3 Efficient Sorting – The Power of Divide & Conquer

17 October 2022

Sebastian Wild

COMP526 (Fall 2022)
University of Liverpool
version `2022-10-17 13:29`

Learning Outcomes

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.

Unit 3: *Efficient Sorting*

Outline

3 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~~ → Unit 5

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

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

→ *sli.do/comp526*

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

Clicker Question

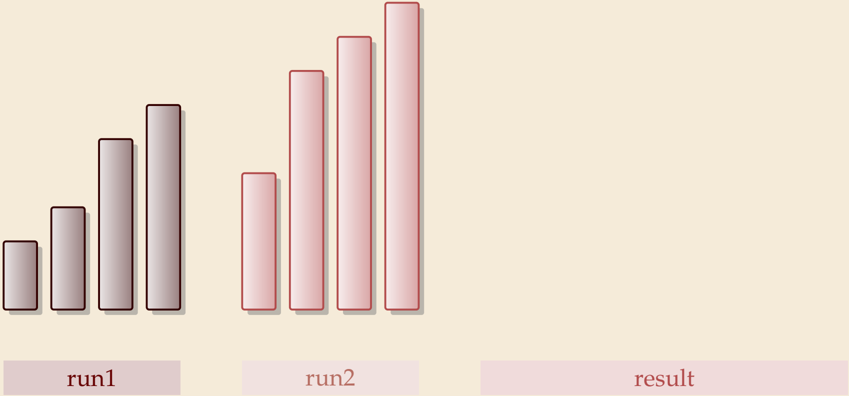
How does mergesort work?

