

Complexity of the Adaptive ShiversSort Algorithm and of its sibling TimSort

Vincent Jugé

LIGM – Université Paris-Est Marne-la-Vallée, ESIEE, ENPC & CNRS

07/01/2020

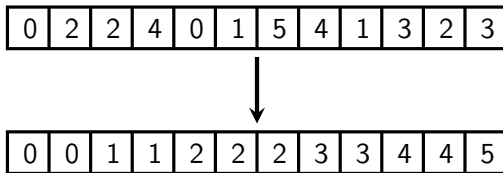
Contents

1 Efficient Merge Sorts

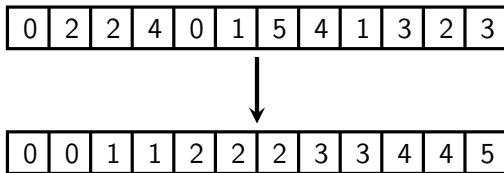
2 TimSort

3 Adaptive ShiversSort

Sorting data



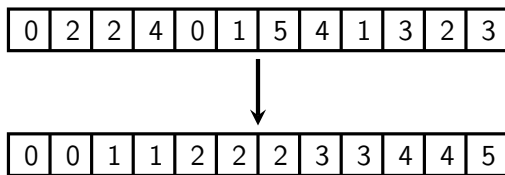
Sorting data



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 $\mathcal{O}(n \log(n))$

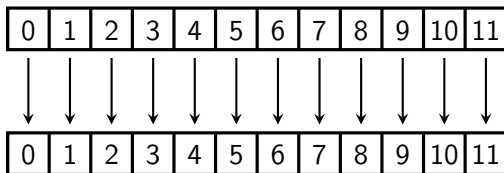
Can we do better? **No!**

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

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

Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

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

- 1 Chunk your data in **non-decreasing runs**
- 2 New parameters: **Number of runs** (ρ) and their **lengths** (r_1, \dots, r_ρ)

Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

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

- 1 Chunk your data in **non-decreasing runs**
- 2 New parameters: **Number of runs** (ρ) and their **lengths** (r_1, \dots, r_ρ)

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!

5 runs of lengths 4, 3, 1, 2 and 2

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

- 1 Chunk your data in **non-decreasing runs**
- 2 New parameters: **Number of runs** (ρ) and their **lengths** (r_1, \dots, r_ρ)

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 [7]

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

Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

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

- 1 Chunk your data in **non-decreasing runs**
- 2 New parameters: **Number of runs** (ρ) and their **lengths** (r_1, \dots, r_ρ)

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 [7]

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

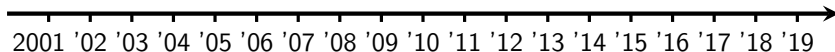
We cannot do better than $\Omega(n + n\mathcal{H})!$ ^[4]

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

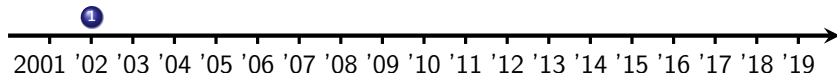
Contents

- 1 Efficient Merge Sorts
- 2 **TimSort**
- 3 Adaptive ShiversSort

A brief history of TimSort

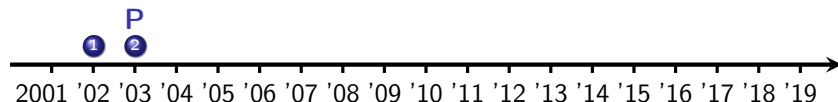


A brief history of TimSort



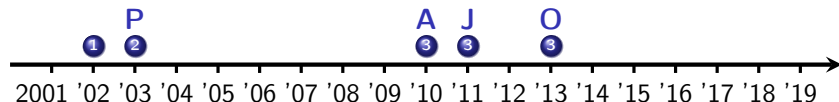
- ① Invented by **Tim Peters**^[3]

