

Quicksort, Timsort, Powersort

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

Sebastian Wild

www.wild-inter.net

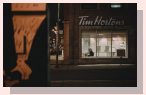
UNIVERSITY OF LIVERPOOL

PyCon US 2023

Outline



1 Sort of a list



2 Timsort



3 Beware, Stackoverflow!



4 Merge policies



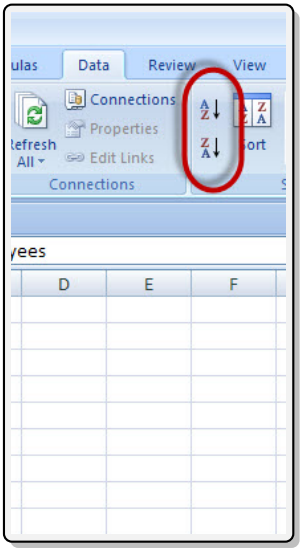
5 Powersort

1 Sort of a list



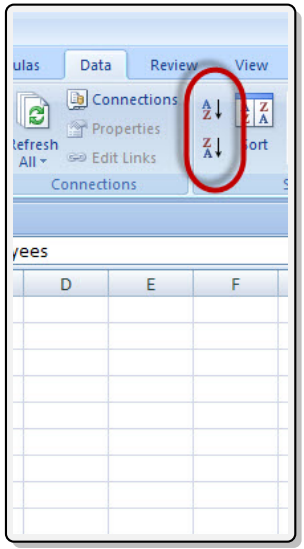
Sorting

- 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, ...)
  - ...



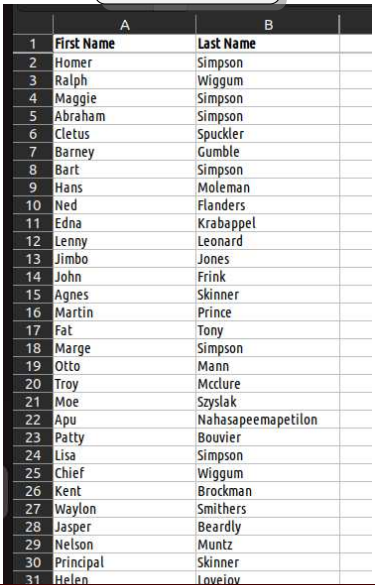
Sorting

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

Stable Sorting

unsorted input

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

sorted by First Name

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

sorted by Last Name

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

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**

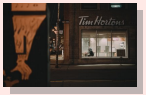
[P01] **Tim Peters et al.**: *listsort.txt*, CPython sources

[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



Sorting Trade-offs

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

Sorting Trade-offs

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

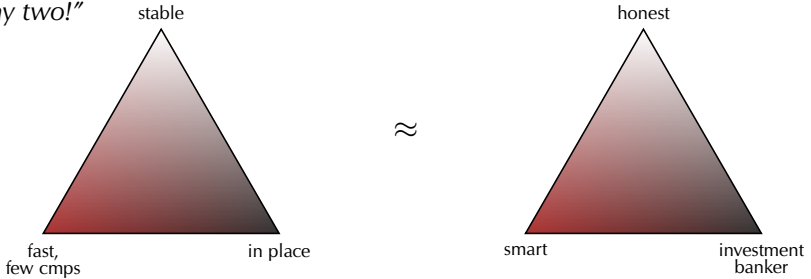
Sorting Trade-offs

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

Sorting Trade-offs

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

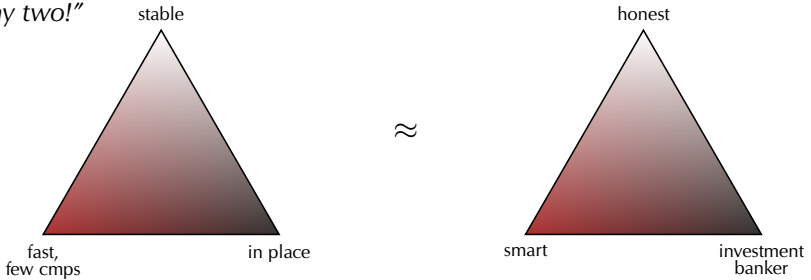
"Pick any two!"



Sorting Trade-offs

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

"Pick any two!"

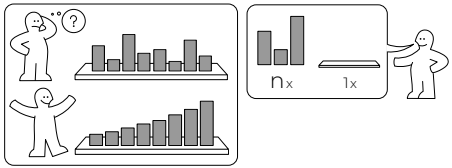


- Python has lots of objects and pointers anyways … "in-place" property least painful to sacrifice

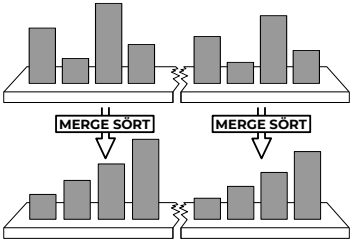
⇝ Mergesort!

MERGE SÖRT

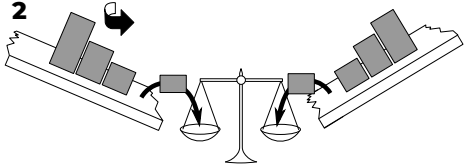
idea-instructions.com/merge-sort/
v1.2, CC by-nc-sa 4.0



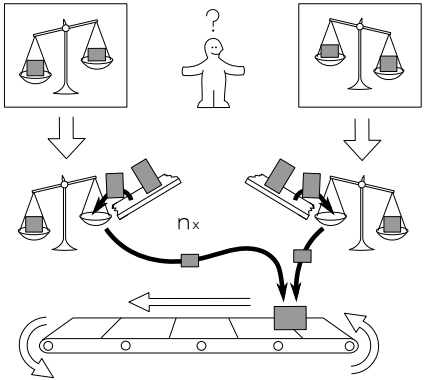
1



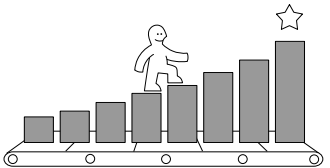
2



3



4



Mergesort++

Timsort

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



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

3. **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

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

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

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

Timsort merge policy (original)

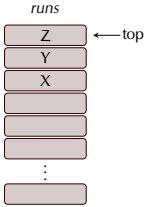
```
def timsort(lst):
    i = 0; runs = []
    while i < len(lst):
        j = extend_run(lst, i)
        runs.append((i,j-i)); i = j
        while Rule A/B/C applicable
            merge X,Y resp. Y,Z
    while len(runs) > 1:
        merge Y,Z
```

full code:
`tiny.cc/timsort`

Timsort merge policy (original)

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

full code:
`tiny.cc/timsort`



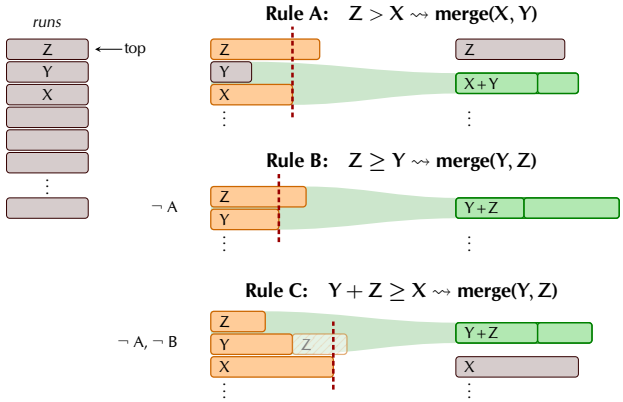
- `extend_run` detects next run
  & boosts to `minrun` if necessary

Timsort merge policy (original)

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

full code:
`tiny.cc/timsort`

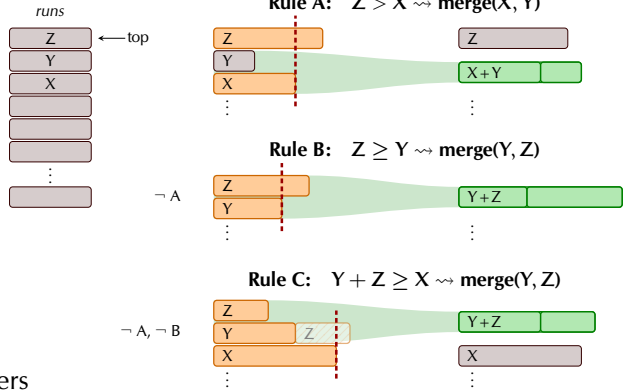
- `extend_run` detects next run & boosts to `minrun` if necessary



Timsort merge policy (original)

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

full code:
`tiny.cc/timsort`



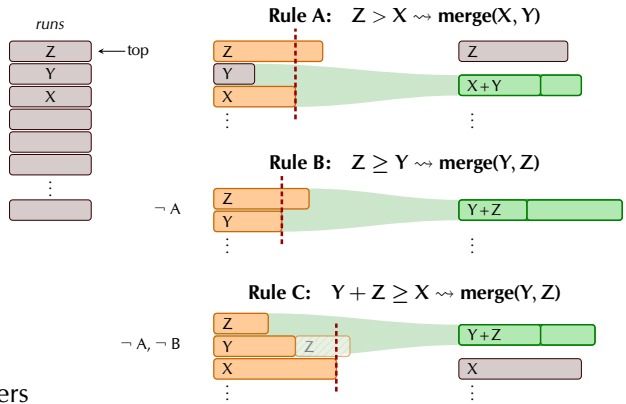
- `extend_run` detects next run & boosts to `minrun` if necessary
- **Goal:** runs on stack grow like Fibonacci numbers
  - Invariant: $\forall j:$ `runs[j]` $\geqslant$ `runs[j+1]` $+$ `runs[j+2]`

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

Timsort merge policy (original)

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

full code:
`tiny.cc/timsort`



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

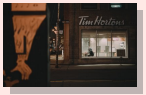
  $\rightsquigarrow$ max stack height $\approx \log_\phi(n)$
- Why exactly these rules?
  - Tim Peters: *"first thing I tried that 'worked well'"*

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

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

Invariant trouble

Recall this?

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

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



not true(!) for naughty pattern of run lengths

Invariant trouble

Recall this?

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

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



not true(!) for naughty pattern of run lengths



KeY project for formal verification in Java

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



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

Invariant trouble

Recall this?

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

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

not true(!) for naughty pattern of run lengths

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)

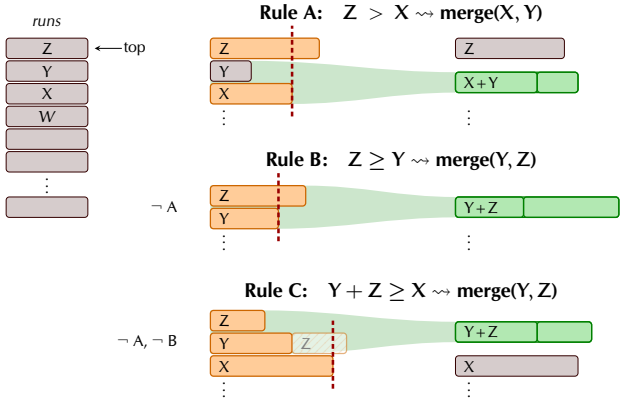
Timsort merge policy (patched)

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

Timsort merge policy (patched)

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

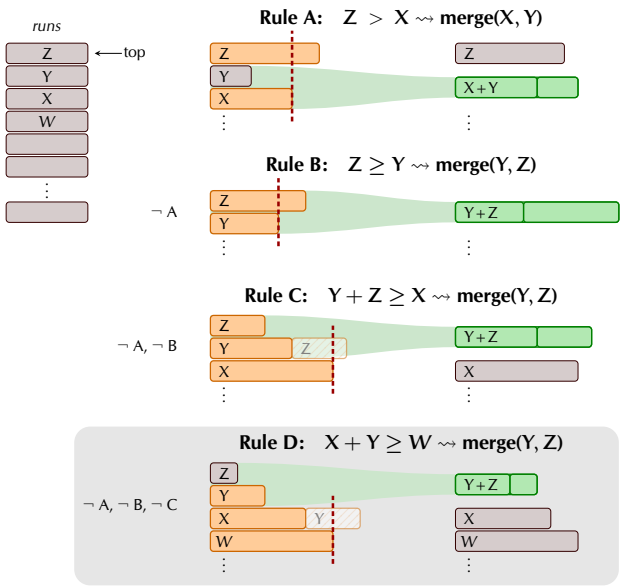
- Need to add a **Rule D**



Timsort merge policy (patched)

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

- Need to add a **Rule D**

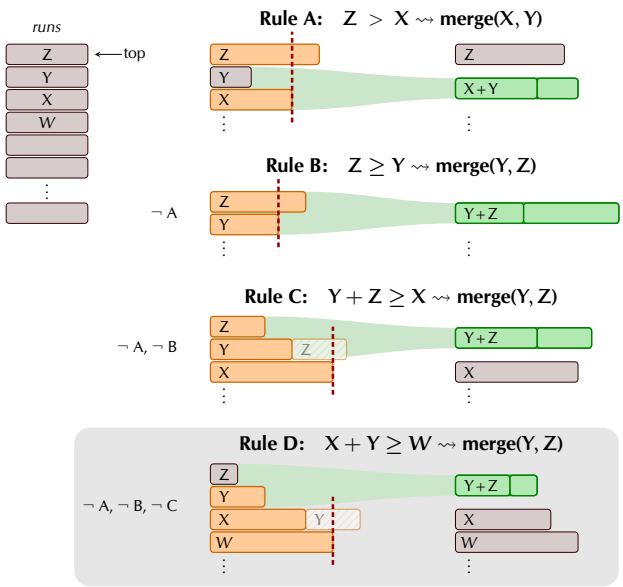


Timsort merge policy (patched)

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

- Need to add a **Rule D**

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



Timsort merge policy (patched)

