So sortiert man heute!

Sebastian Wild

www.wild-inter.net

based on joint work with Markus Nebel, Conrado Martínez, Ian Munro, Tamio Nakajima

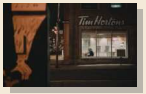
Marburg
University

Outline

1 Context

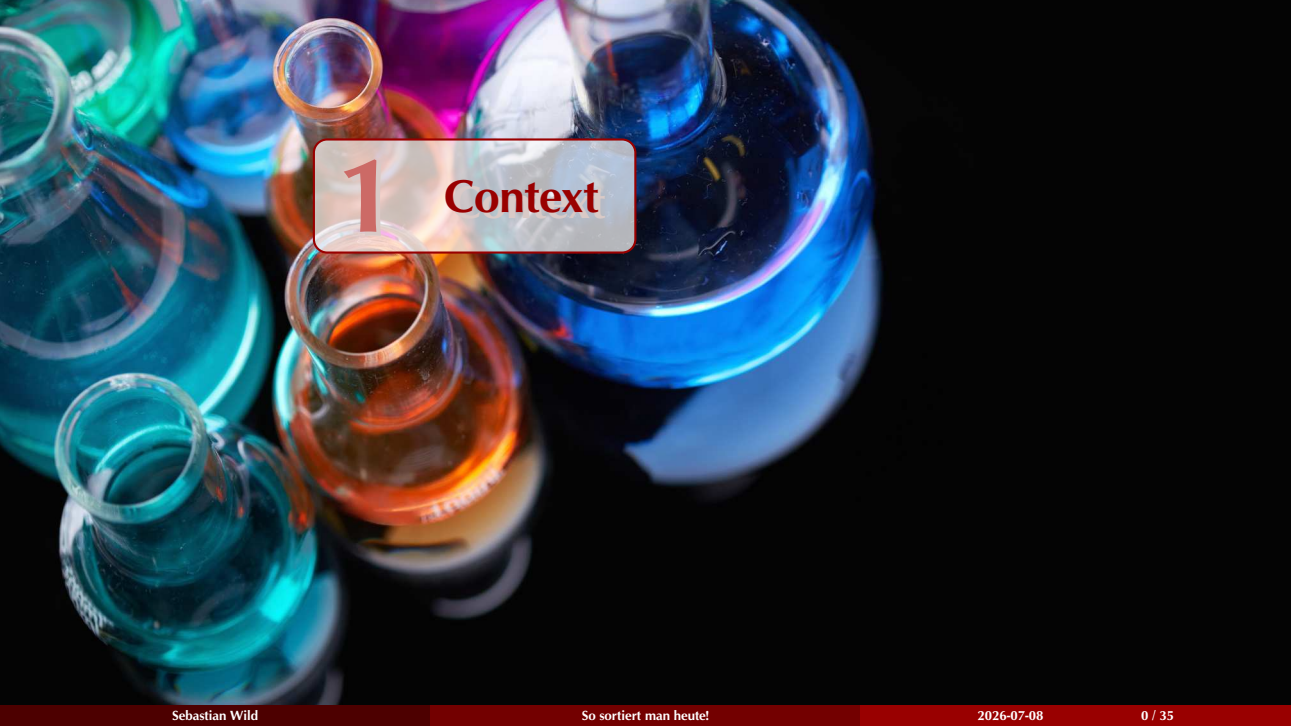
2 Java's Sorting Methods

3 Dual-Pivot Quicksort



4 Timsort

5 Powersort

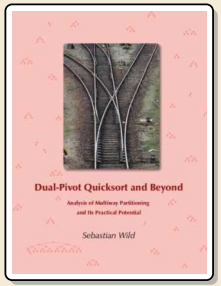


1 Context

Persönliche Geschichte

1. Dual-Pivot Quicksort

- Zufällig von Neuerung für Java 7 erfahren
- offensichtliche Fragen
  - *Warum ist Dual-Pivot Quicksort schneller?*
  - *Warum erst jetzt entdeckt?*
  - *Geht's noch besser?*
- Einige Antworten in meiner Doktorarbeit



2. Timsort

- 10+ Jahre im Einsatz
- **Aber:** Wiederholter Ärger

2015: subtiler algorithmischer Bug entdeckt!
Fix in Python und Java unterschiedlich

2018: subtiler Bug in Java-Fix entdeckt!

- 2018: Timsort bis zu 50% verschwenderisch!

⇝ **Fundamentales Problem in Merge-Policy!**

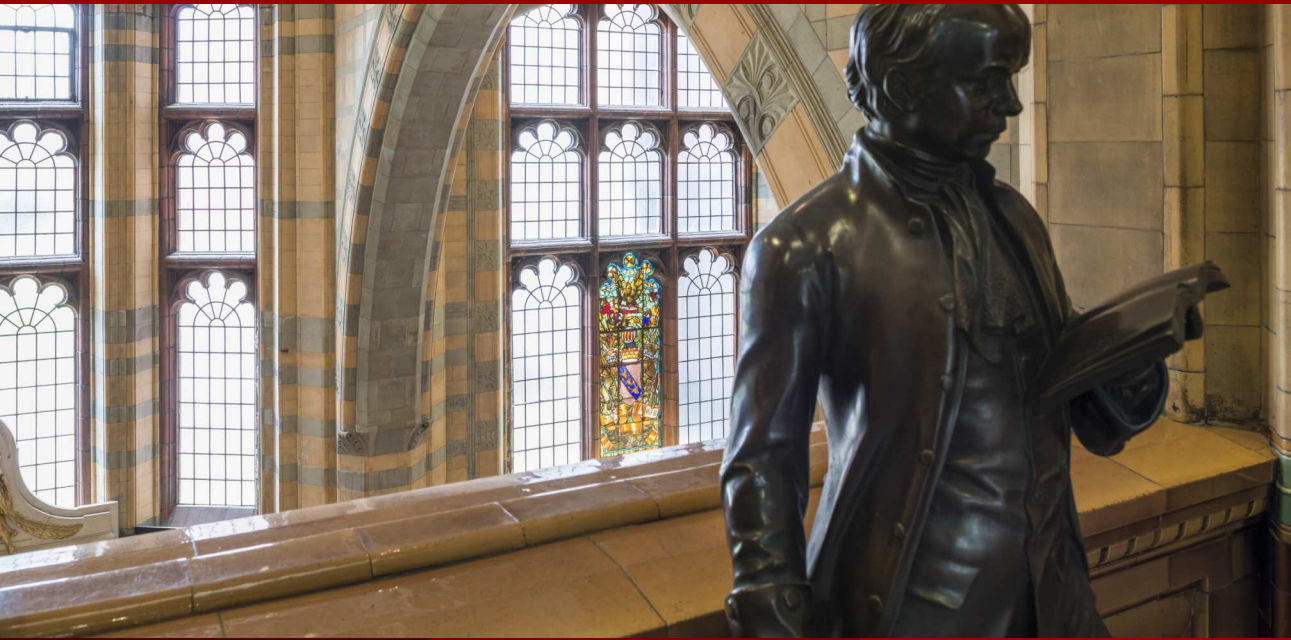
- *Unsere Merge-Policy* ***Powersort*** *behebt all das!*

⇝ ersetzt sukzessive alte Policy in Libraries

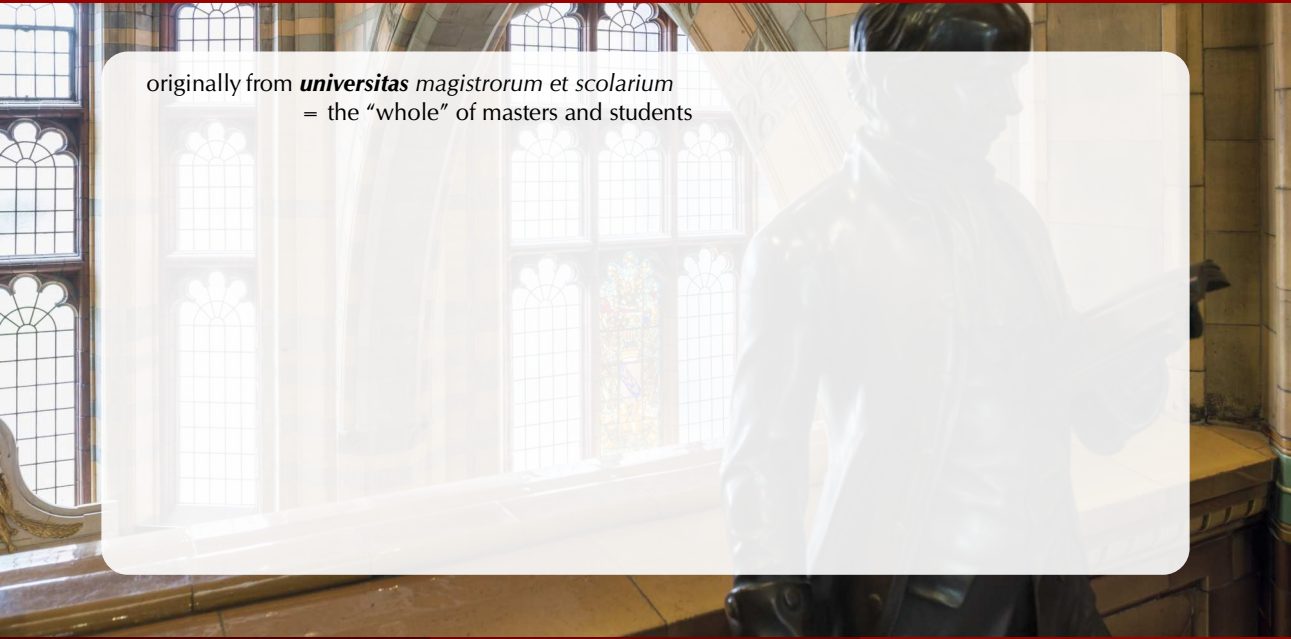
POWER sort=↑

Munro, Wild: *Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs*, ESA **2018**

What Is a University?



What Is a University?



originally from ***universitas*** *magistrorum et scolarium*

= the “whole” of masters and students

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the “whole” of masters and students

last 200 years: *Humboldt’s Ideal*
unity in teaching and research

Wilhelm von Humboldt

educational reformer of
early 19th century Prussia

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the “whole” of masters and students

last 200 years: *Humboldt’s Ideal*
unity in teaching and research

Wilhelm von Humboldt

educational reformer of
early 19th century Prussia

*Why should **students** care?*

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the “whole” of masters and students

last 200 years: *Humboldt’s Ideal*
unity in teaching and research

Wilhelm von Humboldt

educational reformer of
early 19th century Prussia

*Why should **students** care?*

- Researchers are (necessarily!) up to date

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the “whole” of masters and students

last 200 years: *Humboldt’s Ideal*
unity in teaching and research

Wilhelm von Humboldt

educational reformer of
early 19th century Prussia

*Why should **students** care?*

- Researchers are (necessarily!) up to date
- Research *is* learning

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the “whole” of masters and students

last 200 years: *Humboldt’s Ideal*
unity in teaching and research

Wilhelm von Humboldt

educational reformer of
early 19th century Prussia

*Why should **students** care?*

- Researchers are (necessarily!) up to date
- Research *is* learning
- Public funding follows research

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the "whole" of masters and students

last 200 years: *Humboldt's Ideal*
unity in teaching and research

Wilhelm von Humboldt

educational reformer of
early 19th century Prussia

*Why should **students** care?*

- Researchers are (necessarily!) up to date
- Research *is* learning
- Public funding follows research
- Research is the arena for universities
  in the competition for talent

What Is a University?

originally from ***universitas*** *magistrorum et scolarium*
= the “whole” of masters and students

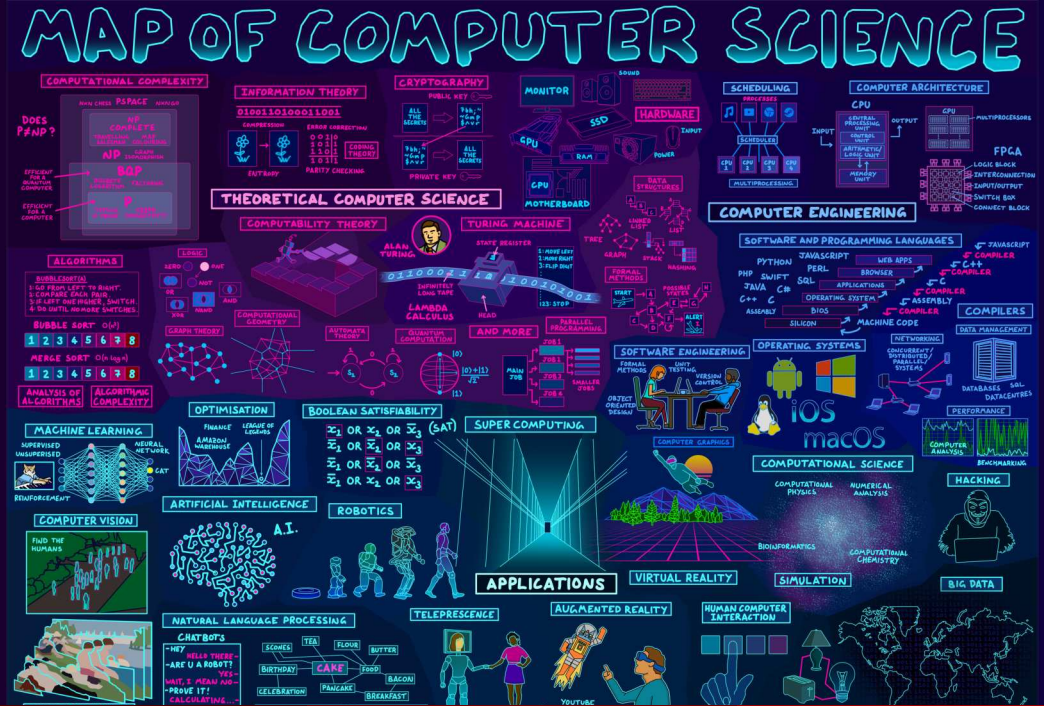
last 200 years: *Humboldt’s Ideal*
unity in teaching and research

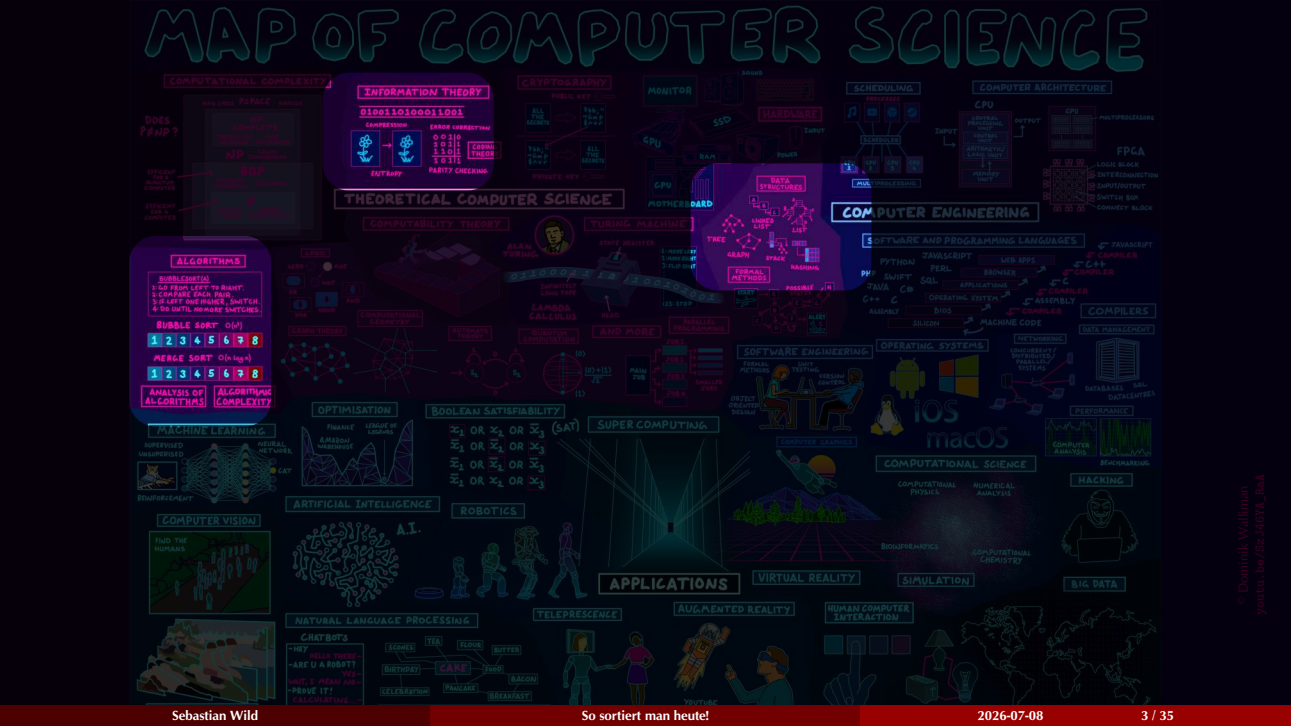
Wilhelm von Humboldt

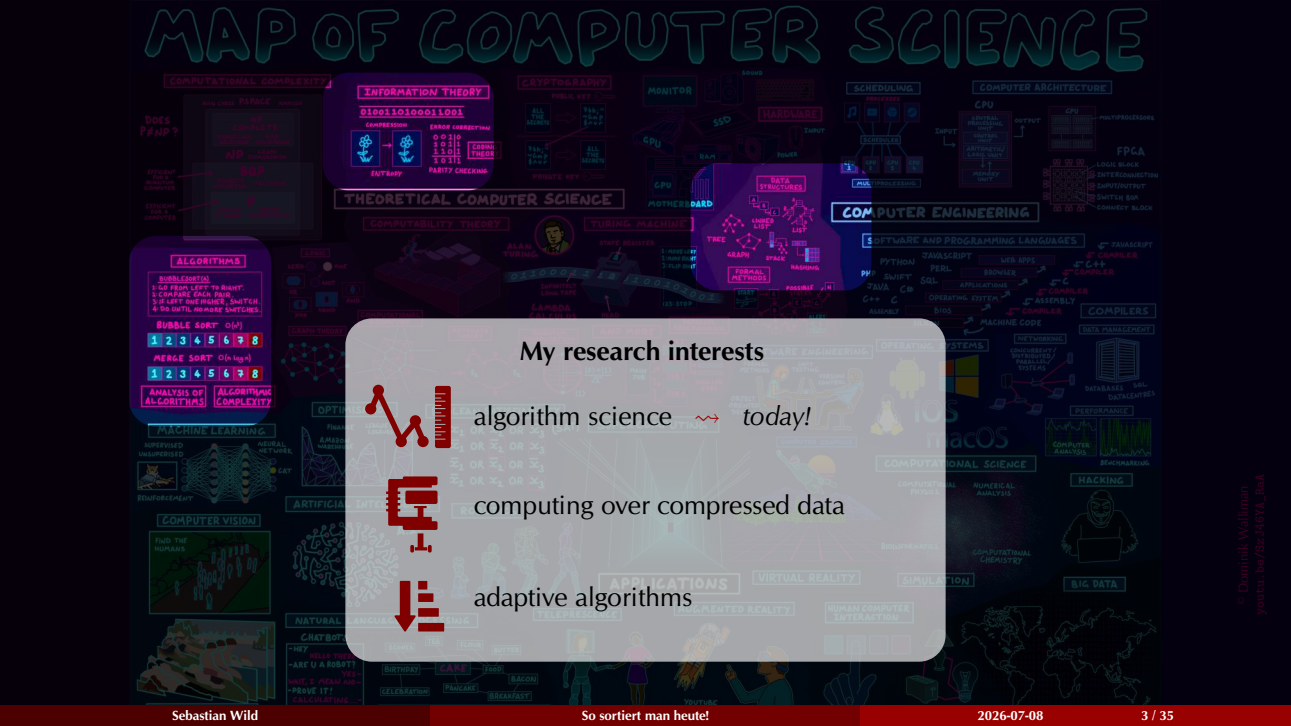
educational reformer of
early 19th century Prussia

*Why should **students** care?*

- Researchers are (necessarily!) up to date
- Research *is* learning
- Public funding follows research
- Research is the arena for universities
  in the competition for talent
- Chance to get involved in forefront of human inquiry!




MAP OF COMPUTER SCIENCE
THEORETICAL COMPUTER SCIENCE
COMPUTATIONAL COMPLEXITY
INFORMATION THEORY
CRYPTOGRAPHY
COMPUTABILITY THEORY
TURING MACHINE
ALGORITHMS
BUBBLESORT(A)
BUBBLE SORT O(n²)
MERGE SORT O(n log n)
ANALYSIS OF ALGORITHMS
ALGORITHMIC COMPLEXITY
DATA STRUCTURES
FORMAL METHODS
COMPUTER ENGINEERING
COMPUTER ARCHITECTURE
SCHEDULING
HARDWARE
SOFTWARE AND PROGRAMMING LANGUAGES
COMPILERS
OPERATING SYSTEMS
SOFTWARE ENGINEERING
MACHINE LEARNING
OPTIMISATION
BOOLEAN SATISFIABILITY
SUPER COMPUTING
ARTIFICIAL INTELLIGENCE
ROBOTICS
COMPUTER VISION
NATURAL LANGUAGE PROCESSING
APPLICATIONS
COMPUTATIONAL SCIENCE
HACKING
VIRTUAL REALITY
SIMULATION
BIG DATA
TELEPRESCENCE
AUGMENTED REALITY
HUMAN COMPUTER INTERACTION
© Dominik Wallman
youtu.be/6z3467A_PaA

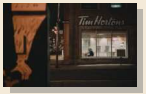

MAP OF COMPUTER SCIENCE
INFORMATION THEORY
THEORETICAL COMPUTER SCIENCE
COMPUTER ENGINEERING
ALGORITHMS
BUBBLE SORT O(n²)
MERGE SORT O(n log n)
ANALYSIS OF ALGORITHMS
ALGORITHMIC COMPLEXITY
MACHINE LEARNING
COMPUTER VISION
COMPUTATIONAL SCIENCE
HACKING
BIG DATA
My research interests
algorithm science ⇝ today!
computing over compressed data
adaptive algorithms
© Dominik Wallinnan
youtu.be/6z14671k_Fxk

Outline

1 Context

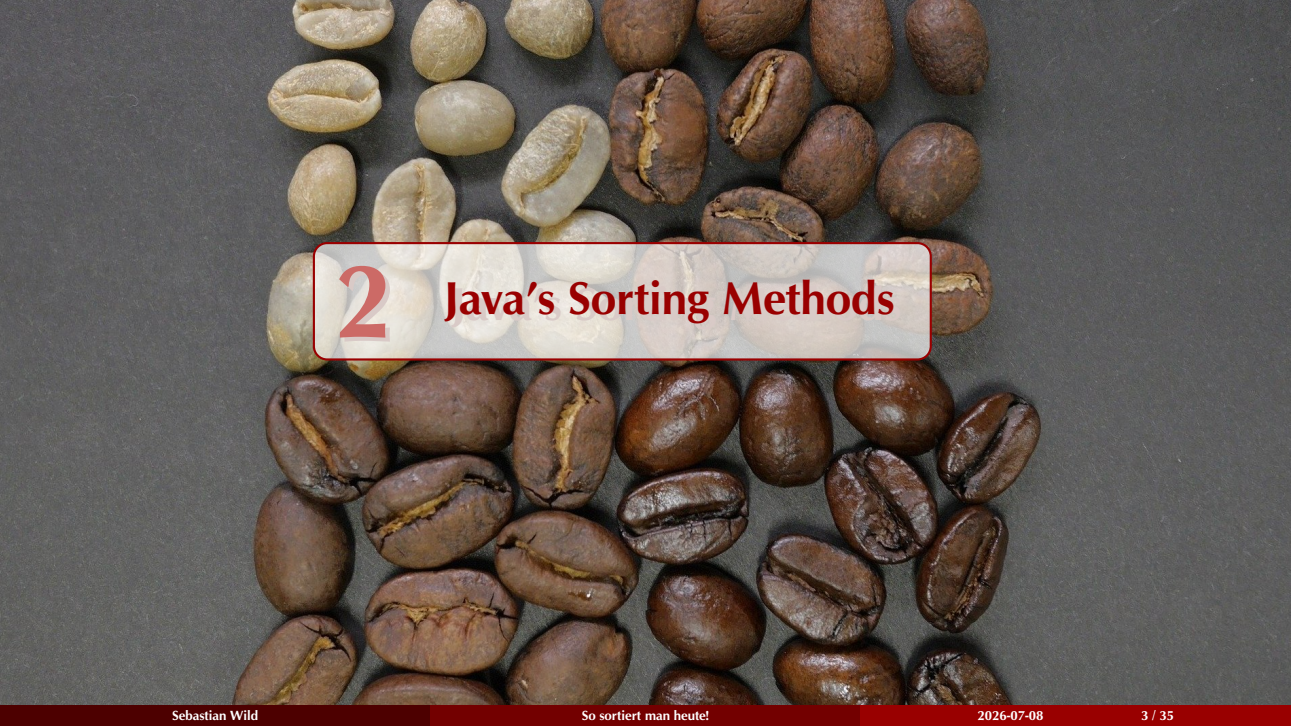
2 Java's Sorting Methods

3 Dual-Pivot Quicksort



4 Timsort

5 Powersort



2 Java's Sorting Methods

Adaptive Sorting

Adaptive algorithm: exploit “structure” of input

Adaptive Sorting

Adaptive algorithm: exploit “structure” of input

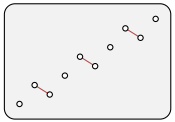
$\rightsquigarrow$ **adaptive sorting**: exploit “presortedness”

Adaptive Sorting

Adaptive algorithm: exploit “structure” of input

$\rightsquigarrow$ **adaptive sorting**: exploit “presortedness”

- few **inversions**

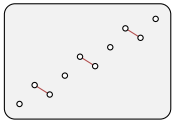


Adaptive Sorting

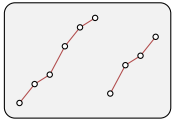
Adaptive algorithm: exploit “structure” of input

$\rightsquigarrow$ **adaptive sorting**: exploit “presortedness”

- few **inversions**



- few **runs**

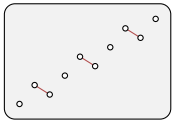


Adaptive Sorting

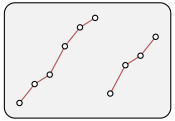
Adaptive algorithm: exploit “structure” of input

$\rightsquigarrow$ **adaptive sorting**: exploit “presortedness”

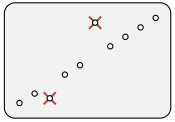
- few **inversions**



- few **runs**



- few **outliers**

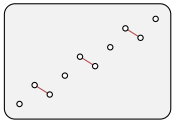


Adaptive Sorting

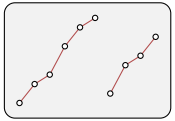
Adaptive algorithm: exploit “structure” of input

$\rightsquigarrow$ **adaptive sorting**: exploit “presortedness”

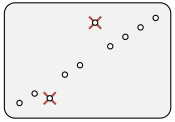
- few **inversions**



- few **runs**



- few **outliers**



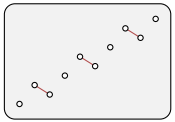
- . . . many more

Adaptive Sorting

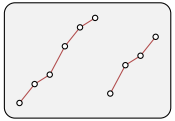
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

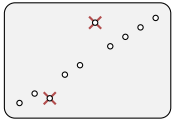
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

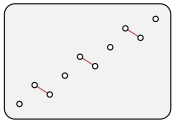
optimal algorithms known for many measures of presortedness

Adaptive Sorting

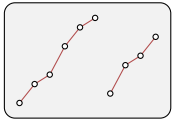
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

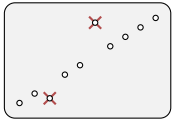
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

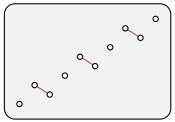
optimal algorithms known for many measures of presortedness

Adaptive Sorting

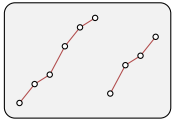
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

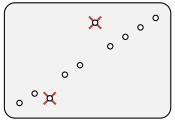
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

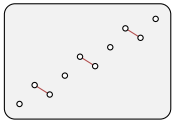
Want:

Adaptive Sorting

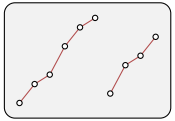
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

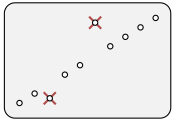
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

Want:

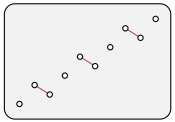
- Optimal up to lower order terms

Adaptive Sorting

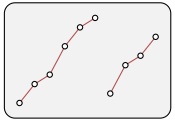
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

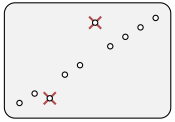
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

Want:

- Optimal up to lower order terms

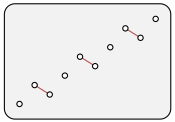
practical methods

Adaptive Sorting

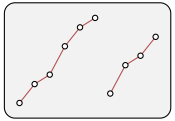
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

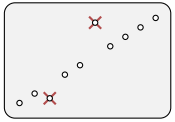
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

Want:

- Optimal up to lower order terms

practical methods

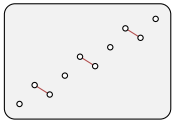
- **low overhead** for detecting presortedness

Adaptive Sorting

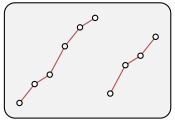
Adaptive algorithm: exploit “structure” of input

⇝ **adaptive sorting**: exploit “presortedness”

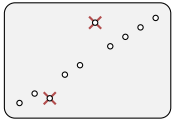
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

Want:

- Optimal up to lower order terms

practical methods

- **low overhead** for detecting presortedness
- competitive on inputs **without** presortedness

Algorithm Science

Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine

Algorithm Science

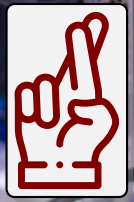
Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?

Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?

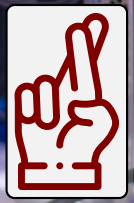


Theory of Algorithms

Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?



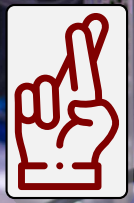
Theory of Algorithms

- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.

Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?



Theory of Algorithms

- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.

Unconditional promise, mathematically proven

Algorithm Science

Algorithm & Systems Engineering

Theory of Algorithms

- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.

👍 Unconditional promise, mathematically proven

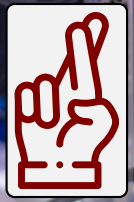
👎 Pessimism often a poor advisor.
Focus on worst case can lead to paralysis.

"Don't bother. In the worst case, only a constant factor faster."

Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?



Theory of Algorithms

- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.

Unconditional promise, mathematically proven

Pessimism often a poor advisor.
Focus on worst case can lead to paralysis.

"Don't bother. In the worst case, only a constant factor faster."

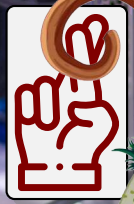
Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?

Theory of Algorithms

- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.



Unconditional promise, mathematically proven

Pessimism often a poor advisor.

worst case can lead to paralysis.

Don't bother. In the [illegible] only a constant factor faster."

Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?

Theory of Algorithms

- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.

👍 Unconditional promise, mathematically proven

Pessimism often a poor advisor.

worst case can lead to paralysis.

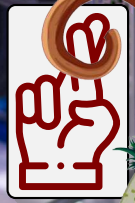
Don't bother, in the ... only a constant factor faster."

1 ***Abstract Cost Models***

precise asymptotic analysis
⇝ quantitative predictions!

