Practical Adaptive Sorting and Searching

Sebastian Wild

www.wild-inter.net

based on joint work with Conrado Martínez, Ian Munro, Markus Nebel, Ralph Neininger

Philipps Universität Marburg

AofA 2025

There and back again . . .



Sebastian Wild

www.wild-inter.net

AG Algorithmik

Philipps Universität Marburg

since 01.09.2024 Professor @

Philipps Universität Marburg

2019 – 2024

(Senior) Lecturer @

UNIVERSITY OF LIVERPOOL

Liverpool, UK

2019

Marburg

Kaiserslautern

2017

2016 PhD

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

2017 – 2019

postdoc @

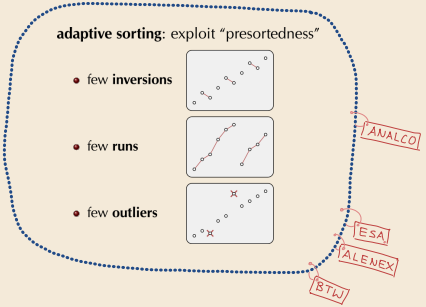
UNIVERSITY OF WATERLOO

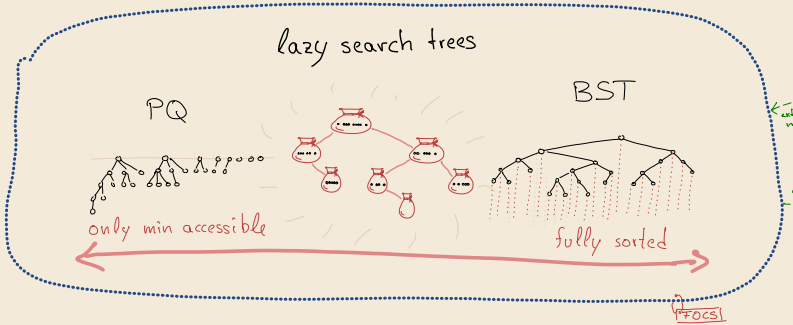
Waterloo, Canada

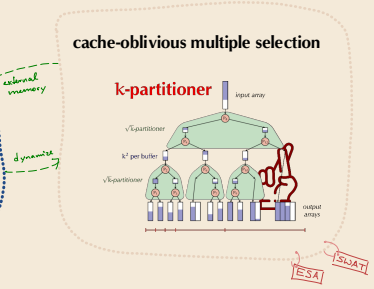
Research Interests: (1) Beyond worst-case algorithms

Goal:
Improve performance when input is “tame” (far form worst case)

My Focus:
Fundamental algorithms & data structures
Dictionaries, Sorting, Searching



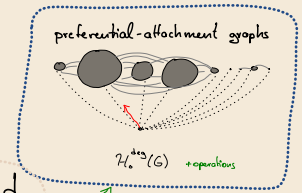


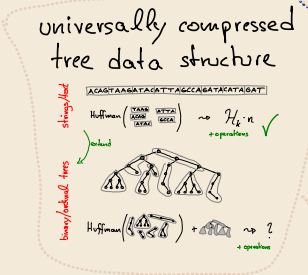


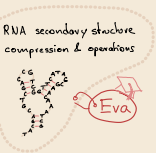
Research Interests: (2) Space-efficient Data Structures

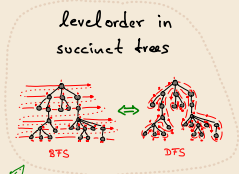
Goal:
Store data in compressed form
But retain efficient access
(without decompressing first)

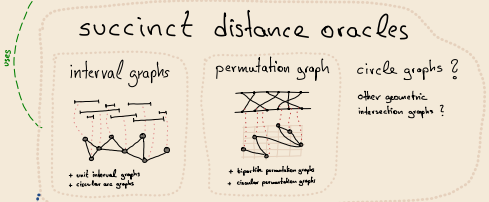
My Focus:
Trees, Graph-structured data

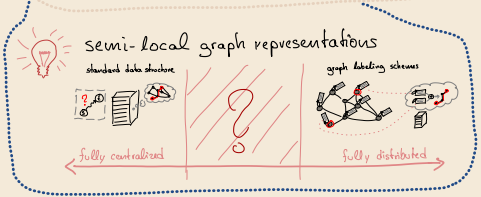








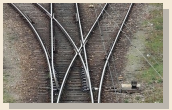




Outline



1 Context



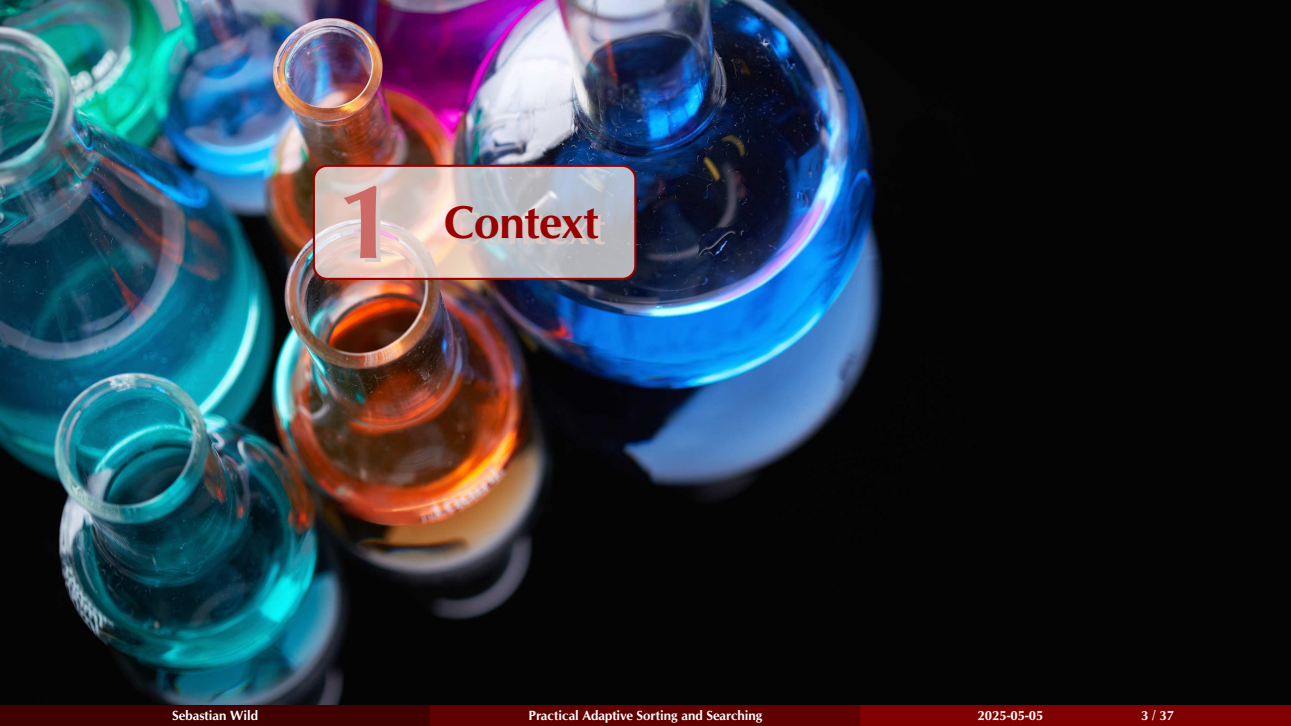
2 Dual-Pivot Quicksort



3 Quicksort with Equal Keys



4 Powersort



1 Context

Adaptive Sorting

Adaptive algorithm: exploit “structure” of input

Adaptive Sorting

Adaptive algorithm: exploit “structure” of input

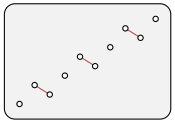
$\rightsquigarrow$ **adaptive sorting**: exploit “presortedness”

Adaptive Sorting

Adaptive algorithm: exploit "structure" of input

$\rightsquigarrow$ **adaptive sorting**: exploit "presortedness"

- few **inversions**

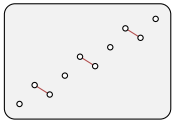


Adaptive Sorting

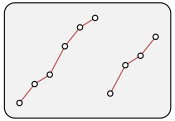
Adaptive algorithm: exploit “structure” of input

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

- few **inversions**



- few **runs**

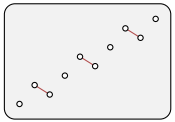


Adaptive Sorting

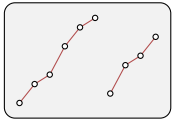
Adaptive algorithm: exploit "structure" of input

$\rightsquigarrow$ **adaptive sorting**: exploit "presortedness"

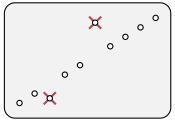
- few **inversions**



- few **runs**



- few **outliers**

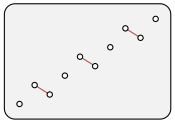


Adaptive Sorting

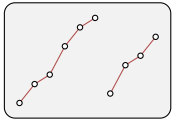
Adaptive algorithm: exploit "structure" of input

$\rightsquigarrow$ **adaptive sorting**: exploit "presortedness"

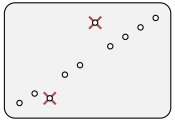
- few **inversions**



- few **runs**



- few **outliers**



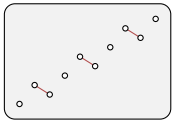
- ...many more

Adaptive Sorting

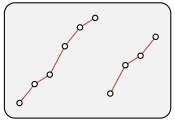
Adaptive algorithm: exploit "structure" of input

$\rightsquigarrow$ **adaptive sorting**: exploit "presortedness"

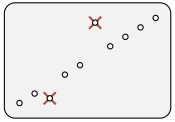
- few **inversions**



- few **runs**



- few **outliers**



- ...many more



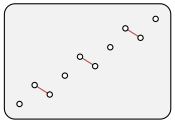
optimal algorithms known for many measures of presortedness

Adaptive Sorting

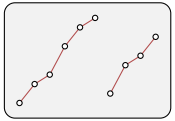
Adaptive algorithm: exploit "structure" of input

⇝ **adaptive sorting**: exploit "presortedness"

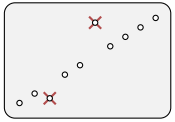
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!



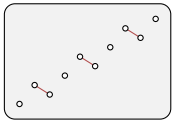
optimal algorithms known for many measures of presortedness

Adaptive Sorting

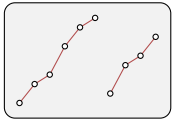
Adaptive algorithm: exploit "structure" of input

⇝ **adaptive sorting**: exploit "presortedness"

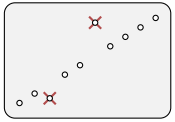
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more



up to constant factors!

optimal algorithms known for many measures of presortedness

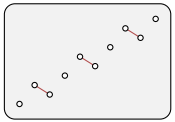
Want:

Adaptive Sorting

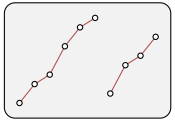
Adaptive algorithm: exploit "structure" of input

$\rightsquigarrow$ **adaptive sorting**: exploit "presortedness"

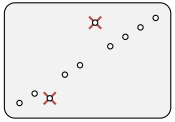
- few **inversions**



- few **runs**



- few **outliers**



- ...many more

up to constant factors!



optimal algorithms known for many measures of presortedness

Want:

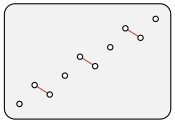
- Optimal up to lower order terms

Adaptive Sorting

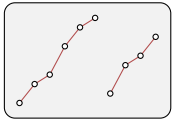
Adaptive algorithm: exploit "structure" of input

$\rightsquigarrow$ **adaptive sorting**: exploit "presortedness"

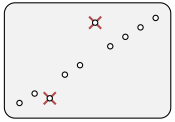
- few **inversions**



- 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

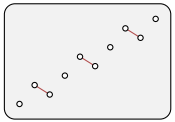
practical methods

Adaptive Sorting

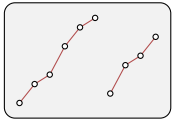
Adaptive algorithm: exploit "structure" of input

⇝ **adaptive sorting**: exploit "presortedness"

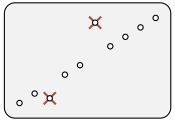
- few **inversions**



- 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

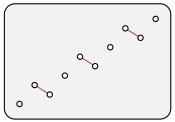
- **low overhead** for detecting presortedness

Adaptive Sorting

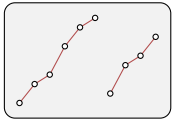
Adaptive algorithm: exploit "structure" of input

⇝ **adaptive sorting**: exploit "presortedness"

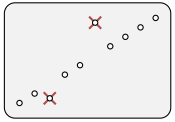
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

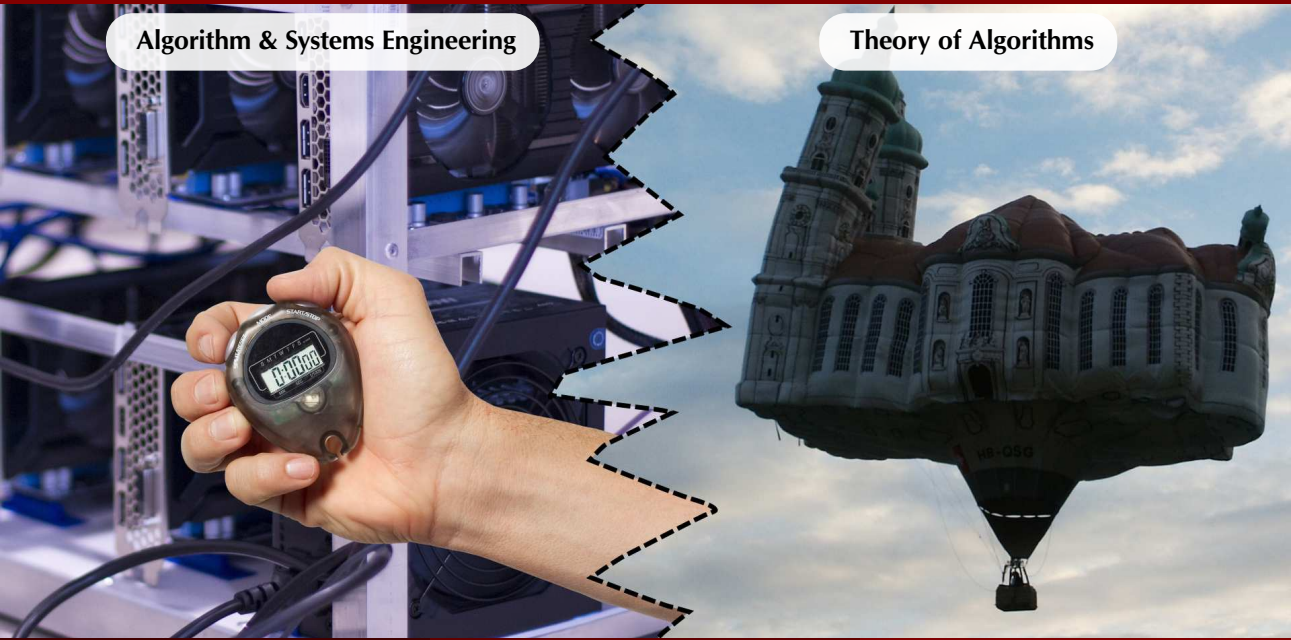
Want:

- Optimal up to lower order terms

practical methods

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

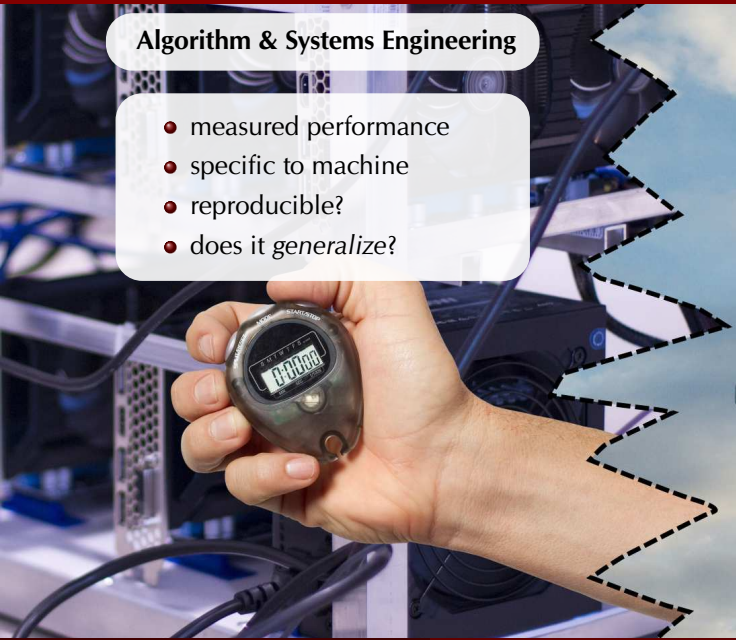
Algorithm Science



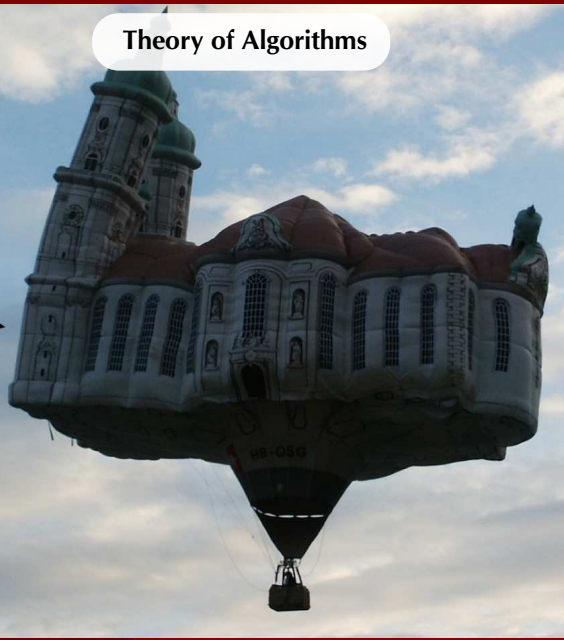
Algorithm Science

Algorithm & Systems Engineering

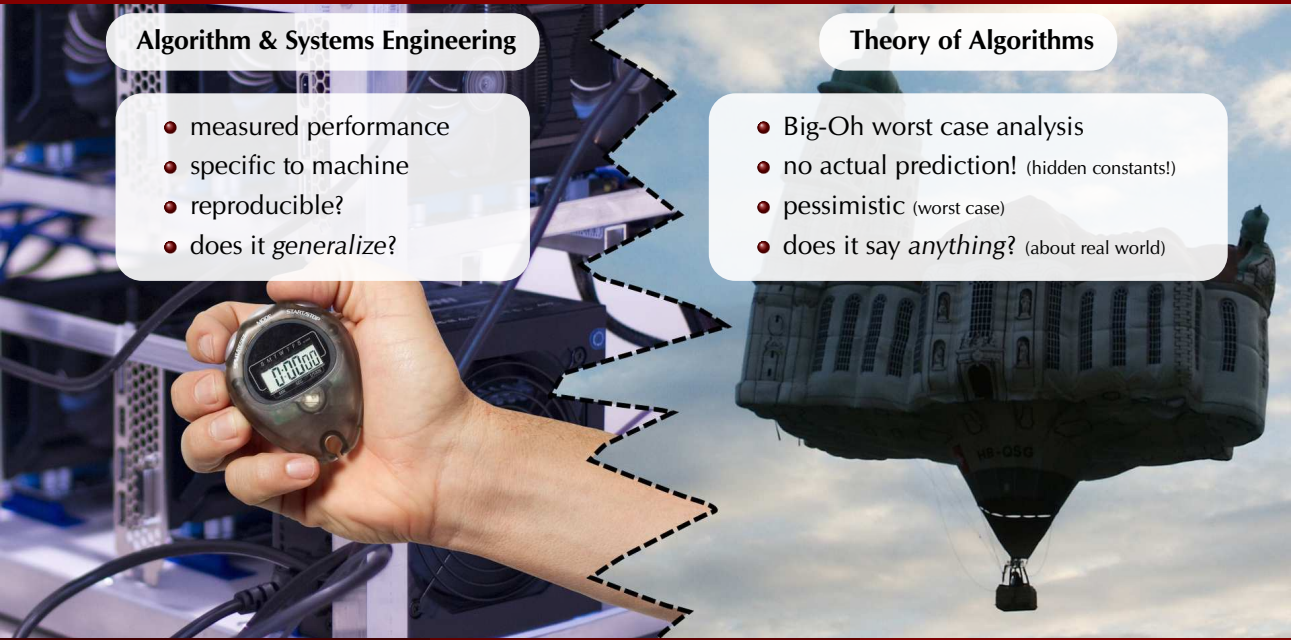
- measured performance
- specific to machine
- reproducible?
- does it *generalize*?



Theory of Algorithms



Algorithm Science



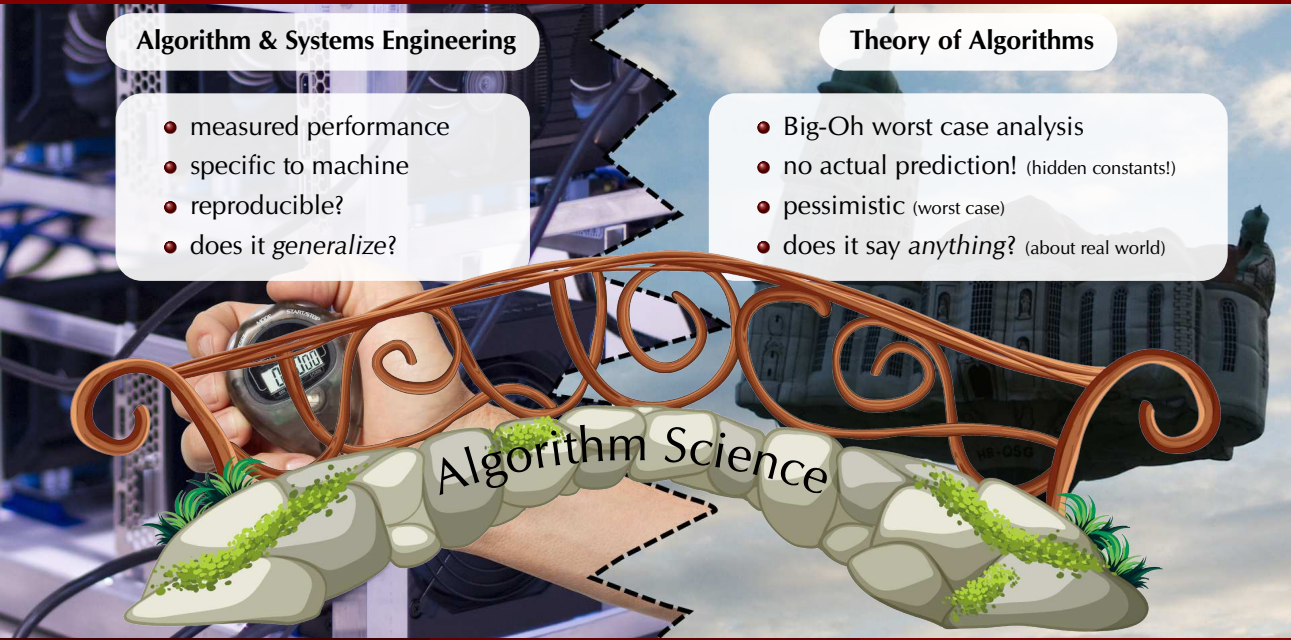
Algorithm & Systems Engineering

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

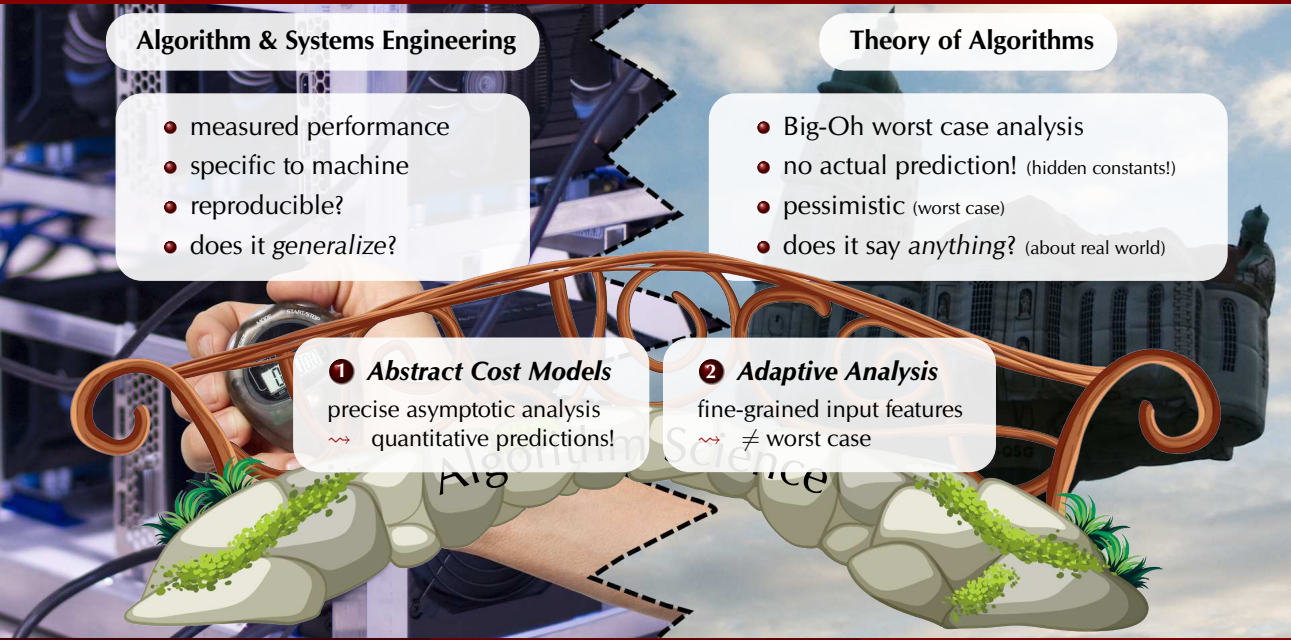
Theory of Algorithms

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)
- pessimistic (worst case)
- does it say *anything*? (about real world)

Algorithm Science



Algorithm Science



Algorithm & Systems Engineering

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Theory of Algorithms

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)
- pessimistic (worst case)
- does it say *anything*? (about real world)

1 ***Abstract Cost Models***

precise asymptotic analysis
⇝ quantitative predictions!

