3 Efficient Sorting

17 February 2022

Sebastian Wild

COMP526 (Spring 2022)
University of Liverpool

version `2022-02-17 09:56`

Learning Outcomes

1. Know principles and implementation of *mergesort*.
2. Know principles and implementation of *quicksort*.
3. Know properties and *performance characteristics* of mergesort and quicksort.
4. Know the comparison model and understand the corresponding *lower bound*.
5. Understand *counting sort* and how it circumvents the comparison lower bound.
6. Understand and use the *parallel random-access-machine* model in its different variants.
7. Be able to *analyze* and compare simple shared-memory parallel algorithms by determining *parallel time and work*.
8. Understand efficient parallel *prefix sum* algorithms.
9. Be able to devise high-level description of *parallel quicksort and mergesort* methods.

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

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 (`__lt__()`)
    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

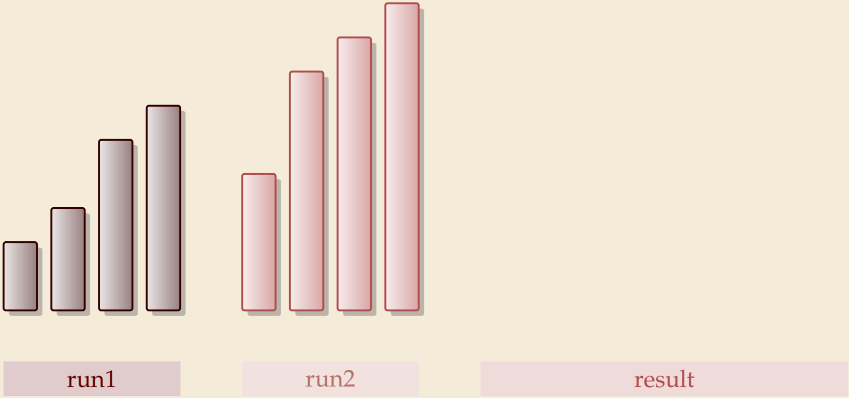
How does mergesort work?

