Nearly-Optimal Mergesorts

Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs

Sebastian Wild

wild@uwaterloo.ca

joint work with Ian Munro

UNIVERSITY OF WATERLOO



ESA 2018

26th Annual European Symposium on Algorithms

Outline

1 Adaptive Sorting – Status Quo

2 Natural Mergesort

3 Peeksort

4 Powersort

5 Experiments

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**

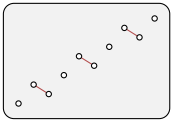
- . . . many more

Adaptive Sorting

Adaptive algorithm: exploit "structure" of input

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

- few **inversions**



- few **runs**

- few **outliers**

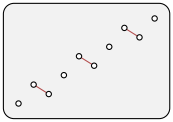
- . . . many more

Adaptive Sorting

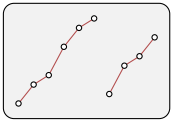
Adaptive algorithm: exploit "structure" of input

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

- few **inversions**



- few **runs**



- few **outliers**

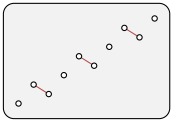
- . . . many more

Adaptive Sorting

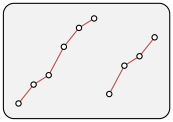
Adaptive algorithm: exploit "structure" of input

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

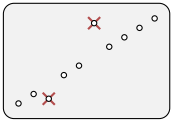
- few **inversions**



- few **runs**



- few **outliers**



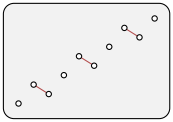
- . . . many more

Adaptive Sorting

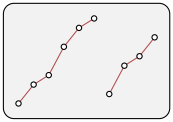
Adaptive algorithm: exploit "structure" of input

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

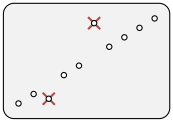
- few **inversions**



- few **runs**



- few **outliers**



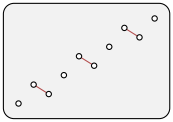
- . . . many more

Adaptive Sorting

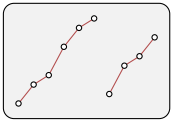
Adaptive algorithm: exploit "structure" of input

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

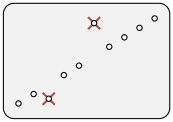
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more



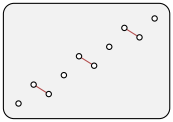
optimal algorithms known for many measures of presortedness

Adaptive Sorting

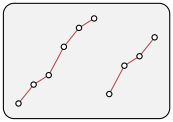
Adaptive algorithm: exploit "structure" of input

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

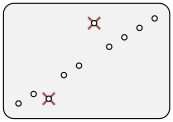
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more



up to constant factors!

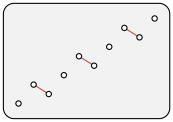
optimal algorithms known for many measures of presortedness

Adaptive Sorting

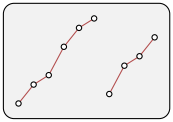
Adaptive algorithm: exploit "structure" of input

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

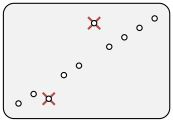
- few **inversions**



- 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

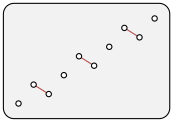
- low overhead for detecting presortedness
- competitive on inputs **without** presortedness

Adaptive Sorting

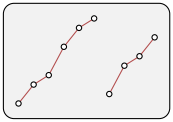
Adaptive algorithm: exploit "structure" of input

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

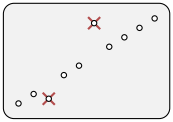
- few **inversions**



- 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

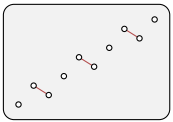
- low overhead for detecting presortedness
- competitive on inputs **without** presortedness

Adaptive Sorting

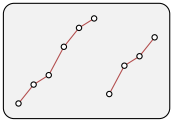
Adaptive algorithm: exploit "structure" of input

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

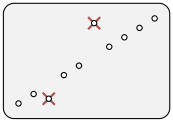
- few **inversions**



- 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

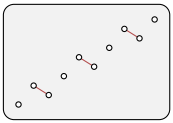
- **low overhead** for detecting presortedness
- competitive on inputs **without** presortedness

Adaptive Sorting

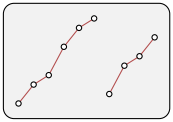
Adaptive algorithm: exploit "structure" of input

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

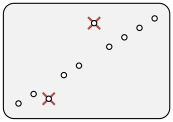
- few **inversions**



- 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

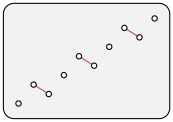
- **low overhead** for detecting presortedness
- competitive on inputs **without** presortedness

Adaptive Sorting

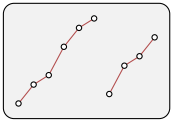
Adaptive algorithm: exploit "structure" of input

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

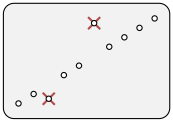
- few **inversions**



- few **runs**



- few **outliers**



- . . . many more

up to constant factors!

optimal algorithms known for many measures of presortedness

Want:

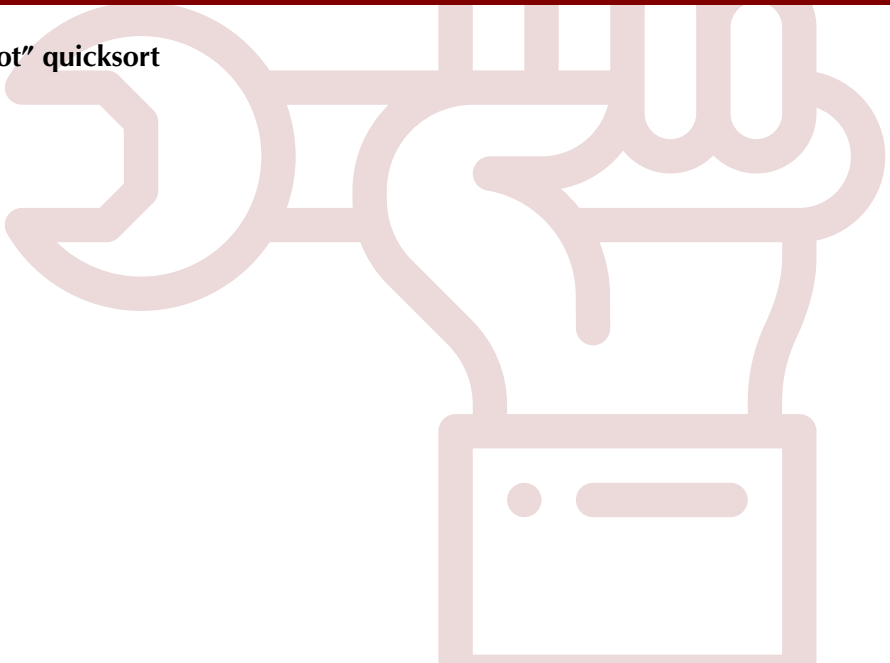
- Optimal up to lower order terms

practical methods

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

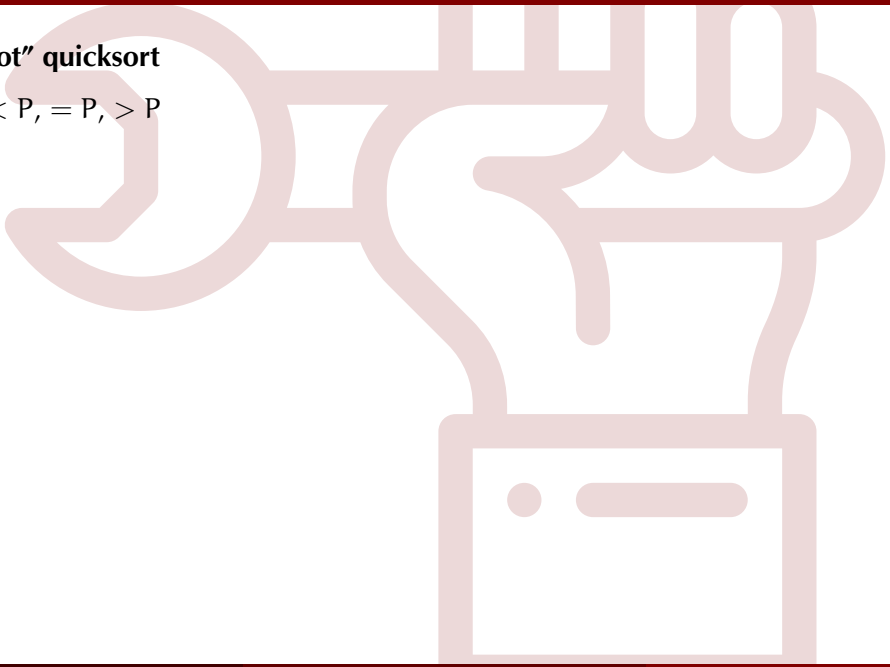
State of the art

1. “fat-pivot” quicksort



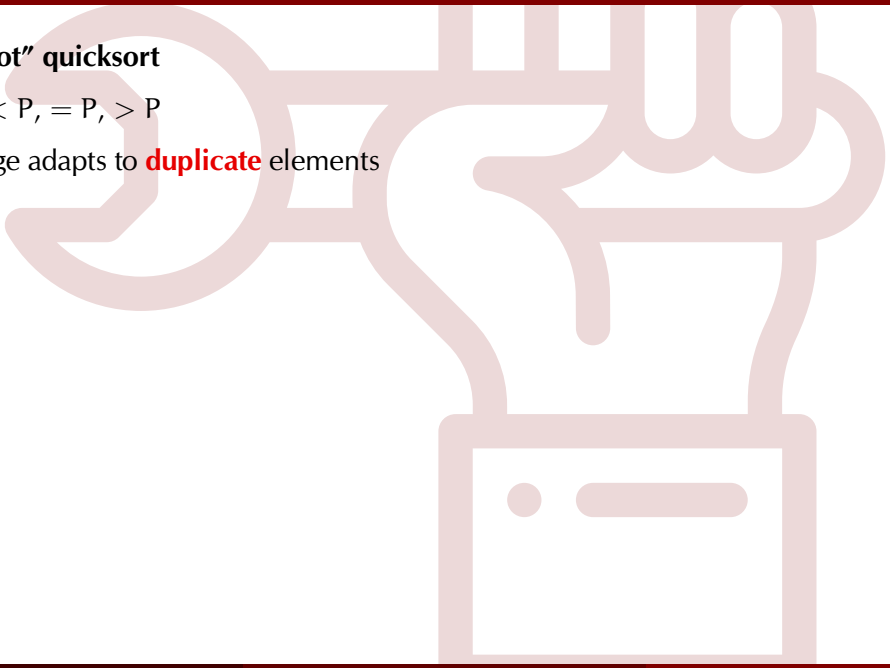
State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$



State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements



State of the art

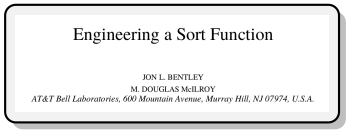
1. **“fat-pivot” quicksort**
   - split $< P$, $= P$, $> P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

1.386 (plain quicksort)
1.188 with median-of-3
1.088 with ninther

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation


Engineering a Sort Function

JON L. BENTLEY
M. DOUGLAS McILROY
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     Engineering a Sort Function

     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

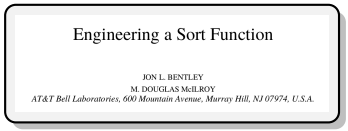
     Engineering a Sort Function

     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...
2. **Timsort**

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     
     Engineering a Sort Function

     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - 

     ...

2. **Timsort**
   - adaptive mergesort variant

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     Engineering a Sort Function

     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...
2. **Timsort**
   - adaptive mergesort variant
   - python, Java, Android, ...

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     
     Engineering a Sort Function

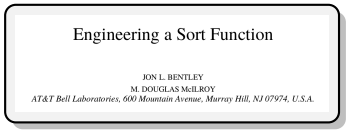
     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...

2. **Timsort**
   - adaptive mergesort variant
   - python, Java, Android, ...
   - adapts to existing **runs**

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     
     Engineering a Sort Function

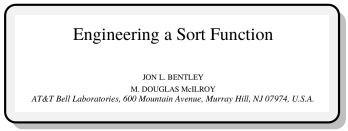
     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...

2. **Timsort**
   - adaptive mergesort variant
   - python, Java, Android, ...
   - adapts to existing **runs**
     - but not optimally!
       factor ≥ 1.5 worse (Buss & Knop 2018)

State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     
     Engineering a Sort Function

     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...

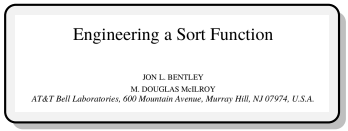
2. **Timsort**
   - adaptive mergesort variant
   - python, Java, Android, ...
   - adapts to existing **runs**
     - but not optimally!
       factor ≥ 1.5 worse (Buss & Knop 2018)



State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation

     
     Engineering a Sort Function

     JON L. BENTLEY
     M. DOUGLAS McILROY
     AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
   - Java, Android, OpenBSD, ...

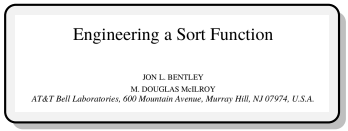
2. **Timsort**
   - adaptive mergesort variant
   - python, Java, Android, ...
   - adapts to existing **runs**
     - but not optimally!
       factor ≥ 1.5 worse (Buss & Knop 2018)



State of the art

1. **“fat-pivot” quicksort**
   - split $< P, = P, > P$
   - average adapts to **duplicate** elements
     - optimal up to small constant factor

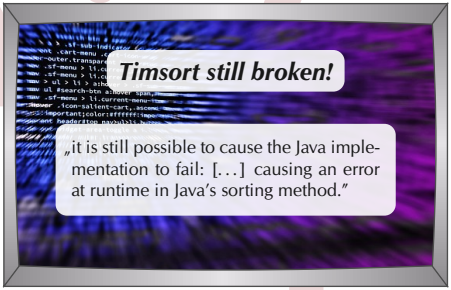
       1.386 (plain quicksort)
       1.188 with median-of-3
       1.088 with ninther
   - low overhead implementation


Engineering a Sort Function

JON L. BENTLEY
M. DOUGLAS McILROY
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

   - Java, Android, OpenBSD, ...

2. **Timsort**
   - adaptive mergesort variant
   - python, Java, Android, ...
   - adapts to existing **runs**
     - but not optimally!
       factor ≥ 1.5 worse (Buss & Knop 2018)



State of the art

1. "fat pivot" quicksort

2. Timsort

Observation:
Timsort's merge rules are quite intricate.

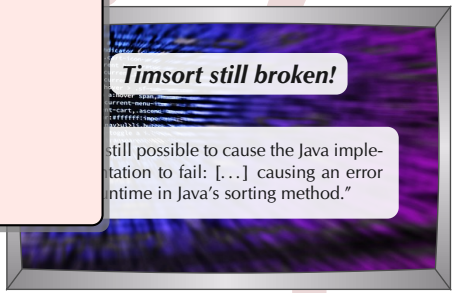
ergesort variant

on™ Java™ ...

existing **runs**

ot optimally!

$\geq$ **1.5 worse** (Buss & Knop 2018)



1. "fat pivot" quicksort

2. Timsort

Observation:
Timsort's merge rules are quite intricate.

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs



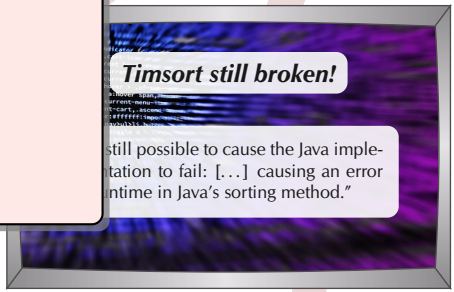
…ergesort variant

…on™ Java™ …

…existing **runs**

…ot optimally!

… $\geq$ **1.5 worse** (Buss & Knop 2018)



1. "fat pivot" quicksort

2. Timsort

Observation:
Timsort's merge rules are quite intricate.

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

❓ *Why these rules?*



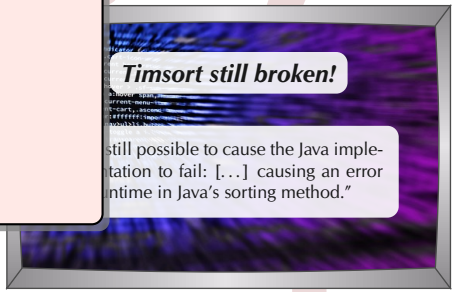
ergesort variant

on™ Java™ ...

existing **runs**

ot optimally!

$\geq$ **1.5 worse** (Buss & Knop 2018)



1. "fat pivot" quicksort

2. Timsort

Observation:
Timsort's merge rules are quite intricate.

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

- *Why these rules?*
- *Why are they so sensitive to small changes?*

cf. Java version!

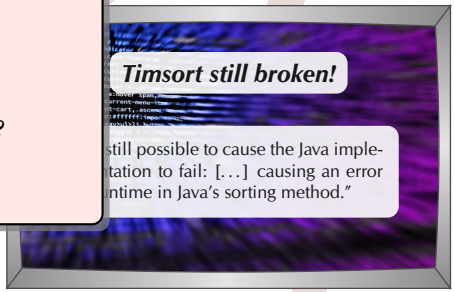
ergesort variant

on™ Java™ ...

existing **runs**

ot optimally!

$\geq$ **1.5 worse** (Buss & Knop 2018)



1. "fat pivot" quicksort

2. Timsort

Observation:
Timsort's merge rules are quite intricate.

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

- *Why these rules?*
- *Why are they so sensitive to small changes?* (cf. Java version!)

*. . . and can't we find **simpler** rules?*



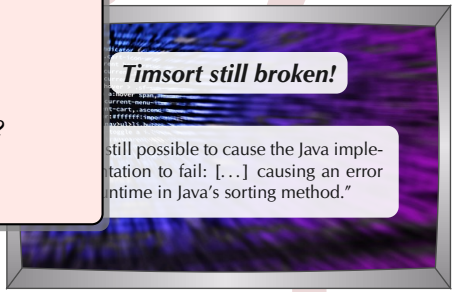
ergesort variant

on™ Java™ . . .

existing **runs**

ot optimally!

