On the Worst-Case Complexity of TimSort

Nicolas Auger
Université Paris-Est, LIGM (UMR 8049), UPEM, F77454 Marne-la-Vallée, France

Vincent Jugé
Université Paris-Est, LIGM (UMR 8049), UPEM, F77454 Marne-la-Vallée, France

Cyril Nicaud
Université Paris-Est, LIGM (UMR 8049), UPEM, F77454 Marne-la-Vallée, France

Carine Pivoteau
Université Paris-Est, LIGM (UMR 8049), UPEM, F77454 Marne-la-Vallée, France

Abstract

TimSort is an intriguing sorting algorithm designed in 2002 for Python, whose worst-case complexity was announced, but not proved until our recent preprint. In fact, there are two slightly different versions of TimSort that are currently implemented in Python and in Java respectively. We propose a pedagogical and insightful proof that the Python version runs in $\mathcal{O}(n \log n)$. The approach we use in the analysis also applies to the Java version, although not without very involved technical details. As a byproduct of our study, we uncover a bug in the Java implementation that can cause the sorting method to fail during the execution. We also give a proof that Python's TimSort running time is in $\mathcal{O}(n + n \log \rho)$, where ρ is the number of runs (i.e. maximal monotonic sequences), which is quite a natural parameter here and part of the explanation for the good behavior of TimSort on partially sorted inputs.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Sorting algorithms, Merge sorting algorithms, TimSort, Analysis of algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.4

1 Introduction

TimSort is a sorting algorithm designed in 2002 by Tim Peters [9], for use in the Python programming language. It was thereafter implemented in other well-known programming languages such as Java. The algorithm includes many implementation optimizations, a few heuristics and some refined tuning, but its high-level principle is rather simple: The sequence S to be sorted is first decomposed greedily into monotonic runs (i.e. nonincreasing or nondecreasing subsequences of S as depicted on Figure 1), which are then merged pairwise according to some specific rules.

The idea of starting with a decomposition into runs is not new, and already appears in Knuth's NaturalMergeSort [6], where increasing runs are sorted using the same mechanism as in MergeSort. Other merging strategies combined with decomposition into runs appear in the literature, such as the MinimalSort of [10] (see also [2, 8] for other considerations on the same topic). All of them have nice properties: they run in $\mathcal{O}(n \log n)$ and even $\mathcal{O}(n + n \log \rho)$, where ρ is the number of runs, which is optimal in the model of sorting by comparisons [7], using the classical counting argument for lower bounds. And yet, among all these merge-based algorithms, TimSort was favored in several very popular programming languages, which suggests that it performs quite well in practice.

26th Annual European Symposium on Algorithms (ESA 2018).
Editors: Yossi Azar, Hannah Bast, and Grzegorz Herman; Article No. 4; pp. 4:1–4:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$$S = (\ \underbrace{12, 10, 7, 5}_{\text{first run}},\ \underbrace{7, 10, 14, 25, 36}_{\text{second run}},\ \underbrace{3, 5, 11, 14, 15, 21, 22}_{\text{third run}},\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}}\ )$$

Figure 1 A sequence and its *run decomposition* computed by TimSort: for each run, the first two elements determine if it is increasing or decreasing, then it continues with the maximum number of consecutive elements that preserves the monotonicity.

TimSort running time was implicitly assumed to be $\mathcal{O}(n \log n)$, but our unpublished preprint [1] contains, to our knowledge, the first proof of it. This was more than ten years after TimSort started being used instead of QuickSort in several major programming languages. The growing popularity of this algorithm invites for a careful theoretical investigation. In the present paper, we make a thorough analysis which provides a better understanding of the inherent qualities of the merging strategy of TimSort. Indeed, it reveals that, even without its refined heuristics,[1] this is an effective sorting algorithm, computing and merging runs on the fly, using only local properties to make its decisions.

As the analysis we made in [1] was a bit involved and clumsy, we first propose in Section 3 a new pedagogical and self-contained exposition that TimSort runs in $\mathcal{O}(n \log n)$ time, which we want both clear and insightful. Using the same approach, we also establish in Section 4 that it runs in $\mathcal{O}(n + n \log \rho)$, a question left open in our preprint and also in a recent work[2] on TimSort [4]. Of course, the first result follows from the second, but since we believe that each one is interesting on its own, we devote one section to each of them. Besides, the second result provides with an explanation to why TimSort is a very good sorting algorithm, worth considering in most situations where in-place sorting is not needed.

To introduce our last contribution, we need to look into the evolution of the algorithm: there are actually not one, but two main versions of TimSort. The first version of the algorithm contained a flaw, which was spotted in [5]: while the input was correctly sorted, the algorithm did not behave as announced (because of a broken invariant). This was discovered by De Gouw and his co-authors while trying to prove formally the correctness of TimSort. They proposed a simple way to patch the algorithm, which was quickly adopted in Python, leading to what we consider to be the real TimSort. This is the one we analyze in Sections 3 and 4. On the contrary, Java developers chose to stick with the first version of TimSort, and adjusted some tuning values (which depend on the broken invariant; this is explained in Sections 2 and 5) to prevent the bug exposed by [5]. Motivated by its use in Java, we explain in Section 5 how, at the expense of very complicated technical details, the elegant proofs of the Python version can be twisted to prove the same results for this older version. While working on this analysis, we discovered yet another error in the correction made in Java. Thus, we compute yet another patch, even if we strongly agree that the algorithm proposed and formally proved in [5] (the one currently implemented in Python) is a better option.

2 TimSort core algorithm

The idea of TimSort is to design a merge sort that can exploit the possible "non randomness" of the data, without having to detect it beforehand and without damaging the performances on random-looking data. This follows the ideas of adaptive sorting (see [7] for a survey on taking presortedness into account when designing and analyzing sorting algorithms).

