

# 3

## Efficient Sorting – The Power of Divide & Conquer

17 October 2022

Sebastian Wild

# 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

- 3.1 Mergesort
- 3.2 Quicksort
- 3.3 Comparison-Based Lower Bound
- 3.4 Integer Sorting
- 3.5 Adaptive Sorting
- 3.6 Python's list sort
- 3.7 Order Statistics
- 3.8 Further D&C Algorithms

# 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
- ▶ 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  $A[0], \dots, A[n - 1]$

(a comparison function)

*Python:* elements support  $\leq$  operator (`__le__()`)

*Java:* Comparable class (`x.compareTo(y) <= 0`)

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

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

## Clicker Question



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



→ *sl.i.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](https://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.~~

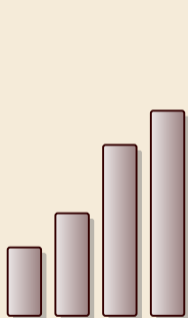


→ [sli.do/comp526](https://sli.do/comp526)

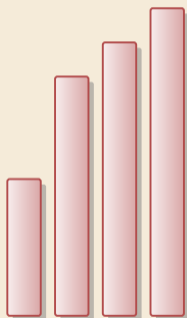
# Merging sorted lists



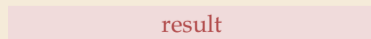
# Merging sorted lists



run1

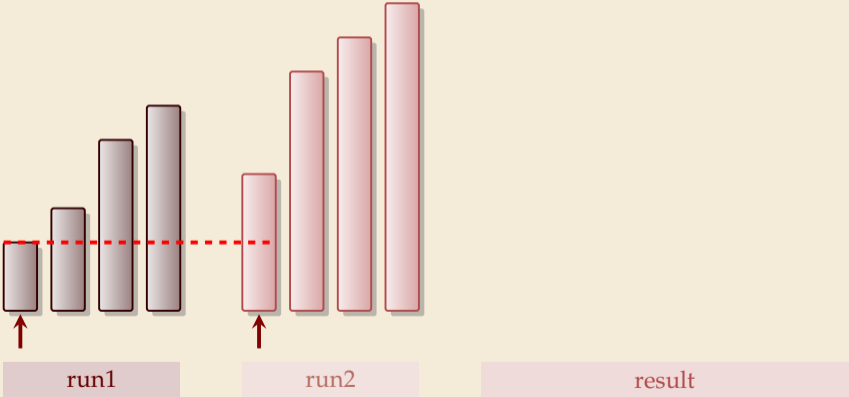


run2



result

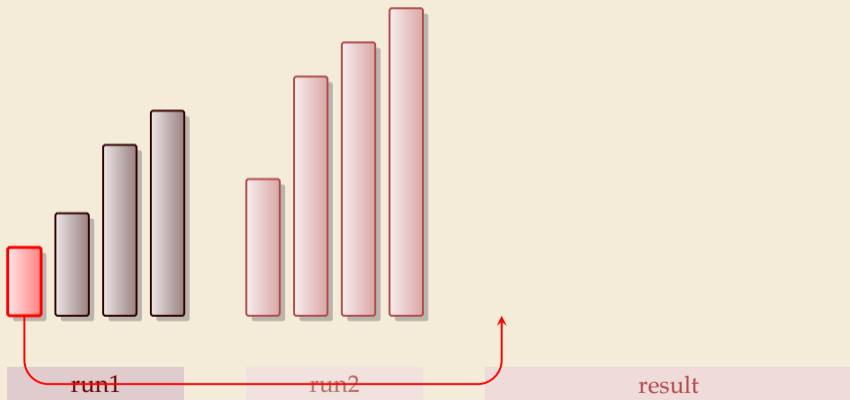
# Merging sorted lists



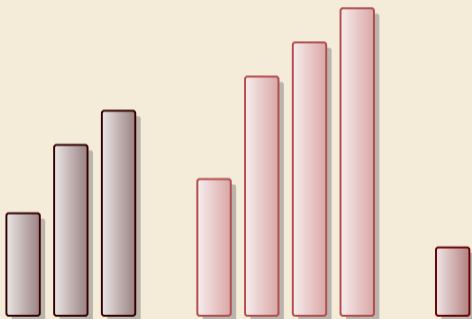
# Merging sorted lists



# Merging sorted lists



# Merging sorted lists



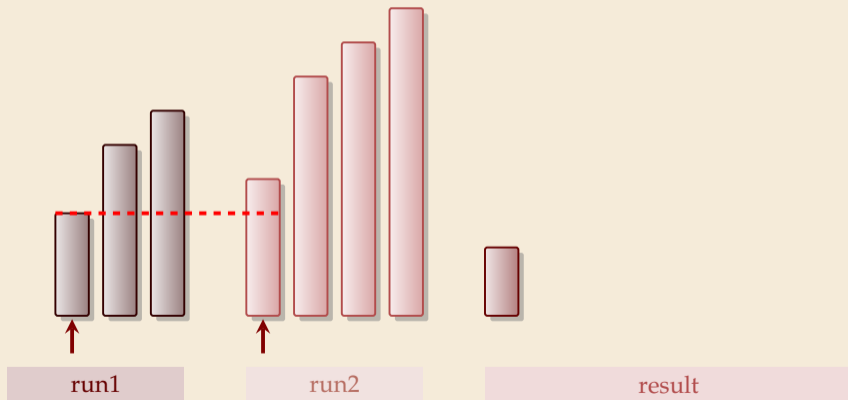
run1

run2

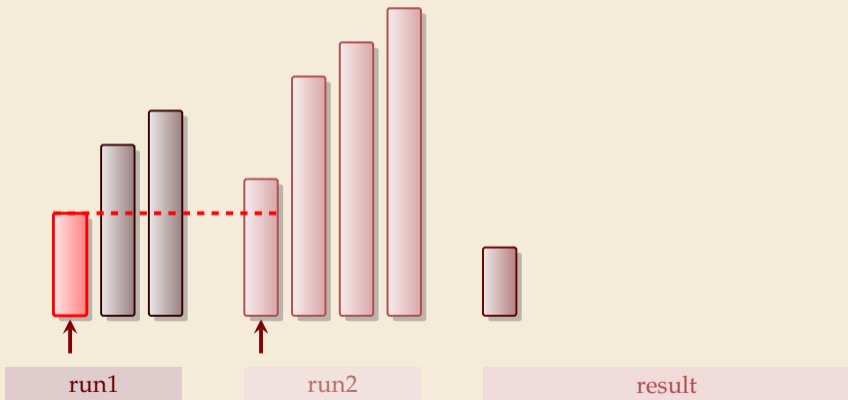
result



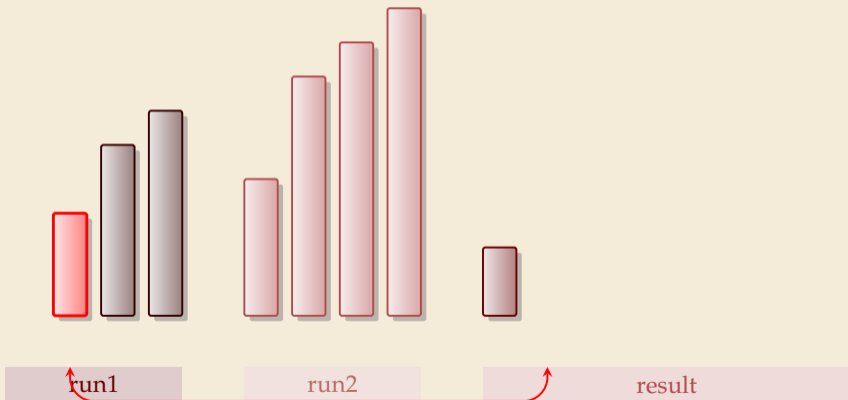
# Merging sorted lists



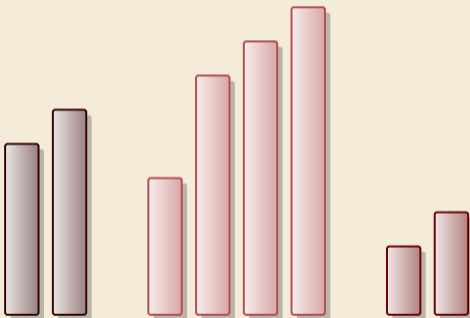
# Merging sorted lists



# Merging sorted lists



# Merging sorted lists

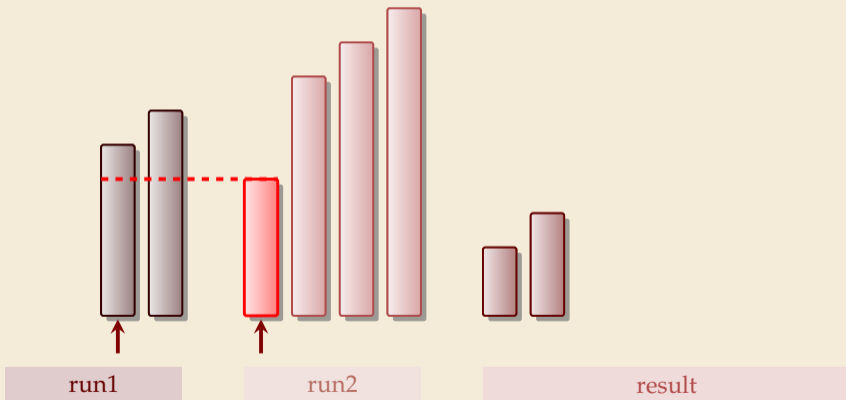


run1

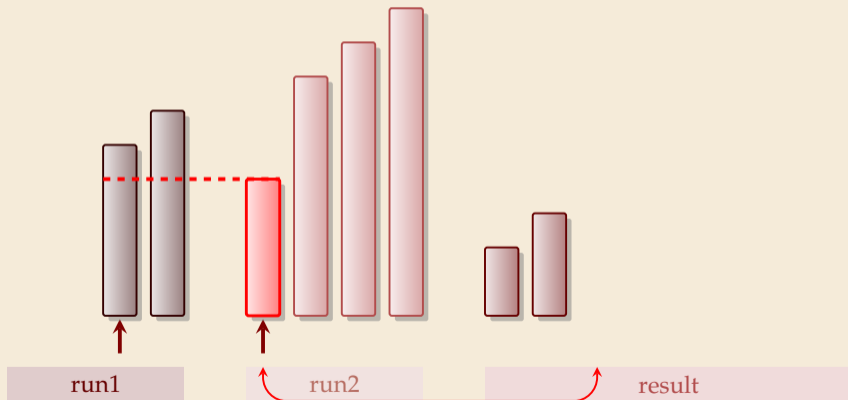
run2

result

# Merging sorted lists



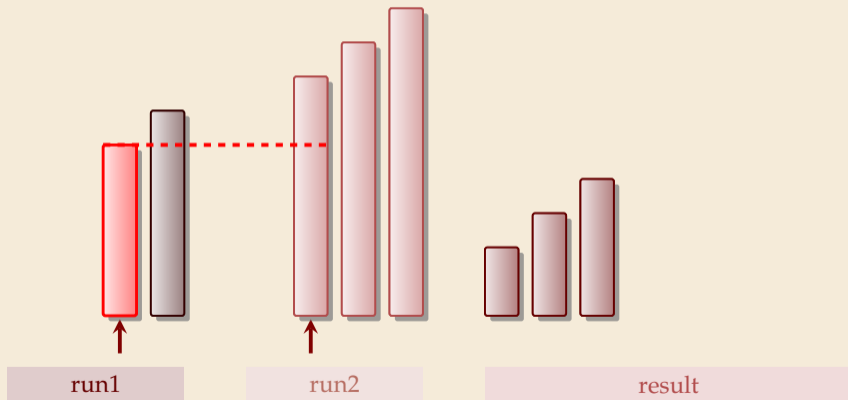
# Merging sorted lists



# Merging sorted lists

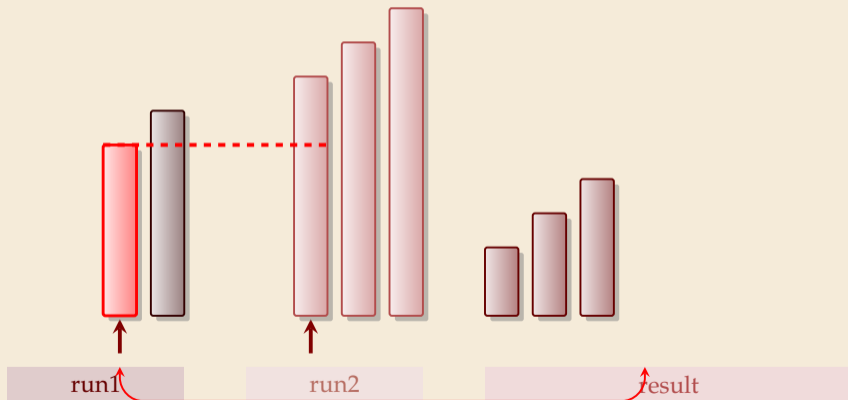


# Merging sorted lists

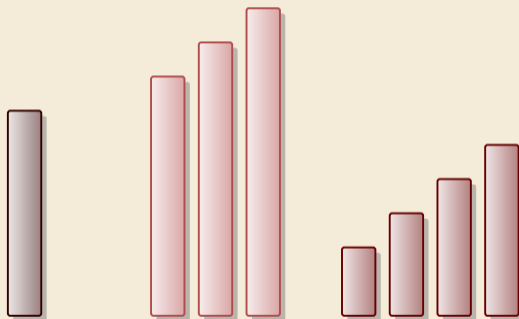




# Merging sorted lists



# Merging sorted lists

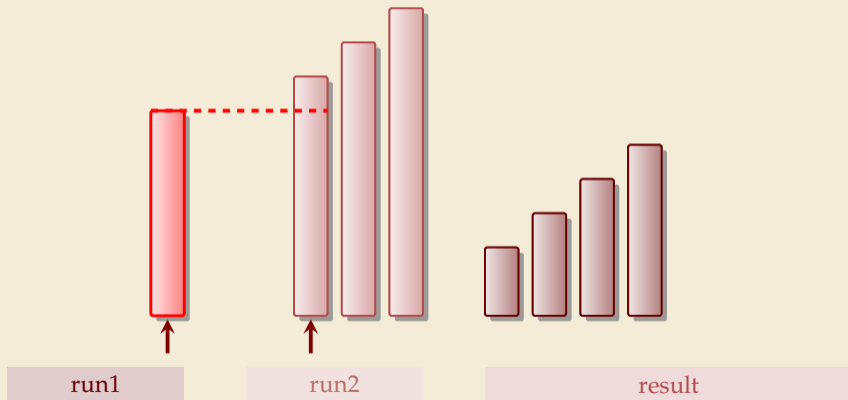


run1

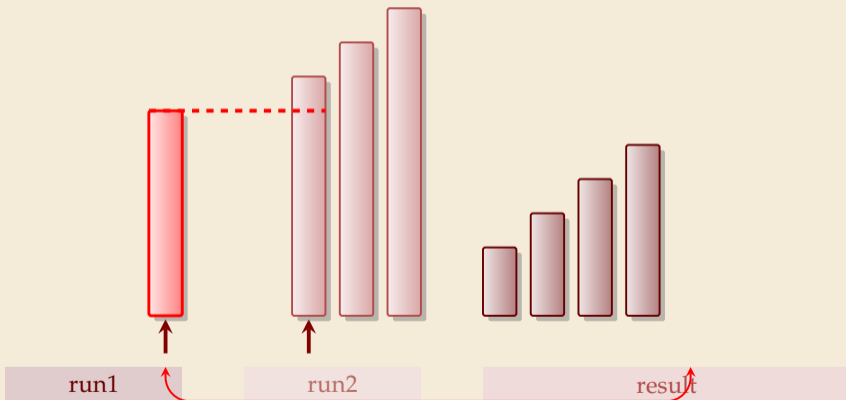
run2

result

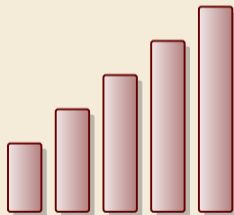
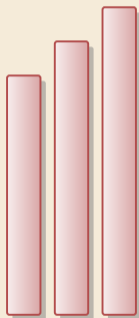
# Merging sorted lists