- **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`

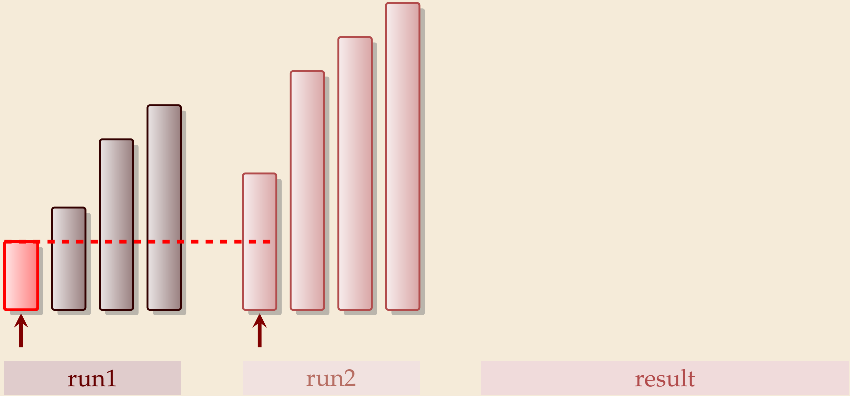
Merging sorted lists

Merging sorted lists

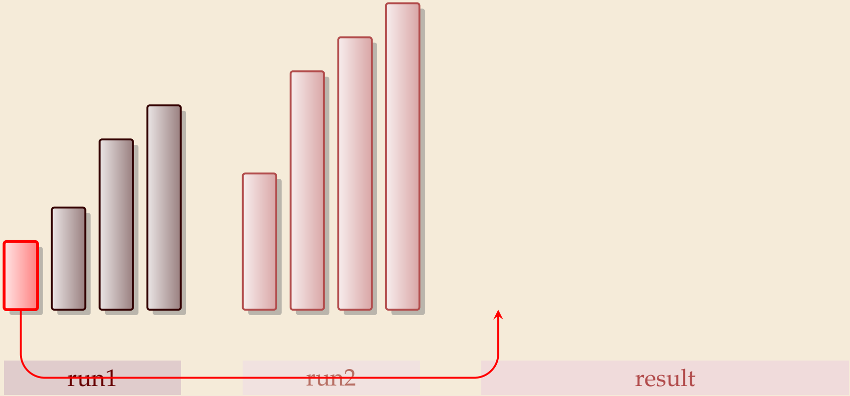


Merging sorted lists

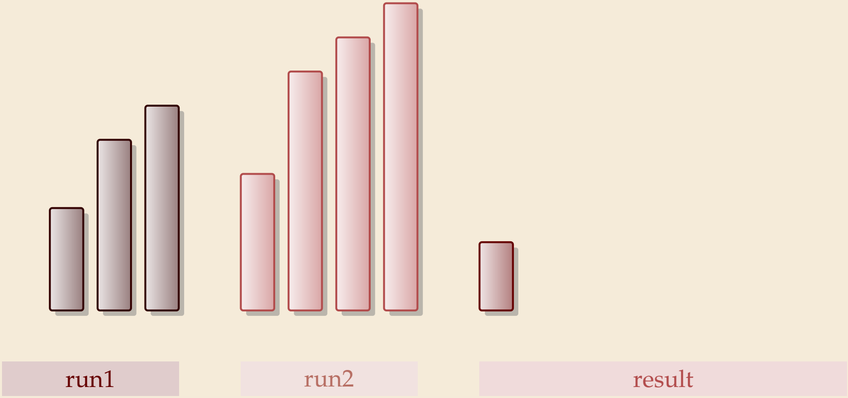
Merging sorted lists



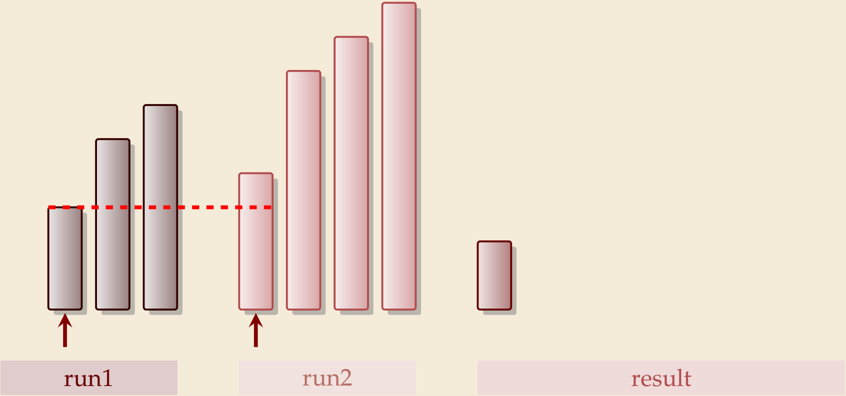
Merging sorted lists



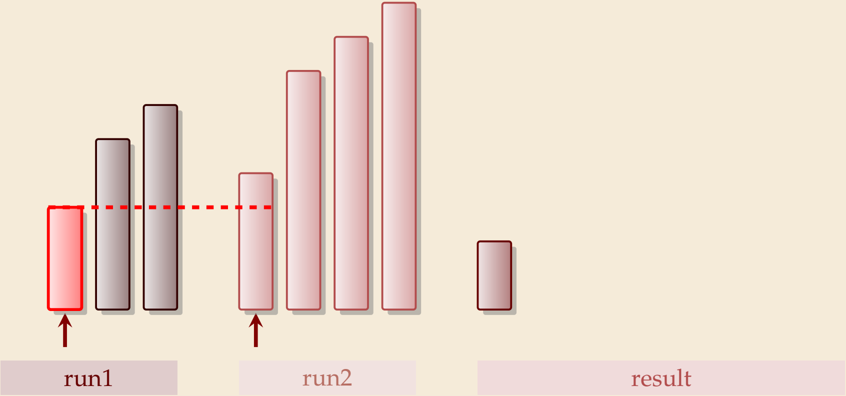
Merging sorted lists



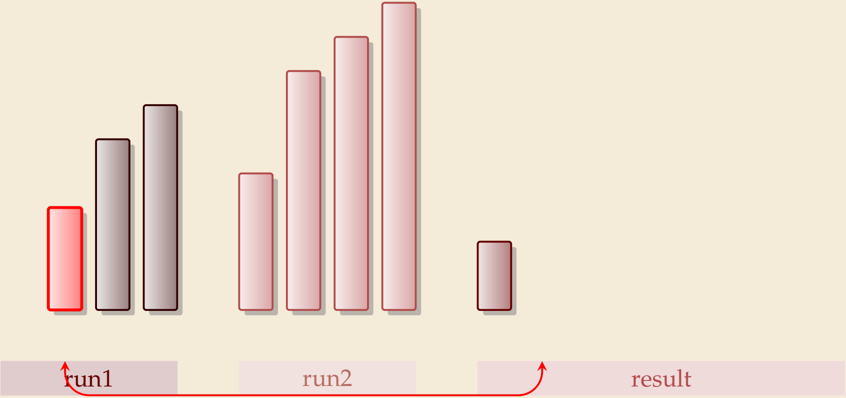
Merging sorted lists



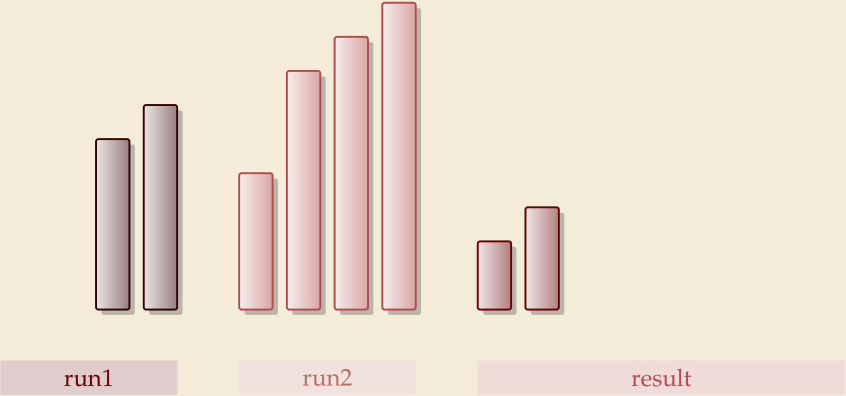
Merging sorted lists



Merging sorted lists

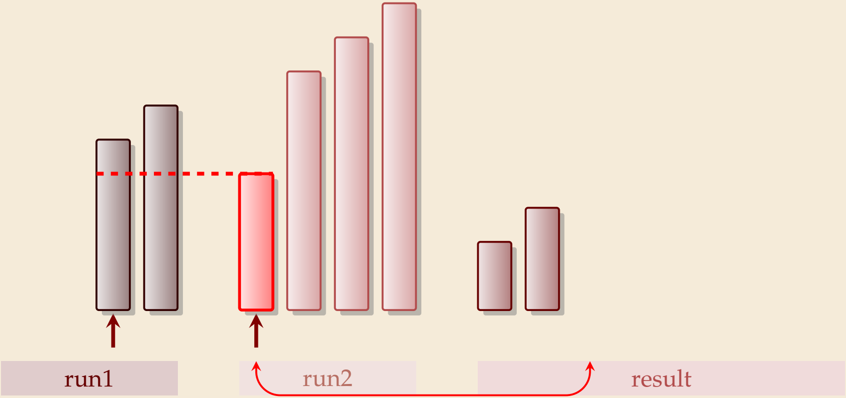


Merging sorted lists

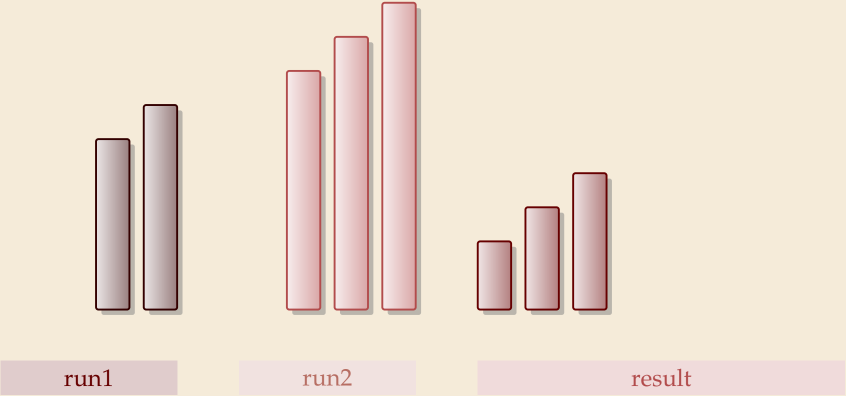


Merging sorted lists