2 ***Adaptive Analysis***

fine-grained input features
⇝ ≠ worst case



Algorithm Science

Algorithm & Systems Engineering

- **Measured Performance:**
  actual time on actual machine
- specific to machine
- reproducible?
- does it *generalize*?

Theory of Algorithms

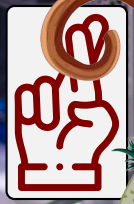
- ***Big-Oh Worst-Case Guarantees:***
  No matter what the input, cost is $O(n^2)$.

Unconditional promise, mathematically proven

Pessimism often a poor advisor.

worst case can lead to paralysis.

only a constant factor faster."



1 ***Abstract Cost Models***

precise asymptotic analysis
⇝ quantitative predictions!

2 ***Adaptive Analysis***

fine-grained input features
⇝ ≠ worst case

Platform-independent performance guarantees

- concrete quantitative model
- validated by experiments

Case Study: Sorting in Java Library

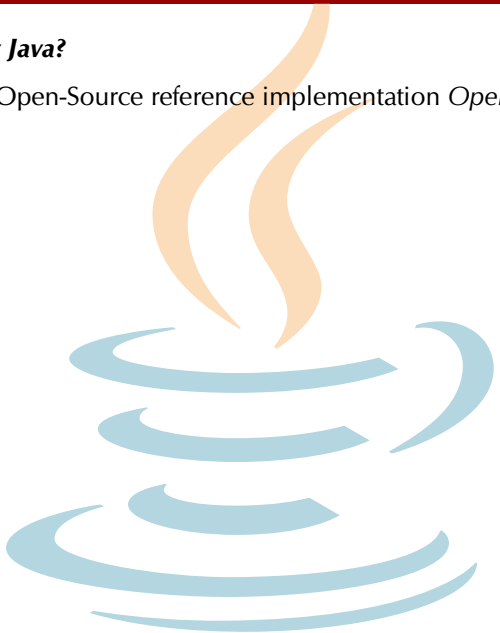
Why Java?

Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*

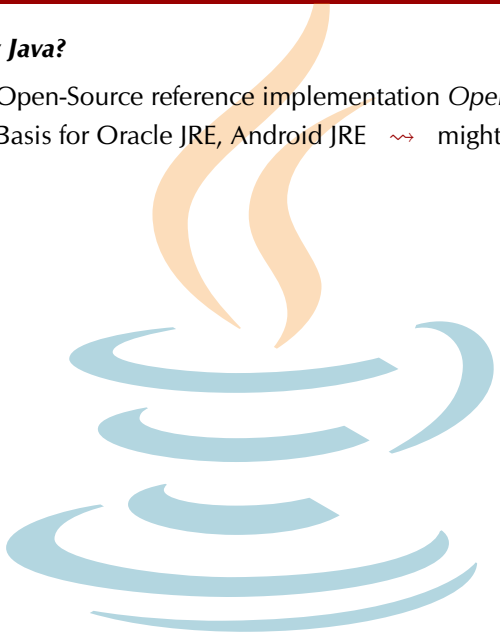


Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!

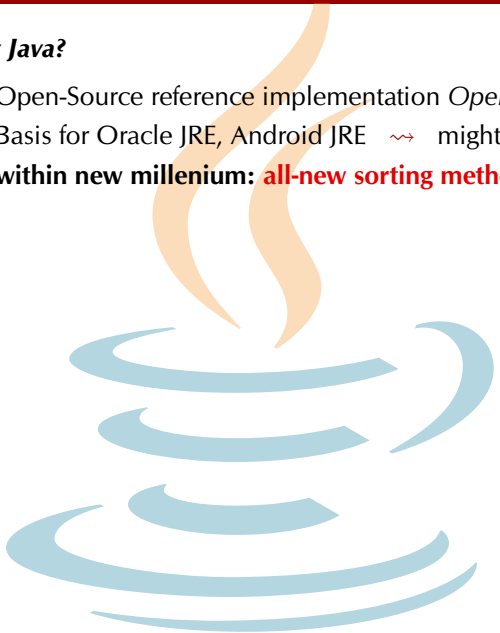


Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**



Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

1 ***Dual-Pivot Quicksort***

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK

Case Study: Sorting in Java Library

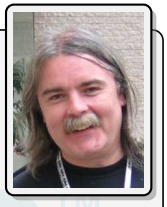
Why Java?

- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

1 ***Dual-Pivot Quicksort***

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK

2 ***Timsort***



- for objects (`Object[]`)
- adaptive stable mergesort
- 2002 developed for `list.sort` in CPython by *Tim Peters*
- since 2009 in OpenJDK

Case Study: Sorting in Java Library

Why Java?

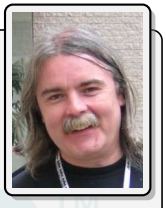
- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

1 *Dual-Pivot Quicksort*

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK

2 *Timsort*

- for objects (`Object[]`)
- adaptive stable mergesort
- 2002 developed for `list.sort` in CPython by *Tim Peters*
- since 2009 in OpenJDK



- both algorithms from **developers**

Case Study: Sorting in Java Library

Why Java?

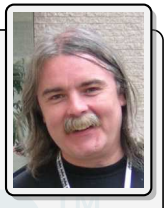
- Open-Source reference implementation *OpenJDK*
- Basis for Oracle JRE, Android JRE $\rightsquigarrow$ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

1 *Dual-Pivot Quicksort*

- for primitive types (`int[]`, `double[]` etc.)
- 2008 Started as hobby project by *Vladimir Yaroslavskiy* (Sun Microsystems)
- since 2009 in OpenJDK

2 *Timsort*

- for objects (`Object[]`)
- adaptive stable mergesort
- 2002 developed for `list.sort` in CPython by *Tim Peters*
- since 2009 in OpenJDK



- both algorithms from **developers**

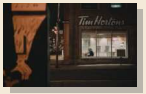
$\rightsquigarrow$ **Focus:** Benchmarks & Unit-Tests not on proofs or analysis

Outline

1 Context

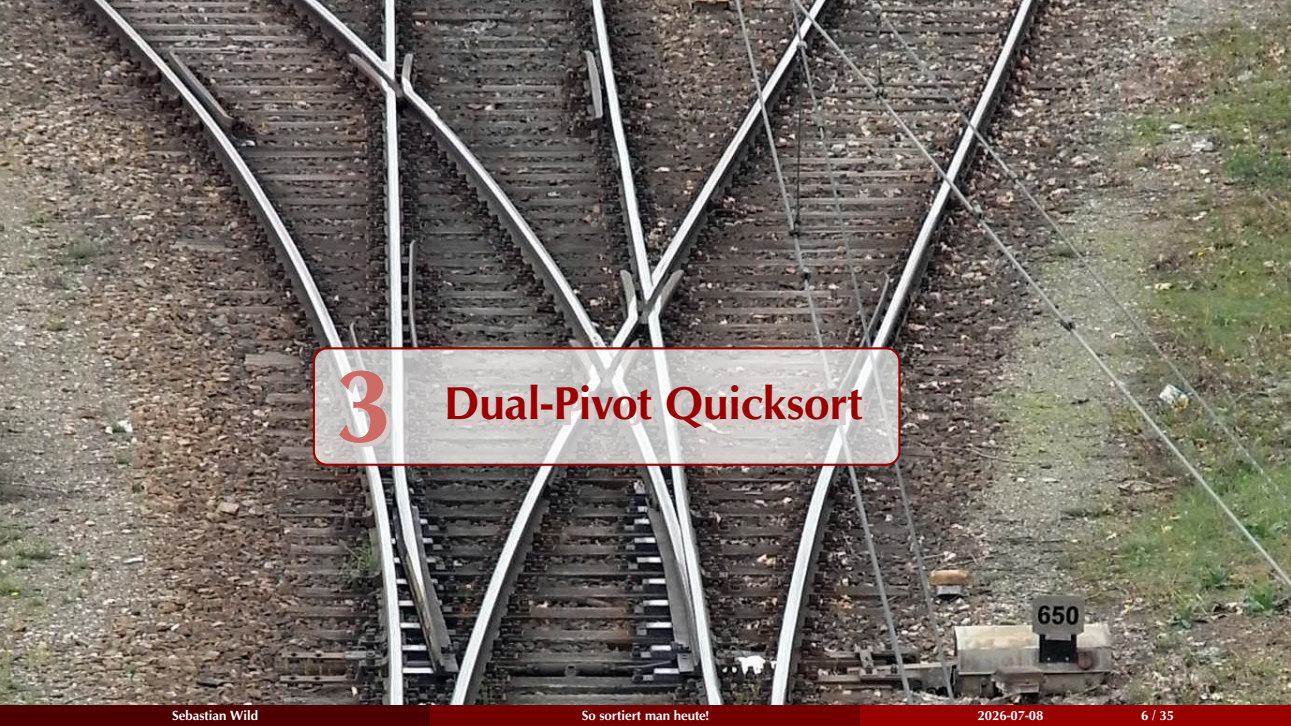
2 Java's Sorting Methods

3 Dual-Pivot Quicksort



4 Timsort

5 Powersort



3 Dual-Pivot Quicksort

Quicksort Implementations in Libraries

Context: Internal, unstable, general-purpose sorting methods

Quicksort Implementations in Libraries

Context: Internal, unstable, general-purpose sorting methods

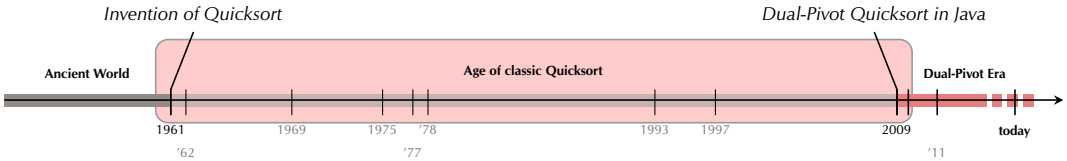
1961,62 **Hoare:** publication, first analysis

1969 **Singleton:** median-of-three & Insertionsort on small subarrays

1975-78 **Sedgewick:** analysis of many optimizations

1993 **Bentley, McIlroy:** duplicate elements & “ninther”

1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

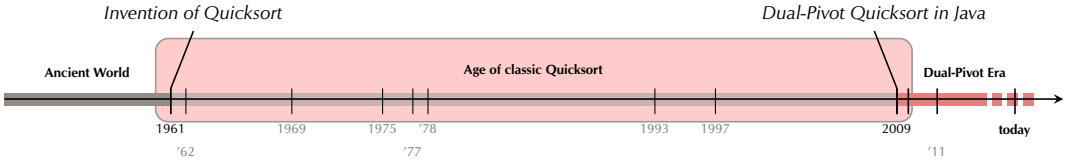


Quicksort Implementations in Libraries

Context: Internal, unstable, general-purpose sorting methods

1961,62 **Hoare:** publication, first analysis
1969 **Singleton:** median-of-three & Insertionsort on small subarrays
1975-78 **Sedgewick:** analysis of many optimizations
1993 **Bentley, McIlroy:** duplicate elements & "ninther"
1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.



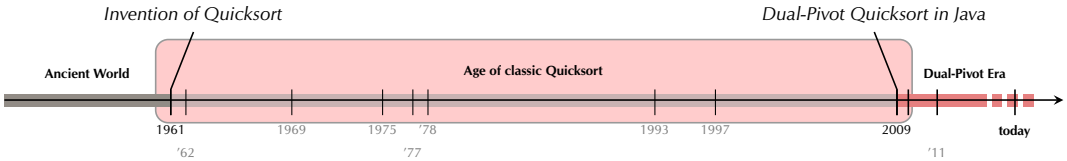
Quicksort Implementations in Libraries

Context: Internal, unstable, general-purpose sorting methods

1961,62 **Hoare:** publication, first analysis
1969 **Singleton:** median-of-three & Insertionsort on small subarrays
1975-78 **Sedgewick:** analysis of many optimizations
1993 **Bentley, McIlroy:** duplicate elements & "ninther"
1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.

Since then: Almost identical in all programming libraries!



Quicksort Implementations in Libraries

Context: Internal, unstable, general-purpose sorting methods

2008 – 2009 **Vladimir Yaroslavskiy** (developer at Sun)
experiments with Quicksort with **two pivots**

2009 – 2011 optimizations by Joshua Bloch, Jon Bentley and others

11 Sep 2009 announcement on Java core library mailing list

29 Oct 2009 **inclusion** in development version of OpenJDK

28 July 2011 **public release** of Java 7 with ***Yaroslavskiy-Bentley-Bloch (YBB) Quicksort***

13 Nov 2019 Major update of Dual-Pivot Quicksort for Java 14

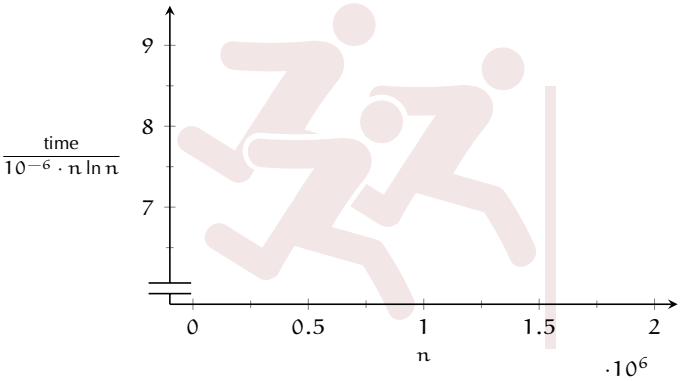
OpenJDK's Dual-Pivot Quicksort

- With Java 7 (2009), tried-and-tested Quicksort implementation in OpenJDK **replaced with *Dual-Pivot Quicksort***

used for primitive types

OpenJDK's Dual-Pivot Quicksort

- With Java 7 (2009), tried-and-tested Quicksort implementation in OpenJDK **replaced with *Dual-Pivot Quicksort***
  used for primitive types
- *Why change a well-tested implementation?*



Normalized Java runtimes (in *ms*).

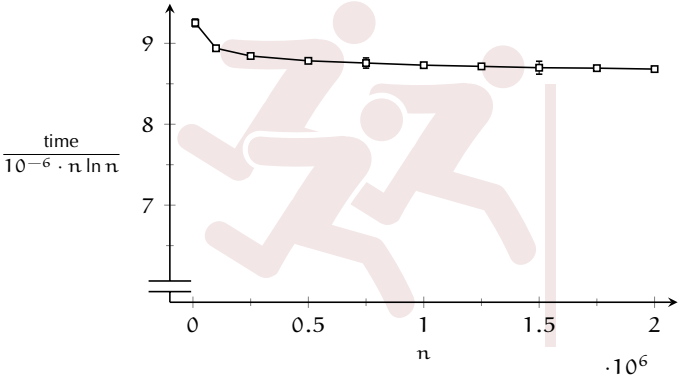
Average of 1000 random permutations per size (`int[]`).

OpenJDK's Dual-Pivot Quicksort

- With Java 7 (2009), tried-and-tested Quicksort implementation in OpenJDK **replaced with *Dual-Pivot Quicksort***

used for primitive types

- *Why change a well-tested implementation?*



Normalized Java runtimes (in *ms*).

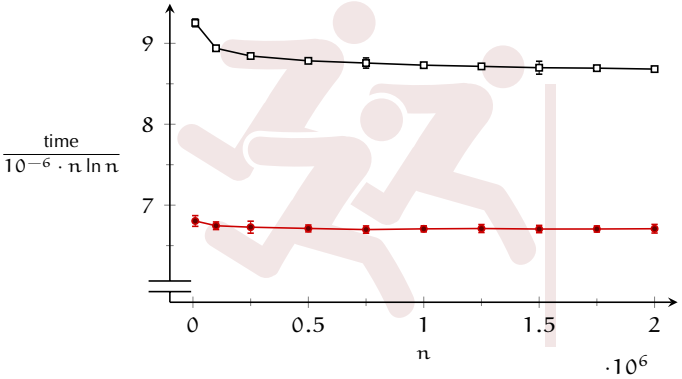
Average of 1000 random permutations per size (`int[]`).

OpenJDK's Dual-Pivot Quicksort

- With Java 7 (2009), tried-and-tested Quicksort implementation in OpenJDK **replaced with *Dual-Pivot Quicksort***

used for primitive types

- *Why change a well-tested implementation?*



Normalized Java runtimes (in *ms*).

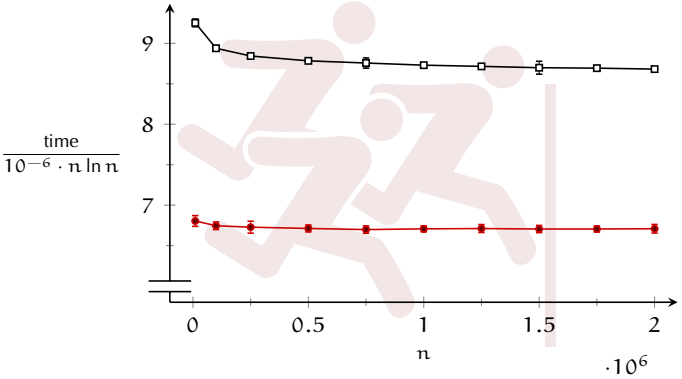
Average of 1000 random permutations per size (`int[]`).

OpenJDK's Dual-Pivot Quicksort

- With Java 7 (2009), tried-and-tested Quicksort implementation in OpenJDK **replaced with *Dual-Pivot Quicksort***

used for primitive types

- *Why change a well-tested implementation?*



Normalized Java runtimes (in *ms*).

Average of 1000 random permutations per size (`int[]`).

⇝ Another Algorithm-Science case study!

Quicksort: Conceptual View

Single-Pivot Quicksort

Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.

Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   by comparing x to P.

Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.

Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   ← assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   (assuming distinct elements in this talk)

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   ← assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class**
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

   by comparing x to **pivots** P_1 and P_2

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class**
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

   by comparing x to **pivots** P_1 and P_2
3. Arrange elements according to classes:

Dijkstra's Dutch National Flag Problem

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P.
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   assuming distinct elements in this talk

   by comparing x to P.
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort

1. Choose **two pivots** $P_1 \leqslant P_2$
2. For each element x, determine its **class**
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

   by comparing x to **pivots** P_1 and P_2
3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dijkstra's Dutch National Flag Problem

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   by comparing x to P.

   assuming distinct elements in this talk
3. Arrange elements according to classes:

4. Sort subarrays recursively.

How to do ❸ efficiently and in place?

Dual-Pivot Quicksort

2. For each element x, determine its **class**
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

   by comparing x to **pivots** P_1 and P_2

Dijkstra's Dutch National Flag Problem

3. Arrange elements according to classes:

4. Sort subarrays recursively.

Dual-Pivot Quicksort: Conceptual View

Single-Pivot Quicksort

1. Choose pivots P
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **large** for $P < x$

   by comparing x to P.

   assuming distinct elements in this talk
3. Arrange elements according to classes:

4. Sort subarrays recursively.

How to do ❸ efficiently and in place? ⇝ ***YBB Partitioning***

Dual-Pivot Quicksort

2. For each element x, determine its **class**
   - **small** for $x < P_1$
   - **medium** for $P_1 < x < P_2$
   - **large** for $P_2 < x$

   by comparing x to **pivots** P_1 and P_2

Dijkstra's Dutch National Flag Problem

3. Arrange elements according to classes:

4. Sort subarrays recursively.

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5        do k := k + 1 while A[k] < P end while
 6        do g := g − 1 while A[g] > P end while
 7        if k ⩾ g then break while end if
 8        Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right )
```

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5        do k := k + 1 while A[k] < P end while
 6        do g := g − 1 while A[g] > P end while
 7        if k ⩾ g then break while end if
 8        Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right )
```

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5        do k := k + 1 while A[k] < P end while
 6        do g := g − 1 while A[g] > P end while
 7        if k ⩾ g then break while end if
 8        Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right )
```

- **Classic Quicksort**

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5        do k := k + 1 while A[k] < P end while
 6        do g := g − 1 while A[g] > P end while
 7        if k ⩾ g then break while end if
 8        Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right )
```

- **Classic Quicksort**
  - *crossing-pointer* scheme 

Basic Algorithms

```
ClassicQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P := A[right]
 3  k := left − 1;  g := right
 4  while true
 5      do k := k + 1 while A[k] < P end while
 6      do g := g − 1 while A[g] > P end while
 7      if k ⩾ g then break while end if
 8      Swap A[k] and A[g]
 9  end while
10  Swap A[k] and A[right]
11  ClassicQuicksort(A,  left  , k − 1)
12  ClassicQuicksort(A, k + 1,  right )
```

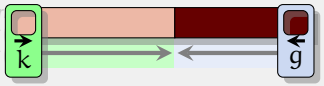
- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment 

Basic Algorithms

ClassicQuicksort(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[r
11 ClassicQuicksort
12 ClassicQuicksort
```

Invariant

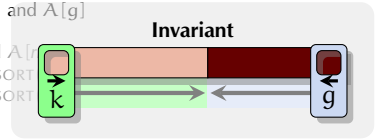


- **Classic Quicksort**
  - *crossing-pointer* scheme
  - symmetric treatment

Basic Algorithms

ClassicQuicksort$(A, \mathit{left}, \mathit{right})$

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[r
11 ClassicQuicksort
12 ClassicQuicksort
```



- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment 
- **YBB Quicksort**

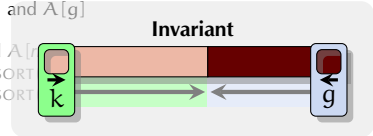
YbbQuicksort$(A, \mathit{left}, \mathit{right})$

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A,  left  , ℓ − 1)
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1,  right )
```

Basic Algorithms

ClassicQuicksort$(A, \mathit{left}, \mathit{right})$

```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[r
11 ClassicQuicksort
12 ClassicQuicksort
```



- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment 
- **YBB Quicksort**

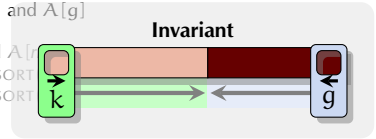
YbbQuicksort$(A, \mathit{left}, \mathit{right})$

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A,  left  , ℓ − 1)
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1,  right )
```

Basic Algorithms

ClassicQuicksort$(A, left, right)$

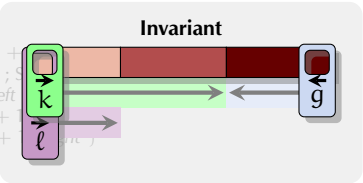
```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[r
11 ClassicQuicksort
12 ClassicQuicksort
```



- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment 
- **YBB Quicksort**

YbbQuicksort$(A, left, right)$

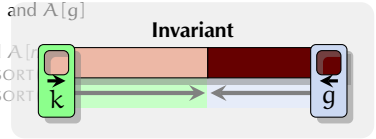
```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g +
20 Swap A[left] and A[ℓ] ; S
21 YbbQuicksort(A,  left
22 YbbQuicksort(A, ℓ +
23 YbbQuicksort(A, g +
```



Basic Algorithms

ClassicQuicksort$(A, left, right)$

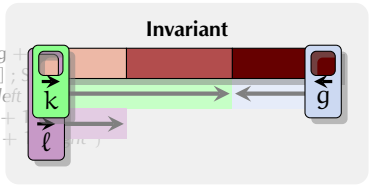
```
1  if right ⩽ left then return
2  P := A[right]
3  k := left − 1;  g := right
4  while true
5      do k := k + 1 while A[k] < P end while
6      do g := g − 1 while A[g] > P end while
7      if k ⩾ g then break while end if
8      Swap A[k] and A[g]
9  end while
10 Swap A[k] and A[r
11 ClassicQuicksort
12 ClassicQuicksort
```



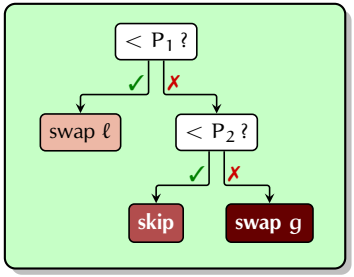
- **Classic Quicksort**
  - *crossing-pointer* scheme 
  - symmetric treatment 
- **YBB Quicksort**
  - . . . looks messy 

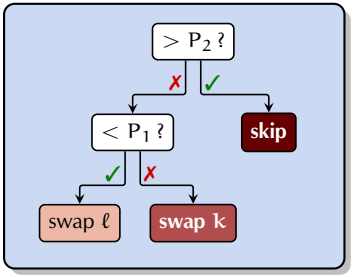
YbbQuicksort$(A, left, right)$

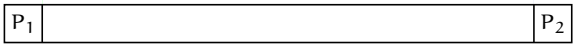
```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g +
20 Swap A[left] and A[ℓ] ; S
21 YbbQuicksort(A,  left
22 YbbQuicksort(A, ℓ +
23 YbbQuicksort(A, g +
```



YBB Partitioning Example

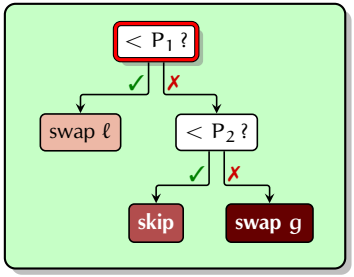


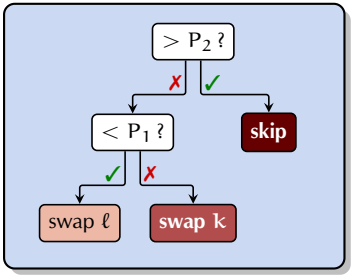


Invariant: 

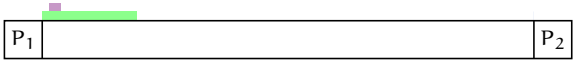

```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



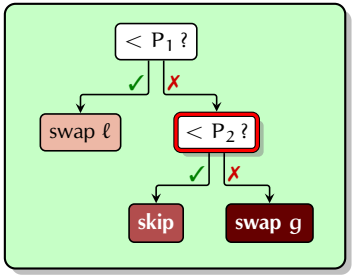


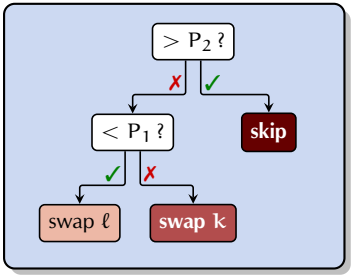
Invariant:



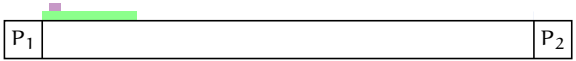

```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6  → if A[k] < P1
 7        Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8    else if A[k] ⩾ P2
 9        while A[g] > P2 and k < g
10            g := g − 1
11        end while
12        Swap A[k] and A[g] ; g := g − 1
13        if A[k] < P1
14            Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15        end if
16    end if
17    k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



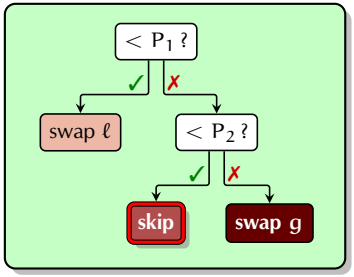


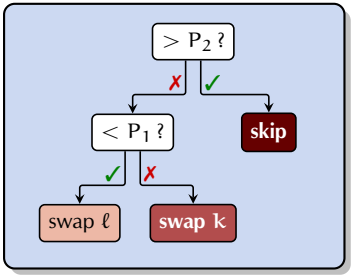
Invariant:



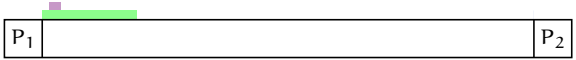

```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



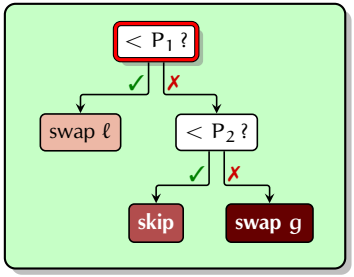


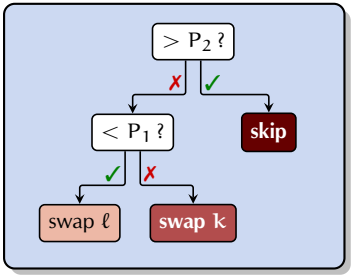
Invariant:

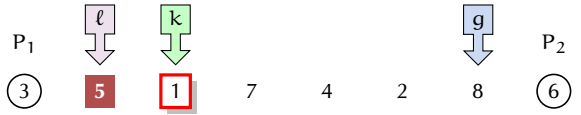


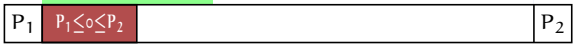

```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



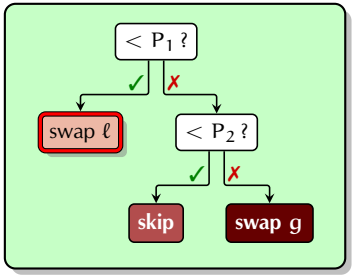


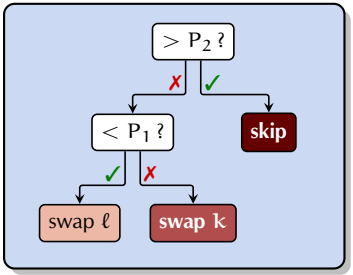


Invariant: 


```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6  → if A[k] < P1
 7        Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8    else if A[k] ⩾ P2
 9        while A[g] > P2 and k < g
10            g := g − 1
11        end while
12        Swap A[k] and A[g] ; g := g − 1
13        if A[k] < P1
14            Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15        end if
16    end if
17    k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example

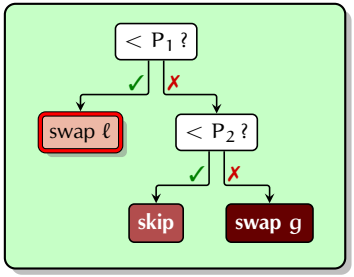


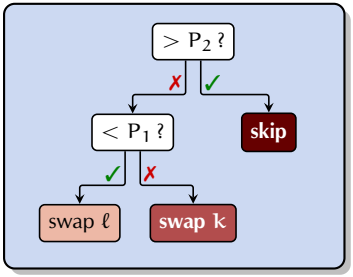


Invariant: $P_1 \leq o \leq P_2$

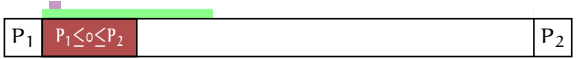
```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



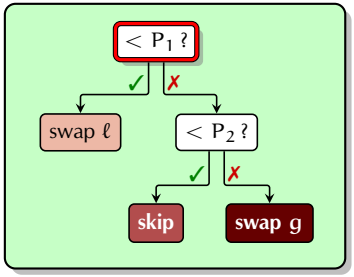


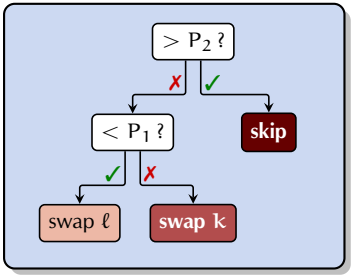
Invariant:

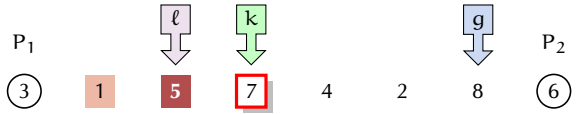



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

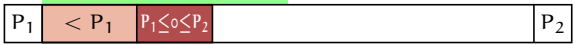
YBB Partitioning Example





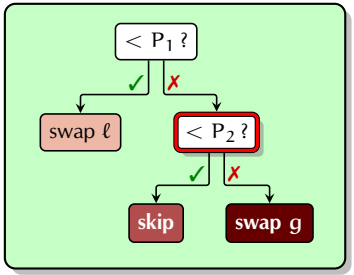


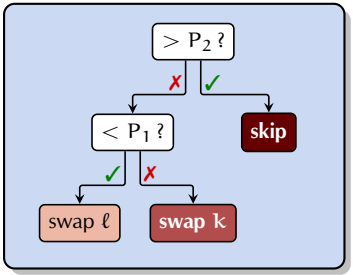
Invariant:

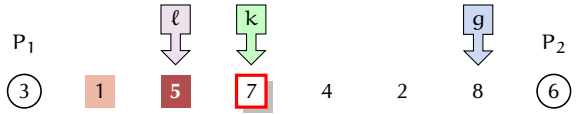



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6  → if A[k] < P1
 7        Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8    else if A[k] ⩾ P2
 9        while A[g] > P2 and k < g
10            g := g − 1
11        end while
12        Swap A[k] and A[g] ; g := g − 1
13        if A[k] < P1
14            Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15        end if
16    end if
17    k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A, left , ℓ − 1)
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1, right )
```

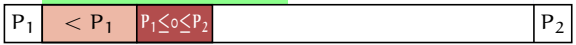
YBB Partitioning Example





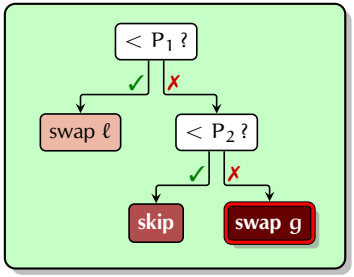


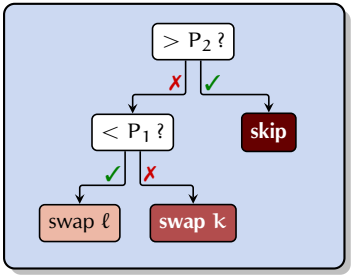
Invariant:

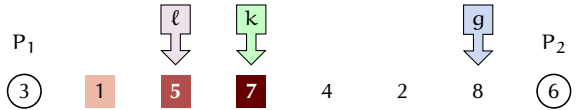



```
YbbQuicksort(A, left, right)
1   if right ⩽ left then return
2   P1 := A[left];   P2 := A[right]
3   if P1 > P2 then Swap P1 and P2 end if
4   ℓ := left + 1;   g := right − 1;   k := ℓ
5   while k ⩽ g
6       if A[k] < P1
7           Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
8       else if A[k] ⩾ P2
9           while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

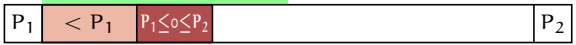
YBB Partitioning Example





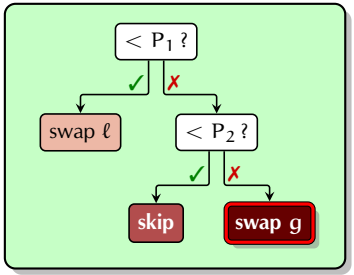


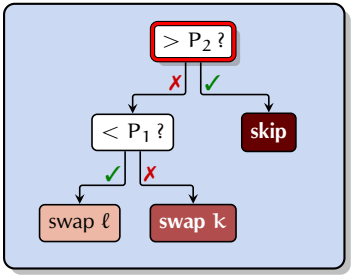
Invariant:

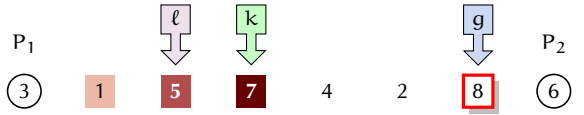



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ;  g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A,  left  , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1,  right )
```

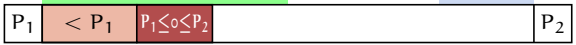
YBB Partitioning Example





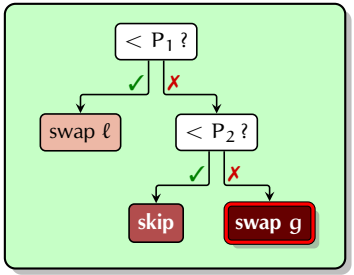


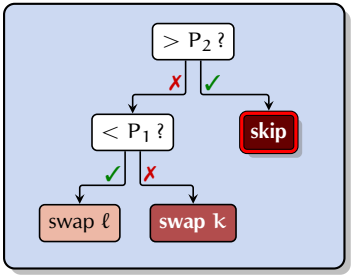
Invariant:

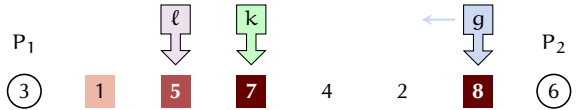



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

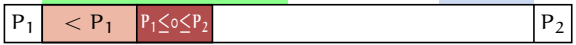
YBB Partitioning Example





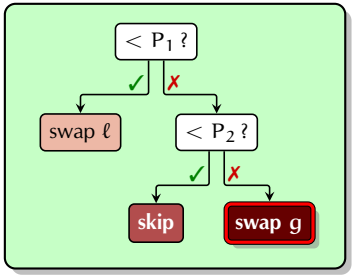


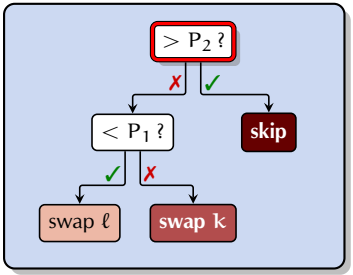
Invariant:

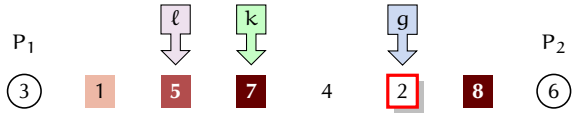


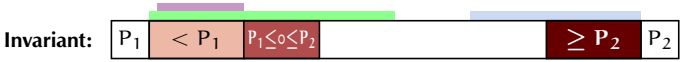

```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ;  g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A,  left  , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1,  right )
```

YBB Partitioning Example



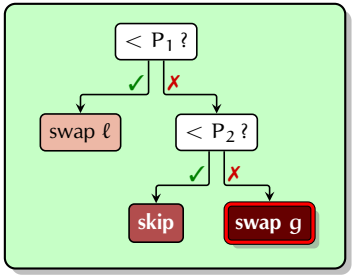


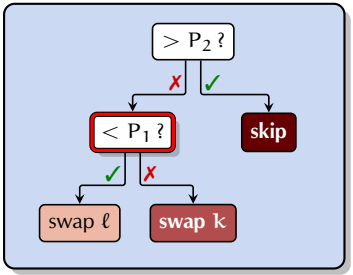


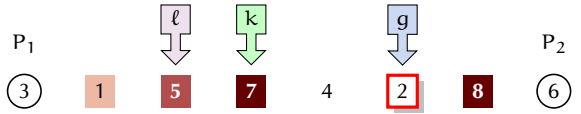



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ;  g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A,  left  , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1,  right )
```

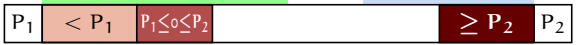
YBB Partitioning Example





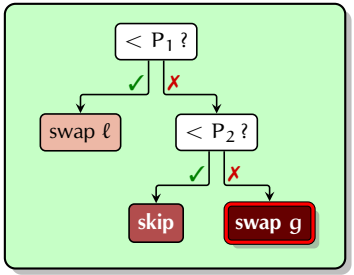


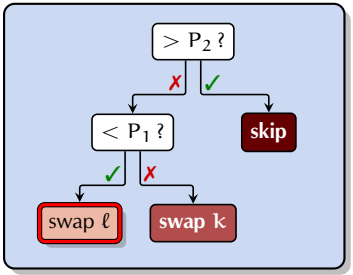
Invariant:

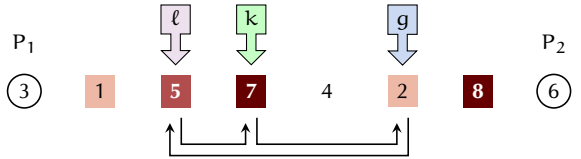



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ;  g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A,  left  , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1,  right )
```

YBB Partitioning Example

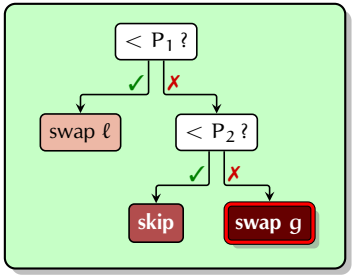


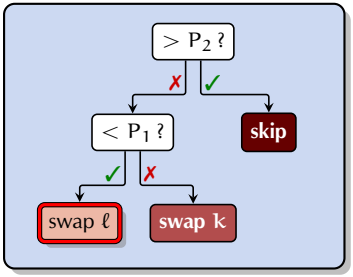


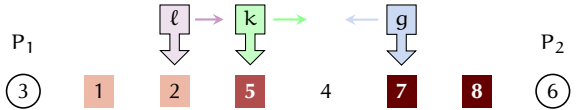



```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



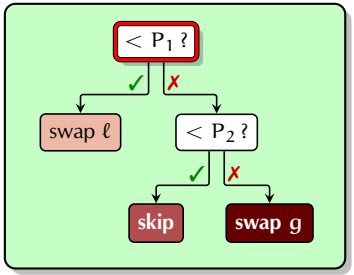


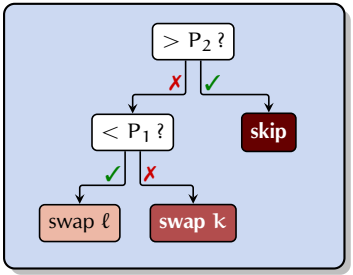


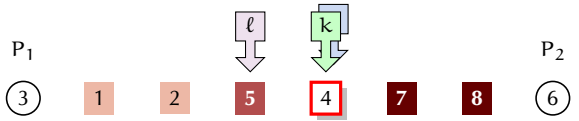
Invariant:


```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



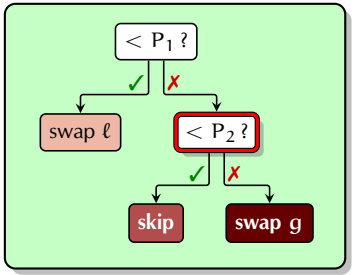


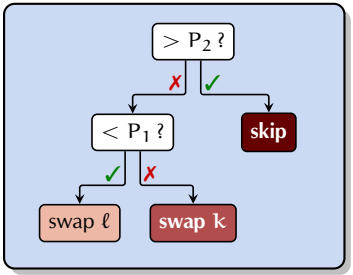


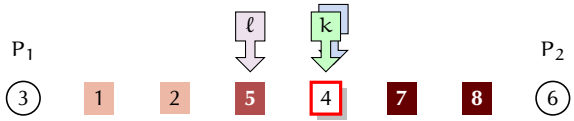
YbbQuicksort$(A, left, right)$

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A, left , ℓ − 1)
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1, right )
```

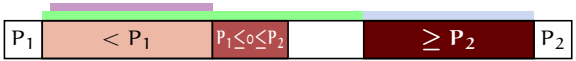
YBB Partitioning Example





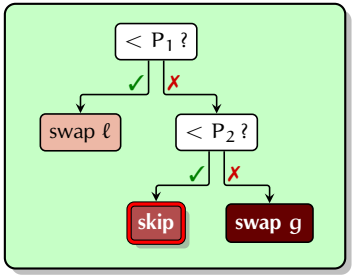


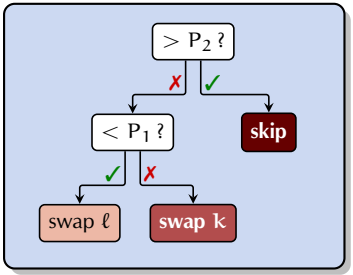
Invariant:

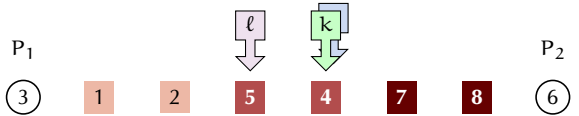


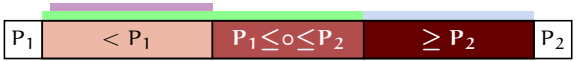

```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example



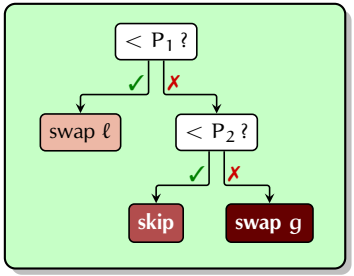


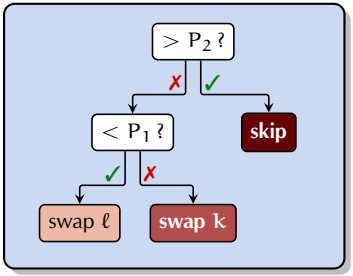





```
YbbQuicksort(A, left, right)
 1  if right ⩽ left then return
 2  P1 := A[left];   P2 := A[right]
 3  if P1 > P2 then Swap P1 and P2 end if
 4  ℓ := left + 1;   g := right − 1;   k := ℓ
 5  while k ⩽ g
 6      if A[k] < P1
 7          Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
 8      else if A[k] ⩾ P2
 9          while A[g] > P2 and k < g
10              g := g − 1
11          end while
12          Swap A[k] and A[g] ; g := g − 1
13          if A[k] < P1
14              Swap A[k] and A[ℓ] ; ℓ := ℓ + 1
15          end if
16      end if
17      k := k + 1
18  end while
19  ℓ := ℓ − 1;   g := g + 1
20  Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21  YbbQuicksort(A, left , ℓ − 1)
22  YbbQuicksort(A, ℓ + 1, g − 1)
23  YbbQuicksort(A, g + 1, right )
```

YBB Partitioning Example

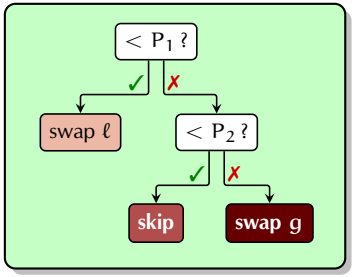


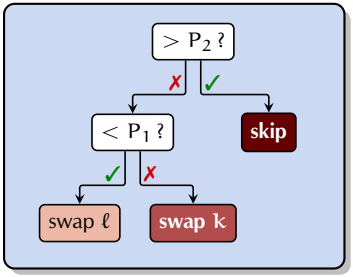


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A,  left  , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1,  right )
```

YBB Partitioning Example

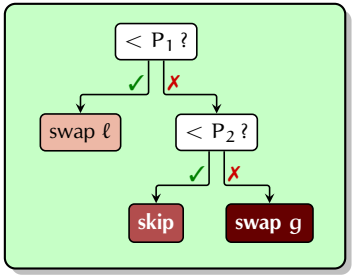


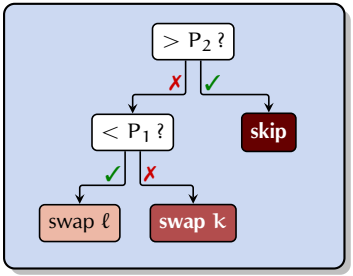


YBBQUICKSORT(A, *left*, *right*)

```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YBBQUICKSORT(A, left , ℓ − 1)
22 YBBQUICKSORT(A, ℓ + 1, g − 1)
23 YBBQUICKSORT(A, g + 1, right )
```

YBB Partitioning Example





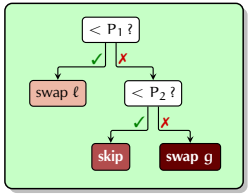
YbbQuicksort(A, *left*, *right*)

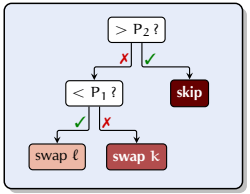
```
1  if right ⩽ left then return
2  P1 := A[left];   P2 := A[right]
3  if P1 > P2 then Swap P1 and P2 end if
4  ℓ := left + 1;   g := right − 1;   k := ℓ
5  while k ⩽ g
6      if A[k] < P1
7          Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
8      else if A[k] ⩾ P2
9          while A[g] > P2 and k < g
10             g := g − 1
11         end while
12         Swap A[k] and A[g] ;  g := g − 1
13         if A[k] < P1
14             Swap A[k] and A[ℓ] ;  ℓ := ℓ + 1
15         end if
16     end if
17     k := k + 1
18 end while
19 ℓ := ℓ − 1;   g := g + 1
20 Swap A[left] and A[ℓ] ; Swap A[right] and A[g]
21 YbbQuicksort(A,  left  , ℓ − 1)
22 YbbQuicksort(A, ℓ + 1, g − 1)
23 YbbQuicksort(A, g + 1,  right )
```

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.

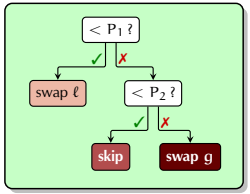


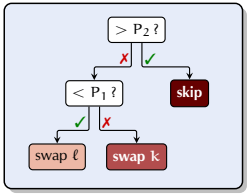


Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.

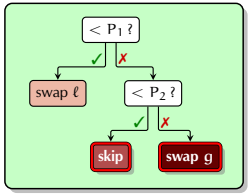


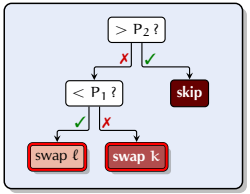


Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.

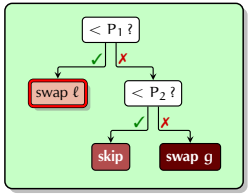


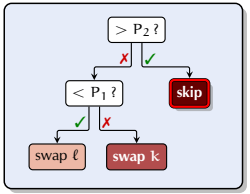


Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.

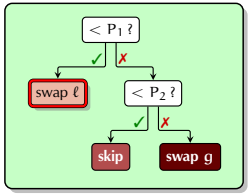


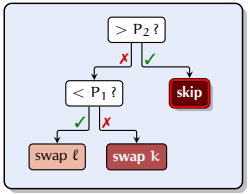


Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

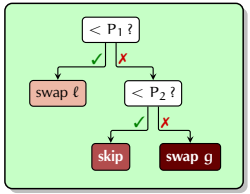


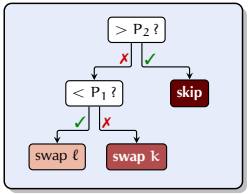


Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?



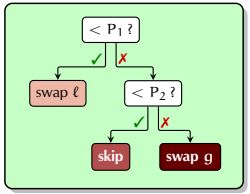


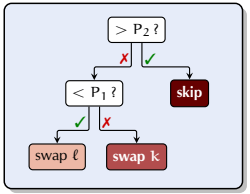
Analysis Sketch:

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





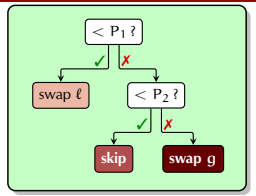
Analysis Sketch:

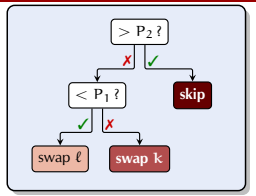
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

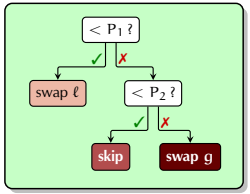
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

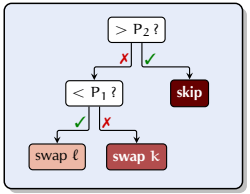
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

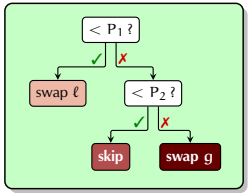
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

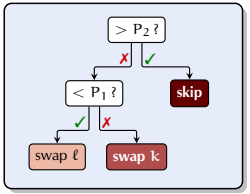
- On average, $\sim \frac{1}{3}n$ elements each kind.

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

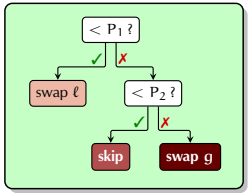
- On average, $\sim \frac{1}{3}n$ elements each kind.

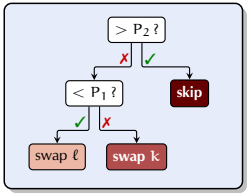
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

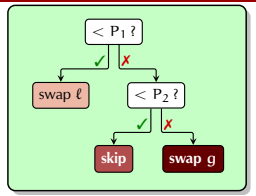
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

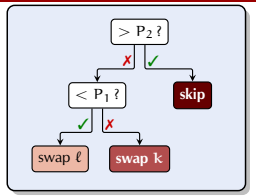
- Sounds plausible?

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?





Analysis Sketch:

- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

⇝ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

⇝ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

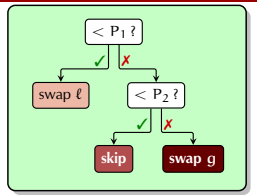
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

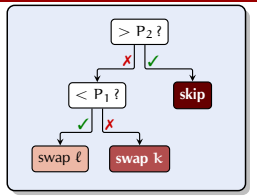
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!





Problem: There are correlations.

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

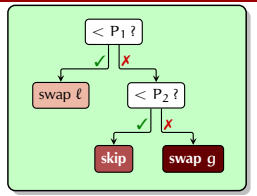
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

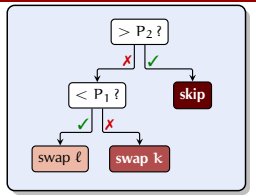
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

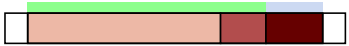
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!





Problem: There are correlations.

- 

many small elements $\rightsquigarrow$ k's range big

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

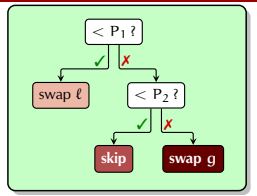
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

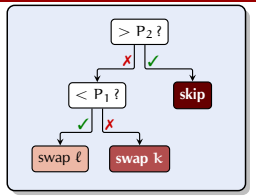
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!





Problem: There are correlations.

- 

many small elements $\rightsquigarrow$ k's range big

- 

many large elements $\rightsquigarrow$ g's range big

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

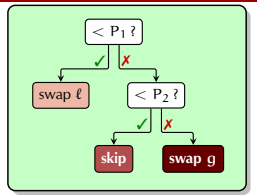
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

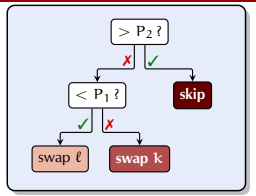
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

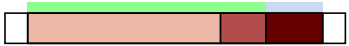
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!





Problem: There are correlations.

- 

many small elements $\rightsquigarrow$ k's range big

- 

many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow$ $> \frac{1}{3}n$ cheap elements on average.

Comparisons

How many comparisons on average?

- Consider **first** partitioning step.
- How many #cmps per element?
  Well, either 2 or 1.
- How many **on average**?

Analysis Sketch:

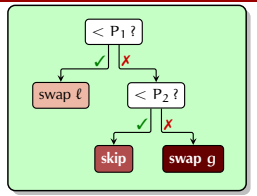
- Have to compare with one pivot first, say P_1.
  (Similar if P_2 used first.)

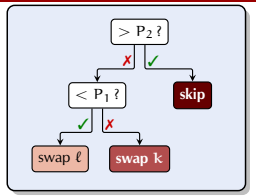
$\rightsquigarrow$ 1 cmp for small elements,
2 cmps for medium and large.

- On average, $\sim \frac{1}{3}n$ elements each kind.

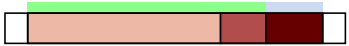
$\rightsquigarrow$ on average $2 \cdot \frac{2}{3} + 1 \cdot \frac{1}{3} = \frac{5}{3}$ cmps per element

- Sounds plausible? . . . but it's wrong!





Problem: There are correlations.

- 

  many small elements $\rightsquigarrow$ k's range big

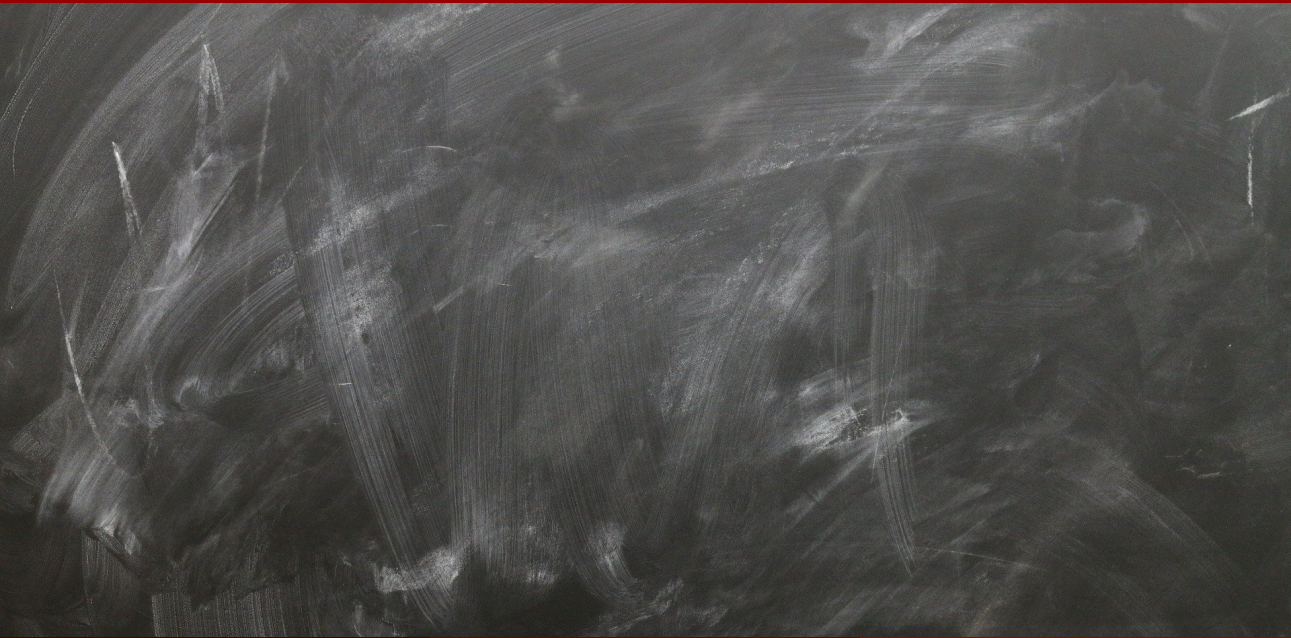
- 

  many large elements $\rightsquigarrow$ g's range big

$\rightsquigarrow$ $> \frac{1}{3}n$ cheap elements on average.

How many? . . . stay tuned.

Mathematical Analysis of Algorithms



Mathematical Analysis of Algorithms

Random Model

- π **i.i.d.** elements chosen **uniformly** in $(0,1)$

- pairwise **distinct** almost surely
- relative ranking is a **random permutation**
- $\rightsquigarrow$ equivalent to classic model

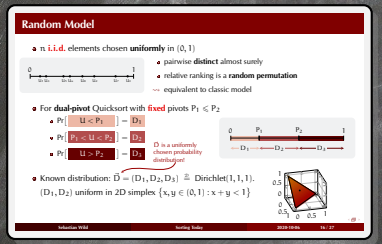
- For **dual-pivot** Quicksort with **fixed** pivots $P_1 \leqslant P_2$
  - $\Pr[\ U_i < P_1\ ] = D_1$
  - $\Pr[\ P_1 < U_i < P_2\ ] = D_2$
  - $\Pr[\ U_i > P_2\ ] = D_3$

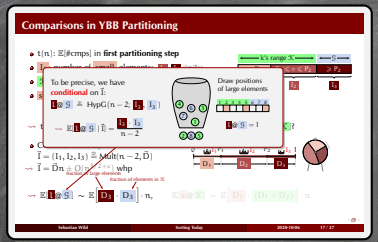
D is a uniformly chosen probability distribution!

- Known distribution: $\vec{D} = (D_1, D_2, D_3) \overset{\mathcal{D}}{=} \text{Dirichlet}(1,1,1)$.
  (D_1, D_2) uniform in 2D simplex $\{x, y \in (0,1) : x + y < 1\}$

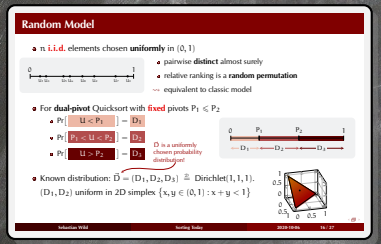
Sebastian Wild | Sorting Today | 2019-10-04 | 9c / 27

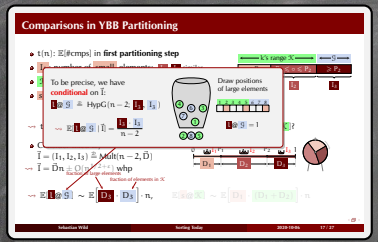
Mathematical Analysis of Algorithms

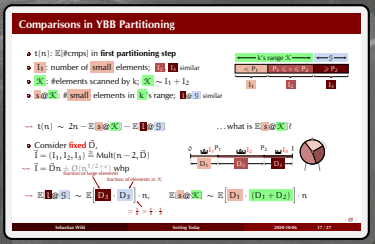




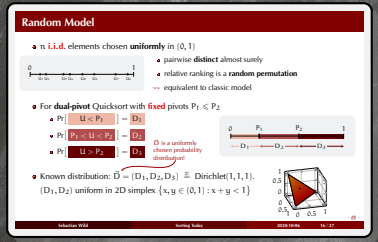
Mathematical Analysis of Algorithms

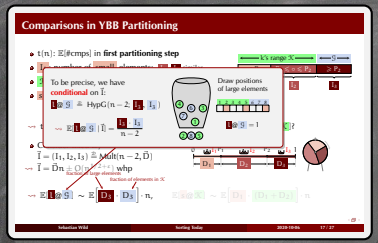


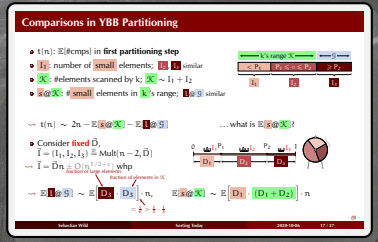


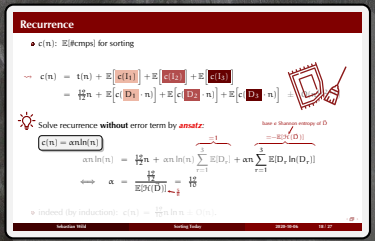


Mathematical Analysis of Algorithms

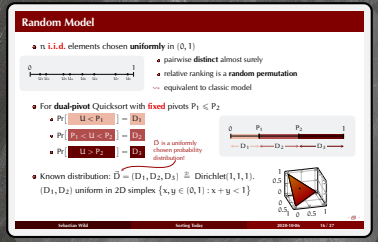


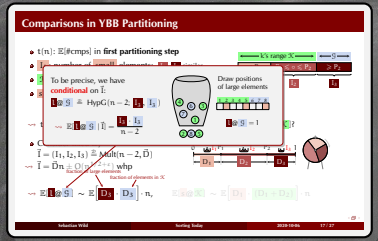


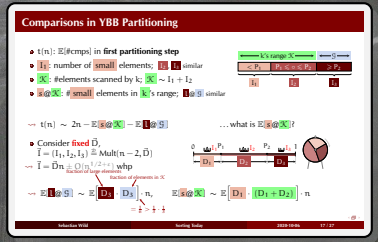


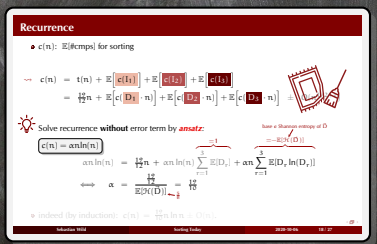


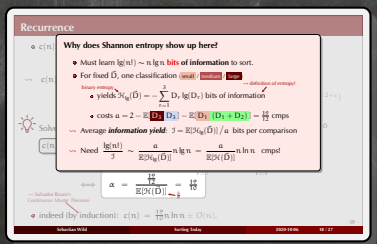
Mathematical Analysis of Algorithms



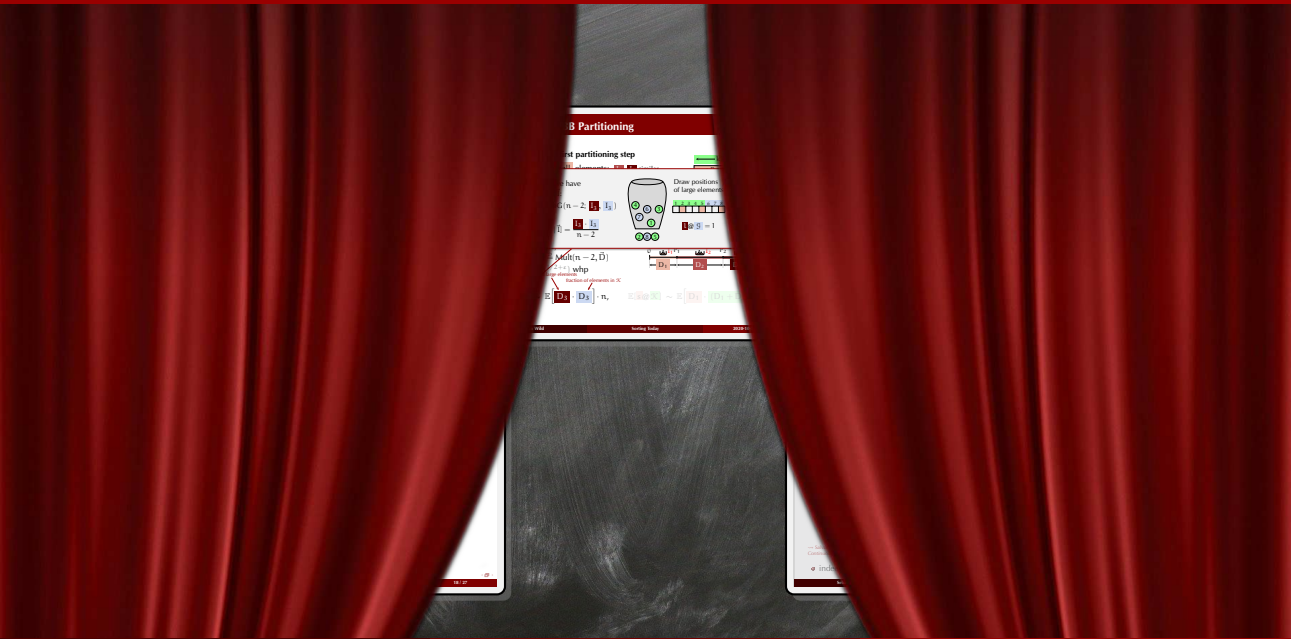




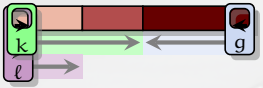
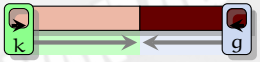
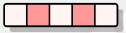




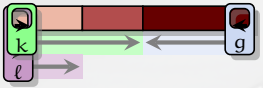
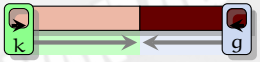
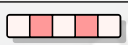
Mathematical Analysis of Algorithms



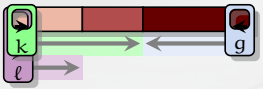
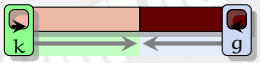
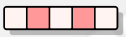
What changes with 2 pivots?