# Merging sorted lists



# Merging sorted lists

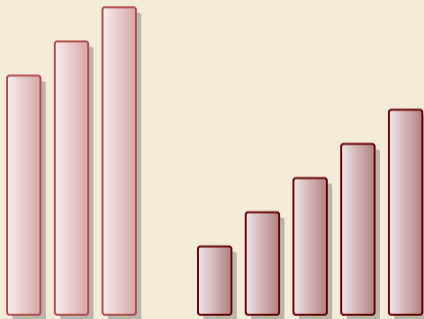


run1

run2

result

# Merging sorted lists

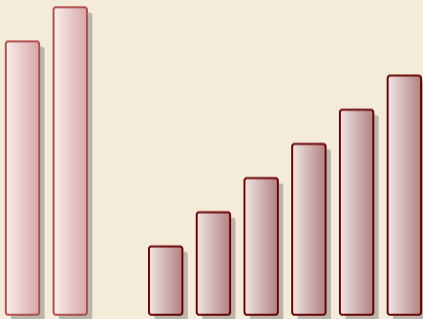


run1

run2

result

# Merging sorted lists

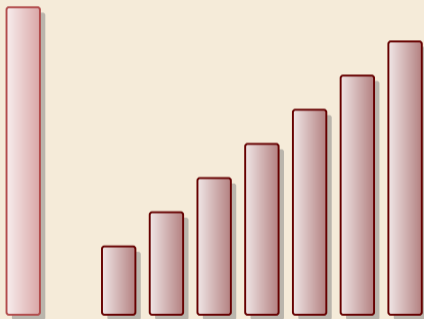


run1

run2

result

# Merging sorted lists



run1

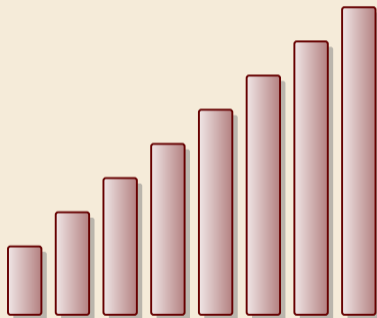
run2

result





# Merging sorted lists



run1

run2

result

## Clicker Question

What is the worst-case running time of mergesort?



**A**  $\Theta(1)$

**B**  $\Theta(\log n)$

**C**  $\Theta(\log \log n)$

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

**E**  $\Theta(n)$

**F**  $\Theta(n \log \log n)$

**G**  $\Theta(n \log n)$

**H**  $\Theta(n \log^2 n)$

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

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

**K**  $\Theta(n^3)$

**L**  $\Theta(2^n)$



→ [sli.do/comp526](https://sli.do/comp526)

