On the Worst-Case Complexity of Timsort

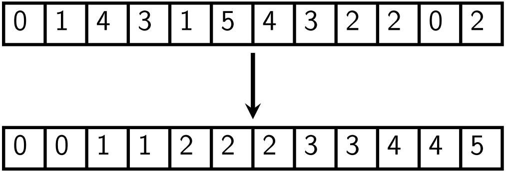
Nicolas Auger, Vincent Jugé, Cyril Nicaud & Carine Pivoteau

LIGM – Université Paris-Est Marne-la-Vallée & CNRS

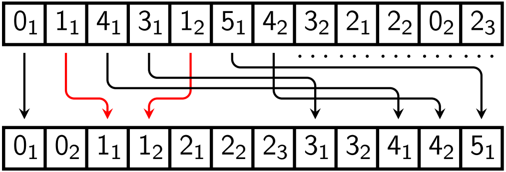
20/08/2018

Contents

Sorting data



Sorting data – in a stable manner



Sorting data – in a stable manner

0_1	1_1	4_1	3_1	1_2	5_1	4_2	3_2	2_1	2_2	0_2	2_3

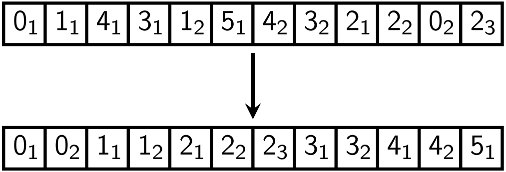
$$\downarrow$$

0_1	0_2	1_1	1_2	2_1	2_2	2_3	3_1	3_2	4_1	4_2	5_1

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

Can we do better?

Sorting data – in a stable manner



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

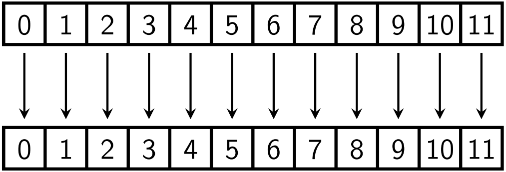
Can we do better? No!

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

Cannot we ever do better?

In some cases, we should. . .



Let us do better!

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

1. Chunk your data in **monotonic runs**

Let us do better!

4 runs of lengths 3, 2, 6 and 1

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

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

Let us do better!

4 runs of lengths **3**, **2**, **6** and **1**

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

1. Chunk your data in **monotonic runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)
   Run-length entropy: $\mathcal{H} = \sum_{k=1}^{\rho} (r_i/n) \log_2(n/r_i)$
   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

Let us do better!

4 runs of lengths 3, 2, 6 and 1

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

1. Chunk your data in **monotonic runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)
   Run-length entropy: $\mathcal{H} = \sum_{k=1}^{\rho} (r_i/n) \log_2(n/r_i)$
   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

Theorem (Auger – Jugé – Nicaud – Pivoteau 2018)

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

Let us do better!

4 runs of lengths **3**, **2**, **6** and **1**

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

1. Chunk your data in **monotonic runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** $(r_1, \ldots, r_\rho)$
   Run-length entropy: $\mathcal{H} = \sum_{k=1}^{\rho} (r_i/n) \log_2(n/r_i)$
   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

Theorem (Auger – Jugé – Nicaud – Pivoteau 2018)

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

Let us do better!

4 runs of lengths **3**, **2**, **6** and **1**

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

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

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

Theorem (Auger – Jugé – Nicaud – Pivoteau 2018)

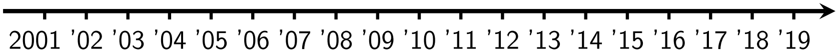
Timsort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

We cannot do better than $\Omega(n + n\,\mathcal{H})$![2]

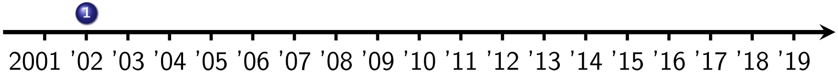
- Reading the whole input requires a time $\Omega(n)$
- There are **X** possible reorderings, with $\mathbf{X} \geqslant 2^{1-\rho}\binom{n}{r_1 \ldots r_\rho} \geqslant 2^{n\,\mathcal{H}/2}$