Merging sorted lists



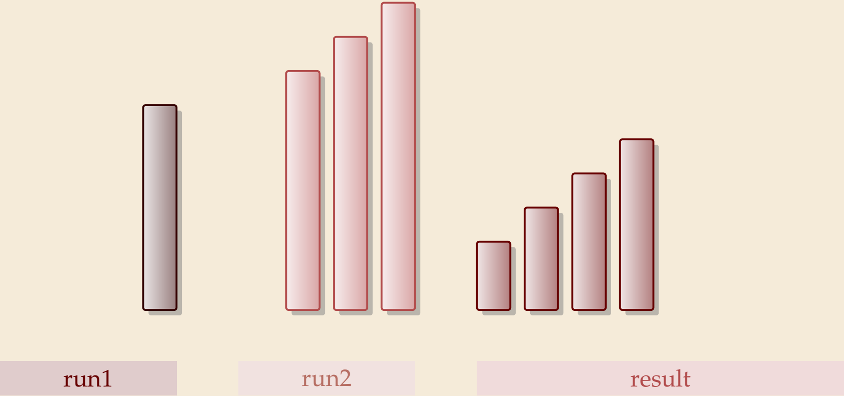
Merging sorted lists



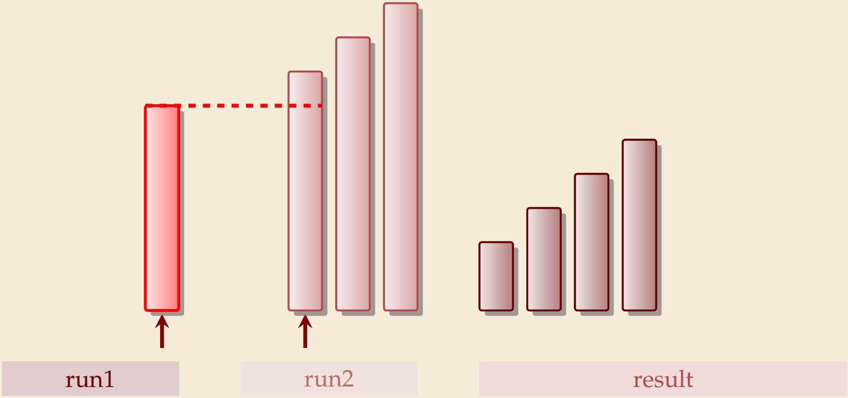
Merging sorted lists

Merging sorted lists

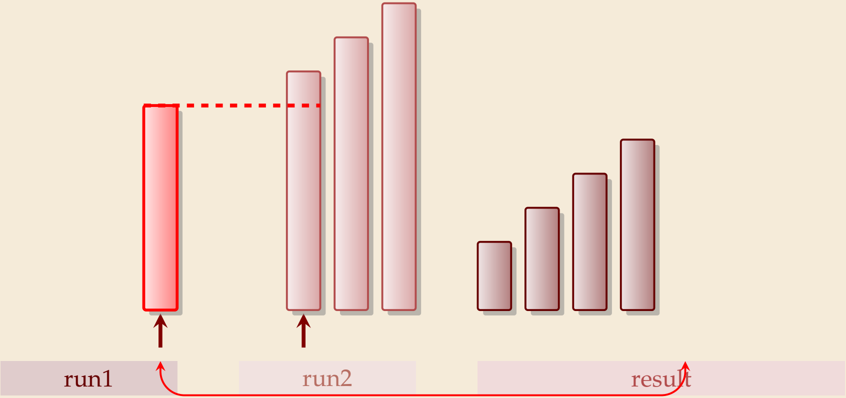
Merging sorted lists



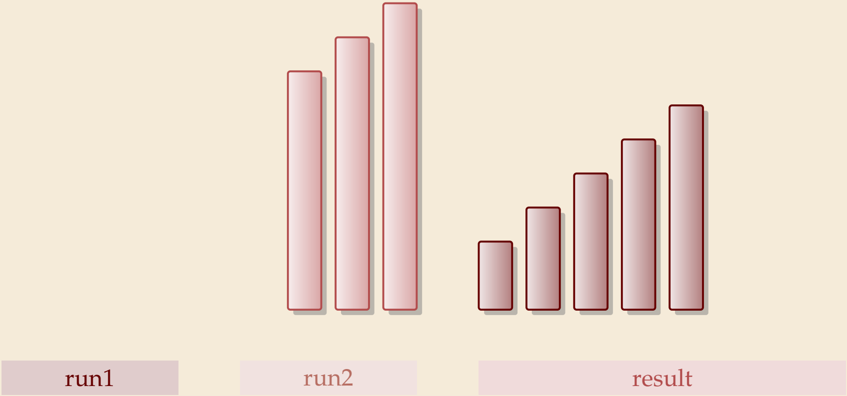
Merging sorted lists



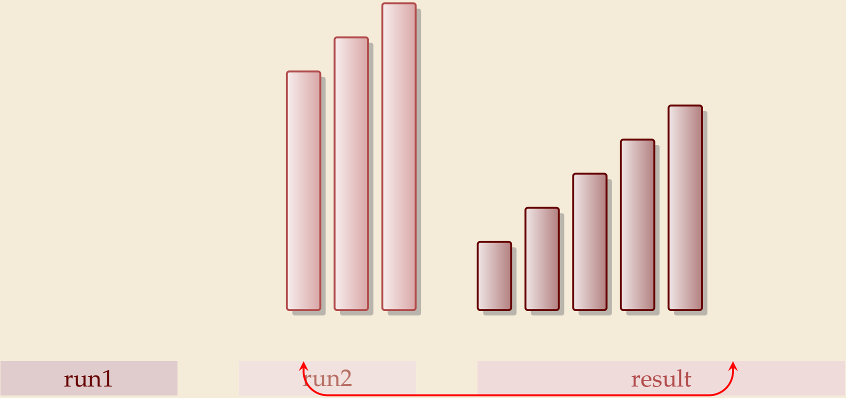
Merging sorted lists



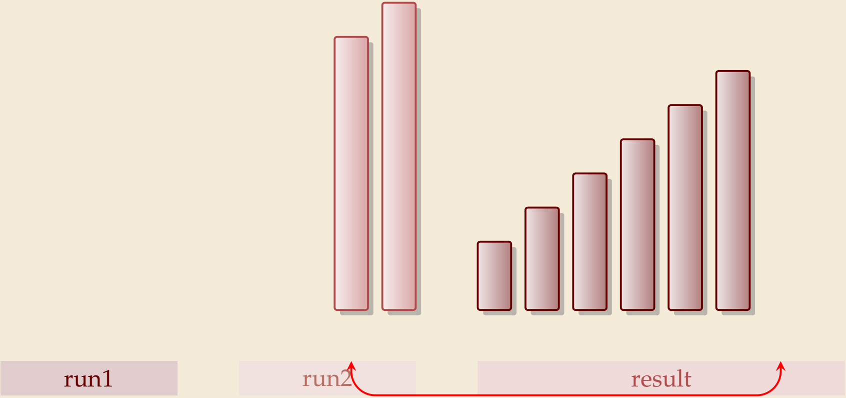
Merging sorted lists



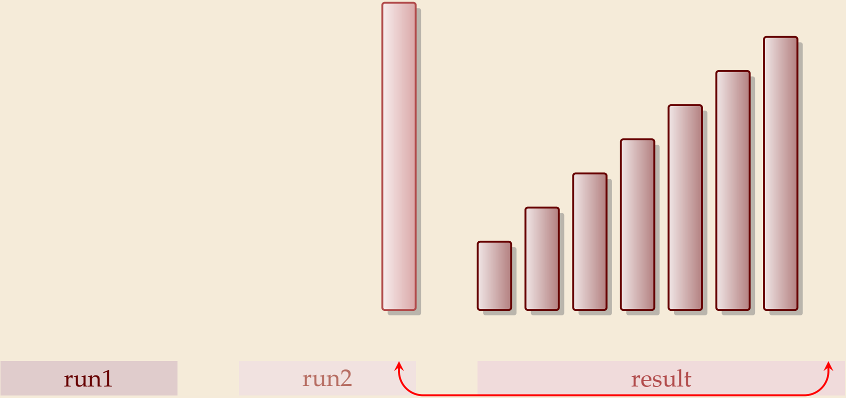
Merging sorted lists



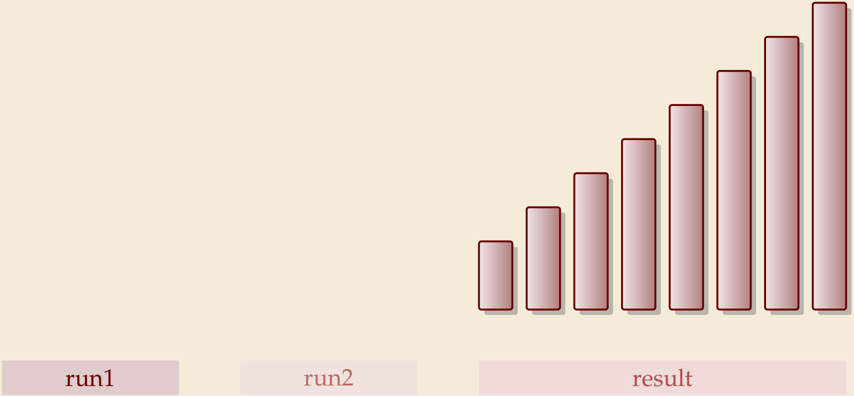
Merging sorted lists



Merging sorted lists



Merging sorted lists



Clicker Question

What is the worst-case running time of mergesort?