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

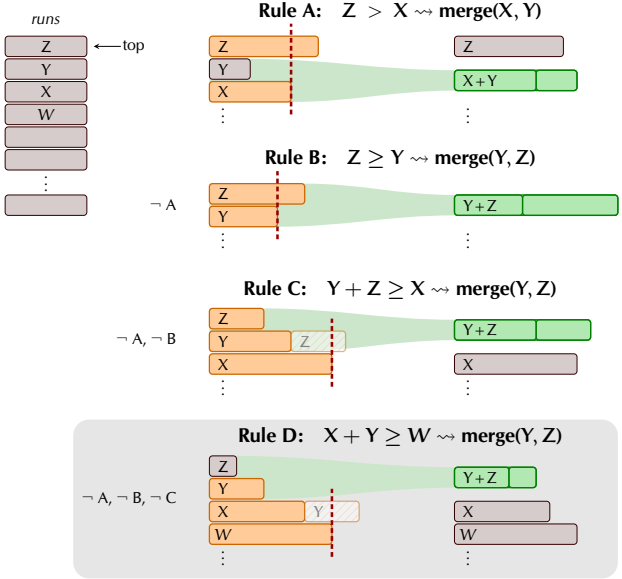
- Need to add a **Rule D**

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

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



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

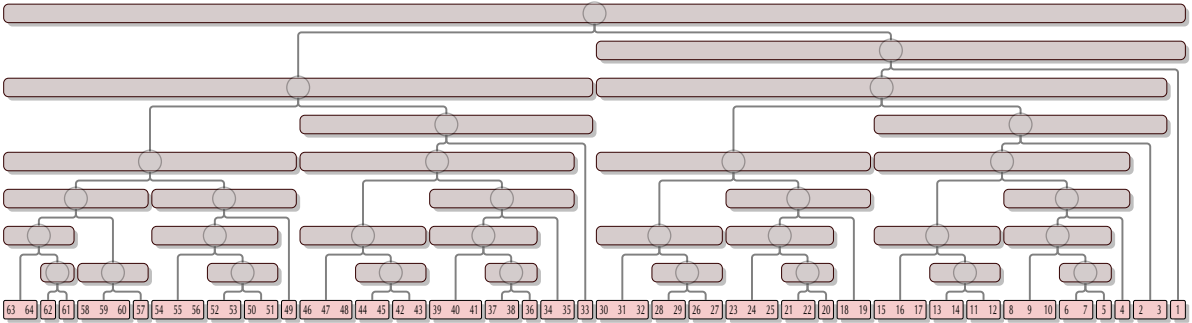


Timsort bad case

- Timsort's merge policy mostly works OK

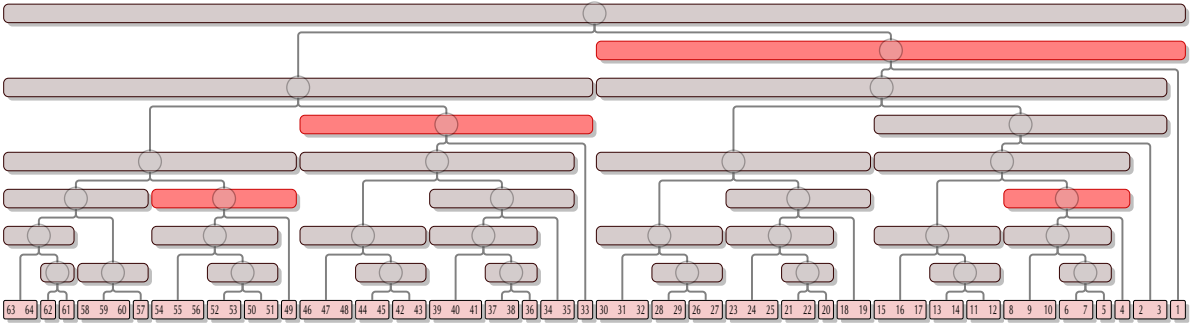
Timsort bad case

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



Timsort bad case

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

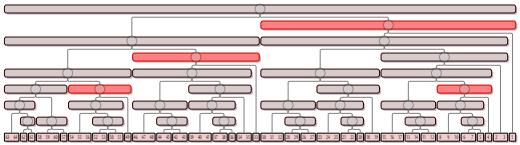


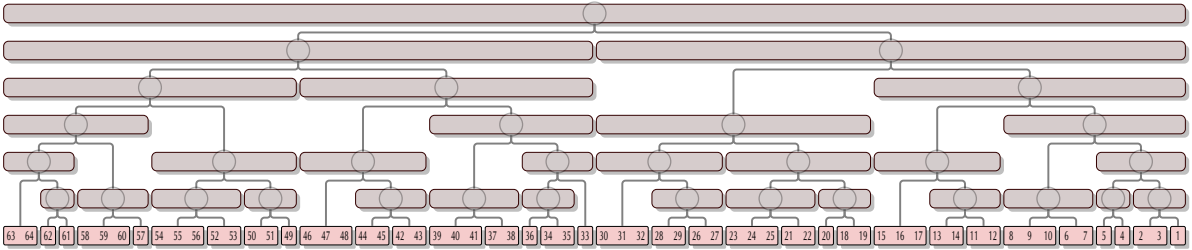
- intuitive problem: regularly very unbalanced merges

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

Timsort bad case

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



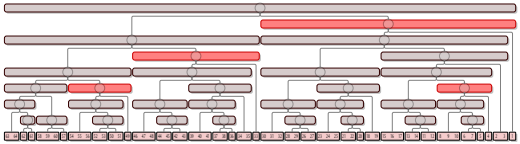


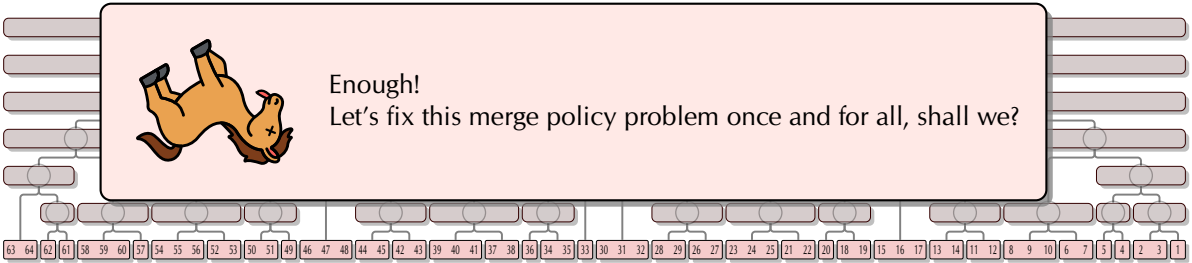
- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

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

Timsort bad case

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





Enough!
Let's fix this merge policy problem once and for all, shall we?

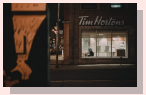
- intuitive problem: regularly very unbalanced merges
- can do much better (merge cost 321 vs. 371)
- As n increases, 50% higher merge cost than standard mergesort

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

Outline



1 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

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:

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**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

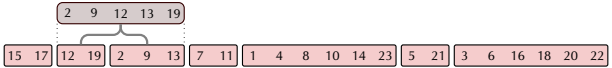
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

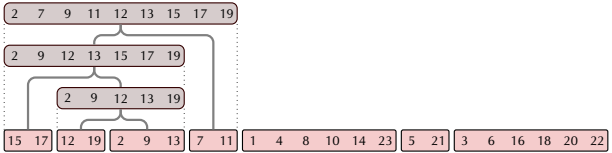
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

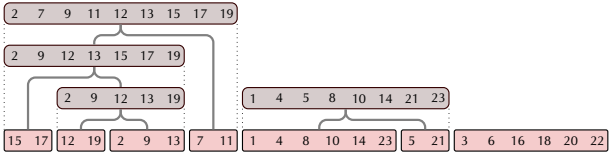
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

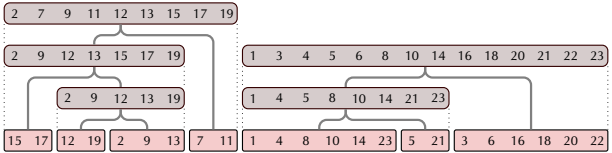
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

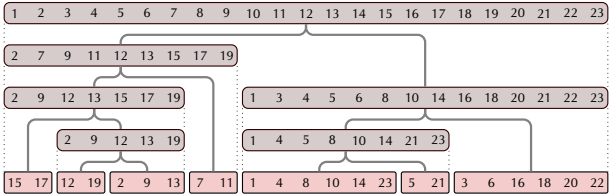
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

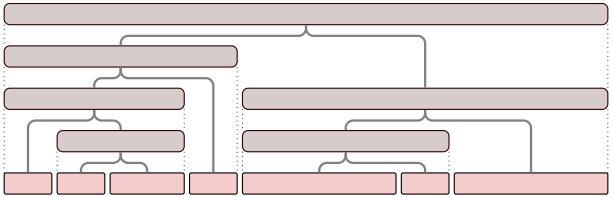
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



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

Merge policies from first principles

Run-adaptive mergesort

interleaved in code

Conceptually two steps:

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

determined by **merge policy**

Here:

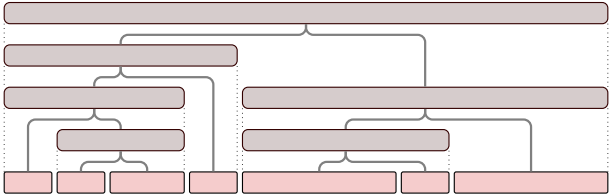
- only binary merges

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

- only stable sorts

⇝ *merge 2 **adjacent** runs*

⇝ Merge order = **merge tree**:



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

Merge costs

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

Merge policies from first principles

Run-adapti

interlea

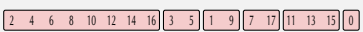
Conceptua

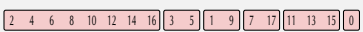
1. *Find r*
2. *Merge*

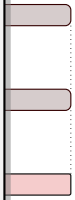
Here:

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

Different merge policies yield different merge costs!







tree

Merge costs

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

Merge policies from first principles

Run-adapti

interlea

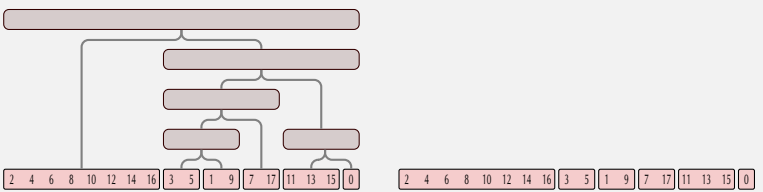
Conceptua

1. *Find r*
2. *Merge*

Here:

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

Different merge policies yield different merge costs!



tree

Merge costs

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

Merge policies from first principles

Run-adapti

interlea

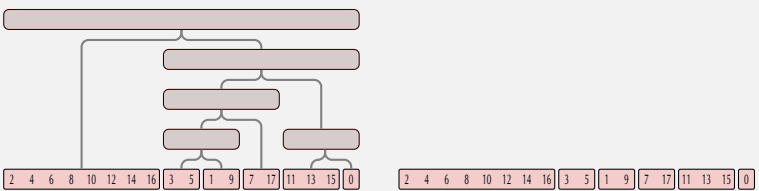
Conceptua

1. *Find r*
2. *Merge*

Here:

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

Different merge policies yield different merge costs!



tree

Merge costs

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

Merge policies from first principles

Run-adapti

interlea

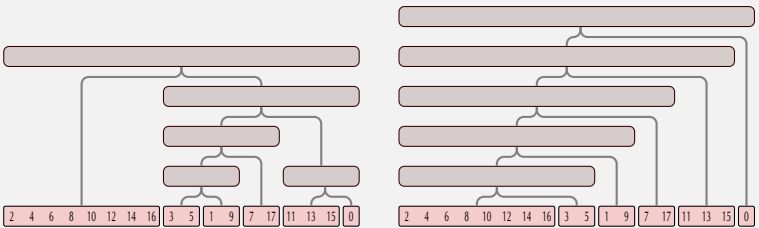
Conceptua

1. *Find r*
2. *Merge*

Here:

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

Different merge policies yield different merge costs!



tree

Merge costs

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

Merge policies from first principles

Run-adapti

interlea

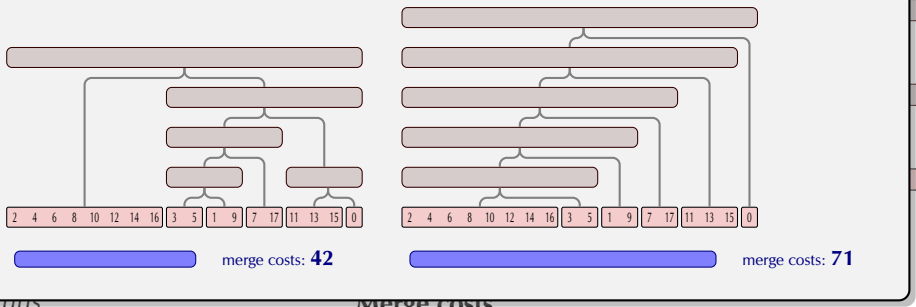
Conceptua

1. *Find r*
2. *Merge*

Here:

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

Different merge policies yield different merge costs!

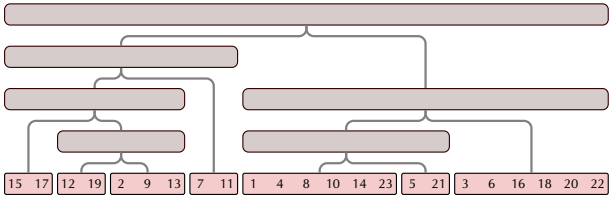


tree

Merge costs

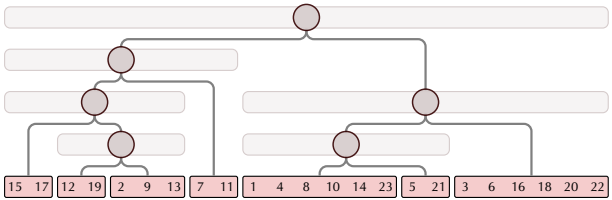
- cost of merge := size of output
  - ≈ memory transfers
  - ⩾ #cmps
- total cost = total area of

Mergesort meets Binary Search Trees



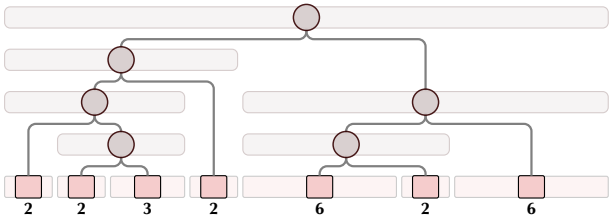
Merge cost = total area of 

Mergesort meets Binary Search Trees



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

Mergesort meets Binary Search Trees



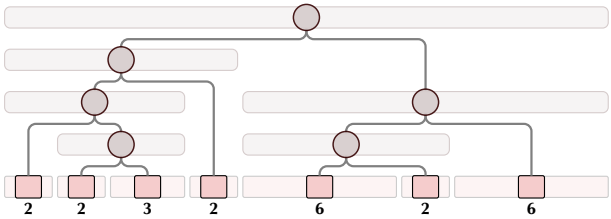
Merge cost $=$ total area of 

$=$ total length of paths to all array entries

$=$ weighted external path length

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

Mergesort meets Binary Search Trees

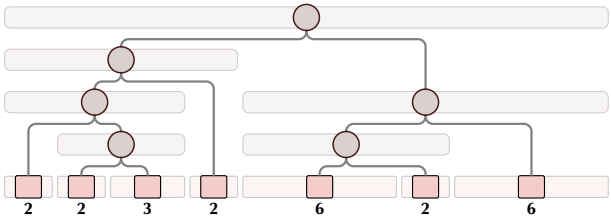


Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

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

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

Mergesort meets Binary Search Trees



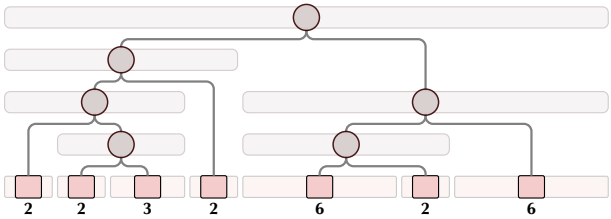
Merge cost = total area of 
= total length of paths to all array entries
= weighted external path length

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

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

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

Mergesort meets Binary Search Trees



How to compute good merge trees?