[1] These heuristics are useful in practice, but do not change the worst-case complexity of the algorithm.
[2] In [4], the authors refined the analysis of [1] to obtain very precise bounds for the complexity of TimSort and of similar algorithms.

Algorithm 1: TimSort. (Python 3.6.5)

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

Note: The function `merge_force_collapse` repeatedly pops the last two runs on the stack $\mathcal{R}$, merges them and pushes the resulting run back on the stack.

`runs` $\leftarrow$ a run decomposition of S
$\mathcal{R} \leftarrow$ an empty stack
while `runs` $\neq \emptyset$ **do** // main loop of TimSort
  remove a run r from `runs` and push r onto $\mathcal{R}$
  `merge_collapse`($\mathcal{R}$)
if `height`($\mathcal{R}$) $\neq 1$ **then** // the height of $\mathcal{R}$ is its number of runs
  `merge_force_collapse`($\mathcal{R}$)

Algorithm 2: The `merge_collapse` procedure. (Python 3.6.5)

Input : A stack of runs $\mathcal{R}$
Result: The invariant of Equations (1) and (2) is established.

Note: The runs on the stack are denoted by $\mathcal{R}[1] \ldots \mathcal{R}[\texttt{height}(\mathcal{R})]$, from top to bottom. The length of run $\mathcal{R}[i]$ is denoted by r_i. The blue highlight indicates that the condition was not present in the original version of TimSort (this will be discussed in section 5).

while `height`($\mathcal{R}$) > 1 **do**
  $n \leftarrow$ `height`($\mathcal{R}$) $- 2$
  if $(n > 0$ `and` $\mathsf{r}_3 \leqslant \mathsf{r}_2 + \mathsf{r}_1)$ `or` $(n > 1$ `and` $\mathsf{r}_4 \leqslant \mathsf{r}_3 + \mathsf{r}_2)$ **then**
    if $\mathsf{r}_3 < \mathsf{r}_1$ **then**
      merge runs $\mathcal{R}[2]$ and $\mathcal{R}[3]$ on the stack
    else merge runs $\mathcal{R}[1]$ and $\mathcal{R}[2]$ on the stack
  else if $\mathsf{r}_2 \leqslant \mathsf{r}_1$ **then**
    merge runs $\mathcal{R}[1]$ and $\mathcal{R}[2]$ on the stack
  else break

The first feature of TimSort is to work on the natural decomposition of the input sequence into maximal runs. In order to get larger subsequences, TimSort allows both nondecreasing and decreasing runs, unlike most merge sort algorithms.

Then, the merging strategy of TimSort (Algorithm 1) is quite simple yet very efficient. The runs are considered in the order given by the run decomposition and successively pushed onto a stack. If some conditions on the lengths of the topmost runs of the stack are not satisfied after a new run has been pushed, this can trigger a series of merges between pairs of runs at the top or right under. And at the end, when all the runs in the initial decomposition have been pushed, the last operation is to merge the remaining runs two by two, starting at the top of the stack, to get a sorted sequence. The conditions on the stack and the merging rules are implemented in the subroutine called `merge_collapse` detailed in Algorithm 2. This is what we consider to be TimSort core mechanism and this is the main focus of our analysis.

Another strength of TimSort is the use of many effective heuristics to save time, such as ensuring that the initial runs are not to small thanks to an insertion sort or using a special technique called “galloping” to optimize the merges. However, this does not interfere with our analysis and we will not discuss this matter any further.

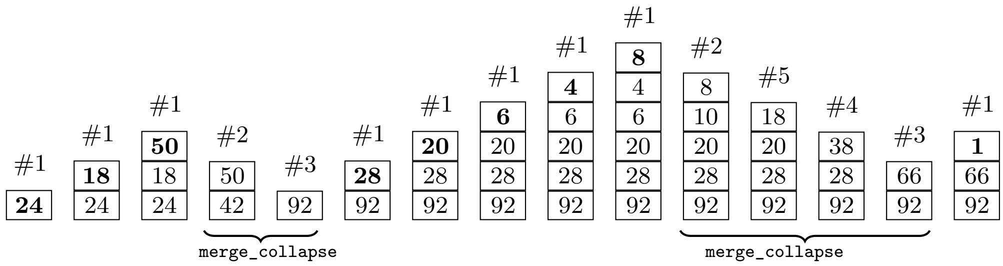

Figure 2 The successive states of the stack $\mathcal{R}$ (the values are the lengths of the runs) during an execution of the main loop of TimSort (Algorithm 1), with the lengths of the runs in `runs` being $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. The label #1 indicates that a run has just been pushed onto the stack. The other labels refer to the different merges cases of `merge_collapse` as translated in Algorithm 3.

Let us have a closer look at Algorithm 2 which is a pseudo-code transcription of the `merge_collapse` procedure found in the latest version of Python (3.6.5). To illustrate its mechanism, an example of execution of the main loop of TimSort (lines 3-5 of Algorithm 1) is given in Figure 2. As stated in its note [9], Tim Peter's idea was that:

> "The thrust of these rules when they trigger merging is to balance the run lengths as closely as possible, while keeping a low bound on the number of runs we have to remember."

To achieve this, the merging conditions of `merge_collapse` are designed to ensure that the following invariant[3] is true at the end of the procedure:

$$\mathsf{r}_{i+2} > \mathsf{r}_{i+1} + \mathsf{r}_i, \tag{1}$$

$$\mathsf{r}_{i+1} > \mathsf{r}_i. \tag{2}$$

This means that the runs lengths r_i on the stack grow at least as fast as the Fibonacci numbers and, therefore, that the height of the stack stays logarithmic (see Lemma 6, section 3).

