Adaptive Shivers Sort: An Alternative Sorting Algorithm*

VINCENT JUGÉ, Univ. Gustave Eiffel, LIGM (UMR 8049), CNRS, France

We present a new sorting algorithm, called adaptive ShiversSort, that exploits the existence of monotonic runs for sorting efficiently partially sorted data. This algorithm is a variant of the well-known algorithm TimSort, which is the sorting algorithm used in standard libraries of programming languages such as Python or Java (for non-primitive types). More precisely, adaptive ShiversSort is a so-called k-aware merge-sort algorithm, a class that captures "TimSort-like" algorithms and that was introduced by Buss and Knop.

In this article, we prove that, although adaptive ShiversSort is simple to implement and differs only slightly from TimSort, its computational cost, in number of comparisons performed, is optimal within the class of *natural* merge-sort algorithms, up to a small additive linear term. This makes adaptive ShiversSort the first k-aware algorithm to benefit from this property, which is also a 33% improvement over TimSort's worst-case. This suggests that adaptive ShiversSort could be a strong contender for being used instead of TimSort.

Then, we investigate the optimality of k-aware algorithms. We give lower and upper bounds on the best approximation factors of such algorithms, compared to optimal stable natural merge-sort algorithms. In particular, we design generalisations of adaptive ShiversSort whose computational costs are optimal up to arbitrarily small multiplicative factors.

CCS Concepts: • **Theory of computation → Sorting and searching**.

ACM Reference Format:
Vincent Jugé. 2024. Adaptive Shivers Sort: An Alternative Sorting Algorithm. *ACM Trans. Algor.* 1, 1 (May 2024), 71 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*An extended abstract of this paper appeared in the proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA 2020).

Author's address: Vincent Jugé, Univ. Gustave Eiffel, LIGM (UMR 8049), CNRS, Marne-la-Vallée, France, vincent.juge@univ-eiffel.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

1 INTRODUCTION

The problem of sorting data has been one of the first and most extensively studied problems in computer science, and sorting is ubiquitous, due to its use as a sub-routine in a wealth of various algorithms. Hence, as early as the 1940s, sorting algorithms were invented, which enjoyed many optimality properties regarding their complexity in time (and, more precisely, in number of comparisons or element moves required) as well as in memory. Every decade or so, a new major sorting algorithm was invented, either using a different approach to sorting or adapting specifically tuned data structures to improve previous algorithms: MergeSort [10], QuickSort [12], HeapSort [23], SmoothSort [7], SplaySort [17], ...

In 2002, Tim Peters, a software engineer, created a new sorting algorithm, which was called TimSort [19]. This algorithm immediately demonstrated its efficiency for sorting actual data, and was adopted as the standard sorting algorithm in core libraries of wide-spread programming languages such as Python and Java. Hence, the prominence of such a custom-made algorithm over previously preferred *optimal* algorithms contributed to the regain of interest in the study of sorting algorithms.

Understanding the reasons behind the success of TimSort is still an ongoing task. These reasons include the fact that TimSort is well adapted to the architecture of computers (e.g., for dealing with cache issues) and to realistic distributions of data. In particular, a model that successfully explains why TimSort is adapted to sorting realistic data involves *run decompositions* [3, 8], as illustrated in Figure 1. Such decompositions were already used in Knuth's NaturalMergeSort [15], which predated TimSort and adapted the traditional MergeSort algorithm as follows: NaturalMergeSort is based on splitting arrays into monotonic subsequences, also called *runs*, and on merging these runs. Thus, all algorithms sharing this feature of NaturalMergeSort are also called *natural* merge sorts.

$$S = (\ \underbrace{12, 7, 6, 5,}_{\text{first run}}\ \underbrace{5, 7, 14, 36,}_{\text{second run}}\ \underbrace{3, 3, 5, 21, 21,}_{\text{third run}}\ \underbrace{20, 8, 5, 1}_{\text{fourth run}}\ )$$

Fig. 1. A sequence and its *run decomposition* computed by a greedy algorithm: for each run, the first two elements determine if it is non-decreasing or decreasing, and then the run continues with the maximum number of consecutive elements that preserves the monotonicity.

In addition to being a natural merge sort, TimSort also includes many optimisations, which were carefully engineered, through extensive testing, to offer the best complexity performances. As a result, the general structure of TimSort can be split into three main components: (i) a complicated variant of an insertion sort, which is used to deal with *small* runs (e.g., runs of length less than 32), (ii) a simple policy for choosing which *large* runs to merge, (iii) a complex

sub-routine for merging these runs. The first and third components were those which were the most finely tuned, hence understanding the subtleties of why they are efficient and how they could be improved seems difficult. The second component, however, is quite simple, and therefore it offers the best opportunities for modifying and improving TimSort.

Context and related work. The success of TimSort has nurtured the interest in the quest for sorting algorithms that would be adapted to arrays with few runs. However, the *ad hoc* conception of TimSort made its complexity analysis less easy than what one might have hoped, and it is only in 2015, a decade after TimSort had been largely deployed, that Auger et al. proved that TimSort required $O(n \log(n))$ comparisons for sorting arrays of length n [2].

Even worse, because of the lack of a systematic and theoretical analysis of this algorithm, several bugs were discovered only recently in both Python and Java implementations of TimSort [1, 6].

Meanwhile, since TimSort was invented, several natural merge sorts have been proposed, all of which were meant to offer easy-to-prove complexity guarantees. Such algorithms include ShiversSort, introduced by Shivers in [20], as well as Takaoka's MinimalSort [21] (equivalent constructions of this algorithm were also obtained in [3]), Buss and Knop's α-MergeSort [5], and the most recent algorithms PeekSort and PowerSort, due to Munro and Wild [18]. Alternatively, as we will mention again, algorithms for constructing *optimal binary search trees*, such as the algorithms of Hu-Tucker [13] and Garsia-Wachs [9], can be adapted to provide natural merge sorts as well. We call MinimalStableSort the merge sort derived from adapting either algorithm.

These algorithms share most of the nice properties of TimSort, as summarised in Table 1 (columns 1–3). For instance, except MinimalSort, these are *stable* algorithms, which means that they sort repeated elements in the same order as these elements appear in the input. This is very important for merge sorts, because only adjacent runs will be merged, which allows merging

Table 1. Properties of a few natural merge sorts. The constant c_α is such that $1.04 < c_\alpha < 1.09$. The *merge cost* of an algorithm is an upper bound on its number of comparisons and element moves.

Algorithm	Time complexity	Stable	k-aware	Worst-case merge cost
NaturalMergeSort	$O(n + n\log(\rho))$	✓	✗	$n\log_2(\rho) + O(n)$
TimSort	$O(n + n\mathcal{H})$	✓	$k = 4$	$3/2\, n\mathcal{H} + O(n)$
ShiversSort	$O(n\log(n))$	✓	$k = 2$	$n\log_2(n) + O(n)$
MinimalSort	$O(n + n\mathcal{H})$	✗	✗	$n\mathcal{H} + O(n)$
MinimalStableSort	$O(n + n\mathcal{H})$	✓	✗	$n\mathcal{H} + O(n)$
α-MergeSort	$O(n + n\mathcal{H})$	✓	$k = 3$	$c_\alpha n\mathcal{H} + O(n)$
PowerSort	$O(n + n\mathcal{H})$	✓	✗	$n\mathcal{H} + O(n)$
PeekSort	$O(n + n\mathcal{H})$	✓	✗	$n\mathcal{H} + O(n)$
Adaptive ShiversSort	$O(n + n\mathcal{H})$	✓	$k = 3$	$n\mathcal{H} + O(n)$

directly arrays instead of having to use linked lists. This feature is also important for merging composite types (e.g., non-primitive types in Java), which might be sorted twice according to distinct comparison measures. Moreover, all these algorithms sort arrays of length n in time $O(n \log(n))$, and, for all of them except ShiversSort, they even do it in time $O(n + n \log(\rho))$, where ρ is the number of runs of the array. This is optimal in the model of sorting by comparisons [16], using the classical counting argument for lower bounds.

Some of these algorithms even adapt to the lengths of the runs, and not only to the number of runs: if the array consists of ρ runs of lengths $r_1, \dots, r_\rho$, these algorithms run in $O(n + n\mathcal{H})$, where $\mathcal{H}$ is defined as $\mathcal{H} = H(r_1/n, \dots, r_\rho/n)$ and $H(x_1, \dots, x_\rho) = -\sum_{i=1}^{\rho} x_i \log_2(x_i)$ is the general entropy function. Considering the number of runs and their lengths as parameters, this finer upper bound is again optimal in the model of sorting by comparisons [3].

Focusing only on the time complexity, six algorithms seem on par with each other, and finer complexity evaluations are required to separate them. Except TimSort, it turns out that these algorithms are, in fact, described only as policies for merging runs, the actual sub-routine used for merging runs being left implicit. Therefore, we settle for the following cost model.

Since naive merging algorithms approximately require $m + n$ element comparisons and element moves for merging two arrays of lengths m and n, and since $m + n$ element moves may be needed in the worst case (for any values of m and n), we measure below the complexity in terms of *merge cost* [2, 5, 11, 18]: the cost of merging two runs of lengths m and n is defined as $m + n$, and we identify the complexity of an algorithm with the sum of the costs of the merges processed while applying the run merge policy of this algorithm.

Of course, this identification can be legitimate only if this sum of merge costs dominates the complexity of deciding which runs should be merged. Fortunately, this is the case in all of the algorithms presented in Table 1. Indeed, for all algorithms presented except MinimalSort and MinimalStableSort, deciding which runs to merge can be done in time $O(n)$. Algorithms MinimalSort and MinimalStableSort have more complicated merge policies, and deciding which runs these algorithms shall merge amounts to computing a Huffman tree (respectively, an optimal binary search tree) with ρ leaves, which can be done in time $O(\rho \log(\rho))$, and therefore in time $O(n\mathcal{H})$ too.

In this new model, every run merge policy can be identified with a bottom-up construction algorithm for binary search trees, an idea that was already noted and used successfully in [3, 18]. In particular, one can prove that the merge cost of any natural merge sort must be at least $n\mathcal{H} + O(n)$. This makes MinimalSort, MinimalStableSort, PeekSort and PowerSort the only sorting algorithms with an optimal merge cost, as shown in the last column of Table 1.

In another direction, and since MinimalStableSort, PeekSort and PowerSort are stable, they could be considered natural options for succeeding TimSort as standard sorting algorithm in

Python or Java. Nevertheless, and although the latter two algorithms have implementations similar to that of TimSort, their merge policies are slightly more complicated, as illustrated in Section 2.

Therefore, there is yet to find a natural merge sort whose structure would be as simple as and extremely close to that of TimSort, and whose merge cost would also be optimal up to an additive term $O(n)$.

A first step towards this goal is using an adequate notion of "TimSort-likeness", and therefore we look at the class of k-aware sorting algorithms. This class of algorithms was invented by Buss and Knop [5], with the explicit goal of characterising those algorithms whose merge policy is similar to that of TimSort. More precisely, TimSort is based on discovering runs on the fly, and "storing" these runs into a stack: if a run spans the i^{th} to j^{th} entries of the array, the stack will contain the pair (i, j). Then, TimSort merges only runs that lie on the top of the stack, and such decisions are based only on the lengths of these top runs.

The rationale behind this process is that processing runs in such a way should be adapted to the architecture of computers, for instance by avoiding cache misses. One says that a natural merge sort is k-aware if deciding which runs should be merged is based only on the lengths of the top k runs of the stack, and if the runs merged belong themselves to these top k runs. The 4^{th} column of Table 1 indicates which algorithms are k-aware for some $k < +\infty$, in which case it also gives the smallest such k.

Focusing on k-aware algorithms seems all the more relevant because some of the nice features of TimSort were also due to the high degree of tuning of the components (i) and (iii). Hence, if one does not modify these components, and if one follows a merge policy that behaves in a way similar to that of TimSort, one may reasonably hope that those nice features of TimSort would be kept intact, even though their causes are not exactly understood. This suggests identifying natural merge sorts with their merge policy, and integrating the components (i) and (iii) later.

Contributions. We propose a new natural merge sort, which we call adaptive ShiversSort. As advertised above, we will identify this algorithm with its run merge policy. Adaptive ShiversSort is a blend between the algorithms TimSort and ShiversSort; the purpose being to borrow nice properties from both algorithms. As a result, the merge policy of adaptive ShiversSort is extremely similar to that of TimSort, which means that switching from one algorithm to the other should be essentially costless, since it would require changing only a dozen lines in the code of Java.

Adaptive ShiversSort is a 3-aware algorithm, which is stable and enjoys an optimal $n\mathcal{H} + O(n)$ upper bound on its merge cost. Hence, adaptive ShiversSort appears as optimal with respect to all the criteria mentioned in Table 1; it is the first known k-aware algorithm with a merge cost of $n\mathcal{H} + O(n)$, thereby answering a question left open by Buss and Knop in [5]. Moreover, and due to its simple policy, the running time complexity proof of adaptive ShiversSort is simple as

well. Below, we propose a short, self-contained version of this proof.

Then, still aiming to compare PeekSort, PowerSort and adaptive ShiversSort, we investigate their *best-case* merge costs. It turns out that the merge costs of PowerSort and of PeekSort, in every case, are bounded between $n\mathcal{H}$ and $n(\mathcal{H}+2)$, these bounds being both close and optimal for any stable merge sort. Similarly, the merge cost of adaptive ShiversSort is only bounded from above by $n(\mathcal{H}+\Delta)$, where $\Delta = 24/5 - \log_2(5) \approx 2.478$, which is also slightly worse than PowerSort. Hence, we design a variant of adaptive ShiversSort, called length-adaptive ShiversSort, which is not 3-aware, but whose merge cost also enjoys an $n(\mathcal{H}+2)$ upper bound.

Finally, we further explore the question, raised by Buss and Knop in [5], of the optimality of k-aware algorithms on *all* arrays. More precisely, this question can be stated as follows: for a given integer k and a real number $\varepsilon > 0$, does there exist a k-aware algorithm whose merge cost is at most $1+\varepsilon$ times the merge cost of any stable merge cost on any array? We prove that the answer is always negative when $k = 2$; for all $k \geqslant 3$, the set of numbers ε for which the answer is positive forms an interval with no upper bound, and we prove that the lower bound of that interval is a positive real number, which tends to 0 when k grows arbitrarily.

Detailed content summary. Section 2 contains brief descriptions of several natural merge sort algorithms mentioned earlier in this introduction. We begin with c-adaptive ShiversSort, a parametrised algorithm that generalises both adaptive ShiversSort and length-adaptive ShiversSort, and present the main properties that explain its efficiency. Subsequent subsections include descriptions of TimSort; α-StackSort and α-MergeSort; ShiversSort; augmented ShiversSort; PowerSort and PeekSort; and the original version of adaptive ShiversSort presented in [14].

Section 3 is devoted to the worst-case analysis of c-adaptive ShiversSort. It culminates with the following two results.

Theorem 5. *The merge costs of adaptive ShiversSort and of length-adaptive ShiversSort are bounded from above by $n(\mathcal{H}+3)$ and by $n(\mathcal{H}+2)$, respectively.*

Theorem 15. *For every value of the parameter c, the merge cost of c-adaptive ShiversSort is bounded from above by $n\,(\mathcal{H}+\Delta)$, where $\Delta = 24/5 - \log_2(5) \approx 2.478$.*

Section 4 is an *intermezzo*, where we study lower bounds on the best- and worst-case merge costs of all natural merge sort algorithms. Although unconventional, the study of the best-case merge cost is meaningful here, because it turns out to be extremely close to the worst-case merge costs of algorithms like adaptive ShiversSort or PowerSort.

Section 5 is focused on studying the entire family of *(k,ℓ)-aware* and *length-(k,ℓ)-aware* algorithms; the former family was proposed by Buss and Knop [5], and is a subset of the latter family, which also contains algorithms like PowerSort and length-adaptive ShiversSort. More

precisely, we wish to find how close to optimal such algorithms can be: an algorithm is ε-optimal if there is no input array on which its merge cost exceeds $1 + \varepsilon$ times the smallest cost of a natural merge-sort algorithm. Table 2 gather our results about the values of ε for some algorithms listed in Section 2. It is followed by three inapproximability and approximability results.

Proposition 29. *Let $k \geqslant 3$ be an integer, and let*

$$\theta_k = 1/((10k + 12)\log_2(2k + 2)).$$

No k-aware sorting algorithm is θ_k-optimal.

Proposition 30. *Let $k \geqslant 3$ be an integer, and let*

$$\varepsilon_k = 1/2^{k+7}.$$

No length-(∞, k)-aware sorting algorithm is ε_k-optimal.

Theorem 33. *Let $k \geqslant 8$ be an integer, and let*

$$\eta_k = (\Delta + 7)/\log_2((k - 3)/4),$$

where we recall that $\Delta = 24/5 - \log_2(5)$. There exists a k-aware sorting algorithm that is η_k-optimal.

Finally, Section 6 is focused on implementation details. It includes an easy analysis of the space complexity of adaptive ShiversSort, as well as suggestions for painlessly switching from TimSort to adaptive ShiversSort or length-adaptive ShiversSort in Python and Java.

Reading itineraries. Given the length of this article and the variety of results listed just above, various readers may prefer different reading strategies. We strongly advise the reader who just discovers this article to first skip the most technical results and proofs, and to focus on Section 2, which gives a nice, yet non-exhaustive overview of various algorithms that may exist; on Table 2, which sums up the results of these algorithms on various input arrays; and on Section 6, in which we discuss actual (and sometimes low-level) implementation details.

Those readers who wish to understand why Theorems 5 and 15 are valid are then invited to read their proofs, which can be found in Sections 3.1 and 3.2, respectively. Although one might be tempted to jump directly to Section 3.2, because Theorem 15 gives us a better complexity bound for adaptive ShiversSort than Theorem 5, we recommend starting with Section 3.1. Indeed, the reader will later find that the proof of Theorem 15, which relies on a carefully crafted *ad hoc* notion of potential, is substantially more technical and difficult, although conceptually not so demanding.

Those who wonder why Theorems 5 and 15 brand adaptive ShiversSort and length-adaptive ShiversSort as excellent merge sort algorithms should definitely read Section 4: this short section, whose content is easy to comprehend, simply indicates that no natural merge sort algorithm,

even in the best case, performs substantially fewer comparisons than adaptive ShiversSort and length-adaptive ShiversSort do in the worst case.

Finally, when having to choose between algorithms such as TimSort, PowerSort or adaptive ShiversSort, the reader might wish to discover *the* ultimate natural merge sort algorithm, maybe restricting this this quest to (k, ℓ)-aware or length-(k, ℓ)-aware algorithms, for implementation reasons. In such a case, the reader should read Section 5, in which it is proved that no such ultimate algorithm exists.

On the one hand, Section 5.1 contains proofs that, once k and ℓ are fixed, length-(k, ℓ)-aware algorithms cannot be ε-optimal for arbitrarily small values of ε. By contrast, Section 5.2 proves that an explicit $2^{O(1/\varepsilon)}$-aware parametrised algorithm is ε-optimal. Thus, the reader may be content with knowing which parameter values make the algorithm ε-optimal, or dive into the step-by-step proof of Theorem 33; in the latter case, we strongly recommend reading Sections 5.2.1 to 5.2.4 one at a time.

2 ADAPTIVE SHIVERSSORT AND RELATED ALGORITHMS

In this section, we describe the run merge policy of the algorithm adaptive ShiversSort and of related algorithms, most of which were mentioned in Section 1.

Here and in subsequent sections, we will always use the following notations. Below, the length of a run R is denoted by r and the integer $\lfloor \log_2(r/c) \rfloor$ is denoted by ℓ. The integer ℓ will be called the *level* of the run R. We adapt readily these notations when the name of the run considered varies, e.g., the length of the run R' is denoted by r' and the integer $\lfloor \log_2(r'/c) \rfloor$ is denoted by ℓ'. In particular, we will commonly note the stack $(R_1, \ldots, R_h)$, where R_k is the k^{th} deepest run of the stack; therefore, the R_h is the top element of the stack and is easy to access, whereas accessing R_1 is much less straightforward. The length of R_k is then denoted by r_k, and we set $\ell_k = \lfloor \log_2(r_k/c) \rfloor$.

2.1 Adaptive ShiversSort

The merge policy of adaptive ShiversSort is depicted in Algorithm 1. For the ease of Section 3.1, and because it does not make any proof harder, we shall consider adaptive ShiversSort as a special case of the parametrised algorithm c-adaptive ShiversSort: in addition to the array to sort, c-adaptive ShiversSort also requires a positive integer c as a parameter.

In subsequent sections, we will consider most specifically two choices for the parameter c: we may either set $c = 1$, thereby obtaining the algorithm adaptive ShiversSort itself, or $c = n + 1$, where n is the length of the array to be sorted, thereby obtaining the algorithm length-adaptive ShiversSort. Except in Section 3.1, where we run a complexity analysis for generic values of the parameter c (thus encompassing both cases $c = 1$ and $c = n + 1$ at once), we will only focus on the algorithm adaptive ShiversSort, i.e., we will set $c = 1$.

Algorithm 1: c-adaptive ShiversSort

Input : Array A to sort, integer parameter c
Result: The array A is sorted into a single run. That run remains on the stack.
Note: Whenever two consecutive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, by the run resulting from the merge. In practice, in $\mathcal{S}$, each run is represented by a pair of pointers to its first and last entries.

```
1  runs ← the run decomposition of A
2  S ← an empty stack
3  while true:                                              ▷ main loop
4      if h ⩾ 3 and ℓ_{h−2} ⩽ max{ℓ_{h−1}, ℓ_h}:
5          merge the runs R_{h−2} and R_{h−1}
6      else if runs ≠ ∅:
7          remove a run R from runs and push R onto S
8      else:
9          break
10 while h ⩾ 2:
11     merge the runs R_{h−1} and R_h
```

One reason for introducing this parameter is as follows. Although choosing $c = 1$ is the most natural choice in general, the resulting algorithm adaptive ShiversSort is *not* scale-invariant: if we triple the size of every run, the merges that adaptive ShiversSort will perform may change. This would not be the case if we had used an optimal sorting algorithm. Relating the parameter c to the length n is a way to recover scale invariance, as will be proved in Section 6.2.

This algorithm is based on discovering monotonic runs and on maintaining a stack of such runs, which may be merged or pushed onto the stack according to whether $\ell_{h-2} \leqslant \max\{\ell_{h-1}, \ell_h\}$. In particular, since this inequality only refers to the values of ℓ_{h-2}, ℓ_{h-1} and ℓ_h, and since only the runs R_{h-2}, R_{h-1} and R_h may be merged, this algorithm falls within the class of 3-aware stable sorting algorithms such as described by Buss and Knop [5] as soon as the value of c does not depend on the input. This is the case of adaptive ShiversSort, but not of other variants such as length-adaptive ShiversSort.

Let us now present briefly some related algorithms. Like adaptive ShiversSort, these algorithms all rely on discovering and maintaining runs in a stack, although their merge policies follow different rules. In fact, each of these policies is obtained by modifying the *main loop* of adaptive ShiversSort. In addition, they may share most of all of the three properties of adaptive ShiversSort:

(1) with the possible exception of the few top runs, the sequence $(r_i)_{i \leqslant h}$ of the lengths of those runs stored in the stack should decrease at exponential speed; this is the *exponential decay* property; it is typically achieved by maintaining some invariant (the fact that $r_i \geqslant r_{i+1} + r_{i+2}$ for all $i \leqslant h - 2$);

(2) when pushing a new run R onto that stack, one should postpone merging it until we can merge it with runs of size approximately equal to R; this is the *late merging* property;
(3) deciding whether two runs should be merged should be done by comparing their levels, instead of comparing directly their lengths; this is the *level-driven merging* property.

The rational of using these three properties is as follows. exponential decay ensures that merging the top runs of the stack (except those top few runs we decided should not be concerned by the exponential decay), say $R_i, R_{i+1}, \dots, R_j$, is not much more expensive than just merging the two runs R_i and R_{i+1}: the runs $R_{i+2}, R_{i+3}, \dots, R_j$ are somehow harmless.

Then, one should definitely avoid merging a new run R, which we just pushed onto a stack $R_1, R_2, \dots, R_h$, if r is much larger than both r_{h-1} and r_h: instead of merging R with R_h and then, presumably, with R_{h-1}, we should rather start by merging R_{h-1} with R_h, and then decide whether the resulting run is worth merging with R_{h-2} or with R itself.

Finally, level-driven merging turns out to be an adequate approximation of the ideal situation where every merge would be perfectly balanced, i.e., one would not merge two runs r and r' unless $r = r'$. Demanding length equality is too much, and we must relax this requirement, for example by merging two runs if they have the same level.

Then, ensuring that the sequence of levels $(\ell_i)_{i \leqslant h}$ is decreasing already ensures the exponential decay property. This makes the urge to merge successive runs R and R' such that $\ell \leqslant \ell'$ even more compelling. If, in addition, one manages to merge runs R and R' only if their levels ℓ and ℓ' are equal to each other, then the resulting run R'' will be of level $\ell'' = \ell + 1$. Thus, when sorting an array of length n, each run R of length r could undergo at most $\lceil \log_2(n/r) \rceil$ such merges before being merged into a run of length n, concluding the algorithm.

Of course, one cannot always ensure that runs R and R' are merged only if their levels are equal to each other. This is the case, for instance if the array contains two runs of lengths 2 and 1. However, it might remain possible to make sure that *most* of the merges will concern runs with equal levels.

2.2 TimSort

The first algorithm we present is TimSort, and is due to Peters [19]. Its main loop is presented in Algorithm 2. This algorithm enjoys the exponential decay and late merging properties, which entail the $O(n + n\mathcal{H})$ worst-case merge cost mentioned in the introduction.

Original versions of TimSort missed the test on lines 10–11, which made the invariant invalid and caused several implementation bugs [1, 6]. Nevertheless, in the full version of [1], Auger et al. proved that these flawed versions of the algorithm miraculously managed to enjoy a $O(n + n\mathcal{H})$ worst-case merge cost. Alas, even in the corrected version of TimSort, the constant hidden in

Algorithm 2: TimSort main loop

```
3  while true:                                                    ▷ main loop
4      if h ⩾ 3 and r_{h-2} < r_h:
5          merge the runs R_{h-2} and R_{h-1}
6      else if h ⩾ 2 and r_{h-1} ⩽ r_h:
7          merge the runs R_{h-1} and R_h
8      else if h ⩾ 3 and r_{h-2} ⩽ r_{h-1} + r_h:
9          merge the runs R_{h-1} and R_h
10     else if h ⩾ 4 and r_{h-3} ⩽ r_{h-2} + r_{h-1}:
11         merge the runs R_{h-1} and R_h
12     else if runs ≠ ∅:
13         remove a run R from runs and push R onto S
14     else:
15         break
```

the O is rather high, as outlined by the following result.

Theorem 1. *The worst-case merge cost of TimSort on inputs of length n is bounded from above by $3/2\, n\mathcal{H} + O(n)$ and bounded from below by $3/2\, n \log_2(n) + O(n)$.*

Proof. The lower bound is proved in [5]. The upper bound is proved in the full version of [1]. □

Hence, and in order to lower the constant from $3/2$ to 1, it was important to look for other merge policies, by keeping the two prominent features of TimSort mentioned above.

2.3 α-StackSort and α-MergeSort

The above considerations led to the invention of the two algorithms α-StackSort [2], which integrates the first feature only, and α-MergeSort [5], which integrates both features.

Like adaptive ShiversSort, these are parametric algorithms, which require a parameter $\alpha > 1$ and, since their structures are very similar, we present both algorithms at once. α-StackSort uses the following main loop:

Algorithm 3: α-StackSort main loop

```
3  while true:                                                    ▷ main loop
4      if h ⩾ 2 and r_{h-1} < α r_h:
5          merge the runs R_{h-1} and R_h
6      else if runs ≠ ∅:
7          remove a run R from runs and push R onto S
8      else:
9          break
```

This sorting algorithm uses a main loop that is quite simpler than that of TimSort, which made it easier to study. However, in spite of enjoying the exponential decay property, it misses the late merging property, and therefore it is *not* adaptive to the number of runs nor to their lengths. We shall see in Section 5.1 that this is the fate of all 2-aware sorting algorithms like α-StackSort.

This problem was circumvented by the 3-aware algorithm α-MergeSort, whose main loop is slightly longer:

Algorithm 4: α-MergeSort main loop

```
3  while true:                                                    ▷ main loop
4      if h ⩾ 3 and r_{h-2} < r_h:
5          merge the runs R_{h-2} and R_{h-1}
6      else if h ⩾ 2 and r_{h-1} < α r_h:
7          merge the runs R_{h-1} and R_h
8      else if h ⩾ 3 and r_{h-2} < α r_{h-1}:
9          merge the runs R_{h-1} and R_h
10     else if runs ≠ ∅:
11         remove a run R from runs and push R onto S
12     else:
13         break
```

However, evaluating precisely the worst-case merge cost of α-MergeSort is challenging. Buss and Knop proved [5] that this merge cost is $c_\alpha n\mathcal{H} + O(n)$ when $\phi < \alpha < 2$, where $\phi = (1+\sqrt{5})/2$ is the Golden ratio, and where

$$c_\alpha = \frac{\alpha + 1}{(\alpha+1)\log_2(\alpha+1) - \alpha\log_2(\alpha)} > 1.$$

When $1 < \alpha \leqslant \phi$, however, it is only known that the worst-case merge cost of α-MergeSort is *at least* $c_\alpha n\mathcal{H} + O(n)$: the precise constant in the $O(n + n\mathcal{H})$ upper bound on the time complexity of α-MergeSort is unknown.

Yet, these partial results already confirm that no value of the parameter $\alpha > 1$ would let α-MergeSort have a worst-case merge cost of $n\mathcal{H} + O(n)$.