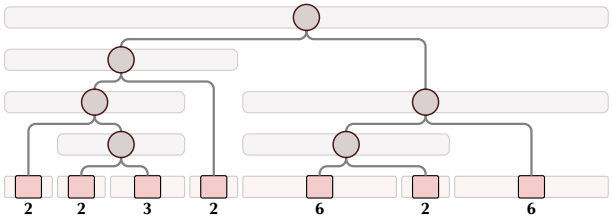
Merge cost = total area of
= total length of paths to all array entries
= weighted external path length

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

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

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

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge trees?

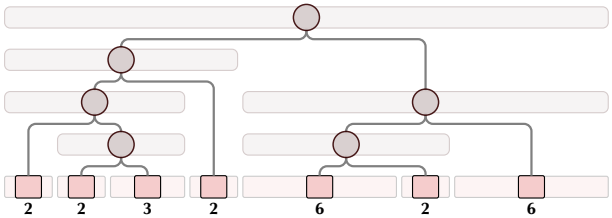


. . . 70s are calling

nearly-optimal BST merge

- simple (greedy) linear-time methods!

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge trees?

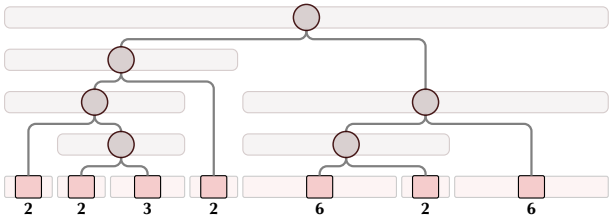


. . . 70s are calling

nearly-optimal BST merge

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

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge trees?



. . . 70s are calling

nearly-optimal BST merge

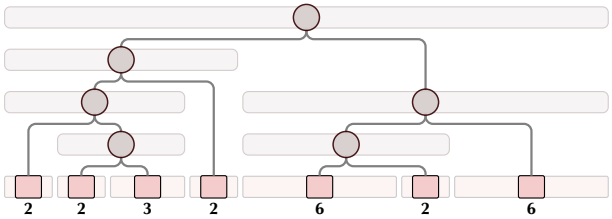
- simple (greedy) linear-time methods!
- almost optimal ($\leqslant \mathcal{H} + 2$)

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



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

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge trees?



. . . 70s are calling

nearly-optimal BST merge

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

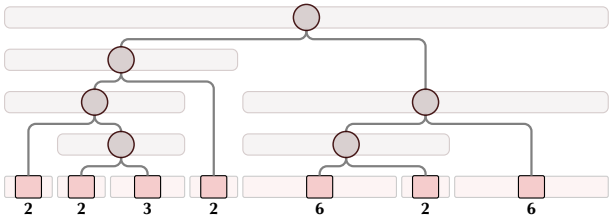
⇝ Powersort based on the **bisection method**



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

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

Mergesort meets Binary Search Trees



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

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

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

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

How to compute good merge trees?



. . . 70s are calling

nearly-optimal BST merge

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

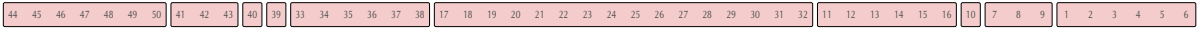
$\rightsquigarrow$ Powersort based on the **bisection method**



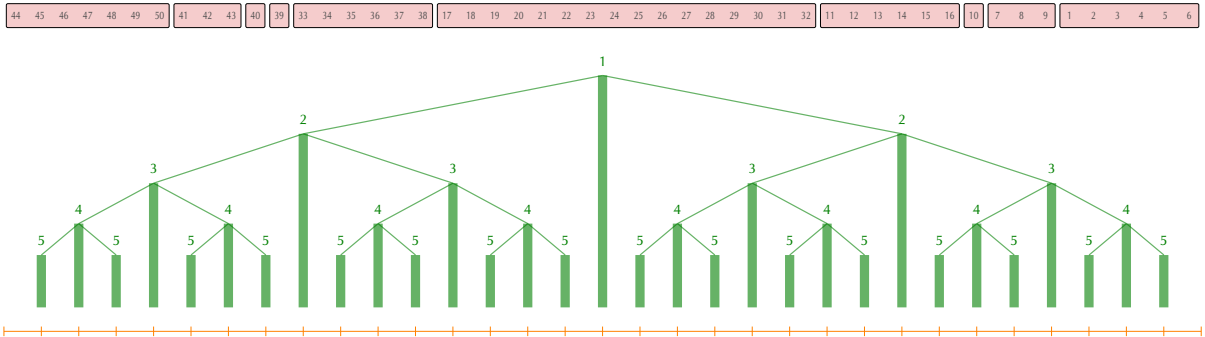
Mehlhorn: *A best possible bound for the weighted path length of binary search trees*, SIAM J Comp **1977**

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

Run-Boundary Powers

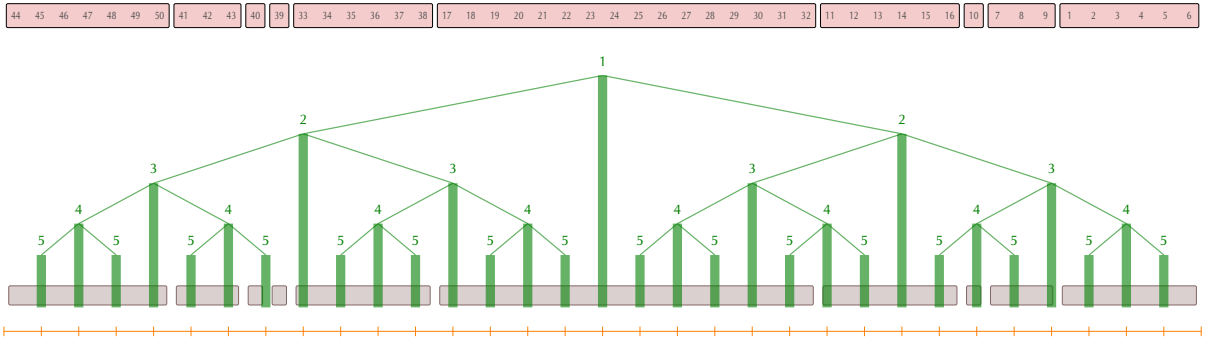


Run-Boundary Powers



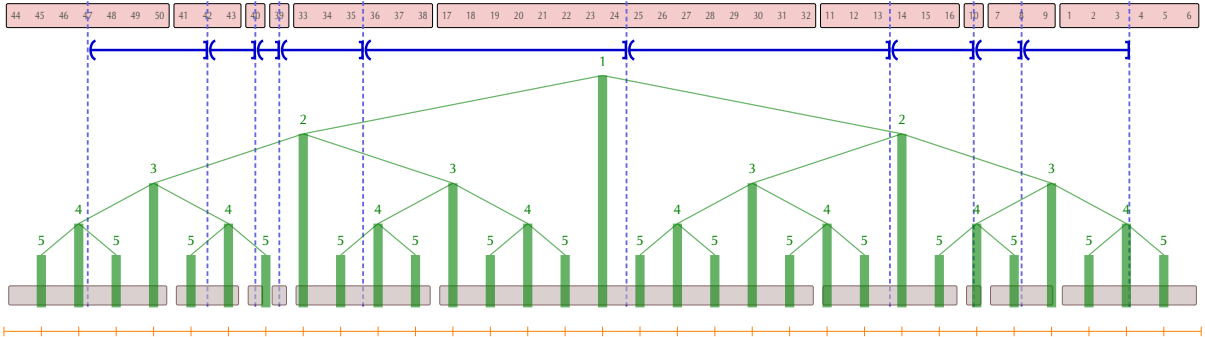
- (virtual) perfect balanced binary tree

Run-Boundary Powers



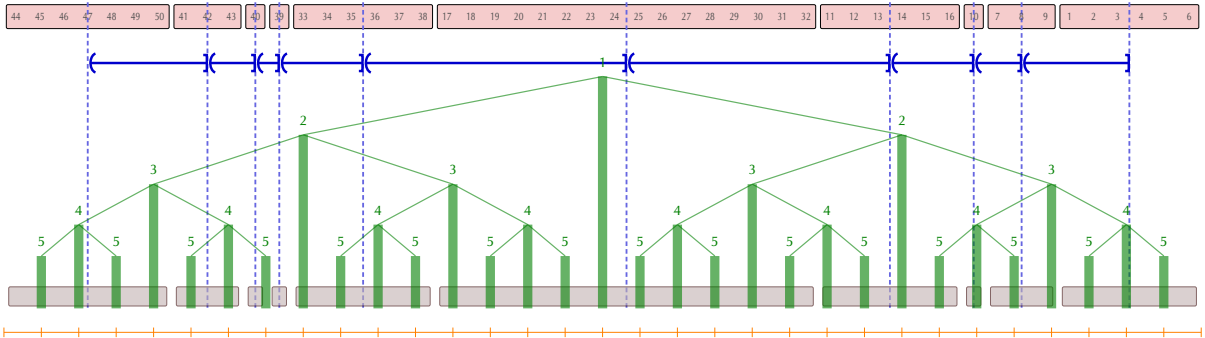
- (virtual) perfect balanced binary tree

Run-Boundary Powers



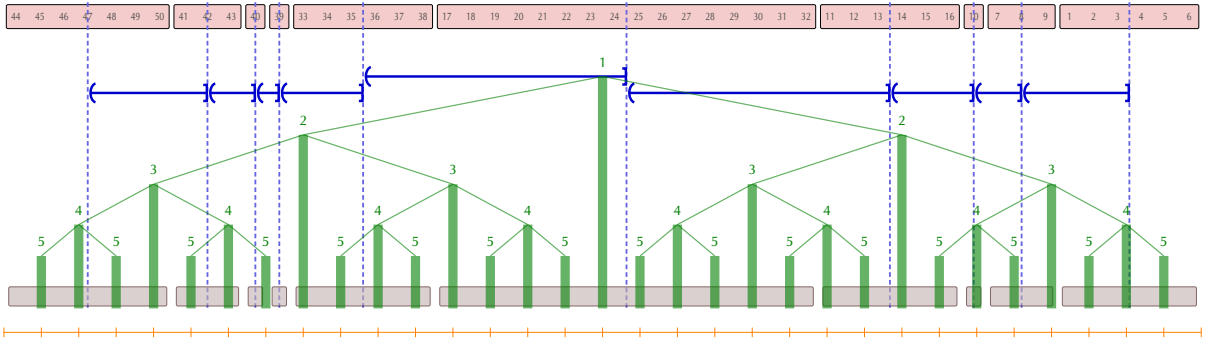
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



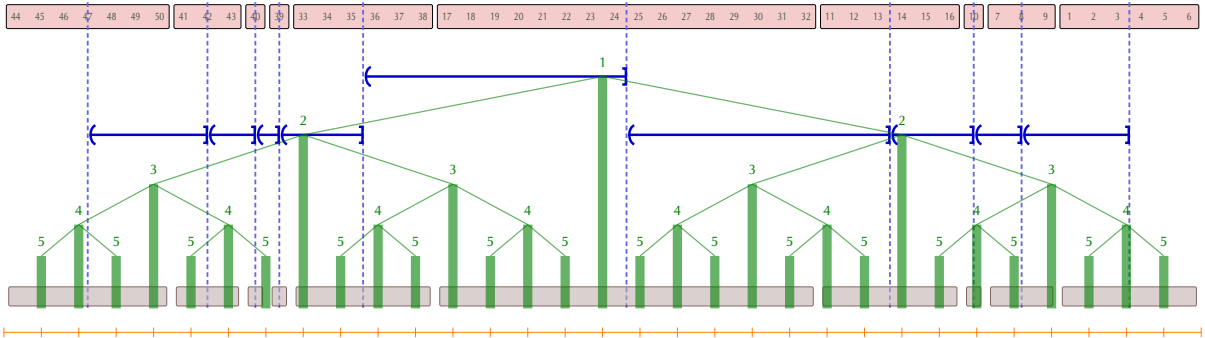
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



