HAL
open science

Merge Strategies: from Merge Sort to TimSort

Nicolas Auger, Cyril Nicaud, Carine Pivoteau

To cite this version:

Nicolas Auger, Cyril Nicaud, Carine Pivoteau. Merge Strategies: from Merge Sort to TimSort. 2015. hal-01212839v1

HAL Id: hal-01212839
https://hal.science/hal-01212839v1

Preprint submitted on 7 Oct 2015 (v1), last revised 9 Dec 2015 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Merge Strategies: from Merge Sort to TimSort

Nicolas Auger, Cyril Nicaud, and Carine Pivoteau

Université Paris-Est, LIGM (UMR 8049), F77454 Marne-la-Vallée, France
{auger,nicaud,pivoteau}@univ-mlv.fr

Abstract. The introduction of TimSort as the standard algorithm for sorting in Java and Python questions the generally accepted idea that merge algorithms are not competitive for sorting in practice. In an attempt to better understand TimSort algorithm, we define a framework to study the merging cost of sorting algorithms that relies on merges of monotonic subsequences of the input. We propose an optimal strategy for lists and a 2-approximation for arrays. We compare them to the merging strategy of TimSort by designing a simpler yet competitive algorithm based on the same ideas. As a side benefit, our framework allows to establish the announced running time of TimSort, that is $\mathcal{O}(n \log n)$.

1 Introduction

TimSort [7] is a sorting algorithm designed in 2002 by Tim Peters, for use in the Python programming language. It was thereafter implemented in other well-known programming languages such as Java. It is quite a strongly engineered algorithm, but its high-level principle is rather simple: The sequence S to be sorted is decomposed into monotonic runs (*i.e.*, nonincreasing or nondecreasing subsequences of S), which are merged pairwise according to some specific rules.

In order to understand and analyze the merging strategy (meaning the order in which the merges are performed) of TimSort, we consider other sorting algorithms that operate in a similar fashion. For instance, Knuth's NaturalMergeSort [6] decomposes the sequence into nondecreasing runs, which are then merged in the same order as in classical MergeSort. We investigate efficient merging strategies for sorting algorithms that start with a decomposition into runs and we consider different settings, based on practical features such as whether the sequence is encoded with a list or with an array. For each setting, we propose algorithms to compute an efficient tree-like merging strategy, with the constraint that the over cost of computing it must be at most $\mathcal{O}(n)$ time, where n is the length of the input, and that it uses the run lengths only: we are not allowed to compare elements of the sequence for establishing the strategy.

The merging strategy of TimSort follows yet another scheme as it relies on a stack to avoid multiple merges of large runs. We give a general framework to describe such algorithms and use it to design a simpler variant and to establish the proof that the running time of TimSort is $\mathcal{O}(n \log n)$.

The article is organized as follows. We explain the general settings in Section 2. In Section 3, we investigate the case where the input is given as a list,

and propose an algorithm based on Huffman coding to compute the optimal strategy. In Section 4, we focus on arrays and only allow merges of consecutive runs; we propose an algorithm that computes a 2-approximation of the optimal strategy in time $\mathcal{O}(n)$. In Section 5, we set up a general framework for algorithms a la TimSort, and propose a new parametric algorithm which is simpler and which has the same kind of properties; both this algorithm and TimSort are then proved to run in $\mathcal{O}(n \log n)$ time.[1] Finally, some experiments and open questions are presented in Section 6.

Due to lack of space, most of the proofs and some algorithms are omitted in this extended abstract.

2 Settings

2.1 Sequences and runs

For every positive integers i and j, let $[i] = \{1, \ldots, i\}$ and let $[i, j] = \{i, \ldots, j\}$. Let $(E, \leq)$ be a totally ordered set. In this article, we consider non-empty finite sequences of elements of E, that is, elements of $E^+ = \cup_{n \geq 1} E^n$. The *length* $|S|$ of such a sequence is its number of elements. A sequence $S = (s_1, \ldots, s_n)$ is *sorted* when, for every $i \in [n-1]$, $s_i \leq s_{i+1}$. We are interested in *sorting* algorithms that, for any given sequence $S \in E^n$, find a permutation σ of $[n]$ such that $(s_{\sigma(1)}, \ldots, s_{\sigma(n)})$ is sorted. Most of the time, we do not want σ explicitly, but instead directly compute the sequence $\mathbf{sort}(S) = (s_{\sigma(1)}, \ldots, s_{\sigma(n)})$.

A *run* of a sequence $S = (s_1, \ldots, s_n)$ is a non-empty integer interval $[i, j]$ such that either $(s_i, \ldots, s_j)$ or $(s_j, \ldots, s_i)$ is sorted. The former is a *nondecreasing run*, and the latter is a *nonincreasing run*.[2] A *run decomposition* of a sequence S of length n is a nonempty sequence $\mathcal{R} = (R_1, \ldots, R_m)$ of elements of E^+ such that each R_i is a run (either nondecreasing or nonincreasing), and such that $S = R_1 \cdot R_2 \cdots R_m$, where $R \cdot R'$ denote the classical concatenation of sequences.

Example 1. $\mathcal{R}_1 = (2, 3, 5, 7, 11) \cdot (10) \cdot (9) \cdot (8, 9, 10)$ and $\mathcal{R}_2 = (2, 3, 5, 7, 11) \cdot (10, 9, 8) \cdot (9, 10)$ are two run decompositions of $S = (2, 3, 5, 7, 11, 10, 9, 8, 9, 10)$.

2.2 Run-merge sorting algorithms and run decomposition strategies

We now equip the runs with a merge operation. If R and R' are two runs, let $\mathbf{merge}(R, R')$ denote the sequence made of the elements of R and R' placed in nondecreasing order, *i.e.*, $\mathbf{merge}(R, R') = \mathbf{sort}(R \cdot R')$.

In this article we are interested in sorting algorithms that follow what we call a *generic run-merge sort* template. Such algorithms consist of two steps: First the sequence is split into a run decomposition. Then, the runs are merged pairwise until only one remains, which is the sorted sequence associated with the input.[3] This generic algorithm is depicted in Algorithm 1.

[1] This fact is a folklore result for TimSort, but it does not seem to appear anywhere.
[2] Observe that we do not require a run to be maximal, though they will usually be.
[3] Except in the very specific case where $\mathcal{R}$ consists of only one nonincreasing run.

Algorithm 1: Generic Run-Merge Sort for S

```
1 R ← run decomposition of S
2 while |R| ≠ 1 do
3     remove two runs R and R' of R
4     add merge(R, R') to R
5 if the unique run R_1 in R is nonincreasing then reverse R_1
6 return R_1
```

To design such an algorithm, the two main concerns are how the run decomposition is computed, and in which order the runs are merged. Observe that several classical algorithms fit in this abstract settings.

MergeSort is a run-merge sorting algorithm in which each run is reduced to a single element. Note that, in this case, the cost of computing the run decomposition is $O(1)$. Then, the runs are merged according to the recursive calls made during the divide and conquer algorithm.

NaturalMergeSort is a variation of merge sort proposed by Knuth [6]. It consists in first decomposing the sequence into maximal nondecreasing runs, then in using the same merging strategy as in MergeSort. The run decomposition can be obtained by scanning the input sequence S from left to right and by starting a new run every time an element is smaller than the one before. This uses $n-1$ comparisons for a sequence of length n.

TimSort [7] is a relatively new algorithm that is implemented in standard libraries of several common programming languages (Python, Java, ...). This algorithm is highly engineered and uses many efficient heuristics, but this is not our purpose to fully describe it here. However, we are quite interested in the run-merging strategy it relies on, which consists in first computing a decomposition into maximal nonincreasing and nondecreasing runs, which are then merged using a stack. The merging strategy is defined by some invariants that must be satisfied within this stack (merges occur when they are not) and we will give more details on this in Section 5.

Concerning the run decomposition, the idea is to take advantage of the natural maximal runs of S, but each run can be either nonincreasing or nondecreasing, according to order of its first two elements.[4] As for the previous solution, $n-1$ comparisons are required to calculate this decomposition. In *Example 1*, $\mathcal{R}_1$ was computed as in NaturalMergeSort and $\mathcal{R}_2$ as in TimSort.

In the sequel, we consider two different data structures for the input sequence. If it is a list, then the run decomposition is encoded as a list of runs, each run being a list of its elements. Otherwise, if we are working on arrays, runs are encoded by pairs of integers that represent their starting and ending positions.

Since the number of useful strategies to compute a run decomposition is limited, we choose to mostly focus on merging strategies in this paper.

[4] Actually, in TimSort, the size of short runs is artificially increased. We do not consider this feature here and focus on the basic ideas of TimSort only.

2.3 Merge tree and merging cost

To an execution of a run-merge algorithm is naturally associated what we call a *merge tree* or a *consecutive-merge tree*. These are complete binary trees whose leaves are the runs of the run decomposition, and where there is an internal node for each merge, whose children are the runs that are being merged. In a consecutive-merge tree, we furthermore ask that each merge involves two consecutive runs. In a merge tree, any two runs can be merged at each step.

