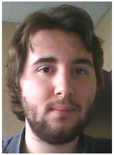
Sorting presorted data

Vincent Jugé

LIGM – Université Gustave Eiffel, ESIEE, ENPC & CNRS

09/12/2021

Joint work with
N. Auger, C. Nicaud, C. Pivoteau, A. Ghasemi & G. Khalighinejad



Université Gustave Eiffel

Sharif University of Technology
Duke University

Sorting data

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

$$\downarrow$$

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

Sorting data

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

$\downarrow$

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

MergeSort has a worst-case time complexity of $\mathcal{O}(n \log(n))$

Can we do better?

Sorting data

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

$\downarrow$

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

MergeSort has a **worst-case time complexity** of $\mathcal{O}(n\log(n))$

Can we do better? No!

Proof:

- There are $n!$ possible reorderings
- Each element comparison gives a 1-bit information
- Thus $\log_2(n!) \sim n\log_2(n)$ tests are required

Sorting data

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

$\downarrow$

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

MergeSort has a **worst-case time complexity** of $\mathcal{O}(n\log(n))$

Can we do better? No!

Proof:

- There are $n!$ possible reorderin…
- Each element compar… …ation
- Thus $\log_2(n!) \sim n\log$… …e required

END OF TALK!

Cannot we ever do better?

In some cases, we should. . .

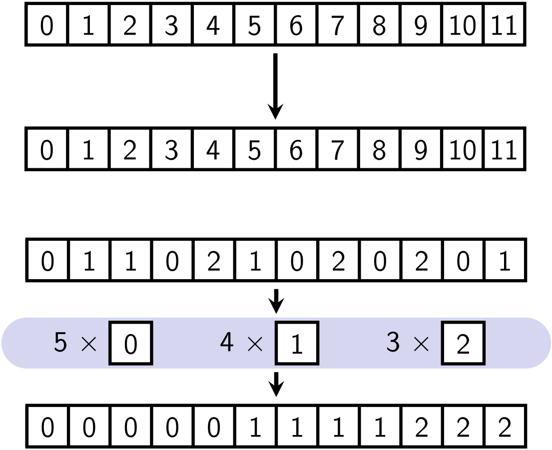
0	1	2	3	4	5	6	7	8	9	10	11

$\downarrow$

0	1	2	3	4	5	6	7	8	9	10	11

Cannot we ever do better?

In some cases, we should…



Let us do better!

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

1. Chunk your data in **non-decreasing runs**

Let us do better!

4 runs of lengths **5**, **3**, **1** and **3**

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

1. Chunk your data in **non-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 **5**, **3**, **1** and **3**

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

1. Chunk your data in **non-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 **5**, **3**, **1** and **3**

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

1. Chunk your data in **non-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 [1, 2, 6, 9, 13]

Some merge sort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

Let us do better!

4 runs of lengths 5, 3, 1 and 3

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

1. Chunk your data in **non-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 [1, 2, 6, 9, 13]

TimSort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

Let us do better!

4 runs of lengths 5, 3, 1 and 3

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

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

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

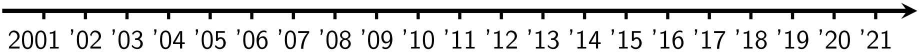
Theorem [1, 2, 6, 9, 13]

TimSort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

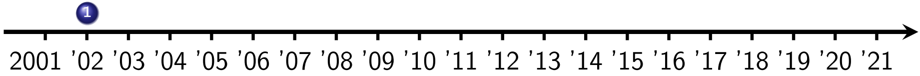
We cannot do better than $\Omega(n + n\,\mathcal{H})$![6]

- Reading the whole input requires a time $\Omega(n)$
- There are X possible reorderings, with $X \geqslant 2^{1-\rho}\binom{n}{r_1 \ldots r_\rho} \geqslant 2^{n\,\mathcal{H}/2}$

