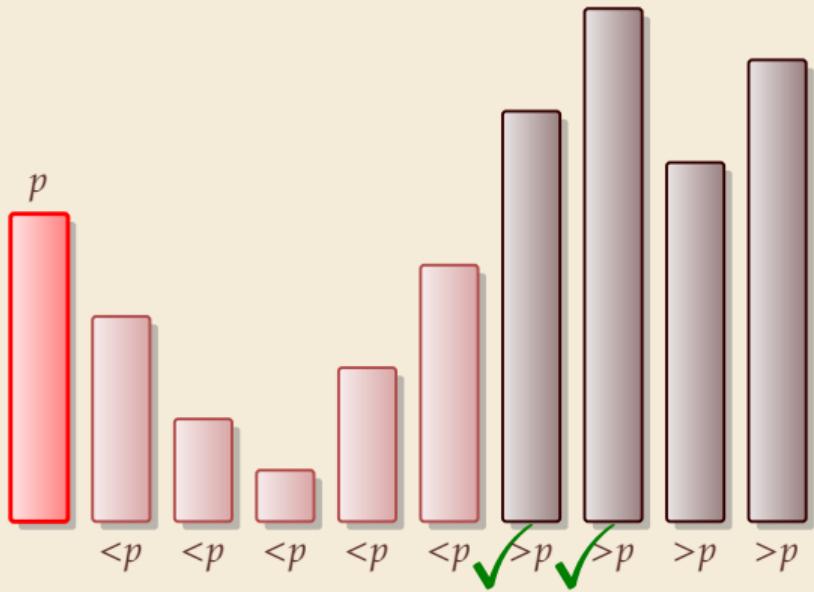
Partitioning around a pivot



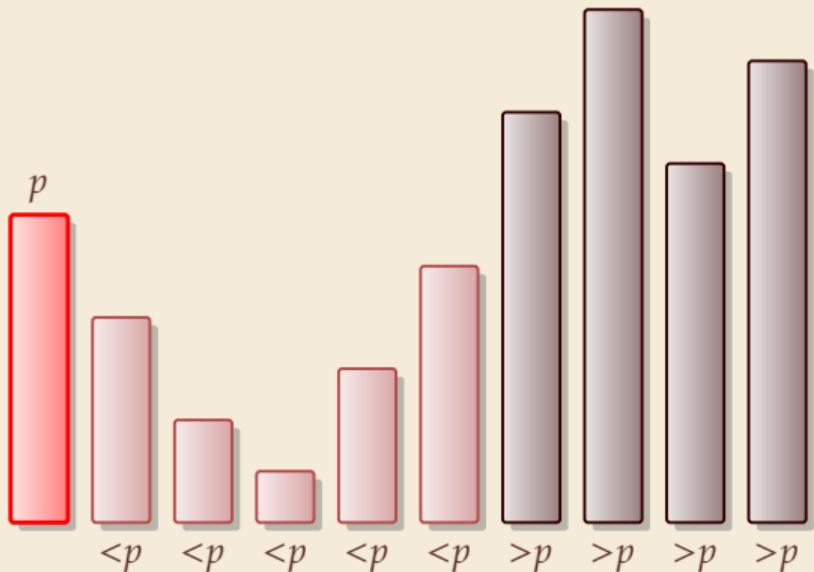
Partitioning around a pivot



