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

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

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.

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

Can we do better?
Sorting data

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

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

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

Can we do better? No!

Proof:
- There are $n!$ possible reordering possibilities.
- Each element comparison gives a 1-bit information.
- Thus $\log_2(n!) \sim n \log_2(n)$ test operations are required.

END OF TALK!
Cannot we ever do better?

In some cases, we should…

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]
Cannot we ever do better?

In some cases, we should…

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

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

```
0 1 1 0 2 1 0 2 0 2 0 1
```

```
5 \times 0
4 \times 1
3 \times 2
```

```
0 0 0 0 0 0 1 1 1 1 2 2 2
```
Let us do better!

Chunk your data in **non-decreasing runs**
Let us do better!

4 runs of lengths 5, 3, 1 and 3

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3
\end{array}
\]

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

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

\[ \leq \log_2(\rho) \leq \log_2(n) \]
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** ($\rho$) and their **lengths** ($r_1, \ldots, r_\rho$)
   
   **Run-length entropy**: $\mathcal{H} = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i)$
   
   $\leq \log_2(\rho) \leq \log_2(n)$

**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 ($\rho$) and their lengths ($r_1, \ldots, r_\rho$)

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

Theorem [1, 2, 6, 9, 13]
TimSort has a worst-case time complexity of $O(n + nH)$
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** ($\rho$) and their **lengths** ($r_1, \ldots, r_{\rho}$)

**Run-length entropy:**

$$H = \sum_{i=1}^{\rho} \frac{r_i}{n} \log_2 \left( \frac{n}{r_i} \right)$$

$$\leq \log_2 (\rho) \leq \log_2 (n)$$

---

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

**TimSort** has a **worst-case time complexity** of $O(n + nH)$

---

We cannot do better than $\Omega(n + nH)!$ [6]

- Reading the whole input requires a time $\Omega(n)$
- There are $X$ possible reorderings, with $X \geq 2^{1-\rho} \binom{n}{r_1 \ldots r_\rho} \geq 2^{nH/2}$
A brief history of TimSort

Invented by Tim Peters

Standard algorithm ———— for non-primitive arrays in Python, Standard algorithm in Android, Java, Octave

1st worst-case complexity analysis ————

Refined worst-case analysis ————

Bugs uncovered in Python & Java implementations

Sorting presorted data
A brief history of TimSort

Invented by Tim Peters\textsuperscript{[5]}

Sorting presorted data
A brief history of TimSort

1. Invented by Tim Peters\textsuperscript{[5]}
2. Standard algorithm in Python
   ————————— for non-primitive arrays in Android, Java, Octave
A brief history of TimSort

1. Invented by Tim Peters\textsuperscript{[5]}
2. Standard algorithm in Python
   ———————————— for non-primitive arrays in Android, Java, Octave
3. 1\textsuperscript{st} worst-case complexity analysis\textsuperscript{[8]} – TimSort works in time $O(n \log n)$
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. 1\textsuperscript{st} worst-case complexity analysis[^8] – TimSort works in time $O(n \log n)$

4. Refined worst-case analysis[^9] – TimSort works in time $O(n + n \mathcal{H})$
A brief history of TimSort

1. Invented by Tim Peters\textsuperscript{[5]}
2. Standard algorithm in Python
   ———————————— for non-primitive arrays in Android, Java, Octave
3. 1\textsuperscript{st} worst-case complexity analysis\textsuperscript{[8]} – TimSort works in time $O(n \log n)$
4. Refined worst-case analysis\textsuperscript{[9]} – TimSort works in time $O(n + nH)$

 strtolles Bugs uncovered in Python & Java implementations\textsuperscript{[7,9]}

V. Jugé

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

Algorithm based on **merging** adjacent runs

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

Run merging algorithm: standard + many optimizations

*time* $O(k + \ell)$

*memory* $O(\min(k, \ell))$

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

**Policy for choosing runs to merge:**
- depends on run lengths only

**Complexity analysis:** Evaluate the total merge cost

Forget array values and only work with run lengths
The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs ✨ **Stable** algorithm

(good for **composite** types)

![Diagram](https://via.placeholder.com/150)

- Run merging algorithm: standard + many optimizations
  - time $O(k + \ell)$
  - memory $O(\min(k, \ell))$

Merge cost: $k + \ell$

Policy for choosing runs to merge:
- depends on run lengths only

Complexity analysis:
- Evaluate the total merge cost
- Forget array values and only work with run lengths
The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs  

