

Theoretical perspectives on algorithmic choices made in programming languages

Cyril Nicaud

LIGM – Université Gustave-Eiffel & CNRS

The Prague Stringology Conference, August 2023

I. Introduction

Introduction

All programming languages (or their standard libraries) include classical solutions and data structures: **sorting algorithms**, **lists**, **hashmaps**, ...

The algorithms behind these tools are studied since the beginning of computer science:

- Very well-known (optimal) solutions
- Detailed theoretical analysis
- Decades of practical uses

Question: if you were to create a new programming language,
which sorting algorithm would you choose?

Which sorting algorithm would you choose?

Remarks:

- The chosen sorting algorithm has to be **generic**
- Other choices: **stable? in place?**
- It may just be a matter of benchmarking

From classical textbooks, two main candidates:

- **QuickSort**
 - ▶ **in place**, usually **unstable**
 - ▶ $\mathcal{O}(n^2)$ worst case, $\mathcal{O}(n \log n)$ in average
- **MergeSort**
 - ▶ **not in place**, **stable**
 - ▶ $\mathcal{O}(n \log n)$ worst case

Which sorting algorithms are chosen (examples)?

QuickSort-like:

- Javascript V8: QuickSort, using InsertionSort if size ≤ 10
- PHP: QuickSort, using InsertionSort if size ≤ 16 , pivot at position $\frac{n}{2}$
- C++: IntroSort a mix of QuickSort, HeapSort and InsertionSort
- Java *primitive types*: dual-pivot QuickSort
- Rust *unstable*: PDQSort, pattern-defeating quicksort

TimSort-like (MergeSort-like?):

- Python: TimSort (until 2021) in cpython
- Java *objects*: TimSort
- Rust *stable*: variant of TimSort

⇒ cpython nowadays uses an implementation of PowerSort (Munro & Wild)

Is there room for theoretical analysis?

- Some **new** algorithms were designed by engineers, ready to be theoretically studied (**TimSort**, **PDQSort**, ...)
- Some choices are motivated by **modern computer architecture**, we can enhance our computational models with **cache**, **branch prediction**, **vectorization**, ... (dual-pivot **QuickSort** of **Java**)
- Data may have some typical structures or patterns in practice: how can we quantify that for a theoretical analysis?

*Let us look at **TimSort** in details to illustrate these ideas*

II. TimSort

with N. Auger, V. Jugé & C. Pivoteau

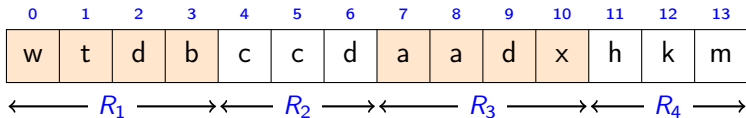
This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than $\lg(N!)$ comparisons needed, and as few as $N-1$), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

timsort.txt, Tim Peters

Monotonic runs

To take some **presortedness** into account, **TimSort** first splits the array into **sequences of monotonic runs**:

- They can be **non-decreasing** or **decreasing**
- They are **maximal**



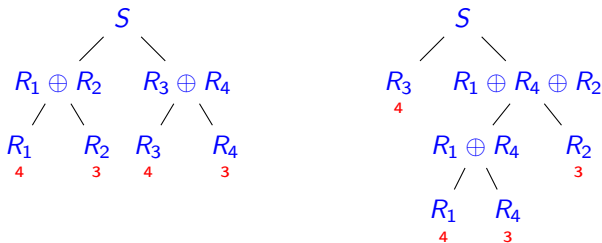
- They are computed greedily from left to right in time $\mathcal{O}(n)$
- Decreasing runs are **reversed** on the fly

⇒ we obtain a sequence of non-decreasing runs to merge

Merge cost and merge tree

- R_i and R_j can be merged as in MergeSort, using $|R_i| + |R_j| - 1$ comparisons in the worst case. We use $c(R_i, R_j) = |R_i| + |R_j|$
- We denote $R_i \oplus R_j$ the result of the merge of R_i and R_j

The **merge tree** represents the merges performed by the algorithm:

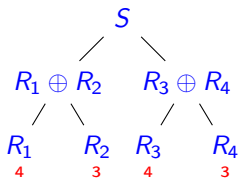


$$\text{Total cost} = \sum_{R_i} |R_i| \times \text{height}(R_i)$$

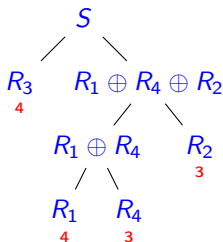
Merge cost and merge tree

- R_i and R_j can be merged as in MergeSort, using $|R_i| + |R_j| - 1$ comparisons in the worst case. We use $c(R_i, R_j) = |R_i| + |R_j|$
- We denote $R_i \oplus R_j$ the result of the merge of R_i and R_j

The **merge tree** represents the merges performed by the algorithm:



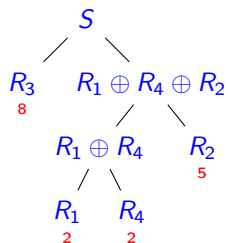
Cost = 28



Cost = 31

$$\text{Total cost} = \sum_{R_i} |R_i| \times \text{height}(R_i)$$

An additional condition



Cost = 30

$$\text{Total cost} = \sum_{R_i} |R_i| \times \text{height}(R_i)$$

- **Huffman** construction optimizes the cost
- It can be computed efficiently
- Discovered several times, Takaota'09, Barbay & Navarro'13, ...
- *Not used in programming languages*

Additional condition: only merge consecutive runs!

- **stability**
- it is easier to manage memory
- runs can be merged when discovered (if needed)
- **cache-friendly**

Merging consecutive runs

An **optimal solution** with this new constraint can be computed using **dynamic programming**, but the cost is prohibitive.

