4 Efficient Sorting

3 November 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

version `2025-09-28 18:05 H`

Learning Outcomes

Unit 4: ***Efficient Sorting***

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.

Outline

4 Efficient Sorting

Why study sorting?

- fundamental problem of computer science that is still not solved

  Algorithm with optimal #comparisons in worst case?
- building brick of many more advanced algorithms
  - for preprocessing
  - as subroutine
- playground of manageable complexity
  to practice algorithmic techniques

Here:

- "classic" fast sorting method
- exploit **partially sorted** inputs
- **parallel** sorting

Part I

The Basics

Rules of the game

- **Given:**
  - array $A[0..n) = A[0..n-1]$ of n objects
  - a total order relation $\le$ among $A[0], \ldots, A[n-1]$

    (a comparison function)

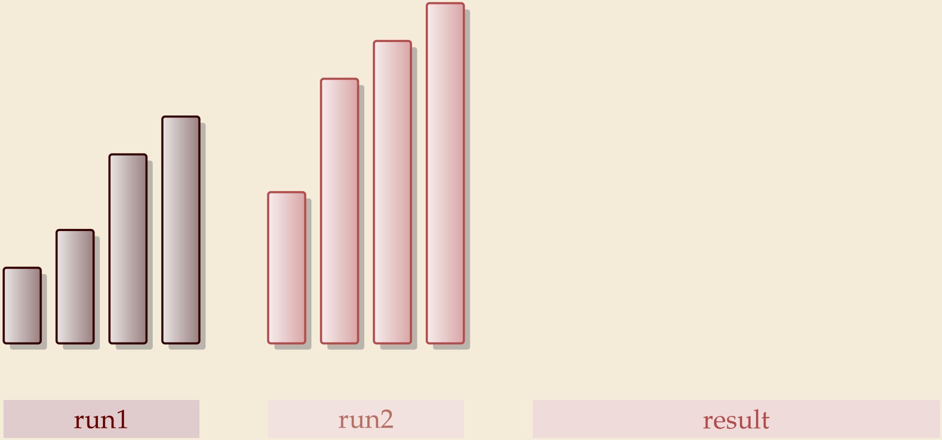
    Python: elements support <= operator (`__le__()`)

    Java: `Comparable` class (`x.compareTo(y) <= 0`)
- **Goal:** rearrange (i. e., permute) elements within A,
  so that A is *sorted*, i. e., $A[0] \le A[1] \le \cdots \le A[n-1]$

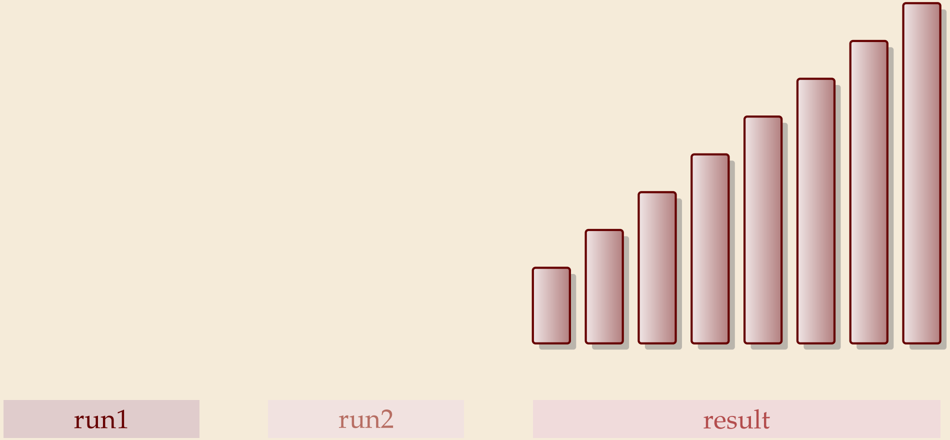
- for now: A stored in main memory (*internal sorting*)
  single processor (*sequential sorting*)

4.1 Mergesort

Merging sorted lists



Merging sorted lists



Mergesort

```
procedure mergesort(A[l..r)):
    n := r − l
    if n ≤ 1 return
    m := l + ⌊n/2⌋
    mergesort(A[l..m))
    mergesort(A[m..r))
    merge(A[l..m), A[m..r), buf)
    copy buf to A[l..r)
```

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

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

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

same for best and worst case!

(precisely(!) solvable *without* assumption $n = 2^k$:
$C(n) = 2n\lg(n) + \bigl(2 - \{\lg(n)\} - 2^{1-\{\lg(n)\}}\bigr)2n$
with $\{x\} := x - \lfloor x \rfloor$)

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

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

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

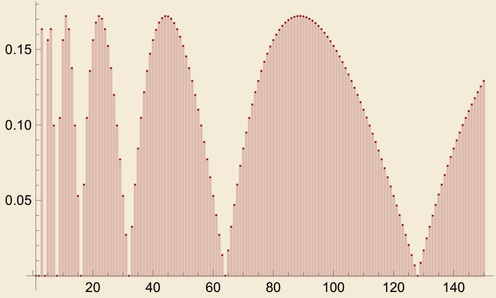
(arbitrary n: $C(n) \le C(\text{next larger power of 2}) \le 4n\lg(n) + 2n = \Theta(n \log n)$)

