Quicksort, Timsort, Powersort

Algorithmic ideas, engineering tricks, and trivia behind CPython's new sorting algorithm

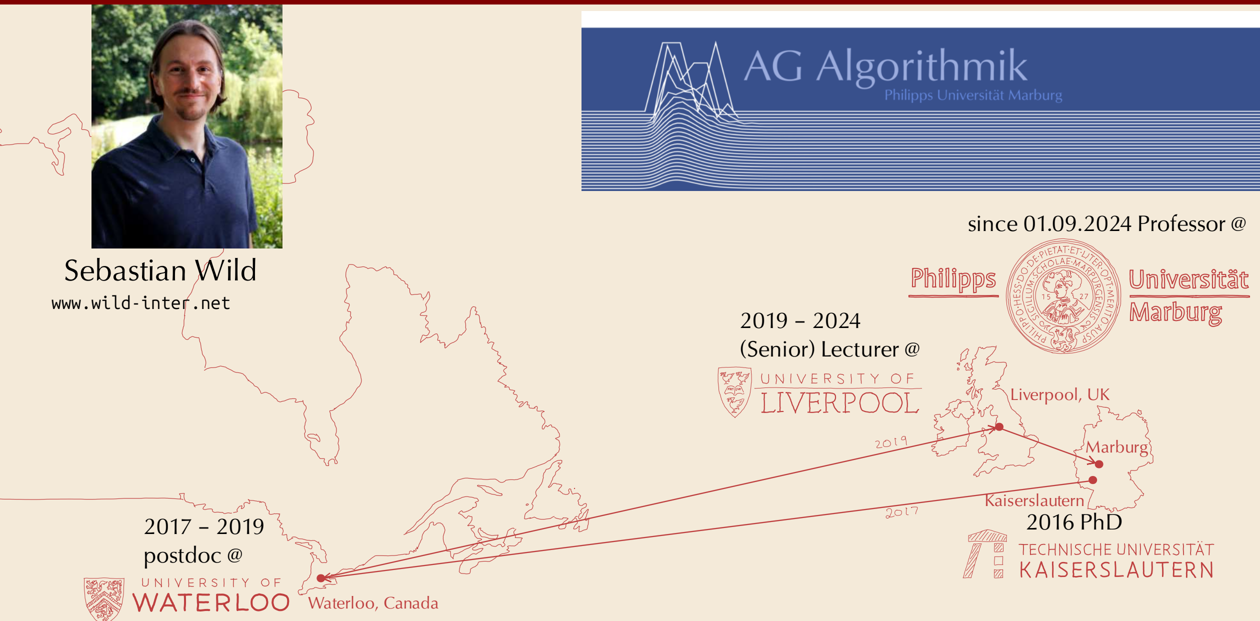
Sebastian Wild

www.wild-inter.net

Philipps Universität Marburg

Theorietag 2025
Friedrich-Schiller-Universität Jena

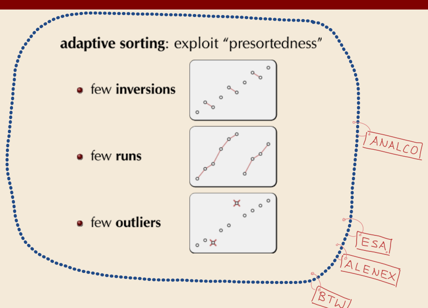
There and back again



Research Interests: (1) Beyond worst-case algorithms

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

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



lazy search trees

PQ

BST

only min accessible

fully sorted

FOCS

external memory

dynamize

cache-oblivious multiple selection

k-partitioner

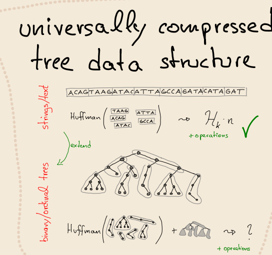
SWAT

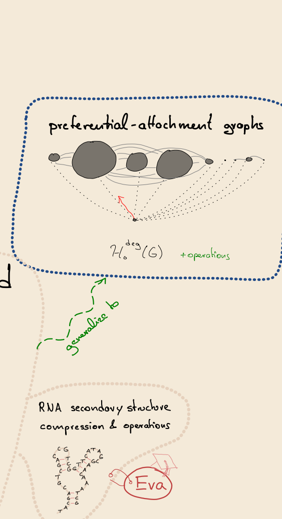
ESA

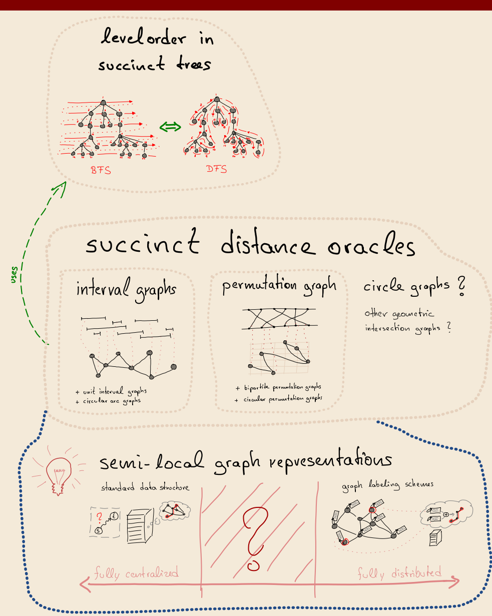
Research Interests: (2) Space-efficient Data Structures

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

My Focus:
Graph-structured data



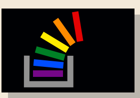




Outline

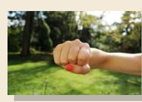
1 Sort of a list

2 Timsort



3 Beware, Stackoverflow!

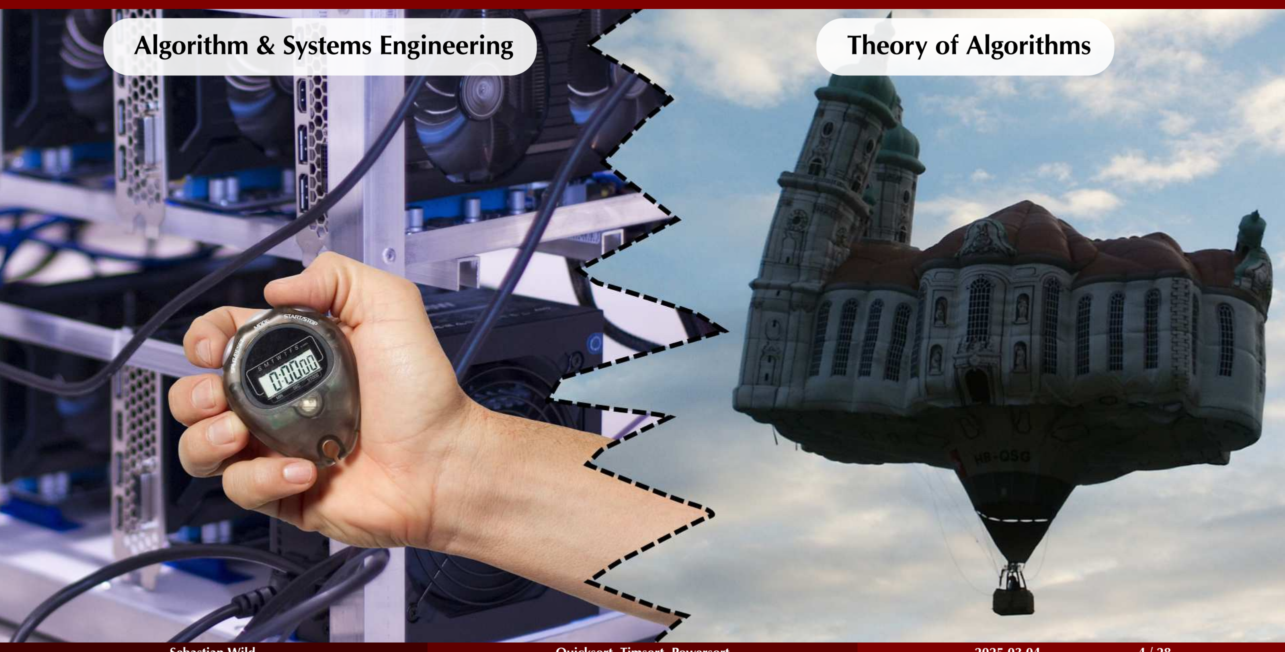
4 Merge policies



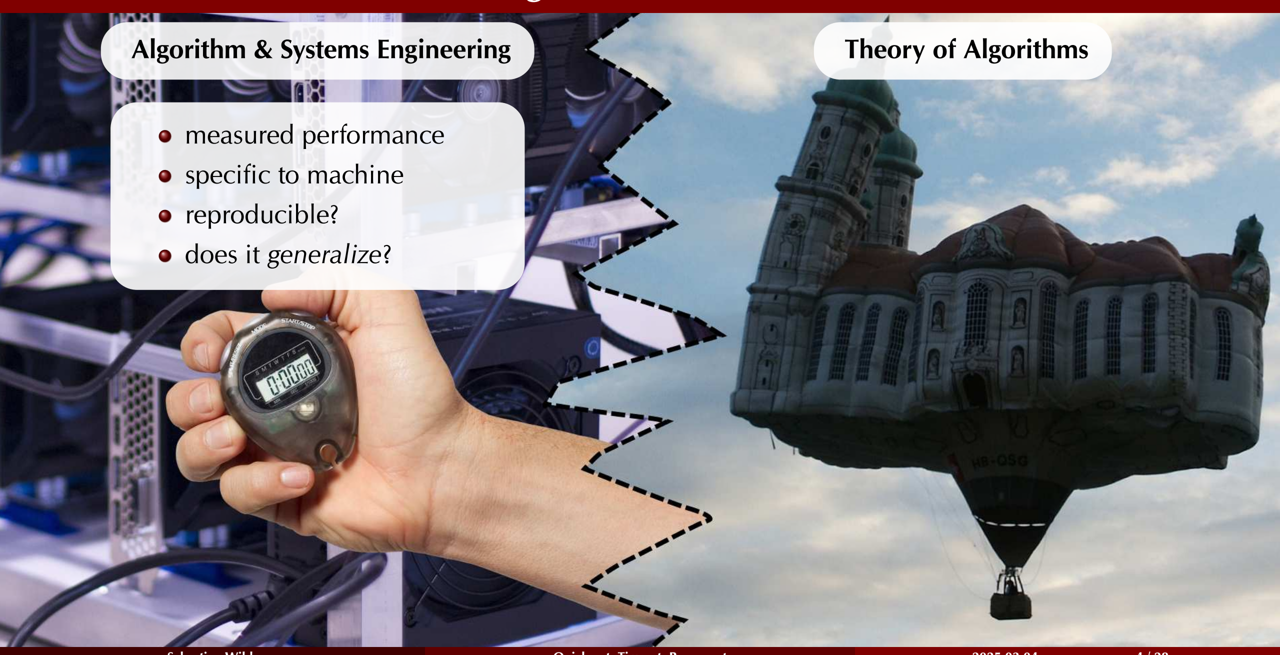
5 Powersort

1
Sort of a list

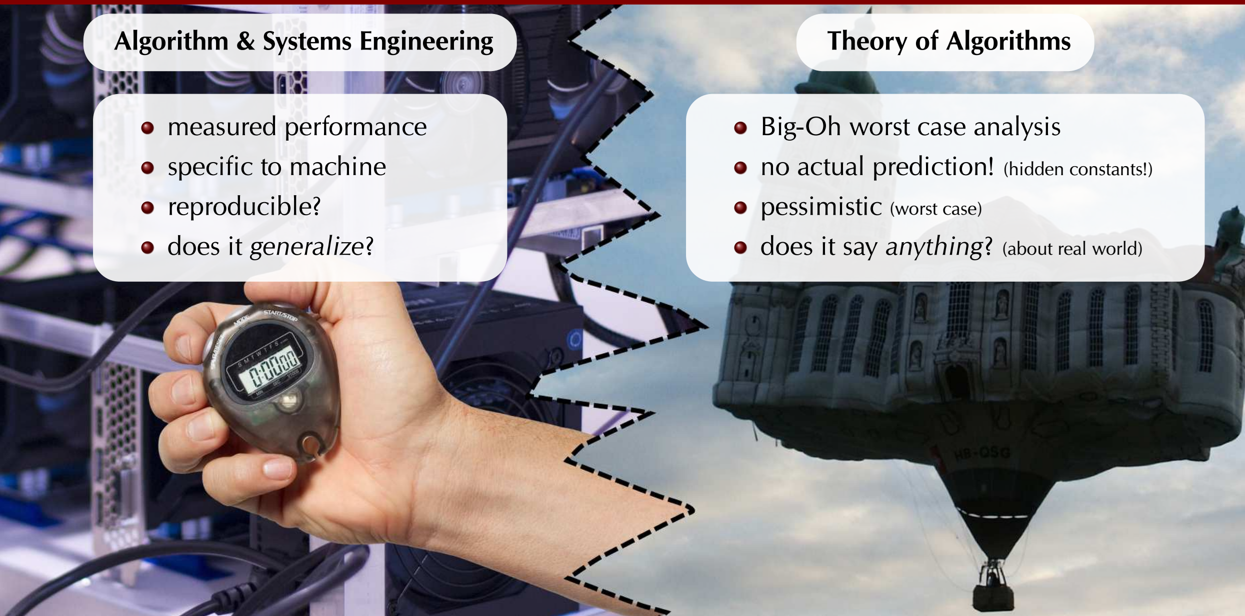
Algorithm Science



Algorithm Science



Algorithm Science



Algorithm Science

Algorithm & Systems Engineering

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

Theory of Algorithms

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

Algorithm Science

Algorithm Science

Algorithm & Systems Engineering

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

Theory of Algorithms

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

1 ***Abstract Cost Models***

precise asymptotic analysis
$\leadsto$ quantitative predictions!

2 ***Adaptive Analysis***

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

Algorithm Science

Algorithm Science

Algorithm & Systems Engineering

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

Theory of Algorithms

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

1 *Abstract Cost Models*

precise asymptotic analysis

$\rightsquigarrow$ quantitative predictions!

2 *Adaptive Analysis*

fine-grained input features

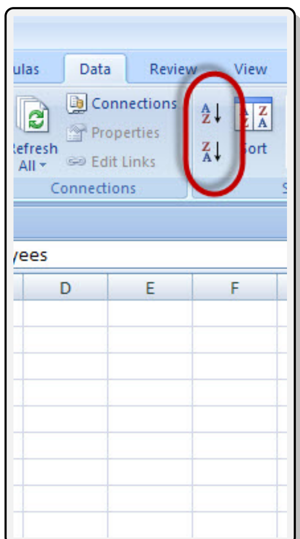
$\rightsquigarrow$ $\neq$ worst case

Platform-independent performance guarantees

- concrete quantitative model
- validated by experiments

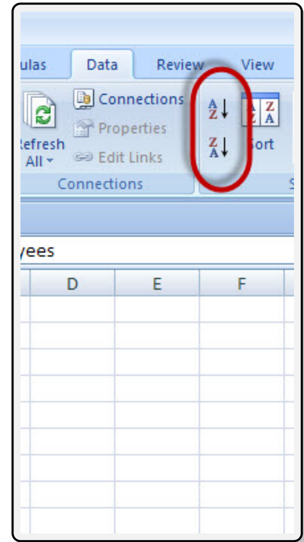
Sorting

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



Sorting

- We use sorting to
  - to make searching faster (binary search!)
  - clean data (remove dups, get canonical form of things)
  - to present data neatly for users
  - as building block in algorithms (database join, sweepline, ...)
  - ...
- built-in functions in Python: `my_list.sort()` and `sorted(my_list)`
  - `key` parameter to specify sorting criterion
  - prefer pairwise comparator function? $\leadsto$ `functools.cmp_to_key`



Stable Sorting

unsorted input

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

Stable Sorting

unsorted input

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

sorted by First Name

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

Stable Sorting

unsorted input

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

sorted by First Name

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

sorted by Last Name

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

CPython Sorting History

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

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

CPython Sorting History

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

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

CPython Sorting History

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

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

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

CPython Sorting History

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

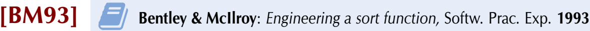
[BM93] **Bentley & McIlroy**: *Engineering a sort function*, Softw. Prac. Exp. **1993**

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

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

CPython Sorting History

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


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

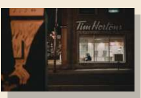
[FM70] **Frazer & McKellar**: *Samplesort: A Sampling Approach to Minimal Storage Tree Sorting*, J. ACM **1970**

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

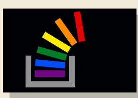
[MW18] **Munro & Wild**: *Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs*, ESA **2018**

Outline

1 Sort of a list

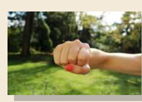


2 Timsort



3 Beware, Stackoverflow!

4 Merge policies



5 Powersort

2 Timsort
Tim Hortons
NO STANDING

Sorting Trade-offs

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

Sorting Trade-offs

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

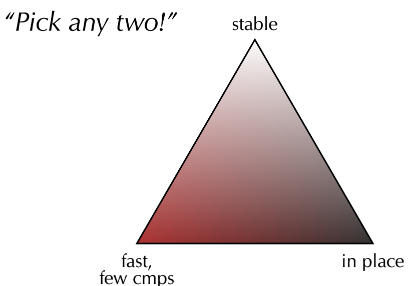
Sorting Trade-offs

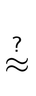
- *Quicksort* is generally **fast** and **in place**, but not stable
- *Mergesort* is generally (quite) **fast** and **stable**, but not in place
- stable and in place is tricky (possible, but relatively slow)

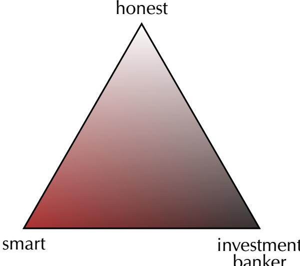
Sorting Trade-offs

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

"Pick any two!"