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

Run merging algorithm: standard + many optimizations

- time $O(k + \ell)$
- memory $O(\min(k, \ell))$

**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)

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

Run merging algorithm: standard + many optimizations

- time $\mathcal{O}(k + \ell)$
- memory $\mathcal{O}(\min(k, \ell))$

**Policy** for choosing runs to merge:

- depends on **run lengths** only

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

Merge cost: $k + \ell$

V. Jugé

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

Algorithm based on **merging** adjacent runs

**Stable** algorithm

(good for **composite** types)

![Run merging example](image)

1. **Run merging** algorithm: standard + many optimizations
   - time $O(k + ℓ)$
   - memory $O(\min(k, ℓ))$
   - **Merge cost**: $k + ℓ$

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**

V. Jugé  Sorting presorted data
The principles of TimSort and its variants (2/2)

Run merge policy of $\alpha$-merge sort$^{[11]}$ for $\alpha = \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618$:

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

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3 \\
\end{array}\equiv\begin{array}{cccccccc}
5 & 3 & 1 & 3 & \infty
\end{array}
\]
The principles of TimSort and its variants (2/2)

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

- Find the least index $k$ such that $r_k \leq \alpha r_{k+1}$ or $r_k \leq \alpha(\alpha - 1)r_{k+2}$
- 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 \infty
```

