On the Worst-Case Complexity of TimSort

Nicolas Auger, Vincent Jugé, Cyril Nicaud, Carine Pivoteau

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...

1 Introduction

TimSort is a sorting algorithm designed in 2002 by Tim Peters [8], 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.

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

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.

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 [9] (see also [2] 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.

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 improve 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.

```
1 runs ← a run decomposition of S
2 R ← an empty stack
3 while runs ≠ ∅ do                                   // main loop of TimSort
4 |   remove a run r from runs and push r onto R
5 |   merge_collapse(R)
6 if height(R) ≠ 1 then                  // the height of R is its number of runs
7 |   merge_force_collapse(R)
```

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 size 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.

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 $\mathcal{R}[1]\dots\mathcal{R}[\texttt{height}(\mathcal{R})]$, 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).

```
1 while height(R) > 1 do
2 |   n ← height(R) − 2
3 |   if (n > 0 and r3 ⩽ r2 + r1) or (n > 1 and r4 ⩽ r3 + r2) then
4 |   |   if r2 < r1 then
5 |   |   |   merge runs R[2] and R[3] on the stack
6 |   |   else merge runs R[1] and R[2] on the stack
7 |   else if r2 ⩽ r1 then
8 |   |   merge runs R[1] and R[2] on the stack
9 |   else break
```

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 dichotomy called "gallop" to optimize the merges. However, this does not interfere with our analysis and we will not discuss this matter any further.

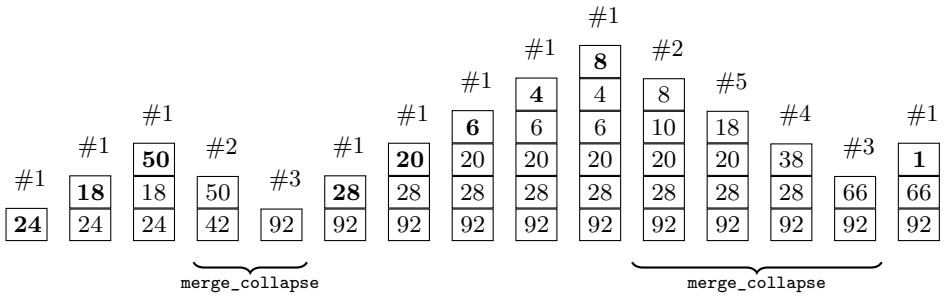

Figure 2 The successive states of the stack $\mathcal{R}$ (the values are the lengths of the runs) during an execution of the main loop of TimSort (Algorithm 1), with the lengths of the runs in `runs` being $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. The label #1 indicates that a run has just been pushed onto 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 [8], 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 size 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 [8], 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

[3] Actually, in [8], the invariant is only stated for the 3 topmost runs of the stack.

result was refined later in [4], where the authors provide lower and upper bounds, including explicit multiplicative constants, for different merge sort algorithms.

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).

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 size of this run is denoted by r_i.

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

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 the same merges, i.e. merges involving the same runs, take place in the same order in both algorithms. ◀

▶ **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 lengths $8n$, $4n$, $2n$ and n (in this order) at the beginning of the decomposition would ensure that $h \geqslant 4$ during the whole loop.

We can now proceed with the core of our proof, which is the analysis of the main loop. We used the framework of *amortized complexity* of algorithms: we 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 ♢-tokens and the ♡-tokens, which can both be used to pay for comparisons.

Two ♢-tokens and one ♡-token are credited to an element when it enters the stack 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 ♢. 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 $r_3 \leqslant r_1$ in this case.
- Case #3: every element of R_1 pays 2 ♢. 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 ♢ and every element of R_2 pays 1 ♡. The cost $r_1 + r_2$ is exactly the number of tokens spent.

▶ **Lemma 4.** *The balances of ♢-tokens and ♡-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 ♢-tokens after the merge is at least the number of ♢-tokens spent. The ♡-tokens are spent in Cases #4 and #5 only: every element of R_2 pays one ♡-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 ♡-token without loosing any, since the topmost run of the stack never pays with ♡-tokens. This proves that, whenever an element pay one ♡-token, the next modification is another merge during which it regains its ♡-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 $2h_{\max}$ ♢-tokens and at most $h_{\max}$ ♡-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 size 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 $\mathcal{O}(nh_{\max})$ comparisons, by bounding from above the total number of tokens credited, and that $h_{\max} = \mathcal{O}(\log n)$, by showing that the run sizes 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 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* $\mathcal{O}(n + n \log \rho)$*.*

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

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. 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).

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

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

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 total number of comparisons performed during 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 size 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 consists in 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 $r \geqslant r_k$, hence applying Lemma 6 to the stack $\overline{\mathcal{S}}$ shows that $r \geqslant \frac{1}{3}\sqrt{2}^{k-i} r_i$ for all $i = 1, \ldots, k$. It follows that

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

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.

▶ **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 $r_1 < r_3 < 3r_3$. Before any merge during an ending sequence, if the stack is $\mathcal{S} = (R_1, \dots R_h)$, then we denote by $\overline{\mathcal{S}} = (\overline{R}_1, \dots, \overline{R}_{h-1})$ 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 $\overline{r}_1 = r_1 + r_2 < r_3 + r_3 + r_4 < 3r_4 = 3\overline{r}_3$ in Cases #3 to #5 (because $r_1 < r_3$), 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

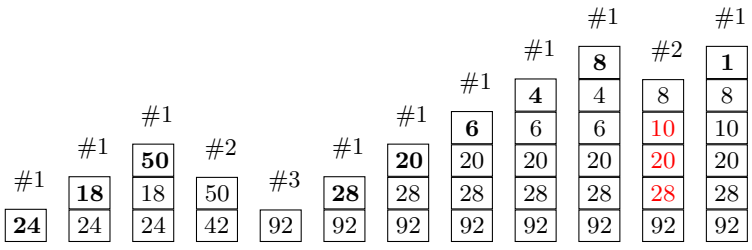

Figure 4 Execution of the main loop of Java's TimSort (Algorithm 3, without merge case #5, at line 9), with the lengths of the runs in `runs` being $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. When the second to last run (of length 8) is pushed onto the stack, the while loop of line 5 stops after only one merge, breaking the invariant (in red), which was not the case with the same example using the Python version of TimSort (see Figure 2).

of `merge_collapse` (and therefore merge case #5 of Algorithm 3) was missing. This version 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.

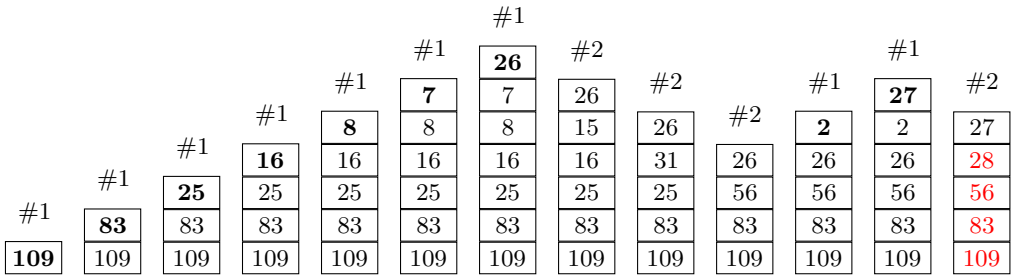

Figure 5 Execution of the main loop of the Java version of TimSort (without merge case #5, at line 9 of Algorithm 3), with the lengths of the runs in `runs` being $(109, 83, 25, 16, 8, 7, 26, 2, 27)$. When the algorithm stops, the invariant is violated twice, for consecutive runs (in red).

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. And, it turns out that the missing invariant was a key ingredient for having a simple an 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* $(2+\sqrt{7})r_i \geqslant r_2 + \ldots + r_{i-1}$.

Full proof in Section A.1.1.

[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`.

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, on the stack size and on the run sizes. 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 *may 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] 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.

Full proof in Sections A.1.2 and A.1.3.

▶ **Theorem 15.** *On an input of size n, Java's* TimSort *may 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 $i+j+1$ is an obstruction index, and 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 sizes 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.

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

References

1 N. Auger, C. Nicaud, and C. Pivoteau. Merge strategies: From Merge Sort to TimSort. Research Report hal-01212839, hal, 2015.

2 J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013.

3 B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

4 S. Buss and A. Knop. Strategies for stable merge sorting. Research Report abs/1801.04641, arXiv, 2018.

5 S. De Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. 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 D. 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 H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Computers*, 34(4):318–325, 1985.

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

9 T. Takaoka. Partial solution and entropy. In R. Královič and D. Niwiński, editors, *Mathematical Foundations of Computer Science 2009*, pages 700–711, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

A Appendix

A.1 Proofs

We provide below complete proofs of the results mentioned in Section 5.

In what follows, we will often refer to so-called *stable* stacks: we say that a stack $\mathcal{S} = (R_1, \ldots, R_h)$ is *stable* if $r_1 + r_2 < r_3$ and $r_1 < r_2$, i.e. if the next operation that will be performed by TimSort is a push operation (Case #1).

A.1.1 Proving Proposition 11

Aiming to prove Proposition 11, and keeping in mind that studying stable stacks may be easier than studying all stacks, a first step is to introduce the following quantities.

▶ **Definition 16.** Let n be a positive integer. We denote by α_n (resp., β_n), the smallest real number m such that, in every stack (resp., stable stack) $\mathcal{S} = (R_1, \ldots, R_h)$ obtained during an execution of TimSort, and for every integer $i \in \{1, \ldots, h\}$ such that $r_i = n$, we have $r_2 + \ldots + r_{i-1} \leqslant m \times r_i$; if no such real number exists, we simply set $\alpha_n = +\infty$ (resp., $\beta_n = +\infty$).

By construction, $\alpha_n \geqslant \beta_n$ for all $n \geqslant 1$. The following lemma proves that $\alpha_n \leqslant \beta_n$.