2.4 ShiversSort

The examples of α-MergeSort and TimSort suggest that enjoying the exponential decay and late merging properties is enough to design an algorithm with a $O(n + n\mathcal{H})$ complexity, but not necessarily with a $n\mathcal{H} + O(n)$ merge cost. Toward achieving this latter goal, one hope comes from the algorithm ShiversSort, which was invented by Shivers [20]. ShiversSort is obtained by slightly simplifying the tests carried in the main loop of adaptive ShiversSort and merging the runs R_{h-1}

and R_h instead of R_{h-2} and R_{h-1}. More precisely, ShiversSort uses the following main loop (and implicitly uses a parameter $c = 1$):

Algorithm 5: ShiversSort main loop

```
while true:                                                    ▷ main loop
    if h ⩾ 2 and ℓ_h ⩾ ℓ_{h−1}:
        merge the runs R_{h−1} and R_h
    else if runs ≠ ∅:
        remove a run R from runs and push R onto S
    else:
        break
```

Like α-StackSort, this algorithm is 2-aware; thus, it misses the late merging property and fails to be adaptive to the number of runs or to their lengths. Nevertheless, it still enjoys the nice property of having a worst-case merge cost that is optimal up to an additive linear term, when the only complexity parameter is n.

Theorem 2. *The worst-case merge cost of ShiversSort on inputs of length n that decompose into ρ monotonic runs is both bounded from above by $n \log_2(n) + O(n)$ and bounded from below by $\omega(n \log_2(\rho))$.*

This result was proved in [5, 20]. Its proof, which we omit here, is very similar to our own analysis of adaptive ShiversSort in Section 3 below. A crucial element of both proofs, as will be stated in Lemma 7, is the fact that the sequence $\ell_1, \ell_2, \ldots, \ell_{h-k}$ shall always be decreasing, for some small integer k: we have $k = 1$ in the proof of [5], and $k = 2$ in Lemma 7. In essence, this invariant is similar to that of TimSort, but it allows decreasing the associated constant hidden in the O notation from $3/2$ to 1.

2.5 Augmented ShiversSort

The very idea of integrating exponential decay, late merging and level-driven merging properties from TimSort and ShiversSort led Buss and Knop to invent the algorithm augmented ShiversSort [5]. Its run merge policy is motivated as follows. When the levels of the two top runs R_{h-1} and R_h obey the inequality $\ell_{h-1} \leqslant \ell_h$, one should merge them, with one possible exception: if R_{h-2} is significantly smaller than R_h, one should merge the runs R_{h-2} and R_{h-1} instead. This is in line with the fact that augmented ShiversSort should be 3-aware instead of only 2-aware.

Based on this motivation, the algorithm augmented ShiversSort is obtained by using the following main loop (like ShiversSort, it implicitly uses a parameter $c = 1$):

Algorithm 6: augmented ShiversSort main loop

```
3  while true:                                                    ▷ main loop
4      if h ⩾ 3, r_{h−2} ⩽ r_h and ℓ_{h−1} ⩽ ℓ_h:
5          merge the runs R_{h−2} and R_{h−1}
6      else if h ⩾ 2 and ℓ_{h−1} ⩽ ℓ_h:
7          merge the runs R_{h−1} and R_h
8      else if runs ≠ ∅:
9          remove a run R from runs and push R onto S
10     else:
11         break
```

The hope here is that both avoiding merging newly pushed runs while they are too large and maintaining an invariant (on the integers ℓ_i) similar to that of ShiversSort would make augmented ShiversSort very efficient. Unfortunately, this algorithm suffers from the same design flaw as the original version of TimSort, and the desired invariant is *not* maintained. Even worse, the effects of not maintaining this invariant are much more severe here, as underlined by the following result.

Theorem 3. *The worst-case merge cost of augmented ShiversSort on inputs of length n is $\Theta(n^2)$.*

Proof. Consider some integer $k \geqslant 1$, and let $n = 8k$ and $\rho = 2k$. Let also $(r_1, \dots, r_\rho)$ be the sequence of run lengths defined by $r_{2i-1} = 6$ and $r_{2i} = 2$ for all $i \leqslant k$. Note that $r_1 + \dots + r_\rho = n$.

Now, let us apply the algorithm augmented ShiversSort on an array of n integers that splits into increasing runs $R_1, \dots, R_\rho$ of lengths exactly $r_1, \dots, r_\rho$. One verifies quickly that the algorithm performs successively the following operations:

(i) push the runs R_1 and R_2;
(ii) for all $i \in \{2, \dots, k\}$, push the run R_{2i-1}, then merge the runs R_{2i-2} and R_{2i-3}, and push the run R_{2i};
(iii) keep merging the last two runs on the stack (line 11): we first merge the runs $R_{\rho-1}$ and R_ρ, and then, the $(m+1)^{\text{th}}$ such merge involves runs of sizes 8 and $8m$.

Therefore, the merge cost of augmented ShiversSort on that array is

$$\mathsf{mc} = \sum_{i=1}^{k} (r_{2i-1} + r_{2i}) + \sum_{m=1}^{k-1} 8(m+1) = n^2/16 + 3n/2 - 8.$$

Conversely, in any (natural or not) merge sort, any element can be merged at most $n-1$ times, and therefore the total merge cost of such a sorting algorithm is at most $n(n-1)$. □

2.6 PowerSort and PeekSort

As mentioned in Section 1, PowerSort and PeekSort enjoy excellent complexity guarantees, with merge costs bounded from above by $n(\mathcal{H} + 2)$. However, these are not k-aware algorithms for any k, and their merge policy is more complicated than that of TimSort.

Unlike TimSort and other k-aware algorithms, PeekSort is better described by adopting a top-down point of view. When PeekSort merges two runs R and R' that resulted from merging runs $R_i, R_{i+1}, \ldots, R_{j-1}$ and $R_j, R_{j+1}, \ldots, R_{k-1}$ respectively, the lengths r and r' shall be as close to each other as possible, i.e., the quantity $|(r_i + r_{i+1} + \ldots + r_{x-1}) - (r_x + r_{x+1} + \ldots + r_{k-1})|$ is minimized when choosing $x = j$. To this day, the merge policy of PeekSort does not seem to have any TimSort-like bottom-up description.

The algorithm PowerSort follows a similar intuition, as described by its authors [18]: when a (created or original) run still need to be merged d times, the positions it spans should be a good approximation of an interval of the form $[kn/2^d, (k+1)n/2^d]$, thereby making every merge somewhat balanced. However, and unlike its sibling PeekSort, the algorithm PowerSort also admits a bottom-up description that makes it quite close to being a 3-aware algorithm. Yet, and due to that intuition, whether PowerSort should merge two consecutive runs R and R' cannot depend only on their lengths. Indeed, it also depends on the *powers* of these runs, which are defined as follows.

Let R be a run resulting from the merge of several (original) runs $R_i, \ldots, R_j$, let pos be the last position contained in the run R, and let n be the length of the array to be sorted. The power of R is defined as the least integer p such that $\lfloor 2^p(2\,\mathrm{pos} - r_j)/n \rfloor < \lfloor 2^p(2\,\mathrm{pos} + r_{j+1})/n \rfloor$. Then, two consecutive runs R and R' should be merged if $p > p'$.

Thus, the merge decisions of PowerSort are based not only on the lengths of the runs, but also on their positions within the array. This prevents PowerSort from being a k-aware algorithm *per se*. However, in practice, allowing the algorithm to base its decisions on the length of the array and on the positions of the runs, which are already stored in the stack used to represent runs, is often harmless. Consequently, in Section 5, we will generalise the notion of k-awareness to include algorithms like PowerSort.

2.7 Alternative constructions and variants

Like augmented ShiversSort, the algorithm adaptive ShiversSort just consists in a new attempt to integrate the three properties of TimSort and ShiversSort: exponential decay, late merging and level-driven merging. Although this attempt was eventually successful, one might have made different choices. Here, we present two such choices.

A first choice consists in the following variant of TimSort's main loop, which we decide to call level-TimSort; it can be used for every value of the parameter c:

Algorithm 7: Level-TimSort main loop

```
3  while true:                                                    ▷ main loop
4      if h ⩾ 3 and ℓ_{h−2} ⩽ ℓ_h:
5          merge the runs R_{h−2} and R_{h−1}
6      else if h ⩾ 2 and ℓ_{h−1} ⩽ ℓ_h:
7          merge the runs R_{h−1} and R_h
8      else if h ⩾ 3 and ℓ_{h−2} ⩽ ℓ_{h−1}:
9          merge the runs R_{h−1} and R_h
10     else if runs ≠ ∅:
11         remove a run R from runs and push R onto S
12     else:
13         break
```

This variant is obtained as follows. Aiming to enjoy the level-driven merging property prescribes replacing every comparison $r_i < r_j$ or $r_i \leqslant r_j$ between run lengths by a comparison $\ell_i \leqslant \ell_j$ between run levels. Similarly, using the approximation $\log_2(r_j + r_k) \approx \max\{\log_2(r_j), \log_2(r_k)\}$, it would be meaningful to replace every comparison $r_i \leqslant r_j + r_k$ between run lengths by a comparison $\ell_i \leqslant \max\{\ell_j, \ell_k\}$ between levels. Doing so and removing unnecessary tests results in level-TimSort. This variant unsurprisingly enjoys a $O(n + n\mathcal{H})$ time complexity and, by mimicking the study of c-adaptive ShiversSort that we conduct below, we might actually prove that its worst-case merge cost is also $n\mathcal{H} + O(n)$.

A second variant may come from the surprise that, both in TimSort and in that first variant, line 9 consists in merging the runs R_{h-1} and R_h although that line was triggered by the fact that R_{h-2} is small: either $r_{h-2} \leqslant r_{h-1} + r_h$ or $\ell_{h-2} \leqslant \ell_{h-1}$. Thus, it is tempting to decide, in that case, that one should merge the runs R_{h-2} and R_{h-1}. As a result, we obtain the following main loop:

Algorithm 8: adaptive ShiversSort main loop – Original version

```
3  while true:                                                    ▷ main loop
4      if h ⩾ 3 and ℓ_{h−2} ⩽ ℓ_h:
5          merge the runs R_{h−2} and R_{h−1}
6      else if h ⩾ 2 and ℓ_{h−1} ⩽ ℓ_h:
7          merge the runs R_{h−1} and R_h
8      else if h ⩾ 3 and ℓ_{h−2} ⩽ ℓ_{h−1}:
9          merge the runs R_{h−2} and R_{h−1}
10     else if runs ≠ ∅:
11         remove a run R from runs and push R onto S
12     else:
13         break
```

Of course, one might also decide to exchange the lines 6–7 and 8–9, which triggered the same merge in TimSort and in the previous algorithm, but trigger now two distinct merges. However, this decision would change nothing, because having failed the test ($h \geqslant 3$ and $\ell_{h-2} \leqslant \ell_h$) already ensures that one cannot pass both tests ($h \geqslant 2$ and $\ell_{h-1} \leqslant \ell_h$) and ($h \geqslant 3$ and $\ell_{h-2} \leqslant \ell_{h-1}$).

This second variant was actually the original version of c-adaptive ShiversSort presented in [14]. We will prove in Section 6.1 that Algorithms 1 and 8 perform the same merges, in the same order, and therefore that both versions have the same merge cost.

In this article, we preferred the current version of c-adaptive ShiversSort, both because its code is simpler and because it makes the analysis carried out in Sections 3 and 5 also simpler.

3 WORST-CASE ANALYSIS OF ADAPTIVE SHIVERSSORT

We stated in the introduction that adaptive ShiversSort enjoys excellent worst-case upper bounds in terms of merge cost. We prove that statement in this section, which is subdivided in two independent parts.

Section 3.1 is devoted to proving Theorem 5, which contains simple yet already excellent upper bounds on the merge costs of adaptive ShiversSort and of length-adaptive ShiversSort. Section 3.2 then consists in proving Theorem 15, which contains an even better upper bound on the merge cost of adaptive ShiversSort. However, the proof of this latter result is less intuitive and more technical than that of Theorem 5, which is why we decided to present it only later in Section 3.

3.1 A first upper bound

Proposition 4. *For every value of the parameter c, the merge cost of c-adaptive ShiversSort is bounded from above by $n(\mathcal{H} + 3 - \{\log_2(n/c)\}) - \rho - 1$, where $\{x\} = x - \lfloor x \rfloor$ denotes the fractional part of the real number x.*

The upper bound provided in Proposition 4 is slightly complicated. Hence, and in particular for $c = 1$ and $c = n + 1$, it can readily be replaced by the following upper bounds, which depend only on n and $\mathcal{H}$.

Theorem 5. *The merge costs of adaptive ShiversSort and of length-adaptive ShiversSort are bounded from above by $n(\mathcal{H} + 3)$ and by $n(\mathcal{H} + 2)$, respectively.*

Proof. Proposition 4 states that adaptive ShiversSort has a merge cost

$$\mathrm{mc} \leqslant n(\mathcal{H} + 3 - \{\log_2(n)\}) - \rho - 1 \leqslant n(\mathcal{H} + 3).$$

It also states that length-adaptive ShiversSort has a merge cost

$$\mathrm{mc} \leqslant n(\mathcal{H} + 3 - \{\log_2(n/(n+1))\}) - \rho - 1.$$

Observing that $\{\log_2(n/(n+1))\} = \log_2(n/(n+1)) - \lfloor \log_2(n/(n+1)) \rfloor = 1 - \log_2(1+1/n) \geqslant 1 - 2/n$, it follows that $\mathrm{mc} \leqslant n(\mathcal{H}+2) - \rho + 1 \leqslant n(\mathcal{H}+2)$. □

In what follows, we fix the value of the parameter c once and for all, and we aim at proving Proposition 4. We first prove two auxiliary results about the levels of the runs manipulated throughout the algorithm.

Lemma 6. *When two runs R and R' are merged into a single run R'', we have $\ell'' \leqslant \max\{\ell, \ell'\} + 1$.*

Proof. Without loss of generality, we assume that $r \leqslant r'$. In that case,

$$2^{\ell''}c \leqslant r'' = r + r' \leqslant 2r' < 2 \times 2^{\ell'+1}c = 2^{\ell'+2}c,$$

and therefore $\ell'' \leqslant \ell' + 1$. □

Lemma 7. *At any time during the main loop of the algorithm c-adaptive ShiversSort, if the run stack is $\mathcal{S} = (R_1, \ldots, R_h)$, we have:*

$$\ell_1 > \ell_2 > \ldots > \ell_{h-3} > \max\{\ell_{h-2}, \ell_{h-1}\}. \tag{1}$$

If, furthermore, $\mathcal{S}$ results from a merge between two runs, then $\ell_{h-2} \geqslant \ell_{h-1}$.

Proof. The proof is done by induction. First, if $h \leqslant 2$, there is nothing to prove: this case occurs, in particular, when the algorithm starts. Now, consider some stack $\mathcal{S} = (R_1, \ldots, R_h)$ that satisfies (1) and is updated into a new stack $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$, either by merging the runs R_{h-2} and R_{h-1}, or by pushing the run $\overline{R}_{\overline{h}}$:

- If the runs R_{h-2} and R_{h-1} were just merged, then $\overline{h} = h - 1$ and $\overline{R}_i = R_i$ for all $i \leqslant h-3$. Thus, the inequalities $\ell_1 > \ell_2 > \ldots > \ell_{h-3}$ immediately rewrite as $\overline{\ell}_1 > \overline{\ell}_2 > \ldots > \overline{\ell}_{\overline{h}-2}$. Meanwhile, $\overline{R}_{\overline{h}-1}$ results from the merge between R_{h-2} and R_{h-1}, and therefore Lemma 6 proves that $\overline{\ell}_{\overline{h}-1} \leqslant \max\{\ell_{h-2}, \ell_{h-1}\} + 1 < \ell_{h-3} + 1$. It follows that $\overline{\ell}_{\overline{h}-2} = \ell_{h-3} \geqslant \overline{\ell}_{\overline{h}-1}$.
- If the run $\overline{R}_{\overline{h}}$ was just pushed, then $\overline{h} = h + 1$ and $\overline{R}_i = R_i$ for all $i \leqslant h$. Thus, the inequalities $\ell_1 > \ell_2 > \ldots > \ell_{h-2}$ already rewrite as $\overline{\ell}_1 > \overline{\ell}_2 > \ldots > \overline{\ell}_{\overline{h}-3}$. Furthermore, since c-adaptive ShiversSort triggered a push operation instead of a merge operation, it must be the case that $\overline{\ell}_{\overline{h}-3} = \ell_{h-2} > \max\{\ell_{h-1}, \ell_h\} = \max\{\overline{\ell}_{\overline{h}-2}, \overline{\ell}_{\overline{h}-1}\}$.

In both cases, the stack $\overline{\mathcal{S}}$ also satisfies (1), which completes the induction. □

Roughly speaking, Lemma 7 states that the lengths of the runs stored in the stack increase at exponential speed (when we start from the top of the stack), with the possible exception of the top run, whose length we have no control over. As suggested in Section 2, this property was already crucial in the complexity proofs of several algorithms such as ShiversSort, TimSort or α-MergeSort.

In addition to these results, we will also need the following technical lemma.

Lemma 8. *For all real numbers x such that $0 \leqslant x \leqslant 1$, we have $2^{1-x} \leqslant 2 - x$.*

Proof. Every function of the form $x \mapsto \exp(tx)$, where t is a fixed real parameter, is convex. Therefore, the function $f : x \mapsto 2^{1-x} - (2 - x)$ is convex too. It follows, for all $x \in [0, 1]$, that $f(x) \leqslant \max\{f(0), f(1)\} = 0$, which completes the proof. □

The proof of Proposition 4 consists now in a careful estimation of the total cost of those merges performed by the algorithm. Intuitively, this proof may be seen as a cost allocation, where each merge between two runs R and R' should be paid for by some run (which may be either R, R', or some other run). In practice, however, assigning the entire cost of a merge to a single run may be too crude for our needs. Therefore, it will be convenient to split the cost of a merge in two parts that will be paid for by different runs.

Hence, below, we artificially split the merge between R and R' into two separate operations, which we call *half-merges*: the half-merge of R with R', for a cost of r, and the half-merge of R' with R, for a cost of r'. Together, these operations indeed consist in merging R and R' with each other. Their costs add up to $r + r'$, as expected and, in what follows, they may be allocated to distinct cost centres.

Then, when merging the run R with a run R' into one bigger run R'', we say that the half-merge of R is *expanding* if $\ell'' \geqslant \ell + 1$, and is *non-expanding* otherwise. Note that, if $\ell \leqslant \ell'$, the half-merge of R with R' is necessarily expanding. Consequently, when two runs R and R' are merged with each other, either the half-merge of R or of R' is expanding. In particular, if $\ell = \ell'$, then both half-merges of R and of R' must be expanding. Hence, we say that the merge between R and R' is *intrinsically expanding* if $\ell = \ell'$.

We first show that, up to a linear term, the announced merge cost is entirely due to expanding merges. To do so, we use the notation $\lambda = \{\log_2(r/c)\}$, where we recall that $\{x\} = x - \lfloor x \rfloor$ denotes the fractional part of x; in other words, $\lambda = \log_2(r/c) - \ell$. Similarly, we set $\lambda' = \{\log_2(r'/c)\} = \log_2(r'/c) - \ell'$ and $\lambda_i = \{\log_2(r_i/c)\} = \log_2(r_i/c) - \ell_i$.

Lemma 9. *The total cost of expanding half-merges is at most $n(\mathcal{H} - \{\log_2(n/c)\}) + \Lambda$, where Λ is defined as $\Lambda = \sum_{i=1}^{\rho} r_i \lambda_i$.*

Proof. While the algorithm is performed, the elements of a run R of initial length r may take part in at most

$$\lfloor \log_2(n/c) \rfloor - \ell = (\log_2(n/c) - \{\log_2(n/c)\}) - (\log_2(r/c) - \lambda) = \log_2(n/r) + \lambda - \{\log_2(n/c)\}$$

expanding half-merges. Consequently, if the array is initially split into runs of lengths $r_1, \dots, r_\rho$, the total cost of expanding half-merges is at most

$$\sum_{i=1}^{\rho} r_i(\log_2(n/r_i) + \lambda_i - \{\log_2(n/c)\}) = n(\mathcal{H} - \{\log_2(n/c)\}) + \Lambda. \qquad \square$$

It remains to prove that the total cost of non-expanding half-merges is at most $3n - \Lambda - \rho - 1$. This requires classifying merges based on the conditions that triggered them, and *binding* some merges to a single run, as follows.

Definition 10. Let $\mathcal{S} = (R_1, \dots, R_h)$ be a stack of runs such that $h \geqslant 3$ and $\ell_{h-2} \leqslant \max\{\ell_{h-1}, \ell_h\}$. By construction, when encountering the stack $\mathcal{S}$ during its main loop, the algorithm c-adaptive ShiversSort performs a merge operation m between the runs R_{h-2} and R_{h-1}. We say that m is

(i) a #1-*merge* if $\ell_{h-2} < \ell_{h-1}$; in that case, we *bind* m to the run R_{h-1};
(ii) a #2-*merge* if $\ell_{h-1} \leqslant \ell_{h-2} \leqslant \ell_h$; in that case, we bind m to the run R_h;
(iii) a #3-*merge* if $\ell_h < \ell_{h-2} = \ell_{h-1}$; in that case, we do not bind m to any run.

We also say that a half-merge m' is a #k-half-merge if the merge m to which m' belongs is a #k-merge. Furthermore, if m is bound to a run R and if m' is non-expanding, we also bind m' to the run R. If m is not bound to any run or if m' is expanding, we do not bind m' to any run.

Observe that the conditions for being a #1-merge, a #2-merge or a #3-merge are mutually exclusive, and that every merge belongs to one of these three classes. By extension, we may also refer to a #4-push operation, so that each update is a #k-update for some $k \leqslant 4$.

Our decision to bind some merges to a given run is motivated as follows. By construction, every #3-merge is intrinsically expanding. Therefore, every non-expanding half-merge m is bound to a run R, and we choose to allocate the cost of m to R. Intuitively, R is the run that was so large that it triggered m: had R been substantially smaller, the half-merge m might not have occurred. Moreover, as we prove below, the run R will necessarily be a run from the original array (i.e., R has not yet been merged before m occurs). These are the main reasons why, among others, we made the otherwise surprising choice to bind every #2-merge to a run R_h that does not even actively participate to the merge.

Lemma 11. *No merge is immediately followed by a #1-merge, and no #3-merge is immediately followed by a #2-merge.*

Proof. Let m be a merge. Let $\mathcal{S} = (R_1, \dots, R_h)$ and $\overline{\mathcal{S}} = (\overline{R}_1, \dots, \overline{R}_{\overline{h}})$ denote the stack just before the merge and just after the merge, respectively, so that $\overline{h} = h - 1$. If $\overline{h} \leqslant 2$, then m cannot be followed by any merge, hence we assume that $\overline{h} \geqslant 3$.

First, Lemma 7 already states that $\overline{\ell}_{\overline{h}-2} \geqslant \overline{\ell}_{\overline{h}-1}$, and therefore m cannot be followed by a #1-merge. Second, if m is a #3-merge, we must have $\ell_{h-1} = \ell_{h-2} > \ell_h$. Since $\overline{R}_{\overline{h}} = R_h$ and $\overline{R}_{\overline{h}-1}$ results from merging the runs R_{h-2} and R_{h-1}, it follows that $\overline{\ell}_{\overline{h}-1} \geqslant \ell_{h-1} > \ell_h = \overline{\ell}_{\overline{h}}$, which shows that m cannot be followed by a #2-merge. □

Lemma 12. *If a push is immediately preceded by a #3-merge, it cannot be immediately followed by a #1-merge.*

Proof. Let p be a push update preceded by a #3-merge m. Let $\mathcal{S} = (R_1, \ldots, R_h)$ denote the stack before m occurs, and $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ the stack after p occurs, so that $\overline{h} = h$.

By construction, the run $\overline{R}_{\overline{h}-2}$ results from merging the runs R_{h-2} and R_{h-1}, and $\overline{R}_{\overline{h}-1} = R_h$, so that $\ell_{h-1} \leqslant \overline{\ell}_{\overline{h}-2}$ and $\ell_h = \overline{\ell}_{\overline{h}-1}$. Since m is a #3-merge, we conclude that $\overline{\ell}_{\overline{h}-1} = \ell_h < \ell_{h-1} \leqslant \overline{\ell}_{\overline{h}-2}$, i.e., that p cannot be immediately followed by a #1-merge. □

A consequence of Lemma 11 is the following one. Let R be some run in the array to be sorted. Just after R has been pushed, there will be, in this order:

(1) zero or one #1-merge, which is *not* bound to R;
(2) an arbitrary number of #2-merges, which *are* bound to R;
(3) an arbitrary number of #3-merges;
(4) a #4-push operation (unless we have reached the end of the array);
(5) zero or one #1-merge, which *is* bound to R;
(6) other merge or push updates, none of these merges being bound to R.

Furthermore, if there is at least one #3-merge at step 3, then there is no #1-merge at step 5. Therefore, the merges bound to R form a contiguous sequence of merge updates which we call *merge sequence* of R. This situation is illustrated in Figure 2.

Lemma 13. *The total cost of non-expanding half-merges bound to a run R is at most $(2 - \lambda)r - 1$.*

Proof. Let mc be the total cost of the non-expanding half-merges bound to R. If no such half-merge exists, then $\mathrm{mc} = 0 \leqslant r - 1 \leqslant (2 - \lambda)r - 1$. Hence, we assume below that such a half-merge exists, and thus that the merge sequence of R, which is denoted by $\mathsf{M}_{\mathrm{seq}}$, contains at least one merge.

Let $\mathcal{S} = (R_1, \ldots, R_h)$ be the stack just after R has been pushed onto the stack or, if a #1-merge immediately follows the push of R, just after that #1-merge has been performed. We know that $R = R_h$. Moreover, due to Lemma 7, and since the first update performed on $\mathcal{S}$ cannot be a #1-merge, we have

$$\ell_1 > \ldots > \ell_{h-2} \geqslant \ell_{h-1}.$$

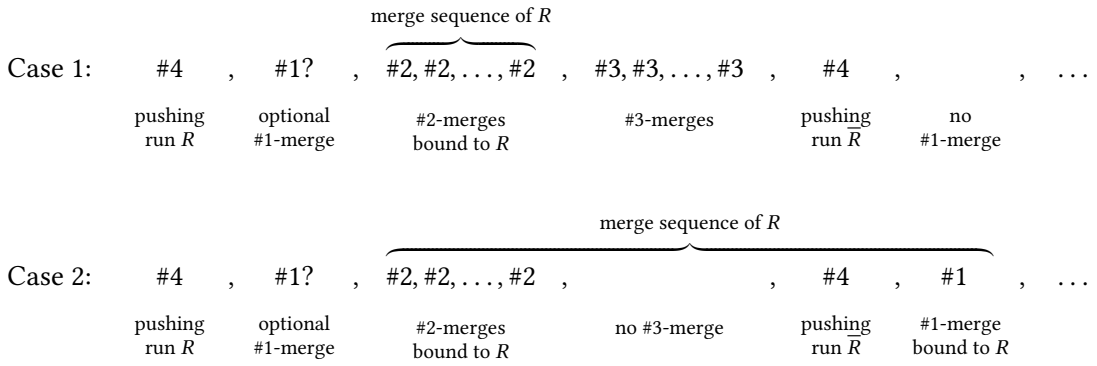


Fig. 2. The run R is pushed. Then come one (optional) #1-merge, a (possibly empty) sequence of #2-merges bound to R, and a (possibly empty) sequence of #3-merges. Finally, a new run $\overline{R}$ is pushed, which may be followed by one (optional) #1-merge bound to R.
In case 1, the latter #1-merge does not exist, and #2-merges bound to R form the merge sequence of R; in case 2, the latter #1-merge exists, the sequence of #3-merges must be empty, and the #2-merges and #1-merge bound to R form the merge sequence of R.

Thus, $\mathsf{M}_{\mathrm{seq}}$ consists in merging R_{h-2} and R_{h-1}, then merging successively the resulting run with $R_{h-3}, R_{h-4}, \dots, R_k$ for some k (we set $k = h-1$ if $\mathsf{M}_{\mathrm{seq}}$ contains no #2-merge nor #3-merge), then possibly with $R = R_h$ itself.

Now, let m' be some non-expanding half-merge bound to R. If m is a #1-merge, then it must be the merge of $R = R_h$ with a smaller run, and its cost is $r = r_h$; if m is a #2-merge, then it must be the merge of some run R_i (with $k \leqslant i \leqslant h-2$) with a smaller run, and its cost is r_i. This proves that $\mathsf{mc} = \sum_{i \in X} r_i$, where the set X is defined by

$$X = \{i \colon 1 \leqslant i \leqslant h \text{ and the half-merge of } R_i \text{ is non-expanding and is bound to } R\}.$$

Then, let m^* be the last non-expanding half-merge bound to R, and let R^* be the run that results from m^*. Just after m^* is completed, all the runs that had been merged since $\mathsf{M}_{\mathrm{seq}}$ started must now belong to the run R^*. These runs must include the run R_{h-1}, which was the first run merged during $\mathsf{M}_{\mathrm{seq}}$ and whose half-merge must have been expanding. It follows that $\sum_{i \in X} r_i \leqslant r^* - r_{h-1} \leqslant r^* - 1$.

Finally, and as mentioned before, m^* must be the non-expanding half-merge of some run R_j: we have $j = h$ if m^* is a #1-merge, and $k \leqslant j \leqslant h-2$ if m^* is a #2-merge. In both cases, $\ell_j \leqslant \ell$. Then, since m^* is non-expanding, we also have $\ell^* \leqslant \ell_j \leqslant \ell$.

Using Lemma 8, we conclude that the total cost of non-expanding half-merges bound to R is

$$\mathsf{mc} = \sum_{i \in X} r_i \leqslant r^* - 1 \leqslant 2^{\ell^*+1} c - 1 \leqslant 2^{\ell+1} c - 1 = 2^{1-\lambda} r - 1 \leqslant (2-\lambda) r - 1. \qquad \square$$

A similar result also holds for those merges performed after the main loop.

Lemma 14. *The total cost of non-expanding half-merges performed in line 11 of Algorithm 1 is at most $n - 1$.*

Proof. Let mc be the total cost of these non-expanding half-merges, and let $\mathcal{S} = (R_1, \ldots, R_h)$ be the stack at the end of the main loop, just before line 11 is reached. Due to Lemma 7 and to the fact that no merge was triggered, we know that

$$\ell_1 > \ldots > \ell_{h-2} > \max\{\ell_{h-1}, \ell_h\}.$$

Consequently, any non-expanding half-merge that takes place in line 11 must be the half-merge of some run R_i with a smaller run. This proves that $\mathrm{mc} = \sum_{i \in X} r_i$, where the set X is defined by

$$\{i \colon 1 \leqslant i \leqslant h \text{ and the half-merge of } R_i \text{ is non-expanding}\}.$$

Since X is a strict subset of $\{1, \ldots, h\}$, it follows that $\mathrm{mc} \leqslant \sum_{i=1}^{h} r_i - 1 = n - 1$. □

We conclude this section by gathering these results as follows.