More formally, to build the *merge tree* of a run decomposition $\mathcal{R}$ for a given run-merge algorithm, we proceed as follows. We start with a forest where every tree is reduced to a root labeled by a run of $\mathcal{R}$ (during the whole process, the runs in $\mathcal{R}$ and their merges label the nodes of the trees). At each step, when merging the runs R and R', we remove the two trees $\mathcal{T}$ and $\mathcal{T}'$ of roots R and R' and replace them by a new tree $\mathcal{T} = \begin{smallmatrix} R'' \\ /\backslash \\ \mathcal{T}\,\mathcal{T}' \end{smallmatrix}$, where $R'' = \mathbf{merge}(R, R')$. If only consecutive runs are merged at each step, the merge tree is called a *consecutive-merge tree* (consecutive-merge trees therefore form a subset of merge trees).

We now turn our attention to the cost of a merge and we distinguish two possible representations of the initial sequence S: lists or arrays. First, if S is encoded using doubly linked lists,[5] the standard merging algorithm applied to R and R' uses at most $|R| + |R'| - 1$ comparisons.

If the sequence S is stored in an array A of size n (and the runs are encoded by their starting and ending indices in A), merging runs that are not consecutive may require to shift the whole array, which make the merge-sort strategy quite impractical. *Therefore, when using arrays, we ask that the algorithm only merges consecutive runs.* The classical implementations of the merging procedure use an auxiliary array of length $\min(|R|, |R'|)$, where the smallest run is copied.[6] In this case, the number of comparisons is also at most $|R| + |R'| - 1$.

In the sequel, we therefore consider that the number of comparisons needed to merge two runs R and R' is $\mathbf{c}(R, R') - 1$, where $\mathbf{c}(R, R') = |R| + |R'|$. We extend inductively the definition of $\mathbf{c}$ to merge trees by (R is a run of the initial run decomposition):

$$\mathbf{c}\,(R) = 0, \quad \text{and} \quad \mathbf{c}\begin{pmatrix} \mathbf{merge}(R', R'') \\ /\backslash \\ \mathcal{T}\,\mathcal{T}' \end{pmatrix} = \mathbf{c}(R', R'') + \mathbf{c}(\mathcal{T}) + \mathbf{c}(\mathcal{T}').$$

Let $h_{\mathcal{T}}(R)$ be the height of the run R in the merge tree $\mathcal{T}$.

Lemma 1. *The number of comparisons done during the merge stage of a run-merge algorithm of tree $\mathcal{T}$ on a run decomposition $\mathcal{R}$ is at most $\mathbf{c}(\mathcal{T}) - |\mathcal{R}| + 1$. Moreover, we have $\mathbf{c}(\mathcal{T}) = \sum_{R \in \mathcal{R}} h_{\mathcal{T}}(R)\,|R|$.*

[5] To use singly linked lists, the runs should be either all nondecreasing or all nonincreasing, to avoid the additional cost induced by computing reverse lists.

[6] This extra memory requirement is a reason why QuickSort is sometimes preferred to MergeSort, even if it performs more comparisons in the worst and average cases.

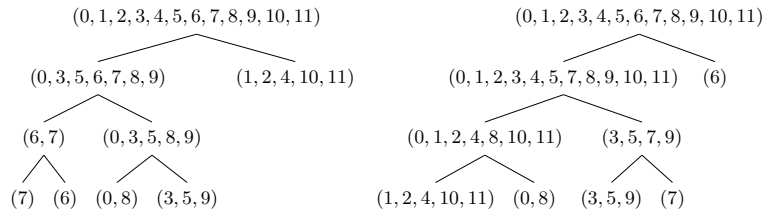

Fig. 1. The merge stage of HuffmanSort (left) and NaturalMergeSort (right) for the initial sequence $S = (1, 2, 4, 10, 11, 0, 8, 3, 5, 9, 7, 6)$. The merge cost is 26 for the left tree and 34 for the right one.

3 Optimal merging strategy for lists: HuffmanSort

In this section, sequences are assumed to be encoded using doubly linked lists. We are therefore allowed to merge runs that are not consecutive.

3.1 A greedy optimal merge algorithm

We propose an algorithm that optimizes the number of comparisons, for a given run decomposition $\mathcal{R}$, when the cost of $\mathbf{merge}(R, R')$ is $|R|+|R'|-1$. This comes down to find a merge tree that minimizes the sum that appears in Lemma 1.

Consider the greedy algorithm that consists in merging the two shortest runs at each step; we call this algorithm HuffmanSort (see example of Fig. 1). The merge tree built by HuffmanSort is exactly the same as the Huffman tree [5] associated to symbols having for weights the $|R_i|$'s, for $\mathcal{R} = (R_1, \ldots, R_m)$. The average code length of this Huffman tree is exactly $\frac{1}{m} \sum_{R \in \mathcal{R}} h(R)\, |R|$ and since Huffman algorithm is optimal for prefix codes, we get the following lemma.

Lemma 2. *For any fixed run decomposition $\mathcal{R}$, the merge tree associated to* HuffmanSort *is a merge tree of $\mathcal{R}$ that minimizes the cost $\boldsymbol{c}$.*

3.2 Reducing additional costs

In this section, we discuss how to implement the greedy merge strategy at a reasonable cost. The classical way of implementing Huffman algorithm consists in using a priority queue. If we do this, the cost of each step is $\mathcal{O}(\log m)$, where m is the number of runs, as we have (1) to extract the shortest run twice and (2) to add their merge. The additional cost is therefore in $\mathcal{O}(m \log m)$, where m is the number of runs. This can be of the same order of magnitude than the worst case complexity of the algorithm, if the number of runs is linear in n, which is the case for uniform random inputs.

Fortunately, this additional cost can be reduced significantly. First recall that van Leeuwen [8] proved that if the weights are sorted in nondecreasing order, then there is a linear algorithm to construct the Huffman tree: We keep a list L

containing the sequence of initial runs in nondecreasing order of their lengths, and a queue Q containing the runs that are built during the merging process, also in nondecreasing order. Q is empty at the beginning. At each step, it is sufficient to check the first elements of L an Q to select the two shortest runs. When these two runs are merged, it creates a new run of length greater than or equal to any other run that have been created before: therefore, it can be safely appended to Q. As every step can be performed in constant time, the running time of the algorithm is linear in the number of runs of $\mathcal{R}$. To apply this idea, we must first sort the initial runs by their lengths. Since these lengths are between 1 and n, this can be done as in CountingSort [1]: initialize an array A of n empty lists, and insert each run R in $A[|R|]$; the queue Q is then obtained by going through A.[7] The whole process requires $\mathcal{O}(n)$ time and space.

We can further reduce this additional cost. Indeed, the size-n array used for sorting the runs according to their length is sparse: as the sum of the run lengths is n, there cannot be many large runs, which can be exploited as follows. Let λ_n be an integer-valued sequence such that $\lambda_n = \Omega(\log n)$ and $\lambda_n = o(n)$. For instance, one can take $\lambda_n = \lceil \sqrt{n} \rceil$. To sort the runs by their lengths, as before, we use an array A to store runs of length at most λ_n, but we put the longest runs into a separate list B, which is sorted using an optimal sorting algorithm. The merges are then performed in optimal order going trough A and then B, using an additional queue, as in the van Leeuwen algorithm. Since there are at most $\frac{n}{\lambda_n}$ elements in the second list, the running time of this construction is $\mathcal{O}(m + \lambda_n + \frac{n}{\lambda_n} \log n)$, and the additional space is $\mathcal{O}(\lambda_n)$. In any case, if $\lambda_n = \Omega(\log n)$, then the over cost is at most $\mathcal{O}(n)$. The sequence λ_n is therefore used to set the trade-off between time and space for the construction of the optimal strategy.

Theorem 1. *Given a run decomposition $\mathcal{R}$ of a size-n sequence into m runs, the algorithm* HuffmanSort *computes a merge tree $\mathcal{T}$ that minimizes $\mathbf{c}(\mathcal{T})$ for this $\mathcal{R}$. Let λ_n be any integer-valued sequence such that $\lambda_n = \Omega(\log n)$ and $\lambda_n = o(n)$. The running time of building $\mathcal{T}$ is $\mathcal{O}(m + \lambda_n + \frac{n}{\lambda_n} \log n)$, and the additional space required is $\mathcal{O}(\lambda_n)$.*

3.3 A note on the uniform random case

In this section we explain how the data structures used in HuffmanSort can be efficiently tuned for random lists, for the uniform distribution. Our remark relies on the following lemma.

Lemma 3. *For given $n \geq 1$, let σ be a random permutation of $[n]$ taken uniformly at random. Then, for any $c > 0$, the probability that there is a run of length greater than $\log n$ in the run decomposition of* NaturalMergeSort *or* TimSort *is $\mathcal{O}(n^{-c})$.*

[7] In this case, we actually do not even need an additional structure for Q, as the runs created by merging can directly be inserted in A.