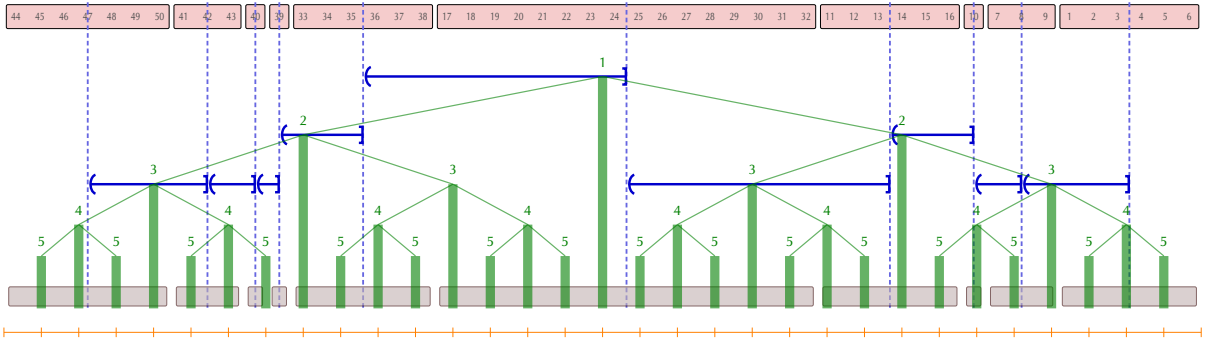
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



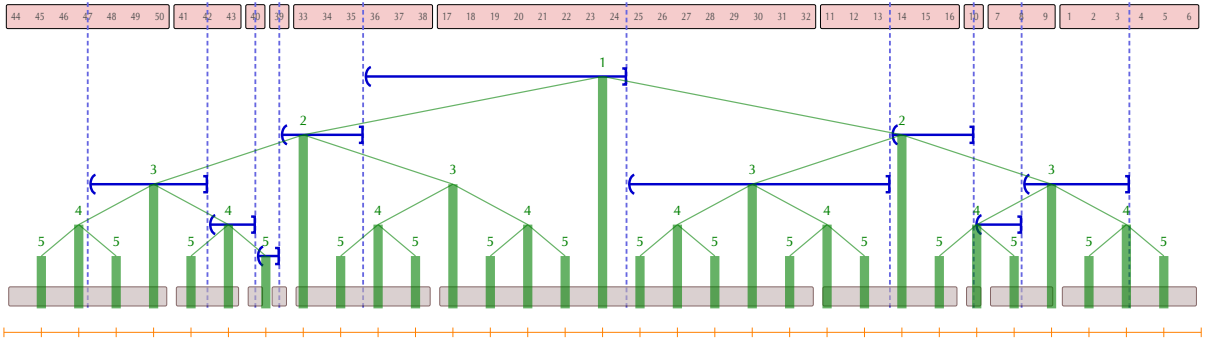
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



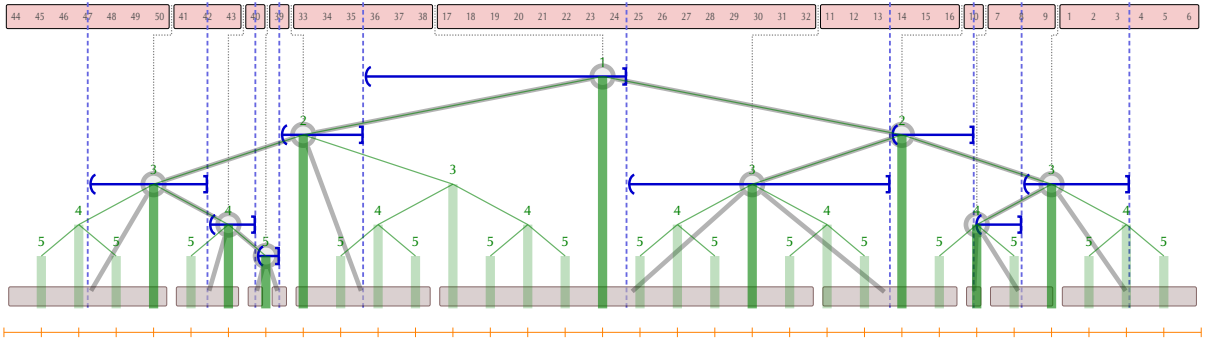
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



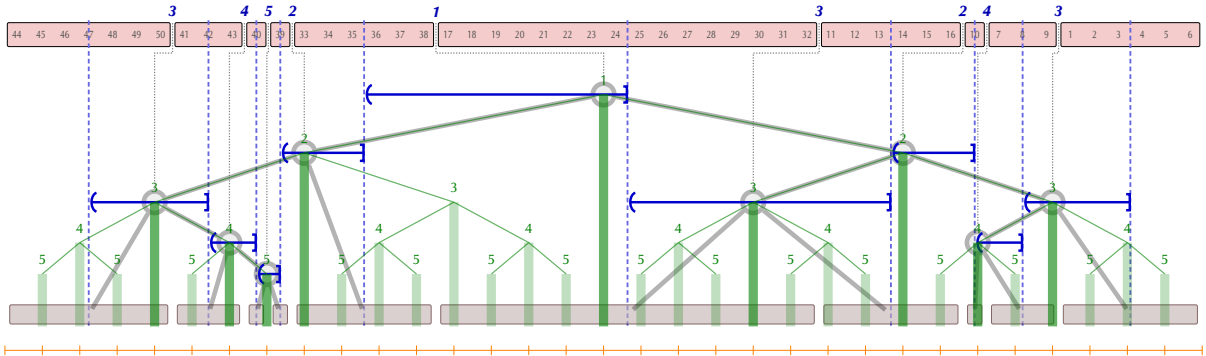
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



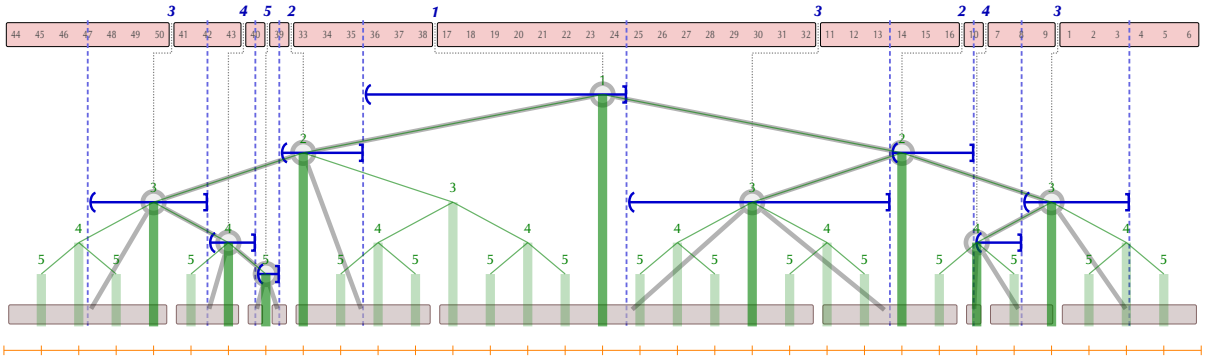
- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node

Run-Boundary Powers



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

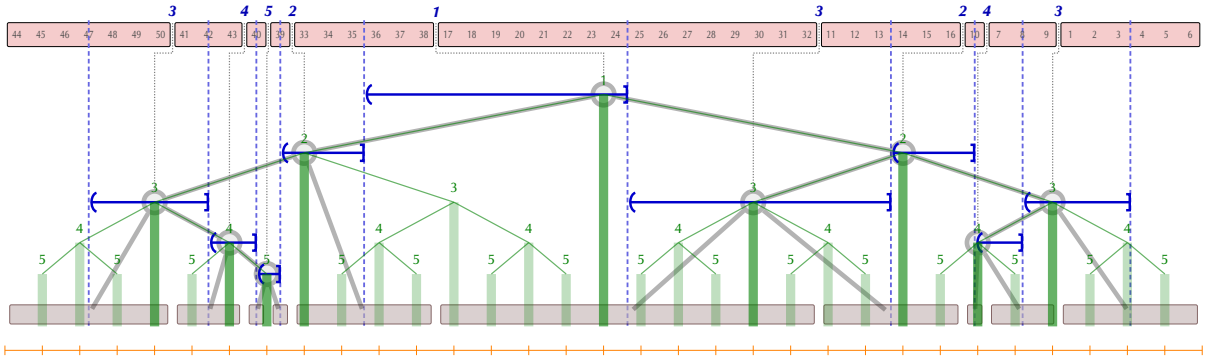
Run-Boundary Powers



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



Run-Boundary Powers

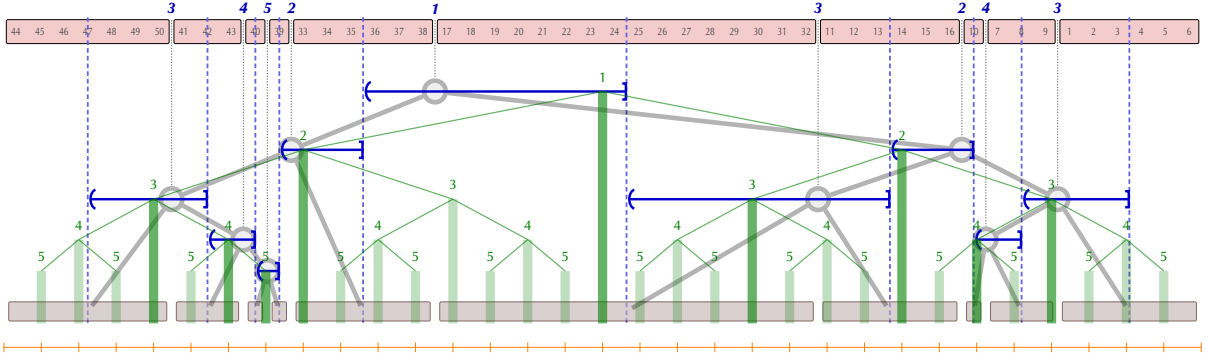


- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its **power**

⇝ merge tree follows virtual tree



Run-Boundary Powers

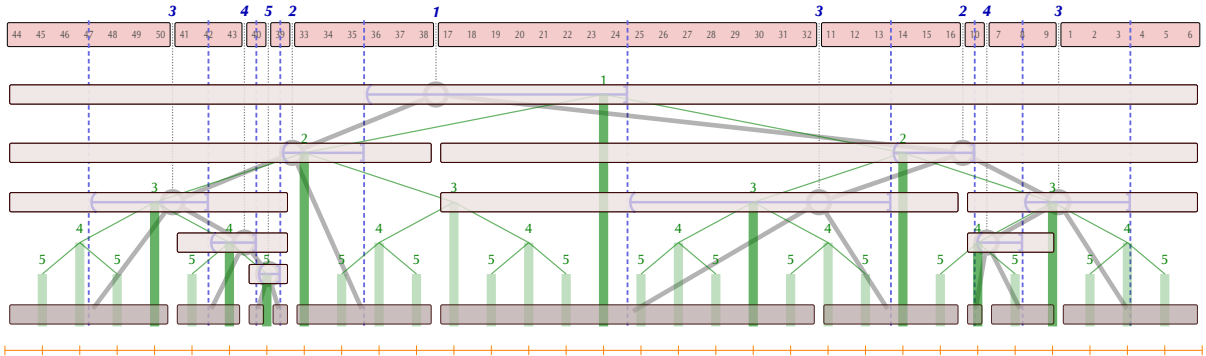


- (virtual) perfect balanced binary tree
- midpoint intervals "snap" to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its **power**

⇝ merge tree follows virtual tree



Run-Boundary Powers

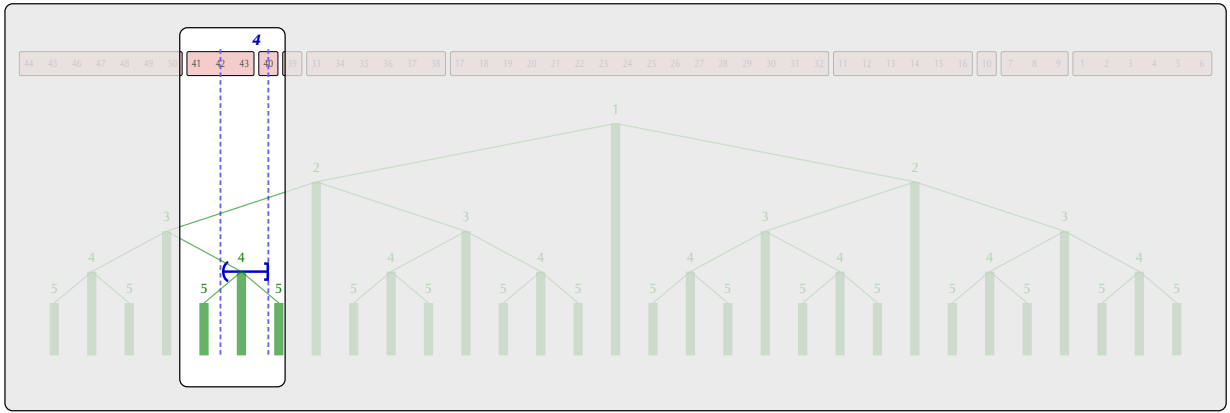


- (virtual) perfect balanced binary tree
- midpoint intervals “snap” to closest virtual tree node
  - ⇝ assigns each run boundary a depth = its **power**

⇝ merge tree follows virtual tree



Run-Boundary Powers are Local

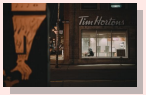


Computation of powers only depends on two adjacent runs.