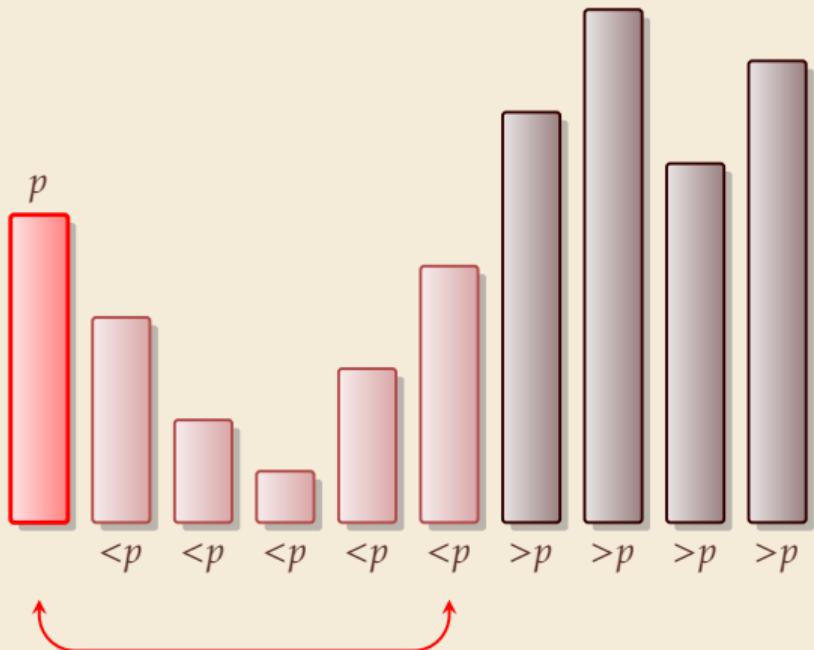
Partitioning around a pivot



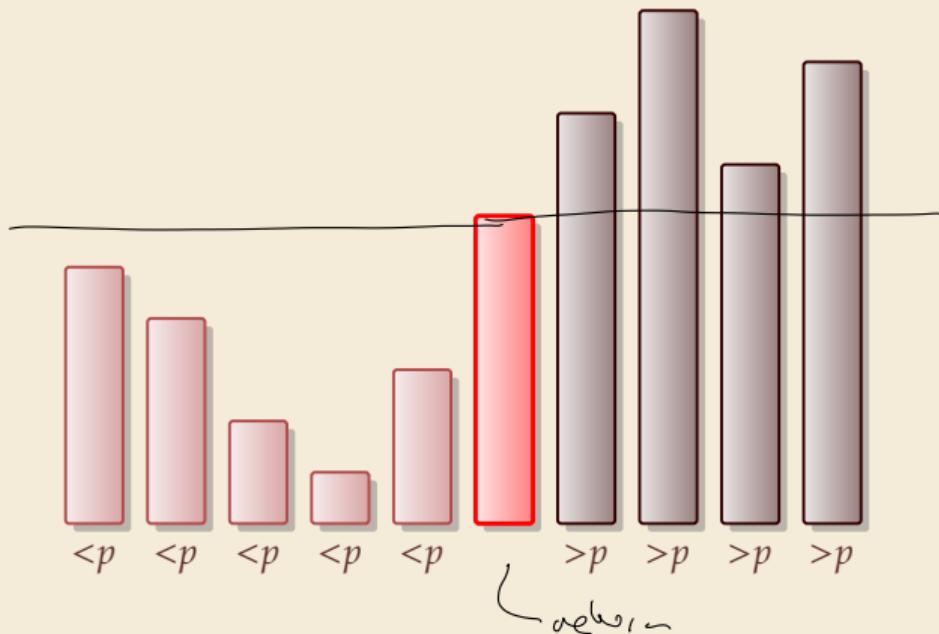
Partitioning around a pivot