As a consequence of Lemma 3, it is unlikely that there are long runs, in the uniform case. Thus HuffmanSort works well with an extra space of $\mathcal{O}(\log n)$, obtained by taking $\lambda_n = \lceil \log n \rceil$. Since there is a linear number of runs in a typical random permutation, the additional time cost of $\mathcal{O}(n)$ is unavoidable. On the other hand, most often, we do not need to sort the list of long runs, since there are none of them with high probability.

4 Merging strategies for arrays

We now consider that the elements of S are stored in an array and we therefore only allow merges of consecutive runs. Indeed, merging non-consecutive runs would require a lot of additional space or some costly shifts. This means that the merging strategies are no longer encoded into merge trees, but into consecutive-merge trees. For a given sequence of runs $\mathcal{R}$, let $\mathbf{Opt}(\mathcal{R})$ be the smallest $\mathbf{c}(\mathcal{T})$, where $\mathcal{T}$ ranges over all possible consecutive-merge trees for $\mathcal{R}$.

4.1 The optimal consecutive merging strategy

The problem of computing the optimal merge tree when only consecutive merges are allowed resemble the classical matrix chain ordering problem [1, Ch. 15.2]. However, the improvements and approximation algorithms proposed in [3,4] do not seem to apply here. As for matrices, the optimal strategy can be computed using a dynamic programming approach: if the run lengths sum to n, the induction is

$$\mathbf{Opt}(\mathcal{R}) = \min_{i \in [m-1]} \Big[\mathbf{Opt}(R_1, \ldots, R_i) + \mathbf{Opt}(R_{i+1}, \ldots, R_m) + n \Big],$$

as the last merge costs n. Unfortunately, this results in an $\mathcal{O}(m^3)$ running time algorithm, which is prohibitive in our settings. Since there is little hope to find a linear algorithm to compute this optimal strategy, we propose, instead, a linear approximation algorithm in the next section.

4.2 An efficient heuristic: GreedyMergeSort

Let $\mathcal{R} = (R_1, \ldots, R_m)$ be a sequence of runs. In this section we analyze the greedy strategy that consists in iteratively merging the two consecutive runs R_i and R_{i+1} that minimize the quantity $|R_i|+|R_{i+1}|$. This is the same as HuffmanSort, except that only consecutive runs can be merged. We call this algorithm GreedyMergeSort and we establish that this is a good approximation of the optimal solution:

Theorem 2. *Let $\mathcal{R}$ be a run decomposition. For any consecutive-merge tree $\mathcal{T}$ of $\mathcal{R}$ produced by* GreedyMergeSort*, we have $\boldsymbol{c}(\mathcal{T}) \leq 2\, \boldsymbol{Opt}(\mathcal{R})$.*

As a consequence, the number of comparisons performed by GreedyMergeSort *during the merge stage is at most $2\, \boldsymbol{Opt}(\mathcal{R}) + |\mathcal{R}| - 1$.*

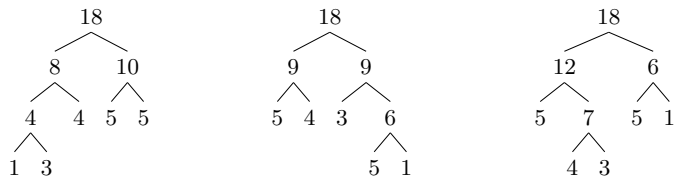

Fig. 2. Merge trees (showing only run lengths) of HuffmanSort (left), $\mathbf{Opt}(\mathcal{R})$ (center) and of GreedyMergeSort (right), for runs of lengths $(5, 4, 3, 5, 1)$. Their merge costs, which are the sum of the internal nodes, are respectively 40, 42 and 43.

In order to pick two consecutive runs with minimal merge cost at each step, our algorithm needs to keep a list of pairs of runs which is sorted according to their merge cost. We use the same idea as for HuffmanSort and we store them in an array A of size n, such that the index of a pair (R_i, R_{i+1}) in A is $|R_i| + |R_{i+1}|$. Then we can go through A to perform the merges using the greedy strategy, but we need to keep A updated after each merge. Indeed, when merging R_i and R_{i+1} into a new run R', we should delete the pairs (R_{i-1}, R_i) and (R_{i+1}, R_{i+2}) and replace them by new pairs (R_{i-1}, R') and (R', R_{i+2}). Since the added pairs have a higher cost than the deleted ones, they are placed further away in A. For any pair of runs, we can do this at cost $O(1)$, by keeping pointers on the previous and next pair of runs in the run decomposition.

Theorem 3. *Building a consecutive-merge tree with* GreedyMergeSort *can be done using $\mathcal{O}(n)$ space and time.*

In Fig. 2, we give an example of the different consecutive-merge trees obtained for the same run decomposition, using each of the strategies that have been discussed so far. Proposition 1 below states the complexity of these three algorithms.

Proposition 1. *The number of comparisons performed during the merge stage of* HuffmanSort*, the optimal consecutive merging and* GreedyMergeSort *is $\mathcal{O}(n \log m)$, where n is the length of the sequence and m is the number of runs.*

5 TimSort-like strategies

As mentioned above, TimSort is a recent sorting algorithm for arrays that follows our generic run-merge template. As in the previous section, only consecutive merges are performed in order to avoid additional costs. It contains many heuristics that will not be discussed here. In the sequel, we will therefore describe a simplified version of TimSort, called SimplifiedTimSort, which focus on the main theoretical ideas.

Before describing SimplifiedTimSort in details, we propose a framework to design a whole class of merge strategies based on the use of a stack, which we call *stack strategies*. SimplifiedTimSort will be an example of such a strategy.

Algorithm 2: Generic Stack Run-Merge Sort for the strategy $\mathfrak{S}$

```
R ← run decomposition of S
X ← ∅
while R ≠ ∅ do
    R ← pop(R)
    Append R to X
    while X violates at least one rule of 𝔖 do
        (ρ, μ) ← first pair such that ρ is not satisfied
        Apply μ to X                                  /* ρ is activated */
while |X| ≥ 1 do
    R, R' ← pop(X), pop(X)
    Append merge(R, R') to X
return the unique element of X
```

5.1 Stack strategies

Let $\mathcal{R} = (R_1, \ldots, R_m)$ be a run decomposition. A stack strategy relies on a stack $\mathcal{X}$ of runs that is initially empty. During the first stage, at each step, a run is extracted from $\mathcal{R}$ and added to the stack. The stack is then updated, by merging runs, in order to assure that some conditions on the top of the stack are satisfied. These conditions and the way runs are merged when they are not satisfied define the strategy. The second stage occurs when there is no more run in $\mathcal{R}$: the runs in $\mathcal{X}$ are then merged pairwise until only one remains.

A *rule* of degree $k \geq 2$ is a property of a stack $\mathcal{X}$ that involves the k topmost elements of $\mathcal{X}$. By convention, the rule is always satisfied when there are less than k elements in $\mathcal{X}$. A *merge strategy* of degree k is the process of merging some of the k topmost runs in a stack $\mathcal{X}$; at least one merge must be performed. A *stack strategy* of degree k consists of a nonempty sequence of s pairs (ρ_i, μ_i), where ρ_i is a rule of degree at most k and μ_i is a merge strategy of degree at most k. The *stack-merge algorithm* associated with the stack strategy $\mathfrak{S} = \langle(\rho_1, \mu_1), \ldots, (\rho_s, \mu_s)\rangle$ is depicted in Algorithm 2. The order of the rules matters: when one or several rules are violated, the merge strategy associated with the first one, and only this one, is performed. This rule is said to be *activated*.

Note that such an algorithm always halts, as the inner loop reduces the number of runs in $\mathcal{X}$: at some point the stack $\mathcal{X}$ contains less elements than the minimal degree of the rules, which are then all satisfied.

Before discussing this family of algorithms further, we provide two examples: SimplifiedTimSort and a new algorithm we called α-StackSort.

5.2 SimplifiedTimSort and α-StackSort

SimplifiedTimSort can be seen as a stack-merge algorithm of degree 4. It is not the way it is usually written, but it is strictly equivalent to the following merge strategy, for a stack that ends with the runs W, X, Y and Z:

	ρ_4 violated			ρ_2 violated	ρ_4 violated	
•	•	•	•	$(8,9)$	•	•
•	$(6,5,2)$	•	$(3,4,10)$	$(3,4,10)$	$(3,4,8,9,10)$	•
$(1,7)$	$(1,7)$	$(1,2,5,6,7)$	$(1,2,5,6,7)$	$(1,2,5,6,7)$	$(1,2,5,6,7)$	$(1,\ldots,10)$

Fig. 3. The stack configurations during the execution of SimplifiedTimSort for the sequence $\mathcal{S} = (1,7,6,5,2,3,4,10,8,9)$.