Note that the bound on the height of the stack is not enough to justify the $\mathcal{O}(n \log n)$ running time of TimSort. Indeed, without the smart strategy used to merge the runs "on the fly", it is easy to build an example using a stack containing at most two runs and that gives a $\Theta(n^2)$ complexity: just assume that all runs have length two, push them one by one onto a stack and perform a merge each time there are two runs in the stack.

We are now ready to proceed with the analysis of TimSort complexity. As mentioned earlier, Algorithm 2 does not correspond to the first implementation of TimSort in Python, nor to the current one in Java, but to the latest Python version. The original version will be discussed in details later, in Section 5.

3 TimSort runs in $\mathcal{O}(n \log n)$

At the first release of TimSort [9], a time complexity of $\mathcal{O}(n \log n)$ was announced with no element of proof given. It seemed to remain unproved until our recent preprint [1], where we provide a confirmation of this fact, using a proof which is not difficult but a bit tedious. This result was refined later in [4], where the authors provide lower and upper bounds, including explicit multiplicative constants, for different merge sort algorithms.

[3] Actually, in [9], the invariant is only stated for the 3 topmost runs of the stack.

Algorithm 3: TimSort: translation of Algorithm 1 and Algorithm 2.

Input : A sequence to S to sort
Result: The sequence S is sorted into a single run, which remains on the stack.
Note: At any time, we denote the height of the stack $\mathcal{R}$ by h and its i^{th} top-most run (for $1 \leqslant i \leqslant h$) by R_i. The length of this run is denoted by r_i.

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

Our main concern is to provide an insightful proof of the complexity of TimSort, in order to highlight how well designed is the strategy used to choose the order in which the merges are performed. The present section is more detailed than the following ones as we want it to be self-contained once TimSort has been translated into Algorithm 3 (see below).

As our analysis is about to demonstrate, in terms of worst-case complexity, the good performances of TimSort do not rely on the way merges are performed. Thus we choose to ignore their many optimizations and consider that merging two runs of lengths r and r' requires both $r + r'$ element moves and $r + r'$ element comparisons. Therefore, to quantify the running time of TimSort, we only take into account the number of comparisons performed.

▶ **Theorem 1.** *The running time of* TimSort *is* $\mathcal{O}(n \log n)$.

The first step consists in rewriting Algorithm 1 and Algorithm 2 in a form that is easier to deal with. This is done in Algorithm 3.

▶ **Claim 2.** *For any input, Algorithms 1 and 3 perform the same comparisons.*

Proof. The only difference is that Algorithm 2 was changed into the `while` loop of lines 5 to 10 in Algorithm 3. Observing the different cases, it is straightforward to verify that merges involving the same runs take place in the same order in both algorithms. Indeed, if $r_3 < r_1$, then $r_3 \leqslant r_1 + r_2$, and therefore line 5 is triggered in Algorithm 2, so that both algorithms merge the 2nd and 3rd runs. On the contrary, if $r_3 \geqslant r_1$, then both algorithms merge the 1st and 2nd runs if and only if $r_2 \leqslant r_1$ or $r_3 \leqslant r_1 + r_2$ (or $r_4 \leqslant r_2 + r_3$). ◀

▶ **Remark 3.** Proving Theorem 1 only requires analyzing the *main loop* of the algorithm (lines 3 to 10). Indeed, computing the run decomposition (line 1) can be done on the fly, by a greedy algorithm, in time linear in n, and the *final loop* (line 11) might be performed in the main loop by adding a fictitious run of length $n + 1$ at the end of the decomposition.

In the sequel, for the sake of readability, we also omit checking that h is large enough to trigger the cases #2 to #5. Once again, such omissions are benign, since adding fictitious runs of respective sizes $8n$, $4n$, $2n$ and n (in this order) at the beginning of the decomposition would ensure that $h \geqslant 4$ during the whole loop.

In Algorithm 3, we can see that the merges performed during Case #2 allow a very large run to be pushed and "absorbed" onto the stack without being merged all the way down, but by collapsing the stack under this run instead. Meanwhile, the purpose of Cases #3 to #5 is mainly to re-establish the invariant of Equations (1) and (2), ensuring an exponential growth of the run lengths within the stack (this duality is made even clearer in the proof of Section 4). Along this process, the cost of keeping the stack in good shape is compensated by the absorption of this large run, which naturally calls for an *amortized complexity* analysis.

To proceed with the core of our proof (that is the amortized analysis of the main loop), we now credit tokens to the elements of the input array, which are spent for comparisons. One token is paid for every comparison performed by the algorithm and each element is given $\mathcal{O}(\log n)$ tokens. Since the balance is always non-negative, we can conclude that at most $\mathcal{O}(n \log n)$ comparisons are performed, in total, during the main loop.

Elements of the input array are easily identified by their starting position in the array, so we consider them as well-defined and distinct entities (even if they have the same value). The *height* of an element is the number of runs that are below it in the stack: the elements belonging to the run R_i in the stack $(R_1, \ldots, R_h)$ have height $h - i$. To simplify the presentation, we also distinguish two kinds of tokens, the $\diamondsuit$-tokens and the $\heartsuit$-tokens, which can both be used to pay for comparisons.