A brief history of TimSort

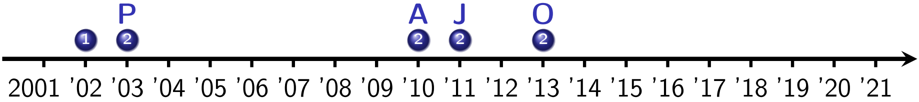


A brief history of TimSort



1. Invented by **Tim Peters**[5]

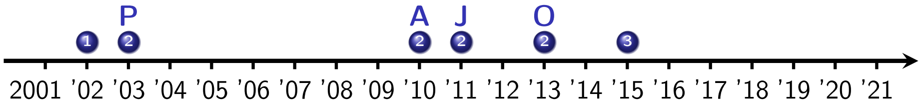
A brief history of TimSort



1. Invented by **Tim Peters**[5]
2. Standard algorithm in **Python**

   —— for non-primitive arrays in **Android**, **Java**, **Octave**

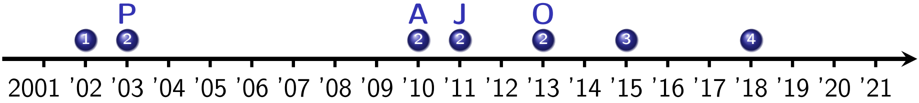
A brief history of TimSort



1. Invented by **Tim Peters**[5]
2. Standard algorithm in **Python**

   for non-primitive arrays in **Android**, **Java**, **Octave**
3. 1st worst-case complexity analysis[8] – TimSort works in time $\mathcal{O}(n \log n)$

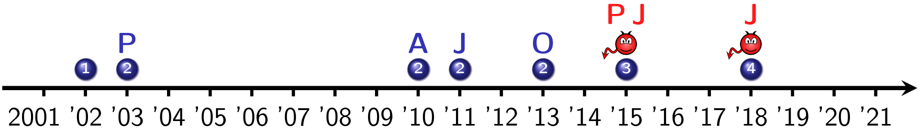
A brief history of TimSort



1. Invented by **Tim Peters**[5]
2. Standard algorithm in **Python**

   ——————— for non-primitive arrays in **Android**, **Java**, **Octave**
3. 1st worst-case complexity analysis[8] – TimSort works in time $\mathcal{O}(n \log n)$
4. Refined worst-case analysis[9] – TimSort works in time $\mathcal{O}(n + n\,\mathcal{H})$

A brief history of TimSort



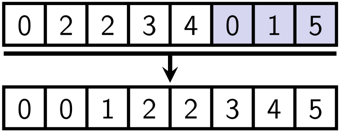
1. Invented by **Tim Peters**[5]
2. Standard algorithm in **Python**

   ——— for non-primitive arrays in **Android**, **Java**, **Octave**
3. 1st worst-case complexity analysis[8] – TimSort works in time $\mathcal{O}(n \log n)$
4. Refined worst-case analysis[9] – TimSort works in time $\mathcal{O}(n + n\mathcal{H})$

- Bugs uncovered in Python & Java implementations[7,9]

The principles of TimSort and its variants (1/2)

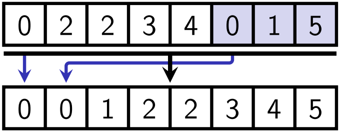
Algorithm based on **merging** adjacent runs



The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

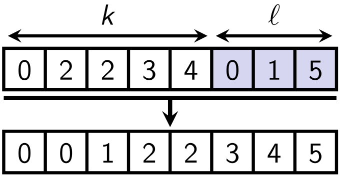
☛ **Stable** algorithm
(good for **composite** types)



The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

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



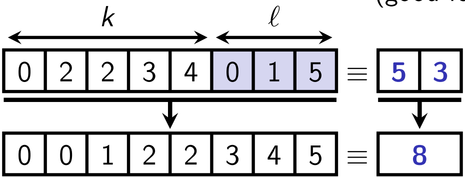
1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   $\Big\}$ **Merge cost:** $k+\ell$

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

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

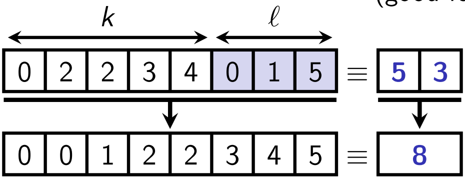


1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   $\Big\}$ **Merge cost:** $k+\ell$
2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs ☛ **Stable** algorithm (good for **composite** types)



