On the Worst-Case Complexity of TimSort

Jan Kukowski, Szymon Wojtulewicz

Marzec 2024

On the Worst-Case Complexity of TimSort

https://drops.dagstuhl.de/storage/00lipics/lipics-vol112-esa2018/LIPIcs.ESA.2018.4/LIPIcs.ESA.2018.4.pdf

Geneza powstania

"Among Python users the most frequent complaint I've heard is that list.sort() isn't stable."

Geneza powstania

"Among Python users the most frequent complaint I've heard is that list.sort() isn't stable."

"I couldn't resist taking a crack at a new algorithm that might be practical, and have something you might call a non-recursive adaptive stable natural mergesort / binary insertion sort hybrid."

Geneza powstania

"Among Python users the most frequent complaint I've heard is that list.sort() isn't stable."

"I couldn't resist taking a crack at a new algorithm that might be practical, and have something you might call a non-recursive adaptive stable natural mergesort / binary insertion sort hybrid."

— Tim Peters

https://mail.python.org/pipermail/python-dev/2002-July/026837.html

Oś czasu

- 2002 - *TimSort* jest wynaleziony

Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*

Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*
- 2011 - *TimSort* domyślnym algorytmem w *javie*

Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*
- 2011 - *TimSort* domyślnym algorytmem w *javie*
- 2015 - odkryto, że *TimSort* zawiera błąd
  http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf

Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*
- 2011 - *TimSort* domyślnym algorytmem w *javie*
- 2015 - odkryto, że *TimSort* zawiera błąd
  `http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf`
- Grudzień 2015 - preprint omawianej pracy i pierwszy dowód złożoności $\mathcal{O}(n \log n)$

Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*
- 2011 - *TimSort* domyślnym algorytmem w *javie*
- 2015 - odkryto, że *TimSort* zawiera błąd
  `http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf`
- Grudzień 2015 - preprint omawianej pracy i pierwszy dowód złożoności $\mathcal{O}(n \log n)$
- 2018 - publikacja pracy i dowód złożoności $\mathcal{O}(n + n \log \rho)$

Co pokażemy

- *TimSort* działa w czasie $\mathcal{O}(n \log n)$

Co pokażemy

- *TimSort* działa w czasie $\mathcal{O}(n \log n)$
- *TimSort* działa w czasie $\mathcal{O}(n + n \log \rho)$

Co pokażemy

- *TimSort* działa w czasie $\mathcal{O}(n \log n)$
- *TimSort* działa w czasie $\mathcal{O}(n + n \log \rho)$

Oczywiście $\mathcal{O}(n + n \log \rho)$ implikuje $\mathcal{O}(n \log n)$

Czym jest ρ

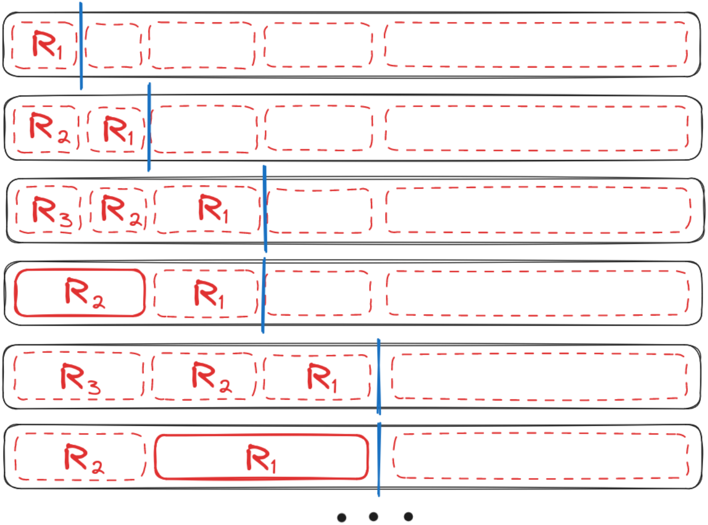
Czym jest ρ

- Algorytm opiera się na podziale danych wejściowych na *monotoniczne podciągi*/*runs*.

Czym jest ρ

- Algorytm opiera się na podziale danych wejściowych na *monotoniczne podciągi/runs*.
- ρ to liczba tych podciągów.