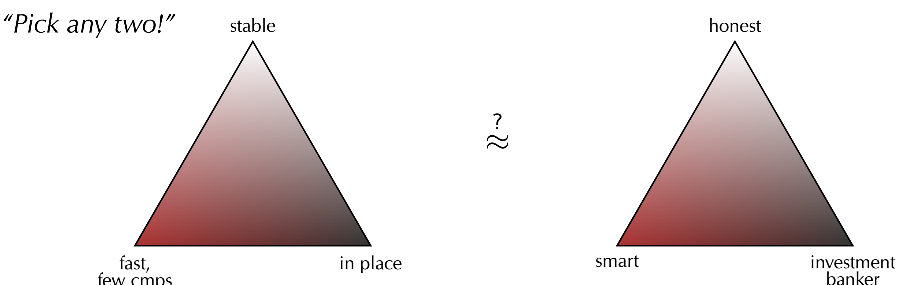






Sorting Trade-offs

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

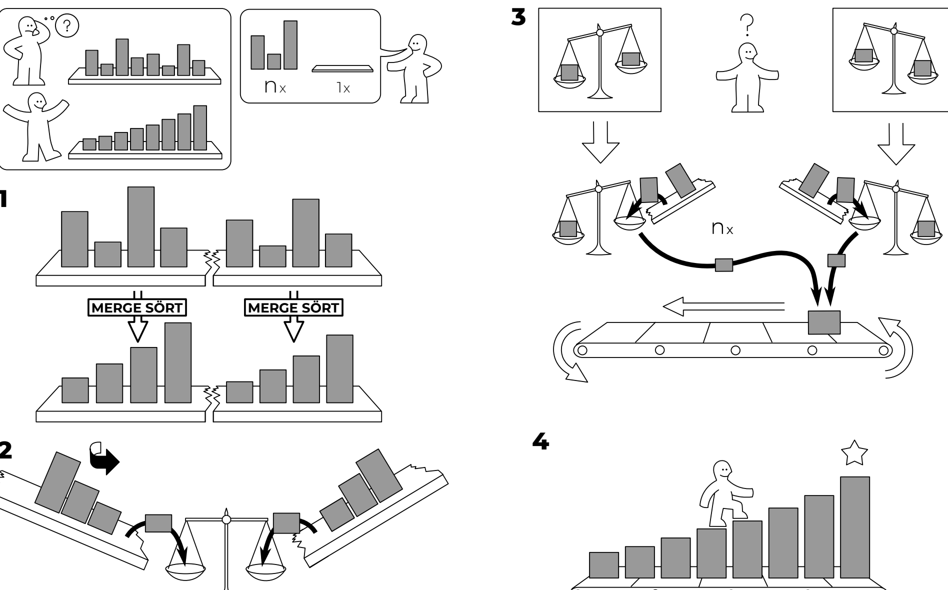


- Python has lots of objects and pointers anyways … “in-place” property least painful to sacrifice

⇝ Mergesort!

MERGE SÖRT

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



Timsort = Mergesort++

Various modifications

1. detect existing **runs**
   run = any (weakly) increasing or
   (strictly) decreasing range

2. **merge policy** decides when to merge which runs

3. keeping runs on a **fixed-size** runstack

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

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

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

Timsort merge policy (original)

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

full code: `tiny.cc/timsort`

Timsort merge policy (original)

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

full code:
`tiny.cc/timsort`

runs

Z ⟵ top
Y
X
⋮

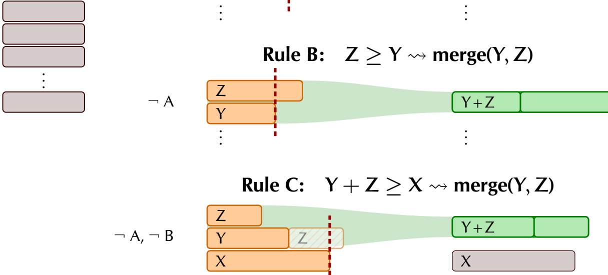
- `extend_run` detects next run
  & boosts to `minrun` if necessary

Timsort merge policy (original)

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

full code: `tiny.cc/timsort`

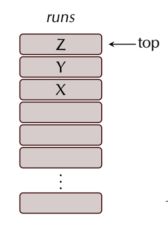
- `extend_run` detects next run & boosts to `minrun` if necessary

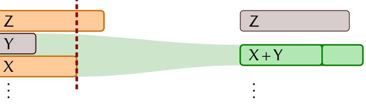


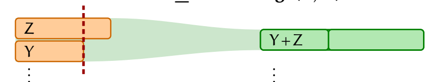
Timsort merge policy (original)

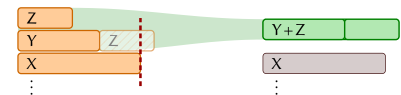
```
1 def timsort(lst):
2     i = 0; runs = []
3     while i < len(lst):
4         j = extend_run(lst, i)
5         runs.append((i,j-i)); i = j
6         while Rule A/B/C applicable
7             merge X,Y resp. Y,Z
8     while len(runs) > 1:
9         merge Y,Z
```

full code: `tiny.cc/timsort`







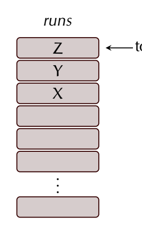


- `extend_run` detects next run & boosts to `minrun` if necessary
- **Goal:** runs on stack grow like Fibonacci numbers
  - Invariant: $\forall j :$ `runs`$[j] \geqslant$ `runs`$[j+1] +$ `runs`$[j+2]$
  - $\rightsquigarrow$ max stack height $\approx \log_\phi(n)$

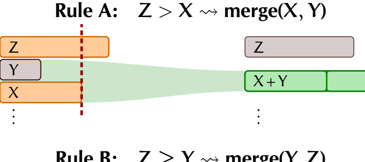
Timsort merge policy (original)

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

full code:
`tiny.cc/timsort`



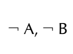
Rule A: $Z > X \rightsquigarrow$ **merge(X, Y)**

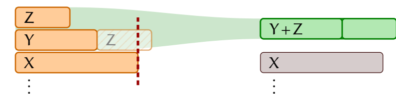


Rule B: $Z \geq Y \rightsquigarrow$ **merge(Y, Z)**

$\neg$ A

Rule C: $Y + Z \geq X \rightsquigarrow$ **merge(Y, Z)**



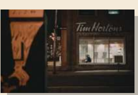


- `extend_run` detects next run & boosts to `minrun` if necessary
- **Goal:** runs on stack grow like Fibonacci numbers
  - Invariant: $\forall j$: `runs[j]` $\geqslant$ `runs[j+1]` $+$ `runs[j+2]`
  - $\rightsquigarrow$ max stack height $\approx \log_\phi(n)$
- Why exactly these rules?
  - Tim Peters: *"first thing I tried that 'worked well'"*

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

Outline

1 Sort of a list



2 Timsort

3 **Beware, Stackoverflow!**

4 Merge policies

5 Powersort

3 Beware, Stackoverflow!

Invariant trouble

Recall this?

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

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

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Invariant trouble

Recall this?

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

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

not true(!) for naughty pattern of run lengths

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Invariant trouble

Recall this?

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

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

not true(!) for naughty pattern of run lengths

KeY project for formal verification in Java

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

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

Invariant trouble

Recall this?

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

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

⚡ not true(!) for naughty pattern of run lengths

KeY project for formal verification in Java

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

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

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

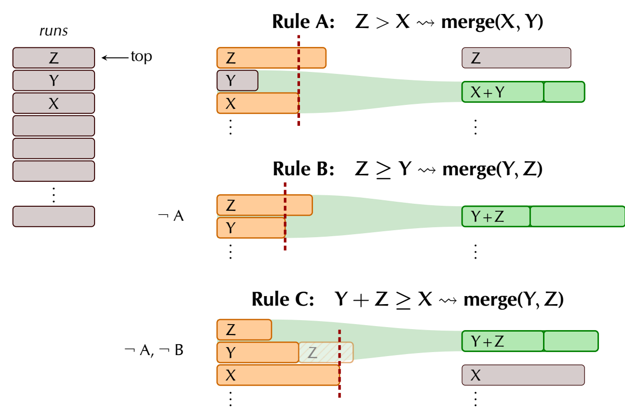
Timsort merge policy (patched)

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

Timsort merge policy (patched)

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

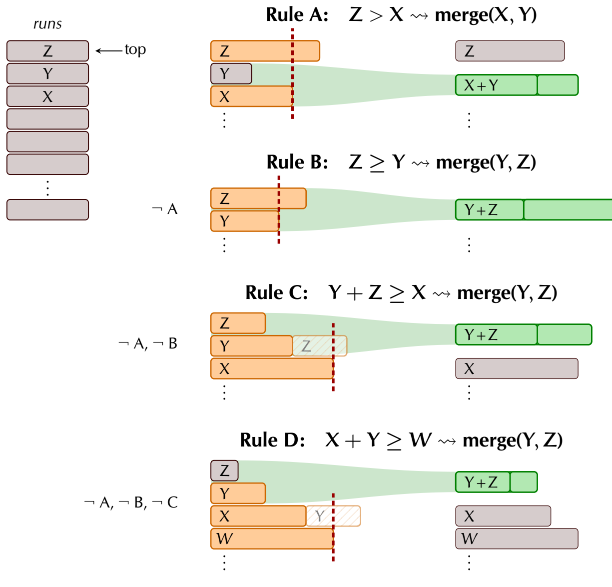
- Need to add a **Rule D**



Timsort merge policy (patched)

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

- Need to add a **Rule D**

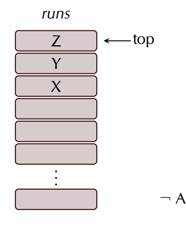


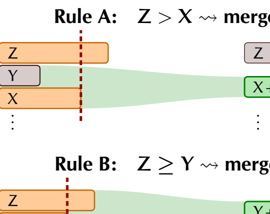
Timsort merge policy (patched)

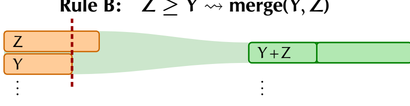
```
1 def timsort(lst):
2     i = 0; runs = []
3     while i < len(lst):
4         j = extend_run(lst, i)
5         runs.apppend((i,j)); i = j
6         while Rule A/B/C/D applicable
7             merge corresponding runs
8     while len(runs) > 1:
9         merge topmost 2 runs
```

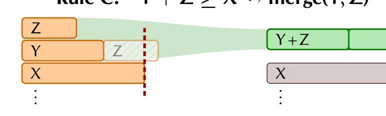
- Need to add a **Rule D**

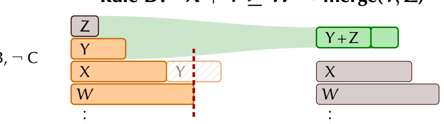
$\rightsquigarrow$ **Invariant:** $\forall j = 0, \ldots, \texttt{len(runs)} - 3$: $\texttt{runs}[j] \geqslant \texttt{runs}[j+1] + \texttt{runs}[j+2]$ ✓











Timsort merge policy (patched)

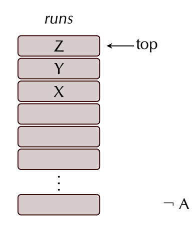
```
1 def timsort(lst):
2     i = 0; runs = []
3     while i < len(lst):
4         j = extend_run(lst, i)
5         runs.apppend((i,j)); i = j
6         while Rule A/B/C/D applicable
7             merge corresponding runs
8     while len(runs) > 1:
9         merge topmost 2 runs
```

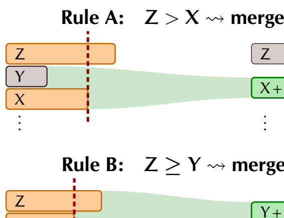
- Need to add a **Rule D**

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

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

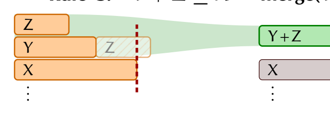
Auger, Jugé, Nicaud, Pivoteau: *On the Worst-Case Complexity of TimSort*, ESA **2018**

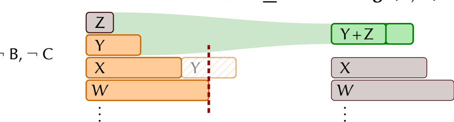




Rule B: $Z \geq Y \rightsquigarrow$ merge(Y, Z)

$\neg$ A
Z
Y
Y+Z



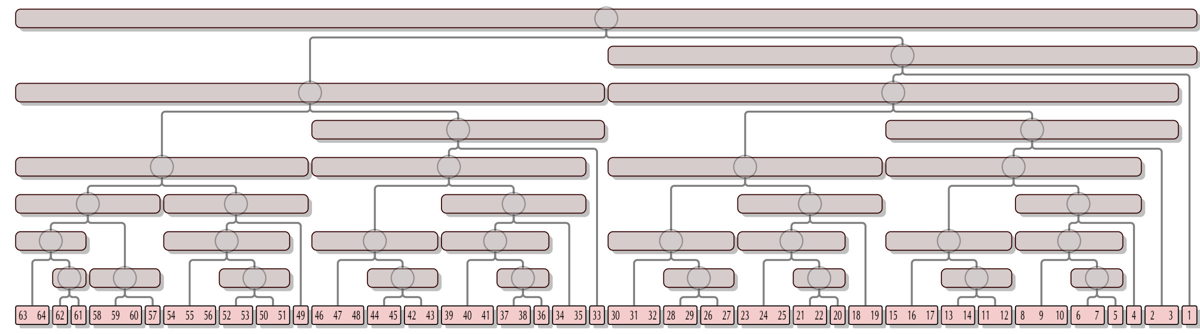


Timsort bad case

- Timsort's merge policy mostly works OK

Timsort bad case

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



Timsort bad case

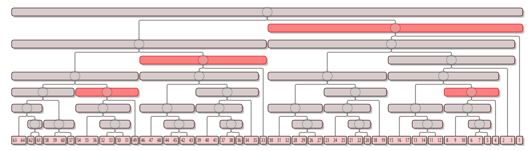
- Timsort's merge policy mostly works OK
- ...but does stupid things on certain inputs:

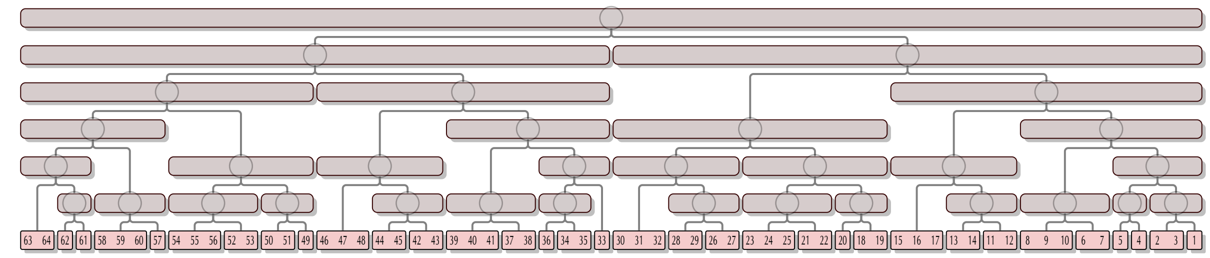
63 64 62 61 58 59 60 57 54 55 56 52 53 50 51 49 46 47 48 44 45 42 43 39 40 41 37 38 36 34 35 33 30 31 32 28 29 26 27 23 24 25 21 22 20 18 19 15 16 17 13 14 11 12 8 9 10 6 7 5 4 2 3 1

- intuitive problem: regularly very unbalanced merges

Timsort bad case

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



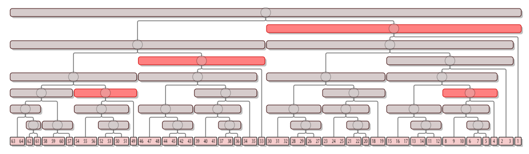


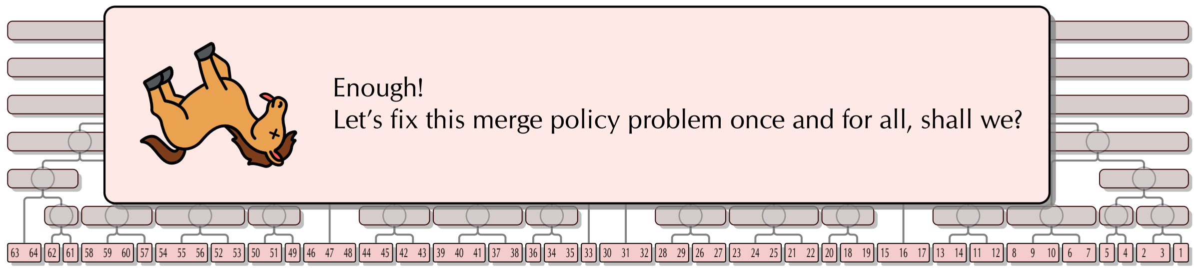
- intuitive problem: regularly very unbalanced merges

- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

Timsort bad case

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

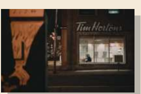




- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

Outline

1 Sort of a list



2 Timsort

3 Beware, Stackoverflow!

4 Merge policies

5 Powersort

4 Merge policies

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

1. *Find runs in input.*

Merge policies from first principles

Run-adaptive mergesort

↙ interleaved in code

Conceptually two steps:

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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

Merge policies from first principles

Run-adaptive mergesort

↙ interleaved in code

Conceptually two steps:

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

↖ determined by **merge policy**

Here:

- only binary merges

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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

$\rightsquigarrow$ 2 becomes:
merge 2 runs,
repeat until single run

- only stable sorts

$\rightsquigarrow$ *merge 2* ***adjacent*** *runs*

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

2 9 12 13 19

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

Merge policies from first principles

Run-adaptive mergesort

Conceptually two steps: ↙ interleaved in code

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:

2 9 12 13 15 17 19

2 9 12 13 19

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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

2 7 9 11 12 13 15 17 19

2 9 12 13 15 17 19

2 9 12 13 19

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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:

2 7 9 11 12 13 15 17 19

2 9 12 13 15 17 19

2 9 12 13 19 | 1 4 5 8 10 14 21 23

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

Merge policies from first principles

Run-adaptive mergesort

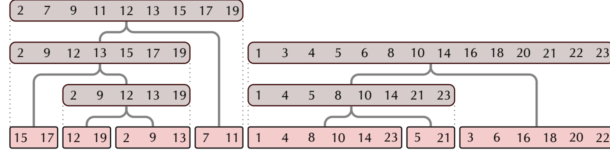
Conceptually two steps: (interleaved in code)

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

Here:

- only binary merges
- ⇝ 2 becomes:
  merge 2 runs,
  repeat until single run
- only stable sorts
- ⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

Conceptually two steps: (interleaved in code)

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

Here:

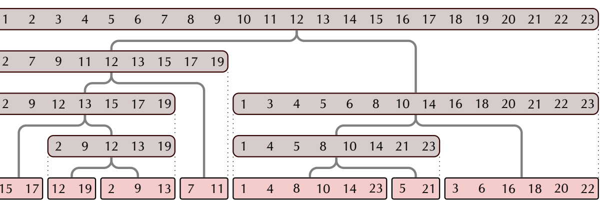
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

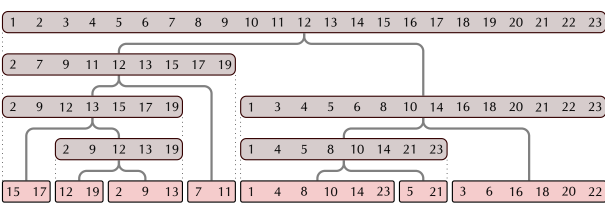
1. *Find runs in input.*
2. *Merge them **in some order**.*

determined by **merge policy**

Here:

- only binary merges
- ⇝ 2 becomes: *merge 2 runs, repeat until single run*
- only stable sorts
- ⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



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

Merge policies from first principles

Run-adaptive mergesort

↙ interleaved in code

Conceptually two steps:

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

↖ determined by **merge policy**

Here:

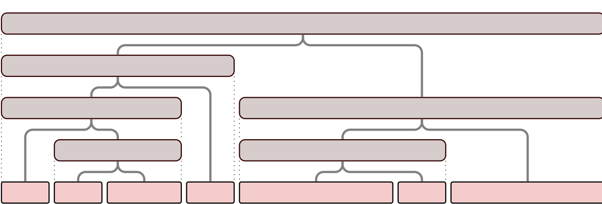
- only binary merges

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

- only stable sorts

⇝ *merge 2* ***adjacent*** *runs*

⇝ Merge order = **merge tree**:



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

Merge costs

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

Merge policies from first principles

Run-adapti

interlea

Conceptua

1. *Find r*
2. *Merge*

Here:

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

- only stable sorts
- ⇝ *merge 2* ***adjacent*** *runs*

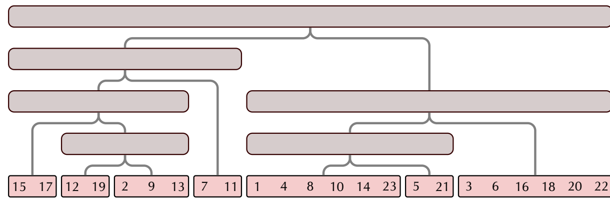
Different merge policies yield different merge costs!

tree

Merge costs

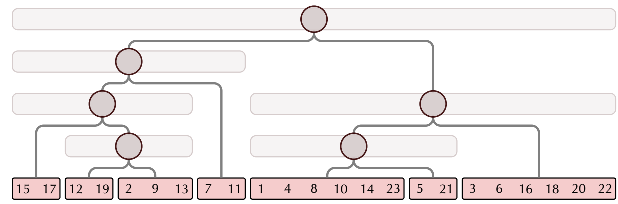
- cost of merge := size of output
  - ≈ memory transfers
  - $\geqslant$ #cmps
- total cost = total area of

Mergesort meets Binary Search Trees



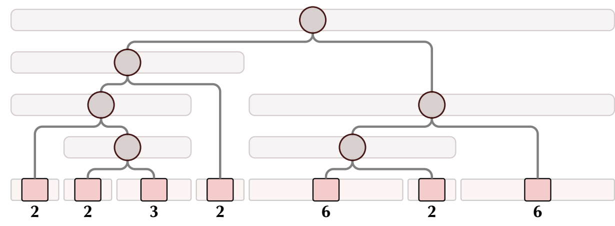
Merge cost = total area of

Mergesort meets Binary Search Trees



Merge cost = total area of [shaded box]
= total length of paths to all array entries

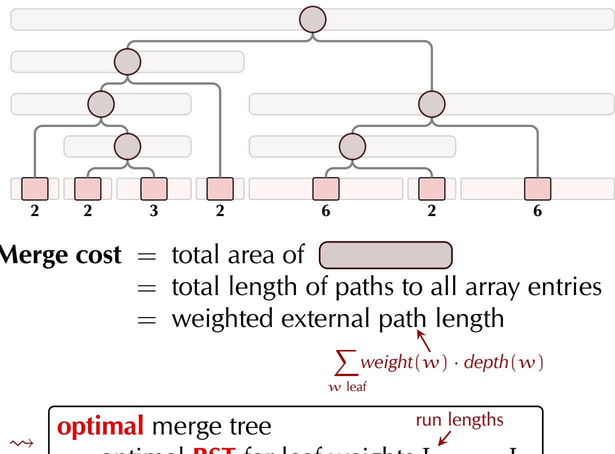
Mergesort meets Binary Search Trees



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

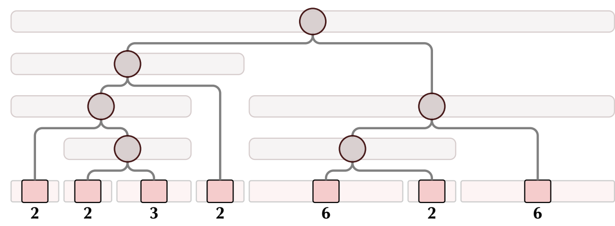
$$\sum_{w \text{ leaf}} \mathit{weight}(w) \cdot \mathit{depth}(w)$$

Mergesort meets Binary Search Trees



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

Mergesort meets Binary Search Trees



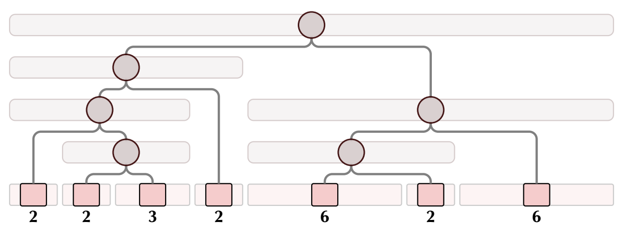
Merge cost = total area of [shaded rectangle]
= total length of paths to all array entries
= weighted external path length

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

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

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

Mergesort meets Binary Search Trees



How to compute good merge tree?

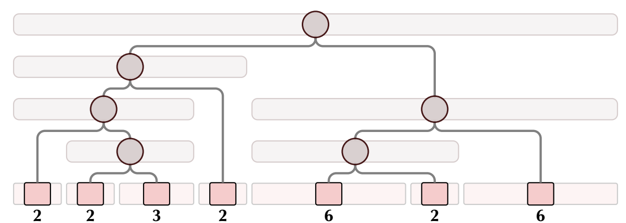
Merge cost = total area of [box]
= total length of paths to all array entries
= weighted external path length

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

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

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

Mergesort meets Binary Search Trees



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

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

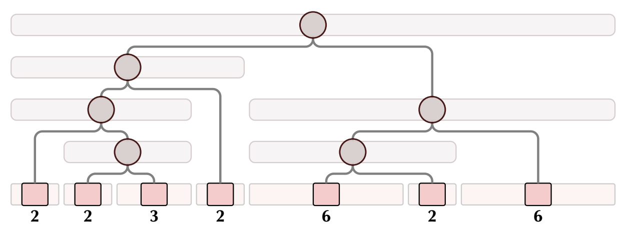
⇝ **optimal** merge tree
= optimal **BST** for leaf weights $L_1, \ldots, L_r$ (run lengths)

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

How to compute good merge tree?

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

Mergesort meets Binary Search Trees



Merge cost = total area of [gray box]
= total length of paths to all array entries
= weighted external path length

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

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

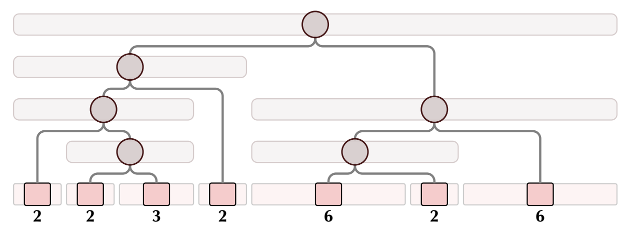
⇝ merge cost $\geqslant \mathcal{H}n$ with $\mathcal{H} = \sum_{i=1}^{r} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right)$

How to compute good merge tree?

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

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

Mergesort meets Binary Search Trees



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

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

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

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

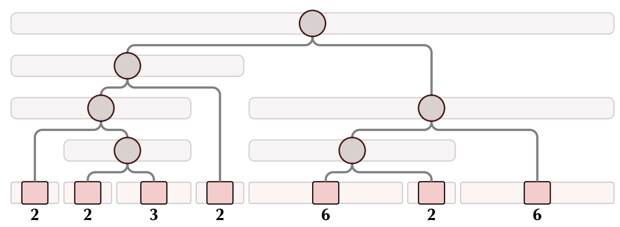
How to compute good merge tree?

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

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

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

Mergesort meets Binary Search Trees



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

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

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

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

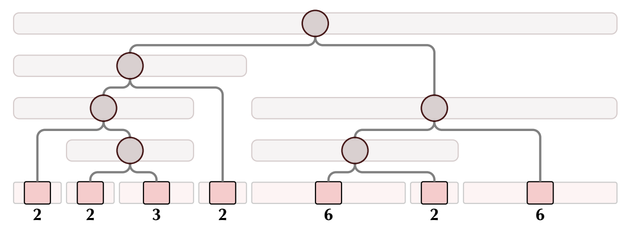
How to compute good merge tree?

- ***Huffman merge***
  - merge shortest runs ← indep. discovered 5x
    - Burge 1958
    - Golin & Sedgewick 1993
    - Takaoka 1998
    - Barbay & Navarro 2009
    - Chandramouli & Goldstein 2014
  - must sort lengths
  - 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!

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge tree?

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

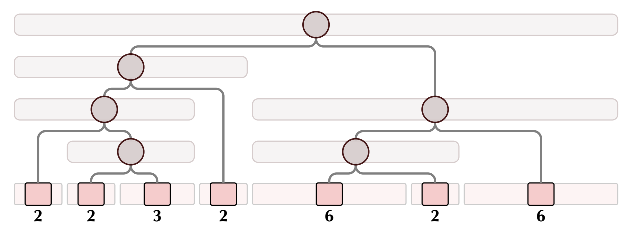
indep. discovered 5x
- Burge 1958
- Golin & Sedgewick 1993
- Takaoka 1998
- Barbay & Navarro 2009
- Chandramouli & Goldstein 2014

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

. . . 70s are calling

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

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge tree?

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

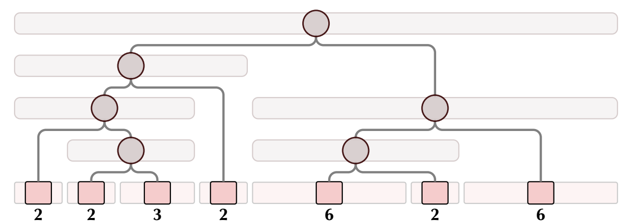
indep. discovered 5x
- Burge 1958
- Golin & Sedgewick 1993
- Takaoka 1998
- Barbay & Navarro 2009
- Chandramouli & Goldstein 2014

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



Merge cost = total area of [shaded rectangle]
= total length of paths to all array entries
= weighted external path length

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

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

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

How to compute good merge tree?

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

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

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

} avoidable!