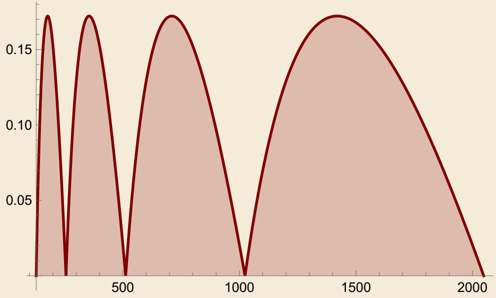
Linear Term of $C(n)$

Recall:

$$C(n) \;=\; 2n\lg(n) + \left(2 - \{\lg(n)\} - 2^{1-\{\lg(n)\}}\right)2n \qquad \text{with } \{x\} := x - \lfloor x \rfloor$$

Plot of $2\left(2 - \{\lg(n)\} - 2^{1-\{\lg(n)\}}\right)$





$\rightsquigarrow$ Can prove: $C(n) \;\le\; 2n\lg n + 0.172n$

Mergesort – Discussion

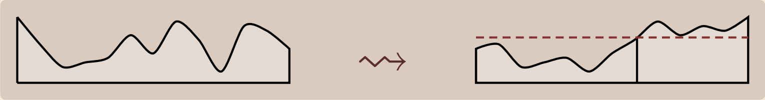
- 👍 optimal time complexity of $\Theta(n \log n)$ in the worst case
- 👍 *stable* sorting method   i. e., retains relative order of equal-key items
- 👍 memory access is sequential (scans over arrays)

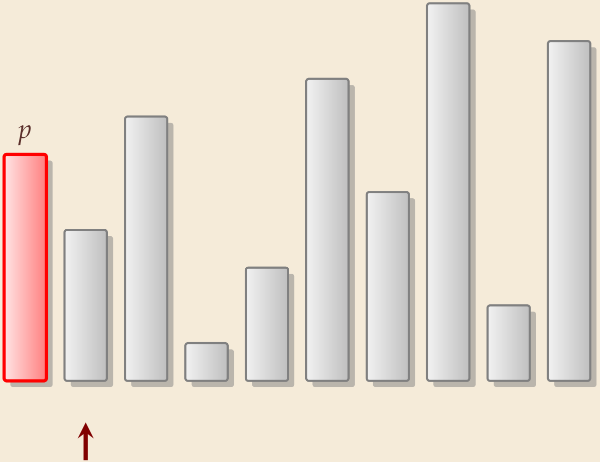
- 👎 requires $\Theta(n)$ extra space

  there are in-place merging methods,
  but they are substantially more complicated
  and not (widely) used

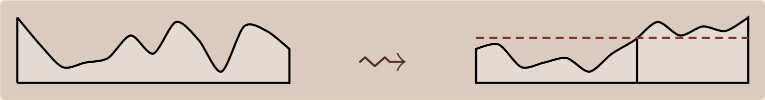
4.2 Quicksort

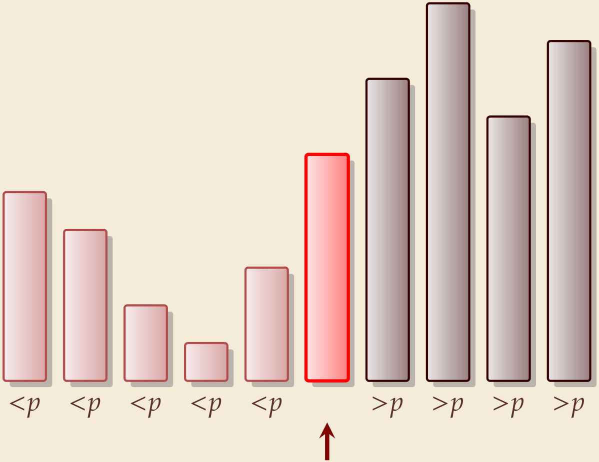
Partitioning around a pivot





Partitioning around a pivot





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

Partitioning – Detailed code

Beware: details easy to get wrong; use this code! (if you ever have to)

```
1  procedure partition(A, b):
2      // input: array A[0..n), position of pivot b ∈ [0..n)
3      swap(A[0], A[b])
4      i := 0,   j := n
5      while true do
6          do i := i + 1 while i < n and A[i] < A[0]
7          do j := j − 1 while j ≥ 1 and A[j] > A[0]
8          if i ≥ j then break    (goto 11)
9          else swap(A[i], A[j])
10     end while
11     swap(A[0], A[j])
12     return j
```

Loop invariant (5–10): A

p	$\le p$	?	$\ge p$
		i	j

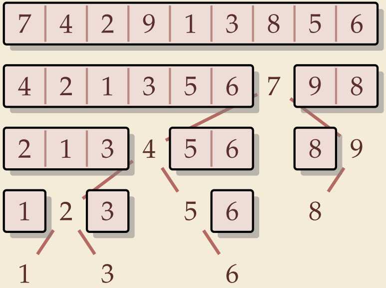
Quicksort

```
procedure quicksort(A[l..r]):
    if r − ℓ ≤ 1 then return
    b := choosePivot(A[l..r])
    j := partition(A[l..r], b)
    quicksort(A[l..j))
    quicksort(A[j + 1..r))
```