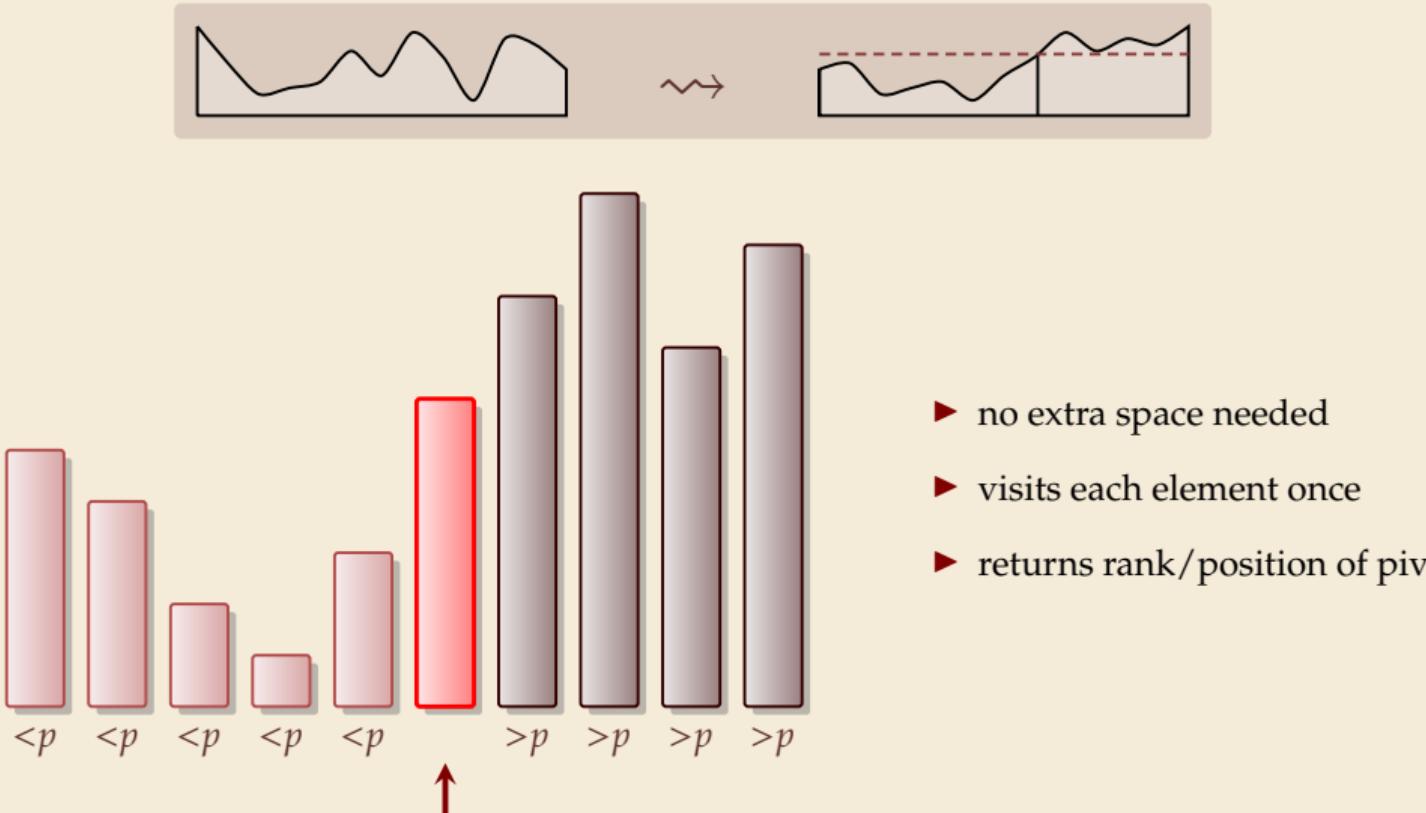
Partitioning around a pivot



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
---	---	---	---	---	---	---	---	---