2 ***Adaptive Analysis***

fine-grained input features
⇝ $\neq$ worst case

Algorithm Science

Algorithm & Systems Engineering

- measured performance
- specific to machine
- reproducible?
- does it *generalize*?

Theory of Algorithms

- Big-Oh worst case analysis
- no actual prediction! (hidden constants!)
- pessimistic (worst case)
- does it say *anything*? (about real world)

1 ***Abstract Cost Models***

precise asymptotic analysis
⇝ quantitative predictions!

2 ***Adaptive Analysis***

fine-grained input features
⇝ $\neq$ worst case



Platform-independent performance guarantees

- concrete quantitative model
- validated by experiments

Case Study: Sorting in Java Library

Why Java?

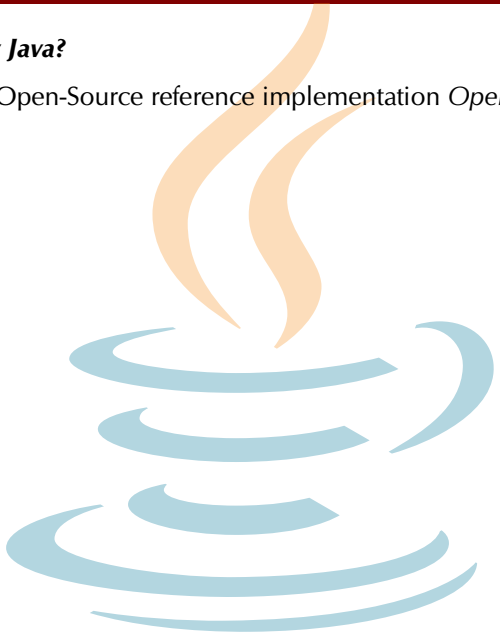


Java™

Case Study: Sorting in Java Library

Why Java?

- Open-Source reference implementation *OpenJDK*

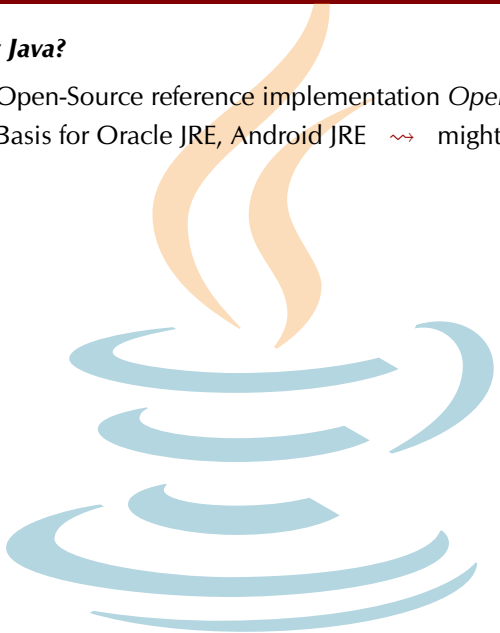


Java™

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!

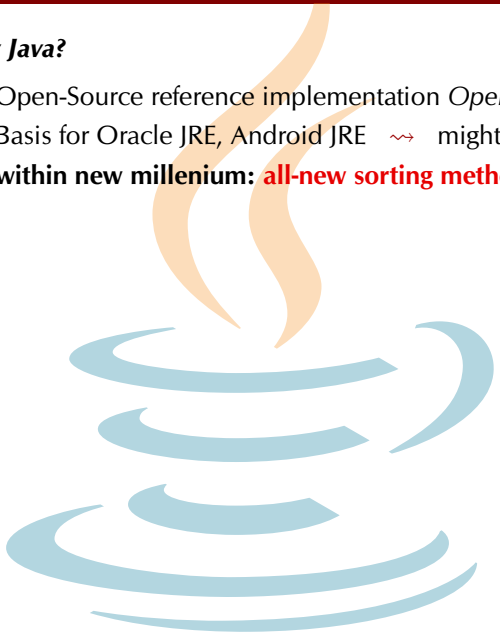


Java™

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!**

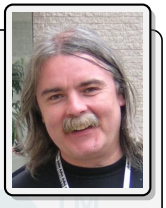
1 ***Dual-Pivot Quicksort***

- 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!**

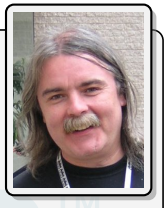
1 *Dual-Pivot Quicksort*

- 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 ⇝ might contain executed algorithms in existence!
- **within new millenium: all-new sorting methods!**

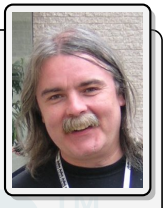
1 *Dual-Pivot Quicksort*

- 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



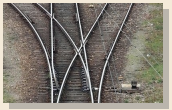
- both algorithms from **developers**

⇝ **Focus:** Benchmarks & Unit-Tests not on proofs or analysis

Outline



1 Context



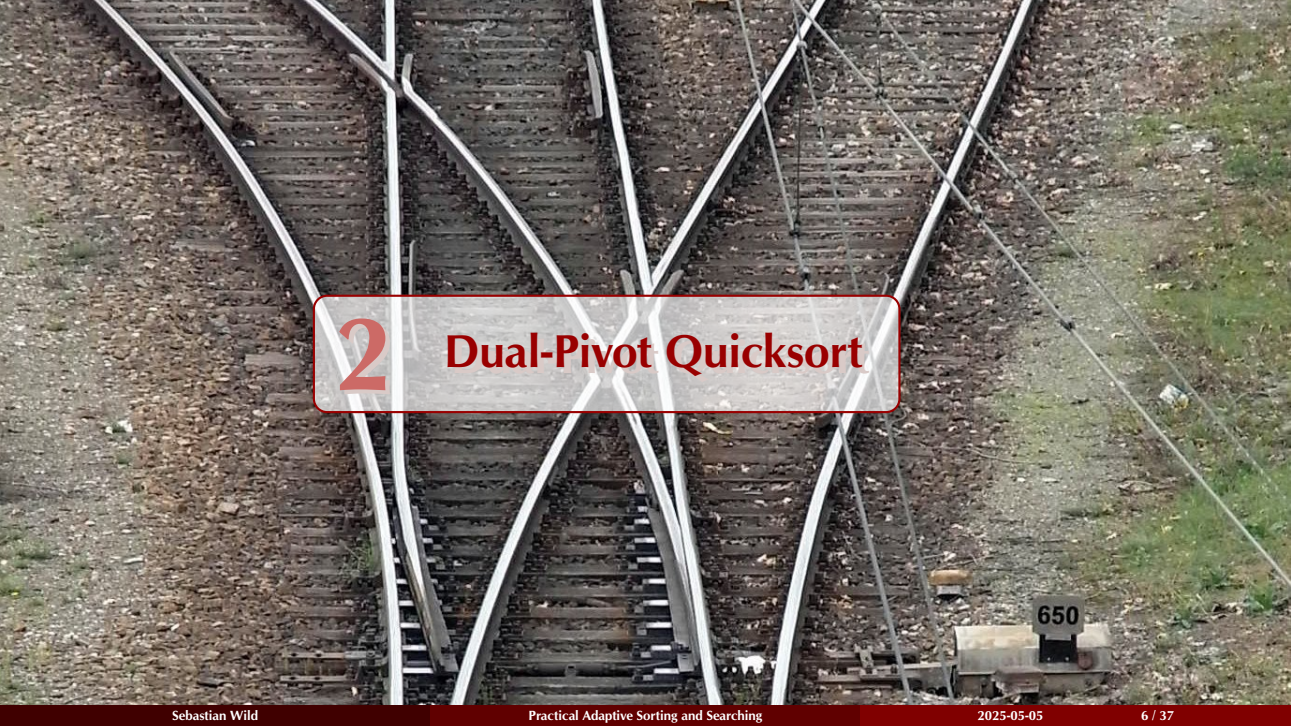
2 Dual-Pivot Quicksort



3 Quicksort with Equal Keys



4 Powersort



2 Dual-Pivot Quicksort

OpenJDK's Dual-Pivot Quicksort

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

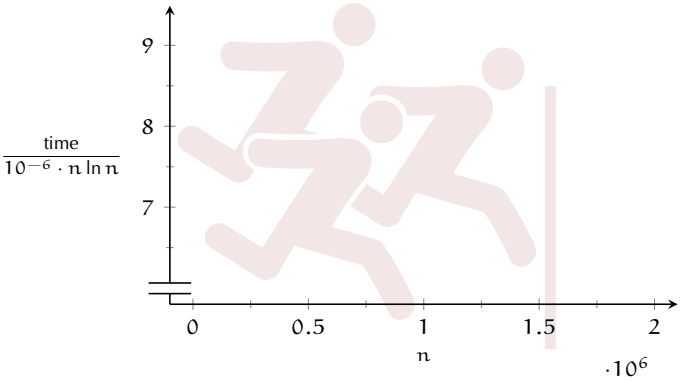
used for primitive types

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*).

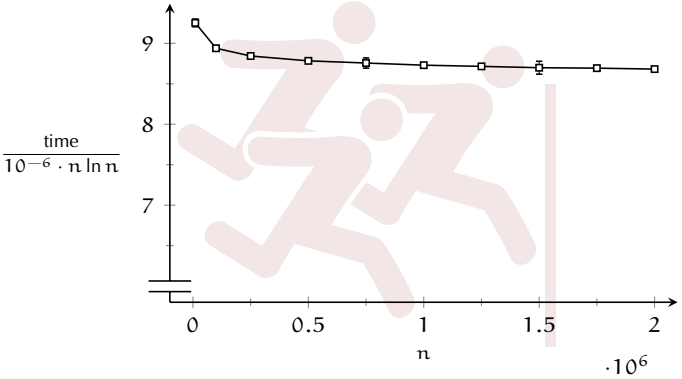
Average of 1000 random permutations per size (`int[]`).

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*).

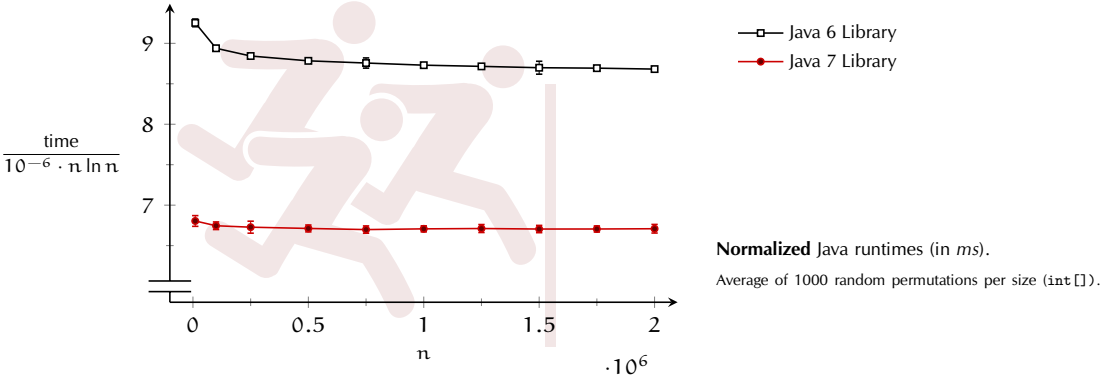
Average of 1000 random permutations per size (`int[]`).

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*).

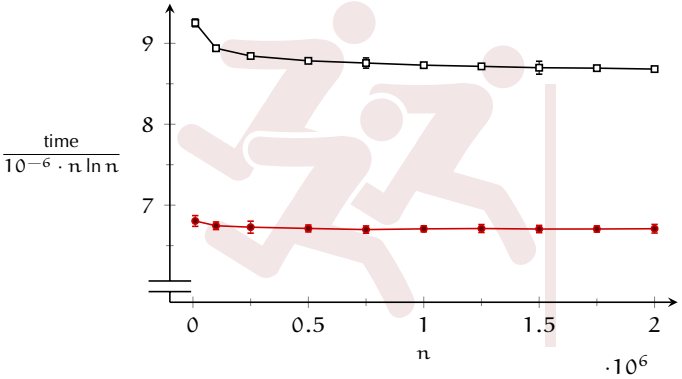
Average of 1000 random permutations per size (`int[]`).

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?*

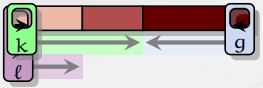
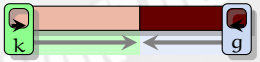
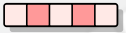


Normalized Java runtimes (in *ms*).

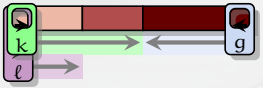
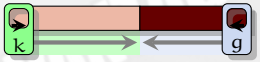
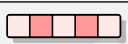
Average of 1000 random permutations per size (`int[]`).

⇝ Another Algorithm-Science case study!

What changes with 2 pivots?

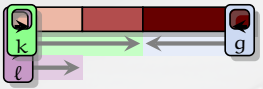
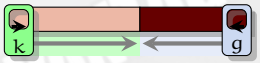
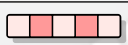
	Dual-Pivot Quicksort	Standard Quicksort	Relative Difference
		 (corrected for pivot Sampling)	
Pivots	 (Tertiles of 5)		

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%**

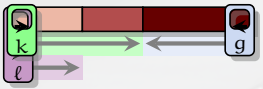
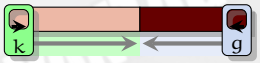
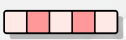
$\cdot n \ln n + O(n)$ *average case results*

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%

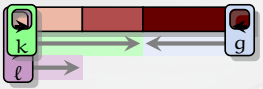
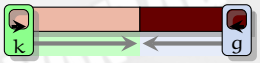
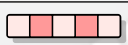
$\cdot n \ln n + O(n)$ *average case results*

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%

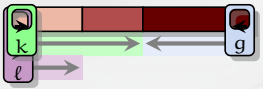
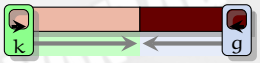
$\cdot n \ln n + O(n)$ *average case results*

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 (≈ 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*

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

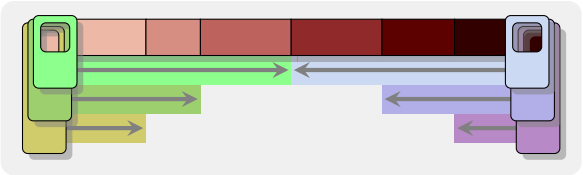
- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right

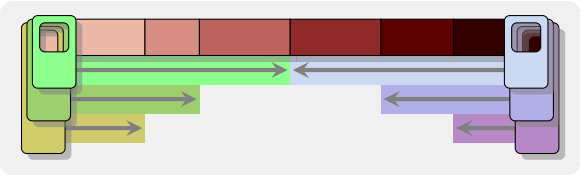


Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right



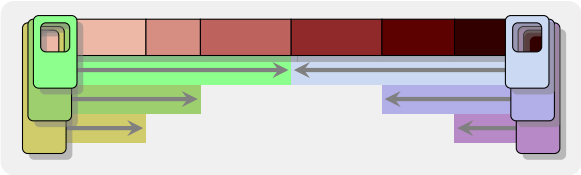
- classify elements using BSTs: λ_k and λ_g

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- $\mathfrak{m}$ segments grow from left, $s - \mathfrak{m}$ from right



- classify elements using BSTs: λ_k and λ_g

Known Quicksort variants

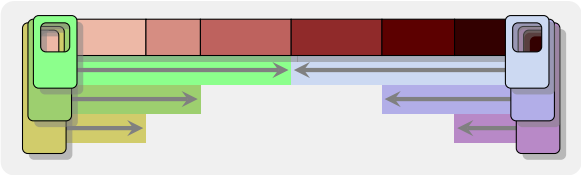
Method	s	$\mathfrak{m}$	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		
Waterloo	4	2		

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way one-pass partitioning*

- **s-way partitioning:** split input into $s \geqslant 2$ parts
- **one pass** over input
- m segments grow from left, $s - m$ from right



- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Known Quicksort variants

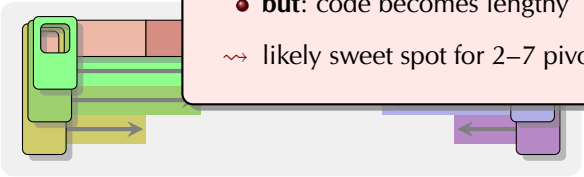
Method	s	m	λ_k	λ_g
Classic	2	1		
Lomuto	2	2		
YBB	3	2		
Waterloo	4	2		

Beyond Dual-Pivot: Multiway Partitioning

Can we do even better than YBB Quicksort?

Setup: *Generic s-way*

- **s-way partitioni**
- **one pass** over in
- m segments gro



- classify elements using BSTs: λ_k and λ_g
- pivots $P_1, \ldots, P_{s-1}$ chosen from sample

Findings for dual-pivot Quicksort are typical:

- **comparisons**: negligible improvements
- **scanned elements**: **significant** improvements
- **but**: code becomes lengthy

⇝ likely sweet spot for 2–7 pivots

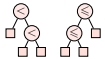
…n Quicksort variants

	s	m	λ_k	λ_g
	2	1		
	2	2		
	3	2		
Waterloo	4	2		

Multiway Quicksort Summary

Multiway Quicksort

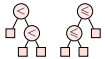
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



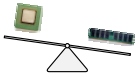
Multiway Quicksort Summary

Multiway Quicksort

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



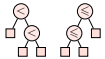
2. evidence for growing weight of memory transfer costs



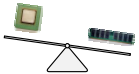
Multiway Quicksort Summary

Multiway Quicksort

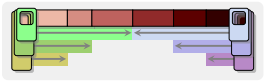
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



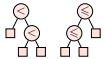
3. **Multiway Quicksort** reduces **memory transfer**



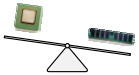
Multiway Quicksort Summary

Multiway Quicksort

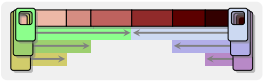
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



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

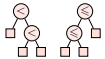


   - No other optimization of Quicksort achieves that

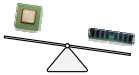
Multiway Quicksort Summary

Multiway Quicksort

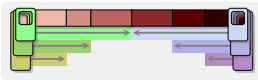
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



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



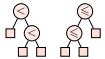
   - No other optimization of Quicksort achieves that

4. *AofA for Algorithm Science!*

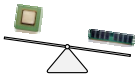
Multiway Quicksort Summary

Multiway Quicksort

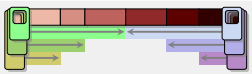
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



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!*

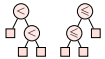
Open Problem

- Automated/Convenient way to determine linear term of cost?

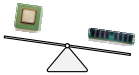
Multiway Quicksort Summary

Multiway Quicksort

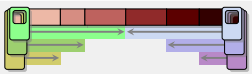
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



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!*

Open Problem

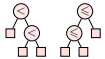
- Automated/Convenient way to determine linear term of cost?

Much more to tell . . .

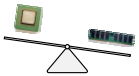
Multiway Quicksort Summary

Multiway Quicksort

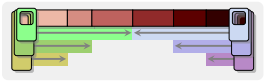
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



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



   - No other optimization of Quicksort achieves that

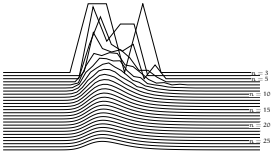
4. *AofA for Algorithm Science!*

Open Problem

- Automated/Convenient way to determine linear term of cost?

Much more to tell …

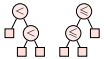
- Limiting **distributions** of costs



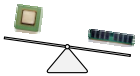
Multiway Quicksort Summary

Multiway Quicksort

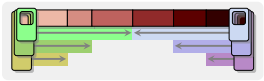
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



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



   - No other optimization of Quicksort achieves that

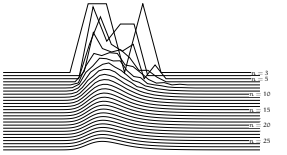
4. *AofA for Algorithm Science!*

Open Problem

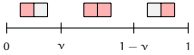
- Automated/Convenient way to determine linear term of cost?

Much more to tell …

- Limiting **distributions** of costs



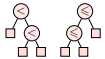
- **Sesquickselect** (cache-efficient selection)



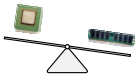
Multiway Quicksort Summary

Multiway Quicksort

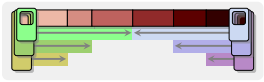
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



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



   - No other optimization of Quicksort achieves that

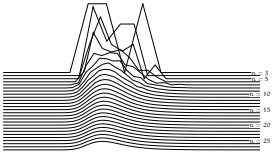
4. *AofA for Algorithm Science!*

Open Problem

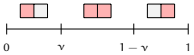
- Automated/Convenient way to determine linear term of cost?

Much more to tell …

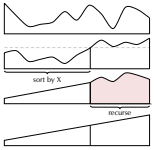
- Limiting **distributions** of costs



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



- ***QuickXsort*** inplace unstable mergesort



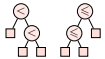
Open Problem

- limit distribution?

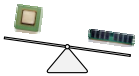
Multiway Quicksort Summary

Multiway Quicksort

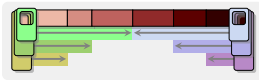
1. tricky **asymmetries** in *YBB Quicksort*
   ⇝ saves a few comparisons



2. evidence for growing weight of memory transfer costs



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



   - No other optimization of Quicksort achieves that

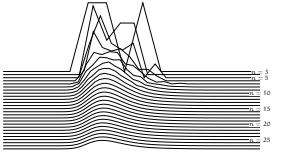
4. *AofA for Algorithm Science!*

Open Problem

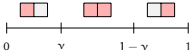
- Automated/Convenient way to determine linear term of cost?

Much more to tell …

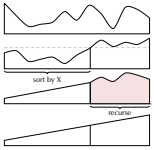
- Limiting **distributions** of costs



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



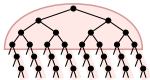
- ***QuickXsort*** inplace unstable mergesort



Open Problem

- limit distribution?