- **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`

Merging sorted lists

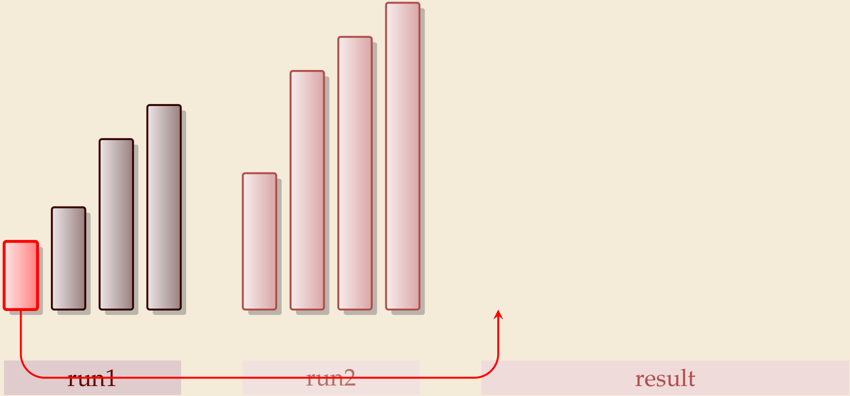
Merging sorted lists



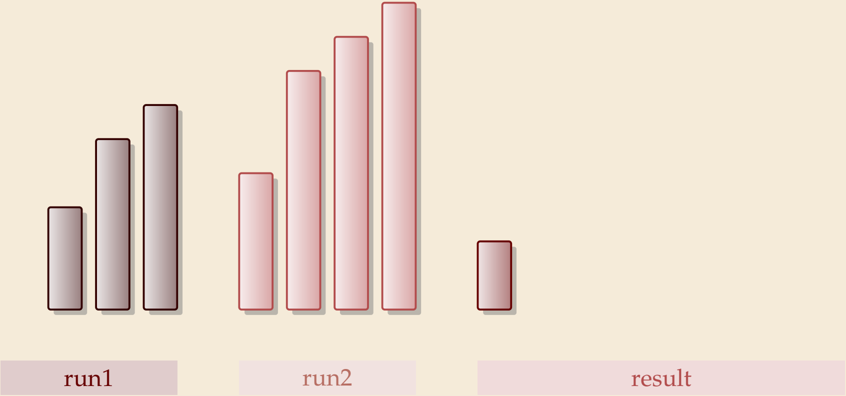
Merging sorted lists

Merging sorted lists

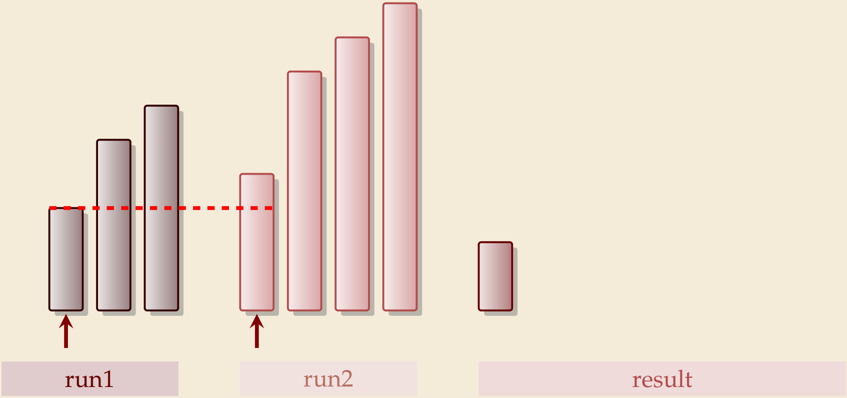
Merging sorted lists



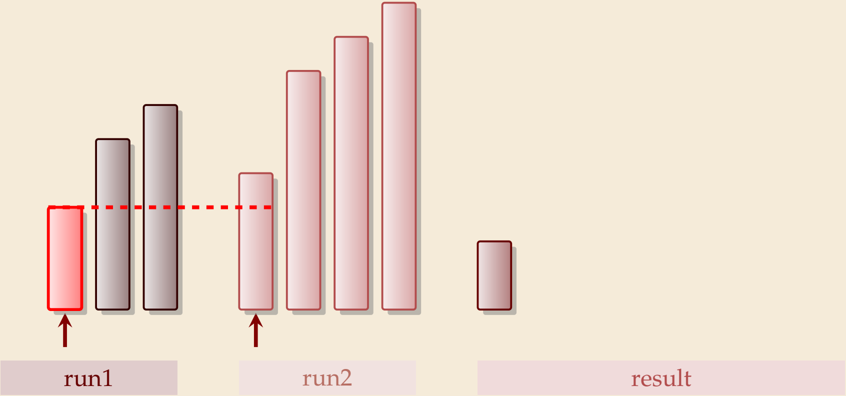
Merging sorted lists



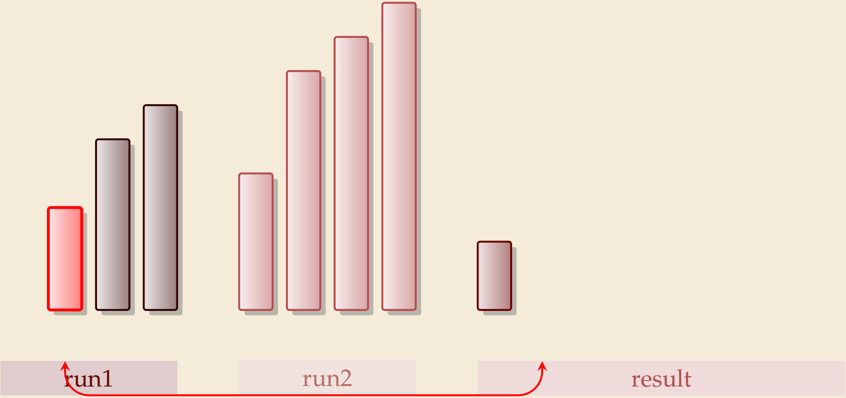
Merging sorted lists



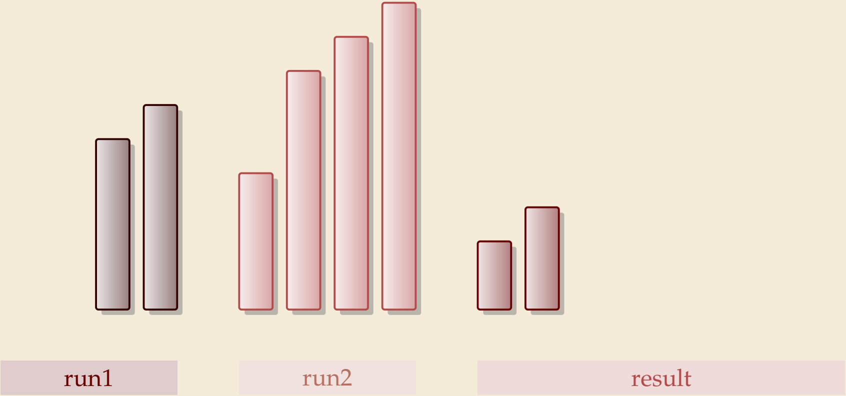
Merging sorted lists



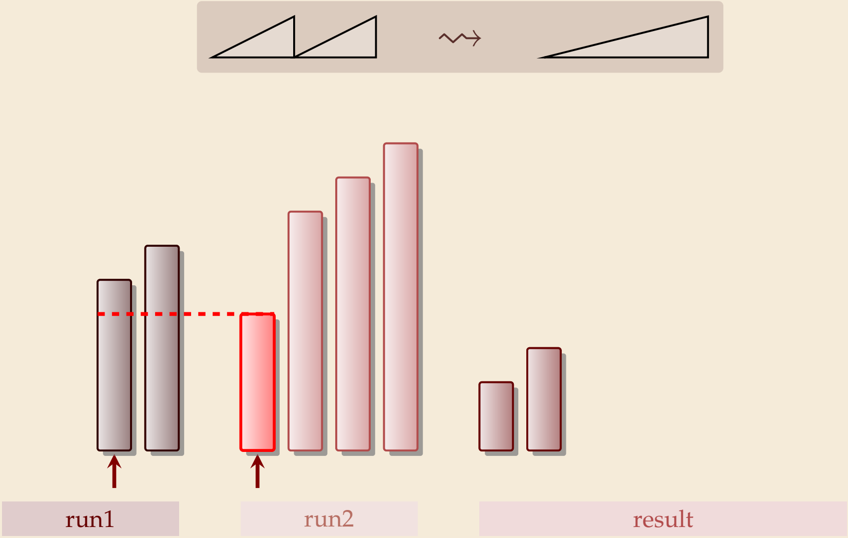
Merging sorted lists



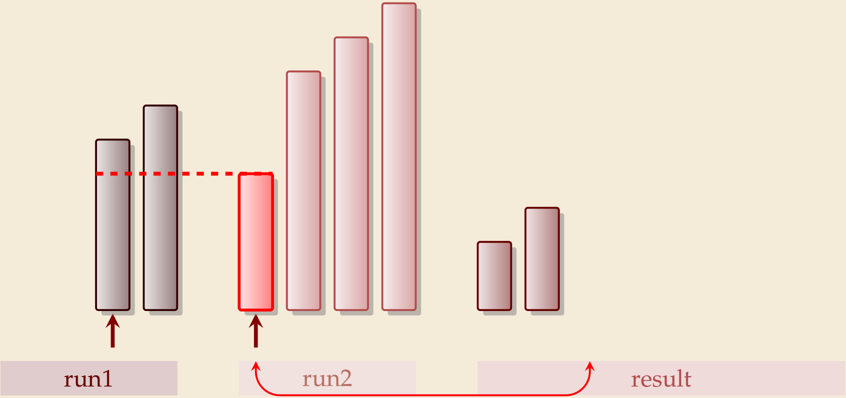
Merging sorted lists



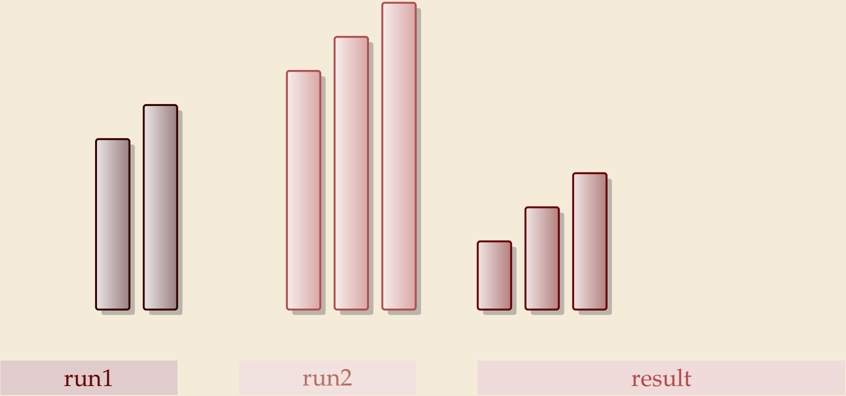
Merging sorted lists



Merging sorted lists



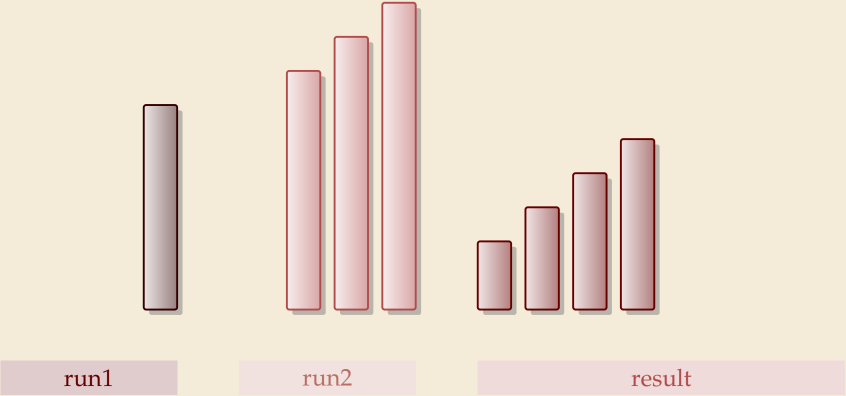
Merging sorted lists



Merging sorted lists

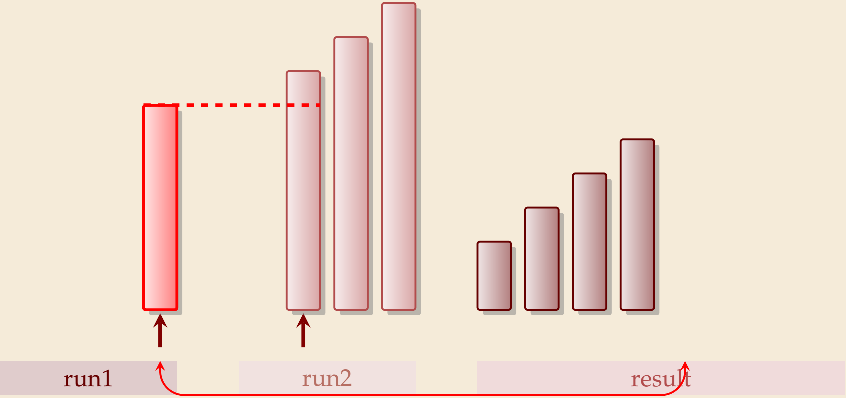
Merging sorted lists

Merging sorted lists

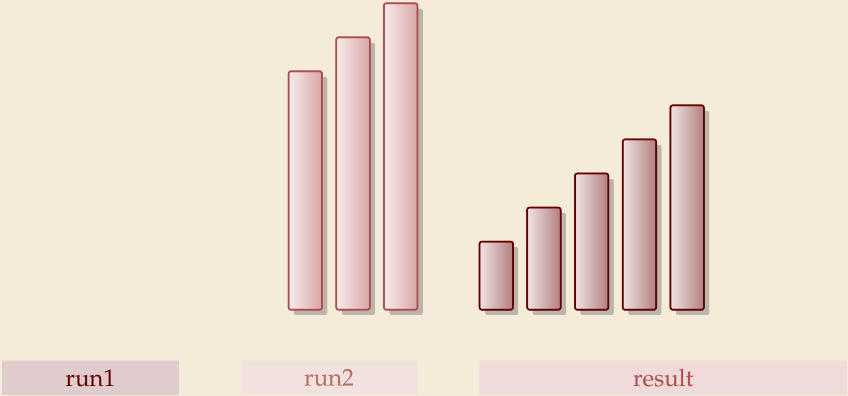