```
0 2 2 3 4 0 1 4 5 1 2 3  \equiv  5 4 3 \infty
```
The principles of TimSort and its variants (2/2)

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

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

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 \quad 4 & 1 & 2 & 3 \\
\text{\( k = 1 \)} & & & & & \quad 5 & 3 & 1 & 3 & \infty \\
0 & 2 & 2 & 3 & 4 & 0 & 1 & 4 & 5 & 1 & 2 & 3 \\
\text{\( k = 0 \)} & & & & & \quad 5 & 4 & 3 & \infty \\
0 & 0 & 1 & 2 & 2 & 3 & 4 & 4 & 5 & 1 & 2 & 3 \\
\quad \equiv & & & & & \quad 9 & 3 & \infty
\end{array}
\]
Run merge policy of $\alpha$-merge sort\textsuperscript{[11]} for $\alpha = \phi = (1 + \sqrt{5})/2 \approx 1.618$:

- Find the least index $k$ such that $r_k \leq \alpha r_{k+1}$ or $r_k \leq \alpha (\alpha - 1) r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$
Run merge policy of $\alpha$-merge sort\[^{[11]}\] for $\alpha = \phi = (1 + \sqrt{5})/2 \approx 1.618$:

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

\[
\begin{array}{cccccccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3 \\
0 & 2 & 2 & 3 & 4 & 0 & 1 & 4 & 5 & 1 & 2 & 3 \\
0 & 0 & 1 & 2 & 2 & 3 & 4 & 4 & 5 & 1 & 2 & 3 \\
0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 4 & 4 & 5
\end{array}
\]

Merge tree

$5 \ 3 \ 1 \ 3$

$4$

$9$

$12$
The principles of TimSort and its variants (2/2)

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

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

\[\begin{array}{cccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3 \\
\end{array}\]

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$

$\equiv \infty$
The principles of TimSort and its variants (2/2)

Run merge policy of $\alpha$-merge sort\textsuperscript{[11]} for $\alpha = \phi = (1 + \sqrt{5})/2 \approx 1.618$:
- Find the least index $k$ such that $r_k \leq \alpha r_{k+1}$ or $r_k \leq \alpha(\alpha - 1)r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

![Merge tree diagram](image-url)
The principles of TimSort and its variants (2/2)

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

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

\[0 2 2 3 4 0 1 5 4 1 2 3\]

$k = 1$

\[0 2 2 3 4 0 1 4 5 1 2 3\]

$k = 0$

\[0 0 1 2 2 3 4 4 5 1 2 3\]

$k = 0$

\[0 0 1 1 2 2 2 3 3 4 4 5\]

- $k^{\text{new}} \geq k^{\text{old}} - 1$ after each merge:
  one can use **stack-based** implementations of $\alpha$-merge sort
Theorem [13]

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

Corollary:

Each run $R$ lies at depth $O\left(\frac{1}{\phi} \log \left(\frac{n}{r}\right)\right)$.

$\phi$-merge sort has a merge cost $O(n + nH)$.
Theorem [13]

In merge trees induced by $\phi$-merge sort, each node is at least $\phi$ times larger than its great-grandchildren.

Proof:

$$\circ \geq a + c \geq 2c$$

Corollary:

Each run $R$ lies at depth $O(\frac{1}{\phi} \log(\frac{n}{r}))$. $\phi$-merge sort has a merge cost $O(n + nH)$.
Fast growth in merge trees (1/2)

Theorem [13]

In merge trees induced by $\phi$-merge sort, each node is at least $\phi$ times larger than its great-grandchildren.

Proof:

\[
\begin{align*}
\circ & \geq a + c \geq 2c \\
\geq & \ a + \max\{b, c\} \\
& \geq \left(1 + \frac{1}{\phi}\right)a
\end{align*}
\]
Fast growth in merge trees (1/2)

**Theorem [13]**

In merge trees induced by $\phi$-merge sort, each node is at least $\phi$ times larger than its great-grandchildren.

**Proof:**

$$a + c \geq 2c$$

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

**Corollary:**

- Each run $R$ lies at depth $O(1 + \log(n/r))$
- $\phi$-merge sort has a merge cost $O(n + nH)$
Fast-growth property

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

- there exists an integer $k \geq 1$ and a real number $\theta > 1$ such that
- in each merge tree induced by $A$, going up $k$ times multiplies the node size by $\theta$ 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 $k \geq 1$ and a real number $\theta > 1$ such that
- in each merge tree induced by $A$,

   going up $k$ times multiplies the node size by $\theta$ or more

Theorem (continued)

Timsort$^5$, $\alpha$-merge sort$^{11}$, adaptive Shivers sort$^{12}$, Peeksort and Powersort$^{10}$ have the fast growth-property

Corollary: These algorithms work in time $O(n + nH)$
What about \[ 0 \ 1 \ 1 \ 0 \ 2 \ 1 \ 0 \ 2 \ 0 \ 2 \ 0 \ 1 \] ?

\[
\begin{align*}
5 \times 0 & \quad 4 \times 1 & \quad 3 \times 2 \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 2
\end{align*}
\]

V. Jugé

Sorting presorted data
What about \( \begin{array}{ccccccccccc}
0 & 1 & 1 & 0 & 2 & 1 & 0 & 2 & 0 & 2 & 0 & 1 \\
\end{array} \) ?

\[
\begin{array}{c}
5 \times 0 \\
4 \times 1 \\
3 \times 2 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \\
\end{array}
\]

Few \textbf{runs} vs few \textbf{values}:
What about \(0\, 1\, 1\, 0\, 2\, 1\, 0\, 2\, 0\, 2\, 0\, 1\) ?

\[
\begin{array}{ccc}
5 \times 0 & 4 \times 1 & 3 \times 2 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \\
\end{array}
\]

Few runs vs few values vs few dual runs:

V. Jugé

Sorting presorted data
Let us do better, dually!

3 dual runs of lengths 5, 4 and 3

Chunk your data in non-decreasing, non-overlapping dual runs

New parameters: Number of dual runs \((\rho^*)\) and their lengths \((r_i^*)\)