A $\Theta(1)$

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 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** $\Theta(n \log n)$ $\checkmark$
- **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 ≤ 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; *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

```
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; *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)

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

same for best and worst case!

$n = 2^k$

$k = \lg(n)$

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

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

$k+1$

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

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`

Clicker Question

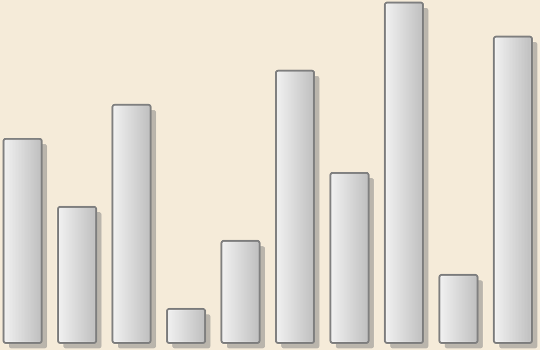
How does quicksort work?

- **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.~~

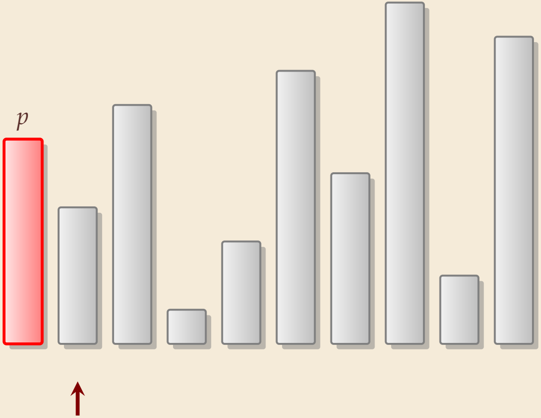
`sli.do/comp526`

Partitioning around a pivot

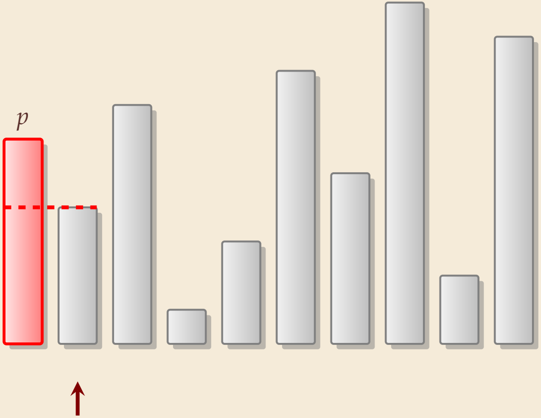
Partitioning around a pivot



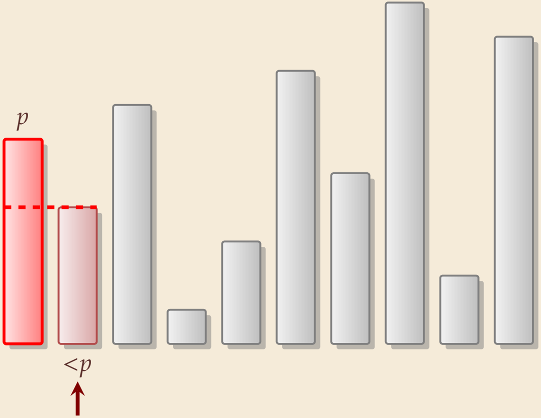
Partitioning around a pivot



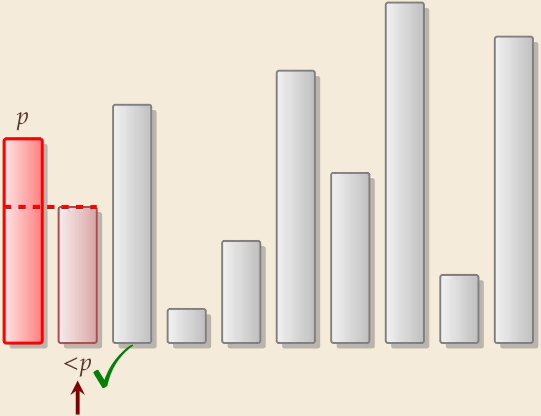
Partitioning around a pivot



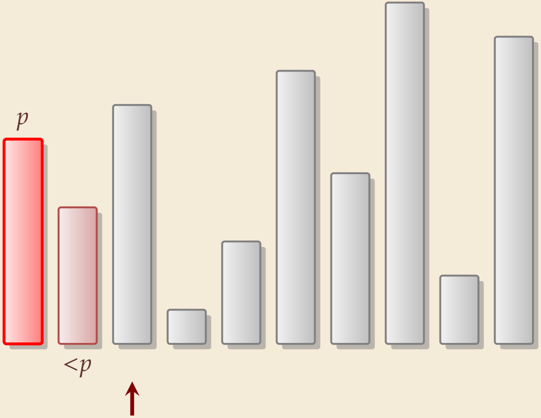
Partitioning around a pivot



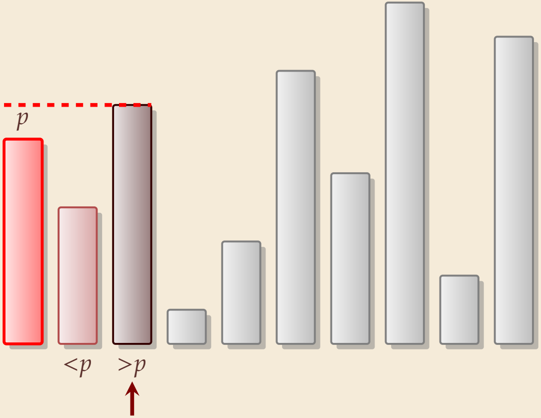
Partitioning around a pivot



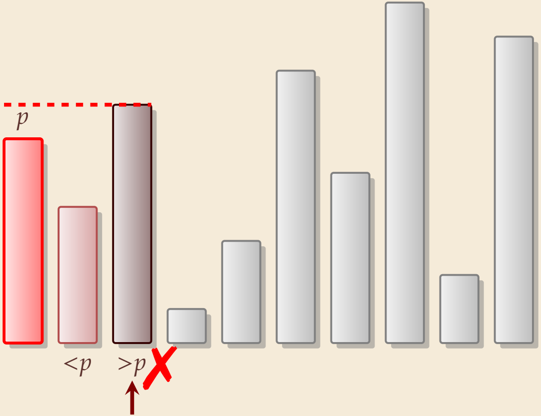
Partitioning around a pivot



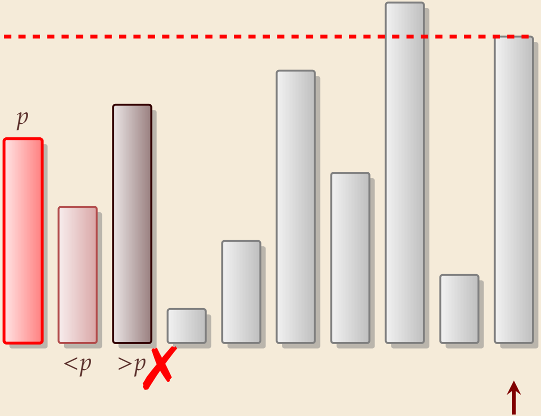
Partitioning around a pivot



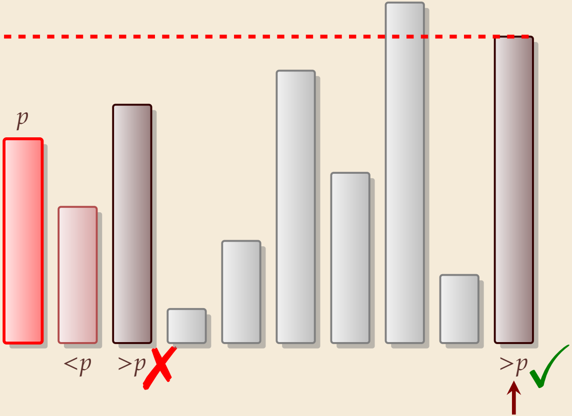
Partitioning around a pivot



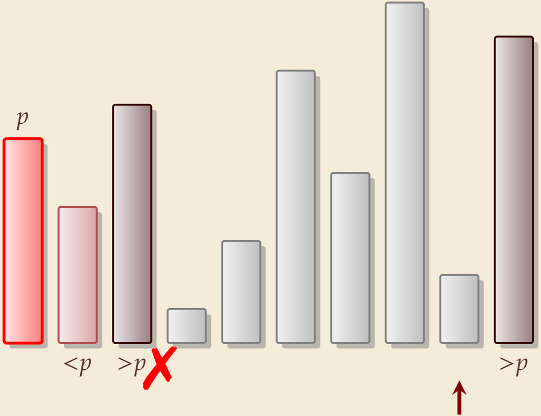
Partitioning around a pivot



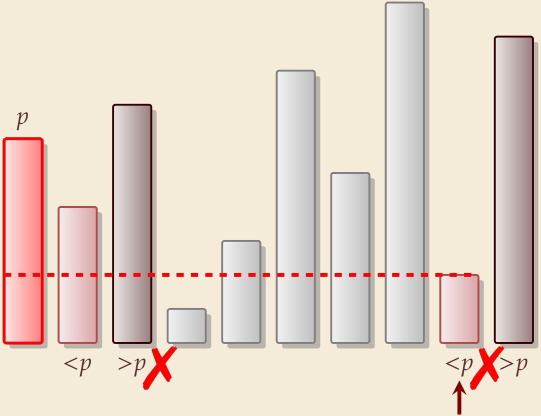
Partitioning around a pivot



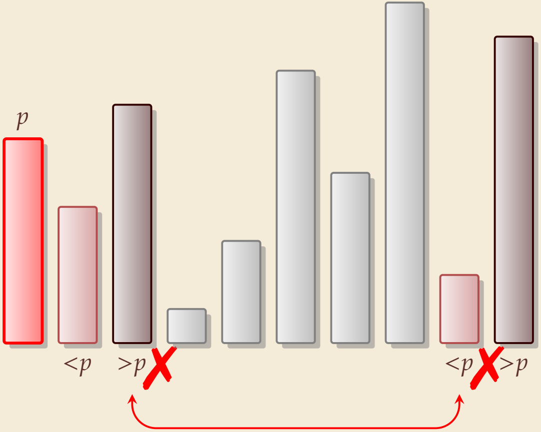
Partitioning around a pivot



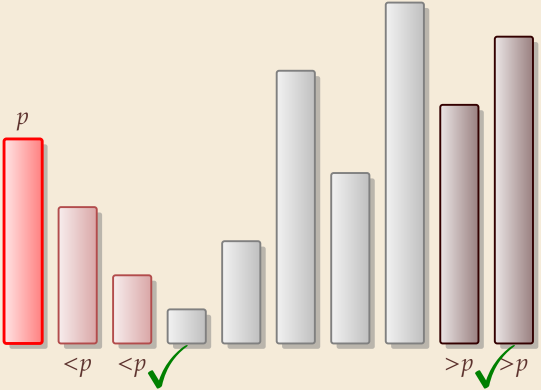
Partitioning around a pivot



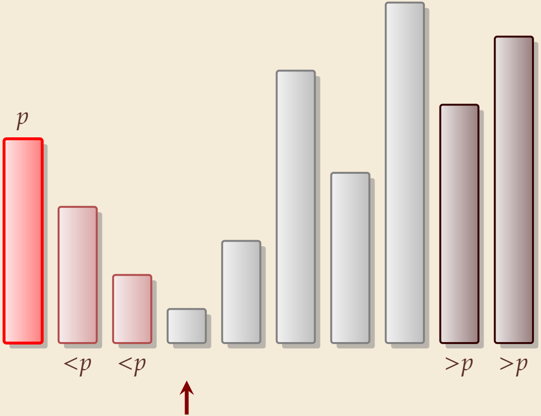
Partitioning around a pivot