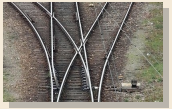
- Intriguing interplay of parameters



Outline



1 Context



2 Dual-Pivot Quicksort



3 Quicksort with Equal Keys



4 Powersort



3 Quicksort with Equal Keys

Adaptive Quicksort

- Extensive literature and results on quicksort

Adaptive Quicksort

- Extensive literature and results on quicksort
- **but:** almost all on **random permutations**!

Adaptive Quicksort

- Extensive literature and results on quicksort
- **but:** almost all on **random permutations**!

Comprehensive literature
on analysis of Quicksort with equal keys:

- Sedgewick 1977:
  Quicksort on Equal Keys
- Sedgewick & Bentley 2002:
  Quicksort is Optimal (Talk at Knuthfest)
- Wild 2018: ANALCO

A bit more on BSTs:

- Burge 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- Kemp 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- Archibald & Clément 2006:
  Average depth in a BST with repeated keys

Adaptive Quicksort

- Extensive literature and results on quicksort
- **but:** almost all on **random permutations**!
- partly justified: should randomize Quicksort!
  (not often done in libraries . . .)
  ⇝ **every** input appears randomly ordered

Comprehensive literature
on analysis of Quicksort with equal keys:

- Sedgewick 1977:
  Quicksort on Equal Keys
- Sedgewick & Bentley 2002:
  Quicksort is Optimal (Talk at Knuthfest)
- Wild 2018: ANALCO

A bit more on BSTs:

- Burge 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- Kemp 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- Archibald & Clément 2006:
  Average depth in a BST with repeated keys

Adaptive Quicksort

- Extensive literature and results on quicksort
- **but:** almost all on **random permutations**!
- partly justified: should randomize Quicksort! (not often done in libraries . . .)
  $\rightsquigarrow$ **every** input appears randomly ordered
- **Catch: Equal keys** don't go away when shuffling input!

Comprehensive literature
on analysis of Quicksort with equal keys:

- Sedgewick 1977:
  Quicksort on Equal Keys
- Sedgewick & Bentley 2002:
  Quicksort is Optimal (Talk at Knuthfest)
- Wild 2018: ANALCO

A bit more on BSTs:

- Burge 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- Kemp 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- Archibald & Clément 2006:
  Average depth in a BST with repeated keys

Adaptive Quicksort

- Extensive literature and results on quicksort
- **but:** almost all on **random permutations**!
- partly justified: should randomize Quicksort! (not often done in libraries . . .)
  $\rightsquigarrow$ **every** input appears randomly ordered
- **Catch: Equal keys** don't go away when shuffling input!

*Here: Quicksort can indeed **exploit** equal keys!*

- a likely key reason its popularity in libraries
- specifically true for dual-pivot quicksort (where it doesn't need special handling)

Comprehensive literature
on analysis of Quicksort with equal keys:

- Sedgewick 1977: *Quicksort on Equal Keys*
- Sedgewick & Bentley 2002: *Quicksort is Optimal* (Talk at Knuthfest)
- Wild 2018: ANALCO

A bit more on BSTs:

- Burge 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- Kemp 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- Archibald & Clément 2006: *Average depth in a BST with repeated keys*

Equal Keys Model

Assumptions:

1. **Input:**

Equal Keys Model

Assumptions:

1. **Input:** (A)



Multiset Model:
Random permutation $U_1, \ldots, U_n$ of **fixed multiset**
$x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$

Equal Keys Model

Assumptions:

1. **Input:** **(A)**



Multiset Model:

Random permutation $U_1, \dots, U_n$ of **fixed multiset**

$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$

profile $\vec{x}$ of input

Equal Keys Model

Assumptions:

1. **Input:**

(A) 

Multiset Model:

Random permutation $U_1, \ldots, U_n$ of **fixed multiset**

$x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$

profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:

$U_1, \ldots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$

$\vec{q} = (q_1, \ldots, q_u)$ a **fixed** universe **distribution**

Equal Keys Model

Assumptions:

1. **Input:**

(A) 

Multiset Model:

Random permutation $U_1, \ldots, U_n$ of **fixed multiset**

$x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$

profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:

$U_1, \ldots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$

$\vec{q} = (q_1, \ldots, q_u)$ a **fixed** universe **distribution**

Equal Keys Model

Assumptions:

1. **Input:**

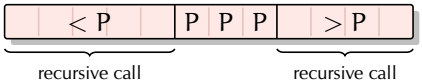
   (A) 

   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
   ↖ profile $\vec{x}$ of input

   (B) 

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning



Equal Keys Model

Assumptions:

1. **Input:**

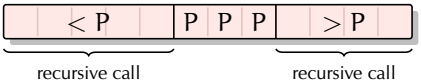
   (A) 

   Multiset Model:
   Random permutation $U_1, \ldots, U_n$ of **fixed multiset**
   $x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$
   ↖ profile $\vec{x}$ of input

   (B) 

   Discrete i.i.d. Model:
   $U_1, \ldots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
   $\vec{q} = (q_1, \ldots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed



Equal Keys Model

Assumptions:

1. **Input:**

(A) 

Multiset Model:

Random permutation $U_1, \ldots, U_n$ of **fixed multiset**

$x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$

↖ profile $\vec{x}$ of input

(B) 

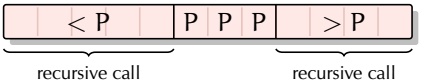
Discrete i.i.d. Model:

$U_1, \ldots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$

$\vec{q} = (q_1, \ldots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



Equal Keys Model

Assumptions:

1. **Input:**

   (A) 

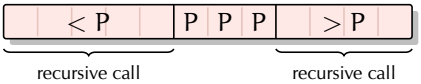
   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
   ↖ profile $\vec{x}$ of input

   (B) 

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\text{iid}}{=} \text{Mult}(n, \vec{q})$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



3. **Cost:** # ternary comparisons (outcome </>/=)

Equal Keys Model

Assumptions:

1. **Input:**

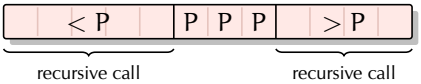
(A) 

Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

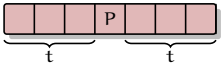
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed
   - ⇝ subproblems of **same** type, (restricted to a sub-universe)



3. **Cost:** # ternary comparisons (outcome </>/=)

Median-of-(2t+1) Quicksort:

Example:
$t = 3$



Equal Keys Model

Assumptions:

1. **Input:**

   (A) 

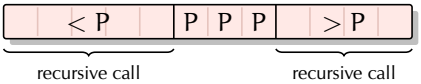
   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
   ↖ profile $\vec{x}$ of input

   (B) 

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \overset{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)

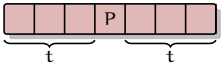


3. **Cost:** # ternary comparisons (outcome </>/=)

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$

Example:
$t = 3$



Equal Keys Model

Assumptions:

1. **Input:**

   (A) 

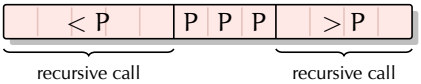
   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
   ↖ profile $\vec{x}$ of input

   (B) 

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)

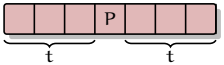


3. **Cost:** # ternary comparisons (outcome </>/=)

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible) ← not today

Example:
$t = 3$



Analysis for classic Quicksort

Classic Quicksort (without pivot sampling):

Analysis for classic Quicksort

Classic Quicksort (without pivot sampling):

Analysis of Quicksort with equal keys

1. Define $C(x_1,\dots,x_n) = C(1,n)$ to be the mean # compares to sort the file

$$C(1,n) = N - 1 + \frac{1}{N}\sum_{1\le j\le n} x_j(C(1,j-1) + C(j+1,n))$$

2. Multiply both sides by $N = x_1 + \dots + x_n$

$$NC(1,n) = N(N-1) + \sum_{1\le j\le n} x_j C(1,j-1) + \sum_{1\le j\le n} x_j C(j+1,n)$$

3. Subtract same equation for $x_2,\dots,x_n$ and let $D(1,n) = C(1,n) - C(2,n)$

$$(x_1 + \dots + x_n)D(1,n) = x_1^2 - x_1 + 2x_1(x_2 + \dots + x_n) + \sum_{2\le j\le n} x_j D(1,j-1)$$

4. Subtract same equation for $x_1,\dots,x_{n-1}$

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,n-1)$$

Analysis for classic Quicksort

Classic Quicksort (without pivot sampling):

Analysis of Quicksort with equal keys

1. Define $C(x_1,\dots,x_n) = C(1,n)$ to be the mean # compares to sort

$$C(1,n) = N - 1 + \frac{1}{N}\sum_{1\le j\le n} x_j\big(C(1,j-1) + C(j+1,n)\big)$$

2. Multiply both sides by $N = x_1 + \dots + x_n$

$$NC(1,n) = N(N-1) + \sum_{1\le j\le n} x_j C(1,j-1) + \sum_{1\le j\le n} x_j C(j+1,n)$$

3. Subtract same equation for $x_2,\dots,x_n$ and let $D(1,n) = C(1,n) - C$

$$(x_1 + \dots + x_n)D(1,n) = x_1^2 - x_1 + 2x_1(x_2 + \dots + x_n) + \sum_{2\le j\le n} x_j D(1,j$$

4. Subtract same equation for $x_1,\dots,x_{n-1}$

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,$$

Analysis of Quicksort with equal keys (cont.)

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,n-1)$$

5. Simplify, divide both sides by $N = x_1 + \dots + x_n$

$$D(1,n) = D(1,n-1) + \frac{2x_1x_n}{x_1 + \dots + x_n}$$

6. Telescope (twice)

$$C(1,n) = N - n + \sum_{1\le k<j\le n} \frac{2x_kx_j}{x_k + \dots + x_j}$$

THEOREM. Quicksort (with 3-way partitioning, randomized) uses

$N - n + 2QN$ compares (where $Q = \sum_{1\le k<j\le n} \frac{p_kp_j}{p_k + \dots + p_j}$, with $p_i = x_i/N$)

to sort an $(x_1,\dots,x_n)$-file, on the average .

Analysis for classic Quicksort

Classic Quicksort (without pivot sampling): Expected comparisons expressible **exactly**.

Analysis of Quicksort with equal keys

1. Define $C(x_1,\dots,x_n) = C(1,n)$ to be the mean # compares to sor

$$C(1,n) = N - 1 + \frac{1}{N}\sum_{1\le j\le n} x_j\,(C(1,j-1) + C(j+1,n))$$

2. Multiply both sides by $N = x_1 + \dots + x_n$

$$NC(1,n) = N(N-1) + \sum x_j C(1,j-1) + \sum x_j C(j+1,n)$$

3. Su

4. Su

Analysis of Quicksort with equal keys (cont.)

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1 x_n + x_n D(1,n-1)$$

5. Simplify, divide both sides by $N = x_1 + \dots + x_n$

$$D(1,n) = D(1,n-1) + \frac{2x_1 x_n}{x_1 + \dots + x_n}$$

$$1 \le k < j \le n$$

THEOREM. Quicksort (with 3-way partitioning, randomized) uses

$$N - n + 2QN \quad \text{compares (where } Q = \sum_{1\le k<j\le n} \frac{p_k p_j}{p_k + \dots + p_j}, \text{ with } p_i = x_i/N)$$

to sort an $(x_1,\dots,x_n)$–file, on the average .

The conjecture of Sedgewick and Bentley

Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.

The conjecture of Sedgewick and Bentley

Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.

The conjecture of Sedgewick and Bentley

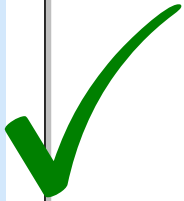
Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.



The conjecture of Sedgewick and Bentley

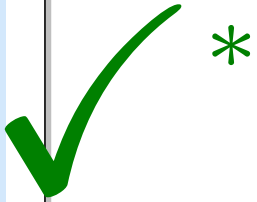
Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.



* subject to some restrictions

Quicksort & search trees

Classic Fact:

- Recursion Tree of Quicksort = Naturally grown BST from input

Quicksort & search trees

Classic Fact:

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST

Quicksort & search trees

Classic Fact:

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

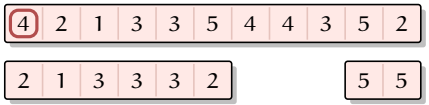


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

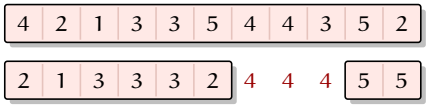


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

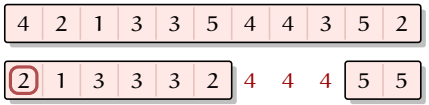


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

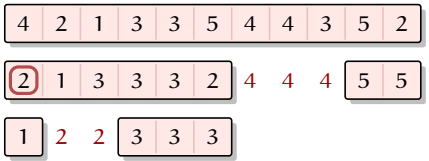


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

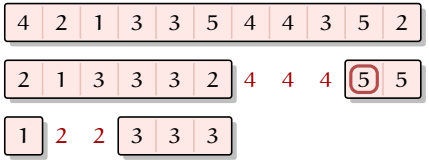


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

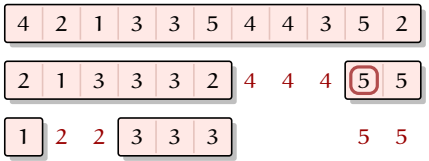


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

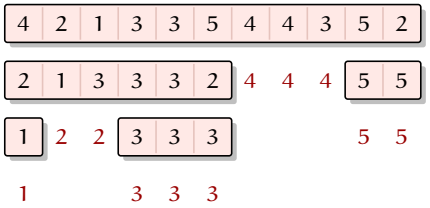


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

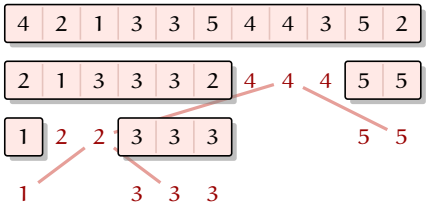


Quicksort & search trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

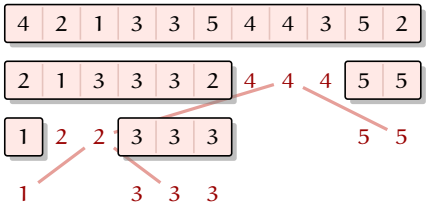


Quicksort & search trees

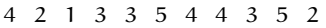
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

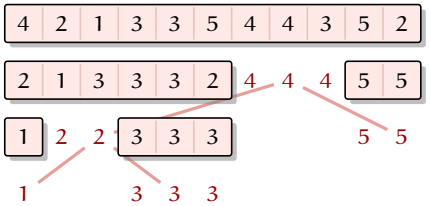


Quicksort & search trees

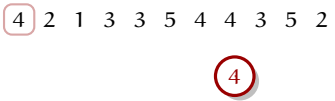
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

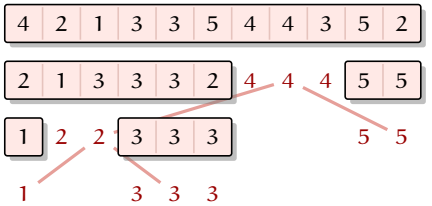


Quicksort & search trees

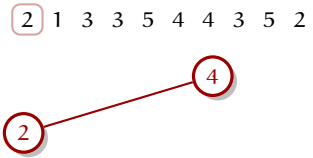
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

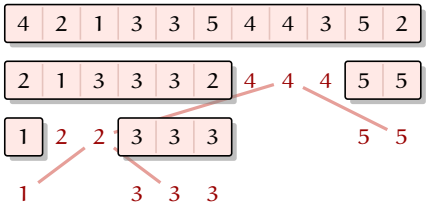


Quicksort & search trees

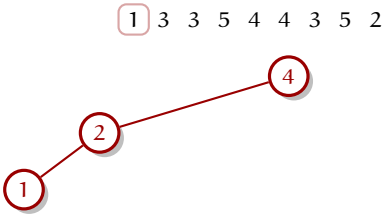
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

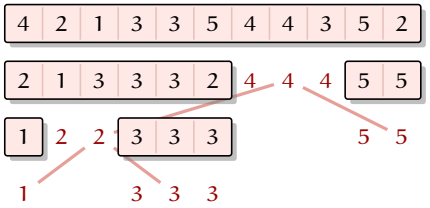


Quicksort & search trees

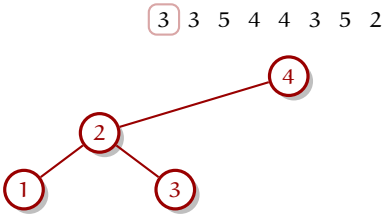
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

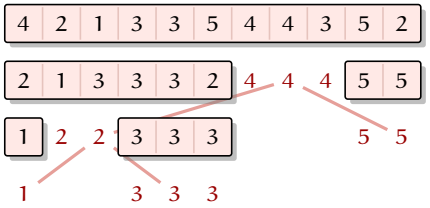


Quicksort & search trees

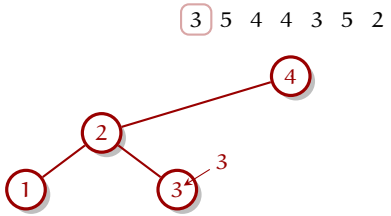
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

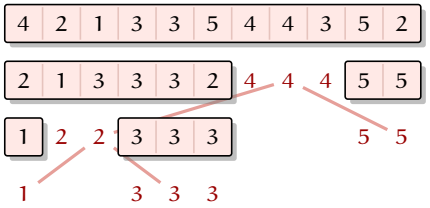


Quicksort & search trees

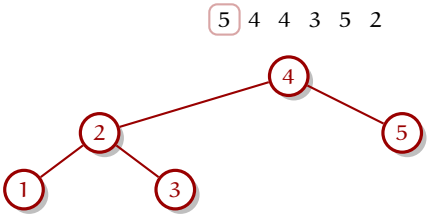
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

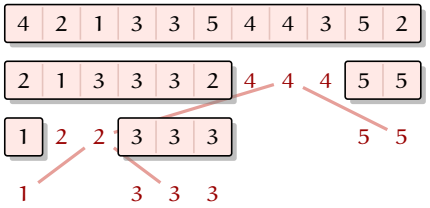


Quicksort & search trees

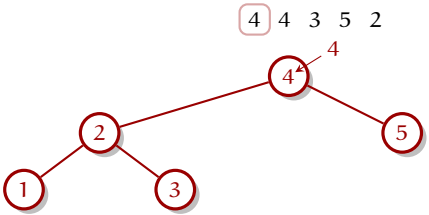
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

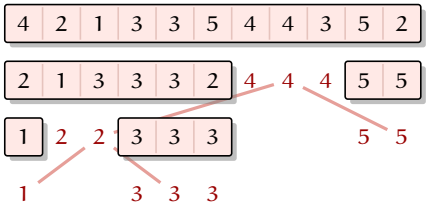


Quicksort & search trees

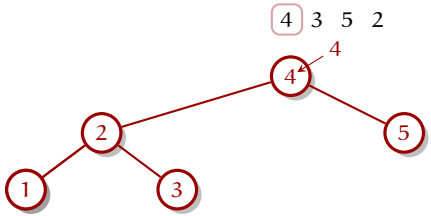
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

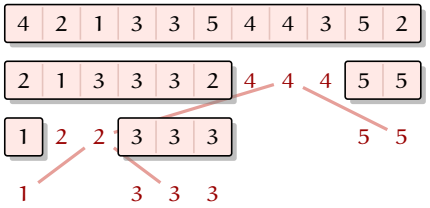


Quicksort & search trees

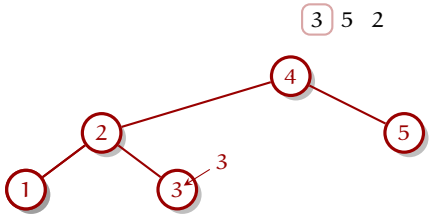
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

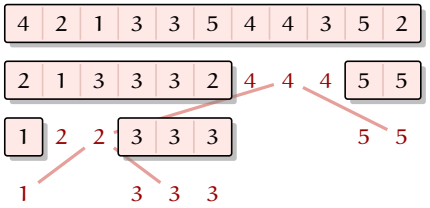


Quicksort & search trees

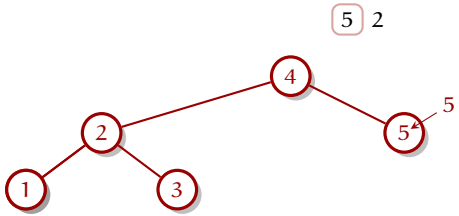
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

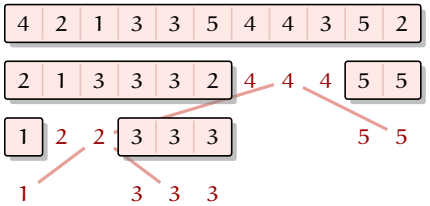


Quicksort & search trees

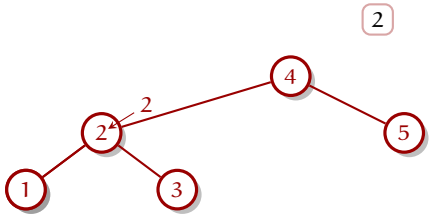
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

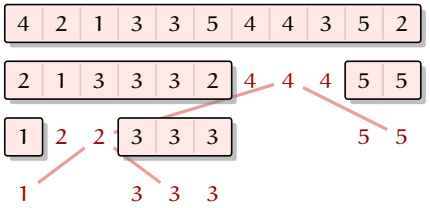


Quicksort & search trees

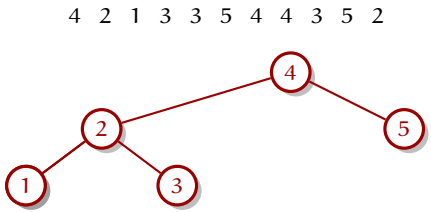
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



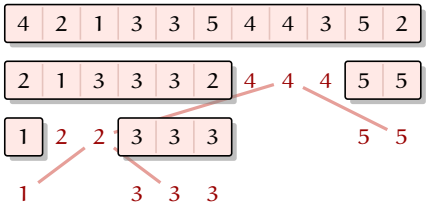
⇝ Equivalence holds also with duplicates.

Quicksort & search trees

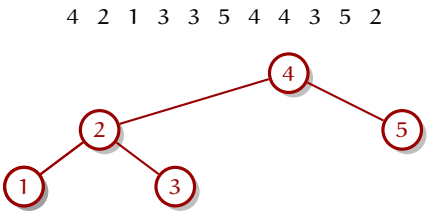
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



⇝ Equivalence holds also with duplicates.

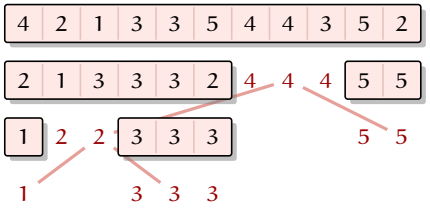
This was only basic Quicksort . . .

Quicksort & search trees

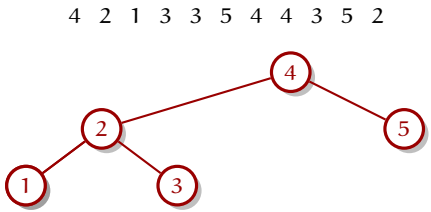
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



⇝ Equivalence holds also with duplicates.

*This was only basic Quicksort . . . how about **pivot sampling**?*

Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.

Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

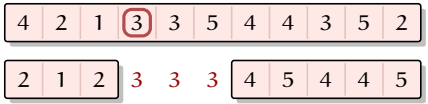


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

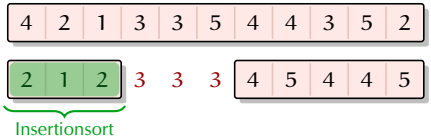


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

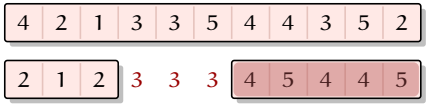


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

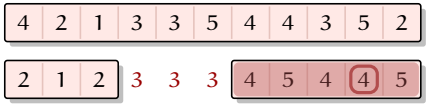


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

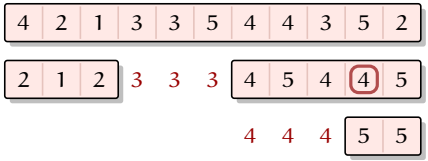


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

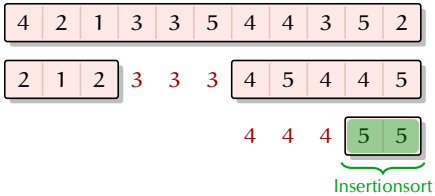


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**

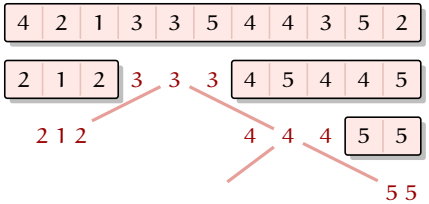


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**

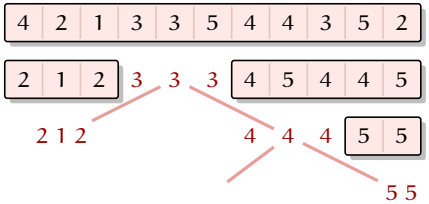


Fringe-balanced trees

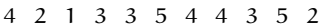
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

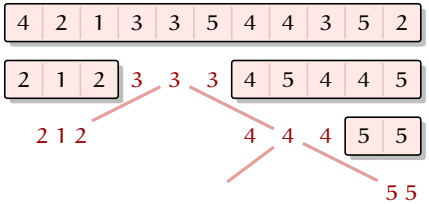


Fringe-balanced trees

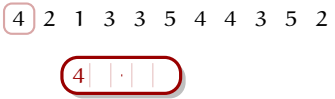
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

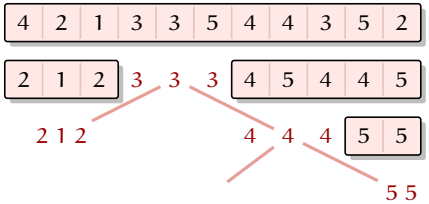


Fringe-balanced trees

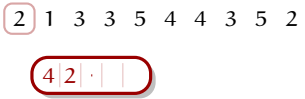
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

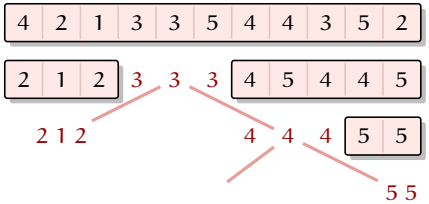


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

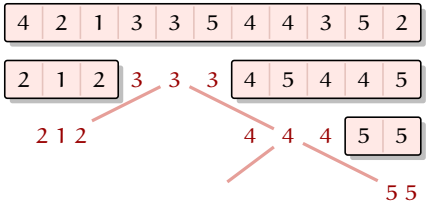


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

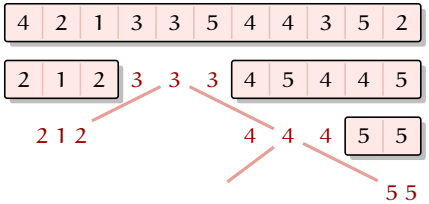


Fringe-balanced trees

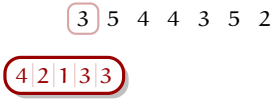
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

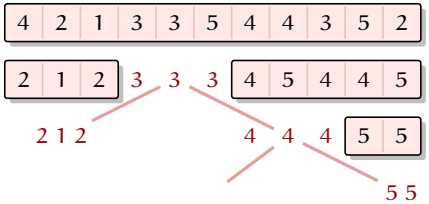


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

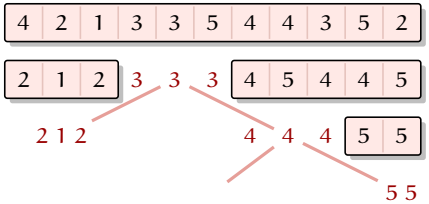


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

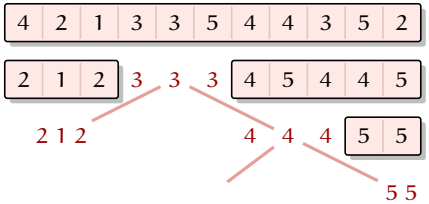


Fringe-balanced trees

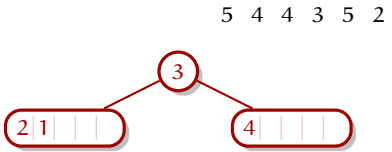
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

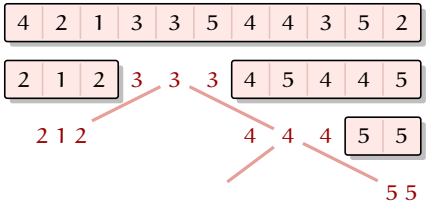


Fringe-balanced trees

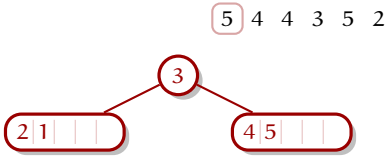
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

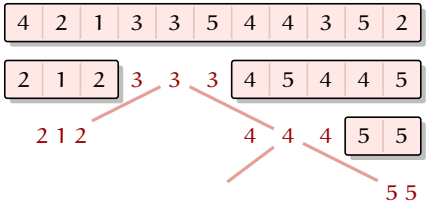


Fringe-balanced trees

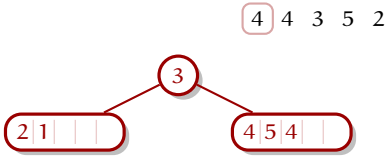
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

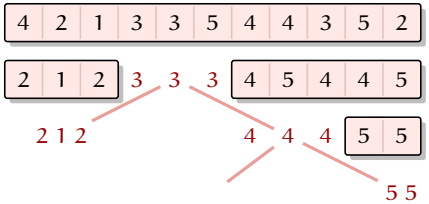


Fringe-balanced trees

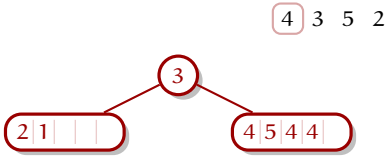
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

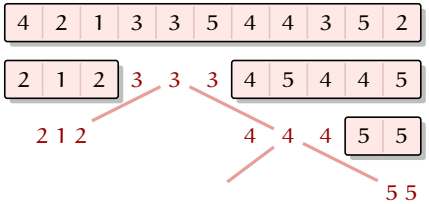


Fringe-balanced trees

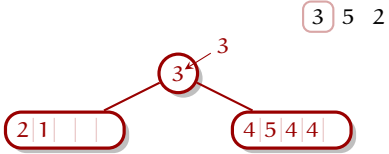
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

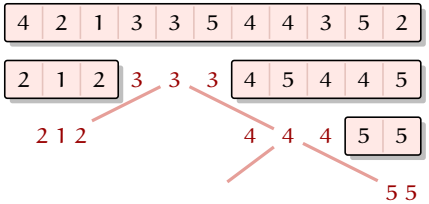


Fringe-balanced trees

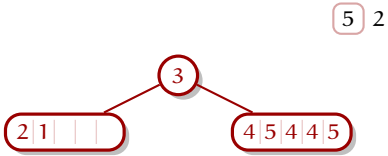
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

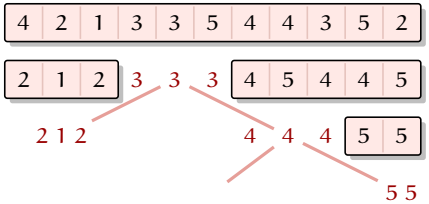


Fringe-balanced trees

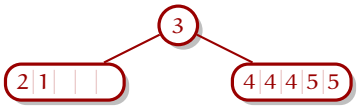
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree



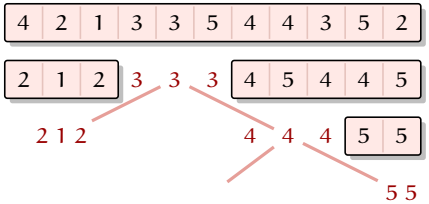
2

Fringe-balanced trees

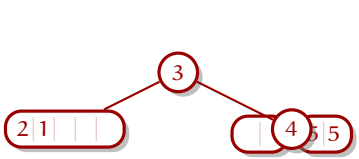
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree



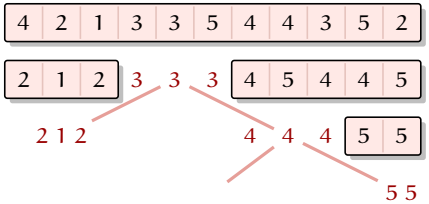
2

Fringe-balanced trees

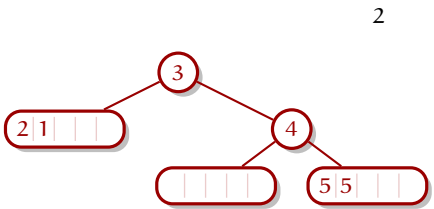
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

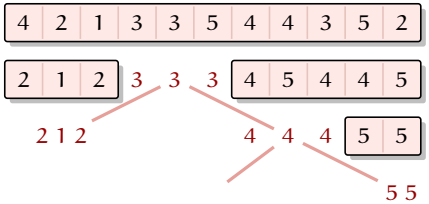


Fringe-balanced trees

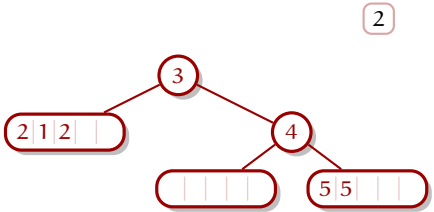
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree

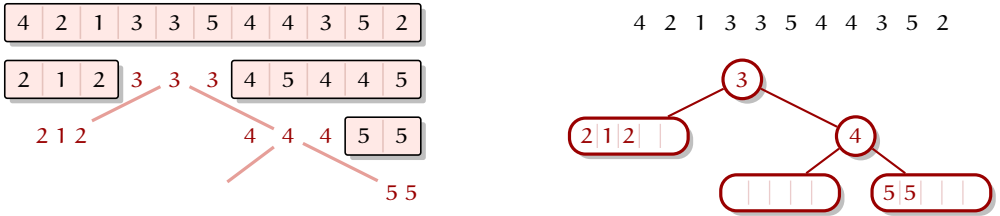


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**



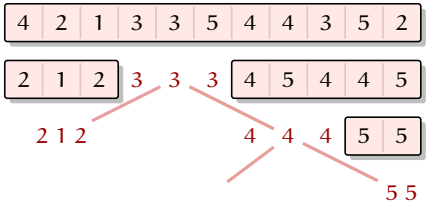
$\rightsquigarrow$ **Correspondence extends to**

Fringe-balanced trees

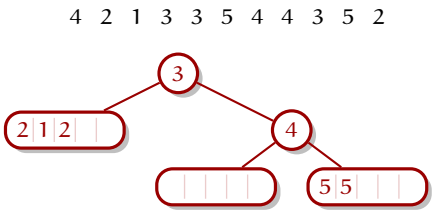
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

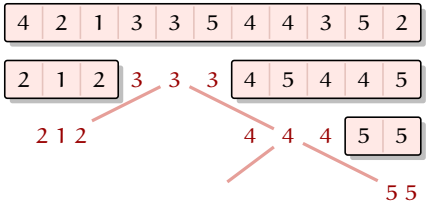
- Pivot Sampling (any scheme, not only median)

Fringe-balanced trees

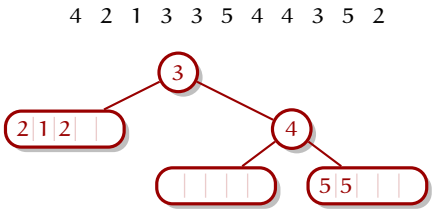
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

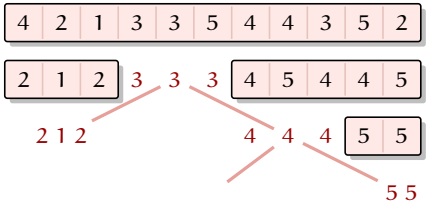
- Pivot Sampling (any scheme, not only median)
- (**s-way Partitioning** $\rightsquigarrow$ s-ary search trees) ← not today

Fringe-balanced trees

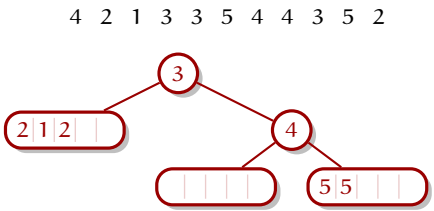
***k*-Fringe-Balanced Search Trees:**

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- (**s-way Partitioning** $\rightsquigarrow$ s-ary search trees) ← not today

$\rightsquigarrow$ Analyze search trees instead of Quicksort.

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort

Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3

Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3

Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

($k = 1$)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Tree-growing and searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - ⇝ direct analysis no simpler than for Quicksort

(k = 1)

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



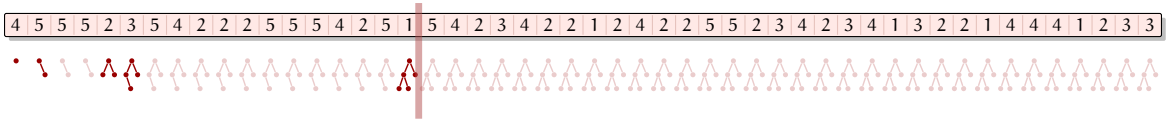
Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
⇝ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times**
(up to k duplicates in buffer)

Tree-growing and searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
- ⇝ direct analysis no simpler than for Quicksort

($k = 1$)



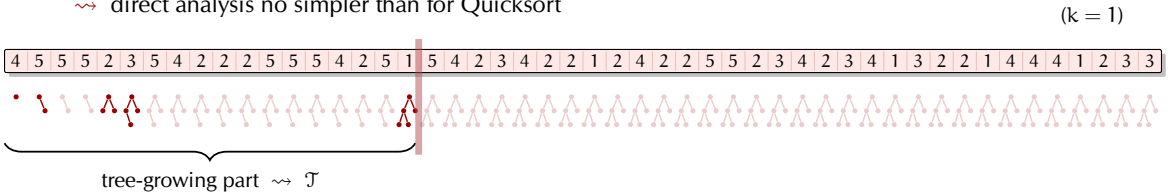
Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
⇝ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times**
(up to k duplicates in buffer)

Split input into **tree-growing** part and **searching** part:

Tree-growing and searching

- $\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times**
(up to k duplicates in buffer)

Split input into **tree-growing** part and **searching** part:

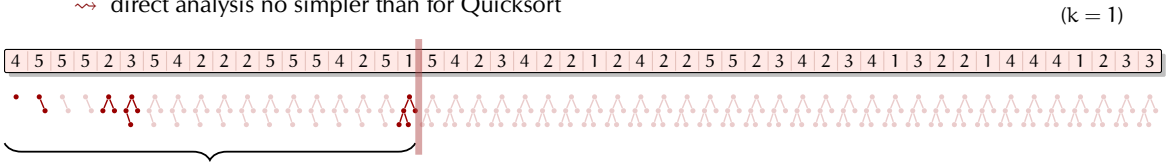
1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times** (up to k duplicates in buffer)



Split input into **tree-growing** part (hopefully a **short** prefix!) and **searching** part:

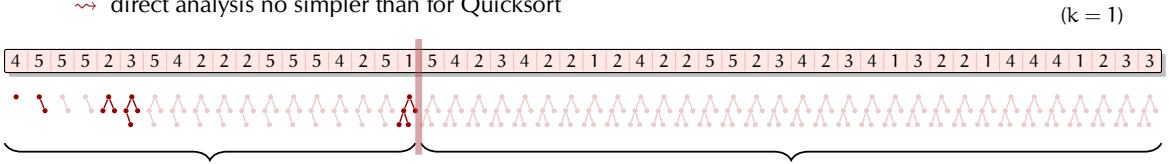
1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times**
(up to k duplicates in buffer)

hopefully a **short** prefix!

Split input into **tree-growing** part and **searching** part:

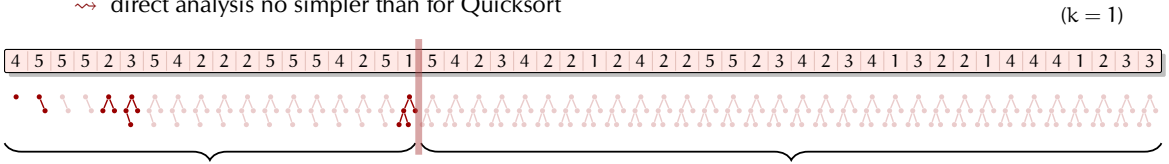
1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ **times**
(up to k duplicates in buffer)

Split input into **tree-growing** part (hopefully a **short** prefix!) and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

Two parts of input always dependent! (profiles must **sum to** $\vec{x}$)

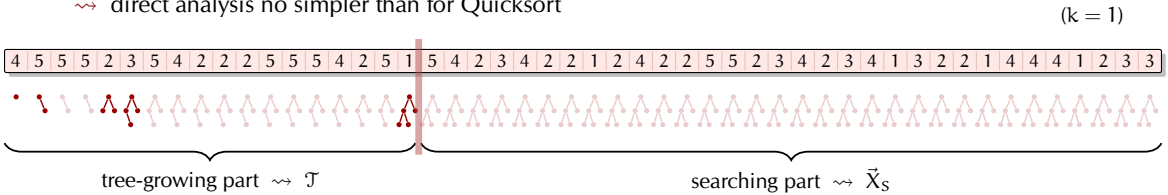
Two parts are **independent** (i.i.d.!)

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ **times**
(up to k duplicates in buffer)

Split input into **tree-growing** part (hopefully a **short** prefix!) and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

~~Two parts of input always dependent! (profiles must **sum to** $\vec{x}$)~~

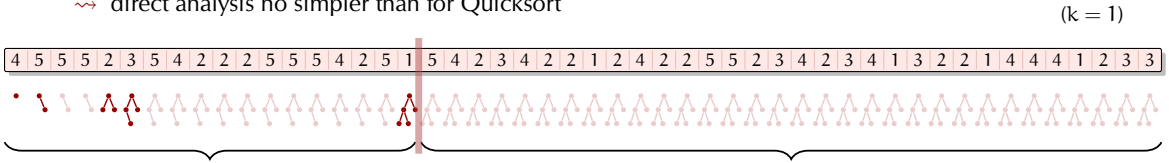
Two parts are **independent** (i.i.d.!)

Tree-growing and searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{u} = (u_1, \dots, u_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{u}$ (Recall: $\mathcal{T}$ is *built* from $\vec{u}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ times
(up to k duplicates in buffer)

hopefully a **short** prefix!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

~~Two parts of input always dependent! (profiles must **sum to** $\vec{x}$)~~

Two parts are **independent** (i.i.d.!)

Results in the i.i.d. Model



Two pieces to the puzzle

Results in the i.i.d. Model



Two pieces to the puzzle

- **Separation Theorem**:
  Quicksort costs
  - in the **i.i.d. model** 
  - with **"many duplicates"**
    ($\Omega(n^{\varepsilon})$ duplicates of each value in expectation)

  are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

  (show: tree is "saturated" after $n^{1-\varepsilon}$ elements whp.)

Results in the i.i.d. Model



Two pieces to the puzzle

- **Separation Theorem**:
  Quicksort costs
  - in the **i.i.d. model** 
  - with **"many duplicates"**
    ($\Omega(n^\varepsilon)$ duplicates of each value in expectation)

  are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

  (show: tree is "saturated" after $n^{1-\varepsilon}$ elements whp.)

- **Average search costs**
  in saturated k-fringe-balanced trees:

(as $\mathcal{H} \to \infty$)

$$\alpha(\vec{q}) \;=\; \alpha_k \cdot \mathcal{H} \;\pm\; O\big(\mathcal{H}^{\frac{t+2}{t+3}} \log \mathcal{H}\big)$$

  - $\mathcal{H} = \mathcal{H}_{\mathsf{Id}}(\vec{q})$
  - $\alpha_k = \dfrac{\ln 2}{H_{k+1} - H_{t+1}}$

  (Compare recurrence of entropy to search cost, tour de force to bound α in terms of $\mathcal{H}$)

Results in the i.i.d. Model



Two pieces to the puzzle

Separation Theorem:
Quicksort costs
- in the **i.i.d. model** 
- with **"many duplicates"**
  ($\Omega(n^\varepsilon)$ duplicates of each value in expectation)

are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

(show: tree is "saturated" after $n^{1-\varepsilon}$ elements whp.)

Average search costs
in saturated k-fringe-balanced trees:

(as $\mathcal{H} \to \infty$)

$$\alpha(\vec{q}) \;=\; \alpha_k \cdot \mathcal{H} \;\pm\; O\big(\mathcal{H}^{\frac{t+2}{t+3}} \log \mathcal{H}\big)$$

- $\mathcal{H} = \mathcal{H}_{\mathsf{Id}}(\vec{q})$
- $\alpha_k = \dfrac{\ln 2}{H_{k+1} - H_{t+1}}$

(Compare recurrence of entropy to search cost, tour de force to bound α in terms of $\mathcal{H}$)

$\rightsquigarrow$

Quicksort Costs (i.i.d. model)

Under the assumptions above, we have for any $\delta \in \big(0, \frac{1}{t+3}\big)$
$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha_k \mathcal{H}_{\mathsf{Id}}(\vec{q}) \cdot n \;\pm\; O\big((\mathcal{H}(\vec{q})^{1-\delta} + 1)n\big)$.

Quicksort with Equal Keys Summary

Findings

- Partial Answer to **conjecture of Sedgewick & Bentley** (for "many duplicates"):
  - $\rightsquigarrow$ Same relative speedup as for random permutations.

  Median-of-k *Quicksort approaches lower bound for* $k \to \infty$.

Quicksort with Equal Keys Summary

Findings

- Partial Answer to **conjecture of Sedgewick & Bentley** (for "many duplicates"):
  - $\rightsquigarrow$ Same relative speedup as for random permutations.

  Median-of-k *Quicksort approaches lower bound for* $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$

Quicksort with Equal Keys Summary

Findings

- Partial Answer to **conjecture of Sedgewick & Bentley** (for "many duplicates"):
  - $\rightsquigarrow$ Same relative speedup as for random permutations.

  Median-of-k *Quicksort approaches lower bound for* $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - extension for multiway partitioning

Quicksort with Equal Keys Summary

Findings

- Partial Answer to **conjecture of Sedgewick & Bentley** (for "many duplicates"):
  - $\rightsquigarrow$ Same relative speedup as for random permutations.

  Median-of-k *Quicksort approaches lower bound for* $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - extension for multiway partitioning
  - better error bounds

Quicksort with Equal Keys Summary

Findings

- Partial Answer to **conjecture of Sedgewick & Bentley** (for "many duplicates"):
  - $\rightsquigarrow$ Same relative speedup as for random permutations.

  Median-of-k *Quicksort approaches lower bound for* $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - extension for multiway partitioning
  - better error bounds

Open Problem

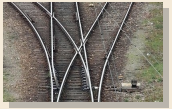
- Get rid of "many duplicates" restriction

  … by going back to fixed profile model?