⇒ Algorithms of that kind are approximations of the optimal solution

Several ideas:

- Knuth's **NaturalMergeSort**: as **MergeSort**, starting with the run decomposition
- Greedily merge the two consecutive runs of smallest total length
- **TimSort** by Peters (2001)
- **PowerSort** by Munro and Wild (2018)
- Several others: **AdaptativeShiverSort** Jugé (2020), ...

Merging consecutive runs

An **optimal solution** with this new constraint can be computed using **dynamic programming**, but the cost is prohibitive.

⇒ Algorithms of that kind are approximations of the optimal solution

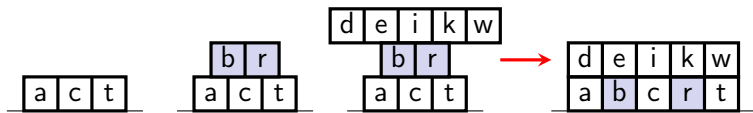
Several ideas:

- Knuth's **NaturalMergeSort**: as **MergeSort**, starting with the run decomposition → **can be really suboptimal**
- Greedily merge the two consecutive runs of smallest total length → **2-approximation**, **cannot be performed as runs are discovered**
- **TimSort** by Peters (2001)
- **PowerSort** by Munro and Wild (2018)
- Several others: **AdaptativeShiverSort** Jugé (2020), ...

TimSort algorithm

a	c	t	r	b	w	k	i	e	d	u	n
---	---	---	---	---	---	---	---	---	---	---	---

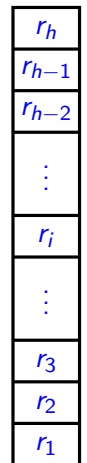
- The input is split into **runs**, which are **monotonic subsequences**
- Every discovered run is added to a **stack**, then some **consecutive** runs can be **merged**



- Merges occur by only looking at the **run lengths**, not the values within
- When there is no more run, the runs in the stack are merged top-down

Remark: **TimSort** also contains a lot of heuristics that we don't consider here (especially in the merge procedure)

Legacy TimSort's Merging Rules



STACK

Notations:

- the run R_i has length r_i
- the stack has height h
- the topmost run is R_h

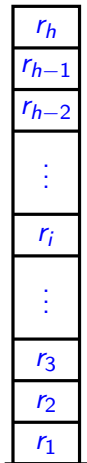
Merges after adding a new run:

- **While true**
 - ▶ **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
 - ▶ **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
 - ▶ **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
 - ▶ **else** break

Remarks:

- we only consider the three topmost runs
- we only merge R_h and R_{h-1} , or R_{h-1} and R_{h-2}

TimSort's Merging Rules



STACK

Merges after adding a new run:

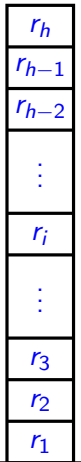
- **While true**

- ▶ **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
- ▶ **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
- ▶ **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
- ▶ **else** break

`timsort.txt`:

"Note that, by induction, it implies the lengths of pending runs form a decreasing sequence. It implies that, reading the lengths right to left, the pending-run lengths grow at least as fast as the Fibonacci numbers. Therefore the stack can never grow larger than about $\log_\phi(N)$ entries"

TimSort's Merging Rules



STACK

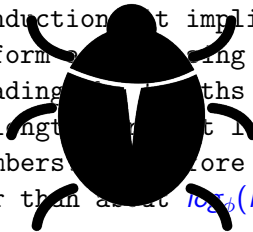
Merges after adding a new run:

- While true

- ▶ if $r_h > r_{h-2}$ then merge R_{h-1} and R_{h-2}
- ▶ else if $r_h \geq r_{h-1}$ then merge R_h and R_{h-1}
- ▶ else if $r_h + r_{h-1} \geq r_{h-2}$ then merge R_h and R_{h-1}
- ▶ else break

`timsort.txt`:


"Note that, by induction, it implies the lengths of pending runs form a Fibonacci-like sequence. It implies that, reading the pending-run lengths right to left, the pending-run length grows at least as fast as the Fibonacci numbers. Therefore the stack can never grow larger than about $\log_b(N)$ entries"



An error in timsort.txt

- **While true**

- ▶ **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
- ▶ **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
- ▶ **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
- ▶ **else** break


 The invariant $r_{i+2} + r_{i+1} < r_i$ **does not hold!**

Discovered by [de Gouw et al \(2015\)](#) while trying to prove (formally) the correctness of Java's **TimSort**, using KeY (formal verification tool)

An error in timsort.txt

- **While true**

- ▶ **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}
- ▶ **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}
- ▶ **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}
- ▶ **else** break

 The invariant $r_{i+2} + r_{i+1} < r_i$ **does not hold!**

Discovered by [de Gouw et al \(2015\)](#) while trying to prove (formally) the correctness of Java's **TimSort**, using KeY (formal verification tool)

Is it a **real** problem?

- In **Python**: not really, the algorithm is still efficient and correct
- In **Java**: they use the invariant to fix the maximum size of the stack, implemented with a static array \Rightarrow [de Gouw et al \(2015\)](#) built an array that produces an error for Java's `sort()`!

Two versions of TimSort

de Gouw et al (2015) proposed two solutions to fix the problem:

1. Adding a new rule (implemented in Python)

• While true

- ▶ if $r_h > r_{h-2}$ then merge R_{h-1} and R_{h-2}
- ▶ else if $r_h \geq r_{h-1}$ then merge R_h and R_{h-1}
- ▶ else if $r_h + r_{h-1} \geq r_{h-2}$ then merge R_h and R_{h-1}
- ▶ else if $r_{h-1} + r_{h-2} \geq r_{h-3}$ then merge R_h and R_{h-1}
- ▶ else break

The invariant now holds, the algorithm is certified in KeY.