1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   Merge cost: $k+\ell$
2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only
3. **Complexity analysis:**
   - ☛ Evaluate the **total merge cost**
   - ☛ Forget array values and only work with **run lengths**

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

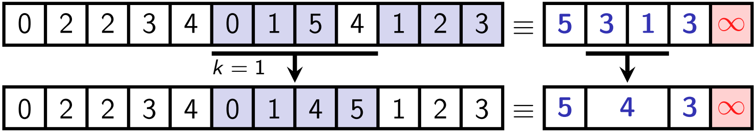
- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}

0	2	2	3	4	0	1	5	4	1	2	3	$\equiv$	**5**	**3**	**1**	**3**	∞

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

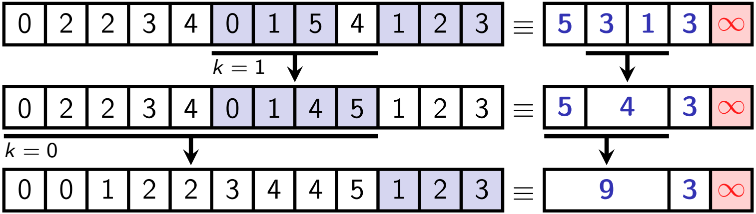
- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

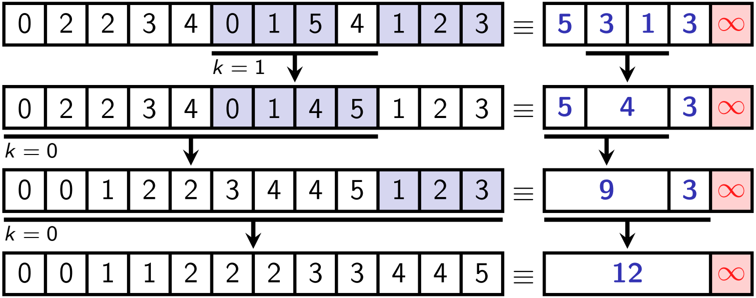
- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha - 1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

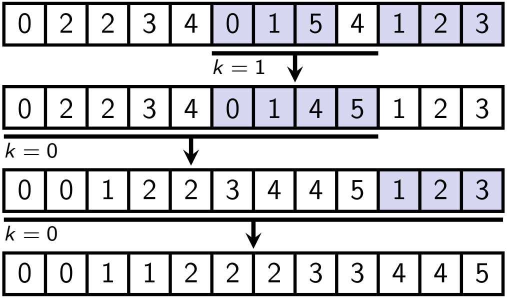
- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



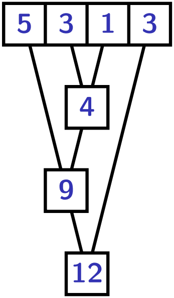
The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



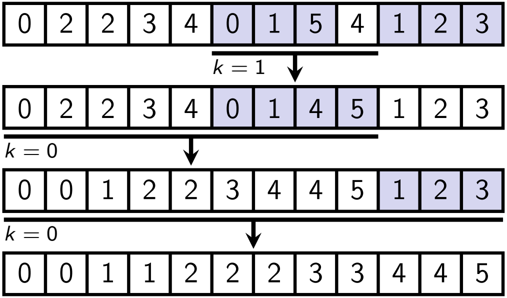
Merge tree



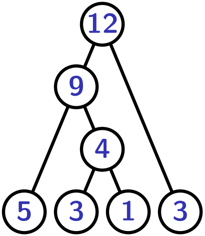
The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



