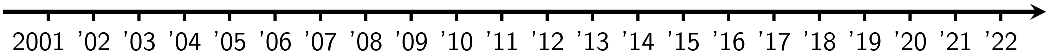
Galloping in fast-growth natural merge sorts

Elahe Ghasemi[2,3], Vincent Jugé[2] & Ghazal Khalighinejad[1,3]

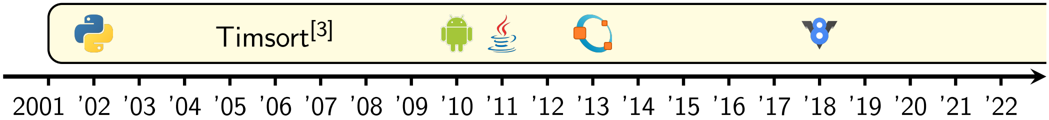
1: Duke University
2: LIGM – Université Gustave Eiffel & CNRS
3: Sharif University of Technology

08/07/2022

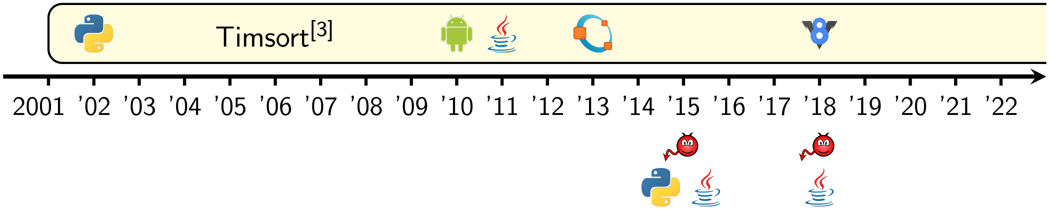
Library sorting algorithms in a few languages (for composite-type arrays)



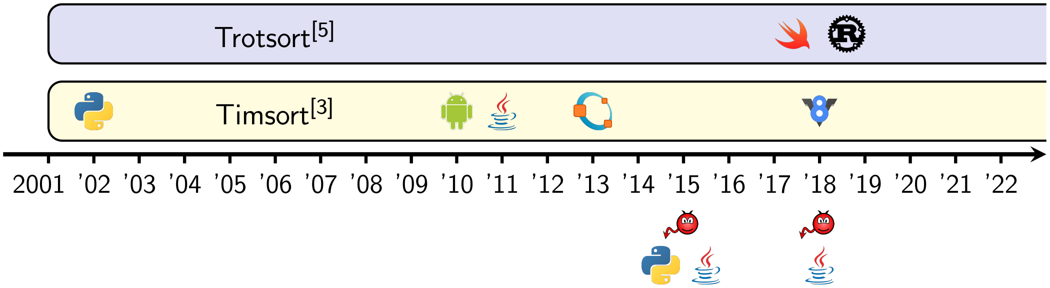
Library sorting algorithms in a few languages (for composite-type arrays)



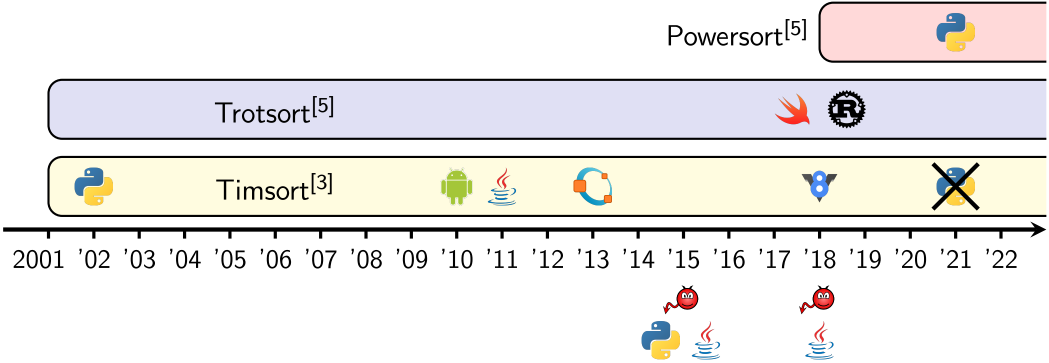
Library sorting algorithms in a few languages (for composite-type arrays)