2. Computing correct maximal heights for the stack (implemented in Java)

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive positions in the stack.

Running time

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than $\lg(N!)$ comparisons needed, and as few as $N-1$), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

timsort.txt – Tim Peters

Theorem (Auger, Nicaud, Pivoteau 2015)

TimSort has a **worst-case running time** of $\mathcal{O}(n \log n)$.

- Our first proof (preprint in 2015) was not very difficult, but hard to read (and to teach!)
- A better proof in ESA'18 proceedings

Running time analysis of TimSort: $O(n \log n)$

We focus on the main loop: other parts are done in $O(n)$ comparisons.

- **While there are remaining runs**

- (#1) Add a new run to the stack

- Repeat until stabilized**

- (#2) **if** $r_h > r_{h-2}$ **then** merge R_{h-1} and R_{h-2}

- (#3) **else if** $r_h \geq r_{h-1}$ **then** merge R_h and R_{h-1}

- (#4) **else if** $r_h + r_{h-1} \geq r_{h-2}$ **then** merge R_h and R_{h-1}

- (#5) **else if** $r_{h-1} + r_{h-2} \geq r_{h-3}$ **then** merge R_h and R_{h-1}

Amortized analysis:

- \diamond -tokens and \heartsuit -tokens are given to the elements of the input
- tokens are used to pay for comparisons
- the total number of tokens granted is our upper bound

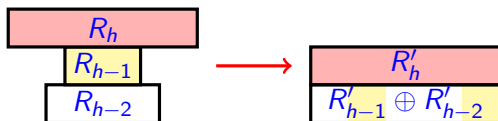
Tokens' rules: an element gets two \diamond and one \heartsuit

- when its run enters the stack
- when its height in the stack decreases

Running time analysis: Case #2

(#2) if $r_h > r_{h-2}$ then merge R_{h-1} and R_{h-2}

Every element of R_h and R_{h-1} pays one \diamond : the merge cost is $r_{h-1} + r_{h-2} \leq r_{h-1} + r_h$, hence it is fully paid

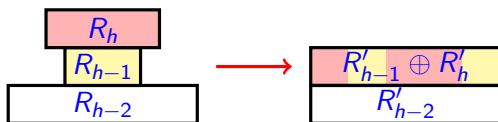


The height of every element that paid one \diamond decreases by one: they all gain two \diamond and one \heartsuit , regaining what they paid

Running time analysis: Case #3

(#3) else if $r_h \geq r_{h-1}$ then merge R_h and R_{h-1}

Every element of R_h pays two \diamond : the merge cost is $r_h + r_{h-1} \leq 2r_h$, hence it is fully paid.

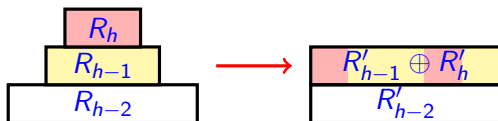


The height of every element that paid two \diamond decreases by one: they all gain two \diamond and one \heartsuit , regaining what they paid

Running time analysis: Case #4 (Case #5 is similar)

(#4) else if $r_h + r_{h-1} \geq r_{h-2}$ then merge R_h and R_{h-1}

Every element of R_h pays one \diamond , every element of R_{h-1} pays one \heartsuit : the merge cost is $r_h + r_{h-1}$, hence it is fully paid.



The height of the elements of R_h decreases by one: ok for \diamond

- Elements that paid one \heartsuit are now in the topmost run
- Elements in the topmost run never pay with \heartsuit
- After the merge, $r_h \geq r_{h-1}$ so #2 or #3 is going to occur immediately
- The height of the new topmost run is going to decrease during this new merge, its elements will regain their \heartsuit (and also two \diamond)

Running time analysis: $\mathcal{O}(n \log n)$

Summary:

- Computing the run decomposition takes $\mathcal{O}(n)$
- For the main loop:
 - ▶ each element gets 2♦ and 1♥ when entering the stack
 - ▶ each merge is paid with ♦ and ♥
 - ▶ when an element pays with ♦, it get it (them) back immediately after
 - ▶ when an element pays with ♥, another merge occurs just after, during which it get it back
- The final merges are done in $\mathcal{O}(n)$ by direct computation

Lemma

At any moment during **TimSort**, the stack has height in $\mathcal{O}(\log n)$.

Proof: the invariant holds.

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of **TimSort** is in $\mathcal{O}(n \log n)$.

Running Time

⇒ TimSort is optimal with a $\mathcal{O}(n \log n)$ running time

What makes it favored to other optimal algorithms?

I believe that lists very often do have exploitable partial order in real life, and this is the strongest argument in favor of timsort

timsort.txt, Tim Peters

- We want to formalize this from a theoretical point of view
- Idea: parameterized complexity to take presortedness into account

Let ρ denote the number of runs, we have:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n + n \log \rho)$.

Run lengths entropy

If the runs have size r_1, \dots, r_ρ , then the **run lengths entropy** of the array is

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

- For run lengths $\frac{n}{11}, \dots, \frac{n}{11}$: $\mathcal{H} = \log_2 11 \approx 3.46$
- For run lengths $\frac{90n}{100}, \frac{n}{100}, \dots, \frac{n}{100}$: $\mathcal{H} \approx 0.80$
- For run lengths $\sqrt{n}, \dots, \sqrt{n}$: $\mathcal{H} = \frac{1}{2} \log_2 n$
- For run lengths $n - 2\sqrt{n}, 2, 2, \dots, 2$: $\mathcal{H} = \mathcal{O}\left(\frac{\log n}{\sqrt{n}}\right)$

Remark:

$$\mathcal{H} \leq \log_2 \rho \leq \log_2 n$$

Timsort running time parameterized by entropy

Theorem (Auger, Jugé, Nicaud, Pivoteau. Talk ESA 2018)

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

Theorem (Barbay, Navarro 2013)

Sorting by comparisons algorithms use more than $n\mathcal{H} - \mathcal{O}(n)$ comparisons.