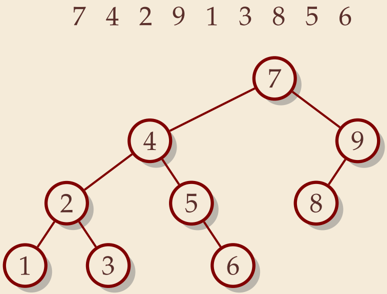
- recursive procedure
- choice of pivot can be
  - fixed position $\rightsquigarrow$ dangerous!
  - random
  - more sophisticated, e. g., median of 3

Quicksort & Binary Search Trees

Quicksort

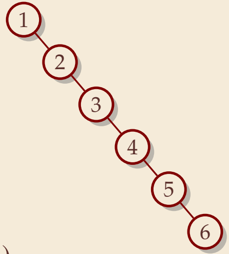


Binary Search Tree (BST)



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

Quicksort – Worst Case



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

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

⇝ 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

⇝ same as randomly *shuffling* input before sorting

Randomized Quicksort – Analysis

- cost measure: element visits (as for mergesort)
- $C(n)$ = #element visits when sorting n randomly permuted elements
  = cost of searching every element in BST build from input

$\rightsquigarrow$ quicksort needs $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ *in expectation*
(see analysis of C_n in Unit 3!)

- 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.

4.3 Comparison-Based Lower Bound

Lower Bounds

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

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

- 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 comparisons.

That's good!
Keeps algorithms general!

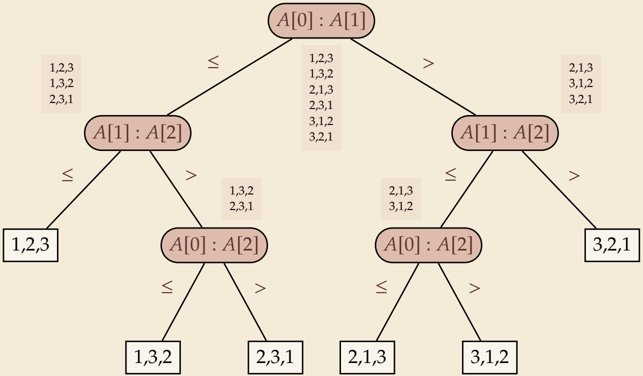
- This makes very few assumptions on the kind of objects we are sorting.
- Mergesort and Quicksort work in the comparison model.

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

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

Comparison Lower Bound

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



- Execution $=$ follow a path in comparison tree.

$\rightsquigarrow$ height of comparison tree $=$ worst-case # comparisons

- comparison trees are *binary* trees

$\rightsquigarrow$ ℓ leaves $\rightsquigarrow$ height $\ge \lceil \lg(\ell) \rceil$

- comparison trees for sorting method must have $\ge n!$ leaves

$\rightsquigarrow$ height $\ge \lg(n!) \sim n \lg n$

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

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

≈ 1.4427

4.4 Integer Sorting

How to beat a lower bound

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

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

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

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

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

Counting sort

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

word size

Counting sort

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

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

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

Larger Universes: Radix Sort

- *MSD Radix Sort:*
  - split numbers into base-R "digits"
  - Use counting sort on <u>m</u>ost <u>s</u>ignificant <u>d</u>igit
    (with variant of counting sort that moves full number)
  - ⇝ integers sorted with respect to first digit
  - recurse on sublist for each digit value, using next digit for counting sort

⇝ After $\lfloor \log_R(U) \rfloor + 1$ levels of counting sort, fully sorted!

- For $R \le 2^w$, all counting sort calls on same level cost total of $O(n)$ time
  (requires care to avoid reinitialization cost of array C)

⇝ total time $O(n \log_R(U)) = O\left(n \frac{\log(U)}{\log(R)}\right)$

⇝ $O(n)$ time sorting possible for numbers in range $U = O(n^c)$ for constant c.

Integer Sorting – State of the art

Algorithm theory

- integer sorting on the w-bit word-RAM
- suppose $U = 2^w$, but w can be an arbitrary function of n
- how fast can we sort n such w-bit integers on a w-bit word-RAM?
  - for $w = O(\log n)$: linear time (*radix/counting sort*)
  - for $w = \Omega(\log^{2+\varepsilon} n)$: linear time (*signature sort*)
  - for w in between: can do $O(n\sqrt{\lg\lg n})$ (very complicated algorithm)
    don't know if that is best possible!

* * *

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

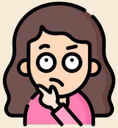
Part II

Exploiting presortedness

4.5 Adaptive Sorting

Adaptive sorting

- Comparison lower bound also holds for the *average case* $\rightsquigarrow$ $\lfloor \lg(n!) \rfloor$ 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 . . .