▶ **Lemma 17.** *At any time during the main loop of* TimSort*, if the stack is* $(R_1, \ldots, R_h)$*, then we have (a)* $r_i < r_{i+1}$ *for all* $i \in \{3, 4, \ldots, h-1\}$ *and (b)* $r_2 + \ldots + r_{i-1} \leqslant \beta_n r_i$ *for all* $n \geqslant 1$ *and* $i \leqslant h$ *such that* $r_i = n$*.*

Proof. Assume that (a) and (b) do not always hold, and consider the first moment where some of them do not hold. When the main loop starts, both (a) and (b) are true. Hence, from a stack $\mathcal{S} = (R_1, \ldots, R_h)$, on which (a) and (b) hold, we carried either a push step (Case #1) or a merging step (Cases #2 to #4), thereby obtaining the new stack $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$. We consider separately these two cases:

- After a push step, we have $\overline{h} = h + 1$, $r_1 + r_2 < r_3$ (otherwise, we would have performed a merging step instead of a push step) and $\overline{r}_i = r_{i-1}$ for all $i \geqslant 2$. It follows that $\overline{r}_3 = r_2 < r_1 + r_2 < r_3 = \overline{r}_4$, and that $\overline{r}_i = r_{i-1} < r_i = \overline{r}_{i+1}$ for all $i \geqslant 4$. This proves that $\overline{\mathcal{S}}$ satisfies (a).
  In addition, the value of $\overline{r}_1$ has no impact on whether $\overline{\mathcal{S}}$ satisfies (b). Hence, we may assume without loss of generality that $\overline{r}_1 < \min\{\overline{r}_2, \overline{r}_3 - \overline{r}_2\}$ (up to doubling the size of every run ever pushed onto the stack so far and setting $\overline{r}_1 = 1$), thereby making $\overline{\mathcal{S}}$ stable. This proves that $\overline{\mathcal{S}}$ satisfies (b).
- After a merging step, we have $\overline{h} = h - 1$, $\overline{r}_2 \leqslant r_2 + r_3$ and $\overline{r}_i = r_{i+1}$ for all $i \geqslant 3$. Hence, $\overline{r}_i = r_{i+1} < r_{i+2} = \overline{r}_{i+1}$ for all $i \geqslant 3$, and $\overline{\mathcal{S}}$ satisfies (a). Furthermore, we have $0 \leqslant \beta_{\overline{r}_2}\overline{r}_2$, and $\overline{r}_2 + \overline{r}_3 + \ldots + \overline{r}_i \leqslant r_2 + r_3 + \ldots + r_{i+1} \leqslant \beta_n r_{i+2} = \beta_n \overline{r}_{i+1}$ whenever $i \geqslant 1$ and $\overline{r}_{i+1} = r_{i+2} = n$. This proves that $\overline{\mathcal{S}}$ also satisfies (b).

Hence, in both cases, (a) and (b) also hold in $\overline{\mathcal{S}}$, which contradicts our assumption and completes the proof. ◀

▶ **Corollary 18.** *For all integers* $n \geqslant 1$*, we have* $\alpha_n = \beta_n$*.*

It remains to prove that $\alpha_n \leqslant \alpha_\infty$ for all $n \geqslant 1$, where $\alpha_\infty = 2 + \sqrt{7}$. This is the object of the next results.

What makes Java's TimSort much harder to study than Python's TimSort is the fact that, during the execution of Java's TimSort algorithm, we may have stacks $\mathcal{S} = (R_1, \ldots, R_h)$ on which the invariant (1) : $r_i > r_{i-1} + r_{i-2}$ fails for many integers $i \geqslant 3$, possibly consecutive. In Section 5, such integers were called *obstruction indices* of the stack $\mathcal{S}$. Hence, we focus on sequences of consecutive obstruction indices.

▶ **Lemma 19.** *Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a stable stack obtained during the main loop of Java's* TimSort*. Assume that $i-k, i+1-k, \ldots, i$ are consecutive obstruction indices of $\mathcal{S}$, and that $\alpha_n \leqslant \alpha_\infty$ for all $n \leqslant r_i - 1$. Then,*

$$r_{i-k-2} \leqslant \frac{\alpha_\infty + 1 - k}{\alpha_\infty + 2} r_{i-1}.$$

Proof. Let T be the number of merge or push operations performed between the start of the main loop and the creation of the stack $\mathcal{S}$. For all $k \in \{0, \ldots, T\}$ and all $j \geqslant 1$, we denote by $\mathcal{S}_k$ the stack after k operations have been performed. We also denote by $P_{j,k}$ the j^{th} bottom-most run of the stack S_k, and by $p_{j,k}$ the size of $P_{j,k}$; we set $P_{j,k} = \emptyset$ and $p_{j,k} = 0$ if S_k has fewer than j runs. Finally, for all $j \leqslant h$, we set $t_j = \min\{k \geqslant 0 \mid \forall \ell \in \{k, \ldots, T\}, p_{j,\ell} = p_{j,T}\}$.

First, observe that $t_j < t_{j+2}$ for all $j \leqslant h-2$, because a run can be pushed or merged only in top or 2^{nd}-to-top position. Second, if $t_j \geqslant t_{j+1}$ for some $j \leqslant h-1$, then the runs P_{j,t_j}, P_{j+1,t_j} are the two top runs of $\mathcal{S}_{t_j}$. Since none of the runs $P_1, \ldots, P_{j+1}$ is modified afterwards, it follows, if $j \geqslant 2$, that $p_{j+1} + p_j = p_{j+1,t_j} + p_{j,t_j} < p_{j-1,t_j} = p_{j-1}$, and therefore that $h + 2 - j$ is not an obstruction index.

Conversely, let $m_0 = h + 3 - i$. We just proved that $t_{m_0-2} < t_{m_0}$ and also that $t_{m_0-1} < t_{m_0} < \ldots < t_{m_0+k}$. Besides, for all $m \in \{m_0, \ldots, m_0 + k\}$, we prove that the $t_m{}^{\text{th}}$ operation was a merge operation of type #2. Indeed, if not, then the run P_{m,t_m} would be the topmost run of $\mathcal{S}_{t_m}$; since the runs P_{m-1} and P_{m-2} were not modified after that, we would have $p_m + p_{m-1} < p_{m-2}$, contradicting the fact that $h + 3 - m$ is an obstruction index. In particular, it follows that $p_{m+1,t_m} = p_{m+2,t_m-1} \geqslant p_{m,t_m-1}$ and that $p_m = p_{m,t_m} \leqslant p_{m-1,t_m} - p_{m+1,t_m} = p_{m-1} - p_{m+1,t_m}$.

Moreover, for $m = m_0$, observe that $p_m = p_{m,t_m} = p_{m,t_m-1} + p_{m+1,t_m-1}$. Applying Lemma 17 on the stacks $\mathcal{S}_T$ and $\mathcal{S}_{t_m-1}$, we know that $p_{m,t_m-1} \leqslant p_m \leqslant p_{m-2} - 1 = r_i - 1$ and that $p_{p+1,t_m-1} \leqslant a_{p_{m,t_m-1}} p_{m,t_m-1} \leqslant \alpha_\infty p_{m,t_m-1}$, which proves that $p_m \leqslant (\alpha_\infty + 1) p_{m,t_m-1} \leqslant (\alpha_\infty + 1) p_{m+1,t_m}$, i.e., $p_{m_0} \leqslant (\alpha_\infty + 1) p_{m_0+1,t_{m_0}}$. Henceforth, we set $\kappa = p_{m_0+1,t_{m_0}}$.

In addition, for all $m \in \{m_0 + 1, \ldots, m_0 + k\}$, observe that the sequence $(p_{m+1,k})_{t_m \leqslant k \leqslant T}$ is non-decreasing. Indeed, when $t_m \leqslant k$, and therefore $t_i \leqslant k$ for all $i \leqslant m$, the run $p_{m+1,k}$ can only be modified by being merged with another run, thereby increasing in size. This proves that $p_{m+2,t_{m+1}} \geqslant p_{m+1,t_{m+1}-1} \geqslant p_{m+1,t_m}$. Hence, an immediate induction shows that $p_{m+1,t_m} \geqslant p_{m_0+1,t_{m_0}} = \kappa$ for all $m \in \{m_0, \ldots, m_0 + k\}$, and it follows that $p_m \leqslant p_{m-1} - \kappa$.

Overall, this implies that $r_{i-k-2} = p_{m_0+k} \leqslant p_{m_0} - k\kappa$. Note that $p_{m_0} \leqslant \min\{(\alpha_\infty + 1)\kappa, p_{m_0-1} - p_{m_0+1,t_{m_0}}\} = \min\{(\alpha_\infty + 1)\kappa, p_{m_0-1} - \kappa\}$. It follows that

$$r_{i-k-2} \leqslant \min\{(\alpha_\infty + 1)\kappa, p_{m_0-1} - \kappa\} - k\kappa \leqslant \min\{(\alpha_\infty + 1 - k)\kappa, r_{i-1} - (k+1)\kappa\},$$

whence $(\alpha_\infty + 2) r_{i-k-2} \leqslant (k+1)(\alpha_\infty + 1 - k)\kappa + (\alpha_\infty + 1 - k)(r_{i-1} - (k+1)\kappa) = (\alpha_\infty + 1 - k) r_{i-1}$. ◀

Lemma 19 paves the way towards a proof by induction that $\alpha_n \leqslant \alpha_\infty$. Indeed, a first, immediate consequence of Lemma 19, is that, provided that $\alpha_n \leqslant \alpha_\infty$ for all $n \leqslant r_i - 1$, then the top-most part $(R_1, \ldots, R_i)$ may not contain more than $\alpha_\infty + 2$ (and therefore no more

than 6) consecutive obstruction indices. This suggests that the sequence $r_1, \ldots, r_i$ should grow "fast enough", which might then be used to prove that $\alpha_{r_i} \leqslant \alpha_\infty$. We present below this inductive proof, which relies on the following objects.

▶ **Definition 20.** We call *expansion function* the function $f : [0,1] \to \mathbb{R}_{\geqslant 0}$ defined by

$$f : x \to \begin{cases} (1+\alpha_\infty)x & \text{if } 0 \leqslant x \leqslant 1/2 \\ x + \alpha_\infty(1-x) & \text{if } 1/2 \leqslant x \leqslant \alpha_\infty/(2\alpha_\infty - 1) \\ \alpha_\infty x & \text{if } \alpha_\infty/(2\alpha_\infty-1) \leqslant x \leqslant 1. \end{cases}$$

In the following, we denote by θ the real number $\alpha_\infty/(2\alpha_\infty - 1)$. Let us first prove two technical results about the expansion function.