Contents

A brief history of Timsort

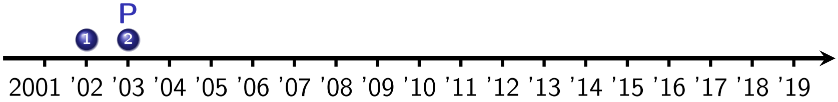


A brief history of Timsort



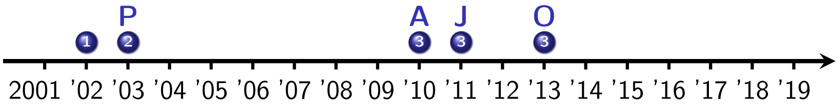
1. Invented by **Tim Peters**[1]

A brief history of Timsort



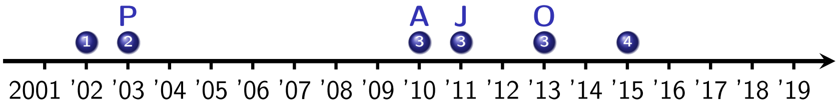
1. Invented by **Tim Peters**[1]
2. Standard algorithm in **Python**

A brief history of Timsort



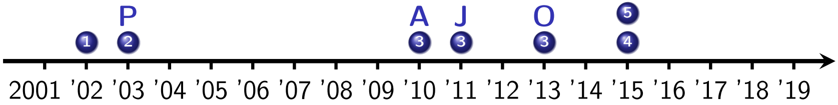
1. Invented by **Tim Peters**[1]
2. Standard algorithm in **Python**
3. ———————— for non-primitive arrays in **Android**, **Java**, **Octave**

A brief history of Timsort



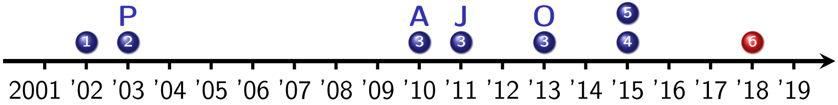
1. Invented by **Tim Peters**[1]
2. Standard algorithm in **Python**
3. ———————— for non-primitive arrays in **Android**, **Java**, **Octave**
4. Stack size bug uncovered – a provably correct fix is suggested:[3]
   - suggested fix implemented in Python (**true** Timsort)
   - custom fix implemented in Java (**Java** Timsort)

A brief history of Timsort



1. Invented by **Tim Peters**[1]
2. Standard algorithm in **Python**
3. ——————— for non-primitive arrays in **Android**, **Java**, **Octave**
4. Stack size bug uncovered – a provably correct fix is suggested:[3]
   - suggested fix implemented in Python (**true** Timsort)
   - custom fix implemented in Java (**Java** Timsort)
5. 1st worst-case complexity analysis[4] – Timsort works in time $\mathcal{O}(n \log n)$

A brief history of Timsort

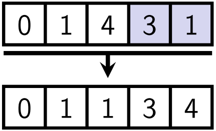


1. Invented by **Tim Peters**[1]
2. Standard algorithm in **Python**
3. ——————— for non-primitive arrays in **Android**, **Java**, **Octave**
4. Stack size bug uncovered – a provably correct fix is suggested:[3]
   - suggested fix implemented in Python (**true** Timsort)
   - custom fix implemented in Java (**Java** Timsort)
5. 1st worst-case complexity analysis[4] – Timsort works in time $\mathcal{O}(n \log n)$
6. Another stack size bug uncovered (Java version)

   Refined worst-case analysis: both versions work in time $\mathcal{O}(n + n\mathcal{H})$

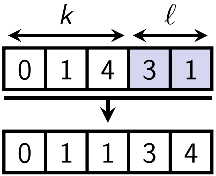
The principles of Timsort (1/3)

Algorithm based on **merging** adjacent runs



The principles of Timsort (1/3)

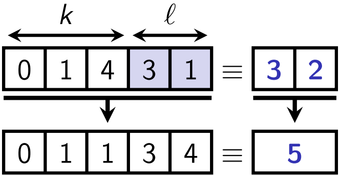
Algorithm based on **merging** adjacent runs



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

The principles of Timsort (1/3)

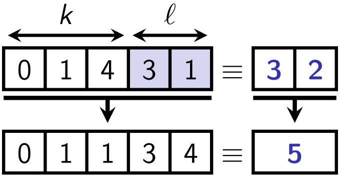
Algorithm based on **merging** adjacent runs



1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k + \ell)$
   - memory $\mathcal{O}(\min(k, \ell))$
2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only

The principles of Timsort (1/3)

Algorithm based on **merging** adjacent runs



1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k + \ell)$
   - memory $\mathcal{O}(\min(k, \ell))$
2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only

Let us forget array values – only remember **run lengths**!