- 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

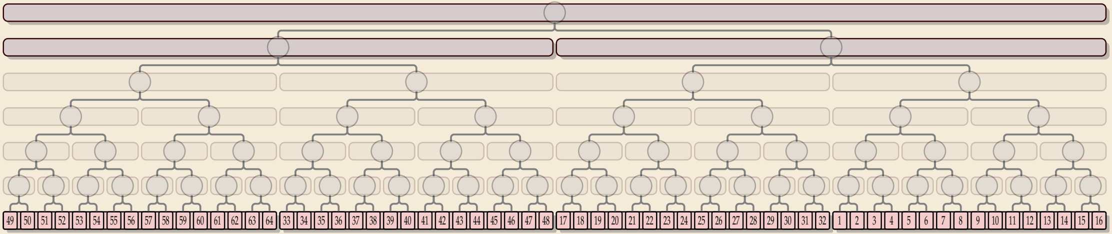
- If last element of first run $\le$ first element of second run, skip merge

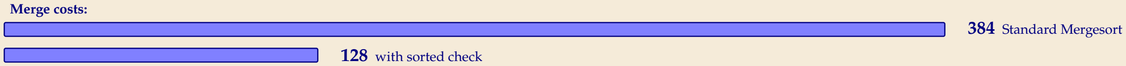
How effective is this idea?

```
procedure mergesortCheck(A[l..r)):
    n := r − l
    if n ≤ 1 return
    m := l + ⌊n/2⌋
    mergesortCheck(A[l..m))
    mergesortCheck(A[m..r))
    if A[m − 1] > A[m]
        merge(A[l..m), A[m..r), buf)
        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:





Sorted check can help a lot!

Alignment issues

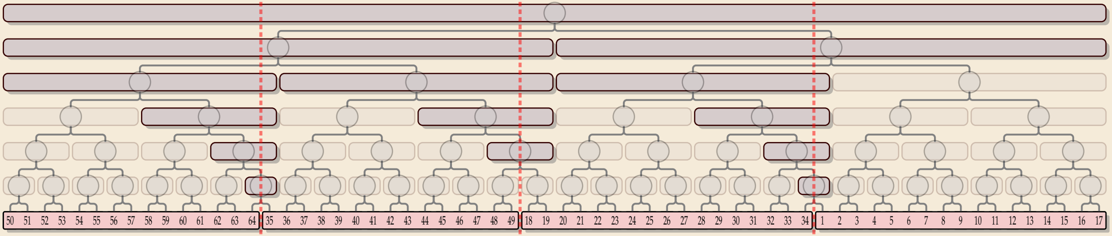
- ▶ In previous example, each run of length ℓ saved us $\ell \lg(\ell)$ in merge cost.
  - $=$ exactly the cost of *creating* this run in mergesort had it not already existed
  - $\rightsquigarrow$ best savings we can hope for!

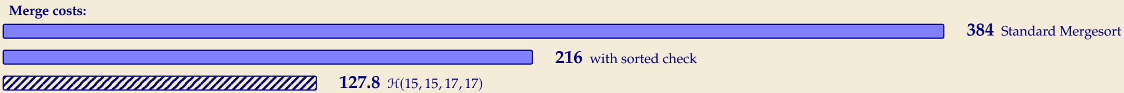
$\rightsquigarrow$ Are overall merge costs

ℓ_i = length of ith run

$$\mathcal{H}(\ell_1,\ldots,\ell_r) \;:=\; \underbrace{n\lg(n)}_{\text{mergesort}} \;-\; \underbrace{\sum_{i=1}^{r} \ell_i \lg(\ell_i)}_{\text{savings from runs}} \;?$$

Unfortunately, not quite:





Bottom-Up Mergesort

- Can we do better by explicitly detecting runs?

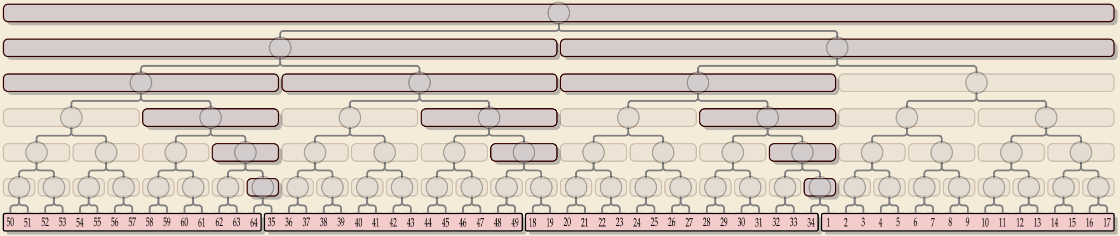
```
procedure bottomUpMergesort(A[0..n)):
    Q := new Queue // runs to merge
    // Phase 1: Enqueue singleton runs
    for i = 0, . . . , n − 1 do
        Q.enqueue((i, i + 1))
    // Phase 2: Merge runs level−wise
    while Q.size() ≥ 2
        Q′ := new Queue
        while Q.size() ≥ 2
            (i₁, j₁) := Q.dequeue()
            (i₂, j₂) := Q.dequeue()
            merge(A[i₁..j₁), A[i₂..j₂), buf)
            copy buf to A[i₁..j₂)
            Q′.enqueue((i₁, j₂))
        if ¬Q.isEmpty() // lonely run
            Q′.enqueue(Q.dequeue())
        Q := Q′
```