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		

What changes with 2 pivots?

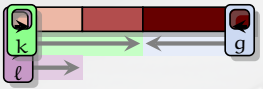
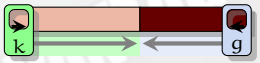
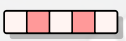
	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		
comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	−2.9%
	$\cdot n \ln n + O(n)$ *average case results*		

What changes with 2 pivots?

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		
comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	−2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	+47.1%

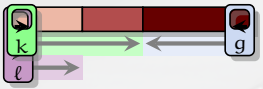
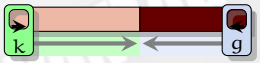
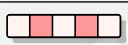
$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		
comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	−2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	+47.1%
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	+15.8%

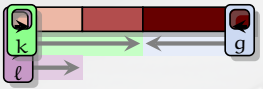
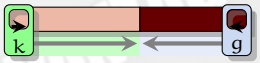
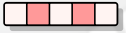
$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		
comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	−2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	+47.1%
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	+15.8%
branch mispredictions (2-bit saturating counter)	$\frac{785\pi}{532} - \frac{535}{133} \approx 0.6131$	$\frac{10\pi}{19} - \frac{20}{19} \approx 0.6008$	+2.0%

$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		
comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	−2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	+47.1%
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	+15.8%
branch mispredictions (2-bit saturating counter)	$\frac{785\pi}{532} - \frac{535}{133} \approx 0.6131$	$\frac{10\pi}{19} - \frac{20}{19} \approx 0.6008$	+2.0%
scanned elements (≈ I/Os)	$\frac{80}{57} \approx 1.4035$	$\frac{100}{57} \approx 1.7544$	−25.0%

$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

Summary:

comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	−2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	+47.1%
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	+15.8%
branch mispredictions (2-bit saturating counter)	$\frac{785\pi}{532} - \frac{535}{133} \approx 0.6131$	$\frac{10\pi}{19} - \frac{20}{19} \approx 0.6008$	+2.0%
scanned elements (≈ I/Os)	$\frac{80}{57} \approx 1.4035$	$\frac{100}{57} \approx 1.7544$	−25.0%

$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

Summary:

1. Only convincing explanation for running times: memory accesses

comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	-2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	$+47.1\%$
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	$+15.8\%$
branch mispredictions (2-bit saturating counter)	$\frac{785\pi}{532} - \frac{535}{133} \approx 0.6131$	$\frac{10\pi}{19} - \frac{20}{19} \approx 0.6008$	$+2.0\%$
scanned elements ($\approx$ I/Os)	$\frac{80}{57} \approx 1.4035$	$\frac{100}{57} \approx 1.7544$	-25.0%

$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

Summary:

1. Only convincing explanation for running times: memory accesses
2. Why was dual-pivot Quicksort not used earlier?

comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	-2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	$+47.1\%$
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	$+15.8\%$
branch mispredictions (2-bit saturating counter)	$\frac{785\pi}{532} - \frac{535}{133} \approx 0.6131$	$\frac{10\pi}{19} - \frac{20}{19} \approx 0.6008$	$+2.0\%$
scanned elements ($\approx$ I/Os)	$\frac{80}{57} \approx 1.4035$	$\frac{100}{57} \approx 1.7544$	-25.0%

$\cdot n \ln n + O(n)$ *average case results*

What changes with 2 pivots?

Summary:

1. Only convincing explanation for running times: memory accesses
2. Why was dual-pivot Quicksort not used earlier?
   Because it was **not faster** then!

comparisons	$\frac{680}{399} \approx 1.7043$	$\frac{100}{57} \approx 1.7544$	-2.9%
write accesses	$\frac{400}{399} \approx 1.0025$	$\frac{272}{399} \approx 0.6817$	$+47.1\%$
Java Bytecode instructions	$\frac{1100}{57} \approx 19.298$	$\frac{2216}{133} \approx 16.662$	$+15.8\%$
branch mispredictions (2-bit saturating counter)	$\frac{785\pi}{532} - \frac{535}{133} \approx 0.6131$	$\frac{10\pi}{19} - \frac{20}{19} \approx 0.6008$	$+2.0\%$
scanned elements ($\approx$ I/Os)	$\frac{80}{57} \approx 1.4035$	$\frac{100}{57} \approx 1.7544$	-25.0%

$\cdot n \ln n + O(n)$ *average case results*

Breaking News

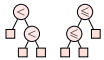
- OpenJDK 23 switched partitioning loop again (to “backwards partitioning”)!
  `https://mail.openjdk.org/pipermail/core-libs-dev/2019-June/060915.html`
- Analytical results (Annika Sabel, 2026) show **worse** comparisons and scanned elements
- better write accesses

⇝ *Java artefact or foundational algorithmic benefit?*

Multiway Quicksort Summary

Multiway Quicksort

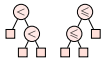
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



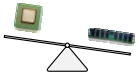
Multiway Quicksort Summary

Multiway Quicksort

1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



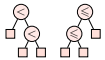
2. evidence for growing weight of memory transfer costs



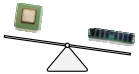
Multiway Quicksort Summary

Multiway Quicksort

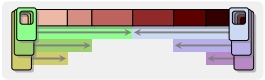
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



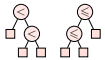
3. **Multiway Quicksort** reduces **memory transfer**



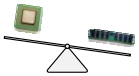
Multiway Quicksort Summary

Multiway Quicksort

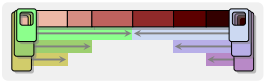
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**

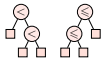


   - No other optimization of Quicksort achieves that

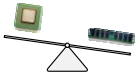
Multiway Quicksort Summary

Multiway Quicksort

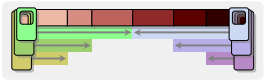
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**



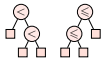
   - No other optimization of Quicksort achieves that

4. *AofA for Algorithm Science!*

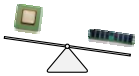
Multiway Quicksort Summary

Multiway Quicksort

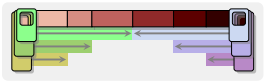
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**



   - No other optimization of Quicksort achieves that

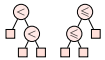
4. *AofA for Algorithm Science!*

Much more to tell . . .

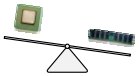
Multiway Quicksort Summary

Multiway Quicksort

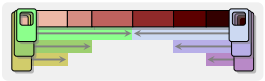
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**

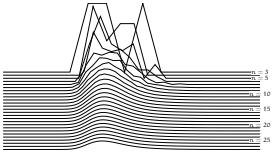


   - No other optimization of Quicksort achieves that

4. *AofA for Algorithm Science!*

Much more to tell …

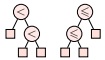
- Limiting **distributions** of costs



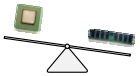
Multiway Quicksort Summary

Multiway Quicksort

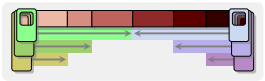
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**

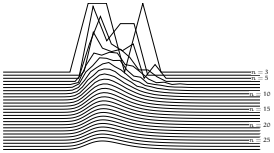


   - No other optimization of Quicksort achieves that

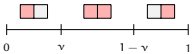
4. *AofA for Algorithm Science!*

Much more to tell …

- Limiting **distributions** of costs



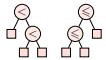
- **Sesquickselect** (cache-efficient selection)



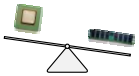
Multiway Quicksort Summary

Multiway Quicksort

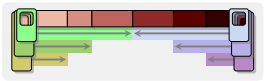
1. tricky **asymmetries** in *YBB Quicksort*
   - ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**

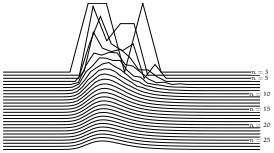


   - No other optimization of Quicksort achieves that

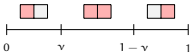
4. *AofA for Algorithm Science!*

Much more to tell …

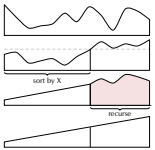
- Limiting **distributions** of costs



- **Sesquickselect** (cache-efficient selection)



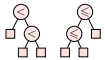
- ***QuickXsort*** inplace unstable mergesort



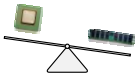
Multiway Quicksort Summary

Multiway Quicksort

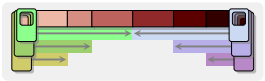
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



3. **Multiway Quicksort** reduces **memory transfer**

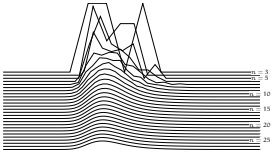


   - No other optimization of Quicksort achieves that

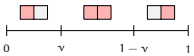
4. *AofA for Algorithm Science!*

Much more to tell …

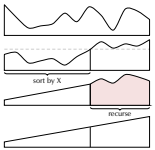
- Limiting **distributions** of costs



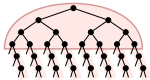
- **Sesquickselect** (cache-efficient selection)



- ***QuickXsort*** inplace unstable mergesort



- Intriguing interplay of parameters

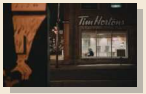


Outline

1 Context

2 Java's Sorting Methods

3 Dual-Pivot Quicksort



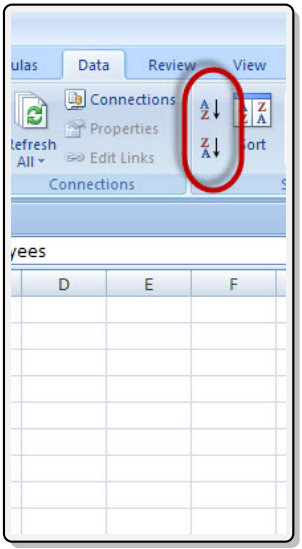
4 Timsort

5 Powersort

4 Timsort

Sorting

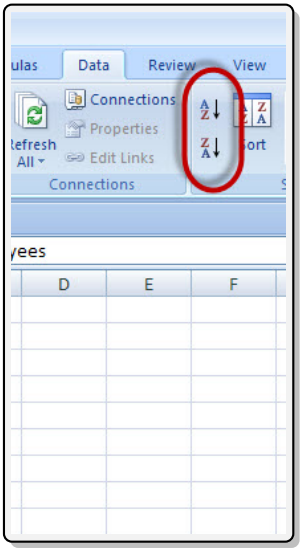
- We use sorting to
  - to make searching faster (binary search!)
  - clean data (remove dups, get canonical form of things)
  - to present data neatly for users
  - as building block in algorithms (database join, sweepline, ...)
  - ...



Sorting

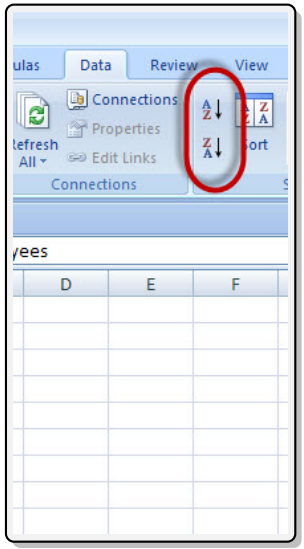
- We use sorting to
  - to make searching faster (binary search!)
  - clean data (remove dups, get canonical form of things)
  - to present data neatly for users
  - as building block in algorithms (database join, sweepline, ...)
  - ...

Sorting in Python



Sorting

- We use sorting to
  - to make searching faster (binary search!)
  - clean data (remove dups, get canonical form of things)
  - to present data neatly for users
  - as building block in algorithms (database join, sweepline, ...)
  - ...



Sorting in Python

- built-in functions: `my_list.sort()` and `sorted(my_list)`
  - `key` parameter to specify sorting criterion
  - prefer pairwise comparator function? ⇝ `functools.cmp_to_key`

CPython Sorting History

Python Version (Year)	Sorting method	Remarks	Stable?
0.9 – 1.4 1991	*qsort*	call to C library, general purpose Quicksort, [BM93]	✗

[BM93] **Bentley & McIlroy**: *Engineering a sort function*, Softw. Prac. Exp. **1993**

CPython Sorting History

Python Version (Year)		Sorting method	Remarks	Stable?
0.9 – 1.4	1991	*qsort*	call to C library, general purpose Quicksort, [BM93]	✗
1.5 – 1.6	1998	custom Quicksort	inspired by Tim Peters, “NEWSORT”	✗

[BM93]  **Bentley & McIlroy**: *Engineering a sort function*, Softw. Prac. Exp. **1993**

CPython Sorting History

Python Version (Year)		Sorting method	Remarks	Stable?
0.9 – 1.4	1991	*qsort*	call to C library, general purpose Quicksort, [BM93]	✗
1.5 – 1.6	1998	custom Quicksort	inspired by Tim Peters, “NEWSORT”	✗
2.0 – 2.2	2000	*Samplesort*	Quicksort with clever pivot choice, [FM70]	✗

[BM93] **Bentley & McIlroy**: *Engineering a sort function*, Softw. Prac. Exp. **1993**

[FM70] **Frazer & McKellar**: *Samplesort: A Sampling Approach to Minimal Storage Tree Sorting*, J. ACM **1970**

CPython Sorting History

Python Version (Year)		Sorting method	Remarks	Stable?
0.9 – 1.4	1991	*qsort*	call to C library, general purpose Quicksort, [BM93]	✗
1.5 – 1.6	1998	custom Quicksort	inspired by Tim Peters, “NEWSORT”	✗
2.0 – 2.2	2000	*Samplesort*	Quicksort with clever pivot choice, [FM70]	✗
2.3.1 – 3.10.2	2003	*Timsort*	custom mergesort by Tim, [P01] **almost 20 years unchanged** (except: caching keys, special pure-type comparisons)	✓

[BM93] **Bentley & McIlroy**: *Engineering a sort function*, Softw. Prac. Exp. **1993**

[FM70] **Frazer & McKellar**: *Samplesort: A Sampling Approach to Minimal Storage Tree Sorting*, J. ACM **1970**

[P01] **Tim Peters et al.**: *listsort.txt*, CPython sources

CPython Sorting History

Python Version (Year)		Sorting method	Remarks	Stable?
0.9 – 1.4	1991	*qsort*	call to C library, general purpose Quicksort, [BM93]	✗
1.5 – 1.6	1998	custom Quicksort	inspired by Tim Peters, "NEWSORT"	✗
2.0 – 2.2	2000	*Samplesort*	Quicksort with clever pivot choice, [FM70]	✗
2.3.1 – 3.10.2	2003	*Timsort*	custom mergesort by Tim, [P01] **almost 20 years unchanged** (except: caching keys, special pure-type comparisons)	✓
3.11.1 – ∞?	2022	***Powersort*** NEW	Timsort with Powersort merge policy [MW18]	✓

[BM93] **Bentley & McIlroy**: *Engineering a sort function*, Softw. Prac. Exp. **1993**

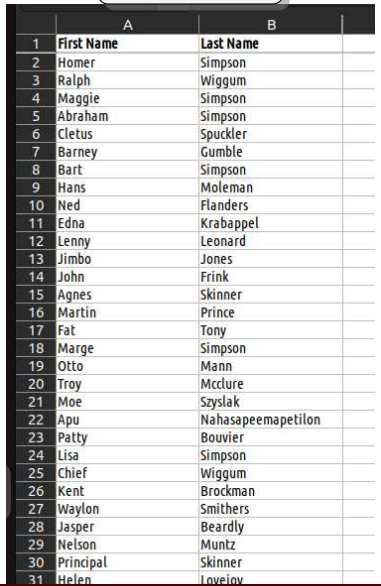
[FM70] **Frazer & McKellar**: *Samplesort: A Sampling Approach to Minimal Storage Tree Sorting*, J. ACM **1970**

[P01] **Tim Peters et al.**: *listsort.txt*, CPython sources

[MW18] **Munro & Wild**: *Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs*, ESA **2018**

Stable Sorting

unsorted input



	A	B
1	First Name	Last Name
2	Homer	Simpson
3	Ralph	Wiggum
4	Maggie	Simpson
5	Abraham	Simpson
6	Cletus	Spuckler
7	Barney	Gumble
8	Bart	Simpson
9	Hans	Moleman
10	Ned	Flanders
11	Edna	Krabappel
12	Lenny	Leonard
13	Jimbo	Jones
14	John	Frink
15	Agnes	Skinner
16	Martin	Prince
17	Fat	Tony
18	Marge	Simpson
19	Otto	Mann
20	Troy	Mcclure
21	Moe	Szyslak
22	Apu	Nahasapeemapetilon
23	Patty	Bouvier
24	Lisa	Simpson
25	Chief	Wiggum
26	Kent	Brockman
27	Waylon	Smithers
28	Jasper	Beardly
29	Nelson	Muntz
30	Principal	Skinner
31	Helen	Lovejoy

Stable Sorting

unsorted input

	A	B
1	First Name	Last Name
2	Homer	Simpson
3	Ralph	Wiggum
4	Maggie	Simpson
5	Abraham	Simpson
6	Cletus	Spuckler
7	Barney	Gumble
8	Bart	Simpson
9	Hans	Moleman
10	Ned	Flanders
11	Edna	Krabappel
12	Lenny	Leonard
13	Jimbo	Jones
14	John	Frink
15	Agnes	Skinner
16	Martin	Prince
17	Fat	Tony
18	Marge	Simpson
19	Otto	Mann
20	Troy	Mcclure
21	Moe	Szyslak
22	Apu	Nahasapeemapetilon
23	Patty	Bouvier
24	Lisa	Simpson
25	Chief	Wiggum
26	Kent	Brockman
27	Waylon	Smithers
28	Jasper	Beardly
29	Nelson	Muntz
30	Principal	Skinner
31	Helen	Lovejoy

sorted by First Name

	A	B
1	First Name	Last Name
2	Abraham	Simpson
3	Agnes	Skinner
4	Apu	Nahasapeemapetilon
5	Barney	Gumble
6	Bart	Simpson
7	Carl	Carlson
8	Charles Montgomery	Burns
9	Chief	Wiggum
10	Cletus	Spuckler
11	Edna	Krabappel
12	Fat	Tony
13	Hans	Moleman
14	Helen	Lovejoy
15	Homer	Simpson
16	Jasper	Beardly
17	Jimbo	Jones
18	John	Frink
19	Kent	Brockman
20	Lenny	Leonard
21	Lionel	Hutz
22	Lisa	Simpson
23	Maggie	Simpson
24	Marge	Simpson
25	Martin	Prince
26	Milhouse	Van Houten
27	Moe	Szyslak
28	Ned	Flanders
29	Nelson	Muntz
30	Otto	Mann
31	Patty	Bouvier

Stable Sorting

unsorted input

	First Name	Last Name
1	Homer	Simpson
2	Ralph	Wiggum
3	Maggie	Simpson
4	Abraham	Simpson
5	Cletus	Spuckler
6	Barney	Gumble
7	Bart	Simpson
8	Hans	Moleman
9	Ned	Flanders
10	Edna	Krabappel
11	Lenny	Leonard
12	Jimbo	Jones
13	John	Frink
14	Agnes	Skinner
15	Martin	Prince
16	Fat	Tony
17	Marge	Simpson
18	Otto	Mann
19	Troy	Mcclure
20	Moe	Szyslak
21	Apu	Nahasapeemapetilon
22	Patty	Bouvier
23	Lisa	Simpson
24	Chief	Wiggum
25	Kent	Brockman
26	Waylon	Smithers
27	Jasper	Beardly
28	Nelson	Muntz
29	Principal	Skinner
30	Helen	Lovejoy

sorted by First Name

	First Name	Last Name
1	Abraham	Simpson
2	Agnes	Skinner
3	Apu	Nahasapeemapetilon
4	Barney	Gumble
5	Bart	Simpson
6	Carl	Carlson
7	Charles Montgomery	Burns
8	Chief	Wiggum
9	Cletus	Spuckler
10	Edna	Krabappel
11	Fat	Tony
12	Hans	Moleman
13	Helen	Lovejoy
14	Homer	Simpson
15	Jasper	Beardly
16	Jimbo	Jones
17	John	Frink
18	Kent	Brockman
19	Lenny	Leonard
20	Lionel	Hutz
21	Lisa	Simpson
22	Maggie	Simpson
23	Marge	Simpson
24	Martin	Prince
25	Milhouse	Van Houten
26	Moe	Szyslak
27	Ned	Flanders
28	Nelson	Muntz
29	Otto	Mann
30	Patty	Bouvier

sorted by Last Name

	First Name	Last Name
1	Jasper	Beardly
2	Patty	Bouvier
3	Selma	Bouvier
4	Kent	Brockman
5	Charles Montgomery	Burns
6	Carl	Carlson
7	Ned	Flanders
8	John	Frink
9	Barney	Gumble
10	Lionel	Hutz
11	Snake	Jailbird
12	Jimbo	Jones
13	Edna	Krabappel
14	Lenny	Leonard
15	Helen	Lovejoy
16	Otto	Mann
17	Troy	Mcclure
18	Hans	Moleman
19	Nelson	Muntz
20	Apu	Nahasapeemapetilon
21	Martin	Prince
22	Abraham	Simpson
23	Bart	Simpson
24	Homer	Simpson
25	Lisa	Simpson
26	Maggie	Simpson
27	Marge	Simpson
28	Agnes	Skinner
29	Principal	Skinner
30	Waylon	Smithers

Sorting Trade-offs

- *Quicksort* is generally **fast** and **in place**, but not stable

Sorting Trade-offs

- *Quicksort* is generally **fast** and **in place**, but not stable
- *Mergesort* is generally (quite) **fast** and **stable**, but not in place

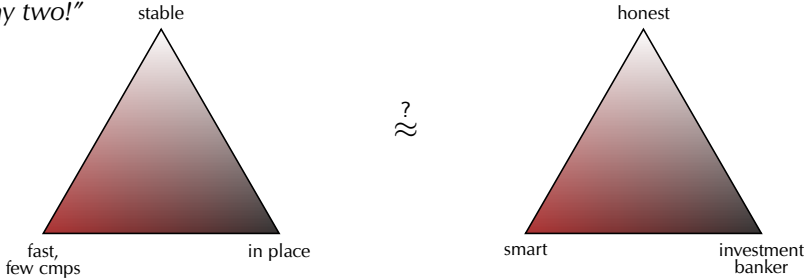
Sorting Trade-offs

- *Quicksort* is generally **fast** and **in place**, but not stable
- *Mergesort* is generally (quite) **fast** and **stable**, but not in place
- stable and in place is tricky (possible, but relatively slow)

Sorting Trade-offs

- *Quicksort* is generally **fast** and **in place**, but not stable
- *Mergesort* is generally (quite) **fast** and **stable**, but not in place
- stable and in place is tricky (possible, but relatively slow)

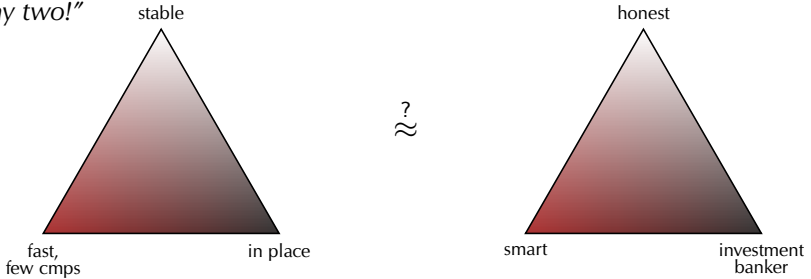
"Pick any two!"



Sorting Trade-offs

- *Quicksort* is generally **fast** and **in place**, but not stable
- *Mergesort* is generally (quite) **fast** and **stable**, but not in place
- stable and in place is tricky (possible, but relatively slow)

"Pick any two!"



- Python has lots of objects and pointers anyways … "in-place" property least painful to sacrifice

⇝ Mergesort!

Timsort = Adaptive Mergesort++

Various modifications

1. detect existing **runs**
   run = any (weakly) increasing or
   (strictly) decreasing range
2. **merge policy** decides when to merge which runs
3. keeping runs on a **fixed-size** stack
4. **galloping** merges
   use exponential searches
   to find position in other run

5. **minimum run lengths** with binary insertion sort
   choose $32 \leqslant \texttt{minrun} \leqslant 64$, so that $\lceil \frac{n}{\texttt{minrun}} \rceil$ is 2^k or slightly smaller
   fill runs to `minrun` elements

Tim Peters et al.: *listsort.txt,* CPython sources

Timsort merge policy (original)

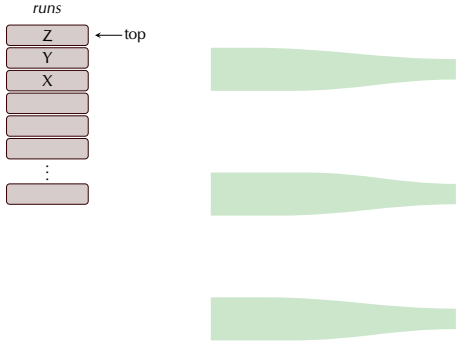
```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.append((i,j-i)); i = j
        while Rule A/B/C applicable
            merge X,Y resp. Y,Z
    while len(runs) > 1:
        merge Y,Z
```

full code:
`tiny.cc/timsort`

Timsort merge policy (original)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.append((i,j-i)); i = j
        while Rule A/B/C applicable
            merge X,Y resp. Y,Z
    while len(runs) > 1:
        merge Y,Z
```

full code:
`tiny.cc/timsort`



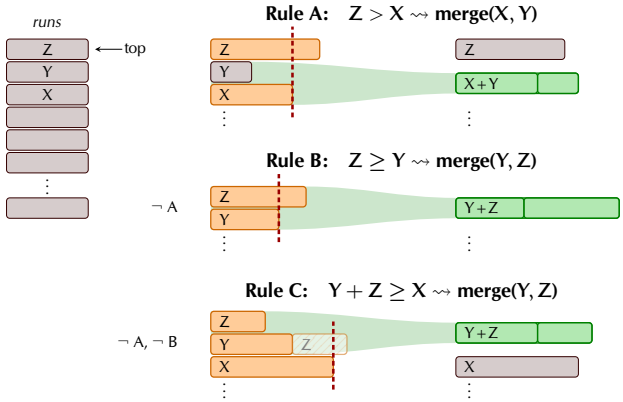
- `extend_run` detects next run
  & boosts to `minrun` if necessary

Timsort merge policy (original)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.append((i,j-i)); i = j
        while Rule A/B/C applicable
            merge X,Y resp. Y,Z
    while len(runs) > 1:
        merge Y,Z
```

full code:
`tiny.cc/timsort`

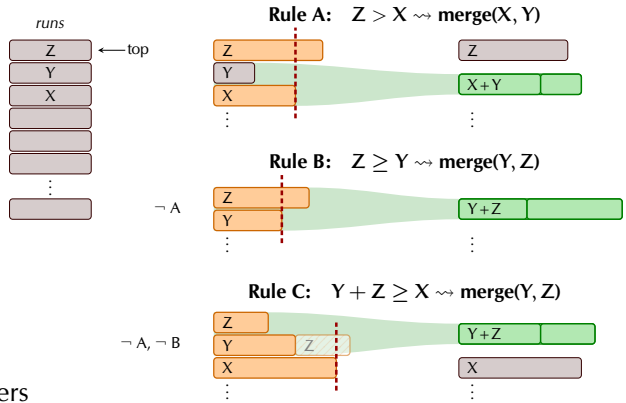
- `extend_run` detects next run & boosts to `minrun` if necessary



Timsort merge policy (original)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.append((i,j-i)); i = j
        while Rule A/B/C applicable
            merge X,Y resp. Y,Z
    while len(runs) > 1:
        merge Y,Z
```

full code:
`tiny.cc/timsort`



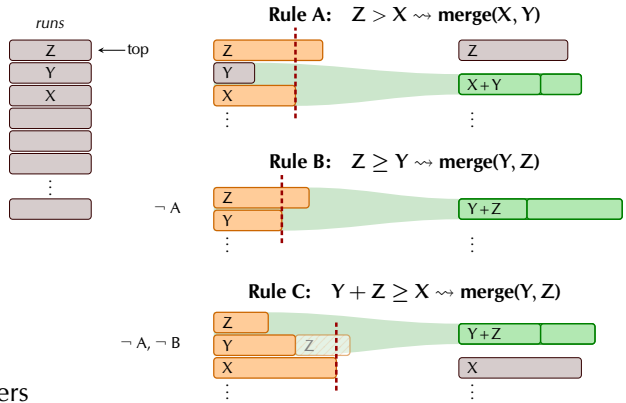
- `extend_run` detects next run & boosts to `minrun` if necessary
- **Goal:** runs on stack grow like Fibonacci numbers
  - Invariant: $\forall j:$ `runs[j]` $\geqslant$ `runs[j+1]` $+$ `runs[j+2]`

  $\rightsquigarrow$ max stack height $\approx \log_\phi(n)$

Timsort merge policy (original)