- $\rho_1 := |X| \geq |Z|$ and μ_1 consists in merging X and Y;
- $\rho_2 := |X| > |Y| + |Z|$ and μ_2 consists in merging Y and Z;
- $\rho_3 := |W| > |X| + |Y|$ and μ_3 consists in merging Y and Z;
- $\rho_4 := |Y| > |Z|$ and μ_4 consists in merging Y and Z;

An example of the successive states of the stack is given in Fig 3. Remark that in the original version of TimSort, the third rule was missing. This lead to some problems; in particular, Lemma 4 did not hold without this rule. This new rule was proposed in [2], where the problem in the invariant was first identified, and quickly corrected in Python.[8]

We propose our own stack-merge algorithm α-StackSort, which is of degree 2. It depends on a fixed parameter $\alpha > 1$, and consists only in one rule ρ which is $|Y| > \alpha\,|Z|$. If it is violated, μ consists in merging Y and Z. The algorithm α-StackSort is therefore a very simple stack-merge algorithm.

5.3 Analysis of SimplifiedTimSort and α-StackSort

The rules and merge strategies of both algorithms are designed in order to ensure that some global invariants hold throughout the stack. This is the meaning of the next two lemmas. The part on SimplifiedTimSort was proven in [2].

Lemma 4. *Let $\mathcal{X} = (x_1, \ldots, x_\ell)$ be the stack configuration at the beginning of any iteration of the while loop at line 3 of Algorithm 2.*
- *For* SimplifiedTimSort *we have $|x_i| > |x_{i+1}| + |x_{i+2}|$, for every $i \in [\ell - 2]$.*
- *For α-*StackSort *we have $|x_i| > \alpha\,|x_{i+1}|$, for every $i \in [\ell - 1]$.*

As remarked by Tim Peters who designed TimSort, the stack contains $\mathcal{O}(\log n)$ runs at any time. This is the same for α-StackSort.

Lemma 5. *At any time during the execution of* SimplifiedTimSort *or α-*StackSort *on a sequence of length n, the stack $\mathcal{X}$ contains $\mathcal{O}(\log n)$ runs.*

Next theorem is a folklore result for TimSort, announced in the first description of the algorithm [7]. However, we could not find its proof anywhere in the literature. The same result holds for α-StackSort. Notice that this is not a direct consequence of Lemma 5: if we merge the runs as they arrive, the stack has size $\mathcal{O}(1)$ but the running time is $\mathcal{O}(n^2)$.

Theorem 4. *The number of comparisons needed for* SimplifiedTimSort *and α-*StackSort *to sort a sequence of length n is $\mathcal{O}(n \log n)$.*

[8] `https://hg.python.org/cpython/file/default/Objects/listobject.c`

Sketch of proof. To analyze the running time of α-StackSort, we rely on a classical technique used for amortized complexity. We define a quantity C that is set to 0 at the beginning of the algorithm. This quantity is increased by $(1+\alpha)\,|\mathcal{X}|\,|R|$ whenever a new run R is added in the stack $\mathcal{X}$. By Lemma 5, this is at most $(1+\alpha)\log n\,|R|$. Hence, the sum of all increases of C is bounded from above by $(1+\alpha)\log n \sum_{R\in\mathcal{R}} |R| = \mathcal{O}(n\log n)$. The quantity C is decreased whenever a merge is performed, by an amount equal to this merge cost. We can readily prove that C is always non-negative. Hence, the total number of comparisons performed in this part is $\mathcal{O}(n\log n)$.

The last while loop also performs at most $\mathcal{O}(n\log n)$ comparisons, as the stack is of length $\mathcal{O}(\log n)$: by Lemma 5, every run is involved in at most $\mathcal{O}(\log n)$ merges during this loop.

If we want to proceed for SimplifiedTimSort as for α-StackSort, there are some technical difficulties inherent to the structure of the rules in SimplifiedTimSort. Again, define a variable C initialized with 0 and which is increased by $3iR$ whenever a run R arrive at position i on the stack. We still remove an amount equal to this merge cost whenever two runs are merged. However, we cannot directly guarantee that C is always positive; for some cases we need to consider several consecutive merges made by the algorithm in order to conclude. Hence, we unroll the main while loop as needed, to obtain an algorithm equivalent to the main while loop, but that is much bigger. On this redundant code we can prove that C remains nonnegative. □

5.4 About TimSort and its variants

There are several reasons why TimSort has been adopted as a standard sorting algorithm in many different languages. An important difference between TimSort and other similar algorithms such as NaturalMergeSort or GreedyMergeSort is the number of cache misses done during their execution. Indeed, in TimSort, runs are computed on the fly, and merges most often apply on the last few computed runs. Hopefully, they are still in the cache when they are needed. Analyzing cache misses is beyond the scope of this article, but we can notice that stack strategies of small degree like α-StackSort have the same kind of behavior, and should be cache-efficient too.

An interesting feature of α-StackSort is that the value of α can be chosen to improve its efficiency, provided we have some knowledge on the distribution of inputs. It is even possible to change the value of α dynamically, if the algorithm finds a better value in view of the first elements. The stack invariant can be violated if α is increased, but this does not affect the complexity of the algorithm.

Also observe that after designing a stack strategy, it is straightforward to take benefit from all the heuristics implemented in TimSort, as we just change the part where the rules are checked and the appropriate merges are performed.

n	k	Huffman	opt. consec.	GreedyMS	TimSort	α-Stack
10,000	-	121,862.02	-	124,292.43	129,271.65	129,178.79
10,000	10	28,277.40	30,013.87	30,284.15	33,581.81	33,479.05
10,000	100	60,852.36	62,103.52	63,200.42	68251.65	67,154.79
100,000	100	607,144.56	619,912.79	630,031.46	678,449.70	669,692.66

Fig. 4. Average number of comparisons performed by merge sorting algorithms ($\alpha = 1.5$, Line 1: random permutations of size n, Lines 2-4: sequences of size n with k runs).

6 Experiments and open questions

We ran a few experiments to measure empirically the differences between our various algorithms. Uniform random permutations of size $10,000$ were used for the first experiments, while we used sequences of exactly k runs in the second ones. The results, given in Fig. 4, indicates the cost $\mathbf{c}$ of the different merging strategies. We also checked on random permutations that the α-StackSort strategy performs as well as the implementation of TimSort in Java.[9]

The first open question we would like to mention is whether the the ratio of approximation of Theorem 2 can be improved. We suspect that it decreases with the number of runs. It is also natural to ask if the number of comparisons performed by TimSort is in $\mathcal{O}(n \log m)$ where m is the number of runs. This is the case for MergeSort, HuffmanSort and GreedyMergeSort. It can be proved that it does not hold for α-StackSort, even though it should be easy to design a $\mathcal{O}(n \log m)$ version of this algorithm.

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition, 2009.
2. 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*. to appear in the *Proceedings of CAV 2015*. `http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf`, 2015.
3. T. C. Hu and M.-t. Shing. Computation of matrix chain products. part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.
4. T. C. Hu and M.-t. Shing. Computation of matrix chain products. part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.
5. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.
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. T. Peters. Timsort description, accessed june 2015. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`.
8. J. van Leeuwen. On the Construction of Huffman Trees. In *ICALP 1976, Edinburgh, July 20-23, 1976*, pages 382–410, 1976.

[9] Benchmark using jmh: `http://openjdk.java.net/projects/code-tools/jmh/`

Appendix A: Algorithms

Auxiliary functions:

- NEXTRUN(A, i) is a function that takes an array A containing lists of runs, an index i and returns the position j of the next run in A (either i if $A[i]$ is not empty or the next $j > i$ such that $A[j]$ is not empty).
- EXTRACTMINRUN(A, B) is a function that takes any two data structures that contain runs and returns the smallest run in the union of A and B.

Algorithm 3: HUFFMANSORT (S, n)

```
smallRuns, largeRuns ← countRuns(S, λ_n)                    /* λ_n ∈]0, n] */
i ← 0
Q ← ∅
while smallRuns is not empty do
    i ← nextRun(smallRuns, i)
    R_1 ← extractMinRun(smallRuns[i], Q)
    if smallRuns is empty then
        add R_1 to largeRuns                                 /* handling the last run */
    else
        i ← nextRun(smallRuns, i)
        R_2 ← extractMinRun(smallRuns[i], Q)
        M ← merge(R_1, R_2)
        add M to Q
/* the runs in Q are sorted by design                        */
sort the runs in largeRuns
while |Q| ≠ 1 or largeRuns is not empty do
    R_1 ← extractMinRun(largeRuns, Q)
    R_2 ← extractMinRun(largeRuns, Q)
    M ← merge(R_1, R_2)
    add M to Q
S ← pop(Q)
```

Algorithm 4: GreedyMergeSort (S, n)

$\mathcal{R} = (R_1, R_2, \ldots, R_m) \leftarrow$ run decomposition of S
$A \leftarrow$ an array of size n filled with empty doubly linked lists
$prev \leftarrow null$
while $|\mathcal{R}| \neq 2$ **do**
  $R_1 \leftarrow \text{pop}(R)$
  $R_2 \leftarrow \text{first}(R)$ `/* $R_2$ is not removed from $R$ */`
  $V \leftarrow (left = R_1, right = R_2, prev = prev, next = null)$
  if $prev \neq null$ **then**
    $prev.next \leftarrow$ a pointer on V
  $prev \leftarrow$ a pointer on V
  add V to $A[|R_1| + |R_2|]$
$i \leftarrow 0$
while $i \neq n$ **do**
  MergeMinAndUpdate(A, i)

`/* this algorith works in place: $S$ is sorted */`

MergeMinAndUpdate (A, i)
  $i \leftarrow$ nextRuns(A, i)
  $V \leftarrow \text{pop}(A[i])$
  $R \leftarrow$ **merge**$(V.left, V.right)$
  $V_1 \leftarrow (V.prev.left, R, V.prev.prev, null)$
  $V_2 \leftarrow (R, V.next.r, V_1, V.next.next)$
  $V_1.next \leftarrow$ a pointer on V_2
  add V_1 to $A[|V_1.left| + |V_1.right|]$
  add V_2 to $A[|V_2.left| + |V_2.right|]$
  delete $V.prev$ and $V.next$

Algorithm 5: `merge_collapse(stack ms)`, the main loop of TimSort for Python (in C)

```
while (n > 1) {
  n = size(ms) - 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;
```

Appendix B: Proofs

▶ Proof of Lemma 1

Every merge of R and R' in $\mathcal{T}$ costs at most one less than $\mathbf{c}(R, R')$. Since there are exactly $|\mathcal{R}| - 1$ merges in total, we directly get that there are at most $\mathbf{c}(\mathcal{T}) - |\mathcal{R}| + 1$ comparisons performed during the merging phase for the strategy given by $\mathcal{T}$.