Proof of Proposition 4. Lemma 9 states that the total cost of expanding half-merges is at most $n(\mathcal{H} - \{\log_2(n/c)\}) + \Lambda$. Then, Lemma 13 states that the total cost of those non-expanding half-merges bound to a given run R is at most $(2-\lambda)r-1$. Taking all runs into account, the total cost of those non-expanding half-merges performed during the main loop of c-adaptive ShiversSort is at most $2n - \Lambda - \rho$. Finally, Lemma 14 states that the total cost of those non-expanding half-merges performed in line 11 is at most $n - 1$. Summing all these costs completes the proof of Proposition 4. □

3.2 A finer upper bound

Now that Theorem 5 has been proved, let us present a finer upper bound on the merge cost of c-adaptive ShiversSort.

Theorem 15. *For every value of the parameter c, the merge cost of c-adaptive ShiversSort is bounded from above by $n\,(\mathcal{H} + \Delta)$, where $\Delta = 24/5 - \log_2(5) \approx 2.478$.*

The proof we draw below relies mainly on the ideas and results already presented in Section 3.1. However, we cannot reuse directly our cost allocation scheme, and we will need a notion of potential instead. As a first step towards defining our potential, we first introduce the notion of *state* of the algorithm.

Definition 16. Consider the execution of the algorithm c-adaptive ShiversSort on a sequence of runs to be sorted. At each step, the algorithm handles both a stack $\mathcal{S} = (R_1, \ldots, R_h)$ of runs and

a sequence $\mathcal{R} = (R_{h+1}, \dots, R_t)$ of those runs that have yet to be discovered and pushed onto the stack. We call *state* of the algorithm, at that step, the sequence $(R_1, \dots, R_t)$, i.e., the concatenation of $\mathcal{S}$ and $\mathcal{R}$.

Finally, two states of the algorithm are said to be *consecutive* if they are distinct from each other and were separated by a single (run push or merge) operation performed by the algorithm.

We immediately see that a push operation does not modify the state of the algorithm. Thus, the operation that separates two consecutive states of the algorithm is necessarily a run merge operation. Moreover, and in order to reduce possible confusions between sequences of runs of different natures, we will stick to the notation $\mathcal{S}$ for stacks and S for states of the algorithm.

We focus now on describing how c-adaptive ShiversSort transforms a state into another one.

Definition 17. Let $\mathcal{R} = (R_1, \dots, R_t)$ be a sequence of runs of length $t \geqslant 2$. We say that an integer x is *dominated* in the sequence $\mathcal{R}$ if $1 \leqslant x \leqslant t-1$ and $\ell_x \leqslant \max\{\ell_{x+1}, \ell_{x+2}\}$, with the convention that $\ell_{t+1} = \infty$ (we may omit mentioning $\mathcal{R}$ when the context is clear). This convention ensures us that $t-1$ is necessarily dominated.

Then, let k be the smallest dominated integer. We say that k is the *merge point* of the sequence $\mathcal{R}$. Finally, we call *successor* of $\mathcal{R}$, and note $\mathrm{succ}(\mathbf{R})$, the sequence of runs

$$(R_1, \dots, R_{k-1}, \overline{R}, R_{k+2}, \dots, R_t),$$

where $\overline{R}$ is the run obtained by merging R_k and R_{k+1}.

Proposition 18. *Let* S *and* $\overline{\mathrm{S}}$ *be two consecutive states encountered during an execution of c-adaptive ShiversSort. We have* $\overline{\mathrm{S}} = \mathit{succ}(\mathrm{S})$.

Proof. Let m be the merge operation that transforms the state S into $\overline{\mathrm{S}}$. Let $\mathcal{S} = (R_1, \dots, R_{k+2})$ be the stack just before m takes place, and let $\mathcal{R} = (R_{k+3}, \dots, R_t)$ be the sequence of those runs that are yet to be pushed onto the stack, so that m consists in merging the runs R_k and R_{k+1}, and that S is the concatenation of $\mathcal{S}$ and $\mathcal{R}$.

Lemma 7 states that $\ell_1 > \ell_2 > \dots > \ell_{k-1} > \max\{\ell_k, \ell_{k+1}\}$, and since c-adaptive ShiversSort performed the merge m, it means that $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\}$. This implies that k is the merge point of S, and therefore that $\overline{\mathrm{S}} = \mathrm{succ}(\mathrm{S})$. □

In addition to the notion of state and to the notations defined in Section 2 (page 8), we will frequently use the following notation. For every run R of length r and level ℓ, we set $r^\bullet = r/(2^\ell c) - 1$. Note that, as R varies over the interval $[2^\ell c, 2^{\ell+1} c)$, the quantity $r^\bullet$ varies over the interval $[0, 1)$. Once again, we will adapt this notation when the name of R varies, e.g., writing $r_i^\bullet$ when considering the run R_i.

We introduce now the notions of potential that we will use in the subsequent proofs.

Definition 19. Let $\Phi : [0,1] \mapsto \mathbb{R}$ be the function defined by $\Phi : x \mapsto \max\{(2-5x)/3, 1/2-x, 0\}$. Then, let $\mathsf{S} = (R_1, \ldots, R_t)$ be a state of the algorithm. By convention, let $\ell_0 = \ell_{t+1} = +\infty$. We define the *potential* of a run R_i in S as the real number $\mathrm{Pot}_0^{\mathsf{S}}(R_i) = \sum_{j=1}^{4} \mathrm{Pot}_j^{\mathsf{S}}(R_i)$, where $\mathrm{Pot}_1^{\mathsf{S}}(R_i) = -\ell_i r_i$ and

$$\mathrm{Pot}_2^{\mathsf{S}}(R_i) = \begin{cases} -r_i & \text{if } \ell_{i-1} \geqslant \ell_i \text{ and } \ell_i < \ell_{i+1}; \\ 0 & \text{if } \ell_{i-1} < \ell_i \text{ or } \ell_i \geqslant \ell_{i+1}; \end{cases}$$

$$\mathrm{Pot}_3^{\mathsf{S}}(R_i) = \begin{cases} 2^{\ell_i} c\, \Phi(r_i^\bullet) & \text{if } \ell_{i-1} \geqslant \ell_i; \\ 2^{\ell_i+1} c & \text{if } \ell_{i-1} < \ell_i; \end{cases}$$

$$\mathrm{Pot}_4^{\mathsf{S}}(R_i) = \begin{cases} 2^{\ell_i} c \left(2\Phi\big((r_i^\bullet + r_{i+1}^\bullet)/2\big) - \Phi(r_i^\bullet) - \Phi(r_{i+1}^\bullet)\right) & \text{if } \ell_{i-1} > \ell_i \text{ and } \ell_i = \ell_{i+1}; \\ 0 & \text{if } \ell_{i-1} \leqslant \ell_i \text{ or } \ell_i \neq \ell_{i+1}. \end{cases}$$

Finally, we call *global potential* of the state S the sum $\mathrm{Pot}(\mathsf{S}) = \sum_{i=1}^{t} \mathrm{Pot}_0^{\mathsf{S}}(R_i)$, i.e., the sum of the potentials of all the runs R_1 to R_t.

Below, and in order to make a good use of the otherwise mysterious function Φ, we will need the following technical lemma.

Lemma 20. *The function Φ is convex and non-increasing. Moreover, for all real numbers x such that $0 \leqslant x \leqslant 1$, we have*

$$1 \geqslant x + 2\Phi(x/2) - \Phi(x) \geqslant x + \Phi(x) \text{ and } \log_2(1+x) + \Phi(x)/(1+x) \geqslant 3 - \Delta.$$

Proof. First, since Φ is a maximum of non-increasing affine functions, it is convex and non-increasing. It already follows that $x + 2\Phi(x/2) - \Phi(x) = (x + \Phi(x)) + 2\left(\Phi(x/2) - \Phi(x)\right) \geqslant x + \Phi(x)$ for all $x \in [0,1]$. It remains to prove that both functions $f : x \mapsto \log_2(1+x) + \Phi(x)/(1+x) + \Delta - 3$ and $g : x \mapsto 1 - x - 2\Phi(x/2) + \Phi(x)$ are non-negative.

First, the function Φ is affine on each of the intervals $[0, 1/4]$, $[1/4, 1/2]$ and $[1/2, 1]$, and thus we study f on each of these intervals:

- If $0 < x < 1/4$, then $\Phi(x) = (2-5x)/3$, and thus

$$(1+x)^2 f'(x) = (1+x)\log_2(e) - 7/3 \leqslant 5/4 \log_2(e) - 7/3 \approx -0.5.$$

  Hence, f is decreasing on $[0, 1/4]$, and $f(x) \geqslant f(1/4) = 0$ when $0 \leqslant x \leqslant 1/4$.
- If $1/4 < x < 1/2$, then $\Phi(x) = 1/2 - x$, and thus

$$(1+x)^2 f'(x) = (1+x)\log_2(e) - 3/2 \geqslant (1+x)\log_2(e) - 3/2 \approx 0.3.$$

Hence, f is increasing on $[1/4, 1/2]$, and $f(x) \geqslant f(1/4) = 0$ when $1/4 \leqslant x \leqslant 1/2$.

▹ If $1/2 < x < 1$, then $\Phi(x) = 0$, and thus

$$(1+x)f'(x) = \log_2(e) > 0.$$

Hence, f is increasing on $[1/2, 1]$, and $f(x) \geqslant f(1/2) \geqslant 0$ when $1/2 \leqslant x \leqslant 1$.

Second, since Φ is affine on each of the intervals $[0, 1/4]$, $[1/4, 1/2]$ and $[1/2, 1]$, so is g. Since $g(0) = 1/3$, $g(1/4) = 1/12$ and $g(1/2) = g(1) = 0$, we conclude that g is non-negative on each of these intervals. □

In particular, the function Φ is convex and bounded from above by $2/3$. Consequently, the following simple observation can be made about the order in which, in Definition 19, we listed the possible values of the real numbers $\mathrm{Pot}_j^{\overline{\mathsf{S}}}(R_i)$ for $j = 2, 3, 4$: the above value is always the smallest one. Indeed, we always have $0 \geqslant -r_i$, $2^{\ell_i+1}c \geqslant 2^{\ell_i+1}c/3 \geqslant 2^{\ell_i}c\,\Phi(r_i^\bullet)$, and $0 \geqslant 2^{\ell_i}c\left(2\Phi\big((r_i^\bullet + r_{i+1}^\bullet)/2\big) - \Phi(r_i^\bullet) - \Phi(r_{i+1}^\bullet)\right)$. Remembering this may help the reader to follow the arguments used in the next lemmas, which concern the variations of potential of a given run when a merge is performed.

More precisely, we prove now that the variation of global potential between two consecutive states separated by a merge operation m is a good over-approximation of the cost of m. This is the object of the two following results: the first one proves that the variation of global potential can be under-approximated by a *local* quantity, and the second result then proves that this is enough.

Lemma 21. *Let* $\mathsf{S} = (R_1, \dots, R_t)$ *and* $\overline{\mathsf{S}} = (\overline{R}_1, \dots, \overline{R}_{t-1})$ *be two consecutive states of c-adaptive ShiversSort, and let* k *be the merge point of* S. *If* $k \leqslant t-2$, *then*

$$\mathrm{Pot}(\mathsf{S}) \geqslant \mathrm{Pot}(\overline{\mathsf{S}}) + \mathrm{Pot}_0^{\mathsf{S}}(R_k) + \mathrm{Pot}_0^{\mathsf{S}}(R_{k+1}) + \mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_k) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1}).$$

Proof. By construction, we know that $R_i = \overline{R}_i$ for all $i \leqslant k-1$, that $R_i = \overline{R}_{i-1}$ for all $i \geqslant k+2$, and that the run $\overline{R}_k$ results from merging the runs R_k and R_{k+1}. Since the potential of a run in a state depends only of the lengths of that run and of its neighbours, it already follows that $\mathrm{Pot}_0^{\mathsf{S}}(R_i) = \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_i)$ for all $i \leqslant k-2$ and that $\mathrm{Pot}_0^{\mathsf{S}}(R_i) = \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_{i-1})$ for all $i \geqslant k+3$.

Then, we prove that $\mathrm{Pot}_j^{\mathsf{S}}(R_{k-1}) \geqslant \mathrm{Pot}_j^{\overline{\mathsf{S}}}(\overline{R}_{k-1})$ for all $j = 1, 2, 3, 4$:

(1) both $\mathrm{Pot}_1^{\mathsf{S}}(R_{k-1})$ and $\mathrm{Pot}_1^{\overline{\mathsf{S}}}(\overline{R}_{k-1})$ are equal to $-\ell_{k-1}r_{k-1}$;
(2) since $\ell_{k-1} \geqslant \ell_k$, we have $\mathrm{Pot}_2^{\mathsf{S}}(R_{k-1}) \geqslant 0 \geqslant \mathrm{Pot}_2^{\overline{\mathsf{S}}}(\overline{R}_{k-1})$;
(3) since $\overline{\ell}_{k-2} = \ell_{k-2} \geqslant \ell_{k-1} = \overline{\ell}_{k-1}$, we have $\mathrm{Pot}_3^{\mathsf{S}}(R_{k-1}) = 2^{\ell_{k-1}}c\Phi(r_{k-1}^\bullet) = \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k-1})$;
(4) since $\ell_{k-1} \neq \ell_k$, we have $\mathrm{Pot}_4^{\mathsf{S}}(R_{k-1}) = 0 \geqslant \mathrm{Pot}_4^{\overline{\mathsf{S}}}(\overline{R}_{k-1})$.

We prove similarly that $\mathrm{Pot}_j^{\mathsf{S}}(R_{k+2}) \geqslant \mathrm{Pot}_j^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$ for all $j = 1, 2, 4$ (recall the convention that $\ell_{k+3} = +\infty$ if $k = t-2$):

(1) both $\mathrm{Pot}_1^{\mathsf{S}}(R_{k+2})$ and $\mathrm{Pot}_1^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$ are equal to $-\ell_{k+2}r_{k+2}$;

(2) if $\ell_{k+1} < \ell_{k+2}$ or $\ell_{k+2} \geqslant \ell_{k+3}$, then $\mathrm{Pot}_1^{\mathsf{S}}(R_{k+2}) = 0 \geqslant \mathrm{Pot}_1^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$; in the contrary case, we know that $\overline{\ell}_k \geqslant \ell_{k+1} \geqslant \ell_{k+2} = \overline{\ell}_{k+1}$ and that $\overline{\ell}_{k+1} = \ell_{k+2} < \ell_{k+3} = \overline{\ell}_{k+2}$, which proves that $\mathrm{Pot}_2^{\mathsf{S}}(R_{k+2}) = -r_{k+2} = \mathrm{Pot}_2^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$;
(4) if $\ell_{k+1} \leqslant \ell_{k+2}$ or $\ell_{k+2} \neq \ell_{k+3}$, then $\mathrm{Pot}_4^{\mathsf{S}}(R_{k+2}) = 0 \geqslant \mathrm{Pot}_4^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$; in the contrary case, we know that $\overline{\ell}_k \geqslant \ell_{k+1} > \ell_{k+2} = \overline{\ell}_{k+1}$ and that $\overline{\ell}_{k+1} = \ell_{k+2} = \ell_{k+3} = \overline{\ell}_{k+2}$, which proves that $\mathrm{Pot}_4^{\mathsf{S}}(R_{k+2}) = 2^{\ell_{k+2}} c \left(2\Phi\Big((r^\bullet_{k+2} + r^\bullet_{k+3})/2\Big) - \Phi(r^\bullet_{k+2}) - \Phi(r^\bullet_{k+3})\right) = \mathrm{Pot}_4^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$.

Consequently, we conclude that

$$\begin{aligned}
\mathrm{Pot}(\mathsf{S}) - \mathrm{Pot}(\overline{\mathsf{S}}) &= \textstyle\sum_{i=1}^{t} \mathrm{Pot}_0^{\mathsf{S}}(R_i) - \sum_{i=1}^{t-1} \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_i) \\
&= \textstyle\sum_{i=k-1}^{k+2} \mathrm{Pot}_0^{\mathsf{S}}(R_i) - \sum_{i=k-1}^{k+1} \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_i) \\
&\geqslant \mathrm{Pot}_0^{\mathsf{S}}(R_k) + \mathrm{Pot}_0^{\mathsf{S}}(R_{k+1}) + \mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_k) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1}). \qquad \square
\end{aligned}$$

Proposition 22. *Let m be a merge operation of cost* mc*, and let* S *and* $\overline{\mathsf{S}}$ *be the states that m separates. It holds that* $\mathrm{Pot}(\mathsf{S}) \geqslant \mathrm{Pot}(\overline{\mathsf{S}}) + \mathrm{mc}$.

Proof. Let $\mathsf{S} = (R_1, \ldots, R_t)$, and $\overline{\mathsf{S}} = (\overline{R}_1, \ldots, \overline{R}_{t-1})$, and let k be the merge point of S. If $k = t-1$, let us append to both states a fictitious run R_∞ of length $2n$, where $n = r_1 + \ldots + r_t$. This does not change the fact that $\overline{\mathsf{S}} = \mathrm{succ}(\mathsf{S})$, and only adds $\mathrm{Pot}_0^{\mathsf{S}}(R_\infty) = \mathrm{Pot}_0^{\overline{\mathsf{S}}}(R_\infty)$ to the global potentials of both states. Then, in view of Lemma 21, it suffices to prove that the real number

$$\Theta = \mathrm{Pot}_0^{\mathsf{S}}(R_k) + \mathrm{Pot}_0^{\mathsf{S}}(R_{k+1}) + \mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_0^{\overline{\mathsf{S}}}(\overline{R}_k) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1}) - \mathrm{mc}$$

is non-negative.

We already evaluate the difference $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$ by distinguishing three subcases, depending on how ℓ_{k+2} compares with ℓ_{k+1} and $\overline{\ell}_k$:

- If $\ell_{k+1} \geqslant \ell_{k+2}$, we have $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) = 2^{\ell_{k+2}} c\, \Phi(r^\bullet_{k+2}) = \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$.
- Similarly, if $\overline{\ell}_k < \ell_{k+2}$, we have $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) = 2^{\ell_{k+2}} c = \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1})$.
- Then, if $\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k$, we have $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1}) = 2^{\ell_{k+2}} c\, (2 - \Phi(r^\bullet_{k+2}))$.

Hence, overall, $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1}) = \mathbf{1}_{\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k} 2^{\ell_{k+2}} c\, (2 - \Phi(r^\bullet_{k+2}))$.

Then, we also evaluate the term $\mathrm{Pot}_4^{\mathsf{S}}(R_{k+1})$ hidden in the term $\mathrm{Pot}_0^{\mathsf{S}}(R_{k+1})$. Indeed, if $\ell_k > \ell_{k+1}$, and since $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\}$, we have $\ell_{k+1} \neq \ell_{k+2}$. Thus, we always have $\mathrm{Pot}_4^{\mathsf{S}}(R_{k+1}) = 0$.

However, going further requires considering separately several cases. In each case, we decompose Θ as a sum, then we evaluate each summand separately.

Case 1: $\ell_k = \ell_{k+1}$. Here, we decompose Θ as the sum

$$\begin{aligned}\Theta = (\ \mathrm{Pot}_1^{\mathrm{S}}(R_k) + \mathrm{Pot}_1^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_1^{\overline{\mathrm{S}}}(\overline{R}_k) - \mathrm{mc}\ ) + \mathrm{Pot}_2^{\mathrm{S}}(R_k) + \mathrm{Pot}_2^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_2^{\overline{\mathrm{S}}}(\overline{R}_k)\\ + (\ \mathrm{Pot}_3^{\mathrm{S}}(R_k) + \mathrm{Pot}_3^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_k) + \mathrm{Pot}_4^{\mathrm{S}}(R_k)\ ) + \mathrm{Pot}_4^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k)\\ + (\ \mathrm{Pot}_3^{\mathrm{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_{k+1})\ ).\end{aligned}$$

- Since $\overline{\ell}_k = \ell_k + 1 = \ell_{k+1} + 1$, we have

$$\begin{aligned}\mathrm{Pot}_1^{\mathrm{S}}(R_k) + \mathrm{Pot}_1^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_1^{\overline{\mathrm{S}}}(\overline{R}_k) - \mathrm{mc} &= -\ell_k r_k - \ell_{k+1} r_{k+1} + \overline{\ell}_k (r_k + r_{k+1}) - (r_k + r_{k+1})\\ &= (\overline{\ell}_k - \ell_k - 1) r_k + (\overline{\ell}_k - \ell_{k+1} - 1) r_{k+1} = 0.\end{aligned}$$

- Since $\overline{\ell}_k = \ell_k + 1 = \ell_{k+1} + 1$, we also have

$$r_k^\bullet + r_{k+1}^\bullet = r_k/(2^{\ell_k}\, c) + r_{k+1}/(2^{\ell_{k+1}}\, c) - 2 = 2\overline{r}_k/(2^{\overline{\ell}_k}\, c) - 2 = 2\overline{r}_k^\bullet.$$

  Since $\ell_{k-1} > \ell_k = \ell_{k+1}$ and $\overline{\ell}_{k-1} \geqslant \overline{\ell}_k$, it follows that

$$\begin{aligned}\mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_k) = 2^{\ell_k+1} c\, \Phi(\overline{r}_k^\bullet) &= 2^{\ell_k} c\, \Phi(r_k^\bullet) + 2^{\ell_k} c\, \Phi(r_{k+1}^\bullet) + 2^{\ell_k} c\, \big(2\Phi(\overline{r}_k^\bullet) - \Phi(r_k^\bullet) - \Phi(r_{k+1}^\bullet)\big)\\ &= \mathrm{Pot}_3^{\mathrm{S}}(R_k) + \mathrm{Pot}_3^{\mathrm{S}}(R_{k+1}) + \mathrm{Pot}_4^{\mathrm{S}}(R_k).\end{aligned}$$

- As mentioned above, since the function Φ is convex, the partial potential Pot_4 is never positive, and thus $\mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k) \leqslant 0$.
- Since $\ell_{k+1} < \ell_{k+2}$ if and only if $\overline{\ell}_k \leqslant \ell_{k+2}$, we have $\mathrm{Pot}_2^{\mathrm{S}}(R_{k+1}) = -\mathbf{1}_{\overline{\ell}_k \leqslant \ell_{k+2}} r_{k+1}$.
- We have $\mathrm{Pot}_2^{\mathrm{S}}(R_k) = 0$, $\mathrm{Pot}_4^{\mathrm{S}}(R_{k+1}) = 0$ and $\mathrm{Pot}_2^{\overline{\mathrm{S}}}(\overline{R}_k) = -\mathbf{1}_{\overline{\ell}_k < \ell_{k+2}} \overline{r}_k$.
- Finally, Φ is bounded from above by 1. Moreover, since ℓ_{k+1} and $\overline{\ell}_k$ are consecutive integers, $\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k$ if and only if $\ell_{k+2} = \overline{\ell}_k$. Thus, since $r_{k+1} \leqslant 2^{\ell_{k+1}+1} c = 2^{\overline{\ell}_k} c$, it follows that

$$\mathrm{Pot}_3^{\mathrm{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_{k+1}) = \mathbf{1}_{\ell_{k+2} = \overline{\ell}_k} 2^{\ell_{k+2}} c\, (2 - \Phi(r_{k+2}^\bullet)) \geqslant \mathbf{1}_{\ell_{k+2} = \overline{\ell}_k} 2^{\overline{\ell}_k} c \geqslant \mathbf{1}_{\ell_{k+2} = \overline{\ell}_k} r_{k+1}.$$

Gathering all these equalities and inequalities and replacing each summand by its value or by a lower bound of that summand, we conclude that

$$\begin{aligned}\Theta &= 0 + 0 - \mathbf{1}_{\overline{\ell}_k \leqslant \ell_{k+2}} r_{k+1} + \mathbf{1}_{\overline{\ell}_k < \ell_{k+2}} \overline{r}_k + 0 + 0 - \mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k) + (\ \mathrm{Pot}_3^{\mathrm{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_{k+1})\ )\\ &\geqslant -\mathbf{1}_{\overline{\ell}_k \leqslant \ell_{k+2}} r_{k+1} + \mathbf{1}_{\overline{\ell}_k < \ell_{k+2}} r_{k+1} + \mathbf{1}_{\overline{\ell}_k = \ell_{k+2}} r_{k+1}\\ &\geqslant 0\end{aligned}$$

whenever $\ell_k = \ell_{k+1}$.

Case 2: $\overline{\ell}_k > \ell_k > \ell_{k+1}$ or $\overline{\ell}_k > \ell_{k+1} > \ell_k$. Here, we decompose Θ as the sum

$$\begin{aligned}\Theta = {} & (\,\mathrm{Pot}_1^{\mathsf{S}}(R_k) + \mathrm{Pot}_1^{\mathsf{S}}(R_{k+1}) - \mathrm{Pot}_1^{\overline{\mathsf{S}}}(\overline{R}_k) - \mathrm{mc}\,) + (\,\mathrm{Pot}_2^{\mathsf{S}}(R_k) + \mathrm{Pot}_2^{\mathsf{S}}(R_{k+1})\,)\\ & + (\,\mathrm{Pot}_3^{\mathsf{S}}(R_{k+1}) - \mathrm{Pot}_2^{\overline{\mathsf{S}}}(\overline{R}_k)\,) + (\,\mathrm{Pot}_3^{\mathsf{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_{k+1})\,)\\ & + \mathrm{Pot}_3^{\mathsf{S}}(R_k) - \mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_k) + \mathrm{Pot}_4^{\mathsf{S}}(R_k) + \mathrm{Pot}_4^{\mathsf{S}}(R_{k+1}) - \mathrm{Pot}_4^{\overline{\mathsf{S}}}(\overline{R}_k).\end{aligned}$$

Below, in order to let the runs R_k and R_{k+1} play somewhat symmetric roles, we set $R_{\max} = R_k$ and $R_{\min} = R_{k+1}$ if $\ell_k > \ell_{k+1}$, and we set $R_{\max} = R_{k+1}$ and $R_{\min} = R_k$ if $\ell_k < \ell_{k+1}$. Accordingly, we set $r_{\max} = \max\{r_k, r_{k+1}\}$, $r_{\min} = \min\{r_k, r_{k+1}\}$, $\ell_{\max} = \max\{\ell_k, \ell_{k+1}\}$ and $\ell_{\min} = \min\{\ell_k, \ell_{k+1}\}$.

- Since $\overline{\ell}_k = \ell_{\max} + 1$ and $\ell_{\max} \geqslant \ell_{\min} + 1$, we have

$$\begin{aligned}\mathrm{Pot}_1^{\mathsf{S}}(R_k) + \mathrm{Pot}_1^{\mathsf{S}}(R_{k+1}) - \mathrm{Pot}_1^{\overline{\mathsf{S}}}(\overline{R}_k) - \mathrm{mc} &= -\ell_k r_k - \ell_{k+1} r_{k+1} + \overline{\ell}_k (r_k + r_{k+1}) - (r_k + r_{k+1})\\ &= (\overline{\ell}_k - \ell_k - 1) r_k + (\overline{\ell}_k - \ell_{k+1} - 1) r_{k+1}\\ &= (\overline{\ell}_k - \ell_{\max} - 1) r_{\max} + (\overline{\ell}_k - \ell_{\min} - 1) r_{\min}\\ &= (\ell_{\max} - \ell_{\min}) r_{\min}\\ &\geqslant r_{\min}.\end{aligned}$$

- Since $\ell_{\min} < \ell_{\max}$, we have $\mathrm{Pot}_2^{\mathsf{S}}(R_{\max}) = 0$. Since $\mathrm{Pot}_2^{\mathsf{S}}(R_{\min}) \geqslant -r_{\min}$ by definition of Pot_2, it follows that

$$\mathrm{Pot}_2^{\mathsf{S}}(R_k) + \mathrm{Pot}_2^{\mathsf{S}}(R_{k+1}) = \mathrm{Pot}_2^{\mathsf{S}}(R_{\max}) + \mathrm{Pot}_2^{\mathsf{S}}(R_{\max}) \geqslant -r_{\min}.$$

- By definition of Pot_2 and Pot_3, we have $\mathrm{Pot}_2^{\overline{\mathsf{S}}}(\overline{R}_k) \leqslant 0$ and $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+1}) \geqslant 0$. Then, we distinguish two sub-cases, depending on how the levels ℓ_{k+2} compares with ℓ_{k+1} and $\overline{\ell}_k$:
  - If $\overline{\ell}_k < \ell_{k+2}$, we have $-\mathrm{Pot}_2^{\overline{\mathsf{S}}}(\overline{R}_k) = \overline{r}_k \geqslant 2^{\overline{\ell}_k} c \geqslant 2^{\overline{\ell}_k + 1} c/3$.
  - If $\ell_{k+2} \leqslant \ell_{k+1}$, we have $\ell_{k+1} = \max\{\ell_{k+1}, \ell_{k+2}\} \geqslant \ell_k$. It follows that $\ell_{k+1} > \ell_k$ and that $\overline{\ell}_k = \ell_{k+1} + 1$, which proves that $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+1}) = 2^{\ell_k + 1} c \geqslant 2^{\overline{\ell}_k + 1} c/3$.

  Hence, overall, we have $\mathrm{Pot}_3^{\mathsf{S}}(R_{k+1}) - \mathrm{Pot}_2^{\overline{\mathsf{S}}}(\overline{R}_k) \geqslant \mathbf{1}_{\overline{\ell}_k < \ell_{k+2} \text{ or } \ell_{k+2} \leqslant \ell_{k+1}} 2^{\overline{\ell}_k + 1} c/3$.
- Since Φ is bounded from above by $2/3$, we have $\mathrm{Pot}_3^{\overline{\mathsf{S}}}(\overline{R}_k) = 2^{\overline{\ell}_k} c\, \Phi(\overline{r}_k^{\bullet}) \leqslant 2^{\overline{\ell}_k + 1}/3$.
- As mentioned above, since the function Φ is convex, the partial potential Pot_4 is never positive, and thus we have $\mathrm{Pot}_4^{\overline{\mathsf{S}}}(\overline{R}_k) \leqslant 0$.
- We have $\mathrm{Pot}_3^{\mathsf{S}}(R_k) \geqslant 0$, $\mathrm{Pot}_4^{\mathsf{S}}(R_{k+1}) = 0$ and $\mathrm{Pot}_4^{\mathsf{S}}(R_k) = 0$.