Outline



1 Sort of a list



2 Timsort



3 Beware, Stackoverflow!



4 Merge policies



5 Powersort

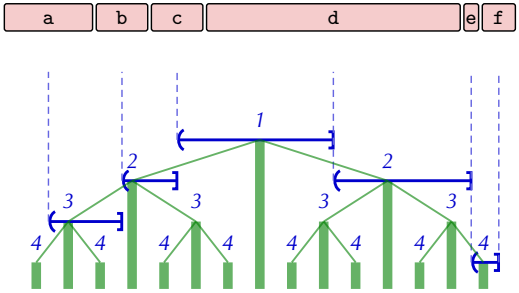


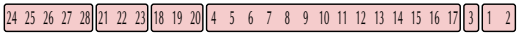
5 Powersort

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`

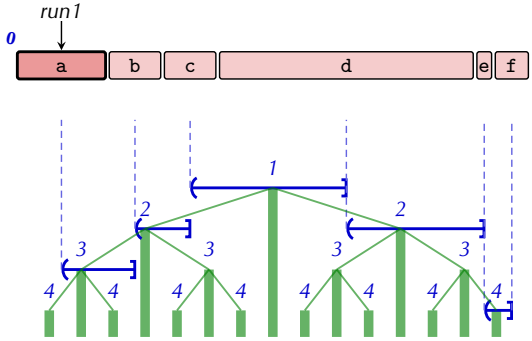




Powersort

```
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)
```

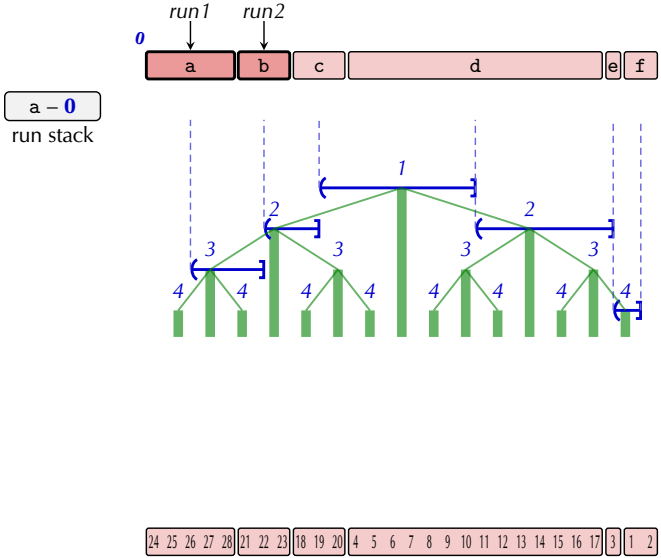
full code:
`tiny.cc/timsort`



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)
```

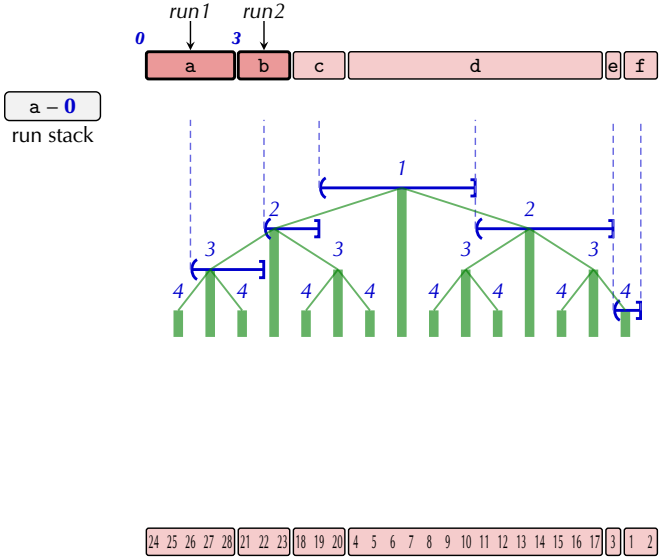
full code:
`tiny.cc/timsort`



Powersort

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

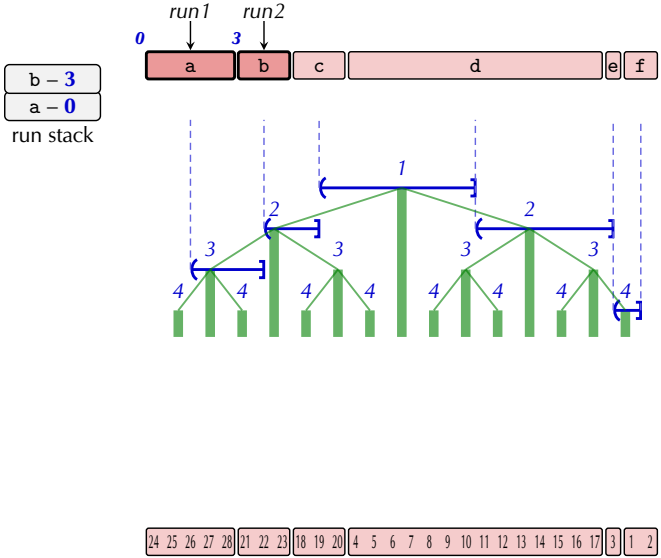
full code:
`tiny.cc/timsort`



Powersort

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

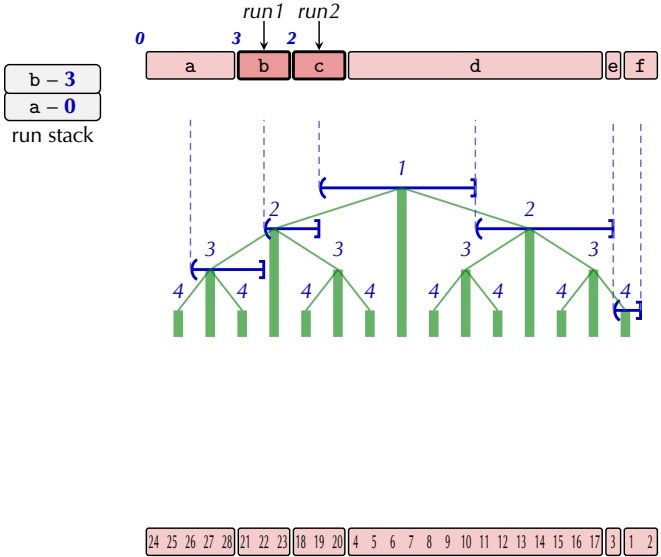
full code:
`tiny.cc/timsort`



Powersort

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

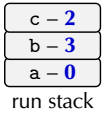
full code:
`tiny.cc/timsort`

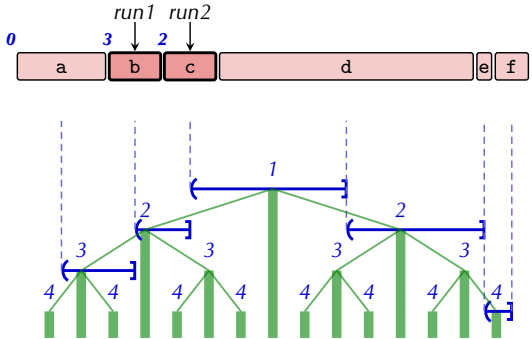


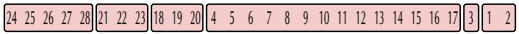
Powersort

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

full code:
`tiny.cc/timsort`



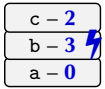




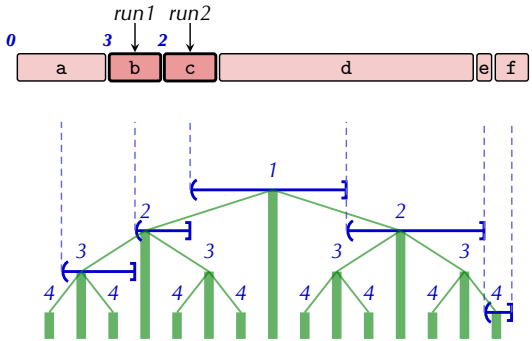
Powersort

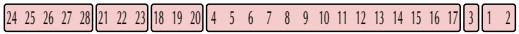
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

full code:
`tiny.cc/timsort`



run stack

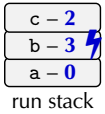


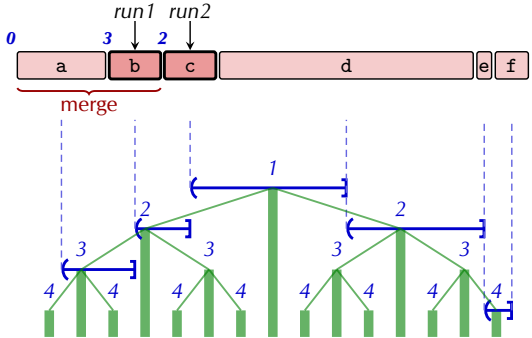


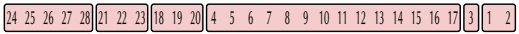
Powersort

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

full code:
`tiny.cc/timsort`



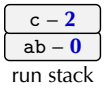


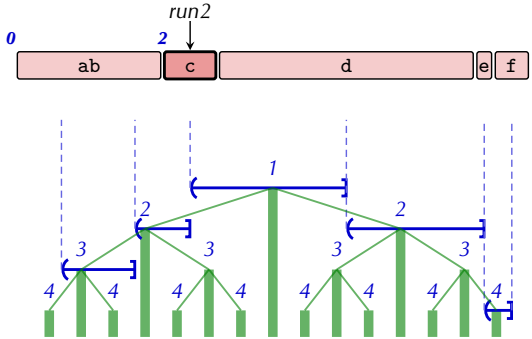


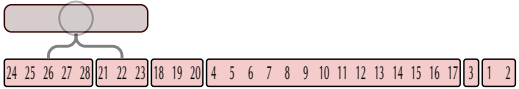
Powersort

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

full code:
`tiny.cc/timsort`



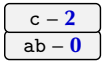


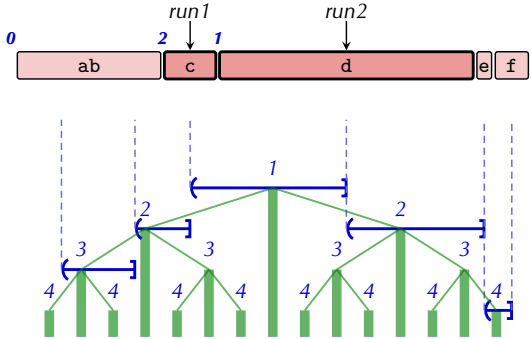


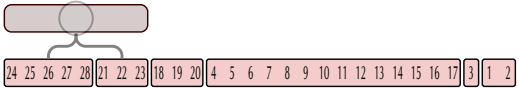
Powersort

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

full code:
`tiny.cc/timsort`



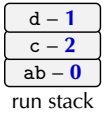




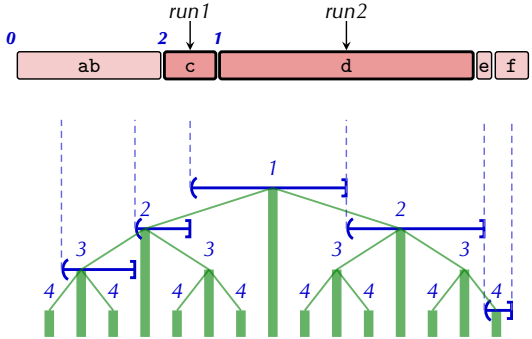
Powersort

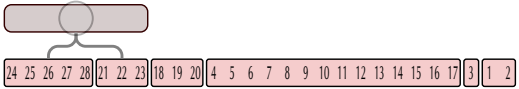
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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

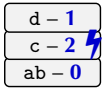




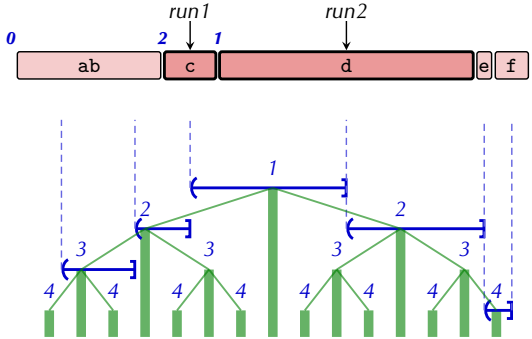
Powersort

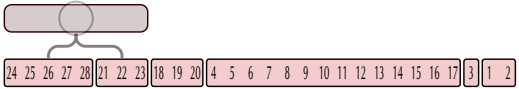
```
1  def powersort(lst)
2      i = 0; n = len(lst)
3      runs = [];
4      j = extend_run(lst, i)
5      runs.append((i,j-i,0)); i = j
6      while i < n:
7          j = extend_run(lst, i)
8          p = power(runs[-1], (i,j-i), n)
9          while p <= runs[-1][2]
10             merge_topmost_2(lst,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



run stack

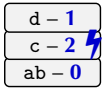


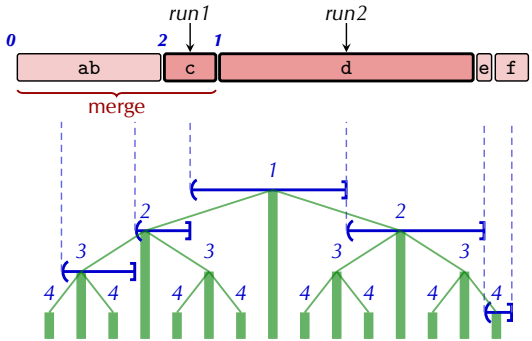


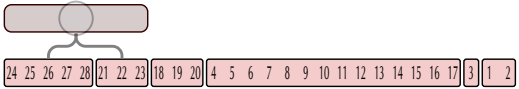
Powersort

```
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`



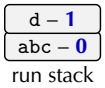


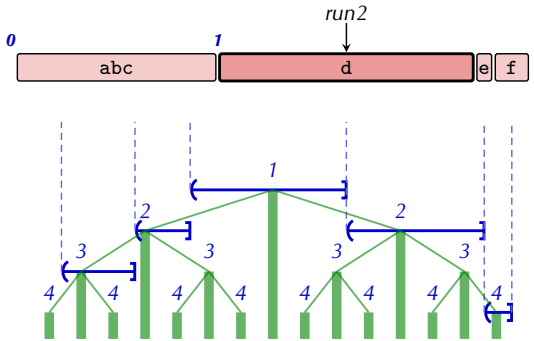


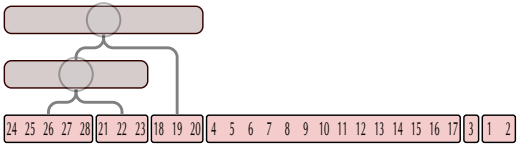
Powersort

```
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`



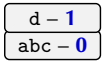


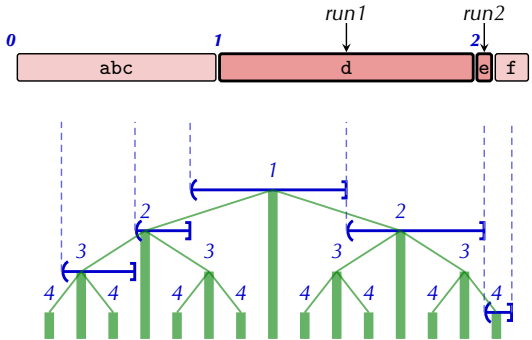


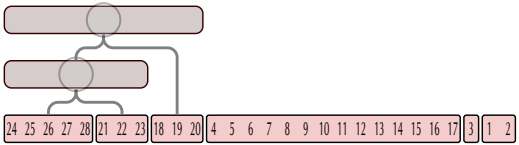
Powersort

```
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`



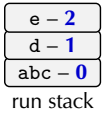


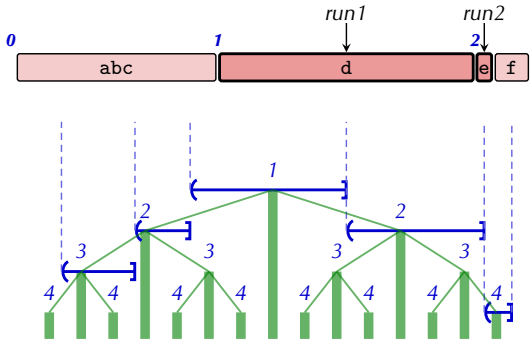


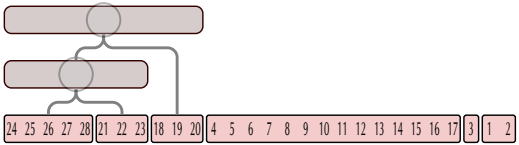
Powersort