The Bisection Method

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

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

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

The Bisection Method

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

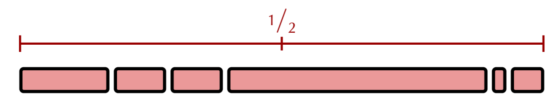
Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

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

The Bisection Method

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

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

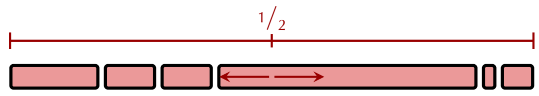


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

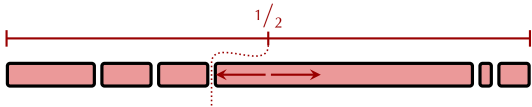
- Powersort is based on the **bisection method**

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



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

The Bisection Method



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

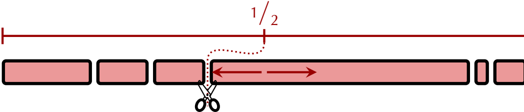
Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

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

The Bisection Method

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

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



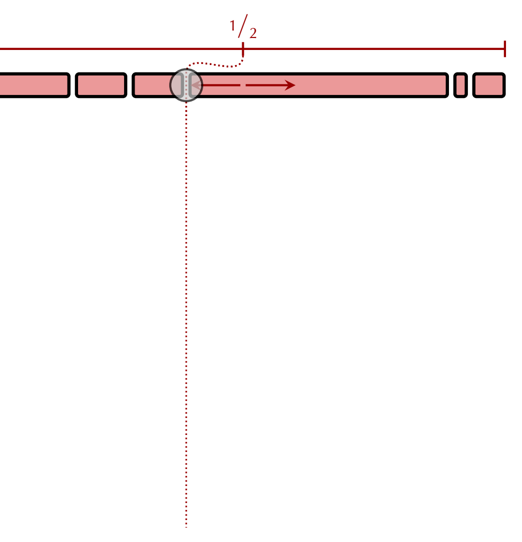
- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

The Bisection Method

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

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

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

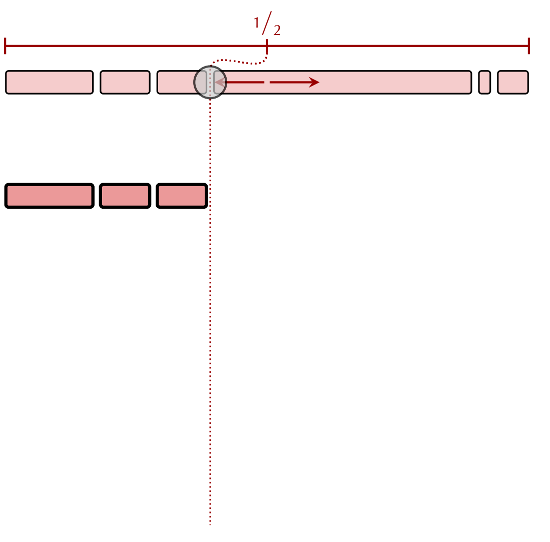


The Bisection Method

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

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

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

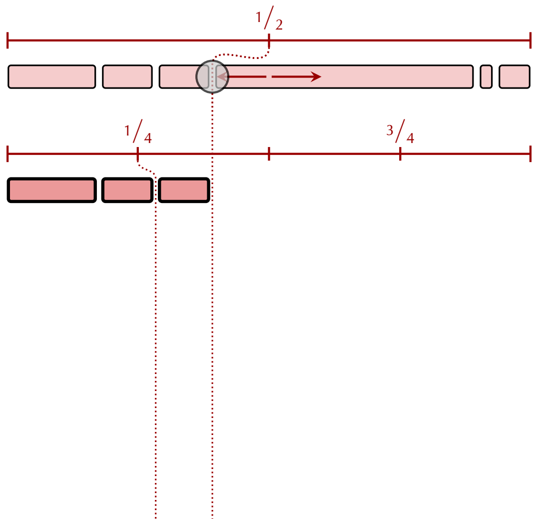


The Bisection Method

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

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

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

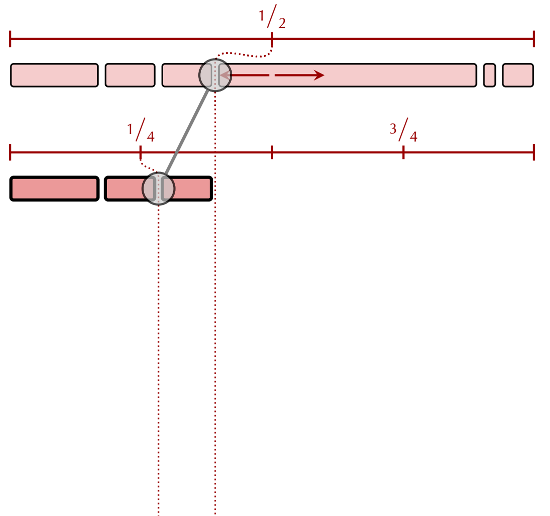


The Bisection Method

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

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

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

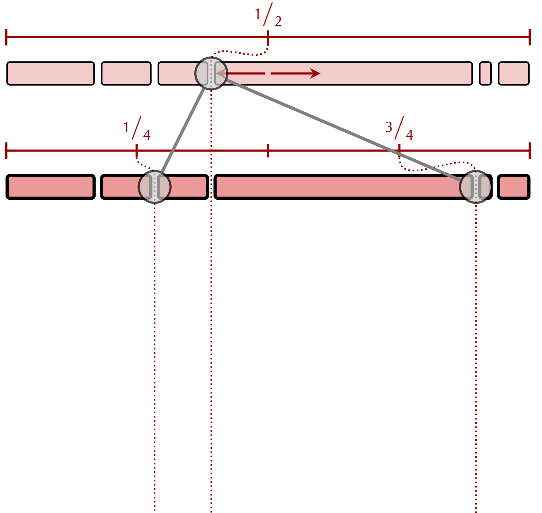


The Bisection Method

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

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

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

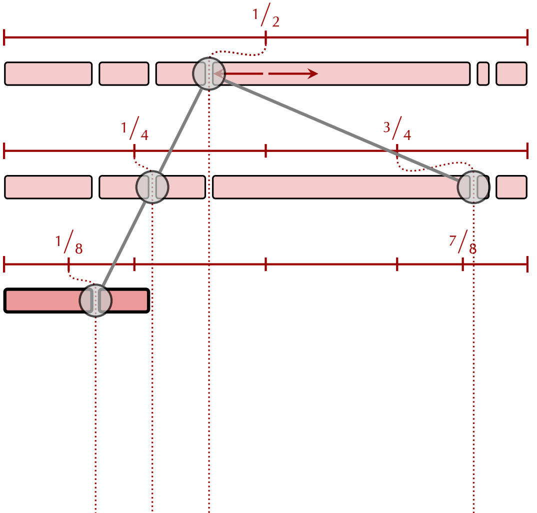


The Bisection Method

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

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

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

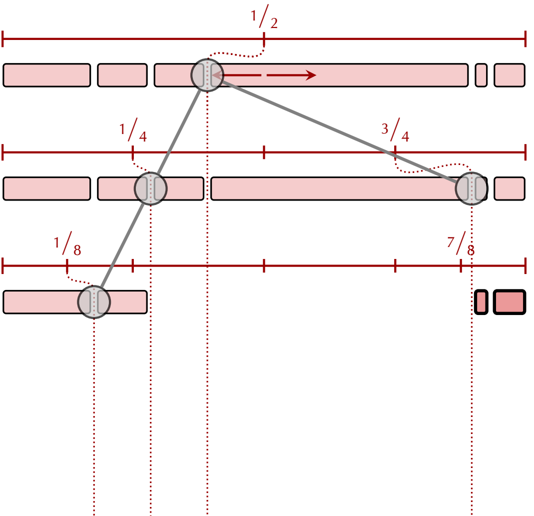


The Bisection Method

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

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

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

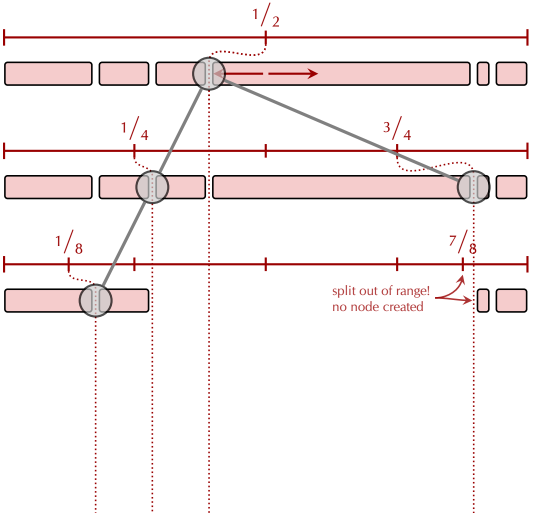


The Bisection Method

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

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

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

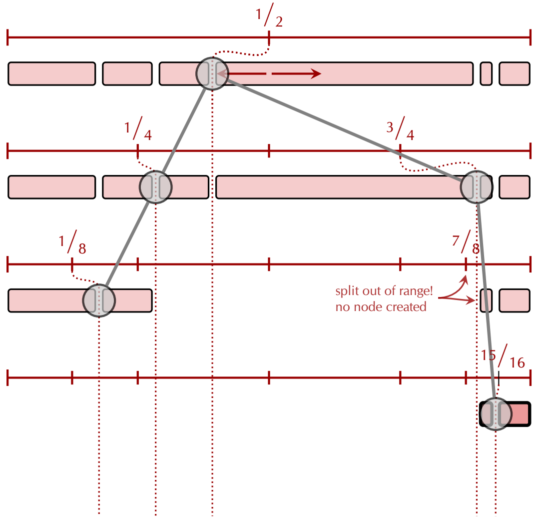


The Bisection Method

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

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

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

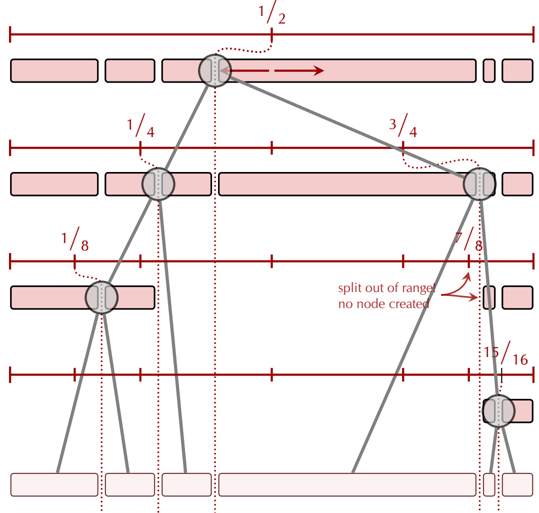


The Bisection Method

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

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

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



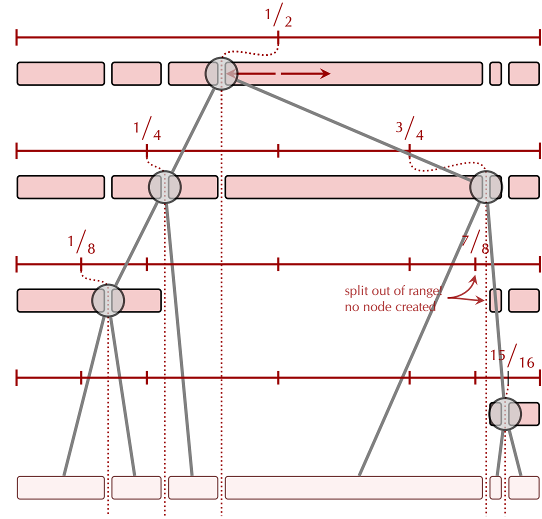
The Bisection Method

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

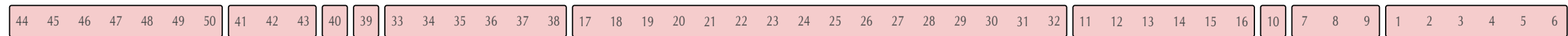
Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

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

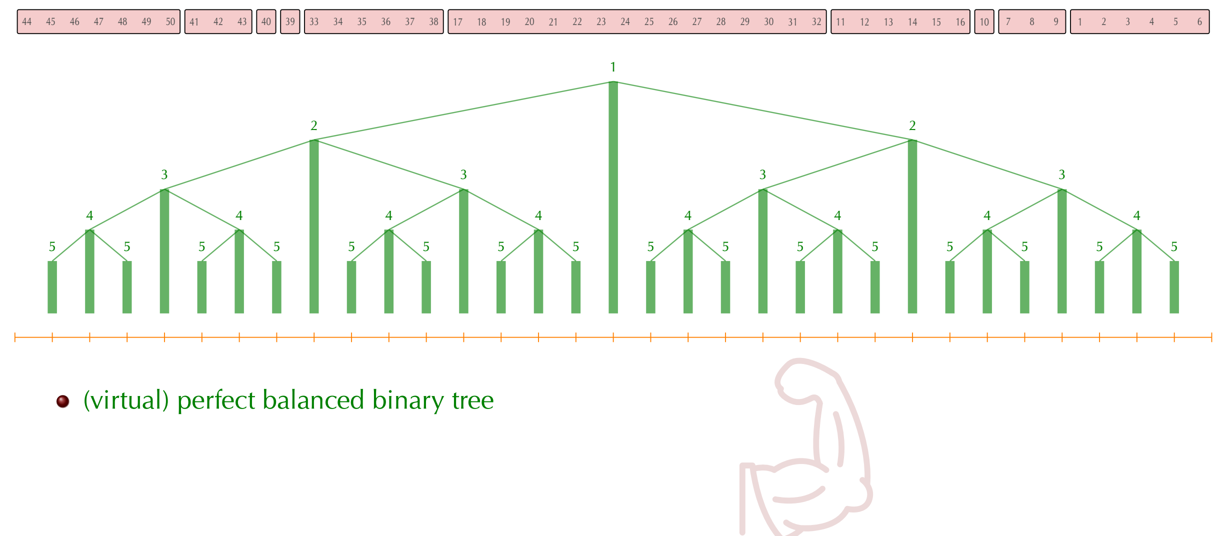
- *Up next:* Don't use recursion!