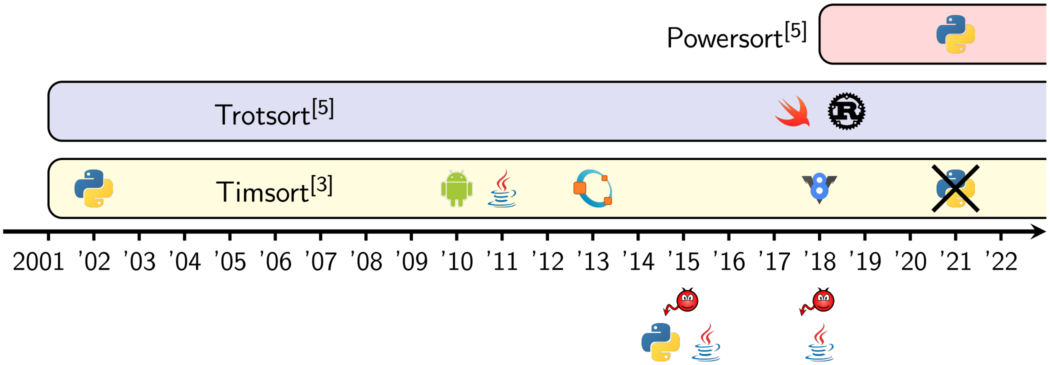
Library sorting algorithms in a few languages (for composite-type arrays)



Library sorting algorithms in a few languages (for composite-type arrays)

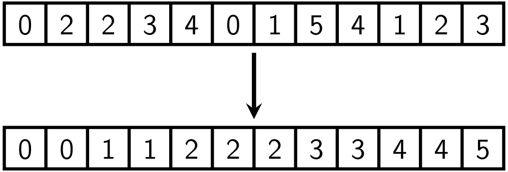


Library sorting algorithms in a few languages (for composite-type arrays)



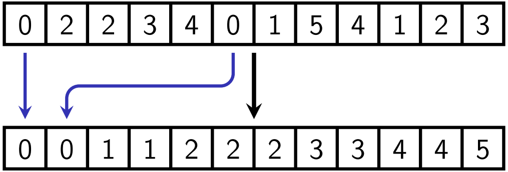
Why don't people just use plain (DualPivot)QuickSort + Heapsort?

Sorting data



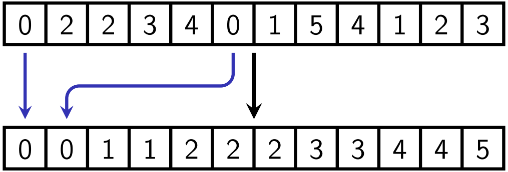
Heapsort and **Mergesort** have a **worst-case time complexity** of $\mathcal{O}(n \log(n))$ and we cannot do better, even on average...

Sorting data in a stable manner

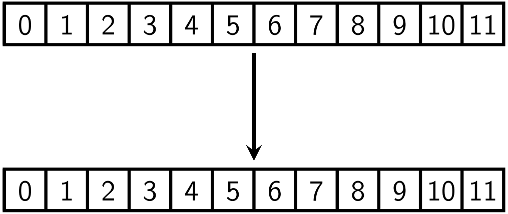


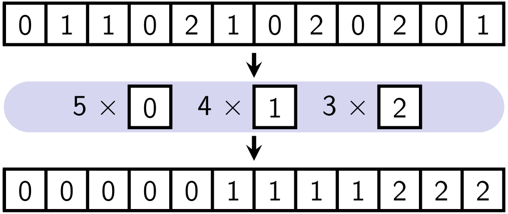
Heapsort and **Mergesort** have a **worst-case time complexity** of $\mathcal{O}(n \log(n))$ and we cannot do better, even on average...

Sorting data in a stable manner



Heapsort and **Mergesort** have a **worst-case time complexity** of $\mathcal{O}(n \log(n))$
and we cannot do better, even on average...
But, sometimes, **we can!**





Let us do better!

0	3	4	4	3	2	1	4	3	2	0	5

1. Subdivide your array in **monotonic** (non-decreasing or decreasing) **runs**.

Let us do better!

4 runs of lengths 4, 3, 4 and 1

0	3	4	4	3	2	1	4	3	2	0	5

1. Subdivide your array in **monotonic** (non-decreasing or decreasing) **runs**.
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)

Let us do better!