```
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`



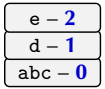


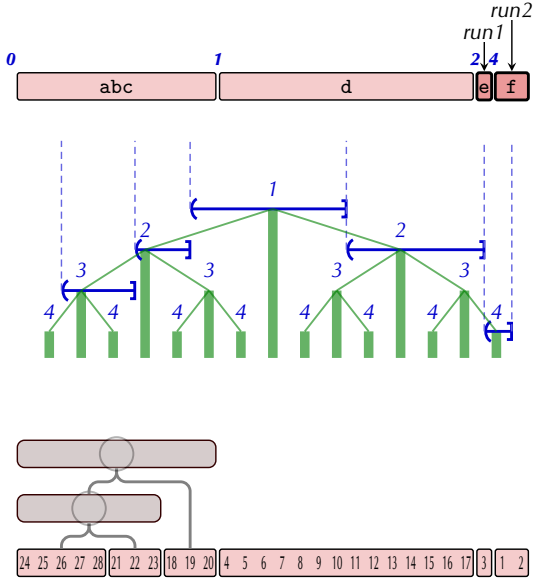


Powersort

```
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`

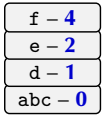




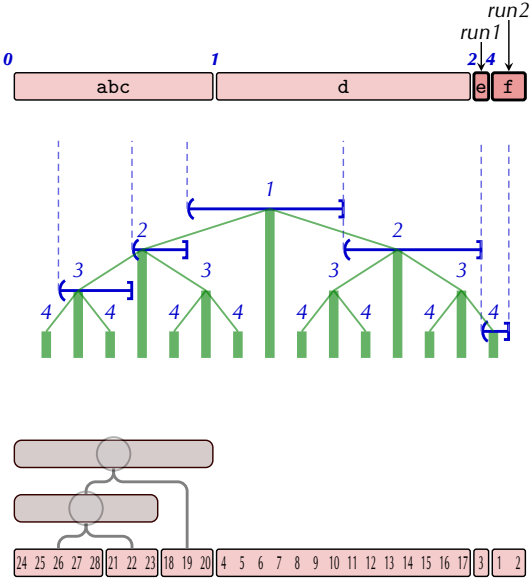
Powersort

```
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`



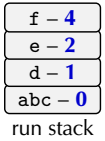
run stack



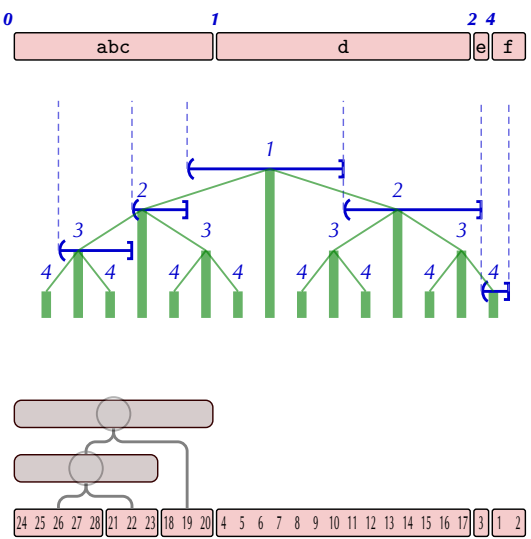
Powersort

```
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`



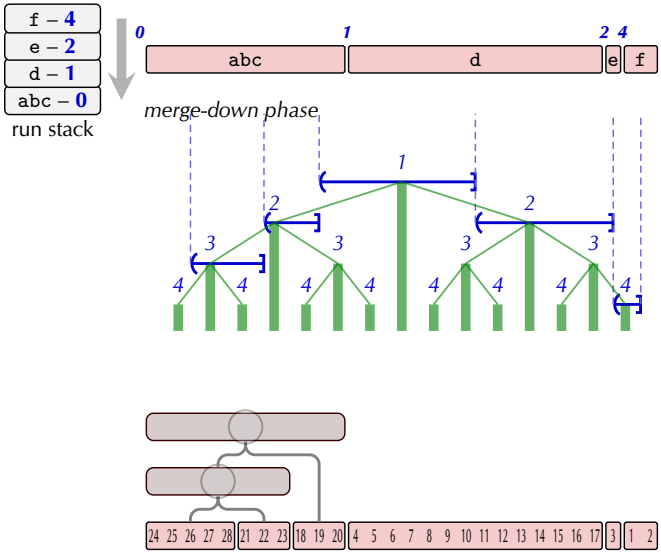
run stack



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)
```

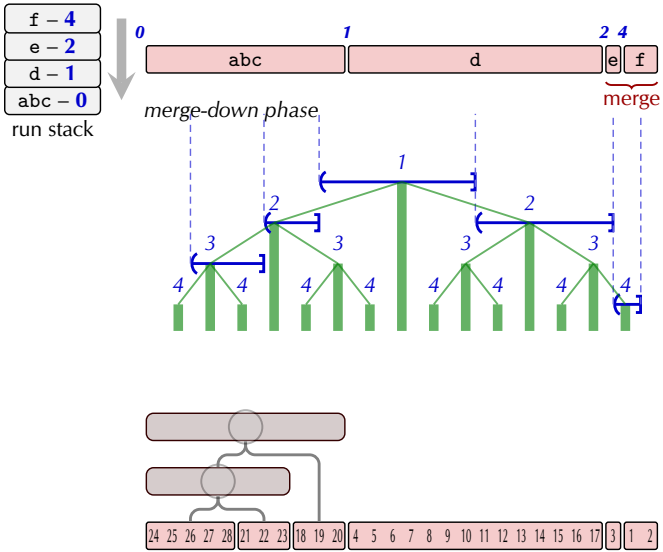
full code:
`tiny.cc/timsort`



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)
```

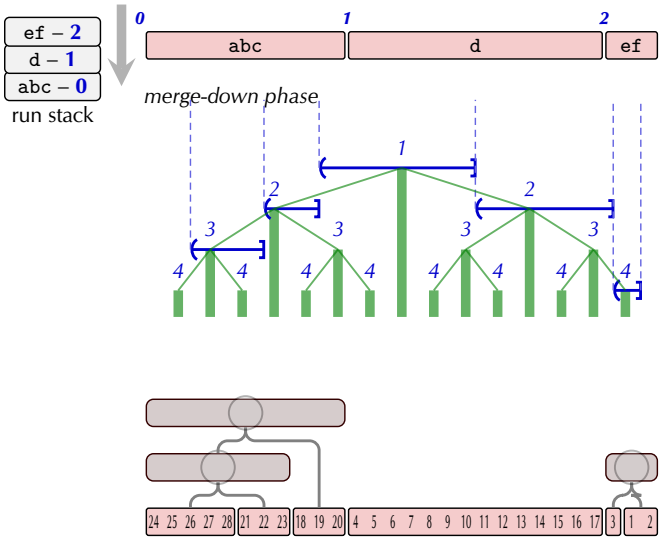
full code:
`tiny.cc/timsort`



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)
```

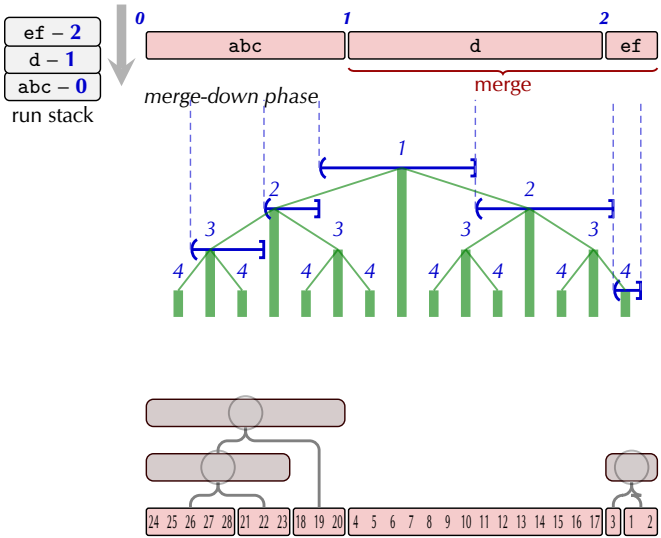
full code:
`tiny.cc/timsort`



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)
```

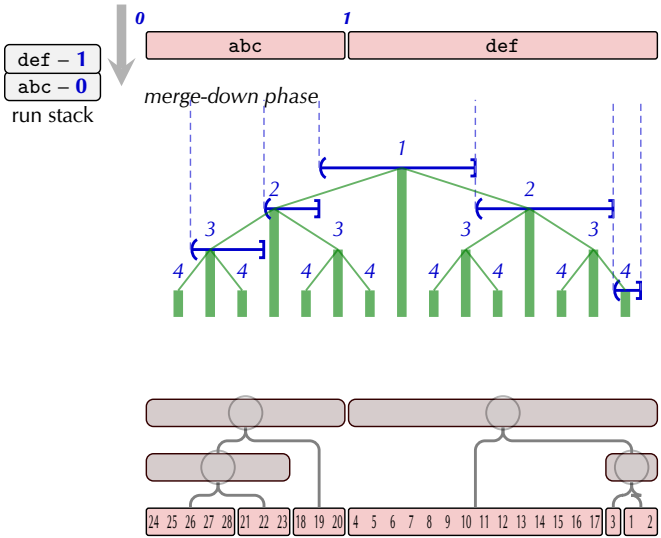
full code:
`tiny.cc/timsort`



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)
```

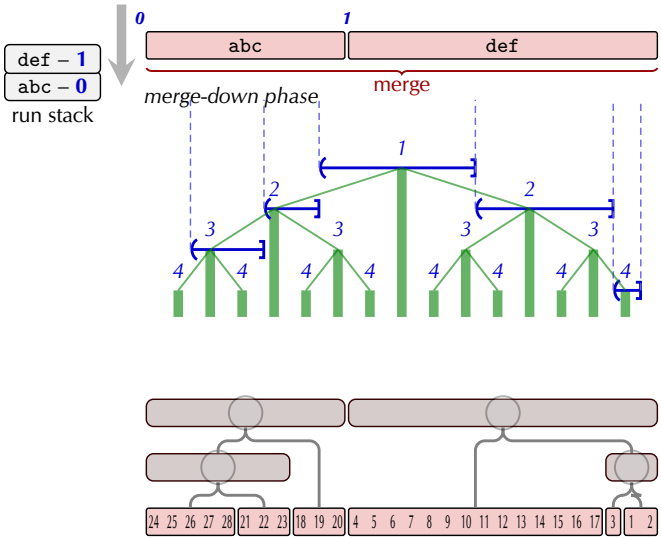
full code:
`tiny.cc/timsort`



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`

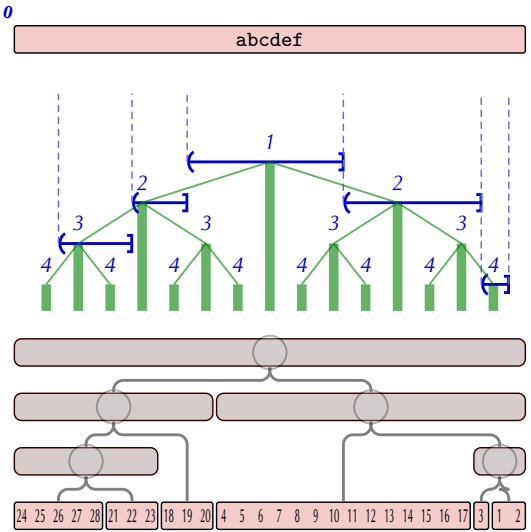


Powersort

```
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`

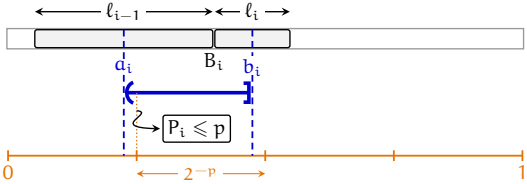
abcdef – **0**
run stack



Computing powers

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



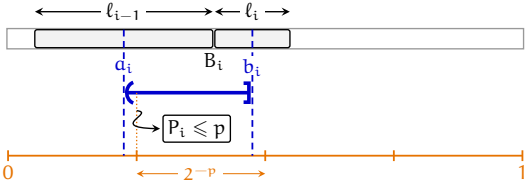



```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      a = (i1 + n1/2) / n
5      b = (i2 + n2/2) / n
6      l = 0
7      while math.floor(a * 2**l) == \
8            math.floor(b * 2**l):
9          l += 1
10     return l
```

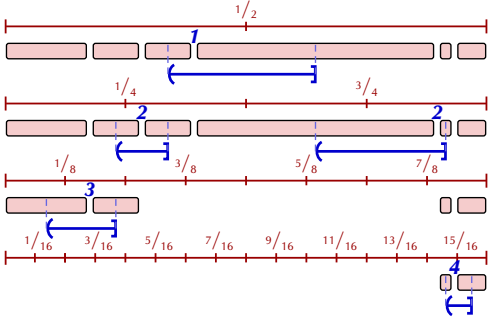
Computing powers

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





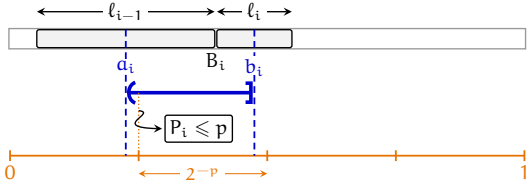