Run-Boundary Powers

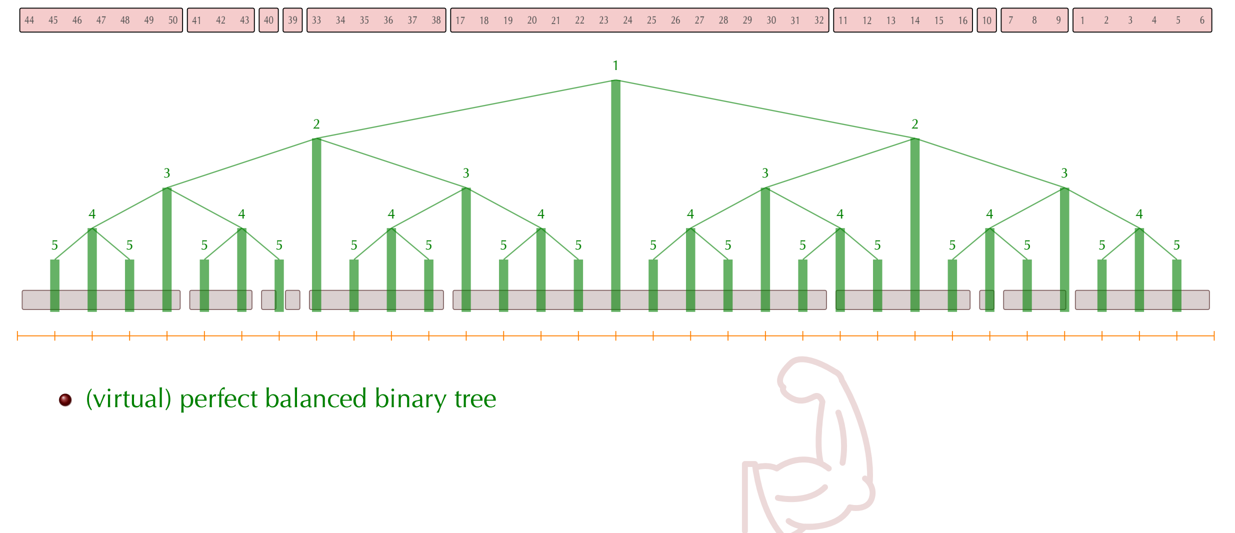


Run-Boundary Powers



- (virtual) perfect balanced binary tree

Run-Boundary Powers



- (virtual) perfect balanced binary tree

Run-Boundary Powers

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

Run-Boundary Powers

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

Run-Boundary Powers

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

Run-Boundary Powers

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

Run-Boundary Powers

44 45 46 47 48 49 50 | 41 42 43 | 40 | 39 | 33 34 35 36 37 38 | 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | 11 12 13 14 15 16 | 10 | 7 8 9 | 1 2 3 4 5 6

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

Run-Boundary Powers

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

Run-Boundary Powers

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

Run-Boundary Powers

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

Run-Boundary Powers

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

Run-Boundary Powers

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

⇝ merge tree follows virtual tree

Run-Boundary Powers

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

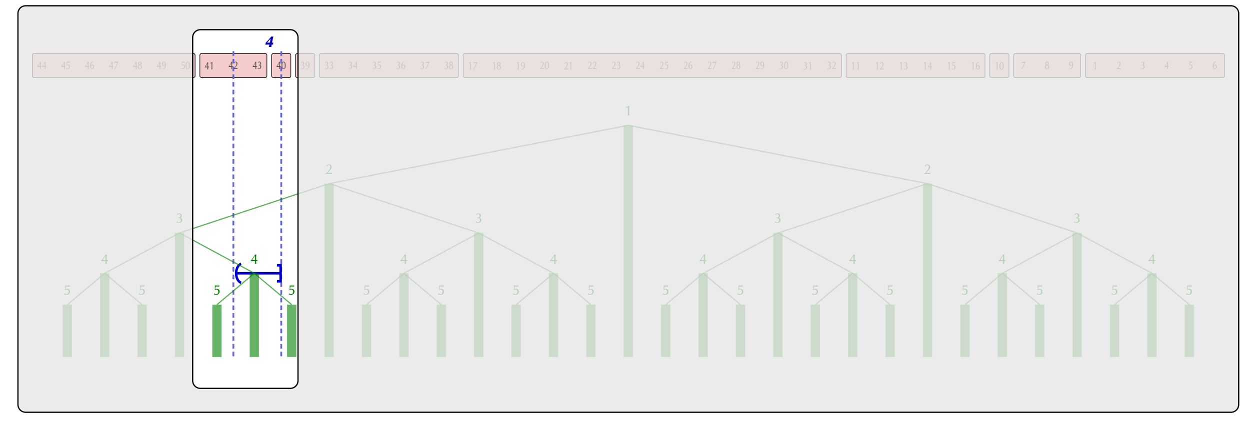
⇝ merge tree follows virtual tree

Run-Boundary Powers

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

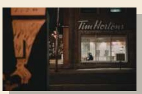
⇝ merge tree follows virtual tree

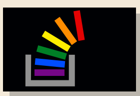
Run-Boundary Powers are Local

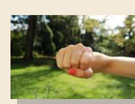


Computation of powers only depends on two adjacent runs.

Outline







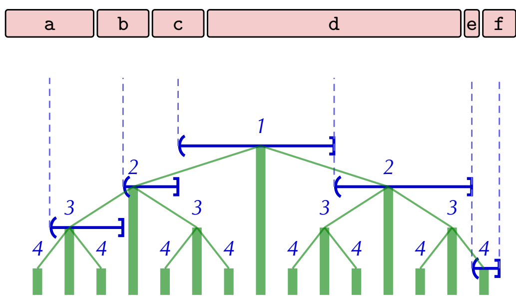
1. Sort of a list
2. Timsort
3. Beware, Stackoverflow!
4. Merge policies
5. Powersort

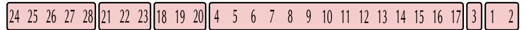
5 Powersort

Powersort

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

full code:
`tiny.cc/timsort`





Powersort

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

full code:
`tiny.cc/timsort`

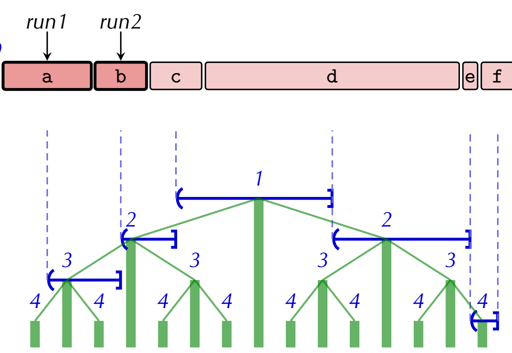
Powersort

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

full code:
`tiny.cc/timsort`

a – **0**

run stack

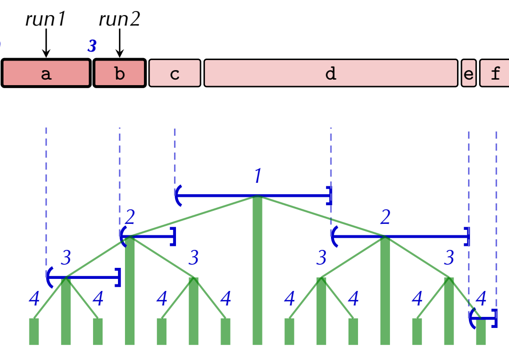


Powersort

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

a – **0**

run stack



24 25 26 27 28 | 21 22 23 | 18 19 20 | 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | 3 | 1 2

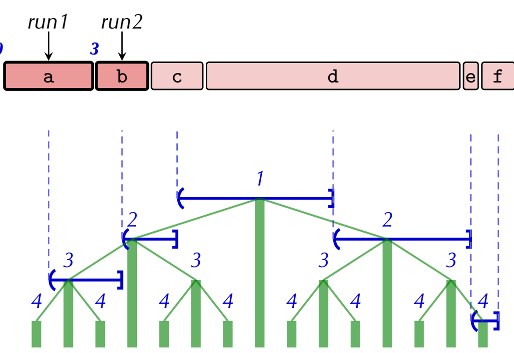
full code:
`tiny.cc/timsort`

Powersort

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

b – **3**
a – **0**

run stack



24 25 26 27 28	21 22 23	18 19 20	4 5 6 7 8 9 10 11 12 13 14 15 16 17	3	1 2

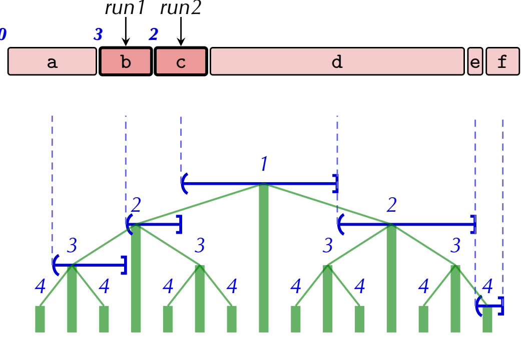
full code:
`tiny.cc/timsort`

Powersort

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

b – **3**
a – **0**

run stack

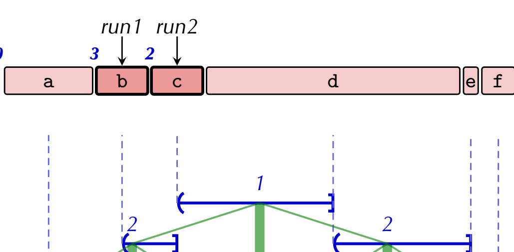


24 25 26 27 28 | 21 22 23 | 18 19 20 | 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | 3 | 1 2

full code:
`tiny.cc/timsort`

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

run stack
c – **2**
b – **3**
a – **0**



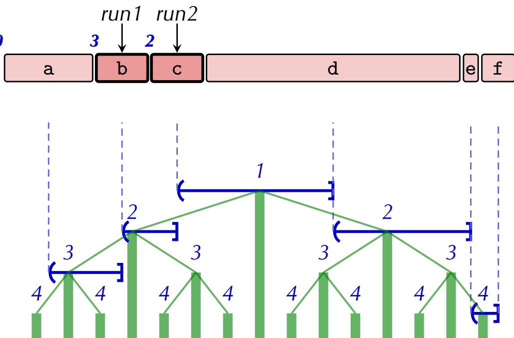
24 25 26 27 28 | 21 22 23 | 18 19 20 | 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | 3 | 1 2

full code:
`tiny.cc/timsort`

Powersort

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

run stack
c – **2**
b – **3**
a – **0**



24 25 26 27 28	21 22 23	18 19 20	4 5 6 7 8 9 10 11 12 13 14 15 16 17	3	1 2

full code:
`tiny.cc/timsort`

Powersort

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

c – **2**
b – **3**
a – **0**
run stack

run1 *run2*

0 a *3* b *2* c d e f

merge

1 *2* *2* *3* *3* *3* *3* *4* *4* *4* *4* *4* *4* *4* *4*

24 25 26 27 28 | 21 22 23 | 18 19 20 | 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | 3 | 1 2

full code:
`tiny.cc/timsort`

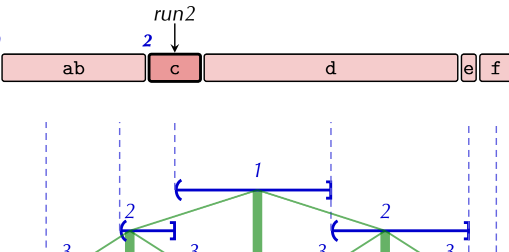
Powersort

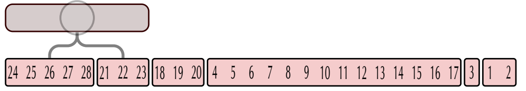
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

c – **2**

ab – **0**

run stack





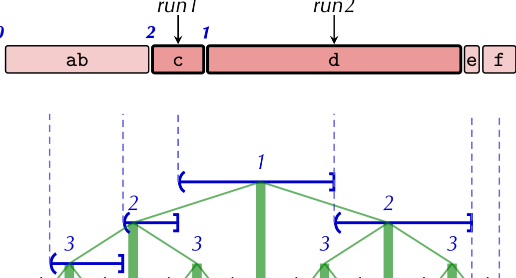
full code:
`tiny.cc/timsort`

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

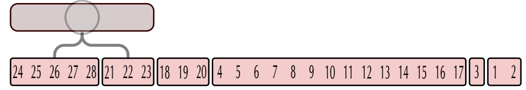
c – **2**

ab – **0**

run stack



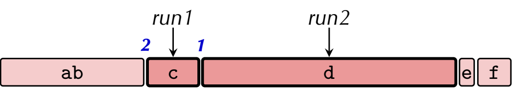
full code:
`tiny.cc/timsort`

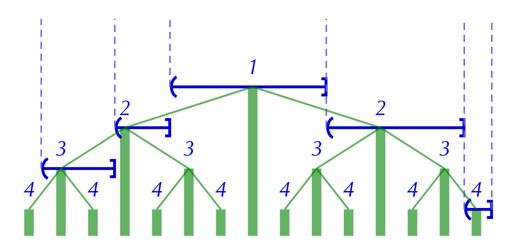


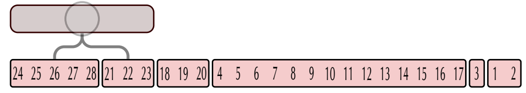
Powersort

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

d – **1**
c – **2**
ab – **0**
run stack





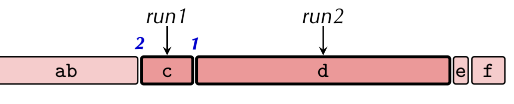


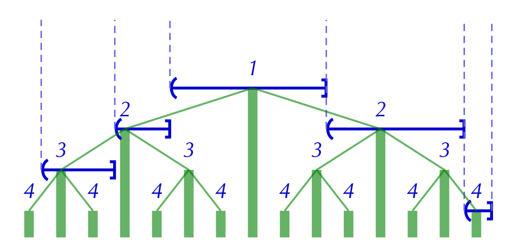
full code:
`tiny.cc/timsort`

Powersort

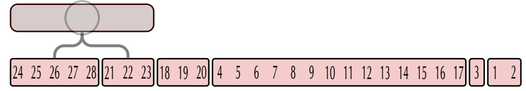
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

run stack
d – **1**
c – **2**
ab – **0**





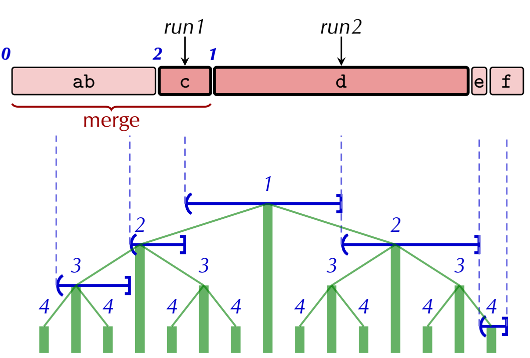
full code:
`tiny.cc/timsort`

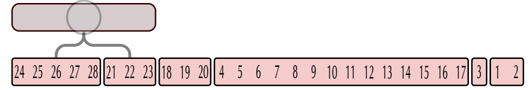


Powersort

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

full code:
`tiny.cc/timsort`



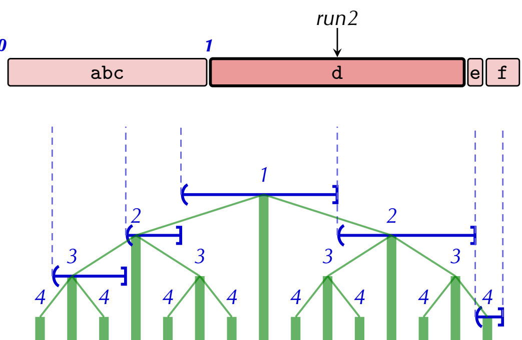


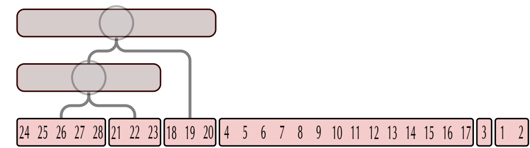
Powersort

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

full code:
`tiny.cc/timsort`

run stack
d – **1**
abc – **0**

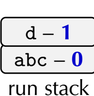




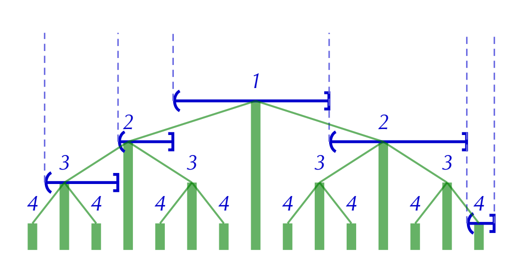
Powersort

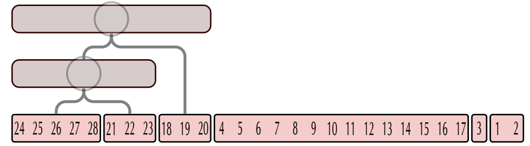
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

full code:
`tiny.cc/timsort`



run stack

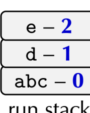


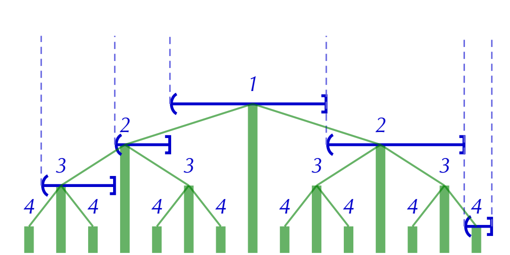


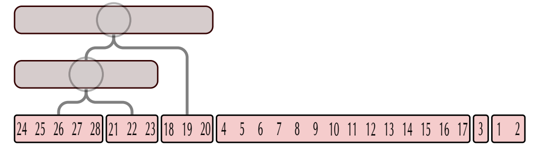
Powersort

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

full code:
`tiny.cc/timsort`



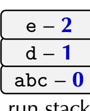


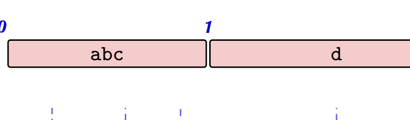


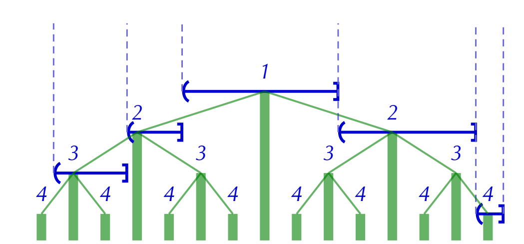
Powersort

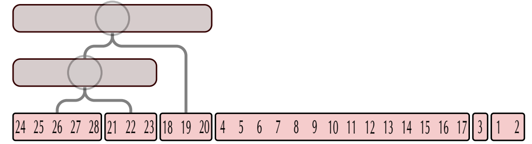
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