Two $\diamondsuit$-tokens and one $\heartsuit$-token are credited to an element when it enters the stack (this is Case #1 of Algorithm 3) or when its height decreases: all the elements of R_1 are credited when R_1 and R_2 are merged, and all the elements of R_1 and R_2 are credited when R_2 and R_3 are merged. Tokens are spent to pay for comparisons, depending on the case triggered:

- Case #2: every element of R_1 and R_2 pays 1 $\diamondsuit$. This is enough to cover the cost of merging R_2 and R_3, since $r_2 + r_3 \leqslant r_2 + r_1$, as $r_3 < r_1$ in this case.
- Case #3: every element of R_1 pays 2 $\diamondsuit$. In this case $r_1 \geqslant r_2$ and the cost is $r_1 + r_2 \leqslant 2r_1$.
- Cases #4 and #5: every element of R_1 pays 1 $\diamondsuit$ and every element of R_2 pays 1 $\heartsuit$. The cost $r_1 + r_2$ is exactly the number of tokens spent.

▶ **Lemma 4.** *The balances of $\diamondsuit$-tokens and $\heartsuit$-tokens of each element remain non-negative throughout the main loop of* TimSort.

Proof. In all four cases #2 to #5, because the height of the elements of R_1 and possibly the height of those of R_2 decrease, the number of credited $\diamondsuit$-tokens after the merge is at least the number of $\diamondsuit$-tokens spent. The $\heartsuit$-tokens are spent in Cases #4 and #5 only: every element of R_2 pays one $\heartsuit$-token, and then belongs to the topmost run $\overline{R}_1$ of the new stack $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{h-1})$ obtained after merging R_1 and R_2. Since $\overline{R}_i = R_{i+1}$ for $i \geqslant 2$, the condition of Case #4 implies that $\overline{r}_1 \geqslant \overline{r}_2$ and the condition of Case #5 implies that $\overline{r}_1 + \overline{r}_2 \geqslant \overline{r}_3$: in both cases, the next modification of the stack $\overline{\mathcal{S}}$ is another merge. This merge decreases the height of $\overline{R}_1$, and therefore decreases the height of the elements of R_2, who will regain one $\heartsuit$-token without losing any, since the topmost run of the stack never pays with $\heartsuit$-tokens. This proves that, whenever an element pay one $\heartsuit$-token, the next modification is another merge during which it regains its $\heartsuit$-token. This concludes the proof by direct induction. ◀

Let $h_{\max}$ be the maximum number of runs in the stack during the whole execution of the algorithm. Due to the crediting strategy, each element is given at most $2h_{\max}$ $\diamondsuit$-tokens and at most $h_{\max}$ $\heartsuit$-tokens in total. So we only need to prove that $h_{\max}$ is $\mathcal{O}(\log n)$ to complete the proof that the main loop running time is in $\mathcal{O}(n \log n)$. This fact is a consequence of TimSort's invariant established with a formal proof in the theorem prover KeY [3, 5]: at the end of any iteration of the main loop, we have $r_i + r_{i+1} < r_{i+2}$, for every $i \geqslant 1$ such that the run R_{i+2} exists.

For completeness, and because the formal proof is not meant to be read by humans, we sketch a "classical" proof of the invariant. It is not exactly the same statement as in [5], since our invariant holds at any time during the main loop: in particular we cannot say anything about R_1, which can have any length when a run has just been added. For technical reasons, and because it will be useful later on, we establish four invariants in our statement.

▶ **Lemma 5.** *At any step during the main loop of* TimSort*, we have* (i) $r_i + r_{i+1} < r_{i+2}$ *for* $i \in \{3, \dots, h-2\}$, (ii) $r_2 < 3r_3$, (iii) $r_3 < r_4$ *and* (iv) $r_2 < r_3 + r_4$.

Proof. The proof is done by induction. It consists in verifying that, if all four invariants hold at some point, then they still hold when an update of the stack occurs in one of the five situations labeled #1 to #5 in the algorithm. This can be done by a straightforward case analysis. We denote by $\overline{\mathcal{S}} = (\overline{R}_1, \dots, \overline{R}_{\overline{h}})$ the new state of the stack after the update:

- If Case #1 just occurred, a new run $\overline{R}_1$ was pushed. This implies that none of the conditions of Cases #2 to #5 hold in $\mathcal{S}$, otherwise merges would have continued. In particular, we have $r_1 < r_2 < r_3$ and $r_2 + r_3 < r_4$. As $\overline{r}_i = r_{i-1}$ for $i \geqslant 2$, and invariant (i) holds for $\mathcal{S}$, we have $\overline{r}_2 < \overline{r}_3 < \overline{r}_4$, and thus invariants (i) to (iv) hold for $\overline{\mathcal{S}}$.
- If one of the Cases #2 to #5 just occurred, we have $\overline{r}_2 = r_2 + r_3$ (in Case #2) or $\overline{r}_2 = r_3$ (in Cases #3 to #5). This implies that $\overline{r}_2 \leqslant r_2 + r_3$. As $\overline{r}_i = r_{i+1}$ for $i \geqslant 3$, and invariants (i) to (iv) hold for $\mathcal{S}$, we have $\overline{r}_2 \leqslant r_2 + r_3 < r_3 + r_4 + r_3 < 3r_4 = 3\overline{r}_3$, $\overline{r}_3 = r_4 \leqslant r_3 + r_4 < r_5 = \overline{r}_4$, and $\overline{r}_2 \leqslant r_2 + r_3 < r_3 + r_4 + r_3 < r_3 + r_5 < r_4 + r_5 = \overline{r}_3 + \overline{r}_4$. Thus, invariants (i) to (iv) hold for $\overline{\mathcal{S}}$. ◀

At this point, invariant (i) can be used to bound $h_{\max}$ from above.

▶ **Lemma 6.** *At any time during the main loop of* TimSort*, if the stack is* $(R_1, \dots, R_h)$ *then we have* $r_2/3 < r_3 < r_4 < \dots < r_h$ *and, for all* $i \geqslant j \geqslant 3$, *we have* $r_i > \sqrt{2}^{\,i-j-1} r_j$. *As a consequence, the number of runs in the stack is always* $\mathcal{O}(\log n)$.