The principles of Timsort (2/3)

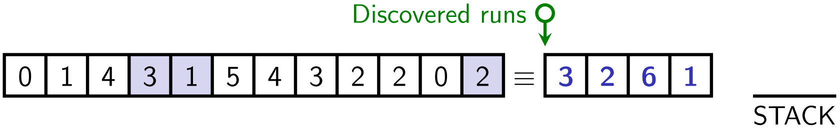
0	1	4	3	1	5	4	3	2	2	0	2	$\equiv$	**3**	**2**	**6**	**1**

STACK

Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1[st] and 2[nd] runs
  3. merge the top 2[nd] and 3[rd] runs

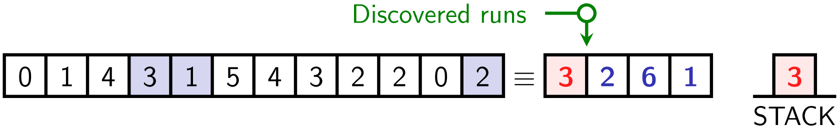
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

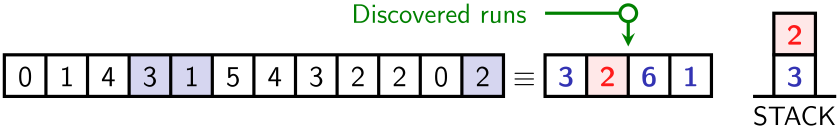
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3nd runs

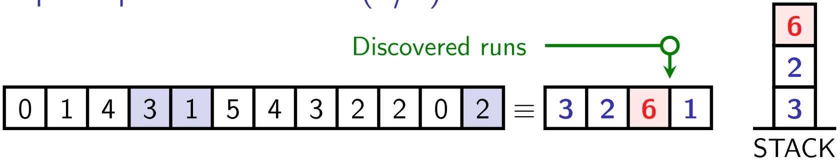
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

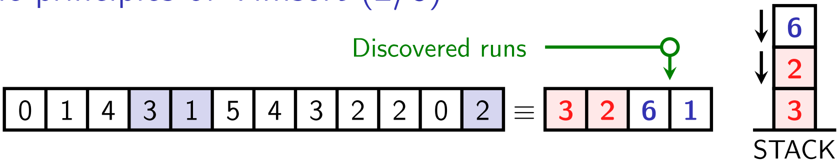
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

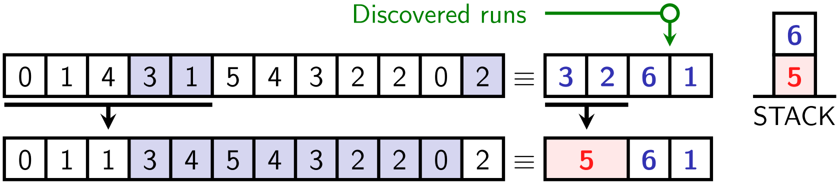
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1[st] and 2[nd] runs
  3. merge the top 2[nd] and 3[nd] runs

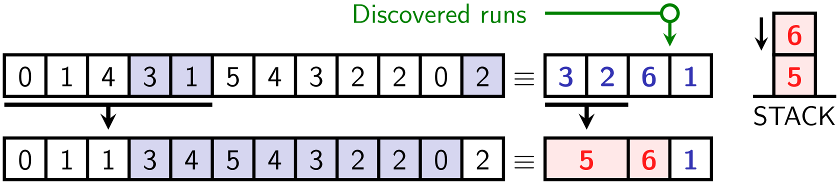
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3nd runs