```
1 def timsort(lst):
2     i = 0; runs = []
3     while i < len(lst):
4         j = extend_run(lst, i)
5         runs.append((i,j-i)); i = j
6         while Rule A/B/C applicable
7             merge X,Y resp. Y,Z
8     while len(runs) > 1:
9         merge Y,Z
```

full code:
`tiny.cc/timsort`



- `extend_run` detects next run & boosts to `minrun` if necessary
- **Goal:** runs on stack grow like Fibonacci numbers
  - Invariant: $\forall j:$ `runs[j]` $\geqslant$ `runs[j+1]` $+$ `runs[j+2]`

  $\rightsquigarrow$ max stack height $\approx \log_\phi(n)$
- Why exactly these rules?
  - Tim Peters: *"first thing I tried that 'worked well'"*

(a) small runs stack,
(b) balanced on equal runs,
(c) $O(n \log n)$ worst case*

Invariant trouble

Recall this?

- **Goal:** runs on stack grow like Fibonacci:

  Invariant: $\forall \mathtt{j} = 0, \ldots, \mathtt{len(runs)} - 3 :$
  $\mathtt{runs}[\mathtt{j}] \geqslant \mathtt{runs}[\mathtt{j}+1] + \mathtt{runs}[\mathtt{j}+2]$

Invariant trouble

Recall this?

- **Goal:** runs on stack grow like Fibonacci:

  Invariant: $\forall \mathtt{j} = 0, \ldots, \mathtt{len(runs)} - 3 :$
  $\mathtt{runs}[\mathtt{j}] \geqslant \mathtt{runs}[\mathtt{j}+1] + \mathtt{runs}[\mathtt{j}+2]$

not true(!) for naughty pattern of run lengths

Invariant trouble

Recall this?

- **Goal:** runs on stack grow like Fibonacci:

  Invariant: $\forall j = 0, \ldots, \texttt{len(runs)} - 3 :$
  $\texttt{runs[j]} \geqslant \texttt{runs[j+1]} + \texttt{runs[j+2]}$

not true(!) for naughty pattern of run lengths

KeY project for formal verification in Java

> Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)
>
> February 24, 2015 Envisage Written by Stijn de Gouw. $s

de Gouw, de Boer, Bubel, Hähnle, Rot, Steinhöfel: *Verifying OpenJDK's Sort Method for Generic Collections,* J Autom Reasoning **2019**

Invariant trouble

Recall this?

- **Goal:** runs on stack grow like Fibonacci:

  Invariant: $\forall j = 0, \ldots, \texttt{len(runs)} - 3 :$
  $\texttt{runs[j]} \geqslant \texttt{runs[j+1]} + \texttt{runs[j+2]}$

not true(!) for naughty pattern of run lengths

KeY project for formal verification in Java

> Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)
>
> February 24, 2015 Envisage Written by Stijn de Gouw. $s

de Gouw, de Boer, Bubel, Hähnle, Rot, Steinhöfel: *Verifying OpenJDK's Sort Method for Generic Collections*, J Autom Reasoning **2019**

- In Java: `Arrays.sort(Object[])`
  could throw `ArrayIndexOutOfBoundException`
  for specific input of size 67,108,864
  - Timsort in Java since 2009, but issue never reported?
  - first (incorrectly!) patched in 2013, then a second time in 2015 ...
- for CPython: patched in 2015
  - mostly a theoretical issue (needs $\geqslant 2^{49}$ elements $\approx 2\,\text{PB}$ ...)

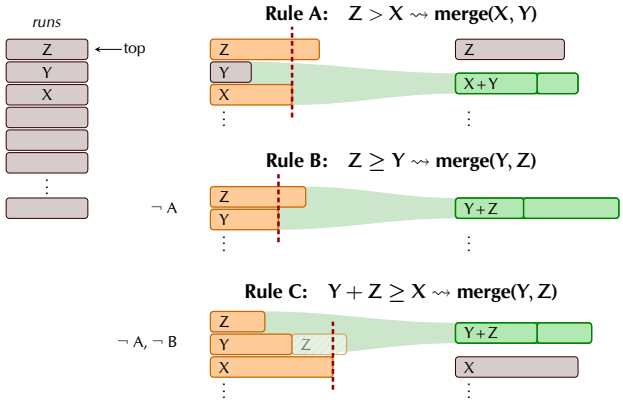
Timsort merge policy (patched)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.apppend((i,j)); i = j
        while Rule A/B/C/D applicable
            merge corresponding runs
    while len(runs) > 1:
        merge topmost 2 runs
```

Timsort merge policy (patched)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.apppend((i,j)); i = j
        while Rule A/B/C/D applicable
            merge corresponding runs
    while len(runs) > 1:
        merge topmost 2 runs
```

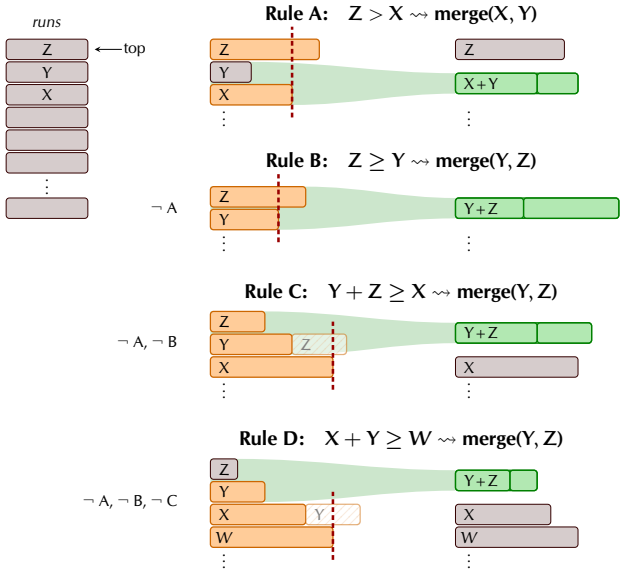
- Need to add a **Rule D**



Timsort merge policy (patched)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.apppend((i,j)); i = j
        while Rule A/B/C/D applicable
            merge corresponding runs
    while len(runs) > 1:
        merge topmost 2 runs
```

- Need to add a **Rule D**

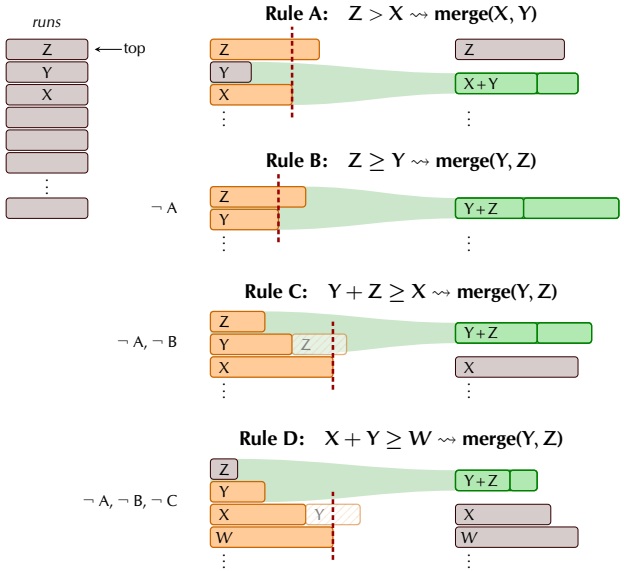


Timsort merge policy (patched)

```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.apppend((i,j)); i = j
        while Rule A/B/C/D applicable
            merge corresponding runs
    while len(runs) > 1:
        merge topmost 2 runs
```

- Need to add a **Rule D**

⇝ **Invariant:** $\forall j = 0, \ldots, \texttt{len(runs)} - 3 :$ $\texttt{runs[j]} \geqslant \texttt{runs[j+1]} + \texttt{runs[j+2]}$ ✓



Timsort merge policy (patched)

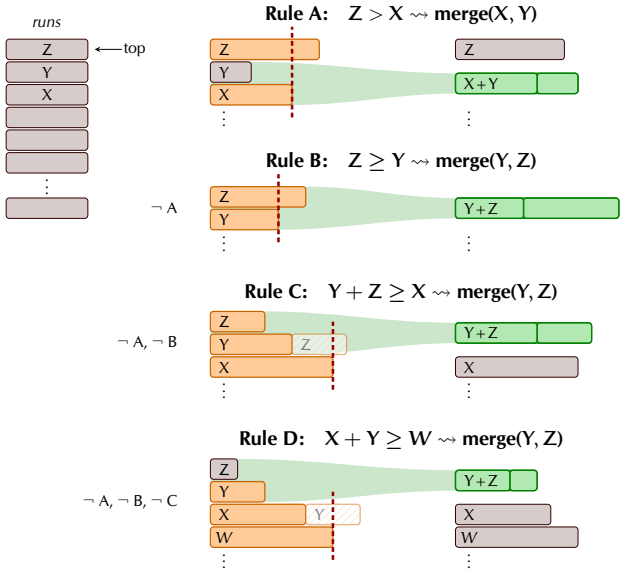
```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.apppend((i,j)); i = j
        while Rule A/B/C/D applicable
            merge corresponding runs
    while len(runs) > 1:
        merge topmost 2 runs
```

- Need to add a **Rule D**

⇝ **Invariant:** $\forall j = 0, \ldots, \texttt{len(runs)} - 3 :$ $\texttt{runs[j]} \geqslant \texttt{runs[j+1]} + \texttt{runs[j+2]}$ ✓

- running time indeed $O(n \log n)$ … *very* complicated to prove

Auger, Jugé, Nicaud, Pivoteau: *On the Worst-Case Complexity of TimSort,* ESA **2018**

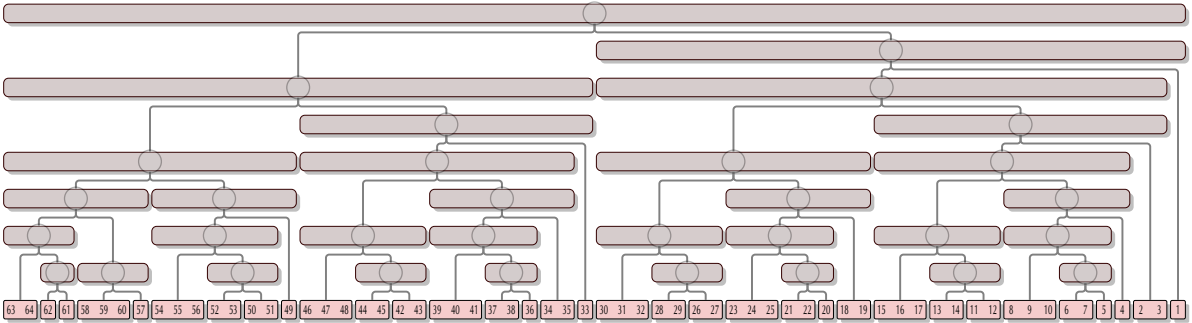


Timsort bad case

- Timsort's merge policy mostly works OK

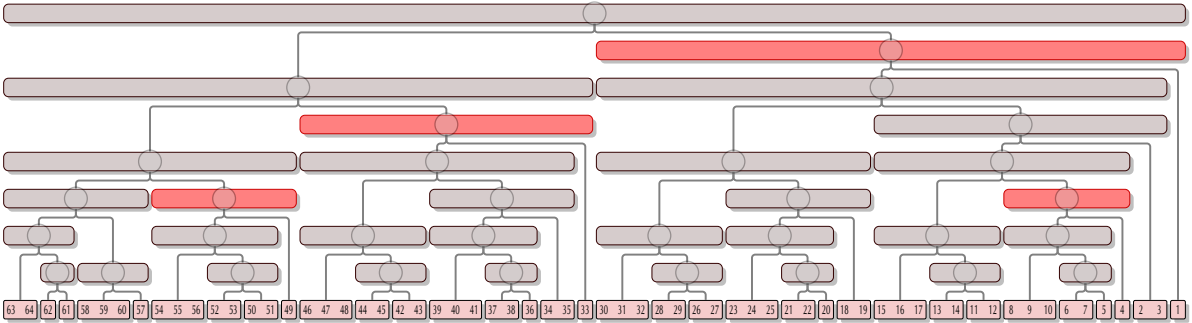
Timsort bad case

- Timsort's merge policy mostly works OK
- . . . but does stupid things on certain inputs:



Timsort bad case

- Timsort's merge policy mostly works OK
- . . . but does stupid things on certain inputs:

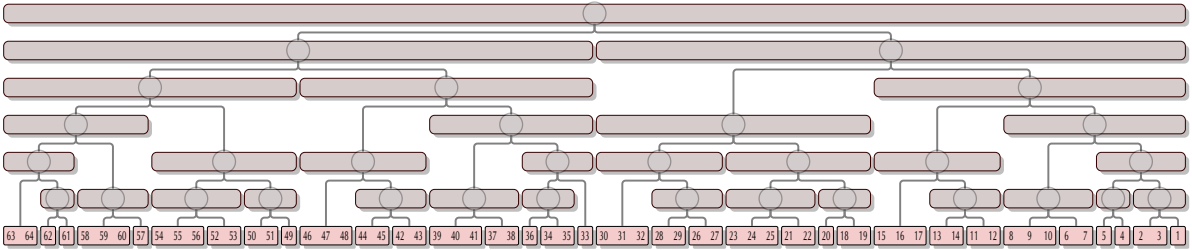


- intuitive problem: regularly very unbalanced merges

Buss, Knop: *Strategies for Stable Merge Sorting*, SODA **2019**

Timsort bad case

- Timsort's merge policy mostly works OK
- . . . but does stupid things on certain inputs:

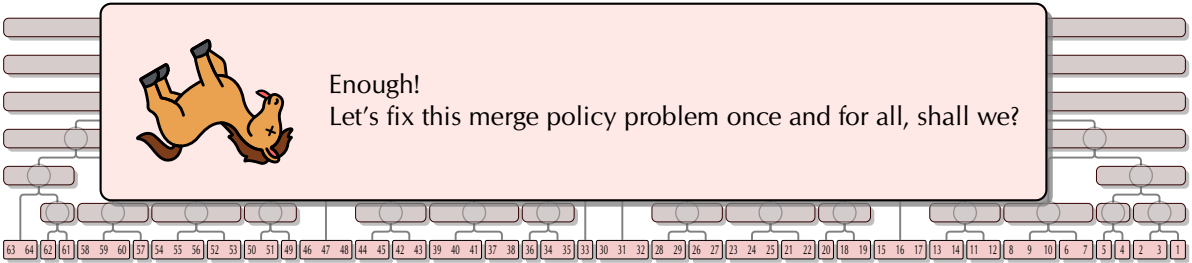


- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

Buss, Knop: *Strategies for Stable Merge Sorting*, SODA **2019**

Timsort bad case

- Timsort's merge policy mostly works OK
- . . . but does stupid things on certain inputs:



- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

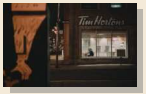
Buss, Knop: *Strategies for Stable Merge Sorting*, SODA **2019**

Outline

1 Context

2 Java's Sorting Methods

3 Dual-Pivot Quicksort



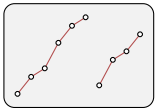
4 Timsort

5 Powersort


POWER
sort
5
Powersort

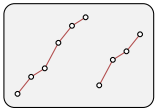
Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

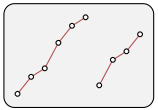
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

Conceptually two steps:

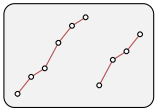
1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

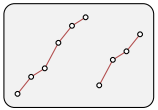
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

15 17 12 19 2 9 13 7 11 1 4 8 10 14 23 5 21 3 6 16 18 20 22

Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

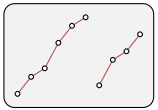
determined by **merge policy**

- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

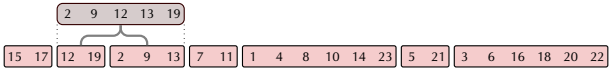
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order***.

determined by **merge policy**

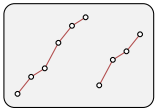
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

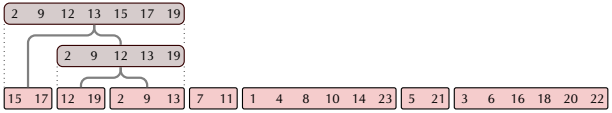
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

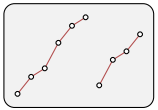
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

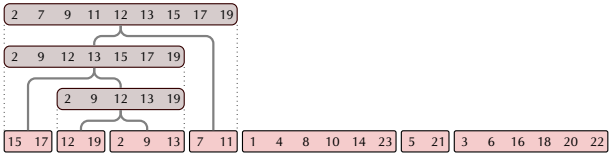
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

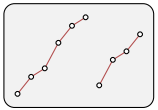
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

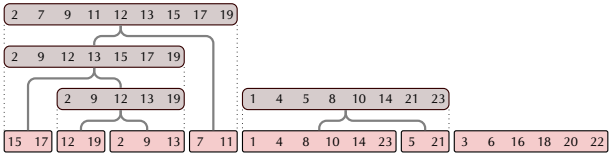
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

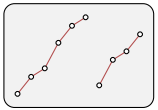
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

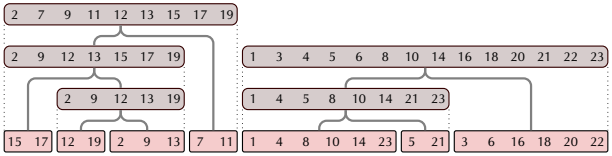
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

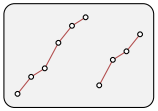
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

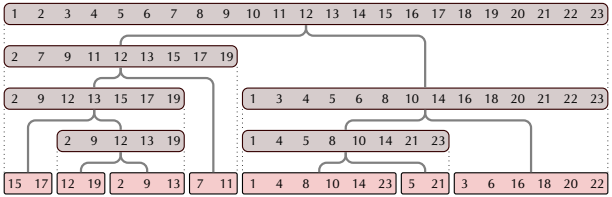
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

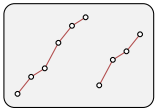
- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

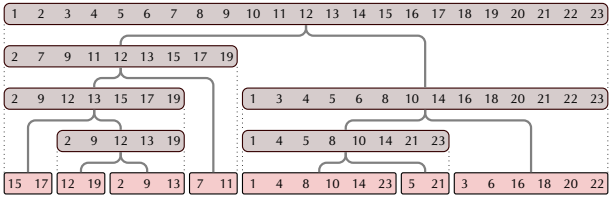
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

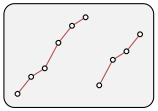
⇝ Merge order = **merge tree**:



⇝ **Merge policy** = algorithm to choose merge tree

Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

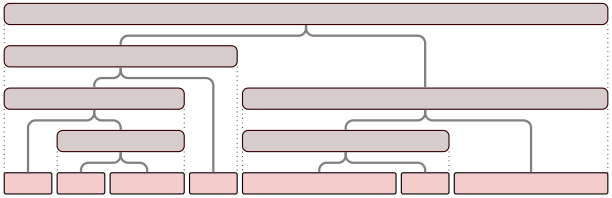
Conceptually two steps:

1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

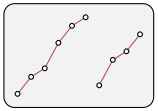


⇝ **Merge policy** = algorithm to choose merge tree

- **Merge cost** of one merge := size of output
  - ≈ memory transfers
  - ⩾ # comparisons
- total cost = total area of 

Merge policies from first principles

Goal: ***Simple*** *stable sorting method that optimally exploits* ***sorted runs***



Run-adaptive mergesort

interleaved in code

Conceptually two steps:

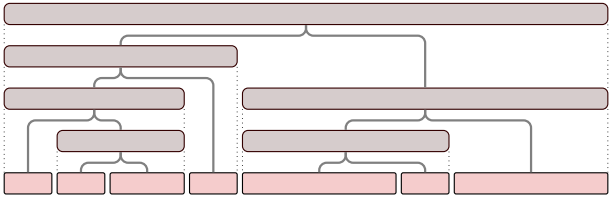
1. *Find runs in input.*
2. *Merge them* ***in some order****.*

determined by **merge policy**

- for stable sort
  - ⇝ *merge 2* ***adjacent*** *runs*

Find good tree with little overhead?

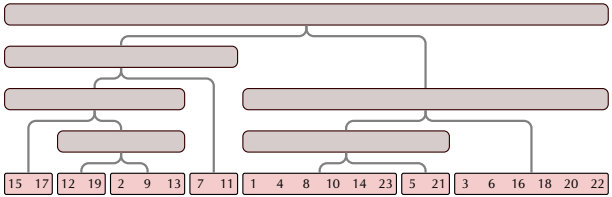
⇝ Merge order = **merge tree**:



⇝ **Merge policy** = algorithm to choose merge tree

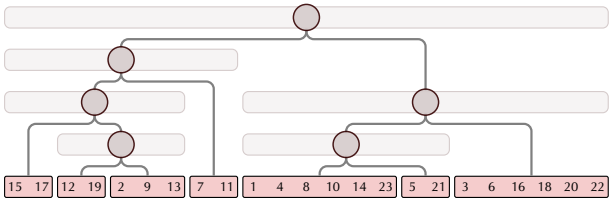
- **Merge cost** of one merge := size of output
  - ≈ memory transfers
  - ⩾ # comparisons
- total cost = total area of 

Mergesort meets Binary Search Trees



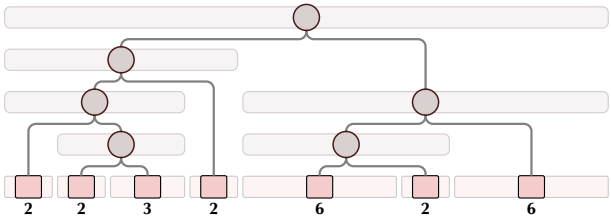
Merge cost = total area of 

Mergesort meets Binary Search Trees



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

Mergesort meets Binary Search Trees



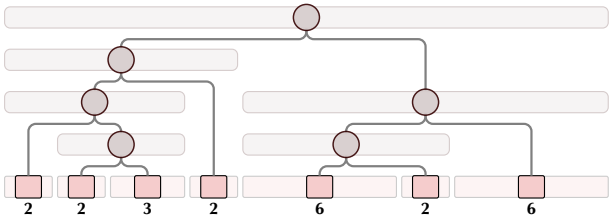
Merge cost $=$ total area of 

$=$ total length of paths to all array entries

$=$ weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

Mergesort meets Binary Search Trees

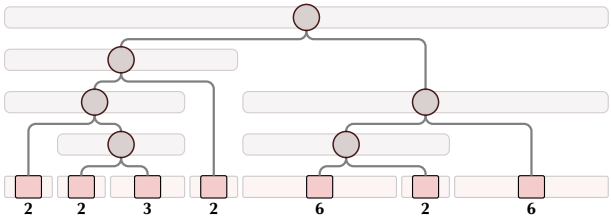


Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

Mergesort meets Binary Search Trees



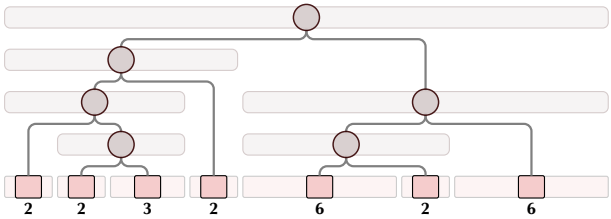
Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

Mergesort meets Binary Search Trees



How to compute good merge trees?

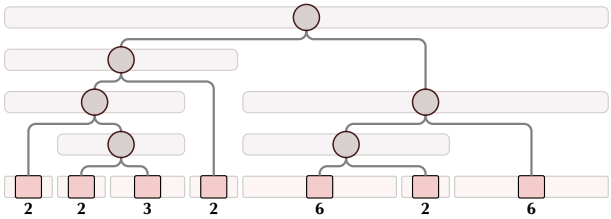
Merge cost = total area of
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

Mergesort meets Binary Search Trees



Merge cost = total area of
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

⇝ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \dots, L_r$ (run lengths)

⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

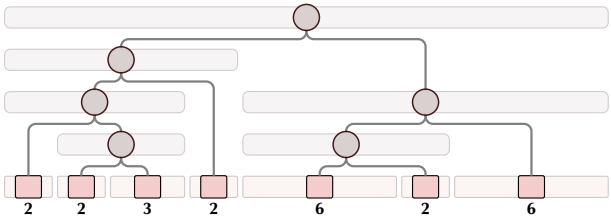
How to compute good merge trees?

. . . 70s are calling

nearly-optimal BST merge

- simple (greedy) linear-time methods!

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

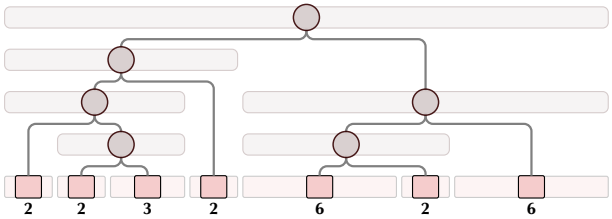
How to compute good merge trees?

. . . 70s are calling

nearly-optimal BST merge

- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)

Mergesort meets Binary Search Trees



Merge cost = total area of
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \dots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H} n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge trees?

. . . 70s are calling

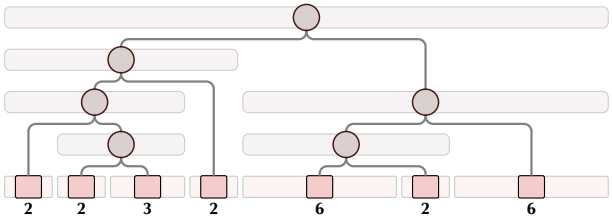
nearly-optimal BST merge

- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)

$\rightsquigarrow$ Powersort based on the **bisection method**

Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

⇝ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \dots, L_r$ (run lengths)

⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge trees?

. . . 70s are calling

nearly-optimal BST merge

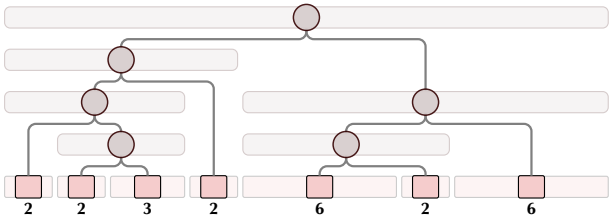
- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)

⇝ Powersort based on the **bisection method**

Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

- **Intuition:**
  "Round" to *perfect* binary tree
  - Find run boundary closest to middle
  - Recurse on both sides

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

⇝ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge trees?

. . . 70s are calling

nearly-optimal BST merge

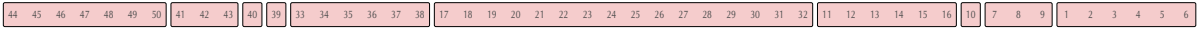
- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)

⇝ Powersort based on the **bisection method**

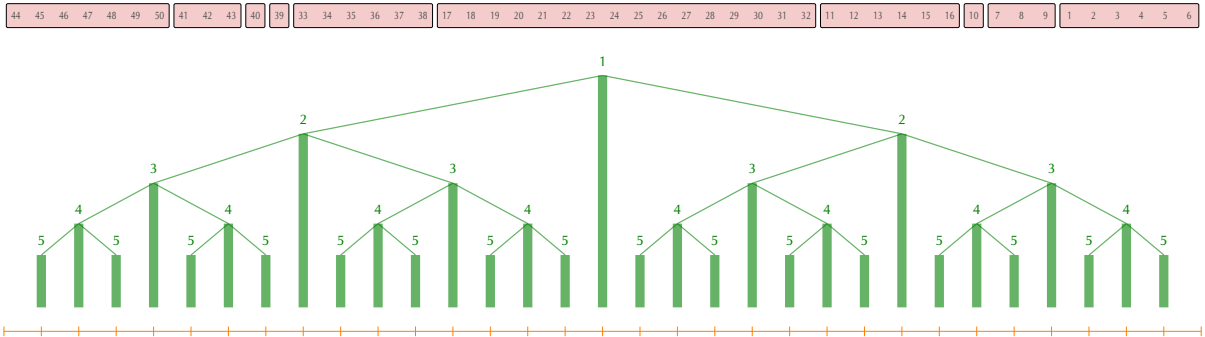
Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

- **Intuition:**
  "Round" to *perfect* binary tree
  - Find run boundary closest to middle
  - Recurse on both sides
- **But:** Can't use recursion!