Proof. By Lemma 5, we have $r_i + r_{i+1} < r_{i+2}$ for $3 \leqslant i \leqslant h-2$. Thus $r_{i+2} - r_{i+1} > r_i > 0$ and the sequence is increasing from index 4: $r_4 < r_5 < r_6 < \dots < r_h$. The increasing sequence of the statement is then obtained using the invariants (ii) and (iii). Hence, for $j \geqslant 3$, we have $r_{j+2} > 2r_j$, from which one can get that $r_i > \sqrt{2}^{\,i-j-1} r_j$. In particular, if $h \geqslant 3$ then $r_h > \sqrt{2}^{\,h-4} r_3$, which yields that the number of runs is $\mathcal{O}(\log n)$ as $r_h \leqslant n$. ◀

Collecting all the above results is enough to prove Theorem 1. First, as mentioned in Remark 3, computing the run decomposition can be done in linear time. Then, we proved that the main loop requires $\mathcal{O}(n h_{\max})$ comparisons, by bounding from above the total number of tokens credited, and that $h_{\max} = \mathcal{O}(\log n)$, by showing that the run lengths grow at exponential speed. Finally, the final merges of line 11 might be taken care of by Remark 3, but they can also be dealt with directly:[4] if we start these merges with a stack $S = (R_1, \dots, R_h)$, then every element of the run R_i takes part in $h + 1 - i$ merges at most, which proves that the overall cost of line 11 is $\mathcal{O}(n \log n)$. This concludes the proof of the theorem.

4 Refined analysis parametrized with the number of runs

A widely spread idea to explain why certain sorting algorithms perform better in practice than expected is that they are able to exploit presortedness [7]. This can be quantified in

[4] Relying on Remark 3 will be necessary only in the next section, where we need more precise computations.

$$\underbrace{\#1\ \#2\ \#2\ \#2}_{\text{starting seq.}}\quad \underbrace{\#3\ \#2\ \#5\ \#2\ \#4\ \#2}_{\text{ending seq.}}\quad \underbrace{\#1\ \#2\ \#2\ \#2\ \#2\ \#2}_{\text{starting seq.}}\quad \underbrace{\#5\ \#2\ \#3\ \#3\ \#4\ \#2}_{\text{ending seq.}}$$

Figure 3 The decomposition of the encoding of an execution into starting and ending sequences.

many ways, the number of runs in the input sequence being one. Since this is the most natural parameter, we now consider the complexity of TimSort, according to it. We establish the following result, which was left open in [1, 4]:

▶ **Theorem 7.** *The complexity of* TimSort *on inputs of size n with ρ runs is $\mathcal{O}(n + n\log\rho)$.*

If $\rho = 1$, then no merge is to be performed, and the algorithm clearly runs in time linear in n. Hence, we assume below that $\rho \geqslant 2$, and we show that the complexity of TimSort is $\mathcal{O}(n\log\rho)$ in this case.

To obtain the $\mathcal{O}(n\log\rho)$ complexity, we need to distinguish several situations. First, consider the sequence of Cases #1 to #5 triggered during the execution of the main loop of TimSort. It can be seen as a word on the alphabet $\{\#1,\ldots,\#5\}$ that starts with #1, which completely encodes the execution of the algorithm. We split this word at every #1, so that each piece corresponds to an iteration of the main loop. Those pieces are in turn split into two parts, at the first occurrence of a symbol #3, #4 or #5. The first half is called a *starting sequence* and is made of a #1 followed by the maximal number of #2's. The second half is called an *ending sequence*, it starts with #3, #4 or #5 (or is empty) and it contains no occurrence of #1 (see Figure 3 for an example).

We treat starting and ending sequences separately in our analysis. The following lemma points out one of the main reasons TimSort is so efficient regarding the number of runs.

▶ **Lemma 8.** *The number of comparisons performed during all the starting sequences is $\mathcal{O}(n)$.*

Proof. More precisely, for a stack $\mathcal{S} = (R_1,\ldots,R_h)$, we prove that a starting sequence beginning with a push of a run R of length r onto $\mathcal{S}$ uses at most γr comparisons in total, where γ is the real constant $3\sqrt{2}\sum_{i\geqslant 0} i/\sqrt{2}^i$. After the push, the stack is $\overline{\mathcal{S}} = (R, R_1,\ldots,R_h)$ and, if the starting sequence contains $k \geqslant 1$ letters, i.e. $k-1$ occurrences of #2, then this sequence amounts to merging the runs R_1, R_2, …, R_k. Since no merge is performed if $k = 1$, we assume below that $k \geqslant 2$.

Looking closely at these runs, we compute that they require a total of

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

comparisons. The last occurrence of Case #2 ensures that $r > r_k$, hence applying Lemma 6 to the stack $\overline{\mathcal{S}}$ shows that $r > \sqrt{2}^{k-i} r_i/3$ for all $i = 1,\ldots,k$. It follows that

$$C/r < 3\sum_{i=2}^{k}(k+1-i)/\sqrt{2}^{k-i} < \gamma.$$

This concludes the proof, since each run is the beginning of exactly one starting sequence, and the sum of their lengths is n. ◀

We can now focus on the cost of ending sequences. Because the inner loop (line 5) of TimSort has already begun, during the corresponding starting sequence, we have some information on the length of the topmost run.