```
procedure naturalMergesort(A[0..n)):
    Q := new Queue;  i := 0
    while i < n
        j := i + 1
        while A[j] ≥ A[j − 1] do j := j + 1
        Q.enqueue((i, j));  i := j
    while Q.size() ≥ 2
        Q′ := new Queue
        while Q.size() ≥ 2
            (i₁, j₁) := Q.dequeue()
            (i₂, j₂) := Q.dequeue()
            merge(A[i₁..j₁), A[i₂..j₂), buf)
            copy buf to A[i₁..j₂)
            Q′.enqueue((i₁, j₂))
        if ¬Q.isEmpty() // lonely run
            Q′.enqueue(Q.dequeue())
        Q := Q′
```

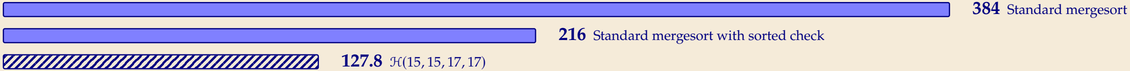
find run $A[i..j)$ starting at i

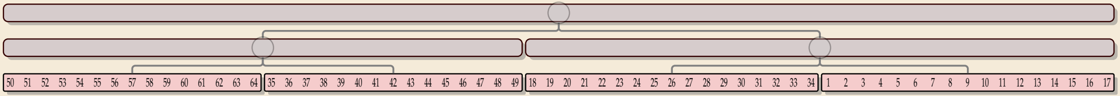
Natural Bottom-Up Mergesort – Analysis

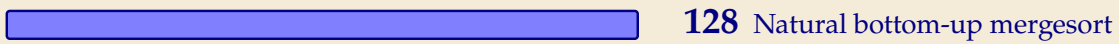
- Works well for runs of roughly equal size, regardless of alignment . . .



Merge costs:

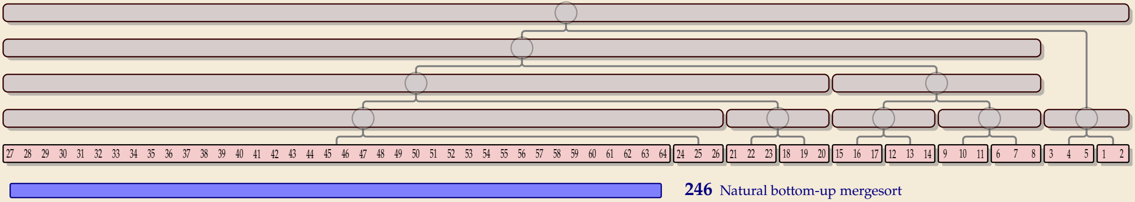


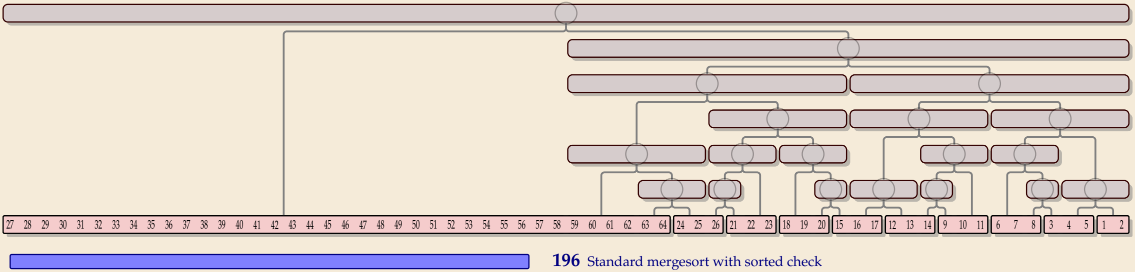




Natural Bottom-Up Mergesort – Analysis [2]

- . . . but less so for widely varying run lengths



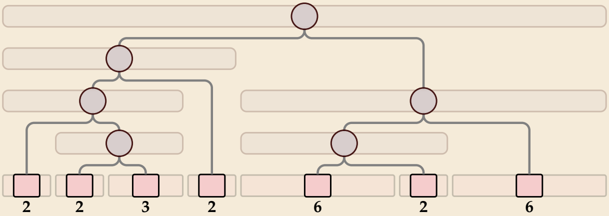


. . . can't we have both at the same time?!

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 $\rightsquigarrow$ $\ell_1, \ldots, \ell_r$ (easy) ✓
  2. **Determine a favorable *order of merges* of runs** ("automatic" in top-down mergesort)

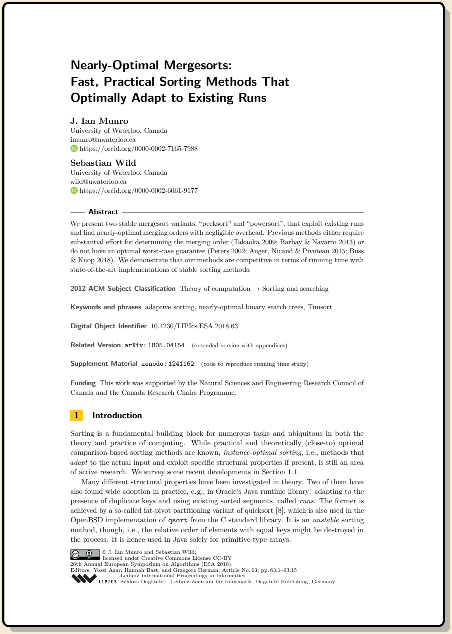


Merge cost = total area of $\boxed{}$
= total length of paths to all array entries
= $\displaystyle\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$

$\rightsquigarrow$ *optimal* merge tree
= optimal *binary search tree*
for leaf weights $\ell_1, \ldots, \ell_r$
(optimal expected search cost)

well-understood problem
with known algorithms

Nearly-Optimal Mergesort



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

J. Ian Munro