Run-Boundary Powers

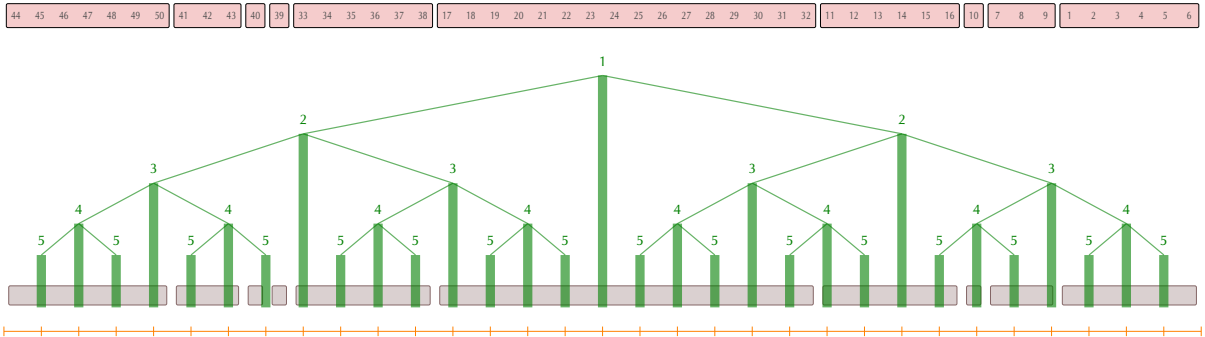


Run-Boundary Powers



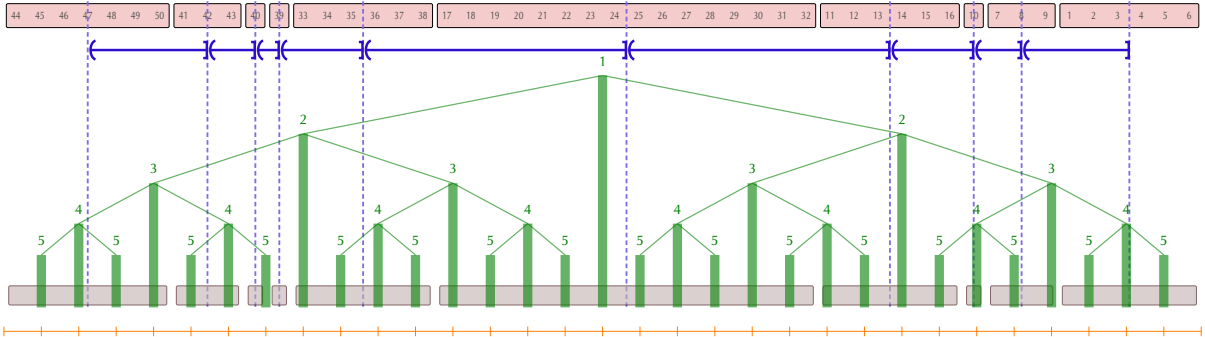
- (virtual) perfect balanced binary tree

Run-Boundary Powers



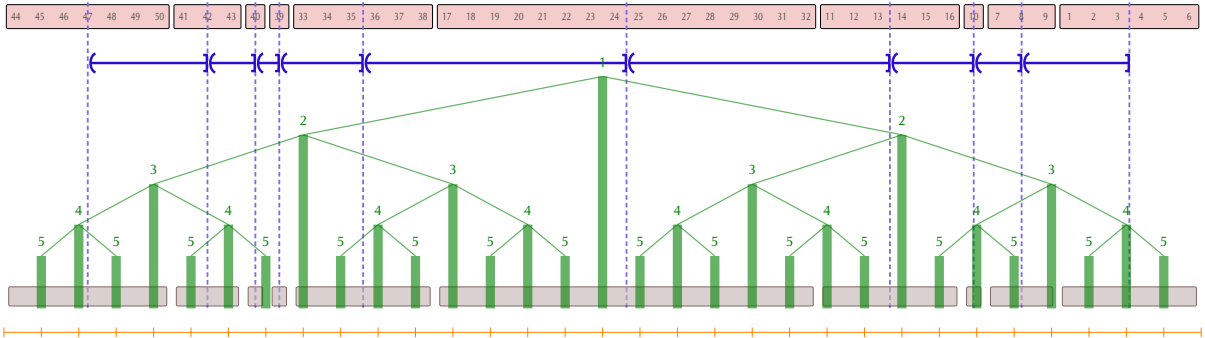
- (virtual) perfect balanced binary tree

Run-Boundary Powers



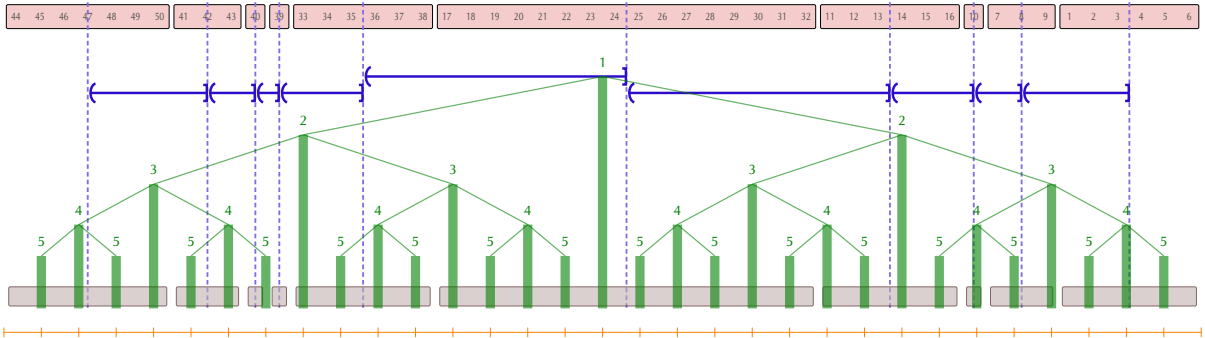
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



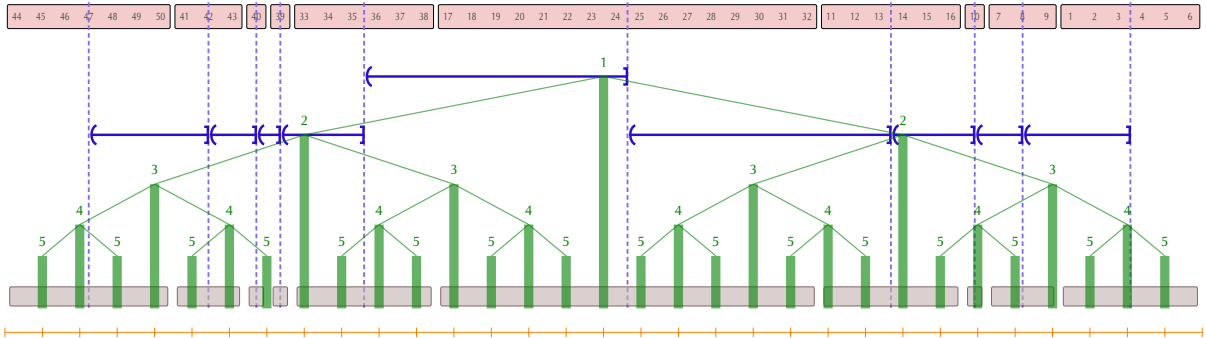
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



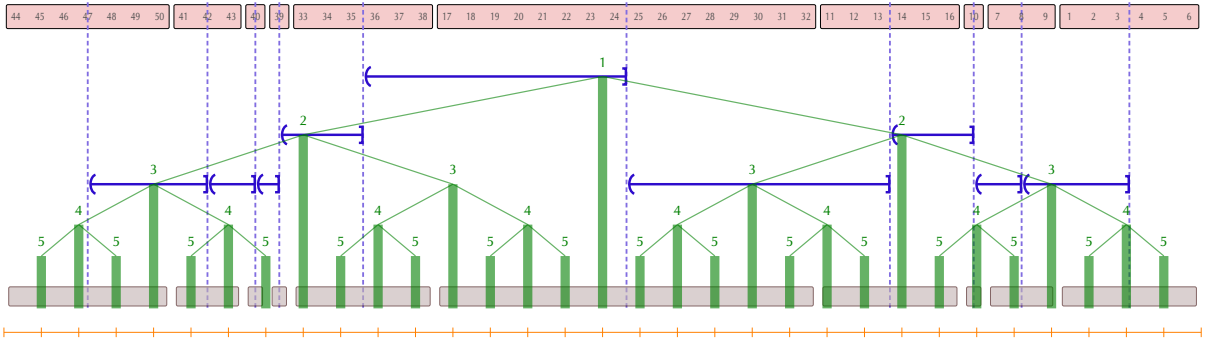
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



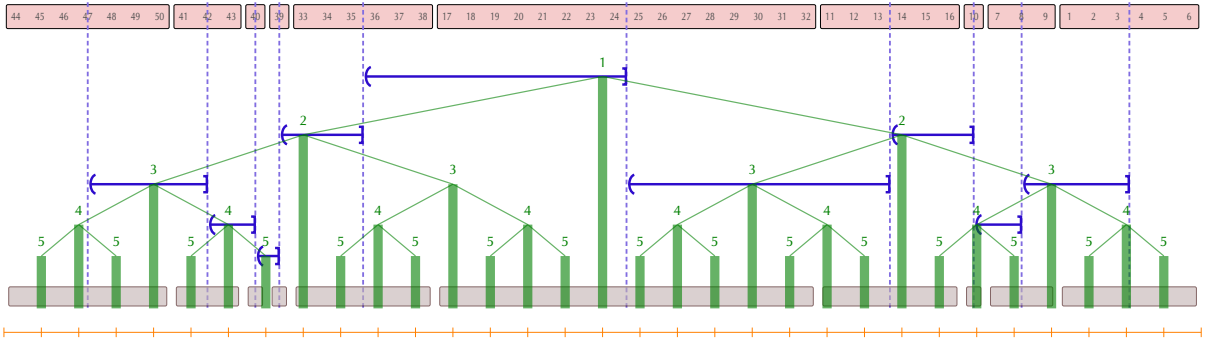
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



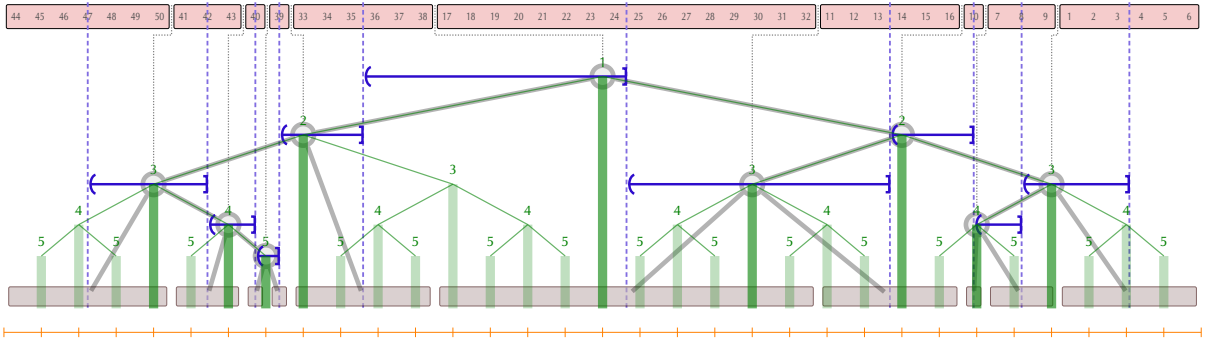
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



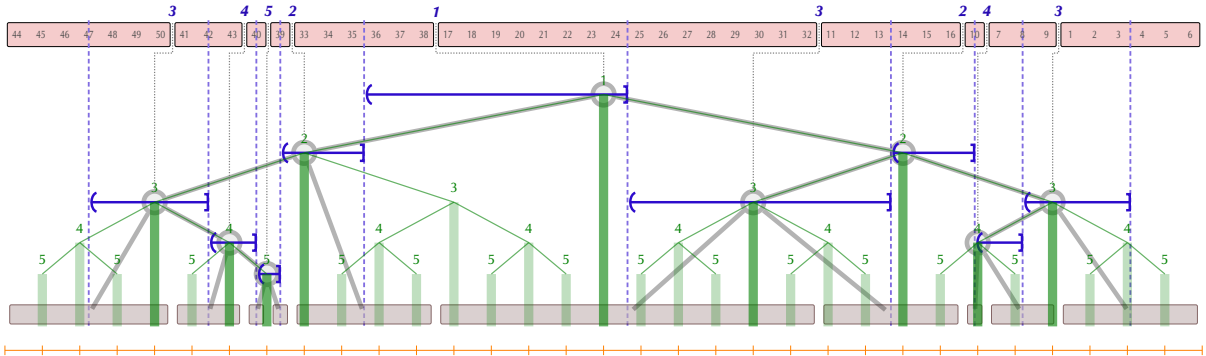
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



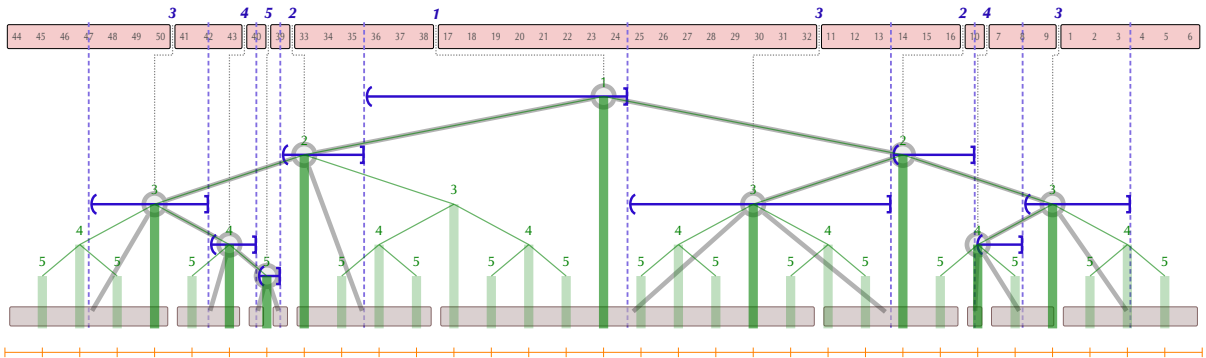
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



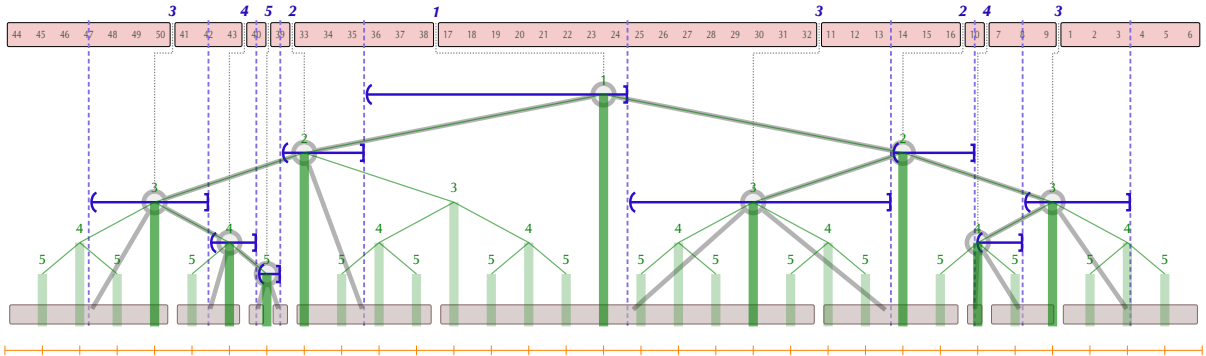
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node
  - ⇝ assigns each run boundary a depth

Run-Boundary Powers



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

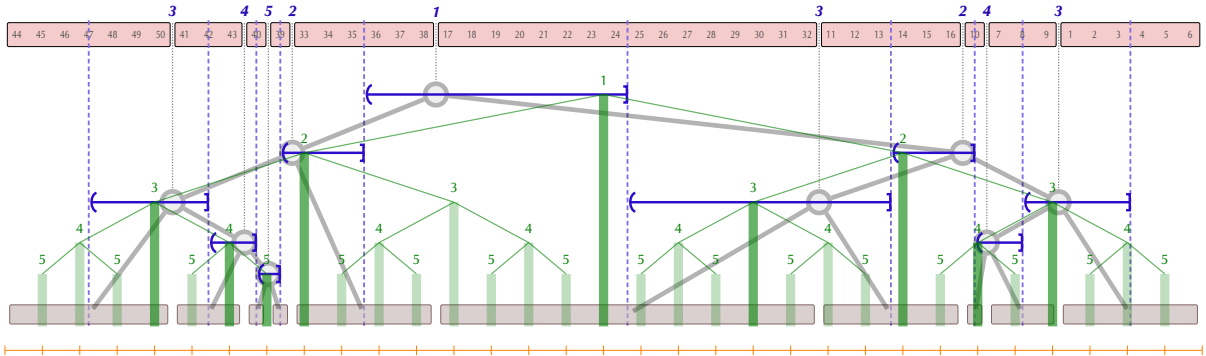
Run-Boundary Powers



- (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

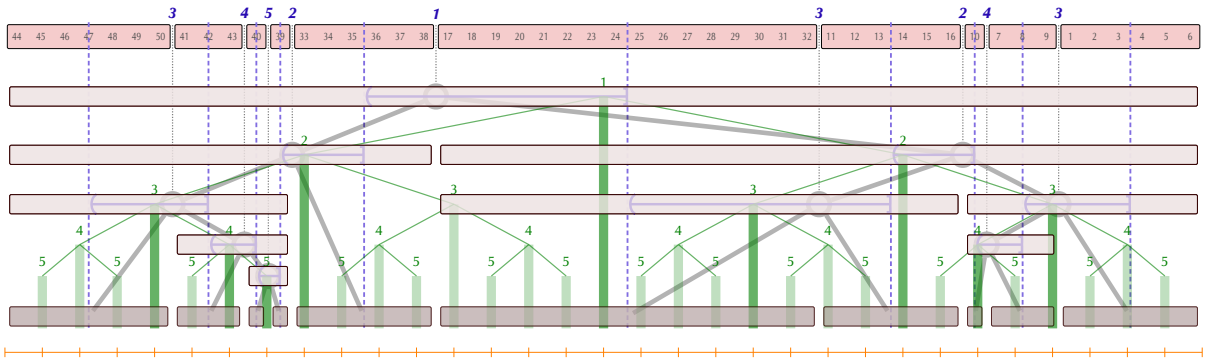
Run-Boundary Powers



- (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

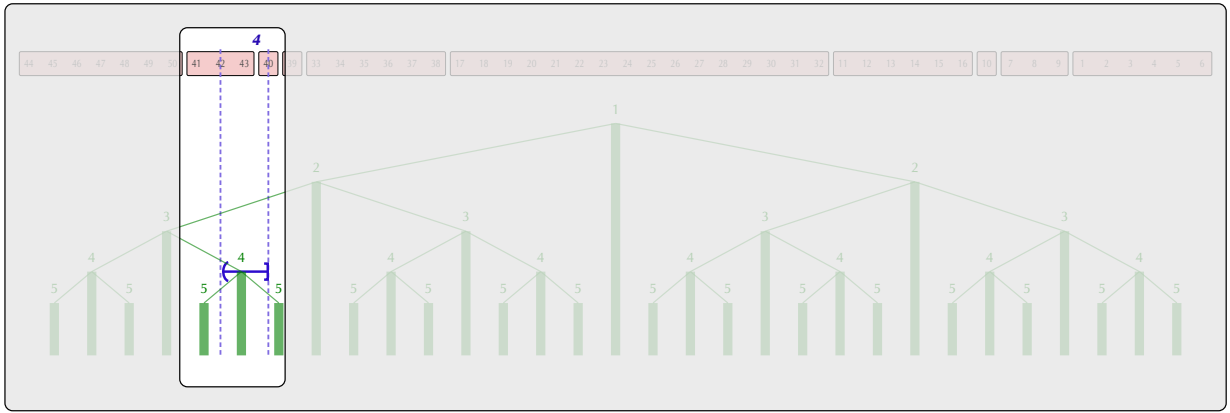
Run-Boundary Powers



- (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

Run-Boundary Powers are Local

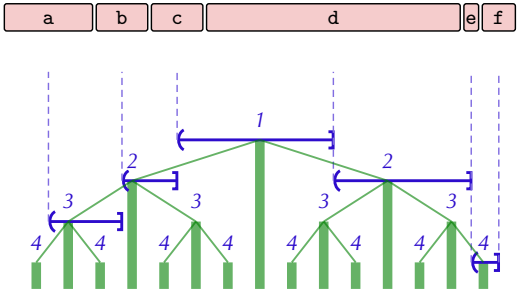


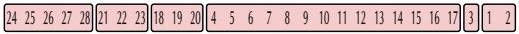
Computation of powers only depends on two adjacent runs.

Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

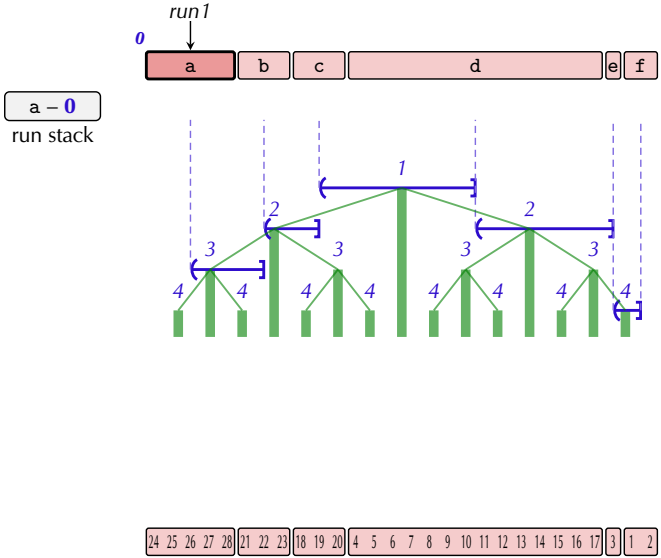




Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

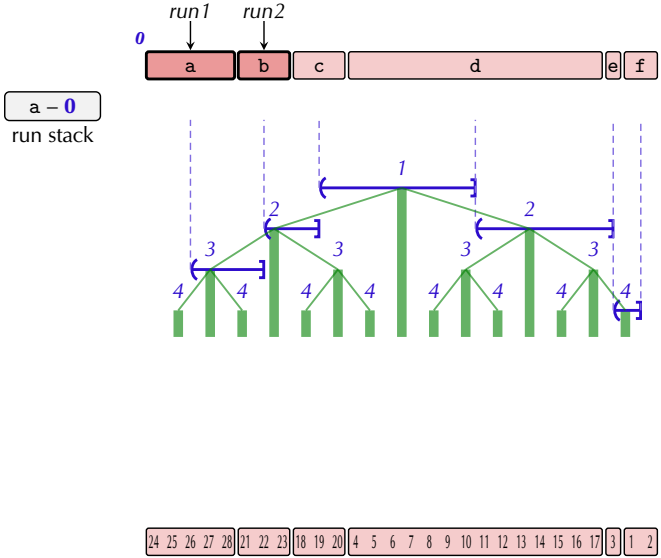
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

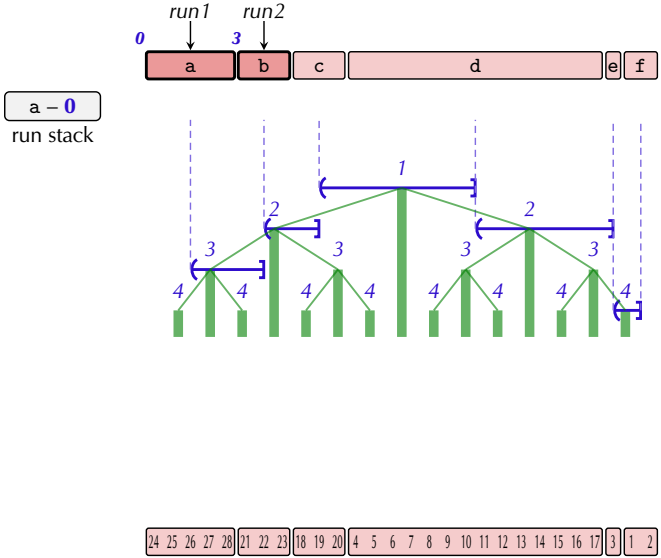
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

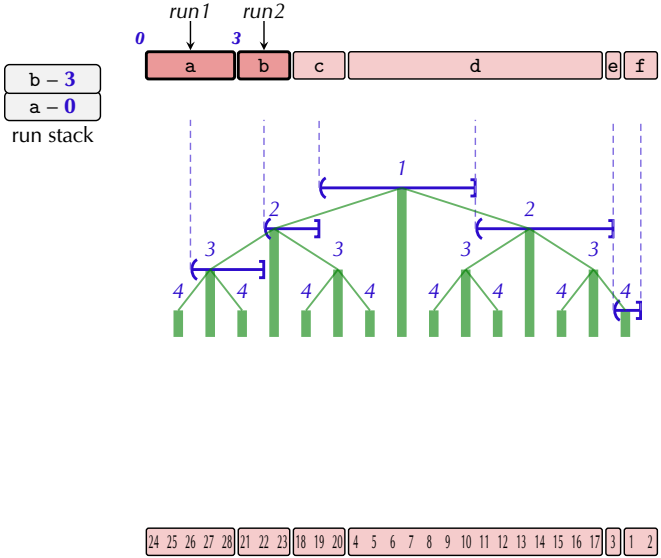
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

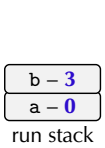
full code:
`https://power-sort.github.io`

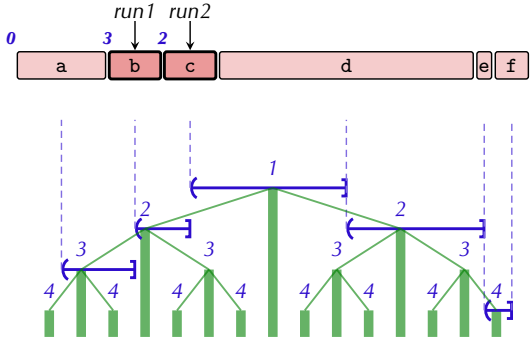


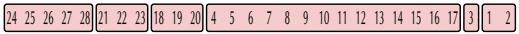
Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

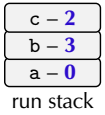


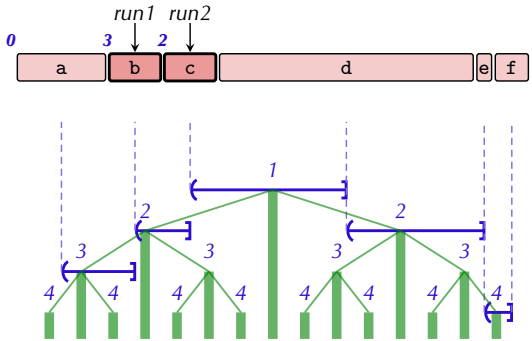




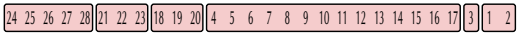
Powersort

```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst, runs)
11         runs.append((i,j-i,p)); i = j
12     while len(runs) >= 2:
13         merge_topmost_2(lst, runs)
```



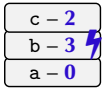


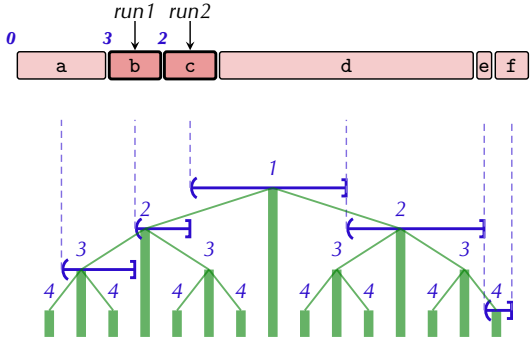
full code:
`https://power-sort.github.io`



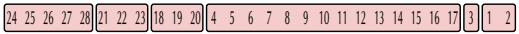
Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```



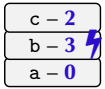


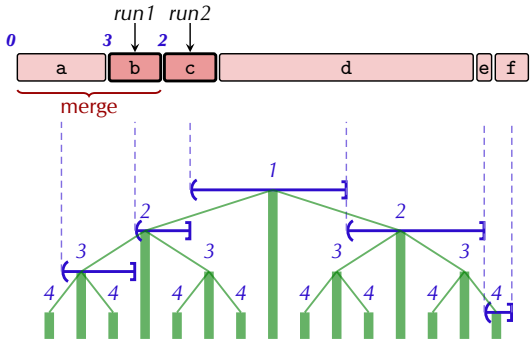
full code:
`https://power-sort.github.io`



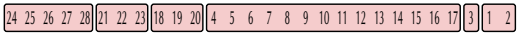
Powersort

```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst, runs)
11         runs.append((i,j-i,p)); i = j
12     while len(runs) >= 2:
13         merge_topmost_2(lst, runs)
```





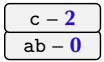
full code:
`https://power-sort.github.io`

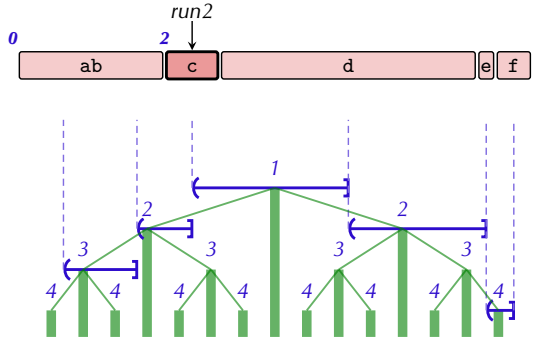


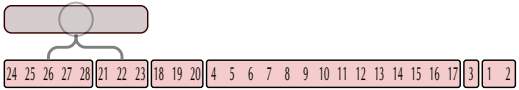
Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`



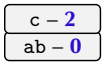


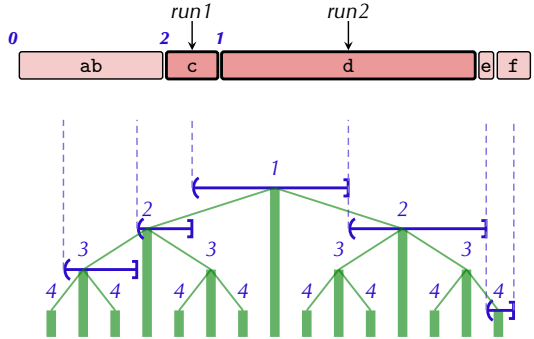


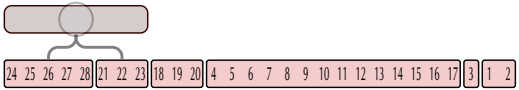
Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

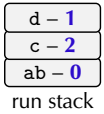


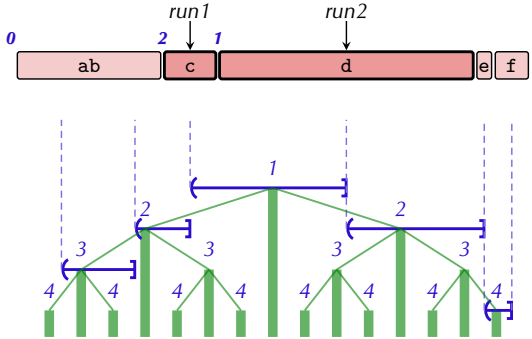




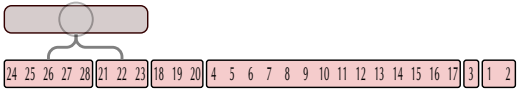
Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```