▶ **Lemma 21.** *We have $\alpha_\infty x \leqslant f(x)$ for all $x \in [0,1]$, $f(x) \leqslant f(1/2)$ for all $x \in [0,\theta]$, $f(x) \leqslant f(1)$ for all $x \in [0,1]$, $x + \alpha_\infty(1-x) \leqslant f(x)$ for all $x \in [1/2,1]$ and $x + \alpha_\infty(1-x) \leqslant f(1/2)$ for all $x \in [1/2,1]$.*

Proof. Since f is piecewise linear, it is enough to check the above inequalities when x is equal to 0, $1/2$, θ or 1, which is immediate. ◀

▶ **Lemma 22.** *For all real numbers $x, y \in [0,1]$ such that $x(y+1) \leqslant 1$, we have $x(1+f(y)) \leqslant \min\{f(1/2), f(x)\}$.*

Proof. We treat three cases separately, relying in each case on Lemma 21:

- if $0 \leqslant x \leqslant 1/2$, then $x(1+f(y)) \leqslant x(1+f(1)) = (1+\alpha_\infty)x = f(x) \leqslant f(1/2)$;
- if $1/2 < x \leqslant 1$ and $f(1/2) < f(y)$, then $\theta \leqslant y \leqslant 1$, hence $x(1+f(y)) = x + \alpha_\infty xy \leqslant x + \alpha_\infty(1-x) \leqslant \min\{f(x), f(1/2)\}$;
- if $0 \leqslant f(y) \leqslant f(1/2)$, and since $\alpha_\infty \geqslant 1$, we have $x(1+f(y)) \leqslant x(1+f(1/2)) = x(3+\alpha_\infty)/2 \leqslant x(1+\alpha_\infty) \leqslant f(x)$; if, furthermore, $y \leqslant 1/2$, then

$$\begin{aligned} x(1+f(y)) &\leqslant (1+(1+\alpha_\infty)y)/(1+y) = (1+\alpha_\infty) - \alpha_\infty/(1+y) \\ &\leqslant (1+\alpha_\infty) - 2\alpha_\infty/3 = (3+\alpha_\infty)/3, \end{aligned}$$

  and if $1/2 \leqslant y$, then $x(1+f(y)) \leqslant (1+f(1/2))/(1+y) \leqslant 2(1+f(1/2))/3 = (3+\alpha_\infty)/3$; since $\alpha_\infty \geqslant 3$, it follows that $x(1+f(y)) \leqslant (3+\alpha_\infty)/3 \leqslant (1+\alpha_\infty)/2 = f(1/2)$ in both cases.

◀

Using Lemma 19 and the above results about the expansion function, we finally get the following result, of which Proposition 11 is an immediate consequence.

▶ **Lemma 23.** *Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a stable stack obtained during the main loop of Java's* TimSort. *For all integers $i \geqslant 2$, we have $r_1 + r_2 + \ldots + r_{i-1} \leqslant r_i f(r_{i-1}/r_i)$, where f is the expansion function. In particular, we have $\alpha_n = \beta_n \leqslant \alpha_\infty$ for all integers $n \geqslant 1$.*

Proof. Lemma 21 proves that $2x \leqslant \alpha_\infty x \leqslant y f(x/y)$ whenever $0 < x \leqslant y$, and therefore the statement of Lemma 23 is immediate when $i \leqslant 3$. Hence, we prove Lemma 23 for $i \geqslant 4$, and proceed by induction on $r_i = n$, thereby assuming that α_{n-1} exists.

Let $x = r_{i-1}/r_i$ and $y = r_{i-2}/r_{i-1}$. By Lemma 17, and since the stack $\mathcal{S}$ is stable, we know that $r_{i-2} < r_{i-1} < r_i$, and therefore that $x < 1$ and $y < 1$. If i is not an obstruction index, then we have $r_{i-2} + r_{i-1} \leqslant r_i$, i.e., $x(1+y) \leqslant 1$ and, using Lemma 22, it follows that $r_1 + \ldots + r_{i-1} = (r_1 + \ldots + r_{i-2}) + r_{i-1} \leqslant f(y) r_{i-1} + r_{i-1} = x(1+f(y))r_i \leqslant f(x) r_i$.

On the contrary, if i is an obstruction index, let k be the smallest positive integer such that $i-k$ is not an obstruction index. Since the stack $\mathcal{S}$ is stable, we have $r_1 + r_2 < r_3$, which means that 3 is not an obstruction index, and therefore $i-k \geqslant 3$. Let $u = r_{i-k-1}/r_{i-k}$ and $v = r_{i-k-2}/r_{i-k-1}$. By construction, we have $r_{i-k-2} + r_{i-k-1} \leqslant r_{i-k}$, i.e., $u(1+v) \leqslant 1$. Using Lemma 19, and since $r_{i-k-1} < r_i$ and $\alpha_{r_i - 1} \leqslant f(1) = \alpha_\infty$ by induction hypothesis, we have

$$\begin{aligned} r_1 + \ldots + r_{i-1} &= (r_1 + \ldots + r_{i-k-2}) + r_{i-k-1} + \ldots + r_{i-1} \leqslant r_{i-k-1} f(v) + r_{i-k-1} + \ldots + r_{i-1} \\ &\leqslant r_{i-k} u(1 + f(v)) + r_{i-k} + \ldots + r_{i-1} \leqslant r_{i-k} f(1/2) + r_{i-k} + \ldots + r_{i-1} \\ &\leqslant \frac{1}{\alpha_\infty + 2} \left((\alpha_\infty + 3 - k) f(1/2) + \sum_{j=1}^{k} (\alpha_\infty + 3 - j) \right) r_{i-1} \\ &\leqslant \frac{1}{2(\alpha_\infty + 2)} \left(\alpha_\infty^2 + (4+k)\alpha_\infty - k^2 + 4k + 3 \right) r_{i-1}. \end{aligned}$$

The function $g : t \to \alpha_\infty^2 + (4+t)\alpha_\infty - t^2 + 4t + 3$ takes its maximal value, on the real line, at $t = (\alpha_\infty + 4)/2 \in (4, 5)$. Consequently, for all integers k, and since $\alpha_\infty \leqslant 5$, we have

$$g(k) \leqslant \max\{g(4), g(5)\} = \alpha_\infty^2 + \max\{8\alpha_\infty + 3, 9\alpha_\infty - 2\} = \alpha_\infty^2 + 8\alpha_\infty + 3.$$

Then, observe that $2(\alpha_\infty + 2)\alpha_\infty = 30 + 12\sqrt{7} = \alpha_\infty^2 + 8\alpha_\infty + 3$. It follows that

$$r_1 + \ldots + r_{i-1} \leqslant \frac{\alpha_\infty^2 + 8\alpha_\infty + 3}{2(\alpha_\infty + 2)} r_{i-1} = \alpha_\infty x r_i \leqslant f(x) r_i.$$

Hence, regardless of whether i is an obstruction index or not, we have $r_1 + \ldots + r_{i-1} \leqslant f(x) r_i \leqslant f(1) r_i = \alpha_\infty r_i$, which completes the proof. ◀

A.1.2 Proving the first part of Theorem 15

We prove below the inequality of Theorem 15; proving that that the constant Δ used in Theorem 15 is optimal will be the done in the next section.

In order to carry out this proof, we need to consider some integers of considerable interest. Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a *stable* stack of runs. We say that an integer $i \geqslant 1$ is a *growth index* if $i+2$ is *not* an obstruction index, and that i is a *strong growth index* if i is a growth index and if, in addition, $i+1$ is an obstruction index. Note that h an $h-1$ are necessarily growth indices, since $h+1$ and $h+2$ are too large to be obstruction indices.

Our aim is now to prove inequalities of the form $r_{i+j} \geqslant \Delta^j r_i$, where $3 \leqslant i \leqslant i+j \leqslant h$. However, such inequalities do not hold in general, hence we restrict the scope of the integers i and $i+j$, which is the subject of the two following results.

▶ **Lemma 24.** *Let i and j be positive integers such that $i+2 \leqslant j \leqslant h$. If no obstruction index k exists such that $i+2 \leqslant k \leqslant j$, then $2\Delta^{j-i-2} r_i \leqslant r_j$.*

Proof. For all $n \geqslant 0$, let F_n denote the n^{th} Fibonacci number, defined by $F_0 = 0$, $F_1 = 1$ and $F_{n+2} = F_n + F_{n+1}$ or, alternatively, by $F_n = (\phi^n - (-\phi)^{-n})/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$ is the Golden ratio. Observe now that

$$F_{j-i+1} r_i \leqslant F_{j-i-1} r_i + F_{j-i} r_{i+1} \leqslant F_{j-i-2} r_{i+1} + F_{j-i-1} r_{i+2} \leqslant \ldots \leqslant F_0 r_{j-1} + F_1 r_j = r_j.$$

Moreover, for all $n \geqslant 3$, we have $F_n = 2F_n/F_3 = 2\phi^{n-3}(1 - (-1)^n \phi^{-2n})/(1 - \phi^{-6}) \geqslant 2\phi^{n-3}$. Since $\Delta < \phi$, it follows that $2\Delta^{j-i-2} r_i \leqslant F_{j-i+1} r_i \leqslant r_j$. ◀

▶ **Lemma 25.** *Let i and j be positive integers such that $1 \leqslant i \leqslant j \leqslant h$. If i is a growth index and j is a strong growth index, then $\Delta^{j-i} r_i \leqslant r_j$.*

Proof. Without loss of generality, let us assume that $i < j$ and that there is no strong growth index k such that $i < k < j$. Indeed, if such an index k exists, then a simple induction completes the proof of Lemma 25.

Let ℓ be the largest integer such that $\ell \leqslant j$ and ℓ is not an obstruction index. Lemmas 19 and 23 prove that $(\alpha_\infty+2)r_\ell \leqslant (\alpha_\infty+2+\ell-j)r_j$ and that $(\alpha_\infty+2)r_{\ell-1} \leqslant (\alpha_\infty+1+\ell-j)r_j$. The latter inequality proves that $j - \ell \leqslant \lfloor \alpha_\infty + 1 \rfloor = 5$.

By construction, we have $i + 2 \leqslant \ell$, and no integer k such that $i + 2 \leqslant k \leqslant \ell$ is an obstruction index. Hence, Lemma 24 proves that $2(\alpha_\infty + 2)\Delta^{\ell-i-2} r_i \leqslant (\alpha_\infty + 2)r_\ell \leqslant (\alpha_\infty + 2 + \ell - j)r_j$. Moreover, simple numerical computations, for $j - \ell \in \{0, \ldots, 5\}$, prove that $\Delta^{j-\ell+2} \leqslant 2(\alpha_\infty + 2)/(\alpha_\infty + 2 + \ell - j)$, with equality when $j - \ell = 3$. It follows that $\Delta^{j-i} r_i = \Delta^{j-\ell+2}\Delta^{\ell-i-2} r_i \leqslant r_j$. ◀

Finally, the inequality of Theorem 15 is an immediate consequence of the following result.

▶ **Lemma 26.** *Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a stack obtained during the main loop of Java's* TimSort. *We have $r_h \geqslant \Delta^{h-3}$.*

Proof. Let us first assume that $\mathcal{S}$ is stable. Then, $r_1 \geqslant 1$, and 1 is a growth index. If there is no obstruction index, then Lemma 24 proves that $r_h \geqslant 2\Delta^{h-3} r_1 \geqslant \Delta^{h-2}$.