▹ Finally, if $\ell_{k+1} < \ell_{k+2}$, we have $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\} = \ell_{k+2}$, and thus $\ell_{k+2} \geqslant \overline{\ell}_k - 1$. Since Φ is bounded from above by $2/3$, it follows that

$$\begin{aligned}\mathrm{Pot}_3^{\mathrm{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_{k+1}) &= \mathbf{1}_{\ell_{k+1}<\ell_{k+2}\leqslant\overline{\ell}_k} 2^{\ell_{k+2}} c\,(2 - \Phi(r^{\bullet}_{k+2})) \\ &\geqslant \mathbf{1}_{\ell_{k+1}<\ell_{k+2}\leqslant\overline{\ell}_k} 2^{\ell_{k+2}} c \times 4/3 \\ &\geqslant \mathbf{1}_{\ell_{k+1}<\ell_{k+2}\leqslant\overline{\ell}_k} 2^{\overline{\ell}_k+1} c/3.\end{aligned}$$

Gathering all these equalities and inequalities and replacing each summand by its value or by a lower bound of that summand, we conclude that

$$\begin{aligned}\Theta &\geqslant r_{\min} - r_{\min} + \mathbf{1}_{\overline{\ell}_k<\ell_{k+2} \text{ or } \ell_{k+2}\leqslant\ell_{k+1}} 2^{\overline{\ell}_k+1} c/3 + \mathbf{1}_{\ell_{k+1}<\ell_{k+2}\leqslant\overline{\ell}_k} 2^{\overline{\ell}_k+1} c/3 + 0 - 2^{\overline{\ell}_k+1} c/3 + 0 + 0 - 0 \\ &\geqslant 0\end{aligned}$$

whenever $\ell_k \neq \ell_{k+1}$ and $\overline{\ell}_k > \max\{\ell_k, \ell_{k+1}\}$.

Case 3: $\overline{\ell}_k = \ell_k > \ell_{k+1}$ or $\overline{\ell}_k = \ell_{k+1} > \ell_k$. Here, we decompose Θ as the sum

$$\begin{aligned}\Theta = {} & (\,\mathrm{Pot}_1^{\mathrm{S}}(R_k) + \mathrm{Pot}_1^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_1^{\overline{\mathrm{S}}}(\overline{R}_k) - \mathrm{mc}\,) + (\,\mathrm{Pot}_2^{\mathrm{S}}(R_k) + \mathrm{Pot}_2^{\mathrm{S}}(R_{k+1})\,) - \mathrm{Pot}_2^{\overline{\mathrm{S}}}(\overline{R}_k) \\ & + (\,\mathrm{Pot}_3^{\mathrm{S}}(R_k) + \mathrm{Pot}_3^{\mathrm{S}}(R_{k+1})\,) + (\,\mathrm{Pot}_3^{\mathrm{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_{k+1})\,) \\ & - (\,\mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_k) + \mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k)\,) + \mathrm{Pot}_4^{\mathrm{S}}(R_k) + \mathrm{Pot}_4^{\mathrm{S}}(R_{k+1}).\end{aligned}$$

Like in case 2, we use the notations $R_{\max}$, $R_{\min}$, $r_{\max}$, $r_{\min}$, $\ell_{\max}$ and $\ell_{\min}$. In addition, we will often distinguish sub-cases, based on how ℓ_{k+2} compares with $\overline{\ell}_k$ and ℓ_{k+1}: either (i) $\overline{\ell}_k < \ell_{k+2}$, or (ii) $\ell_{k+2} \leqslant \ell_{k+1}$, or (iii) $\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k$. Since $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\}$ and $\overline{\ell}_k = \max\{\ell_k, \ell_{k+1}\}$, the latter case arises if and only if $\overline{\ell}_k = \ell_k = \ell_{k+2} > \ell_{k+1}$.

▹ Since $\overline{\ell}_k = \ell_{\max}$ and $\ell_{\max} \geqslant \ell_{\min} + 1$, we have

$$\begin{aligned}\mathrm{Pot}_1^{\mathrm{S}}(R_k) + \mathrm{Pot}_1^{\mathrm{S}}(R_{k+1}) - \mathrm{Pot}_1^{\overline{\mathrm{S}}}(\overline{R}_k) - \mathrm{mc} + {} &= -\ell_k r_k - \ell_{k+1} r_{k+1} + \overline{\ell}_k (r_k + r_{k+1}) - (r_k + r_{k+1}) \\ &= (\overline{\ell}_k - \ell_k - 1) r_k + (\overline{\ell}_k - \ell_{k+1} - 1) r_{k+1} \\ &= (\overline{\ell}_k - \ell_{\max} - 1) r_{\max} + (\overline{\ell}_k - \ell_{\min} - 1) r_{\min} \\ &= -r_{\max} + (\ell_{\max} - \ell_{\min} - 1) r_{\min} \\ &\geqslant -r_{\max}.\end{aligned}$$

▹ Since $\ell_{\min} < \ell_{\max}$, we have $\mathrm{Pot}_2^{\mathrm{S}}(R_{\max}) = 0$. Since $\mathrm{Pot}_2^{\mathrm{S}}(R_{\min}) \geqslant -r_{\min}$ by definition of Pot_2, it follows that

$$\mathrm{Pot}_2^{\mathrm{S}}(R_k) + \mathrm{Pot}_2^{\mathrm{S}}(R_{k+1}) = \mathrm{Pot}_2^{\mathrm{S}}(R_{\max}) + \mathrm{Pot}_2^{\mathrm{S}}(R_{\max}) \geqslant -r_{\min}.$$

▹ We distinguish two sub-cases, depending on how ℓ_k compares with ℓ_{k+1}:

- If $\overline{\ell}_k = \ell_k > \ell_{k+1}$, since Φ is non-decreasing and $r_k^\bullet = r_k/(2^{\overline{\ell}_k}c) - 1 \leqslant \overline{r}_k/(2^{\overline{\ell}_k}c) - 1 = \overline{r}_k^\bullet$, we have $\mathrm{Pot}_3^{\mathrm{S}}(R_k) + \mathrm{Pot}_3^{\mathrm{S}}(R_{k+1}) = 2^{\ell_k} c\,\Phi(r_k^\bullet) + 2^{\ell_{k+1}} c\,\Phi(r_{k+1}^\bullet) \geqslant 2^{\overline{\ell}_k} c\,\Phi(r_k^\bullet) \geqslant 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet)$.
- If $\overline{\ell}_k = \ell_{k+1} > \ell_k$, we have $\mathrm{Pot}_3^{\mathrm{S}}(R_k) + \mathrm{Pot}_3^{\mathrm{S}}(R_{k+1}) = 2^{\ell_k} c\,\Phi(r_k^\bullet) + 2^{\ell_{k+1}+1} c \geqslant 2^{\overline{\ell}_k+1} c \geqslant 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet)$ anyway, because Φ is bounded from above by $2/3$.

- We also distinguish two cases, depending on how ℓ_{k+2} compares with ℓ_{k+1} and $\overline{\ell}_k$:
  - In general, since the function Φ is convex and the partial potential function Pot_4 is never positive, we have $\mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k) \leqslant 0$, and thus $\mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_k) + \mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k) \leqslant \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_k) = 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet)$.
  - If $\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k$, we mentioned above that $\overline{\ell}_k = \ell_{k+2}$. Then, since Φ is non-increasing, we have

$$\begin{aligned}
\mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_k) + \mathrm{Pot}_4^{\overline{\mathrm{S}}}(\overline{R}_k) &= 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet) + 2^{\overline{\ell}_k} c\left(2\Phi((\overline{r}_k^\bullet + r_{k+2}^\bullet)/2) - \Phi(\overline{r}_k^\bullet) - \Phi(r_{k+2}^\bullet)\right) \\
&= 2^{\overline{\ell}_k} c\left(2\Phi((\overline{r}_k^\bullet + r_{k+2}^\bullet)/2 - \Phi(r_{k+2}^\bullet)\right) \\
&\leqslant 2^{\overline{\ell}_k} c\left(2\Phi(\overline{r}_k^\bullet/2) - \Phi(r_{k+2}^\bullet)\right).
\end{aligned}$$

- We have $\mathrm{Pot}_2^{\overline{\mathrm{S}}}(\overline{R}_k) = -\mathbf{1}_{\overline{\ell}_k < \ell_{k+2}}\overline{r}_k$, $\mathrm{Pot}_4^{\mathrm{S}}(R_{k+1}) = 0$ and $\mathrm{Pot}_4^{\overline{\mathrm{S}}}(R_k) = 0$.
- Finally, we proved in the preamble of Case 3 that, if $\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k$, then $\ell_{k+2} = \overline{\ell}_k$. Hence,

$$\mathrm{Pot}_3^{\mathrm{S}}(R_{k+2}) - \mathrm{Pot}_3^{\overline{\mathrm{S}}}(\overline{R}_{k+1}) = \mathbf{1}_{\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k} 2^{\overline{\ell}_k} c\,(2 - \Phi(r_{k+2}^\bullet)).$$

Gathering all these equalities and inequalities and replacing each summand by its value or by a lower bound of that summand, we conclude that:

- If $\overline{\ell}_k < \ell_{k+2}$, we have

$$\Theta \geqslant -r_{\max} - r_{\min} + \overline{r}_k + 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet) + 0 - 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet) + 0 + 0 = 0.$$

- If $\ell_k < \ell_{k+1}$, we have

$$\Theta \geqslant -r_{\max} - r_{\min} + 0 + 2^{\overline{\ell}_k+1} c + 0 - 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet) + 0 + 0 = 2^{\overline{\ell}_k} c\,(1 - \overline{r}_k^\bullet - \Phi(\overline{r}_k^\bullet)) \geqslant 0.$$

- If $\ell_{k+2} \leqslant \overline{\ell}_k$ and $\ell_{k+1} < \ell_k$, the inequality $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\}$ proves that $\ell_{k+1} < \ell_{k+2} \leqslant \overline{\ell}_k$. Thus, we have

$$\begin{aligned}
\Theta \geqslant\ & -r_{\max} - r_{\min} + 0 + 2^{\overline{\ell}_k} c\,\Phi(\overline{r}_k^\bullet) + 2^{\overline{\ell}_k} c\,(2 - \Phi(r_{k+2}^\bullet)) \\
&\qquad - 2^{\overline{\ell}_k} c\,(2\Phi(\overline{r}_k^\bullet/2) - \Phi(r_{k+2}^\bullet)) + 0 + 0 \\
&\geqslant 2^{\overline{\ell}_k} c\,(1 + \Phi(\overline{r}_k^\bullet) - \overline{r}_k^\bullet - 2\Phi(\overline{r}_k^\bullet/2)) \\
&\geqslant 0.
\end{aligned}$$

This proves that $\Theta \geqslant 0$ whenever $\ell_k \neq \ell_{k+1}$ and $\overline{\ell}_k = \max\{\ell_k, \ell_{k+1}\}$. □

Equipped with these hard-won results, it is now time to prove Theorem 15 itself.

Proof of Theorem 15. Let $\mathrm{S} = (R_1, \ldots, R_\rho)$ be the initial state, i.e., the run decomposition of the array to sort, and let $\overline{\mathrm{S}} = (\overline{R}_{\mathsf{end}})$ be the last state encountered in the algorithm, whose only run has length $r_{\mathsf{end}} = r_1 + \ldots + r_\rho = n$.

For every run R_i, we have

$$\begin{aligned}\mathrm{Pot}_0^{\mathrm{S}}(R_i) &\leqslant 2^{\ell_i+1}c - \ell_i r_i = \left(2/(1+r_i^\bullet) + \log_2(c) - \log_2(r_i) + \log_2(1+r_i^\bullet)\right) r_i \\ &\leqslant \left(2 + \log_2(c) - \log_2(r_i)\right) r_i \qquad \text{by applying Lemma 8 to } x = \log_2(1+r_i^\bullet).\end{aligned}$$

Therefore, it holds that $\mathrm{Pot}(\mathrm{S}) \leqslant \sum_{i=1}^{\rho}(2+\log_2(c))r_i - r_i\log_2(r_i) = (2+\log_2(c)+\mathcal{H}-\log_2(n))n$. Then, we also verify that

$$\begin{aligned}\mathrm{Pot}(\overline{\mathrm{S}}) &= 2^{\ell_{\mathsf{end}}} c\,\Phi(r_{\mathsf{end}}^\bullet) - (\ell_{\mathsf{end}}+1)r_{\mathsf{end}} \\ &= \left(\Phi(n^\bullet)/(1+n^\bullet) - \log_2(n) + \log_2(1+n^\bullet) + \log_2(c) - 1\right) n \\ &\geqslant \left(3 - \Delta - \log_2(n) + \log_2(c) - 1\right) n.\end{aligned}$$

Consequently, it follows from Proposition 22 that the total merge cost of c-adaptive ShiversSort is $\mathrm{mc} \leqslant \mathrm{Pot}(\mathrm{S}) - \mathrm{Pot}(\overline{\mathrm{S}}) \leqslant n(\mathcal{H}+\Delta)$, which completes the proof. □

4 BEST-CASE AND WORST-CASE MERGE COSTS

In the introduction, we mentioned that, unlike PowerSort and length-adaptive ShiversSort, the algorithm adaptive ShiversSort is 3-aware. This suggests that it might be preferred to PowerSort and length-adaptive ShiversSort. However, the worst-case merge cost of PowerSort and length-adaptive ShiversSort is at most $n(\mathcal{H}+2)$, whereas the above analysis only proves that the merge cost of adaptive ShiversSort is bounded from above by $n(\mathcal{H}+\Delta)$. Hence, and most notably in cases where $\mathcal{H}$ is small, PowerSort or length-adaptive ShiversSort might be significantly better options. Furthermore, in other cases than the worst case, we have little information on the relative costs of adaptive ShiversSort, PowerSort and length-adaptive ShiversSort.

We address these problems as follows. First, we derive lower bounds on the best-case merge cost of any merge policy. Then, we prove that the worst-case merge cost of PowerSort and length-adaptive ShiversSort is actually optimal among all the stable natural merge-sort algorithms.

4.1 Best-case merge cost

Given a sequence $\mathbf{r} = (r_1, \ldots, r_\rho)$ of run lengths and an array of length $n = r_1 + \ldots + r_\rho$ that splits into monotonic runs of lengths $r_1, \ldots, r_\rho$, what is the best merge cost of any merge policy? An answer to this question is given by Theorem 24, which is a rephrasing of results from [3, 9, 13, 16, 18], adapted to the context of sorting algorithms and merge costs.

This answer comes from the analysis of *merge trees*, which we describe below, and of two

algorithms: MinimalSort [3, 21], which is *not* a stable natural merge sort, and whose merge policy is described in Algorithm 9; and MinimalStableSort, which *is* a stable natural merge sort, and whose merge policy can be computed by following either the Hu-Tucker [13] or Garsia-Wachs [9] algorithms for constructing optimal binary search trees.

Note that MinimalSort may require merging *non-adjacent* runs, which might therefore be less easy to implement than just merging adjacent runs. Yet, such merges (between runs of lengths m and n) can still be carried in time $m + n$, for instance by using linked lists, and therefore the merge cost remains an adequate measure of complexity for this algorithm.

Algorithm 9: MinimalSort

Input : Array to A to sort
Result: The array A is sorted into a single run.

```
runs ← the run decomposition of A
while runs contains at least two runs:
    merge the two shortest runs in runs
```

Definition 23. Let $\mathcal{M}$ be a merge policy and $\mathcal{R}$ a sequence of runs. We define the *merge tree* induced by $\mathcal{M}$ on $\mathcal{R}$ as the following binary rooted tree. Every node of the tree is identified with a run, either present in the initial sequence or created by the merge policy. The runs in the sequence $\mathcal{R}$ are the leaves of the tree, and when two runs R_1 and R_2 are merged together in a run $\overline{R}$, the run $\overline{R}$ is identified with the internal node whose children are R_1 and R_2: if the run R_1 was placed to the left of R_2 in the sequence to be sorted, then R_1 is the left sibling of R_2.

For example, Figure 3 presents the merge tree induced by adaptive ShiversSort on a sequence of runs of lengths $(2, 2m - 7, 2, m, 2, 2m, 1)$.

Any merge tree is the tree of a binary prefix encoding on a text on the alphabet $\{1, \dots, \rho\}$, which contains r_i characters i for all $i \leqslant \rho$. Furthermore, denoting by d_i the depth of the leaf R_i, both the length of this code and the merge cost of the associated merge policy are equal to $\sum_{i=1}^{\rho} d_i r_i$. Hence, character encoding algorithms unsurprisingly yield efficient sorting algorithms.

Similarly, if $\mathcal{M}$ is a stable merge policy, and considering a node R_i to be smaller than another node R_j whenever $i < j$, the merge tree induced by $\mathcal{M}$ on $\mathcal{R}$ is a binary search tree. Then, assuming that there will be r_i queries for finding the leaf R_i in that tree, the total cost of these queries is also equal to $\sum_{i=1}^{\rho} d_i r_i$, whence the usefulness of algorithms for constructing optimal binary search trees.

In particular, from now on, we identify every merge tree $\mathcal{T}$ with a collection of merges (these are the merges between the nodes R_1 and R_2 that are siblings in $\mathcal{T}$), and we define the *cost* of that

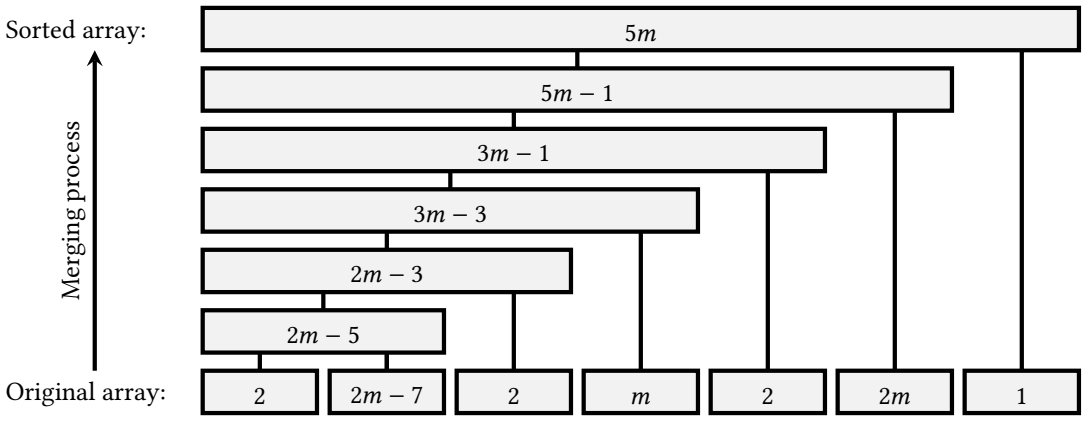


Fig. 3. Adaptive ShiversSort keeps merging the two leftmost runs of a 7-run array of length $n = 5m$, for a total merge cost $\mathrm{mc} = n(\mathcal{H} + \Delta + o(1))$. Each run (original or created) is labelled by its length.

tree as the sum of the costs of these merges. Equivalently, the cost of $\mathcal{T}$ is the sum of the lengths of the runs that are internal nodes of $\mathcal{T}$.

Theorem 24. *The merge cost of any (stable or not) merge policy on a non-sorted array is minimised by the algorithm MinimalSort, and the merge cost of any stable merge policy on a non-sorted array is minimised by the algorithm MinimalStableSort. Both costs are at least* $\max\{n, n\mathcal{H}\}$.

Proof. The merge trees of the algorithms MinimalSort and MinimalStableSort are a Huffman tree and an optimal binary search tree. This means that MinimalSort (respectively, MinimalStableSort) is indeed the natural (respectively, stable natural) merge sort with the least merge cost, and that this merge cost is the length of a Huffman code for a text containing r_i occurrences of the character i. Such a text has entropy $\mathcal{H}$, and therefore Shannon theorem proves that the associated Huffman code has length at least $n\mathcal{H}$. Furthermore, it is clear that the merge cost of MinimalSort must be at least n, which completes the proof. □

4.2 Optimality of worst-case merge costs

The lower bound provided by Theorem 24 matches quite well the worst-case merge costs of both PowerSort and adaptive ShiversSort. In particular, and independently of the sequence to be sorted, the merge cost of PowerSort (respectively, adaptive ShiversSort) lies between $n\mathcal{H}$ and $n(\mathcal{H} + 2)$ (respectively, $n(\mathcal{H} + \Delta)$).

This shows, among others, that both PowerSort and adaptive ShiversSort are very close to optimal when $\mathcal{H}$ is large. When $\mathcal{H}$ is small, however, the respective performances of PowerSort and adaptive ShiversSort are still worth investigating. In particular, and given the tiny margin of

freedom between these lower and upper bounds, it becomes meaningful to check whether our upper bounds are indeed optimal.

A first result in that direction is the following one, which implies that the worst-case merge costs of PowerSort and of length-adaptive ShiversSort are optimal.

Proposition 25. *Let $\mathcal{S}$ be a stable merge policy. Assume that there exist real constants α and β such that the merge cost of $\mathcal{S}$ can be bounded from above by $n(\alpha\mathcal{H} + \beta)$. We have $\alpha \geqslant 1$ and $\beta \geqslant 2$.*

Proof. First, Theorem 24 proves, by considering arbitrarily large values of $\mathcal{H}$, that $\alpha \geqslant 1$. Second, let $n \geqslant 6$ be some integer, and let $\mathbf{a}$ be some array of data that splits into runs of lengths 2, $n-4$ and 2. One checks easily that $n\mathcal{H} \leqslant 4\log_2(n) + 2$ and that every stable merge policy has a merge cost $2n - 2$ when sorting $\mathbf{a}$. Hence, considering arbitrarily large values of n proves that $\beta \geqslant 2$. □

While Proposition 25 indeed proves that the worst-case merge cost of PowerSort and of length-adaptive ShiversSort is optimal, addressing the optimality of the worst-case merge cost of adaptive ShiversSort requires considering another example.

Proposition 26. *Let β be a real constant such that the merge cost of adaptive ShiversSort can be bounded from above by $n(\mathcal{H} + \beta)$ when n is large enough. We have $\beta \geqslant \Delta$.*

Proof. Let $k \geqslant 3$ be some integer, and let $m = 2^k$, so that $m \geqslant 8$. Then, consider an array that decomposes into seven runs $R_1, \ldots, R_7$ with lengths 2, $2m-7$, 2, m, 2, $2m$ and 1 respectively. This array is represented in Figure 3. Its length is $n = 5m = 5 \cdot 2^k$, and its entropy is

$$\mathcal{H} = -\sum_{i=1}^{7} \log_2(r_i/n)\, r_i/n = \log_2(5) - 4/5 + o(1).$$

Meanwhile, the merge cost of adaptive ShiversSort is equal to $\mathrm{mc} = 20m - 13 = n\,(4 + o(1))$. Since $\mathrm{mc} \leqslant n(\mathcal{H} + \beta)$, it follows that $\beta \geqslant 4 - (\log_2(5) - 4/5) = \Delta$. □

Hence, the worst-case merge cost of adaptive ShiversSort is *not* optimal. This is not very surprising since, unlike PowerSort and length-adaptive ShiversSort, the algorithm adaptive ShiversSort cannot take into account the total length of the input until that end is indeed reached.

5 APPROXIMATELY OPTIMAL SORTING ALGORITHMS

In previous sections, we have shown that adaptive ShiversSort is both very easy to obtain by modifying the code of TimSort and very effective. Yet, and although adaptive ShiversSort is optimal up to an additive term of at most Δn, this term may still be of importance when considering arrays of data with small run-length entropy $\mathcal{H}$.

For example, in Table 2, we present the run lengths of arrays on which adaptive ShiversSort, PowerSort and TimSort have significantly different merge costs: the overhead of each algorithm,

compared to the others, can climb up to between 40% and 100% (at least), and each algorithm can be quite better or quite worse than the other two. This table also illustrates that, although they are similar to each other, the algorithms adaptive ShiversSort and length-adaptive ShiversSort, PowerSort and PeekSort, or even α-MergeSort for various values of α, can still have substantially behaviours on well-chosen arrays.

For space reasons, and using the convention that $n = 2^k$ and $k = 2^\ell$, we define here some sequences and merge costs mentioned in Table 2:

- the sequences $\mathbf{a}_{n,m} = (mn/4 - 1, mn/8 - 1, \dots, 2m - 1, m - 1, m - 2, 1, 1, \dots, 1)$ with k terms "1" at the end;
- the sequences $\mathbf{b}_{n,m} = (6n - 1, 1, 1, 3n - 2, 1, 1, 3n/2 - 2, 1, 1, 3n/4 - 1, \dots, 3 \cdot 2^{m+1} - 2, 1, 1, 3 \cdot 2^m - 3, 2, 1, 3 \cdot 2^m - 1)$;
- the sequences $\mathbf{c}_n = (4n + 2, 2n - 2, 1, 1, n - 2, 1, 1, n/2 - 2, \dots, 2^3 - 2, 1, 1, 2^2 - 2, 1, 1, 2)$;
- the sequences $\mathbf{d}_n = (7, 1, 1, 2^3 - 2, 1, 1, 2^4 - 2, 1, 1, \dots, n/2 - 2, 1, 1, 2n - 2, 1, 1, n/2 - 2, 1, 1, n/4 - 2, \dots, 2^4 - 2, 1, 1, 2^3 - 2, 1, 1, 7)$;
- the merge costs $\alpha_{n,m} = (2n - 2)m + k(2\ell + k - 3)$ and $\alpha'_{n,m} = (3n - 4)m + k(2\ell - k - 3) + 2$.

Moreover, arrays with small run-length entropy may have arbitrarily large lengths, but also arbitrarily many monotonic runs. This is, for instance, the case of arrays whose run lengths form the sequence $(2, 2, \dots, 2, k^2)$, where the k first terms are integers 2: although this sequence contains $k + 1$ terms, it is associated with a value of $\mathcal{H} \approx 4 \log_2(k)/k$.

Hence, and since the parameters n, ρ and $\mathcal{H}$ may vary more or less independently of each other (up to the rather loose inequalities $(\rho - 1) \log_2(n)/n \leqslant \mathcal{H} \leqslant \log_2(\rho) \leqslant \log_2(n)$), we aim for the uniform approximation result captured by the following definition.

Definition 27. Let $\mathcal{A}$ be a stable natural merge sort, and let $\varepsilon \geqslant 0$ be a real number. We say that $\mathcal{A}$ is *ε-optimal* if, for every stable natural merge sort $\mathcal{B}$ and every array to be sorted, the respective merge costs mc_a and mc_b of $\mathcal{A}$ and $\mathcal{B}$ satisfy the inequality $\mathsf{mc}_a \leqslant (1 + \varepsilon)\mathsf{mc}_b$.

Below, we study the ε-optimality of algorithms such as TimSort, adaptive ShiversSort, α-MergeSort, or even PowerSort and length-adaptive ShiversSort. To that aim, we first define the family of k-aware algorithms, to which all these algorithms belong; our notion subsumes and generalises slightly the notion of *awareness* of Buss and Knop [5]. While TimSort and adaptive ShiversSort are $(4, 3)$- and $(3, 3)$-aware algorithms in the sense of Buss and Knop, this novel notion also captures PowerSort and length-adaptive ShiversSort, which are respectively length-$(\infty, 3)$- and length-$(3, 3)$-aware algorithms. It fails, however, to capture PeekSort.

Definition 28. Let k and ℓ be elements of the set $\{0, 1, 2, \dots\} \cup \{\infty\}$, with $k \geqslant \ell$. A deterministic sorting algorithm is said to be *(k, ℓ)-aware* (or simply *k-aware* if $k = \ell$) if it sorts arrays of data

Table 2. Merge costs on a few sequences of run lengths. Overheads indicated hold when $n \to +\infty$. Gray cells indicate the worst performance (i.e., the largest overhead) of each algorithm.