Theorem (Auger, Jugé, Nicaud, Pivoteau. Buss, Knop 2019)

TimSort uses $1.5n\mathcal{H} + \mathcal{O}(n)$ comparisons in the worst case.

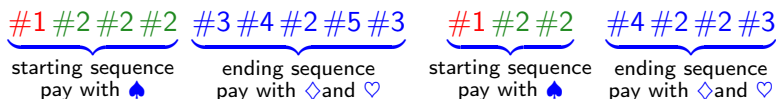
- Lower bound by Buss and Knop
- We proved the upper bound

Running time analysis: $\mathcal{O}(n + n\mathcal{H})$

Recall: #1 is the insertion of a new run in the stack

Recall: $\mathcal{H} = -\sum \frac{r_i}{n} \log \frac{r_i}{n}$

We use the following decomposition of the sequence of events:



Two lemmas (both consequences of the invariant):

- The total cost in ♠-tokens is **linear**
- The **height** of the stack at the **beginning of the ending sequence** after inserting a run of length r is $\mathcal{O}(\log \frac{n}{r})$.

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of TimSort is in $\mathcal{O}(n + n\mathcal{H})$.

And for Legacy TimSort?

For the legacy version of TimSort, we just have:

Theorem (Auger, Jugé, Nicaud, Pivoteau 2018)

The running time of LegacyTimSort is in $\mathcal{O}(n + n \log \rho)$.

but wait a minute ...

Another bug in Java's TimSort

Lemma

Throughout execution of **TimSort**, the invariant cannot be violated at two consecutive positions in the stack.

Another bug in Java's TimSort

Lemma

Throughout execution of **TimSort**, the invariant cannot be violated at two consecutive positions in the stack.



The lemma is incorrect!

⇒ We built an array **that produces an error to Java's (patched) TimSort!**

Another bug in Java's TimSort

Lemma

Throughout execution of **TimSort**, the invariant cannot be violated at two consecutive positions in the stack.



The lemma is incorrect!

⇒ We built an array that produces an error to Java's (patched) TimSort!

The screenshot shows a JIRA issue page for JDK-8203864. The issue title is "Execution error in Java's Timsort". The status is "RESOLVED". The issue type is "Bug", priority is "P3", and it affects version "11". The component is "core-libs" and the subcomponent is "java.util.collections". The issue was introduced in version "6" and resolved in build "b20".

Details

- Type: Bug
- Priority: P3
- Affects Version/s: None
- Component/s: core-libs
- Labels: None
- Subcomponent: java.util.collections
- Introduced In: 6
- Version: 11
- Resolved in Build: b20

Status: RESOLVED

- Resolution: Fixed
- Fix Version/s: 11

People

- Assignee: Martin Buchholz
- Reporter: Rémi Forax
- Votes: 0
- Watchers: 0

Backports

Issue	Fix Version	Assignee	Priority	Status	Resolution	Resolved In Build
JDK-8206770 12		Doug Lea	P3	Resolved	Fixed	team
JDK-8206547 11 0.1		Doug Lea	P3	Resolved	Fixed	b01

Description

Carine Pivoteau wrote:
While working on a proper complexity analysis of the algorithm, we realised that there was an error in the last paper reporting such a bug (<http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf>). This implies that the correction implemented in the java source code (changing Timsort stack size) is wrong and that it is still possible

Another bug in Java's TimSort

Lemma

Throughout execution of **TimSort**, the **invariant** cannot be violated at two consecutive positions in the stack. 

The lemma is incorrect!

⇒ We built an array that produces an error to Java's (patched) TimSort!



The screenshot shows a JIRA issue page for 'Execution error in Java's Timsort' (JDK-8206770). The issue is categorized as a 'Bug' and is marked as 'Resolved' in build b20. The 'Details' section shows the issue type as 'Bug', priority as 'P', and it affects version 11.0.1. The 'Backlogs' table lists the issue as resolved in version 11.0.1 by Doug Lea. The 'Description' mentions a paper by Carine Pivoteau. A large, tilted overlay contains a 'Hacker News' article snippet titled 'On the Worst-Case Complexity of TimSort (2016)'. The article discusses a bug in Java 10.0.2 related to a 'java.lang.ArrayIndexOutOfBoundsException: 519' and mentions a 'really huge array (>4GB)'. It also references a paper by Carine Pivoteau and a link to a GitHub repository.

Down for Maintenance on Friday January 25 from 3-5 PM PST

JDK / JDK-8203864

Execution error in Java's Timsort

login

Details

Type: **B** Bug
Priority: **P** P
Affects Version/s: None
Component/s: core-ii
Labels: None
Subcomponent: java.util
Introduced In: 6
Version:
Resolved in Build: b20

Backlogs

Issue	Fix Version	Assignee
JDK-8206770 12		Doug Lea
JDK-8206547 11 0.1		Doug Lea

Description

Carine Pivoteau wrote:
While working on a proper complexity ana. was an error in the last paper reporting suc. content/uploads/2015/02/errring.pdf. This is the java source code (changing TimSort.stc.