Otherwise, let ℓ be the largest obstruction index. Then, $\ell - 1$ is a strong growth index, and Lemma 25 proves that $r_{\ell-1} \geqslant \Delta^{\ell-2} r_1 \geqslant \Delta^{\ell-2}$. If $\ell = h$, then $r_h \geqslant r_{\ell-1} \geqslant \Delta^{h-2}$, and if $\ell \leqslant h - 1$, then Lemma 24 also proves that $r_h \geqslant 2\Delta^{h-\ell-1} r_{\ell-1} \geqslant \Delta^{h-\ell}\Delta^{\ell-2} = \Delta^{h-2}$.

Finally, if $\mathcal{S}$ is not stable, the result is immediate for $h \leqslant 3$, hence we assume that $h \geqslant 4$. The stack $\mathcal{S}$ was obtained by pushing a run onto a stable stack $\mathcal{S}'$ of size at least $h - 1$, then merging runs from $\mathcal{S}'$ into the runs R_1 and R_2. It follows that R_h was already the largest run of S', and therefore that $R_h \geqslant \Delta^{h-3}$. ◀

A.1.3 Proving the second part of Theorem 15

We finally focus on proving that the constant Δ of Theorem 15 is optimal. The most important step towards this result consists in proving that $\alpha_\infty = \lim_{n\to\infty} \alpha_n$, with the real numbers α_n introduced in Definition 16 and $\alpha_\infty = 2 + \sqrt{7}$. Since it is already proved, in Lemma 23, that $\alpha_n \leqslant \alpha_\infty$ for all $n \geqslant 1$, it remains to prove that $\alpha_\infty \leqslant \liminf_{n\to\infty} \alpha_n$. We obtain this inequality by constructing explicitly, for k large enough, a stable sequence of runs $(R_1, \ldots, R_h)$ such that $r_h = k$ and $r_2 + \ldots + r_{h-1} \approx \alpha_\infty k$. Hence, we focus on constructing sequences of runs.

In addition, let us consider the stacks of runs created by the main loop of Java's TimSort on a sequence of runs $P_1, \ldots, P_n$. We say below that the sequence $P_1, \ldots, P_k$ *produces* a stack of runs $\mathcal{S} = (R_1, \ldots, R_h)$ if the stack $\mathcal{S}$ is obtained after each of the runs $P_1, \ldots, P_n$ has been pushed; observe that the sequence $P_1, \ldots, P_n$ produces exactly one stable stack. We also say that a stack of runs is *producible* if it is produced by some sequence of runs.

Finally, in what follows, we are only concerned with run sizes. Hence, we abusively identify runs with their sizes. For instance, in Figure 5, the sequence $(109, 83, 25, 16, 8, 7, 26, 2, 27)$ produces the stacks $(27, 2, 26, 56, 83, 109)$ and $(27, 28, 56, 83, 109)$; only the latter stack is stable.

We review now several results that will provide us with convenient and powerful ways of constructing producible stacks.

▶ **Lemma 27.** *Let $\mathcal{S} = (r_1, \dots, r_h)$ be a stable stack produced by a sequence of runs $p_1, \dots, p_n$. Assume that n is minimal among all sequences that produce $\mathcal{S}$. Then, when producing $\mathcal{S}$, no merging step #3 or #4 was performed.*

Moreover, for all $k \leqslant n-1$, after the run p_{k+1} has been pushed onto the stack, the elements coming from p_k will never belong to the topmost run of the stack.

Proof. We begin by proving the first statement of Lemma 27 by induction on n, which is immediate for $n = 1$. Hence, we assume that $n \geqslant 2$, and we further assume, for the sake of contradiction, that some merging step #3 or #4 took place. Let $\mathcal{S}' = (r'_1, \dots, r'_\ell)$ be the stable stack produced by the sequence $p_1, \dots, p_{n-1}$. By construction, this sequence is as short as possible, and therefore no merging step #3 or #4 was used so far. Hence, consider the last merging step #3 or #4, which necessarily appears after p_n was pushed onto the stack. Just after this step has occurred, we have a stack $\mathcal{S}'' = (r''_1, \dots, r''_m)$, with $r'_i = r''_j$ whenever $j \geqslant 2$ and $i + m = j + \ell$, and the run r''_1 was obtained by merging the runs $p_n, r'_1, \dots, r'_{\ell+1-m}$.

Let $p_1, \dots, p_k$ be the runs that had been pushed onto the stack when the run $r''_2 = r'_{m+2-\ell}$ was created. This creation was the result of either a push step or a merging step #2. In both cases, and until $\mathcal{S}'$ is created, no merging step #3 or #4 ever involves any element placed within or below r''_2. Then, in the case of a push step, we have $p_k = r''_2$, and therefore the sequence $\mathcal{P}_{\#1} = (p_1, \dots, p_k, r''_1)$ also produces the stack $\mathcal{S}'$. In the case of a merging step #2, it is the sequence $\mathcal{P}_{\#2} = (p_1, \dots, p_{k-1}, r''')$ that also produces the stack $\mathcal{S}''$, where r''' is obtained by merging the runs $p_k, \dots, p_n$.

In both cases, since the sequences $\mathcal{P}_{\#1}$ and $\mathcal{P}_{\#2}$ produce $\mathcal{S}''$, they also produce $\mathcal{S}$. It follows that $k + 1 \geqslant n$ (in the first case) or $k \geqslant n$ (in the second case). Moreover, the run r''_2 was not modified between pushing the run p_n and before obtaining the stack $\mathcal{S}''$, hence $k \leqslant n - 1$. This proves that $k = n - 1$ and that the run r''_2 was obtained through a push step, i.e. $p_{n-1} = r''_2$. But then, the run r''_1 may contain elements of p_n only, hence is not the result of a merging step #3 or #4: this disproves our assumption and proves the first statement of Lemma 27.

Finally, observe that push steps and merging steps #2 never allow a run in 2^{nd}-to-top position or below to go to the top position. This completes the proof of Lemma 27. ◀

▶ **Lemma 28.** *Let $\mathcal{S} = (r_1, \dots, r_h)$ be a stable stack produced by a sequence of runs $p_1, \dots, p_n$. Assume that n is minimal among all sequences that produce $\mathcal{S}$. There exist integers $i_0, \dots, i_h$ such that $0 = i_h < i_{h-1} < \dots < i_0 = n$ and such that, for every integer $k \leqslant h$, (a) the runs $p_{i_k+1}, \dots, p_{i_{k-1}}$ were merged into the run r_k, and (b) $i_{k-1} = i_k + 1$ if and only if $k + 2$ is not an obstruction index.*

Proof. The existence (and uniqueness) of integers $i_0, \dots, i_h$ satisfying (a) is straightforward, hence we focus on proving (b). That property is immediate if $h = 1$, hence we assume that $2 \leqslant h \leqslant n$. Checking that the sequence $r_h, r_{h-1}, p_{i_{h-2}+1}, p_{i_{h-2}+2}, \dots, p_n$ produces the stack $\mathcal{S}$ is immediate, and therefore $i_{h-1} = 1$ and $i_{h-2} = 2$, i.e., $r_h = p_1$ and $r_{h-1} = p_2$.

Consider now some integer $k \leqslant h - 2$, and let $\mathcal{S}'$ be the stable stack produced by $p_1, \dots, p_{i_k+1}$. From that point on, the run $p_{i_{k-1}+1}$ will never be the topmost run, and the runs p_j with $j \leqslant i_{k-1}$, which can never be merged together with the run $p_{i_{k-1}+1}$, will never be modified again. This proves that $\mathcal{S}' = (p_{i_{k-1}+1}, r_{k+1}, \dots, r_h)$.

Then, assume that $i_{k-1} = i_k + 1$, and therefore that $p_{i_{k-1}+1} = r_k$. Since $\mathcal{S}'$ is stable, we know that $r_k + r_{k+1} < r_{k+2}$, which means that $k + 2$ is not an obstruction index. Conversely, if $k+2$ is not an obstruction index, both sequences $p_1, \dots, p_{i_{k+1}+1}$ and $p_1, \dots, p_{i_{k-1}}, r_k, p_{i_{k+1}+1}$ produce the stack $(p_{i_{k+1}+1}, r_k, \dots, r_h)$ and, since n is minimal, $i_{k-1} = i_k + 1$. ◀

▶ **Lemma 29.** *Let $\mathcal{S} = (r_1, \ldots, r_h)$ be a producible stable stack of height $h \geqslant 3$. There exists an integer $\kappa \in \{1, 4\}$ and a producible stable stack $\mathcal{S}' = (r'_1, \ldots, r'_\ell)$ such that $\ell \geqslant 2$, $r_h = r'_\ell$, $r_{h-1} = r'_{\ell-1}$ and $r_1 + \ldots + r_{h-2} = r'_1 + \ldots + r'_{\ell-2} + \kappa$.*

Proof. First, Lemma 29 is immediate if $h = 3$, since the sequence of runs $(r_3, r_2, r_1 - 1)$ produces the stack $(r_1 - 1, r_2, r_3)$. Hence, we assume that $h \geqslant 4$. Let $p_1, \ldots, p_n$ be a sequence of runs, with n minimal, that produces $\mathcal{S}$. We prove Lemma 29 by induction on n.

If the last step carried when producing $\mathcal{S}$ was pushing the run P_n onto the stack, then the sequence $p_1, \ldots, p_{n-1}, p_n - 1$ produces the stack $r_1 - 1, r_2, \ldots, r_h$, and we are done in this case. Hence, assume that the last step carried was a merging step #2.

Let $\mathcal{S}' = (q_1, \ldots, q_m)$ be the stable stack produced by the sequence $p_1, \ldots, p_{n-1}$, and let i be the largest integer such that $q_i < p_n$. After pushing p_n, the runs $q_1, \ldots, q_i$ are merged into one single run r_2, and we also have $p_n = r_1$ and $q_{i+j} = r_{2+j}$ for all $j \geqslant 1$. Incidentally, this proves that $m = h + i - 2$ and, since $h \geqslant 4$, that $i \leqslant m - 2$. We also have $i \geqslant 2$, otherwise, if $i = 1$, we would have had a merging step #3 instead.

If $r_1 = p_n \geqslant q_i + 2$, then the sequence $p_1, \ldots, p_{n-1}, p_n - 1$ also produces the stack $(r_1 - 1, r_2, \ldots, r_h)$, and we are done in this case. Hence, we further assume that $r_1 = p_n = q_i + 1$. Since $q_{i-1} + q_i \leqslant r_2 < r_3 = q_{i+1}$, we know that $i + 1$ is not an obstruction index of S. Let $a \leqslant n - 1$ be a positive integer such that $p_1 + \ldots + p_a = q_i + q_{i+1} + \ldots + q_m$. Lemma 28 states that $p_{a+1} = q_{i-1}$, and therefore that the sequence of runs $p_1, \ldots, p_{a+1}$ produces the stack $(q_{i-1}, \ldots, q_m)$.