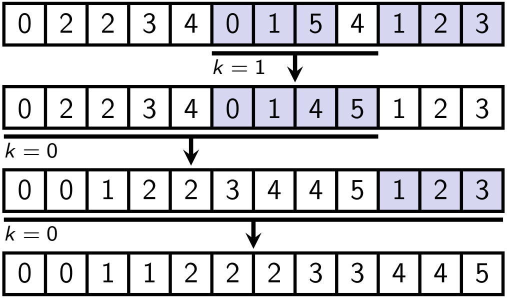
Merge tree



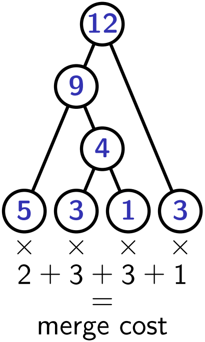
The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



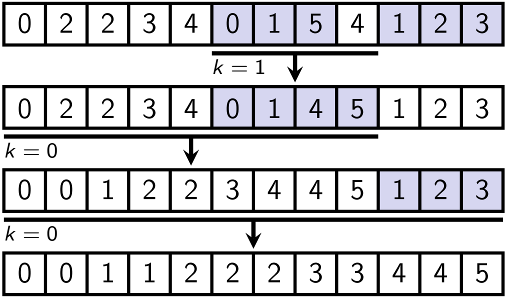
Merge tree



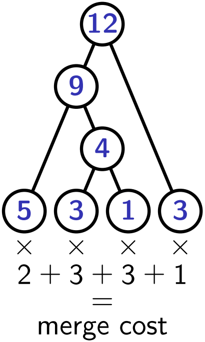
The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[11] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant \alpha(\alpha-1) r_{k+2}$
- Merge the runs R_k and R_{k+1}



Merge tree



- $k^{\text{new}} \geqslant k^{\text{old}} - 1$ after each merge:

  one can use **stack-based** implementations of α-merge sort

Fast growth in merge trees (1/2)

Theorem [13]

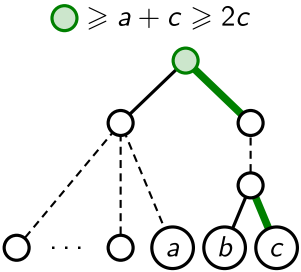
In merge trees induced by ϕ-merge sort, each node is at least ϕ times larger than its great-grandchildren

Fast growth in merge trees (1/2)

Theorem [13]

In merge trees induced by ϕ-merge sort, each node is at least ϕ times larger than its great-grandchildren

Proof:

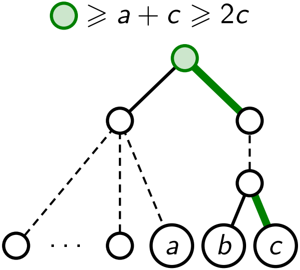


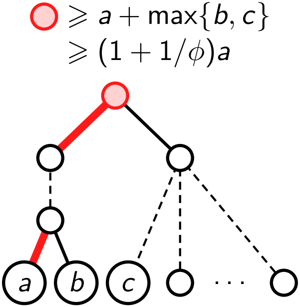
Fast growth in merge trees (1/2)

Theorem [13]

In merge trees induced by ϕ-merge sort, each node is at least ϕ times larger than its great-grandchildren

Proof:





Fast growth in merge trees (1/2)

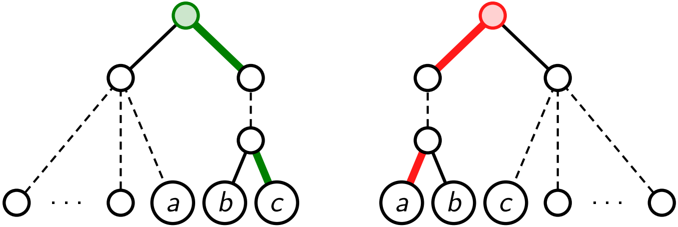
Theorem [13]

In merge trees induced by ϕ-merge sort, each node is at least ϕ times larger than its great-grandchildren

Proof:

$\geqslant a + c \geqslant 2c$

$\geqslant a + \max\{b, c\}$
$\geqslant (1 + 1/\phi)a$



Corollary:

- Each run R lies at depth $\mathcal{O}(1 + \log(n/r))$
- ϕ-merge sort has a merge cost $\mathcal{O}(n + n\mathcal{H})$