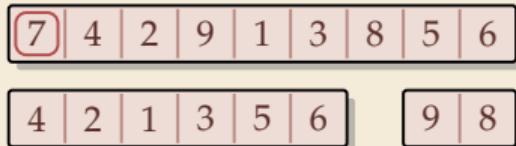
Quicksort & Binary Search Trees

Quicksort

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

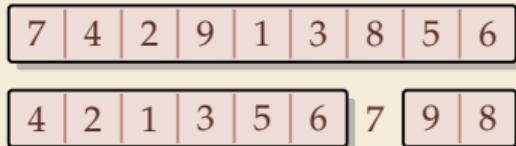
Quicksort & Binary Search Trees

Quicksort



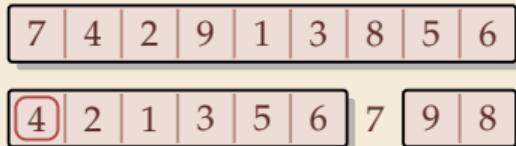
Quicksort & Binary Search Trees

Quicksort



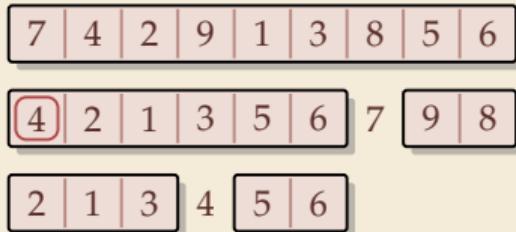
Quicksort & Binary Search Trees

Quicksort



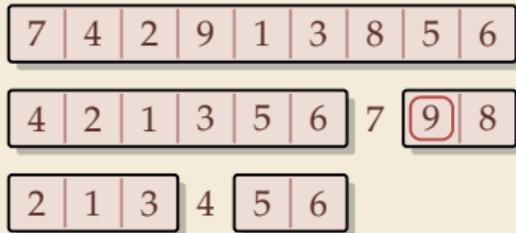
Quicksort & Binary Search Trees

Quicksort



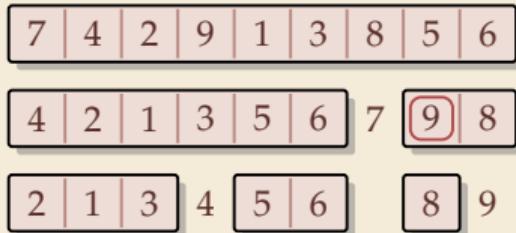
Quicksort & Binary Search Trees

Quicksort



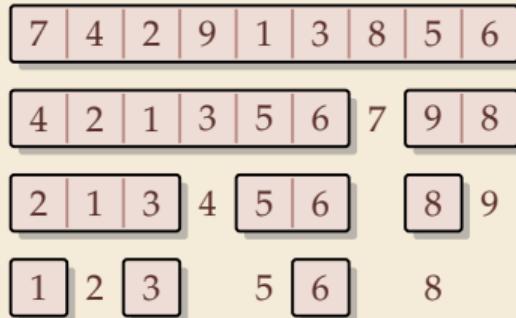
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort



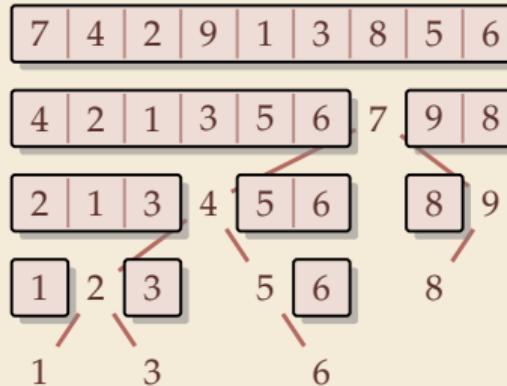
Quicksort & Binary Search Trees

Quicksort



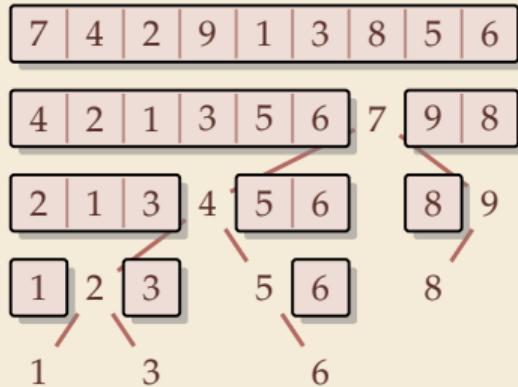
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

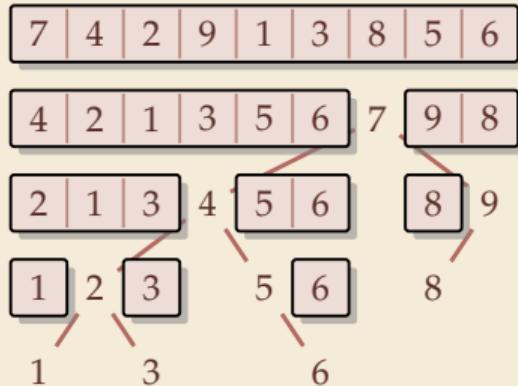


Binary Search Tree (BST)