A brief history of TimSort



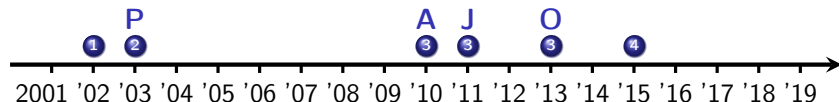
- 1 Invented by **Tim Peters**^[3]
- 2 Standard algorithm in **Python**

A brief history of TimSort



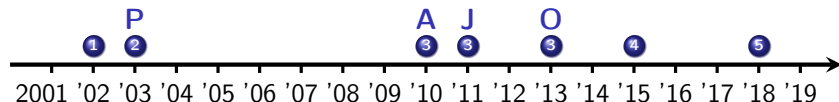
- ① Invented by **Tim Peters**^[3]
- ② Standard algorithm in **Python**
- ③ ————— for non-primitive arrays in **Android, Java, Octave**

A brief history of TimSort



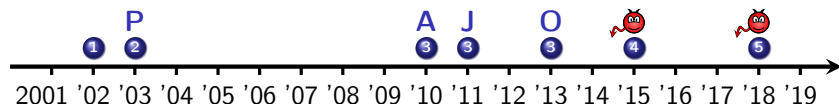
- ❶ Invented by **Tim Peters**^[3]
- ❷ Standard algorithm in **Python**
- ❸ ————— for non-primitive arrays in **Android, Java, Octave**
- ❹ 1st worst-case complexity analysis^[6] – TimSort works in time $\mathcal{O}(n \log n)$

A brief history of TimSort



- ① Invented by **Tim Peters**^[3]
- ② Standard algorithm in **Python**
- ③ ————— for non-primitive arrays in **Android, Java, Octave**
- ④ 1st worst-case complexity analysis^[6] – TimSort works in time $\mathcal{O}(n \log n)$
- ⑤ Refined worst-case analysis^[7] – TimSort works in time $\mathcal{O}(n + n\mathcal{H})$

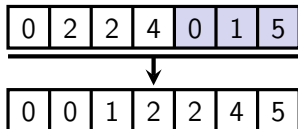
A brief history of TimSort



- ❶ Invented by **Tim Peters**^[3]
- ❷ Standard algorithm in **Python**
- ❸ ————— for non-primitive arrays in **Android, Java, Octave**
- ❹ 1st worst-case complexity analysis^[6] – TimSort works in time $\mathcal{O}(n \log n)$
- ❺ Refined worst-case analysis^[7] – TimSort works in time $\mathcal{O}(n + n\mathcal{H})$
- 👹 Bugs uncovered in Python & Java implementations^[5, 7]

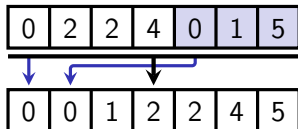
The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on **merging** adjacent runs



The principles of TimSort and of adaptive ShiversSort (1/2)

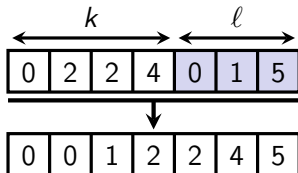
Algorithm based on **merging** adjacent runs ➡ **Stable** algorithm
(good for **composite** types)



The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on **merging** adjacent runs


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

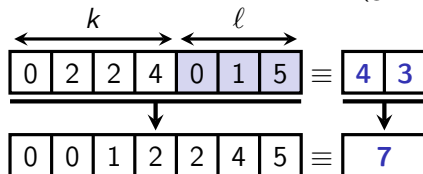


① **Run merging** algorithm: standard + many optimizations

- ▶ time $\mathcal{O}(k + \ell)$
 - ▶ memory $\mathcal{O}(\min(k, \ell))$
- } **Merge cost:** $k + \ell$

The principles of TimSort and of adaptive ShiversSort (1/2)