Hacker News [new](#) | [comments](#) | [ask](#) | [show](#) | [jobs](#) | [submit](#)

▲ On the Worst-Case Complexity of TimSort (2016) [dot] 204 points by pablos 4 months ago | [hide](#) | [edit](#) | [web](#) | [favorite](#) | 74 comments

▲ (Self 4 months ago [-])
The linked java test file, <http://openjdk.org/jdk-9/branches/jdk-9/103864/TestJava> - still crashes the latest Java 10.0.2 with an "Exception in thread "main": java.lang.ArrayIndexOutOfBoundsException: 519". Amazing! I wonder if this makes some web services vulnerable - if the user can submit a just-so array of ints to be sorted? But it does seem like it would require uploading a really huge array (>4GB)

▲ (Self 4 months ago [-])
This link like <https://www.openjdk.java.net/jvms/8203864/>, which has the following additional information:
"While working on a proper complexity analysis of the algorithm, we realised that there was an error in the last paper reporting such a bug (<http://openjdk.org/content/uploads/2015/02/errring.pdf>). This implies that the correction implemented in the Java source code (changing TimSort stack size) is wrong and that it is still possible to make it break. This is explained in full details in our analysis: <https://arxiv.org/pdf/1805.08612.pdf>".

▲ andrea14 4 months ago [-]
That is not additional information. That bug was created by the authors of this post, and the link is a reference to the exact paper linked to in this post.

▲ (Self 4 months ago [-])
... doesn't really seem like an actionable attack vector, unless that is the application invocation.

Another bug in Java's TimSort

Lemma

Throughout execution of TimSort, the invariant cannot be violated at two consecutive positions in the stack.



The lemma is incorrect!

⇒ We built an array that produces an error to Java's (patched) TimSort!

The screenshot shows a bug report for JDK-8203864, titled "Execution error in Java's Timsort". The report details the issue, including its priority, component (core-ii), and version (6). A table of backports shows the issue was fixed in JDK-8206770 and JDK-8206547. The description mentions a linked Java test file and a reference to a paper on the worst-case complexity of TimSort.

Overlaid on the bug report is a code snippet for the `mergeCollapse()` method. The code includes a comment about the method being called frequently and a list of contributors: Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hahnle. The code itself is as follows:

```
private void mergeCollapse() {
    while (stackSize > 1) {
        int n = stackSize - 2;
        if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
            if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1] ||
                n > 1 && runLen[n-2] <= runLen[n] + runLen[n-1]) {
                if (runLen[n-1] < runLen[n+1]) {
                    n--;
                    mergeAt(n);
                } else if (runLen[n] <= runLen[n+1]) {
                    mergeAt(n);
                } else {
                    mergeAt(n);
                } else if (n < 0 || runLen[n] > runLen[n+1]) {
                    break; // Invariant
                }
            }
        }
    }
}
```

Conclusion on TimSort

- TimSort is an **efficient** algorithm, in theory and in practice
- It is not **entropy-optimal**, but not far from it
- There are many optimisations, for instance on merges ([Ghasemi, Jugé, & Khalighinejad \[ICALP'22\]](#))
- There were **two consecutive bugs** in Java's version, due to improper analysis of the algorithm

- Every new (used) algorithm deserves a fine grain analysis
- **run lengths entropy** plays a major role

Epilogue for sorting algorithms

- There is a gap in the leading asymptotic term between **TimSort** in $\sim 1.5 n\mathcal{H}$ and the lower bound in $\sim n\mathcal{H}$
- Several algorithms are tight for this measure of complexity $n\mathcal{H} + \mathcal{O}(n)$:
 - ▶ Takaota [MFCS'09] Huffman
 - ▶ Barbay & Navarro [TCS'13] Huffman with contiguous runs
 - ▶ Jugé [SODA'20] **AdaptiveShiverSort**
- in 2021, cpython programmers decided to change their sorting algorithm to **PowerSort** Munro & Wild [ESA'18], also in $n\mathcal{H} + \mathcal{O}(n)$

III. Lua's Table

with C. Martínez & P. Rotondo

Maps

A **map** (**associative array**, **dictionary**, ...) is a data structure to encode a **partial mapping** from a set of **keys** to a set of **values**

It supports the operations of **initialization**, **insertion** of $k \rightarrow v$, **search** for the value associated to a key, and **delete** a key

Maps are usually encoded using **Hashtables** (more rarely, **balanced trees**)

Question: if you were to create a new programming language,
what kind of hashtables would you implement?

Hashtables

- We assume that we have **efficient hash functions**
- The **load factor** $\alpha := \frac{N}{M}$, where M is the size of the hashtable, and N is the number of keys it contains
- The table has a given starting size (**capacity**), which is **doubled** whenever we reach a given **load factor** α

Dealing with collisions:

- **separate chaining**: use a linked list in each bucket
- **open addressing**: put the key somewhere else if its bucket is already taken: **linear probing**, **quadratic probing**, **double hashing**, ...

How are they implemented (examples)?

Separate chaining:

- PHP, $\alpha = 1$
- C++ **unordered map** in std, $\alpha = 1$
- Java, $\alpha = 3/4$, *balanced trees* if too many keys

Open-addressing:

- Javascript V8: Deterministic Hash Tables, quadratic probing
- Python **dictionary**: random probing, $\alpha = 2/3$
- Rust: clusters of 16 entries (SIMD)

→ *let us look at LUA's tables*

Lua (programming language)

From Wikipedia, the free encyclopedia

The article is about the Lua programming language itself. For its use in Wikipedia, see [Wikipedia:Lua](#).

Lua (/ˈluːə/ *LOO-ə*; from Portuguese: *lua* [ˈlu.(w)ɐ] meaning *moon*) is a **lightweight, high-level, multi-paradigm programming language** designed primarily for **embedded use** in applications.^[3] Lua is **cross-platform**, since the **interpreter** of **compiled bytecode** is written in **ANSI C**,^[4] and Lua has a relatively simple **C API** to embed it into applications.^[5]

Lua was originally designed in 1993 as a language for extending **software applications** to meet the increasing demand for customization at the time. It provided the basic facilities of most **procedural programming** languages, but more complicated or **domain-specific** features were not included; rather, it included mechanisms for extending the language, allowing programmers to implement such features. As Lua was intended to be a general embeddable extension language, the designers of Lua focused on improving its **speed**, **portability**, extensibility, and ease-of-use in development.