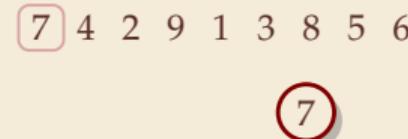
7 4 2 9 1 3 8 5 6

Quicksort & Binary Search Trees

Quicksort

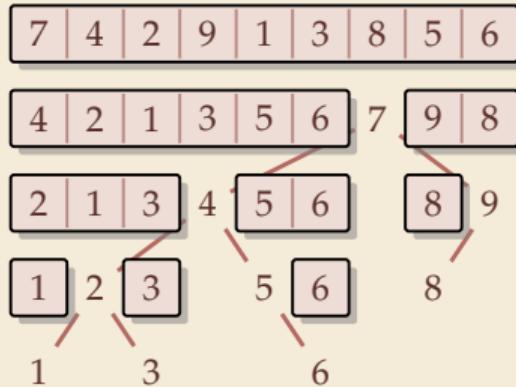


Binary Search Tree (BST)

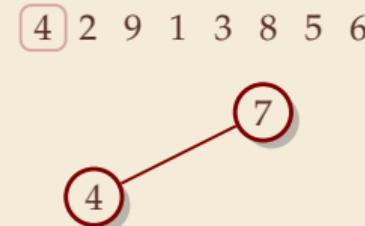


Quicksort & Binary Search Trees

Quicksort

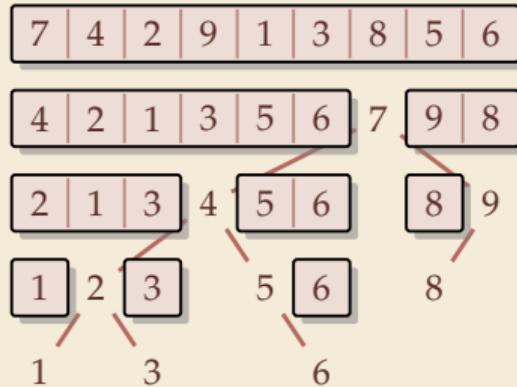


Binary Search Tree (BST)

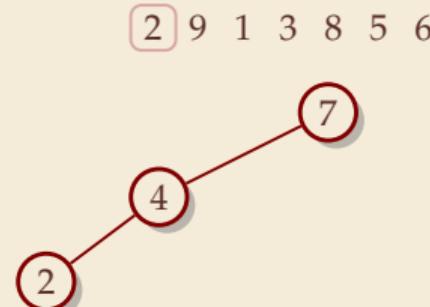


Quicksort & Binary Search Trees

Quicksort

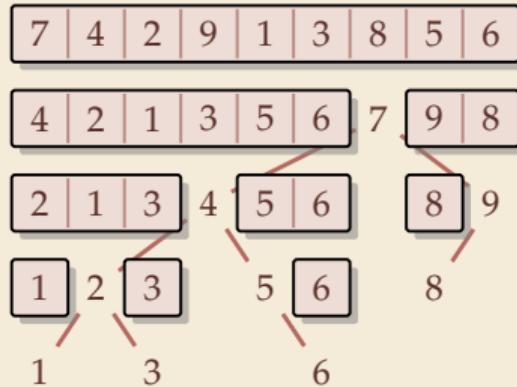


Binary Search Tree (BST)

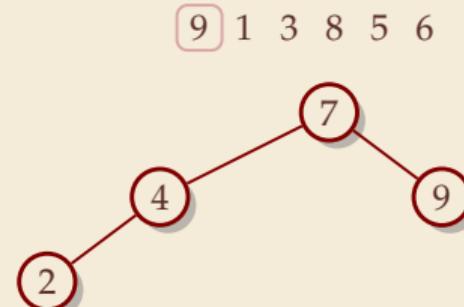


Quicksort & Binary Search Trees

Quicksort

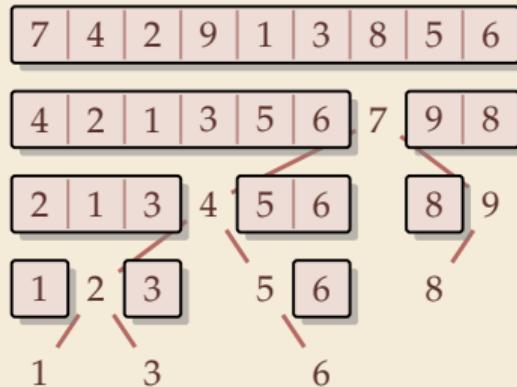


Binary Search Tree (BST)