## 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](https://sli.do/comp526)

# Mergesort

---

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

```
1 procedure mergesort(A[l..r])
2   n := r - l
3   if n ≤ 1 return
4   m := l + ⌊ $\frac{n}{2}$ ⌋
5   mergesort(A[l..m])
6   mergesort(A[m..r])
7   merge(A[l..m], A[m..r], buf)
8   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 access*  
**Analysis:** count “element visits” (read and/or write)

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

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

same for best and worst case!

Simplification  $n = 2^k$

$$C(2^k) = \begin{cases} 0 & k \leq 0 \\ 2 \cdot C(2^{k-1}) + 2 \cdot 2^k & k \geq 1 \end{cases} = 2 \cdot 2^k + 2^2 \cdot 2^{k-1} + 2^3 \cdot 2^{k-2} + \dots + 2^k \cdot 2^1 = 2k \cdot 2^k$$

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

$$C(2^k) \stackrel{!}{=} 2 C(2^{k-1}) + 2 \cdot 2^k$$

$k \geq 2$

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

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

$2^{k+1} \qquad 2^{k+1}$

# 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](https://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.~~

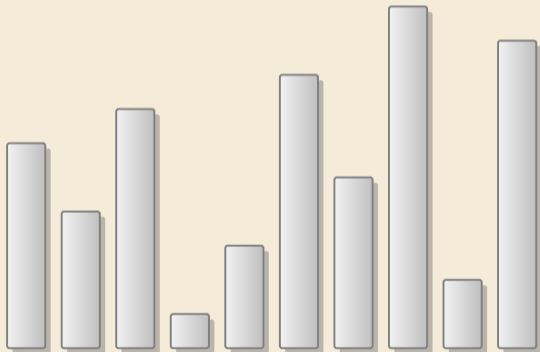


→ [sli.do/comp526](https://sli.do/comp526)

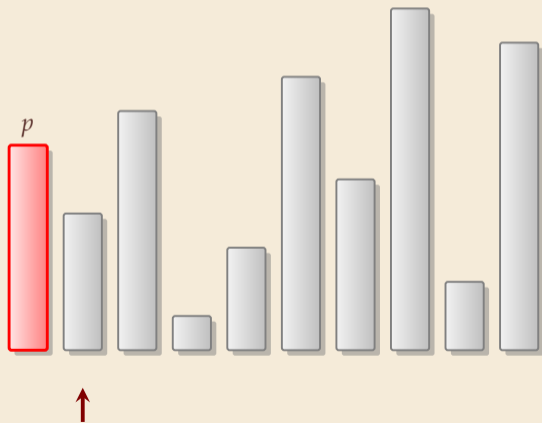
## Partitioning around a pivot



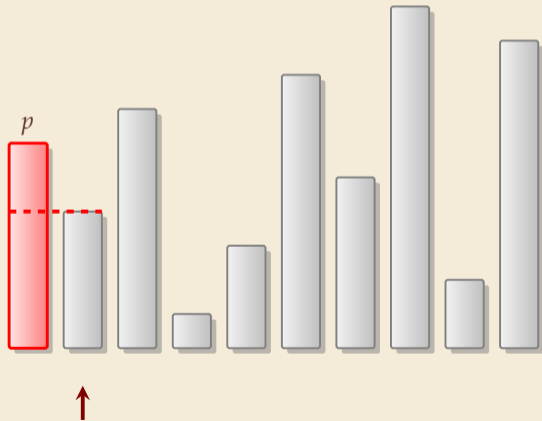
## Partitioning around a pivot



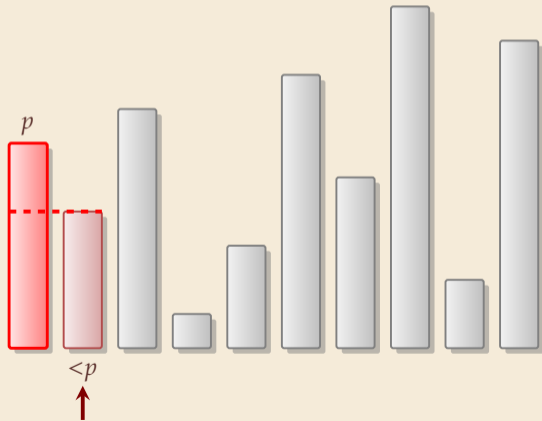
## Partitioning around a pivot



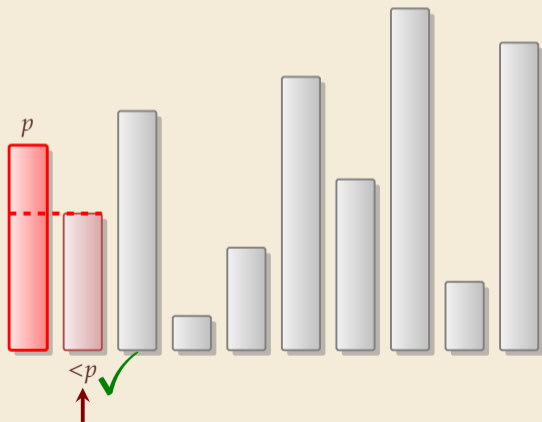
## Partitioning around a pivot



# Partitioning around a pivot

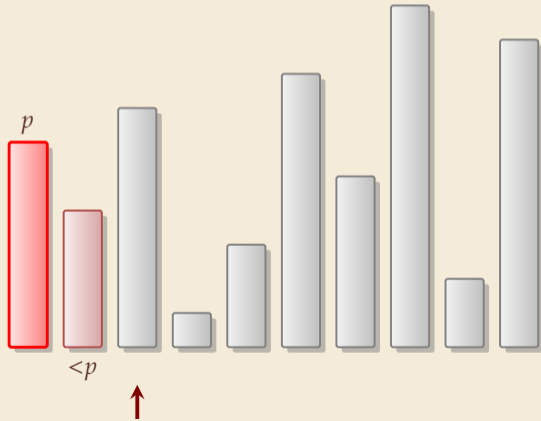


# Partitioning around a pivot

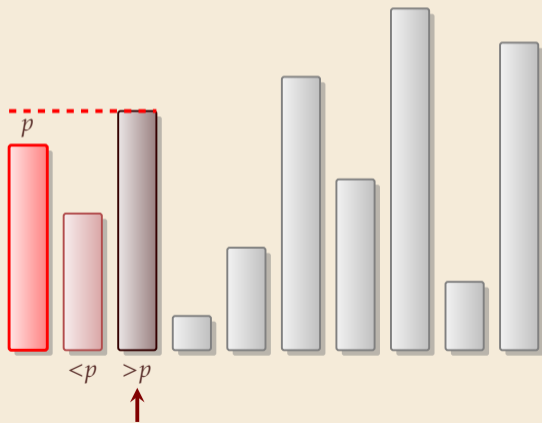




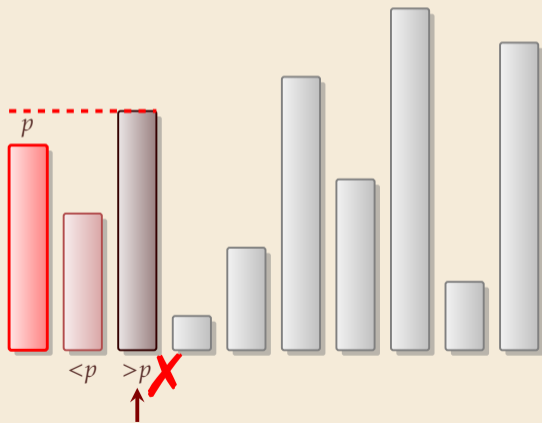
# Partitioning around a pivot



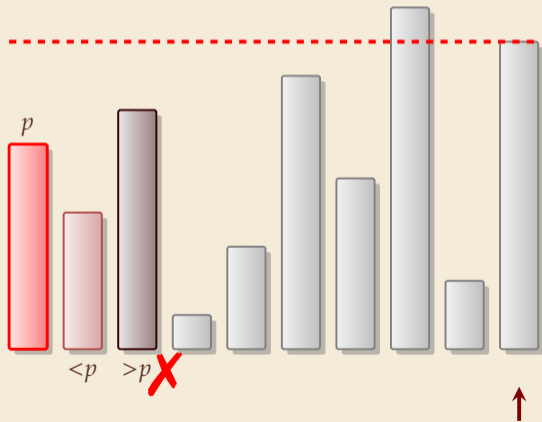
# Partitioning around a pivot



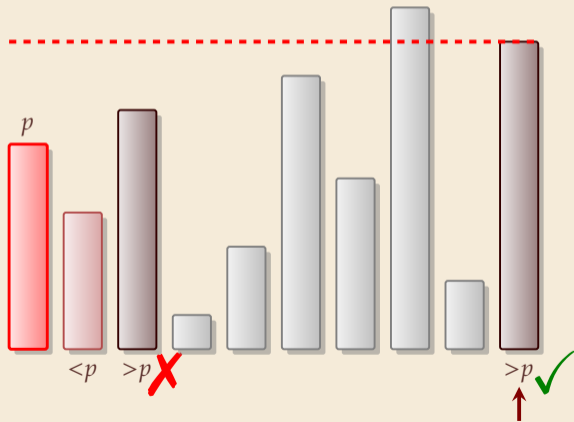
# Partitioning around a pivot



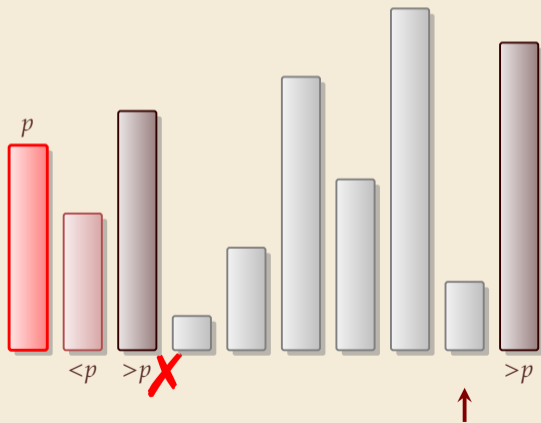
# Partitioning around a pivot



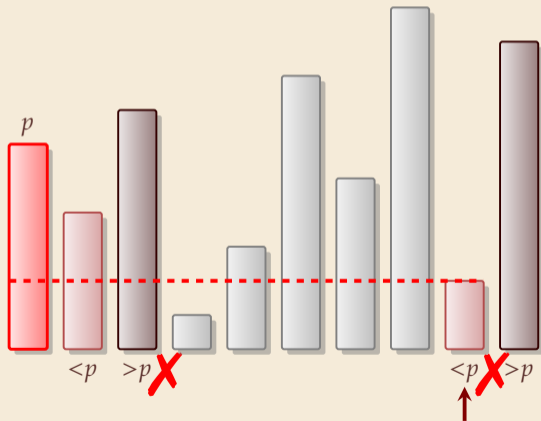
# Partitioning around a pivot



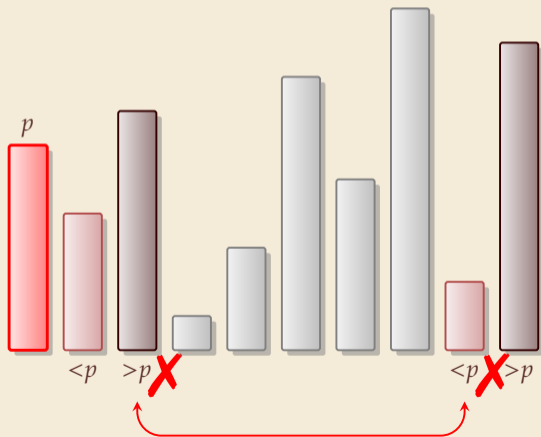
# Partitioning around a pivot



# Partitioning around a pivot

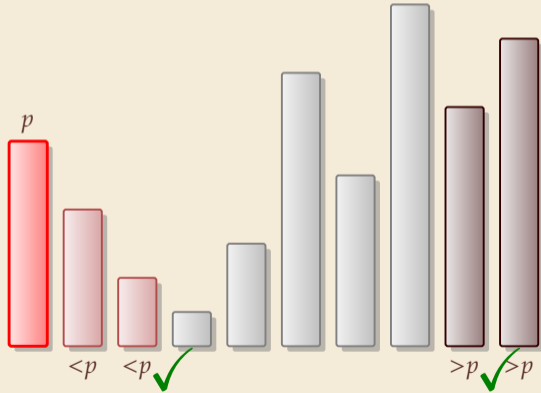


# Partitioning around a pivot

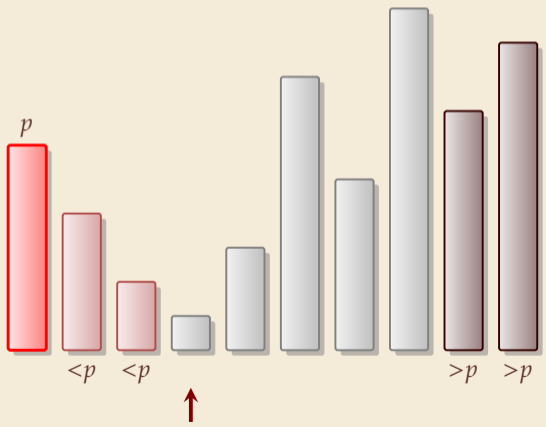




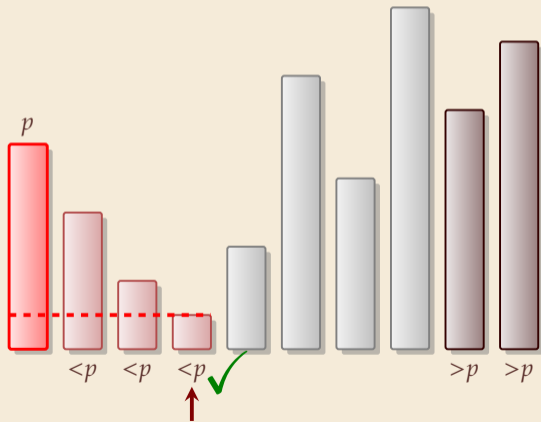
# Partitioning around a pivot



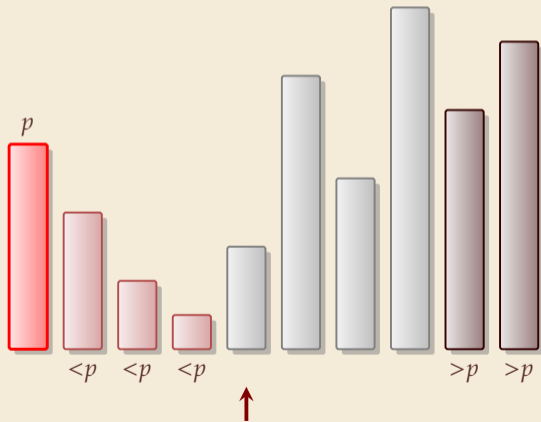
# Partitioning around a pivot



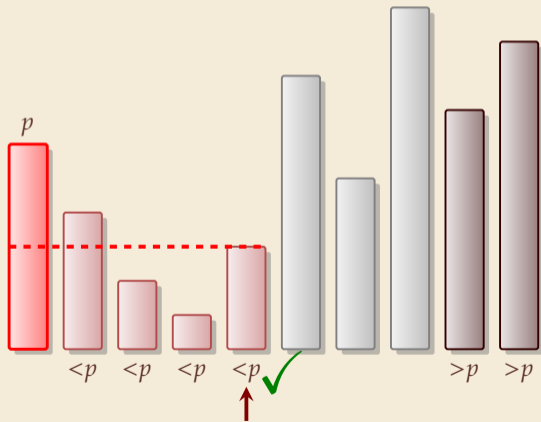
# Partitioning around a pivot



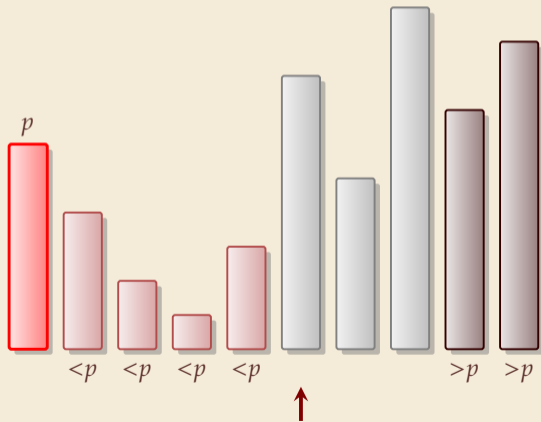
# Partitioning around a pivot



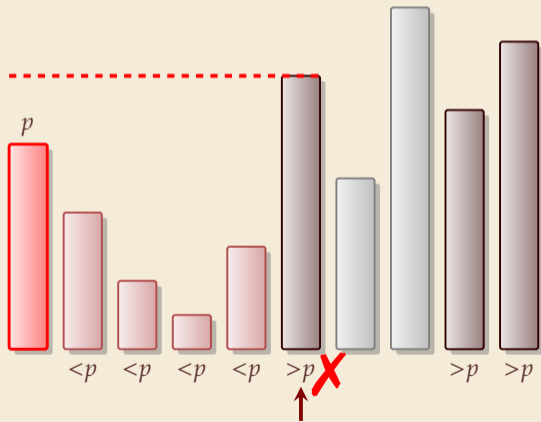
# Partitioning around a pivot



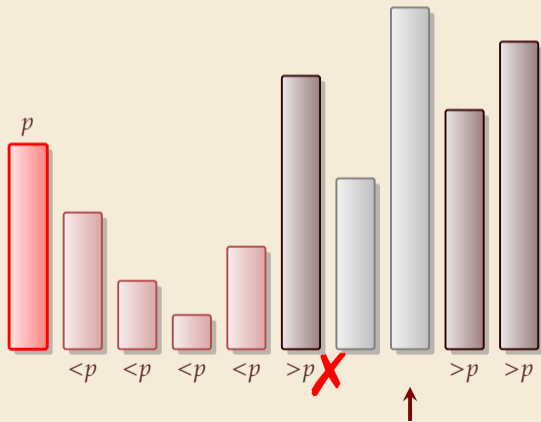
# Partitioning around a pivot



# Partitioning around a pivot

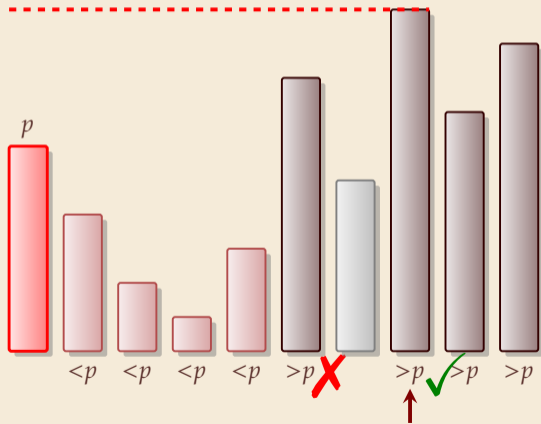


# Partitioning around a pivot

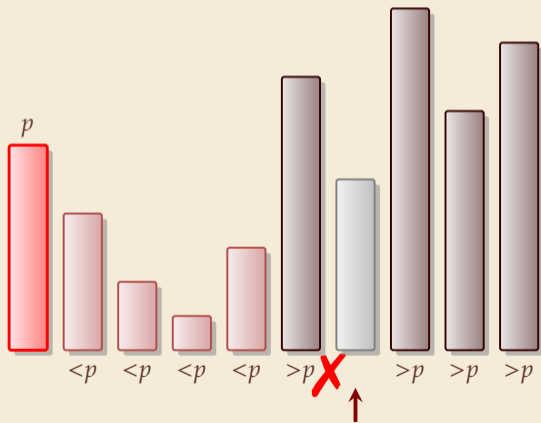




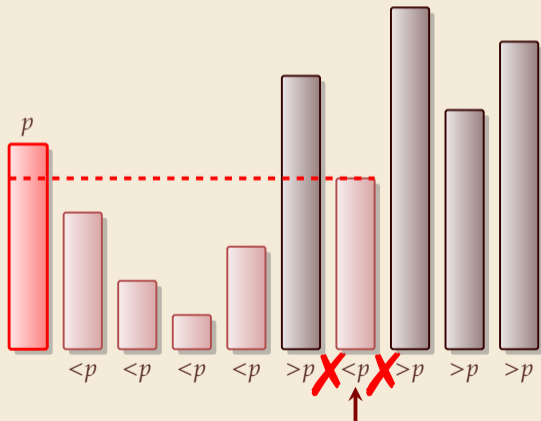
# Partitioning around a pivot



# Partitioning around a pivot

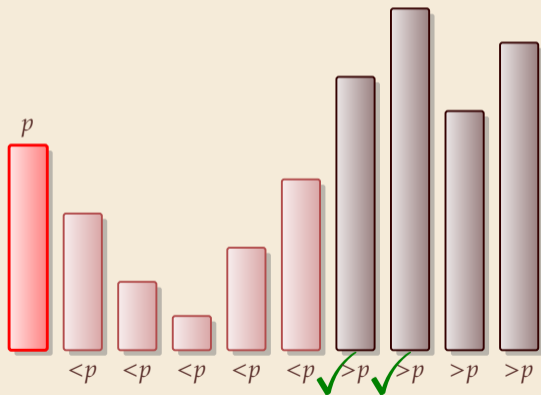


# Partitioning around a pivot

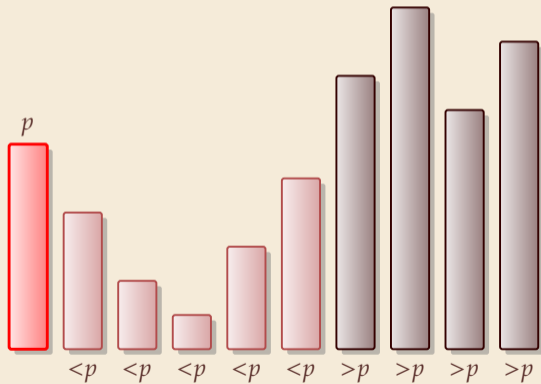




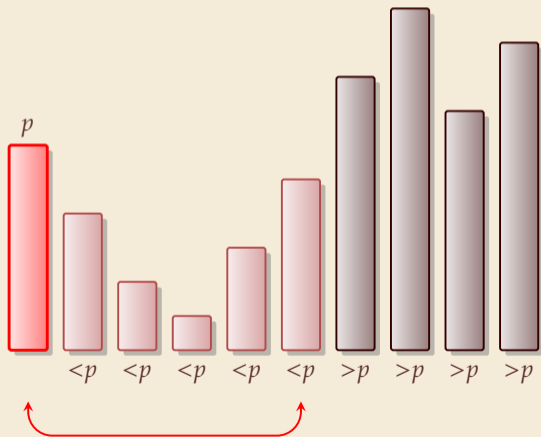
# Partitioning around a pivot



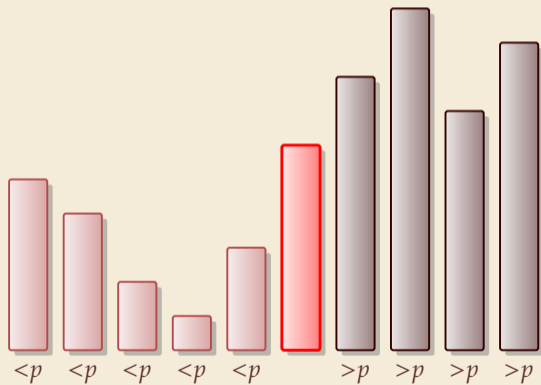
## Partitioning around a pivot



# Partitioning around a pivot

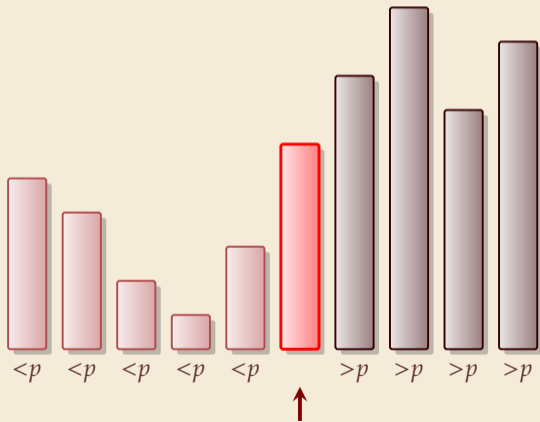


## Partitioning around a pivot





## Partitioning around a pivot



- ▶ 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!

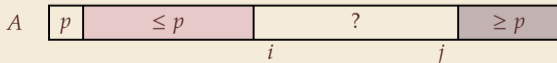
(if you ever have to)

---

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



# Quicksort

---

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



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



→ [sli.do/comp526](https://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](https://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

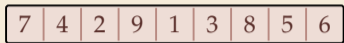
## Quicksort

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

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

# Quicksort & Binary Search Trees

## Quicksort



# 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

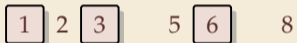
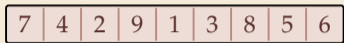
7	4	2	9	1	3	8	5	6
---	---	---	---	---	---	---	---	---

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

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

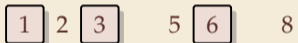
# Quicksort & Binary Search Trees

## Quicksort



# Quicksort & Binary Search Trees

## Quicksort



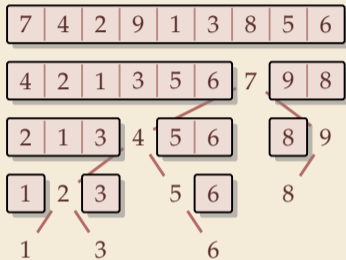
1

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)

4 2 9 1 3 8 5 6

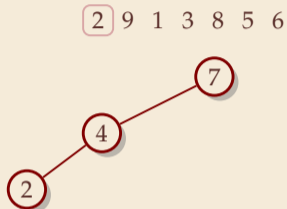


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)

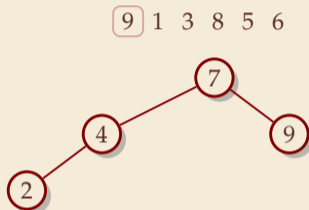


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)

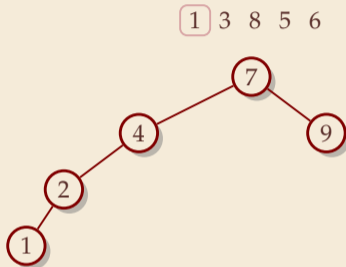


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)

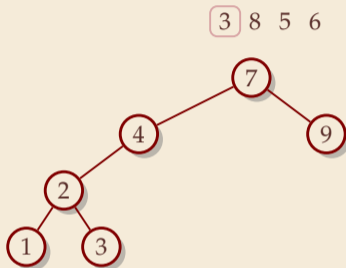


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)

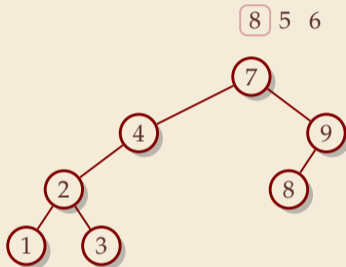


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)



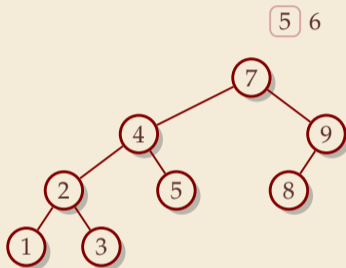


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)

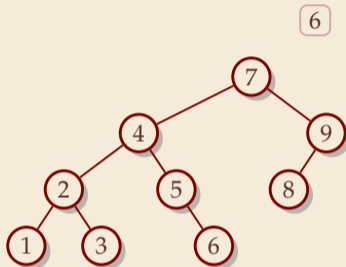


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)

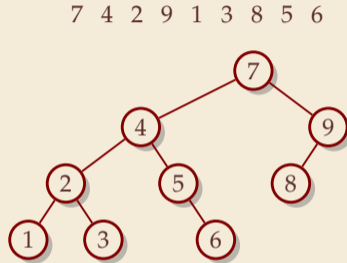


# Quicksort & Binary Search Trees

## Quicksort



## Binary Search Tree (BST)



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

## Quicksort – Worst Case

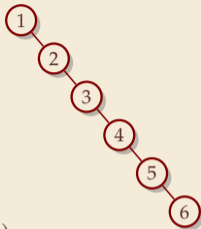
► Problem: BSTs can degenerate

► 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  $\sim 2 \ln(2) \cdot n \lg n \approx 1.39n \lg n$  *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.

## 3.3 Comparison-Based Lower Bound

# Lower Bounds

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



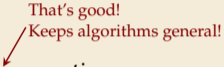
# Lower Bounds

- ▶ **Lower bound:** mathematical proof that *no algorithm* can do better.
  - ▶ very powerful concept: bulletproof *impossibility* result  
≈ *conservation of energy* in physics
  - ▶ **(unique?) feature of computer science:**  
for many problems, solutions are known that (asymptotically) **achieve the lower bound**  
↪ can speak of “*optimal algorithms*”
  
- ▶ To prove a statement about *all algorithms*, we must precisely define what that is!
  
- ▶ already know one option: the word-RAM model
  
- ▶ Here: use a simpler, more restricted model.

# The Comparison Model

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

# The Comparison Model

- ▶ In the *comparison model* data can only be accessed in two ways:
  - ▶ comparing two elements
  - ▶ moving elements around (e. g. copying, swapping)
  - ▶ Cost: number of these operations.
- ▶ This makes very few assumptions on the kind of objects we are sorting. 
- ▶ 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~~ <sup>comparisons (here)</sup>.

That's good!  
Keeps algorithms general!

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

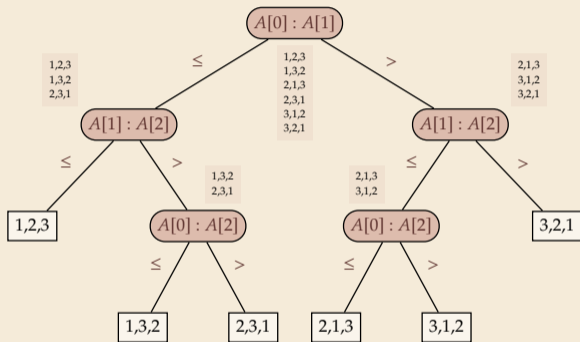
▶ Mergesort and Quicksort work in the comparison model.

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

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

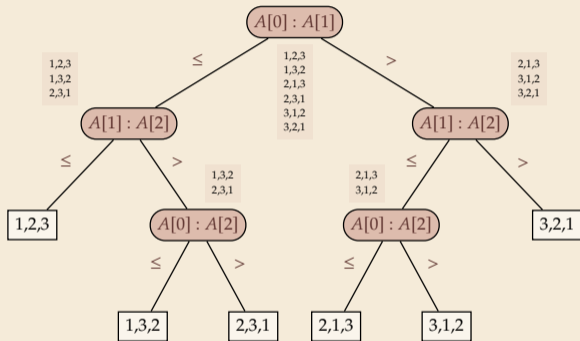
# Comparison Lower Bound

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



# Comparison Lower Bound

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



▶ Execution = follow a path in comparison tree.

↪ height of comparison tree = worst-case # comparisons

▶ comparison trees are *binary* trees

↪  $\ell$  leaves  $\rightsquigarrow$  height  $\geq \lceil \lg(\ell) \rceil$

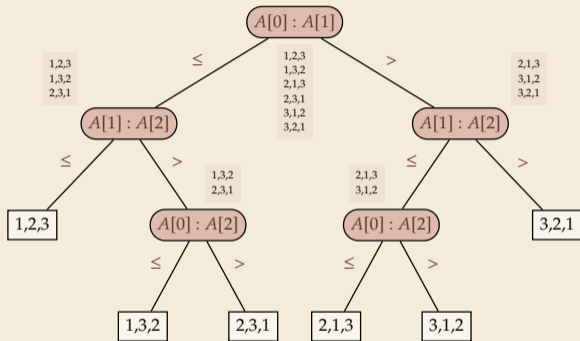
▶ comparison trees for sorting method must have  $\geq n!$  leaves