4 runs of lengths 4, 3, 4 and 1

0	3	4	4	3	2	1	4	3	2	0	5

1. Subdivide your array in **monotonic** (non-decreasing or decreasing) **runs**.
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)

   Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i) \leqslant \log_2(\rho) \leqslant \log_2(n)$

Let us do better!

4 runs of lengths 4, 3, 4 and 1

0	3	4	4	3	2	1	4	3	2	0	5

1. Subdivide your array in **monotonic** (non-decreasing or decreasing) **runs**.
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)

Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho}(r_i/n)\log_2(n/r_i) \leqslant \log_2(\rho) \leqslant \log_2(n)$

Theorem[5]

Powersort uses $\mathcal{O}(n + n\,\mathcal{H})$ element moves and $\mathcal{O}(n) + n\,\mathcal{H}$ comparisons.

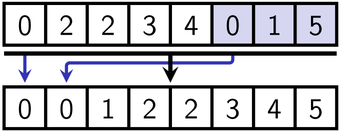
We cannot do better than $\mathcal{O}(n) + n\,\mathcal{H}$ comparisons![4]

There are X possible reorderings, with $X \geqslant 2^{1-\rho}\binom{n}{r_1 \ldots r_\rho} \geqslant 2^{(\mathcal{H}-5)n}$.

The principles of Timsort, Trotsort, Powersort et al.

Algorithms based on **merging** adjacent runs

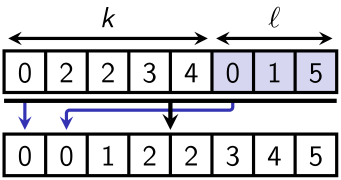
☛ **Stable** algorithms
(good for **composite** types)



The principles of Timsort, Trotsort, Powersort et al.

Algorithms based on **merging** adjacent runs

☛ **Stable** algorithms (good for **composite** types)



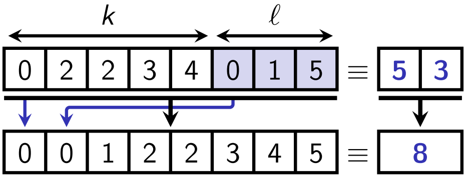
1. **Extend small runs** to save time $\mathcal{O}(n)$, and make them non-decreasing
2. **Run merging** sub-routine: naïve (Trotsort) or optimised (Timsort & Powersort)
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   } **Merge cost:** $k+\ell \geqslant$ #comparisons
3. **Policy** for choosing runs to merge:
   - depends on **run lengths** and **positions** only

The principles of Timsort, Trotsort, Powersort et al.

Algorithms based on **merging** adjacent runs

☛ **Stable** algorithms (good for **composite** types)

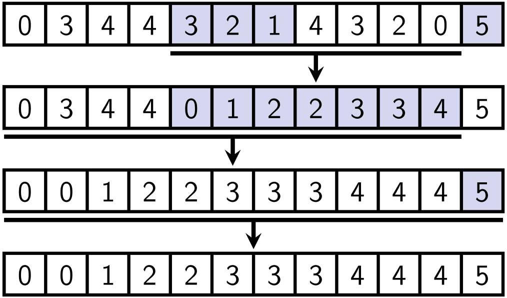


1. **Extend small runs** to save time $\mathcal{O}(n)$, and make them non-decreasing
2. **Run merging** sub-routine: naïve (Trotsort) or optimised (Timsort & Powersort)
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   } **Merge cost:** $k+\ell \geqslant$ #comparisons
3. **Policy** for choosing runs to merge:
   - depends on **run lengths** and **positions** only
4. **Complexity analysis:**
   - ☛ Evaluate the **total merge cost**
   - ☛ Just work with **run lengths**