University of Waterloo, Canada
imunro@uwaterloo.ca
https://orcid.org/0000-0002-7165-7988

Sebastian Wild
University of Waterloo, Canada
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; Barbay & Navarro 2013) or do not have an optimal worst-case guarantee (Peters 2002; Auger, Nicaud & Pivoteau 2015; Buss & Knop 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 trees, Timsort

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.63

Related Version `arXiv: 1805.04154` (extended version with appendices)

Supplement Material `zenodo: 1241162` (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 fat-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;
licensed under Creative Commons License CC-BY

26th Annual European Symposium on Algorithms (ESA 2018).
Editors: Yossi Azar, Hannah Bast, and Grzegorz Herman; Article No. 63; pp. 63:1–63:15
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

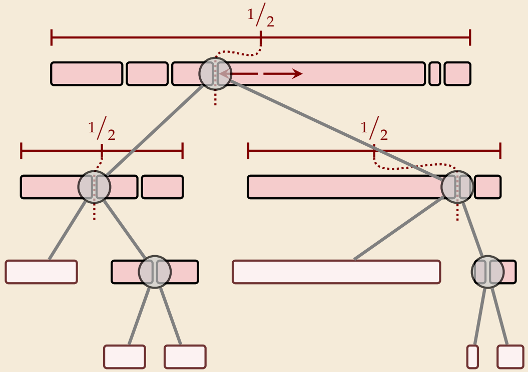
$\rightsquigarrow$ 2 new algorithms: *Peeksort* and *Powersort*

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

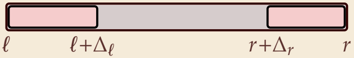
Peeksort

- based on top-down mergesort
- “peek” at middle of array
  & find closest run boundary

⇝ split there and recurse
  (instead of at midpoint)



- 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

procedure peeksort$(A[\ell..r), \Delta_\ell, \Delta_r)$:

  if $r - \ell \le 1$ **then return**

  if $\ell + \Delta_\ell == r \lor \ell == r + \Delta_r$ **then return**

  $m := \ell + \lfloor (r - \ell)/2 \rfloor$

  $i := \begin{cases} \ell + \Delta_\ell & \text{if } \ell + \Delta_\ell \ge m \\ \text{extendRunLeft}(A, m) & \text{else} \end{cases}$

  $j := \begin{cases} r + \Delta_r \le m & \text{if } r + \Delta_r \le m \le m \\ \text{extendRunRight}(A, m) & \text{else} \end{cases}$