↪ height  $\geq \lg(n!) \sim n \lg n$

more precisely:  $\lg(n!) = \underbrace{n \lg n - \lg(e)n + O(\log n)}_{\sim n \lg n}$

# Comparison Lower Bound

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



▶ Execution = follow a path in comparison tree.

↪ height of comparison tree = worst-case # comparisons

▶ comparison trees are *binary* trees

↪  $\ell$  leaves ↪ height  $\geq \lceil \lg(\ell) \rceil$

▶ comparison trees for sorting method must have  $\geq n!$  leaves

↪ height  $\geq \lg(n!) \sim n \lg n$

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

▶ Mergesort achieves  $\sim n \lg n$  comparisons ↪ asymptotically comparison-optimal!

▶ Open (theory) problem: Sorting algorithm with  $n \lg n - \lg(e)n + o(n)$  comparisons?

$\approx 1.4427$

## Clicker Question



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

**A** Yes

**B** No



→ [sli.do/comp526](https://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](https://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*!
  - ↪ Lower bounds show where to *change* the model!

## How to beat a lower bound

- ▶ Does the above lower bound mean, sorting always takes time  $\Omega(n \log n)$ ?
- ▶ **Not necessarily;** only in the *comparison model*!
  - ↪ Lower bounds show where to *change* the model!
- ▶ Here: sort  $n$  integers
  - ▶ can do *a lot* with integers: add them up, compute averages, ... (full power of word-RAM)
  - ↪ we are **not** working in the comparison model
  - ↪ *above lower bound does not apply!* /

## How to beat a lower bound

- ▶ Does the above lower bound mean, sorting always takes time  $\Omega(n \log n)$ ?
- ▶ **Not necessarily;** only in the *comparison model*!
  - ↪ Lower bounds show where to *change* the model!
- ▶ Here: sort  $n$  **integers**
  - ▶ can do *a lot* with integers: add them up, compute averages, ... (full power of word-RAM)
  - ↪ we are **not** working in the comparison model
  - ↪ *above lower bound does not apply!*
  - ▶ but: a priori unclear how much arithmetic helps for sorting ...

# Counting sort

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

# Counting sort

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

word size

## Counting sort

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

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

$$\Theta(U + \sum_{k=0} C[k]) = \Theta(U + n)$$

$\rightsquigarrow$  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  $w = \Omega(\log^{2+\epsilon} n)$ : 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!

⊕ 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  $w = \Omega(\log^{2+\epsilon} n)$ : linear time (*signature sort*)
    - ▶ for  $w$  in between: can do  $O(n\sqrt{\lg \lg n})$  (very complicated algorithm)  
don't know if that is best possible!

\* \* \*

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

## Clicker Question

Which statements are correct? Select all that apply.

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



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



→ [sli.do/comp526](https://sli.do/comp526)

## Clicker Question

Which statements are correct? Select all that apply.

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



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



→ [sli.do/comp526](https://sli.do/comp526)

# Part II

*Exploiting presortedness*

## 3.5 Adaptive Sorting

## Adaptive sorting

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

## Adaptive sorting

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



Can we do better if the input is already “*almost sorted*”?

Scenarios where this may arise naturally:

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

$\rightsquigarrow$  Ideally, algorithms should *adapt* to input: *the more sorted the input, the faster the algorithm*  
... but how to do that!?



## Warmup: check for sorted inputs

- ▶ Any method could first check if input already completely in order!



Best case becomes  $\Theta(n)$  with  $n - 1$  comparisons!



Usually  $n - 1$  extra comparisons and pass over data “wasted”



Only catches a single, extremely special case . . .

## Warmup: check for sorted inputs

- ▶ Any method could first check if input already completely in order!
  - 👍 Best case becomes  $\Theta(n)$  with  $n - 1$  comparisons!
  - 👎 Usually  $n - 1$  extra comparisons and pass over data “wasted”
  - 👎 Only catches a single, extremely special case . . .
  
- ▶ For divide & conquer algorithms, could check in each recursive call!
  - 👍 Potentially exploits partial sortedness!
  - 👎 usually adds  $\Omega(n \log n)$  extra comparisons

## Warmup: check for sorted inputs

- ▶ Any method could first check if input already completely in order!



Best case becomes  $\Theta(n)$  with  $n - 1$  comparisons!



Usually  $n - 1$  extra comparisons and pass over data “wasted”



Only catches a single, extremely special case . . .

- ▶ For divide & conquer algorithms, could check in each recursive call!



Potentially exploits partial sortedness!



usually adds  $\Omega(n \log n)$  extra comparisons



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

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

*How effective is this idea?*

---

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

---

## Mergesort with sorted check – Analysis

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



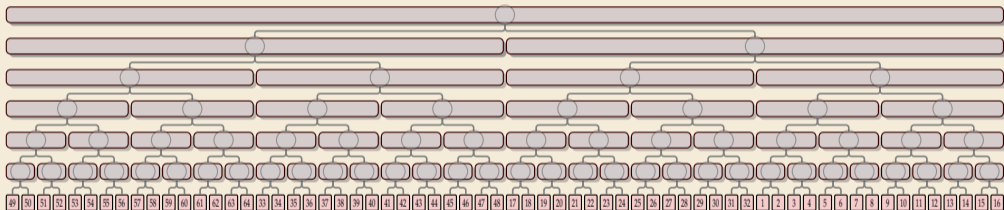
## Mergesort with sorted check – Analysis

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

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

# Mergesort with sorted check – Analysis

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



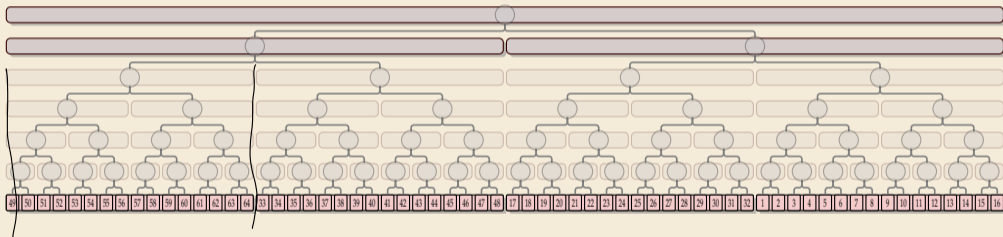
Merge costs:



384 Standard Mergesort

# Mergesort with sorted check – Analysis

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



Merge costs:



384 Standard Mergesort

128 with sorted check

*Sorted check can help a lot!*

## Alignment issues

- ▶ In previous example, each run of length  $\ell$  saved us  $\ell \lg(\ell)$  in merge cost.

= exactly the cost of *creating* this run in mergesort had it not already existed

↪ best savings we can hope for!

↪ Are overall merge costs  $\mathcal{H}(\ell_1, \dots, \ell_r) := \underbrace{n \lg(n)}_{\text{mergesort}} - \underbrace{\sum_{i=1}^r \ell_i \lg(\ell_i)}_{\text{savings from runs}} ?$

$\ell_i = \text{length of } i\text{th run}$



# Alignment issues

- ▶ In previous example, each run of length  $\ell$  saved us  $\ell \lg(\ell)$  in merge cost.

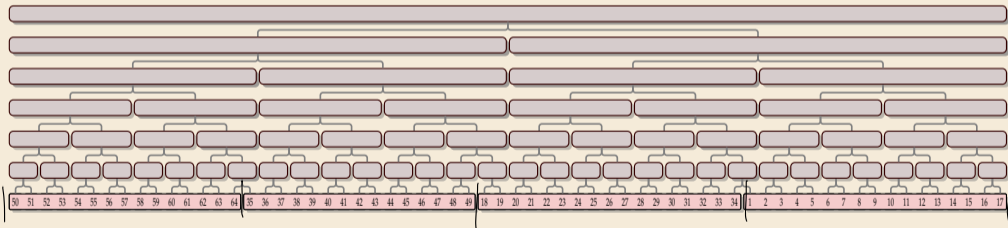
= exactly the cost of *creating* this run in mergesort had it not already existed

↪ best savings we can hope for!

↪ Are overall merge costs  $\mathcal{H}(\ell_1, \dots, \ell_r) := \underbrace{n \lg(n)}_{\text{mergesort}} - \underbrace{\sum_{i=1}^r \ell_i \lg(\ell_i)}_{\text{savings from runs}} ?$

$\ell_i = \text{length of } i\text{th run}$

Unfortunately, not quite:



Merge costs:



384 Standard Mergesort



127.8  $\mathcal{H}(15, 15, 17, 17)$

# Alignment issues

- ▶ In previous example, each run of length  $\ell$  saved us  $\ell \lg(\ell)$  in merge cost.

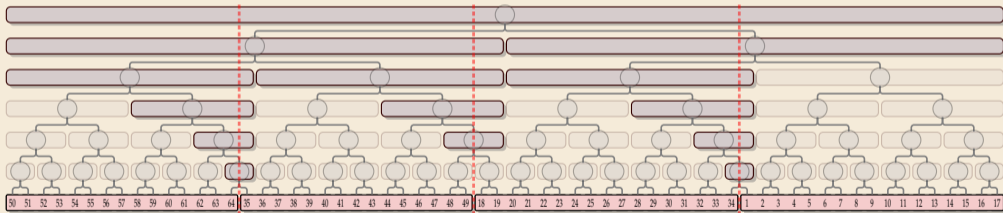
= exactly the cost of *creating* this run in mergesort had it not already existed

↪ best savings we can hope for!

↪ Are overall merge costs  $\mathcal{H}(\ell_1, \dots, \ell_r) := \underbrace{n \lg(n)}_{\text{mergesort}} - \underbrace{\sum_{i=1}^r \ell_i \lg(\ell_i)}_{\text{savings from runs}} ?$

$\ell_i = \text{length of } i\text{th run}$

Unfortunately, not quite:



Merge costs:



384 Standard Mergesort



216 with sorted check



127.8  $\mathcal{H}(15, 15, 17, 17)$

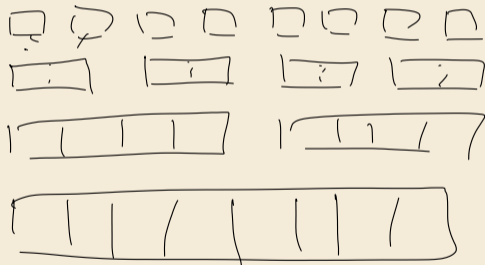
# Natural Bottom-Up Mergesort

- Can we do better by explicitly detecting runs?

---