```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      a = (i1 + n1/2) / n
5      b = (i2 + n2/2) / n
6      l = 0
7      while math.floor(a * 2**l) == \
8            math.floor(b * 2**l):
9          l += 1
10     return l
```



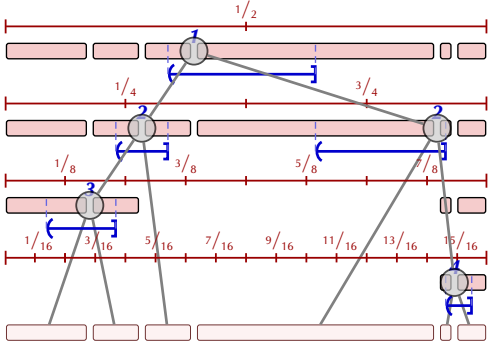
Computing powers

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





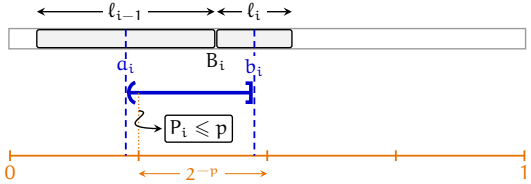

```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      a = (i1 + n1/2) / n
5      b = (i2 + n2/2) / n
6      l = 0
7      while math.floor(a * 2**l) == \
8            math.floor(b * 2**l):
9          l += 1
10     return l
```



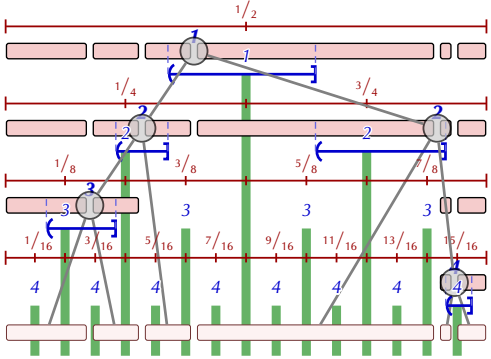
Computing powers

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





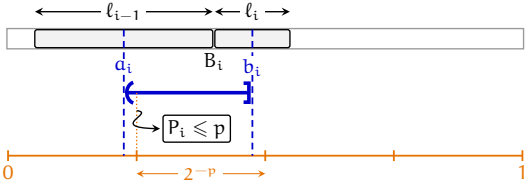

```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      a = (i1 + n1/2) / n
5      b = (i2 + n2/2) / n
6      l = 0
7      while math.floor(a * 2**l) == \
8            math.floor(b * 2**l):
9          l += 1
10     return l
```



Computing powers

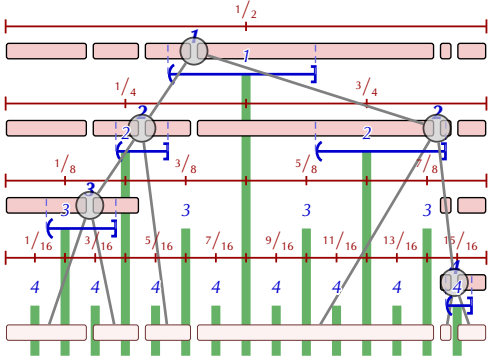
- Computing the power of (run boundary between) two runs
  - $\longleftrightarrow$ = normalized midpoint interval
  - power = min ℓ s.t. $\longleftrightarrow$ contains $c \cdot 2^{-\ell}$