Schemat działania



Pseudokod

Input: A sequence S to sort

Result: The sequence S is sorted into a single run, which remains on the stack.

Note: At any time, we denote the height of the stack $\mathcal{R}$ by h and its i^{th} top-most run (for $1 \leqslant i \leqslant h$) by R_i. The size of this run is denoted by r_i.

```
runs ← the run decomposition of S
ℛ ← an empty stack
while runs ≠ ∅ do                                           // main loop of TimSort
    remove a run r from runs and push r onto ℛ                                  // #1
    while true do
        if h ⩾ 3 and r1 > r3 then merge the runs R2 and R3                      // #2
        else if h ⩾ 2 and r1 ⩾ r2 then merge the runs R1 and R2                 // #3
        else if h ⩾ 3 and r1 + r2 ⩾ r3 then merge the runs R1 and R2            // #4
        else if h ⩾ 4 and r2 + r3 ⩾ r4 then merge the runs R1 and R2            // #5
        else break
while h ≠ 1 do merge the runs R1 and R2
```

TimSort działa w czasie $\mathcal{O}(n \log n)$

Każdy element otrzymuje 2 $\diamondsuit$-tokeny i 1 $\heartsuit$-token, gdy jest wstawiany na stos w #1, oraz za każdym razem gdy zmniejsza się jego wysokość w stosie

TimSort działa w czasie $\mathcal{O}(n \log n)$

Każdy element otrzymuje 2 $\diamondsuit$-tokeny i 1 $\heartsuit$-token, gdy jest wstawiany na stos w #1, oraz za każdym razem gdy zmniejsza się jego wysokość w stosie

- #2 Każdy element R_1 oraz R_2 płaci 1 $\diamondsuit$-token. Wystarczy tokenów ponieważ $r_3 < r_1$, więc $r_2 + r_3 \le r_1 + r_2$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Każdy element otrzymuje 2 $\diamondsuit$-tokeny i 1 $\heartsuit$-token, gdy jest wstawiany na stos w #1, oraz za każdym razem gdy zmniejsza się jego wysokość w stosie

- #2 Każdy element R_1 oraz R_2 płaci 1 $\diamondsuit$-token. Wystarczy tokenów ponieważ $r_3 < r_1$, więc $r_2 + r_3 \leq r_1 + r_2$
- #3 Każdy element R_1 płaci 2 $\diamondsuit$-tokeny. Wystarczy tokenów ponieważ $r_2 \leq r_1$, więc $r_2 + r_1 \leq 2r_1$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Każdy element otrzymuje 2 $\diamondsuit$-tokeny i 1 $\heartsuit$-token, gdy jest wstawiany na stos w #1, oraz za każdym razem gdy zmniejsza się jego wysokość w stosie

- #2 Każdy element R_1 oraz R_2 płaci 1 $\diamondsuit$-token. Wystarczy tokenów ponieważ $r_3 < r_1$, więc $r_2 + r_3 \leq r_1 + r_2$
- #3 Każdy element R_1 płaci 2 $\diamondsuit$-tokeny. Wystarczy tokenów ponieważ $r_2 \leq r_1$, więc $r_2 + r_1 \leq 2r_1$
- #4 i #5 Każdy element R_1 płaci 1 $\diamondsuit$-token, a każdy element R_2 płaci 1 $\heartsuit$-token. Wydajemy dokładnie $r_1 + r_2$ tokenów

TimSort działa w czasie $\mathcal{O}(n \log n)$

Każdy element otrzymuje 2 $\diamondsuit$-tokeny i 1 $\heartsuit$-token, gdy jest wstawiany na stos w #1, oraz za każdym razem gdy zmniejsza się jego wysokość w stosie

- #2 Każdy element R_1 oraz R_2 płaci 1 $\diamondsuit$-token. Wystarczy tokenów ponieważ $r_3 < r_1$, więc $r_2 + r_3 \leq r_1 + r_2$
- #3 Każdy element R_1 płaci 2 $\diamondsuit$-tokeny. Wystarczy tokenów ponieważ $r_2 \leq r_1$, więc $r_2 + r_1 \leq 2r_1$
- #4 i #5 Każdy element R_1 płaci 1 $\diamondsuit$-token, a każdy element R_2 płaci 1 $\heartsuit$-token. Wydajemy dokładnie $r_1 + r_2$ tokenów