```
1 procedure bottomUpMergesort( $A[0..n]$ )
2    $Q :=$  new Queue // runs to merge
3   // Phase 1: Enqueue singleton runs
4   for  $i = 0, \dots, n - 1$  do
5      $Q.enqueue((i, i + 1))$ 
6   // Phase 2: Merge runs level-wise
7   while  $\neg Q.isEmpty()$ 
8      $Q' :=$  new Queue
9     while  $Q.size() \geq 2$ 
10       $(i_1, j_1) := Q.dequeue()$ 
11       $(i_2, j_2) := Q.dequeue()$ 
12      merge( $A[i_1..j_1], A[i_2..j_2], buf$ )
13      copy  $buf$  to  $A[i_1..j_2]$ 
14       $Q'.enqueue((i_1, j_2))$ 
15     if  $\neg Q.isEmpty()$ 
16        $Q'.enqueue(Q.dequeue())$ 
17    $Q := Q'$ 
```

---



# Natural Bottom-Up Mergesort

- Can we do better by explicitly detecting runs?

---

```
1 procedure bottomUpMergesort( $A[0..n]$ )
2    $Q :=$  new Queue // runs to merge
3   // Phase 1: Enqueue singleton runs
4   for  $i = 0, \dots, n - 1$  do
5      $Q.enqueue((i, i + 1))$ 
6   // Phase 2: Merge runs level-wise
7   while  $\neg Q.isEmpty()$ 
8      $Q' :=$  new Queue
9     while  $Q.size() \geq 2$ 
10       $(i_1, j_1) := Q.dequeue()$ 
11       $(i_2, j_2) := Q.dequeue()$ 
12       $merge(A[i_1..j_1], A[i_2..j_2], buf)$ 
13      copy  $buf$  to  $A[i_1..j_2]$ 
14       $Q'.enqueue((i_1, j_2))$ 
15   if  $\neg Q.isEmpty()$ 
16      $Q'.enqueue(Q.dequeue())$ 
17    $Q := Q'$ 
```

---

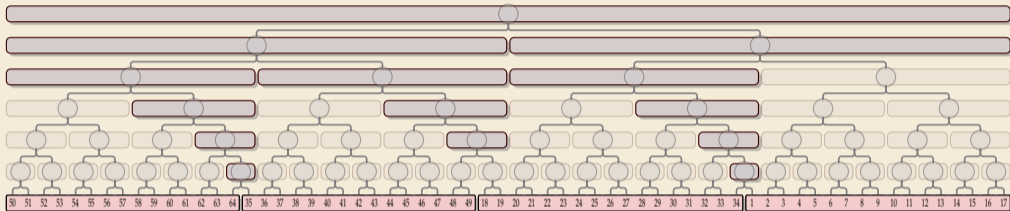
---

```
1 procedure naturalMergesort( $A[0..n]$ )
2    $Q :=$  new Queue;  $i := 0$  find run  $A[i..j]$ 
3   while  $i < n$  starting at  $i$ 
4      $j := i + 1$ 
5     while  $A[j] \geq A[j - 1]$  do  $j := j + 1$ 
6      $Q.enqueue((i, j)); i := j$ 
7   while  $\neg Q.isEmpty()$ 
8      $Q' :=$  new Queue
9     while  $Q.size() \geq 2$ 
10       $(i_1, j_1) := Q.dequeue()$ 
11       $(i_2, j_2) := Q.dequeue()$ 
12       $merge(A[i_1..j_1], A[i_2..j_2], buf)$ 
13      copy  $buf$  to  $A[i_1..j_2]$ 
14       $Q'.enqueue((i_1, j_2))$ 
15   if  $\neg Q.isEmpty()$ 
16      $Q'.enqueue(Q.dequeue())$ 
17    $Q := Q'$ 
```

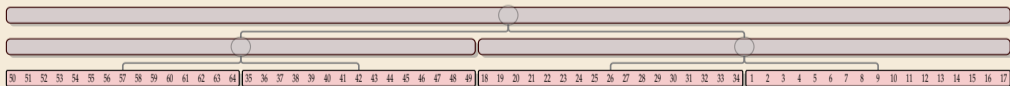
---

# Natural Bottom-Up Mergesort – Analysis

- Works well runs of roughly equal size, regardless of alignment . . .

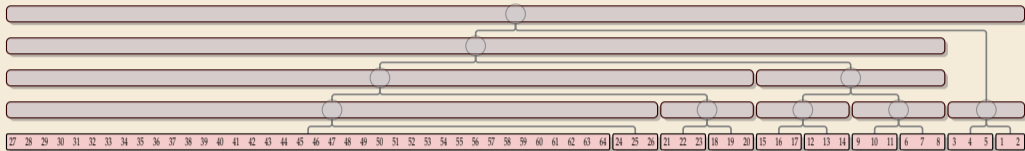


Merge costs:

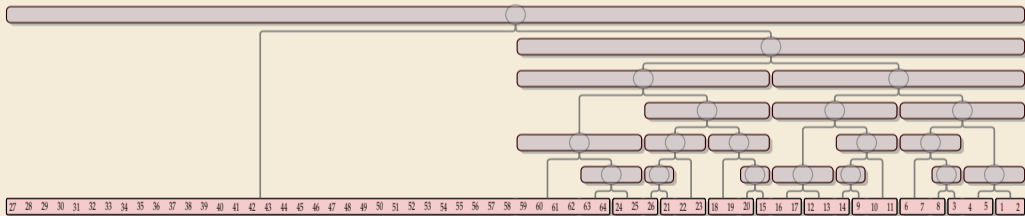


# Natural Bottom-Up Mergesort – Analysis [2]

► ... but less so for uneven run lengths



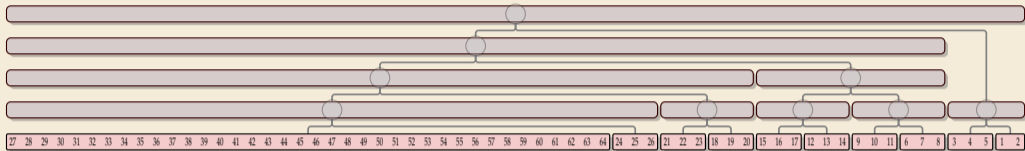
246 Natural bottom-up mergesort



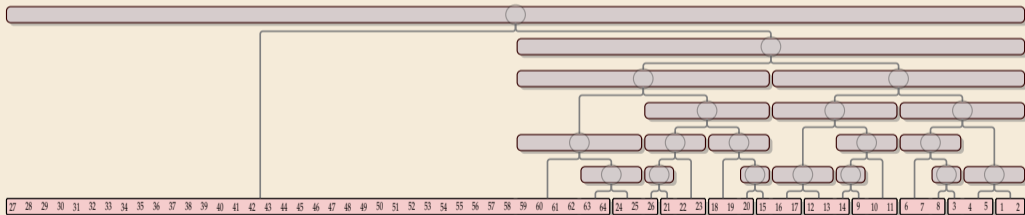
196 Standard mergesort with sorted check

# Natural Bottom-Up Mergesort – Analysis [2]

► ... but less so for uneven run lengths



246 Natural bottom-up mergesort



196 Standard mergesort with sorted check

... can't we have both at the same time?!

## Good merge orders

◀◀ *Let's take a step back and breathe.*



## Good merge orders

◀◀ *Let's take a step back and breathe.*

► Conceptually, there are two tasks:

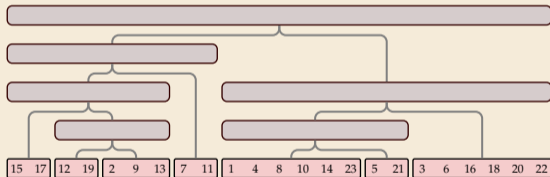
1. Detect and use existing runs in the input  $\rightsquigarrow \ell_1, \dots, \ell_r$  (easy)
2. Determine a favorable *order of merges* of runs (“automatic” in top-down mergesort)

# Good merge orders

◀◀ Let's take a step back and breathe.

► Conceptually, there are two tasks:

1. Detect and use existing runs in the input  $\rightsquigarrow l_1, \dots, l_r$  (easy) ✓
2. Determine a favorable *order of merges of runs* ("automatic" in top-down mergesort)



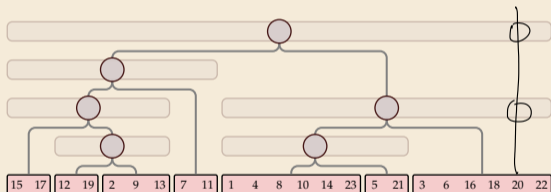
Merge cost = total area of


## Good merge orders

◀◀ Let's take a step back and breathe.

► Conceptually, there are two tasks:

1. Detect and use existing runs in the input  $\rightsquigarrow l_1, \dots, l_r$  (easy) ✓
2. Determine a favorable *order of merges of runs* ("automatic" in top-down mergesort)



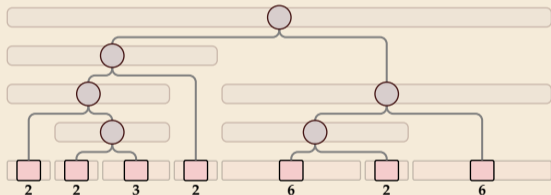
**Merge cost** = total area of   
= total length of paths to all array entries


# Good merge orders

◀◀ Let's take a step back and breathe.

► Conceptually, there are two tasks:

1. Detect and use existing runs in the input  $\rightsquigarrow l_1, \dots, l_r$  (easy) ✓
2. Determine a favorable *order of merges* of runs ("automatic" in top-down mergesort)



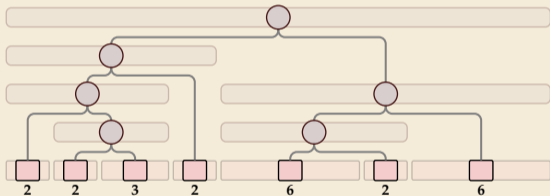
**Merge cost** = total area of   
= total length of paths to all array entries  
=  $\sum_{w \text{ leaf}} \text{weight}(w) \cdot \text{depth}(w)$


# Good merge orders

◀ Let's take a step back and breathe.

► Conceptually, there are two tasks:

1. Detect and use existing runs in the input  $\rightsquigarrow l_1, \dots, l_r$  (easy) ✓
2. Determine a favorable *order of merges of runs* ("automatic" in top-down mergesort)



**Merge cost** = total area of   
= total length of paths to all array entries  
=  $\sum_{w \text{ leaf}} \text{weight}(w) \cdot \text{depth}(w)$

well-understood problem  
with known algorithms

$\rightsquigarrow$  *optimal* merge tree  
= optimal *binary search tree*  
for leaf weights  $l_1, \dots, l_r$   
(optimal expected search cost)

# Nearly-Optimal Mergesort

## Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs

J. Ian Munro

University of Waterloo, Canada

[jmunro@uwaterloo.ca](mailto:jmunro@uwaterloo.ca)

<https://orcid.org/0000-0002-7165-7988>

Sebastian Wild

University of Waterloo, Canada

[wild@uwaterloo.ca](mailto:wild@uwaterloo.ca)

<https://orcid.org/0000-0002-6061-9177>

### Abstract

We present two stable mergesort variants, “peeksort” and “powersort”, that exploit existing runs and find nearly-optimal merging orders with negligible overhead. Previous methods either require substantial effort for determining the merging order (Takaoka 2009; Burdaj & Némethy 2013) or do not have an optimal worst-case guarantee (Peterson 2002; Anger, Nicolai & Probst 2015; Bae & Knap 2018). We demonstrate that our methods are competitive in terms of running time with state-of-the-art implementations of stable sorting methods.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases adaptive sorting, nearly-optimal binary search tree, TimSort

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.63

Related Version arXiv: 1805.04154 (extended version with appendix)

Supplement Material zenodo: 1241102 (code to reproduce running time study)

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs Programme.

### 1 Introduction

Sorting is a fundamental building block for numerous tasks and ubiquitous in both the theory and practice of computing. While practical and theoretically (close-to) optimal comparison-based sorting methods are known, *instance-optimal sorting*, i.e. methods that adapt to the actual input and exploit specific structural properties if present, is still an area of active research. We survey some recent developments in Section 1.1.

Many different structural properties have been investigated in theory. Two of them have also found wide adoption in practice, e.g., in Oracle’s Java runtime library: adapting to the presence of duplicate keys and using existing sorted segments, called *runs*. The former is achieved by a so-called *fast-pivot partitioning* variant of quicksort [8], which is also used in the OpenBSD implementation of *qsort* from the C standard library. It is an unstable sorting method, though, i.e., the relative order of elements with equal keys might be destroyed in the process. It is hence used in Java solely for primitive-type arrays.

© J. Ian Munro and Sebastian Wild.

Revised under Creative Commons License CC BY

2018 Annual European Symposium on Algorithms (ESA 2018).

Editors: Sándor Aladi, Henrik Eick, and Giuseppe Ottaviano, Article No. 63, pp. 63:1–63:13

LIPIcs International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

► In 2018, with Ian Munro, I combined research on nearly-optimal BSTs with mergesort

↪ 2 new algorithms: *Peeksort* and *Powersort*

- both adapt provably optimal to existing runs even in worst case:  
 $\text{mergcost} \leq \mathcal{H}(\ell_1, \dots, \ell_r) + 2n$
- both are lightweight extensions of existing methods with negligible overhead
- both fast in practice

# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary




↪ split there and recurse  
(instead of at midpoint)



# Peeksort

- ▶ based on top-down mergesort

- ▶ “peek” at middle of array & find closest run boundary 


↪ split there and recurse  
(instead of at midpoint)



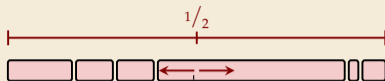


# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary 

↪ split there and recurse  
(instead of at midpoint)



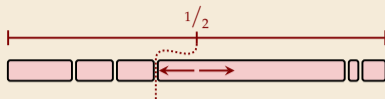
# Peeksort

▶ based on top-down mergesort


▶ “peek” at middle of array  
& find closest run boundary

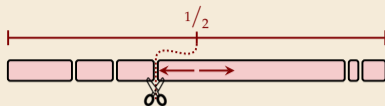


↪ split there and recurse  
(instead of at midpoint)



# Peeksort

- ▶ based on top-down mergesort
- ▶ “peek” at middle of array & find closest run boundary 
- ↪ split there and recurse  
(instead of at midpoint)



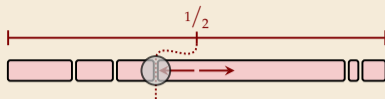
# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary




↪ split there and recurse  
(instead of at midpoint)

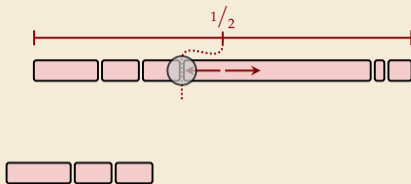


# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary 

↪ split there and recurse  
(instead of at midpoint)



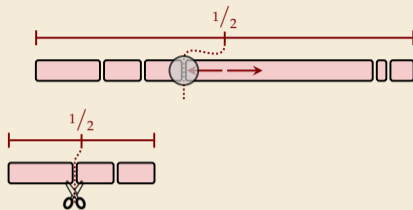
# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary




↪ split there and recurse  
(instead of at midpoint)

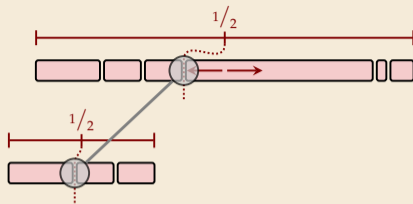


# Peeksort

▶ based on top-down mergesort


▶ “peek” at middle of array  
& find closest run boundary 

↪ split there and recurse  
(instead of at midpoint)

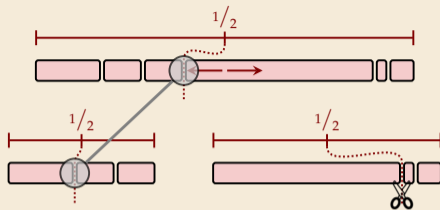


# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array & find closest run boundary 

↪ split there and recurse  
(instead of at midpoint)





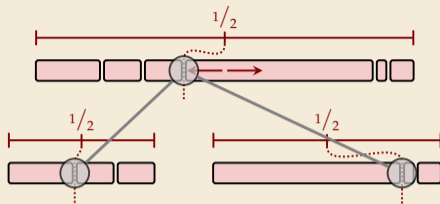
# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary



↪ split there and recurse  
(instead of at midpoint)



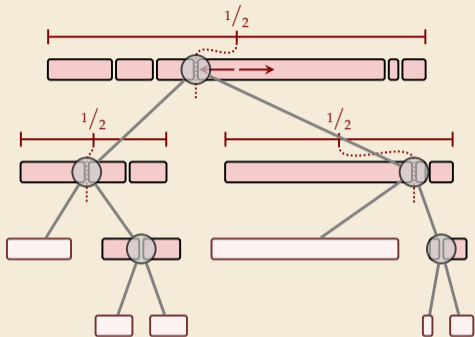
# Peeksort

▶ based on top-down mergesort

▶ “peek” at middle of array  
& find closest run boundary




↪ split there and recurse  
(instead of at midpoint)



# Peeksort

- ▶ based on top-down mergesort

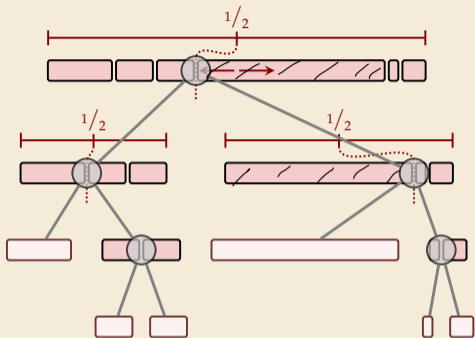
- ▶ “peek” at middle of array & find closest run boundary 

↪ split there and recurse  
(instead of at midpoint)

- ▶ can avoid scanning runs repeatedly:
  - ▶ find full run straddling midpoint
  - ▶ remember length of known runs at boundaries



↪ with clever recursion, scan each run only once.



# Peeksort – Code