Merging sorted lists

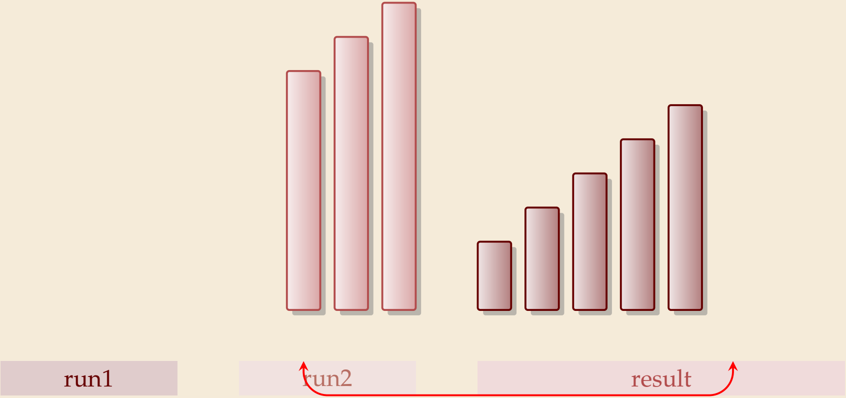
Merging sorted lists



Merging sorted lists

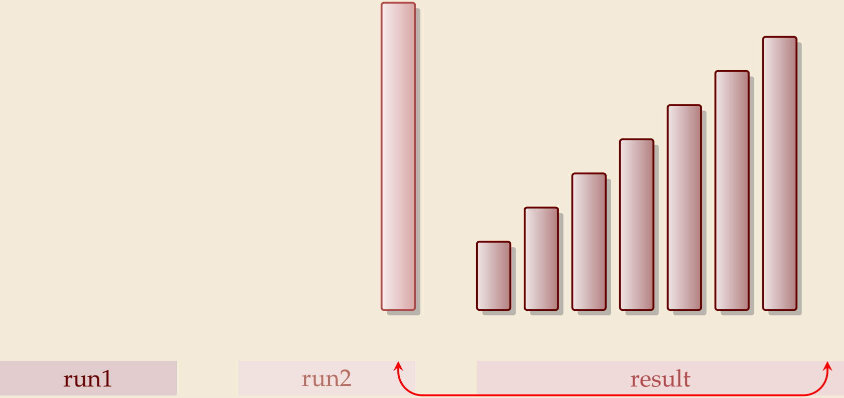


Merging sorted lists

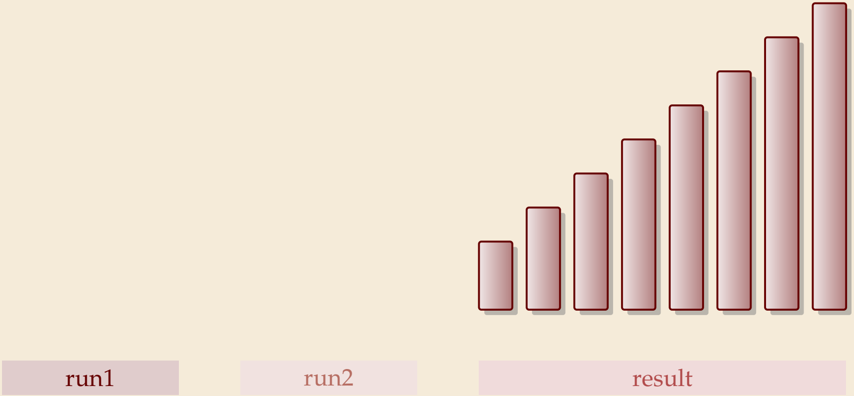


Merging sorted lists

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/comp526`

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/comp526`

Mergesort

procedure mergesort($A[l..r)$)
    $n := r - l$
    if $n \le 1$ **return**
    $m := l + \left\lfloor \frac{n}{2} \right\rfloor$
    mergesort($A[l..m)$)
    mergesort($A[m..r)$)
    merge($A[l..m)$, $A[m..r)$, buf)
    copy buf to $A[l..r)$

proc sort($A[0..n)$)
    mergesort($A[0..n)$)

- 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)

array accesses

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

$C(n) \sim \boxed{2n \lg n}$

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

same for best and worst case!