Run lengths ($n = 2^k$, $k = 2^\ell$, $\ell \geqslant 2$)	Merge cost			
	adaptive ShiversSort	length-adaptive ShiversSort	PowerSort	PeekSort
$(2n-1, n, 1)$	$6n-1$ (50% overhead)	$4n+1$ (optimal)	$4n+1$ (optimal)	$4n+1$ (optimal)
$(2n, 2, 1, n-1, 1, 1, n-3, 3)$	$9n+16$ (optimal)	$10n+14$ (11% overhead)	$12n+8$ (33% overhead)	$12n+8$ (33% overhead)
$(2n-1, n/2, n/4, \ldots, 16, 8, 4, 2, 2, 1)$	$8n-5$ (60% overhead)	$6n-3$ (20% overhead)	$5n+k-2$ (optimal)	$5n+k-2$ (optimal)
Sequence $\mathbf{a}_{n,n+1}$ (see page 36)	$\alpha_{n,n+1}$ (optimal)	$\alpha'_{n,n+1}$ (50% overhead)	$\alpha_{n,n+1}$ (optimal)	$\alpha_{n,n+1}$ (optimal)
Sequence $\mathbf{a}_{n,n}$ (see page 36)	$\alpha'_{n,n}$ (50% overhead)	$\alpha'_{n,n}$ (50% overhead)	$\alpha_{n,n}$ (optimal)	$\alpha_{n,n}$ (optimal)
Sequence $\mathbf{b}_{n,2}$ (see page 36)	$30n+3k-23$ (optimal)	$33n+3k-25$ (10% overhead)	$42n-k-36$ (40% overhead)	$42n-k-36$ (40% overhead)
Sequence $\mathbf{b}_{n,1}$ (see page 36)	$42n-2k-21$ (40% overhead)	$33n+3k-10$ (10% overhead)	$42n-k-19$ (40% overhead)	$42n-k-19$ (40% overhead)
Sequence $\mathbf{c}_n$ (see page 36)	$19n+2k-12$ (6% overhead)	$19n+2k-12$ (6% overhead)	$18n+2k-10$ (optimal)	$22n-2k-14$ (22% overhead)
Sequence $\mathbf{d}_n$ (see page 36)	$13n+6k-23$ (optimal)	$13n+6k-23$ (optimal)	$18n-2k-21$ (38% overhead)	$14n+2k-18$ (8% overhead)

Run lengths ($n = 2^k$, $k = 2^\ell$, $\ell \geqslant 2$)	Merge cost			
	TimSort	2-MergeSort	ϕ-MergeSort ($\phi = (1+\sqrt{5})/2$)	Minimal StableSort
$(2n-1, n, 1)$	$4n+1$ (optimal)	$6n-1$ (50% overhead)	$4n+1$ (optimal)	$4n+1$
$(2n, 2, 1, n-1, 1, 1, n-3, 3)$	$12n+12$ (33% overhead)	$10n+14$ (11% overhead)	$12n+12$ (33% overhead)	$9n+16$
$(2n-1, n/2, n/4, \ldots, 16, 8, 4, 2, 2, 1)$	$6n-3$ (20% overhead)	$8n-5$ (60% overhead)	$6n-3$ (20% overhead)	$5n+k-28$
Sequence $\mathbf{a}_{n,n+1}$ (see page 36)	$4n^2-2n-6k-2$ (100% overhead)	$\alpha'_{n,n+1}$ (50% overhead)	$\alpha'_{n,n+1}$ (50% overhead)	$\alpha_{n,n+1}$
Sequence $\mathbf{a}_{n,n}$ (see page 36)	$4n^2-6n-6k+4$ (100% overhead)	$\alpha'_{n,n}$ (50% overhead)	$\alpha'_{n,n}$ (50% overhead)	$\alpha_{n,n}$
Sequence $\mathbf{b}_{n,2}$ (see page 36)	$30n+3k-23$ (optimal)	$33n+3k-25$ (10% overhead)	$30n+3k-23$ (optimal)	$30n+3k-23$
Sequence $\mathbf{b}_{n,1}$ (see page 36)	$30n+3k-8$ (optimal)	$33n+3k-10$ (10% overhead)	$30n+3k-8$ (optimal)	$30n+3k-8$
Sequence $\mathbf{c}_n$ (see page 36)	$18n+2k-10$ (optimal)	$22n+2k-10$ (22% overhead)	$18n+2k-10$ (optimal)	$18n+2k-10$
Sequence $\mathbf{d}_n$ (see page 36)	$14n+4k-23$ (8% overhead)	$14n+4k-24$ (8% overhead)	$14n+4k-24$ (8% overhead)	$13n+6k-23$

by manipulating a stack of runs (where each run is represented by its first and last indices) and operating as follows:

- the algorithm discovers, from the left to the right, the monotonic runs in which the array is split, and it pushes these runs on the stack when discovering them;
- the algorithm is allowed to merge two consecutive runs in the ℓ top runs of its stack only, and its decision may be based only on the lengths of the top k runs of the stack, and on whether the algorithm already discovered the entire array;
- if $\ell = \infty$, the algorithm may merge any two consecutive runs in its stack; if $k = \infty$, it is granted an infinite memory, and thus its decisions may be based all the push or merge operations it performed (and on the lengths of the runs involved in these operations).

If, furthermore, the algorithm is given access to the length of the array and can base its decisions on this information, then we say that it is a *length-(k, ℓ)-aware* algorithm.

In particular, note that PowerSort needs to remember not only the lengths of the array and of the runs stored in its stack, but also their powers (or, alternatively, the positions they span in the array), which does not fall into the scope of length-$(3, 3)$-awareness. Our new notion of awareness could be further generalised in a meaningful way that would make PowerSort a length-$(3, 3)$-aware algorithm. However, our results below already apply to all length-(∞, ℓ)-aware algorithms, and therefore such generalisations are not needed within the framework of this article.

We present now results going in opposite directions, and which, taken together, form a first step towards finding the best approximation factor of k-aware algorithms. One direction is explored in Section 5.1, where we prove that, once the integer k is fixed, k-aware algorithms cannot be ε-optimal for arbitrarily small values of ε. The other direction is explored in Section 5.2, where we prove that, for all ε, there exists a ε-optimal algorithm that is k-aware for some k.

5.1 Inapproximability bounds

Below, we present a few *inapproximability* results. We first investigate lower bounds on those numbers ε such that, for a given integer $k \geqslant 3$, there exists a ε-optimal algorithm that is k-aware (in Proposition 29) or length-(∞, k)-aware (in Proposition 30). Then, we focus on the case $k = 2$, and we prove that length-$(\infty, 2)$-aware algorithms cannot be ε-optimal for *any* ε.

Proposition 29. *Let $k \geqslant 3$ be an integer, and let*

$$\theta_k = 1/((10k + 12)\log_2(2k + 2)).$$

No k-aware sorting algorithm is θ_k-optimal.

Proof. Let $h = \lceil \log_2(k+2) \rceil$, and let $\rho = 2^h - 1$. We design six arrays $\mathbf{A}^{x,y}$, where $x \in \{1, 3\}$ and $y \in \{0, 2, 4\}$. Each array will have ρ runs, and we prove below that no k-aware algorithm can approach the merge cost of MinimalStableSort by a factor θ_k on those six arrays.

For all x and y, let $r_1 = x + 5$, $r_2 = \ldots = r_{\rho-1} = 5$, and $r_\rho = y + 5$. We build the array $\mathbf{A}^{x,y}$ as any array whose run lengths form the sequence $(r_1, r_2, \ldots, r_\rho)$. Thus, the length of this array is $n = 5\rho + x + y$.

The merge tree induced by MinimalStableSort on the array $\mathbf{A}^{x,y}$ is a Huffman tree. That tree is a perfectly balanced binary tree of height h, except that its two leftmost (if $x > y$) or rightmost (if $x < y$) leaves have been deleted, their parent thereby becoming a new leaf. Then, the merge cost of MinimalStableSort on the array $\mathbf{A}^{x,y}$ is simply $\mathsf{mc}_{\text{opt}} = hn - 5 - \max\{x, y\}$. In particular, note that $h \leqslant \log_2(k+1) + 1 = \log_2(2k+2)$, so that $\rho \leqslant 2k + 1$ and that $n \leqslant 5\rho + 7 \leqslant 10k + 12$. It follows that

$$\mathsf{mc}_{\text{opt}} = hn - 5 - \max\{x, y\} < hn \leqslant 1/\theta_k.$$

Hence, let $\mathcal{A}$ be some θ_k-optimal algorithm, and let mc_a be its merge cost on the array $\mathbf{A}^{x,y}$. Since $\mathsf{mc}_{\text{opt}} \leqslant \mathsf{mc}_a \leqslant (1 + \theta_k)\mathsf{mc}_{\text{opt}} < \mathsf{mc}_{\text{opt}} + 1$, it follows that $\mathsf{mc}_a = \mathsf{mc}_{\text{opt}}$. In particular, this means that $\mathcal{A}$ must sort $\mathbf{A}^{x,y}$ optimally, even in the class of not necessarily stable merge algorithms.

Let R_i denote the i^{th} run of $\mathbf{A}^{x,y}$. Then, let d_i denote the depth of the run R_i in the merge tree associated with $\mathcal{A}$, i.e., the number of merges into which the elements of R_i have been involved. We know that $\mathsf{mc}_a = \sum_{i=1}^{\rho-1} r_i d_i$ and, by minimality of mc_a, we must have $d_i \geqslant d_j$ whenever $r_i < r_j$; otherwise, by exchanging the runs R_i and R_j, we would strictly decrease the merge cost of $\mathcal{A}$ (although this might require merging non-adjacent runs). Now, we prove that $\mathcal{A}$ must merge the same pairs of runs as MinimalStableSort. We treat the case where $x < y$, the case $x > y$ being entirely analogous.

Let $\mathbf{D}$ be the largest of the integers d_i, and let $\mathbf{I}$ be the number of indices i such that $d_i = \mathbf{D}$. We have $\mathbf{D} \geqslant d_i \geqslant d_1 \geqslant d_\rho$ for all $i = 2, 3, \ldots, \rho - 1$. Then, if $d_\rho \neq \mathbf{D}$, let R_i and R_{i+1} be two runs that $\mathcal{A}$ merges with each other, and such that $d_i = d_{i+1} = \mathbf{D}$. The run resulting from the merge is (strictly) larger than R_ρ, and therefore, again by minimality of mc_a, we know that $d_\rho \geqslant d_i - 1 = \mathbf{D} - 1$.

Finally, Kraft equality states that $1 = \sum_{i=1}^{\rho} 2^{-d_i}$. Since every integer d_i is equal either to $\mathbf{D}$ or to $\mathbf{D} - 1$, this means that $1 = 2^{-\mathbf{D}}\mathbf{I} + 2^{1-\mathbf{D}}(\rho - \mathbf{I})$, i.e., that $2^{\mathbf{D}} = 2\rho - \mathbf{I}$. Since $1 \leqslant \mathbf{I} \leqslant \rho$, it follows that $2^{h+1} = 2\rho + 2 > 2^{\mathbf{D}} \geqslant \rho > 2^{h-1}$, and therefore we conclude that $\mathbf{D} = h$ and that $\mathbf{I} = 2\rho - 2^{\mathbf{D}} = 2^h - 2 = \rho - 1$. Hence, we know that $d_1 = \ldots = d_{\rho-1} = h$ and that $d_\rho = h - 1$. Remembering that $\mathcal{A}$ can merge adjacent runs only, this means that $\mathcal{A}$ must perform the same merges as MinimalStableSort, as announced above.

Let us now further assume that $\mathcal{A}$ is k-aware. Then, when scanning some array $\mathbf{A}^{x,y}$, it cannot distinguish between the arrays $\mathbf{A}^{x,x-1}$ and $\mathbf{A}^{x,x+1}$ until it discovers the rightmost run R_ρ. Moreover,

no two adjacent runs R_i and R_{i+1} would ever be merged by MinimalStableSort in both arrays $\mathbf{A}^{x,x-1}$ and $\mathbf{A}^{x,x+1}$. Thus, $\mathcal{A}$ must wait until discovering the run R_ρ before it can perform a single merge.

Imagine now that $y = 2$, i.e., $y = x + 1$ if $x = 1$, or $y = x - 1$ if $x = 3$. When discovering the run R_ρ, the run R_1 lies at the bottom of the stack, which contains $\rho \geqslant k + 1$ runs. Therefore, $\mathcal{A}$ cannot distinguish any more between the arrays $\mathbf{A}^{y-1,y}$ and $\mathbf{A}^{y+1,y}$. Once again, no two adjacent runs R_i and R_{i+1} would ever be merged by MinimalStableSort in both arrays $\mathbf{A}^{y-1,y}$ and $\mathbf{A}^{y+1,y}$.

This proves that $\mathcal{A}$ cannot make sure that it will perform the same merges as MinimalStableSort on all the arrays $\mathbf{A}^{x,y}$, which completes the proof. □

Proposition 30. *Let $k \geqslant 3$ be an integer, and let*

$$\varepsilon_k = 1/2^{k+7}.$$

No length-(∞, k)-aware sorting algorithm is ε_k-optimal.

Proof. The proof below follows similar lines as the proof of Proposition 29, but its details are often quite different.

We design two arrays $\mathbf{A}^-$ and $\mathbf{A}^+$ and prove below that no length-(∞, k)-aware algorithm can approach the merge cost of MinimalStableSort by a factor ε_k on those two arrays. The array $\mathbf{A}^-$ can be any array whose run lengths form the sequence $(r_1, r_2, \ldots, r_{2k+4})$ defined by $r_i = 2^{k+4-i}$ for $i = 1, 2, \ldots, k$; $r_{k+1} = 9$; $r_{k+2} = 8$; $r_{k+3} = 11$; $r_{k+4} = 18$; $r_{k+4+i} = 2^{i+4}$ for $i = 1, 2, \ldots, k-2$; $r_{2k+3} = 2^{k+2} + 2$; and $r_{2k+4} = 2^{k+3}$. The array $\mathbf{A}^+$ is any array whose run lengths form the sequence $(r_1, r_2, \ldots, r_{2k+2}, r_{\mathsf{end}})$, where $r_{\mathsf{end}} = r_{2k+3} + r_{2k+4} = 3 \times 2^{k+2} + 2$. Hence, the length of both arrays is $n = 9 \times 2^{k+2}$. In what follows, we will note R_i the i^{th} run of $\mathbf{A}^-$, and R_{end} the rightmost run of $\mathbf{A}^+$.

When sorting either array $\mathbf{A}^+$ or $\mathbf{A}^-$, the algorithm MinimalStableSort performs the same merges as those of a Huffman tree. That tree looks like a bunch of grapes, as illustrated in Figure 4. Thus, the merge costs of the algorithms MinimalStableSort and MinimalSort are equal to each other. These merge costs are $\mathsf{mc}^-_{\mathsf{opt}} = 29 \times 2^{k+2} - 2k - 48$ on the array $\mathbf{A}^-$, and $\mathsf{mc}^+_{\mathsf{opt}} = 26 \times 2^{k+2} - 2k - 45$ on the array $\mathbf{A}^+$: in both cases, we have $\mathsf{mc}_{\mathsf{opt}} < 2^{k+7} = 1/\varepsilon_k$.

Hence, let $\mathcal{A}$ be some ε_k-optimal algorithm. Since $\mathsf{mc}_{\mathsf{opt}} \leqslant \mathsf{mc}_a \leqslant (1 + \varepsilon_k)\mathsf{mc}_{\mathsf{opt}} < \mathsf{mc}_{\mathsf{opt}} + 1$, it follows that $\mathsf{mc}_a = \mathsf{mc}_{\mathsf{opt}}$. In particular, this means that $\mathcal{A}$ must sort $\mathbf{A}^\pm$ optimally, even in the class of not necessarily stable merge algorithms. We prove now the following claims:

(i) when sorting $\mathbf{A}^-$, the algorithm $\mathcal{A}$ must merge the runs R_{k+2} and R_{k+3};
(ii) when sorting $\mathbf{A}^+$, the algorithm $\mathcal{A}$ must merge the run R_{k+3} successively with the runs $R_{k+4}, R_{k+5}, \ldots, R_{2k+2}, R_{\mathsf{end}}$.

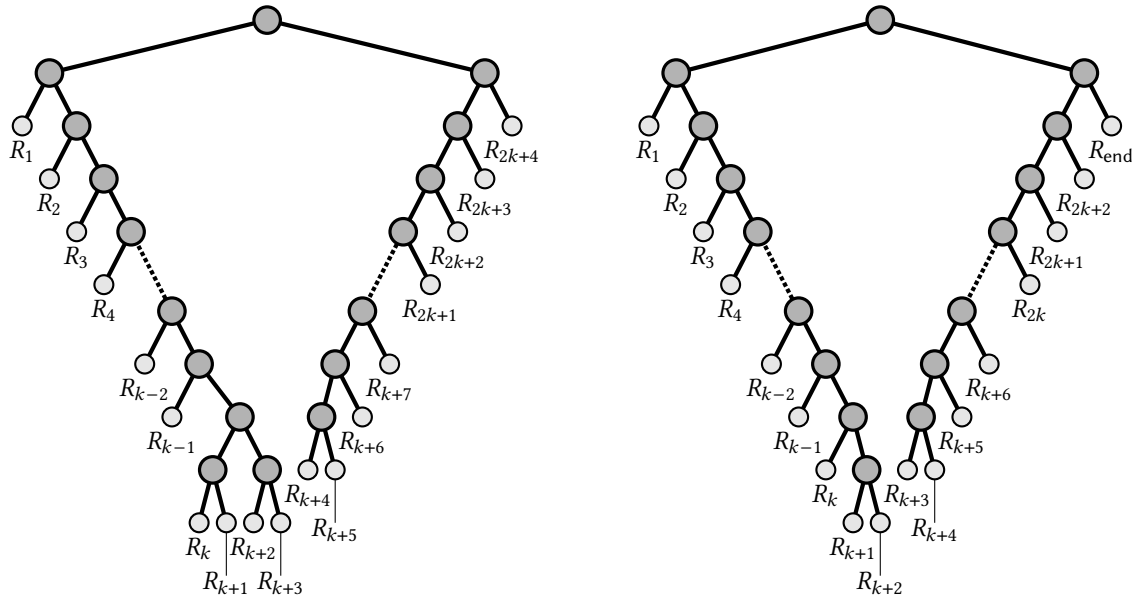


Fig. 4. Merge trees associated with MinimalStableSort when sorting $\mathbf{A}^-$ (left) and $\mathbf{A}^+$ (right).

Let us first prove Claim (i). In order to do so, let us note d_i the depth of the run R_i in the merge tree associated with $\mathcal{A}$ when sorting $\mathbf{A}^-$. Since $\mathcal{A}$ is optimal among all merge algorithms when sorting $\mathbf{A}^-$, we can prove the following statement: if R and R' are two runs ever manipulated by $\mathcal{A}$ (either because the are runs of the original array $\mathbf{A}^-$ or because they result from some merge operation), and if $r < r'$, then $d \geqslant d'$. Indeed, both inequalities $r < r'$ and $d < d'$ were to hold, exchanging the runs R and R' would provide us with a smaller total merge cost.

Since R_{k+1} and R_{k+2} are the two shortest runs of the array, they must be the deepest runs as well, and thus $d_{k+1} = d_{k+2}$. Then, the run R_{k+2} must be merged with either R_{k+1} or R_{k+3}. For the sake of contradiction, assume that R_{k+1} and R_{k+2} were merged with each other, and let $\overline{R}$ be the run resulting from that merge.

We know that $r_{k+4} > \overline{r} = 17 > r_{k+3}$, and therefore that $d_{k+4} \leqslant \overline{d} = d_{k+1} - 1 \leqslant d_{k+3} \leqslant d_{k+1}$. Then, since, $d_{k+1} - d_{k+3} \equiv \mathrm{mc}^-_{\mathrm{opt}} \equiv 0 \pmod 2$, it follows that $d_{k+3} = d_{k+1}$. Consequently, R_{k+3} is also the deepest run of the array, and since it is not merged with R_{k+2}, it must be merged with R_{k+4}. But this is impossible since $d_{k+4} \geqslant \overline{d} > d_{k+3}$. Hence, our assumption was incorrect, which proves that $\mathcal{A}$ must merge R_{k+2} with R_{k+3}.

Now that Claim (i) is proved, let us also prove Claim (ii). Like above, d_i denotes the depth of the run R_i, and d_{end} denotes the depth of the run R_{end}. Once again, the runs R_{k+1} and R_{k+2} are the deepest runs of the array, and thus $d_{k+1} = d_{k+2}$. Yet, this time, $d_{k+1} - d_{k+3} \equiv \mathrm{mc}^+_{\mathrm{opt}} \equiv 1 \pmod 2$,

and therefore $d_{k+3} < d_{k+1}$. Hence, the runs R_{k+1} and R_{k+2} must be merged together.

Below, let us note $\overline{R}$ the parent of every run R. Since $\overline{r}_{k+1} = 17 > r_k > r_{k+3}$, we have $\overline{d}_{k+1} \leqslant d_k \leqslant d_{k+3}$. Then, since $d_{k+3} < d_{k+1} = \overline{d}_{k+1} + 1$, this even proves that $\overline{d}_{k+1} = d_k = d_{k+3}$. Similarly, for all $i \leqslant k-1$ or $i \geqslant k+4$, we know that $r_i > \overline{r}_{k+1}$, and thus that $d_i \leqslant \overline{d}_{k+1}$. Consequently, the run R_{k+3} must be merged with either $\overline{R}_{k+1}$ or R_{k+4}.

In both cases, we have $\overline{r}_{k+3} \leqslant r_{k+3} + \max\{\overline{r}_{k+1}, r_{k+4}\} = 29$, and therefore $r_i > \overline{r}_{k+3}$ for all $i \leqslant k-1$ or $i \geqslant k+5$, thereby proving that $d_i \geqslant \overline{d}_{k+3}$. Consequently, the only runs that lie at depth $\overline{d}_{k+1}$ are R_k, $\overline{R}_{k+1}$, R_{k+3}, and possibly R_{k+4}. Since there must be an even number of such runs, it follows at once that all four runs lie at depth $\overline{d}_{k+1}$ and that $\mathcal{A}$ merges R_k with $\overline{R}_{k+1}$, and R_{k+3} with R_{k+4}.

Let us further prove that $\mathcal{A}$ merges R_{k-1} with $\overline{R}_k$, and $\overline{R}_{k+4}$ with R_{k+5}. Indeed, both runs $\overline{R}_k$ and $\overline{R}_{k+4}$ lie at depth $\overline{d}_k$, and since $\overline{r}_k = 2^5 + 1 > r_{k-1} = r_{k+5} > 2^5 - 3 = \overline{r}_{k+4}$, the runs R_{k-1} and R_{k+5} also lie at depth $\overline{d}_k$. Hence, the run $\overline{R}_{k+4}$ will be merged with either $\overline{R}_k$ or R_{k+5}. In both cases, we have $\overline{\overline{r}}_{k+4} \leqslant \overline{r}_{k+4} + \max\{\overline{r}_k, r_{k+5}\} < 2^6 \leqslant r_i$ for all $i \leqslant k-2$ or $i \geqslant k+6$. Consequently, the runs R_{k-1}, $\overline{R}_k$, $\overline{R}_{k+4}$ and R_{k+5} are the only runs at depth $\overline{d}_k$, and $\mathcal{A}$ merges R_{k-1} with $\overline{R}_k$, and $\overline{R}_{k+4}$ with R_{k+5}.

Repeating the same arguments verbatim, we prove that $\mathcal{A}$ merges R_{k-1-i} with $\overline{R}_{k-i}$, and $\overline{R}_{k+4+i}$ with R_{k+5+i}, for all $i = 0, 1, \ldots, k-2$. Eventually, we end up with the runs R_0, $\overline{R}_1$, $\overline{R}_{2k+2}$ and R_{end}, whose lengths are $r_0 = 2^{k+3}$, $\overline{r}_1 = 2^{k+3} + 1$, $\overline{r}_{2k+2} = 2^{k+3} - 1$ and $r_{\mathsf{end}} = 3 \times 2^{k+2} + 2$. The only optimal way to merge these runs is to merge R_0 with $\overline{R}_1$, and $\overline{R}_{2k+2}$ with $\overline{R}_{\mathsf{end}}$. This completes the proof of Claim (ii).

Let us now further assume that $\mathcal{A}$ is length-(∞, k)-aware. When scanning either array $\mathbf{A}^{\pm}$, and since both arrays $\mathbf{A}^-$ and $\mathbf{A}^+$ have the same length, $\mathcal{A}$ cannot distinguish between them until it discovers the run R_{2k+3} or R_{end}. In particular, it must wait until discovering that run before it can merge the run R_{k+3}.

Imagine now that $\mathcal{A}$ is sorting the array $\mathbf{A}^+$. When $\mathcal{A}$ merges the run R_{k+3}, the k runs $R_{k+4}, \ldots, R_{2k+2}, R_{\mathsf{end}}$ have already been pushed onto the stack, and none of them can be merged before R_{k+3} is merged. Thus, if the algorithm $\mathcal{A}$ is to merge R_{k+3}, it could in fact not be length-(∞, k)-aware, which completes the proof. □

Unsurprisingly, this result can be strengthened dramatically in the case of length-$(\infty, 2)$-aware algorithms, which are not ε-optimal for any ε.

Lemma 31. *Consider the following dynamic system. Starting with one empty stack $\mathcal{S}$, we successively perform operations of the following type: either (i) if $\mathcal{S}$ contains at least two elements, we may remove the top element of the stack, or (ii) if we have already pushed the integers $0, 1, \ldots, \ell - 1$ onto $\mathcal{S}$ (some of which may have been removed), we may push the integer ℓ on the top of the stack.*

Then, for all integers h and all functions $f : \mathbb{Z}_{\geqslant 0} \mapsto \mathbb{Z}_{\geqslant 0}$, there exists an integer $m_{f,h}$ such that, when the integer $m_{f,h}$ is pushed onto the stack, either the stack $\mathcal{S}$ has been of height h at some point, or some integer k has been the top element of the stack after $f(k)$ operations of type (i).

Proof. Since, at every step, we have the choices between options (i) and (ii), the set of executions of the system can be seen as an infinite tree in which every node has two children, one per option. Let us restrict this tree to the subtree $\mathcal{T}$ containing those (finite or infinite) executions where the stack is always of height smaller than h, and where no integer k ever becomes the top element after $f(k)$ or more operations of type (i).

First, we show that every branch (i.e., execution) of $\mathcal{T}$ is finite. Indeed, consider some infinite execution where the stack height is always smaller than h. At most $h-1$ integers will stay forever on the stack after they have been pushed onto it, and 0 is one of these integers. Hence, let k be the largest such integer. Every integer ever placed just above k must have been removed from the stack at some point, hence k must have been the top element of the stack infinitely many times. Therefore, this execution is not a branch of $\mathcal{T}$.

Finally, since the branching degree of $\mathcal{T}$ is 2, and since its branches are all finite, König's lemma proves that $\mathcal{T}$ itself is finite. Defining $m_{f,h}$ as the maximal length of $\mathcal{T}$'s branches completes the proof. □

Proposition 32. *Let $\mathcal{A}$ be a length-$(\infty, 2)$-aware stable merge sort algorithm. The worst-case merge cost of $\mathcal{A}$ is bounded from below by $\omega(n(\mathcal{H}+1))$. In particular, $\mathcal{A}$ is not ε-optimal for any real number ε.*

Proof. For the sake of contradiction, we assume in the entire proof that there exist an integer $\mathbf{z}$ and a length-$(\infty, 2)$-aware algorithm $\mathcal{A}$ whose merge cost is bounded from above by $\mathbf{z}n(\mathcal{H}+1)$. Then, for all integers $k \leqslant \ell$, let $\mathbf{A}^{k,\ell}$ be an array that decomposes into $k+1$ runs $R_0, R_1, \ldots, R_k$ of respective lengths $2^{\ell-1}, 2^{\ell-2}, \ldots, 2^{\ell-k}$ and $2^\ell + 2^{\ell-k}$: the array $\mathbf{A}^{k,\ell}$ has length $2^{\ell+1}$.

The entropy of any array $\mathbf{A}^{k,\ell}$ is defined by

$$\mathcal{H}^{k,\ell} = \sum_{i=2}^{k+1} \frac{1}{2^i} \log_2(2^i) - \frac{1+2^{-k}}{2} \log_2\left(\frac{1+2^{-k}}{2}\right) < \sum_{i \geqslant 2} \frac{i}{2^i} + 1 = \frac{5}{2},$$

and therefore the cost of those merges used by $\mathcal{A}$ for sorting $\mathbf{A}^{k,\ell}$ is smaller than $7\mathbf{z}2^\ell$. Moreover, if the stack of $\mathcal{A}$ is of height h when the last run of $\mathbf{A}^{k,\ell}$ is discovered, then that run will take part in h merges, for a total cost of $2^\ell h$ at least. It follows that $h < 7\mathbf{z}$.

However, once the integer ℓ is fixed, and provided that the algorithm $\mathcal{A}$ is executed on some array $\mathbf{A}^{k,\ell}$, it cannot distinguish between the arrays $\mathbf{A}^{0,\ell}, \ldots, \mathbf{A}^{\ell,\ell}$ until it discovers their last run (or the second last run of $\mathbf{A}^{\ell,\ell}$). In particular, if, when treating some array $\mathbf{A}^{k,\ell}$, and just before pushing its i^{th} run (with $i \leqslant k$), the stack of $\mathcal{A}$ turns out to be of height $h \geqslant 7\mathbf{z}$, then $\mathcal{A}$ might

as well discover that it was, in fact, treating the array $\mathbf{A}^{i,\ell}$, contradicting the previous paragraph. Therefore, the stack of $\mathcal{A}$ may *never* be of height $7\mathbf{z}$ or more.

At the same time, if the elements of some run R_j take part in $7\mathbf{z}2^{j+1}$ merges, then of course these merges have a total cost of $7\mathbf{z}2^\ell$ at least. Hence, the elements of every run R_j can take part in at most $f(j)$ merges, where $f(j) = 7\mathbf{z}2^{j+1}$. Furthermore, our stack follows exactly the dynamics described in Lemma 31, where choosing the option (i) means that we merge the top two elements of the stack: if the element j becomes the new top element after such an operation, this means that we have just merged the elements of R_j (among others). It follows from Lemma 31 that $\ell \leqslant m_{f,7\mathbf{z}}$.

Consequently, and when ℓ is large enough, we know that there must exist arrays on which the merge cost of $\mathcal{A}$ is at least $\mathbf{z}n(\mathcal{H} + 1)$. In particular, since PowerSort would have sorted $\mathcal{A}$ for a cost of $n(\mathcal{H} + 2)$, it follows that $\mathcal{A}$ is not $(\mathbf{z}/2 - 1)$-optimal. □

5.2 Approximability bounds

We focus now on one *approximability* result, which states that there exists a k-aware algorithm that matches some approximability bound. Furthermore, this result has a constructive proof, which consists in providing an effective algorithm and proving that this algorithm indeed matches the approximability bound.

Theorem 33. *Let $k \geqslant 8$ be an integer, and let*

$$\eta_k = (\Delta + 7)/\log_2((k-3)/4),$$

where we recall that $\Delta = 24/5 - \log_2(5)$. There exists a k-aware sorting algorithm that is η_k-optimal.

Note that, if $k \geqslant 3$, then Theorems 15 and 24 already prove that adaptive ShiversSort, which is a 3-aware (and thus a k-aware) algorithm, is also Δ-optimal. In particular, numerical computations show that $\Delta \leqslant \eta_k$ when $k \leqslant 59$, which already proves Theorem 33 in that case. Furthermore, and although the constants θ_k and η_k become arbitrarily small when $k \to \infty$, there is still an exponential gap between these constants.

Our proof consists in showing that the parametrised algorithm κ-stack ShiversSort depicted in Algorithm 10, which is visibly $(2\kappa + 2)$-aware, is $\eta_{2\kappa+3}$-optimal. In order to do so, we go through several phases:

(1) In Section 5.2.1, we prove that adaptive ShiversSort enjoys *independence* or *stability* properties, which describe the behaviour of adaptive ShiversSort if some merges have been performed in advance, or if one uses adaptive ShiversSort to sort a sub-array instead of the entire array.
(2) In Section 5.2.2, we prove a variant of Proposition 18 that characterises the behaviour of κ-stack ShiversSort during its so-called *phase 1*, which consists of those merges performed in lines 7, 11 or 28.