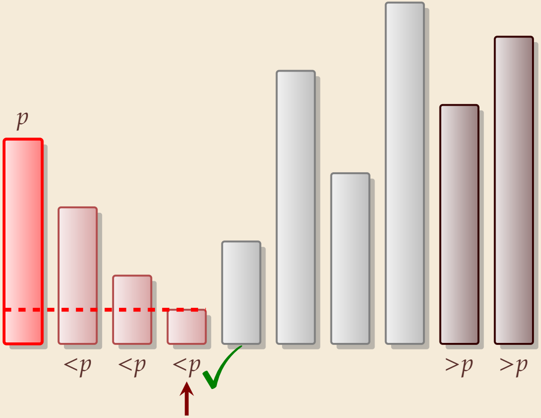
Partitioning around a pivot



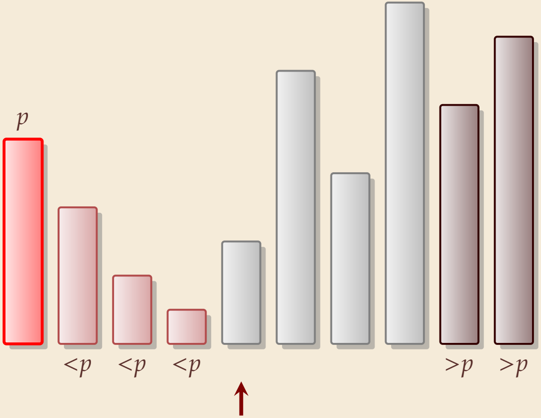
Partitioning around a pivot



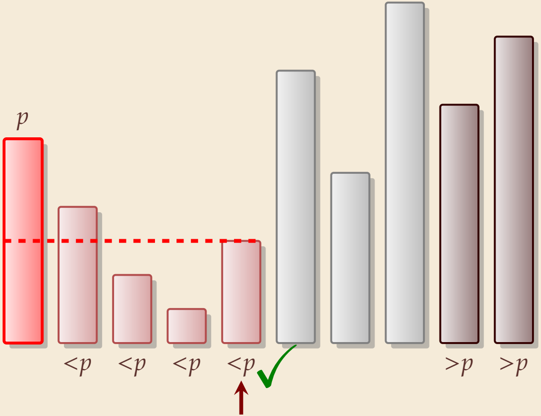
Partitioning around a pivot



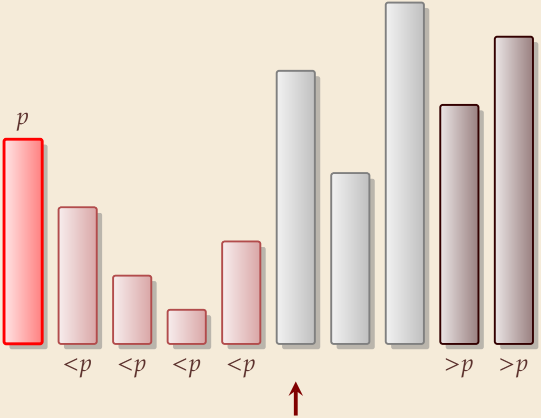
Partitioning around a pivot



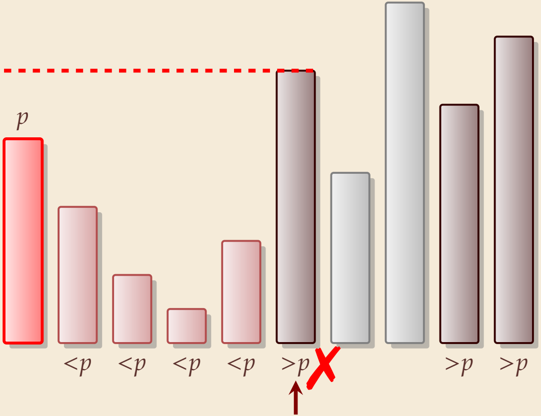
Partitioning around a pivot



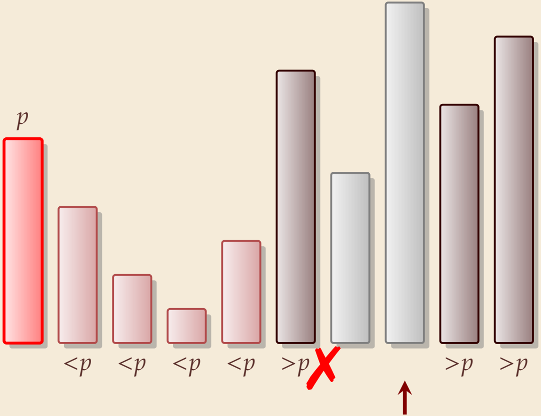
Partitioning around a pivot



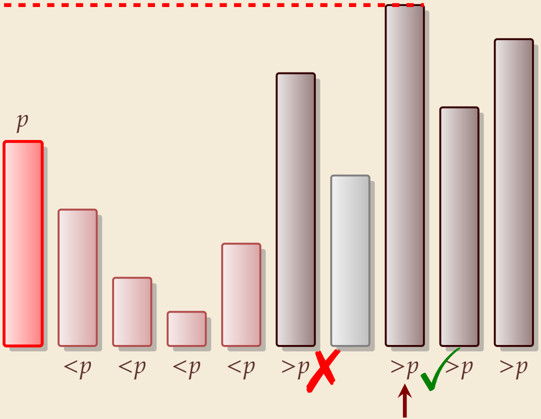
Partitioning around a pivot



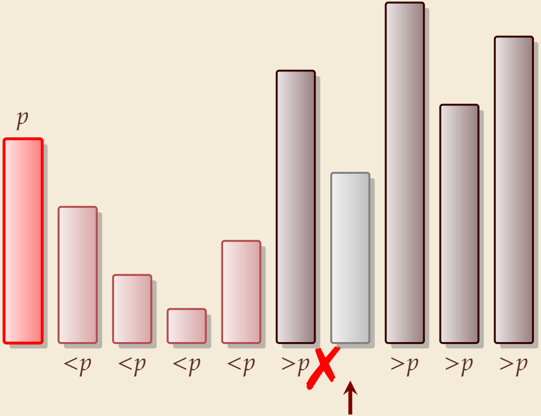
Partitioning around a pivot



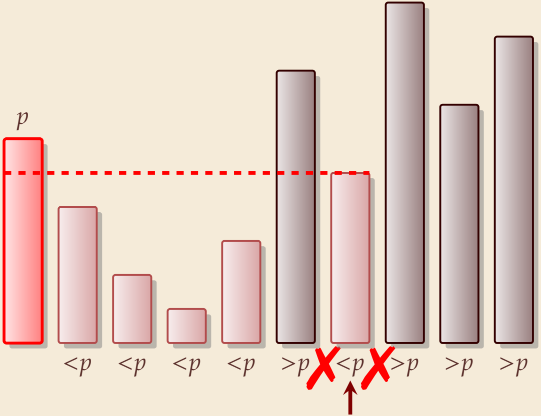
Partitioning around a pivot



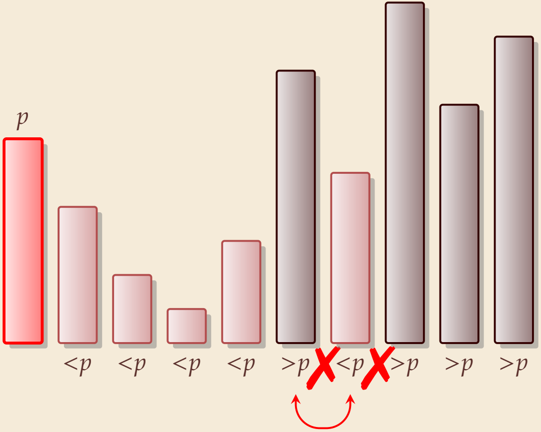
Partitioning around a pivot



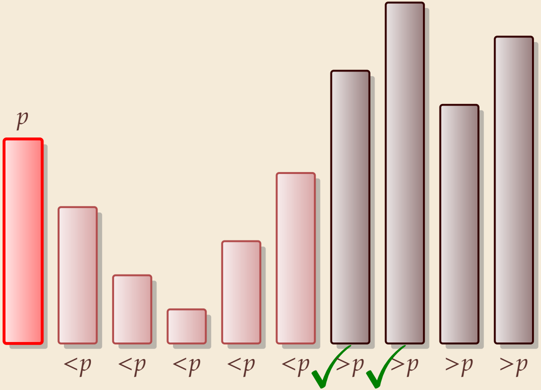
Partitioning around a pivot



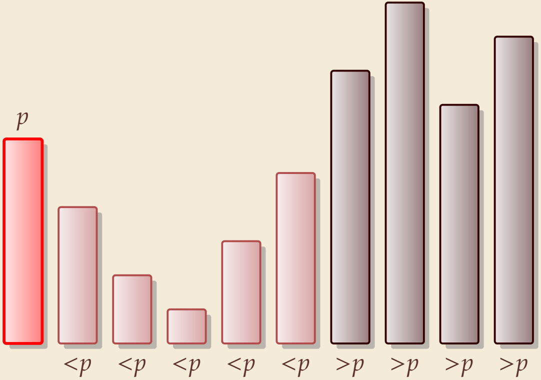
Partitioning around a pivot



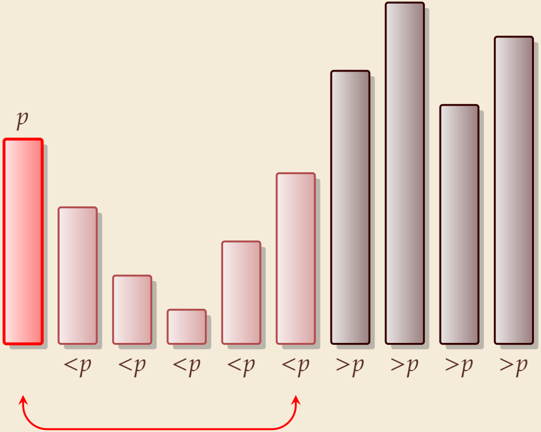
Partitioning around a pivot



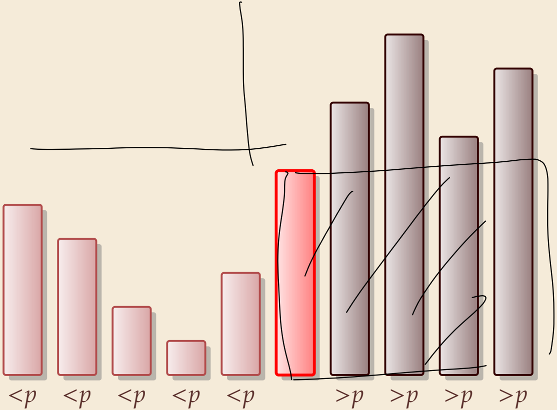
Partitioning around a pivot



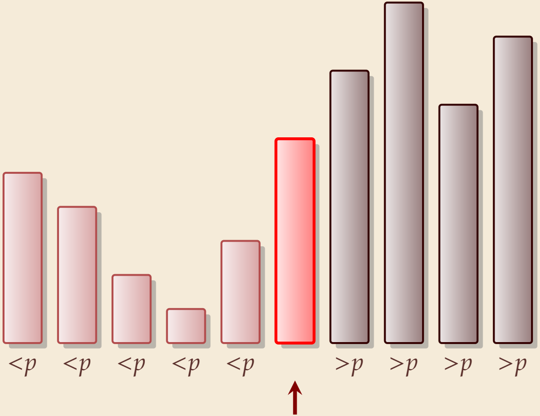
Partitioning around a pivot



Partitioning around a pivot



Partitioning around a pivot



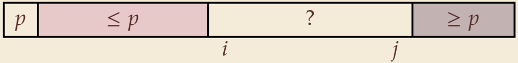
- no extra space needed
- visits each element once
- returns rank/position of pivot

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



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; *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

Clicker Question

What is the worst-case running time of quicksort?

A $\cancel{\Theta(1)}$

B $\cancel{\Theta(\log n)}$

C $\cancel{\Theta(\log \log n)}$

D $\cancel{\Theta(\sqrt{n})}$

E $\cancel{\Theta(n)}$

F $\cancel{\Theta(n \log \log n)}$

G $\cancel{\Theta(n \log n)}$

H $\cancel{\Theta(n \log^2 n)}$

I $\cancel{\Theta(n^{1+\epsilon})}$

J $\Theta(n^2)$ $\checkmark$

K $\cancel{\Theta(n^3)}$

L $\cancel{\Theta(2^n)}$

`sli.do/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