By structural induction. If $\mathcal{T}$ is reduced to a single node R, then $\mathcal{R} = (R)$ and $h_{\mathcal{T}}(R) = 0 = \mathbf{c}(R)$. Assume now that the property hold for $\mathcal{A}$ and $\mathcal{A}'$ and that $\mathcal{T} = \overset{\mathbf{merge}(R,R')}{\underset{\mathcal{A}\ \mathcal{A}'}{/\backslash}}$, where R and R' are the runs at the roots of $\mathcal{A}$ and $\mathcal{A}'$, respectively. By definition of $\mathbf{c}$, we have $\mathbf{c}(\mathcal{T}) = \mathbf{c}(R, R') + \mathbf{c}(\mathcal{A}) + \mathbf{c}(\mathcal{A}')$. Hence, by induction hypothesis,

$$\mathbf{c}(\mathcal{T}) = \mathbf{c}(R, R') + \sum_{X \text{ leaf of } \mathcal{A}} h_{\mathcal{A}}(X)\,|X| + \sum_{X \text{ leaf of } \mathcal{A}'} h_{\mathcal{A}'}(X)\,|X|.$$

But $\mathbf{c}(R, R')$ is the sum of the lengths of the runs of $\mathcal{R}$, and thus $\mathbf{c}(R, R') = \sum_{X \text{ leaf of } \mathcal{A}} |X| + \sum_{X \text{ leaf of } \mathcal{A}'} |X|$. Therefore,

$$\mathbf{c}(\mathcal{T}) = \sum_{X \text{ leaf of } \mathcal{A}} (1 + h_{\mathcal{A}}(X))\,|X| + \sum_{X \text{ leaf of } \mathcal{A}'} (1 + h_{\mathcal{A}'}(X))\,|X|.$$

This conclude the proof, as for a leaf X of $\mathcal{A}$ (resp. of $\mathcal{A}'$), we have $1 + h_{\mathcal{A}}(X) = h_{\mathcal{T}}(X)$ (resp. $1 + h_{\mathcal{A}'}(X) = h_{\mathcal{T}}(X)$). □

▶ Proof of Lemma 3

Let $\ell \geq 2$. For any $i \in [n - \ell + 1]$, let S_i be the set of permutations σ of $[n]$ such that σ is monotonic on $[i, i + \ell - 1]$. If σ is taken uniformly at random, then the ordering of $(\sigma(i), \ldots, \sigma(i + 1))$ is also a uniform order on a set of size ℓ. Hence it is monotonic with probability $\frac{2}{\ell!}$.

If there is a run of length greater than ℓ in the run decomposition of NaturalMergeSort or TimSort, then σ belongs to a S_i for some $i \in [i, i + \ell - 1]$. By the union bound, this happens with probability at most $\frac{2n}{\ell!}$. Taking $\ell = \lceil \log n \rceil$ concludes the proof by Stirling formula. □

▶ Proof of Theorem 2

For any positive integer $i < m$, we denote by

$$\mathcal{R}_{[i]} = (R_1, \ldots, R_{i-1}, R_i + R_{i+1}, R_{i+2}, \ldots, R_m),$$

the sequence of length $m - 1$ obtained after replacing R_i and R_{i+1} by $R_i + R_{i+1}$. We say that (R_i, R_{i+1}) is a *minimal pair* of $\mathcal{R}$ if for all positive $j < m$, $R_j + R_{j+1} \geq R_i + R_{i+1}$.

We establish several lemmas, whose proofs are quite similar. Nonetheless, we think it is clearer to do this way than to try to put everything in one lemma.

In the sequel, if $\mathcal{A}$ is a merging tree, let $\|\mathcal{A}\|$ be the sum of the lengths of its runs (*i.e.* the lengths of the leaves of $\mathcal{A}$).

First observe that the first merge made by GreedyMergeSort, which involves R_i and R_{i+1}, can be either a choice that belongs to an optimal strategy or not. In the former case, $\mathbf{Opt}(\mathcal{R}) = \mathbf{Opt}(\mathcal{R}_{[i]}) + R_i + R_{i+1}$. In the latter case, $\mathbf{Opt}(\mathcal{R}) < \mathbf{Opt}(\mathcal{R}_{[i]}) + R_i + R_{i+1}$, as merging R_i and R_{i+1} is not an optimal choice.

Lemma 6. *Let $\mathcal{R} = (R_1, \ldots, R_m)$ be a sequence of runs and let (R_i, R_{i+1}) be a minimal pair of $\mathcal{R}$. If $\boldsymbol{Opt}(\mathcal{R}) < \boldsymbol{Opt}(R_{[i]}) + R_i + R_{i+1}$, then in every optimal merging tree $\mathcal{T}$ of $\mathcal{R}$, R_i is the right child of its father and R_{i+1} is the left child of its father.*

Proof. By symmetry we only need to prove that R_i is the right child of its father in $\mathcal{T}$. Assume by contradiction that it is the left child of its father x. The right child $\mathcal{A}$ of x cannot be a tree reduced to R_{i+1}, as by hypothesis, R_i and R_{i+1} are not siblings in an optimal tree. Hence, R_{i+1} is the left child of a node y of $\mathcal{A}$. Let $\mathcal{B}$ be its associated right subtree: we have $\overset{x}{\underset{R_i\ \mathcal{A}}{/\backslash}}$ and $\overset{y}{\underset{R_{i+1}\ \mathcal{B}}{/\backslash}}$, and the height $h_{\mathcal{T}}(x)$ of x in $\mathcal{T}$ is smaller than the height $h_{\mathcal{T}}(y)$ of y. Let $\mathcal{T}'$ be the tree obtained by changing the node R_i into $\overset{\bullet}{\underset{R_i\ R_{i+1}}{/\backslash}}$ and y into $\mathcal{B}$. The tree $\mathcal{T}'$ is still a merging tree of $\mathcal{R}$. By construction of $\mathcal{T}'$ and by Lemma 1, we have

$$\mathbf{c}(\mathcal{T}') - \mathbf{c}(\mathcal{T}) = |R_i| + \big(h_{\mathcal{T}}(x) - h_{\mathcal{T}}(y) + 1\big)\,|R_{i+1}| - \|\mathcal{B}\|.$$

Since $h_{\mathcal{T}}(x) - h_{\mathcal{T}}(y) + 1 \leq 0$, then $\mathbf{c}(\mathcal{T}') - \mathbf{c}(\mathcal{T}) \leq |R_i| - \|\mathcal{B}\|$. But $\mathcal{B}$ contains R_{i+2} as it is not empty, and $|R_{i+2}| \geq |R_i|$ by minimality of (R_i, R_{i+1}). Hence $\|\mathcal{B}\| \geq |R_i|$ and therefore $\mathbf{c}(\mathcal{T}') \leq \mathbf{c}(\mathcal{T})$. Thus $\mathcal{T}'$ is optimal, which is a contradiction with the hypothesis, since R_i and R_{i+1} are siblings in $\mathcal{T}'$. □

Lemma 7. *Let $\mathcal{R} = (R_1, \ldots, R_m)$ be a sequence of runs and let (R_i, R_{i+1}) be a minimal pair of $\mathcal{R}$. Assume that $\boldsymbol{Opt}(\mathcal{R}) < \boldsymbol{Opt}(R_{[i]}) + R_i + R_{i+1}$. Then R_i and R_{i+1} have same height in any optimal merging tree.*