```
1 procedure peeksort( $A[\ell..r]$ ,  $\Delta_\ell$ ,  $\Delta_r$ )
2   if  $r - \ell \leq 1$  then return
3   if  $\ell + \Delta_\ell == r \vee \ell == r + \Delta_r$  then return
4    $m := \ell + \lfloor (r - \ell) / 2 \rfloor$ 
5    $i := \begin{cases} \ell + \Delta_\ell & \text{if } \ell + \Delta_\ell \geq m \\ \text{extendRunLeft}(A, m) & \text{else} \end{cases}$ 
6    $j := \begin{cases} r + \Delta_r \leq m & \text{if } r + \Delta_r \leq m \leq m \\ \text{extendRunRight}(A, m) & \text{else} \end{cases}$ 
7    $g := \begin{cases} i & \text{if } m - i < j - m \\ j & \text{else} \end{cases}$ 
8    $\Delta_g := \begin{cases} j - i & \text{if } m - i < j - m \\ i - j & \text{else} \end{cases}$ 
9   peeksort( $A[\ell..g]$ ,  $\Delta_\ell$ ,  $\Delta_g$ )
10  peeksort( $A[g..r]$ ,  $\Delta_g$ ,  $\Delta_r$ )
11  merge( $A[\ell, g]$ ,  $A[g..r]$ , buf)
12  copy buf to  $A[\ell..r]$ 
```

► Parameters:



► initial call:

peeksort( $A[0..n]$ ,  $\Delta_0$ ,  $\Delta_n$ ) with  
 $\Delta_0 = \text{extendRunRight}(A, 0)$   
 $\Delta_n = n - \text{extendRunLeft}(A, n)$

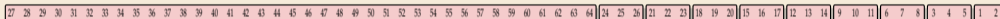
► helper procedure