$\geq$ **1.5 worse** (Buss & Knop 2018)



Run-Length Entropy



Run-Length Entropy



Our measure of unsortedness: **runs**

Run-Length Entropy



Our measure of unsortedness: **runs**

maximal contiguous sorted range

Run-Length Entropy

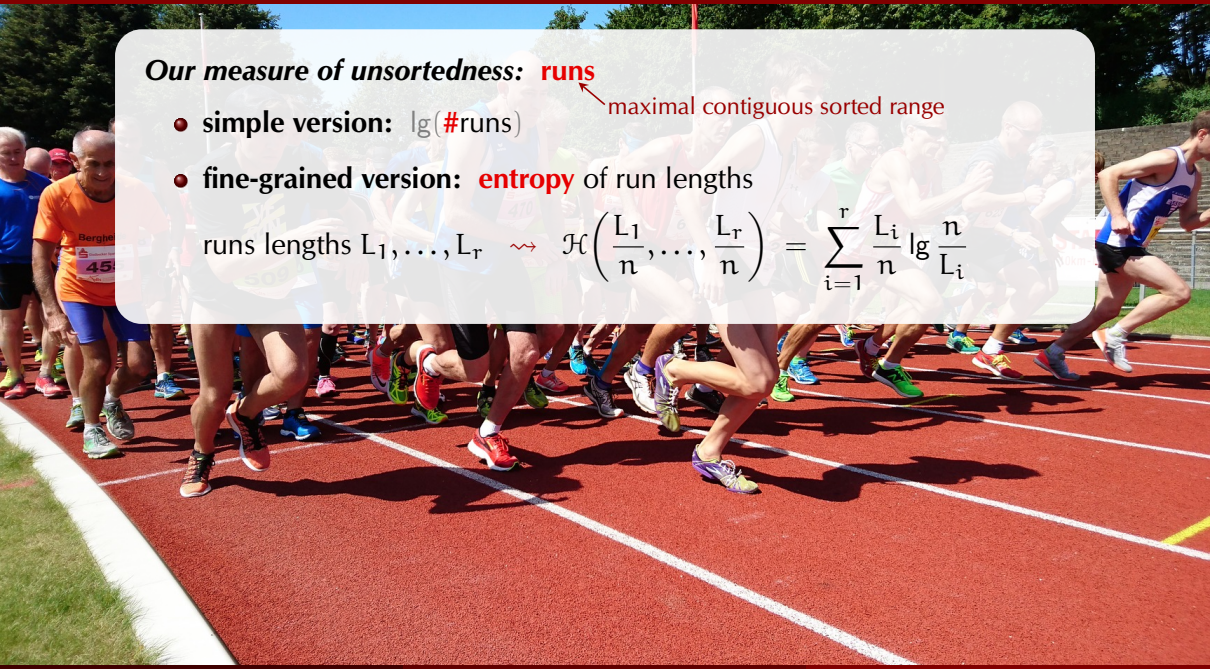


Our measure of unsortedness: **runs**

maximal contiguous sorted range

- **simple version:** $\lg(\#\text{runs})$

Run-Length Entropy



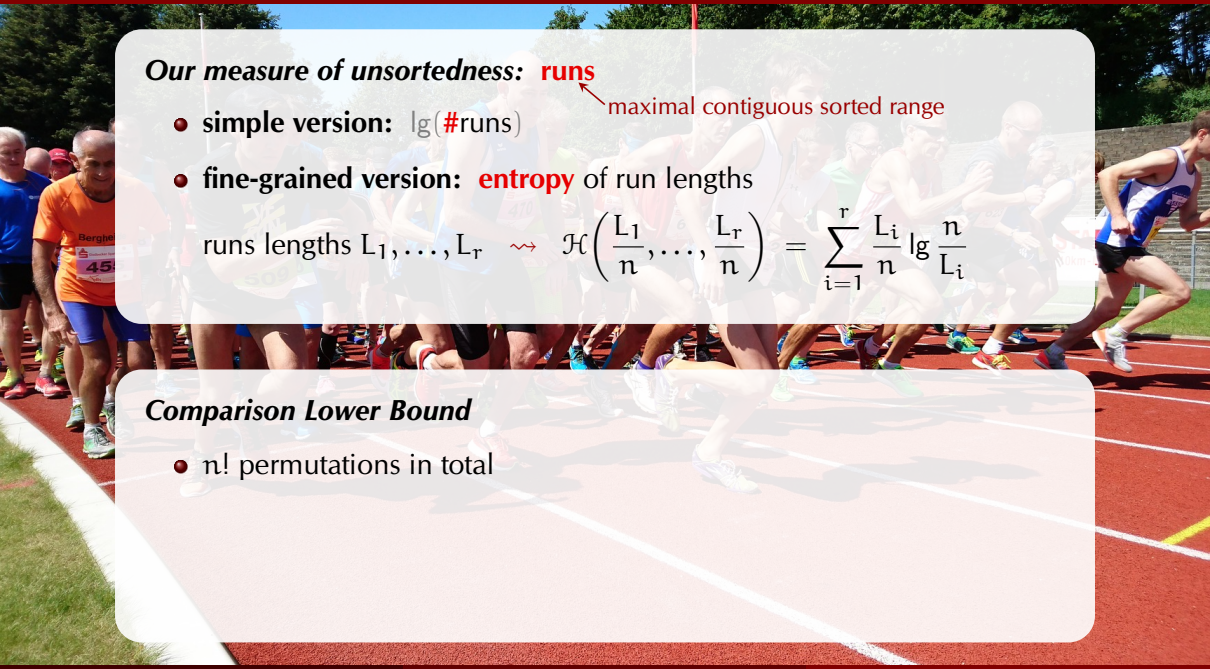
Our measure of unsortedness: **runs**

maximal contiguous sorted range

- **simple version:** $\lg(\#\text{runs})$
- **fine-grained version:** **entropy** of run lengths

runs lengths $L_1, \ldots, L_r \quad \rightsquigarrow \quad \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) = \sum_{i=1}^{r} \frac{L_i}{n} \lg \frac{n}{L_i}$

Run-Length Entropy



Our measure of unsortedness: **runs**

maximal contiguous sorted range

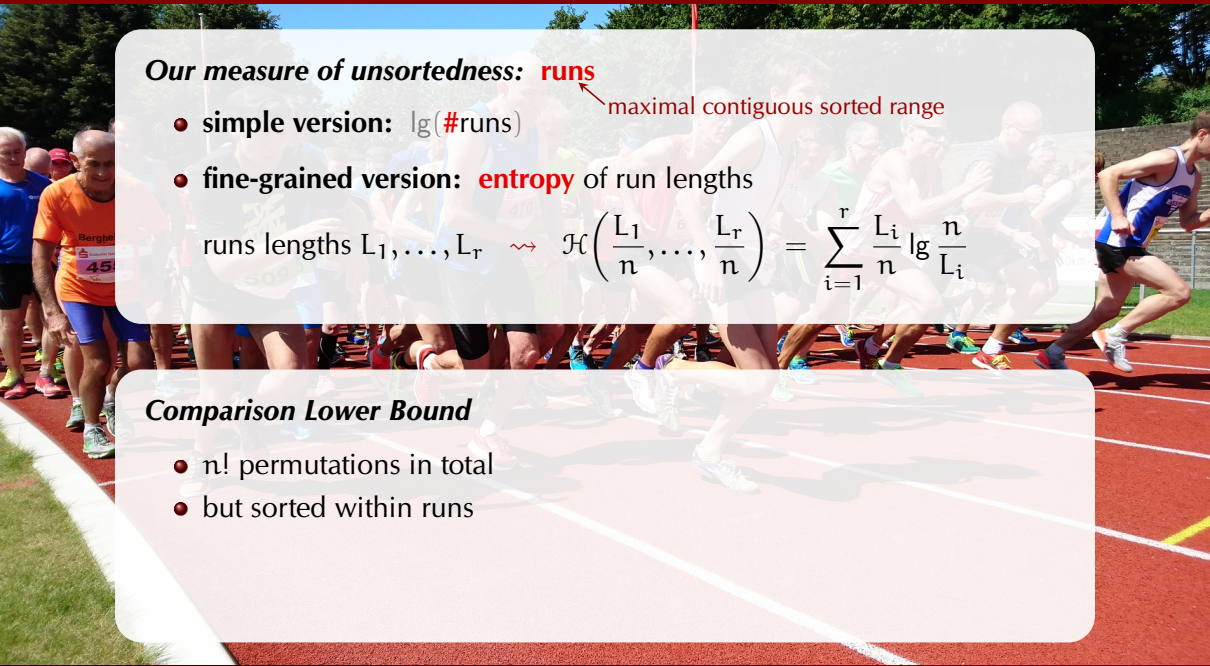
- **simple version:** $\lg(\#\text{runs})$
- **fine-grained version:** **entropy** of run lengths

runs lengths $L_1, \dots, L_r \quad \rightsquigarrow \quad \mathcal{H}\left(\frac{L_1}{n}, \dots, \frac{L_r}{n}\right) = \sum_{i=1}^{r} \frac{L_i}{n} \lg \frac{n}{L_i}$

Comparison Lower Bound

- $n!$ permutations in total

Run-Length Entropy



Our measure of unsortedness: **runs**

maximal contiguous sorted range

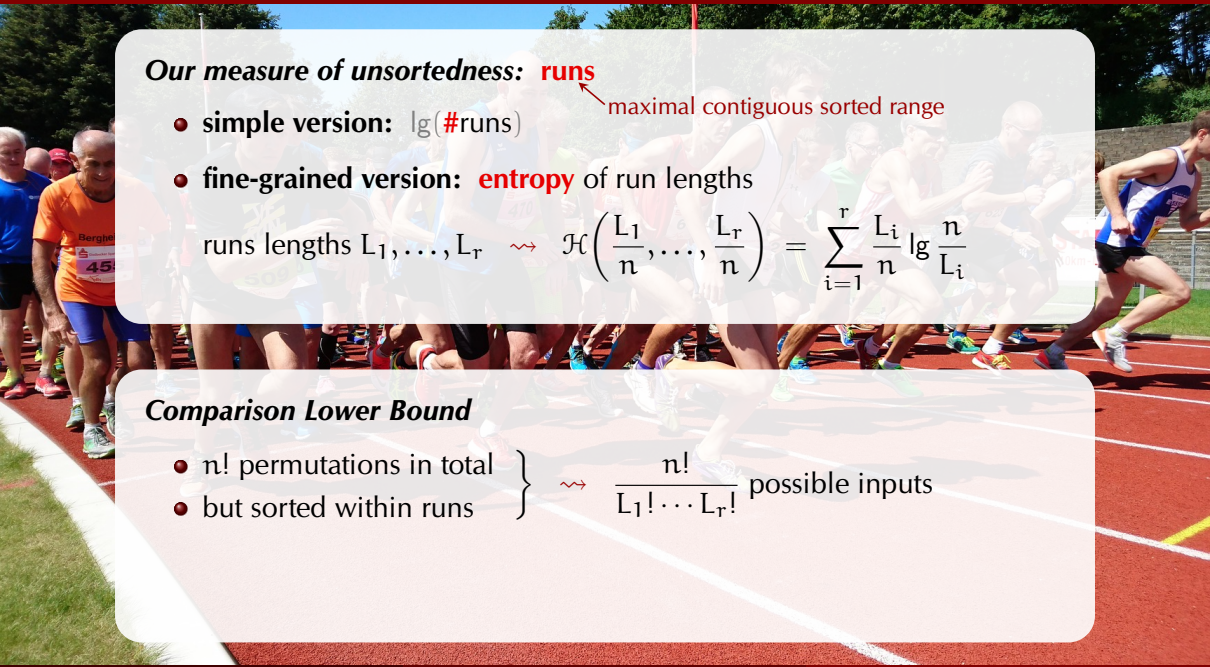
- **simple version:** $\lg(\#\text{runs})$
- **fine-grained version:** **entropy** of run lengths

runs lengths $L_1, \ldots, L_r \quad \rightsquigarrow \quad \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) = \sum_{i=1}^{r} \frac{L_i}{n} \lg \frac{n}{L_i}$

Comparison Lower Bound

- $n!$ permutations in total
- but sorted within runs

Run-Length Entropy



Our measure of unsortedness: **runs**

maximal contiguous sorted range

- **simple version:** $\lg(\#\text{runs})$
- **fine-grained version:** **entropy** of run lengths

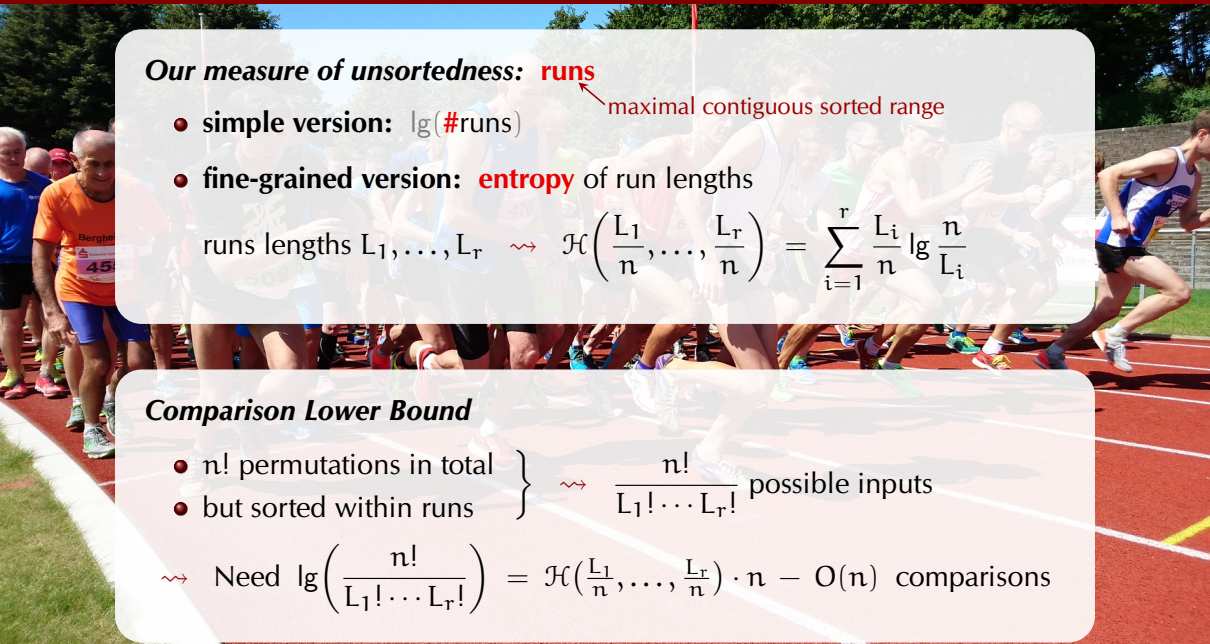
runs lengths $L_1, \ldots, L_r \rightsquigarrow \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) = \sum_{i=1}^{r} \frac{L_i}{n} \lg \frac{n}{L_i}$

Comparison Lower Bound

- $n!$ permutations in total
- but sorted within runs

$\rightsquigarrow \frac{n!}{L_1! \cdots L_r!}$ possible inputs

Run-Length Entropy



Our measure of unsortedness: **runs**

maximal contiguous sorted range

- **simple version:** $\lg(\#\text{runs})$
- **fine-grained version:** **entropy** of run lengths

runs lengths $L_1, \ldots, L_r \rightsquigarrow \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) = \sum_{i=1}^{r} \frac{L_i}{n} \lg \frac{n}{L_i}$

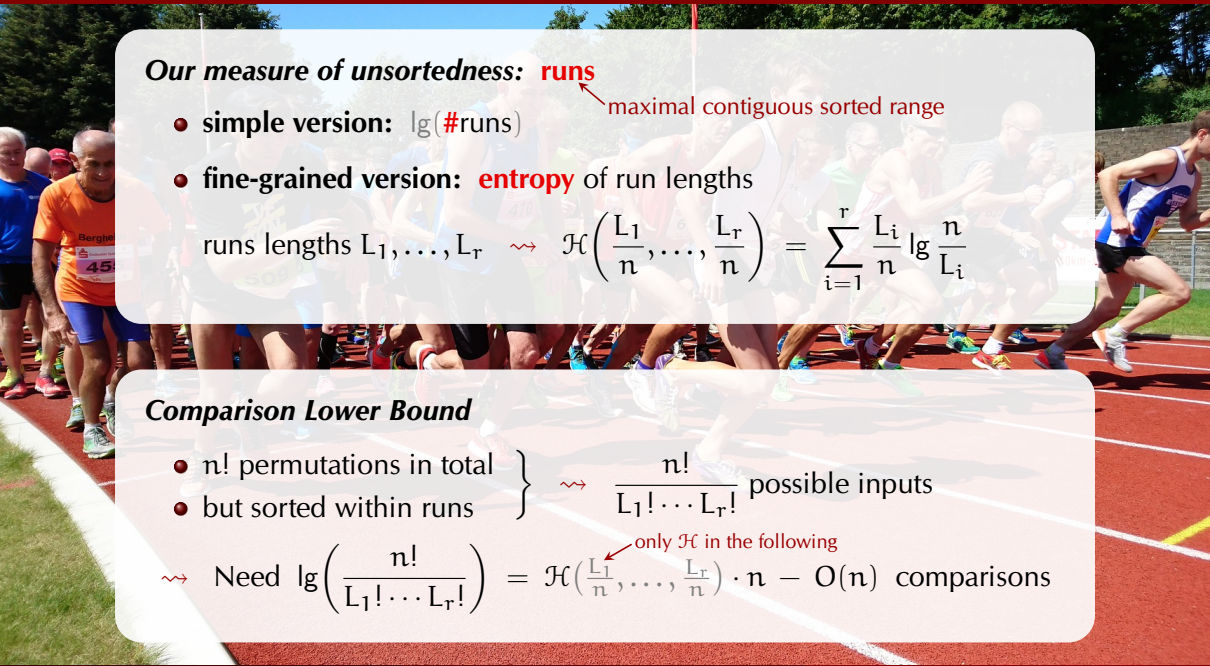
Comparison Lower Bound

- $n!$ permutations in total
- but sorted within runs

$\rightsquigarrow \frac{n!}{L_1! \cdots L_r!}$ possible inputs

$\rightsquigarrow$ Need $\lg\left(\frac{n!}{L_1! \cdots L_r!}\right) = \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) \cdot n - O(n)$ comparisons

Run-Length Entropy



Our measure of unsortedness: **runs**

maximal contiguous sorted range

- **simple version:** $\lg(\#\text{runs})$
- **fine-grained version:** **entropy** of run lengths

runs lengths $L_1, \ldots, L_r \rightsquigarrow \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) = \sum_{i=1}^{r} \frac{L_i}{n} \lg \frac{n}{L_i}$

Comparison Lower Bound

- $n!$ permutations in total
- but sorted within runs

$\rightsquigarrow$ $\frac{n!}{L_1! \cdots L_r!}$ possible inputs

only $\mathcal{H}$ in the following

$\rightsquigarrow$ Need $\lg\left(\frac{n!}{L_1! \cdots L_r!}\right) = \mathcal{H}\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) \cdot n - O(n)$ comparisons

Outline

1 Adaptive Sorting – Status Quo

2 Natural Mergesort

3 Peeksort

4 Powersort

5 Experiments

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

Conceptually two steps:

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

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

interleaved in code

Conceptually two steps:

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

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

interleaved in code

Conceptually two steps:

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

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

interleaved in code

Conceptually two steps:

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

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

interleaved in code

Conceptually two steps:

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

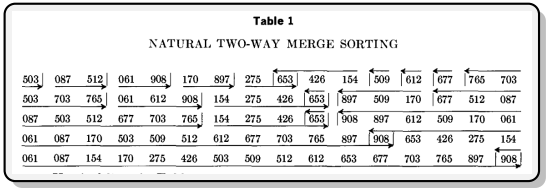


Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort (Knuth 1973)

interleaved in code

Conceptually two steps:

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

(Knuth: simple bottom-up)

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:

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

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

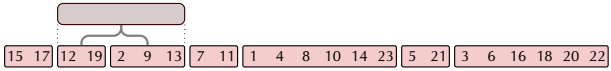
- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

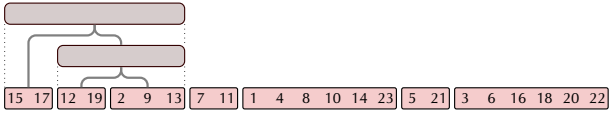
- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