$\boxed{4}$ 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

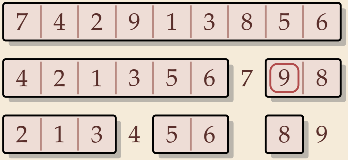
2	1	3

4

5	6

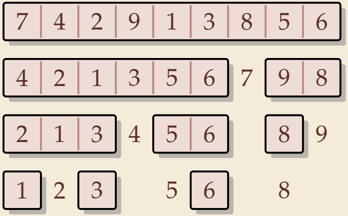
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

7	4	2	9	1	3	8	5	6

4	2	1	3	5	6

7

9	8

2	1	3

4

5	6

8

9

1

2

3

5

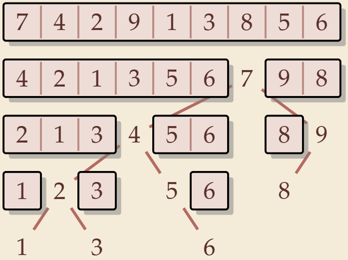
6

8

1 3 6

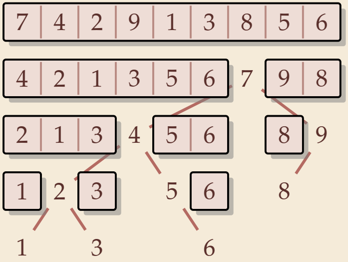
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

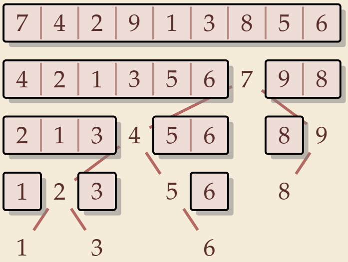


Binary Search Tree (BST)

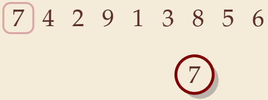
7 4 2 9 1 3 8 5 6

Quicksort & Binary Search Trees

Quicksort

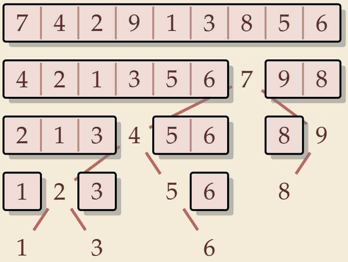


Binary Search Tree (BST)

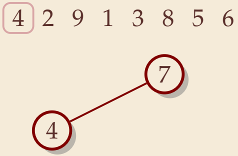


Quicksort & Binary Search Trees

Quicksort

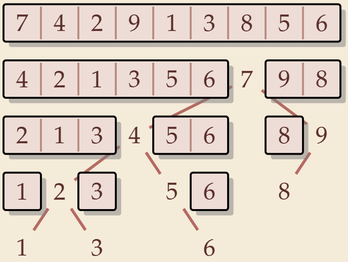


Binary Search Tree (BST)

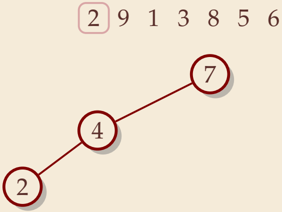


Quicksort & Binary Search Trees

Quicksort

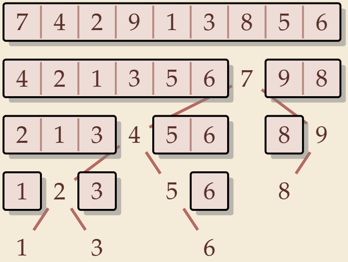


Binary Search Tree (BST)

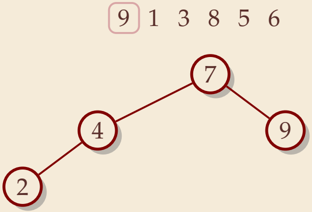


Quicksort & Binary Search Trees

Quicksort

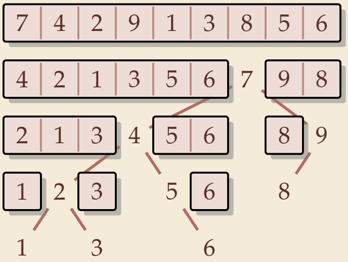


Binary Search Tree (BST)

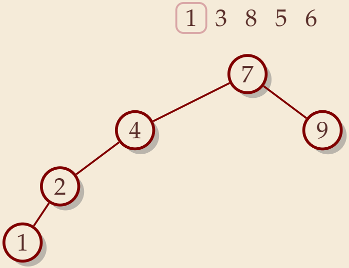


Quicksort & Binary Search Trees

Quicksort

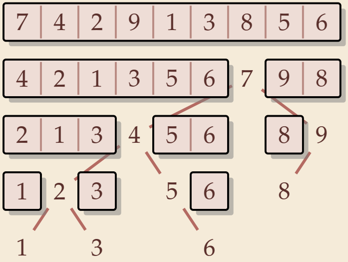


Binary Search Tree (BST)

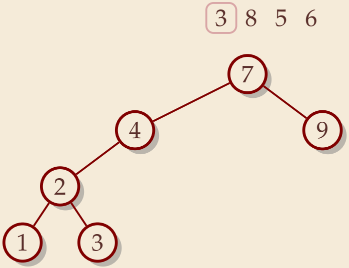


Quicksort & Binary Search Trees

Quicksort

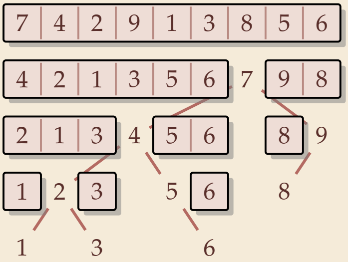


Binary Search Tree (BST)

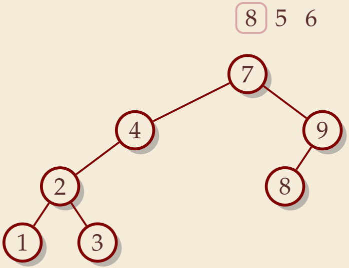


Quicksort & Binary Search Trees

Quicksort

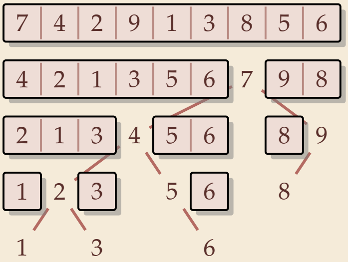


Binary Search Tree (BST)

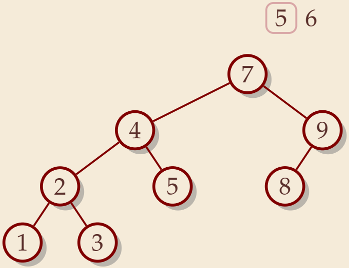


Quicksort & Binary Search Trees

Quicksort

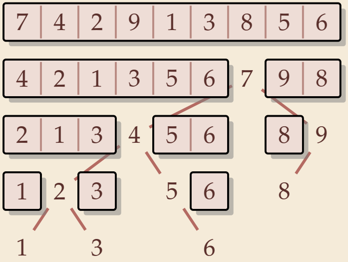


Binary Search Tree (BST)

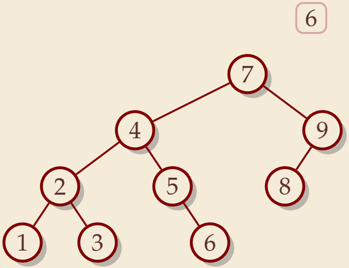


Quicksort & Binary Search Trees

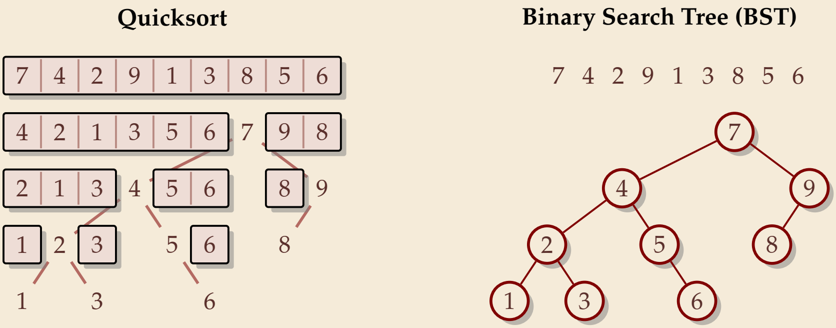
Quicksort



Binary Search Tree (BST)