```
1 procedure extendRunRight( $A[0..n]$ ,  $i$ )
2    $j := i + 1$ 
3   while  $j < n \wedge A[j - 1] \leq A[j]$ 
4      $j := j + 1$ 
5   return  $j$ 
```

(extendRunLeft similar)

## Peeksort – Analysis

- ▶ Consider tricky input from before again:



**144.5**  $\mathcal{H}(38, 3, 3, 3, 3, 3, 3, 3, 2)$



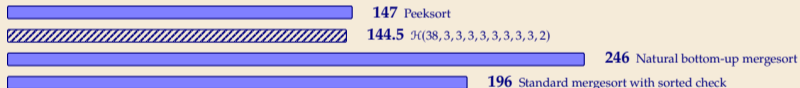
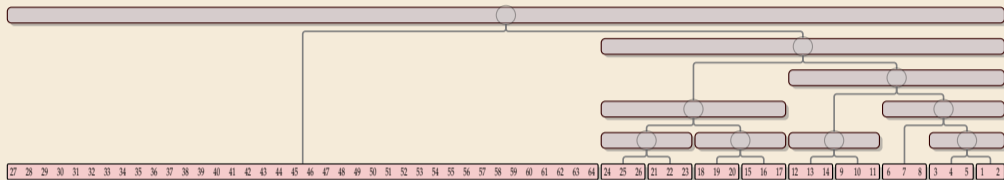
**246** Natural bottom-up mergesort



**196** Standard mergesort with sorted check

# Peeksort – Analysis

- ▶ Consider tricky input from before again:



- ▶ One can prove: Mergecost always  $\leq \mathcal{H}(\ell_1, \dots, \ell_r) + 2n$

↪ We can have the best of both worlds!

## 3.6 Python's list sort

# Sorting in Python

- ▶ *CPython*
  - ▶ *Python* is only a specification of a programming language
  - ▶ The Python Foundation maintains *CPython* as the official reference implementation of the Python programming language
  - ▶ If you don't specifically install something else, python will be CPython
- ▶ part of Python are `list.sort` resp. `sorted` built-in functions
  - ▶ implemented in C
  - ▶ use *Timsort*, custom Mergesort variant by Tim Peters



# Sorting in Python

- ▶ *CPython*
  - ▶ *Python* is only a specification of a programming language
  - ▶ The Python Foundation maintains *CPython* as the official reference implementation of the Python programming language
  - ▶ If you don't specifically install something else, python will be CPython
- ▶ part of Python are `list.sort` resp. `sorted` built-in functions
  - ▶ implemented in C
  - ▶ use *Timsort*, custom Mergesort variant by Tim Peters



Sept 2021: **Python uses Powersort!**  
in CPython 3.11 and PyPy 7.3.6

**msg400864** - (view) Author: **Tim Peters (tim.peters)** • 🌐 Date: 2021-09-01 19:43

I created a PR that implements the powersort merge strategy:  
<https://github.com/python/cpython/pull/28108>

Across all the time this issue report has been open, that strategy continues to be the top contender. Enough already ;-) It's indeed a more difficult change to make to the code, but that's in relative terms. In absolute terms, it's not at all a hard change.

Laurent, if you find that some variant of ShiversSort actually runs faster than that, let us know here! I'm a big fan of Vincent's innovations too, but powersort seems to do somewhat better "on average" than even his length-adaptive ShiversSort (and implementing that too would require changing code outside of `merge_collapse()`).

# Timsort (original version)

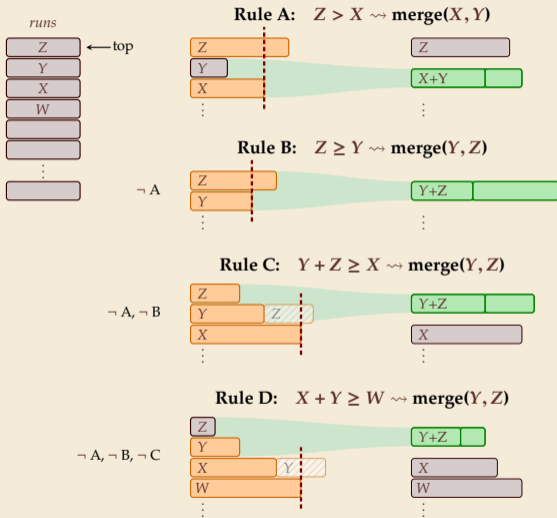
```
1 procedure Timsort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3   while  $i < n$ 
4      $j := \text{ExtendRunRight}(A, i)$ 
5      $runs.push(i, j)$ ;  $i := j$ 
6     while rule A/B/C/D applicable
7       merge corresponding runs
8   while  $runs.size() \geq 2$ 
9     merge topmost 2 runs
```

- ▶ above shows the core algorithm;  
many more algorithm engineering tricks

- ▶ **Advantages:**

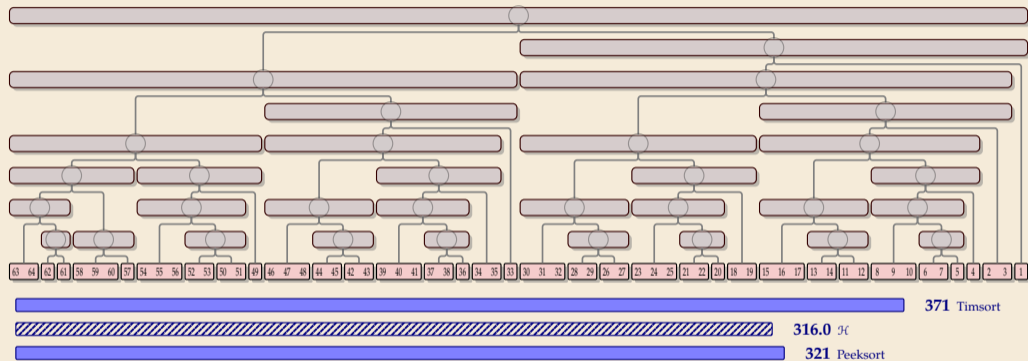
- ▶ profits from existing runs
- ▶ *locality of reference* for merges

- ▶ **But:** *not* optimally adaptive! (next slide)  
Reason: Rules A–D (Why exactly these?!)



## Timsort bad case

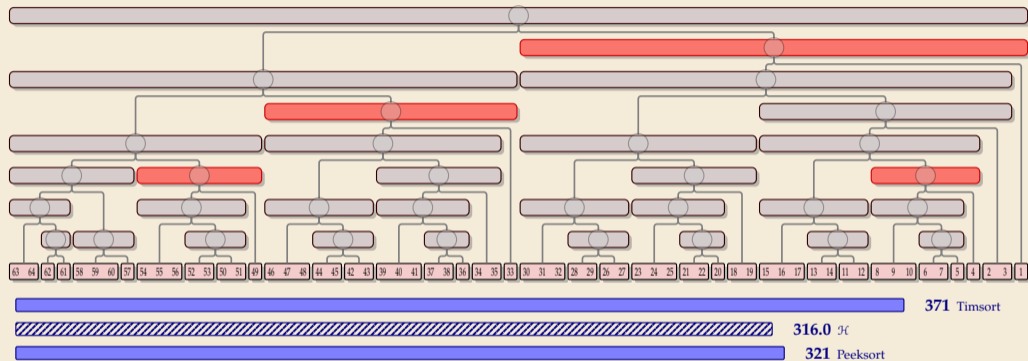
- ▶ On certain inputs, Timsort's merge rules don't work well:



- ▶ As  $n$  increases, Timsort's cost approach  $1.5 \cdot \mathcal{H}$ , i. e., 50% more merge costs than necessary

## Timsort bad case

- ▶ On certain inputs, Timsort's merge rules don't work well:



- ▶ As  $n$  increases, Timsort's cost approach  $1.5 \cdot \mathcal{H}$ , i. e., 50% more merge costs than necessary
  - ▶ intuitive problem: regularly very unbalanced merges

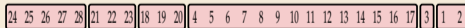
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



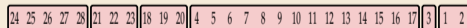
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



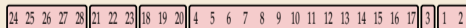
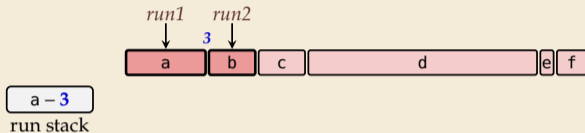
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



# Powersort

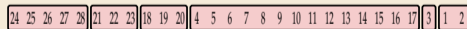
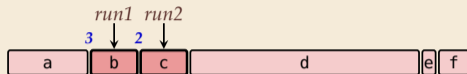
↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

a-3  
run stack





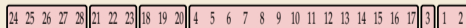
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



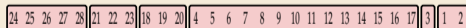
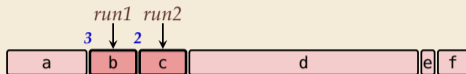
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



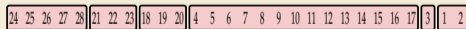
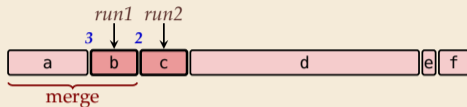
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



# Powersort

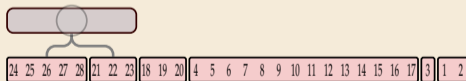
↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

ab - 2  
run stack



# Powersort

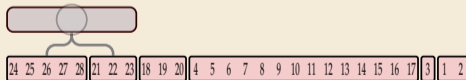
↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

ab - 2  
run stack



# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



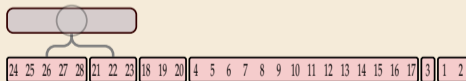
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



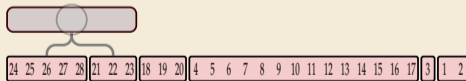
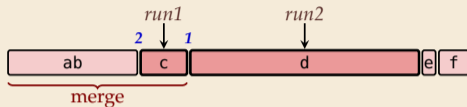
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---





# Powersort

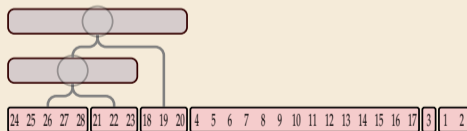
↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

abc - 1  
run stack



# Powersort

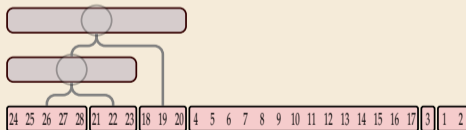
↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

abc - 1  
run stack



# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

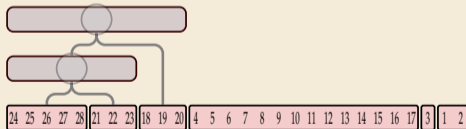
---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

d - 2
abc - 1

run stack



# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

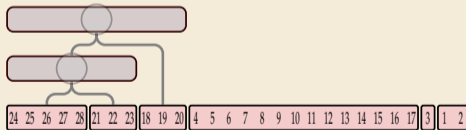
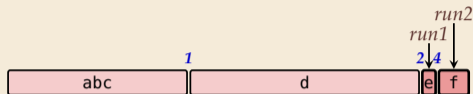
---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

d - 2
abc - 1

run stack



# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

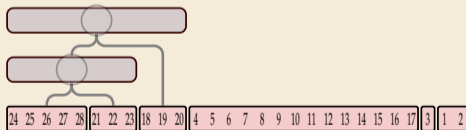
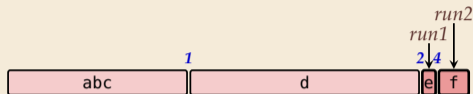
---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

e - 4
d - 2
abc - 1

run stack



# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

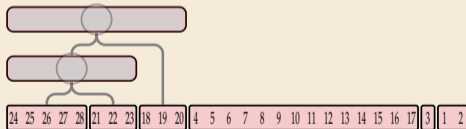
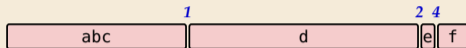
---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---

e - 4
d - 2
abc - 1

run stack



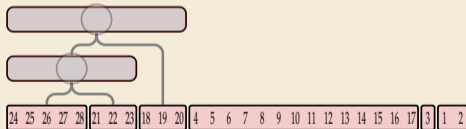
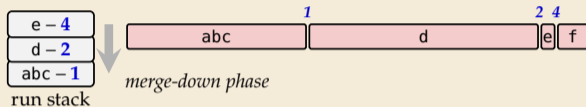
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



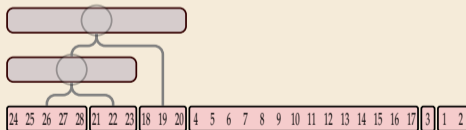
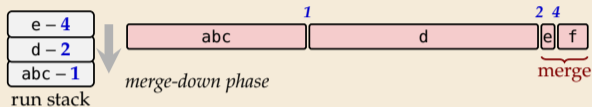
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---





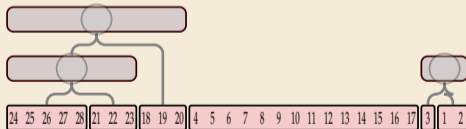
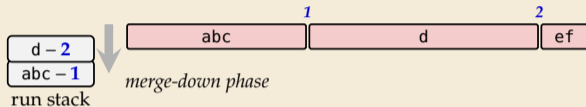
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



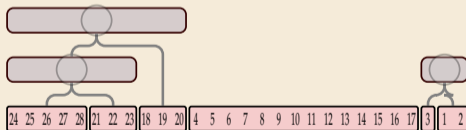
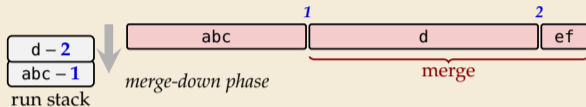
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



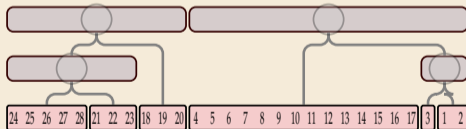
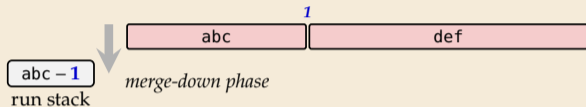
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



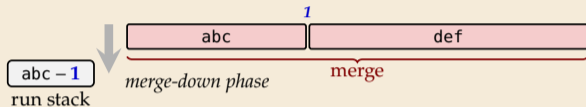
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



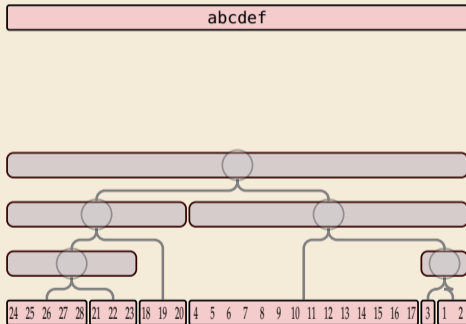
# Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?

---

```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```

---



# Powersort – Computing powers

► Computing the power of (the node between) two runs  $A[i_1..j_1]$  and  $A[i_2..j_2]$

►  $\left[ \leftarrow \rightarrow \right]$  = normalized midpoint interval

► power =  $\min \ell$  s.t.  $\left[ \leftarrow \rightarrow \right]$  contains  $c \cdot 2^{-\ell}$



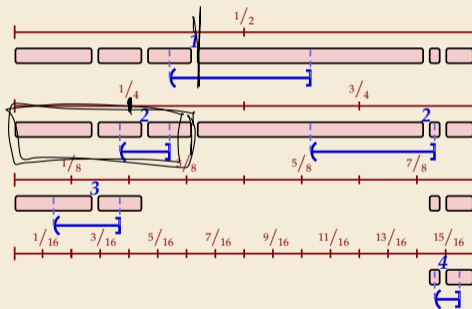

---

```

1 procedure power((i1, j1), (i2, j2), n)
2   n1 := j1 - i1 + 1
3   n2 := j2 - i2 + 1
4   a := (i1 + 1/2 * n1 - 1) / n
5   b := (i2 + 1/2 * n2 - 1) / n // interval (a, b)
6   l := 0
7   while [a * 2^l] == [b * 2^l]
8     l := l + 1
9   return l

```


---



# Powersort – Computing powers

- ▶ Computing the power of (the node between) two runs  $A[i_1..j_1]$  and  $A[i_2..j_2]$

- ▶  $\left[ \text{---} \right]$  = normalized midpoint interval

- ▶ power =  $\min \ell$  s.t.  $\left[ \text{---} \right]$  contains  $c \cdot 2^{-\ell}$  

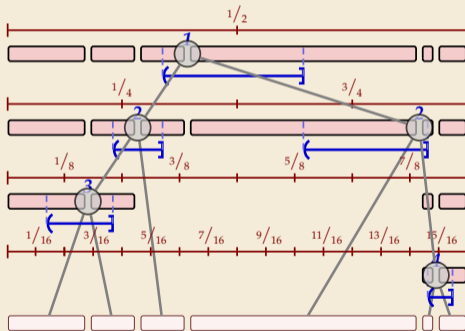
---

```

1 procedure power( $(i_1, j_1), (i_2, j_2), n$ )
2    $n_1 := j_1 - i_1 + 1$ 
3    $n_2 := j_2 - i_2 + 1$ 
4    $a := \frac{i_1 + \frac{1}{2}n_1 - 1}{n}$ 
5    $b := \frac{i_2 + \frac{1}{2}n_2 - 1}{n}$  // interval  $(a, b)$ 
6    $\ell := 0$ 
7   while  $\lfloor a \cdot 2^\ell \rfloor \neq \lfloor b \cdot 2^\ell \rfloor$ 
8      $\ell := \ell + 1$ 
9   return  $\ell$ 

```


---



# Powersort – Computing powers

- ▶ Computing the power of (the node between) two runs  $A[i_1..j_1]$  and  $A[i_2..j_2]$

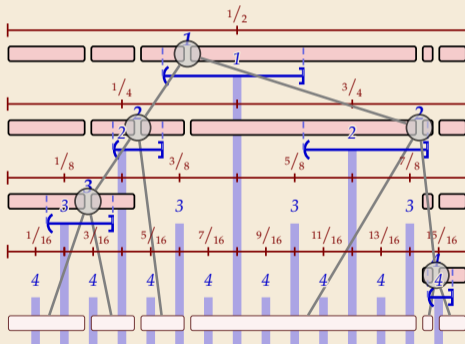
- ▶  $\left[ \begin{array}{|} \hline \leftarrow \rightarrow \\ \hline \end{array} \right]$  = normalized midpoint interval

- ▶ power =  $\min \ell$  s.t.  $\left[ \begin{array}{|} \hline \leftarrow \rightarrow \\ \hline \end{array} \right]$  contains  $c \cdot 2^{-\ell}$  

---

```
1 procedure power( $(i_1, j_1), (i_2, j_2), n$ )
2    $n_1 := j_1 - i_1 + 1$ 
3    $n_2 := j_2 - i_2 + 1$ 
4    $a := \frac{i_1 + \frac{1}{2}n_1 - 1}{n}$ 
5    $b := \frac{i_2 + \frac{1}{2}n_2 - 1}{n}$  // interval  $(a, b)$ 
6    $\ell := 0$ 
7   while  $\lfloor a \cdot 2^\ell \rfloor \neq \lfloor b \cdot 2^\ell \rfloor$ 
8      $\ell := \ell + 1$ 
9   return  $\ell$ 
```

---





## Powersort – Discussion



Retains all advantages of Timsort

- ▶ good locality in memory accesses
- ▶ no recursion
- ▶ all the tricks in Timsort



optimally adapts to existing runs



minimal overhead for finding merge order

# Part III

*Divide & Conquer beyond sorting*

# Divide and conquer

**Divide and conquer** *idiom* (Latin: *divide et impera*)

to make a group of people disagree and fight with one another  
so that they will not join together against one

(Merriam-Webster Dictionary)

↪ in politics as in algorithms, many independent, small problems are better than a big one!