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

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

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

$$C(2^k) \underset{k \geq 2}{=} 2\,C(2^{k-1}) + 2 \cdot 2^k$$

$$= 2\left(2 \cdot C(2^{k-2}) + 2 \cdot 2^{k-1}\right) + 2 \cdot 2^k$$

$$= 2^2 \cdot C(2^{k-2}) + \underbrace{2^2 \cdot 2^{k-1}}_{= 2^{k+1}} + \underbrace{2 \cdot 2^k}_{= 2^{k+1}}$$

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*

Clicker Question

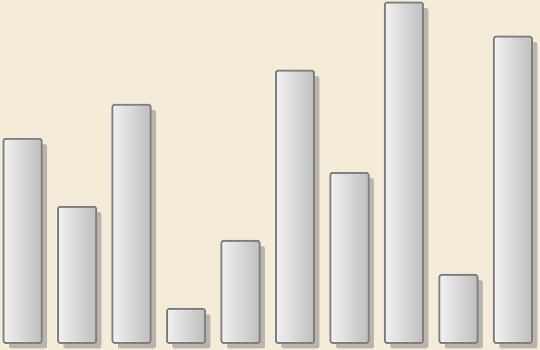
How does quicksort work?

- **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.~~

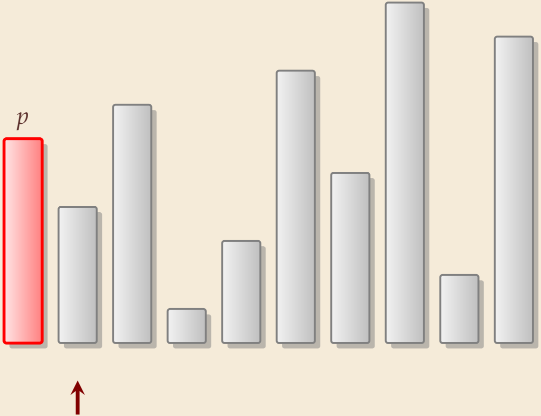
→ sli.do/comp526

Partitioning around a pivot

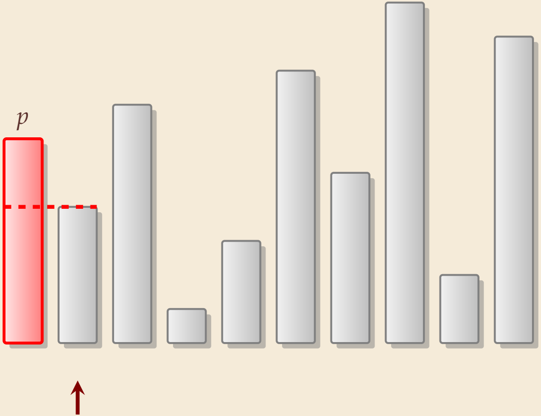
Partitioning around a pivot



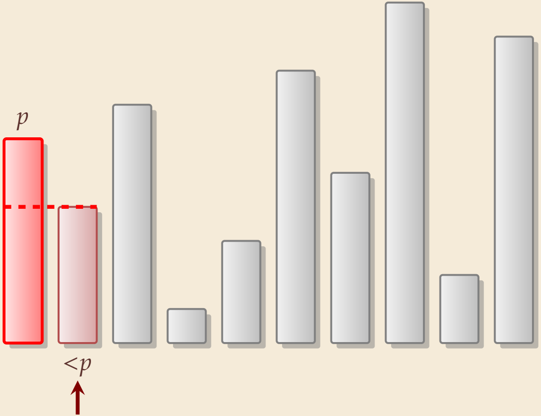
Partitioning around a pivot



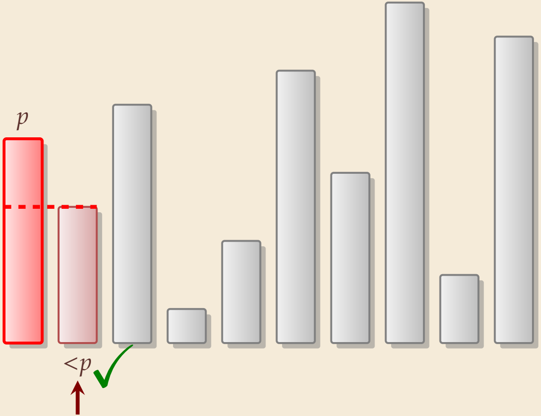
Partitioning around a pivot