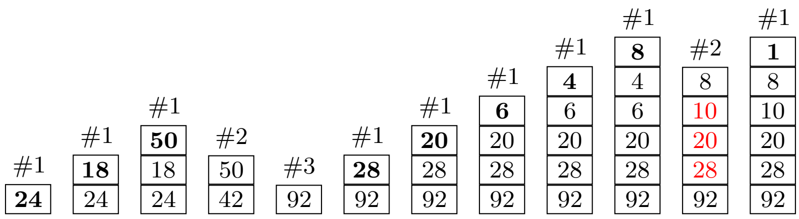

Figure 4 Execution of the main loop of Java's TimSort (Algorithm 3, without merge case #5, at line 9), with the lengths of the runs in `runs` being $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. When the second to last run (of length 8) is pushed onto the stack, the while loop of line 5 stops after only one merge, breaking the invariant (in red), unlike what we see in Figure 2 using the Python version of TimSort.

▶ **Lemma 9.** *At any time during an ending sequence, including just before it starts and just after it ends, we have $r_1 < 3r_3$.*

Proof. The proof is done by induction. At the beginning of the ending sequence, the condition of #2 cannot be true, so $r_1 \leqslant r_3 < 3r_3$. Before any merge during an ending sequence, if the stack is $\mathcal{S} = (R_1, \ldots R_h)$, then we denote by $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{h-1})$ the stack after that merge. If the invariant holds before the merge, and since $r_2 < r_3 + r_4$ and $r_3 < r_4$ by Lemma 5, we have $\overline{r}_1 = r_1 < 3r_3 < 3r_4 = 3\overline{r}_3$ in Case #2, and $\overline{r}_1 = r_1 + r_2 \leqslant r_3 + r_2 < r_3 + r_3 + r_4 < 3r_4 = 3\overline{r}_3$ in Cases #3 to #5 (since $r_1 \leqslant r_3$, as Case #2 does not apply), concluding the proof. ◀

In order to obtain a suitable upper bound for the merges that happen during ending sequences, we refine the analysis of the previous section. We still use ♢-tokens and ♡-tokens to pay for comparisons when the stack is not too high, and we use different tokens otherwise:

- at the beginning of the algorithm, a common pool is credited with $24\,n$ ♣-tokens,
- all elements are still credited two ♢-tokens and one ♡-token when entering the stack,
- no token (of any kind) is credited nor spent during merges of starting sequences (the cost of such sequences is already taken care of by Lemma 9),
- if the stack has height less than $\kappa = \lceil 2 \log_2 \rho \rceil$, elements are credited ♢-tokens and ♡-tokens and merges (of ending sequences) are paid in the same fashion as in Section 3,
- if the stack has height at least κ, then merges (of ending sequences) are paid using ♣-tokens, and elements are not credited any token when a merge decreases their height.

By the analysis of the previous section, at most $\mathcal{O}(n\kappa)$ comparisons are paid with ♢-tokens and ♡-tokens. Hence, using Remark 3, we complete the proof of Theorem 7 by checking that we initially credited enough ♣-tokens. This is a direct consequence of the following lemma, since at most ρ merges are paid by ♣-tokens.

▶ **Lemma 10.** *A merge performed during an ending sequence with a stack containing at least κ runs costs at most $24n/\rho$ comparisons.*

Proof. Lemmas 5 and 9 prove that $r_2 < 3r_3$ and $r_1 < 3r_3$. Since a merging step either merges R_1 and R_2, or R_2 and R_3, it requires at most $6r_3$ comparisons. By Lemma 6, we have $r_h \geqslant \sqrt{2}^{h-4} r_3$, whence $6r_3 \leqslant 24\sqrt{2}^{-h} r_h \leqslant 24n\sqrt{2}^{-\kappa} \leqslant 24n/\rho$. ◀

5 About the Java version of TimSort

Algorithm 2 (and therefore Algorithm 3) does not correspond to the original TimSort. Before release 3.4.4 of Python, the second part of the condition (in blue) in the test at line 3 of `merge_collapse` (and therefore merge case #5 of Algorithm 3) was missing. This version

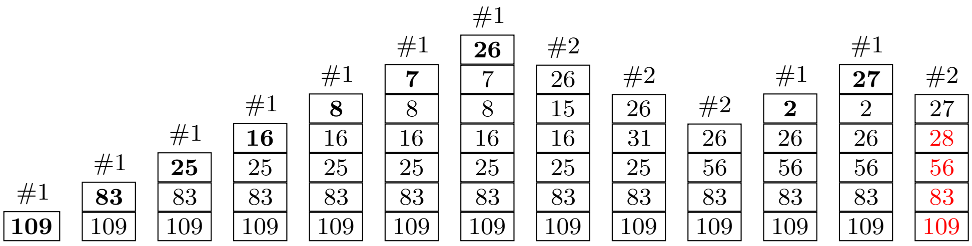

Figure 5 Execution of the main loop of the Java version of TimSort (without merge case #5, at line 9 of Algorithm 3), with the lengths of the runs in `runs` being $(109, 83, 25, 16, 8, 7, 26, 2, 27)$. When the algorithm stops, the invariant is violated twice, for consecutive runs (in red).

of the algorithm worked fine, meaning that it did actually sort arrays, but the invariant given by Equation (1) did not hold. Figure 4 illustrates the difference caused by the missing condition when running Algorithm 3 on the same input as in Figure 2.

This was discovered by de Gouw *et al.* [5] when trying to prove the correctness of the Java implementation of TimSort (which is the same as in the earlier versions of Python). And since the Java version of the algorithm uses the (wrong) invariant to compute the maximum size of the stack used to store the runs, the authors were able to build a sequence of runs that causes the Java implementation of TimSort to crash. They proposed two solutions to fix TimSort: reestablish the invariant, which led to the current Python version, or keep the original algorithm and compute correct bounds for the stack size, which is the solution that was chosen in Java 9 (note that this is the second time these values had to be changed). To do the latter, the developers used the claim in [5] that the invariant cannot be violated for two consecutive runs on the stack, which turns out to be false,[5] as illustrated in Figure 5. Thus, it is still possible to cause the Java implementation to fail: it uses a stack of runs of size at most 49 and we were able to compute an example requiring a stack of size 50 (see `http://igm.univ-mlv.fr/~pivoteau/Timsort/Test.java`), causing an error at runtime in Java's sorting method.