```
1  def power(run1, run2, n):
2      i1, n1 = run1
3      i2, n2 = run2
4      a = (i1 + n1/2) / n
5      b = (i2 + n2/2) / n
6      l = 0
7      while math.floor(a * 2**l) == \
8            math.floor(b * 2**l):
9          l += 1
10     return l
```

- More efficient implementation possible (avoid multiplication / division in loop)



Some performance data

Summary claims:

1. Typical improvement from Powersort:
   0-5% fewer comparisons; occasionally 20–30%. 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%)

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



☐ = 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)

Bonus: Multiway powersort

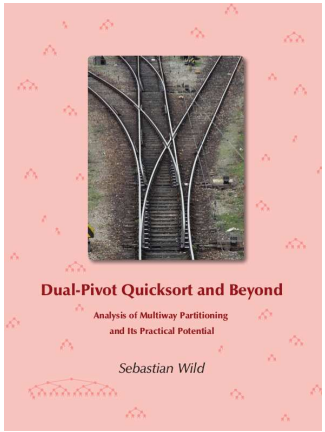
- Timsort has been highly successful export from Python



☐ = 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



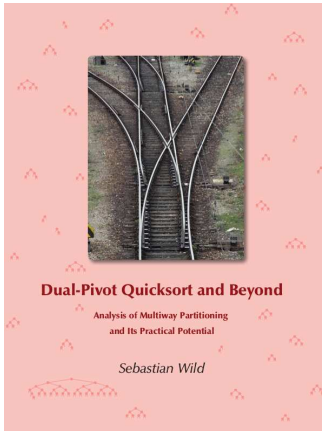
☐ = now using Powersort merge policy

- 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)

⇝ Easy to do with Powersort while keeping adaptivity!

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

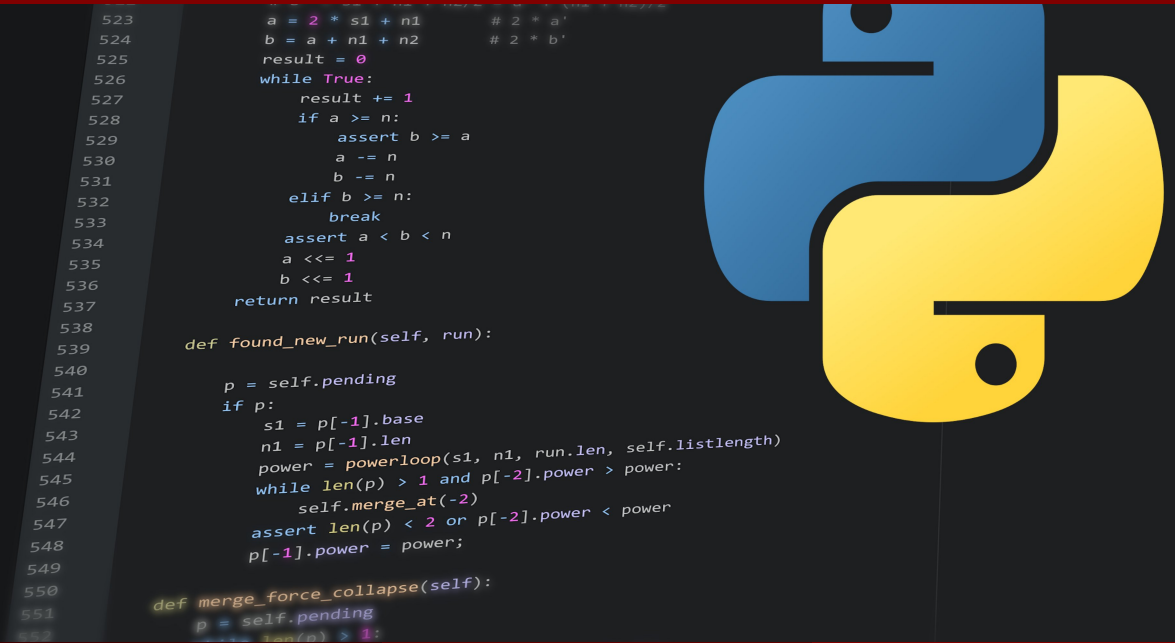
*



(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


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



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 every 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 every 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 every 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?

Conclusion

Summary

- Timsort is … method of choice
- its ori… … ideal
  - co… … / analysis
  - bl…
- Powersc…



Instructions to contribute:

tiny.cc/sort-lake-city

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?
   - **Please help me with your data!**

I only care about **relative ordering**

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

    def found_new_run(self, run):

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

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





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**

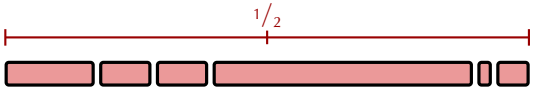


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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**

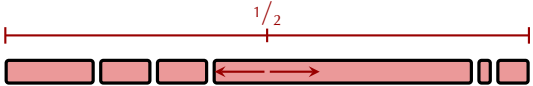


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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**

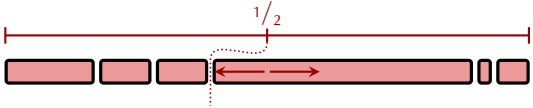


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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**

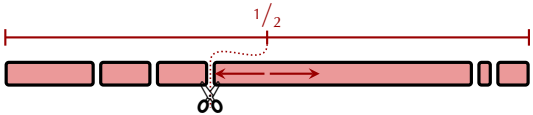


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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**

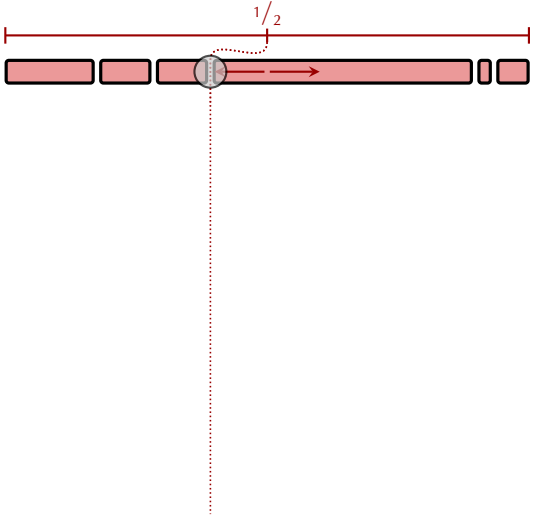


- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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**



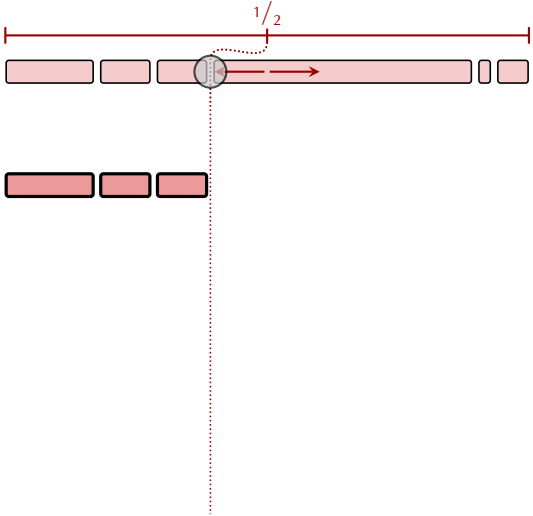
- **Intuition:**
  "Round" to a perfectly balanced binary tree
  - Pretend array is interval $[0, 1]$
  - Find run boundaries closest to middle
  - Recurse on both sides

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

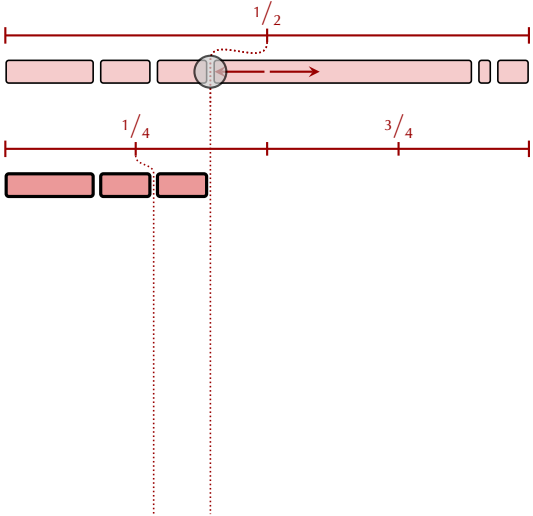


The Bisection Method

- 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

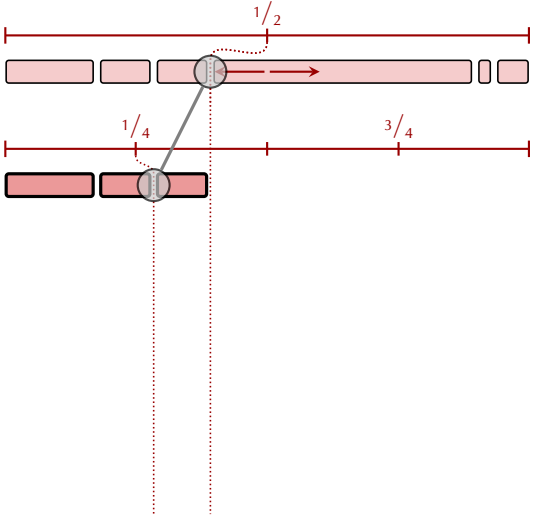


The Bisection Method

- 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

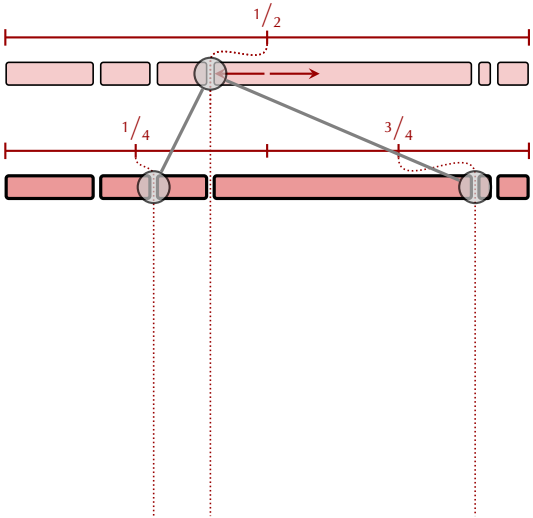


The Bisection Method

- 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

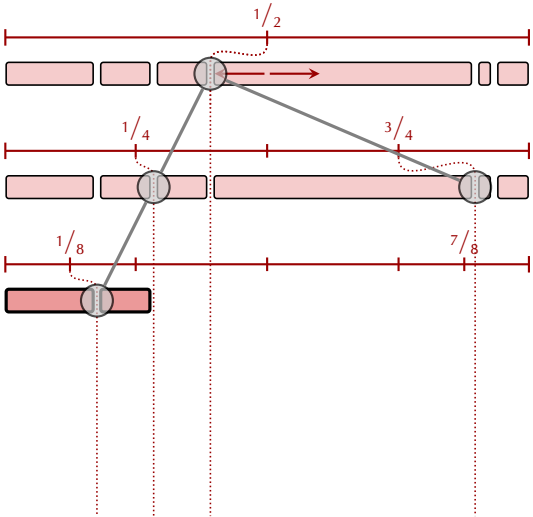


The Bisection Method

- 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

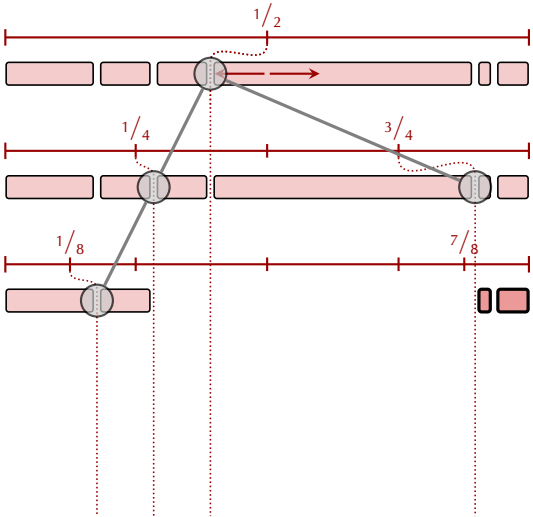


The Bisection Method

- 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

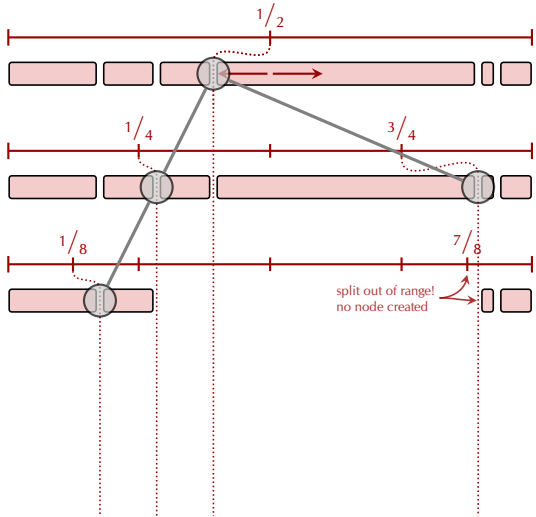


The Bisection Method

- 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

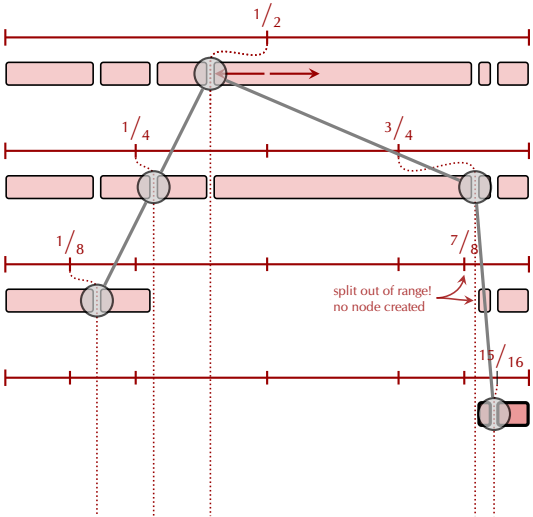


The Bisection Method

- 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

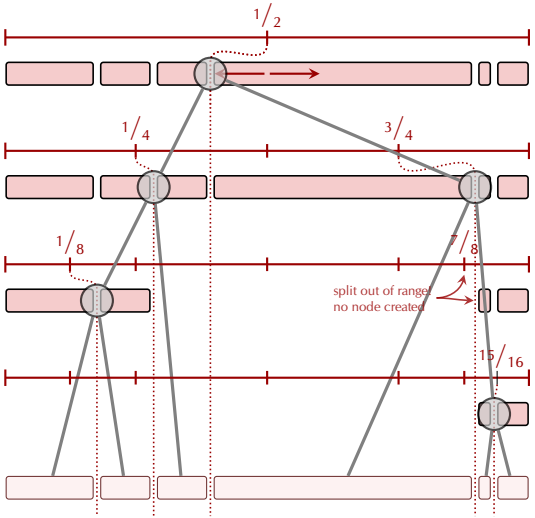


The Bisection Method

- 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

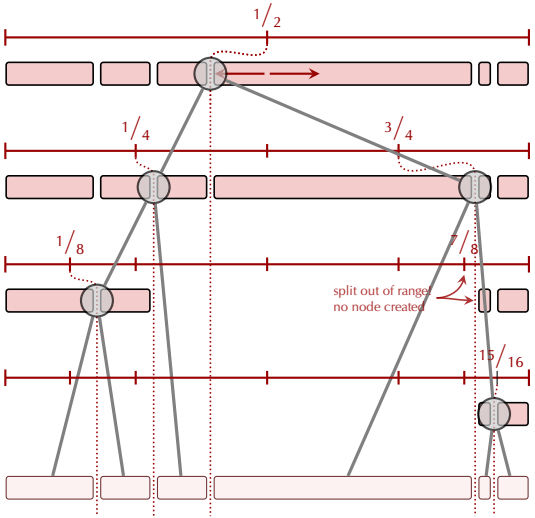


The Bisection Method

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

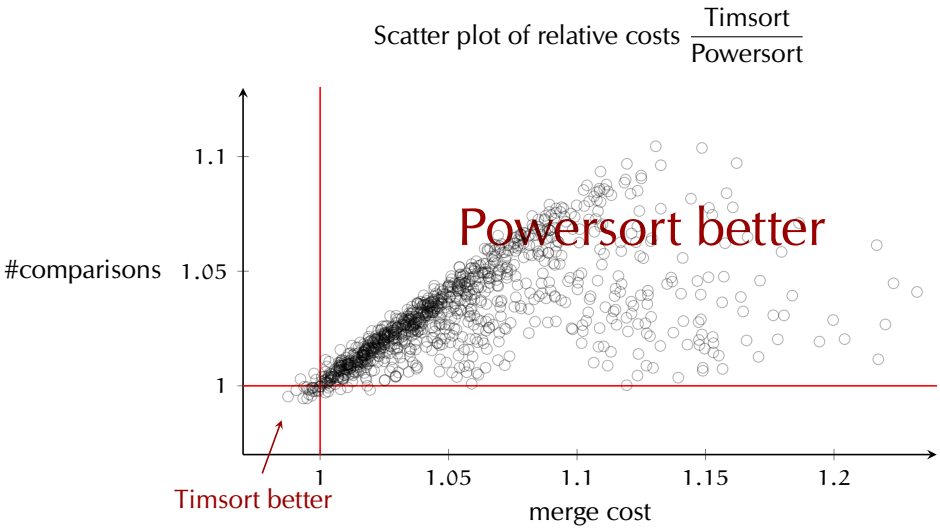
> **Mehlhorn**: *A best possible bound for the weighted path length of binary search trees,* SIAM J Comp **1977**

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



Abstract cost measures

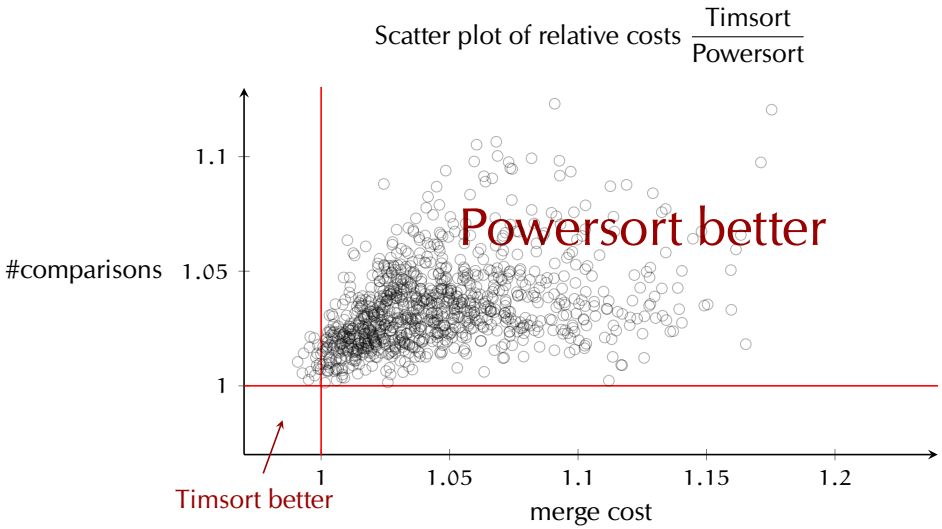
back



- 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

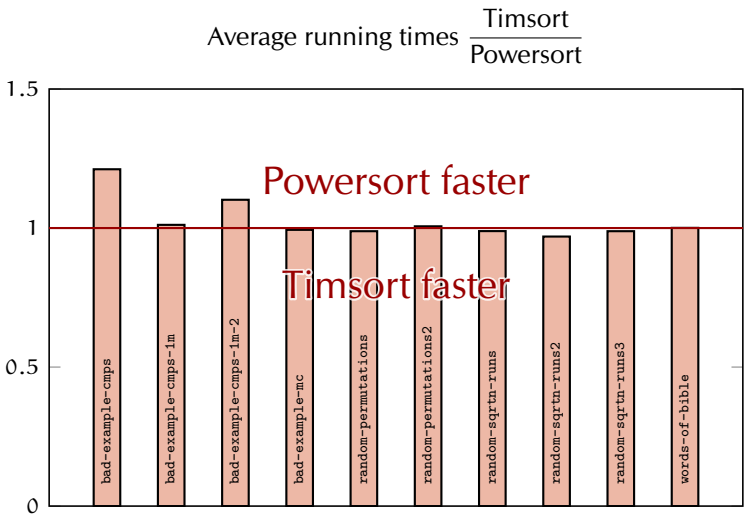


back

- 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

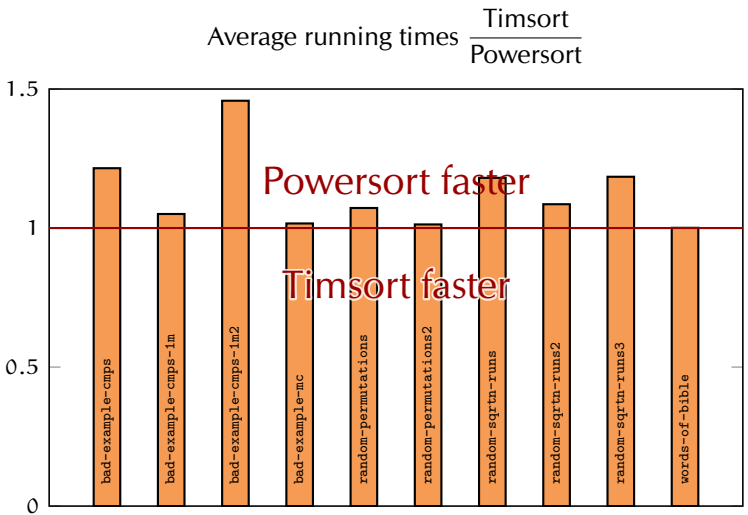


back

- CPython 3.11 with Powersort resp. Timsort
- selection of some “random” inputs
- all using `int` (cheap comparisons)
- machine-dependent, but qualitatively stable

Machine 1

Running times

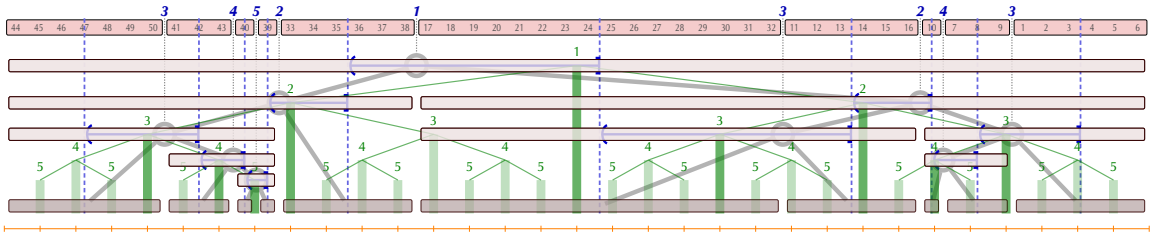


back

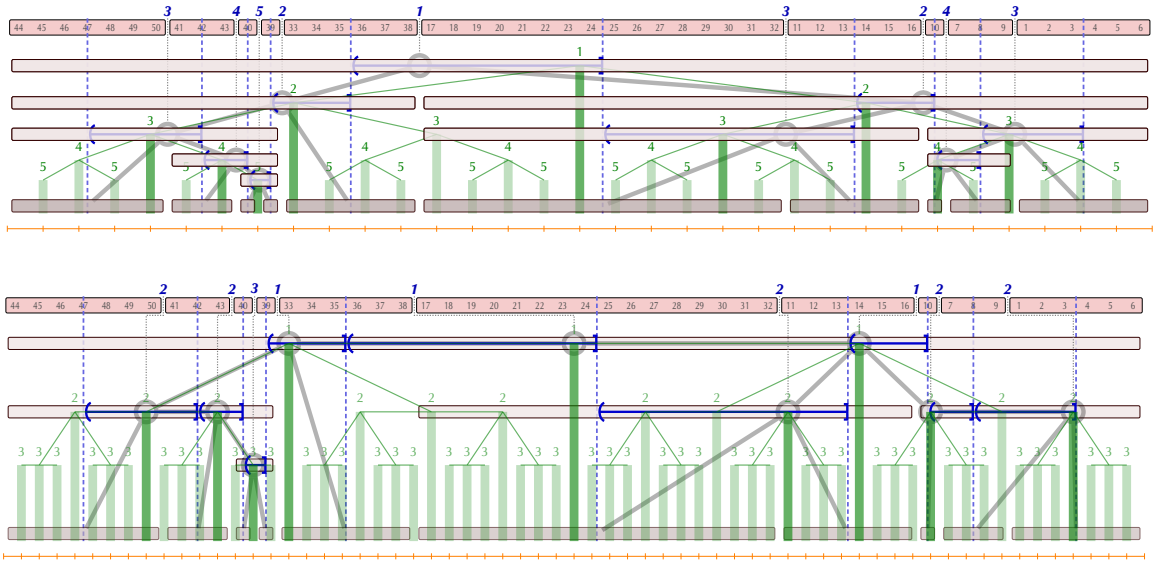
- CPython 3.11 with Powersort resp. Timsort
- selection of some “random” inputs
- all using `int` (cheap comparisons)
- machine-dependent, but qualitatively stable

Machine 2

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`.