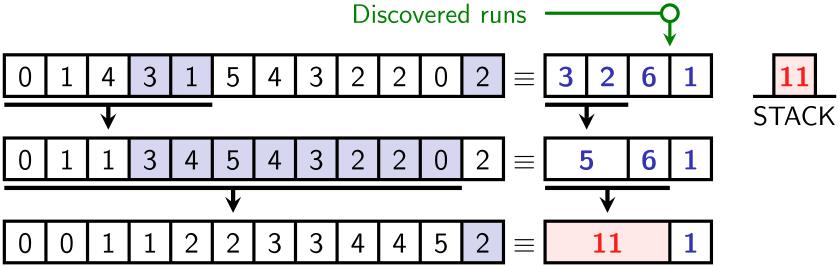
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

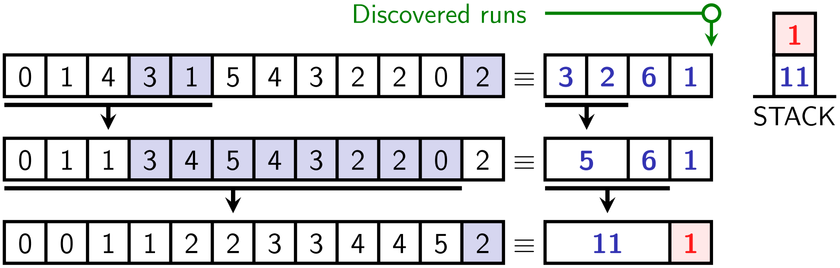
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

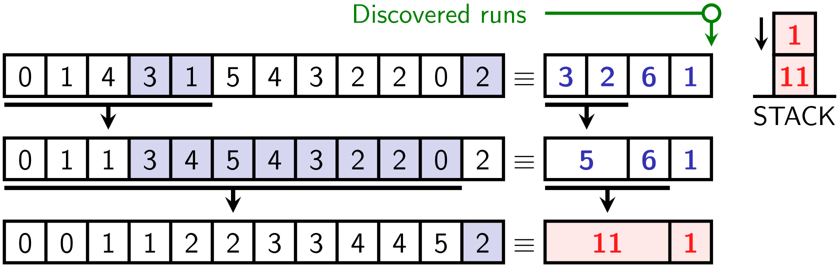
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

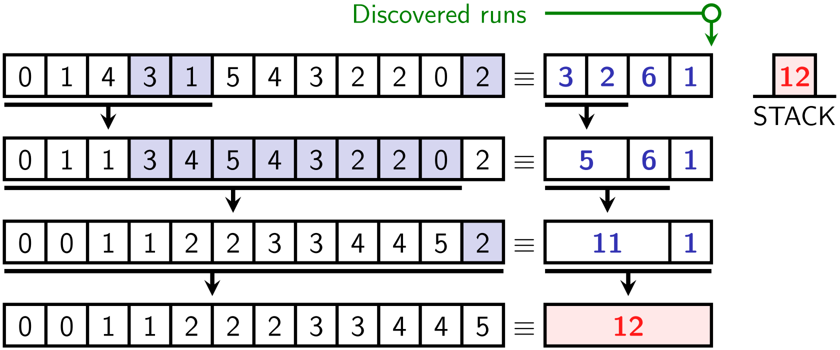
The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

The principles of Timsort (2/3)



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
  1. discover & push a new run length onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

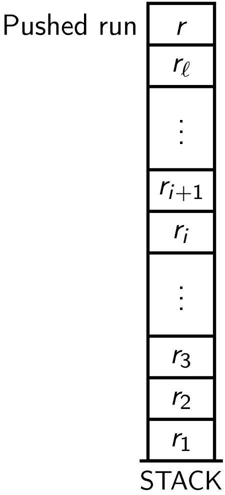
Intermezzo: Intelligent design & amortized analysis

Key ideas:

Intermezzo: Intelligent design & amortized analysis

Key ideas:

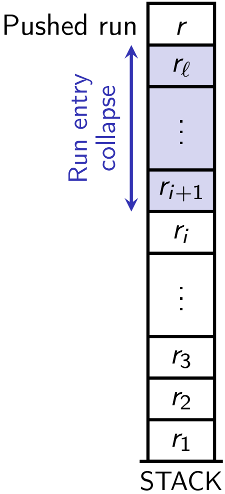
- Each run r **pays** $\mathcal{O}(r)$ to
  - **enter** the stack (before its 1st merge)
  - **go down** 1 floor (after its 1st merge)