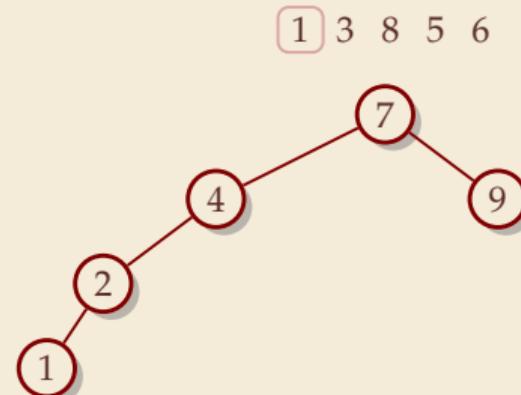


Quicksort & Binary Search Trees

Quicksort

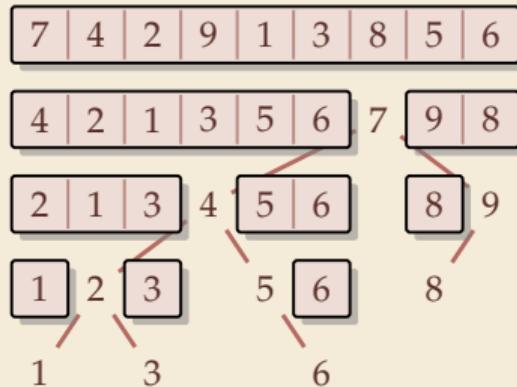


Binary Search Tree (BST)

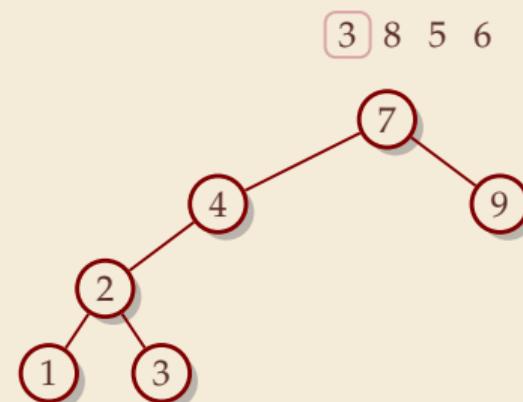


Quicksort & Binary Search Trees

Quicksort

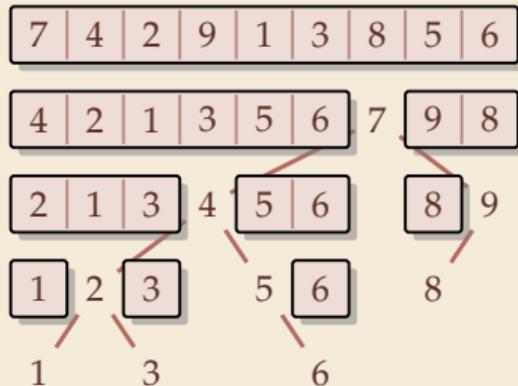


Binary Search Tree (BST)

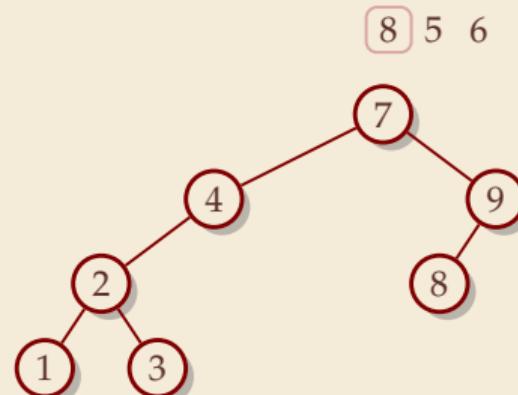


Quicksort & Binary Search Trees

Quicksort

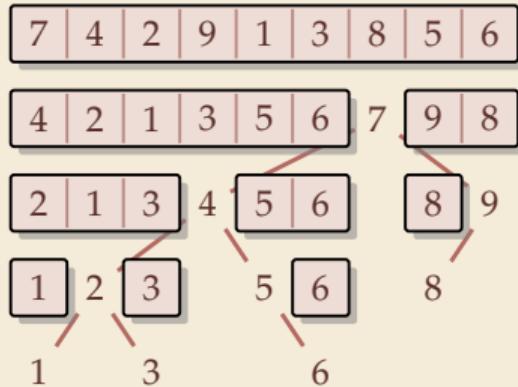


Binary Search Tree (BST)