Lua: website

❖ What is Lua?

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua is fast

Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

Lua's Tables

- **Tables** are the main (only) data structuring mechanism in Lua
- Until Lua 4.0 → Hashmaps
- Lua 5.0 → hybrid data-structure with an array part and a hash part

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>H</i>	120 20		PR 8	WF 8	EF 21	CM 11					CP 4	VJ 8	ZS 7		FK 7	

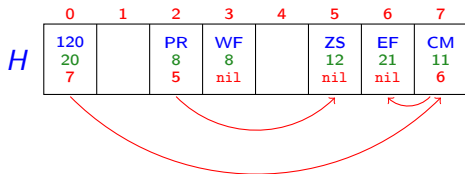
	1	2	3	4	5	6	7	8
<i>A</i>	6		7	16	21	22		4

$$\begin{aligned} f(CM) &= 11 & f(CP) &= 4 & f(GC) &= \text{nil} \\ f(5) &= 21 & f(120) &= 20 & f(7) &= \text{nil} \end{aligned}$$

In the array part *A*, the **keys** are the **indices**.

Lua's hashmaps

Internal chaining: each spot contains the index of the next key in the list

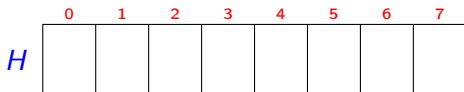


- Internal chaining is also classical (in textbooks *TAOCP vol. 3*)
- Several ways to handle collisions leading to coalescent chaining, or separate chaining

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$



$$\begin{array}{lll} h(ZS) = 2 & h(FK) = 4 & h(PR) = 4 \\ h(VJ) = 6 & h(WF) = 4 & h(CP) = 7 \end{array}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$

	0	1	2	3	4	5	6	7
H			ZS 12 nil					

$$\begin{array}{lll} h(ZS) = 2 & h(FK) = 4 & h(PR) = 4 \\ h(VJ) = 6 & h(WF) = 4 & h(CP) = 7 \end{array}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$

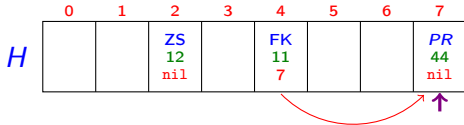
	0	1	2	3	4	5	6	7
H			ZS 12 nil		FK 11 nil			

$$\begin{array}{lll} h(ZS) = 2 & h(FK) = 4 & h(PR) = 4 \\ h(VJ) = 6 & h(WF) = 4 & h(CP) = 7 \end{array}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$

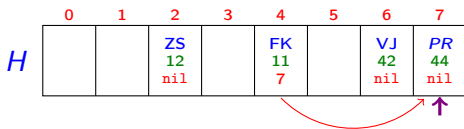


$$\begin{array}{lll} h(ZS) = 2 & h(FK) = 4 & h(PR) = 4 \\ h(VJ) = 6 & h(WF) = 4 & h(CP) = 7 \end{array}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$



$$\begin{array}{lll} h(ZS) = 2 & h(FK) = 4 & h(PR) = 4 \\ h(VJ) = 6 & h(WF) = 4 & h(CP) = 7 \end{array}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$

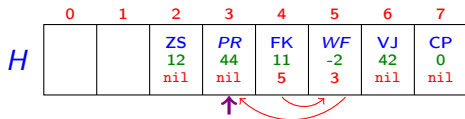
	0	1	2	3	4	5	6	7
H			ZS 12 nil		FK 11 5	WF -2 7	VJ 42 nil	PR 44 nil

$$\begin{aligned} h(ZS) &= 2 & h(FK) &= 4 & h(PR) &= 4 \\ h(VJ) &= 6 & h(WF) &= 4 & h(CP) &= 7 \end{aligned}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: **insert there**
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$



$$\begin{aligned}h(ZS) &= 2 & h(FK) &= 4 & h(PR) &= 4 \\h(VJ) &= 6 & h(WF) &= 4 & h(CP) &= 7\end{aligned}$$

Insertion in Lua's hashmap

Insertion of key x :

- If spot $h(x)$ is available: insert there
- If y is already at spot $h(x)$:
 - ▶ if $h(y) = h(x)$ then add x to a free spot and change the chain $y \rightarrow z \rightarrow \dots$ into $y \rightarrow x \rightarrow z \rightarrow \dots$ [*y at its main position*]
 - ▶ if $h(y) \neq h(x)$ find the predecessor w of y , y in a free spot, update the successor of w and place x at its main position $h(x)$
- Use an **index** to find a **free position**: it starts at the end and move step by step to the right until it finds a free spot (or exit the array)
- A newly inserted key is either in its **main position**, or at the **second position** in its list

Lua's hashmap analysis (insertion/search)

This is a classical setting, under the **uniform hashing assumption**, the **expected number of probes** is:

- **Unsuccessful** search: $U_\alpha \approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}$
- **Successful** search: $S_\alpha \approx \frac{1}{\alpha}(e^\alpha - 1)$

→ Efficient even if the hashtable is full!

Lua's hashmap analysis (insertion/search)

This is a classical setting, under the **uniform hashing assumption**, the **expected number of probes** is:

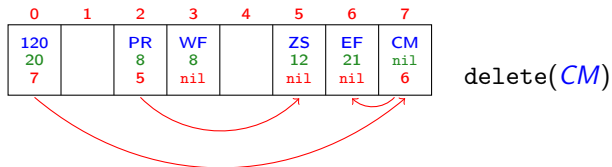
- **Unsuccessful** search: $U_\alpha \approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}$
- **Successful** search: $S_\alpha \approx \frac{1}{\alpha}(e^\alpha - 1)$

→ Efficient even if the hashtable is full!

What about deletions?