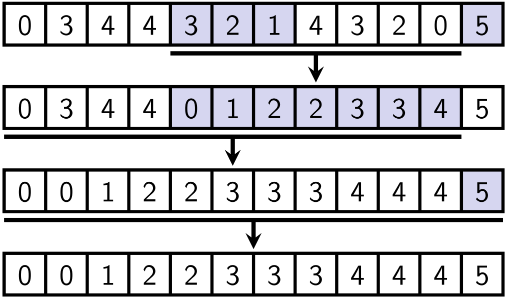
Merge trees and fast growth

Timsort merges

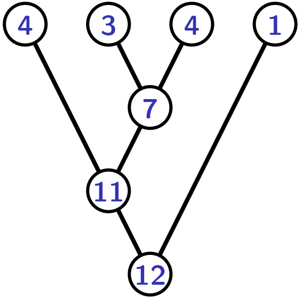


Merge trees and fast growth

Timsort merges

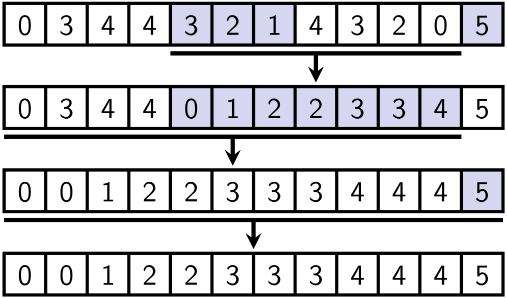


Timsort merge tree

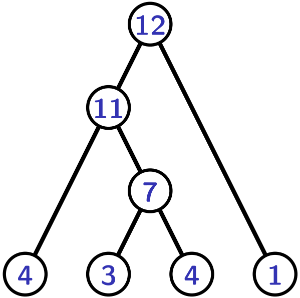


Merge trees and fast growth

Timsort merges

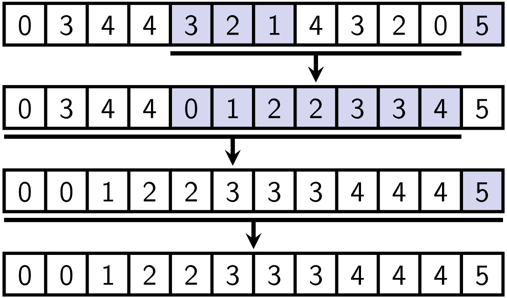


Timsort merge tree

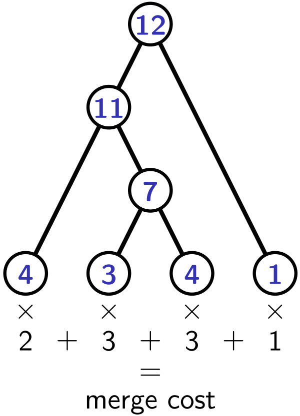


Merge trees and fast growth

Timsort merges

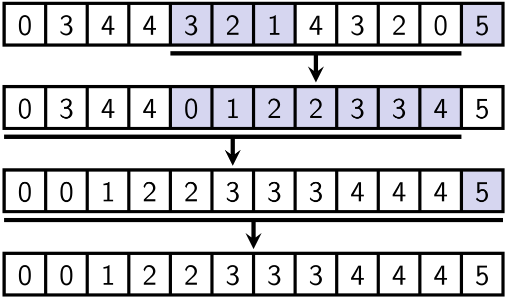


Timsort merge tree

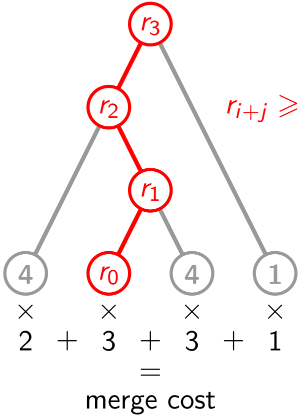


Merge trees and fast growth

Timsort merges



Timsort merge branch



$$r_{i+j} \geqslant (14/13)^{j-2} r_i$$

Fast growth and merge cost

Fast growth[6]

An natural merge sort is **fast-growing** if node sizes grow exponentially fast on its branches:

$$r_{i+j} \geqslant a^{j-b} \times r_i \text{ for some constants } a > 1 \text{ and } b \geqslant 0.$$

Fast growth and merge cost

Fast growth[6]

An natural merge sort is **fast-growing** if node sizes grow exponentially fast on its branches:

$$r_{i+j} \geqslant a^{j-b} \times r_i \text{ for some constants } a > 1 \text{ and } b \geqslant 0.$$

- ☛ Each leaf of size r lies at depth $d \leqslant \log_a(n/r) + b$.
- ☛ Such an algorithm has a merge cost $\leqslant \log_a(2) n \mathcal{H} + bn$.