full code:
`tiny.cc/timsort`



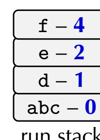


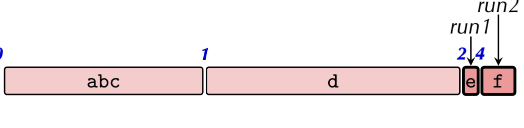


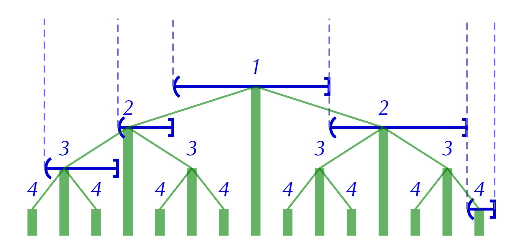


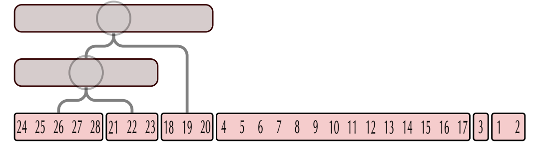
Powersort

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







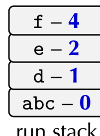


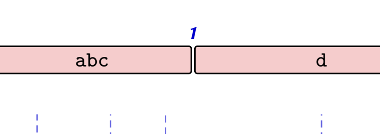
full code:
`tiny.cc/timsort`

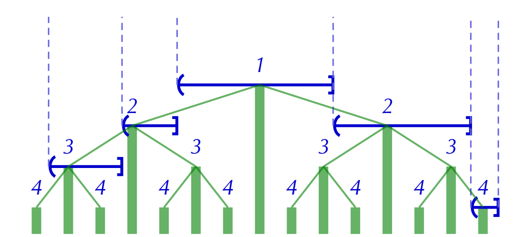
Powersort

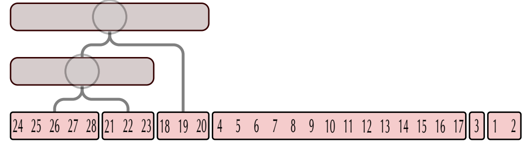
```
1 def powersort(lst)
2     i = 0; n = len(lst)
3     runs = [];
4     j = extend_run(lst, i)
5     runs.append((i,j-i,0)); i = j
6     while i < n:
7         j = extend_run(lst, i)
8         p = power(runs[-1], (i,j-i), n)
9         while p <= runs[-1][2]
10            merge_topmost_2(lst, runs)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

full code:
`tiny.cc/timsort`





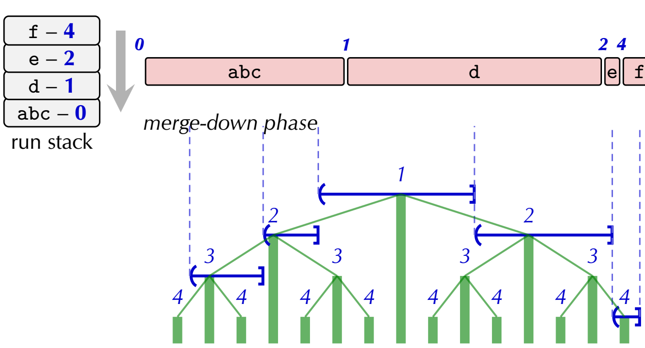


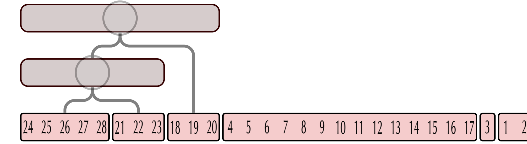


Powersort

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

full code:
`tiny.cc/timsort`

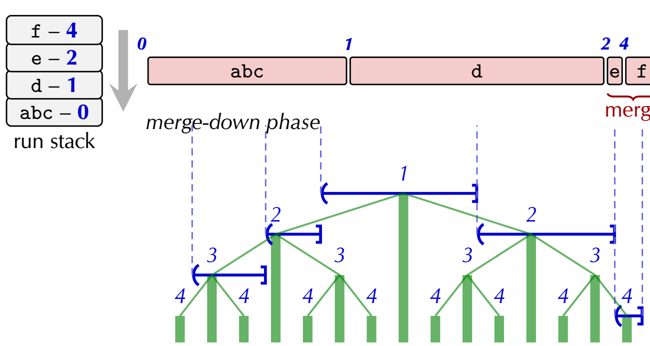


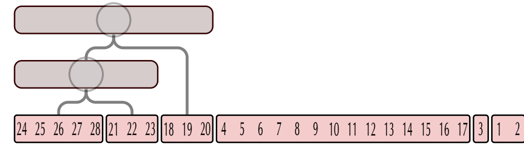


Powersort

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

full code:
`tiny.cc/timsort`

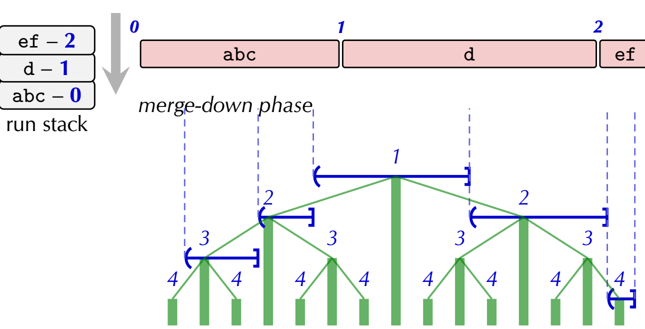


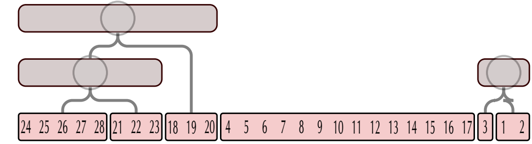


Powersort

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

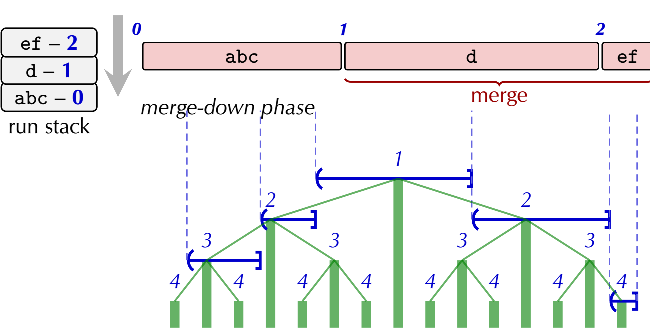
full code:
`tiny.cc/timsort`

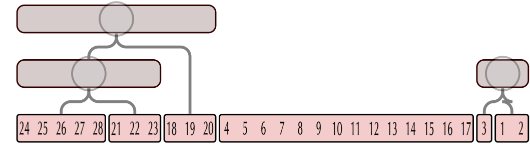




Powersort

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



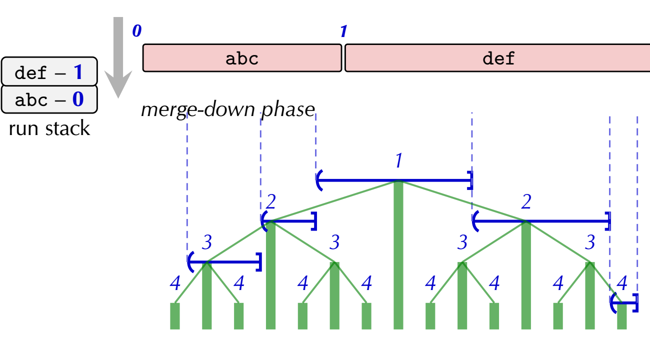


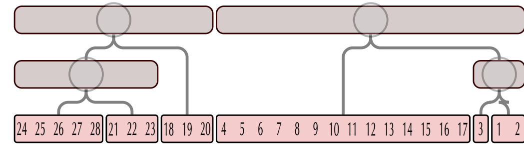
full code:
`tiny.cc/timsort`

Powersort

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

full code:
`tiny.cc/timsort`

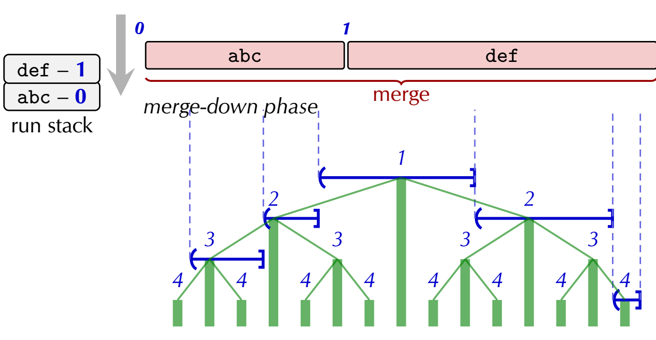


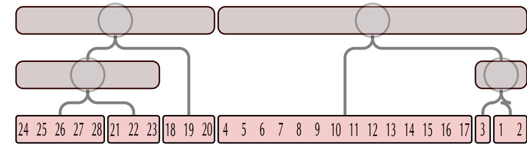


Powersort

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

full code:
`tiny.cc/timsort`





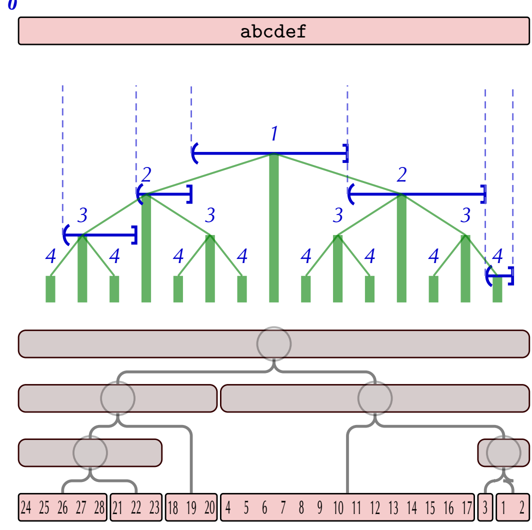
Powersort

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

abcdef – **0**

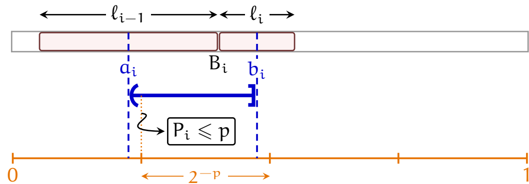
run stack

full code:
`tiny.cc/timsort`



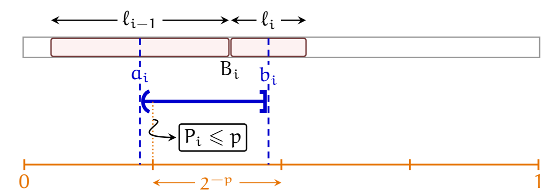
Computing powers

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



Computing powers

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

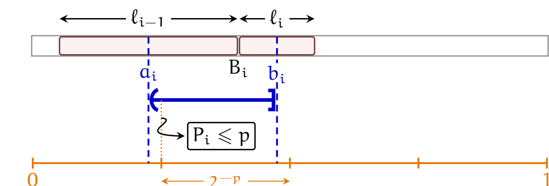



```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      A = 2 * i1 + n1    # A = 2na_i
5      B = A + n1 + n2    # B = 2nb_i
6      p = 0
7      while True:
8          p += 1
9          if A >= n:
10             A -= n; B -= n
11         elif B >= n:
12             break
13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

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

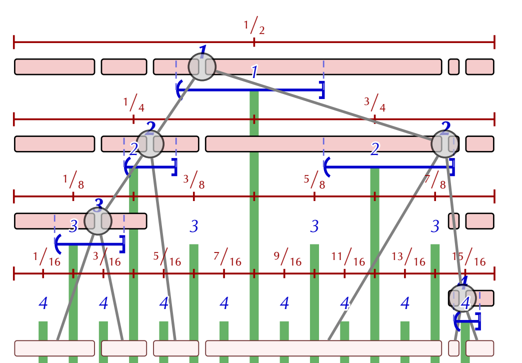
Computing powers

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




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

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



Mergecost Analysis

- **Given:** input with runs $R_1, \ldots, R_r$ of
  lengths $L_1, \ldots, L_r$; $\quad n = L_1 + \cdots + L_r$

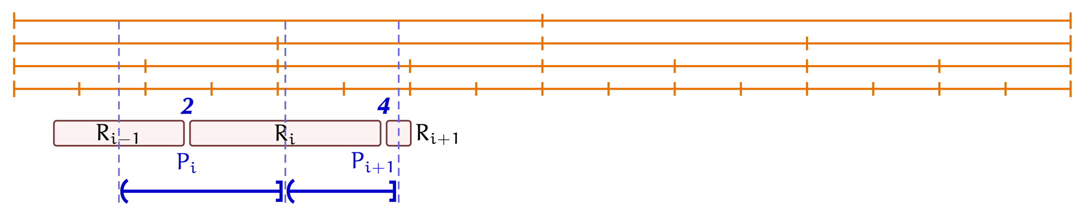
- Mergecost: $M = \sum_{r=1}^{r} L_i \cdot \mathrm{depth}(R_i)$

Mergecost Analysis

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

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

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



Mergecost Analysis

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

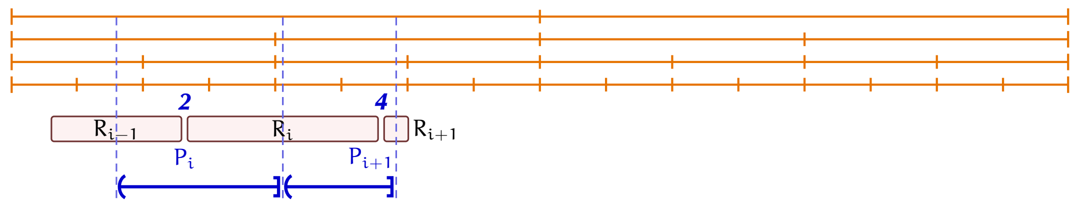
$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

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

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

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

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



Mergecost Analysis

- **Given:** input with runs $R_1, \ldots, R_r$ of **lengths** $L_1, \ldots, L_r$; $\quad n = L_1 + \cdots + L_r$

- Mergecost: $M = \sum_{r=1}^{r} L_i \cdot \text{depth}(R_i)$

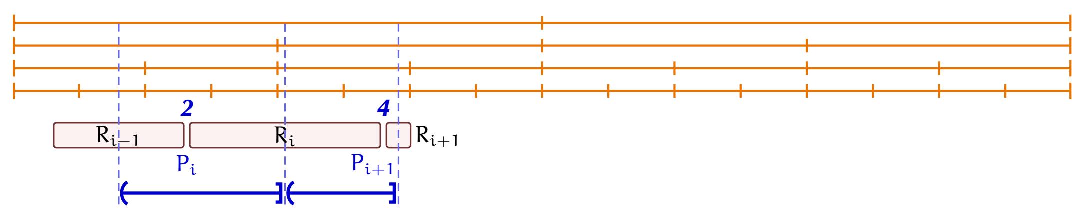
$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

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

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

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

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



Mergecost Analysis

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

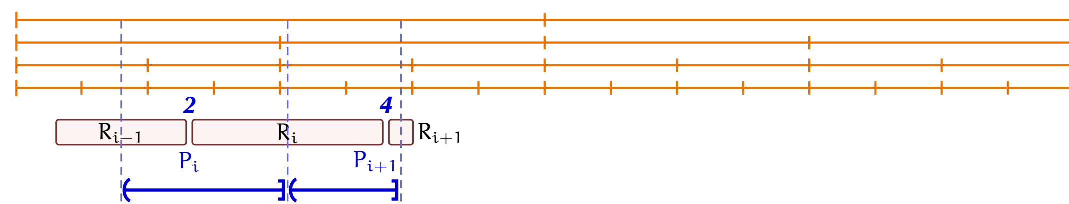
$\rightsquigarrow$ $M \leqslant \sum_{r=1}^{r} L_i \cdot \hat{P}_i$, where $\hat{P}_i = \max\{P_i, P_{i+1}\}$