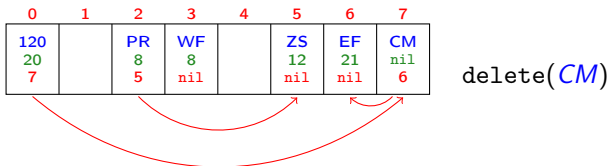
Deletion in Lua's hashmap

- To **delete** the **key** x , just set its **value** to **nil**



Deletion in Lua's hashmap

- To **delete** the **key** x , just set its **value** to **nil**

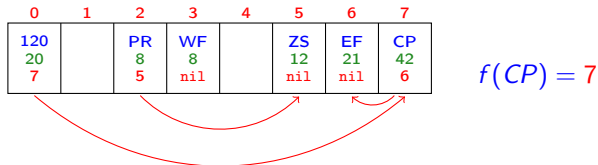
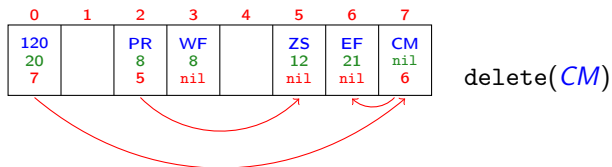


Insertion of key x :

- If spot $h(x)$ is **free** or the **value** is **nil**: insert there
- Otherwise, proceed as previously
- When looking for a free spot **do not consider** spots with a **nil** **value**

Deletion in Lua's hashmap

- To **delete** the **key** x , just set its **value** to **nil**



- The lists of hashvalue **0** and **7** coalesce!
- No more separate chaining if there are deletions

Rehashing

- When the pointer exits the hashmap (values -1), a **rehash** occurs
- The hashtable is then full (possibly with some **nil** values)
- To rehash:
 - ▶ compute the number n of **used keys** (with no **nil** value)
 - ▶ build a new hashtable of size $M = 2^m$, where m is the smallest integer such that $n + 1 \leq 2^m \leftarrow +1$ for the newly inserted key
 - ▶ reinsert all **used keys**

Rehashing

- When the pointer exits the hashmap (values -1), a **rehash** occurs
- The hashtable is then full (possibly with some **nil** values)
- To rehash:
 - ▶ compute the number n of **used keys** (with no **nil** value)
 - ▶ build a new hashtable of size $M = 2^m$, where m is the smallest integer such that $n + 1 \leq 2^m$ ← **+1** the newly inserted key
 - ▶ reinsert all **used keys**



Rehashing

- When the pointer exits the hashmap (values -1), a **rehash** occurs
- The hashtable is then full (possibly with some **nil** values)
- To rehash:
 - ▶ compute the number n of **used keys** (with no **nil** value)
 - ▶ build a new hashtable of size $M = 2^m$, where m is the smallest integer such that $n + 1 \leq 2^m$ ← **+1** the newly inserted key
 - ▶ reinsert all **used keys**



Bad worst-case scenario:

- insert until reaching a hashtable of size $M = 2^m$
 - alternate M deletion/insertion
 - Each insertion induces a rehash (if not right in the deleted bucket)
 - Complexity of $\Theta(M^2)$ for $3M$ operations.
- That's bad, but it's not a very realistic scenario in practice

A randomized scenario

We considered the following process:

- fix some $p \in (\frac{1}{2}, 1)$
- at each of the n steps
 - ▶ add a new element with probability p (or 1 if the map is empty)
 - ▶ remove an element of the map, uniformly, with probability $1 - p$

Not necessarily **realistic**, but we do want a hashtable to **perform well for this scenario**

A randomized scenario

We considered the following process:

- fix some $p \in (\frac{1}{2}, 1)$
- at each of the n steps
 - ▶ add a new element with probability p (or 1 if the map is empty)
 - ▶ remove an element of the map, uniformly, with probability $1 - p$

Not necessarily **realistic**, but we do want a hashtable to **perform well for this scenario**

Theorem (Martinez, Nicaud, Rotondo 2021)

With high probability, Lua uses $\Omega(n \log n)$ time for this process

→ not good, obviously!

Proof sketch

Theorem (Martínez, Nicaud, Rotondo 2021)

With high probability, Lua uses $\Omega(n \log n)$ time for this process

- The number of keys in the hashmap after t operations is $\approx (2p - 1)t$
- So we reach some size linear in t for the hashmap
- the first time we reach a size of 2^m , the hashmap contains $2^{m-1} + 1$ keys and $2^{m-1} - 1$ free spots

Proof sketch

Theorem (Martínez, Nicaud, Rotondo 2021)

With high probability, Lua uses $\Omega(n \log n)$ time for this process

- The number of keys in the hashmap after t operations is $\approx (2p - 1)t$
- So we reach some size linear in t for the hashmap
- the first time we reach a size of 2^m , the hashmap contains $2^{m-1} + 1$ keys and $2^{m-1} - 1$ free spots

Lemma

If the hashmap has size M and just after a rehash it contains $f \gg \sqrt{M}$ free spots, then at the next rehash it still has size M and contains at least γf free spots (whp).

→ Requires $\log M$ iterations to increase the hashmap size

Proof sketch

Lemma

If the hashmap has size M and just after a rehash it contains $f \gg \sqrt{M}$ free spots, then at the next rehash it still has size M and contains at least γf free spots (whp).