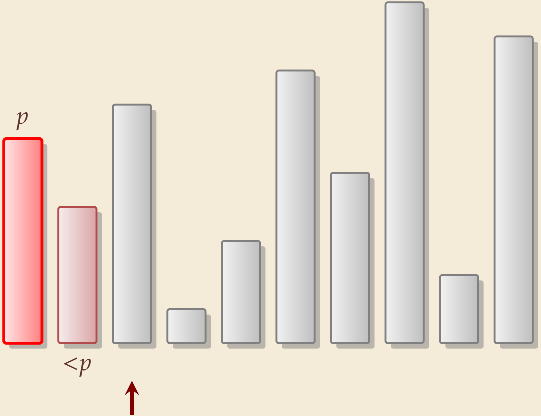
Partitioning around a pivot



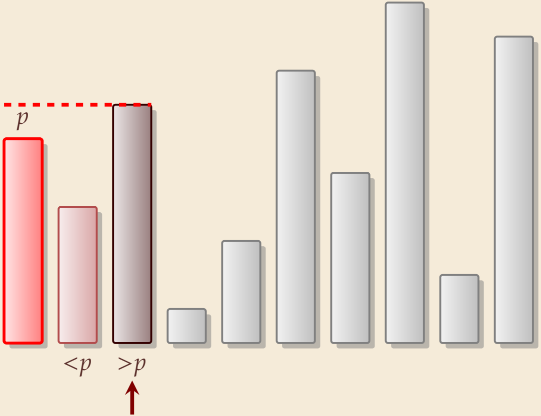
Partitioning around a pivot



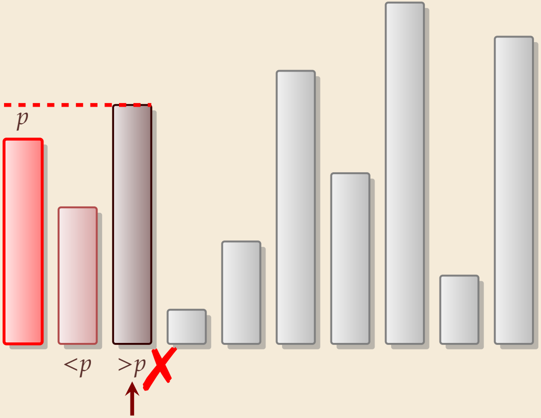
Partitioning around a pivot



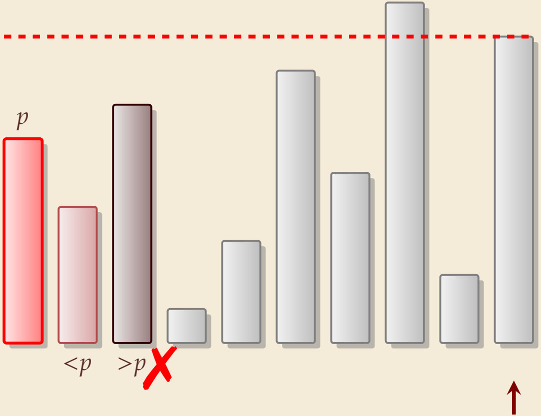
Partitioning around a pivot



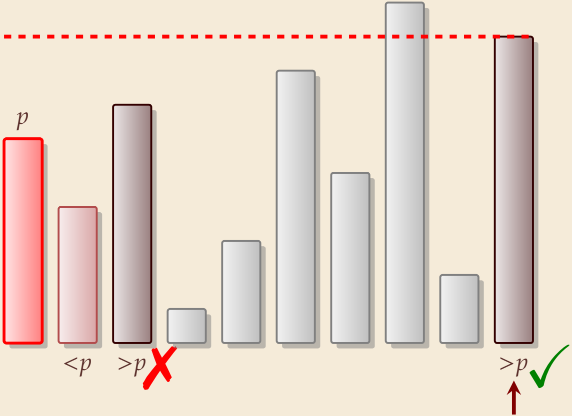
Partitioning around a pivot



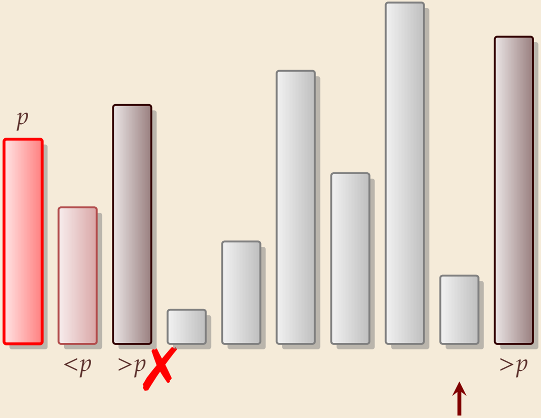
Partitioning around a pivot



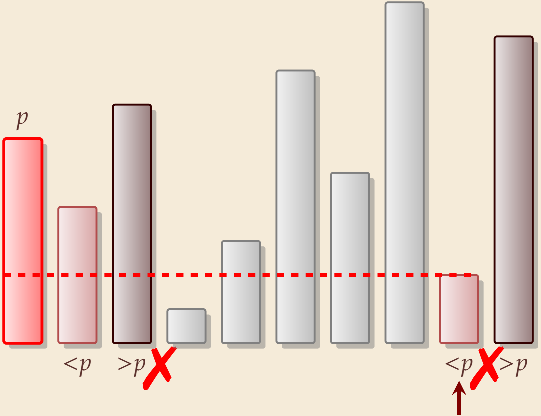
Partitioning around a pivot



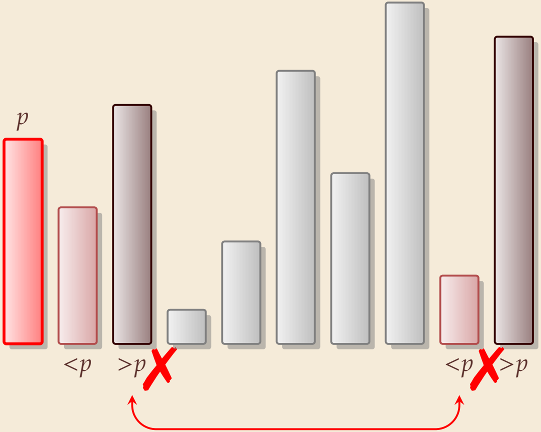
Partitioning around a pivot



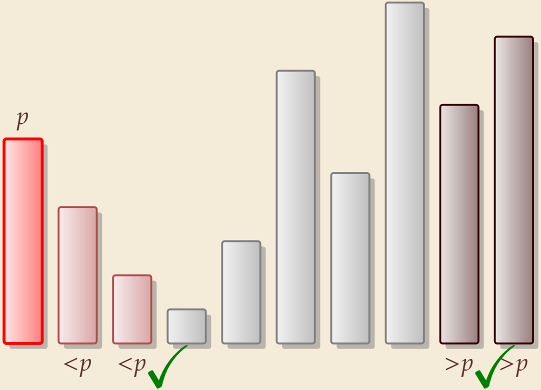
Partitioning around a pivot



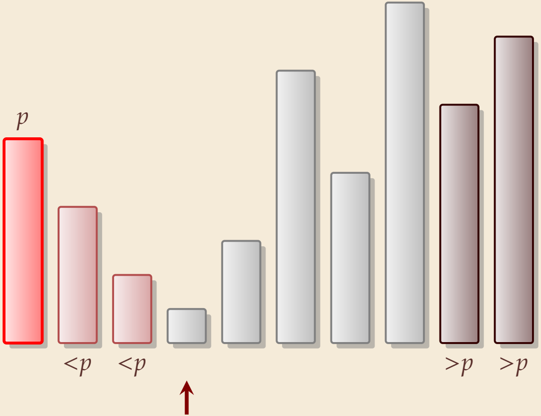
Partitioning around a pivot



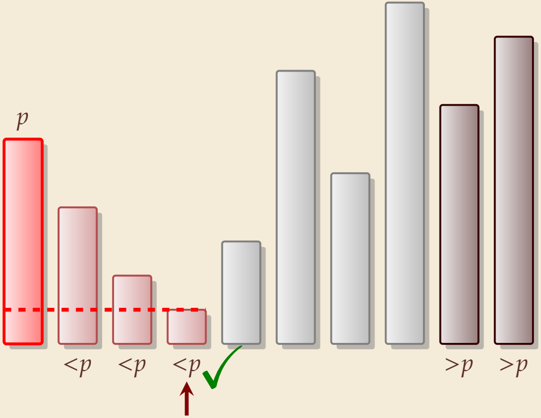
Partitioning around a pivot



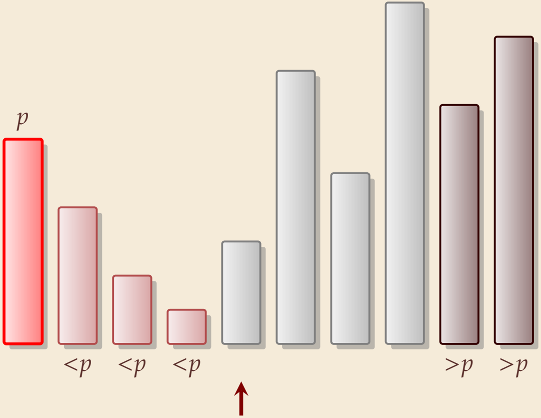
Partitioning around a pivot



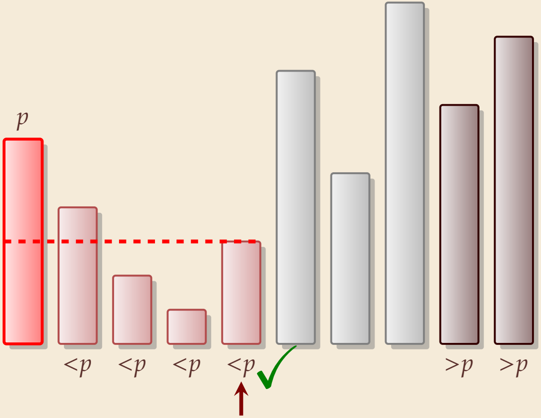
Partitioning around a pivot