Lemma (4)

W każdym obrocie głównej pętli zachowujemy nieujemny bilans $\diamondsuit$-tokenów i $\heartsuit$-tokenów

TimSort działa w czasie $\mathcal{O}(n \log n)$

Lemma (5)

W każdym obrocie głównej pętli zachodzą 4 niezmienniki:

1. $r_i + r_{i+1} < r_{i+2}$ *dla* $i \in \{3, \ldots, h-2\}$
2. $r_2 < 3r_3$
3. $r_3 < r_4$
4. $r_2 < r_3 + r_4$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Dowód.

Jeśli weszliśmy w #1, to żaden z warunków z #2 - #5 nie zachodził w $\mathcal{S}$. W szególności zachodzi $r_1 < r_2 < r_3$ oraz $r_2 + r_3 < r_4$. Po dodaniu nowego run mamy $\overline{r}_2 < \overline{r}_3 < \overline{r}_4$ co daje niezmienniki $2 - 4$ oraz $\overline{r}_3 + \overline{r}_4 < \overline{r}_5$ co daje niezmiennik 1

TimSort działa w czasie $\mathcal{O}(n \log n)$

Jeśli weszliśmy w jeden z #2 - #5, to $\overline{r}_2 = r_2 + r_3$ (dla #2) albo $\overline{r}_2 = r_3$ (dla #3 - #5). W takim razie $\overline{r}_2 \leq r_2 + r_3$.

1. Spełniony bo $\overline{r}_i = r_{i+1}$ dla $i \geq 3$
2. $\overline{r}_2 \leq r_2 + r_3 < r_3 + r_4 + r_3 < 3r_4 = 3\overline{r}_3$
3. $\overline{r}_3 = r_4 \leq r_3 + r_4 < r_5 = \overline{r}_4$
4. $\overline{r}_2 \leq r_2 + r_3 < r_3 + r_4 + r_3 < r_3 + r_5 < r_4 + r_5 = \overline{r}_3 + \overline{r}_4$

$\square$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Lemma (6)

$r_2/3 < r_3 < r_4 < r_5 < \cdots < r_h$ oraz dla $k \geq i \geq 3$ zachodzi $r_k > \sqrt{2}^{k-i-1} r_i$. Wobec tego, rozmiar stosu jest $\mathcal{O}(logn)$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Lemma (6)

$r_2/3 < r_3 < r_4 < r_5 < \cdots < r_h$ oraz dla $k \geq i \geq 3$ zachodzi $r_k > \sqrt{2}^{k-i-1} r_i$. Wobec tego, rozmiar stosu jest $\mathcal{O}(logn)$

Dowód.

- Z niezmiennika 1. $r_i + r_{i+1} < r_{i+2}$ dla $i \in \{3, \ldots, h-2\}$, więc $r_{i+2} - r_{i+1} > r_i > 0$, czyli $r_4 < r_5 < \cdots < r_h$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Lemma (6)

$r_2/3 < r_3 < r_4 < r_5 < \cdots < r_h$ oraz dla $k \geq i \geq 3$ zachodzi $r_k > \sqrt{2}^{k-i-1} r_i$. Wobec tego, rozmiar stosu jest $\mathcal{O}(logn)$

Dowód.

- Z niezmiennika 1. $r_i + r_{i+1} < r_{i+2}$ dla $i \in \{3, \ldots, h-2\}$, więc $r_{i+2} - r_{i+1} > r_i > 0$, czyli $r_4 < r_5 < \cdots < r_h$
- Z niezmiennikami 2. i 3. uzyskujemy $r_2/3 < r_3 < r_4 < r_5 < \cdots < r_h$

TimSort działa w czasie $\mathcal{O}(n \log n)$

Lemma (6)

$r_2/3 < r_3 < r_4 < r_5 < \cdots < r_h$ oraz dla $k \geq i \geq 3$ zachodzi $r_k > \sqrt{2}^{k-i-1} r_i$. Wobec tego, rozmiar stosu jest $\mathcal{O}(logn)$

Dowód.