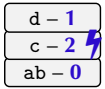


full code:
`https://power-sort.github.io`

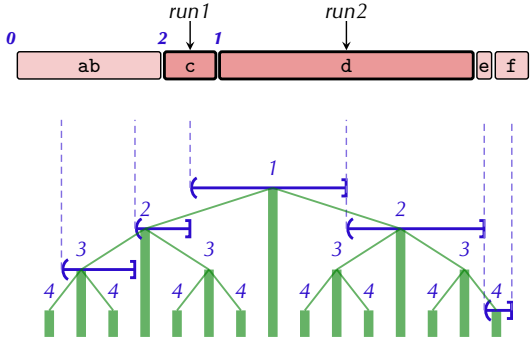


Powersort

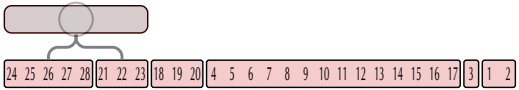
```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```



run stack

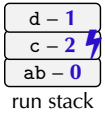


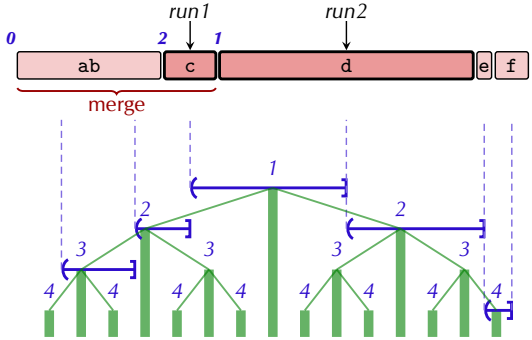
full code:
https://power-sort.github.io



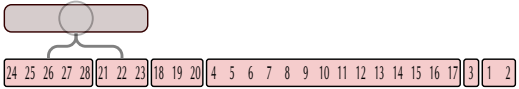
Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```





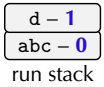
full code:
`https://power-sort.github.io`

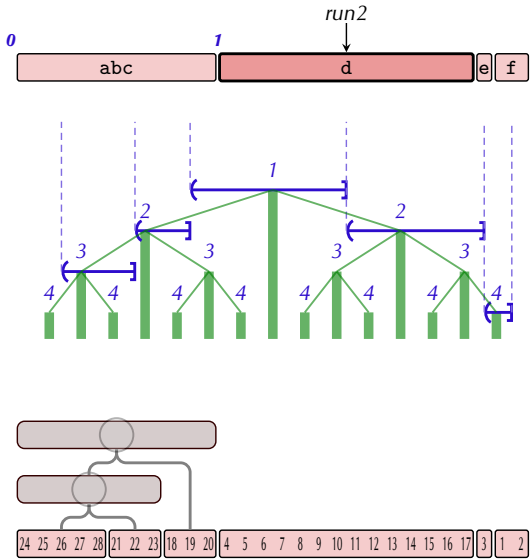


Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

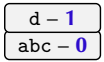


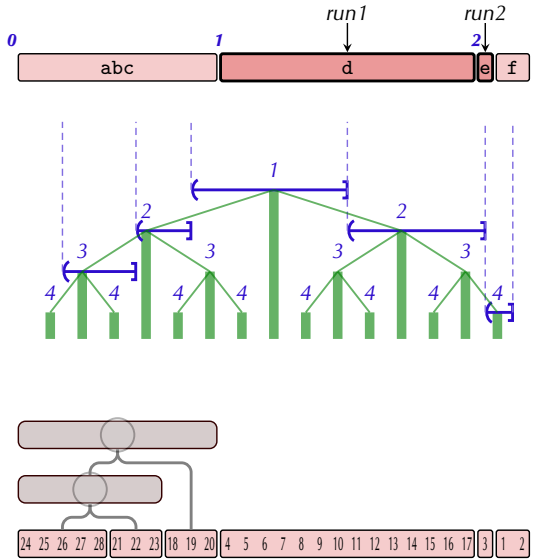


Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

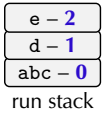


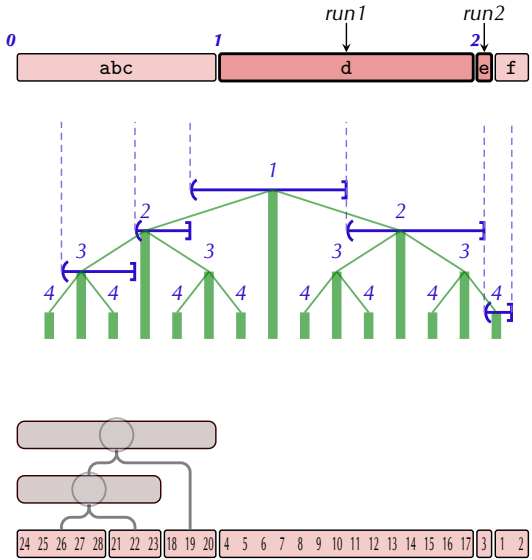


Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
https://power-sort.github.io

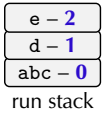


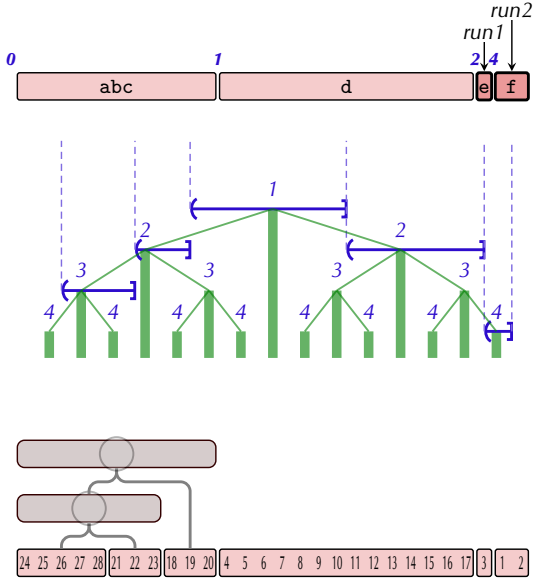


Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

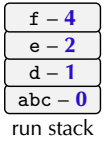


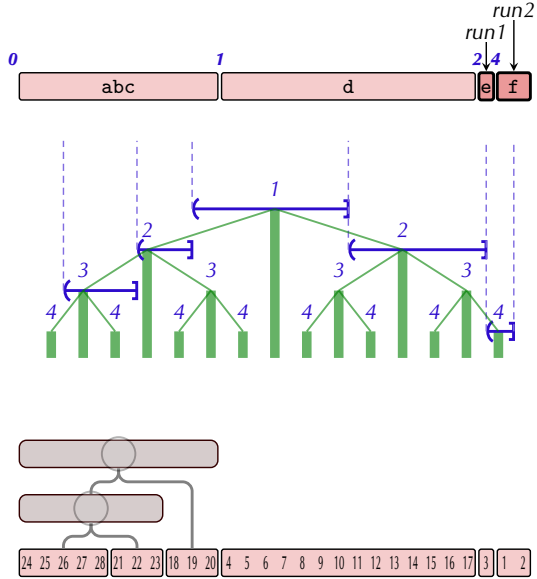


Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

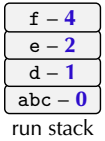


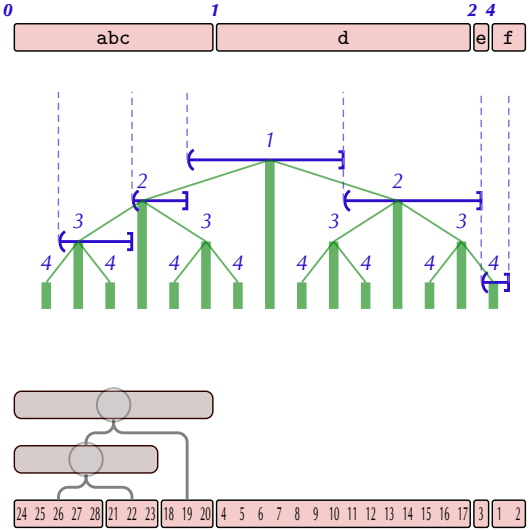


Powersort

```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst, runs)
11         runs.append((i,j-i,p)); i = j
12     while len(runs) >= 2:
13         merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

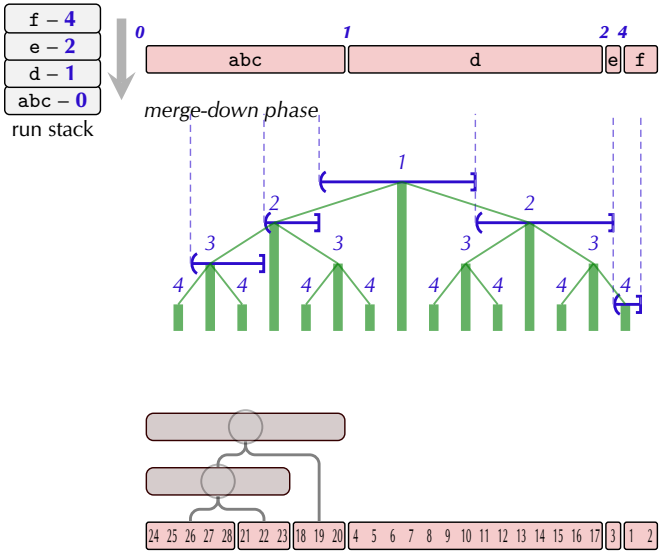




Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

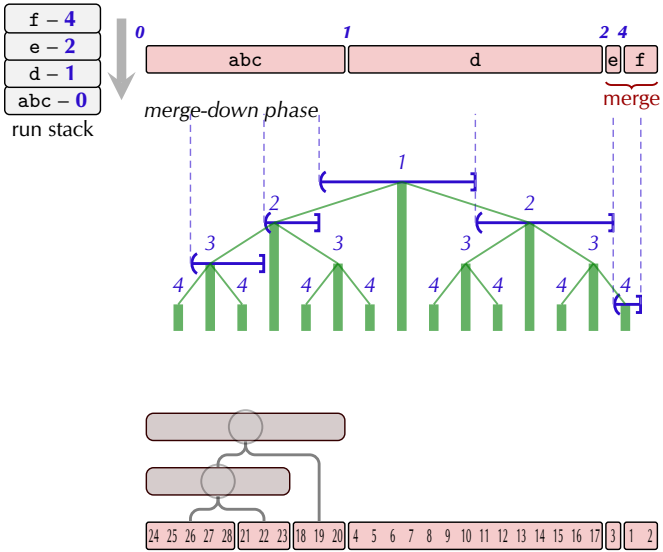
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

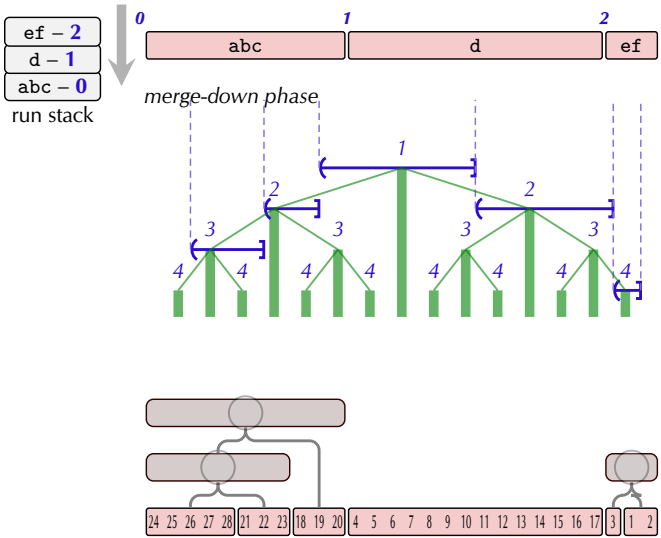
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

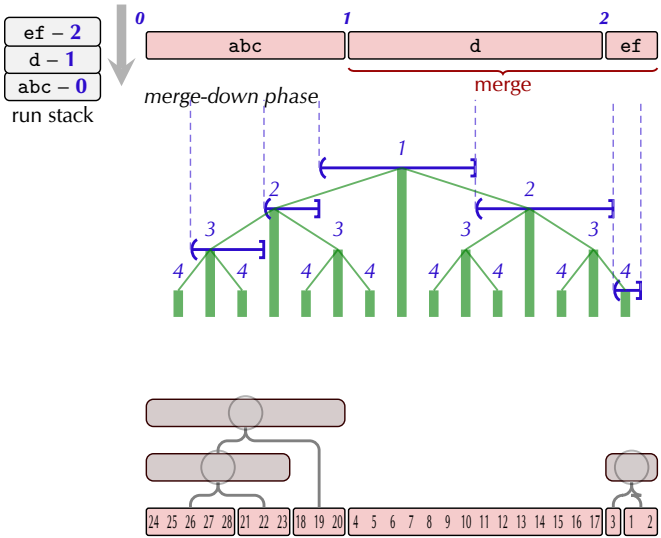
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

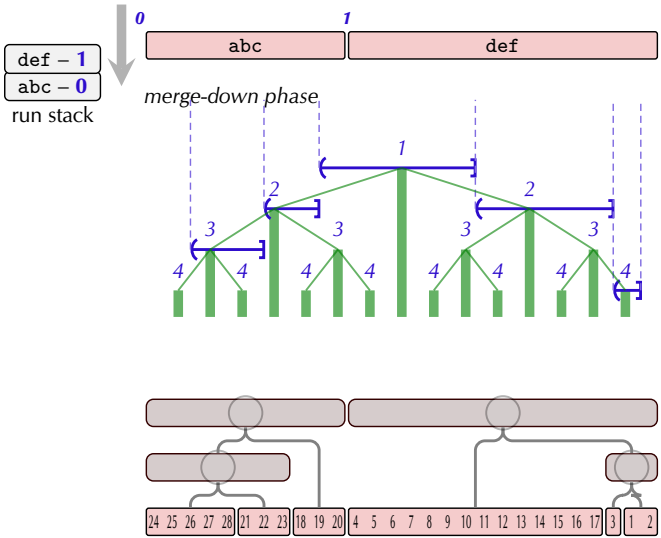
full code:
https://power-sort.github.io



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

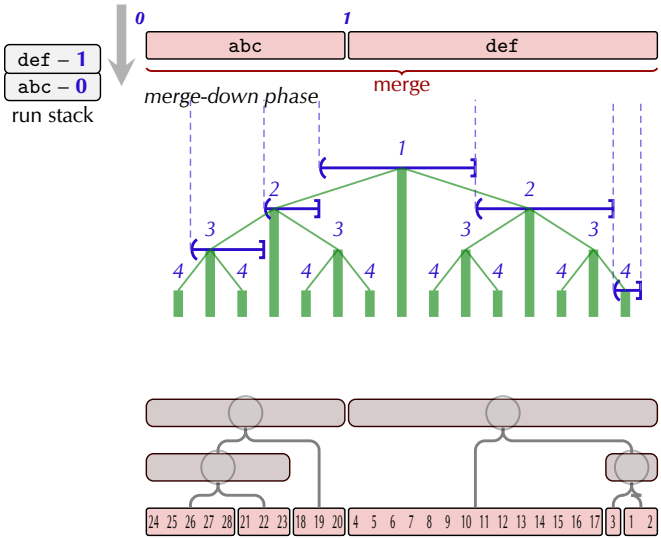
full code:
`https://power-sort.github.io`



Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

full code:
`https://power-sort.github.io`

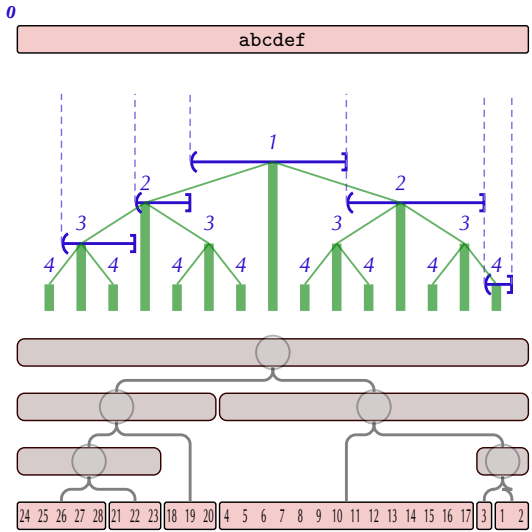


Powersort

```
def powersort(lst)
    i = 0; n = len(lst)
    runs = [];
    j = extend_run(lst, i)
    runs.append((i,j-i,0)); i = j
    while i < n:
        j = extend_run(lst, i)
        p = power(runs[-1], (i,j-i), n)
        while p <= runs[-1][2]
            merge_topmost_2(lst, runs)
        runs.append((i,j-i,p)); i = j
    while len(runs) >= 2:
        merge_topmost_2(lst, runs)
```

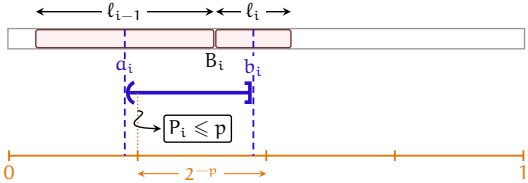
full code:
`https://power-sort.github.io`

abcdef – 0
run stack



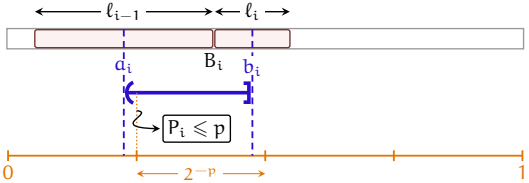
Computing powers

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



Computing powers

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

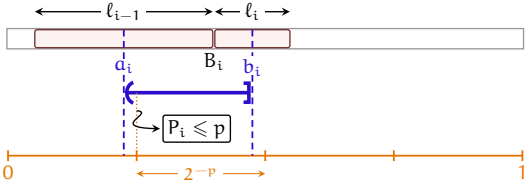



```
def power(run1, run2, n):
    i1, n1 = run1
    i2, n2 = run2
    A = 2 * i1 + n1   # A = 2na_i
    B = A + n1 + n2   # B = 2nb_i
    p = 0
    while True:
        p += 1
        if A >= n:
            A -= n; B -= n
        elif B >= n:
            break
        A <<= 1; B <<= 1
    return p
```

- with bitwise trickery O(1) time possible

Computing powers

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




```
def power(run1, run2, n):
    i1, n1 = run1
    i2, n2 = run2
    A = 2 * i1 + n1  # A = 2n a_i
    B = A + n1 + n2  # B = 2n b_i
    p = 0
    while True:
        p += 1
        if A >= n:
            A -= n; B -= n
        elif B >= n:
            break
        A <<= 1; B <<= 1
    return p
```

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

Obvious analysis

- $0 \leqslant P_i \leqslant \lceil \lg n \rceil$
- powers on stack strictly increasing

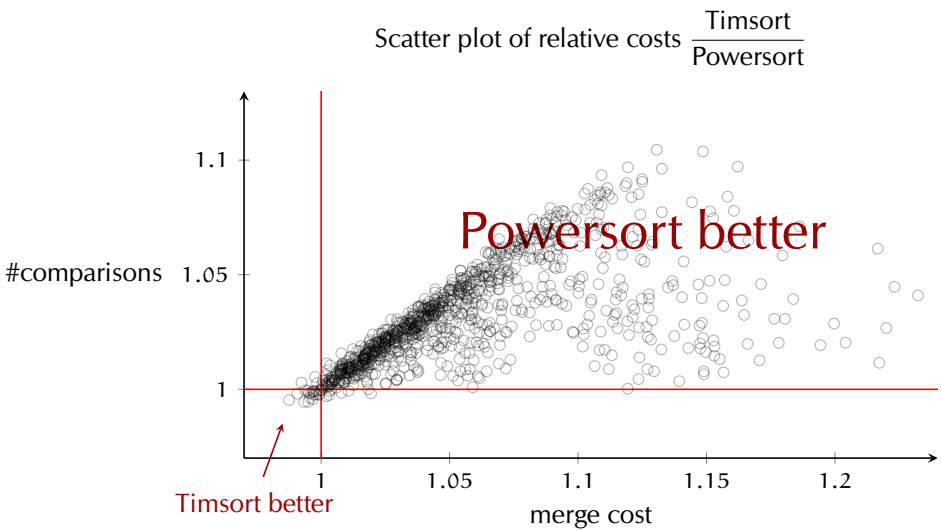
$\rightsquigarrow$ Stack height $\leqslant \lceil \lg n \rceil + 1$

Some performance data

Summary claims:

1. Typical improvement from Powersort:
   0-5% fewer comparisons; occasionally 20–60%. Data
   (One can contrive inputs where Powersort does worse; seems inevitable)
2. No running time regressions: never measurably slower in actual running time Data
3. Sometimes substantially faster (20–40%)

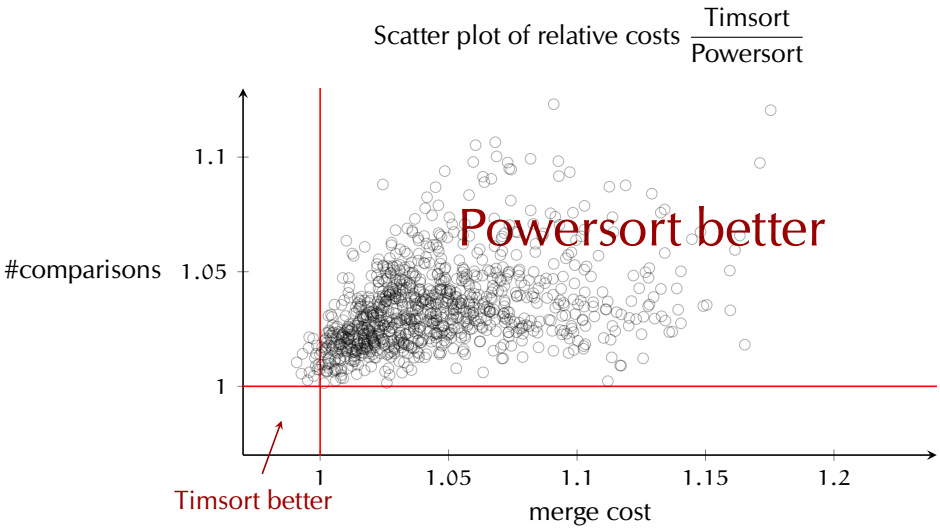
Abstract cost measures



- Worse on 2-3%, but if so, only slightly.
- On average, 3% fewer cmps and 5% less merge cost.

Input: Runs of expected length $\sqrt{n}$, $n = 10^5$

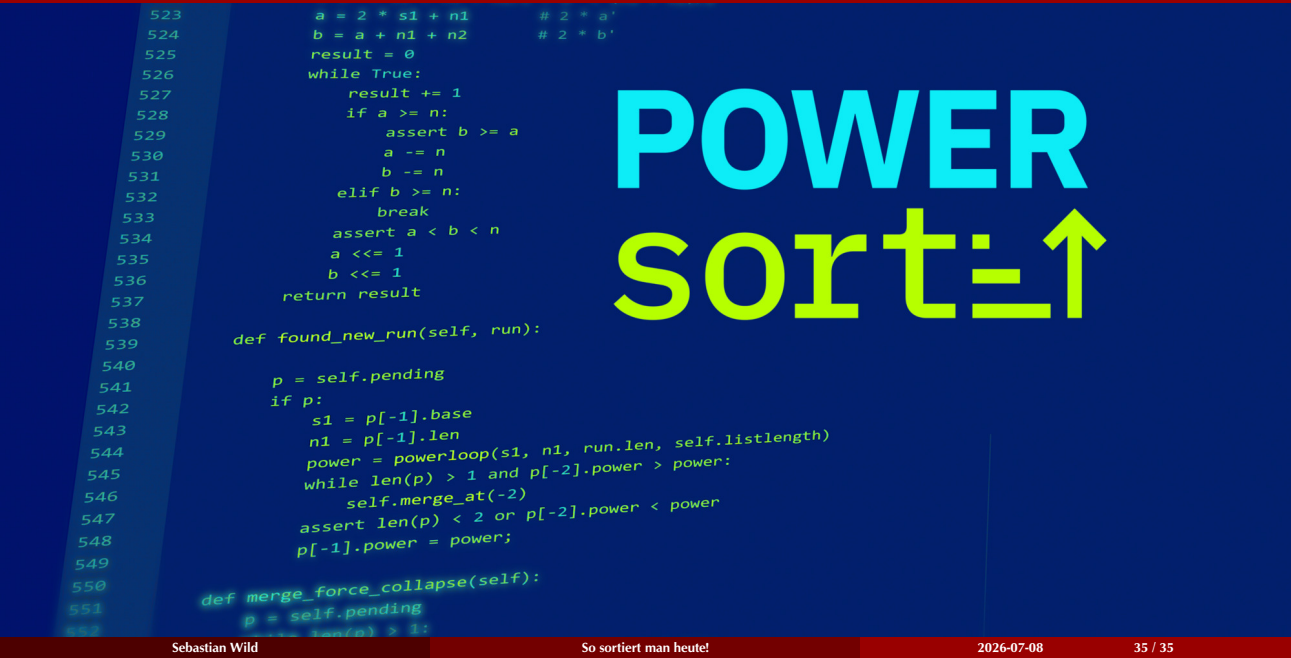
Abstract cost measures



- Worse on 2-3%, but if so, only slightly.
- On average, 3% fewer cmps and 5% less merge cost.

Input: Tim's mixture of long and short runs

Conclusion



Conclusion

Summary

Conclusion

Summary

- Timsort is stable sorting method of choice

Conclusion

Summary

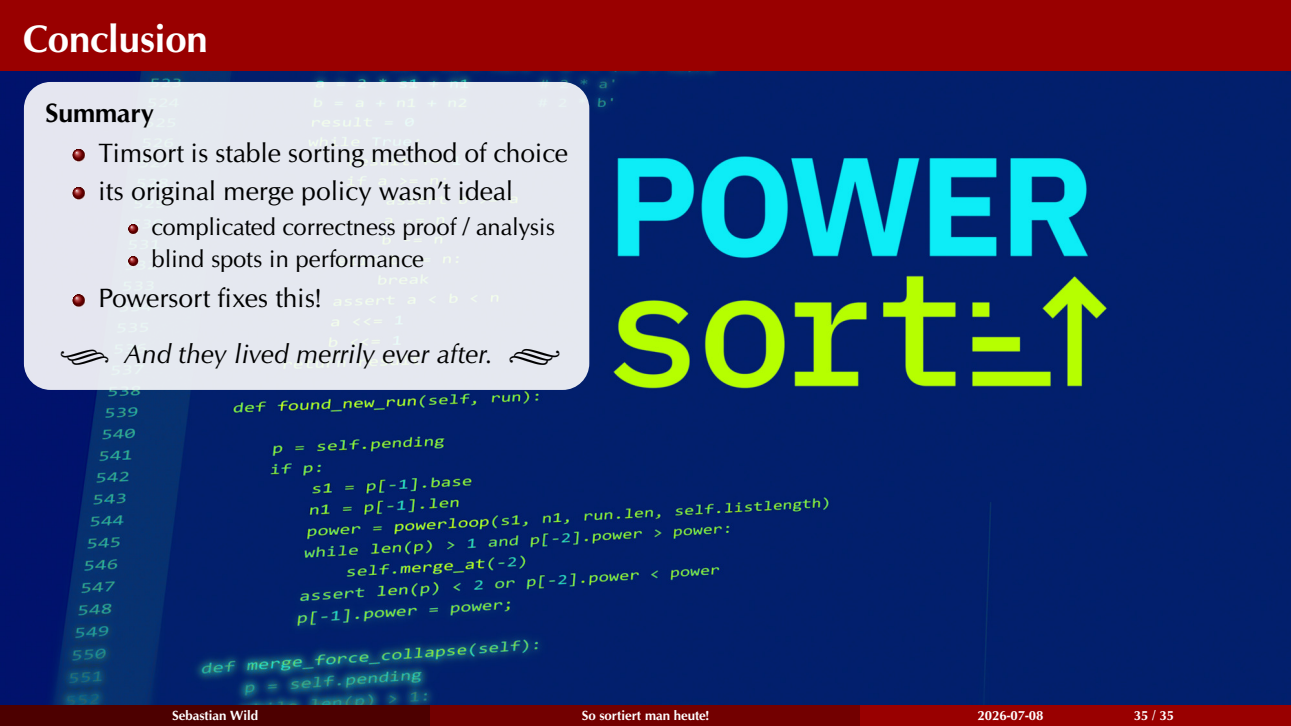
- Timsort is stable sorting method of choice
- its original merge policy wasn't ideal
  - complicated correctness proof / analysis
  - blind spots in performance

Conclusion

Summary

- Timsort is stable sorting method of choice
- its original merge policy wasn't ideal
  - complicated correctness proof / analysis
  - blind spots in performance
- Powersort fixes this!

And they lived merrily ever after.



Conclusion

Summary

- Timsort is stable sorting method of choice
- its original merge policy wasn't ideal
  - complicated correctness proof / analysis
  - blind spots in performance
- Powersort fixes this!

And they lived merrily ever after.

POWER

Goals

1. Powersort in **other libraries** using Timsort?
   - numpy & pandas ✓
   - OpenJDK
   - Android Java runtime
   - V8 JavaScript engine
   - GNU STL `std::stable_sort`

Conclusion

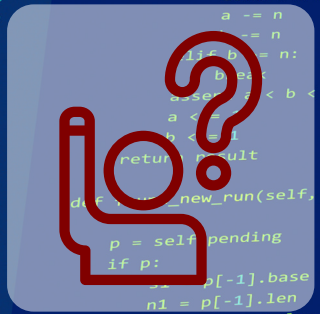
Summary

- Timsort is stable sorting method of choice
- its original merge policy wasn't ideal
  - complicated correctness proof / analysis
  - blind spots in performance
- Powersort fixes this!

And they lived merrily ever after.

Goals

1. Powersort in **other libraries** using Timsort?
   - numpy & pandas ✓
   - OpenJDK
   - Android Java runtime
   - V8 JavaScript engine
   - GNU STL `std::stable_sort`
2. Where else can adaptive-algorithms ideas spark practical improvements?



POWER sort

Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

↓

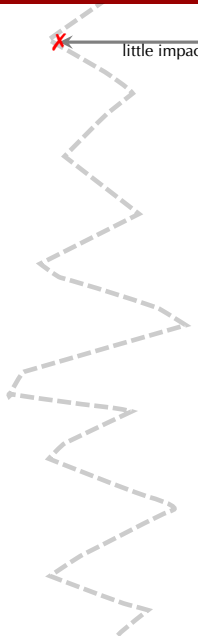
Timsort in CPython 2001
ad-hoc rules for merge policy

Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

↓

Timsort in CPython 2001
ad-hoc rules for merge policy



1990s: Theoretical literature on adaptive sorting

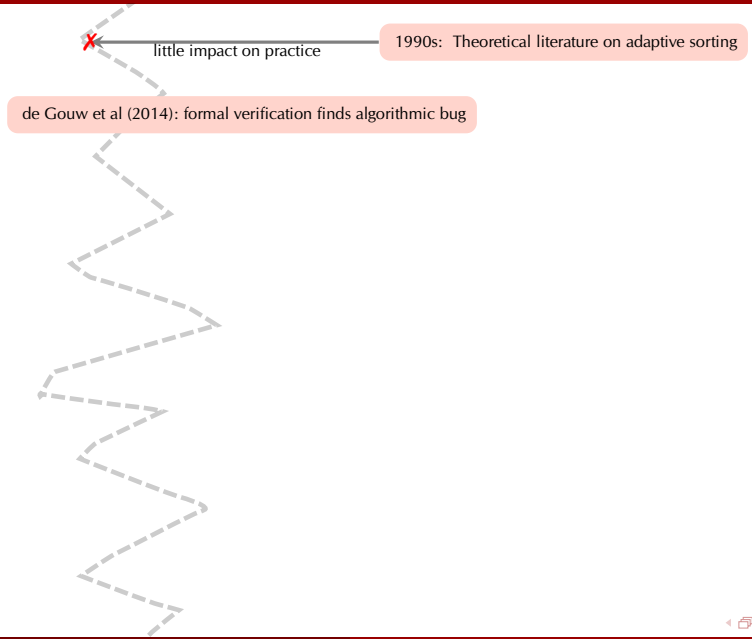
✗ little impact on practice

Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

↓

Timsort in CPython 2001
ad-hoc rules for merge policy



Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

Timsort in CPython 2001
ad-hoc rules for merge policy

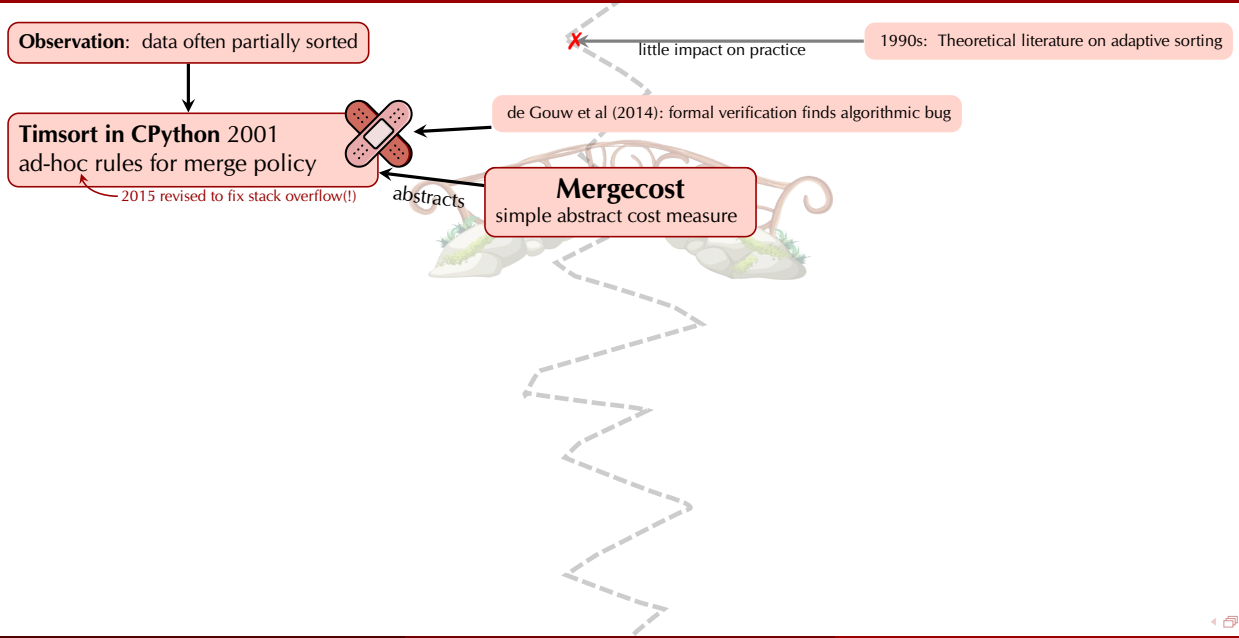
2015 revised to fix stack overflow(!)