Outline



1 Context



2 Dual-Pivot Quicksort



3 Quicksort with Equal Keys

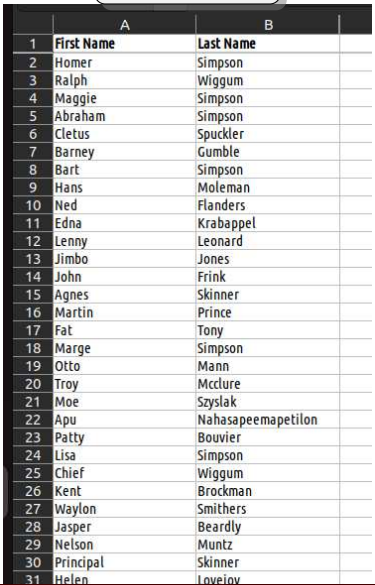


4 Powersort


4 Powersort

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

	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

sorted by Last Name

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

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**

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

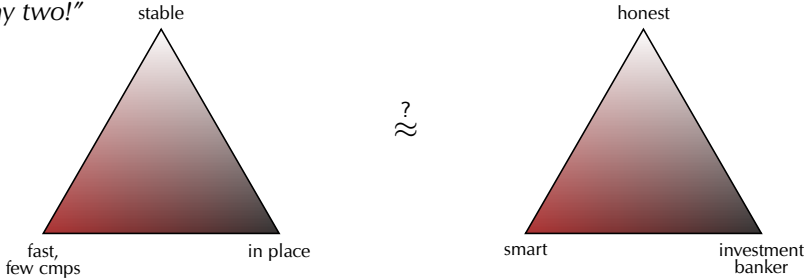
Sorting Trade-offs

- *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)

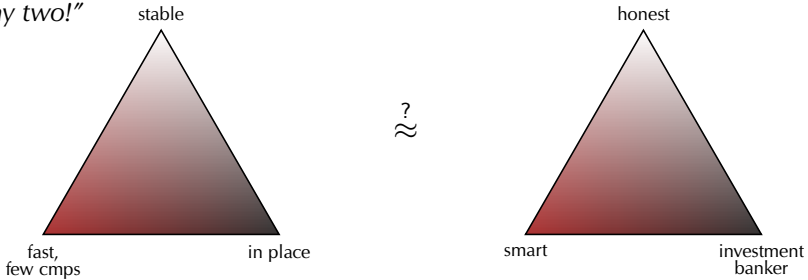
"Pick any two!"



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!"

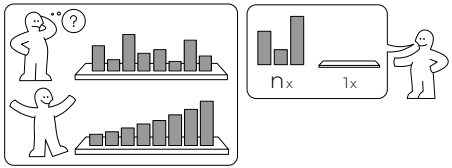


- Python has lots of objects and pointers anyways … "in-place" property least painful to sacrifice

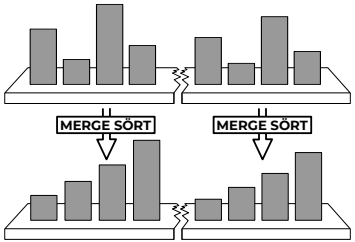
⇝ Mergesort!

MERGE SÖRT

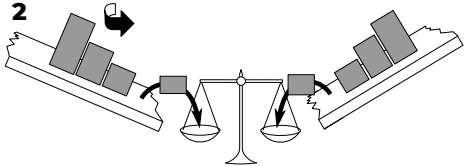
idea-instructions.com/merge-sort/
v1.2, CC by-nc-sa 4.0



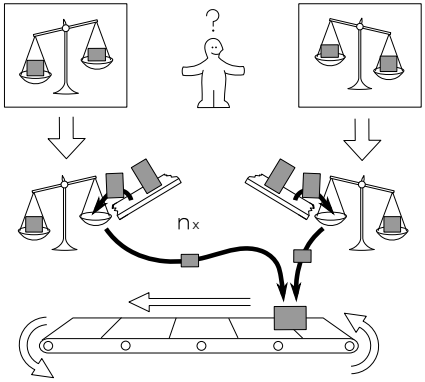
1



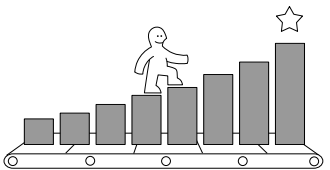
2



3



4



Timsort = 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** runstack
4. **galloping** merges
   use exponential searches
   to find position in other run



Much more on galloping ⇝ ask Vincent

Ghasemi, Jugé, Khalighinejad, Yazdanyar: *Galloping in Fast-Growth Natural Merge Sorts*, Algorithmica **2025**

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)

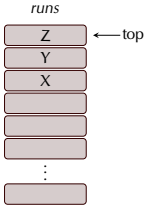
```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.append((i,j-i)); i = j
        while Rule A/B/C applicable
            merge X,Y resp. Y,Z
    while len(runs) > 1:
        merge Y,Z
```

full code:
`tiny.cc/timsort`

Timsort merge policy (original)

```
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`



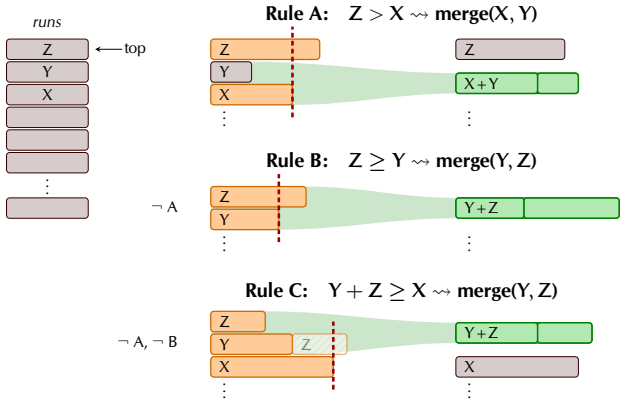
- `extend_run` detects next run
  & boosts to `minrun` if necessary

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`

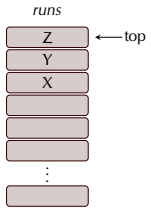
- `extend_run` detects next run & boosts to `minrun` if necessary

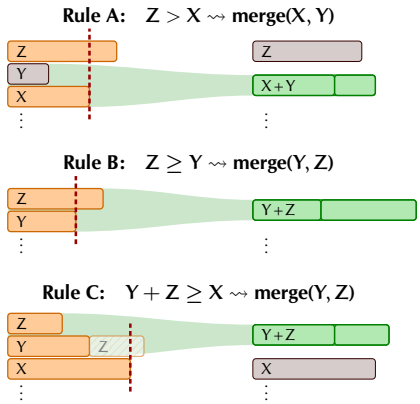


Timsort merge policy (original)

```
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`





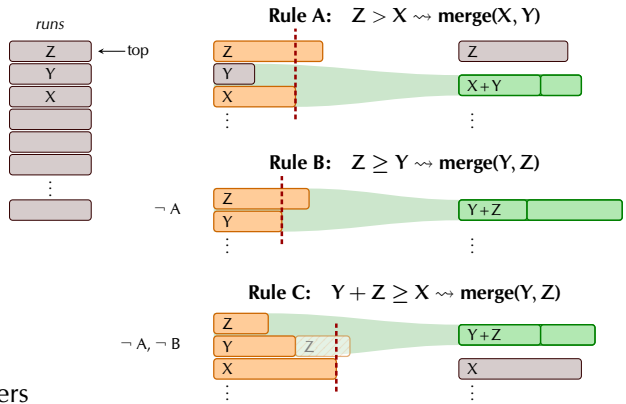
- `extend_run` detects next run
  & boosts to `minrun` if necessary
- **Goal:** runs on stack grow like Fibonacci numbers
  - Invariant: $\forall j:$ `runs[j]` $\geqslant$ `runs[j+1]` $+$ `runs[j+2]`

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

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

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

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

- 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)

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**

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

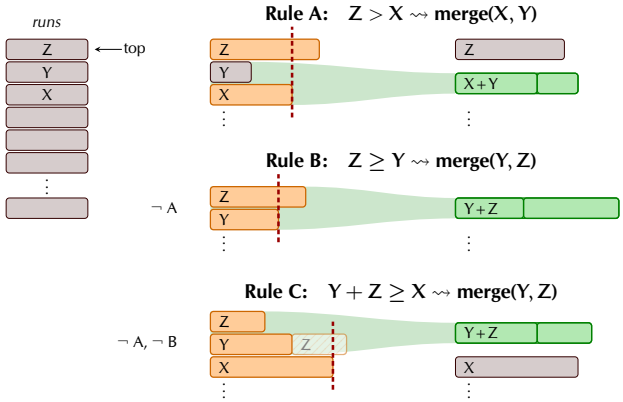
Timsort merge policy (patched)

```
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)

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

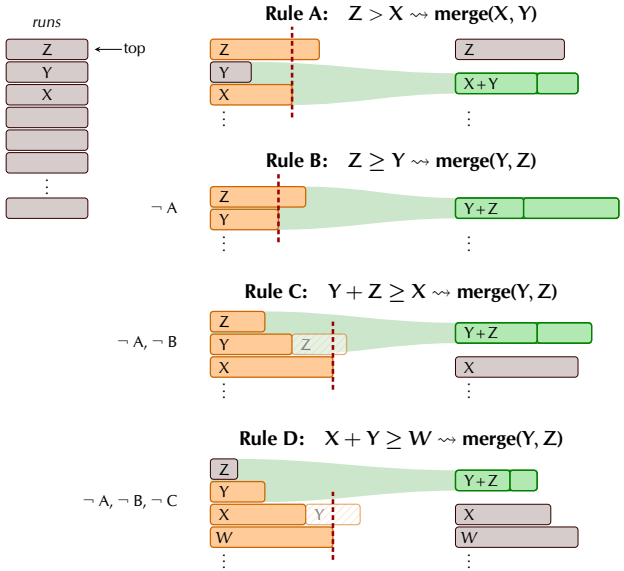
- Need to add a **Rule D**



Timsort merge policy (patched)

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

- Need to add a **Rule D**

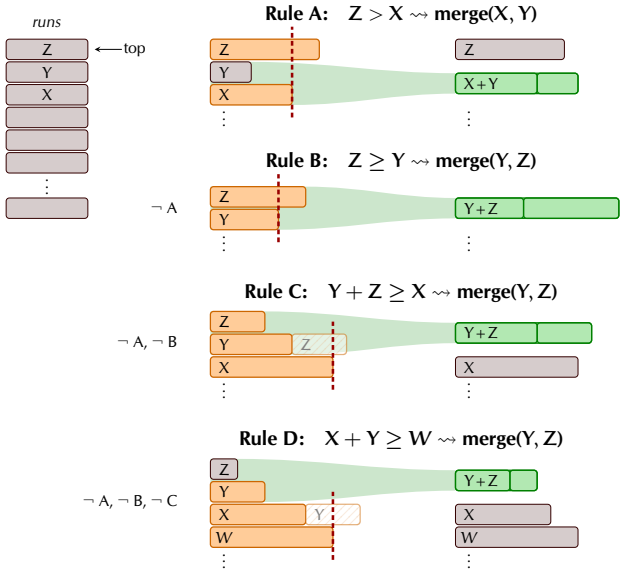


Timsort merge policy (patched)

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

- Need to add a **Rule D**

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



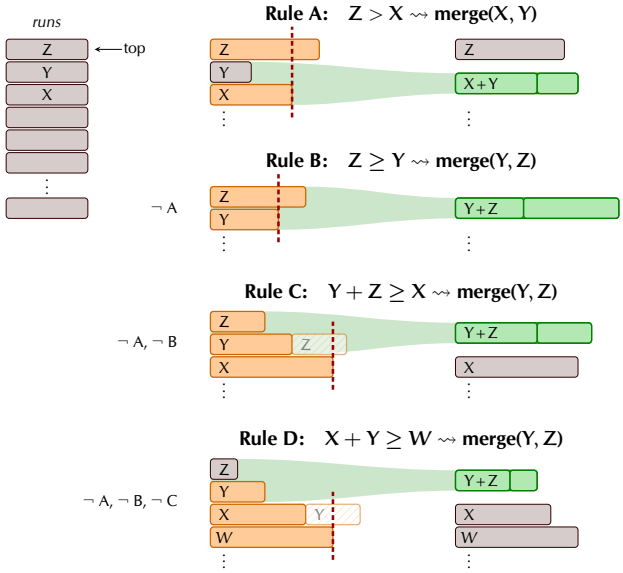
Timsort merge policy (patched)

```
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**

Merge policies from first principles

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

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

Conceptually two steps: (interleaved in code)

1. *Find runs in input.*
2. *Merge them **in some order**.* (determined by **merge policy**)

Here:

- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

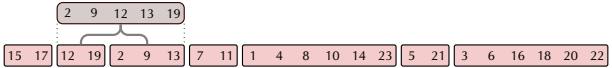
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

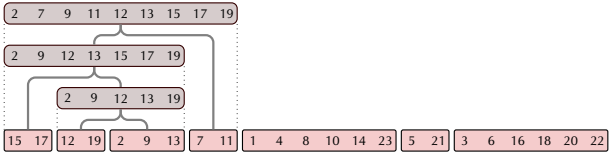
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

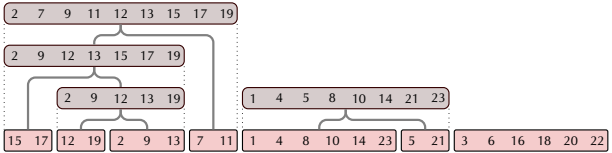
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

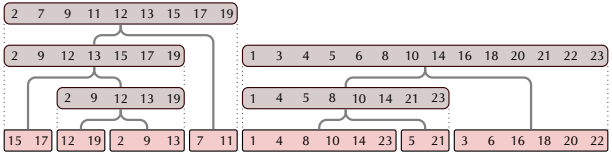
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

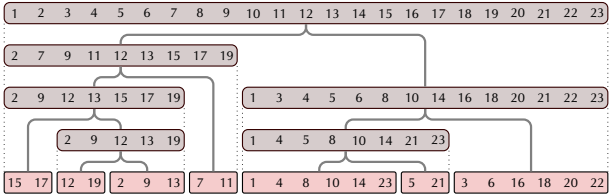
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

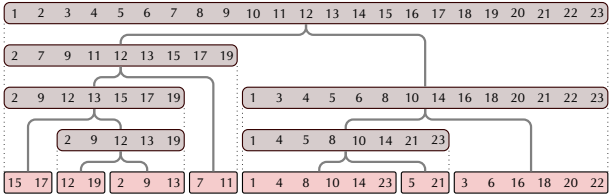
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

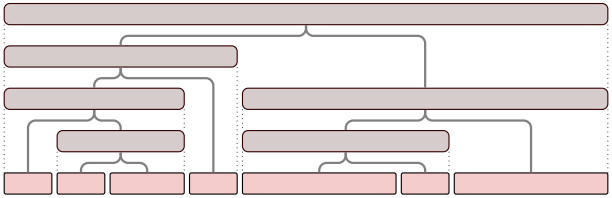
- only binary merges

⇝ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



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

Merge costs

- cost of merge := size of output
  - ≈ memory transfers
  - ⩾ #cmps
- total cost = total area of 

Merge policies from first principles

Run-adapti

interlea

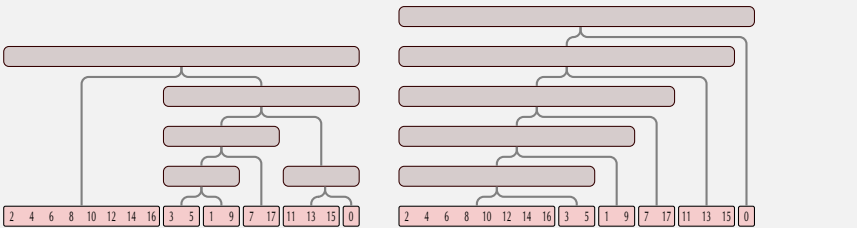
Conceptua

1. *Find r*
2. *Merge*

Here:

- only b
- ⇝ 2 bec
  merge 2 runs,
  repeat until single run
- only stable sorts
- ⇝ *merge 2* ***adjacent*** *runs*

Different merge policies yield different merge costs!



tree

Merge costs

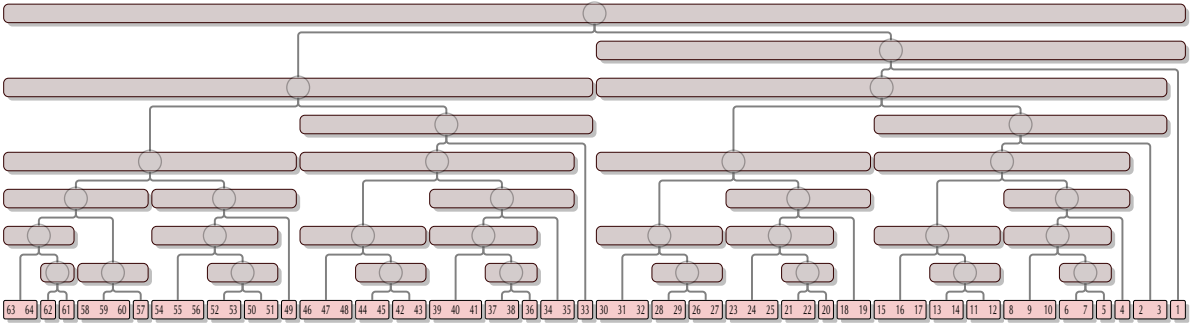
- cost of merge := size of output
  - ≈ memory transfers
  - ⩾ #cmps
- total cost = total area of

Timsort bad case

- Timsort's merge policy mostly works OK

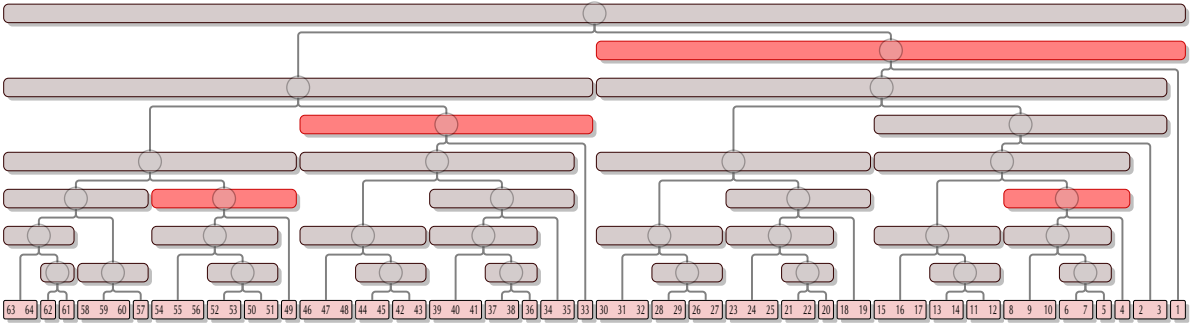
Timsort bad case

- 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:

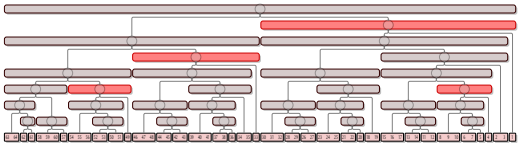


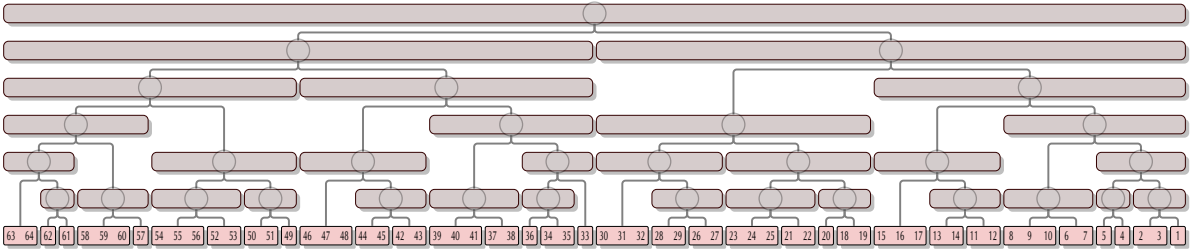
- intuitive problem: regularly very unbalanced merges

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:



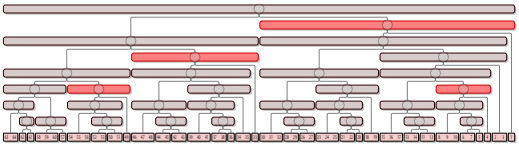


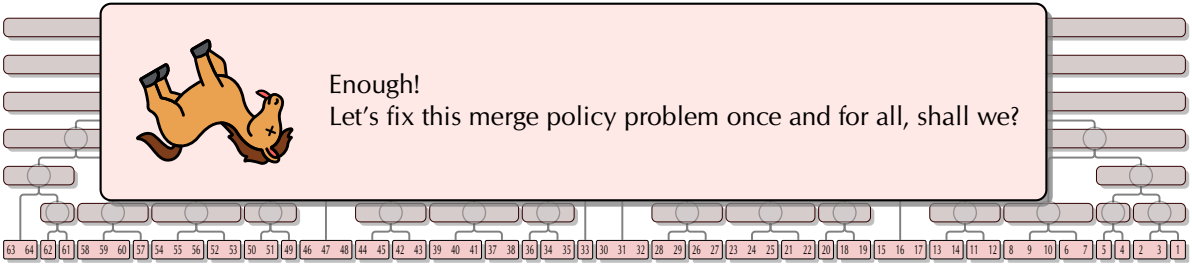
- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

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:

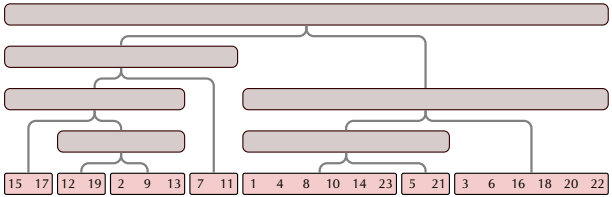




- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

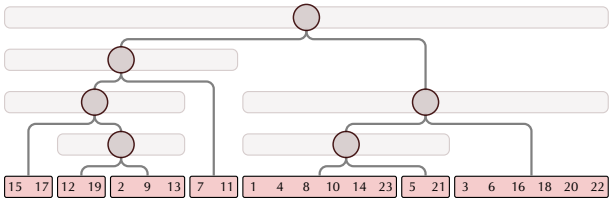
Buss, Knop: *Strategies for Stable Merge Sorting*, SODA **2019**

Mergesort meets Binary Search Trees



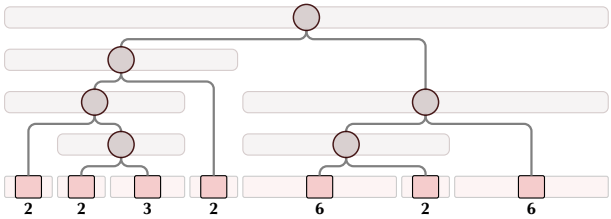
Merge cost = total area of 

Mergesort meets Binary Search Trees



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

Mergesort meets Binary Search Trees



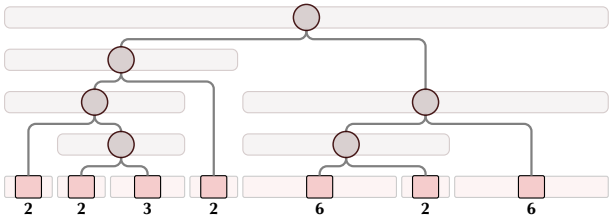
Merge cost $=$ total area of 

$=$ 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

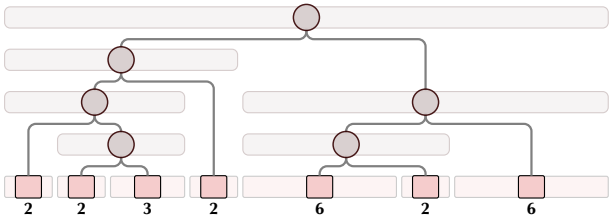


Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

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

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

Mergesort meets Binary Search Trees



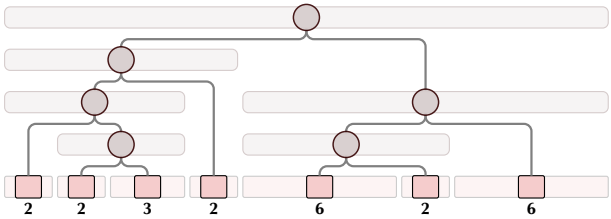
Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

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

$\rightsquigarrow$ **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?