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

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

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

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



Mergecost Analysis

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

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

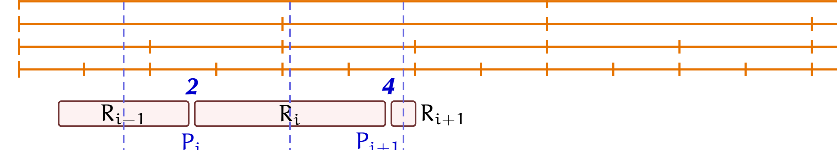
P_i = power of boundary between R_{i-1} and R_i, $i = 2, \ldots, r$
$P_1 = P_{r+1} = 0$

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

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

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

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



Some performance data

Summary claims:

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

Abstract cost measures

Scatter plot of relative costs $\frac{\text{Timsort}}{\text{Powersort}}$

#comparisons

1.1

1.05

1

Powersort better

Timsort better

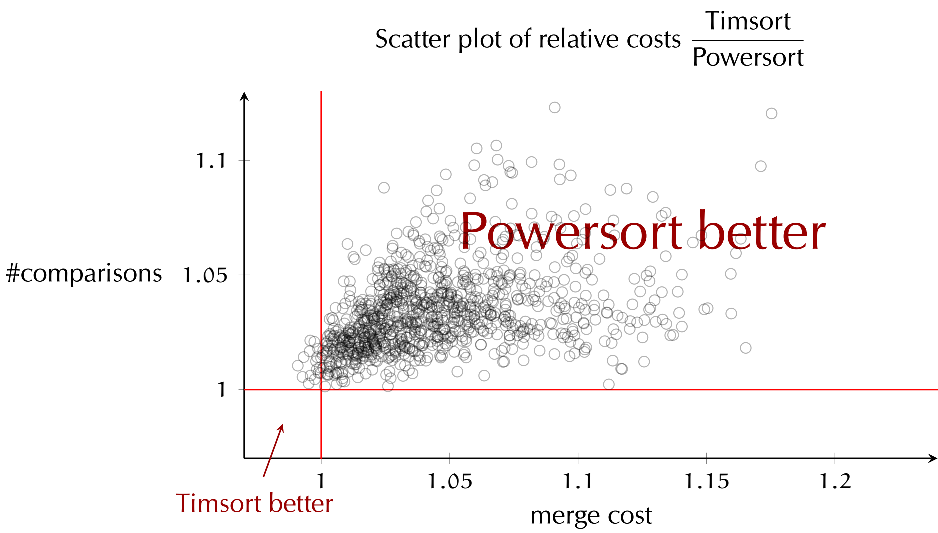
1 1.05 1.1 1.15 1.2

merge cost

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

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

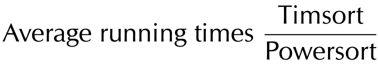
Abstract cost measures



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

Input: Tim's mixture of long and short runs

Running times



1.5
1
0.5
0
Powersort faster
Timsort faster
bad-example-cmps
bad-example-cmps-1m
bad-example-cmps-1m2
bad-example-mc
random-permutations
random-permutations2
random-sqrtn-runs
random-sqrtn-runs2
random-sqrtn-runs3
words-of-bible

- CPython 3.11 with Powersort resp. Timsort
- selection of some "random" inputs
- most inputs `ints` (cheap comparisons)
- machine-dependent, but qualitatively stable

Machine 1

Running times

1.5
1
0.5
0

Powersort faster

Timsort faster

bad-example-cmps
bad-example-cmps-1m
bad-example-cmps-1m-2
bad-example-mc
random-permutations
random-permutations2
random-sqrtn-runs
random-sqrtn-runs2
random-sqrtn-runs3
words-of-bible

- CPython 3.11 with Powersort resp. Timsort
- selection of some "random" inputs
- most inputs `ints` (cheap comparisons)
- machine-dependent, but qualitatively stable

Machine 2

Bonus: Powersort in Databases

- in the context of databases,
  adaptive sorting not as widely used
- but tricks to avoid costly or redundant comparisons are widely adopted
  - *normalized keys*
  - *offset-value coding*

Bonus: Powersort in Databases

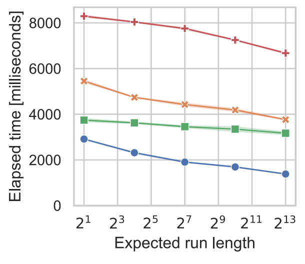
- in the context of databases, adaptive sorting not as widely used
- but tricks to avoid costly or redundant comparisons are widely adopted
  - *normalized keys*
  - *offset-value coding*

⇝ Can be combined, with cumulative benefit!

Kuhrt, Seeger, Wild, Graefe: *Adaptive sorting for large keys, strings, and database rows*, BTW **2025**

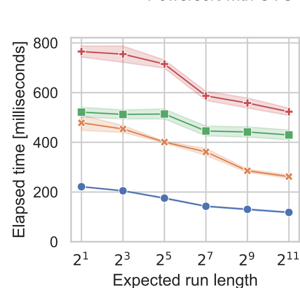
Time as function of presortedness

std::stable_sort
std::sort
Powersort
Powersort with OVC



German words

words from news articles from Leipniz Corpora Collection



Database rows

database benchmark TPC(SF = 1) normalized keys of `catalog_sales`

Bonus: Multiway powersort

- Timsort has been highly successful export from Python

...

Bonus: Multiway powersort

- Timsort has been highly successful export from Python

...

▢ = now using Powersort merge policy

Bonus: Multiway powersort

- Timsort has been highly successful export from Python

...

- In other contexts, comparisons can be *much* **cheaper**
  - ⇝ need to economize on memory transfers*
  - ⇝ can profit from **multiway merging** (instead of 2 runs at a time)

Bonus: Multiway powersort

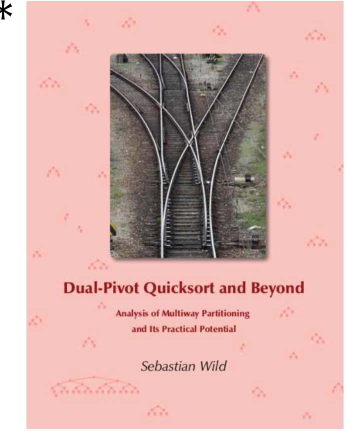
- Timsort has been highly successful export from Python

...

☐ = now using Powersort merge policy

- In other contexts, comparisons can be *much* **cheaper**
  - ⇝ need to economize on memory transfers*
  - ⇝ can profit from **multiway merging** (instead of 2 runs at a time)

*



(My PhD was on how this affects Quicksort)

Wild: *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential*, PhD thesis **2016**

Bonus: Multiway powersort

- Timsort has been highly successful export from Python

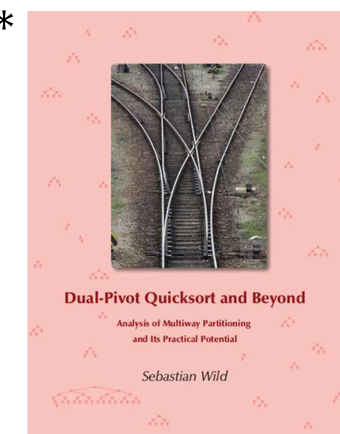
...

- In other contexts, comparisons can be *much* **cheaper**
  - ⇝ need to economize on memory transfers*
  - ⇝ can profit from **multiway merging** (instead of 2 runs at a time)

⇝ Easy to do with Powersort while keeping adaptivity!

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

*



(My PhD was on how this affects Quicksort)

Wild: *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential,* PhD thesis **2016**

Conclusion

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

    def found_new_run(self, run):

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

    def merge_force_collapse(self):
        p = self.pending
```

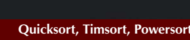
Conclusion

Summary

Conclusion

Summary

- Timsort is stable sorting method of choice



Conclusion

Summary

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

```
def found_new_run(self, run):

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

def merge_force_collapse(self):
    p = self.pending
```

Conclusion

Summary

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

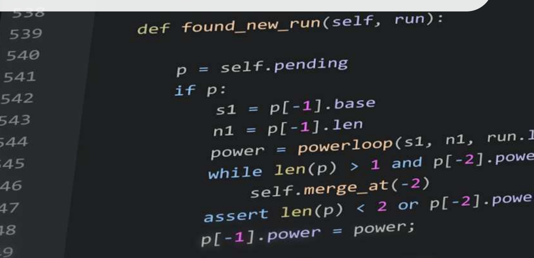
And they lived merrily ever after.

Conclusion

Summary

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

And they lived merrily ever after.



Goals

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

Conclusion

Summary

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

And they lived merrily ever after.

Goals

1. Powersort in **other libraries** using Timsort?
   - numpy & pandas
   - OpenJDK
   - Android Java runtime
   - V8 JavaScript engine
   - GNU STL `std::stable_sort`
2. How much does adaptive sorting help?
   - What are typical inputs like in difference applications?
   - Is sorting used at all?

```
def found_new_run(self, run):

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

def merge_force_collapse(self):
    p = self.pending
```

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

    def found_new_run(self, run):

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

    def merge_force_collapse(self):
        p = self.pending
```

Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

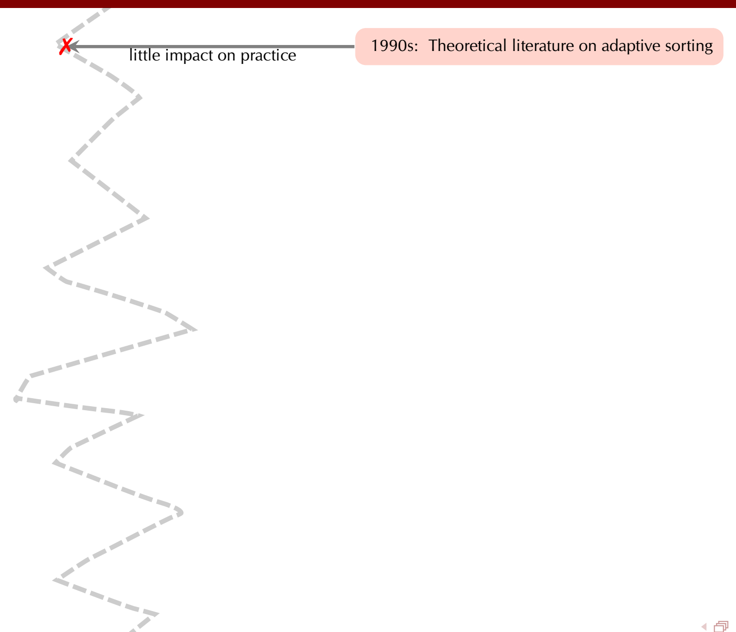
↓

Timsort in CPython 2001
ad-hoc rules for merge policy

Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

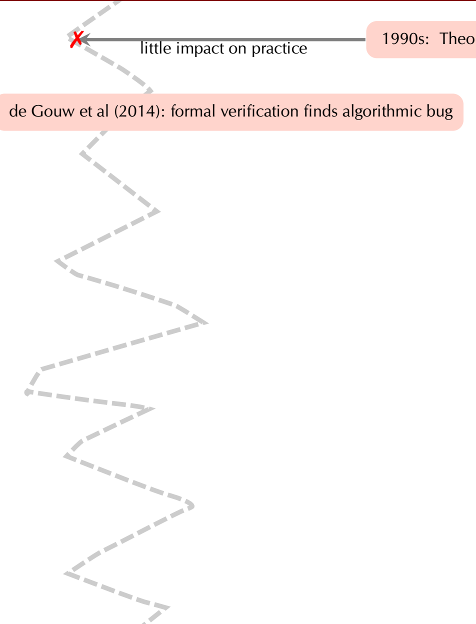
Timsort in CPython 2001
ad-hoc rules for merge policy



Powersort: An Algorithm Science Success Story

Observation: data often partially sorted

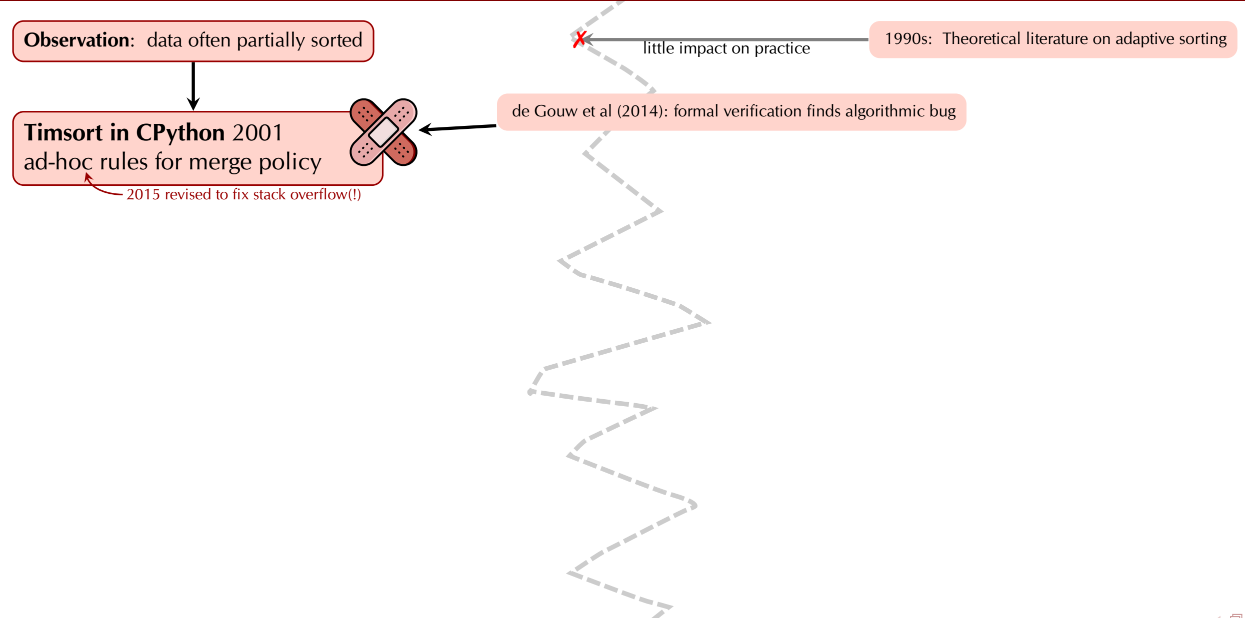
Timsort in CPython 2001
ad-hoc rules for merge policy



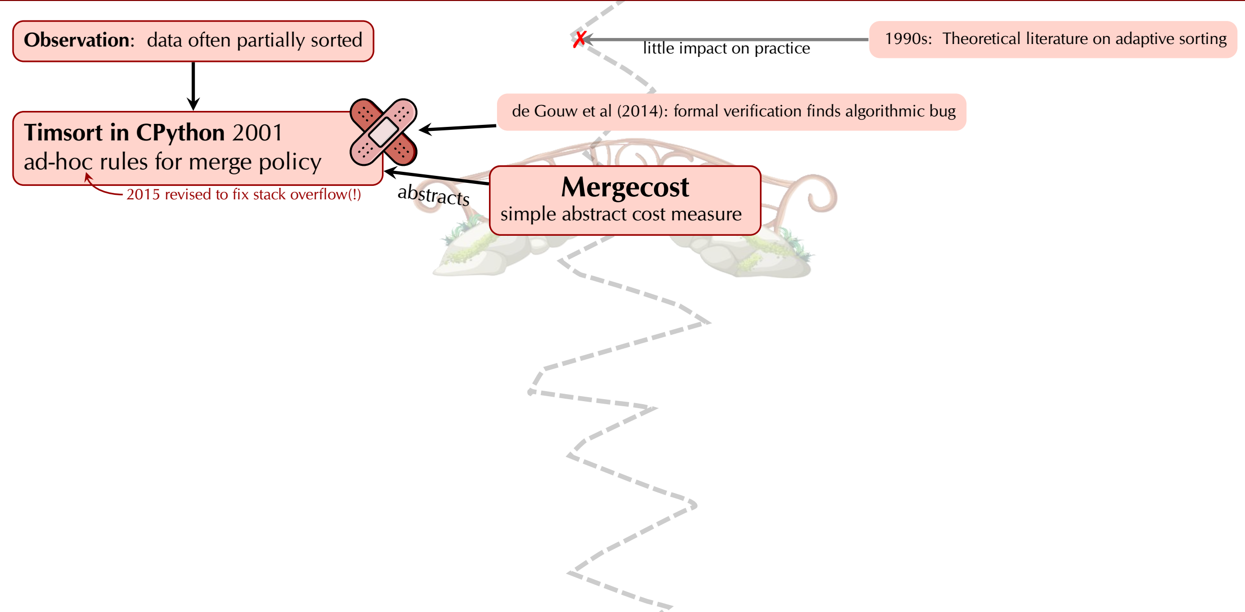
1990s: Theoretical literature on adaptive sorting