Proof. Let $\mathcal{T}$ be any optimal tree for $\mathcal{R}$. By Lemma 6, R_i is the right child of its father x and R_{i+1} is the left child of its father y. Let $\mathcal{A}$ be the left subtree of x and let $\mathcal{B}$ be the right subtree of y: we have $\overset{x}{\underset{\mathcal{A}\ R_i}{/\backslash}}$ and $\overset{y}{\underset{R_{i+1}\ \mathcal{B}}{/\backslash}}$ in $\mathcal{T}$. Let also $h_{\mathcal{T}}(x)$ and $h_{\mathcal{T}}(y)$ be the respective heights of x and y in $\mathcal{T}$. Assume by contradiction that $h_{\mathcal{T}}(x) \neq h_{\mathcal{T}}(y)$, and assume by symmetry that $h_{\mathcal{T}}(x) > h_{\mathcal{T}}(y)$. Consider the tree $\mathcal{T}'$ obtained from $\mathcal{T}$ by changing the node R_i into $\overset{\bullet}{\underset{R_i\ R_{i+1}}{/\backslash}}$ and y into $\mathcal{B}$. The tree $\mathcal{T}'$ is still a merging tree of $\mathcal{R}$. Moreover, by construction of $\mathcal{T}'$, we have

$$\mathbf{c}(\mathcal{T}') - \mathbf{c}(\mathcal{T}) = |R_i| + (h_{\mathcal{T}}(x) - h_{\mathcal{T}}(y) + 1)\,|R_{i+1}| - \|\mathcal{B}\|.$$

This is not possible, for the same reasons as in the proof of Lemma 6. □

Lemma 8. *Let $\mathcal{R} = (R_1, \ldots, R_m)$ be a sequence of runs and let (R_i, R_{i+1}) be a minimal pair of $\mathcal{R}$. The following inequality holds:*

$$\boldsymbol{Opt}(\mathcal{R}) \geq \boldsymbol{Opt}(\mathcal{R}_{[i]}) + \frac{|R_i| + |R_{i+1}|}{2}. \tag{1}$$

Proof. If there is an optimal strategy for which R_i and R_{i+1} are siblings, then its cost is $\mathbf{Opt}(\mathcal{R}) = \mathbf{Opt}(\mathcal{R}_{[i]}) + |R_i| + |R_{i+1}|$, and therefore Equation (1) holds trivially in this case.

We now prove that Equation (1) also holds when there are no optimal strategy for which R_i and R_{i+1} are siblings. In this case, let $\mathcal{T}$ be any optimal merging tree for $\mathcal{R}$. By Lemma 6 and Lemma 7, R_i is the right child of its father x, R_{i+1} is the left child of its father y, and $h_{\mathcal{T}}(x) = h_{\mathcal{T}}(y)$: we have $\overset{x}{\underset{\mathcal{A}\ R_i}{/\backslash}}$ and $\overset{y}{\underset{R_{i+1}\ \mathcal{B}}{/\backslash}}$ in $\mathcal{T}$. Assume by symmetry that $|R_i| \geq |R_{i_+1}|$. Consider the tree $\mathcal{T}'$ obtained from $\mathcal{T}$ by changing the node R_i into $\overset{\bullet}{\underset{R_i\ R_{i+1}}{/\backslash}}$ and y into $\mathcal{B}$. The tree $\mathcal{T}'$ is still a merging tree of $\mathcal{R}$. Moreover, by construction of $\mathcal{T}'$ and Lemma 1, we have

$$\mathbf{c}(\mathcal{T}') - \mathbf{c}(\mathcal{T}) = |R_i| + (h_{\mathcal{T}}(x) - h_{\mathcal{T}}(y) + 1)\,|R_{i+1}| - \|\mathcal{B}\| = |R_i| + |R_{i+1}| - \|\mathcal{B}\|.$$

But $\mathcal{T}'$ is a merging tree for $\mathcal{R}$, in which R_i and R_{i+1} are siblings. We therefore have $\mathbf{c}(\mathcal{T}') \geq \mathbf{Opt}(\mathcal{R}_{[i]}) + |R_i| + |R_{i+1}|$. Thus,

$$\mathbf{c}(\mathcal{T}) + |R_i| + |R_{i+1}| - \|\mathcal{B}\| \geq \mathbf{Opt}(\mathcal{R}_{[i]}) + |R_i| + |R_{i+1}|.$$

Hence, as $\mathbf{c}(\mathcal{T}) = \mathbf{Opt}(\mathcal{R})$, we have $\mathbf{Opt}(\mathcal{R}) \geq \mathbf{Opt}(\mathcal{R}_{[i]}) + \|\mathcal{B}\|$. We assumed that $|R_i| \geq |R_{i_+1}|$. Moreover, the minimality of (R_i, R_{i+1}) ensures that $|R_i| \leq |R_{i+2}|$. Since R_{i+2} is a leaf of $\mathcal{B}$, this yields

$$\|\mathcal{B}\| \geq |R_{i+2}| \geq \max\big(|R_i|, |R_{i_+1}|\big) \geq \frac{|R_i| + |R_{i+1}|}{2},$$

which concludes the proof. □

The proof of Theorem 2 is a direct consequence of Lemma 8. Indeed if we follow GreedyMergeSort, we start with $\mathcal{R}^{(0)} = \mathcal{R}$ then select $i_1 \in [m-1]$ such that $(R^{(0)}_{i_1}, R^{(0)}_{i_1+1})$ is a minimal pair and merge these two runs to produce $\mathcal{R}^{(1)} = \mathcal{R}^{(0)}_{[i_1]}$. And we repeat this construction until we reach $\mathcal{R}^{(m-1)}$ which contains exactly one run: at step $k < m$, we select a minimal pair $(R^{(k-1)}_{i_k}, R^{(k-1)}_{i_k+1})$ from $\mathcal{R}^{(k-1)}$, and build the new sequence of runs $\mathcal{R}^{(k)} = \mathcal{R}^{(k-1)}_{[i_k]}$. The cost of each step is $|R^{(k-1)}_{i_k}| + |R^{(k-1)}_{i_k+1}|$.

Let $(i_1, \dots, i_{m-1})$ be the indices chosen during the execution of GreedyMergeSort in the description above. By Lemma 8 we have

$$\begin{aligned}
\mathbf{Opt}(\mathcal{R}^{(0)}) &\geq \mathbf{Opt}(\mathcal{R}^{(1)}) + \frac{|R^{(0)}_{i_1}| + |R^{(0)}_{i_1+1}|}{2} \\
&\geq \mathbf{Opt}(\mathcal{R}^{(2)}) + \frac{|R^{(1)}_{i_2}| + |R^{(1)}_{i_2+1}|}{2} + \frac{|R^{(0)}_{i_1}| + |R^{(0)}_{i_1+1}|}{2} \\
&\geq \dots \\
&\geq \frac{1}{2} \sum_{k=1}^{m-1} \left(|R^{(k-1)}_{i_k}| + |R^{(k-1)}_{i_k+1}| \right).
\end{aligned}$$

This concludes the proof as by construction, $\sum_{k=1}^{m-1} \left(|R^{(k-1)}_{i_k}| + |R^{(k-1)}_{i_k+1}| \right)$ is exactly the cost of the merging tree associated with GreedyMergeSort. □

▶ Proof of Proposition 1

First, since GreedyMergeSort always performs at least as many comparisons as the other two algorithms, it is sufficient to prove the statement for GreedyMergeSort.

Let $\mathcal{R} = (R_1, \dots, R_m)$ be a run decomposition of S with $m \geq 2$ (if $m = 1$, no comparison is performed, hence the result). Consider the set $E(\mathcal{R}) = \{(R_{2i+1}, R_{2i+2}) : \ 0 \leq i \leq \lfloor \frac{m}{2} \rfloor - 1\}$. Since the run lengths sum to n, if we sum the length of all pairs of $E(\mathcal{R})$, the result is at most n. Hence, the pair of minimal sum (R_{2j+1}, R_{2j+2}) of E is such that

$$|R_{2j+1}| + |R_{2j+2}| \leq \frac{n}{\lfloor \frac{m}{2} \rfloor} \leq \frac{2n}{m-1},$$

since $\lfloor \frac{m}{2} \rfloor \geq \frac{m-1}{2}$. Hence, if (R_i, R_{i+1}) is a minimal pair of $\mathcal{R}$, then

$$|R_i| + |R_{i+1}| \leq |R_{2j+1}| + |R_{2j+2}| \leq \frac{2n}{m-1}.$$

This yields that when performing the first step of GreedyMergeSort, we go from $\mathcal{R}$ to $\mathcal{R}_{[i]}$ at a cost bounded from above by $\frac{2n}{m-1}$. Observe that the resulting sequence of runs is made of $m-1$ runs whose lengths sum to n. Hence, a direct induction yields that the total number of comparisons performed by GreedyMergeSort is bounded from above by

$$\frac{2n}{m-1} + \frac{2n}{m-2} + \dots + \frac{2n}{1} = 2n \sum_{k=1}^{m-1} \frac{1}{k} = \mathcal{O}(n \log m).$$

This concludes the proof. □

Algorithm 6: merge_collapse translated

```
R ← run decomposition of S
X ← ∅
while R ≠ ∅ do
    R ← pop(R)
    Append R to X
    while True do
        if |X| ≤ |Y| + |Z| or |W| ≤ |X| + |Y| then
            if |X| < |Z| then  Merge X and Y
            else  Merge Y and Z
        else if |Y| ≤ |Z| then
            Merge Y and Z
        else
            break
```

▶ Proof that the strategy of Section 5.2 is equivalent to SimplifiedTimSort

Algorithm 5 is the code found in Python for the main loop of TimSort. We "translate" it into pseudo-code, using the convention that $\mathcal{X} = (x_1, \cdots, x_{\ell-4}, W, X, Y, Z)$. Hence W, X, Y, and Z are the four topmost elements of the stack, Z being on top. The result is given in Algorithm 6.
Important: whenever a predicate in a `if` conditional uses a variable that is not available because the stack is too small, for instance $|W| \leq |X| + |Y|$ for a stack of size 3, we consider that this predicate is false. This convention is useful to avoid many tests on the length of the stack.