Algorithm 10: κ-stack ShiversSort and its auxiliary functions

Input : An array to A to sort, integer parameter κ
Result: The array A is sorted into a single run, which remains on the stack.

Note: Whenever two successive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, by the run resulting from the merge. In practice, in $\mathcal{S}$, each run is represented by a pair of pointers to its first and last entries.

```
1  runs ← the run decomposition of A
2  S ← an empty stack
3  while true:                                                    ▷ main loop
4     if h = 2κ + 1, runs ≠ ∅ and mustMerge:                      ▷ phase 1
5        doMerge
6     else if h ⩾ 2κ + 2 and ℓ_{h−2} ⩽ max{ℓ_{h−1}, ℓ_h}:
7        merge the runs R_{h−2} and R_{h−1}
8     else if runs ≠ ∅:
9        remove a run R from runs and push R onto S
10    else if h ⩾ 2κ + 2:
11       merge the runs R_{h−1} and R_h
12    else if h ⩾ 2:                                              ▷ phase 2
13       perform the first merge prescribed by MinimalStableSort
14    else:
15       break
```

```
16 Function mustMerge:                                  ▷ called only with h = 2κ + 1
17    r_large ← ⌊(r_1 + r_2 + ... + r_{2κ})/κ⌋
18    ℓ_large ← ⌊log_2(r_large)⌋
19    for i ← 1, 2, ..., 2κ − 1:
20       if ℓ_i ⩽ max{ℓ_{i+1}, ℓ_{i+2}} and max{ℓ_i, ℓ_{i+1}} ⩽ ℓ_large:
21          return true
22    return false
```

```
23 Function doMerge:                                    ▷ called only with h = 2κ + 1
24    r_large ← ⌊(r_1 + r_2 + ... + r_{2κ})/κ⌋
25    ℓ_large ← ⌊log_2(r_large)⌋
26    for i ← 1, 2, ..., 2κ − 1:
27       if ℓ_i ⩽ max{ℓ_{i+1}, ℓ_{i+2}} and max{ℓ_i, ℓ_{i+1}} ⩽ ℓ_large:
28          merge the runs R_i and R_{i+1}
29          break
```

(3) In Section 5.2.3, we introduce an object called the *least high border* of a merge tree, which consists of *large enough* runs. We prove that this object is a reasonable approximation of an actual state of MinimalStableSort, and we evaluate the quality of this approximation.

(4) In Section 5.2.4, we use this approximation to find good upper bounds on the merge cost of κ-stack ShiversSort, thereby proving Theorem 33 itself.

5.2.1 Independence and stability properties. In this section, and before focusing on κ-stack ShiversSort itself, we explore some behavioural properties of the algorithm adaptive ShiversSort. We start this exploration by recalling some key tools already introduced in previous sections: the notions of *state*, *merge point*, *successor* and *merge tree*.

Definition 16. Consider the execution of the algorithm c-adaptive ShiversSort on a sequence of runs to be sorted. At each step, the algorithm handles both a stack $\mathcal{S} = (R_1, \dots, R_h)$ of runs and a sequence $\mathcal{R} = (R_{h+1}, \dots, R_t)$ of those runs that have yet to be discovered and pushed onto the stack. We call *state* of the algorithm, at that step, the sequence $(R_1, \dots, R_t)$, i.e., the concatenation of $\mathcal{S}$ and $\mathcal{R}$.

Finally, two states of the algorithm are said to be *consecutive* if they are distinct from each other and were separated by a single (run push or merge) operation performed by the algorithm.

Definition 17. Let $\mathcal{R} = (R_1, \dots, R_t)$ be a sequence of runs of length $t \geqslant 2$. We say that an integer x is *dominated* in the sequence $\mathcal{R}$ if $1 \leqslant x \leqslant t-1$ and $\ell_x \leqslant \max\{\ell_{x+1}, \ell_{x+2}\}$, with the convention that $\ell_{t+1} = \infty$ (we may omit mentioning $\mathcal{R}$ when the context is clear). This convention ensures us that $t-1$ is necessarily dominated.

Then, let k be the smallest dominated integer. We say that k is the *merge point* of the sequence $\mathcal{R}$. Finally, we call *successor* of $\mathcal{R}$, and note $\mathrm{succ}(\mathbf{R})$, the sequence of runs

$$(R_1, \dots, R_{k-1}, \overline{R}, R_{k+2}, \dots, R_t),$$

where $\overline{R}$ is the run obtained by merging R_k and R_{k+1}.

Proposition 18. *Let* S *and* $\overline{\mathrm{S}}$ *be two consecutive states encountered during an execution of c-adaptive ShiversSort. We have* $\overline{\mathrm{S}} = \mathit{succ}(\mathrm{S})$.

Definition 23. Let $\mathcal{M}$ be a merge policy and $\mathcal{R}$ a sequence of runs. We define the *merge tree* induced by $\mathcal{M}$ on $\mathcal{R}$ as the following binary rooted tree. Every node of the tree is identified with a run, either present in the initial sequence or created by the merge policy. The runs in the sequence $\mathcal{R}$ are the leaves of the tree, and when two runs R_1 and R_2 are merged together in a run $\overline{R}$, the run $\overline{R}$ is identified with the internal node whose children are R_1 and R_2: if the run R_1 was placed to the left of R_2 in the sequence to be sorted, then R_1 is the left sibling of R_2.

Proposition 18 clearly marks *states* as a crucial object for studying the dynamics of adaptive ShiversSort, since they seem to capture the right amount of information manipulated by the algorithm at a given point in time. On the other hand, *merge trees* are generic objects, which may

be constructed for every (stable) merge policy, and allow us to capture at once the entire list of merges performed by the algorithm, with little regard for the temporality of these merges: when two runs in the tree are not related by the ancestor relation, and if neither is a leaf of the tree, we cannot know which run was created first.

A natural further step is then to bind both of these objects: this is the aim of *borders*, illustrated in Figure 5.

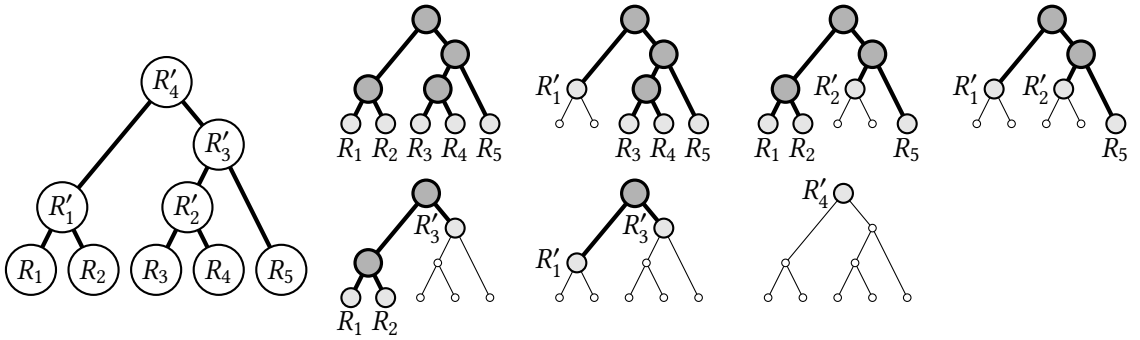


Fig. 5. A merge tree with seven borders, and corresponding ancestor sub-trees.

Definition 34. Let $\mathcal{R} = (R_1, \ldots, R_\rho)$ be a sequence of runs, and let $\mathcal{T}$ be a merge tree induced by some merge sort on $\mathcal{R}$. A *border* of $\mathcal{T}$ is a maximal set $\mathcal{B}$ of pairwise incomparable (for the strict ancestor relation) nodes of $\mathcal{T}$, i.e., a maximal set $\mathcal{B}$ such that no strict ancestor of a node in $\mathcal{B}$ belongs to $\mathcal{B}$. We call the set of ancestors of nodes in $\mathcal{B}$, including nodes in $\mathcal{B}$ themselves, the *ancestor sub-tree* of $\mathcal{B}$ in $\mathcal{T}$.

Every border or, more generally, every set of pairwise incomparable nodes of a merge tree $\mathcal{T}$, is naturally endowed with a left-to-right ordering. Hence, we will often identify a border $\mathcal{B}$ with the unique sequence of runs $(R'_1, \ldots, R'_k)$ such that each R'_i lies to the left of R'_{i+1} and such that $\mathcal{B} = \{R'_1, \ldots, R'_k\}$. In particular, every state encountered during an execution of adaptive ShiversSort is a border of $\mathcal{R}$.

Theorem 35. *Let $\mathcal{T}$ be the merge tree induced by adaptive ShiversSort on a sequence of runs $\mathcal{R}$, and let $\mathcal{R}'$ be a border of $\mathcal{T}$. The merge tree induced by adaptive ShiversSort on $\mathcal{R}'$ coincides with the ancestor sub-tree of $\mathcal{R}'$ in $\mathcal{T}$.*

Proof. Let $(R_1, \ldots, R_\rho)$ be our sequence of runs $\mathcal{R}$, and let $\rho = |\mathcal{R}|$ and $s = |\mathcal{R}'|$. Then, if $\rho \geqslant 2$, let k be the merge point of $\mathcal{R}$, and let $\overline{R}$ be the run obtained by merging R_k and R_{k+1}, i.e., their parent in the tree $\mathcal{T}$; if $\rho = 1$, we just set $k = 0$. Since $k \leqslant \rho$, we prove Theorem 35 by induction on the triple $(\rho, \rho - s, k)$.

First, if $\rho = s$, we have $\mathcal{R} = \mathcal{R}'$, and the result is immediate.

Second, let us assume that $\rho = s + 1$. In that case, there exists an integer $i \leqslant \rho - 1$ such that $\mathcal{B} = \mathcal{R} \setminus \{R_i, R_{i+1}\} \cup \{R'\}$, where R' is the run obtained by merging R_i and R_{i+1}, i.e., their parent in $\mathcal{T}$. By minimality of k, we have $i \geqslant k$, and since R_k and R_{k+1} are already merged with each other, we cannot have $i = k + 1$. Thus, either $i = k$ or $i \geqslant k + 2$:

- If $i = k$, Proposition 18 proves that merging the runs R_i and R_{i+1} into one single run R' is the first merge operation performed while applying adaptive ShiversSort on the sequence $\mathcal{R}$. Thus, for all $a \geqslant 1$, the a^{th} merge performed while executing adaptive ShiversSort on $\mathcal{R}'$ is also the $(a+1)^{\text{th}}$ merge performed while executing adaptive ShiversSort on $\mathcal{R}$. This completes the proof in that case.
- If $i \geqslant k + 2$, let also $\overline{\mathcal{R}} = \mathcal{R} \setminus \{R_k, R_{k+1}\} \cup \{\overline{R}\}$ and $\overline{\mathcal{R}}' = \mathcal{R} \setminus \{R_k, R_{k+1}, R_i, R_{i+1}\} \cup \{\overline{R}, R'\}$ be borders of $\mathcal{T}$. Let $\mathcal{T}'$, $\overline{\mathcal{T}}$ and $\overline{\mathcal{T}}'$ be the merge trees induced by adaptive ShiversSort on $\mathcal{R}'$, $\overline{\mathcal{R}}$ and $\overline{\mathcal{R}}'$. We will show that they are all ancestor sub-trees of $\mathcal{T}$, as illustrated in Figure 6.

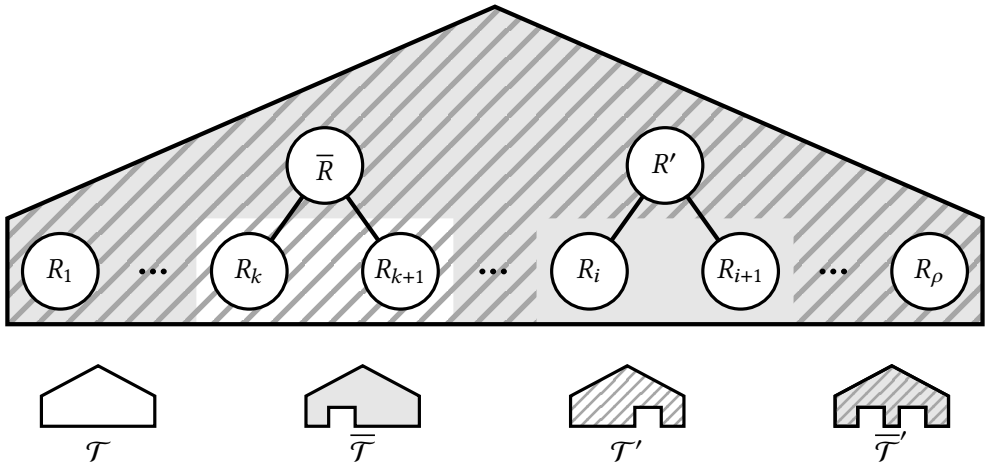


Fig. 6. Embedding the trees $\mathcal{T}'$, $\overline{\mathcal{T}}$ and $\overline{\mathcal{T}}' = \mathcal{T}' \cap \overline{\mathcal{T}}'$ as sub-trees of $\mathcal{T}$.

The induction hypothesis proves that $\overline{\mathcal{T}}$ is the ancestor sub-tree of $\overline{\mathcal{R}}$ in $\mathcal{T}$. Since $\overline{\mathcal{R}}'$ is a border of $\overline{\mathcal{T}}$, the induction hypothesis also proves that $\overline{\mathcal{T}}'$ is the ancestor sub-tree of $\overline{\mathcal{R}}'$ in $\overline{\mathcal{T}}$ (and thus in $\mathcal{T}$ too).

Then, Proposition 18 proves that the runs R_k and R_{k+1} are siblings in $\mathcal{T}'$. It follows that $\overline{\mathcal{R}}'$ is a border of $\mathcal{T}'$ and, by induction hypothesis, that $\overline{\mathcal{T}}'$ is the ancestor sub-tree of $\overline{\mathcal{R}}'$ in $\mathcal{T}'$. Hence, $\overline{\mathcal{T}}'$ is a sub-tree of both trees $\mathcal{T}$ and $\mathcal{T}'$. Then, since the only nodes of $\mathcal{T}'$ that do not belong to $\mathcal{T}'$ are siblings in $\mathcal{T}$ (these are the leaves R_i and R_{i+1}), it means that $\mathcal{T}'$ is itself a sub-tree of $\mathcal{T}$, as desired.

Finally, we assume that $\rho \geqslant s+2$. Let R be an internal node of $\mathcal{T}$ that belongs to $\mathcal{R}'$, and let $R_\oplus$ and $R_\ominus$ be its children. The set $\mathcal{R}'' = \mathcal{R}' \setminus \{R\} \cup \{R_\oplus, R_\ominus\}$ is a border of $\mathcal{T}$ of cardinality $s+1$. Let also $\mathcal{T}'$ and $\mathcal{T}''$ be the merge trees respectively induced by adaptive ShiversSort on $\mathcal{R}'$ and $\mathcal{R}''$.

The induction hypothesis proves that $\mathcal{T}''$ is the ancestor sub-tree of $\mathcal{R}''$ in $\mathcal{T}$. Therefore, $\mathcal{R}'$ is a border of $\mathcal{T}''$, and the induction hypothesis proves that $\mathcal{T}'$ is the ancestor sub-tree of $\mathcal{R}'$ in $\mathcal{T}''$. Consequently, $\mathcal{T}'$ is also the ancestor sub-tree of $\mathcal{R}'$ in $\mathcal{T}$. □

When $\mathcal{R}'$ is a state encountered while executing adaptive ShiversSort on the sequence $\mathcal{R}$, Theorem 35 means that interrupting and then restarting this execution would not change the list of those merges that adaptive ShiversSort performs. However, $\mathcal{R}'$ does not need to be such a state. Thus, Theorem 35 provides us with the following remarkable *stability* property of adaptive ShiversSort.

Corollary 36. *Let $\mathcal{A}$ be some stable natural merge sort algorithm, which consists in (i) applying some (arbitrary) sequence of merges that would have been performed by adaptive ShiversSort, and then (ii) applying adaptive ShiversSort on the resulting state. The algorithm $\mathcal{A}$ simply performs the same merges as adaptive ShiversSort, although in a possibly different order.*

It also provides us with the following *independence* property.

Proposition 37. *Let $\mathcal{T}$ be the merge tree induced by adaptive ShiversSort on a sequence $\mathcal{R}$, let R be a node of the tree $\mathcal{T}$, and let $\mathcal{R}'$ be the subsequence of $\mathcal{R}$ that consists of those runs that descend from R. The sub-tree of $\mathcal{T}$ rooted at R coincides with the merge tree induced by adaptive ShiversSort on the sequence $\mathcal{R}'$.*

Proof. Let d be the depth of the node R in the tree $\mathcal{T}$, let ρ be the number of leaves of $\mathcal{T}$, and let $\mathcal{T}'$ be the sub-tree of $\mathcal{T}$ rooted at R. We prove Proposition 37 by induction on the pair (d, ρ). First, if $d = 0$, then R is the root of $\mathcal{T}$, and the result clearly holds.

Second, if $d = 1$, let S be the sibling of R. If S is a leaf of $\mathcal{T}$, it is either the leftmost or rightmost element of $\mathcal{R}$. Therefore, replacing S by an arbitrarily large run would not alter the dynamics of adaptive ShiversSort, and thus the conclusion of Proposition 37 comes immediately.

If, however, S is an internal node of $\mathcal{T}$, let $\mathcal{B}$ be the border of $\mathcal{T}$ that consists of $\mathcal{R}'$ (i.e., the leaves of $\mathcal{T}'$) and of the node S: this border contains at most $\rho - 1$ nodes. Theorem 35 states that the merge tree $\mathcal{T}''$ induced by adaptive ShiversSort on the sequence $\mathcal{B}$ coincides with the ancestor sub-tree of $\mathcal{B}$ in $\mathcal{T}$, and thus $\mathcal{T}'$ is also the sub-tree of $\mathcal{T}''$ rooted at R. Hence, the induction hypothesis proves the desired result in that case too.

Finally, if $d \geqslant 2$, let R° be the parent of R, let $\mathcal{T}^\circ$ be the sub-tree of $\mathcal{T}$ rooted at R°, and let $\mathcal{R}^\circ$ be the subsequence of $\mathcal{R}$ that consists of those runs that descend from R°, i.e., of those leaves of $\mathcal{T}^\circ$. The induction hypothesis proves first that $\mathcal{T}^\circ$ coincides with the merge tree induced by

adaptive ShiversSort on the sequence $\mathcal{R}^\circ$, and then that $\mathcal{T}'$ coincides with the merge tree induced by adaptive ShiversSort on the sequence $\mathcal{R}'$. □

Note that MinimalStableSort also enjoy the stability and independence properties mentioned in Corollary 36 and Proposition 37. Actually, and except in the rare case where several merge trees have the optimal merge cost, every optimal merge-sort must have these properties.

5.2.2 Phase 1 in κ-stack ShiversSort. In this section, we study more closely the behaviour of κ-stack ShiversSort. Our first step consists in observing that, as soon as $\mathtt{runs} = \emptyset$ and $h \leqslant 2\kappa + 1$, the algorithm κ-stack ShiversSort will keep performing the merges of line 13 until $h = 1$, thereby following the merge policy of MinimalStableSort itself. Therefore, we split κ-stack ShiversSort in two phases: *phase 1* consists in the (merge or push) operations performed while $\mathtt{runs} \neq \emptyset$ or $h \geqslant 2\kappa + 2$, in lines 7, 9, 11 or 28 of Algorithm 10, whereas *phase 2* consists in the merge operations performed while $h \leqslant 2\kappa + 2$ and $\mathtt{runs} = \emptyset$, in line 13.

Unsurprisingly, the dynamics of these two phases are quite different: phase 1 keeps the flavour of adaptive ShiversSort, whereas phase 2 is just a brutal application of MinimalStableSort. However, the notion of *successor* used when studying adaptive ShiversSort cannot be reused verbatim. Thus, we present a variant of the notion of successor, which will be well adapted to κ-stack ShiversSort and will lead to a variant of Proposition 18.

Definition 38. Let $\mathcal{R} = (R_1, \dots, R_t)$ be a sequence of runs of length $t \geqslant 2\kappa + 1$. Recall that an integer x is *dominated* in the sequence $\mathcal{R}$ if $1 \leqslant x \leqslant t - 1$ and $\ell_x \leqslant \max\{\ell_{x+1}, \ell_{x+2}\}$, with the convention that $\ell_{t+1} = \infty$.

Let us also define the integers

$$r_{\mathsf{large}} = \lfloor (r_1 + r_2 + \dots + r_{2\kappa})/\kappa \rfloor$$

and $\ell_{\mathsf{large}} = \lfloor \log_2(r_{\mathsf{large}}) \rfloor$. We say that an integer x is *low* in the sequence $\mathcal{R}$ if $1 \leqslant x \leqslant t - 1$ and if $\max\{\ell_x, \ell_{x+1}\} \leqslant \ell_{\mathsf{large}}$ (we may omit mentioning $\mathcal{R}$ when the context is clear).

Lemma 39. *Let $\mathcal{R} = (R_1, \dots, R_t)$ be a sequence of runs of length $t \geqslant 2\kappa + 1$. There exists at least one integer $x \leqslant 2\kappa - 1$ that is low in $\mathcal{R}$.*

Proof. Let $s = r_1 + r_2 + \dots + r_{2\kappa}$, and consider the set

$$X = \{x \colon 1 \leqslant x \leqslant 2\kappa \text{ and } \ell_x > \ell_{\mathsf{large}}\}.$$

For all $x \in X$, we have $\log_2(r_x) > \log_2(r_{\mathsf{large}})$, i.e., $r_x > r_{\mathsf{large}}$, and therefore $r_x > s/\kappa$. Therefore,

$$s \geqslant \sum_{x \in X} r_x > s|X|/\kappa,$$

which means that $|X| \leqslant \kappa - 1$.

Among the κ pairs $\{1,2\}, \{3,4\}, \ldots, \{2\kappa-1, 2\kappa\}$, at most $\kappa-1$ contain an element of X. Thus, there exists an integer $i \leqslant \kappa$ such that neither $2i-1$ nor $2i$ belong to X, i.e., such that the integer $x = 2i-1$ is low. □

Lemma 40. *Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a stack of height $h = 2\kappa+1$ encountered while κ-stack ShiversSort calls the function mustMerge, and let $\mathrm{S} = (R_1, \ldots, R_t)$ be the associated state. If there exists a low integer $x \leqslant 2\kappa - 1$ such that $x+1$ is not low in S, that call to mustMerge returns **true**.*

Proof. If x is low in S and $x+1$ is not low in S, then $\ell_x \leqslant \max\{\ell_x, \ell_{x+1}\} < \ell_{\text{large}} \leqslant \max\{\ell_{x+1}, \ell_{x+2}\}$. Therefore, x is both low in S and dominated, and mustMerge necessarily returns **true**. □

Definition 41. Let $\mathcal{R} = (R_1, \ldots, R_t)$ be a sequence of runs of length $t \geqslant 2\kappa + 1$, and let k be the least integer that is both low in $\mathcal{R}$ and dominated, if such an integer exists; it not, we simply set $k = 0$. The integer k is called the *κ-merge point* of $\mathcal{R}$. Furthermore, if $k \neq 0$, we call *κ-successor* of $\mathcal{R}$, and note $\mathrm{succ}_\kappa(\mathcal{R})$, the sequence of runs

$$(R_1, \ldots, R_{k-1}, R', R_{k+2}, \ldots, R_t),$$

where R' is the run obtained by merging R_k and R_{k+1}; if $k = 0$, the κ-successor of $\mathcal{R}$ is not defined.

Lemma 42. *Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a stack encountered during phase 1 of κ-stack ShiversSort, let $\mathrm{S} = (R_1, \ldots, R_t)$ be the associated state, and let k be the κ-merge point of S. If $h \geqslant 2\kappa + 2$, then*

(i) the integer k is not equal to any of the integers $1, 2, \ldots, 2\kappa - 1$;
(ii) the integer $2\kappa - 1$ is low in S;
(iii) we have:

$$\ell_{2\kappa-1} > \ell_{2\kappa} > \ldots > \ell_{h-3} > \max\{\ell_{h-2}, \ell_{h-1}\}. \tag{2}$$

Proof. The proof is done by induction. First, if $h \leqslant 2\kappa + 1$, there is nothing to prove: this case occurs, in particular, when the algorithm starts. Now, consider some stack $\mathcal{S} = (R_1, \ldots, R_h)$ that is updated into a new stack $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ of height $\overline{h} \geqslant 2\kappa + 2$, either by merging the runs R_{h-2} and R_{h-1} or by pushing the run $\overline{R}_{\overline{h}}$, and let $\overline{\mathrm{S}}$ be the state associated with $\overline{\mathcal{S}}$. Note that, if the update consists in a push operation, the states S and $\overline{\mathrm{S}}$ coincide with eath other. Let also k and $\overline{k}$ be the respective κ-merge points of S and $\overline{\mathrm{S}}$, and assume that $\mathcal{S}$ satisfies Lemma 42:

- If $h = 2\kappa + 1$, then $\overline{h} = 2\kappa + 2$ and $\overline{\mathcal{S}}$ was obtained by pushing $\overline{R}_{\overline{h}}$, so that $\overline{\mathrm{S}} = \mathrm{S}$. This run push must have been preceded by a call to the function mustMerge, which consisted in checking that k differs from the integers $1, 2, \ldots, 2\kappa - 1$, and then returning the value **false**.
  Since Lemma 39 states that there exists a integer $x \leqslant 2\kappa - 1$ that is low in $\mathrm{S} = \overline{\mathrm{S}}$, Lemma 40 then proves that $2\kappa - 1$ is low in $\mathrm{S} = \overline{\mathrm{S}}$. Then, since $k \neq 2\kappa - 1$, it must be that $2\kappa - 1$ is not dominated, i.e., that $\overline{\ell}_{2\kappa-1} > \max\{\overline{\ell}_{2\kappa}, \overline{\ell}_{2\kappa+1}\}$, which means that $\overline{\mathcal{S}}$ satisfies (2).

- If $h \geqslant 2\kappa + 2$ and $\overline{\mathcal{S}}$ was obtained by pushing $\overline{R}_{\overline{h}}$, then $\overline{h} = h + 1$ and $\overline{\mathrm{S}} = \mathrm{S}$. It already follows that $2\kappa - 1$ is low in $\mathrm{S} = \overline{\mathrm{S}}$, that $\overline{k} = k \notin \{1, 2, \ldots, 2\kappa - 1\}$ and that $\overline{\ell}_{2\kappa-1} > \overline{\ell}_{2\kappa} > \ldots > \overline{\ell}_{\overline{h}-3}$. Finally, since κ-stack ShiversSort triggered a push operation instead of a merge operation, it must be the case that $\ell_{h-2} > \max\{\ell_{h-1}, \ell_h\}$, i.e., that $\overline{\ell}_{\overline{h}-3} > \max\{\overline{\ell}_{\overline{h}-2}, \overline{\ell}_{\overline{h}-1}\}$. This proves that $\overline{\mathcal{S}}$ satisfies (2).
- If $\overline{\mathcal{S}}$ was obtained by merging the runs R_{h-2} and R_{h-1}, then $\overline{h} = h - 1$ and $\overline{R}_i = R_i$ for all $i \leqslant \overline{h} - 2$, and in particular for all $i \leqslant 2\kappa$. It follows that (i) each integer $x \leqslant \overline{h} - 4$ is dominated in $\overline{\mathrm{S}}$ if and only if x is dominated in S, and that (ii) $\ell_{\mathsf{large}} = \overline{\ell}_{\mathsf{large}}$, thereby proving that each integer $x \leqslant \overline{h} - 3$ is low in $\overline{\mathrm{S}}$ if and only if x is low in S. Since $\overline{h} \geqslant 2\kappa + 2$, it already follows that $\overline{k} \notin \{1, 2, \ldots, 2\kappa - 2\}$ and that $2\kappa - 1$ is low in $\overline{\mathrm{S}}$.
  Finally, the inequalities $\ell_{2\kappa-1} > \ell_{2\kappa} > \ldots > \ell_{h-3}$ rewrite as $\overline{\ell}_{2\kappa-1} > \overline{\ell}_{2\kappa} > \ldots > \overline{\ell}_{\overline{h}-2}$. Then, since the run $\overline{R}_{\overline{h}-1}$ results from the merge between R_{h-2} and R_{h-1}, Lemma 6 proves that $\overline{\ell}_{\overline{h}-1} \leqslant \max\{\ell_{h-2}, \ell_{h-1}\} + 1 \leqslant \ell_{h-3} = \overline{\ell}_{\overline{h}-2}$. This proves that $\overline{\mathcal{S}}$ satisfies (2), from which it also follows that $2\kappa - 1$ is not dominated in $\overline{\mathrm{S}}$, thereby proving that $\overline{k} \neq 2\kappa - 1$. □

Proposition 43. *Let* S *and* $\overline{\mathrm{S}}$ *be two consecutive states encountered during an execution of the phase 1 of* κ*-stack ShiversSort. We have* $\overline{\mathrm{S}} = \mathit{succ}_\kappa(\mathrm{S})$.

Proof. Let m be the merge operation that transforms S into $\overline{\mathrm{S}}$. Let $\mathcal{S} = (R_1, \ldots, R_h)$ be the stack just before m is performed. Let k be the κ-merge point of S, and let R_i and R_{i+1} be the runs that are merged when m is performed. We shall prove that $i = k$.

If $h = 2\kappa + 1$, the merge m was triggered by a call to mustMerge. During that call, the function mustMerge scans the integers $1, 2, \ldots, 2\kappa - 1$ one by one until it finds a low dominated integer is identified. That integer is equal to i, and by construction it is also equal to k.

Then, if $h \geqslant 2\kappa + 2$, we know that $i = h - 2$. Furthermore, in that case, Lemma 42 states that either $k = 0$ or $k \geqslant 2\kappa$, and that the inequalities (2) are satisfied. In particular, k cannot be equal to any of the integers $2\kappa, \ldots, h - 3$, which are not even dominated, whereas $h - 2$ is both low and dominated. This means that $k = h - 2$, which completes the proof. □

Due to the characterisation provided by Proposition 43, we focus now on proving that every merge performed in phase 1 of κ-stack ShiversSort would also have been performed by adaptive ShiversSort.