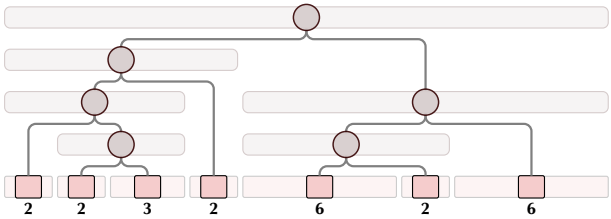
Merge cost = total area of
= total length of paths to all array entries
= weighted external path length

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

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

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

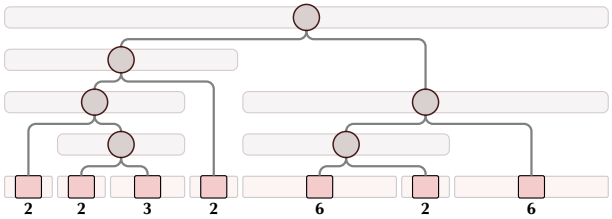
$\rightsquigarrow$ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

$\rightsquigarrow$ 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

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

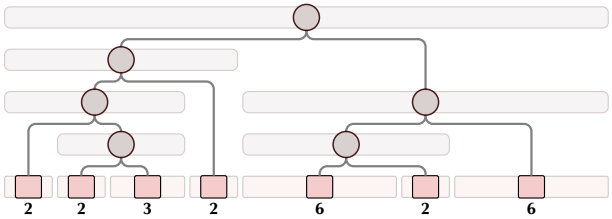
⇝ **optimal** merge tree = optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

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

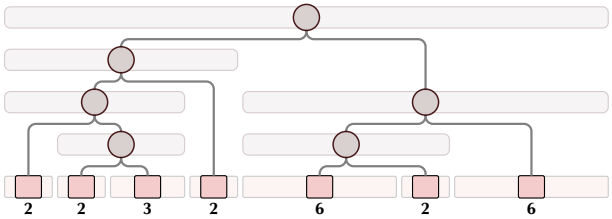
How to compute good merge tree?

- ***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

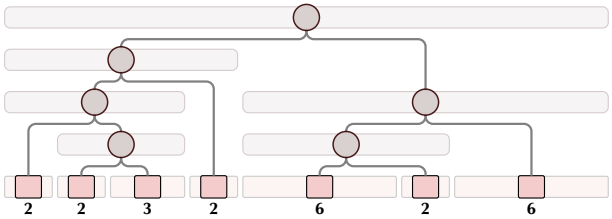


…70s are calling

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

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

⇝ **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

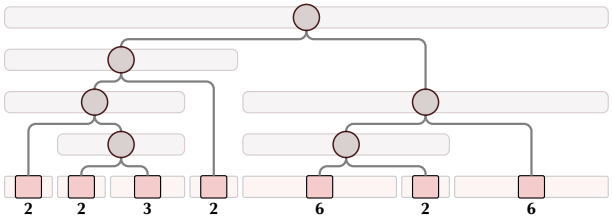


…70s are calling

nearly-optimal BST merge

- simple (greedy) linear-time methods!
- 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

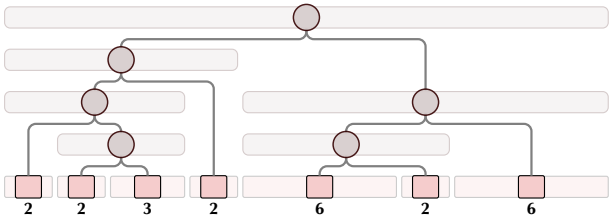
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm



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



…70s are calling

nearly-optimal BST merge

- 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**

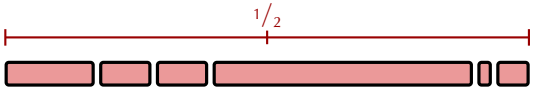
> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

- **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**

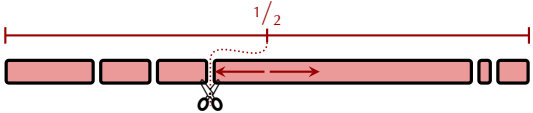


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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**



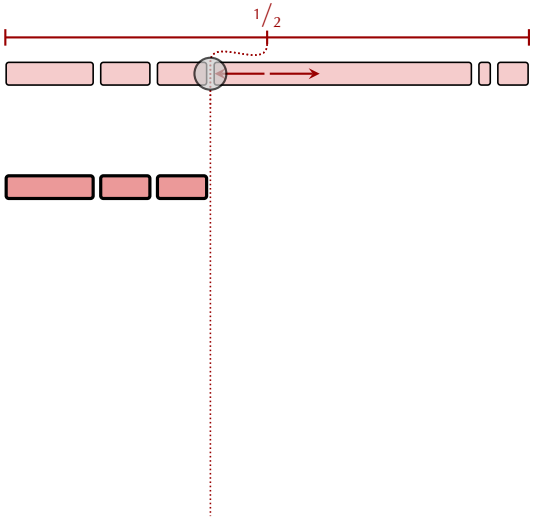
- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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

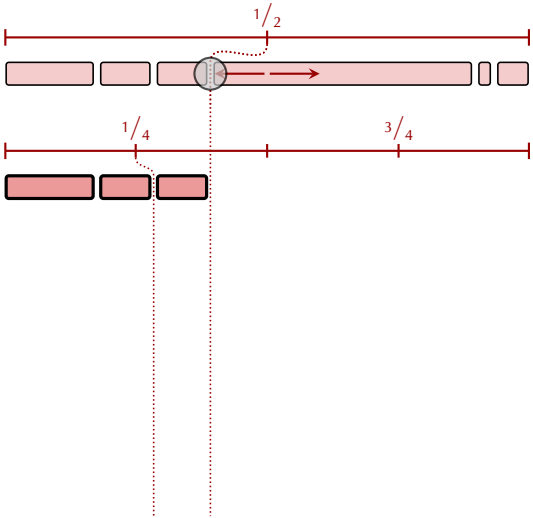


The Bisection Method

- 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

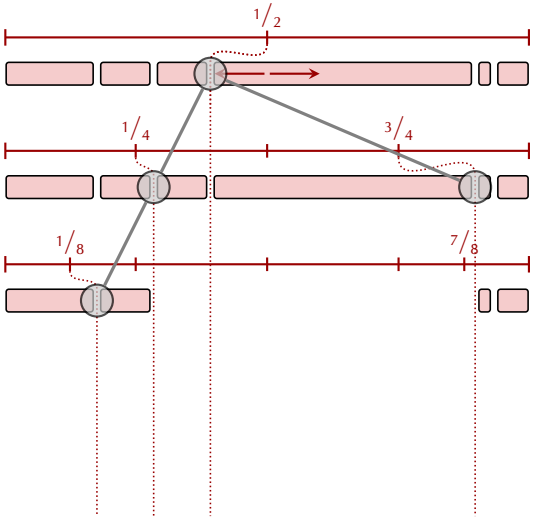


The Bisection Method

- 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

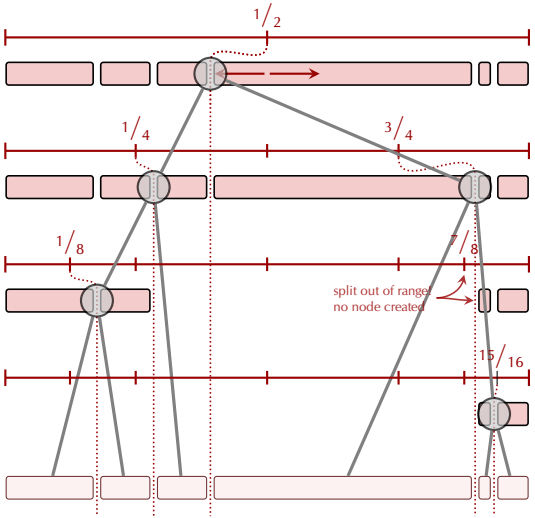


The Bisection Method

- 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



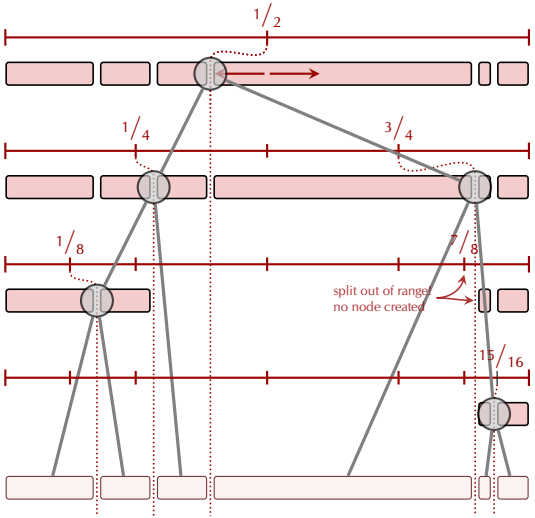
The Bisection Method

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

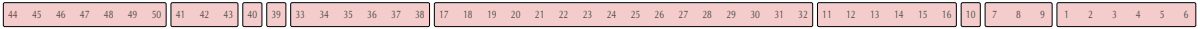
> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

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

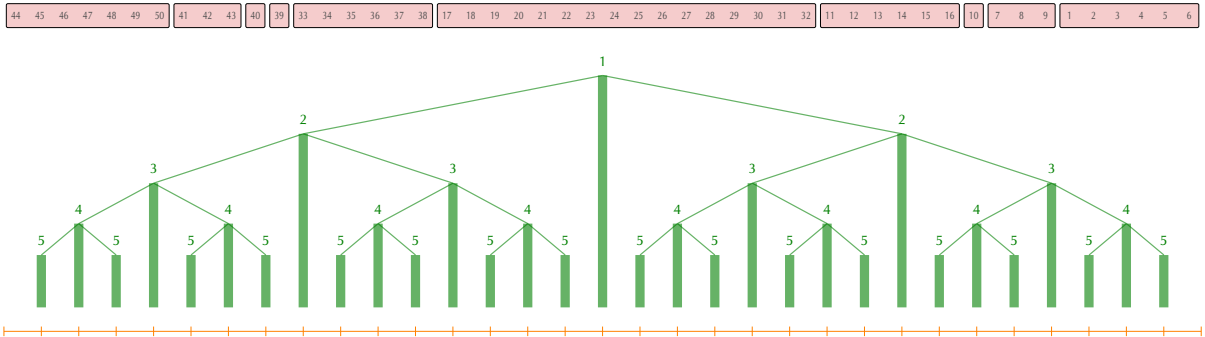
- *Up next:* Don't use recursion!



Run-Boundary Powers

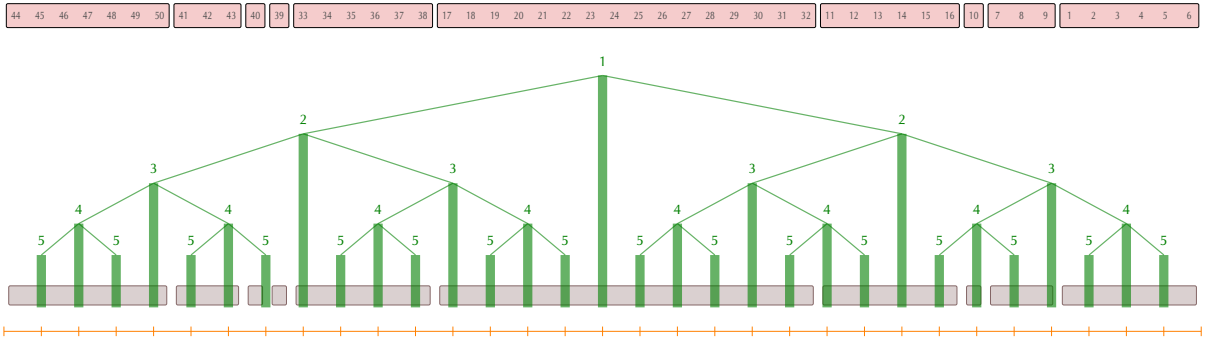


Run-Boundary Powers



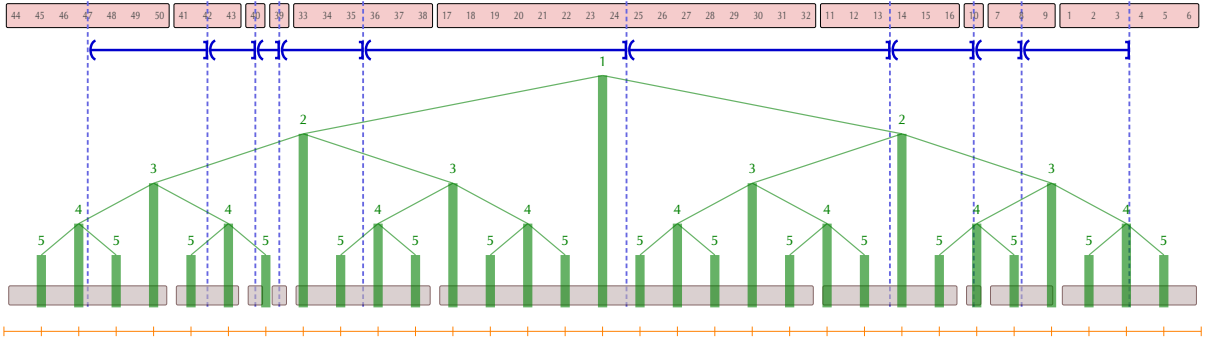
- (virtual) perfect balanced binary tree

Run-Boundary Powers



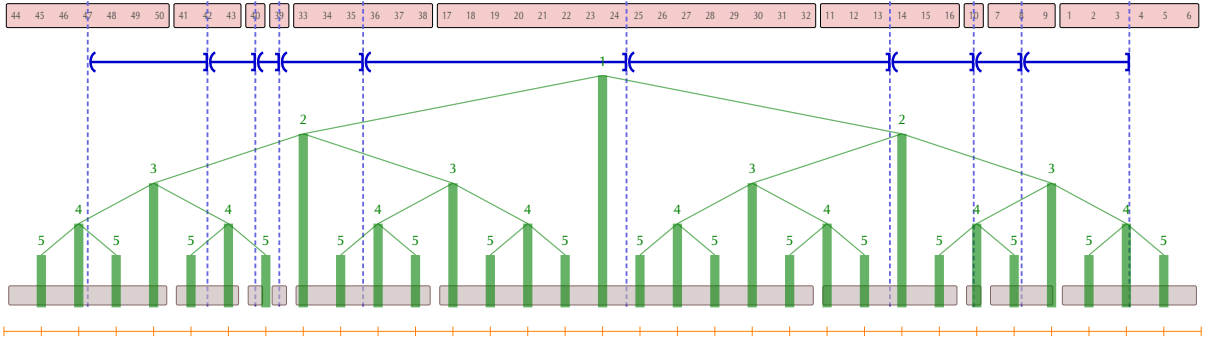
- (virtual) perfect balanced binary tree

Run-Boundary Powers



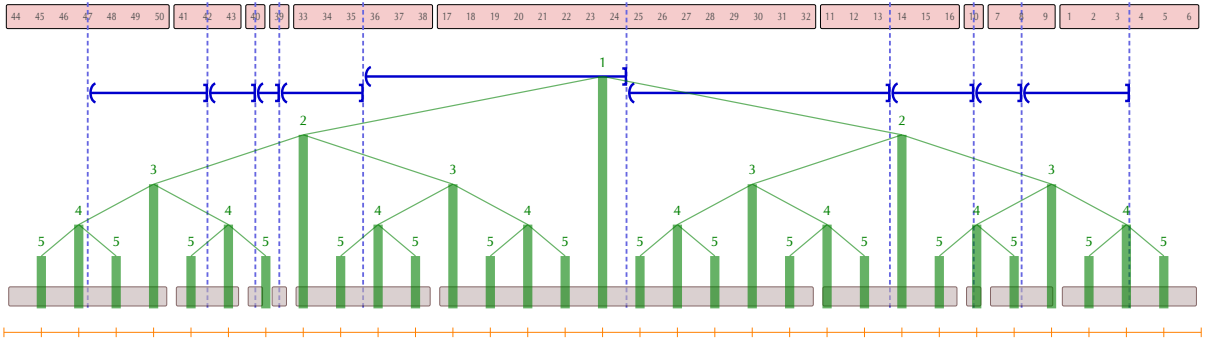
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



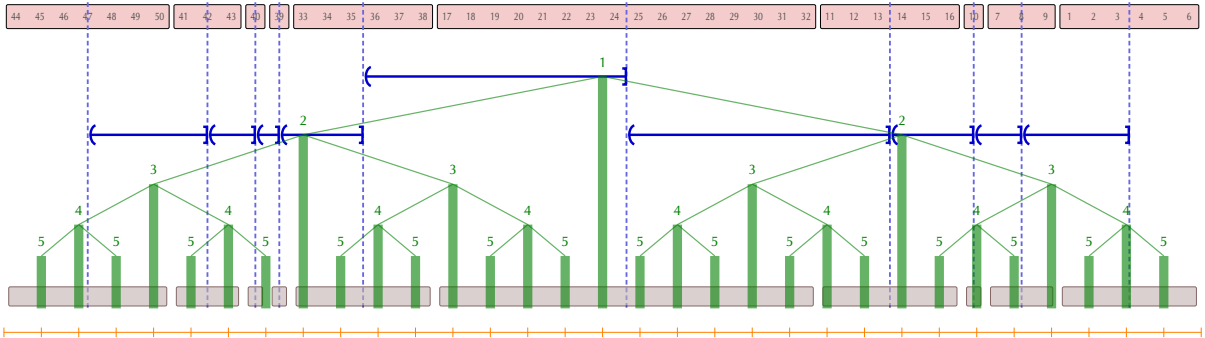
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



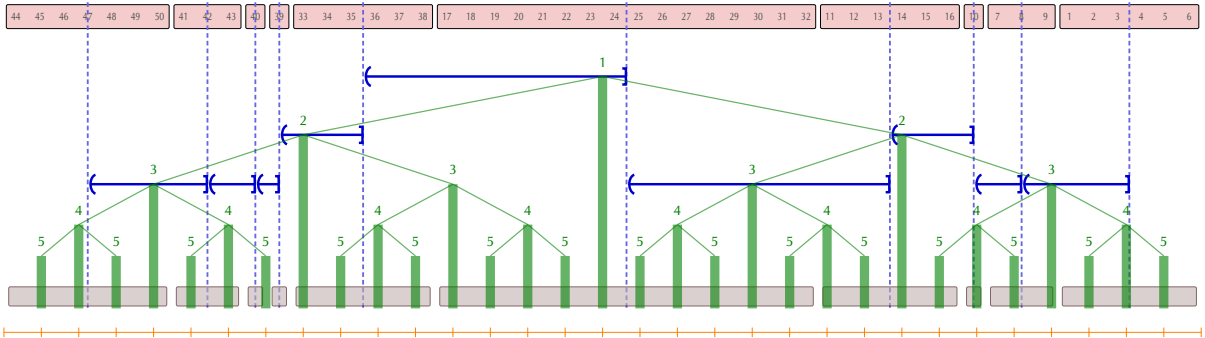
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



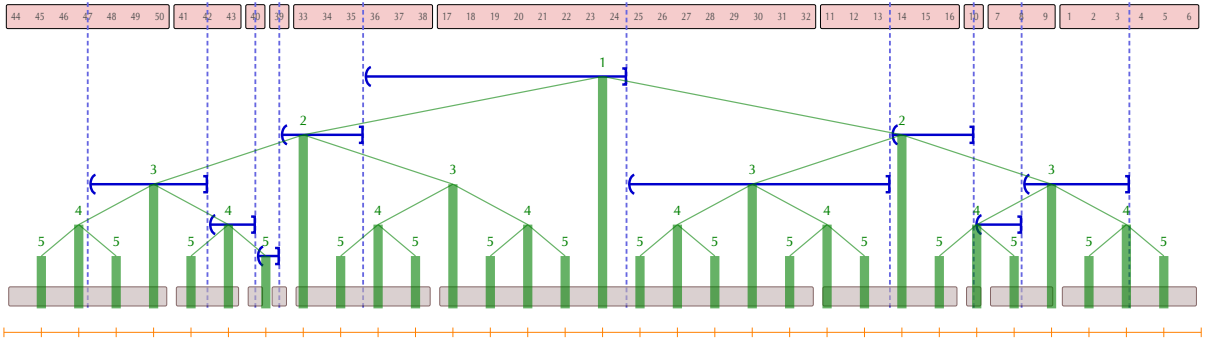
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



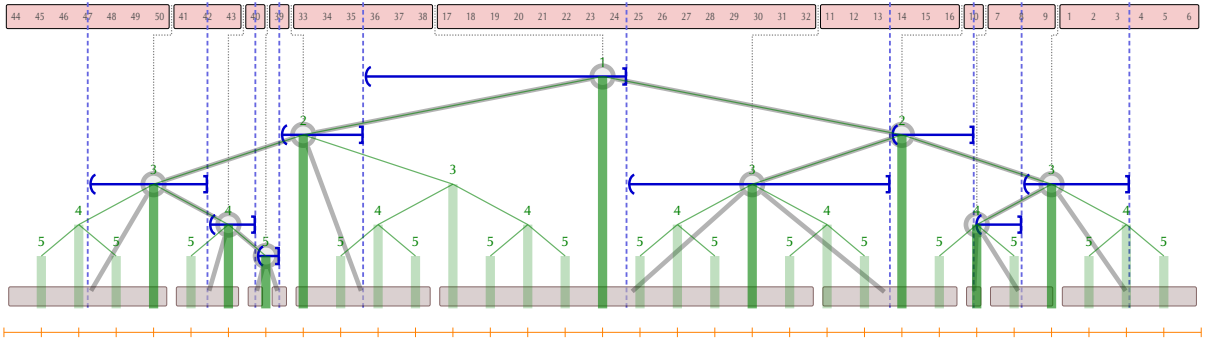
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



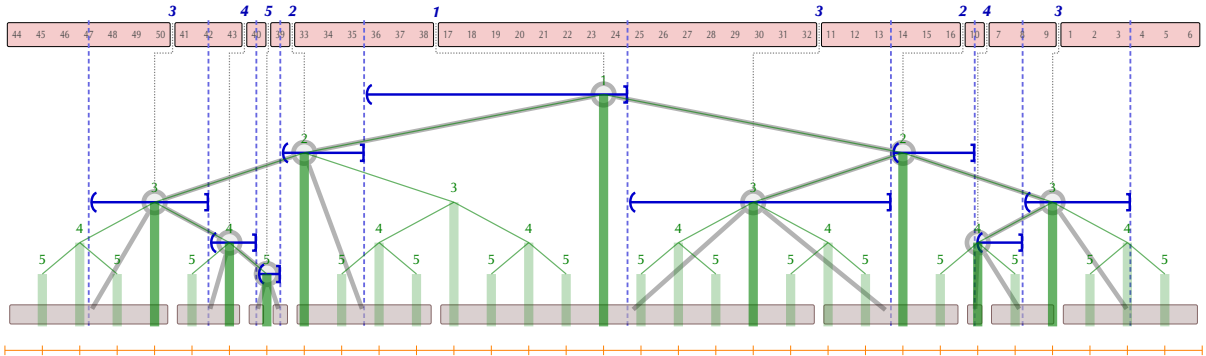
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



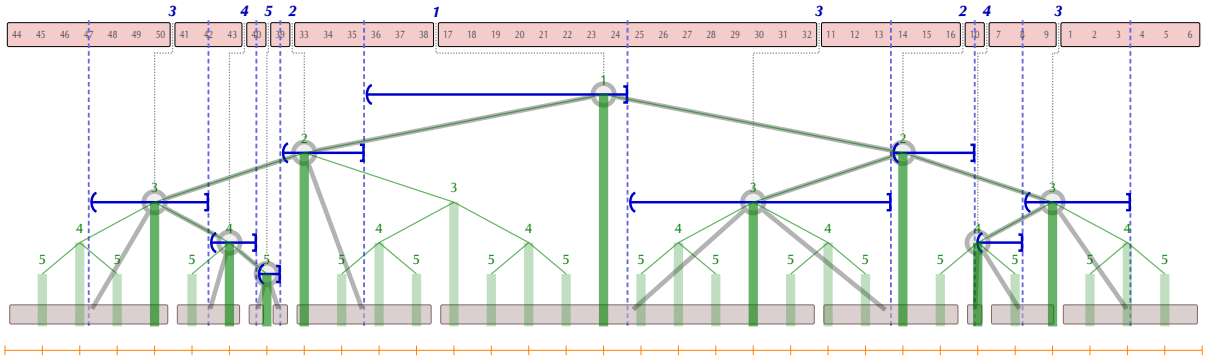
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