- Z niezmiennika 1. $r_i + r_{i+1} < r_{i+2}$ dla $i \in \{3, \ldots, h-2\}$, więc $r_{i+2} - r_{i+1} > r_i > 0$, czyli $r_4 < r_5 < \cdots < r_h$
- Z niezmiennikami 2. i 3. uzyskujemy $r_2/3 < r_3 < r_4 < r_5 < \cdots < r_h$
- Otrzymujemy $r_{j+2} > 2r_j$, a więc $r_k > \sqrt{2}^{k-i-1} r_i$ i w takim razie ilość runs w stosie jest ograniczona $\mathcal{O}(logn)$

□

TimSort działa w czasie $\mathcal{O}(n + n \log \rho)$

Input: A sequence S to sort

`runs` $\leftarrow$ the run decomposition of S

$\mathcal{R} \leftarrow$ an empty stack

while *runs* $\neq \emptyset$ **do** // main loop of *TimSort*

    remove a run r from `runs` and push r onto $\mathcal{R}$ // #1

    while ***true*** **do**

        if $h \geqslant 3$ *and* $r_1 > r_3$ **then** merge the runs R_2 and R_3 // #2

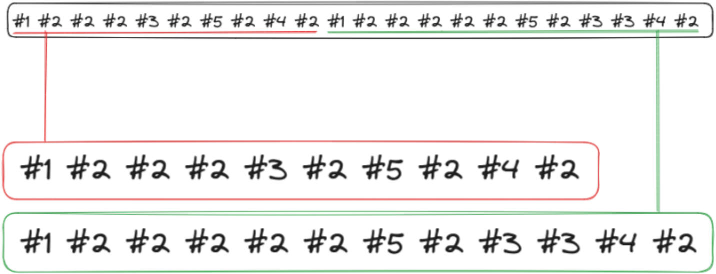
        else if $h \geqslant 2$ *and* $r_1 \geqslant r_2$ **then** merge the runs R_1 and R_2 // #3

        else if $h \geqslant 3$ *and* $r_1 + r_2 \geqslant r_3$ **then** merge the runs R_1 and R_2 // #4

        else if $h \geqslant 4$ *and* $r_2 + r_3 \geqslant r_4$ **then** merge the runs R_1 and R_2 // #5

        else break

Podział na sekwencje



Podział na sekwencje

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

- Kładziemy na stos *run* R o długości $r \overset{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

- Kładziemy na stos *run* R o długości $r \overset{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.
- $\mathcal{S} = (R_1, \ldots, R_h) \xrightarrow{R} \overline{\mathcal{S}} = (R, R_1, \ldots, R_h)$

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

- Kładziemy na stos *run* R o długości $r \stackrel{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.
- $\mathcal{S} = (R_1, \ldots, R_h) \xrightarrow{R} \overline{\mathcal{S}} = (R, R_1, \ldots, R_h)$

$$\underbrace{\#1 \; \#2 \; \#2 \; \ldots \; \#2 \; \#2}_{k}$$

-

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

- Kładziemy na stos *run* R o długości $r \overset{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.
- $\mathcal{S} = (R_1, \dots, R_h) \xrightarrow{R} \overline{\mathcal{S}} = (R, R_1, \dots, R_h)$

$$\underbrace{\boxed{\#1\ \ \#2\ \ \#2\ \ \dots\ \ \#2\ \ \#2}}_{k}$$

-
- $\mathcal{C} = (k-1)r_1 + (k-1)r_2 + (k-2)r_3 + \ldots + r_k \leqslant \sum_{i=1}^{k}(k+1-i)r_i$.

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

- Kładziemy na stos *run* R o długości $r \overset{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.
- $\mathcal{S} = (R_1, \dots, R_h) \xrightarrow{R} \overline{\mathcal{S}} = (R, R_1, \dots, R_h)$

$$\underbrace{\boxed{\#1\ \ \#2\ \ \#2\ \ \dots\ \ \#2\ \ \#2}}_{k}$$

-
- $\mathcal{C} = (k-1)r_1 + (k-1)r_2 + (k-2)r_3 + \dots + r_k \leqslant \sum_{i=1}^{k}(k+1-i)r_i.$
- $\dots \#2 \implies r > r_k$