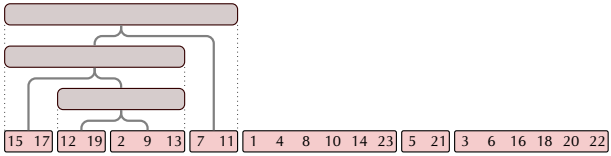
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

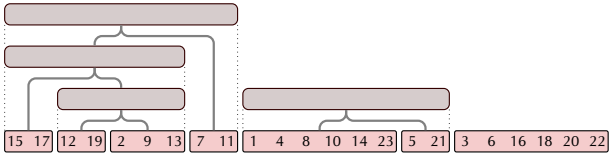
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

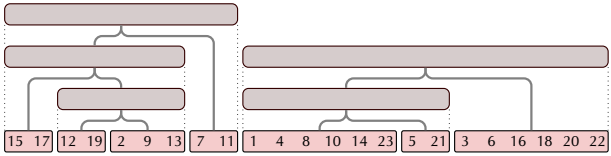
- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

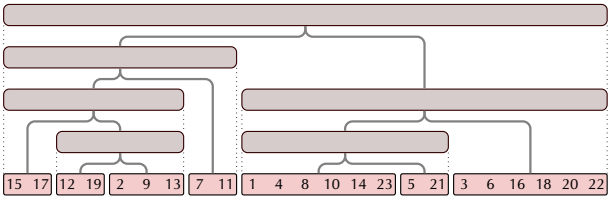
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

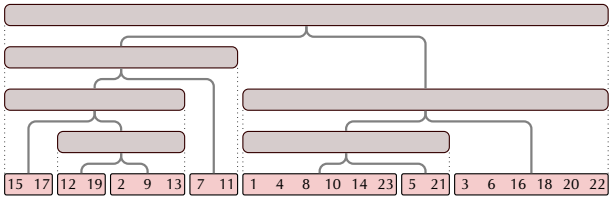
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Merge costs

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

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

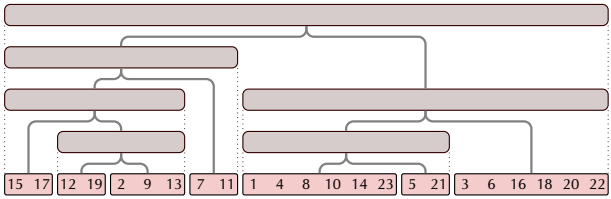
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Merge costs

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

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

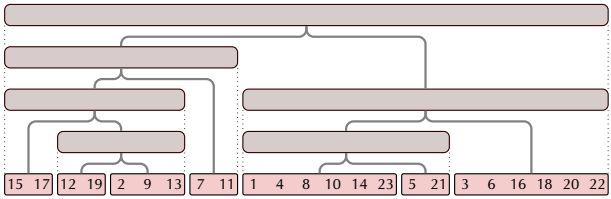
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Merge costs

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

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

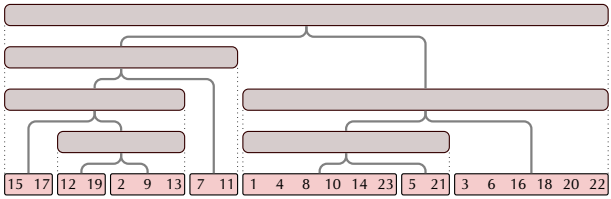
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Merge costs

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

Natural Mergesort

"natural" mergesort = run-adaptive mergesort

Conceptually two steps:

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

Here:

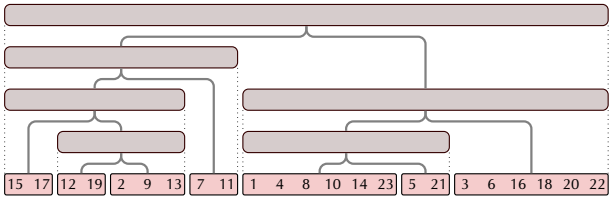
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge trees:



Merge costs

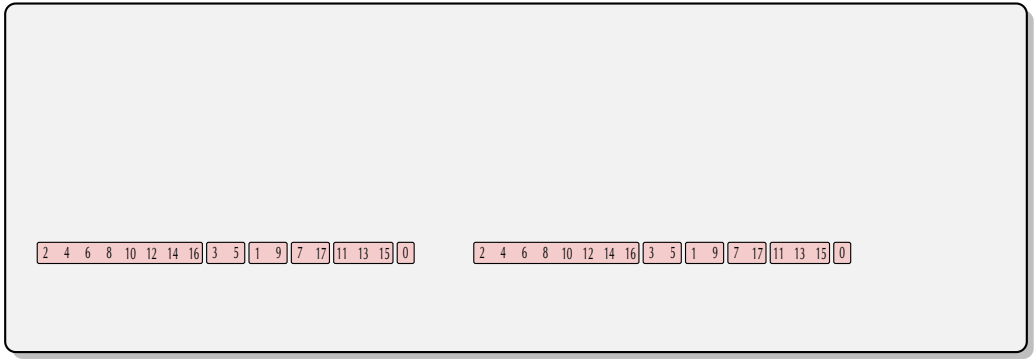
- cost of merge := size of output
  - ≈ memory transfers
  - ⩾ #cmps
- total cost = total area of 

Mergesort meets search trees

Different merge trees yield different cost!

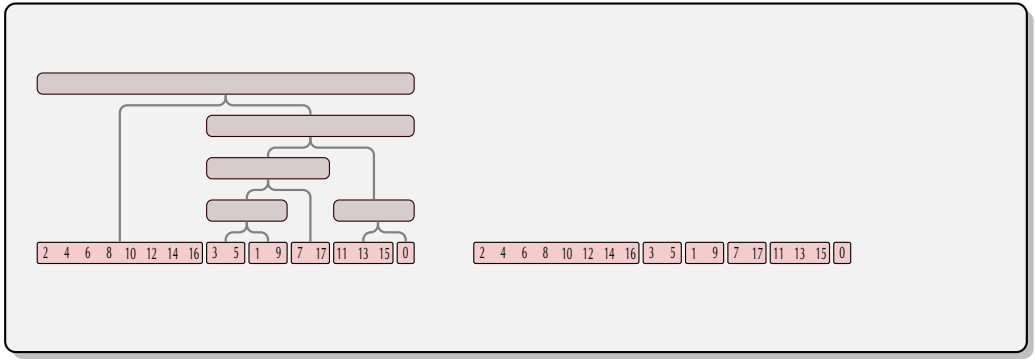
Mergesort meets search trees

Different merge trees yield different cost!



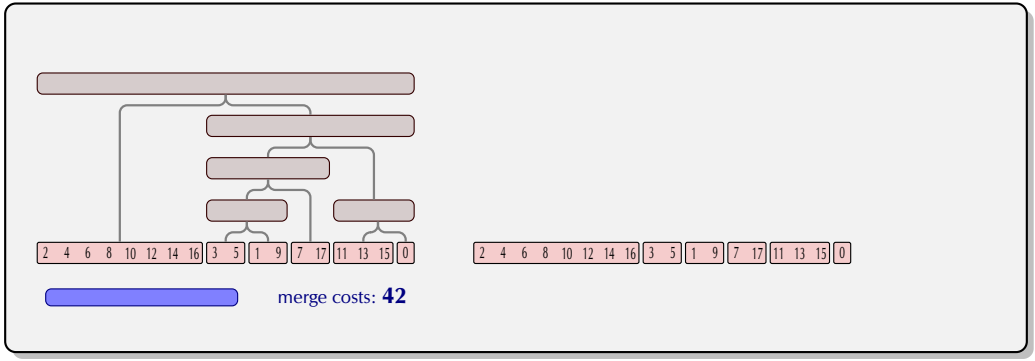
Mergesort meets search trees

Different merge trees yield different cost!



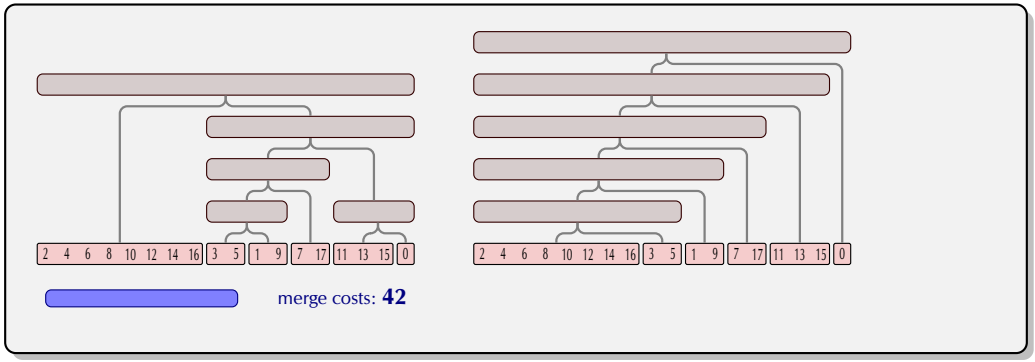
Mergesort meets search trees

Different merge trees yield different cost!



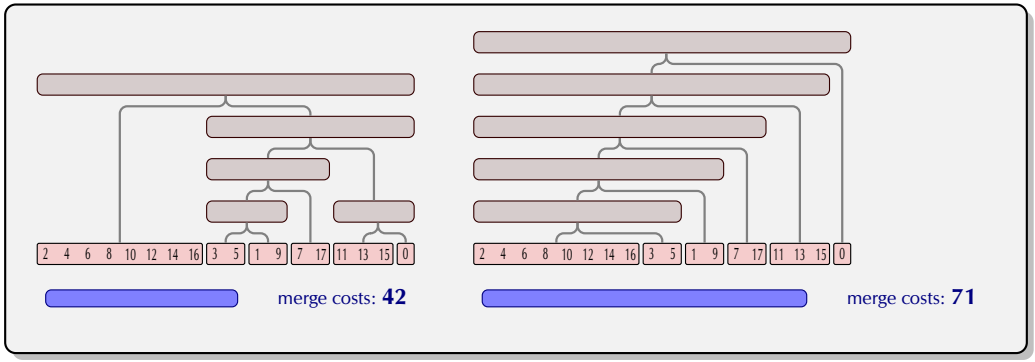
Mergesort meets search trees

Different merge trees yield different cost!



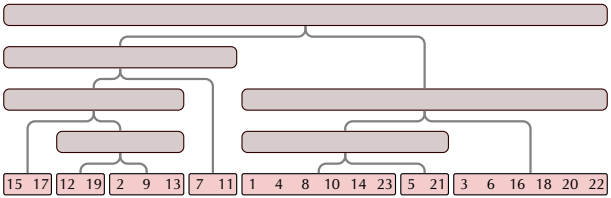
Mergesort meets search trees

Different merge trees yield different cost!



Mergesort meets search trees

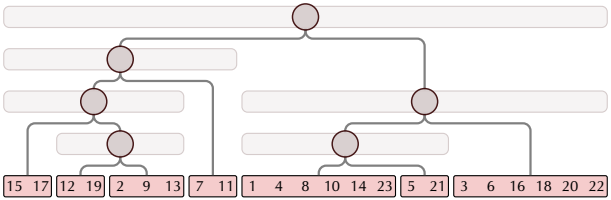
Different merge trees yield different cost!



Merge cost $=$ total area of 

Mergesort meets search trees

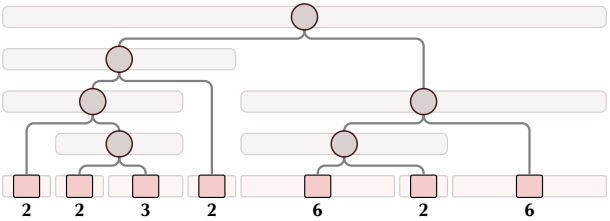
Different merge trees yield different cost!



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

Mergesort meets search trees

Different merge trees yield different cost!



Merge cost = total area of 

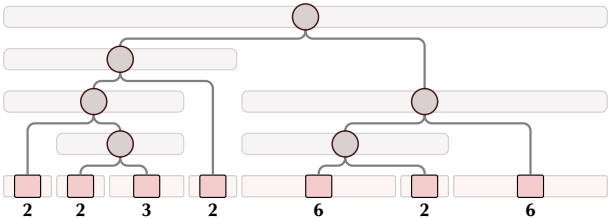
= total length of paths to all array entries

= weighted external path length

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

Mergesort meets search trees

Different merge trees yield different cost!



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

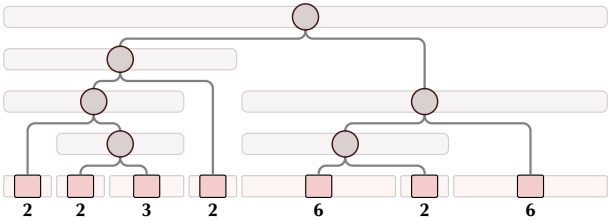
$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

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

Mergesort meets search trees

Different merge trees yield different cost!

How to compute good merge tree?



Merge cost = total area of 

= total length of paths to all array entries

= weighted external path length

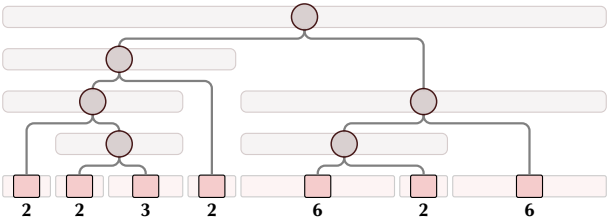
$$\sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ **optimal** merge tree

= optimal **BST** for leaf weights $L_1, \ldots, L_r$

Mergesort meets search trees

Different merge trees yield different cost!



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

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

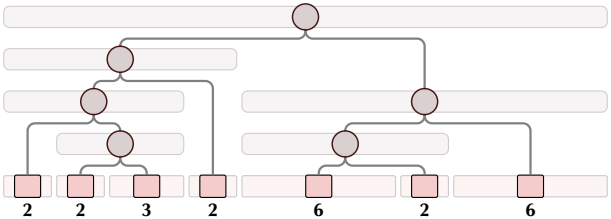
$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***

Mergesort meets search trees

Different merge trees yield different cost!



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

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

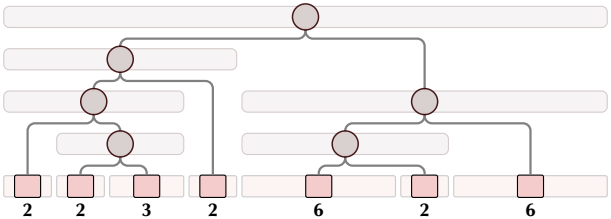
$\rightsquigarrow$ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$

How to compute good merge tree?

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

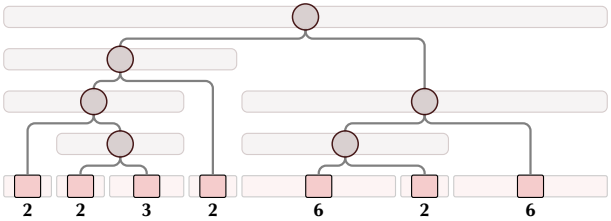
How to compute good merge tree?

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

indep. discovered

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

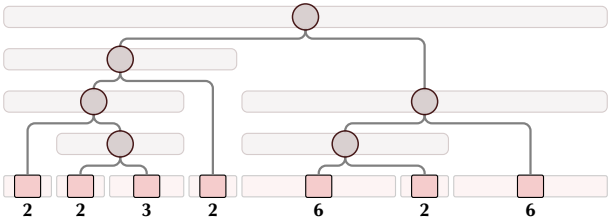
How to compute good merge tree?

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

indep. discovered
- Golin & Sedgewick 1993

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

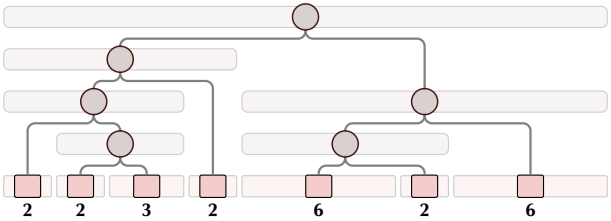
How to compute good merge tree?

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

indep. discovered
- Golin & Sedgewick 1993
- Takaoka 1998

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

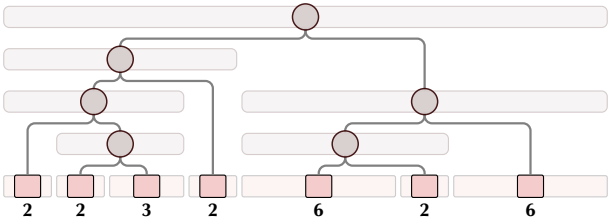
How to compute good merge tree?

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

indep. discovered
- Golin & Sedgewick 1993
- Takaoka 1998
- Barbay & Navarro 2009

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

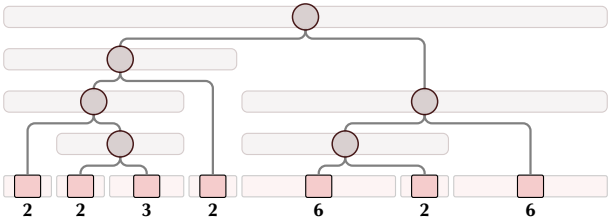
How to compute good merge tree?

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

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

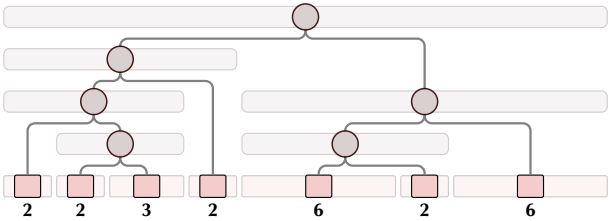
How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

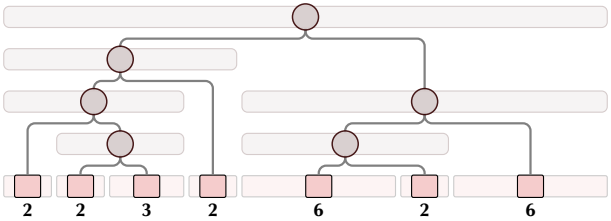
- ***Huffman merge***
  - merge shortest runs
  - must sort lengths

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

- **not stable**

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

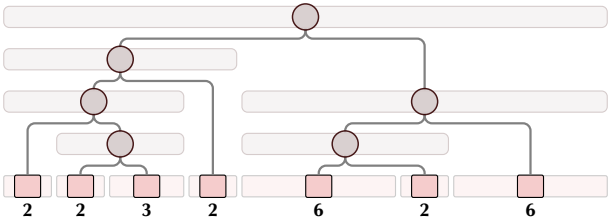
- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - **not stable**

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

- ***Hu-Tucker merge***

Mergesort meets search trees

Different merge trees yield different cost!