Lemma 44. *Let* $\mathcal{R} = (R_1, \ldots, R_\rho)$ *be a sequence of runs with* κ*-merge point* $k \neq 0$. *At some point when sorting that sequence, the algorithm adaptive ShiversSort will merge the runs* R_k *and* R_{k+1}.

Proof. If $k = 1$, then k is the least dominated integer, and thus adaptive ShiversSort starts by merging R_k and R_{k+1}. Hence, we assume that $k \geqslant 2$.

If $k-1$ is dominated, then $\ell_{k-1} \leqslant \max\{\ell_k, \ell_{k+1}\} \leqslant \ell_{\text{large}}$, and thus $\max\{\ell_{k-1}, \ell_k\} \leqslant \ell_{\text{large}}$, proving that $k-1$ is also low. Since k is the minimal low dominated integer, it follows that $k-1$ cannot be dominated.

In particular, an immediate induction on the number of operations performed by adaptive ShiversSort proves that, in every state encountered until adaptive ShiversSort ever merges one of the runs R_k or R_{k+1}, the run R_k will always be preceded by a run R such that $\ell > \max\{\ell_k, \ell_{k+1}\}$. Indeed, the only way to modify that run is to merge it with other runs, which cannot decrease its level.

Now, let m be the first merge performed by adaptive ShiversSort that involves some of the runs $R_k, R_{k+1}, \ldots, R_\rho$. We shall prove that m consists in merging the runs R_k and R_{k+1}. The state just before m takes place is a sequence of the form $\overline{\mathrm{S}} = (\overline{R}_1, \ldots, \overline{R}_t, R_k, R_{k+1}, \ldots, R_\rho)$ for some $t \leqslant k-1$. First, m cannot merge any of the runs $\overline{R}_1, \ldots, \overline{R}_{t-1}$. Second, since $\overline{\ell}_t > \max\{\ell_k, \ell_{k+1}\}$, it cannot merge $\overline{R}_t$ either. Consequently, and since $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\}$, it must merge R_k and R_{k+1}. □

Proposition 45. *Every merge performed during phase 1 of κ-stack ShiversSort on a sequence of runs $\mathcal{R}$ would also have been performed by adaptive ShiversSort on the sequence $\mathcal{R}$.*

Proof. Let $\mathrm{S}_0, \ldots, \mathrm{S}_p$ be the sequence of states encountered in phase 1 of κ-stack ShiversSort, with $\mathrm{S}_0 = \mathcal{R}$. For all $i \leqslant p-1$, since $\mathrm{S}_{i+1} = \mathrm{succ}_\kappa(\mathrm{S}_i)$, Lemma 44 proves that S_{i+1} is a border of the merge tree induced by adaptive ShiversSort on S_i. It follows at once that every state S_i is a border of the tree $\mathcal{T}$ induced by adaptive ShiversSort on $\mathcal{R}$, which completes the proof. □

As a nice consequence of Proposition 45, we can already observe that using κ-stack ShiversSort instead of adaptive ShiversSort comes with no additional merge cost.

Corollary 46. *The merge cost of κ-stack ShiversSort cannot exceed the cost of adaptive ShiversSort.*

Proof. If phase 2 of κ-stack ShiversSort consisted in running adaptive ShiversSort instead of MinimalStableSort, the algorithms adaptive ShiversSort and κ-stack ShiversSort would perform the same list of merges, possibly in a different order. Since running MinimalStableSort cannot increase the total merge cost of those merges performed in phase 2, the result follows. □

5.2.3 Least high border and lattice of partitions into intervals. In order to go further in our proof, we need a way to compare two states S_1 and S_2 obtained while executing two algorithms, although there might exist no merge tree of which both S_1 and S_2 are borders. This requires finding a framework in which all borders of all merge trees can simultaneously exist.

To do this, we consider a given sequence of runs $\mathcal{R} = (R_1, \ldots, R_\rho)$, that will be fixed once and for all, and therefore left implicit whenever possible. This allows us to identify every run R_i with the integer i, or even with the singleton set $\{i\}$. We also identify every interval $X = \{x, x+1, \ldots, y\}$ with

the run R created by merging the consecutive runs $R_x, R_{x+1}, \dots, R_y$. In order to avoid ambiguities, and depending on the context, we may note this run R_X or directly X. Building on this notation, the length of the run R_X is denoted by $r_X = \sum_{x \in X} r_x$, and we set $\ell_X = \lfloor \log_2(r_X) \rfloor$.

Then, we extend these notations to sequences of runs and to partitions into intervals. We thereby also identify every partition $\mathbf{I} = (I_1, \dots, I_s)$ of X into intervals with the sequence of runs $(R_{I_1}, \dots, R_{I_s})$. Again, in order to avoid ambiguities, and depending on the context, we may note this sequence $\mathcal{R}_\mathbf{I}$ or directly by $\mathbf{I}$. Furthermore, if $\mathbf{I}$ is the finest partition of X, we may simply note $\mathcal{R}_X$ the sequence $\mathcal{R}_\mathbf{I}$: this is the sub-sequence $(R_x, R_{x+1}, \dots, R_y)$ of the sequence $\mathcal{R}$. In particular, this identification between sequences of runs and partitions applies to every border of every merge tree induced on $\mathcal{R}_X$, and each such border is now seen as a partition of X into intervals.

The interest of this identification also comes from the *refinement* ordering on such partitions: we say that a partition $\mathbf{I} = (I_1, \dots, I_s)$ refines a partition $\mathbf{J} = (J_1, \dots, J_t)$, which is denoted by $\mathbf{I} \sqsubseteq \mathbf{J}$, if every set I_i is contained in some set J_j. Partitions of $\{1, \dots, \rho\}$ into intervals form a lattice for the refinement ordering. Below, let start be the finest partition $\{\{1\}, \dots, \{\rho\}\}$, and end be the coarsest partition $\{\{1, \dots, \rho\}\}$: these are respectively the $\sqsubseteq$-minimal and the $\sqsubseteq$-maximal elements of the lattice. The sequence $\mathcal{R}_{\mathsf{start}}$ is simply the sequence $\mathcal{R}$, whereas $\mathcal{R}_{\mathsf{end}}$ is the sequence whose only run is the sorted array itself.

Then, we need a simple characterisation of the state obtained at the end of phase 1. Such a characterisation relies on the notions of *high* nodes in a merge tree and of *least high border*.

Definition 47. Let $\mathcal{T}$ be a merge tree induced by some stable algorithm on the sequence $\mathcal{R}$. Let n be the sum of the lengths or the runs in $\mathcal{R}$, and let $\ell^\star_{\mathsf{large}} = \lfloor \log_2(n/\kappa) \rfloor$. Finally, let R be a node of $\mathcal{T}$, with level ℓ. We say that R is *immense* if $\ell \geqslant \ell^\star_{\mathsf{large}} + 2$, that R is *very high* if $\ell \geqslant \ell^\star_{\mathsf{large}} + 1$, and that R is *high* in $\mathcal{T}$ if R or its sibling (in case R is not the root of $\mathcal{T}$) is very high; whenever the context is clear enough, we will omit mentioning the tree $\mathcal{T}$, and just say that R is high.

Definition 48. Let $\mathcal{T}$ be a merge tree induced by some stable algorithm, and let $\mathcal{T}'$ be the sub-tree that consists of the high nodes of $\mathcal{T}$. The border of $\mathcal{T}$ that consists of the leaves of $\mathcal{T}'$ is called the *least high border* of $\mathcal{T}$.

Equipped with these two notions, we will focus on the two trees $\mathcal{T}_{\mathsf{ass}}$ and $\mathcal{T}_{\mathsf{opt}}$ respectively induced by adaptive ShiversSort and MinimalStableSort on $\mathcal{R}$, and four partitions of $\{1, \dots, \rho\}$, the first three of which we introduce as states or borders:

- the last state $\mathsf{S}_{\mathsf{phase\,1}}$ encountered in phase 1 of κ-stack ShiversSort when sorting $\mathcal{R}$;
- the least high border $\mathcal{B}_{\mathsf{ass}}$ of the tree $\mathcal{T}_{\mathsf{ass}}$;
- the least high border $\mathcal{B}_{\mathsf{opt}}$ of the tree $\mathcal{T}_{\mathsf{opt}}$;

▹ the coarsest partition **K** that refines both $\mathcal{B}_{\text{ass}}$ and $\mathcal{B}_{\text{opt}}$, i.e., their $\sqsubseteq$-greatest lower bound.

The interest of the least high border $\mathcal{B}_{\text{ass}}$ comes from the following result, illustrated in Figure 7.

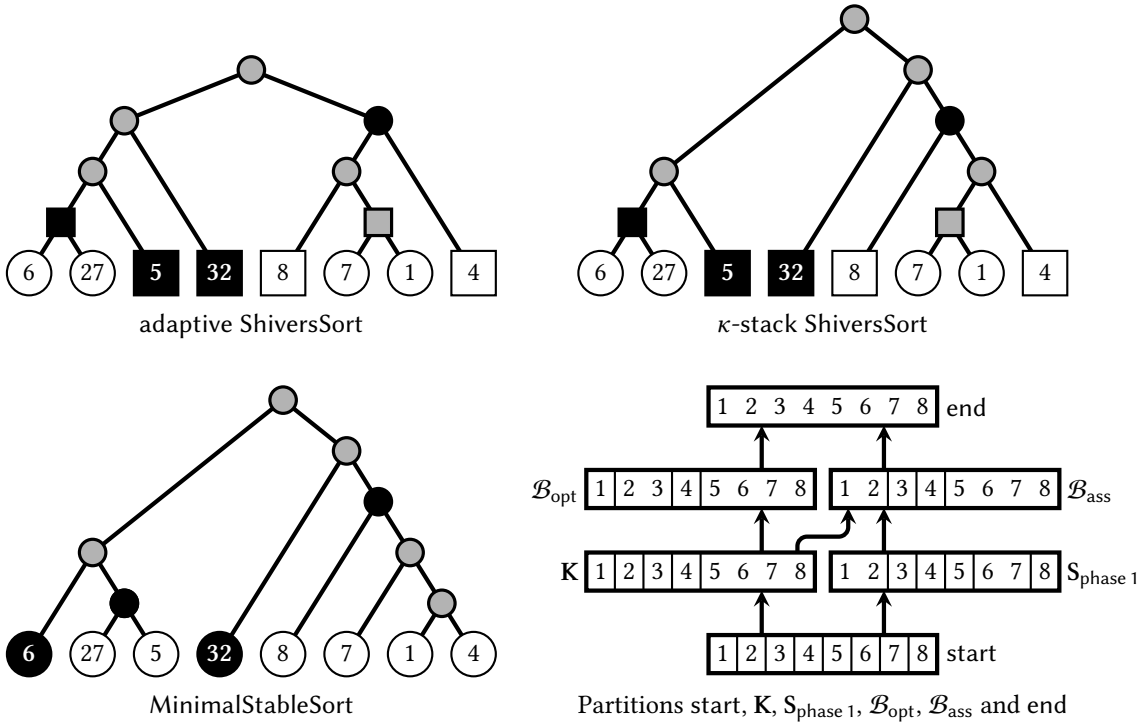


Fig. 7. Case $\kappa = 3$: Merge trees induced by adaptive ShiversSort, κ-stack ShiversSort and MinimalStableSort, and remarkable partitions. Least high borders are represented by black nodes and, in the two top trees, the border S is represented by square nodes. Tree leaves are labelled by the lengths of the runs they represent. Finally, upward arrows between partitions indicate refinement relations.

Proposition 49. *The partition* $\text{S}_{\text{phase 1}}$ *refines the partition* $\mathcal{B}_{\text{ass}}$.

Proof. Let us assume, for the sake of contradiction, that some run R of $\text{S}_{\text{phase 1}}$ is a strict ancestor, in the tree $\mathcal{T}_{\text{ass}}$, of a node in $\mathcal{B}_{\text{ass}}$. By definition, both children of R are high nodes of $\mathcal{T}_{\text{ass}}$. Hence, one of them, say R', is a run of level $\ell' > \ell^\star_{\text{large}}$, where we recall that $\ell^\star_{\text{large}} = \lfloor \log_2(n/\kappa) \rfloor$ and n is the sum of the lengths of the runs in the sequence $\mathcal{R}$.

Now, consider the state $\overline{\text{S}} = (\overline{R}_1, \ldots, \overline{R}_t)$ just before κ-stack ShiversSort proceeded to merging R'. By construction, at this point in time, we have $\overline{r}_{\text{large}} = \lfloor (\overline{r}_1 + \ldots + \overline{r}_{2\kappa})/\kappa \rfloor \leqslant (\overline{r}_1 + \ldots + \overline{r}_t)/\kappa = n/\kappa$, which proves that $\overline{\ell}_{\text{large}} = \lfloor \log_2(\overline{r}_{\text{large}}) \rfloor \leqslant \ell^\star_{\text{large}}$. It follows that $\ell' > \overline{\ell}_{\text{large}}$, contradicting the fact that κ-stack ShiversSort might have merged R' in phase 1. This completes the proof. □

Another interest of the notion of least high border is that it can be defined in every merge tree, which allows considering both trees $\mathcal{T}_{\mathsf{ass}}$ and $\mathcal{T}_{\mathsf{opt}}$ at once. Then, the partition $\mathbf{K}$, which does not need to be a border in either tree, is meant to provide a convenient intermediate step towards proving that the partitions $\mathcal{B}_{\mathsf{ass}}$ and $\mathcal{B}_{\mathsf{opt}}$ are similar to each other.

Let us make this statement precise by using adequate metrics.

Definition 50. Let $\mathbf{I} = (I_1, \dots, I_s)$ and $\mathbf{J} = (J_1, \dots, J_t)$ be two partitions of $\{1, \dots, \rho\}$ into intervals, such that $\mathbf{I}$ refines $\mathbf{J}$. For all $j \leqslant t$, we set $\mathrm{d}_j = |\{i \colon I_i \subseteq J_j\}|$, i.e., d_j is the number of intervals I_i in which J_j is subdivided. We say that $\mathbf{I}$ is a d-*refinement* of $\mathbf{J}$ if $\mathrm{d}_j \leqslant \mathrm{d}$ for all $j \leqslant t$.

Lemma 51. *Let $\mathcal{T}$ be a merge tree induced on the sequence $\mathcal{R}$, and let $\mathcal{B}$ be its least high border:*

(i) each very high run of $\mathcal{R}$ also belongs to $\mathcal{B}$, and
(ii) each immense run of $\mathcal{B}$ also belongs to $\mathcal{R}$.

Proof. Claim (i) immediately follows from the definition of the least high border. Then, if an immense run R of $\mathcal{T}$ has two children $R_\ominus$ and $R_\oplus$, Lemma 6 proves that

$$\max\{\ell_\ominus, \ell_\oplus\} \geqslant \ell - 1 \geqslant \ell^\star_{\mathsf{large}} + 1.$$

This means that $R_\ominus$ and $R_\oplus$ are high nodes, and thus that R cannot belong to the border $\mathcal{B}$, thereby proving claim (ii). □

Lemma 52. *Among any two consecutive runs of $\mathcal{B}_{\mathsf{ass}}$, at least one is very high in the tree $\mathcal{T}_{\mathsf{ass}}$.*

Proof. Let us assume that there exist two consecutive runs R and R' of $\mathcal{B}_{\mathsf{ass}}$ that are not very high. Let us apply the algorithm adaptive ShiversSort on the border $\mathcal{B}_{\mathsf{ass}}$, and let S be the state obtained just before one of the runs R or R' is merged. Three cases are *a priori* possible:

- If the run R is merged with R', neither run is high in $\mathcal{T}_{\mathsf{ass}}$.
- If the run R' has a right neighbour R'' in S and is merged with R'', we must have $\ell > \max\{\ell', \ell''\} \geqslant \ell''$, so that R'' is not very high, and neither R' nor R'' is high in $\mathcal{T}_{\mathsf{ass}}$.
- If the run R has a left neighbour $\overline{R}$ in S and is merged with $\overline{R}$, we have $\overline{\ell} \leqslant \max\{\ell, \ell'\}$, so that $\overline{R}$ is not very high, and neither $\overline{R}$ nor R is high in $\mathcal{T}_{\mathsf{ass}}$.

Thus, our initial assumption is disproved in every case. □

Lemma 53. *Let R be some run in the tree $\mathcal{T}_{\mathsf{opt}}$, and let $\overline{R}_1, \dots, \overline{R}_k$ be the runs with which MinimalStableSort successively merges R (i.e., the sibling of R is $\overline{R}_1$, the uncle of R is $\overline{R}_2$, and so on). For all $i \geqslant 3$, we have $\overline{r}_i \geqslant r$.*

Proof. Let us first assume that $\overline{r}_3 < r$. Then, the total cost of the merges of R with $\overline{R}_1$, $\overline{R}_2$ and $\overline{R}_3$ is $\mathsf{mc} = 3(r + \overline{r}_1) + 2\overline{r}_2 + \overline{r}_3$. However, if we had used a balanced binary tree of height 2 for

merging these four runs, (i.e., merging first the two leftmost runs, then the two rightmost runs, and finally the two resulting runs), each run would have participated to 2 merges only, for a total cost of $\mathrm{mc}' = 2(r + \overline{r}_1 + \overline{r}_2 + \overline{r}_3) = \mathrm{mc} - (r + \overline{r}_1 - \overline{r}_3) < \mathrm{mc}$. This contradicts the optimality of our merge policy, which proves that $\overline{r}_3 \geqslant r$.

The same reasoning, applied to the run obtained by merging R and $\overline{R}_1, \ldots, \overline{R}_i$, shows that $\overline{r}_{i+3} \geqslant r + \overline{r}_1 + \ldots + \overline{r}_i \geqslant r$ for all $i \leqslant k - 3$, which completes the proof. □

Corollary 54. *Among any five consecutive runs of $\mathcal{B}_{\mathrm{opt}}$, at least one is very high in the tree $\mathcal{T}_{\mathrm{opt}}$.*

Proof. Let $\mathcal{T}'$ be the ancestor sub-tree of $\mathcal{B}_{\mathrm{opt}}$ in $\mathcal{T}_{\mathrm{opt}}$, and let us assume that there exist five consecutive runs R_{i-2}, R_{i-1}, R_i, R_{i+1} and R_{i+2} of $\mathcal{B}_{\mathrm{opt}}$ that are not very high. Let R' be the parent of R_i, and let $\langle R' \rangle$ be the sub-tree of $\mathcal{T}'$ rooted at R'. Among any two sibling leaves of $\langle R' \rangle$ of maximal depth, one of them, say $\overline{R}$, must be very high.

Since $\overline{\ell}_i \leqslant \ell^{\star}_{\mathrm{large}} < \overline{\ell}$, and thus $\overline{r}_i < \overline{r}$, Lemma 53 proves that R_i is either the sibling or the uncle of $\overline{R}$. This means that $\langle R' \rangle$ has either two or three leaves and, since these leaves that consecutive runs of $\mathcal{B}$, they must all belong to the set $\{R_{i-2}, R_{i-1}, R_i, R_{i+1}, R_{i+2}\}$. Thus, one of these runs was in fact very high, which invalidates our initial assumption and completes the proof. □

Proposition 55. *The partition* $\mathbf{K}$ *is a* 11*-refinement of* $\mathcal{B}_{\mathrm{ass}}$ *and a* 7*-refinement of* $\mathcal{B}_{\mathrm{opt}}$.

Proof. First, assume that some interval I, belonging to the partition $\mathcal{B}_{\mathrm{ass}}$, contains at least 12 sub-intervals $K_1, \ldots, K_{12}$ of the partition $\mathbf{K}$. By definition of $\mathbf{K}$, each of the intervals $K_2, \ldots, K_{11}$ belongs to $\mathcal{B}_{\mathrm{opt}}$. Corollary 54 proves that one of the runs $R_{K_2}, \ldots, R_{K_6}$, say R_{K_x}, is very high. Similarly, one of the runs $R_{K_7}, \ldots, R_{K_{11}}$, say R_{K_y}, is very high too. It follows that

$$r_I \geqslant \sum_{k=1}^{12} r_{K_k} \geqslant r_{K_x} + r_{K_y} \geqslant 2 \times 2^{\ell^{\star}_{\mathrm{large}}+1} = 2^{\ell^{\star}_{\mathrm{large}}+2},$$

which means that R_I is immense. Hence, Lemma 51 proves that R_I also belongs to $\mathcal{R}$ and to $\mathcal{B}_{\mathrm{opt}}$, which is a contradiction.

Likewise, assume some interval J of the partition $\mathcal{B}_{\mathrm{opt}}$ contains at least 8 sub-intervals $K_1, \ldots, K_8$ of the partition $\mathbf{K}$. Then, all sub-intervals $K_2, \ldots, K_7$ also belong to $\mathcal{B}_{\mathrm{ass}}$. Lemma 52 proves that one of the runs R_{K_2}, R_{K_3} or R_{K_4}, say R_{K_x}, is very high, and that one of the runs R_{K_5}, R_{K_6} or R_{K_7}, say R_{K_y}, is very high too. But then

$$r_J \geqslant \sum_{k=1}^{8} r_{K_k} \geqslant r_{K_x} + r_{K_y} \geqslant 2 \times 2^{\ell^{\star}_{\mathrm{large}}+1} = 2^{\ell^{\star}_{\mathrm{large}}+2},$$

which means that R_J is immense, again leading to a contradiction. □

5.2.4 The cost of approximations. Grossly speaking, Theorem 33 says that the merge cost of κ-stack ShiversSort is very good, which is what we prove below. Before writing this proof itself, we first provide an intuition of how the main steps this proof consists of.

(1) κ-stack ShiversSort consists in applying merges prescribed by adaptive ShiversSort until one reaches the partition $\mathbf{S}_{\text{phase 1}}$, in phase 1, and then applying MinimalStableSort, in phase 2. Instead of this phase 2, we might just continue applying merges prescribed by adaptive ShiversSort until one reaches the partition $\mathcal{B}_{\text{ass}}$, and then apply MinimalStableSort on this partition. Doing so will only increase the total cost of the merges performed.
(2) By independence property, using merges from adaptive ShiversSort to obtain the partition $\mathcal{B}_{\text{ass}}$ amounts to using adaptive ShiversSort to sort, one by one, the sub-sequences of $\mathcal{R}$ spanned by the runs in $\mathcal{B}_{\text{ass}}$. Theorem 15 states that using adaptive ShiversSort to sort each such sub-sequence is not much more expensive than using MinimalStableSort.
(3) Since $\mathbf{K}$ is a d-refinement of both partitions $\mathcal{B}_{\text{ass}}$ and $\mathcal{B}_{\text{opt}}$ for small values of d, using MinimalStableSort to obtain (the sequence of runs that we identify with) the partition $\mathbf{K}$ is approximately as expensive as using MinimalStableSort to obtain either the partition $\mathcal{B}_{\text{ass}}$ or $\mathcal{B}_{\text{opt}}$, and the partition $\mathbf{K}$ is approximately as expensive to sort as either the partition $\mathcal{B}_{\text{ass}}$ or $\mathcal{B}_{\text{opt}}$.
(4) Since $\mathcal{B}_{\text{opt}}$ is a border of the tree $\mathcal{T}_{\text{opt}}$, and up to changing the order in which MinimalStableSort performs merges, we may assume that it uses $\mathcal{B}_{\text{opt}}$ as an intermediate state, so that the smallest merge cost to sort $\mathcal{R}$ is the sum of the costs to obtain the partition $\mathcal{B}_{\text{opt}}$ and to sort this partition.

Most of these steps consist in evaluating the total cost of those merges used by a given algorithm to transform a partition $\mathbf{I}$ into another partition $\mathbf{J}$. Thus, if $\mathcal{R}_{\mathbf{J}}$ is a border of the merge tree induced by adaptive ShiversSort on the sequence $\mathcal{R}_{\mathbf{I}}$, let $\mathsf{mc}_{\text{ass}}(\mathbf{I} \to \mathbf{J})$ denote the total cost of those merges required by adaptive ShiversSort to transform $\mathbf{I}$ into $\mathbf{J}$. As mentioned in step 2 above, and since adaptive ShiversSort has the independence property, this amounts to considering one by one the intervals J in $\mathbf{J}$, the partition $\{I \in \mathbf{I} \colon I \subseteq J\}$ of J into intervals taken from the partition $\mathbf{I}$, and to sorting the sequence $\mathcal{R}_{\{I \in \mathbf{I} \colon I \subseteq J\}}$ (which consists of those runs of $\mathcal{R}_{\mathbf{I}}$ spanned by the run R_J), thereby obtaining the run R_J.

Similarly, if $\mathbf{I}$ refines $\mathbf{I}$, and even if $\mathbf{J}$ is not a border of the merge tree induced by MinimalStableSort on the sequence $\mathcal{R}_{\mathbf{I}}$, let $\mathsf{mc}_{\text{opt}}(\mathbf{I} \to \mathbf{J})$ denote the total cost of those merges required by MinimalStableSort to transform $\mathbf{I}$ into $\mathbf{J}$, i.e., the merge cost paid by considering one by one the runs R in $R_{\mathbf{J}}$, selecting the sub-sequence of $\mathcal{R}_{\mathbf{I}}$ of those runs spanned by the run R, and sorting that sub-sequence.

Finally, given any two partitions $\mathbf{I}$ and $\mathbf{J}$ of $\{1, \dots, \rho\}$ into intervals, we call *distortion* between $\mathbf{I}$

and $\mathbf{J}$, and note $\delta(\mathbf{I},\mathbf{J})$, the sum of the lengths of the runs R that belong to the sequence $\mathcal{R}_\mathbf{I}$ but not to the sequence $\mathcal{R}_\mathbf{J}$. Observe that this function is symmetric, i.e., that $\delta(\mathbf{I},\mathbf{J}) = \delta(\mathbf{J},\mathbf{I})$.

With these notations in mind, Proposition 45 has the following consequence, which give us a better bound than Corollary 46.

Corollary 46$^\mathbf{b}$. *The merge cost of κ-stack ShiversSort cannot exceed*

$$\mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathcal{B}_{\mathrm{ass}}) + \mathrm{mc}_{\mathrm{opt}}(\mathcal{B}_{\mathrm{ass}} \to \mathrm{end}).$$

Proof. Let mc_κ be he merge cost of κ-stack ShiversSort when sorting the sequence $\mathcal{R}$. Since forcing MinimalStableSort to use the sequence $\mathcal{B}_{\mathrm{opt}}$ as an intermediate step during the phase 2 of κ-stack ShiversSort cannot decrease the resulting merge cost, and since the stability property of adaptive ShiversSort shows that $\mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathbf{S}) + \mathrm{mc}_{\mathrm{ass}}(\mathbf{S} \to \mathcal{B}_{\mathrm{ass}}) = \mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathcal{B}_{\mathrm{ass}})$, we have

$$\begin{aligned}
\mathrm{mc}_\kappa &= \mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathbf{S}) + \mathrm{mc}_{\mathrm{opt}}(\mathbf{S} \to \mathrm{end}) \\
&\leqslant \mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathbf{S}) + \mathrm{mc}_{\mathrm{opt}}(\mathbf{S} \to \mathcal{B}_{\mathrm{ass}}) + \mathrm{mc}_{\mathrm{opt}}(\mathcal{B}_{\mathrm{ass}} \to \mathrm{end}) \\
&\leqslant \mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathbf{S}) + \mathrm{mc}_{\mathrm{ass}}(\mathbf{S} \to \mathcal{B}_{\mathrm{ass}}) + \mathrm{mc}_{\mathrm{opt}}(\mathcal{B}_{\mathrm{ass}} \to \mathrm{end}) \\
&\leqslant \mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathcal{B}_{\mathrm{ass}}) + \mathrm{mc}_{\mathrm{opt}}(\mathcal{B}_{\mathrm{ass}} \to \mathrm{end}).
\end{aligned}$$

□

Proposition 56. *Let $\mathbf{I}$ be a partition of the set $\{1,\dots,\rho\}$ into intervals, such that $\mathcal{R}_\mathbf{I}$ is a border of $\mathcal{T}_{\mathrm{ass}}$. We have*

$$\mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathbf{I}) \leqslant \mathrm{mc}(\mathrm{start} \to \mathbf{I}) + \Delta\,\delta(\mathbf{I},\mathrm{start}),$$

where we recall that $\Delta = 24/5 - \log_2(5)$.

Proof. Transforming $\mathcal{R}$ into $\mathcal{R}_\mathbf{I}$ amounts to sorting the sequence $\mathcal{R}_I$ for each interval $I \in \mathbf{I}$. Theorem 24 states that MinimalStableSort performs this transformation for a cost that is at least $r_I\mathcal{H}_I$, where we set

$$\mathcal{H}_I = -\sum_{i\in I} r_i/r_I \log_2(r_i/r_I).$$

Meanwhile, adaptive ShiversSort performs the same transformation for a cost that is 0 if $I \in \mathrm{start}$, and at most $r_i(\mathcal{H}_I + \Delta)$ in general.

We conclude that

$$\mathrm{mc}_{\mathrm{ass}}(\mathrm{start} \to \mathbf{I}) \leqslant \sum_{I\in\mathbf{I}} r_I(\mathcal{H}_I + \Delta)\mathbf{1}_{I\in\mathrm{start}} \leqslant \mathrm{mc}(\mathrm{start} \to \mathbf{I}) + \Delta\,\delta(\mathbf{I},\mathrm{start}).$$

□

Our next move consists in proving a monotonicity statement: whenever a partition $\mathbf{I}$ refines another partition $\mathbf{J}$, obtaining the partition $\mathbf{I}$ never costs more than obtaining $\mathbf{J}$, and sorting the partition $\mathbf{J}$ never costs more than sorting $\mathbf{I}$.

Lemma 57. *Let* $\mathbf{I}$ *and* $\mathbf{J}$ *be two partitions of the set* $\{1, \dots, \rho\}$ *into intervals, such that* $\mathbf{I}$ *refines* $\mathbf{J}$*. We have* $\mathsf{mc}(\mathsf{start} \to \mathbf{I}) \leqslant \mathsf{mc}(\mathsf{start} \to \mathbf{J})$ *and* $\mathsf{mc}(\mathbf{I} \to \mathsf{end}) \geqslant \mathsf{mc}(\mathbf{J} \to \mathsf{end})$.