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

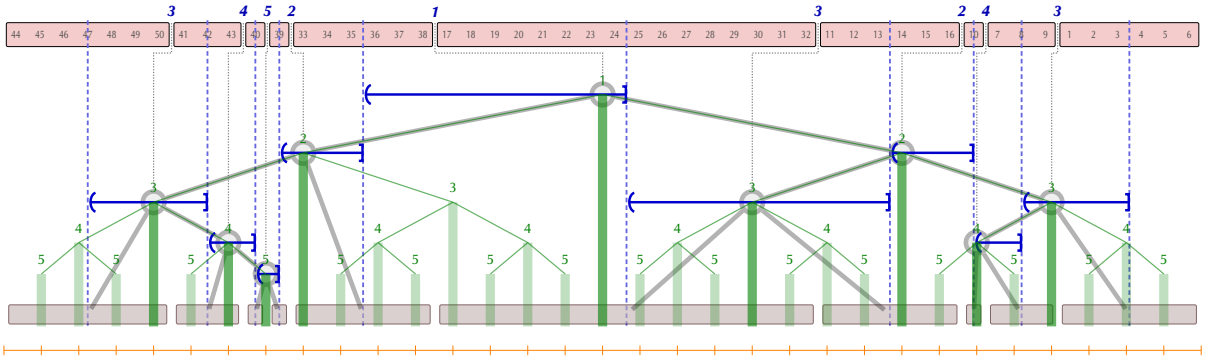
Run-Boundary Powers



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



Run-Boundary Powers

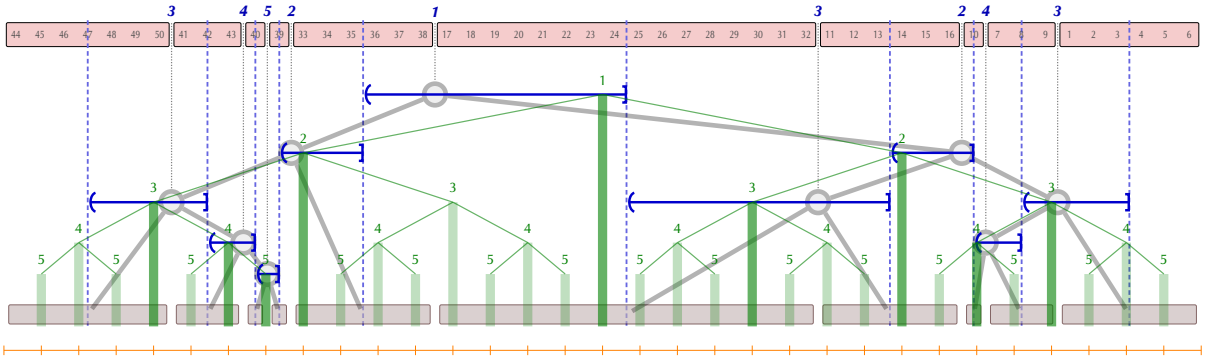


- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its **power**

⇝ merge tree follows virtual tree



Run-Boundary Powers

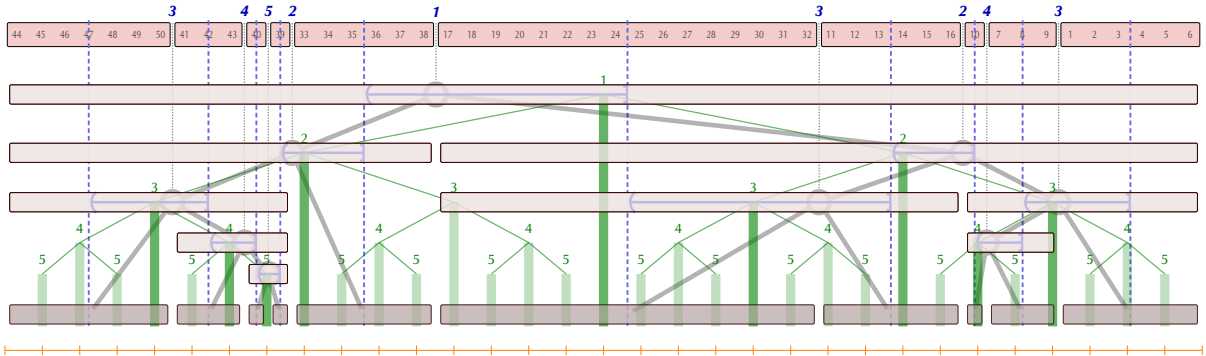


- (virtual) perfect balanced binary tree
- midpoint intervals "snap" to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its **power**

⇝ merge tree follows virtual tree



Run-Boundary Powers

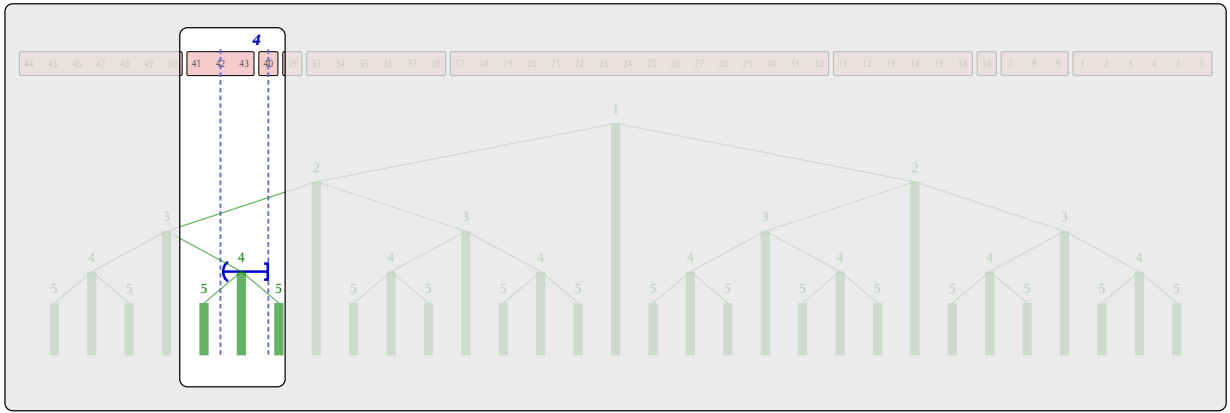


- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its **power**

⇝ merge tree follows virtual tree



Run-Boundary Powers are Local

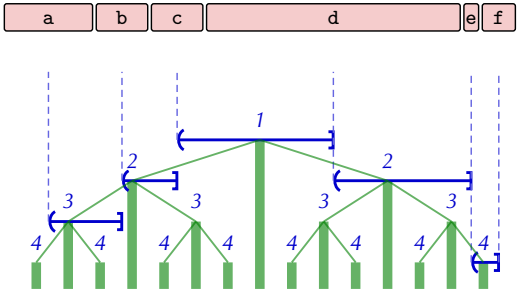


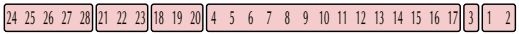
Computation of powers only depends on two adjacent runs.

Powersort

```
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:
`tiny.cc/timsort`

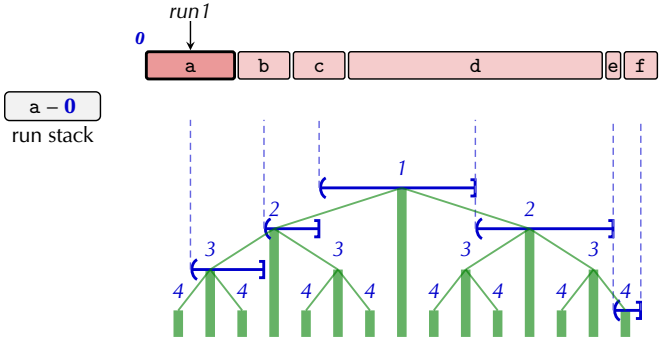


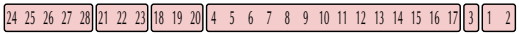


Powersort

```
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:
`tiny.cc/timsort`

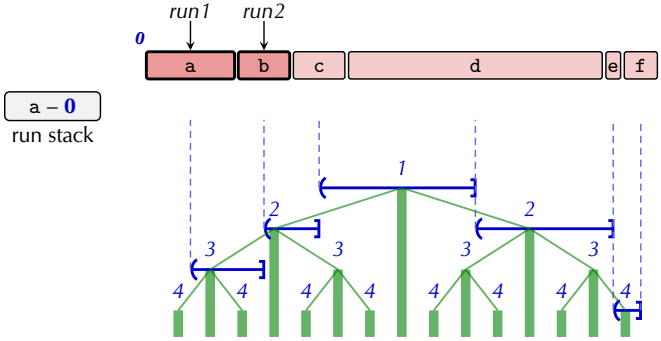


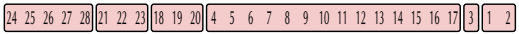


Powersort

```
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:
`tiny.cc/timsort`

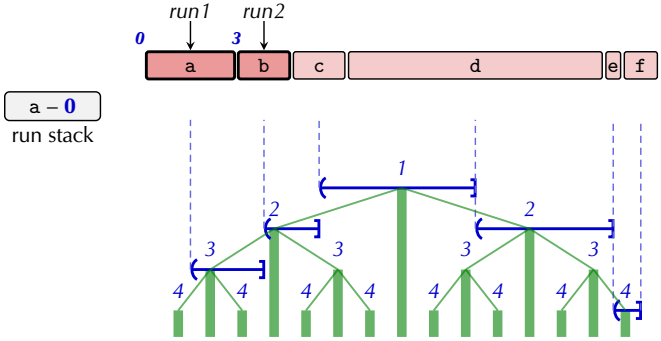


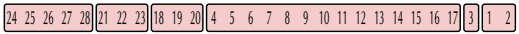


Powersort

```
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:
`tiny.cc/timsort`

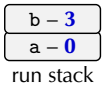


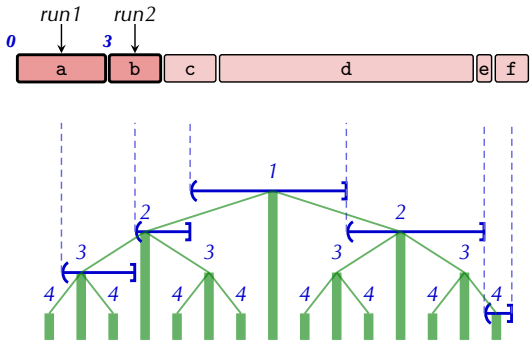


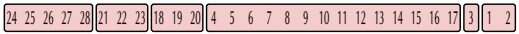
Powersort

```
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:
`tiny.cc/timsort`



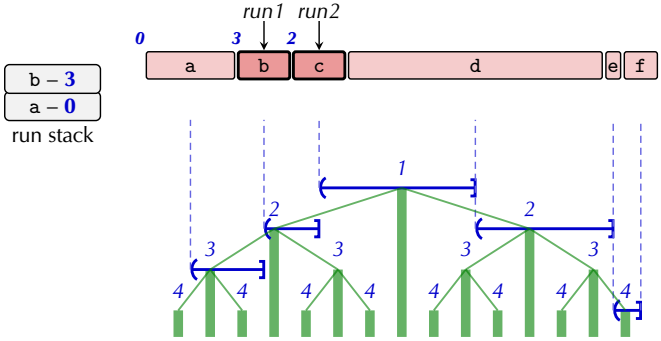


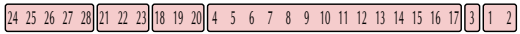


Powersort

```
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:
`tiny.cc/timsort`

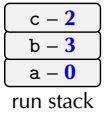


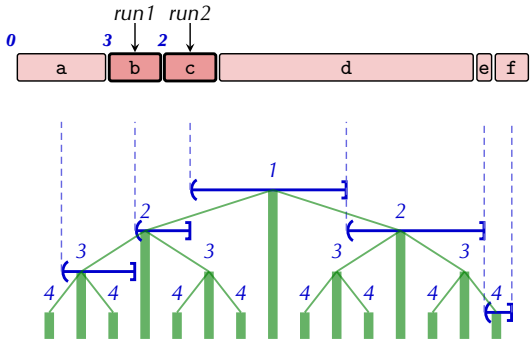


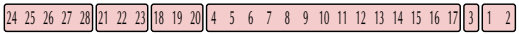
Powersort

```
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:
`tiny.cc/timsort`



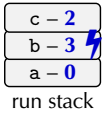


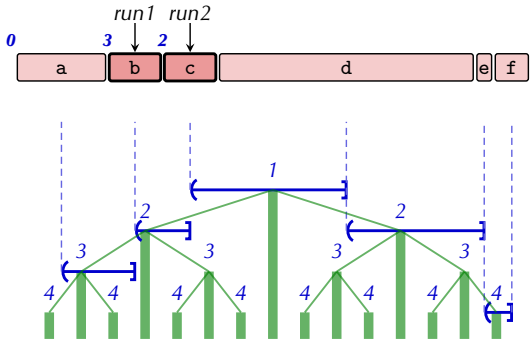


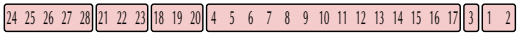
Powersort

```
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:
`tiny.cc/timsort`



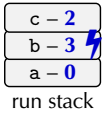


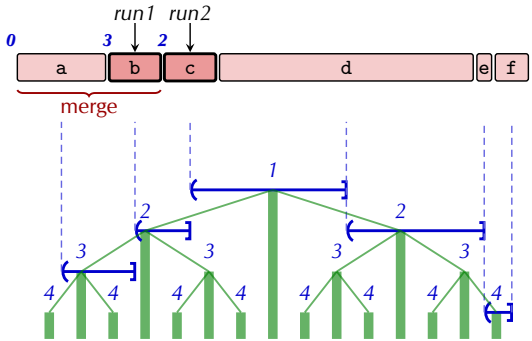


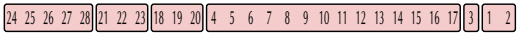
Powersort

```
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:
`tiny.cc/timsort`



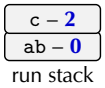


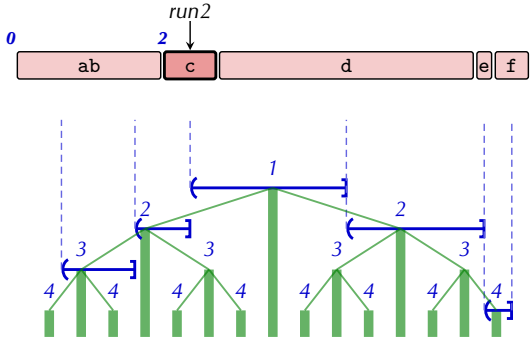


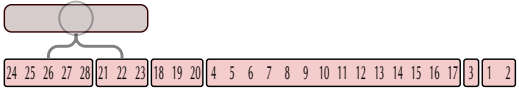
Powersort

```
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:
`tiny.cc/timsort`



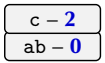


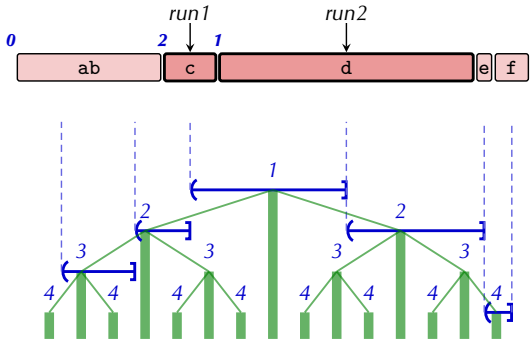


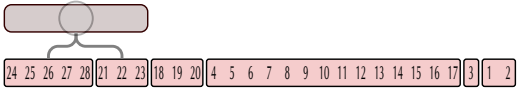
Powersort

```
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:
`tiny.cc/timsort`



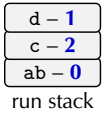


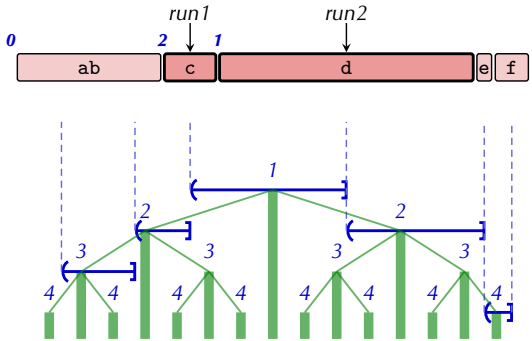


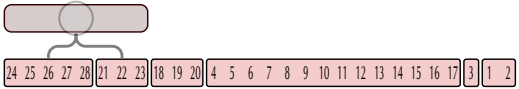
Powersort

```
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:
`tiny.cc/timsort`



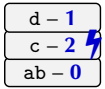




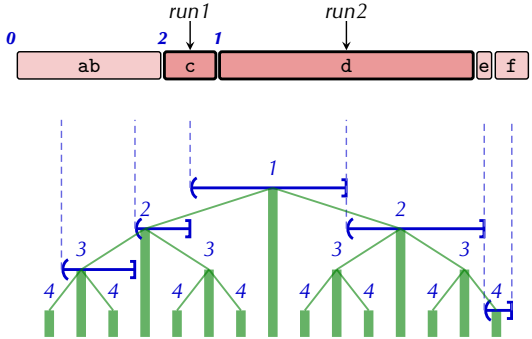
Powersort

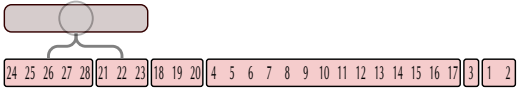
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

full code:
`tiny.cc/timsort`



run stack

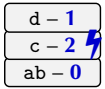


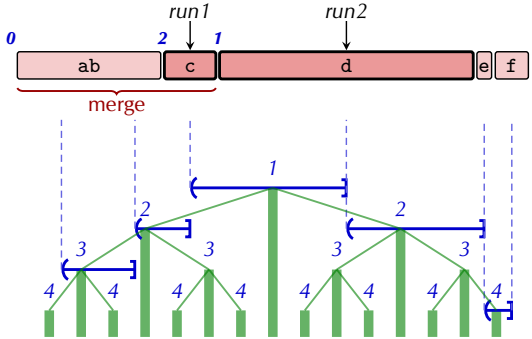


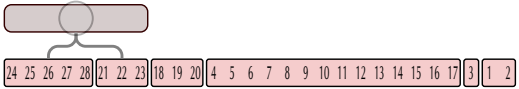
Powersort

```
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:
`tiny.cc/timsort`



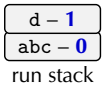


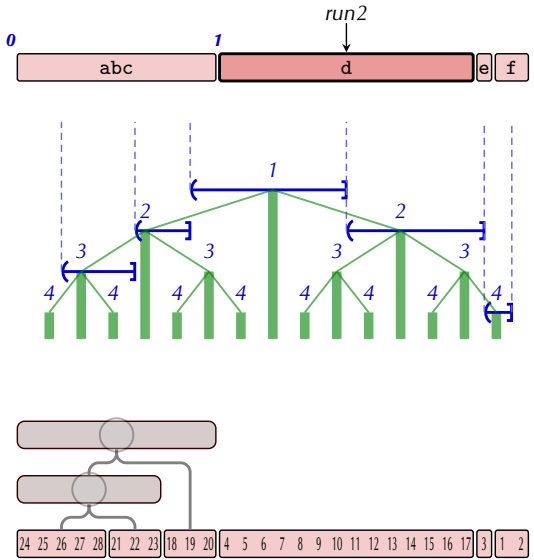


Powersort

```
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:
`tiny.cc/timsort`

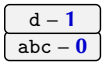




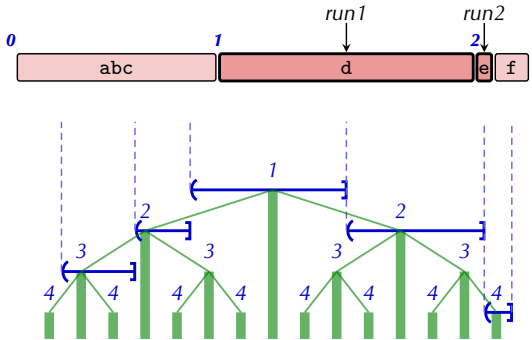
Powersort

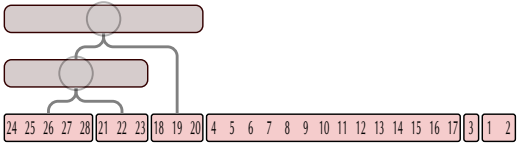
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

full code:
`tiny.cc/timsort`



run stack

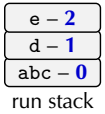


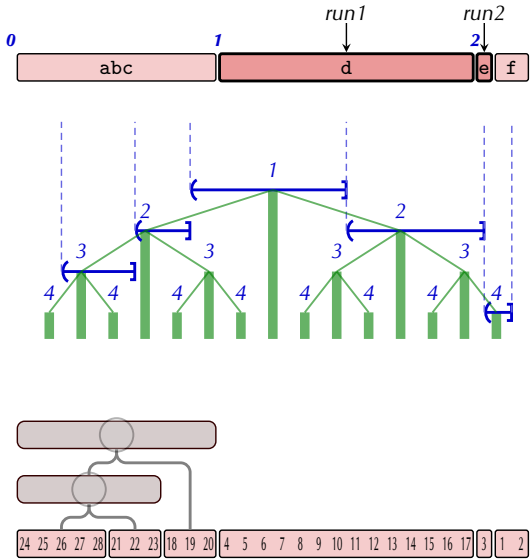


Powersort

```
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:
`tiny.cc/timsort`

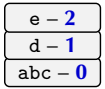


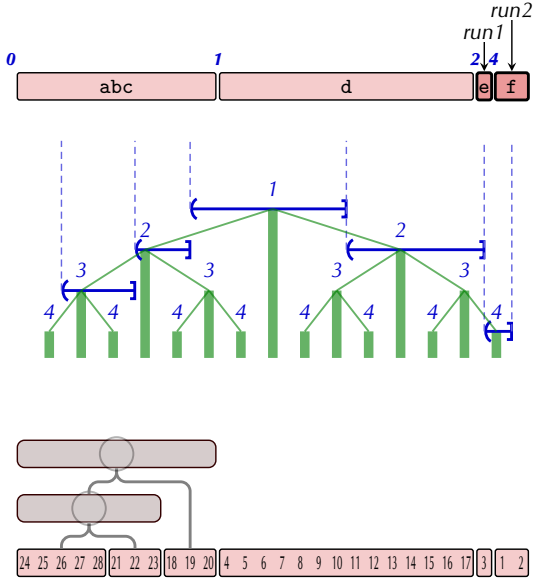


Powersort

```
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:
`tiny.cc/timsort`

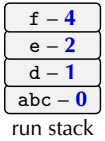


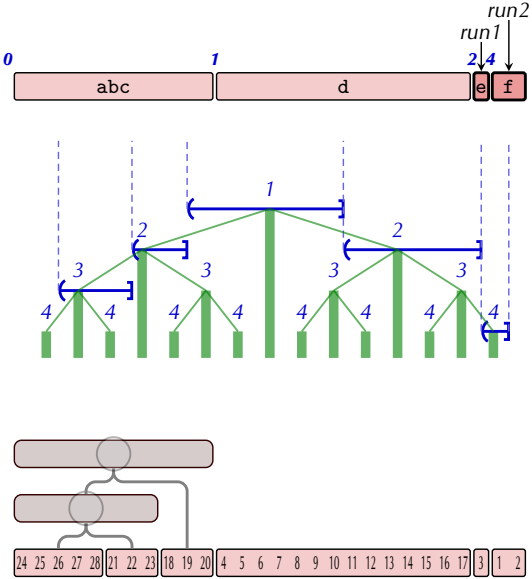


Powersort

```
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:
`tiny.cc/timsort`

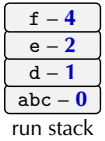


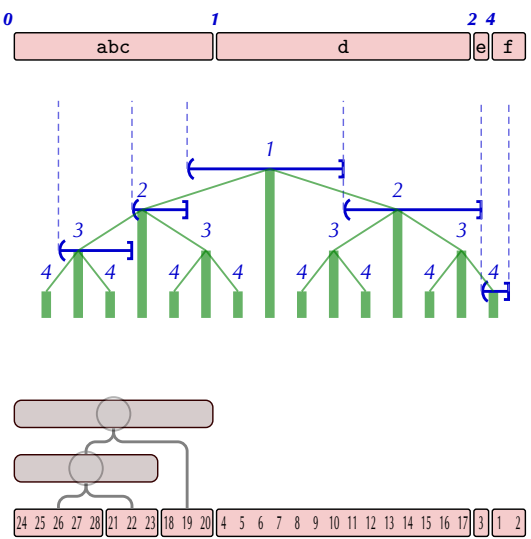


Powersort