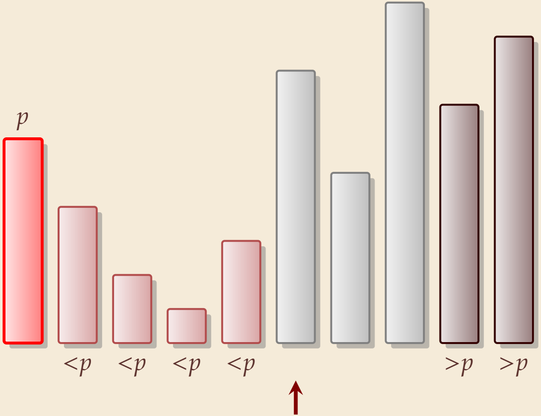
Partitioning around a pivot



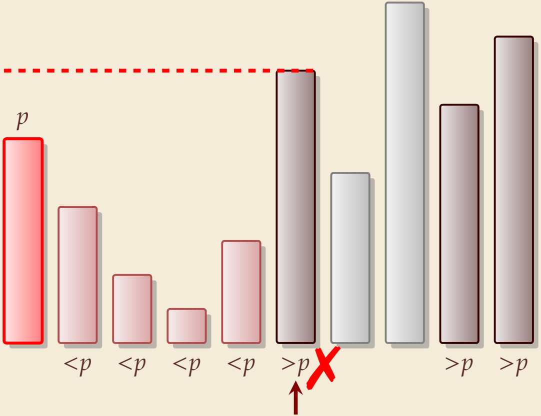
Partitioning around a pivot



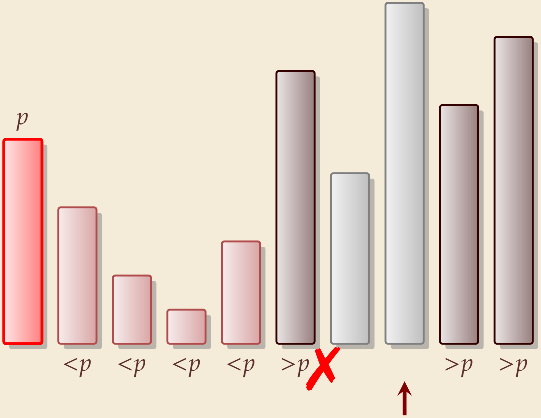
Partitioning around a pivot



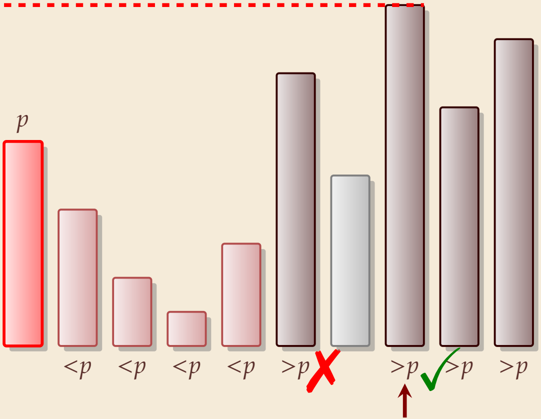
Partitioning around a pivot



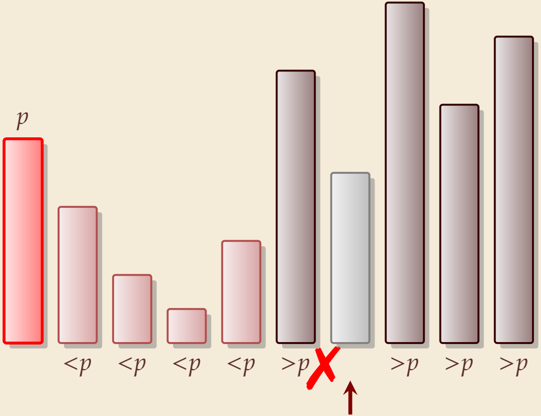
Partitioning around a pivot



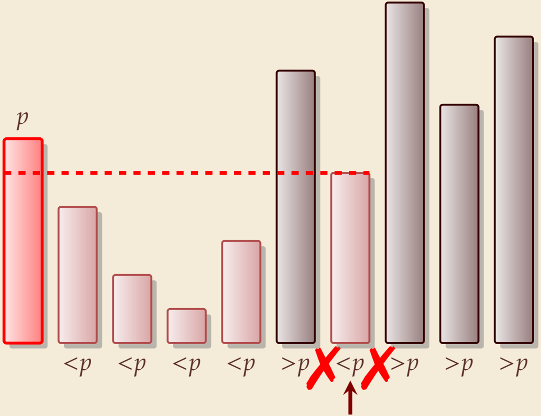
Partitioning around a pivot



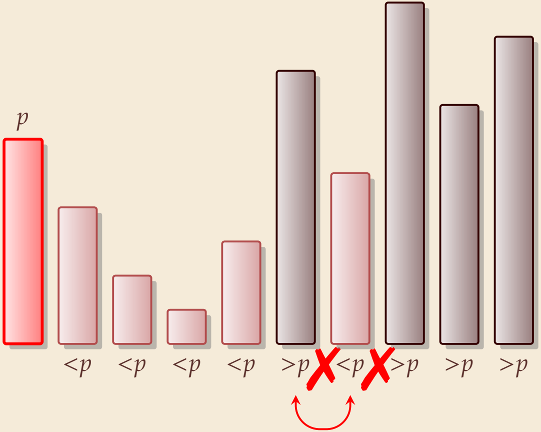
Partitioning around a pivot



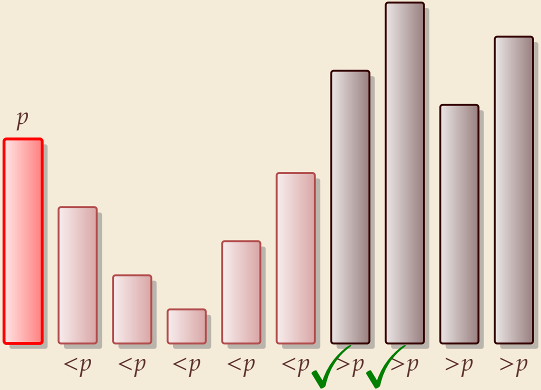
Partitioning around a pivot



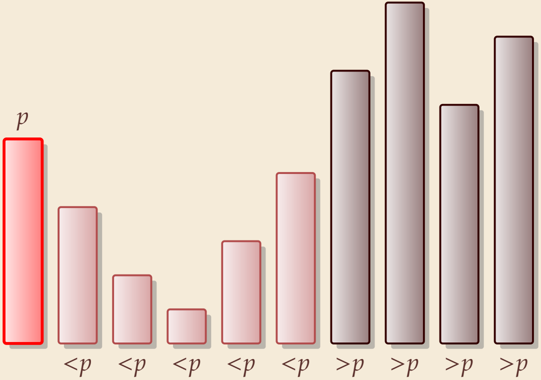
Partitioning around a pivot



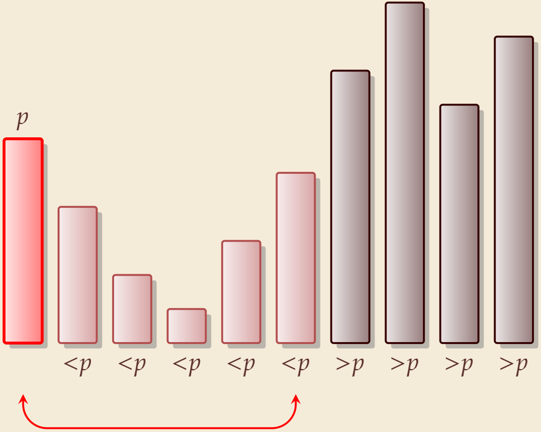
Partitioning around a pivot



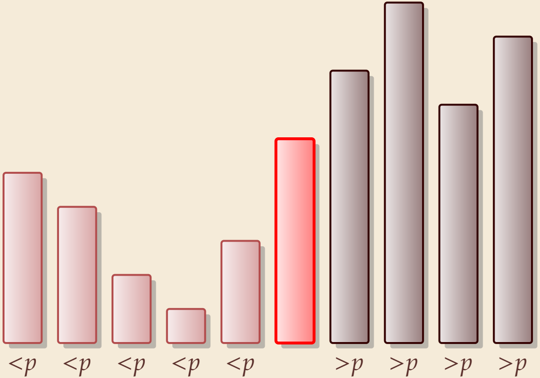
Partitioning around a pivot



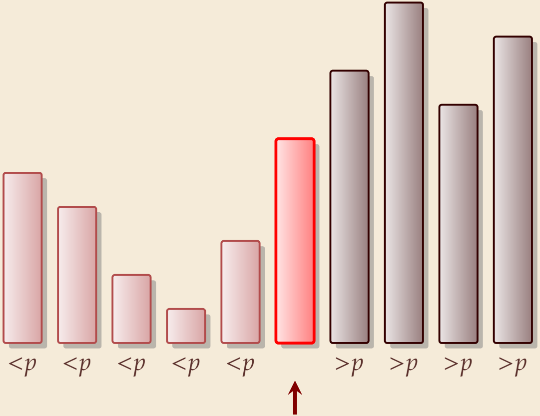
Partitioning around a pivot



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/comp526`