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

- Kładziemy na stos *run* R o długości $r \overset{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.
- $\mathcal{S} = (R_1, \ldots, R_h) \xrightarrow{R} \overline{\mathcal{S}} = (R, R_1, \ldots, R_h)$

$$\underbrace{\boxed{\#1\ \#2\ \#2\ \ldots\ \#2\ \#2}}_{k}$$

-
- $\mathcal{C} = (k-1)r_1 + (k-1)r_2 + (k-2)r_3 + \ldots + r_k \leqslant \sum_{i=1}^{k}(k+1-i)r_i$.
- $\ldots \#2 \implies r > r_k$
- Z części $\mathcal{O}(n \log n)$: $r \geqslant r_k \geqslant \sqrt{2}^{k-i} r_i / 3$

Liczba porównań w trakcie *sekwencji startowych* jest $\mathcal{O}(n)$

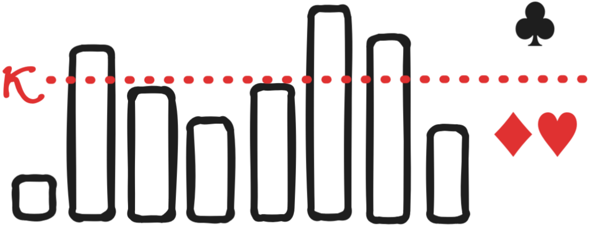
- Kładziemy na stos *run* R o długości $r \overset{?}{\Longrightarrow} \mathcal{O}(r)$ porównań.
- $\mathcal{S} = (R_1, \dots, R_h) \xrightarrow{R} \overline{\mathcal{S}} = (R, R_1, \dots, R_h)$

  $$\underbrace{\boxed{\#1\ \#2\ \#2\ \dots\ \#2\ \#2}}_{k}$$

  -
- $\mathcal{C} = (k-1)r_1 + (k-1)r_2 + (k-2)r_3 + \dots + r_k \leqslant \sum_{i=1}^{k}(k+1-i)r_i$.
- $\dots \#2 \Longrightarrow r > r_k$
- Z części $\mathcal{O}(n \log n)$: $r \geqslant r_k \geqslant \sqrt{2}^{k-i} r_i/3$
- $$\mathcal{C}/r \leqslant 3\sum_{i=1}^{k}(k+1-i)/\sqrt{2}^{k-i} \leqslant 3\sqrt{2}\sum_{i \geqslant 0} i/\sqrt{2}^{i} = \gamma$$

Liczba porównań w trakcie *sekw. końcowych* jest $\mathcal{O}(n \log \rho)$

$\kappa = \lceil 2 \log_2 \rho \rceil$



Globalna pula $24n\clubsuit$ tokenów

Liczba porównań w trakcie *sekw. końcowych* jest $\mathcal{O}(n \log \rho)$

Lemma (10)

Merge podczas sekwencji końcowej, gdzie wysokość stosu $h \geq \kappa$*, kosztuje co najwyżej* $24n/\rho$ *porównań.*

Liczba porównań w trakcie *sekw. końcowych* jest $\mathcal{O}(n \log \rho)$

Lemma (10)

Merge podczas sekwencji końcowej, gdzie wysokość stosu $h \geq \kappa$, kosztuje co najwyżej $24n/\rho$ porównań.

Dowód.

Mamy $r_2 < 3r_3$ (*Lemma 5*) i $r_1 < 3r_2$. *Merge* R_1 z R_2 lub R_2 z R_3 kosztuje co najwyżej $6r_3$ porównań.

$r_h \geq \sqrt{2}^{h-4} r_3 \implies 6r_3 \leq 24\sqrt{2}^{-h} r_h \leq 24n\sqrt{2}^{-\kappa} \leq 24n/\rho$ □

Liczba porównań w trakcie *sekw. końcowych* jest $\mathcal{O}(n \log \rho)$

Lemma (10)

Merge podczas sekwencji końcowej, gdzie wysokość stosu $h \geq \kappa$, kosztuje co najwyżej $24n/\rho$ porównań.

Dowód.

Mamy $r_2 < 3r_3$ (*Lemma 5*) i $r_1 < 3r_2$. *Merge* R_1 z R_2 lub R_2 z R_3 kosztuje co najwyżej $6r_3$ porównań.

$$r_h \geq \sqrt{2}^{h-4} r_3 \implies 6r_3 \leq 24\sqrt{2}^{-h} r_h \leq 24n\sqrt{2}^{-\kappa} \leq 24n/\rho \qquad \square$$

Ponieważ liczba *merge*'y opłacanych przez ♣ jest ograniczona przez ρ to przyznaliśmy wystarczającą liczbę tych tokenów.