```
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:
`tiny.cc/timsort`

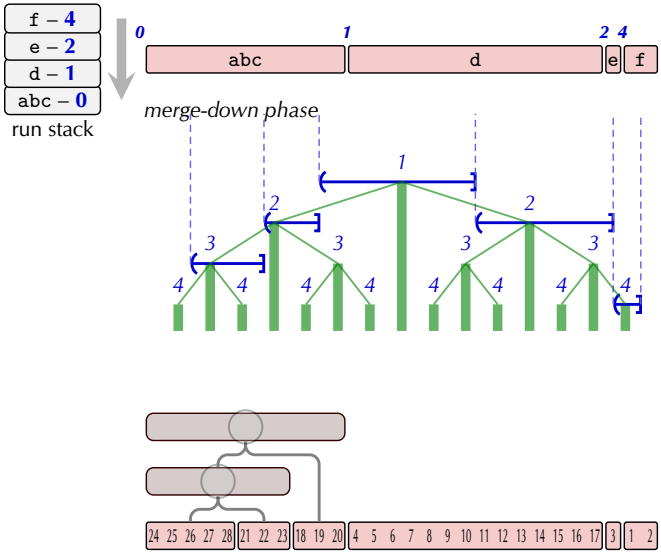




Powersort

```
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)
```

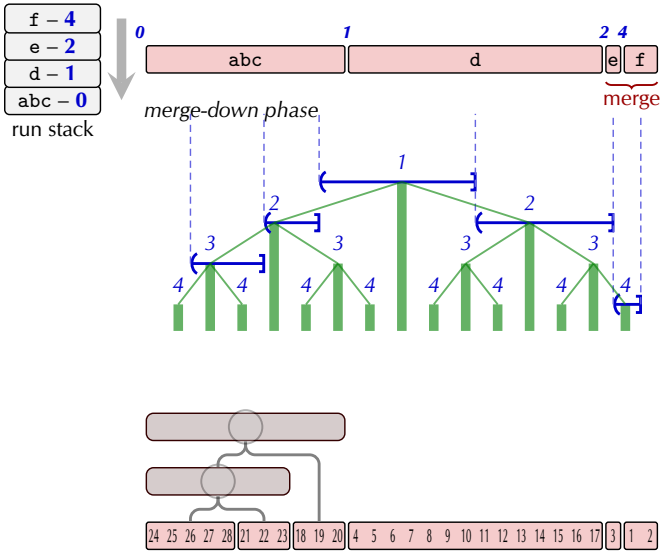
full code:
`tiny.cc/timsort`



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)
```

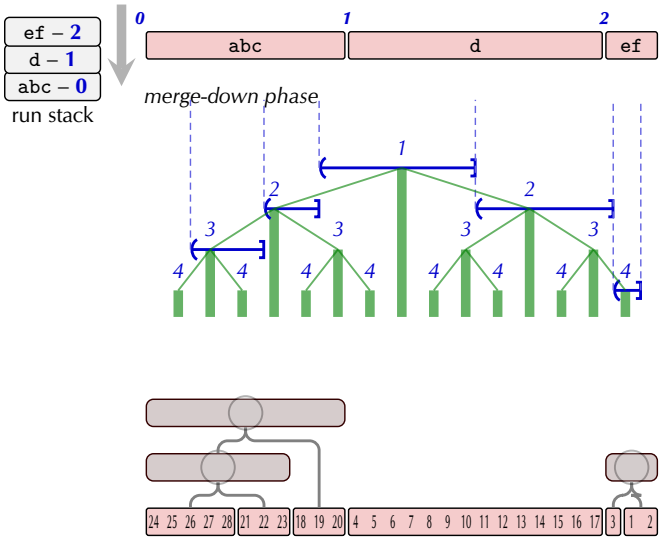
full code:
`tiny.cc/timsort`



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)
```

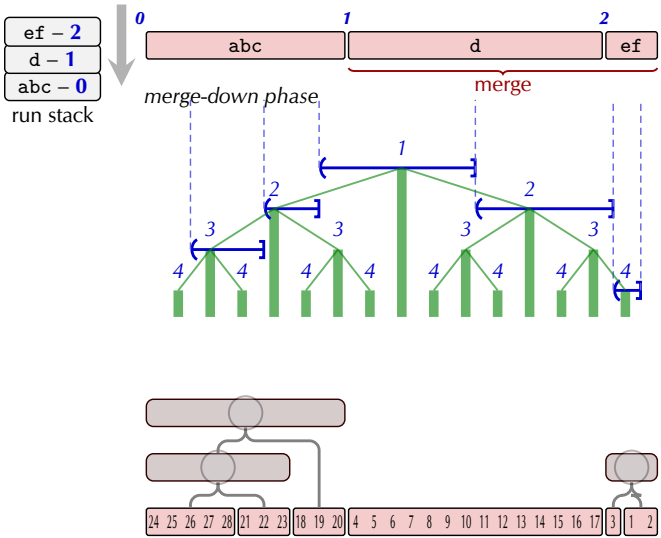
full code:
`tiny.cc/timsort`



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)
```

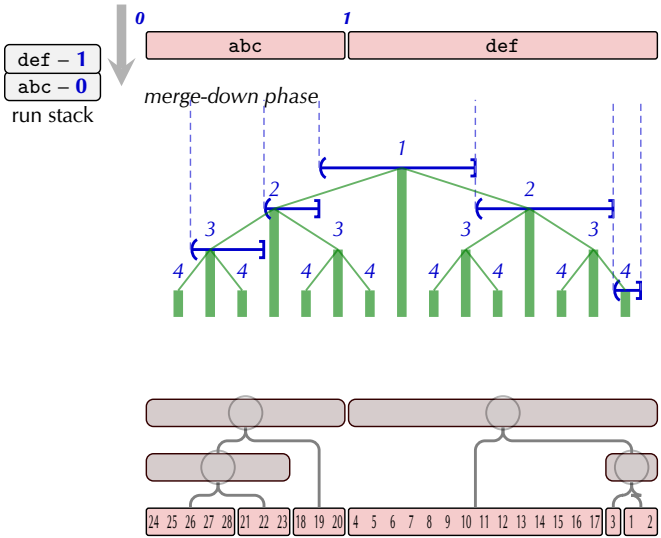
full code:
`tiny.cc/timsort`



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)
```

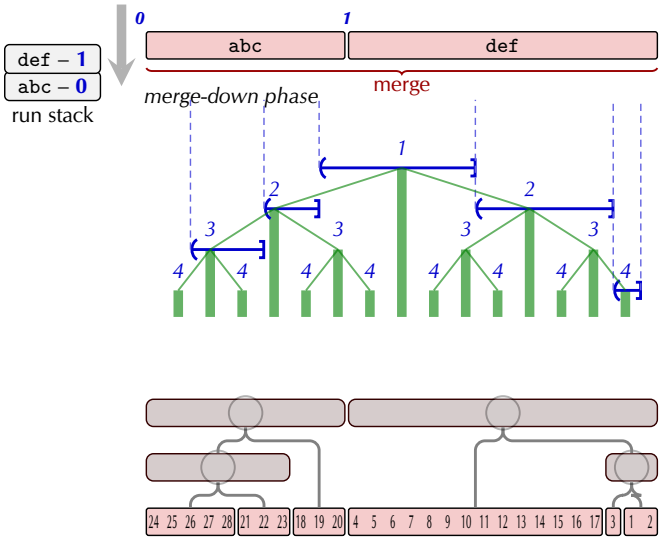
full code:
`tiny.cc/timsort`



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:
`tiny.cc/timsort`

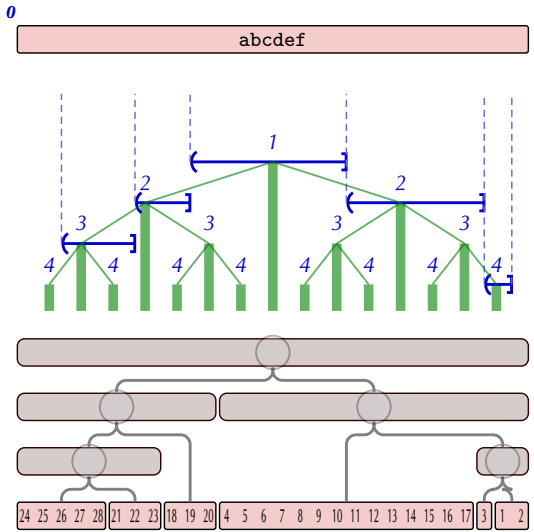


Powersort

```
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:
`tiny.cc/timsort`

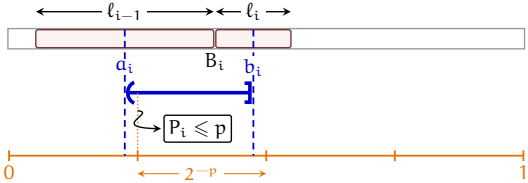
abcdef – **0**
run stack



Computing Powers

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

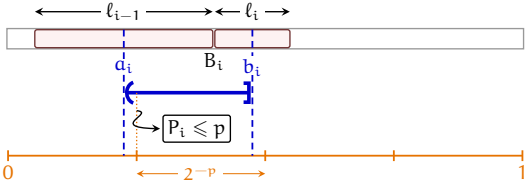




Computing Powers

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






```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      A = 2 * i1 + n1    # A = 2na_i
5      B = A + n1 + n2    # B = 2nb_i
6      p = 0
7      while True:
8          p += 1
9          if A >= n:
10             A -= n; B -= n
11         elif B >= n:
12             break
13         A <<= 1; B <<= 1
14     return p
```

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

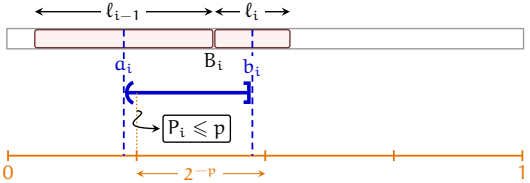
Computing Powers

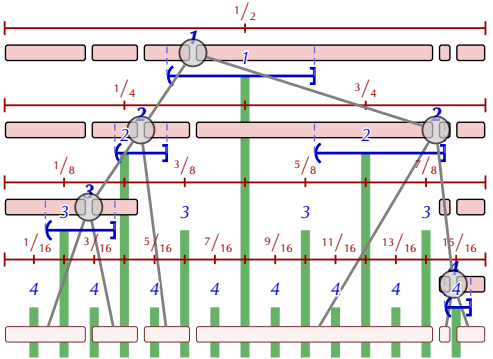
- Computing the power of (run boundary between two runs
  - $\left[\!\left[\right.\right.\left.\left.\right]\!\right]$ = normalized midpoint interval
  - power = min ℓ s.t. $\left[\!\left[\right.\right.\left.\left.\right]\!\right]$ contains $c \cdot 2^{-\ell}$



```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      A = 2 * i1 + n1    # A = 2na_i
5      B = A + n1 + n2    # B = 2nb_i
6      p = 0
7      while True:
8          p += 1
9          if A >= n:
10             A -= n; B -= n
11         elif B >= n:
12             break
13         A <<= 1; B <<= 1
14     return p
```

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





Mergecost Analysis

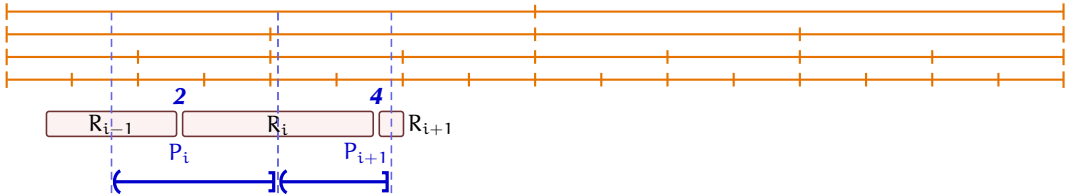
- **Given:** input with runs $R_1, \ldots, R_r$ of **lengths** $L_1, \ldots, L_r$; $n = L_1 + \cdots + L_r$
- Mergecost: $M = \sum_{i=1}^{r} L_i \cdot \text{depth}(R_i)$

Mergecost Analysis

- **Given:** input with runs $R_1, \ldots, R_r$ of **lengths** $L_1, \ldots, L_r$; $n = L_1 + \cdots + L_r$
- Mergecost: $M = \sum_{i=1}^{r} L_i \cdot \operatorname{depth}(R_i)$

$\rightsquigarrow$ $M \leqslant \sum_{i=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

P_i = power of boundary between R_{i-1} and R_i, $i = 2, \ldots, r$
$P_1 = P_{r+1} = 0$



Mergecost Analysis

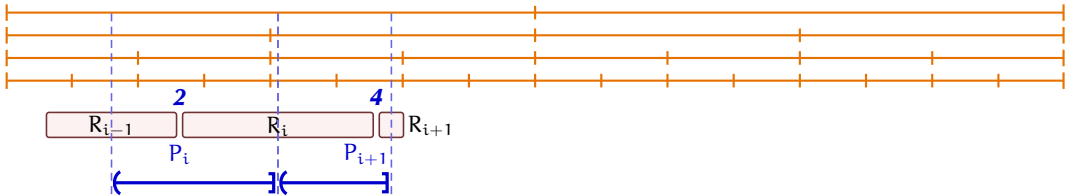
- **Given:** input with runs $R_1, \dots, R_r$ of **lengths** $L_1, \dots, L_r$; $n = L_1 + \cdots + L_r$
- Mergecost: $M = \sum_{r=1}^{r} L_i \cdot \text{depth}(R_i)$

$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

P_i = power of boundary between R_{i-1} and R_i, $i = 2, \dots, r$
$P_1 = P_{r+1} = 0$

$\rightsquigarrow$ To prove $M \leqslant n(\mathcal{H} + 2)$,
it suffices to show:

$$P_i \leqslant \lceil \log_2(\tfrac{n}{L_i}) + 1 \rceil \leqslant \log_2(\tfrac{n}{L_i}) + 2$$
$$P_{i+1} \leqslant \lceil \log_2(\tfrac{n}{L_i}) + 1 \rceil \leqslant \log_2(\tfrac{n}{L_i}) + 2$$



Mergecost Analysis

- **Given:** input with runs $R_1, \dots, R_r$ of **lengths** $L_1, \dots, L_r$; $n = L_1 + \cdots + L_r$
- Mergecost: $M = \sum_{r=1}^{r} L_i \cdot \text{depth}(R_i)$

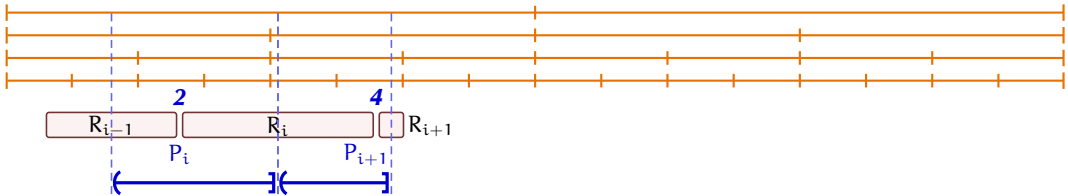
$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

P_i = power of boundary between R_{i-1} and R_i, $i = 2, \dots, r$
$P_1 = P_{r+1} = 0$

$\rightsquigarrow$ To prove $M \leqslant n(\mathcal{H}+2)$, it suffices to show:

$$P_i \leqslant \lceil \log_2(\tfrac{n}{L_i}) + 1 \rceil \leqslant \log_2(\tfrac{n}{L_i}) + 2$$
$$P_{i+1} \leqslant \lceil \log_2(\tfrac{n}{L_i}) + 1 \rceil \leqslant \log_2(\tfrac{n}{L_i}) + 2$$

- by def.: $\frac{|\text{⟷}|}{n} \geqslant 2^{-p}$ $\rightsquigarrow$ $P_i \leqslant p$

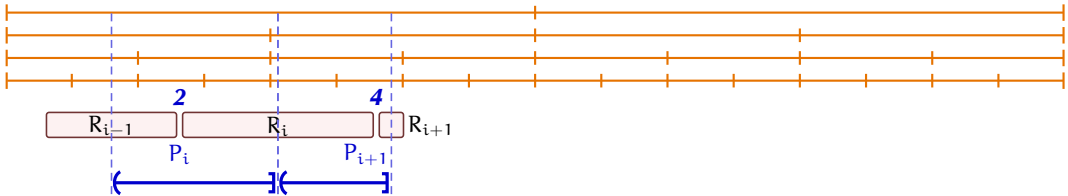


Mergecost Analysis

- **Given:** input with runs $R_1, \dots, R_r$ of **lengths** $L_1, \dots, L_r$; $n = L_1 + \cdots + L_r$
- Mergecost: $M = \sum_{r=1}^{r} L_i \cdot \mathrm{depth}(R_i)$

$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

P_i = power of boundary between R_{i-1} and R_i, $i = 2, \dots, r$
$P_1 = P_{r+1} = 0$



$\rightsquigarrow$ To prove $M \leqslant n(\mathcal{H}+2)$, it suffices to show:

$$P_i \leqslant \left\lceil \log_2\left(\tfrac{n}{L_i}\right) + 1 \right\rceil \leqslant \log_2\left(\tfrac{n}{L_i}\right) + 2$$
$$P_{i+1} \leqslant \left\lceil \log_2\left(\tfrac{n}{L_i}\right) + 1 \right\rceil \leqslant \log_2\left(\tfrac{n}{L_i}\right) + 2$$

- by def.: $\frac{|\longleftrightarrow|}{n} \geqslant 2^{-p}$ $\rightsquigarrow$ $P_i \leqslant p$
- we have: $|\longleftrightarrow| \geqslant L_i/2$

Mergecost Analysis

- **Given:** input with runs $R_1, \dots, R_r$ of **lengths** $L_1, \dots, L_r$; $n = L_1 + \cdots + L_r$
- Mergecost: $M = \sum_{r=1}^{r} L_i \cdot \text{depth}(R_i)$

$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

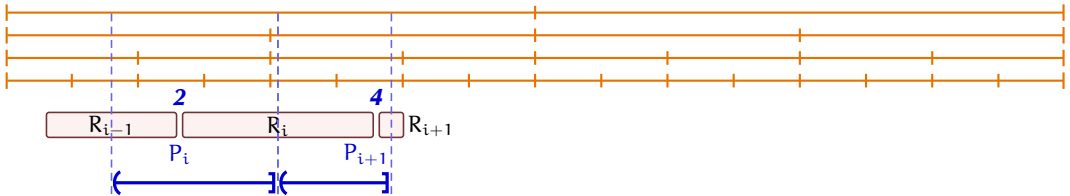
P_i = power of boundary between R_{i-1} and R_i, $i = 2, \dots, r$
$P_1 = P_{r+1} = 0$

$\rightsquigarrow$ To prove $M \leqslant n(\mathcal{H}+2)$, it suffices to show:

$$P_i \leqslant \lceil \log_2\left(\tfrac{n}{L_i}\right) + 1 \rceil \leqslant \log_2\left(\tfrac{n}{L_i}\right) + 2$$
$$P_{i+1} \leqslant \lceil \log_2\left(\tfrac{n}{L_i}\right) + 1 \rceil \leqslant \log_2\left(\tfrac{n}{L_i}\right) + 2$$

- by def.: $\dfrac{|\longleftrightarrow|}{n} \geqslant 2^{-p}$ $\rightsquigarrow$ $P_i \leqslant p$
- we have: $|\longleftrightarrow| \geqslant L_i/2$

$\rightsquigarrow$ $P_i \leqslant \lceil \log_2\left(\tfrac{n}{L_i}\right) \rceil + 1$ □



Conclusion



```
            a = 2 * s1 + n1          # 2 * a'
            b = a + n1 + n2          # 2 * b'
            result = 0
            while True:
                result += 1
                if a >= n:
                    assert b >= a
                    a -= n
                    b -= n
                elif b >= n:
                    break
                assert a < b < n
                a <<= 1
                b <<= 1
            return result

    def found_new_run(self, run):

        p = self.pending
        if p:
            s1 = p[-1].base
            n1 = p[-1].len
            power = powerloop(s1, n1, run.len, self.listlength)
            while len(p) > 1 and p[-2].power > power:
                self.merge_at(-2)
            assert len(p) < 2 or p[-2].power < power
            p[-1].power = power;

    def merge_force_collapse(self):
        p = self.pending
        while len(p) > 1:
```

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

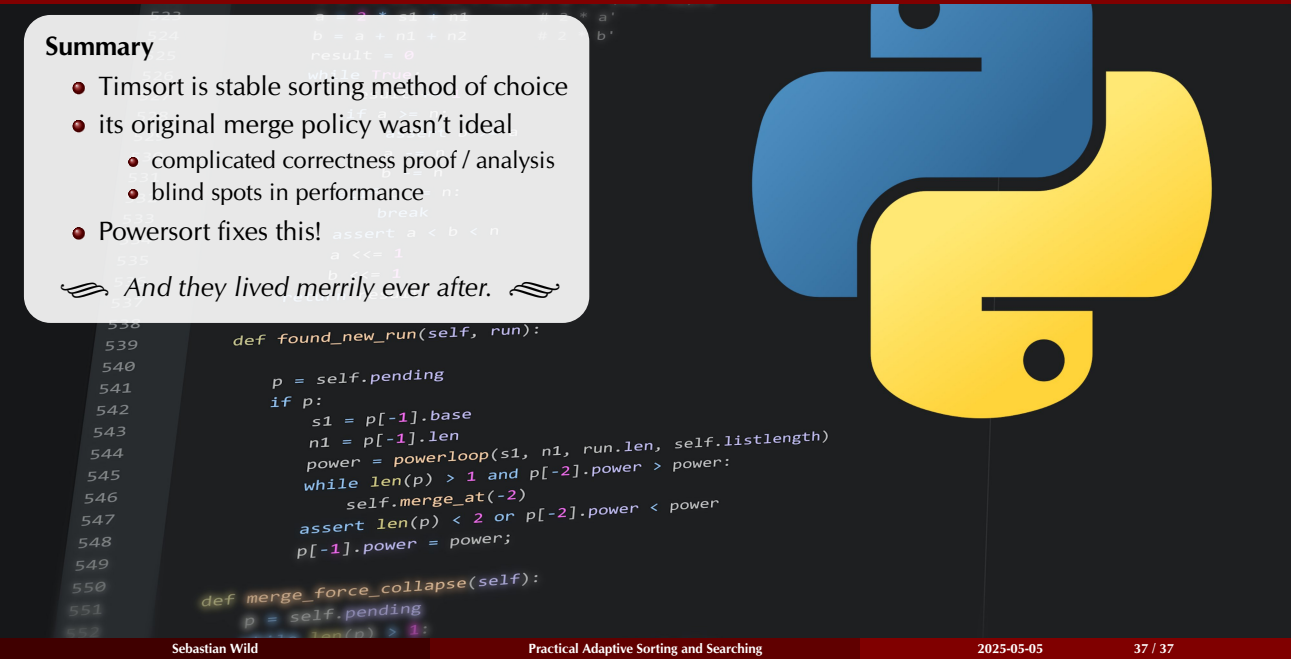


Conclusion

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.

Powersort Bonus

- Adaptive **Multiway** merging version easy



Cawley Gelling, Nebel, Smith, Wild: *Multiway Powersort*, ALENEX 2023

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.

Powersort Bonus

- Adaptive **Multiway** merging version easy



Cawley Gelling, Nebel, Smith, Wild: *Multiway Powersort*, ALENEX **2023**

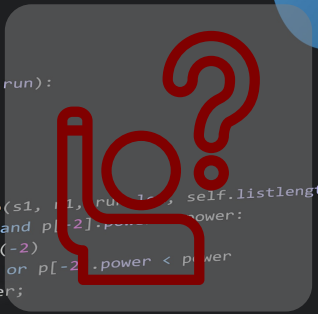
Open problems

1. Good **probabilistic models** for inputs with runs?
2. Analyze **minrunlen** heuristic
3. Powersort in **other libraries** using Timsort
   - numpy & pandas
   - OpenJDK
   - Android Java runtime
   - V8 JavaScript engine
   - GNU STL `std::stable_sort`

I'm hiring postdocs!

- **2 positions**, both can be AofA topics
  1. 2 year project *Computing over compressed graph-structured data*
  2. flexible post, with teaching (»Landesstelle«)
- Ample travel budget (for more AofAs ;))

Marburg sounds cool? → Talk to me!



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`.