If $i = 2$ and if $q_1 \geqslant 3$, then $(q_1 - 1) + q_2 \geqslant q_2 + 2 > r_1$, and therefore the sequence of runs $p_1, \ldots, p_a, q_1 - 1, r_1$ produces the stable stack $(r_1, r_2 - 1, r_3, \ldots, r_h)$. However, if $i = 2$ and $q_1 \leqslant 2$, then the sequence of runs $p_1, \ldots, p_a, r_1 - 2$ produces the stable stack $(r_1 - 2, r_2 - 2, \ldots, r_h)$. Hence, in both cases, we are done, by choosing respectively $\kappa = 1$ and $\kappa = 4$.

Les us now assume that $i \geqslant 3$. Observe that $n \geqslant a + i \geqslant 3$ since, after the stack $(q_{i-1}, \ldots, q_m)$ has been created, it remains to create runs $q_1, \ldots, q_{i-2}$ and finally, r_1, which requires pushing at least $i - 1$ runs in addition to the $a + 1$ runs already pushed. Therefore, we must have $q_1 + \ldots + q_{i-1} \geqslant q_i$, unless what the sequence $p_1, \ldots, p_a, q_1 + \ldots + q_{i-1}, p_n$ would have produced the stack $\mathcal{S}$, despite being of length $a + 2 < n$. In particular, since $q_1 + q_2 < q_3$, it follows that $i \geqslant 4$. Consequently, we have $q_1 \geqslant 1$, $q_2 \geqslant 2$, $q_3 \geqslant 4$, and therefore $q_1 + \ldots + q_{i-1} \geqslant 7$, i.e. $r_2 \geqslant q_2 + 7 = r_1 + 6$.

Finally, by induction hypothesis, there exists a sequence $p'_1, \ldots, p'_u$, with u minimal, that produced the stack $(q'_1, \ldots, q'_v)$ such that $q'_v = q_i$, $q'_{v-1} = q_{i-1}$ and $q_1 + \ldots + q_{i-2} = q'_1 + \ldots + q'_{v-2} + \kappa$ for some $\kappa \in \{1, 4\}$. Lemma 28 also states that $p'_1 = q_i$ and that $p'_2 = q_{i-1}$. It is then easy to check that the sequence of runs $p_1, \ldots, p_{b+1}, p'_3, \ldots, p'_u$ produces the stable stack $(q'_1, \ldots, q'_{v-2}, q_{i-1}, q_i, \ldots, q_m)$. Since $q'_j < q_{i-1} < q_i < r_1 < r_3 = q_{i+1}$ for all $j \leqslant v - 2$, pushing the run $p_n = r_1$ onto that stack and letting merging steps #2 occur then gives the stack $(r_1, r_2 - \kappa, r_3, \ldots, r_h)$, which completes the proof since $r_1 \leqslant r_2 - 6 < r_2 - \kappa$. ◀

In what follows, we will only consider stacks that are producible and stable. Hence, from this point on, we omit mentioning that they systematically must be producible and stable, and we will say that "the stack $\mathcal{S}$ exists" in order to say that "the stack $\mathcal{S}$ is producible and stable".

▶ **Lemma 30.** *Let $\mathcal{S} = (r_1, \ldots, r_h)$ and $\mathcal{S}' = (r'_1, \ldots, r'_\ell)$ be two stacks. Then (a) for all $i \leqslant h$, there exists a stack $(r_1, \ldots, r_i)$, and (b) if $r_{h-1} = r'_1$ and $r_h = r'_2$, then there exists a stack $(r_1, \ldots, r_h, r'_3, \ldots, r'_\ell)$.*

Proof. Let $p_1, \ldots, p_m$ and $p'_1, \ldots, p'_n$ be two sequences that respectively produce $\mathcal{S}$ and $\mathcal{S}'$. Let us further assume that m is minimal. First, consider some integer $i \leqslant h$, and let a be the integer such that $p_1 + \ldots + p_a = r_{i+1} + \ldots + r_h$. It comes at once that the sequence $p_{a+1}, \ldots, p_m$ produces the stack $(r_1, \ldots, r_i)$. Second, since m is minimal, Lemma 28 proves that $p_1 = r_h = r'_2$ and that $p_2 = r_{h-1} = r'_1$ and, once again, the sequence $p'_1, \ldots, p'_n, p_3, \ldots, p_3$ produces the stack $(r_1, \ldots, r_h, r'_3, \ldots, r'_\ell)$, which is also stable. ◀

▶ **Lemma 31.** *For all positive integers k and ℓ such that $k \leqslant \ell\alpha_\ell$, there exists a stack $(r_1, \ldots, r_h)$ such that $r_h = \ell$, $k - 3 \leqslant r_1 + \ldots + r_{h-1} \leqslant k$, and $r_1 + \ldots + r_{h-1} = k$ if $k = \ell\alpha_\ell$.*

Proof. First, if $\ell = 1$, then $\alpha_\ell = 0$, and therefore Lemma 31 is vacuously true. Hence, we assume that $\ell \geqslant 2$. Let Ω be the set of integers k for which some sequence of runs $p_1, \ldots, p_m$ produces a stack $\mathcal{S} = (r_1, \ldots, r_h)$ such that $r_1 + \ldots + r_{h-1} = k$ and $r_h = \ell$. First, if $k \leqslant \ell - 1$, the sequence ℓ, k produces the stack (ℓ, k), thereby proving that $\{1, 2, \ldots, \ell - 1\} \in \Omega$. Second, it follows from Lemma 30 that $\ell\alpha_\ell \in \Omega$.

Finally, consider some integer $k \in \Omega$ such that $k \geqslant \ell$, and let $p_1, \ldots, p_m$ be a sequence of runs that produces a stack $(r_1, \ldots, r_h)$ such that $r_1 + \ldots + r_{h-1} = k$ and $r_h = \ell$. Since $k \geqslant \ell = r_h > r_{h-1}$, we know that $h \geqslant 3$. Hence, due to Lemma 29, either $k - 1$ or $k - 4$ (or both) belongs to Ω. This completes the proof of Lemma 31. ◀

▶ **Lemma 32.** *For all positive integers k and ℓ such that $k \leqslant \ell$, we have $\alpha_\ell \geqslant (1 - k/\ell)\alpha_k$ and $\alpha_\ell \geqslant k\alpha_k/\ell$.*

Proof. Let $n = \lfloor \ell/k \rfloor$. Using Lemma 31, there exists a sequence of runs $p_1, \ldots, p_m$ that produces a stack $(r_1, \ldots, r_h)$ such that $r_h = k$ and $r_1 + \ldots + r_{h-1} = k\alpha_k$. By choosing m minimal, Lemma 28 further proves that $p_1 = r_h = k$. Consequently, the sequence of runs $np_1, \ldots, np_{m-1}, \ell$ produces the stack $(nr_1, \ldots, nr_{h-1}, \ell)$, and therefore we have $\ell\alpha_\ell \geqslant n(r_1 + \ldots + r_{h-1}) = nk\alpha_k \geqslant \max\{1, \ell/k - 1\}k\alpha_k = \ell \max\{k/\ell, 1 - k/\ell\}\alpha_k$. ◀

A first intermediate step towards proving that $\lim_{n\to\infty} \alpha_n = \alpha_\infty$ is the following result, which is a consequence of the above Lemmas.

▶ **Proposition 33.** *Let $\overline{\alpha} = \sup\{\alpha_n \mid n \in \mathbb{N}^*\}$. We have $1 + \alpha_\infty/2 < \overline{\alpha} \leqslant \alpha_\infty$, and $\alpha_n \to \overline{\alpha}$ when $n \to +\infty$.*

Proof. Let $\overline{\alpha} = \sup\{\alpha_n \mid n \in \mathbb{N}^*\}$. Lemma 23 proves that $\overline{\alpha} \leqslant \alpha_\infty$. Then, let ε be a positive real number, and let k be a positive integer such that $\alpha_k \geqslant \overline{\alpha} - \varepsilon$. Lemma 32 proves that $\liminf \alpha_n \geqslant \alpha_k \geqslant \overline{\alpha} - \varepsilon$, and therefore $\liminf \alpha_n \geqslant \overline{\alpha}$. This proves that $\alpha_n \to \overline{\alpha}$ when $n \to +\infty$.

Finally, it is tedious yet easy to verify that the sequence 360, 356, 3, 2, 4, 6, 10, 2, 1, 22, 4, 2, 1, 5, 1, 8, 4, 2, 1, 73, 4, 2, 5, 7, 2, 16, 3, 2, 4, 6, 21, 4, 2, 22, 4, 2, 1, 5, 8, 3, 2, 79, 3, 2, 4, 6, 2, 10, 6, 3, 2, 33, 4, 2, 5, 7, 1, 13, 4, 2, 1, 5, 1, 80, 4, 2, 5, 7, 1, 95, 3, 2, 4, 6, 10, 20, 4, 2, 5, 7, 3, 2, 26, 6, 3, 1, 31, 3, 2, 4, 6, 2, 1, 12, 4, 2, 5 produces the stack (5, 6, 12, 18, 31, 36, 68, 95, 99, 195, 276, 356, 360). Moreover, since $(20\sqrt{7})^2 = 2800 < 2809 = 53^2$, it follows that $80 + 20\sqrt{7} < 133$, i.e., that $1 + \alpha_\infty/2 = 2 + \sqrt{7}/2 < 133/40$. This proves that

$$\overline{\alpha} \geqslant \alpha_{360} \geqslant \frac{5 + 6 + 12 + 18 + 31 + 36 + 68 + 95 + 99 + 195 + 276 + 356}{360} = \frac{133}{40} > 1 + \alpha_\infty/2.$$

◀

▶ **Lemma 34.** *There exists a positive integer N such that, for all integers $n \geqslant N$ and $k = \lfloor (n - 6)/(\overline{\alpha} + 2) \rfloor$, the stack $(k + 1, k+, \alpha_k, n - 4, n)$ exists.*

Proof. Proposition 33 proves that there exists a positive real number $\nu > 1 + \alpha_\infty/2$ and a positive integer $L \geqslant 256$ such that $\alpha_\ell \geqslant \nu$ for all $\ell \geqslant L$. Then, we set $N = \lceil (\overline{\alpha} + 2)L \rceil + 6$. Consider now some integer $n \geqslant N$, and let $k = \lfloor (n-6)/(\overline{\alpha}+2) \rfloor$. By construction, we have $k \geqslant L$, and therefore $\alpha_k \geqslant \nu$.