Intermezzo: Intelligent design & amortized analysis

Key ideas:

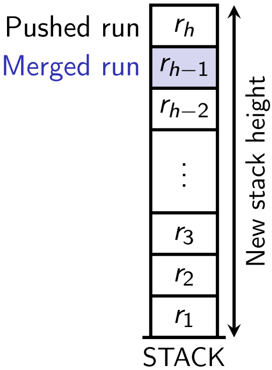
- Each run r **pays** $\mathcal{O}(r)$ to
  - **enter** the stack (before its 1st merge)
  - **go down** 1 floor (after its 1st merge)
- Stack **height** $h = \mathcal{O}(\log(n/r))$ when the **run entry phase** ends



Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays** $\mathcal{O}(r)$ to
  - **enter** the stack (before its 1st merge)
  - **go down** 1 floor (after its 1st merge)
- Stack **height** $h = \mathcal{O}(\log(n/r))$ when the **run entry phase** ends



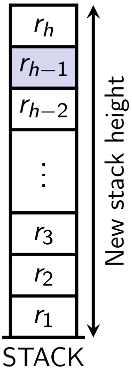
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays** $\mathcal{O}(r)$ to
  - **enter** the stack (before its 1st merge)
  - **go down** 1 floor (after its 1st merge)
- Stack **height** $h = \mathcal{O}(\log(n/r))$ when the **run entry phase** ends
- Ensure that
  - $(r_i)_{i \geqslant 1}$ has **exponential** decay when r is pushed
  - $r = r_h \leqslant r_{h-\mathcal{O}(1)}$ when the **run entry phase** ends

Pushed run

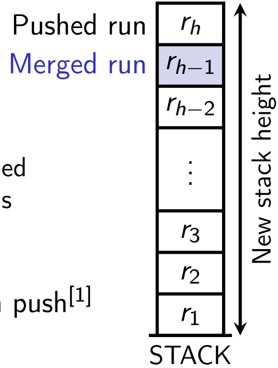
Merged run



Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays** $\mathcal{O}(r)$ to
  - **enter** the stack (before its 1st merge)
  - **go down** 1 floor (after its 1st merge)
- Stack **height** $h = \mathcal{O}(\log(n/r))$ when the **run entry phase** ends
- Ensure that
  - $(r_i)_{i \geqslant 1}$ has **exponential** decay when r is pushed
  - $r = r_h \leqslant r_{h-2}$ when the **run entry phase** ends



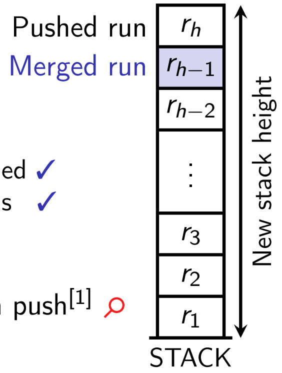
Implementation in **Timsort**:

- **Fibonacci constraints** $r_i > r_{i+1} + r_{i+2}$ on run push[1]
- Merge r_{h-2} and r_{h-1} whenever $r_{h-2} \leqslant r_h$

Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays** $\mathcal{O}(r)$ to
  - **enter** the stack (before its 1st merge) ✓
  - **go down** 1 floor (after its 1st merge) 🔍
- Stack **height** $h = \mathcal{O}(\log(n/r))$ when the **run entry phase** ends ✓
- Ensure that
  - $(r_i)_{i \geqslant 1}$ has **exponential** decay when r is pushed ✓
  - $r = r_h \leqslant r_{h-2}$ when the **run entry phase** ends ✓



Implementation in **Timsort**:

- **Fibonacci constraints** $r_i > r_{i+1} + r_{i+2}$ on run push[1] 🔍
- Merge r_{h-2} and r_{h-1} whenever $r_{h-2} \leqslant r_h$ ✓

The principles of Timsort (3/3)

Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1$^{\text{st}}$ and 2$^{\text{nd}}$ runs
3. merge the top 2$^{\text{nd}}$ and 3$^{\text{nd}}$ runs