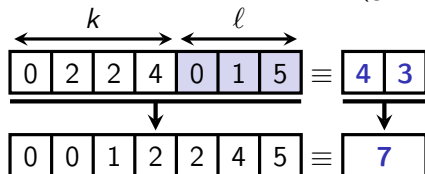
Algorithm based on **merging** adjacent runs  **Stable** algorithm
(good for **composite** types)



- 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

The principles of TimSort and of adaptive ShiversSort (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:**

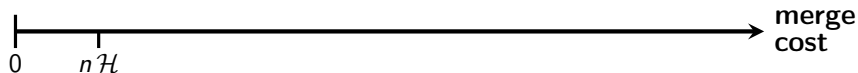
- 🐼 Evaluate the **total merge cost**
- 🐼 Forget array values and only work with **run lengths**

Some results about merge costs

Best-case merge costs:

- **Every algorithm** has a **best-case** merge cost of at least $n\mathcal{H}^{[4,10]}$

Worst-case merge costs:



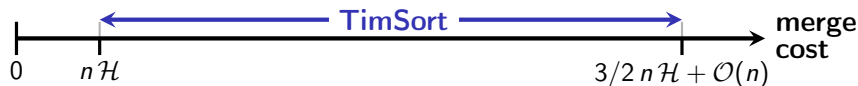
Some results about merge costs

Best-case merge costs:

- **Every algorithm** has a **best-case** merge cost of at least $n\mathcal{H}^{[4,10]}$

Worst-case merge costs:

- **TimSort** has a **worst-case** merge cost of $\frac{3}{2}n\mathcal{H} + \mathcal{O}(n)^{[7,9]}$



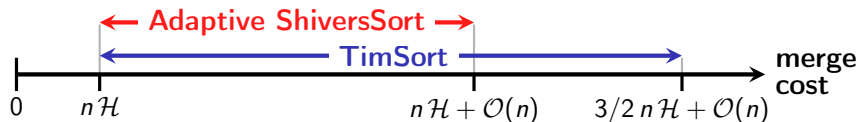
Some results about merge costs

Best-case merge costs:

- **Every algorithm** has a **best-case** merge cost of at least $n\mathcal{H}^{[4,10]}$

Worst-case merge costs:

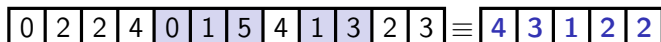
- **TimSort** has a **worst-case** merge cost of $\frac{3}{2}n\mathcal{H} + \mathcal{O}(n)^{[7,9]}$
- **Adaptive ShiversSort** has a **worst-case** merge cost of $n\mathcal{H} + \mathcal{O}(n)^{[10]}$



Contents

- 1 Efficient Merge Sorts
- 2 TimSort
- 3 Adaptive ShiversSort

The principles of adaptive ShiversSort and of TimSort (2/2)



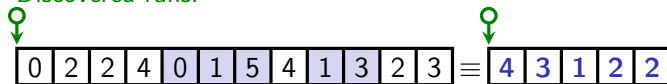
STACK

Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:



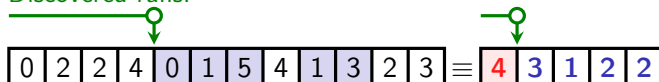
STACK

Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

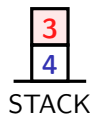
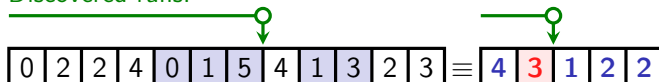