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

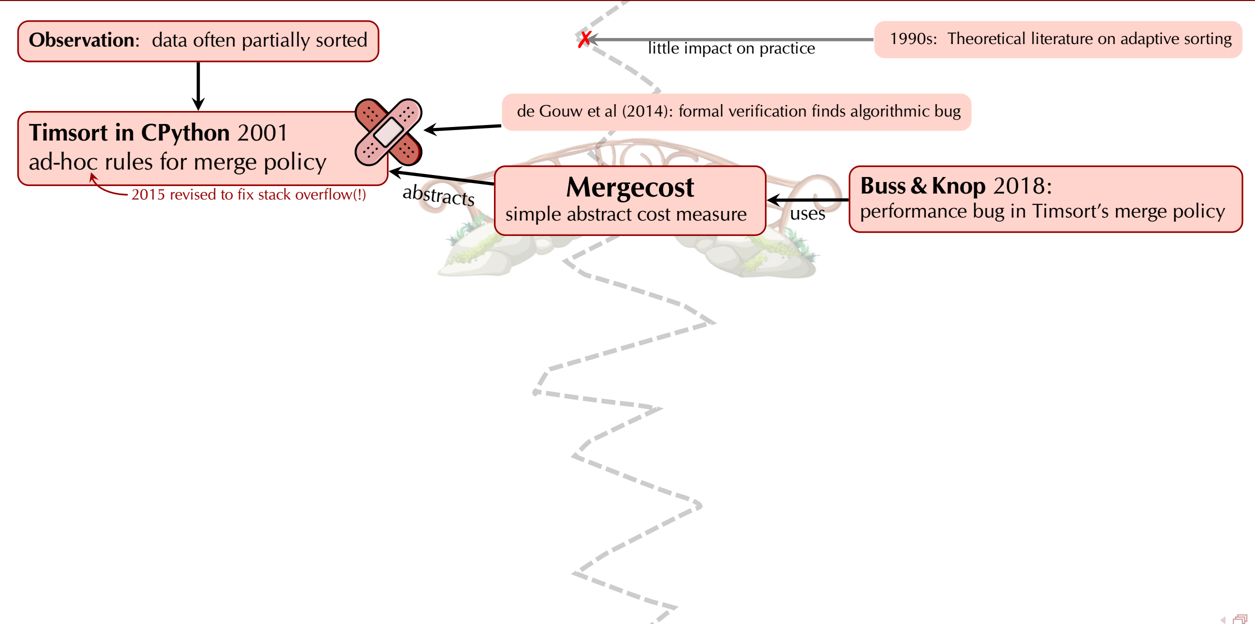
Powersort: An Algorithm Science Success Story



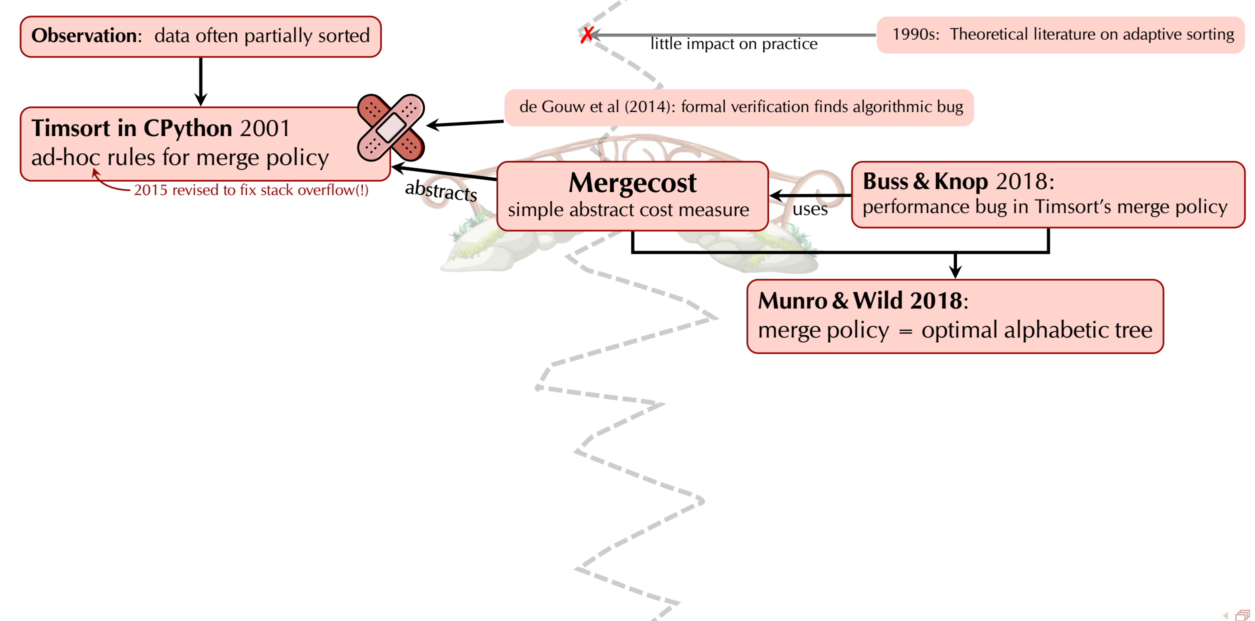
Powersort: An Algorithm Science Success Story



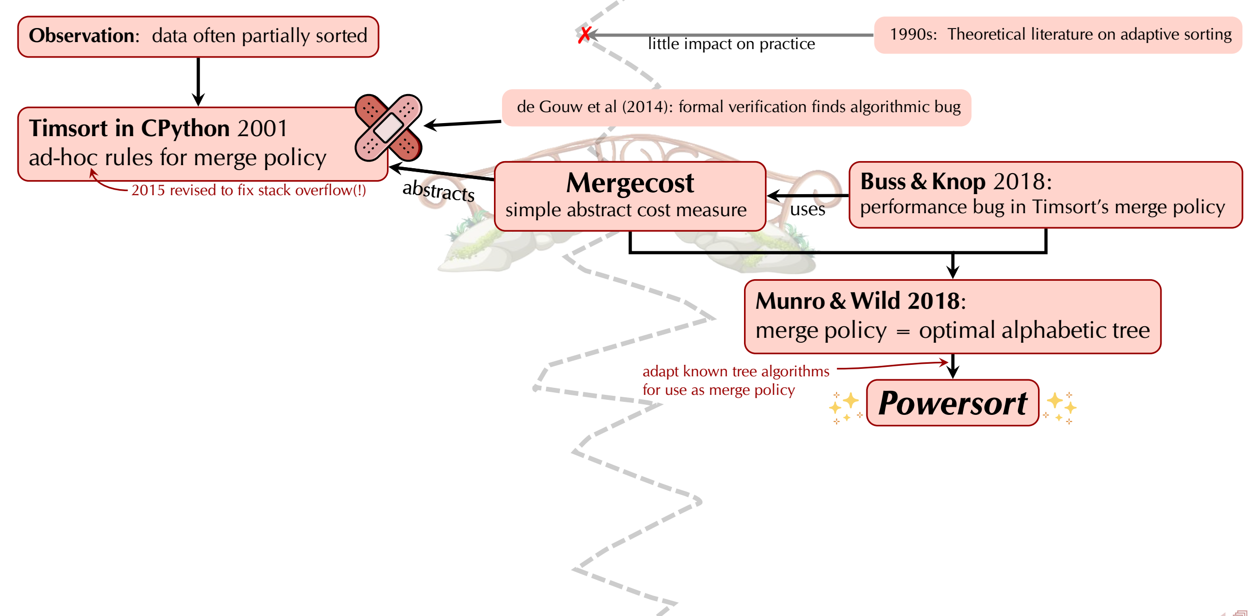
Powersort: An Algorithm Science Success Story



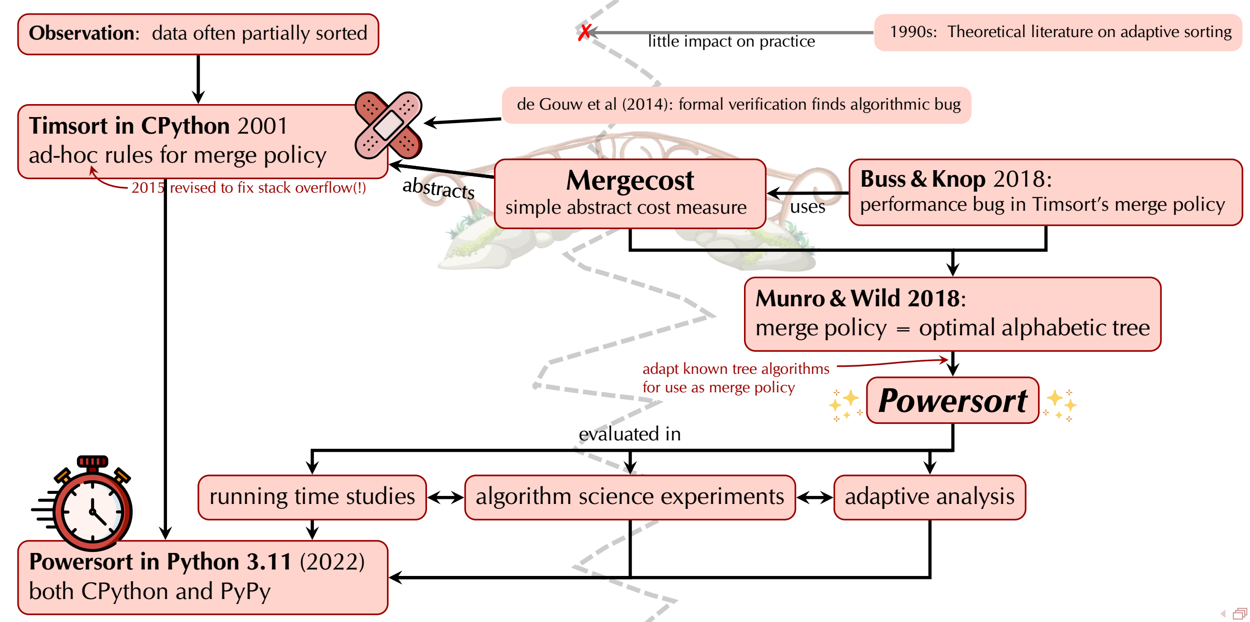
Powersort: An Algorithm Science Success Story



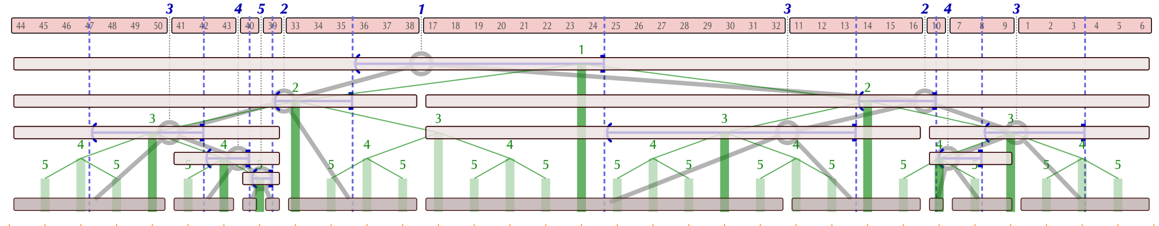
Powersort: An Algorithm Science Success Story



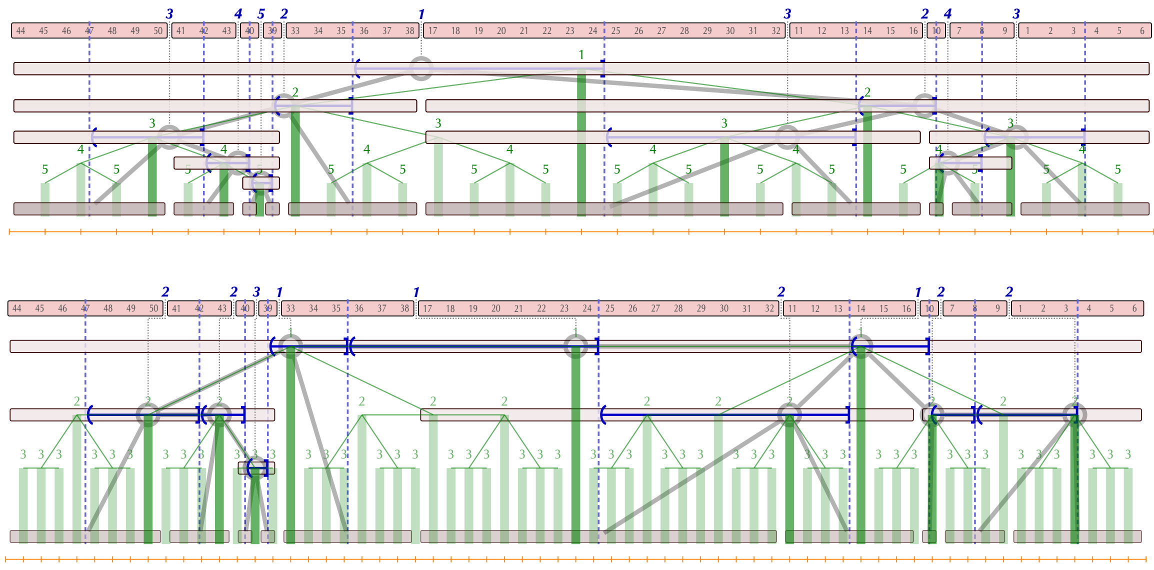
Powersort: An Algorithm Science Success Story

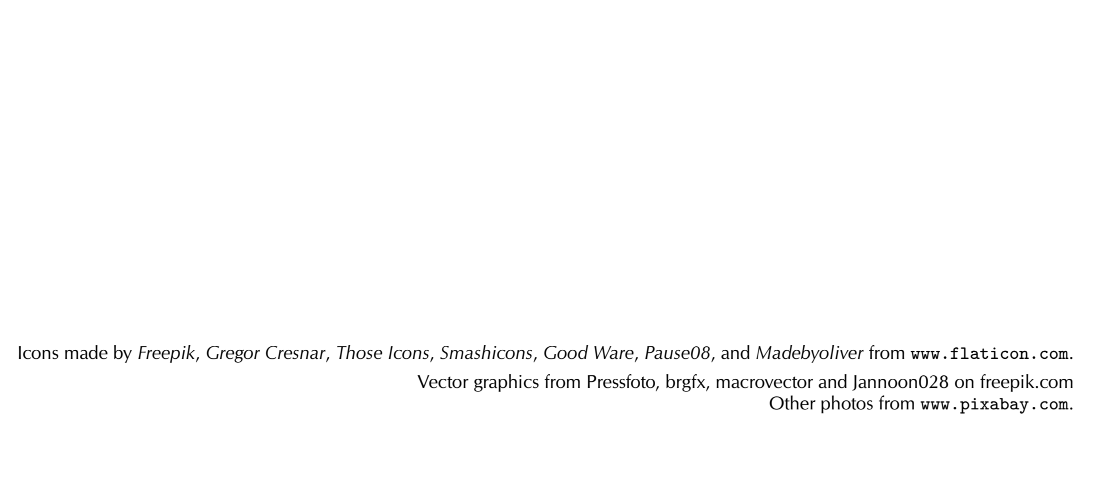


4-way Powersort



4-way Powersort





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

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