Błąd w implementacji w Javie

Input: A sequence S to sort

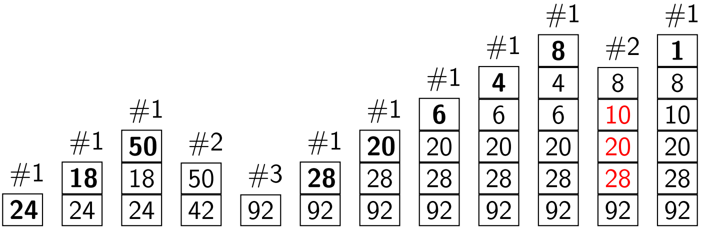
Result: The sequence S is sorted into a single run, which remains on the stack.

Note: At any time, we denote the height of the stack $\mathcal{R}$ by h and its i^{th} top-most run (for $1 \leqslant i \leqslant h$) by R_i. The size of this run is denoted by r_i.

runs $\leftarrow$ the run decomposition of S
$\mathcal{R} \leftarrow$ an empty stack
while *runs* $\neq \varnothing$ **do** // main loop of *TimSort*
    remove a run r from runs and push r onto $\mathcal{R}$ // #1
    while ***true*** **do**
        if $h \geqslant 3$ *and* $r_1 > r_3$ **then** merge the runs R_2 and R_3 // #2
        else if $h \geqslant 2$ *and* $r_1 \geqslant r_2$ **then** merge the runs R_1 and R_2 // #3
        else if $h \geqslant 3$ *and* $r_1 + r_2 \geqslant r_3$ **then** merge the runs R_1 and R_2 // #4
        else if $h \geqslant 4$ *and* $r_2 + r_3 \geqslant r_4$ **then** merge the runs R_1 and R_2 // #5
        else break
while $h \neq 1$ **do** merge the runs R_1 and R_2

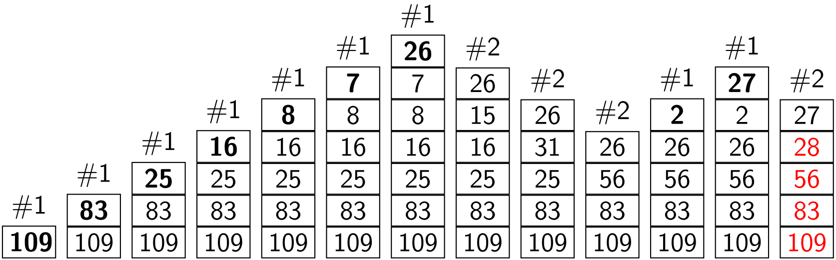
Błąd w implementacji w Javie

Pomijając warunek #5 nie otrzymujemy niezmiennika $r_i + r_{i+1} < r_{i+2}$



Błąd w implementacji w Javie

Co więcej, błąd może się powtarzać na kilku indeksach z rzędu



Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*
- 2011 - *TimSort* domyślnym algorytmem w *javie*
- 2015 - odkryto, że *TimSort* zawiera błąd
  `http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf`
- Grudzień 2015 - preprint omawianej pracy i pierwszy dowód złożoności $\mathcal{O}(n \log n)$
- 2018 - publikacja pracy i dowód złożoności $\mathcal{O}(n + n \log \rho)$

Oś czasu

- 2002 - *TimSort* jest wynaleziony
- 2003 - *TimSort* domyślnym algorytmem w *pythonie*
- 2011 - *TimSort* domyślnym algorytmem w *javie*
- 2015 - odkryto, że *TimSort* zawiera błąd
  http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf
- Grudzień 2015 - preprint omawianej pracy i pierwszy dowód złożoności $\mathcal{O}(n \log n)$
- 2018 - publikacja pracy i dowód złożoności $\mathcal{O}(n + n \log \rho)$
- **2022 - wraz z *pythonem 3.11* *TimSort* przestaje być domyślnym algorytmem dla tego języka :((**

Dziękujemy za uwagę!