de Gouw et al (2014): formal verification finds algorithmic bug

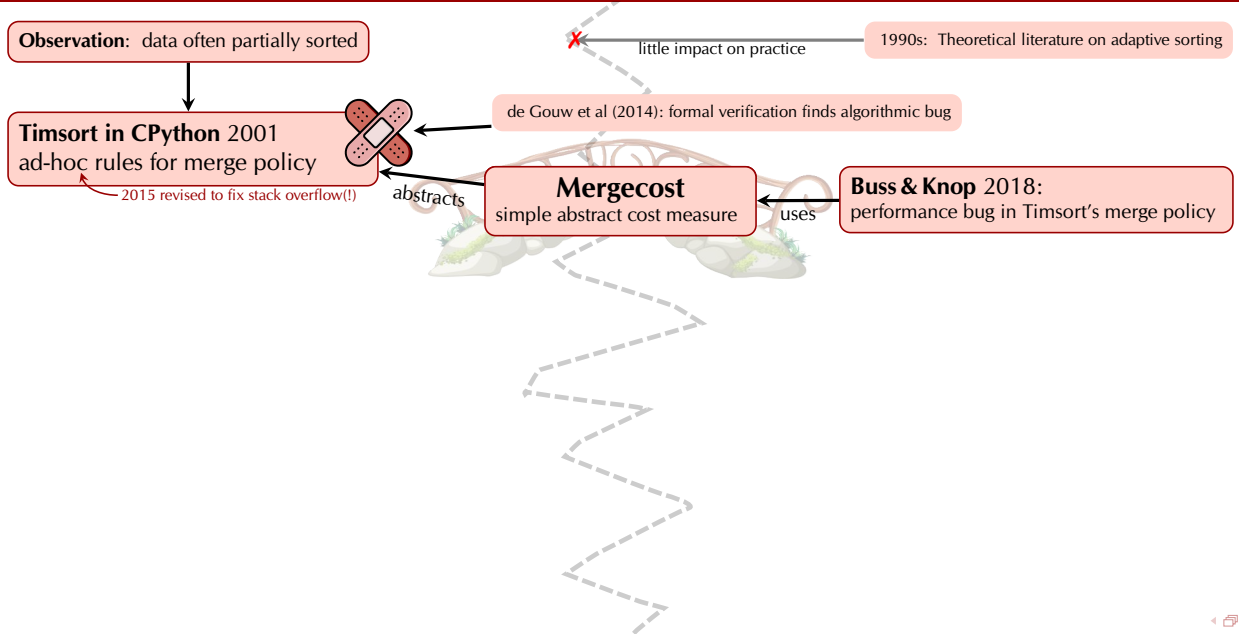
little impact on practice

1990s: Theoretical literature on adaptive sorting

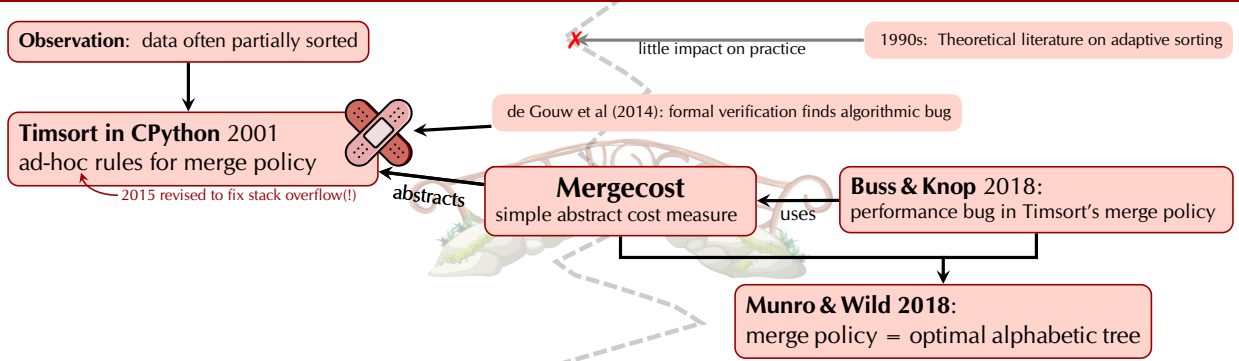
Powersort: An Algorithm Science Success Story



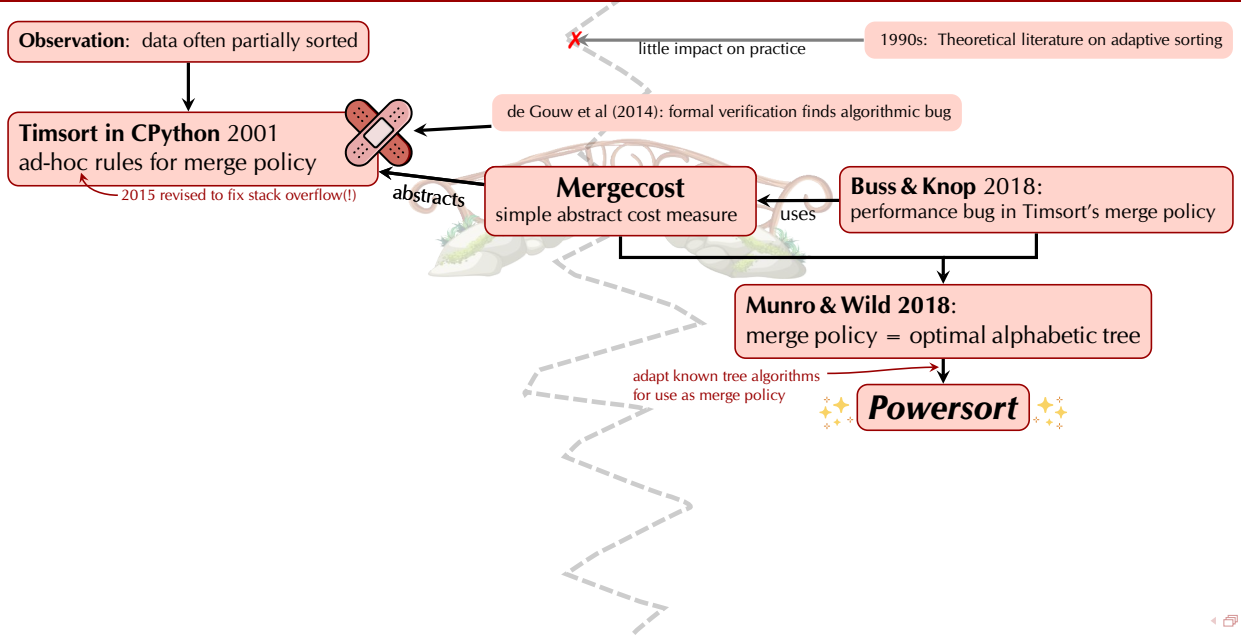
Powersort: An Algorithm Science Success Story



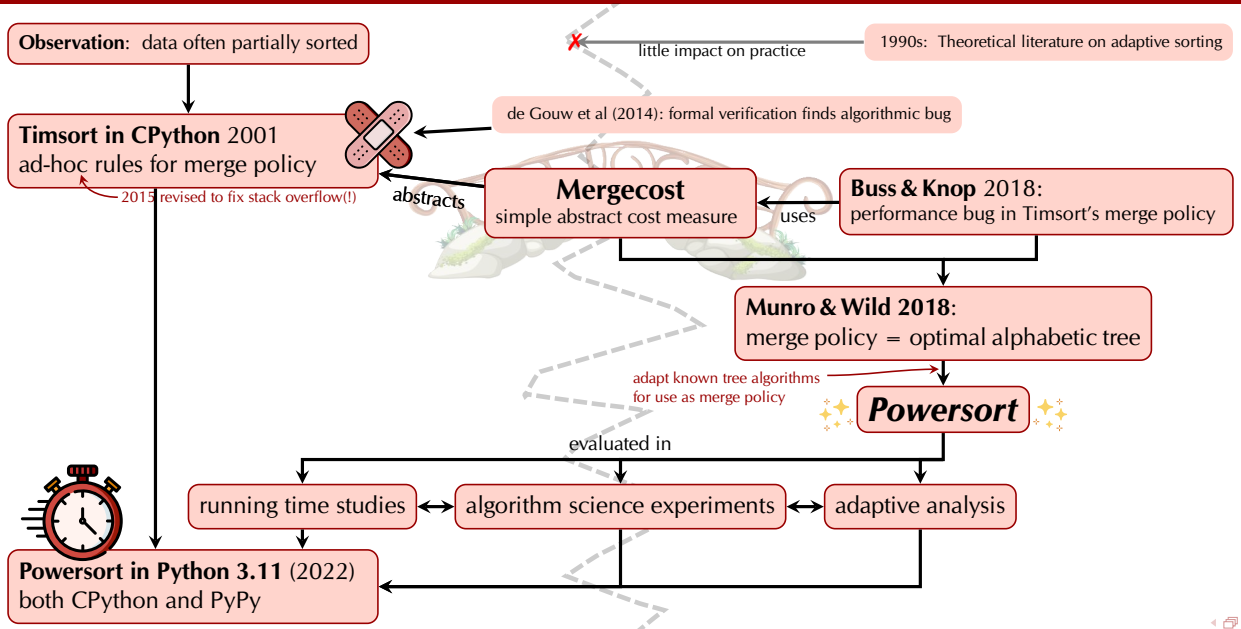
Powersort: An Algorithm Science Success Story



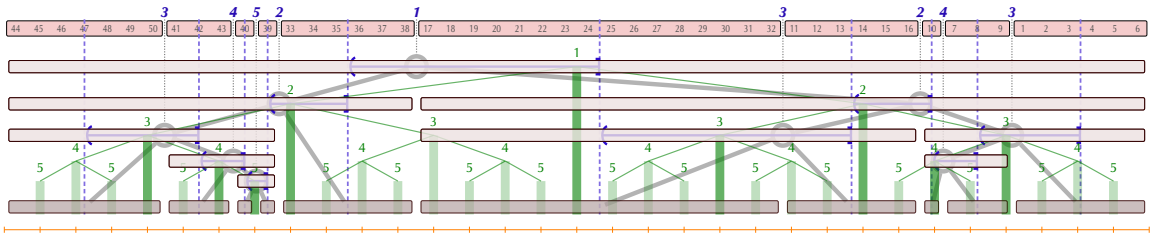
Powersort: An Algorithm Science Success Story



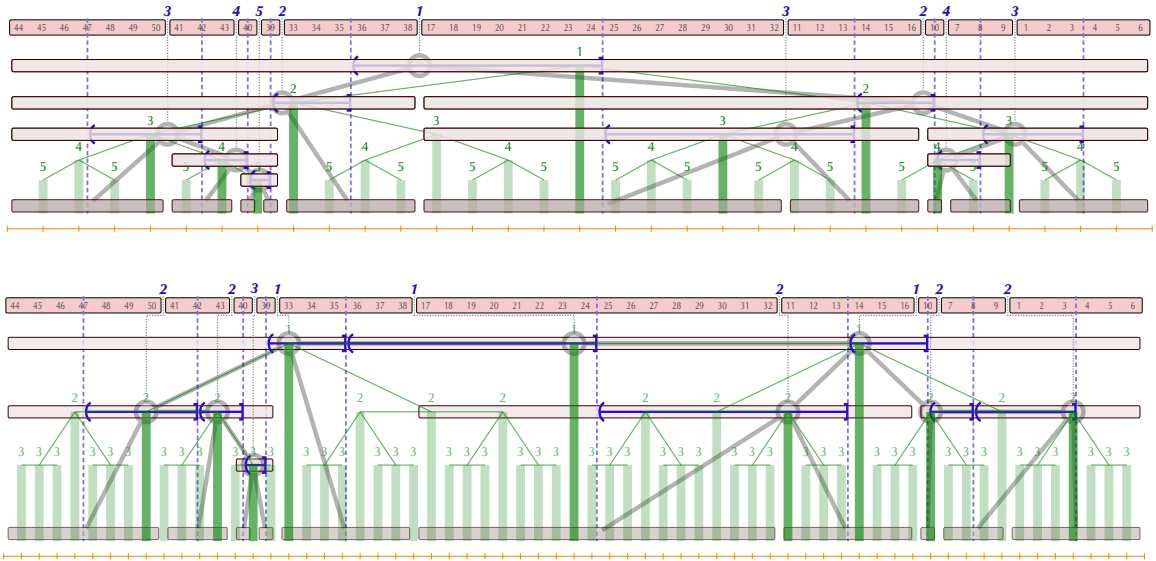
Powersort: An Algorithm Science Success Story



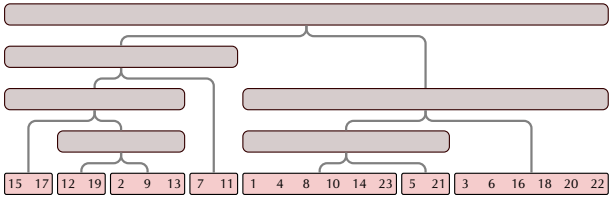
4-way Powersort



4-way Powersort

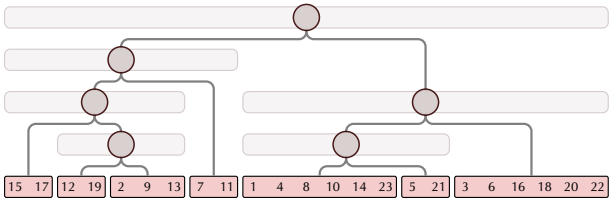


Mergesort meets Binary Search Trees



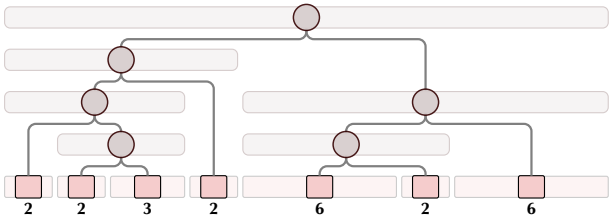
Merge cost = total area of 

Mergesort meets Binary Search Trees



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

Mergesort meets Binary Search Trees



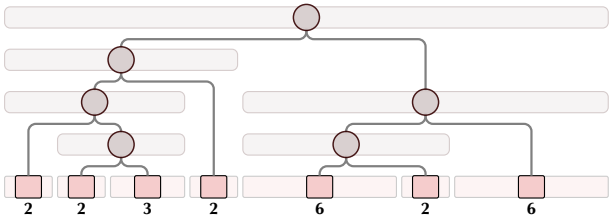
Merge cost = total area of 

= total length of paths to all array entries

= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

Mergesort meets Binary Search Trees

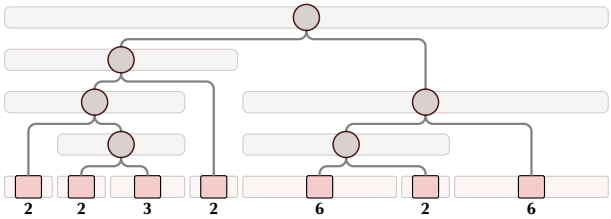


Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

Mergesort meets Binary Search Trees



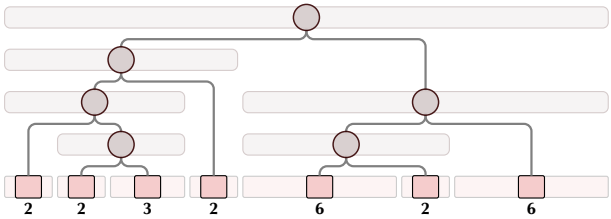
Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

Mergesort meets Binary Search Trees



How to compute good merge tree?

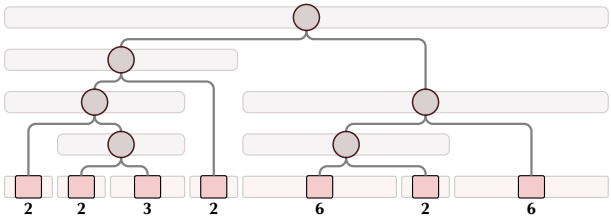
Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \dots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

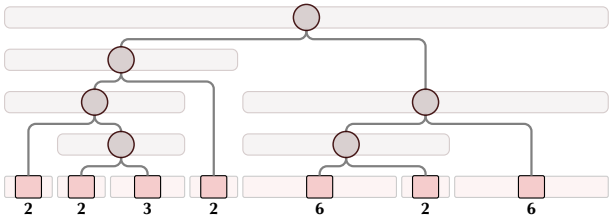
$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

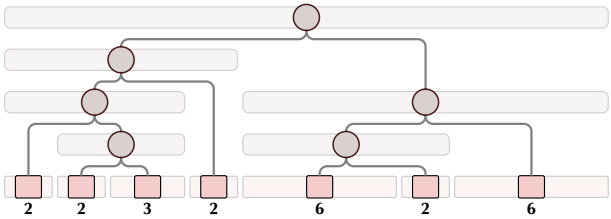
$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs ← indep. discovered 5x
    - Burge 1958
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
  - must sort lengths
  - **not stable**

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

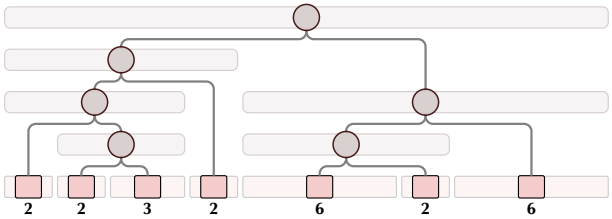
How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - ↯ **not stable**

  indep. discovered 5x
  - Burge 1958
  - Golin & Sedgewick 1993
  - Takaoka 1998
  - Barbay & Navarro 2009
  - Chandramouli & Goldstein 2014

- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

⇝ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - ↯ **not stable**

  indep. discovered 5x
  - Burge 1958
  - Golin & Sedgewick 1993
  - Takaoka 1998
  - Barbay & Navarro 2009
  - Chandramouli & Goldstein 2014

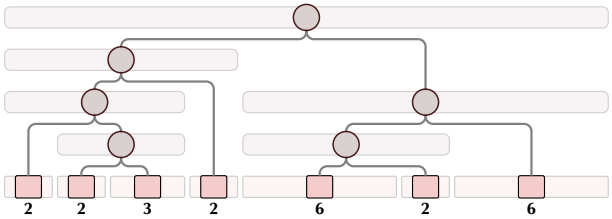
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

…70s are calling

nearly-optimal BST merge

- simple (greedy) linear-time methods!

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - ↯ **not stable**

  indep. discovered 5x
  - Burge 1958
  - Golin & Sedgewick 1993
  - Takaoka 1998
  - Barbay & Navarro 2009
  - Chandramouli & Goldstein 2014

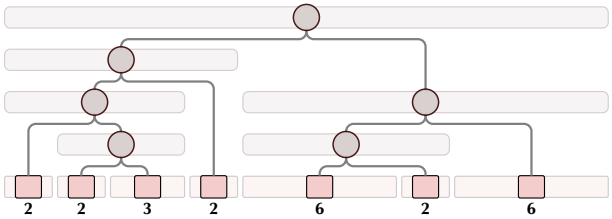
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

…70s are calling

nearly-optimal BST merge

- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

⇝ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - ↯ **not stable**

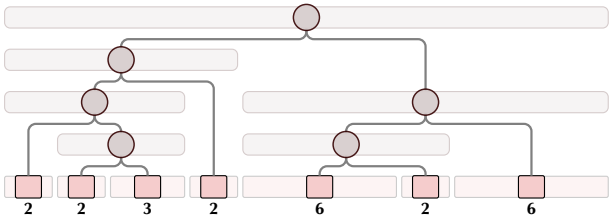
  indep. discovered 5x
  - Burge 1958
  - Golin & Sedgewick 1993
  - Takaoka 1998
  - Barbay & Navarro 2009
  - Chandramouli & Goldstein 2014

- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

nearly-optimal BST merge (…70s are calling)

- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)
- 👎 have to store lengths
- 👎 extra scan to detect runs

Mergesort meets Binary Search Trees



Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

⇝ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - ↯ **not stable**

  indep. discovered 5x
  - Burge 1958
  - Golin & Sedgewick 1993
  - Takaoka 1998
  - Barbay & Navarro 2009
  - Chandramouli & Goldstein 2014

- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

nearly-optimal BST merge (…70s are calling)

- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)
- 👎 have to store lengths
- 👎 extra scan to detect runs

} avoidable!

The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

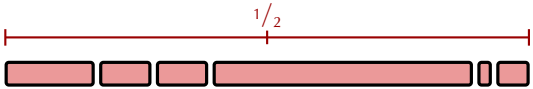
> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

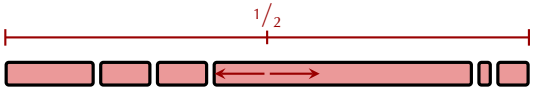


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

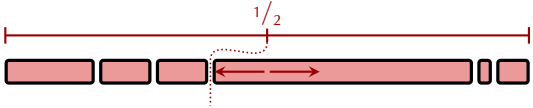


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

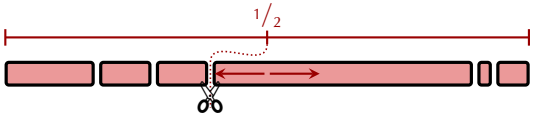


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

  Mehlhorn: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

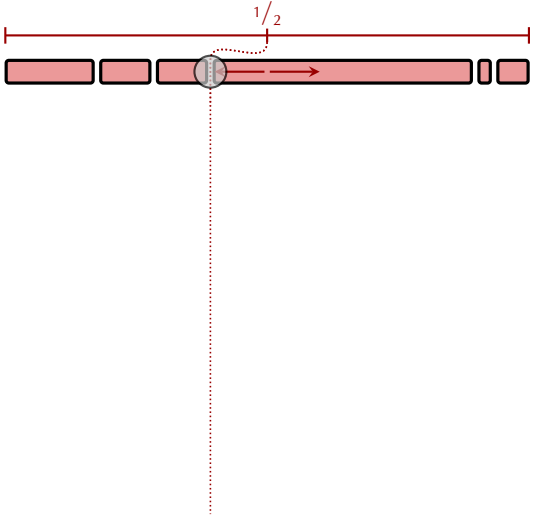


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**



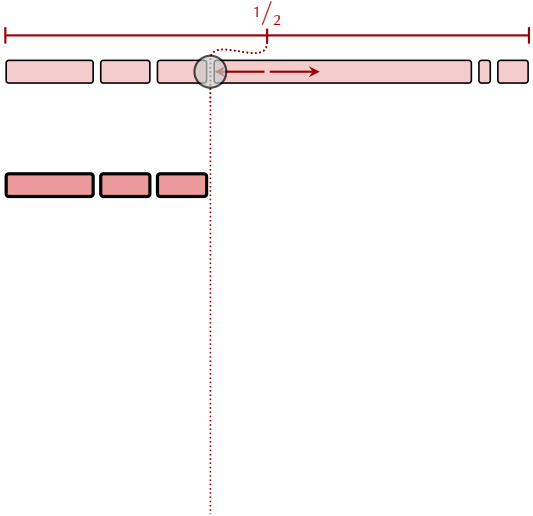
- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

- Powersort is based on the **bisection method**

  Mehlhorn: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

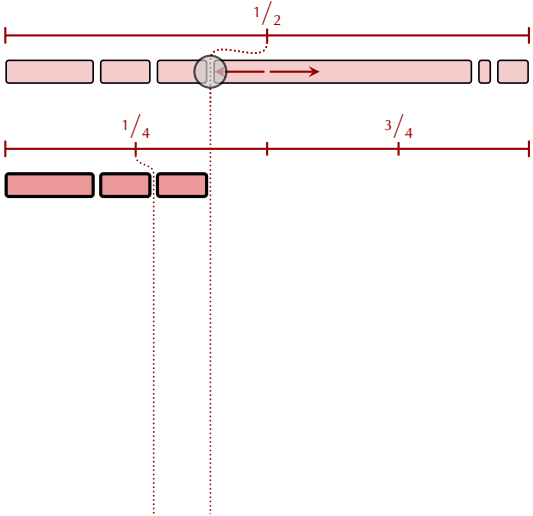


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

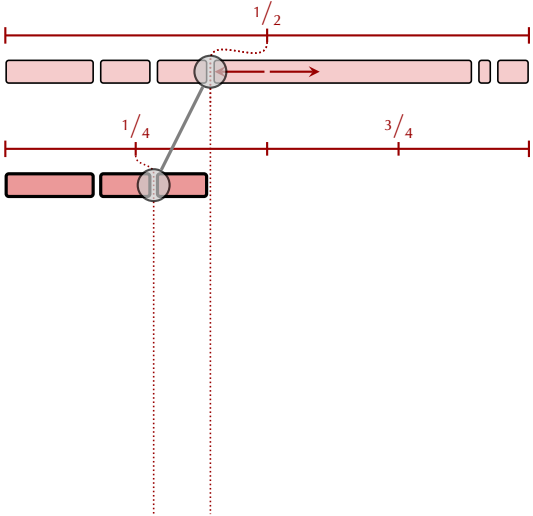


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

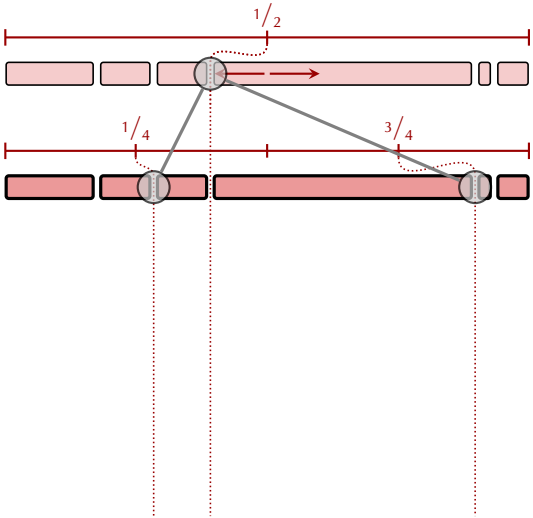


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

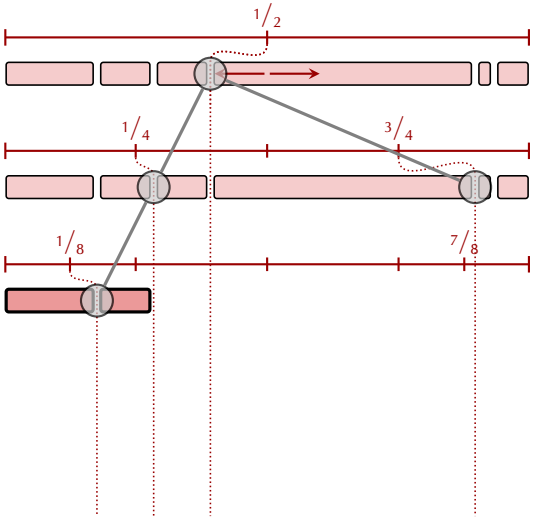


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

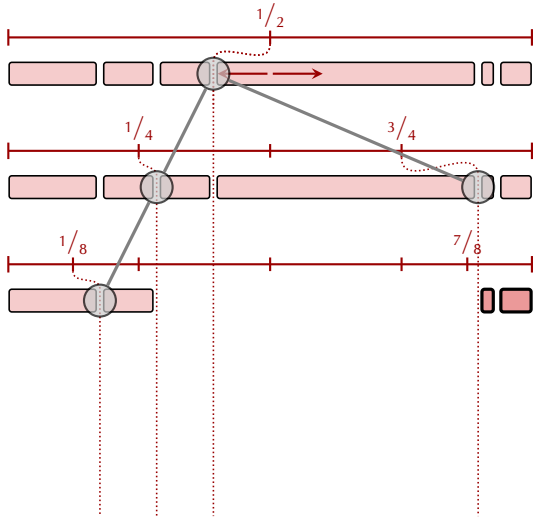


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

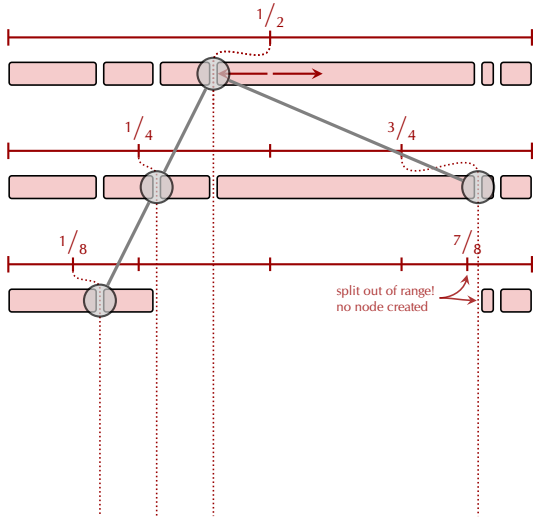


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

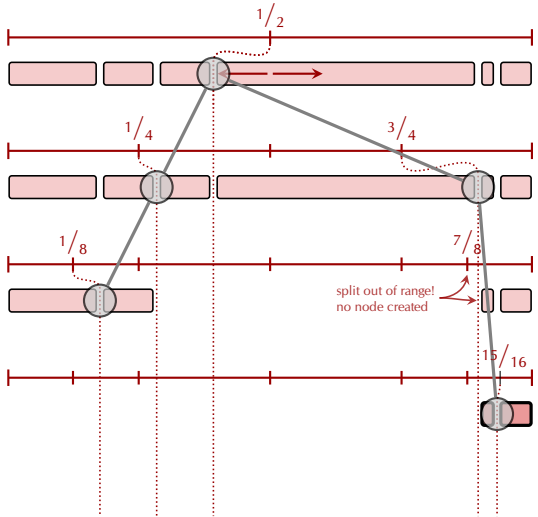


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

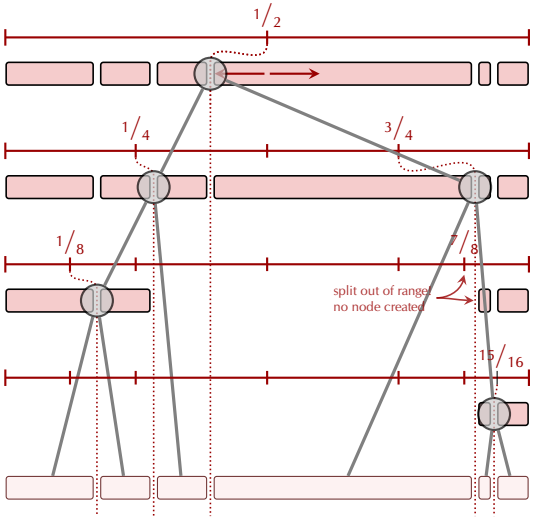


The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides



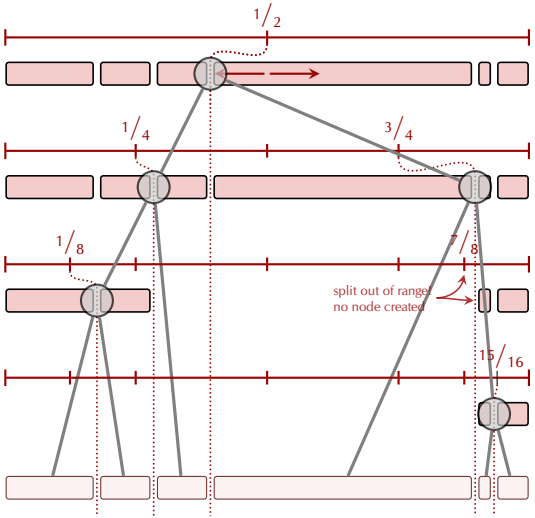
The Bisection Method

- Powersort is based on the **bisection method**

> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

- *Up next:* Don't use recursion!



Icons made by *Freepik*, *Gregor Cresnar*, *Those Icons*, *Smashicons*, *Good Ware*, *Pause08*, and *Madebyoliver* from `www.flaticon.com`.

Vector graphics from Pressfoto, brgfx, macrovector and Jannoon028 on freepik.com
Other photos from `www.pixabay.com`.