Fast growth in merge trees (2/2)

Fast-growth property

A merge algorithm **A** has the **fast-growth property** if

- there exists an integer $k \geqslant 1$ and a real number $\theta > 1$ such that
- in each merge tree induced by **A**,

going up k times multiplies the node size by θ or more

Fast growth in merge trees (2/2)

Fast-growth property

A merge algorithm **A** has the **fast-growth property** if

- there exists an integer $k \geqslant 1$ and a real number $\theta > 1$ such that
- in each merge tree induced by **A**,

going up k times multiplies the node size by θ or more

Theorem (continued)

Timsort[5], **α-merge sort**[11], **adaptive Shivers sort**[12], **Peeksort** and **Powersort**[10] have the fast growth-property

Corollary: These algorithms work in time $\mathcal{O}(n + n\mathcal{H})$

What about $\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{2}\boxed{1}\boxed{0}\boxed{2}\boxed{0}\boxed{2}\boxed{0}\boxed{1}$?

$$\downarrow$$

$$5 \times \boxed{0} \qquad 4 \times \boxed{1} \qquad 3 \times \boxed{2}$$

$$\downarrow$$

$$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{2}\boxed{2}\boxed{2}$$

What about

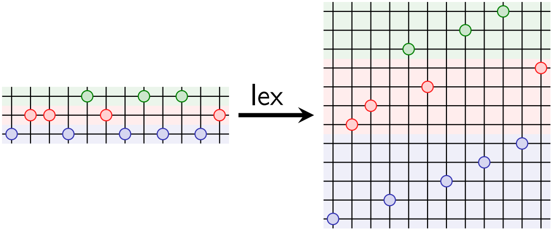
0	1	1	0	2	1	0	2	0	2	0	1

?

$5 \times \boxed{0}$ $\quad 4 \times \boxed{1}$ $\quad 3 \times \boxed{2}$

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

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

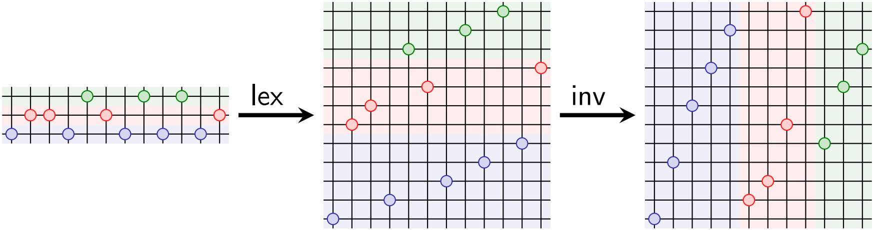


What about | 0 | 1 | 1 | 0 | 2 | 1 | 0 | 2 | 0 | 2 | 0 | 1 | ?

$5 \times \boxed{0}$ $4 \times \boxed{1}$ $3 \times \boxed{2}$

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

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. Chunk 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. Chunk 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 [13]

Every **fast-growth** merge sort requires $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons if it uses **Timsort's optimized run-merging routine**

and we still cannot do better than $\Omega(n + n\,\mathcal{H}^\star)$

Fast merging procedure

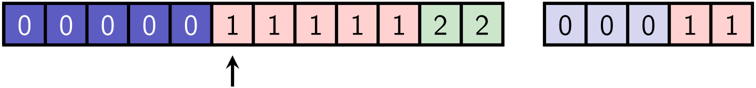
Merging ≈ finding an integer (several times)[3, 4]

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

0	0	0	1	1

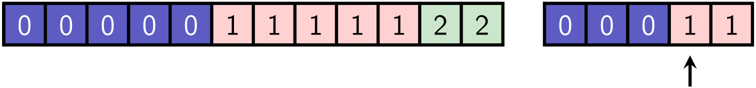
Fast merging procedure

Merging ≈ finding an integer (several times)[3, 4]



Fast merging procedure

Merging ≈ finding an integer (several times)[3, 4]