## Divide-and-conquer algorithms:

1. Break problem into smaller, independent subproblems. (Divide!)
2. Recursively solve all subproblems. (Conquer!)
3. Assemble solution for original problem from solutions for subproblems.

## Examples:

- ▶ Mergesort
- ▶ Quicksort
- ▶ Binary search
- ▶ (arguably) Tower of Hanoi

## 3.7 Order Statistics

# Selection by Rank

- ▶ Standard data summary of numerical data: (Data scientists, listen up!)

- ▶ mean, standard deviation

- ▶ min/max (range)

- ▶ histograms

- ▶ median, quartiles, other quantiles  
(a.k.a. order statistics)

} easy to compute in  $\Theta(n)$  time



computable in  $\Theta(n)$  time?

# Selection by Rank

- ▶ Standard data summary of numerical data: (Data scientists, listen up!)

- ▶ mean, standard deviation
  - ▶ min/max (range)
  - ▶ histograms
- } easy to compute in  $\Theta(n)$  time

- ▶ median, quartiles, other quantiles  
(a.k.a. order statistics)



computable in  $\Theta(n)$  time?

## General form of problem: Selection by Rank

- ▶ **Given:** array  $A[0..n)$  of numbers and number  $k \in [0..n)$ .
  - ▶ **Goal:** find element that would be in position  $k$  if  $A$  was sorted ( $k$ th smallest element).
- but 0-based & counting dups
- ▶  $k = \lfloor n/2 \rfloor \rightsquigarrow$  median;     $k = \lfloor n/4 \rfloor \rightsquigarrow$  lower quartile  
 $k = 0 \rightsquigarrow$  minimum;     $k = n - \ell \rightsquigarrow \ell$ th largest

# Quickselect

- ▶ Key observation: Finding the element of rank  $k$  seems hard.  
But computing the rank of a given element is easy!

↪ Pick any element  $A[b]$  and find its rank  $j$ .

count smaller elements

- ▶  $j = k?$  ↪ Lucky Duck! Return chosen element and stop
- ▶  $j < k?$  ↪ ... not done yet. But: The  $j + 1$  elements smaller than  $\leq A[b]$  can be excluded!
- ▶  $j > k?$  ↪ similarly exclude the  $n - j$  elements  $\geq A[b]$

# Quickselect

- ▶ Key observation: Finding the element of rank  $k$  seems hard.  
But computing the rank of a given element is easy!

↪ Pick any element  $A[b]$  and find its rank  $j$ .

↖ count smaller elements

- ▶  $j = k$ ? ↪ Lucky Duck! Return chosen element and stop
- ▶  $j < k$ ? ↪ ... not done yet. But: The  $j + 1$  elements smaller than  $\leq A[b]$  can be excluded!
- ▶  $j > k$ ? ↪ similarly exclude the  $n - j$  elements  $\geq A[b]$

▶ partition function from Quicksort:

- ▶ returns the rank of pivot
- ▶ separates elements into smaller/larger

↪ can use same building blocks

(recursion can be replaced by loop)


---

```
1 procedure quickselect( $A[l..r]$ ,  $k$ )
2   if  $r - l \leq 1$  then return  $A[l]$ 
3    $b :=$  choosePivot( $A[l..r]$ )
4    $j :=$  partition( $A[l..r]$ ,  $b$ )
5   if  $j == k$ 
6     return  $A[j]$ 
7   else if  $j < k$ 
8     quickselect( $A[j + 1..n]$ ,  $k - j - 1$ )
9   else //  $j > k$ 
10    quickselect( $A[0..j]$ ,  $k$ )
```


---




## Quickselect Discussion

  $\Theta(n^2)$  worst case (like Quicksort)

 can prove: expected cost  $\Theta(n)$

 no extra space needed

 adaptations possible to find several order statistics



For practical purposes, Quickselect is fine.



Yeah . . . maybe. But can we select by rank in  $O(n)$  worst case?

## Better Pivots

It turns out, we can!

- ▶ All we need is better pivots!
  - ▶ If pivot was the exact median, we would at least halve #elements in each step
  - ▶ Then the total cost of all partitioning steps is  $\leq 2n = \Theta(n)$ .

## Better Pivots

It turns out, we can!

- ▶ All we need is better pivots!
  - ▶ If pivot was the exact median, we would at least halve #elements in each step
  - ▶ Then the total cost of all partitioning steps is  $\leq 2n = \Theta(n)$ .



But: finding medians is (basically) our original problem!

# Better Pivots

It turns out, we can!

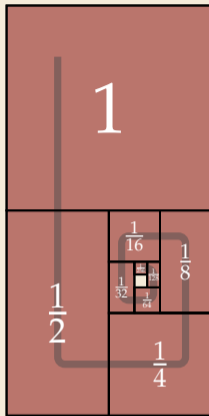
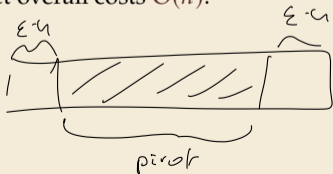
- ▶ All we need is better pivots!
  - ▶ If pivot was the exact median, we would at least halve #elements in each step
  - ▶ Then the total cost of all partitioning steps is  $\leq 2n = \Theta(n)$ .



But: finding medians is (basically) our original problem!



It totally suffices to find an element of rank  $\alpha n$  for  $\alpha \in (\epsilon, 1 - \epsilon)$  to get overall costs  $\Theta(n)$ !

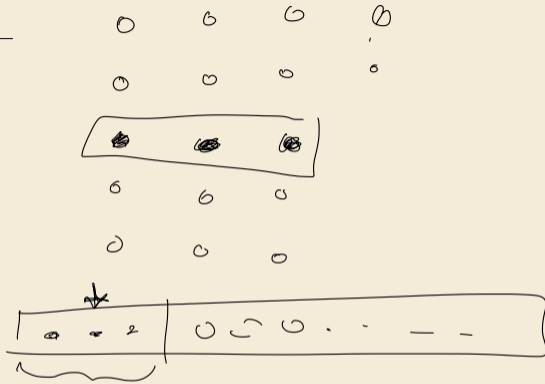


$$\sum_{i=0}^n (\alpha)^i < \sum_{i=0}^{\infty} (\alpha)^i = \frac{1}{1-\alpha}$$

$\alpha = \frac{1}{2}$

# The Median-of-Medians Algorithm

```
1 procedure choosePivotMoM(A[l..r])
2    $m := \lfloor n/5 \rfloor$ 
3   for  $i := 0, \dots, m - 1$ 
4     sort(A[5i..5i + 4])
5     // collect median of 5
6     Swap A[i] and A[5i + 2]
7   return quickselectMoM(A[0..m],  $\lfloor \frac{m-1}{2} \rfloor$ )
8
9 procedure quickselectMoM(A[l..r], k)
10  if  $r - l \leq 1$  then return A[l]
11   $b :=$  choosePivotMoM(A[l..r])
12   $j :=$  partition(A[l..r], b)
13  if  $j == k$ 
14    return A[j]
15  else if  $j < k$ 
16    quickselectMoM(A[j + 1..r], k - j - 1)
17  else //  $j > k$ 
18    quickselectMoM(A[l..j], k)
```



# The Median-of-Medians Algorithm

---

```
1 procedure choosePivotMoM(A[l..r])
2    $m := \lfloor n/5 \rfloor$ 
3   for  $i := 0, \dots, m - 1$ 
4     sort(A[5i..5i + 4])
5     // collect median of 5
6     Swap A[i] and A[5i + 2]
7   return quickselectMoM(A[0..m],  $\lfloor \frac{m-1}{2} \rfloor$ )
8
9 procedure quickselectMoM(A[l..r], k)
10  if  $r - \ell \leq 1$  then return A[l]
11   $b :=$  choosePivotMoM(A[l..r])
12   $j :=$  partition(A[l..r], b)
13  if  $j == k$ 
14    return A[j]
15  else if  $j < k$ 
16    quickselectMoM(A[j + 1..n], k - j - 1)
17  else //  $j > k$ 
18    quickselectMoM(A[0..j], k)
```

---

## Analysis:

- Note: 2 mutually recursive procedures  
     $\rightsquigarrow$  effectively 2 recursive calls!
- 1. recursive call inside choosePivotMoM  
    on  $m \leq \frac{n}{5}$  elements

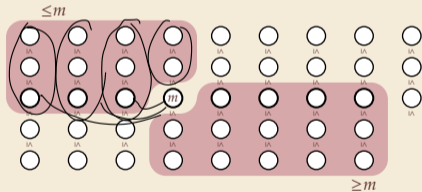
# The Median-of-Medians Algorithm

```
1 procedure choosePivotMoM(A[l..r])
2    $m := \lfloor n/5 \rfloor$ 
3   for  $i := 0, \dots, m - 1$ 
4     sort(A[5i..5i + 4])
5     // collect median of 5
6     Swap A[i] and A[5i + 2]
7   return quickselectMoM(A[0..m],  $\lfloor \frac{m-1}{2} \rfloor$ )
8
9 procedure quickselectMoM(A[l..r], k)
10  if  $r - l \leq 1$  then return A[l]
11   $b :=$  choosePivotMoM(A[l..r])
12   $j :=$  partition(A[l..r], b)
13  if  $j == k$ 
14    return A[j]
15  else if  $j < k$ 
16    quickselectMoM(A[j + 1..r], k - j - 1)
17  else //  $j > k$ 
18    quickselectMoM(A[l..j], k)
```

## Analysis:

► Note: 2 mutually recursive procedures  
↪ effectively 2 recursive calls!

1. recursive call inside choosePivotMoM on  $m \leq \frac{n}{5}$  elements
2. recursive call inside quickselectMoM



↪ partition excludes  $\sim 3 \cdot \frac{m}{2} \sim \frac{3}{10}n$  elem.

# The Median-of-Medians Algorithm

```

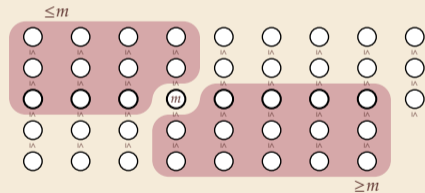
1 procedure choosePivotMoM(A[l..r])
2   m := ⌊n/5⌋
3   for i := 0, ..., m - 1
4     sort(A[5i..5i + 4])
5     // collect median of 5
6     Swap A[i] and A[5i + 2]
7   return quickselectMoM(A[0..m], ⌊ $\frac{m-1}{2}$ ⌋)
8
9 procedure quickselectMoM(A[l..r], k)
10  if r - l ≤ 1 then return A[l]
11  b := choosePivotMoM(A[l..r])
12  j := partition(A[l..r], b)
13  if j == k
14    return A[j]
15  else if j < k
16    quickselectMoM(A[j + 1..r], k - j - 1)
17  else // j > k
18    quickselectMoM(A[l..j], k)

```

## Analysis:

► Note: 2 mutually recursive procedures  
 $\rightsquigarrow$  effectively 2 recursive calls!

1. recursive call inside choosePivotMoM on  $m \leq \frac{n}{5}$  elements
2. recursive call inside quickselectMoM



$\rightsquigarrow$  partition excludes  $\sim 3 \cdot \frac{m}{2} \sim \frac{3}{10}n$  elem.

$$\begin{aligned}
 \rightsquigarrow C(n) &\leq \Theta(n) + C\left(\frac{1}{5}n\right) + C\left(\frac{7}{10}n\right) \\
 &\leq \Theta(n) + C\left(\frac{1}{5}n + \frac{7}{10}n\right) \\
 &= \Theta(n) + C\left(\frac{9}{10}n\right) \rightsquigarrow C(n) = \Theta(n)
 \end{aligned}$$

ansatz: overall cost linear



## 3.8 Further D&C Algorithms

# Majority

- ▶ **Given:** Array  $A[0..n)$  of objects
- ▶ **Goal:** Check if there is an object  $x$  that occurs at  $> \frac{n}{2}$  positions in  $A$   
if so, return  $x$
- ▶ Naive solution: check each  $A[i]$  whether it is a majority  $\rightsquigarrow \Theta(n^2)$  time

# Majority

- ▶ **Given:** Array  $A[0..n)$  of objects
- ▶ **Goal:** Check if there is an object  $x$  that occurs at  $> \frac{n}{2}$  positions in  $A$   
if so, return  $x$
- ▶ Naive solution: check each  $A[i]$  whether it is a majority  $\rightsquigarrow \Theta(n^2)$  time

Can be solved faster using a simple Divide & Conquer approach:

- ▶ If  $A$  has a majority, that element must also be a majority of at least one half of  $A$ .  
 $\rightsquigarrow$  Can find majority (if it exists) of left half and right half recursively  
 $\rightsquigarrow$  Check these  $\leq 2$  candidates.
- ▶ Costs similar to mergesort  $\Theta(n \log n)$

$$C(n) \approx 2 \cdot C\left(\frac{n}{2}\right) + \Theta(n)$$

# Majority – Linear Time

We can actually do much better!

---

```
1 def MJRTY(A[0..n])
2   c := 0
3   for i := 1, ..., n - 1
4     if c == 0
5       x := A[i]; c := 1
6     else
7       if A[i] == x then c := c + 1 else c := c - 1
8   return x
```

---



- ▶  $\text{MJRTY}(A[0..n])$  returns *candidate* majority element
- ▶ either that candidate is the majority element or none exists(!)

👍 Clearly  $\Theta(n)$  time

## Closest pair

- ▶ **Given:** Array  $P[0..n)$  of points in the plane  
each has  $x$  and  $y$  coordinates:  $P[i].x$  and  $P[i].y$
- ▶ **Goal:** Find pair  $P[i], P[j]$  that is closest in (Euclidean) distance
- ▶ Naive solution: compute distance of each pair  $\rightsquigarrow \Theta(n^2)$  time
- ▶ Can be done in  $O(n \log n)$  time using a clever divide & conquer algorithm.  
(Details not part of the module material.)