Fast growth and merge cost

Fast growth[6]

An natural merge sort is **fast-growing** if node sizes grow exponentially fast on its branches:

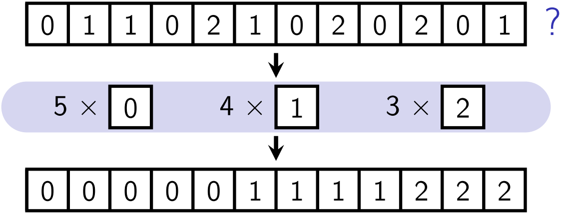
$$r_{i+j} \geqslant a^{j-b} \times r_i \text{ for some constants } a > 1 \text{ and } b \geqslant 0.$$

- Each leaf of size r lies at depth $d \leqslant \log_a(n/r) + b$.
- Such an algorithm has a merge cost $\leqslant \log_a(2) n \mathcal{H} + bn$.

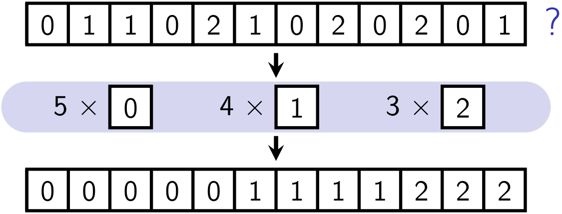
- **Fast-growing algorithms** work in time $\mathcal{O}(n + n\mathcal{H})$.

  Examples: Timsort, α-Mergesort, Powersort, Peeksort, adaptive Shiverssort
- **Powersort** performs no more than $n(\mathcal{H} + 4)$ comparisons (because $a = 2$ and $b = 4$).
- **Peeksort** and **adaptive Shiverssort** perform only $\mathcal{O}(n) + n\mathcal{H}$ comparisons (but $a > 2$).

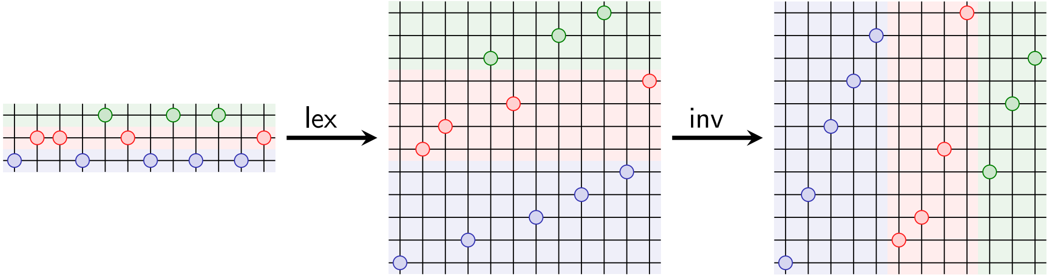
What about ?



What about 0 1 1 0 2 1 0 2 0 2 0 1 ?



Few **runs** vs few **values** vs few **dual runs**:



Let us do better, dually!

3 dual runs of lengths **5**, **4** and **3**

0	1	1	0	2	1	0	2	0	2	0	1

1. Subdivide your data in non-decreasing, non-overlapping **dual runs**
2. New parameters: **Number of dual runs** ($\rho^\star$) and their **lengths** ($r_i^\star$)

Dual-run entropy: $\mathcal{H}^\star = \sum_{i=1}^{\rho^\star} (r_i^\star/n) \log_2(n/r_i^\star) \leqslant \log_2(\rho^\star) \leqslant \log_2(n)$

Let us do better, dually!

3 dual runs of lengths **5**, **4** and **3**

0	1	1	0	2	1	0	2	0	2	0	1

1. Subdivide your data in non-decreasing, non-overlapping **dual runs**
2. New parameters: **Number of dual runs** ($\rho^\star$) and their **lengths** ($r_i^\star$)

Dual-run entropy: $\mathcal{H}^\star = \sum_{i=1}^{\rho^\star} (r_i^\star / n) \log_2(n / r_i^\star) \leqslant \log_2(\rho^\star) \leqslant \log_2(n)$

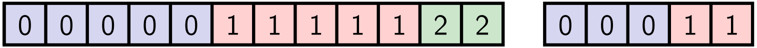
Theorem[6]

Fast-growing merge sorts require $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons if they use Timsort's **galloping** run-merging routine*.
*we are slightly cheating

and we cannot use less than $\mathcal{O}(n) + n\,\mathcal{H}^\star$ comparisons in general.