Dual-run entropy: \(H^* = \sum_{i=1}^{\rho^*} (r_i^*/n) \log_2(n/r_i^*) \leq \log_2(\rho^*) \leq \log_2(n)\)
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 \((\rho^*)\) and their lengths \((r_i^*)\)

Dual-run entropy: \(H^* = \sum_{i=1}^{\rho^*} (r_i^*/n) \log_2(n/r_i^*) \leq \log_2(\rho^*) \leq \log_2(n)\)

Theorem [13]

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

and we still cannot do better than \(\Omega(n + nH^*)\)
Fast merging procedure

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

![Diagram of merging process]
Fast merging procedure

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

Find an integer $x$ by asking $y$ and being told whether $y \geq x$:

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

Timsort merging procedure $\approx$ methods 1 + 2 with threshold $t$\textsuperscript{[4,5]:}

Ask $y = 1, 2, \ldots, t + 1, t + 2, t + 4, t + 8, \ldots$, then find the bits of $x - t$ (time $\log_2(x)$)

Merge cost: 

$$
\sum_i \log_2(1 + k_{\rightarrow i}) + \log_2(1 + \ell_{\rightarrow i})
$$

V. Jugé

Sorting presorted data
Fast merging procedure

Merging $\approx$ finding an integer (several times)$^{[3,4]}$
Fast merging procedure

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

![Image of two merged arrays](image)

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Timsort merging procedure**
  - \(\approx\) methods 1 + 2 with threshold \(t\)\[^4,5\]:
    - Ask \(y = 1, 2, 4, 8,...\)
    - Find the bits of \(x\) (time \(\log_2(x)\))

- \(\sum_i \log_2(1 + k \rightarrow i) + \log_2(1 + \ell \rightarrow i)\)
Fast merging procedure

Merging \(\approx\) finding an integer (several times)\(^{[3,4]}\)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\end{array}
\] 

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccc}
2 & 2
\end{array}
\]
Fast merging procedure

Merging $\approx$ finding an integer (several times)$^{[3,4]}$
Fast merging procedure

Merging \(\approx\) finding an integer (several times)\(^{[3,4]}\)

Finding an integer \(x\) by asking \(y\) and being told whether \(y \geq x\):

1. Ask \(y = 1, 2, 3, 4 \ldots\) (time \(x\))
Fast merging procedure

Merging ≈ finding an integer (several times)\(^{[3,4]}\)

Finding an integer \(x\) by asking \(y\) and being told whether \(y \geq x\):

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 \(\approx\) finding an integer (several times)\(^{[3,4]}\)

Finding an integer \(x\) by asking \(y\) and being told whether \(y \geq x\):

1. Ask \(y = 1, 2, 3, 4 \ldots\) \hspace{1cm} (time \(x\))
2. First ask \(y = 1, 2, 4, 8, \ldots\), then find the bits of \(x\) \hspace{1cm} (time \(2 \log_2(x)\))
3. Find \(\log_2(x)\) with method 1, then find the bits of \(x\)
Finding an integer $x$ by asking $y$ and being told whether $y \geq x$:

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

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

Finding an integer \( x \) by asking \( y \) and being told whether \( y \geq x \):

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

Timsort merging procedure \( \approx \) methods 1 + 2 with threshold \( t^{[4,5]} \):

4. Ask \( y = 1, 2, \ldots, t + 1, t + 2, t + 4, t + 8, \ldots \), then find the bits of \( x - t \)

Merge cost: \[ \sum_i \log_2(1 + k_{\rightarrow i}) + \log_2(1 + \ell_{\rightarrow i}) \]
Amortized cost evaluation

How much time is spent comparing elements from a dual run $R^*$?
Amortized cost evaluation

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

Overall, we perform at most:

\[ \text{comparisons with } R^* \]

In total:

\[ O(n + n H^*) \]
Amortized cost evaluation

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

\[
\leq w \log_2 (1 + \frac{r^*}{w})
\]

Overall, we perform at most:

\[
\sum_{i=1}^{k \log_\theta (n/r^*)} + \mathcal{O}(n + n H^*)
\]

comparisons in total.
Amortized cost evaluation

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

$$\leq w \log_2(1 + r*/w)$$

$$\leq w \log_2(1 + r*/w)$$

$$\leq r^*$$

$$\leq r^*$$

$$w \leq \frac{r^*}{\sqrt{\theta}}$$

Overall, we perform at most:

$$+$$ comparisons with $R$ and $O(n + nH^*)$ comparisons in total.
Amortized cost evaluation

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

Overall, we perform at most:

- $k \, r^* \log_\theta(n/r^*) + \sum_{h \geq 0} r^*/\theta^{h/k} \log_2(1 + \theta^{h/k})$ comparisons with $R^*$
Amortized cost evaluation

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

Overall, we perform at most:

- $O(r^* \log(n/r^*) + r^*)$ comparisons with $R^*$
- $O(n + nH^*)$ comparisons in total
Conclusion

- **TimSort** is good in practice and in theory: \( \mathcal{O}(n + nH) \) merge cost
  \( \mathcal{O}(n + nH^*) \) comparisons
Conclusion

- **TimSort** is good in practice and in theory: $O(n + nH)$ merge cost
  $O(n + nH^*)$ comparisons
- Both its **merging policy** and **merging routine** are great!
Conclusion

- **TimSort** is good in practice and in theory: $O(n + nH)$ merge cost
  $O(n + nH^*)$ 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: $O(n + nH)$ merge cost $O(n + nH^*)$ 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 $O$ notations (WIP)
Some references

MERCI POUR VOTRE ATTENTION !

NE POSEZ PAS DE QUESTIONS DIFFICILES S'IL VOUS PLAÎT !