  $g := \begin{cases} i & \text{if } m - i < j - m \\ j & \text{else} \end{cases}$

  $\Delta_g := \begin{cases} j - i & \text{if } m - i < j - m \\ i - j & \text{else} \end{cases}$

  peeksort$(A[\ell..g), \Delta_\ell, \Delta_g)$

  peeksort$(A[g, r), \Delta_g, \Delta_r)$

  merge$(A[\ell, g), A[g..r), \mathit{buf})$

  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

procedure extendRunRight$(A[0..n), i)$:

  $j := i + 1$

  while $j < n \land A[j-1] \le A[j]$

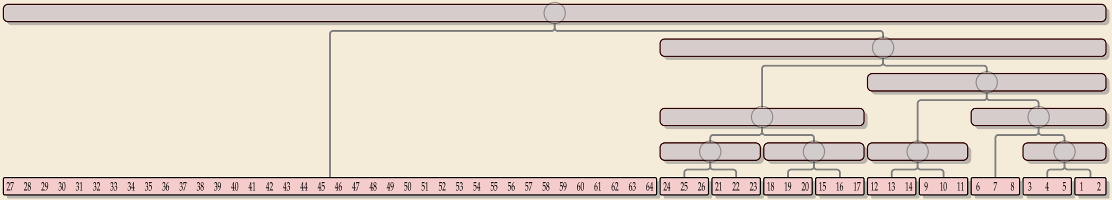
    $j := j + 1$

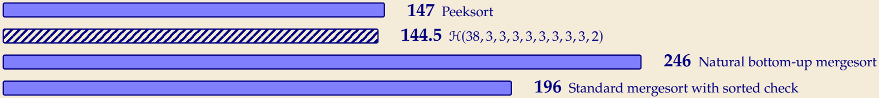
  return j

(extendRunLeft similar)

Peeksort – Analysis

- Consider tricky input from before again:





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

$\rightsquigarrow$ We can have the best of both worlds!

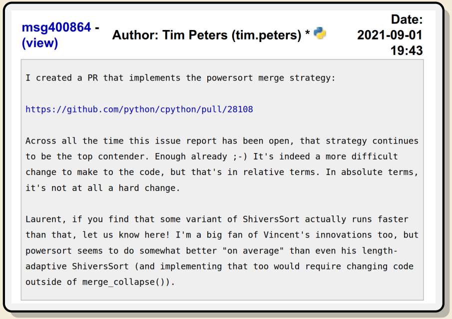
4.6 Python’s list sort

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

Sept 2021: **Python uses *Powersort*!**

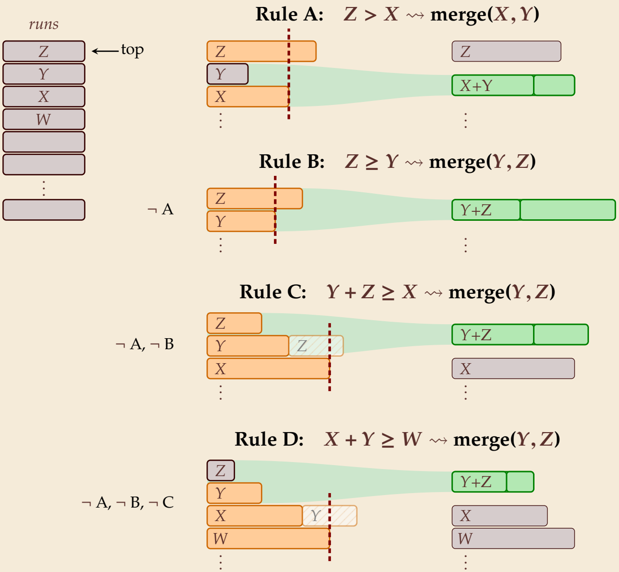
since CPython 3.11 and PyPy 7.3.6


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)

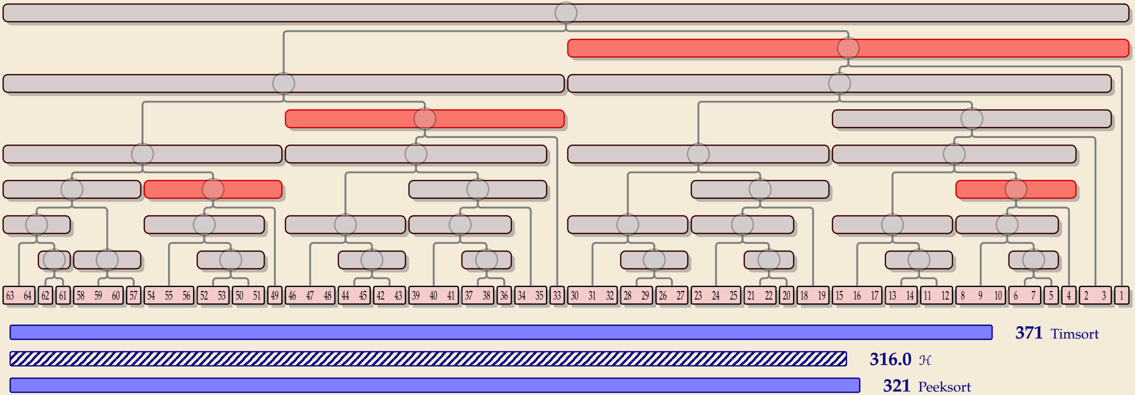
```
procedure Timsort(A[0..n)):
    i := 0; runs := new Stack()
    while i < n
        j := ExtendRunRight(A, i)
        runs.push(i, j); i := j
        while rule A/B/C/D applicable
            merge corresponding runs
    while runs.size() ≥ 2
        merge topmost 2 runs
```



- 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:

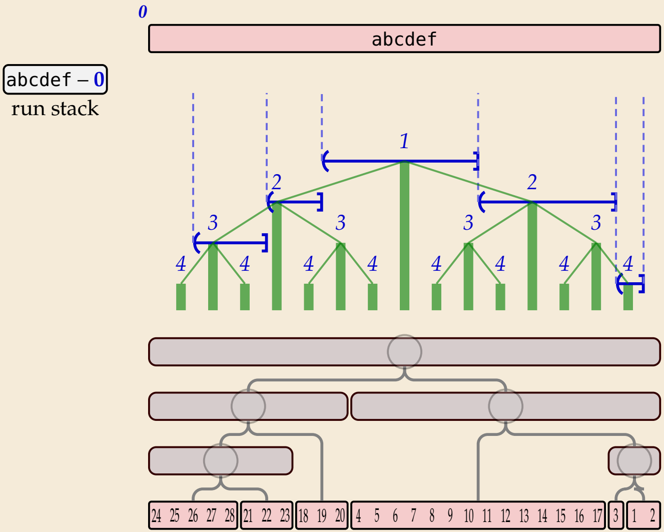


- As n increases, Timsort's cost approach $1.5 \cdot \mathcal{H}$, i. e., 50% more merge costs than necessary
  - intuitive problem: regularly very unbalanced merges

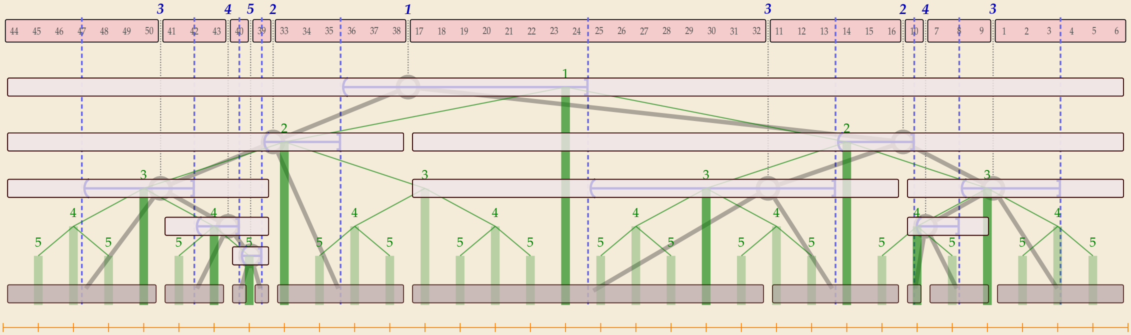
Powersort

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