Fast merging procedure

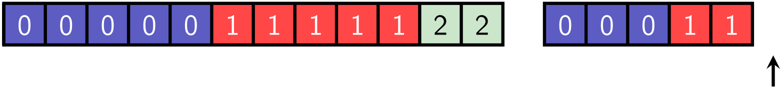
Merging ≈ finding an integer (several times)[3,4]

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

0	0	0	1	1

Fast merging procedure

Merging ≈ finding an integer (several times)[3, 4]



Fast merging procedure

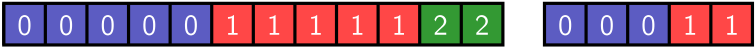
Merging ≈ finding an integer (several times)[3,4]

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

0	0	0	1	1

Fast merging procedure

Merging ≈ finding an integer (several times)[3,4]



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

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

Fast merging procedure

Merging $\approx$ finding an integer (several times)[3, 4]

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

0	0	0	1	1

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 $2\log_2(x)$)

Fast merging procedure

Merging ≈ finding an integer (several times)[3,4]

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

0	0	0	1	1

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

Fast merging procedure

Merging ≈ finding an integer (several times)[3, 4]

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

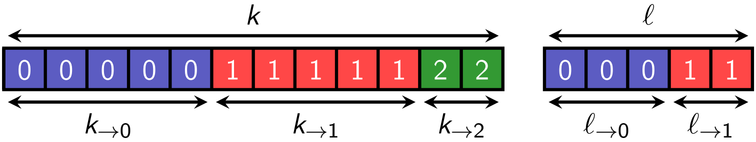
0	0	0	1	1

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 $2 \log_2(x)$)
2. 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 $\log_2(x)$)

Fast merging procedure

Merging ≈ finding an integer (several times)[3, 4]



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 $2 \log_2(x)$)
2. 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 $\log_2(x)$)

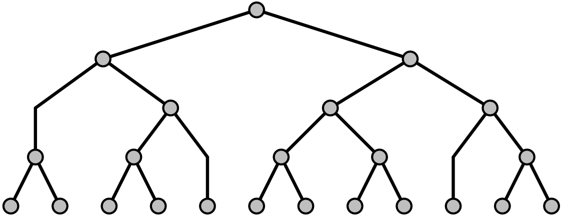
Timsort merging procedure ≈ methods **1** + **2** with threshold **t**[4, 5]:

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:** $\sum_i \log_2(1 + k_{\to i}) + \log_2(1 + \ell_{\to i})$

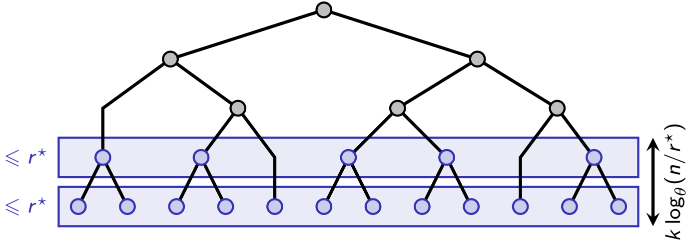
Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^{\star}$?



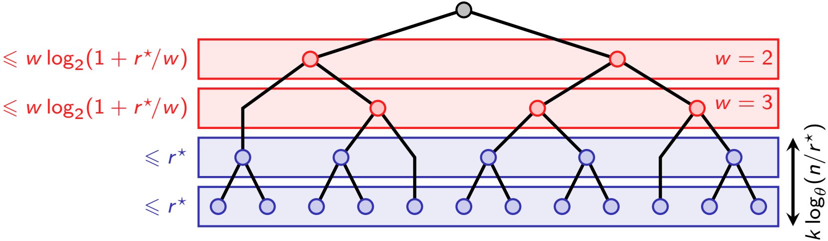
Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^\star$?



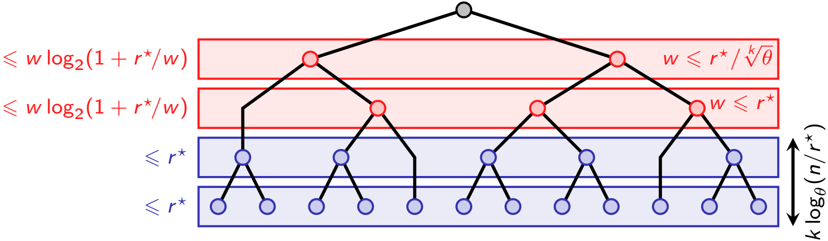
Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^\star$?