Quicksort & Binary Search Trees

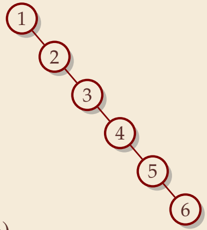


- recursion tree of quicksort $=$ binary search tree from successive insertion
- comparisons in quicksort $=$ comparisons to built BST
- comparisons in quicksort $\approx$ comparisons to search each element in BST

Quicksort – Worst Case

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

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



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

terribly slow!

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)

$\rightsquigarrow$ 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: $\Pr[\text{cost} > 10 n \lg n] = O(n^{-2.5})$
  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.

3.3 Comparison-Based Lower Bound

Lower Bounds

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

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

Lower Bounds

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

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

- To prove a statement about *all algorithms*, we must precisely define what that is!

- already know one option: the word-RAM model

- Here: use a simpler, more restricted model.

The Comparison Model

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

That's good!
Keeps algorithms general!

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

The Comparison Model

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

That's good!
Keeps algorithms general!

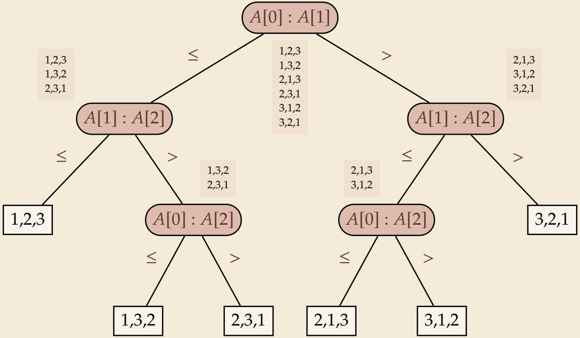
- This makes very few assumptions on the kind of objects we are sorting.
- Mergesort and Quicksort work in the comparison model.

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

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

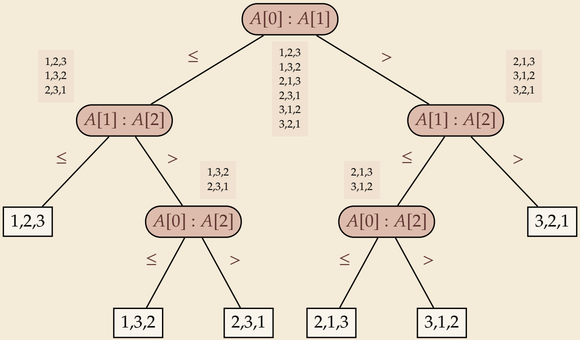
Comparison Lower Bound

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



Comparison Lower Bound

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