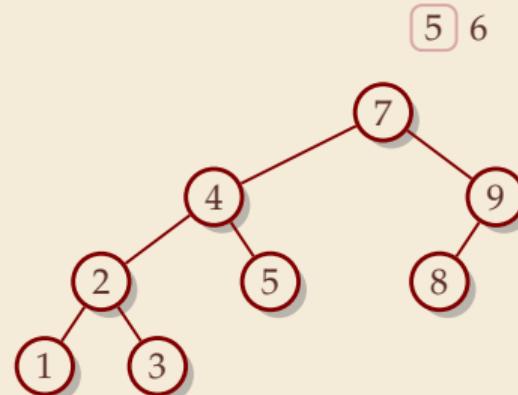


Quicksort & Binary Search Trees

Quicksort

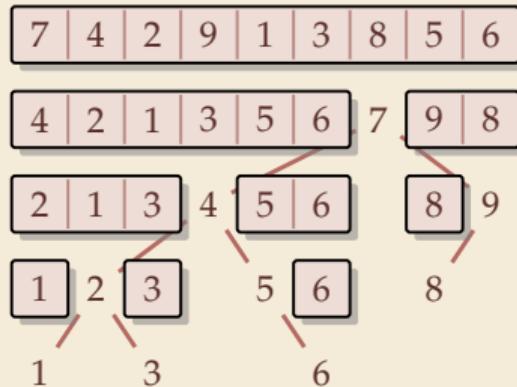


Binary Search Tree (BST)

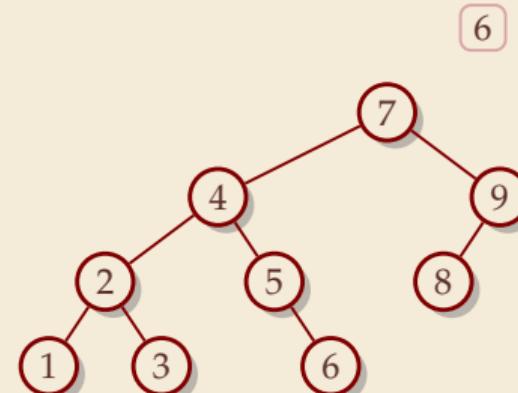


Quicksort & Binary Search Trees

Quicksort

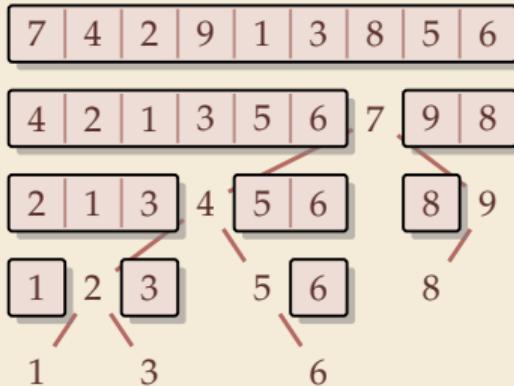


Binary Search Tree (BST)

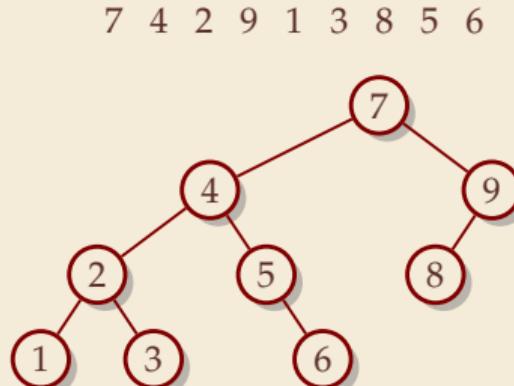


Quicksort & Binary Search Trees

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

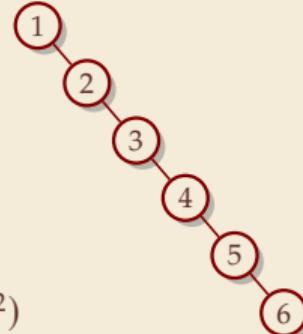
- ▶ Problem: BSTs can degenerate

- ▶ 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.