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

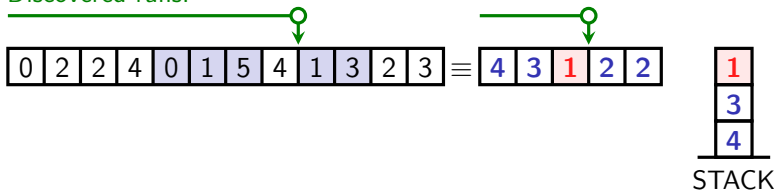


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

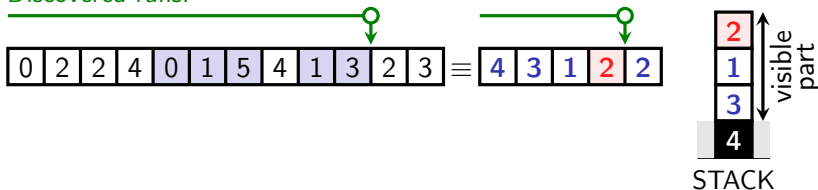


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

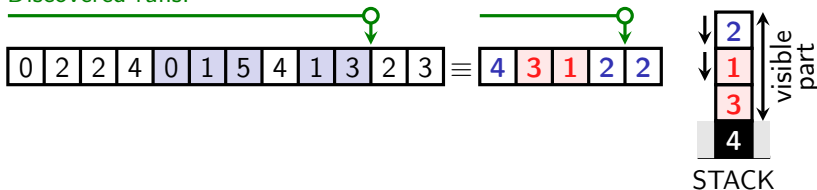


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

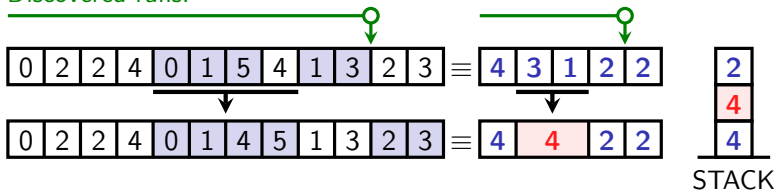


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

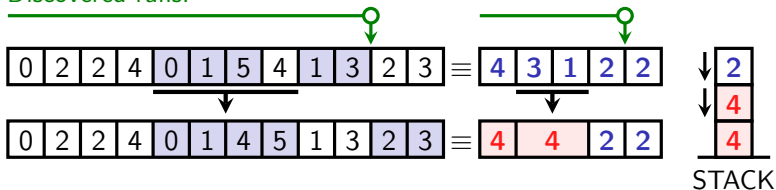


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

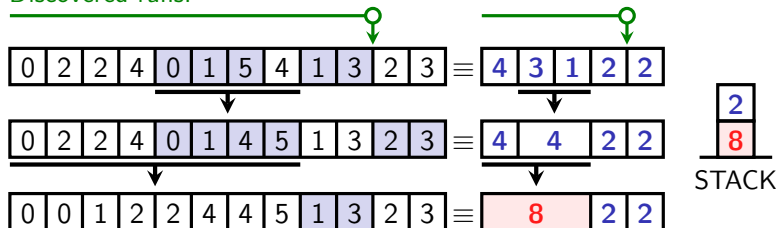


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

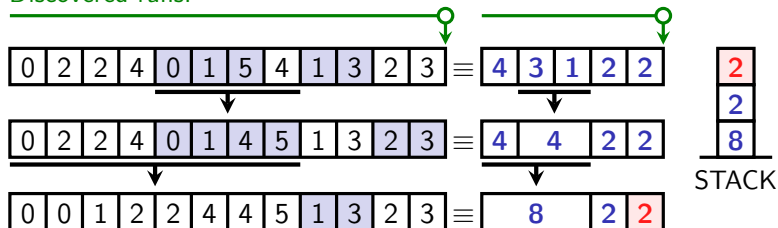


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

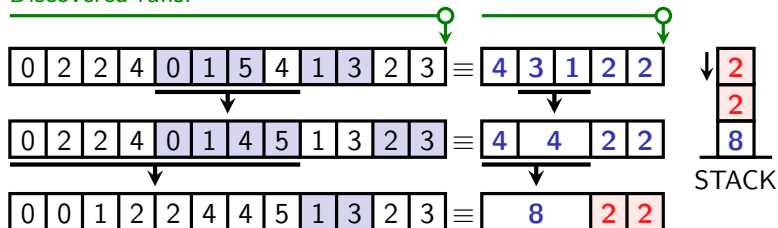


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

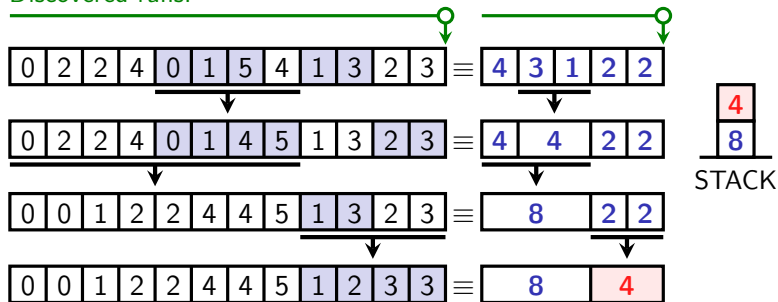


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