Let $p_1, \ldots, p_m$ be a sequence of runs that produces a stack $(r_1, \ldots, r_h)$ such that $r_h = k$ and $r_1 + \ldots + r_{h-1} = k\alpha_k$. Lemma 23 proves that $\alpha_k \leqslant f(r_{h-1}/k)$, where f is the expansion function of Definition 20. Since $\alpha_k \geqslant \nu > 1 + \alpha_\infty/2 = f(1/2)$, it follows that $k > r_{h-1} > \theta k$. Assuming that m is minimal, Lemma 28 proves that $p_1 = r_h = k$ and that $p_2 = r_{h-1}$.

Now, let $k' = \lfloor k/2 \rfloor + 1$, and let ℓ be the largest integer such that $2^{\ell+4} \leqslant k'$. Since $k \geqslant L \geqslant 256$, we know that $k' \geqslant 128$, and therefore that $\ell \geqslant 3$. Observe also that, since $\theta = \alpha_\infty/(2\alpha_\infty - 1) > 11/20$ and $k' \geqslant 20$, we have $r_{m-1} > \theta k \geqslant \lfloor k/2 \rfloor + k/20 \geqslant k'$. We build a stack of runs $p_2, k, n-4, n$ by distinguishing several cases, according to the value of $k'/2^\ell$.

- If $16 \times 2^\ell \leqslant k' \leqslant 24 \times 2^\ell + 1$, let x be the smallest integer such that $x \geqslant 2$ and $k' < 2(9 \times 2^\ell + x + 1)$. Since $k' \leqslant 24 \times 2^\ell + 1$, we know that $x \leqslant 3 \times 2^\ell$, and that $x = 2$ if $k \leqslant 18 \times 2^\ell + 1$. Therefore, the sequence of runs $(n, n-4, 3, 2, 4, 6, 10, 3\times 8, 3 \times 16, \ldots, 3 \times 2^\ell, 3 \times 2^\ell + x)$ produces the stack $(3 \times 2^\ell + x, 3 \times 2^{\ell+1} + 1, n-4, n)$. Moreover, observe that $(3 \times 2^{\ell+1} + 1) + (9 \times 2^\ell + x + 1) = 15 \times 2^\ell + 4 < k'$ if $16 \times 2^\ell \leqslant k' \leqslant 18 \times 2^\ell + 1$, and that $(3 \times 2^{\ell+1} + 1) + (9 \times 2^\ell + x + 1) \leqslant 18 \times 2^\ell + 1 < k'$ if $18 \times 2^\ell + 2 \leqslant k'$. Since, in both cases, we also have $k' < 2(9 \times 2^\ell + x + 1)$, it follows that $3 \times 2^{\ell+1} + 1 < k' - (9 \times 2^\ell + x + 1) < 9 \times 2^\ell + x + 1$. Consequently, pushing an additional run of size $k' - (9 \times 2^\ell + x + 1)$ produces the stack $k' - (9 \times 2^\ell + x + 1), 9 \times 2^\ell + x + 1, n - 4, n$. Finally, the inequalities $2k' > k$, $k - k' \geqslant k/2 - 1 \geqslant k' - 2 \geqslant 18 \times 2^\ell$ and $x \leqslant 3 \times 2^\ell$ prove that

  $$9 \times 2^\ell + x + 1 \leqslant 12 \times 2^\ell + 1 < 16 \times 2^\ell - 2 \leqslant k - k' < k'.$$

  Recalling that $k > p_2 > k'$, it follows that pushing additional runs of sizes $k - k'$ and p_2 produces the stack $(p_2, k, n-4, n)$.
- If $24 \times 2^\ell + 2 \leqslant k' < 32 \times 2^\ell$, let x be the smallest integer such that $x \geqslant 2$ and $k' < 2(12 \times 2^\ell + x + 1)$. Since $k' < 32 \times 2^\ell$, we know that $x + 1 \leqslant 2^{\ell+2}$. Therefore, the sequence of runs $n, n-4, 3, 2, 4, 8, 16, 32, \ldots, 2^{\ell+2}, 2^{\ell+2} + x$ produces the stack $(2^{\ell+2} + x, 2^{\ell+3} + 1, n-4, n)$. Moreover, the inequalities

  $$(2^{\ell+3} + 1) + (3 \times 2^{\ell+2} + x + 1) \leqslant 6 \times 2^{\ell+2} + 1 < k' < 2(3 \times 2^{\ell+2} + x + 1)$$

  prove that $2^{\ell+3} + 1 < k' - (3 \times 2^{\ell+2} + x + 1) < 3 \times 2^{\ell+2} + x + 1$.
  Consequently, pushing an additional run of size $k' - (3 \times 2^{\ell+2} + x + 1)$ produces the stack $(k' - (3 \times 2^{\ell+2} + x + 1), 3 \times 2^{\ell+2} + x + 1, n-4, n)$. Finally, the inequalities $2k' > k$, $k - k' \geqslant k/2 - 1 \geqslant k' - 2 \geqslant 6 \times 2^{\ell+2}$ and $x + 1 \leqslant 2^{\ell+2}$ prove that

  $$3 \times 2^{\ell+2} + x + 1 \leqslant 4 \times 2^{\ell+2} < 6 \times 2^{\ell+2} \leqslant k - k' < k'.$$

  Recalling once again that $k > p_2 > k'$, it follows that pushing additional runs of sizes $k - k'$ and p_2 produces the stack $(p_2, k, n-4, n)$ in this case too.

Finally, after having obtained the stack $(p_2, k, n-4, n)$, let us add the sequence of runs $p_3, \ldots, p_m, k+1$. Since $k(1+\alpha_k) + (k+1) \leqslant k(\overline{\alpha}+2) + 1 \leqslant n - 6 + 1 < n - 4$, adding these runs produces the stack $(k+1, k+k\alpha_k, n-4, n)$, which completes the proof. ◀

▶ **Lemma 35.** *For all integers $n \geqslant N$ and $k = \lfloor (n-6)/(\overline{\alpha}+2) \rfloor$ there exists a stack $(k+3, k(\alpha_k - 1) - 7 - x - y, k\alpha_k - 3 - x, k(1+\alpha_k), n-4, n)$ for some integers $x, y \in \{0, 1, 2, 3\}$.*

Proof. Consider some integer $n \geqslant N$, and let $k = \lfloor (n-6)/(\alpha_\infty + 2) \rfloor$. By Lemma 34, there exists a stack $(k+1, k+k\alpha_k, n-4, n)$.

Lemma 32 proves then that $k\alpha_k \geqslant (k+1)\alpha_{k+1}$. Therefore, Lemma 31 proves that there exists a stack $(r_1, \ldots, r_h)$ such that $r_h = k+1$ and $r_1 + \ldots + r_{h-1} = k(\alpha_k - 1) - 4 - x$ for some integer $x \in \{0,1,2,3\}$. By construction, we have $r_1 < r_2 < \ldots < r_h$, hence $r_{h-1} + r_h < 2(k+1) < k(1+\alpha_k)$. Consequently, there also exists a stack $(r_1, r_2, \ldots, r_{h-1}, k+1, k(1+\alpha_k), n-4, n)$. Then, $k+2+r_1+\ldots+r_h \leqslant k+2+k(\alpha_k-1)-4+k+1 = k(1+\alpha_k)-1 < k(1+\alpha_k)$, and therefore pushing an additional run of size $k+2$ produces a stack $(k+2, k\alpha_k - 3 - x, k(1+\alpha_k), n-4, n)$.

Once again, there exists a stack $(r'_1, \ldots, r'_{h'})$ such that $r'_{h'} = k+2$ and $r'_1 + \ldots + r'_{h'-1} = k(\alpha_k - 2) - 9 - x - y$ for some integer $y \in \{0,1,2,3\}$. By construction, we have $r'_1 < r'_2 < \ldots < r'_{h'}$, hence $r'_{h'-1} + r'_{h'} < 2(k+2) < k\alpha_k - 3 - x$, and therefore there exists a stack $(r'_1, r'_2, \ldots, r'_{h'-1}, k+2, k\alpha_k - 3 - x, k(1+\alpha_k), n-4, n)$. Then, $k+3+r'_1+\ldots+r'_{h'} \leqslant k+3+k(\alpha_k-2)-9-x-y+k+2 = k\alpha_k - 4 - x - y < k\alpha_k - 3 - x$, and therefore pushing an additional run of size $k+3$ produces a stack $(k+3, k(\alpha_k-1)-7-x-y, k\alpha_k-3-x, k(1+\alpha_k), n-4, n)$. ◀

We introduce now a variant of the real numbers α_n and β_n, adapted to our construction.

▶ **Definition 36.** Let n be a positive integer. We denote by γ_n the smallest real number m such that, in every stack $\mathcal{S} = (r_1, \ldots, r_h)$ such that $h \geqslant 2$, $r_h = n$ and $r_{h-1} = n-4$, we have $r_1 + \ldots + r_{h-1} \leqslant m \times n$. If no such real number exists, we simply set $\gamma_n = +\infty$.

▶ **Lemma 37.** *Let* $\underline{\gamma} = \liminf \gamma_n$. *We have* $\underline{\gamma} \geqslant (\overline{\alpha}\underline{\gamma} + 9\overline{\alpha} - \underline{\gamma} + 3)/(2\overline{\alpha} + 4)$.

Proof. Let ε be a positive real number, and let $N_\varepsilon \geqslant N$ be an integer such that $\alpha_\ell \geqslant \overline{\alpha} - \varepsilon$ and $\gamma_\ell \geqslant \underline{\gamma} - \varepsilon$ for all $\ell \geqslant \lfloor (N-6)/(\alpha_\infty + 2) \rfloor$. Since $\overline{\alpha} > 3$, and up to increasing the value of N_ε, we may further assume that $\ell(\alpha_\ell - 3) \geqslant 30$ for every such integers ℓ.

Then, consider some integer $n \geqslant N_\varepsilon$, and let $k = \lfloor (n-6)/(\overline{\alpha} + 2) \rfloor$. By Lemma 35, there exists a stack $(k+3, k(\alpha_k - 1) - 7 - x - y, k\alpha_k - 3 - x, k(1+\alpha_k), n-4, n)$ for some integers $x, y \in \{0,1,2,3\}$. Let also $k' = \lfloor (k(\alpha_k - 1) - 7 - x - y)/2 \rfloor$. Since $k(\alpha_k - 3) \geqslant 30$, it follows that $2(k'-4) \geqslant k(\alpha_k - 1) - 7 - x - y - 2 - 8 \geqslant k(\alpha_k - 1) - 23 \geqslant 2k + 7 > 2(k+3)$. Similarly, and since $\alpha_k \leqslant \alpha_\infty < 5$, we have $k' \leqslant k(\alpha_k - 1)/2 < 2k < 2(k+3)$. Consequently, pushing additional runs of sizes $k' - (k+3)$ and $k' - 4$ produces the stack $(k'-4, k', k(\alpha_k - 1) - 7 - x - y, k\alpha_k - 3 - x, k(1+\alpha_k), n-4, n)$.