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

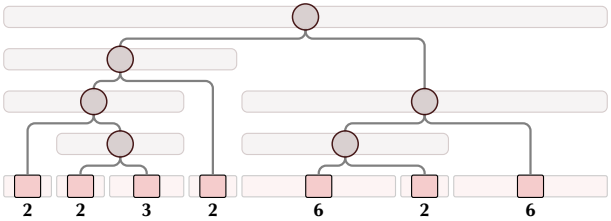
= optimal **BST** for leaf weights $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths ← indep. discovered
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
  - ϟ **not stable**
- ***Hu-Tucker merge***
  - optimal alphabetic tree

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

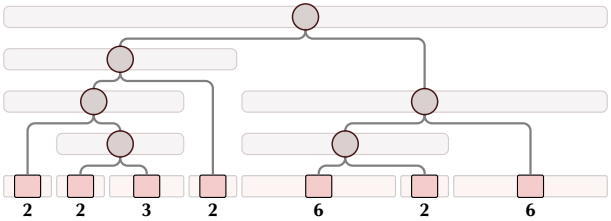
- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - **not stable**

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

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

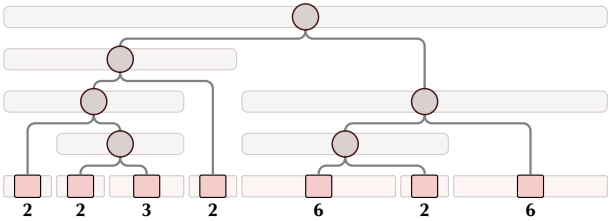
- ***Huffman merge***
  - merge shortest runs
  - must sort lengths
  - ↯ **not stable**

indep. discovered
- 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 search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

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

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

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

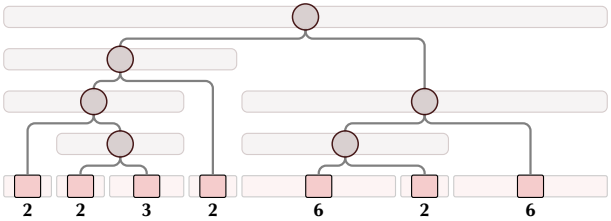


. . . 70s are calling

nearly-optimal BST merge

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths ← indep. discovered
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
  - ↯ **not stable**
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

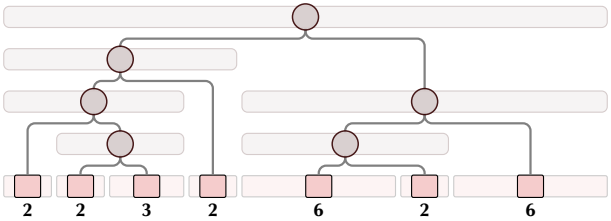


…70s are calling

- ***nearly-optimal BST merge***
  - simple (greedy) linear-time methods!

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths ← indep. discovered
  - ↯ **not stable**
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ↯ **complicated** algorithm

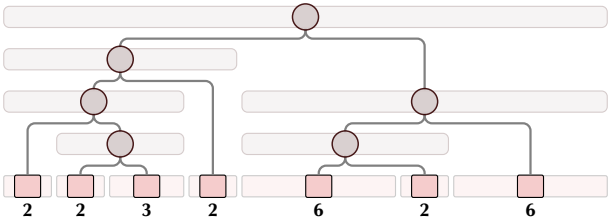


…70s are calling

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths ← indep. discovered
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
  - ϟ **not stable**
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ϟ **complicated** algorithm



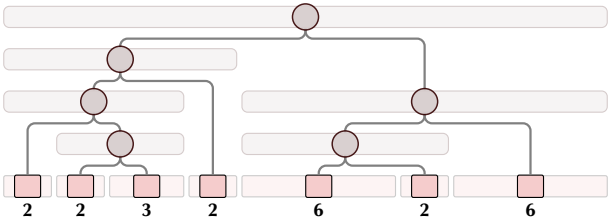
…70s are calling

nearly-optimal BST merge

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths ← indep. discovered
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
  - ϟ **not stable**
- ***Hu-Tucker merge***
  - optimal alphabetic tree
  - have to store lengths
  - ϟ **complicated** algorithm



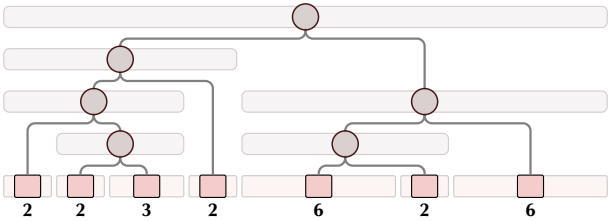
…70s are calling

nearly-optimal BST merge

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

Mergesort meets search trees

Different merge trees yield different cost!



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 $L_1, \ldots, L_r$

How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs
  - must sort lengths ← indep. discovered
  - ↯ **not stable**
    - 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?

Outline

1 Adaptive Sorting – Status Quo

2 Natural Mergesort

3 Peeksort

4 Powersort

5 Experiments

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

Peeksort

Method 1: *weight-balancing*
(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

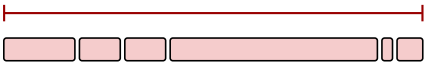


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

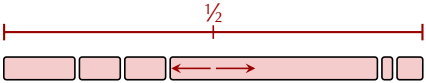


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

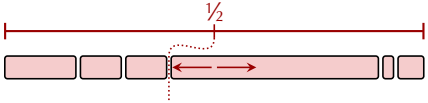


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

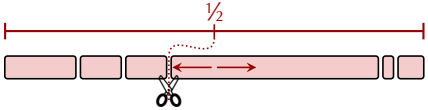


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

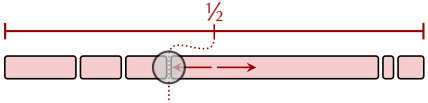


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

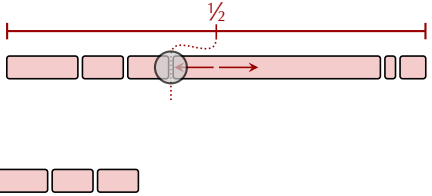


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

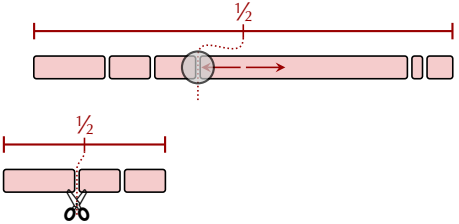


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

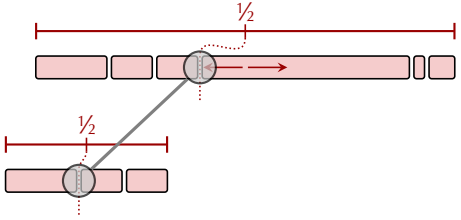


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

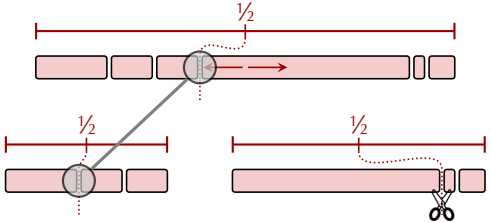


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

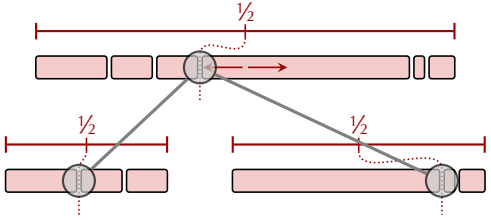


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

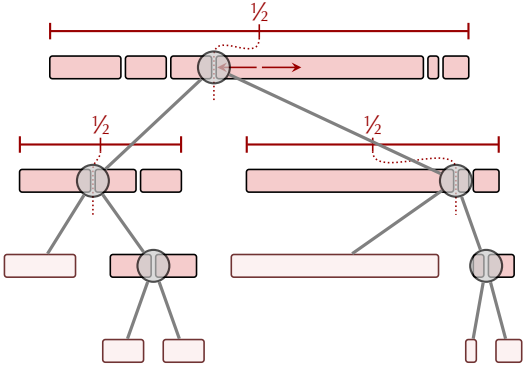


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees

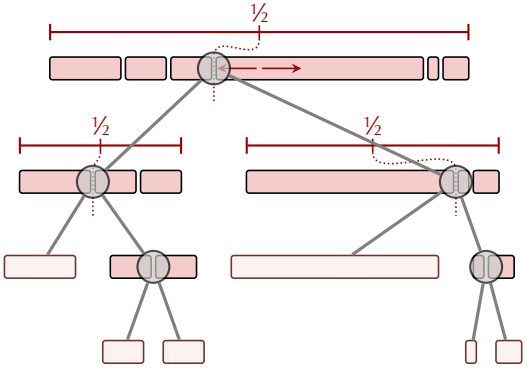


Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort

can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary

⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:

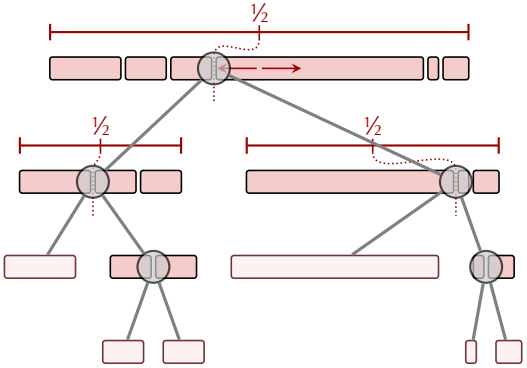
⇝ stores outermost runs

⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*
(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary

⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:

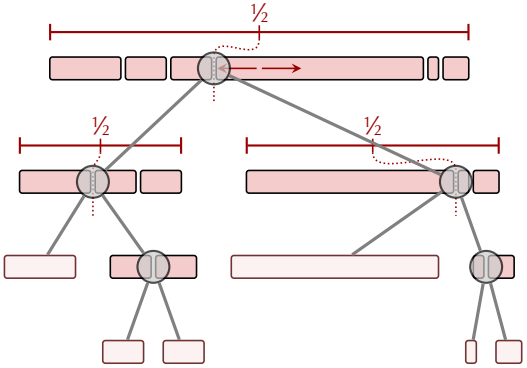
⇝ stores outermost runs
⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:

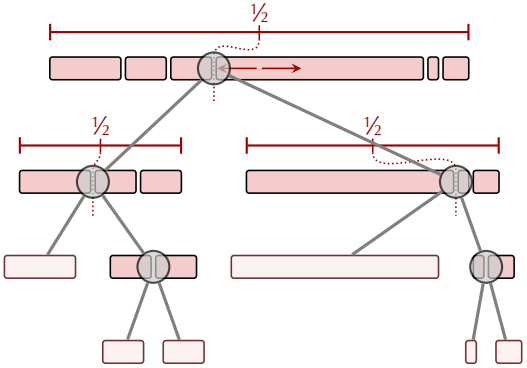
⇝ stores outermost runs

⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*
(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:

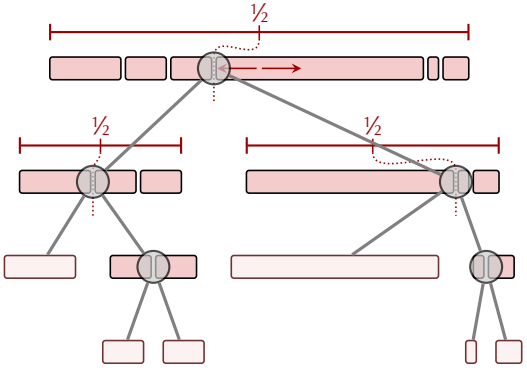
  - ⇝ stores outermost runs
  - ⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:

⇝ stores outermost runs

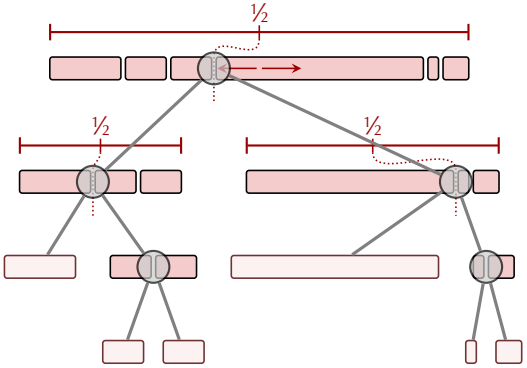
⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:

⇝ stores outermost runs

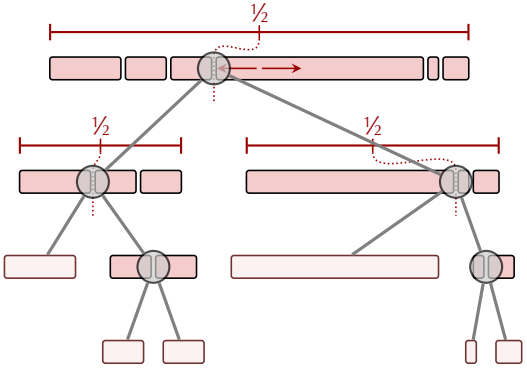
⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:



⇝ stores outermost runs

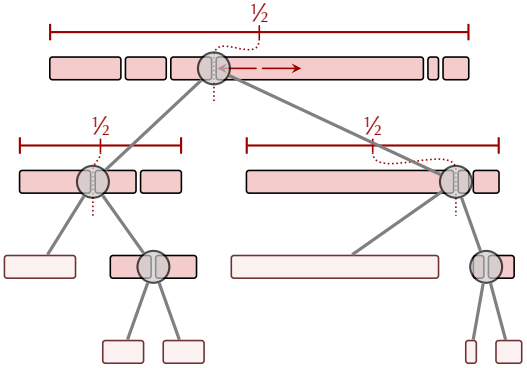
⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:



⇝ stores outermost runs

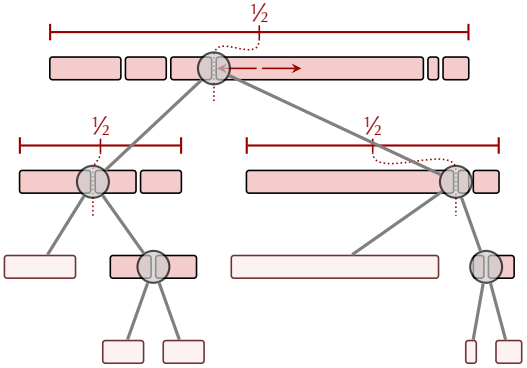
⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



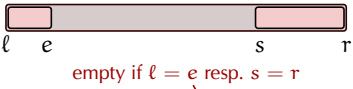
can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:



⇝ stores outermost runs

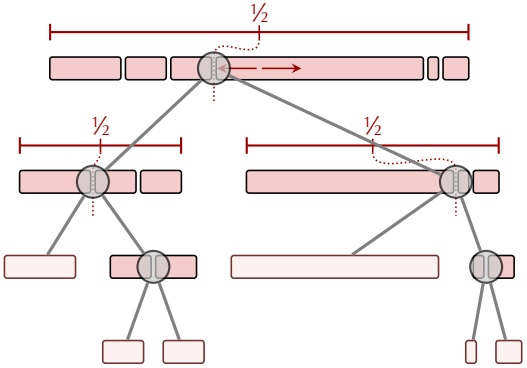
⇝ each run scanned only once

Peeksort

Method 1: *weight-balancing*

(Mehlhorn 1975, Bayer 1975)

- choose root to balance subtree weights
- recurse on subtrees



Peeksort



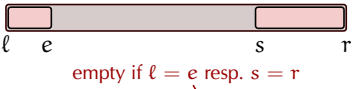
can simulate weight-balancing
without knowing/storing all runs!

- "peek" at middle of array
  to find closest run boundary



⇝ split there and recurse

- can avoid redundant work:
  - find full run straddling midpoint
  - 4 parameters for recursive calls:



  ⇝ stores outermost runs

  ⇝ each run scanned only once

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

$\rightsquigarrow$ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

- both are optimal up to $O(n)$ terms

$\rightsquigarrow$ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

- both are optimal up to $O(n)$ terms

⇝ *Peeksort **exploits** existing **runs** **optimally** up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort **exploits** existing **runs** **optimally** up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

Are we done then?

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Are we done then?

- have to store lengths

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

Are we done then?

~~have to store lengths~~

$\rightsquigarrow$ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs

merge cost

- both are optimal up to $O(n)$ terms

$\rightsquigarrow$ *Peeksort* ***exploits*** *existing* ***runs*** ***optimally*** *up to lower order terms!*

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs — merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs optimally*** *up to lower order terms!*

Are we done then?

~~have to store lengths~~

extra scan to detect runs

- one run at a time
- we load runs (peeking) **without putting memory transfers to good use**

… can't we do better?

- Timsort does better: newly detected run usually merged soon after

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs — merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs optimally*** *up to lower order terms!*

Are we done then?

~~have to store lengths~~

extra scan to detect runs

- one run at a time
- we load runs (peeking) **without putting memory transfers to good use**

… can't we do better?

- Timsort does better: newly detected run usually merged soon after

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs — merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs optimally*** *up to lower order terms!*

Are we done then?

~~👎 have to store lengths~~

👎 extra scan to detect runs

- one run at a time
- we load runs (peeking) **without putting memory transfers to good use**

… can't we do better?

- Timsort does better: newly detected run usually merged soon after

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs — merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs optimally*** *up to lower order terms!*

Are we done then?

~~have to store lengths~~

- extra scan to detect runs
  - one run at a time
  - we load runs (peeking) **without putting memory transfers to good use**



… can't we do better?

- Timsort does better: newly detected run usually merged soon after

Analysis of peeksort

Theorem (Horibe 1977, Bayer 1975:)

Weight-balancing on leaf probabilities $\alpha_1, \dots, \alpha_r$ *yields a BST with search cost* $C \leqslant \mathcal{H}(\alpha_1, \dots, \alpha_r) + 2$.

exp. #cmps to find random leaf chosen with prob. α_i.

⇝ immediate corollary:

Peeksort incurs merge cost $M \leqslant (\mathcal{H} + 2)n$.

Peeksort needs $C \leqslant n + (\mathcal{H} + 2)n$ cmps.

detect runs — merge cost

- both are optimal up to $O(n)$ terms

⇝ *Peeksort* ***exploits*** *existing* ***runs optimally*** *up to lower order terms!*

Are we done then?

~~have to store lengths~~

- extra scan to detect runs
  - one run at a time
  - we load runs (peeking) **without putting memory transfers to good use**



… can't we do better?