As Algorithm 6 is not exactly in the form of pairs (rules,merge), we rewrite the algorithm as depicted in Algorithm 7. The rules and merge strategies are, in order:

$$\begin{array}{ll} \rho_1 := |X| \geq |Z| & \mu_1 = \textbf{merge}(X, Y) \\ \rho_2 := |X| > |Y| + |Z| & \mu_2 = \textbf{merge}(Y, Z) \\ \rho_3 := |W| > |X| + |Y| & \mu_3 = \textbf{merge}(Y, Z) \\ \rho_4 := |Y| > |Z| & \mu_4 = \textbf{merge}(Y, Z) \end{array}$$

Lemma 9. *Algorithm 6 and Algorithm 7 are equivalent: for given sequence of runs $\mathcal{R}$, they are both characterized by the same merge tree.*

Proof. Straightforward. Just check that the conditions that leads to the merge of X and Y, Y and Z or no merge are equivalent. □

▶ Proof of Lemma 4

As already stated, the case of SimplifiedTimSort was proved in [7].

Let us consider α-StackSort. The proof is done by induction on the iteration t of the while loop. The result holds trivially for $t = 1$, since at the first iteration, the stack is empty. Assume it holds at iteration t and consider iteration $t+1$. During the $(t+1)$-th iteration, we append a new run R at the end of the stack. If $|x_\ell| > \alpha\,|R|$ then we are done. Otherwise, the inner while loop merge the two rightmost runs Y and Z until they verify $|Y| > \alpha|Z|$. As the condition is satisfied everywhere before by induction hypothesis, this ensures that the condition is satisfied everywhere. □

Algorithm 7: SIMPLIFIEDTIMSORT (S, n)

```
1  R ← run decomposition of S
2  X ← ∅
3  while R ≠ ∅ do
4      R ← pop(R)
5      Append R to X
6      while X violates at least one rule of 𝔖 do
7          if |X| < |Z| then                                  /* ρ1 is activated */
8              Merge X and Y
9          else if |X| ≤ |Y| + |Z| then                       /* ρ2 is activated */
10             Merge Y and Z
11         else if |W| ≤ |X| + |Y| then                       /* ρ3 is activated */
12             Merge Y and Z
13         else if |Y| ≤ |Z| then                             /* ρ4 is activated */
14             Merge Y and Z
```

► Proof of Lemma 5

It is sufficient to prove the lemma at the beginning of each iteration of the while loop of line 3 in Algorithm 2, as the stack can have at most one more element, when it is inserted at line 5.

For SIMPLIFIEDTIMSORT, Lemma 4 ensures that if the stack is $\mathcal{X} = (x_1, \cdots, x_\ell)$, then $|x_i| > |x_{i+1}|+|x_{i+2}|$ for every $i \in [\ell - 2]$. Moreover $|x_{\ell-1}| > |x_\ell|$. Hence, by direct induction, for every $i \in [\ell]$, $|x_{\ell-i+1}| \geq |x_\ell|\, F_i$, where F_i is the i-th Fibonacci number. As $F_n \geq c\phi^n$ for $n \geq 1$ and some well chosen c, we have

$$\sum_{i=1}^{\ell} |x_i| \geq |x_\ell| \sum_{i=1}^{\ell} c\phi^i \geq c\,\frac{\phi^\ell - 1}{\phi - 1} = \Omega(\phi^\ell).$$

Since the sum of the run lengths is at most n, we get that $\ell = \mathcal{O}(\log n)$.

The proof for α-STACKSORT is similar, α playing the role of ϕ. □

► Proof of Theorem 4 for α-StackSort

We start with α-STACKSORT. The run decomposition uses only $n-1$ comparisons. To analyze the complexity of the while loop of line 3, we rely on a classical technique used for amortized complexity and rewrite this part of the algorithm as in Algorithm 8. In blue have been added some computation on a variable C. Observe first that C is decreased at line 9 every time a merge is done, by an amount equal to the cost of this merge.

We now prove that after each blue instruction (Line 2, Line 6 and Line 9), we have, if the current stack is $\mathcal{X} = (x_1, \ldots, x_\ell)$,

$$C \geq \sum_{i=1}^{\ell} (1+\alpha) i |x_i| \tag{2}$$

We prove this property by induction: It clearly holds after Line 2. Observe that any time the stack is altered, C is updated immediately after. Hence we just have to prove that if the property holds before an alteration, then it still holds when C is updated just after:

- For Lines 5-6: if $\mathcal{X} = (x_1, \ldots, x_\ell)$ before Line 5, then $\mathcal{X} = (x_1, \ldots, x_\ell, R)$ after Line 6. By induction hypothesis $C \geq \sum_{i=1}^{\ell}(1+\alpha)i|x_i|$ before Line 5, and it is increased by $(1+\alpha)(\ell+1)|R|$ at Line 6. Therefore the property still holds after Line 6.

Algorithm 8: Main loop of α-StackSort

```
1 𝒳 ← ∅
2 C ← 0
3 while ℛ ≠ ∅ do
4     R ← pop(ℛ)
5     Append R to 𝒳
6     C ← C + (1 + α) |𝒳| |R|                    /* used for the proof only */
7     while 𝒳 violates the rule |Y| ≥ α |Z| do
8         Merge Y and Z
9         C ← C − (|Y| + |Z| − 1)                 /* used for the proof only */
```

– For Lines 8-9: if $\mathcal{X} = (x_1, \ldots, x_{\ell-2}, Y, Z)$ before Line 8, then after Line 9 the stack is

$$\mathcal{X} = (x_1, \ldots, x_{\ell-2}, \mathbf{merge}(Y, Z)).$$

By induction hypothesis, before Line 8 we have

$$\begin{aligned} C &\geq \sum_{i=1}^{\ell-2}(1+\alpha)i|x_i| + (1+\alpha)(\ell-1)|Y| + (1+\alpha)\ell|Z| \\ &\geq \sum_{i=1}^{\ell-2}(1+\alpha)i|x_i| + (1+\alpha)(\ell-1)(|Y|+|Z|) + (1+\alpha)|Z|. \end{aligned}$$

But we are in the case where the rule is activated, hence $|Y| < \alpha\,|Z|$. Thus $(1+\alpha)|Z| > |Y| + |Z| > |Y| + |Z| - 1$. This yields

$$C - (|Y| + |Z| - 1) \geq \sum_{i=1}^{\ell-2}(1+\alpha)i|x_i| + (1+\alpha)(\ell-1)(|Y|+|Z|).$$

Hence, the property still holds after Line 9.

The quantity C is increased on Line 6 only. By Lemma 5, when a new run R is added in $\mathcal{X}$, C is increased by at most $K \log n\,|R|$, for some positive constant K. Hence, the sum of all increases of C is bounded from above by $K \log n \sum_{R\in\mathcal{R}} |R| = \mathcal{O}(n \log n)$. The quantity C is decreased whenever a merge is performed, by an amount equal to this merge cost. As we just proved that Equation (2) always holds after an update of C, C is non-negative at the end of this part of the algorithm. Hence, the total number of comparisons performed in this part is $\mathcal{O}(n \log n)$.

The last while loop of Algorithm 2 also performs at most $\mathcal{O}(n \log n)$ comparisons, as the stack is of length $\mathcal{O}(\log n)$: every run is involved in at most $\mathcal{O}(\log n)$ merges during this loop.

▶ Proof of Theorem 4 for SimplifiedTimSort

We want to proceed for SimplifiedTimSort as for α-StackSort, but there are some technical difficulties inherent to the structure of the rules in SimplifiedTimSort. We still define a variable C initialized with 0 and which is increased by $3iR$ whenever a run R arrive at position i on the stack. We still remove $|R|+|R'|-1$ from C whenever R and R' are merged. However, we cannot directly guarantee that C is always positive; for some cases we need to consider several consecutive merges made by the algorithm in order to conclude. Hence, we will unroll the main while loop as needed, to obtain an algorithm equivalent to the main while

loop, but that is bigger. On this redundant code we then prove that at the end of any iteration, if the stack is $\mathcal{X} = (x_1, \ldots, x_\ell)$ then

$$C \geq \sum_{i=1}^{\ell} 3i|x_i|. \tag{3}$$

We first establish three lemmas, which give hints of what happens, in certain cases, during two consecutive iterations of the while loop.