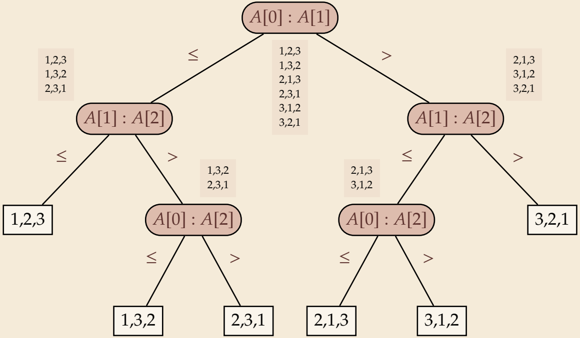


- Execution = follow a path in comparison tree.
- ⇝ height of comparison tree = worst-case # comparisons
- comparison trees are *binary* trees
- ⇝ ℓ leaves ⇝ height $\ge \lceil \lg(\ell) \rceil$
- comparison trees for sorting method must have $\ge n!$ leaves
- ⇝ height $\ge \lg(n!) \sim n \lg n$

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

Comparison Lower Bound

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



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

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

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

≈ 1.4427

Clicker Question

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

A Yes

B No

`sli.do/comp526`

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

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!*
  - $\rightsquigarrow$ Lower bounds show where to *change* the model!

How to beat a lower bound

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

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

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

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

How to beat a lower bound

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

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

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

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

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

Counting sort

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

Counting sort

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

word size

Counting sort

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

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

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

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!

$\notin$ 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!