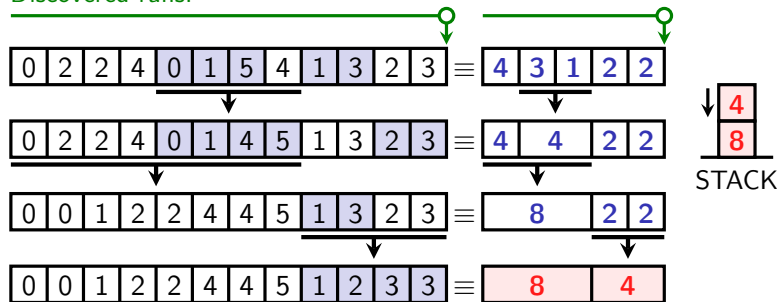


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

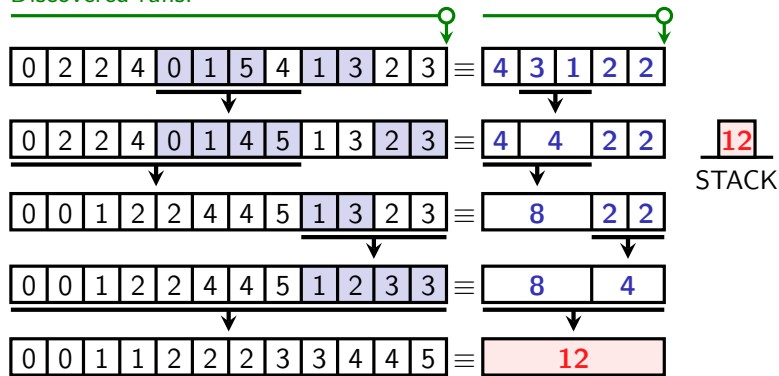


Run merge policy:

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

The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:



Run merge policy:

- Maintain a **stack** of runs
- Until the array is sorted, either:
 - 1 discover & push a new run 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:

- Each run r **pays** its share of the total merge cost

Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$

Cost analysis:

- Each run r **pays**
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$

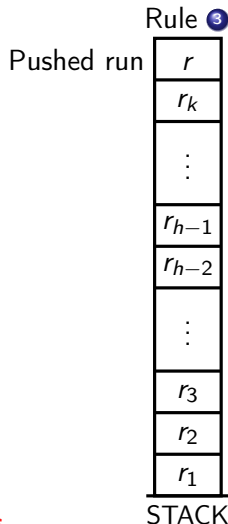
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase:**

Cost analysis:

- Each run r pays
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$



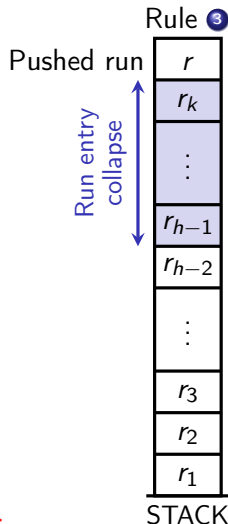
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase:**