Lemma 10. *If rule ρ_2 is activated, then rule ρ_4 is violated at the next iteration of the while loop.*

Proof. If rule ρ_2 is activated, then the runs Y and Z are merged. The new stack is $\mathcal{X}' = (x_1, \ldots, x_{\ell-3}, Y', Z')$ with $Y' = X$ and $Z' = \mathbf{merge}(Y, Z)$. Since ρ_2 is violated, we have $|X| \leq |Y| + |Z|$, thus $|Y'| \leq |Z'|$ which is the negation of ρ_4. □

Lemma 11. *If rule ρ_3 is activated, then the rule ρ_2 is violated at the next iteration.*

Proof. If rule ρ_3 is activated, then the runs Y and Z are merged. The new stack is $\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, X', Y', Z')$ with $X' = W$, $Y' = X$, and $Z' = \mathbf{merge}(Y, Z)$. Since ρ_3 is violated, we have $|W| \leq |X|+|Y| \leq |X|+|Y|+|Z|$. Thus $|X'| \leq |Y'| + |Z'|$, and therefore ρ_2 is violated at the next iteration. □

Using this lemmas, we rewrite Algorithm 7 by unrolling some loops. More precisely, we obtained Algorithm 9 page 23 the following way:

- if ρ_1 is activated, then we merge X and Y and we are done.
- if ρ_2 is activated, then we merge Y and Z: the stack is now $\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, W, X, \mathbf{merge}(Y, Z))$. We unroll the loop once since Lemma 10 ensures that another rule is violated after the merge. We check whether ρ_1 is violated, if not we are sure that either ρ_2, ρ_3 or ρ_4 is not satisfied; in every of these cases X and $\mathbf{merge}(Y, Z)$ are merged. We just have to be careful to use $\mathcal{X}'$ for writing the nested conditions. For instance the nested condition for ρ_1 Line 13 is $|X'| < |Z'|$, for $\mathcal{X}' = (\ldots, X', Y', Z')$, which rewrites $|W| < |Y| + |Z|$.
- if ρ_3 is activated, then we merge Y and Z: the stack is now $\mathcal{X}' = (x_1, \ldots, x_{\ell-5}, V, W, X, \mathbf{merge}(Y, Z))$. By Lemma 11, we know that ρ_2 is violated on next iteration. Hence, we unroll once. The nested test for ρ_1 is done as previously. If ρ_1 is satisfied we know that ρ_2 is activated and then the stack is now $\mathcal{X}'' = (x_1, \ldots, x_{\ell-5}, V, W, \mathbf{merge}(X, Y, Z))$. We unroll once more (that is, three nested `if`), using the properties ensured when ρ_2 is activated, as before.
- if ρ_4 is activated, then we merge Y and Z and do not unroll the loop.

In this complicated version of SimplifiedTimSort, which is strictly equivalent to SimplifiedTimSort, we removed from C the costs of the merges that have been performed. What remains to prove, as we did for α-StackSort, is that Equation (3) holds after each update of C. This is done by induction. As C is always decreased just before ending an iteration of the main loop, we assume the property holds at the beginning of the `while` loop, and verify that it still holds when C is updated. There are seven cases, which we detail in the following.

For a given stack configuration $\mathcal{X} = (x_1, \ldots, x_\ell)$, let $f(\mathcal{X}) = \sum_{i \in [\ell]} 3i|x_i|$. By induction hypothesis, we assume that at the beginning of an iteration of the main loop, $C \geq f(\mathcal{X})$. We now check for the different cases, which is tedious but straightforward. $\mathbf{merge}(X, Y, Z)$ denote the result of merging X, Y and Z.

- **Line 10:** the stack goes from $\mathcal{X} = (x_1, \ldots, x_{\ell-3}, X, Y, Z)$ to $\mathcal{X}' = (x_1, \ldots, x_{\ell-3}, \mathbf{merge}(X, Y), Z)$. Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|Y| + 3|Z|$. As rule ρ_1 is violated in this case, we have $|X| < |Z|$. Thus, the cost paid at Line 10 satisfies $|X| + |Y| - 1 < |Y| + |Z| < 3|Y| + 3|Z|$. Hence, the property still holds after Line 10.
- **Line 15:** $\mathcal{X} = (x_1, \ldots, x_{\ell-4}, W, X, Y, Z)$ becomes $\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, \mathbf{merge}(W, X), \mathbf{merge}(Y, Z))$. Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|X| + 3|Y| + 6|Z|$. As $|W| < |Y| + |Z|$ in this case, the cost paid at Line 15 satisfies $|W| + |X| + |Y| + |Z| - 2 < |X| + 2|Y| + 2|Z| < 3|X| + 3|Y| + 6|Z|$. Hence, the property still holds after Line 15.

- **Line 18:** $\mathcal{X} = (x_1, \ldots, x_{\ell-4}, W, X, Y, Z)$ becomes $\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, W, \mathbf{merge}(X, Y, Z))$. Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|Y| + 6|Z|$. As $|X| \leq |Y| + |Z|$ in this case, the cost paid at Line 18 satisfies $|X| + 2|Y| + 2|Z| - 2 < 3|Y| + 3|Z| < 3|Y| + 6|Z|$. Hence, the property still holds after Line 18.
- **Line 23:** This is exactly the same as for Line 15.
- **Line 28:** $\mathcal{X} = (x_1, \ldots, x_{\ell-5}, V, W, X, Y, Z)$ becomes $\mathcal{X}' = (x_1, \ldots, x_{\ell-5}, \mathbf{merge}(V, W), \mathbf{merge}(X, Y, Z))$. Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|W| + 3|X| + 6|Y| + 9|Z|$. As $|V| < |X| + |Y| + |Z|$ in this case, the cost paid at Line 28 satisfies $|V| + |W| + |X| + 2|Y| + 2|Z| - 3 < |W| + 2|X| + 3|Y| + 3|Z| < 3|W| + 3|X| + 6|Y| + 9|Z|$. Hence, the property still holds after Line 28.
- **Line 31:** $\mathcal{X} = (x_1, \ldots, x_{\ell-5}, V, W, X, Y, Z)$ becomes $\mathcal{X}' = (x_1, \ldots, x_{\ell-5}, V, \mathbf{merge}(W, X, Y, Z))$. Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|X| + 6|Y| + 9|Z|$. As $|W| \leq |X| + |Y|$ in this case, the cost paid at Line 31 satisfies $|W| + 2|X| + 3|Y| + 3|Z| - 3 < 3|X| + 4|Y| + 3|Z| < 3|X| + 6|Y| + 9|Z|$. Hence, the property still holds after Line 31.
- **Line 34:** $\mathcal{X} = (x_1, \ldots, x_{\ell-2}, Y, Z)$ becomes $\mathcal{X}' = (x_1, \ldots, x_{\ell-2}, \mathbf{merge}(Y, Z))$. Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|Z|$. As $|Y| \leq |Z|$ in this case, the cost paid at Line 34 satisfies $|Y| + |Z| - 1 < 2|Z| < 3|Z|$. Hence, the property still holds after Line 34.

We conclude as for α-StackSort: Equation (3) ensures that $C \geq 0$ when SimplifiedTimSort halts. Moreover, the sum of all increases of C is $\mathcal{O}(n \log n)$ and the number of comparisons is at most the sum of all decreases of C. Also, as for α-StackSort, the last stage of the algorithm where the remaining runs are merged, is $\mathcal{O}(n \log n)$. $\square$

Algorithm 9: Main loop of unrolled-SimplifiedTimSort

```
X ← ∅
C ← 0
while R ≠ ∅ do
    R ← pop(R)
    Append R to X
    C ← C + 3 |X| |R|
    while X violates at least one rule of 𝔖 do
        if |X| < |Z| then                                   /* ρ1 is activated */
            Merge X and Y
            C ← C − (|X| + |Y| − 1)
        else if |X| ≤ |Y| + |Z| then                        /* ρ2 is activated */
            Merge Y and Z
            if |W| < |Y| + |Z| then                         /* ρ1 is activated */
                Merge W and X
                C ← C − (|W| + |X| + |Y| + |Z| − 2)
            else                                            /* ρ2, ρ3 or ρ4 is activated */
                Merge X and merge(Y, Z)
                C ← C − (|X| + 2|Y| + 2|Z| − 2)
        else if |W| ≤ |X| + |Y| then                        /* ρ3 is activated */
            Merge Y and Z
            if |W| < |Y| + |Z| then                         /* ρ1 is activated */
                Merge W and X
                C ← C − (|W| + |X| + |Y| + |Z| − 2)
            else                                            /* ρ2 is activated */
                Merge X and merge(Y, Z)
                if |V| < |X| + |Y| + |Z| then               /* ρ1 is activated */
                    Merge V and W
                    C ← C − (|V| + |W| + |X| + 2|Y| + 2|Z| − 3)
                else                                        /* ρ2, ρ3 or ρ4 is activated */
                    Merge W and merge(X, merge(Y, Z))
                    C ← C − (|W| + 2|X| + 3|Y| + 3|Z| − 3)
        else if |Y| ≤ |Z| then                              /* ρ4 is activated */
            Merge Y and Z
            C ← C − (|Y| + |Z| − 1)
```