```
procedure Powersort(A[0..n)):
    i := 0;  runs := new Stack()
    j := ExtendRunRight(A, i)
    runs.push((i, j), 0);  i := j
    while i < n
        j := ExtendRunRight(A, i)
        p := power(runs.top(), (i, j), n)
        while p ≤ runs.top().power
            merge topmost 2 runs
        runs.push((i, j), p);  i := j
    while runs.size() ≥ 2
        merge topmost 2 runs
```



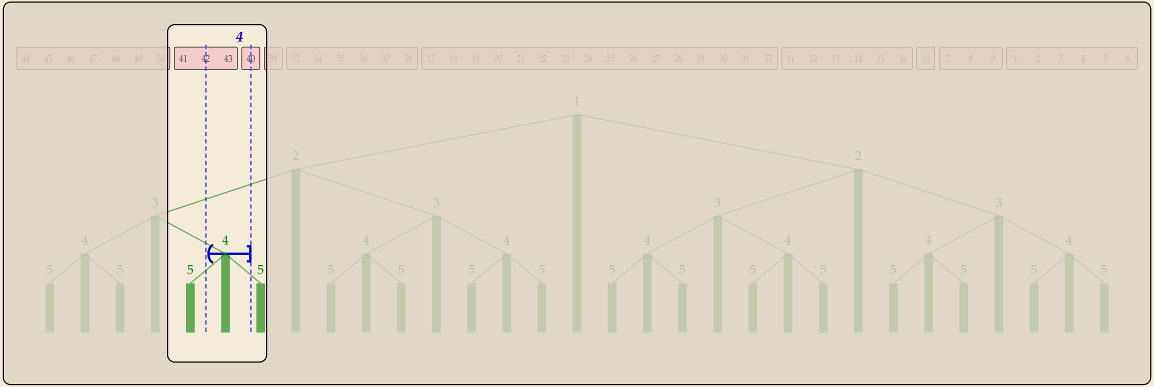
Powersort – Run-Boundary Powers



- (virtual) perfect balanced binary tree
- midpoint intervals "snap" to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its *power*

⇝ merge tree follows virtual tree

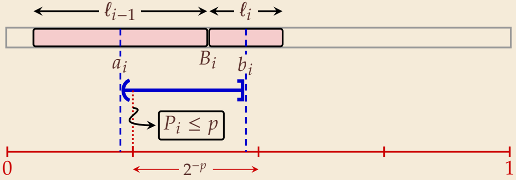
Powersort – Run-Boundary Powers are Local



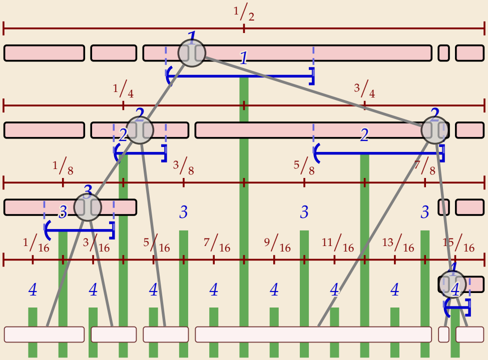
Computation of powers only depends on two adjacent runs.

Powersort – Computing powers

- Computing the power of (run boundary between) two runs
  - = normalized midpoint interval
  - power = min ℓ s.t. contains $c \cdot 2^{-\ell}$




```
procedure power((i_1, j_1), (i_2, j_2), n):
    n_1 := j_1 − i_1
    n_2 := j_2 − i_2
    a := (i_1 + ½ n_1 − 1) / n
    b := (i_2 + ½ n_2 − 1) / n      // interval (a, b]
    ℓ := 0
    while ⌊a · 2^ℓ⌋ == ⌊b · 2^ℓ⌋
        ℓ := ℓ + 1
    return ℓ
```



- with bitwise trickery $O(1)$ time possible

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