Cost analysis:

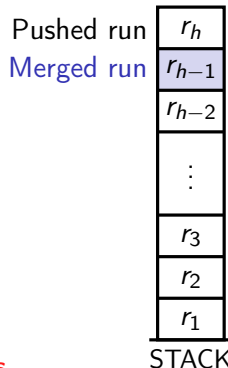
- Each run r pays
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$



Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase:**



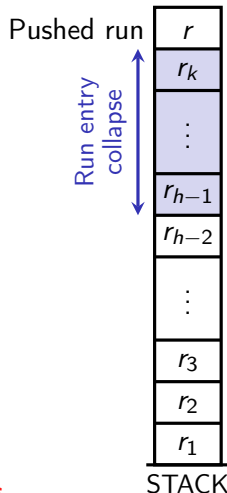
Cost analysis:

- Each run r pays
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$

Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase**: ensure that
 - ▶ r pays for every merge
 - ▶ $(r_i)_{i \geq 1}$ has **exponential** decay when r is pushed
 - ▶ runs **smaller** than r are merged



Cost analysis:

- Each run r pays
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$

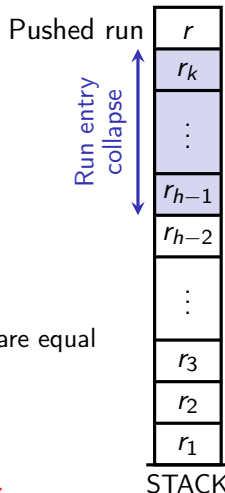
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase**: ensure that
 - ▶ r pays for every merge
 - ▶ $(r_i)_{i \geq 1}$ has **exponential** decay when r is pushed
 - ▶ runs **smaller** than r are merged
- **Growth phase**: ensure that
 - ▶ r_i and r_{i+1} are merged only if their **bit lengths** are equal

Cost analysis:

- Each run r pays
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$



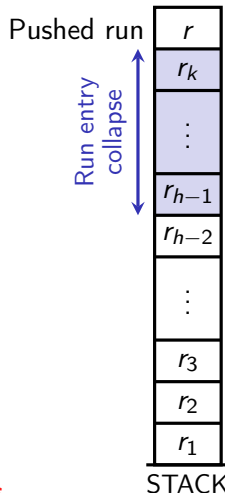
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase**: ensure that
 - ▶ r pays for every merge
 - ▶ $(\ell_i)_{i \geq 1}$ is **decreasing** when r is pushed
 - ▶ runs r_i with $\ell_i \leq \ell$ are merged
- **Growth phase**: ensure that
 - ▶ r_i and r_{i+1} are merged only if $\ell_i = \ell_{i+1}$

Cost analysis:

- Each run r pays
 - ☞ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☞ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$



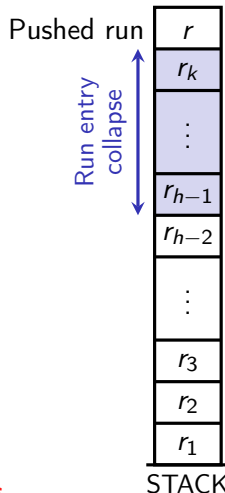
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run r **pays**
 - ▶ $\mathcal{O}(r)$ to **enter** the stack (**entry phase**)
 - ▶ r to increase its **bit length** (**growth phase**)
bit length of r : $\ell = \lfloor \log_2(r) \rfloor$
- **Entry phase**: ensure that
 - ▶ r pays for every merge
 - ▶ $(\ell_i)_{i \geq 1}$ is **decreasing** when r is pushed ☹
 - ▶ runs r_i with $\ell_i \leq \ell$ are merged
- **Growth phase**: ensure that
 - ▶ r_i and r_{i+1} are merged only if $\ell_i = \ell_{i+1}$ ☹☹

Cost analysis:

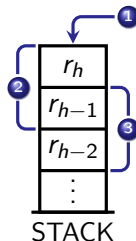
- Each run r pays
 - ☹ $\mathcal{O}(r)$ during its own **run entry phase**
 - ☹ at most $r \lceil \log_2(n/r) \rceil$ during the **growth phases**
- **Total merge cost** of $n\mathcal{H} + \mathcal{O}(n)$



The details of Adaptive ShiversSort

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

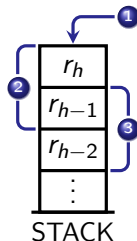
if $\ell_h \geq \ell_{h-2}$ **or** $\ell_{h-1} \geq \ell_{h-2}$: choose ③
else if $\ell_h \geq \ell_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

where $\ell_i = \lfloor \log_2(r_i) \rfloor$

The details of Adaptive ShiversSort

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 $\ell_h \geq \ell_{h-2}$ or $\ell_{h-1} \geq \ell_{h-2}$: choose ③

else if $\ell_h \geq \ell_{h-1}$: choose ②

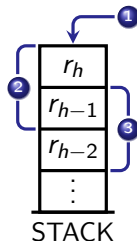
else: choose ① (or ② if ① is unavailable)

where $\ell_i = \lfloor \log_2(r_i) \rfloor$

The details of Adaptive ShiversSort

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 $\ell_h \geq \ell_{h-2}$ **or** $\ell_{h-1} \geq \ell_{h-2}$: choose ③

else if $\ell_h \geq \ell_{h-1}$: choose ②

else: choose ① (or ② if ① is unavailable)

where $\ell_i = \lfloor \log_2(r_i) \rfloor$

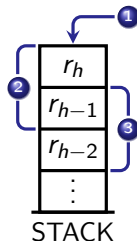
Bit-length constraints:

- $\ell_1 > \ell_2 > \dots > \ell_{h-2} \geq \ell_{h-1}$ (induction)
- $\ell_1 > \ell_2 > \dots > \ell_h$ **on run push** ☹
- $\ell_{h-1} \geq \ell_h$ and $\ell_{h-2} > \ell_h$ **during growth** (induction) ☹☹

The details of Adaptive ShiversSort

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 $\ell_h \geq \ell_{h-2}$ **or** $\ell_{h-1} \geq \ell_{h-2}$: choose ③
else if $\ell_h \geq \ell_{h-1}$: choose ②
else: choose ① (or ② if ① is unavailable)

where $\ell_i = \lfloor \log_2(r_i) \rfloor$

Bit-length constraints:

- $\ell_1 > \ell_2 > \dots > \ell_{h-2} \geq \ell_{h-1}$ (induction)
- $\ell_1 > \ell_2 > \dots > \ell_h$ **on run push** ∅
- $\ell_{h-1} \geq \ell_h$ and $\ell_{h-2} > \ell_h$ **during growth** (induction) ∅∅

END OF PROOF!

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\mathcal{H})$ merge cost
- **Adaptive ShiversSort** is **better** than and **very similar** to TimSort

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\mathcal{H})$ merge cost
- **Adaptive ShiversSort** is **better** than and **very similar** to TimSort

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] Tim Peters' description of TimSort,
svn.python.org/projects/python/trunk/Objects/listsort.txt (2001)
- [4] *On compressing permutations and adaptive sorting*, Barbay & Navarro (2013)
- [5] *OpenJDK's java.utils.Collection.sort() is broken*, de Gouw et al. (2015)
- [6] *Merge strategies: from merge sort to TimSort*, Auger et al. (2015)
- [7] *On the worst-case complexity of TimSort*, Auger et al. (2018)
- [8] *Nearly-optimal mergesorts*, Munro & Wild (2018)
- [9] *Strategies for stable merge sorting*, Buss & Knop (2019)
- [10] *Adaptive ShiversSort: an alternative sorting algorithm*, Jugué (2020)

Thank
you