Even if the bug we highlighted in Java's TimSort is very unlikely to happen, this should be corrected. And, as advocated by de Gouw *et al.* and Tim Peters himself,[6] we strongly believe that the best solution would be to correct the algorithm as in the current version of Python, in order to keep it clean and simple. However, since this is the implementation of Java's sort for the moment, there are two questions we would like to tackle: Does the complexity analysis holds without the missing condition? And, can we compute an actual bound for the stack size? We first address the complexity question. It turns out that the missing invariant was a key ingredient for having a simple and elegant proof.

▶ **Proposition 11.** *At any time during the main loop of Java's* TimSort*, if the stack of runs is* $(R_1, \ldots, R_h)$ *then we have* $r_3 < r_4 < \ldots < r_h$ *and, for all* $i \geqslant 3$*, we have* $(2 + \sqrt{7})r_i \geqslant r_2 + \ldots + r_{i-1}$.

Proof ideas. The proof of Proposition 11 is much more technical and difficult than insightful, and therefore we just summarize its main steps. As in previous sections, this proof relies on several inductive arguments, using both inductions on the number of merges performed,

[5] This is the consequence of a small error in the proof of their Lemma 1. The constraint $C_1 > C_2$ has no reason to be. Indeed, in our example, we have $C_1 = 25$ and $C_2 = 31$.
[6] Here is the discussion about the correction in Python: `https://bugs.python.org/issue23515`.

on the stack size and on the run lengths. The inequalities $r_3 < r_4 < \ldots < r_h$ come at once, hence we focus on the second part of Proposition 11.

Since separating starting and ending sequences was useful in Section 4, we first introduce the notion of *stable* stacks: a stack $\mathcal{S}$ is stable if, when operating on the stack $\mathcal{S} = (R_1, \ldots, R_h)$, Case #1 is triggered (i.e. Java's TimSort is about to perform a *run push* operation).

We also call *obstruction indices* the integers $i \geqslant 3$ such that $r_i \leqslant r_{i-1} + r_{i-2}$: although they do not exist in Python's TimSort, they may exist, and even be consecutive, in Java's TimSort. We prove that, if $i-k, i-k+1, \ldots, i$ are obstruction indices, then the stack sizes $r_{i-k-2}, \ldots, r_i$ grow "at linear speed". For instance, in the last stack of Figure 5, obstruction indices are 4 and 5, and we have $r_2 = 28$, $r_3 = r_2 + 28$, $r_4 = r_3 + 27$ and $r_5 = r_4 + 26$.

Finally, we study so-called *expansion functions*, i.e. functions $f : [0,1] \mapsto \mathbb{R}$ such that, for every stable stack $\mathcal{S} = (R_1, \ldots, R_h)$, we have $r_2 + \ldots + r_{h-1} \leqslant r_h f(r_{h-1}/r_h)$. We exhibit an explicit function f such that $f(x) \leqslant 2 + \sqrt{7}$ for all $x \in [0,1]$, and we prove by induction on r_h that f is an expansion function, from which we deduce Proposition 11. ◀

Once Proposition 11 is proved, we easily recover the following variant of Lemmas 6 and 9.

▶ **Lemma 12.** *At any time during the main loop of Java's* TimSort, *if the stack is* $(R_1, \ldots, R_h)$ *then we have* $r_2/(2+\sqrt{7}) \leqslant r_3 < r_4 < \ldots < r_h$ *and, for all* $i \geqslant j \geqslant 3$, *we have* $r_i \geqslant \delta^{i-j-4} r_j$, *where* $\delta = \big(5/(2+\sqrt{7})\big)^{1/5} > 1$. *Furthermore, at any time during an ending sequence, including just before it starts and just after it ends, we have* $r_1 \leqslant (2+\sqrt{7}) r_3$.

Proof. The inequalities $r_2/(2+\sqrt{7}) \leqslant r_3 < r_4 < \ldots < r_h$ are just a (weaker) restatement of Proposition 11. Then, for $j \geqslant 3$, we have $(2+\sqrt{7}) r_{j+5} \geqslant r_j + \ldots + r_{j+4} \geqslant 5 r_j$, i.e. $r_{j+5} \geqslant \delta^5 r_j$, from which one gets that $r_i \geqslant \delta^{i-j-4} r_j$.

Finally, we prove by induction that $r_1 \leqslant (2+\sqrt{7}) r_3$ during ending sequences. First, when the ending sequence starts, $r_1 < r_3 \leqslant (2+\sqrt{7}) r_3$. Before any merge during this sequence, if the stack is $\mathcal{S} = (R_1, \ldots R_h)$, then we denote by $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{h-1})$ the stack after the merge. If the invariant holds before the merge, in Case #2, we have $\overline{r}_1 = r_1 \leqslant (2+\sqrt{7}) r_3 \leqslant (2+\sqrt{7}) r_4 = (2+\sqrt{7}) \overline{r}_3$; and using Proposition 11 in Cases #3 and #4, we have $\overline{r}_1 = r_1 + r_2$ and $r_1 \leqslant r_3$, hence $\overline{r}_1 = r_1 + r_2 \leqslant r_2 + r_3 \leqslant (2+\sqrt{7}) r_4 = (2+\sqrt{7}) \overline{r}_3$, concluding the proof. ◀

We can then recover a proof of complexity for the Java version of TimSort, by following the same proof as in Sections 3 and 4, but using Lemma 12 instead of Lemmas 6 and 9.