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

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

7	4	2	9	1	3	8	5	6

4	2	1	3	5	6

9	8

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	7	9	8

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

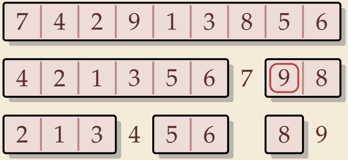
7	4	2	9	1	3	8	5	6

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

[2 | 1 | 3] 4 [5 | 6]

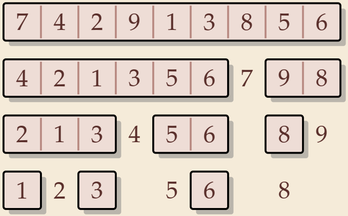
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

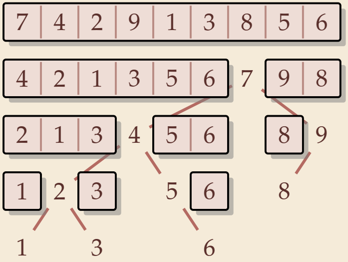


Quicksort & Binary Search Trees

Quicksort

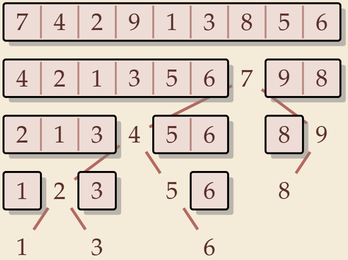
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

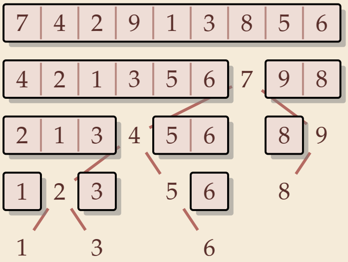


Binary Search Tree (BST)

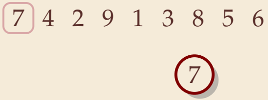
7 4 2 9 1 3 8 5 6

Quicksort & Binary Search Trees

Quicksort

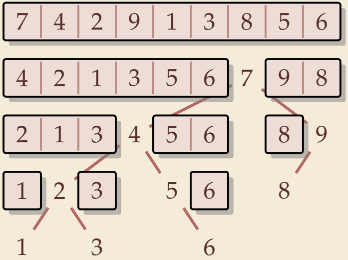


Binary Search Tree (BST)