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

The principles of Timsort (3/3)

Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

Fibonacci constraints:

- $r_i > r_{i+1} + r_{i+2}$ for all $i \leqslant h - 4$ (induction)
- $r_i > r_{i+1} + r_{i+2}$ for $i \geqslant h - 3$ on run push

The principles of Timsort (3/3)

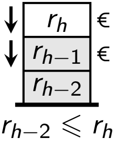
Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

Choice algorithm

if $\underline{r_{h-2} \leqslant r_h}$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

Making runs pay for going down:



The principles of Timsort (3/3)

Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1^{st} and 2^{nd} runs
3. merge the top 2^{nd} and 3^{nd} runs

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $\underline{r_{h-1} \leqslant r_h}$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

Making runs pay for going down:

The principles of Timsort (3/3)

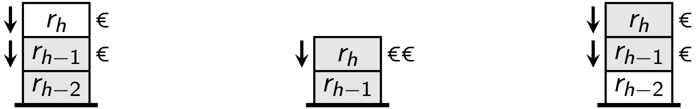
Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $\underline{r_{h-2} \leqslant r_{h-1} + r_h}$ or $\underline{r_{h-3} \leqslant r_{h-2} + r_{h-1}}$: choose ②
else: choose ① (or ② if ① is unavailable)

Making runs pay for going down:



$r_{h-2} \leqslant r_h$ | $r_{h-1} \leqslant r_h$ | $r_{h-2} \leqslant r_{h-1} + r_h$, $r_{h-3} \leqslant r_{h-2} + r_{h-1}$

The principles of Timsort (3/3)

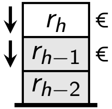
Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1^{st} and 2^{nd} runs
3. merge the top 2^{nd} and 3^{rd} runs

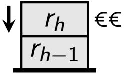
Choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $\underline{r_{h-2} \leqslant r_{h-1} + r_h}$ or $\underline{r_{h-3} \leqslant r_{h-2} + r_{h-1}}$: choose ②
else: choose ① (or ② if ① is unavailable)

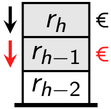
Making runs pay (with 1-step delay) **for going down**:



$r_{h-2} \leqslant r_h$



$r_{h-1} \leqslant r_h$



$r_{h-2} \leqslant r_{h-1} + r_h$
$r_{h-3} \leqslant r_{h-2} + r_{h-1}$

Contents

Stack size bugs in Java Timsort

Java choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$ or $r_{h-3} \leqslant r_{h-2} + r_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

Stack size bugs in Java Timsort

Java choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$: choose ②
else: choose ① (or ② if ① is unavailable)

Fibonacci constraints fail!

Stack size bugs in Java Timsort

Java choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$: choose ②
else: choose ① (or ② if ① is unavailable)

Fibonacci constraints fail!

- Stack height may be higher than forecast

Stack size bugs in Java Timsort

Java choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$: choose ②
else: choose ① (or ② if ① is unavailable)

Fibonacci constraints fail!

- Stack height may be higher than forecast
- Suggested fix: add the $r_{h-3} \leqslant r_{h-2} + r_{h-1}$ test[3]

Stack size bugs in Java Timsort

Java choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$: choose ②
else: choose ① (or ② if ① is unavailable)

Fibonacci constraints fail!

- Stack height may be higher than forecast
- Suggested fix: add the $r_{h-3} \leqslant r_{h-2} + r_{h-1}$ test[3]
- Custom Java fix: increase maximal stack size[3]

Stack size bugs in Java Timsort

Java choice algorithm

if $r_{h-2} \leqslant r_h$: choose ③
else if $r_{h-1} \leqslant r_h$, $r_{h-2} \leqslant r_{h-1} + r_h$: choose ②
else: choose ① (or ② if ① is unavailable)

Fibonacci constraints fail!

- Stack height may be higher than forecast
- Suggested fix: add the $r_{h-3} \leqslant r_{h-2} + r_{h-1}$ test[3]
- Custom Java fix: increase maximal stack size[3]

The increase was not sufficient!