- δ_t the number of deleted spots (`nil` value)
- at rehash time t_0 we have $\delta_{t_0} = 0$

$$\delta_{t+1} = \begin{cases} \delta_t - 1 & \text{with probability } \frac{p\delta_t}{M} & \text{[insertion in a deleted key],} \\ \delta_t & \text{with probability } p \left(1 - \frac{\delta_t}{M}\right) & \text{[insertion in a free cell],} \\ \delta_t + 1 & \text{with probability } 1 - p & \text{[deletion].} \end{cases}$$

Rehash before we reach the equilibrium point at $\delta_t \approx \frac{1-p}{p} M$

Lua's hybrid tables

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>H</i>	120 20		PR 8	WF 8	EF 21	CM 11					CP 4	VJ 8	ZS 7		FK 7	

	1	2	3	4	5	6	7	8
<i>A</i>	6		7	16	21	22		4

- the array part is used for **keys** that are small integers
- the hash part for all other **keys**
- **idea**: automatically use an array if more convenient, the details are hidden to the programmer (simplicity & efficiency)
- we need a mechanism to choose the length of **A**

Lua's hybrid tables

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	120 20		PR 8	WF 8	EF 21	CM 11					CP 4	VJ 8	ZS 7		FK 7	
	1	2	3	4	5	6	7	8								
A	6		7	16	21	22		4								

- Size of the array part chosen **only when there is a rehash**
- Find the largest size 2^a s.t. there are more than 2^{a-1} keys in $[1 \dots 2^a]$

Analysis with only insertions

Even with no deletion, we have the same kind of problems as before

First example: insert $-(2^k - 1), -(2^k - 2), \dots, -1, 0, 1, \dots, 2^k$
→ this is done in $\Theta(k2^k) = \Theta(n \log n)$ time

Proposition

Lua's hybrid table needs $\mathcal{O}(n \log n)$ time to insert n keys, in the worst case.

Permutations and random permutations

Second example: inserting a permutation of $[1, \dots, n]$

Take $n = 3 \cdot 2^k$ and consider the order

$$2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \dots, 3 \cdot 2^k, 1, 2, \dots, 3 \cdot 2^k$$

Permutations and random permutations

Second example: inserting a permutation of $[1, \dots, n]$

Take $n = 3 \cdot 2^k$ and consider the order

$$2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \dots, 3 \cdot 2^k, 1, 2, \dots, 3 \cdot 2^k$$

- The 2^k first keys create a **full** hash-part of size 2^k
- With the insertion of $1, 2, \dots$ it behaves as for the previous example
- Need $\Theta(n \log n)$ time

Permutations and random permutations

Second example: inserting a permutation of $[1, \dots, n]$

Take $n = 3 \cdot 2^k$ and consider the order

$$2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \dots, 3 \cdot 2^k, 1, 2, \dots, 3 \cdot 2^k$$

- The 2^k first keys create a **full** hash-part of size 2^k
- With the insertion of $1, 2, \dots$ it behaves as for the previous example
- Need $\Theta(n \log n)$ time

Third example: uniform random permutation

Theorem (Martinez, Nicaud, Rotondo 2021)

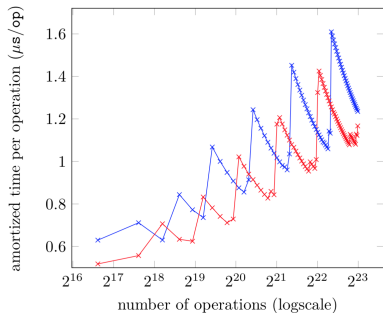
For any sequence $g(n) \rightarrow +\infty$, with high probability Lua's hybrid table insert a uniform random order on $[1, \dots, n]$ in time $\mathcal{O}(n \cdot g(n))$.

→ not so bad for once!

Experiments – random insertions/deletions

Theorem (Martinez, Nicaud, Rotondo 2021)

With high probability, Lua uses $\Omega(n \log n)$ time when inserting with probability $p \in (\frac{1}{2}, 1)$ and deleting with probability $1 - p$ (n operations)



$p = 0.75$

$p = 0.9$

Experiments – array part

```
m = 1<<24
tab = {}

for i=(-m+1),m do
    tab[i] = 1
end
```

LUA code for Example 1

- We insert $\approx 34 \cdot 10^6$ integer keys
- On a personal laptop: 21 seconds
- Ensuring at least 20% free space after a rehash: 3.5 seconds

⇒ Just change the number of used keys n to $5n/4$ before rehashing


Lua's tables: conclusion

- The hybrid data-structure is an interesting idea
- The problems in the hash algorithms can be fixed by **allowing more room** when rehashing
- This would also fix the hybrid part


From a theoretical point of view:

- We had to find **convincing models**
- Develop the probabilistic tools to analyze it

General conclusion

- There were **surprises** in the implementation choices made
- These innovations are often interesting
- Lots of them are motivated by recent changes in **computer architecture**
- A theoretical analysis seems mandatory to prevent **bugs** 
- Not always easy to be convincing: programmers are reluctant to modify algorithms used by lots of users

General conclusion

- There were **surprises** in the implementation choices made
- These innovations are often interesting
- Lots of them are motivated by recent changes in **computer architecture**
- A theoretical analysis seems mandatory to prevent **bugs** 
- Not always easy to be convincing: programmers are reluctant to modify algorithms used by lots of users

Thank you!



The Art Of Computer Programming, vol. 3: Sorting And Searching. Knuth. Addison-Wesley, (1973)



Partial solution and entropy. Takaoka. MFCS'19



On compressing permutations and adaptive sorting. Barbay, Navarro. TCS (2013)



OpenJDK's `Java.util.Collection.sort()` is broken: The good, the bad and the worst case. De Gouw, Rot, de Boer, Bubel, Hähnle. CAV'15



On the Worst-Case Complexity of TimSort. Auger, Jugé, Nicaud, Pivoteau. ESA'18



Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs. Munro, Wild. ESA'18



Strategies for stable merge sorting. Buss, Knop. SODA'19



Adaptive Shivers Sort: An alternative sorting algorithm. Jugé. SODA'20



Galloping in fast-growth natural merge sorts. Jugé, Khalighinejad, Ghasemi. ICALP'22



A Probabilistic Model Revealing Shortcomings in Lua's Hybrid Tables. Martínez, Nicaud, Rotondo. COCOON'22