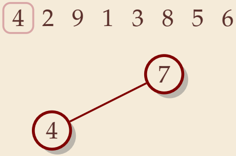


Quicksort & Binary Search Trees

Quicksort

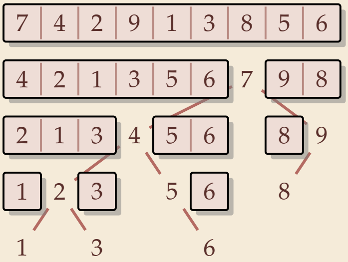


Binary Search Tree (BST)

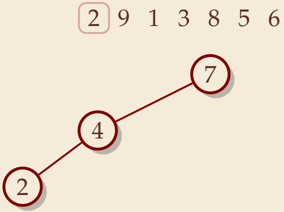


Quicksort & Binary Search Trees

Quicksort

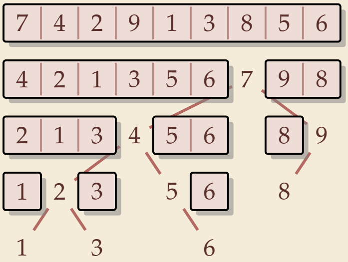


Binary Search Tree (BST)

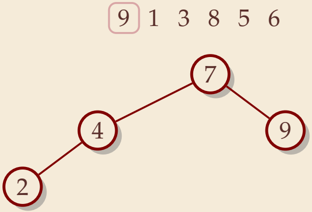


Quicksort & Binary Search Trees

Quicksort

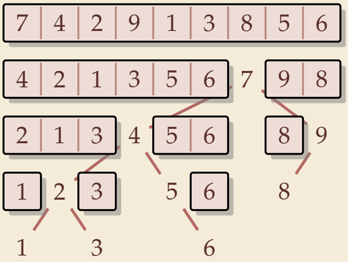


Binary Search Tree (BST)

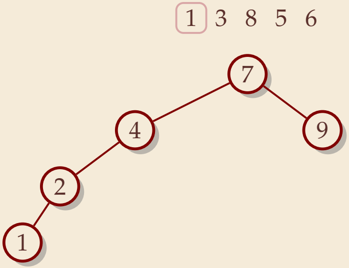


Quicksort & Binary Search Trees

Quicksort

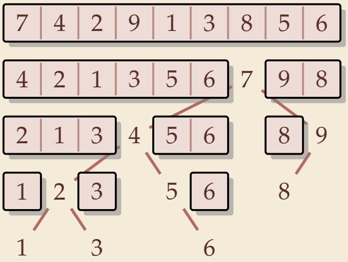


Binary Search Tree (BST)

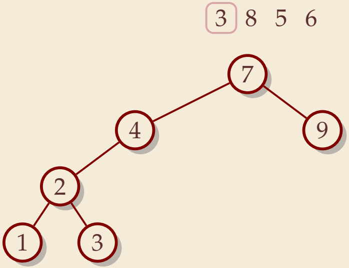


Quicksort & Binary Search Trees

Quicksort

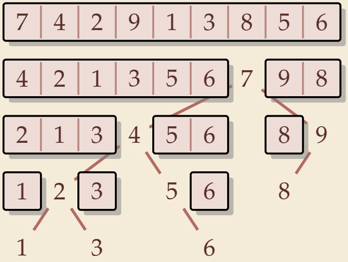


Binary Search Tree (BST)

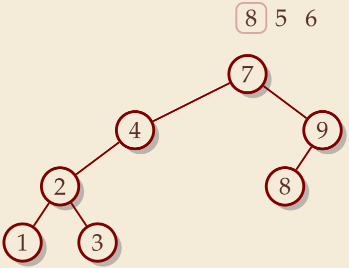


Quicksort & Binary Search Trees

Quicksort

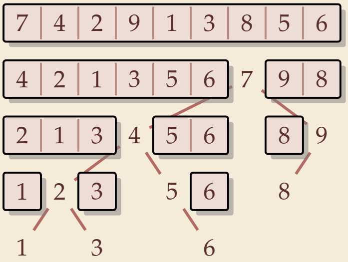


Binary Search Tree (BST)

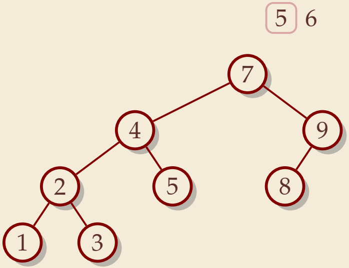


Quicksort & Binary Search Trees

Quicksort

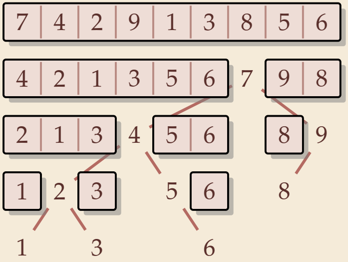


Binary Search Tree (BST)

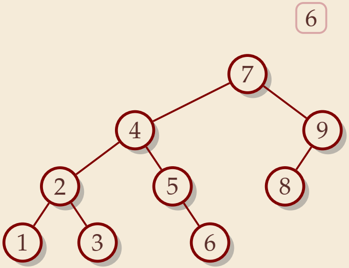


Quicksort & Binary Search Trees

Quicksort

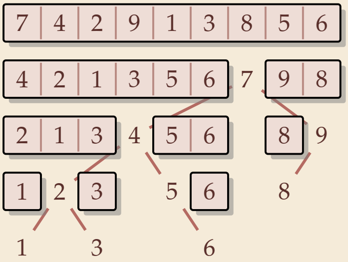


Binary Search Tree (BST)

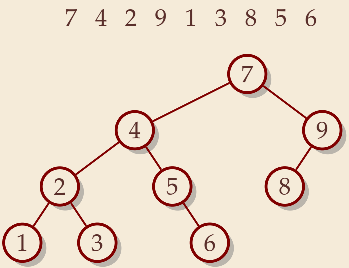


Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)

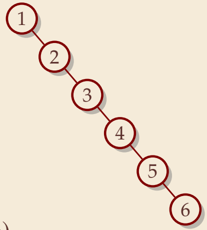


- recursion tree of quicksort $=$ binary search tree from successive insertion
- comparisons in quicksort $=$ comparisons to built BST
- comparisons in quicksort $\approx$ comparisons to search each element in BST

Quicksort – Worst Case

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

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



$\leadsto$ 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

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

Randomized Quicksort – Analysis

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

$\rightsquigarrow$ quicksort needs $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ *in expectation*

- 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

- 👍 fastest general-purpose method
- 👍 $\Theta(n \log n)$ average case
- 👍 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.