Sorting presorted data

Vincent Jugé

LIGM – Université Gustave Eiffel, ESIEE, ENPC & CNRS

14/06/2021

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

Université Gustave Eiffel

Sharif University of Technology
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!
**Sorting data**

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

Can we do better?
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

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

**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.
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
```
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 × [0] 4 × [1] 3 × [2]

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

V. Jugé

Sorting presorted data
Let us do better!

Chunk your data in non-decreasing runs

Theorem [1,2,4,7,11] Some merge sort has a worst-case time complexity of $O(n + nH)$.

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

[4]

Reading the whole input requires a time $\Omega(n)$. There are $X$ possible reorderings, with $X \geq 2^{1 - \rho(\text{nr}1\ldots\text{nr}\rho)} \geq 2^{nH/2}$. 

V. Jugé

Sorting presorted data
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 \((r_1, \ldots, r_\rho)\)
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: $\mathcal{H} = \sum_{i=1}^{\rho} \left( \frac{r_i}{n} \right) \log_2 \left( \frac{n}{r_i} \right)$

\[ \leq \log_2(\rho) \leq \log_2(n) \]
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, 4, 7, 11]

Some merge sort has a worst-case time complexity of $O(n + n \mathcal{H})$
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 [1,2,4,7,11]**

**TimSort** has a **worst-case time complexity** of $O(n + nH)$
Let us do better!

4 runs of lengths 5, 3, 1 and 3

chunk your data in non-decreasing runs

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 [1,2,4,7,11]

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

We cannot do better than $\Omega(n + nH)!$\cite{4}

- 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 ———— for non-primitive arrays 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

1 Invented by Tim Peters[3]

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

Refined worst-case analysis[7] – TimSort works in time $O(n + nH)$

Bugs uncovered in Python & Java implementations[5,7]

V. Jugé

Sorting presorted data
A brief history of TimSort

1. Invented by Tim Peters\textsuperscript{[3]}
2. Standard algorithm in Python
   ———— for non-primitive arrays in Android, Java, Octave
A brief history of TimSort

1. Invented by Tim Peters[^3]
2. Standard algorithm in **Python**
   ————————— for non-primitive arrays in **Android, Java, Octave**
3. 1\textsuperscript{st} worst-case complexity analysis[^6] – TimSort works in time $\mathcal{O}(n \log n)$
A brief history of TimSort

1. Invented by Tim Peters[3]
2. Standard algorithm in Python
   ———— for non-primitive arrays in Android, Java, Octave
3. 1st worst-case complexity analysis[6] – TimSort works in time $O(n \log n)$
4. Refined worst-case analysis[7] – TimSort works in time $O(n + n \mathcal{H})$
A brief history of TimSort

1. Invented by Tim Peters\(^3\)
2. Standard algorithm in **Python**
   
   ———— for non-primitive arrays in **Android, Java, Octave**
3. 1\(^{st}\) worst-case complexity analysis\(^6\) – TimSort works in time \(O(n \log n)\)
4. Refined worst-case analysis\(^7\) – TimSort works in time \(O(n + n \mathcal{H})\)

Bugs uncovered in Python & Java implementations\(^{[5,7]}\)
The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

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

Algorithm based on **merging** adjacent runs

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

\[
\begin{align*}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 \\
\downarrow & & & & & & & \\
0 & 0 & 1 & 2 & 2 & 3 & 4 & 5
\end{align*}
\]
The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs  

![Diagram of TimSort algorithm]

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

1. **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)

### 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
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:** $k + \ell$

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

2. **Complexity analysis:**
   - Evaluate the **total merge cost**
   - Forget array values and only work with **run lengths**

V. Jugé

Sorting presorted data
Run merge policy of $\alpha$-merge sort$^9$ 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

\[ \begin{array}{cccccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3 & \equiv & 5 & 3 & 1 & 3 & \infty
\end{array} \]
Run merge policy of $\alpha$-merge sort\textsuperscript{[9]} 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$
The principles of TimSort and its variants (2/2)

Run merge policy of $\alpha$-merge sort\(^9\) 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3 \\
0 & 2 & 2 & 3 & 4 & 0 & 1 & 4 & 5 & 1 & 2 & 3 \\
0 & 0 & 1 & 2 & 2 & 3 & 4 & 4 & 5 & 1 & 2 & 3 \\
5 & 3 & 1 & 3 & \infty \\
5 & 4 & 3 & \infty \\
9 & 3 & \infty \\
\end{array}
\]
The principles of TimSort and its variants (2/2)

Run merge policy of $\alpha$-merge sort$^9$ 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

$$0 \ 2 \ 2 \ 3 \ 4 \ 0 \ 1 \ 5 \ 4 \ 1 \ 2 \ 3 \ \equiv \ \begin{array}{cccccccc}
0 & 2 & 2 & 3 & 4 & 0 & 1 & 5 & 4 & 1 & 2 & 3 \\
\end{array}$$

$$\begin{array}{cccccccc}
5 & 3 & 1 & 3 & \infty \\
\end{array}$$

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

$$\begin{array}{cccccccc}
5 & 4 & 3 & \infty \\
\end{array}$$

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

$$\begin{array}{cccccccc}
9 & 3 & \infty \\
\end{array}$$

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

$$\begin{array}{cccccccc}
12 & \infty \\
\end{array}$$
The principles of TimSort and its variants (2/2)

Run merge policy of $\alpha$-merge sort\[^9\] 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

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

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

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

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

\[\begin{array}{ccccccc}
5 & 3 & 1 & 3 \\
\end{array}\]

\[\begin{array}{cccc}
4 & 9 & 12 \\
\end{array}\]

\[\begin{array}{cccc}
\times & \times & \times & \times \\
\end{array}\]

\[\text{merge cost} \ \alpha \geq \phi \Rightarrow k_{\text{new}} \geq k_{\text{old}} - 1 \text{ after each merge} \Rightarrow \text{one can use stack-based implementations of } \alpha \text{-merge sort}\]

V. Jugé

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

Run merge policy of $\alpha$-merge sort\textsuperscript{[9]} 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

![Merge tree diagram]

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

![Merge tree diagram]

- Merge the runs $R_k$ and $R_{k+1}$

![Merge tree diagram]

- Merge the runs $R_k$ and $R_{k+1}$

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

Run merge policy of $\alpha$-merge sort$^{[9]}$ 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