- Timsort does better: newly detected run usually merged soon after

Outline

1 Adaptive Sorting – Status Quo

2 Natural Mergesort

3 Peeksort

4 Powersort

5 Experiments

The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details 

⇝ cannot use weight-balancing

The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details 

⇝ cannot use weight-balancing

The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

$\rightsquigarrow$ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)

The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

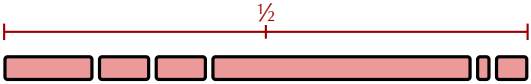
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

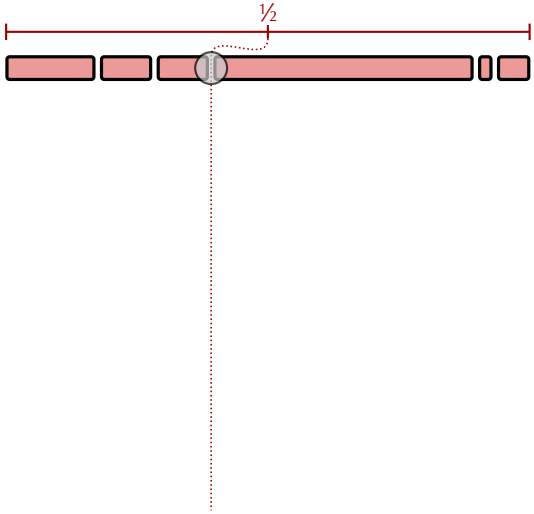
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

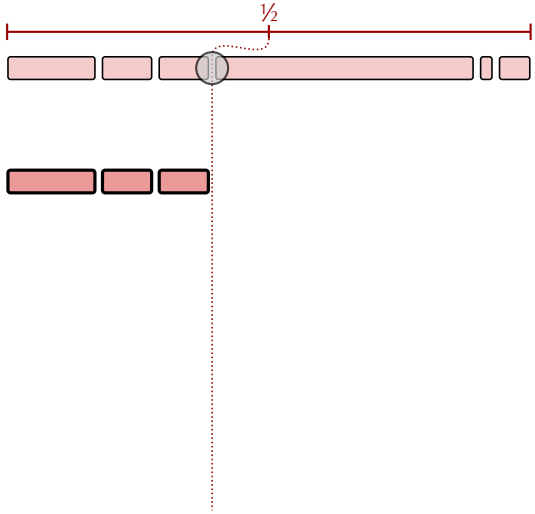
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

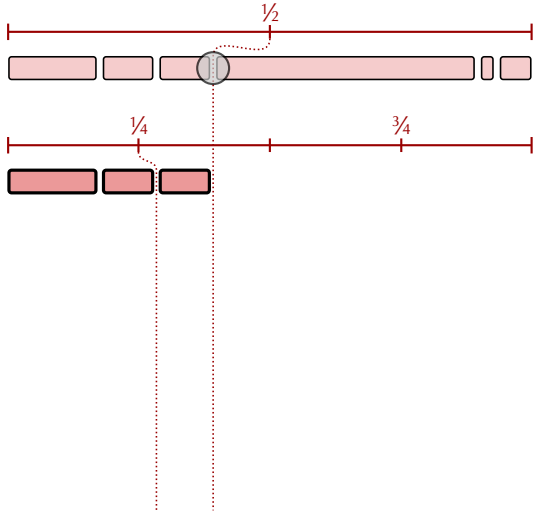
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

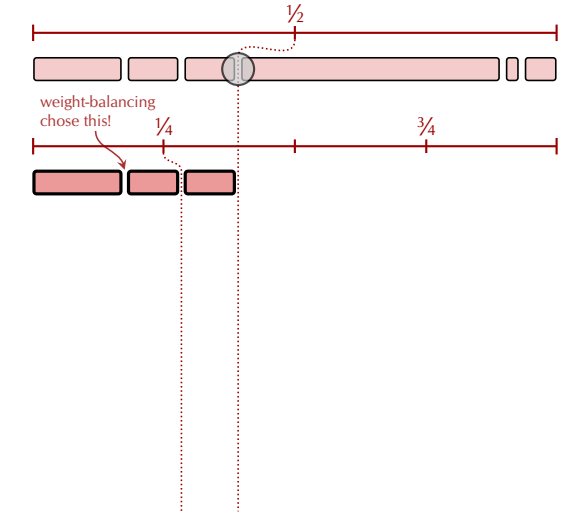
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

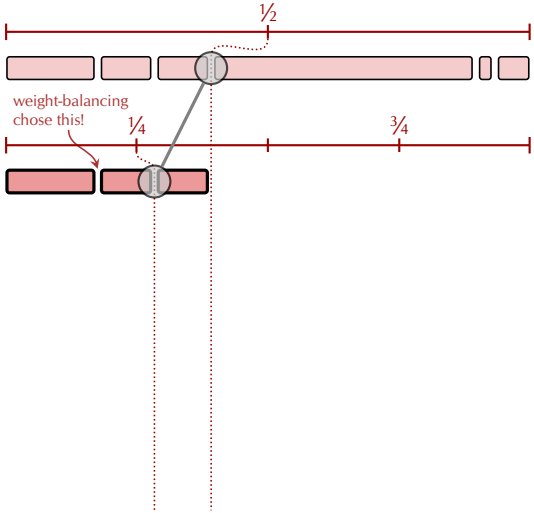
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

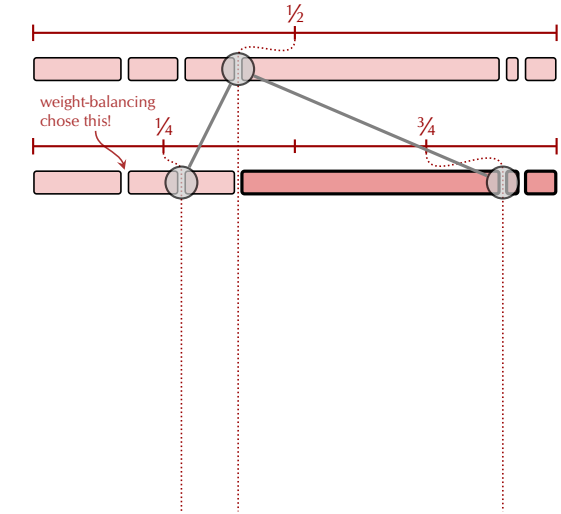
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

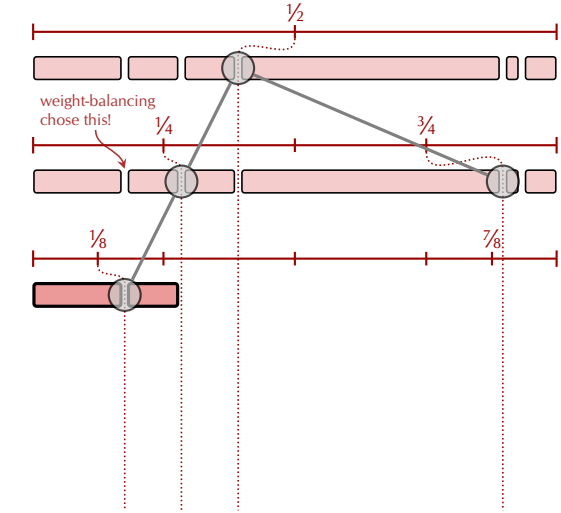
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

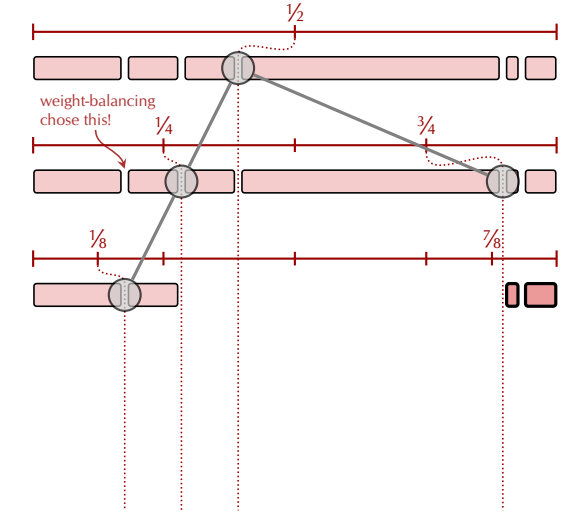
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

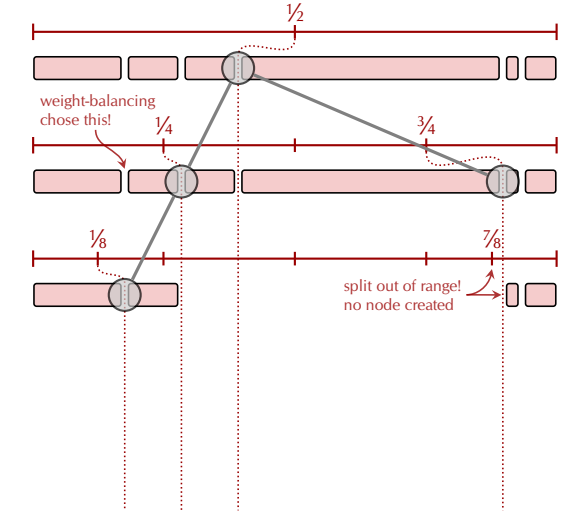
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

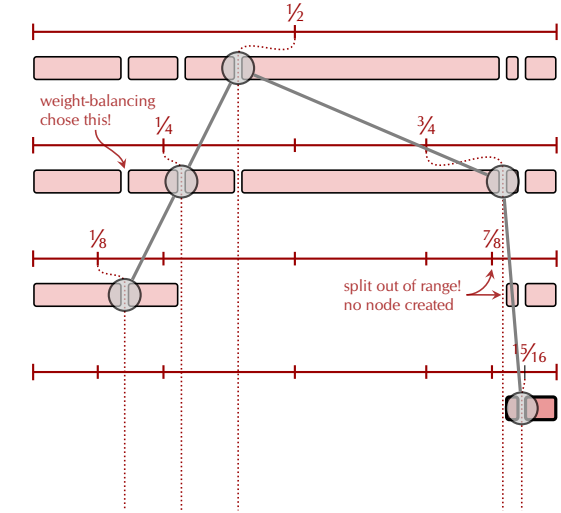
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

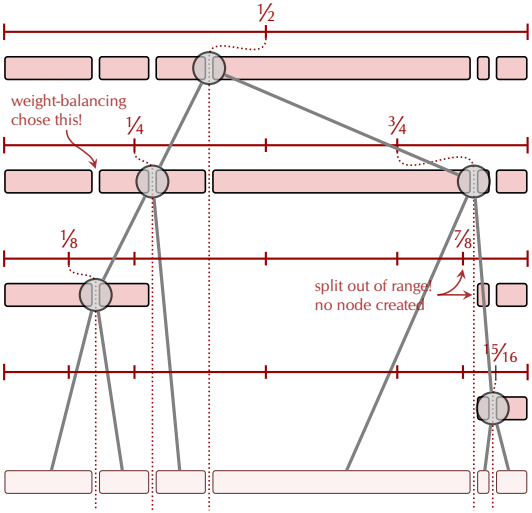
- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Alternative view: *node powers*

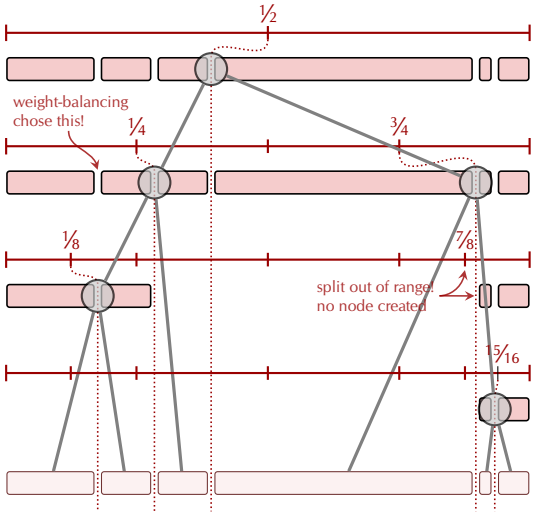
- inner node ⇝ midpoint interval
  ⟝⟞ = normalized interval $[1..n] \mapsto [0,1]$
- power = min ℓ s.t. ⟝⟞ contains $c \cdot 2^{-\ell}$

⇝ depends only on 2 runs



Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

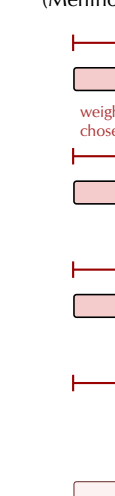
devil is in the details

⇝ cannot use weight-balancing

Alternative view: *node powers*

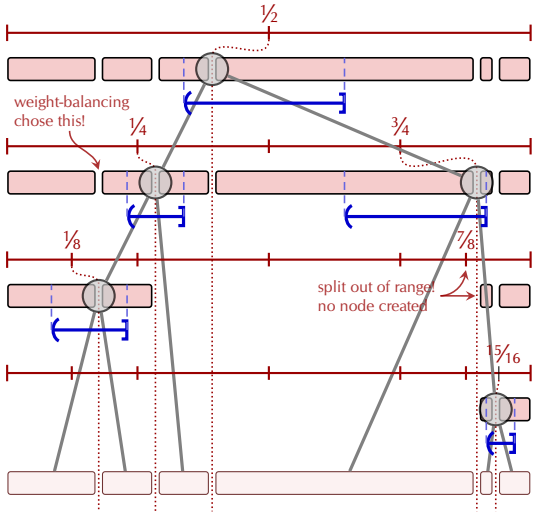
- inner node ⇝ midpoint interval
  ⟼ = normalized interval $[1..n] \mapsto [0, 1]$
- power = min ℓ s.t. ⟼ contains $c \cdot 2^{-\ell}$

⇝ depends only on 2 runs



Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

$\rightsquigarrow$ cannot use weight-balancing

Alternative view: *node powers*

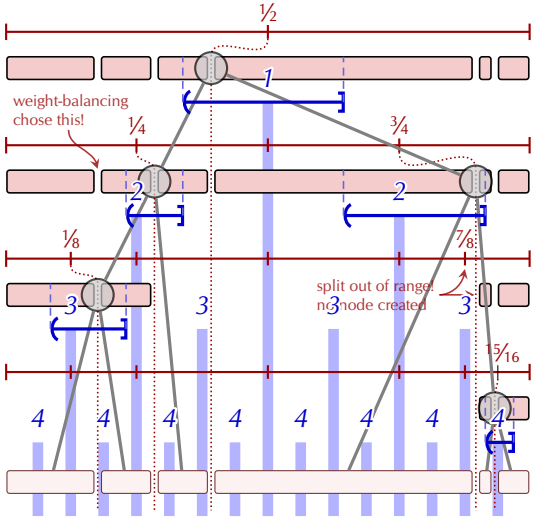
- inner node $\rightsquigarrow$ midpoint interval
  $\longleftrightarrow$ = normalized interval
  $[1..n] \mapsto [0, 1]$
- power = min ℓ s.t. $\longleftrightarrow$ contains $c \cdot 2^{-\ell}$

$\rightsquigarrow$ depends only on 2 runs



Method 2′: *bisection*

(Mehlhorn 1977)



The bisection heuristic

Timsort

- proceed left to right:
  - detect the next run
  - push it onto stack of runs
  - merge some runs from the stack

devil is in the details

⇝ cannot use weight-balancing

Alternative view: *node powers*

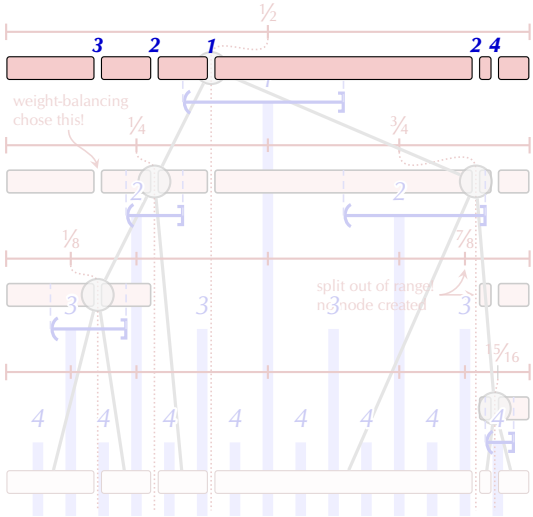
- inner node ⇝ midpoint interval
  ⟷ = normalized interval
  $[1..n] \mapsto [0,1]$
- power = min ℓ s.t. ⟷ contains $c \cdot 2^{-\ell}$



⇝ depends only on 2 runs

Method 2′: *bisection*

(Mehlhorn 1977)



Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

More good properties:

- power efficient to compute; (count leading zeros)
  $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic
    (highest on top)
  - stack height $\leq$ max power $\leq \lceil \lg n \rceil + 1$

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

More good properties:

- power efficient to compute; $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic (lowest on top)
  - stack height $\leq$ max power $\leq \lg n$ (+ 1)

Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

Powersort

Powersort

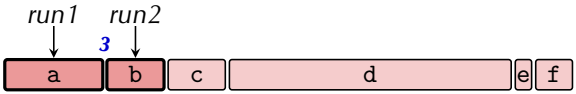
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

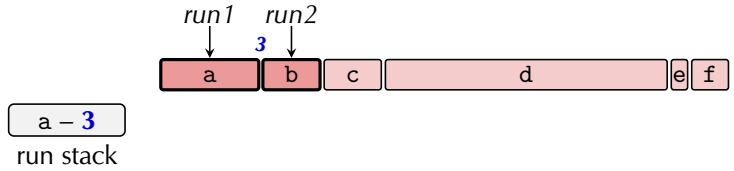
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

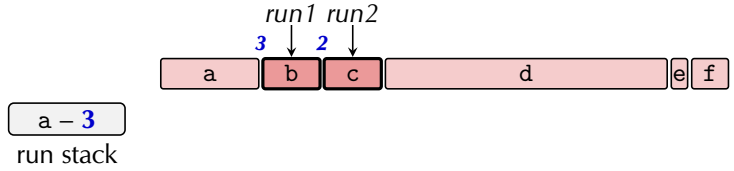
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

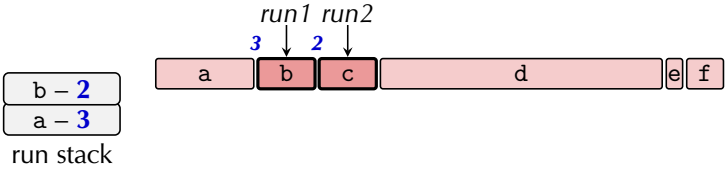
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



More good properties:

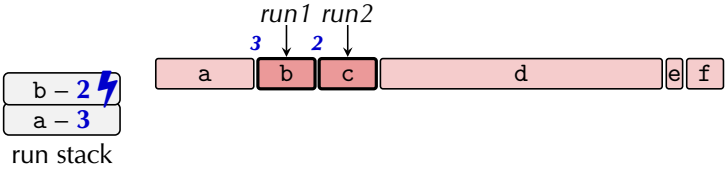
- power efficient to compute; O(1) with bitwise tricks (`clz`)
- never stores more than lg n runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leq$ max power $\leq$ $\lfloor \lg n \rfloor$ + 1

Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



More good properties:

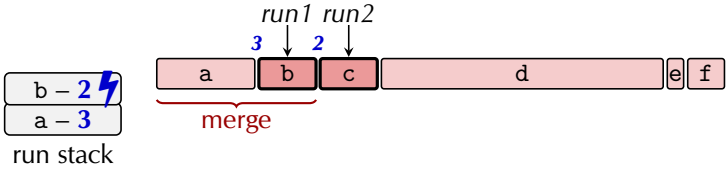
- power efficient to compute; O(1) with bitwise tricks (`clz`)
- never stores more than lg n runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leq$ max power $\leq$ $\lfloor \lg n \rfloor$ + 1

Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

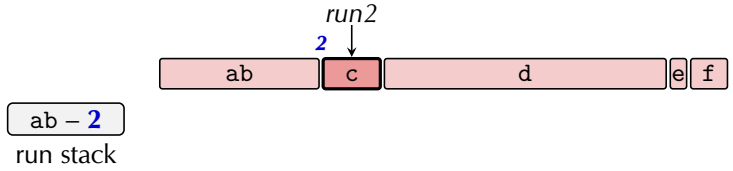
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



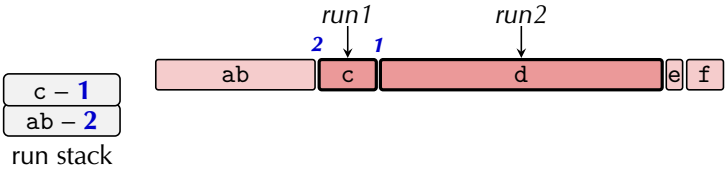
ab – **2**

run stack

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



More good properties:

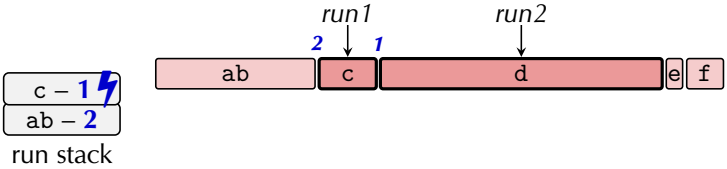
- power efficient to compute; O(1) with bitwise tricks (`clz`)
- never stores more than lg n runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leq$ max power $\leq$ lg n (lg n − 1)

Sebastian Wild | **Nearly-Optimal Mergesorts** | **2018-08-20** | **9 / 15**

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



More good properties:

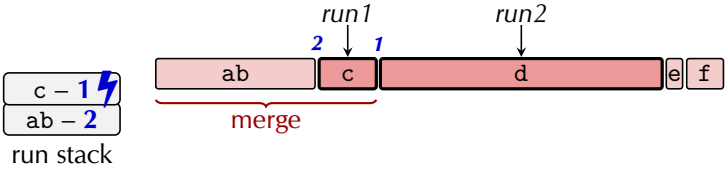
- power efficient to compute; O(1) with bitwise tricks (`clz`)
- never stores more than lg n runs:
  - powers on stack strictly increasing (highest on top)
  - stack height $\leq$ max power $\leq$ lg n (lg n + 1)

Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



More good properties:

- power efficient to compute; O(1) with bitwise tricks (`clz`)
- never stores more than lg n runs:
  - powers on stack strictly increasing (lowest on top)
  - stack height $\leq$ max power $\leq$ lg n ($\rightarrow$ 1)

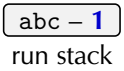
Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs





Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



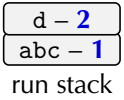
`abc` – **1**

run stack

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

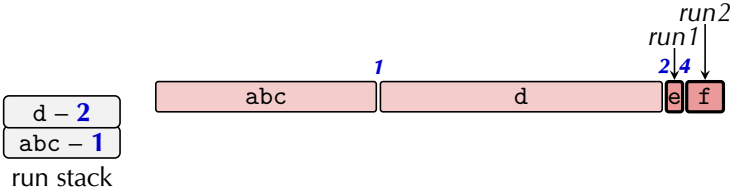




Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



More good properties:

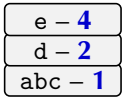
- power efficient to compute; O(1) with bitwise tricks (`clz`)
- never stores more than lg n runs:
  - powers on stack strictly monotonic (largest on top)
  - stack height $\leq$ max power $\leq$ $\lfloor \lg n \rfloor$ + 1

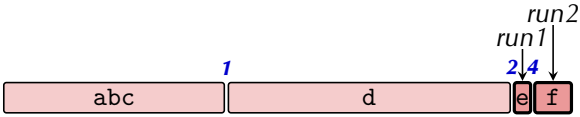
Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

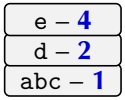




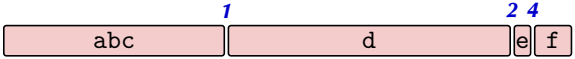
Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



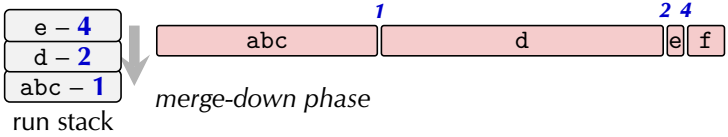
run stack



Powersort

Powersort

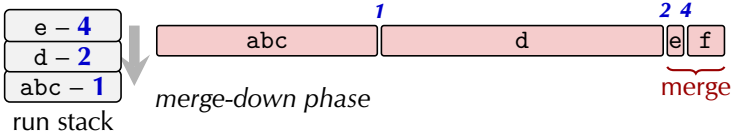
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

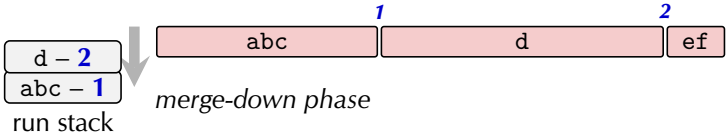
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

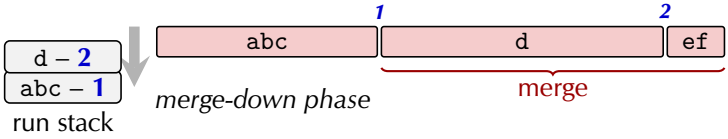
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

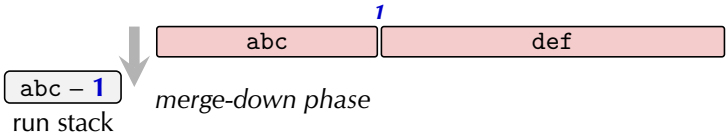
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

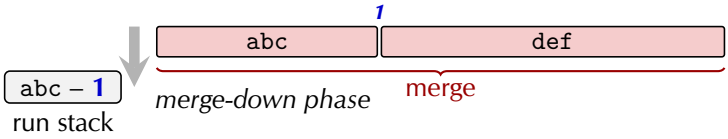
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

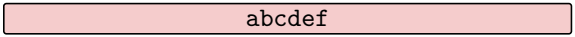
- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

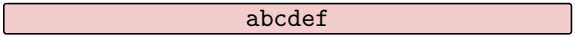


Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

abcdef



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

⇝ same merge/cmps cost as Peeksort

⇝ exploit runs **optimally** up to lower order terms!

- **but:** detects runs lazily!
  ⇝ no extra scan!

More good properties:

- power efficient to compute; $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - $\lg n$ is the number of bits of n (more than $\lg n$ runs)
  - $\lg n$ is the number of bits of n (more than $\lg n$ runs)

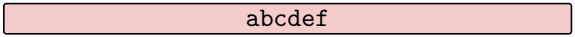
(clz: count leading zeros)

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

abcdef



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

⇝ same merge/cmps cost as Peeksort

⇝ exploit runs **optimally** up to lower order terms!

- **but:** detects runs lazily!
  ⇝ no extra scan!

More good properties:

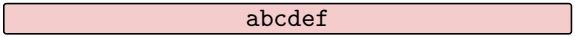
- power efficient to compute; $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - $\lg n$ is the maximum power of the bisection heuristic
  - $\lg n$ is the maximum power of the bisection heuristic

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

abcdef



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

⇝ same merge/cmps cost as Peeksort

⇝ exploit runs **optimally** up to lower order terms!

- **but:** detects runs lazily!
  ⇝ no extra scan!

More good properties:

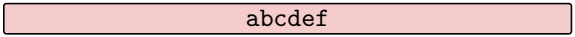
- power efficient to compute; $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - $\lg n$ is the most strictly increasing integer on $\mathbb{N}$
  - $\lg n$ is the most strictly increasing integer on $\mathbb{N}$
  - $\lg n$ is the most strictly increasing integer on $\mathbb{N}$

Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

- ⇝ same merge/cmps cost as Peeksort
- ⇝ exploit runs **optimally** up to lower order terms!
- **but:** detects runs lazily!
  - ⇝ no extra scan!

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

abcdef

Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

⇝ same merge/cmps cost as Peeksort

⇝ exploit runs **optimally** up to lower order terms!

- **but:** detects runs lazily!
  ⇝ no extra scan!



More good properties:

- power efficient to compute; $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - $\lg n$ runs are stored in $\mathcal{O}(n)$ time
  - $\lg n$ runs are stored in $\mathcal{O}(n)$ time

Sebastian Wild | Nearly-Optimal Mergesorts | 2018-08-20 | 9 / 15

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs

`abcdef`

Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

- ⇝ same merge/cmps cost as Peeksort
- ⇝ exploit runs **optimally** up to lower order terms!
- **but:** detects runs lazily!
  - ⇝ no extra scan!



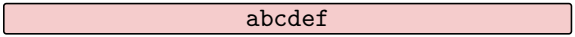
More good properties:

- power efficient to compute; count leading zeros
  $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leqslant$ max power $\leqslant \lfloor \lg n \rfloor + 1$

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

- ⇝ same merge/cmps cost as Peeksort
- ⇝ exploit runs **optimally** up to lower order terms!
- **but:** detects runs lazily!
  - ⇝ no extra scan!



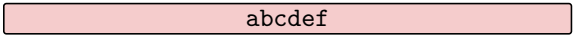
More good properties:

- power efficient to compute; $O(1)$ with bitwise tricks (`clz`) — count leading zeros
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leqslant$ max power $\leqslant \lfloor \lg n \rfloor + 1$

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

- ⇝ same merge/cmps cost as Peeksort
- ⇝ exploit runs **optimally** up to lower order terms!
- **but:** detects runs lazily!
  - ⇝ no extra scan!



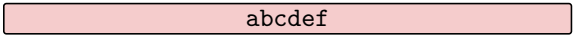
More good properties:

- power efficient to compute; count leading zeros
  $O(1)$ with bitwise tricks (`clz`)
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leqslant$ max power $\leqslant \lfloor \lg n \rfloor + 1$

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

- ⇝ same merge/cmps cost as Peeksort
- ⇝ exploit runs **optimally** up to lower order terms!
- **but:** detects runs lazily!
  - ⇝ no extra scan!



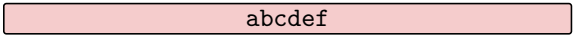
More good properties:

- power efficient to compute; $O(1)$ with bitwise tricks (`clz`) — count leading zeros
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leqslant$ max power $\leqslant \lfloor \lg n \rfloor + 1$

Powersort

Powersort

- proceed left to right:
  - detect next run & **compute power**
  - push run onto stack of runs
  - while new node **less powerful**: merge topmost runs



Theorem (Mehlhorn 1977:)

The bisection heuristic yields a BST with search cost $C \leqslant \mathcal{H}(\alpha_1, \ldots, \alpha_r) + 2$.

- ⇝ same merge/cmps cost as Peeksort
- ⇝ exploit runs **optimally** up to lower order terms!
- **but:** detects runs lazily!
  - ⇝ no extra scan!



More good properties:

- power efficient to compute; $O(1)$ with bitwise tricks (`clz`) — count leading zeros
- never stores more than $\lg n$ runs:
  - powers on stack strictly monotonic (highest on top)
  - stack height $\leqslant$ max power $\leqslant \lfloor \lg n \rfloor + 1$

Outline

1 Adaptive Sorting – Status Quo

2 Natural Mergesort

3 Peeksort

4 Powersort

5 Experiments

Experimental Evaluation



Experimental Evaluation



Hypotheses:

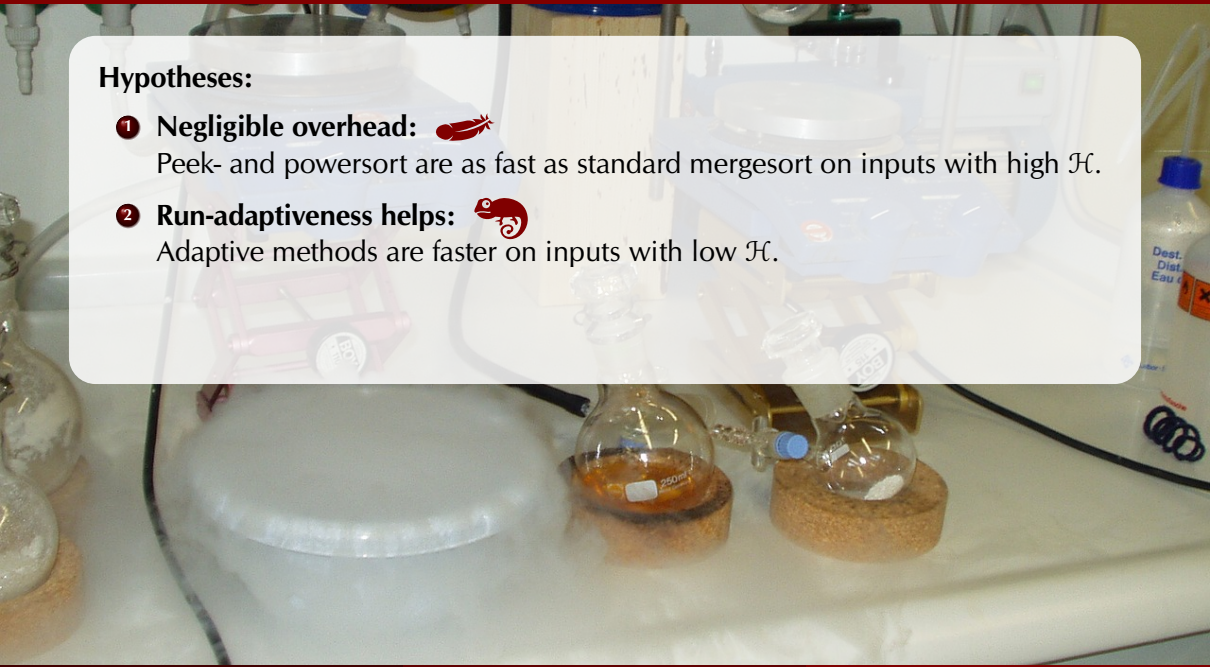
Experimental Evaluation



Hypotheses:

1. **Negligible overhead:**
   Peek- and powersort are as fast as standard mergesort on inputs with high $\mathcal{H}$.

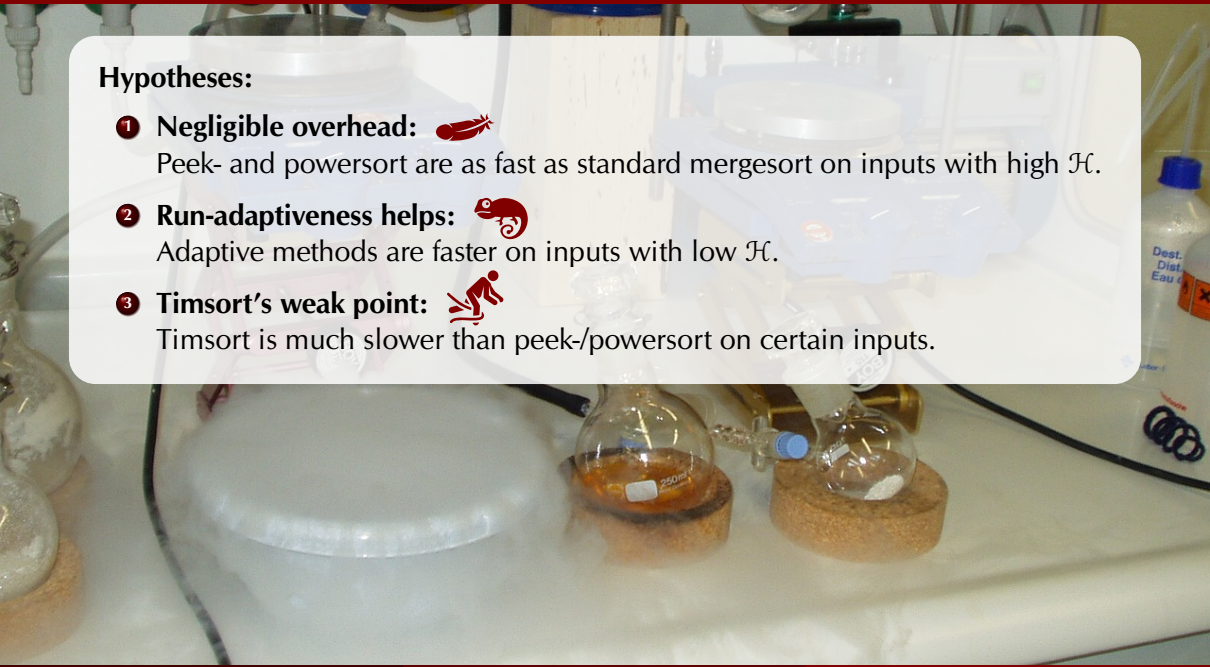
Experimental Evaluation



Hypotheses:

1. **Negligible overhead:**
   Peek- and powersort are as fast as standard mergesort on inputs with high $\mathcal{H}$.
2. **Run-adaptiveness helps:**
   Adaptive methods are faster on inputs with low $\mathcal{H}$.

Experimental Evaluation



Hypotheses:

1. **Negligible overhead:**
   Peek- and powersort are as fast as standard mergesort on inputs with high $\mathcal{H}$.
2. **Run-adaptiveness helps:**
   Adaptive methods are faster on inputs with low $\mathcal{H}$.
3. **Timsort's weak point:**
   Timsort is much slower than peek-/powersort on certain inputs.

Experimental Evaluation

Hypotheses:

1. **Negligible overhead:** 
   Peek- and powersort are as fast as standard mergesort on inputs with high $\mathcal{H}$.
2. **Run-adaptiveness helps:** 
   Adaptive methods are faster on inputs with low $\mathcal{H}$.
3. **Timsort's weak point:** 
   Timsort is much slower than peek-/powersort on certain inputs.