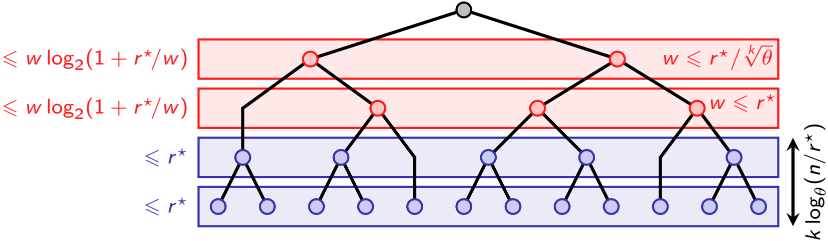
Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^\star$?



Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^\star$?

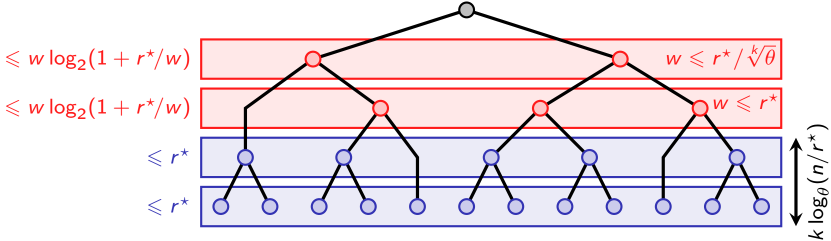


Overall, we perform at most:

- $k\, r^\star \log_\theta(n/r^\star) + \sum_{h \geqslant 0} r^\star/\theta^{h/k} \log_2(1 + \theta^{h/k})$ comparisons with $R^\star$

Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^\star$?



Overall, we perform at most:

- $\mathcal{O}(r^\star \log(n/r^\star) + r^\star)$ comparisons with $R^\star$
- $\mathcal{O}(n + n\mathcal{H}^\star)$ comparisons in total

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\,\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\,\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons
- Both its **merging policy** and **merging routine** are great!

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\,\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons
- Both its **merging policy** and **merging routine** are great!
- Use TimSort's merging routine in **Swift** and **Rust**!

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\,\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons
- Both its **merging policy** and **merging routine** are great!
- Use TimSort's merging routine in **Swift** and **Rust**!

☛ We still need to evaluate constants hidden in the $\mathcal{O}$ notations (WIP)

Some references

[1] *Optimal computer search trees and variable-length alphabetical codes*, Hu & Tucker (1971)
[2] *A new algorithm for minimum cost binary trees*, Garsia & Wachs (1973)
[3] *An almost optimal algorithm for unbounded searching*, Bentley & Yao (1976)
[4] *Optimistic Sorting and Information Theoretic Complexity*, McIlroy (1993)
[5] Tim Peters' description of TimSort, `svn.python.org/projects/python/trunk/Objects/listsort.txt` (2001)
[6] *On compressing permutations and adaptive sorting*, Barbay & Navarro (2013)
[7] *OpenJDK's java.utils.Collection.sort() is broken*, de Gouw et al. (2015)
[8] *Merge strategies: from merge sort to TimSort*, Auger et al. (2015)
[9] *On the worst-case complexity of TimSort*, Auger et al. (2018)
[10] *Nearly-optimal mergesorts*, Munro & Wild (2018)
[11] *Strategies for stable merge sorting*, Buss & Knop (2019)
[12] *Adaptive ShiversSort: an alternative sorting algorithm*, Jugé (2020)
[13] *Galloping in natural merge sorts*, Ghasemi, Jugé & Khalighinejad (2022^+)


MERCI POUR VOTRE ATTENTION !
NE POSEZ PAS DE QUESTIONS DIFFICILES S'IL VOUS PLAÎT !