```
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
```

Merge tree

```
12
```

```
9
```

```
4
```

```
5 3 1 3
```

$2 + 3 + 3 + 1 = \text{merge cost}$
The principles of TimSort and its variants (2/2)

Run merge policy of $\alpha$-merge sort\(^{[9]}\) 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 $r_k \leq r_{k+2}$
- Merge the runs $R_k$ and $R_{k+1}$

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

Merge tree

\[
\begin{array}{c}
\times \\
\times \\
\times \\
\times \\
\end{array}
\]

\[
= \text{merge cost}
\]

$\alpha \geq \phi \Rightarrow k^{\text{new}} \geq k^{\text{old}} - 1$ after each merge

$\Rightarrow$ one can use stack-based implementations of $\alpha$-merge sort
Fast growth in merge trees (1/2)

Theorem [11]

In merge trees induced by $\alpha$-merge sort for $\alpha \geq \phi$, each node is at least $(\alpha + 1)/\alpha$ times larger than its great-grandchildren.

Proof:

\begin{align*}
\geq a + c \\
\geq 2c \\
\geq a + \max\{b, c\} \\
\geq \left(\frac{\alpha + 1}{\alpha}\right)a
\end{align*}

Corollary:

Each run $R$ lies at depth $O(\frac{1}{\alpha} \log(\frac{n}{r}))$. $\alpha$-merge sort has a merge cost $O(n + nH)$. 

V. Jugé

Sorting presorted data
Fast growth in merge trees (1/2)

**Theorem [11]**

In merge trees induced by $\alpha$-merge sort for $\alpha \geq \phi$, each node is at least $(\alpha + 1)/\alpha$ times larger than its great-grandchildren.

**Proof:**

\[ a + c \geq 2c \]

![Diagram of merge tree structure with nodes labeled a, b, and c, illustrating the theorem's claim.]
Theorem [11]

In merge trees induced by $\alpha$-merge sort for $\alpha \geq \phi$, each node is at least $(\alpha + 1)/\alpha$ times larger than its great-grandchildren.

Proof:

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

Theorem [11]
In merge trees induced by $\alpha$-merge sort for $\alpha \geq \phi$, each node is at least $(\alpha + 1)/\alpha$ times larger than its great-grandchildren.

Proof:

Corollary:
- Each run $R$ lies at depth $O(1 + \log(n/r))$
- $\alpha$-merge sort has a merge cost $O(n + nH)$
Fast growth in merge trees (2/2)

Fast-growth property

A merge algorithm $A$ has the \textbf{fast-growth property} if

- there exists an integer $k \geq 1$ and a real number $\varepsilon > 1$ such that

in each merge tree induced by $A$, going up $k$ times multiplies the node size by $\varepsilon$ 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 $\varepsilon > 1$ such that
- in each merge tree induced by $A$,

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

Theorem (continued)

Timsort$^{[3]}$, $\alpha$-merge sort$^{[9]}$ (when $\alpha \geq \phi$), adaptive Shivers sort$^{[10]}$, Peeksort and Powersort$^{[8]}$ have the fast growth-property

Corollary: These algorithms work in time $O(n + nH)$
What about 

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

? 

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

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \\
\end{array}
\]
What about \(0\ 1\ 1\ 0\ 2\ 1\ 0\ 2\ 0\ 2\ 0\ 1\)?

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

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

Few runs vs few values:

V. Jugé  
Sorting presorted data
What about \[ 0 \ 1 \ 1 \ 0 \ 2 \ 1 \ 0 \ 2 \ 0 \ 2 \ 0 \ 1 \]?

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

\[
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 2
\]

Few **runs** vs few **values** vs few **dual runs**:
Let us do better, dually!

3 dual runs of lengths 5, 4 and 3

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

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:**
\[
\mathcal{H}^* = \sum_{i=1}^{\rho^*} \left( \frac{r_i^*}{n} \right) \log_2 \left( \frac{n}{r_i^*} \right) \\
\leq \log_2 (\rho^*) \leq \log_2 (n)
\]
Let us do better, dually!

3 dual runs of lengths 5, 4 and 3

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: $\mathcal{H}^* = \sum_{i=1}^{\rho^*} \left( r_i^* / n \right) \log_2 \left( n / r_i^* \right) \\ \leq \log_2 (\rho^*) \leq \log_2 (n)$

Theorem [11]
Every fast-growth merge sort requires $O(n + n \mathcal{H}^*)$ comparisons if it uses Timsort’s optimized run-merging routine

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

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

Some references:

Conclusion

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

- Both its **merging policy** and **merging routine** are great!
Conclusion

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

- Both its **merging policy** and **merging routine** are great!

Some references:

1. *Optimal computer search trees and variable-length alphabetical codes*, Hu & Tucker (1971)
5. *OpenJDK’s java.util.Collection.sort() is broken*, de Gouw et al. (2015)
MERCI POUR VOTRE ATTENTION !

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