Setup:

- Java implementations, reproduced in C++
- mildly hand-tuned code
- sorting `int[]`s, length around 10^7

Negligible Overhead

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, Java runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

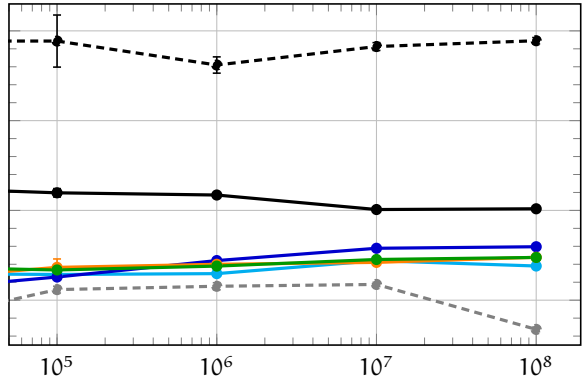
C++

Negligible Overhead

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, Java runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

C++

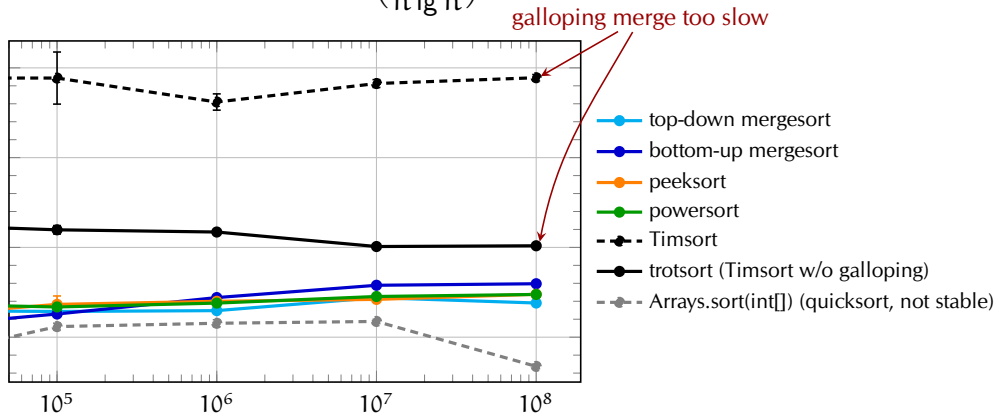


Negligible Overhead

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, Java runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

C++

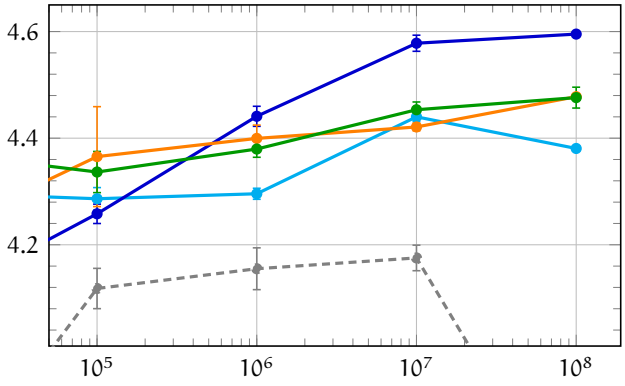


Negligible Overhead

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, Java runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

C++

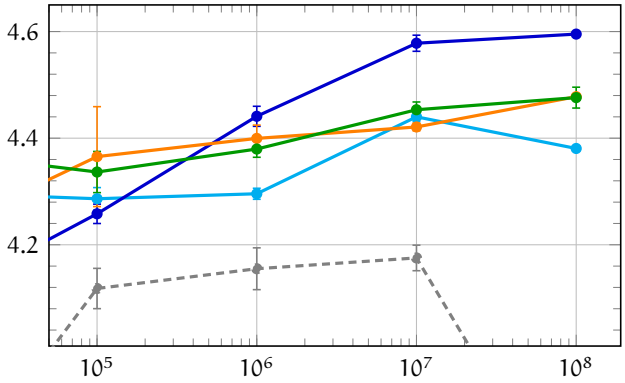


Negligible Overhead

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, Java runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

C++



$\rightsquigarrow$ **no significant difference** to standard mergesort

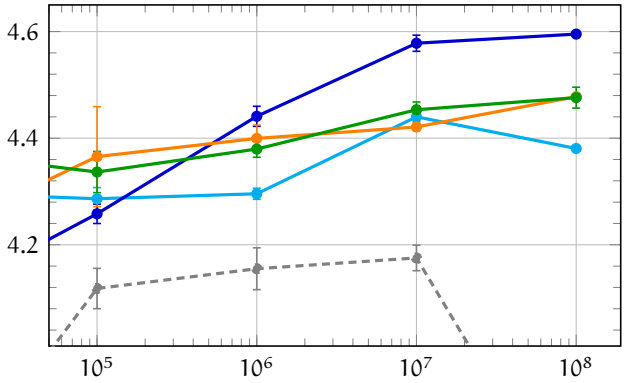
Negligible Overhead

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.



Study: random permutations, Java runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

C++



$\rightsquigarrow$ **no significant difference** to standard mergesort

Run-adaptiveness helps

❷ **Run-adaptiveness helps:** 
Adaptive methods are faster on inputs with low $\mathcal{H}$.

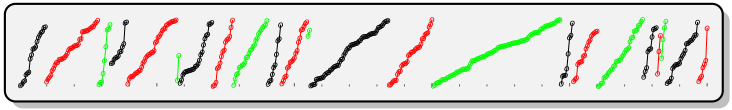
Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$ ▸ C++

$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness

Run-adaptiveness helps

❷ **Run-adaptiveness helps:** 
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$ ▸ C++

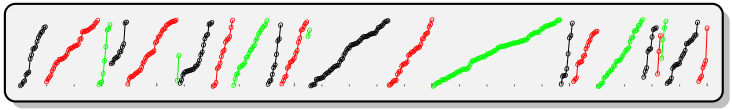


$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness

Run-adaptiveness helps

❷ **Run-adaptiveness helps:** 
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$ ▸ C++

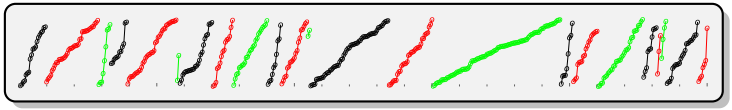


$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness

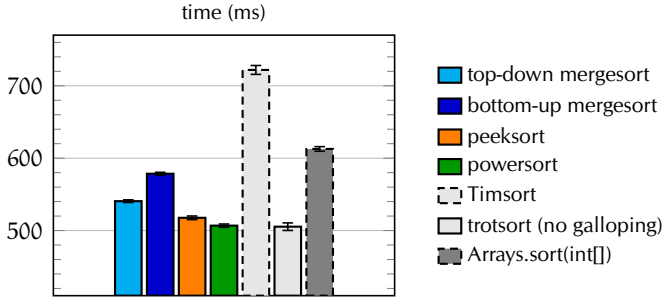
Run-adaptiveness helps

❷ **Run-adaptiveness helps:**
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$ ▸ C++



$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness

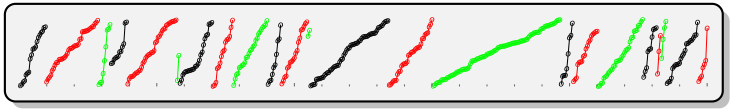


▸ merge cost

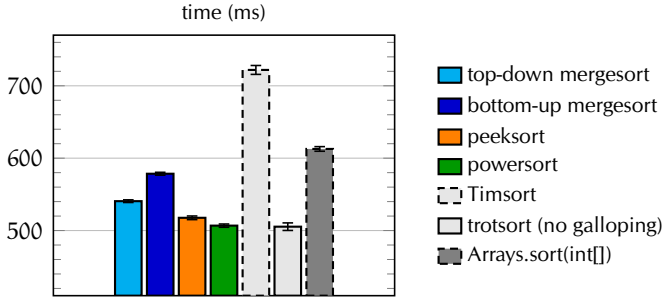
Run-adaptiveness helps

❷ **Run-adaptiveness helps:**
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$ ▸ C++



$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness



▸ merge cost

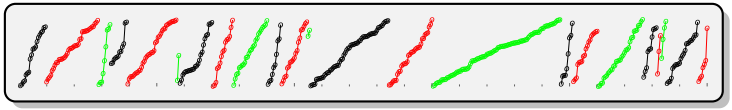
$\rightsquigarrow$ **beat quicksort** by 20%

Run-adaptiveness helps

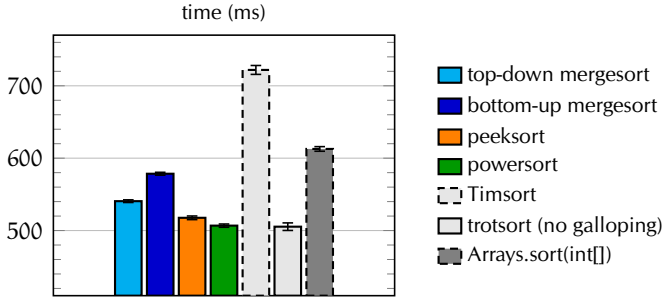
❷ **Run-adaptiveness helps:**
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$

▸ C++



$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness



▸ merge cost

$\rightsquigarrow$ **beat quicksort** by 20%

Timsort's weak point

❸ **Timsort's weak point:**

Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$ ▸ C++
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018

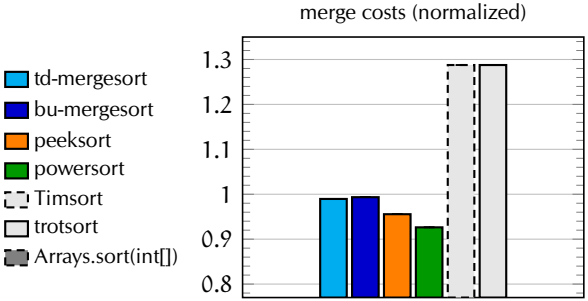
Timsort's weak point

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \ldots, L_r$
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

▸ C++

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018



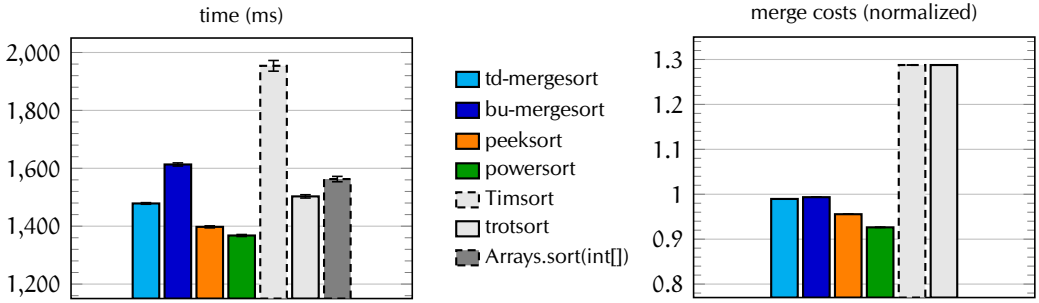
Timsort's weak point

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018

▸ C++



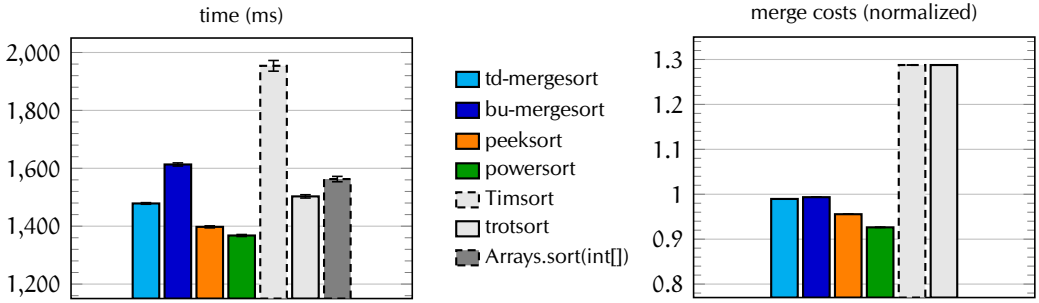
Timsort's weak point

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018

▸ C++



⇝ Timsort/trotsort has
- **40% higher merge cost**
- 10% higher running time in Java

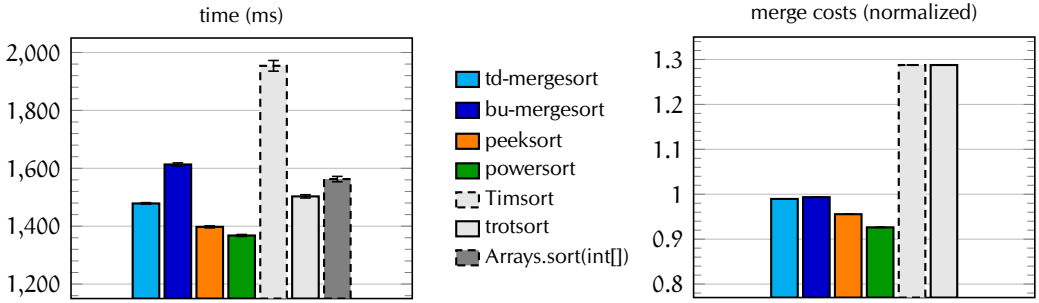
Timsort's weak point

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018

▸ C++



⇝ Timsort/trotsort has { **40% higher merge cost**; 10% higher running time in Java

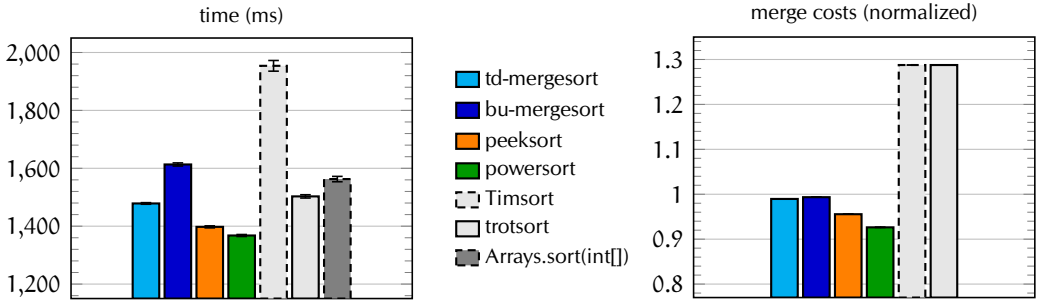
Timsort's weak point

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018

▸ C++



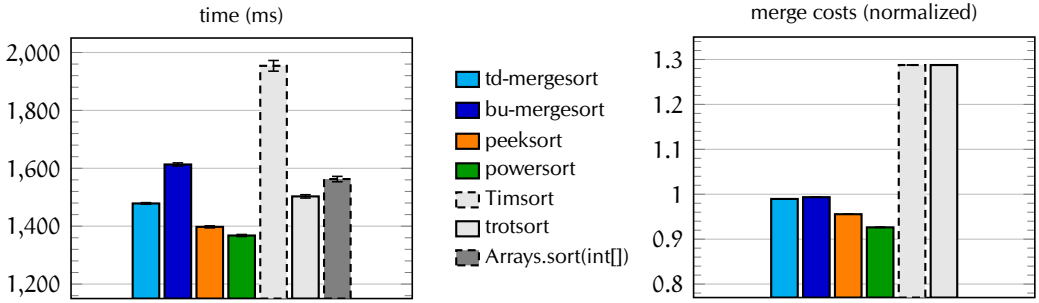
⇝ Timsort/trotsort has
- **40% higher merge cost**
- 10% higher running time in Java
- **40% higher running time** in C++

Timsort's weak point

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$ ▸ C++
Java runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018



⇝ Timsort/trotsort has
- **40% higher merge cost**
- 10% higher running time in Java
- **40% higher running time** in C++

Conclusion

We have seen

- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor



What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen

- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor



What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen



- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor

What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen



- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor

What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen



- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor

What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen



- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor

What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen



- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor

What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?

Conclusion

We have seen



- optimal run-**adaptivity** by conceptually **simple** methods
- with **negligible** overhead in running time
- correctness and performance bounds easy to prove, including constant factor

What's next?

- improvements **during** merge?
- nearly-optimal **multiway** merge?
- **preprocessing** by patience sort to get longer runs?
- practical method that adapts to **duplicates and runs**?



Icons made by *Freepik* and *Gregor Cresnar* from `www.flaticon.com`.

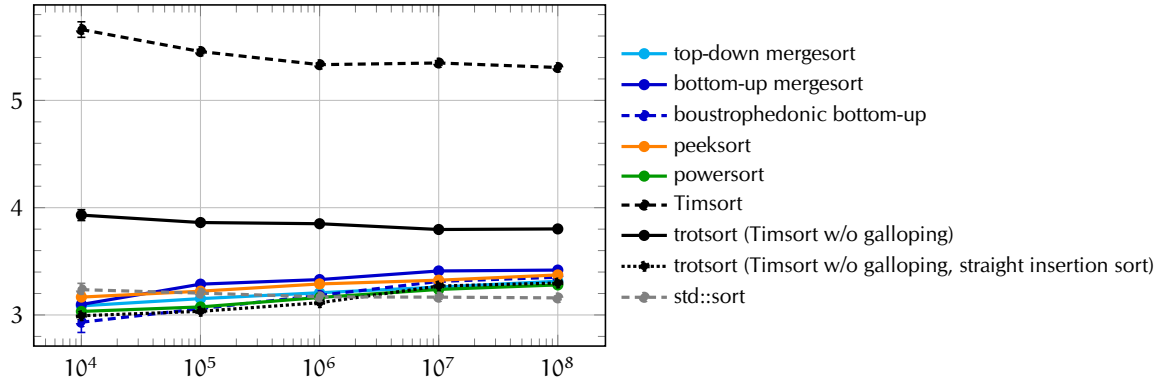
Squid photo by *LuqueStock* / *Freepik*

Appendix: Negligible Overhead – C++

1. **Negligible overhead:**
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, **C++** runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

Java

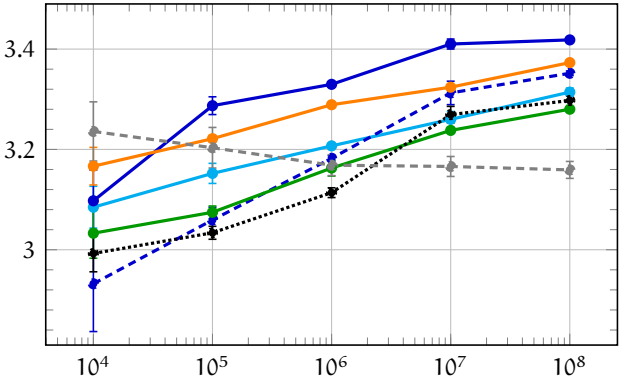


Appendix: Negligible Overhead – C++

1. **Negligible overhead:** 
   Peek- and powersort are as good as standard mergesort on inputs with high $\mathcal{H}$.

Study: random permutations, **C++** runtimes $\left(\frac{\text{time}}{n \lg n}\right)$

Java

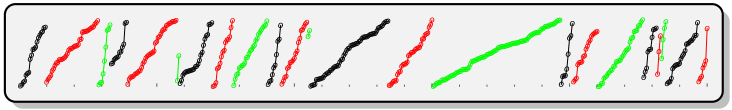


Appendix: Run-adaptiveness helps – merge cost

❷ **Run-adaptiveness helps:**
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, Java runtimes, $n = 10^7$

▸ C++



$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness



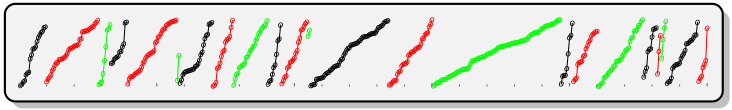
$\rightsquigarrow$ **beat quicksort** by 20%

▸ back

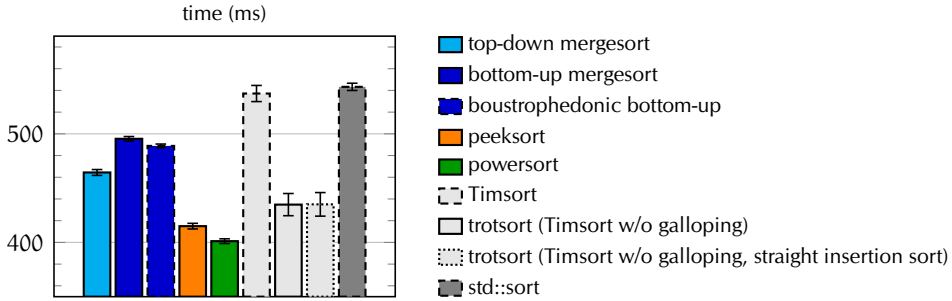
Appendix: Run-adaptiveness helps – C++

❷ **Run-adaptiveness helps:**
Adaptive methods are faster on inputs with low $\mathcal{H}$.

Study: "random runs": rp w/ ranges of Geo($1/\sqrt{n}$) sorted, **C++** runtimes, $n = 10^7$ ▸ Java



$\rightsquigarrow$ $\approx \sqrt{n}$ runs, avg length $\sqrt{n}$
10% of runs $< 0.1\sqrt{n}$
5% of runs $> 5\sqrt{n}$
moderate presortedness

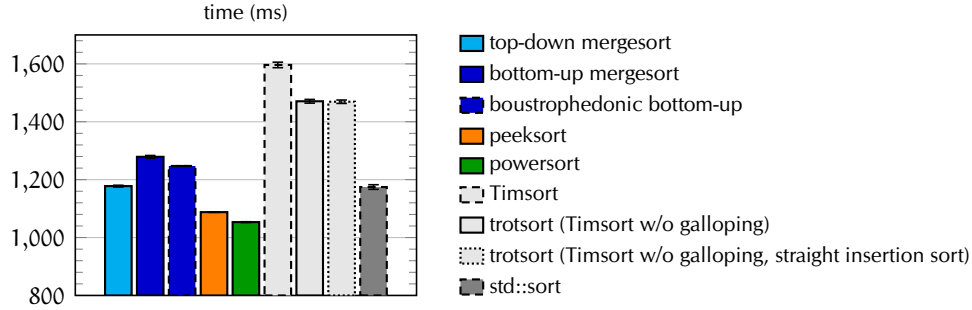


Appendix: Timsort's weak point – C++

❸ **Timsort's weak point:**
Timsort is much slower than peek-/powersort on certain inputs.