Finally, by definition of γ_n, there exists a sequence of runs $p_1, \ldots, p_m$ that produces a stack $(r_1, \ldots, r_h)$ such that $p_1 = r_h = k'$, $p_2 = r_{h-1} = k' - 4$ and $r_1 + \ldots + r_{h-1} = k'\gamma_{k'}$. Hence, pushing the runs $p_3, \ldots, p_m$ produces the stack $(r_1, \ldots, r_{h-1}, k', k(\alpha_k - 1) - 7 - x - y, k\alpha_k - 3 - x, k(1+\alpha_k), n-4, n)$.

Then, recall that that $\underline{\gamma} \leqslant \overline{\alpha}$, that $3 < \overline{\alpha} \leqslant \alpha_\infty < 5$ and that $0 \leqslant x, y \leqslant 3$. It follows that $k \geqslant (n-6)/(\overline{\alpha}+2) - 1 \geqslant n/(\overline{\alpha}+2) - 3$ and that $k' \geqslant (k(\alpha_k - 1) - 7 - x - y)/2 - 1 \geqslant k(\alpha_k - 1)/2 - 8$. This proves that

$$\begin{aligned}
n\gamma_n &\geqslant r_1 + \ldots + r_{h-1} + k' + k(\alpha_k - 1) - 7 - x - y + k\alpha_k - 3 - x + k(1+\alpha_k) + n - 4 \\
&\geqslant k'\gamma_{k'} + k' + k(\alpha_k - 1) - 13 + k\alpha_k - 6 + k(1+\alpha_k) + n - 4 \\
&\geqslant k'(1 + \underline{\gamma} - \varepsilon) + 3k\alpha_k + n - 23 \\
&\geqslant (k(\alpha_k - 1)/2 - 8)(1 + \underline{\gamma} - \varepsilon) + 3k\alpha_k + n - 23 \\
&\geqslant k(3\alpha_k + (1 + \underline{\gamma} - \varepsilon)(\alpha_k - 1)/2) + n - 23 - 8 \times 6
\end{aligned}$$

$$\begin{aligned} n\gamma_n &\geqslant (n/(\overline{\alpha}+2)-3)(3\overline{\alpha}-3\varepsilon+(1+\underline{\gamma}-\varepsilon)(\overline{\alpha}-\varepsilon-1)/2)+n-71 \\ &\geqslant \big(6\overline{\alpha}-6\varepsilon+(1+\underline{\gamma}-\varepsilon)(\overline{\alpha}-\varepsilon-1)+2\overline{\alpha}+4\big)\,n/(2\overline{\alpha}+4)- \\ &\qquad 3(3\overline{\alpha}+(1+\underline{\gamma})(\overline{\alpha}-1)/2)-71 \\ &\geqslant \big(\overline{\alpha}\underline{\gamma}+9\overline{\alpha}-\underline{\gamma}+3-(\overline{\alpha}+\underline{\gamma}-\varepsilon)\varepsilon\big)\,n/(2\overline{\alpha}+4)-152. \end{aligned}$$

Hence, by choosing n arbitrarily large, then ε arbitrarily small, Lemma 37 follows. ◀

From Lemma 37, we derive the asymptotic evaluation of the sequence (α_n), as announced at the beginning of Section A.1.3.

▶ **Proposition 38.** *We have* $\overline{\alpha} = \alpha_\infty = 2+\sqrt{7}$.

Proof. Lemma 37 states that $\underline{\gamma} \geqslant (\overline{\alpha}\underline{\gamma}+9\overline{\alpha}-\underline{\gamma}+3)/(2\overline{\alpha}+4)$ or, equivalently, that $\underline{\gamma} \geqslant (9\overline{\alpha}+3)/(\overline{\alpha}+5)$. Since $\overline{\alpha} \geqslant \underline{\gamma}$, it follows that $\overline{\alpha} \geqslant (9\overline{\alpha}+3)/(\overline{\alpha}+5)$, i.e., that $\overline{\alpha} \geqslant 2+\sqrt{7}$ or that $\overline{\alpha} \leqslant 2-\sqrt{7} < 0$. The latter case is obviously impossible, hence $\overline{\alpha} = \alpha_\infty = 2+\sqrt{7}$. ◀

Finally, we may prove that the constant Δ of Theorem 15 is optimal, as a consequence of the following result.

▶ **Lemma 39.** *For all real numbers* $\Lambda > \Delta$*, there exists a positive real number* K_Λ *such that, for all* $h \geqslant 1$*, there is a stack* $(r_1, \ldots, r_h)$ *for which* $r_h \leqslant K_\Lambda \Lambda^h$.

Proof. Let ε be an arbitrarily small real number such that $0 < \varepsilon < \Lambda/\Delta - 1$, and let N_ε be a large enough integer such that $\alpha_\ell \geqslant \alpha_\infty - \varepsilon$ and for all $\ell \geqslant \lfloor (N-6)/(\alpha_\infty+2) \rfloor$. Then, consider some integer $n_0 \geqslant N_\varepsilon$, and let $k = \lfloor (n-6)/(\alpha_\infty+2) \rfloor$. As shown in the proof of Lemma 37, there are integers $x, y \in \{0,1,2,3\}$, and $n_1 = \lfloor (k(\alpha_k-1)-7-x-y)/2 \rfloor$, such that there exists a stack $(n_1-4, n_1, k(\alpha_k-1)-7-x-y, k\alpha_k-3-x, k(1+\alpha_k), n_0-4, n_0)$. Since $\alpha_\infty \leqslant 5$, we have then

$$\begin{aligned} \Delta^5 n_1 &\geqslant \Delta^5(k(\alpha_\infty-1-\varepsilon)/2-8) \geqslant \Delta^5\left((\alpha_\infty-1-\varepsilon)/(2\alpha_\infty+4)n_0-16\right) \\ &\geqslant (\alpha_\infty-1-\varepsilon)/(\alpha_\infty-1)n_0 - 16\Delta^5. \end{aligned}$$

It follows that $n_0 \leqslant \Delta^5(n_1+16)(\alpha_\infty-1)/(\alpha_\infty-1-\varepsilon) \leqslant \Delta^5(1+\varepsilon)^2 n_1 \leqslant \Lambda^5 n_1$.

Then, we repeat this construction, but replacing n_0 by n_1, thereby constructing a new integer n_2, then replacing n_0 by n_2, and so on, until we construct an integer n_ℓ such that $n_\ell < N_\varepsilon$. Doing so, we built a stack $(n_\ell - 4, n_\ell, \ldots, n_0-4, n_0)$ of size $5\ell+2$, and we also have $\Lambda^{5\ell} N_\varepsilon \geqslant \Lambda^{5\ell} n_\ell \geqslant \Lambda^{5(\ell-1)} n_{\ell-1} \geqslant \ldots \geqslant n_0$. Choosing $K_\Lambda = N_\varepsilon$ completes the proof. ◀

Proof of Theorem 15. We focus here on proving the second part of Theorem 15. Consider a real constant $\Delta' > \Delta$, and let $\Lambda = (\Delta+\Delta')/2$. If h is large enough, then $hK_\Lambda\Lambda^h \leqslant (\Delta')^{h-4}$. Then, let $(r_1, \ldots, r_h)$ be a stack such that $r_h \leqslant K_\Lambda\Lambda^h$, and let n be some integer such that $(\Delta')^{h-4} \leqslant n < (\Delta')^{h-3}$, if any. Considering the stack $(r_1, \ldots, r_h+m)$, where $m = n - (r_1 + r_2 + \ldots + r_h)$, we deduce that $h_{\max} \geqslant h > 3 + \log_{\Delta'}(n)$. Therefore, if n is large enough, we complete the proof by choosing $h = \lfloor \log_{\Delta'}(n) \rfloor + 3$. ◀

A.2 Programs

Algorithm 4: C++ code, file `listobject.c`, python 3.6.5

```
/* Merge the two runs at stack indices i and i+1.
 * Returns 0 on success, -1 on error.
 */
static Py_ssize_t merge_at(MergeState *ms, Py_ssize_t i);

/* Examine the stack of runs waiting to be merged, merging
 * adjacent runs until the stack invariants are re-established:
 *
 * 1. len[-3] > len[-2] + len[-1]
 * 2. len[-2] > len[-1]
 *
 * See listsort.txt for more info.
 *
 * Returns 0 on success, -1 on error.
 */
static int merge_collapse(MergeState *ms){
    struct s_slice *p = ms->pending;

    assert(ms);
    while (ms->n > 1) {
        Py_ssize_t n = ms->n - 2;
        if ((n > 0 && p[n-1].len <= p[n].len + p[n+1].len) ||
            (n > 1 && p[n-2].len <= p[n-1].len + p[n].len)) {
            if (p[n-1].len < p[n+1].len)
                --n;
            if (merge_at(ms, n) < 0)
                return -1;
        }
        else if (p[n].len <= p[n+1].len) {
                 if (merge_at(ms, n) < 0)
                        return -1;
        }
        else
            break;
    }
    return 0;
}
```

Algorithm 5: Java code, file `java.util.TimSort`, jdk 9

```
/* Allocate runs-to-be-merged stack (which cannot be expanded).
 * The stack length requirements are described in listsort.txt.
 * The C version always uses the same stack length (85), but
 * this was measured to be too expensive when sorting "mid-sized"
 * arrays (e.g., 100 elements) in Java. Therefore, we use smaller
 * (but sufficiently large) stack lengths for smaller arrays.
 * The "magic numbers" in the computation below must be changed
 * if MIN_MERGE is decreased. See the MIN_MERGE declaration above
 * for more information. The maximum value of 49 allows for an
 * array up to length  Integer.MAX_VALUE-4, if array is filled by
 * the worst-case stack size  increasing scenario. More explana-
 * tions are given in section 4 of: http://envisage-project.eu/
 * wp-content/uploads/2015/02/sorting.pdf
 */
int stackLen = (len <    120  ?  5 :
                len <   1542  ? 10 :
                len < 119151  ? 24 : 49);
runBase = new int[stackLen];
runLen = new int[stackLen];
...

/**
 * Examines the stack of runs waiting to be merged and merges
 * adjacent runs until the stack invariants are reestablished:
 *
 *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
 *     2. runLen[i - 2] > runLen[i - 1]
 *
 * This method is called each time a new run is pushed onto
 * the stack, so the invariants are guaranteed to hold
 * for i < stackSize upon entry to the method.
 */
private void mergeCollapse() {
    while (stackSize > 1) {
        int n = stackSize - 2;
        if (n > 0 && runLen[n-1] <= 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 {
            break; // Invariant is established
        }
    }
}
```