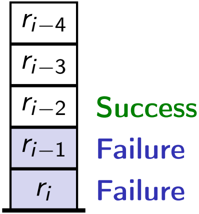
Bug raised by `igm.univ-mlv.fr/~pivoteau/Timsort/TimSort.java`

Java Timsort complexity analysis

Key steps:

- Study of the creation of **consecutive** Fibonacci **constraint failures**

$$r_{i-4} + r_{i-3} < r_{i-2}$$
$$r_{i-3} + r_{i-2} \geqslant r_{i-1}$$
$$r_{i-2} + r_{i-1} \geqslant r_i$$

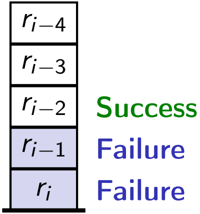


Java Timsort complexity analysis

Key steps:

- Study of the creation of **consecutive** Fibonacci **constraint failures**
- At most 6 consecutive contraint failures

$$r_{i-4} + r_{i-3} < r_{i-2}$$
$$r_{i-3} + r_{i-2} \geqslant r_{i-1}$$
$$r_{i-2} + r_{i-1} \geqslant r_i$$



Java Timsort complexity analysis

Key steps:

- Study of the creation of **consecutive** Fibonacci **constraint failures**
- At most 6 consecutive contraint failures
- $(r_i)_{i \geqslant 1}$ has still exponential decay

Java Timsort complexity analysis

Key steps:

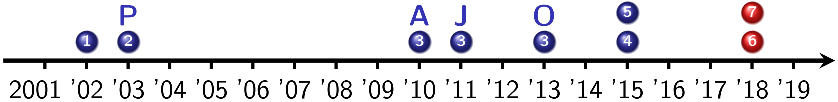
- Study of the creation of **consecutive** Fibonacci **constraint failures**
- At most 6 consecutive contraint failures
- $(r_i)_{i \geqslant 1}$ has still exponential decay
- Tight upper bound on stack size!

Java Timsort complexity analysis

Key steps:

- Study of the creation of **consecutive** Fibonacci **constraint failures**
- At most 6 consecutive contraint failures
- $(r_i)_{i \geqslant 1}$ has still exponential decay
- Tight upper bound on stack size!

7 Suggested fix[3] now implemented in Java (JDK 11)!



Conclusion

- Timsort is good **in practice**

Conclusion

- Timsort is good **in practice**
- ———————— **in theory**: $\mathcal{O}(n + n\mathcal{H})$ worst-case time complexity

Conclusion

- Timsort is good **in practice**
- ———————— **in theory**: $\mathcal{O}(n + n\mathcal{H})$ worst-case time complexity
- **Every algorithm** deserves a **proof** of **correctness** and **complexity**

Conclusion

- Timsort is good **in practice**
- ——————— **in theory**: $\mathcal{O}(n + n\mathcal{H})$ worst-case time complexity
- **Every algorithm** deserves a **proof** of **correctness** and **complexity**

Some references:

[1] Tim Peters' description of Timsort,
`svn.python.org/projects/python/trunk/Objects/listsort.txt` (2001)
[2] *On compressing permutations and adaptive sorting*, Barbay & Navarro (2013)
[3] *OpenJDK's java.utils.Collection.sort() is broken*, de Gouw et al. (2015)
[4] *Merge Strategies: from Merge Sort to Timsort*, Auger et al. (2015)
[5] *Strategies for stable merge sorting*, Buss & Knop (2018)
[6] *Nearly-optimal mergesorts*, Munro & Wild – to be presented **now** (2018)

Conclusion

- Timsort is good **in practice**
- ———————— **in theory**: $\mathcal{O}(n + n\mathcal{H})$ worst-case time complexity
- **Every algorithm** deserves a **proof** of **correctness** and **complexity**

Some references:

[1] Tim Peters' description of Timsort,
`svn.python.org/projects/python/trunk/Objects/listsort.txt` (2001)
[2] *On compressing permutations and adaptive sorting*, Barbay & Navarro (2013)
[3] *OpenJDK's java.utils.Collection.sort() is broken*, de Gouw et al. (2015)
[4] *Merge Strategies: from Merge Sort to Timsort*, Auger et al. (2015)
[5] *Strategies for stable merge sorting*, Buss & Knop (2018)
[6] *Nearly-optimal mergesorts*, Munro & Wild – to be presented **now** (2018)