▶ **Theorem 13.** *The complexity of Java's* TimSort *on inputs of size* n *with* ρ *runs is* $\mathcal{O}(n + n \log \rho)$.

Another question is that of the stack size requirements of Java's TimSort, i.e. computing $h_{\max}$. A first result is the following immediate corollary of Lemma 12.

▶ **Corollary 14.** *On an input of size* n, *Java's* TimSort *will create a stack of runs of maximal size* $h_{\max} \leqslant 7 + \log_\delta(n)$, *where* $\delta = \big(5/(2+\sqrt{7})\big)^{1/5}$.

Proof. At any time during the main loop of Java's TimSort on an input of size n, if the stack is $(R_1, \ldots, R_h)$ and $h \geqslant 3$, it follows from Lemma 12 that $n \geqslant r_h \geqslant \delta^{h-7} r_3 \geqslant \delta^{h-7}$. ◀

Unfortunately, for integers smaller than 2^{31}, Corollary 14 only proves that the stack size will never exceed 347. However, in the comments of Java's implementation of TimSort,[7]

[7] Comment at line 168: `http://igm.univ-mlv.fr/~pivoteau/Timsort/TimSort.java`.

there is a remark that keeping a short stack is of some importance, for practical reasons, and that the value chosen in Python– 85 – is “too expensive”. Thus, in the following, we go to the extent of computing the optimal bound. It turns out that this bound cannot exceed 86 for such integers. This bound could possibly be refined slightly, but definitely not to the point of competing with the bound that would be obtained if the invariant of Equation (1) were correct. Once more, this suggests that implementing the new version of TimSort in Java would be a good idea, as the maximum stack height is smaller in this case.

▶ **Theorem 15.** *On an input of size n, Java's* TimSort *will create a stack of runs of maximal size $h_{\max} \leqslant 3 + \log_\Delta(n)$, where $\Delta = (1 + \sqrt{7})^{1/5}$. Furthermore, if we replace Δ by any real number $\Delta' > \Delta$, the inequality fails for all large enough n.*

Proof ideas. The first part of Theorem 15 is proved as follows. Ideally, we would like to show that $r_{i+j} \geqslant \Delta^j r_i$ for all $i \geqslant 3$ and some fixed integer j. However, these inequalities do not hold for all i. Yet, we prove that they hold if $i + 2$ and $i + j + 2$ are not obstruction indices and if $i + j + 1$ is an obstruction index. It follows quickly that $r_h \geqslant \Delta^{h-3}$.

The optimality of Δ is much more difficult to prove. It turns out that the constants $2+\sqrt{7}$, $(1 + \sqrt{7})^{1/5}$, and the expansion function referred to in the proof of Proposition 11 were constructed as least fixed points of non-decreasing operators, although this construction needed not be explicit for using these constants and function. Hence, we prove that Δ is optimal by inductively constructing sequences of run lengths that show that $\limsup\{\log(r_h)/h\} \geqslant \Delta$; much care is required for proving that our constructions are indeed feasible. ◀

6 Conclusion

At first, when we learned that Java's QuickSort had been replaced by a variant of MergeSort, we thought that this new algorithm – TimSort – should be really fast and efficient in practice, and that we should look into its average complexity to confirm this from a theoretical point of view. Then, we realized that its worst-case complexity had not been formally established yet and we first focused on giving a proof that it runs in $\mathcal{O}(n \log n)$, which we wrote in a preprint [1]. In the present article, we simplify this preliminary work and provide a short, simple and self-contained proof of TimSort's complexity, which sheds some light on the behavior of the algorithm. Based on this description, we were also able to answer positively a natural question, which was left open so far: does TimSort runs in $\mathcal{O}(n + n \log \rho)$, where ρ is the number of runs? We hope our theoretical work highlights that TimSort is actually a very good sorting algorithm. Even if all its fine-tuned heuristics are removed, the dynamics of its merges, induced by a small number of local rules, results in a very efficient global behavior, particularly well suited for *almost sorted* inputs.

Besides, we want to stress the need for a thorough algorithm analysis, in order to prevent errors and misunderstandings. As obvious as it may sound, the three consecutive mistakes on the stack height in Java illustrate perfectly how the best ideas can be spoiled by the lack of a proper complexity analysis.

Finally, following [5], we would like to emphasize that there seems to be no reason not to use the recent version of TimSort, which is efficient in practice, formally certified and whose optimal complexity is easy to understand.

References

1 Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: From Merge Sort to TimSort. Research Report hal-01212839, hal, 2015. URL: `https://hal-upec-upem.archives-ouvertes.fr/hal-01212839`.

2 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013. `doi:10.1016/j.tcs.2013.10.019`.

3 Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

4 Sam Buss and Alexander Knop. Strategies for stable merge sorting. Research Report abs/1801.04641, arXiv, 2018. URL: `http://arxiv.org/abs/1801.04641`.

5 Stijn De Gouw, Jurriaan Rot, Frank S de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In *International Conference on Computer Aided Verification*, pages 273–289. Springer, 2015.

6 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publish. Co., Redwood City, CA, USA, 1998.

7 Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Computers*, 34(4):318–325, 1985. `doi:10.1109/TC.1985.5009382`.

8 J. Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In Hannah Bast Yossi Azar and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 63:1–63:15, 2018.

9 Tim Peters. Timsort description, accessed june 2015. URL: `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`.

10 Tadao Takaoka. Partial solution and entropy. In Rastislav Královič and Damian Niwiński, editors, *Mathematical Foundations of Computer Science 2009*, pages 700–711, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.