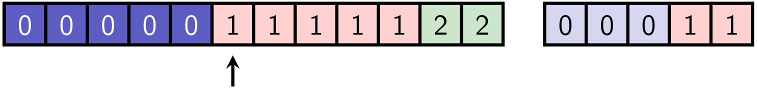
Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]



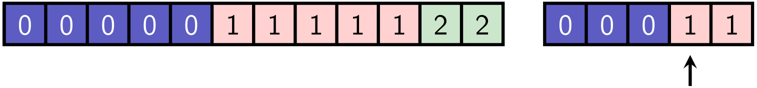
Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]



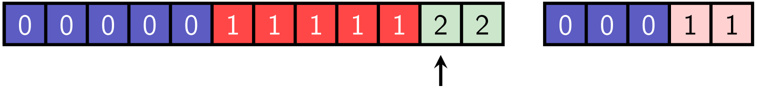
Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]



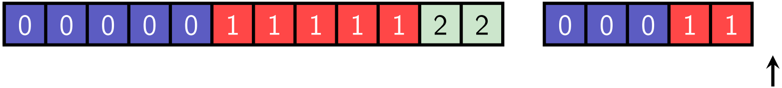
Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]



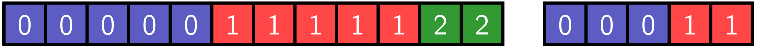
Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]



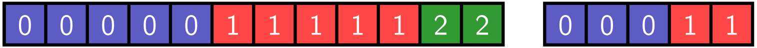
Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]



Galloping merging procedure

Merging runs $\approx$ **finding an integer** (several times)[1,2]

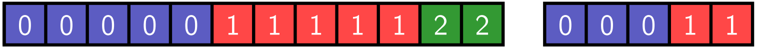


Finding an integer $\boldsymbol{x}$ by asking y and being told whether $y \geqslant x$:

1. Ask $y = 1, 2, 3, 4 \ldots$ (time x)

Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]

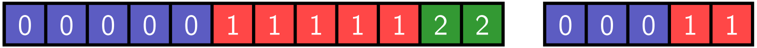


Finding an integer x by asking y and being told whether $y \geqslant x$:

1. Ask $y = 1, 2, 3, 4 \ldots$ (time x)
2. First ask $y = 1, 2, 4, 8, \ldots$, then find the bits of x (time $2\log_2(x)$)

Galloping merging procedure

Merging runs ≈ finding an integer (several times)[1,2]

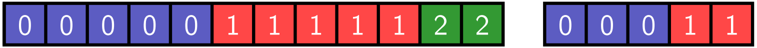


Finding an integer *x* by asking y and being told whether $y \geqslant x$:

1. Ask $y = 1, 2, 3, 4 \ldots$ (time x)
2. First ask $y = 1, 2, 4, 8, \ldots$, then find the bits of x (time $2 \log_2(x)$)
   Find $\log_2(x)$ with method 1, then find the bits of x

Galloping merging procedure

Merging runs $\approx$ **finding an integer** (several times)[1,2]



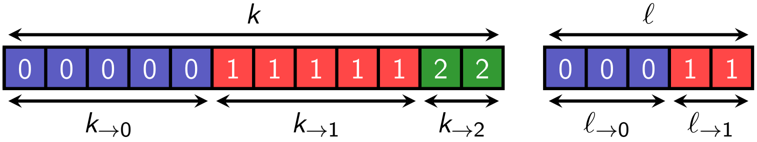
Finding an integer x by asking y and being told whether $y \geqslant x$:

1. Ask $y = 1, 2, 3, 4 \ldots$ (time x)
2. First ask $y = 1, 2, 4, 8, \ldots$, then find the bits of x (time $2\log_2(x)$)

   Find $\log_2(x)$ with method 1, then find the bits of x
3. Find $\log_2(x)$ with method 2, then find the bits of x (time $\log_2(x)$)

Galloping merging procedure

Merging runs ≈ **finding an integer** (several times)[1,2]



Finding an integer x by asking y and being told whether $y \geqslant x$:

1. Ask $y = 1, 2, 3, 4 \ldots$ (time x)
2. First ask $y = 1, 2, 4, 8, \ldots$, then find the bits of x (time $2\log_2(x)$)
   Find $\log_2(x)$ with method 1, then find the bits of x
3. Find $\log_2(x)$ with method 2, then find the bits of x (time $\log_2(x)$)

Timsort merging procedure ≈ methods 1 + 2 with threshold **t**[2,3]:

4. Ask $y = 1, 2, \ldots, \mathbf{t}+1, \mathbf{t}+2, \mathbf{t}+4, \mathbf{t}+8, \ldots$, then find the bits of $x - \mathbf{t}$

☛ **Merge cost:** $\sum_i \min\{(1+\mathbf{t}^{-1})(k_{\to i} + \ell_{\to i}), 6\mathbf{t} + 4\log_2(k_{\to i} + \ell_{\to i} + 1)\} \geqslant$ #comparisons

Conclusions (after a few more computations)

- ☛ For fixed thresholds **t**, fast-growth natural merge sorts require $\mathcal{O}(n + n\mathcal{H}^\star)$ comparisons.
- ☛ Choosing adequate choices of **t**, **Powersort** requires $\mathcal{O}(n) + (1 + o(1))n\mathcal{H}^\star$ comparisons.

Choose $\mathbf{t} \approx \log(k + \ell)$ to merge runs of lengths k and ℓ

Conclusions (after a few more computations)

- For fixed thresholds **t**, fast-growth natural merge sorts require $\mathcal{O}(n + n\mathcal{H}^\star)$ comparisons.
- Choosing adequate choices of **t**, **Powersort** requires $\mathcal{O}(n) + (1 + o(1))n\mathcal{H}^\star$ comparisons.

  Choose $\mathbf{t} \approx \log(k + \ell)$ to merge runs of lengths k and ℓ

- Timsort updates **t** in a way that makes the $\mathcal{O}(n + n\mathcal{H}^\star)$ upper bound look dubious.
- Trotsort requires $\Omega(n \log(n))$ comparisons to sort 0101010101010101...

Conclusions (after a few more computations)

- For fixed thresholds **t**, fast-growth natural merge sorts require $\mathcal{O}(n + n\mathcal{H}^\star)$ comparisons.
- Choosing adequate choices of **t**, **Powersort** requires $\mathcal{O}(n) + (1 + o(1))n\mathcal{H}^\star$ comparisons.

  Choose **t** $\approx \log(k + \ell)$ to merge runs of lengths k and ℓ

- Timsort updates **t** in a way that makes the $\mathcal{O}(n + n\mathcal{H}^\star)$ upper bound look dubious.
- Trotsort requires $\Omega(n \log(n))$ comparisons to sort 0101010101010101...

- Studying widely-used algorithms/heuristics rocks!

Conclusions (after a few more computations) and references

- For fixed thresholds **t**, fast-growth natural merge sorts require $\mathcal{O}(n + n\mathcal{H}^\star)$ comparisons.
- Choosing adequate choices of **t**, **Powersort** requires $\mathcal{O}(n) + (1 + o(1))n\mathcal{H}^\star$ comparisons.

  Choose $\mathbf{t} \approx \log(k + \ell)$ to merge runs of lengths k and ℓ

- Timsort updates **t** in a way that makes the $\mathcal{O}(n + n\mathcal{H}^\star)$ upper bound look dubious.
- Trotsort requires $\Omega(n \log(n))$ comparisons to sort 0101010101010101...

- Studying widely-used algorithms/heuristics rocks!

[1] *An almost optimal algorithm for unbounded searching*, Bentley & Yao (1976)
[2] *Optimistic Sorting and Information Theoretic Complexity*, McIlroy (1993)
[3] *Description of TimSort*, Peters
`svn.python.org/projects/python/trunk/Objects/listsort.txt` (2001)
[4] *On compressing permutations and adaptive sorting*, Barbay & Navarro (2013)
[5] *Nearly-optimal mergesorts*, Munro & Wild (2018)
[6] *Galloping in natural merge sorts*, Ghasemi, Jugé & Khalighinejad (2022)


THANK YOU FOR YOUR ATTENTION!
DO YOU HAVE ANY EASY QUESTIONS?