Algorithm 6: Java code that raises a `java.lang.ArrayIndexOutOfBoundsException`, file `http://igm.univ-mlv.fr/~pivoteau/Timsort/Test.java`

```
import java.util.Arrays;
public class Test {
  final static int[] runLengths = new int[] { 76405736, 74830360, 1181532, 787688, 1575376, 2363064, 3938440,
    6301504, 1181532, 393844, 15753760, 1575376, 787688, 393844, 1969220, 3150752, 1181532, 787688, 5513816,
    3938440, 1181532, 787688, 1575376, 18116824, 1181532, 787688, 1575376, 2363064, 3938440, 787688, 26781392,
    1181532, 787688, 1575376, 2363064, 393844, 4332284, 1181532, 787688, 1575376, 12209164, 1181532, 787688,
    1575376, 2363064, 787688, 393844, 4726128, 1575376, 787688, 1969220, 76405758, 53168940, 1181532, 787688,
    1575376, 2363064, 3938440, 1575376, 787688, 393844, 10633788, 1181532, 787688, 1575376, 2363064, 4332284,
    1181532, 787688, 1575376, 12996852, 1181532, 787688, 1575376, 2363064, 393844, 17329136, 1575376, 787688,
    393844, 1969220, 3150752, 1181532, 393844, 7483036, 1575376, 787688, 1969220, 2756908, 1181532, 787600,
    76405780, 38202802, 114608494, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 11264, 22528, 45056, 90112,
    180224, 360448, 720896, 1441792, 2883584, 5767168, 11387222, 22495132, 319836, 213224, 426448, 639672,
    1066120, 1705792, 426448, 213224, 106612, 4584316, 426448, 213224, 106612, 533060, 106612, 852896, 426448,
    213224, 1599180, 1172732, 319836, 213224, 426448, 5223988, 319836, 213224, 426448, 639672, 1066120, 319836,
    213224, 7782676, 426448, 213224, 533060, 746284, 213224, 1705792, 319836, 213224, 426448, 639672, 2238852,
    426448, 213224, 106612, 2345464, 426448, 213224, 106612, 533060, 106612, 852896, 426448, 213224, 106612,
    22921602, 15245516, 319836, 213224, 426448, 639672, 1172732, 319836, 213224, 426448, 3304972, 319836,
    213224, 426448, 639672, 213224, 1279344, 426448, 213224, 533060, 3838032, 319836, 213224, 426448, 639672,
    213224, 106612, 5330600, 319836, 213224, 426448, 639672, 1066120, 213224, 2345464, 426448, 213224, 106612,
    533060, 106612, 852896, 426448, 213224, 106524, 22921624, 11460724, 34382260, 66, 44, 88, 176, 352, 704,
    1408, 2816, 5632, 11264, 22528, 45056, 90112, 180224, 360448, 720896, 1001792, 1783584, 2649020, 6739370,
    102630, 68420, 136840, 205260, 342100, 547360, 102630, 68420, 1436820, 102630, 68420, 136840, 205260,
    342100, 547360, 102630, 68420, 136840, 205260, 68420, 34210, 1607870, 102630, 68420, 136840, 205260, 342100,
    68420, 34210, 2428910, 102630, 68420, 136840, 205260, 34210, 410520, 102630, 68420, 136840, 1094720, 102630,
    68420, 136840, 205260, 68420, 34210, 444730, 136840, 68420, 34210, 171050, 34210, 6876232, 4618350, 102630,
    68420, 136840, 205260, 34210, 342100, 136840, 68420, 34210, 992090, 102630, 68420, 136840, 205260, 68420,
    342100, 205260, 102630, 68420, 1163140, 102630, 68420, 136840, 205260, 68420, 1607870, 102630, 68420,
    136840, 205260, 342100, 34210, 684200, 136840, 68420, 171050, 239470, 102630, 68332, 6876254, 3438028,
    10314194, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 11264, 22528, 45056, 90112, 180224, 360448, 500896,
    840554, 2018720, 32736, 21824, 43648, 65472, 21824, 10912, 141856, 43648, 21824, 10912, 54560, 10912,
    425568, 43648, 21824, 54560, 76384, 21824, 10912, 185504, 32736, 21824, 43648, 65472, 21824, 10912, 491040,
    32736, 21824, 43648, 65472, 109120, 10912, 731104, 32736, 21824, 43648, 65472, 120032, 32736, 21824, 43648,
    327360, 32736, 21824, 43648, 65472, 21824, 130944, 43648, 21824, 54560, 2062390, 1396736, 32736, 21824,
    43648, 65472, 109120, 43648, 21824, 10912, 294624, 32736, 21824, 43648, 65472, 120032, 32736, 21824, 43648,
    360096, 32736, 21824, 43648, 65472, 10912, 480128, 32736, 21824, 43648, 65472, 109120, 196416, 65472, 32736,
    10912, 76384, 21824, 10824, 2062412, 1031118, 3093442, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 11264,
    22528, 45056, 90112, 180224, 258170, 605616, 9768, 6512, 13024, 19536, 6512, 3256, 42328, 13024, 6512, 3256,
    16280, 3256, 126984, 13024, 6512, 16280, 22792, 6512, 3256, 55352, 9768, 6512, 13024, 19536, 6512, 3256,
    146520, 9768, 6512, 13024, 19536, 32560, 3256, 218152, 9768, 6512, 13024, 19536, 35816, 9768, 6512, 13024,
    100936, 9768, 6512, 13024, 19536, 6512, 3256, 39072, 13024, 6512, 16280, 618662, 416768, 9768, 6512, 13024,
    19536, 32560, 13024, 6512, 3256, 87912, 9768, 6512, 13024, 19536, 35816, 9768, 6512, 13024, 107448, 9768,
    6512, 13024, 19536, 3256, 143264, 13024, 6512, 3256, 16280, 26048, 9768, 3256, 61864, 13024, 6512, 16280,
    22792, 9768, 3168, 618684, 309254, 927850, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632, 11264, 22440, 23056,
    45056, 72314, 181632, 2838, 1892, 3784, 5676, 9460, 15136, 2838, 946, 37840, 3784, 1892, 946, 4730, 7568,
    2838, 1892, 13244, 9460, 2838, 1892, 3784, 43516, 2838, 1892, 3784, 5676, 9460, 1892, 65274, 2838, 1892,
    3784, 5676, 946, 10406, 2838, 1892, 3784, 30272, 2838, 1892, 3784, 5676, 1892, 946, 12298, 3784, 1892, 946,
    4730, 185438, 127710, 2838, 1892, 3784, 5676, 9460, 3784, 1892, 946, 26488, 2838, 1892, 3784, 5676, 946,
    10406, 2838, 1892, 3784, 31218, 2838, 1892, 3784, 5676, 946, 42570, 2838, 1892, 3784, 5676, 9460, 17974,
    3784, 1892, 4730, 6622, 2838, 1804, 185460, 92642, 278014, 66, 44, 88, 176, 352, 704, 1408, 2816, 5632,
    9064, 11528, 23606, 54340, 858, 572, 1144, 1716, 2860, 4576, 858, 286, 11440, 1144, 572, 286, 1430, 2288,
    858, 572, 4004, 2860, 858, 572, 1144, 13156, 858, 572, 1144, 1716, 2860, 572, 19448, 858, 572, 1144, 1716,
    286, 3146, 858, 572, 1144, 8866, 858, 572, 1144, 1716, 572, 286, 3432, 1144, 572, 1430, 55506, 38610, 858,
    572, 1144, 1716, 2860, 1144, 572, 286, 7722, 858, 572, 1144, 1716, 3146, 858, 572, 1144, 9438, 858, 572,
    1144, 1716, 286, 12584, 1144, 572, 286, 1430, 2288, 858, 286, 5434, 1144, 572, 1430, 2002, 858, 484, 55528,
    27676, 83116, 66, 44, 88, 176, 352, 704, 1408, 1716, 3872, 8118, 16192, 264, 176, 352, 528, 176, 88, 1144,
    352, 176, 88, 440, 88, 3432, 352, 176, 440, 616, 176, 88, 1408, 264, 176, 352, 528, 176, 88, 3960, 264, 176,
    352, 528, 880, 88, 5808, 264, 176, 352, 528, 968, 264, 176, 352, 2640, 264, 176, 352, 528, 176, 1056, 352,
    176, 440, 16566, 11264, 264, 176, 352, 528, 880, 352, 176, 2376, 264, 176, 352, 528, 968, 264, 176, 352,
    2816, 264, 176, 352, 528, 88, 3872, 264, 176, 352, 528, 880, 1584, 528, 264, 88, 616, 176, 16588, 8206,
    24706, 66, 44, 88, 176, 352, 704, 1408, 2090, 4708, 66, 44, 88, 132, 220, 352, 66, 44, 88, 990, 66, 44, 88,
    132, 220, 418, 88, 44, 110, 154, 66, 44, 1122, 66, 44, 88, 132, 220, 88, 44, 22, 1716, 88, 44, 110, 154, 44,
    352, 66, 44, 88, 132, 44, 22, 462, 110, 44, 22, 528, 88, 44, 22, 110, 22, 176, 88, 44, 22, 4950, 3256, 66,
    44, 88, 132, 22, 242, 66, 44, 88, 704, 66, 44, 88, 132, 44, 22, 264, 88, 44, 110, 814, 88, 44, 110, 154, 22,
    1144, 66, 44, 88, 132, 220, 44, 22, 506, 88, 44, 22, 110, 22, 176, 88, 44, 22, 4972, 2398, 7282, 66, 44, 88,
    176, 242, 418, 660, 1496, 66, 44, 88, 132, 242, 66, 44, 88, 682, 66, 44, 88, 132, 44, 22, 264, 88, 44, 110,
    1716, 858, 88, 44, 110, 154, 44, 22, 352, 66, 44, 88, 132, 44, 22, 1738, 198, 1760, 175, 156, 18, 17, 19,
    36, 65, 21, 20, 22, 18, 452, 114, 95, 18, 17, 21, 36, 18, 17, 115, 76, 144, 44, 38, 61, 20, 19, 21, 17 };

  public static void main(String[] args) {
    Integer[] arrayToSort = new Integer[1091482190];
    Arrays.fill(arrayToSort, 0);
    int sum = -1;
    for (int i : runLengths) {
      sum += i;
      arrayToSort[sum] = 1;
    }
    Arrays.sort(arrayToSort);
  }
}
```