Study: "Timsort-drags": known family of bad-case sequences $L_1, \dots, L_r$ ▸ Java
C++ runtimes, $n = 2^{24} \approx 1.6 \cdot 10^7$

$\mathcal{R}_{\text{tim}}(n)$ by Buss & Knop 2018



Timsort

How does Timsort work?

Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While (A)/(B)/(C)/(D) applicable: Merge corresponding runs

Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs

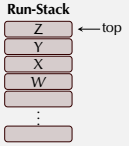


Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs

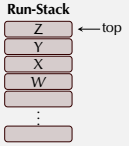


Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While (A)/(B)/(C)/(D) applicable: Merge corresponding runs

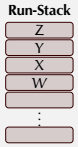


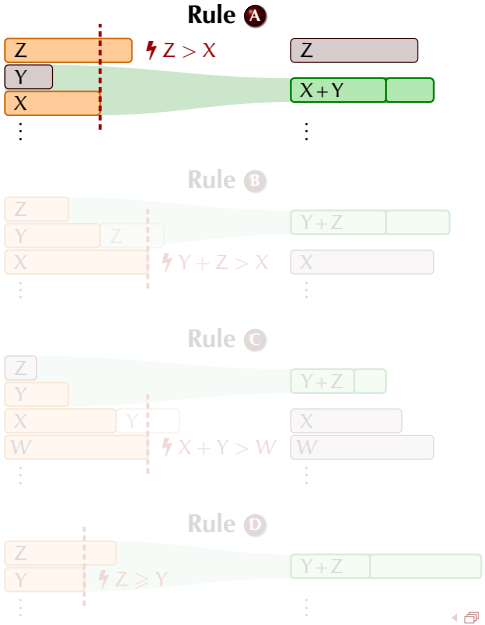
Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs



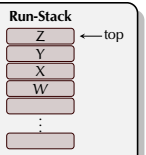


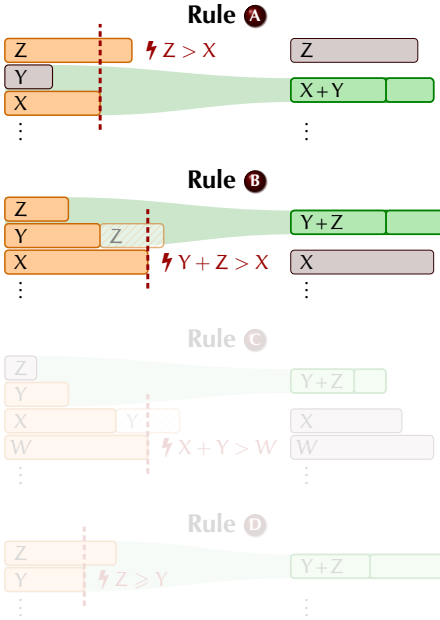
Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs



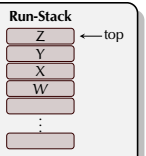


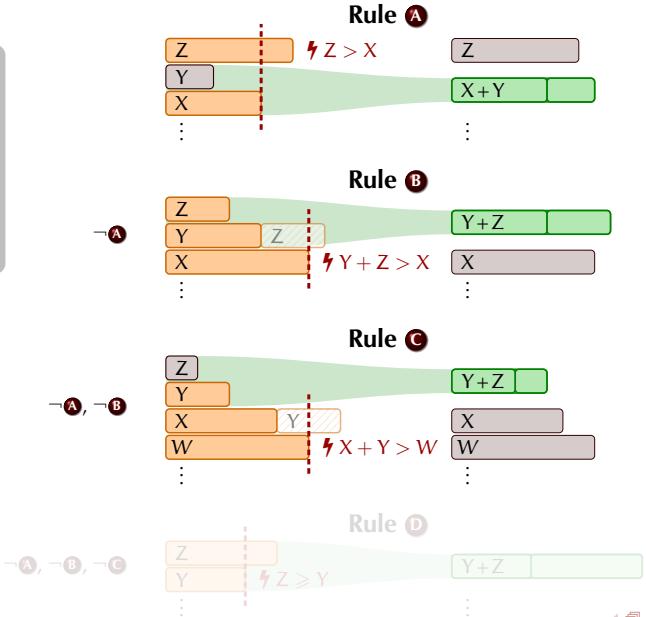
Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs





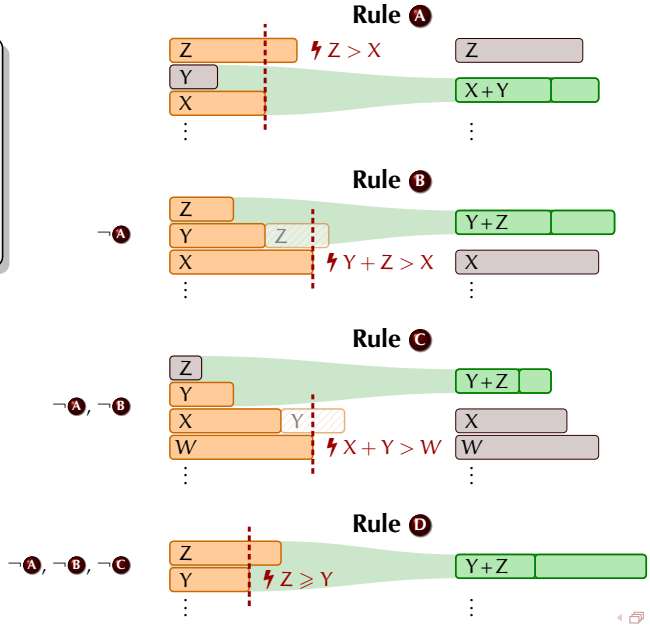
Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs





Timsort

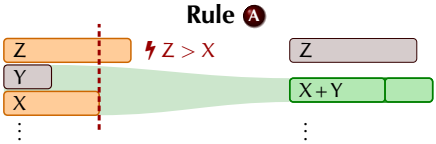
How does Timsort work?

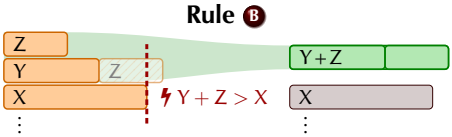
Timsort

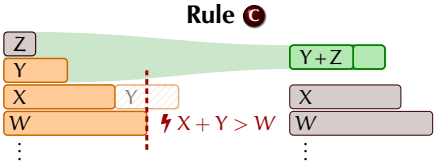
- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs

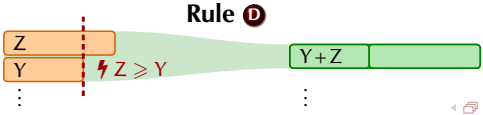


- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci
  - $\rightsquigarrow$ stack height $\approx \log_\varphi(n)$









Timsort

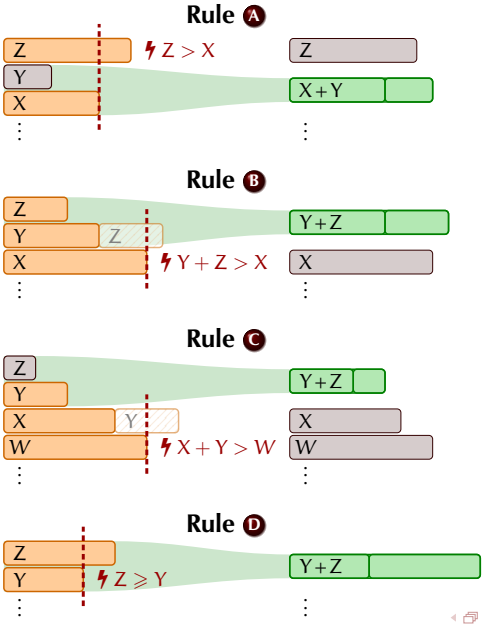
How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci
  - $\rightsquigarrow$ stack height $\approx \log_\varphi(n)$

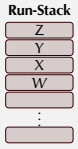


Timsort

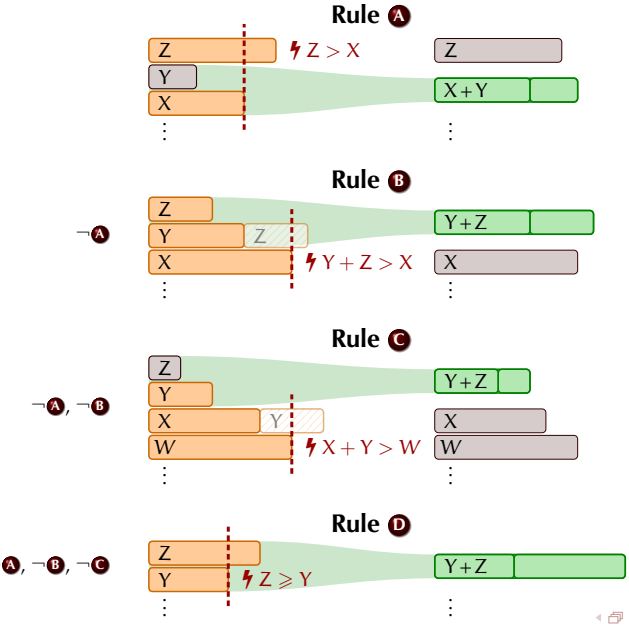
How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While Ⓐ/Ⓑ/Ⓒ/Ⓓ applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci
  - $\rightsquigarrow$ stack height $\approx \log_\varphi(n)$



Timsort

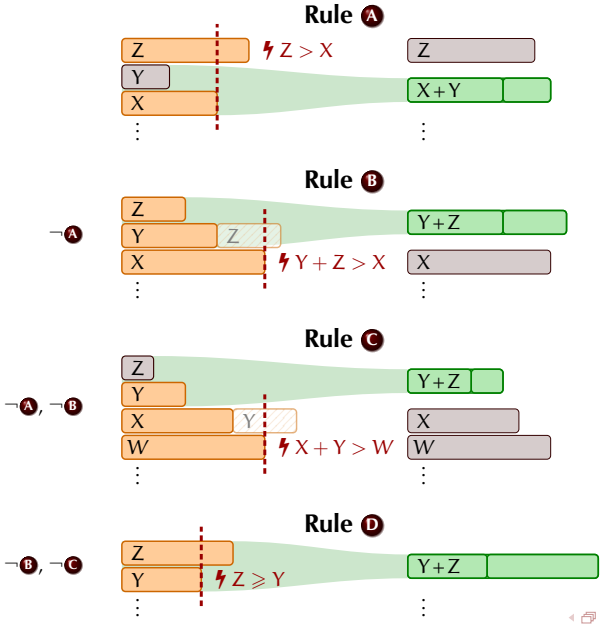
How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci
  - $\rightsquigarrow$ stack height $\approx \log_\phi(n)$

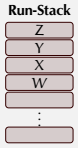


Timsort

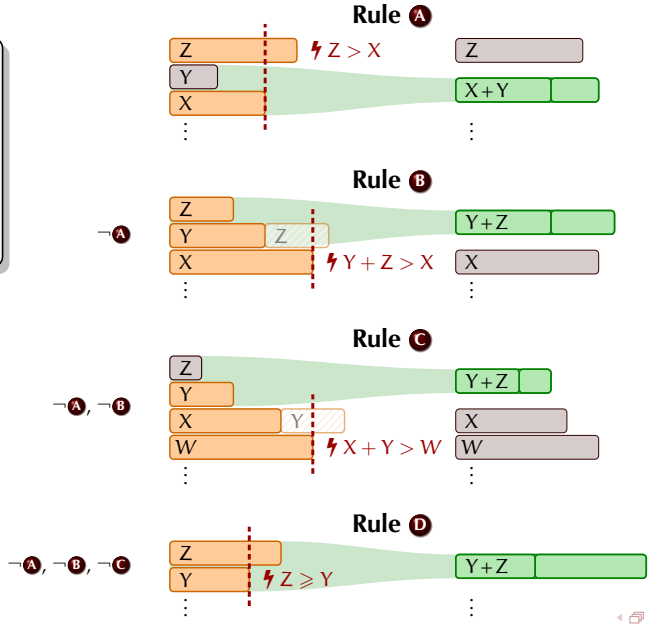
How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While Ⓐ/Ⓑ/Ⓒ/Ⓓ applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque rules!**
  - Goal: run lengths $\geqslant$ Fibonacci
  - ⇝ stack height $\approx \log_\phi(n)$

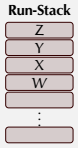


Timsort

How does Timsort work?

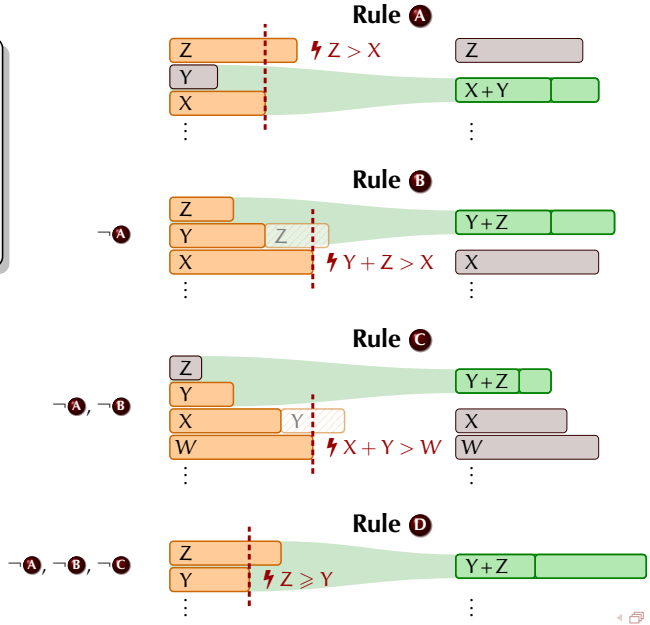
Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While Ⓐ/Ⓑ/Ⓒ/Ⓓ applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque rules!**
  - Goal: run lengths $\geqslant$ Fibonacci

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



Timsort

How does Timsort work?

Timsort

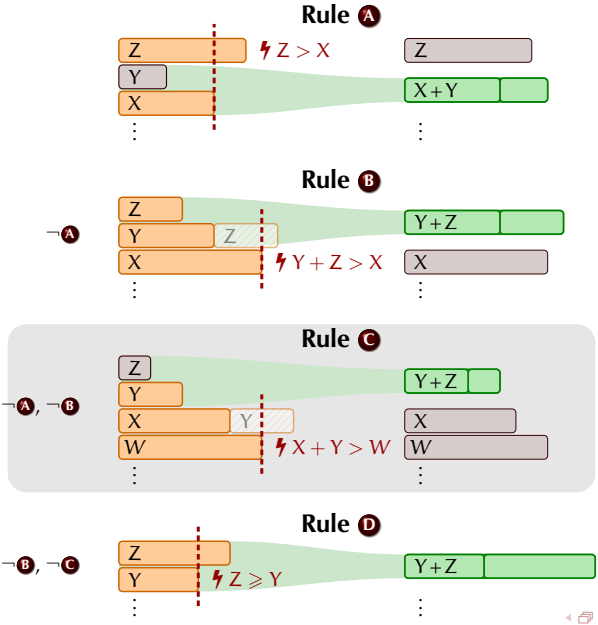
- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While Ⓐ/Ⓑ/Ⓒ/Ⓓ applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci

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

2015 ↯ invariant does **not** hold w/o rule C!

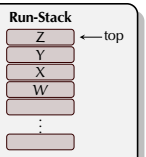


Timsort

How does Timsort work?

Timsort

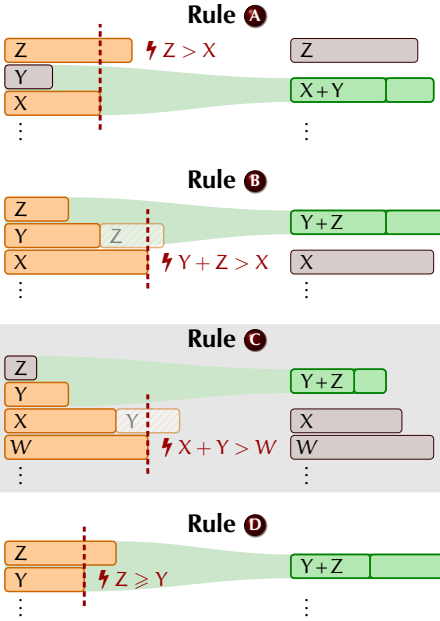
- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci

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

2015 ↯ invariant does **not** hold w/o rule C!
- *CPython*: ✓



Timsort

How does Timsort work?

Timsort

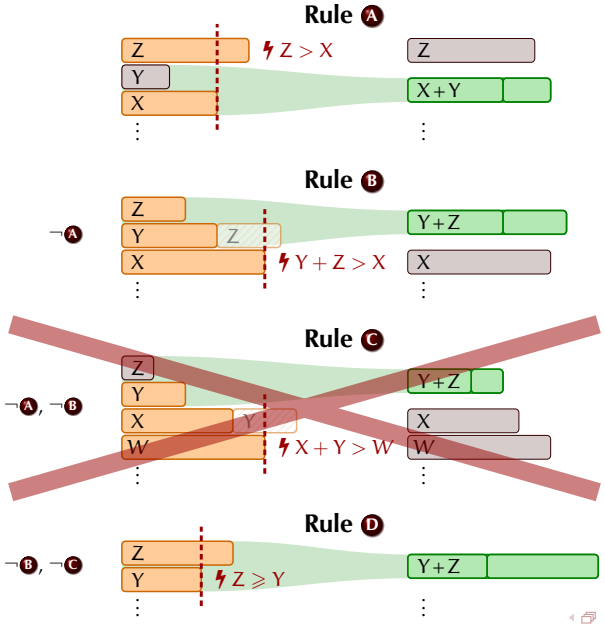
- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While A/B/C/D applicable: Merge corresponding runs



- **Advantages:**
  - profits from existing runs
  - *locality of reference* for merges
- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci

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

2015 ↯ invariant does **not** hold w/o rule C!
- *CPython*: ✓
- *OpenJDK*: rather increase stack …

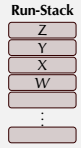


Timsort

How does Timsort work?

Timsort

- Proceed left to right:
  - Find next Run
  - Push Run onto stack
  - While Ⓐ/Ⓑ/Ⓒ/Ⓓ applicable:





Stack still too small!

(Auger et al. **ESA 2018**)

- **But: opaque** rules!
  - Goal: run lengths $\geqslant$ Fibonacci
  - $\rightsquigarrow$ stack height $\approx \log_\phi(n)$

2015 ϟ invariant does **not** hold w/o rule C!

- *CPython*: ✓
- *OpenJDK*: rather increase stack …