Proof. Let $\mathbf{I} = (I_1, \dots, I_n)$ and $\mathbf{J} = (J_1, \dots, J_m)$ be our two partitions of the set $\{1, \dots, \rho\}$. We prove Lemma 57 by induction on n.

First, if $n \leqslant m$, then $\mathbf{I} = \mathbf{J}$, and the result is immediate. Then, if $n \geqslant m + 2$, let $\mathbf{L}$ be a partition that lies between the partitions $\mathbf{I}$ and $\mathbf{J}$ (i.e., $\mathbf{I}$ strictly refines $\mathbf{L}$, which strictly refines $\mathbf{J}$). The induction hypothesis ensures that $\mathsf{mc}(\mathsf{start} \to \mathbf{I}) \leqslant \mathsf{mc}(\mathsf{start} \to \mathbf{L}) \leqslant \mathsf{mc}(\mathsf{start} \to \mathbf{J})$ and $\mathsf{mc}(\mathbf{I} \to \mathsf{end}) \geqslant \mathsf{mc}(\mathbf{L} \to \mathsf{end}) \geqslant \mathsf{mc}(\mathbf{J} \to \mathsf{end})$.

Finally, let us assume that $n = m + 1$. We first prove that $\mathsf{mc}(\mathbf{I} \to \mathsf{end}) \geqslant \mathsf{mc}(\mathbf{J} \to \mathsf{end})$. Up to redefining the sequence $\mathcal{R}$ as the sequence $\mathcal{R}_{\mathbf{I}}$, we assume without loss of generality that $\mathbf{I} = \mathsf{start}$. In that case, let J_j be the only interval such that $\mathsf{d}_j = 2$: it is the disjoint union of the intervals $I_j = \{j\}$ and $I_{j+1} = \{j + 1\}$. Then, recall that $\mathcal{T}_{\mathsf{opt}}$ is the merge tree generated by MinimalStableSort on the sequence $\mathcal{R}$. Up to replacing $\mathcal{R}$ by its mirror (i.e., by the sequence $(R_n, R_{n-1}, \dots, R_1)$, and denoting by δ_j and δ_{j+1} the respective depths of the leaves R_j and R_{j+1} in the tree $\mathcal{T}_{\mathsf{opt}}$, we assume without loss of generality that $\delta_j \geqslant \delta_{j+1}$.

Then, let us consider the sequence of merges performed by MinimalStableSort when sorting the sequence $\mathcal{R}$: these are the merges between two runs R_X and R_Y that are siblings in the tree $\mathcal{T}_{\mathsf{opt}}$. We modify that sequence as follows. First, the run R_j is deleted, and its (unique) merge with another run is also deleted. Second, and for every run R_X ever involved in a merge:

- if X is of the form $\{j + 1, \dots, x\}$ (with $x \geqslant j + 1$) we replace R_X by the run $R_{\{j,\dots,x\}}$;
- if X is of the form $\{x, \dots, j\}$ (with $x \leqslant j - 1$) we replace R_X by the run $R_{\{x,\dots,j-1\}}$;
- otherwise, we keep the run R_X as is.

This situation is illustrated in Figure 8, where merges of the the former sequence are gathered in the tree $\mathcal{T}_{\mathsf{opt}}$ (on the left) and the merges of the latter (modified) sequence are gathered in the tree $\mathcal{T}'$ (on the right).

In this new tree $\mathcal{T}'$, the depth of each run R_i either does not change or decreases (the former runs R_j and R_{j+1}, which do not longer exist, are seen as both belonging to the run $R_{\{j,j+1\}}$). Consequently, by following this new sequence of merges, we transform the sequence $\mathcal{R}_{\mathbf{J}}$ into the sequence $\mathcal{R}_{\mathsf{end}}$, for a total cost mc such that $\mathsf{mc}_{\mathsf{opt}}(\mathbf{J} \to \mathsf{end}) \leqslant \mathsf{mc} \leqslant \mathsf{mc}_{\mathsf{opt}}(\mathbf{I} \to \mathsf{end})$.

Second, we prove that $\mathsf{mc}(\mathsf{start} \to \mathbf{I}) \leqslant \mathsf{mc}(\mathsf{start} \to \mathbf{J})$. This time, the independence property of MinimalStableSort ensures that we can work independently on all the sub-sequences $\mathcal{R}_J$, for each interval $J \in \mathbf{J}$. Thus, and up to redefining the sequence $\mathcal{R}$ as the sequence $\mathcal{R}_J$, we assume without loss of generality, we assume that $\mathbf{J} = \mathsf{end}$, and thus that $\mathbf{I} = \{\{1, \dots, z\}, \{z + 1, \dots, \rho\}\}$ for some integer z. Then, let us consider the sequence of merges performed by MinimalStableSort when sorting the sequence $\mathcal{R}$. We modify that sequence as follows:

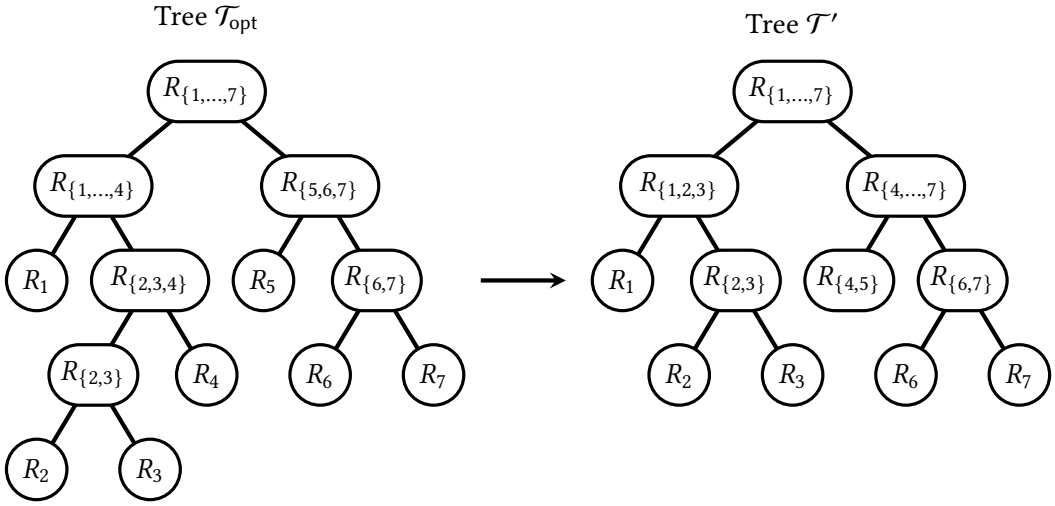


Fig. 8. Transforming the tree $\mathcal{T}_{\text{opt}}$ into a new tree $\mathcal{T}'$ of smaller merge cost (in case $j = 4$).

- the (unique) merge between runs R_X and R_Y such that $z \in X$ and $z+1 \in Y$ is deleted;
- every merge between runs R_X and R_Y such that $\{z, z+1\} \subseteq X$ is replaced by a merge between the runs $R_{\{x \in X\colon x \geqslant z+1\}}$ and R_Y;
- every merge between runs R_X and R_Y such that $\{z, z+1\} \subseteq Y$ is replaced by a merge between the runs R_X and $R_{\{y \in Y\colon y \leqslant z\}}$;
- other merges are not modified.

This situation is illustrated in Figure 9, where merges of the the former sequence are gathered in the tree $\mathcal{T}_{\text{opt}}$ (on the left) and the merges of the latter (modified) sequence are gathered in the forest $\mathcal{F}$ (on the right), which consists of two trees: one that gathers the runs R_X such that $X \subseteq \{1, \dots, z\}$ and one that gathers the runs R_X such that $X \subseteq \{z+1, \dots, \rho\}$.

Again, the depth of each run R_i in this new forest either does not change or decreases. Consequently, by following this new sequence of merges, we transform the initial sequence $\mathcal{R}$ into the sequence $\mathcal{R}_{\mathbf{J}}$, for a total cost mc such that $\mathsf{mc}(\mathsf{start} \to \mathbf{I}) \leqslant \mathsf{mc} \leqslant \mathsf{mc}(\mathsf{start} \to \mathbf{J})$. □

Lemma 58. *Let* $\mathbf{I}$ *and* $\mathbf{J}$ *be two partitions of the set* $\{1, \dots, \rho\}$ *into intervals, such that* $\mathbf{I}$ *refines* $\mathbf{J}$. *If* $\mathbf{I}$ *is a* d*-refinement of* $\mathbf{J}$, *then* $\mathsf{mc}(\mathbf{I} \to \mathbf{J}) \leqslant \lceil \log_2(\mathsf{d}) \rceil \delta(\mathbf{I}, \mathbf{J})$.

Proof. Let $\mathbf{I} = (I_1, \dots, I_n)$ and $\mathbf{J} = (J_1, \dots, J_m)$ be our two partitions of the set $\{1, \dots, \rho\}$. Consider some integer J_j such that $\mathsf{d}_j \geqslant 2$, and let i be the integer such that J_j is the disjoint union of $I_{i+1}, I_{i+2}, \dots, I_{i+\mathsf{d}_j}$. We can merge $R_{I_{i+1}}, \dots, R_{I_{i+\mathsf{d}_j}}$ into one unique run R_{J_j} by using a balanced binary merge tree of height $\lceil \log_2(\mathsf{d}_j) \rceil$. The total merge cost of these operations is thus bounded

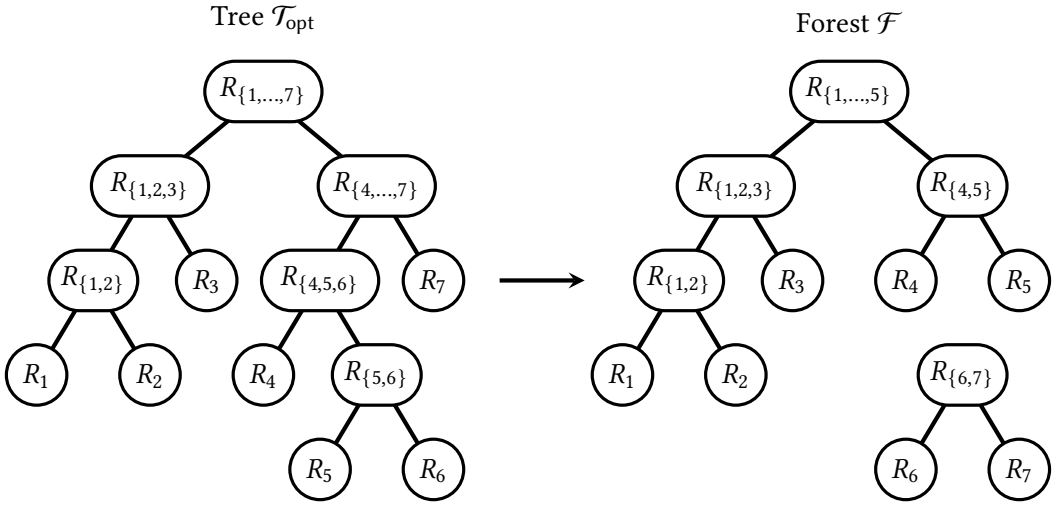


Fig. 9. Transforming the tree $\mathcal{T}_{\text{opt}}$ into a new forest $\mathcal{F}$ of smaller merge cost (in case $z = 5$).

from above by $\lceil \log_2(\mathrm{d}) \rceil\, r_{J_j}$. Proceeding in this way for every interval J_j such that $\mathrm{d}_j \geqslant 2$, we transform the sequence $\mathcal{R}_{\mathbf{I}}$ into $\mathcal{R}_{\mathbf{J}}$ for a cost of $\lceil \log_2(\mathrm{d}) \rceil\, \delta(\mathbf{I}, \mathbf{J})$ or less. □

Theorem 59. *The algorithm κ-stack ShiversSort is $\eta_{2\kappa+3}$-optimal.*

Proof. Let $\mathcal{R} = (R_1, \dots, R_\rho)$ be a sequence of runs to sort, of total length n and entropy $\mathcal{H}$, and let mc_κ be the merge cost of κ-stack ShiversSort when sorting the sequence $\mathcal{R}$. Let also $X = \{i \colon R_i$ is not very high$\}$, and let $n^\star = \sum_{i \in X} r_X$ be the sum of the lengths of those runs R_i that are not very high. For all $i \in X$, we know that $\lfloor \log_2(r_i) \rfloor = \ell_i \leqslant \ell^\star_{\text{large}} \leqslant \log_2(n/\kappa)$, and thus that $r_i \leqslant 2n/\kappa$. It follows that

$$n\mathcal{H} = \sum_{i=1}^{\rho} r_i \log_2(n/r_i) \geqslant \sum_{i \in X} r_i \log_2(n/r_i) \geqslant \sum_{i \in X} r_i \log_2(\kappa/2) = n^* \log_2(\kappa/2).$$

Then, Lemma 51 proves that each very high run R_i belongs to both $\mathcal{B}_{\text{ass}}$ and $\mathcal{B}_{\text{opt}}$. and thus the pairwise distortions between the partitions $\mathcal{B}_{\text{ass}}$, $\mathcal{B}_{\text{opt}}$, $\mathbf{K}$ and start are bounded from above by $n^\star$.

It follows that

$$\begin{aligned}
\mathsf{mc}_\kappa &\leqslant \mathsf{mc}_{\mathsf{ass}}(\mathsf{start} \to \mathcal{B}_{\mathsf{ass}}) + \mathsf{mc}_{\mathsf{opt}}(\mathcal{B}_{\mathsf{ass}} \to \mathsf{end}) && \text{by Corollary 46}^{\mathrm{b}} \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathcal{B}_{\mathsf{ass}}) + \mathsf{mc}_{\mathsf{opt}}(\mathcal{B}_{\mathsf{ass}} \to \mathsf{end}) + \Delta n^\star && \text{by Proposition 56} \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathbf{K}) + \mathsf{mc}_{\mathsf{opt}}(\mathbf{K} \to \mathcal{B}_{\mathsf{ass}}) + \mathsf{mc}_{\mathsf{opt}}(\mathcal{B}_{\mathsf{ass}} \to \mathsf{end}) + \Delta n^\star && \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathbf{K}) + \mathsf{mc}_{\mathsf{opt}}(\mathcal{B}_{\mathsf{ass}} \to \mathsf{end}) + (\Delta + 4)n^\star && \text{by Proposition 55} \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathcal{B}_{\mathsf{opt}}) + \mathsf{mc}_{\mathsf{opt}}(\mathbf{K} \to \mathsf{end}) + (\Delta + 4)n^\star && \text{by Lemma 57} \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathcal{B}_{\mathsf{opt}}) + \mathsf{mc}_{\mathsf{opt}}(\mathbf{K} \to \mathcal{B}_{\mathsf{opt}}) + \mathsf{mc}_{\mathsf{opt}}(\mathcal{B}_{\mathsf{opt}} \to \mathsf{end}) + (\Delta + 4)n^\star && \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathcal{B}_{\mathsf{opt}}) + \mathsf{mc}_{\mathsf{opt}}(\mathcal{B}_{\mathsf{opt}} \to \mathsf{end}) + (\Delta + 7)n^\star && \text{by Proposition 55} \\
&\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathsf{end}) + (\Delta + 7)n^\star && \text{since } \mathcal{B}_{\mathsf{opt}} \text{ is a border of } \mathcal{T}_{\mathsf{opt}}.
\end{aligned}$$

Since Theorem 24 proves that $\mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathsf{end}) \geqslant n\mathcal{H} \geqslant n^* \log_2(\kappa/2)$, we conclude that

$$\begin{aligned}
\mathsf{mc}_\kappa &\leqslant \mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathsf{end}) + (\Delta + 7)n^\star \\
&\leqslant (1 + (\Delta + 7)/\log_2(\kappa/2))\mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathsf{end}) \\
&\leqslant (1 + \eta_{2\kappa+3})\mathsf{mc}_{\mathsf{opt}}(\mathsf{start} \to \mathsf{end}),
\end{aligned}$$

which means that κ-stack ShiversSort is $\eta_{2\kappa+3}$-optimal. □

In conclusion, the algorithm κ-stack ShiversSort is $(2\kappa + 2)$-aware and $\eta_{2\kappa+3}$-optimal. Hence, it is also $(2\kappa + 3)$-aware and $\eta_{2\kappa+2}$-optimal, which proves Theorem 33.

6 IMPLEMENTATION DETAILS AND SIMPLIFICATIONS

One of the reasons for introducing the algorithm adaptive ShiversSort is that it might be a good substitute to TimSort, being both more efficient in the worst case and simple to implement by modifying the code used for TimSort.

This quest for simplicity led us to adopt the current presentation of adaptive ShiversSort instead of the original version presented in [14]. We prove below that Algorithms 1 and 8 perform the same merge operations, in the same relative order, a claim that was made in Section 2.7. The only behavioural difference between these two variants is that a push operation, which would happen after a given merge operation in the original algorithm, may now happen before that merge operation.

Then, we focus more specifically on the space complexity of adaptive ShiversSort. Indeed past versions of TimSort in languages such as Python or Java suffered from implementation bugs [1, 6], which involved the time complexity and, most importantly, the space complexity of TimSort. In both languages, the stack $\mathcal{S}$ used in TimSort is simulated by a fixed-size array. Allocating enough

memory to that array is therefore a crucial step. This task requires bounding from above the size that $\mathcal{S}$ may ever take during the execution of TimSort, before even starting to sort the array A (i.e., when the only thing known about A is its length). However, that step was incorrectly performed, which led to the bugs mentioned above. That is why, in order to avoid similar problems in the future, we study this problem below.

Another critical point, if one were to replace TimSort by adaptive ShiversSort or length-adaptive ShiversSort, consists in making sure that as few code lines as possible be modified, and that switching between the two algorithms be straightforward. This point was already taken care of since, as mentioned in the introduction, the only part that distinguishes adaptive ShiversSort and length-adaptive ShiversSort from TimSort is the merge policy used for choosing which *large* runs to merge. Below, we complete this task and actually provide the few code lines that should be used in order to use adaptive ShiversSort or length-adaptive ShiversSort instead of TimSort*.

6.1 Comparing the two versions of c-adaptive ShiversSort

In Section 2.7 and in the dicussion above, we claim that, when Algorithms 1 and 8 are used to sort a given array, they perform the same merge operations, in the same order. This amounts to proving the following variant of Proposition 18.

Proposition 18$^{\mathbf{b}}$. *Let* S *and* $\overline{\mathrm{S}}$ *be two consecutive states encountered during an execution of Algorithm 8. We have* $\overline{\mathrm{S}} = \mathit{succ}(\mathrm{S})$.

In order to prove Proposition 18$^{\mathrm{b}}$, we first prove a variant of Lemma 7.

Lemma 7$^{\mathbf{b}}$. *At any time during the main loop of Algorithm 8, if the run stack is* $\mathcal{S} = (R_1, \dots, R_h)$*, we have:*

$$\ell_1 > \ell_2 > \ldots > \ell_{h-2}. \tag{3}$$

Proof. The proof is done by induction. First, if $h \leqslant 3$, there is nothing to prove: this case occurs, in particular, when the algorithm starts. Now, consider some stack $\mathcal{S} = (R_1, \dots, R_h)$ that satisfies (3) and is updated into a new stack $\overline{\mathcal{S}} = (\overline{R}_1, \dots, \overline{R}_{\overline{h}})$, either by merging two of the runs R_{h-2}, R_{h-1} and R_h, or by pushing the run $\overline{R}_{\overline{h}}$:

- If a run merge was just performed, then $\overline{h} = h - 1$ and $\overline{R}_i = R_i$ for all $i \leqslant h - 3$. Thus, the inequalities $\ell_1 > \ell_2 > \ldots > \ell_{h-3}$ immediately rewrite as $\overline{\ell}_1 > \overline{\ell}_2 > \ldots > \overline{\ell}_{\overline{h}-2}$.
- If the run $\overline{R}_{\overline{h}}$ was just pushed, then $\overline{h} = h + 1$ and $\overline{R}_i = R_i$ for all $i \leqslant h$. Thus, the inequalities $\ell_1 > \ell_2 > \ldots > \ell_{h-2}$ already rewrite as $\overline{\ell}_1 > \overline{\ell}_2 > \ldots > \overline{\ell}_{\overline{h}-3}$. Furthermore, since Algorithm 8 triggered a push operation instead of a merge operation, it must be the case that $\overline{\ell}_{\overline{h}-3} = \ell_{h-2} > \ell_{h-1} = \overline{\ell}_{\overline{h}-2}$.

*Since this article was written, PowerSort has been chosen as the new standard sorting algorithm in Python. TimSort remains the standard sorting algorithm for non-primitive types in Java.

In both cases, the stack $\overline{\mathcal{S}}$ also satisfies (3), which completes the induction. □

Proof of Proposition 18[B]. Let m be the merge operation that transforms the state S into $\overline{\mathrm{S}}$. Let $\mathcal{S} = (R_1, \ldots, R_{k+i})$ be the run stack just before m takes place, with $i = 1$ or $i = 2$, and let $\mathcal{R} = (R_{k+i+1}, \ldots, R_t)$ be the sequence of those runs that are yet to be pushed onto the stack, so that m consists in merging the runs R_k and R_{k+1}, and that S is the concatenation of $\mathcal{S}$ and $\mathcal{R}$.

- If $i = 1$, Lemma 7[b] states that $\ell_1 > \ell_2 > \ldots > \ell_{k-1}$, and since Algorithm 8 performed the merge m, it means that $\ell_{k-1} > \ell_{k+1} \geqslant \ell_k$.
- If $i = 2$, Lemma 7[b] states that $\ell_1 > \ell_2 > \ldots > \ell_k$, and since Algorithm 8 performed the merge m, it means that $\ell_k \leqslant \max\{\ell_{k+1}, \ell_{k+2}\}$.

Both cases imply that k is the merge point of S, and therefore that $\overline{\mathrm{S}} = \mathrm{succ}(\mathrm{S})$. □

6.2 Scale invariance

In the introduction, we claimed that length-adaptive ShiversSort is a scale-invariant merge sort, i.e., that the sequence of merges it performs does not change if we multiply the length of each run by a constant a. Being scale-invariant is a desirable property shared by all the algorithms reviewed in Table 1 and in Section 2, except ShiversSort and adaptive ShiversSort. The fact that length-adaptive ShiversSort is indeed scale-invariant is a direct consequence of the following result.

Lemma 60. *Let r and n be positive integers, such that $r \leqslant n$. For all integers $a \geqslant 1$, we have*

$$\lfloor \log_2(r/(n+1)) \rfloor = \lfloor \log_2(ar/(an+1)) \rfloor.$$

Proof. Let $\ell = \lfloor \log_2(r/(n+1)) \rfloor$ and $\ell' = \lfloor \log_2(ar/(an+1)) \rfloor$. Since

$$\frac{ar}{an+1} = \frac{r}{n+1} + \frac{(a-1)r}{(n+1)(an+1)} \geqslant \frac{r}{n+1},$$

we know that $\ell' \geqslant \ell$. Conversely, recall that $r/(n+1) < 2^{\ell+1}$ and that $\ell \leqslant -1$. This means that $n + 1 > 2^{-\ell-1}r$ and that $2^{-\ell-1}$ is an integer, so that $n \geqslant 2^{-\ell-1}r$. It follows that

$$\frac{ar}{an+1} < \frac{ar}{an} = \frac{r}{n} \leqslant 2^{\ell+1},$$

and therefore that $\ell' \leqslant \ell$, which completes the proof. □

6.3 Stack size

In Section 3, we focused on the *time* complexity of adaptive ShiversSort, which is obviously an important parameter. However, for the reasons mentioned just above, evaluating precisely the *space* complexity of adaptive ShiversSort is also important. Thus, we first provide upper bounds on the stack size that might be required while sorting an array of size n.

Since, as mentioned in Section 1, TimSort is also based on dealing with small runs with an *ad hoc* sub-routine, we also take into account the minimal size $\mathrm{s}_{\min}$ that characterizes runs *large enough* to be considered by our merge policy (except the last run, which may be of any size). We also note $\ell_{\min} = \lfloor \log_2(\mathrm{s}_{\min}) \rfloor$ the *minimal level* of such runs.

Proposition 61. *At any time while sorting an array of size $n \geqslant \mathrm{s}_{\min}$ by considering runs of size $\mathrm{s}_{\min}$ or more (except, possibly, the rightmost run), the stack size required by the algorithm adaptive ShiversSort is at most $\lceil \log_2(n) \rceil + 1 - \ell_{\min}$.*

Proof. Let $\mathcal{S} = (R_1, \dots, R_h)$ be some stack encountered while executing adaptive ShiversSort. An immediate consequence of Lemma 7 is that $\ell_1 > \ldots > \ell_{h-2}$. It follows that

$$\ell_i \geqslant \ell_{h-2} + (h-2-i) \geqslant \ell_{\min} + (h-2-i)$$

for all $i \leqslant h-2$, and thus that $r_i \geqslant 2^{\ell_i} \geqslant 2^{h-2-i+\ell_{\min}}$ as well. Consequently,

$$n \geqslant r_1 + \ldots + r_h \geqslant \left(\sum_{i=1}^{h-2} r_i \right) + \mathrm{s}_{\min} + 1 \geqslant \left(\sum_{i=1}^{h-2} 2^{h-2-i+\ell_{\min}} \right) + 2^{\ell_{\min}} + 1 = 2^{h-2+\ell_{\min}} + 1,$$

and therefore $h < \log_2(n) + 2 - \ell_{\min}$. □

As an immediate consequence, and since $\mathrm{s}_{\min} \geqslant 1$, i.e., $\ell_{\min} \geqslant 0$, we already know that no stack of size larger than $\lceil \log_2(n) \rceil + 1$ will ever be required. Then, in practice, it remains to check whether the stack sizes currently used in the implementations of TimSort in both languages Python and Java would be sufficient for sorting arrays of any size. In fact, it might even be possible to use smaller arrays than those currently in use, but whether making such a change would be worth the effort is not clear. Finally, we only focus here on arrays of *meaningful* length, which means that Python and Java cannot deal with arrays of arbitrary sizes; we make that point clearer below.

Corollary 62. *Whenever sorting an array of* meaningful *length, the size of the array used to implement TimSort's stack in both languages Python and Java is large enough to also implement the stack of adaptive ShiversSort.*

Proof. Let n be the length of the array to be sorted. In Python, TimSort's stack is simulated by an array of size $h_{\max} = 85$ and contains only runs of size at least 32 [22], i.e., $\ell_{\min} = 5$. Thus, Proposition 61 proves that this array is large enough whenever $n \leqslant 2^{89}$: this is more than could ever be handled by an actual computer, and therefore large enough for all reasonable purposes.

In Java, the situation is slightly different, because we only have $\ell_{\min} = 4$, and TimSort's stack is simulated by an array whose size depends on n and is quite smaller [4]. More precisely, this size is

the integer $h_{\max}$ defined by

$$h_{\max} = \begin{cases} 5 & \text{if } n \leqslant 119, \\ 10 & \text{if } 120 \leqslant n \leqslant 1541, \\ 24 & \text{if } 1542 \leqslant n \leqslant 119150, \\ 49 & \text{if } 119151 \leqslant n \leqslant 2^{31} - 5. \end{cases}$$

Finally, Java fails to handle and sort arrays of length $n \geqslant 2^{31} - 4$, which makes these cases irrelevant for our purposes. We complete the proof by checking that $h_{\max} \geqslant \lceil \log_2(n) \rceil + 1 - \ell_{\min}$ when n is equal to 119, 1541, 119150 or $2^{31} - 5$, and therefore in all cases where $n \leqslant 2^{31} - 5$. □

6.4 Switching from TimSort to (length-)adaptive ShiversSort in Python and Java

The general structures of the merge policies of TimSort, adaptive ShiversSort and length-adaptive ShiversSort are remarkably similar. However, a crucial difference that adaptive ShiversSort and length-adaptive ShiversSort have with TimSort is that, instead of comparing directly the lengths of the runs involved, they require comparing their levels. More precisely, a key step is to check efficiently whether $\ell_{h-2} \leqslant \max\{\ell_h, \ell_{h-1}\}$, which might be bothersome if implemented carelessly. Fortunately, given three integers r_i, r_{i+1} and r_{i+2}, checking whether $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$ is made very easy by the use of boolean integer operations. When $c = 1$, this is the object of the following two-line algorithm.

Algorithm 11: Checking whether $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, when $c = 1$

Input : Integers r_i, r_{i+1} and r_{i+2}
Result: **true** if $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, and **false** otherwise.
Note: We use bit-wise **and**, **or** and **not** binary operations on integers.

1 $x \leftarrow r_{i+1}$ **or** r_{i+2}
2 **return** $x > (r_i$ **and** (**not** $x))$

Proposition 63. *When given positive integers r_i, r_{i+1} and r_{i+2} as input, and provided that $c = 1$, Algorithm 11 returns **true** if $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, and **false** otherwise.*

Proof. Let ℓ, x' and ℓ' be the integers defined by $\ell = \lfloor \log_2(x) \rfloor$, $x' = (r_i$ **and** (**not** $x))$, and $\ell' = \lfloor \log_2(x') \rfloor$. By construction, we have $\ell = \max\{\ell_{i+1}, \ell_{i+2}\}$. Thus, we shall prove that $\ell_i \leqslant \ell$, which is the inequality that Algorithm 11 is supposed to check, if and only if $x' < x$, which is what Algorithm 11 actually checks.

Below, we write integers in base 2: the bit of weight 2^i of an integer n is denoted by $\mathrm{b}_i(n)$, so that $n = \sum_{i \geqslant 0} 2^i \mathrm{b}_i(n)$. We complete the proof by distinguishing two cases:

- If $\ell_i \leqslant \ell$, we have $\ell' \leqslant \ell_i$ by definition of x', and thus $\ell' \leqslant \ell$. Since $\mathrm{b}_{\ell'}(x') = 1 = \mathrm{b}_\ell(x) \neq \mathrm{b}_\ell(x')$, we also have $\ell \neq \ell'$. It follows that $\ell' < \ell$, and thus that $x' < x$.
- If $\ell_i > \ell$, we have $\mathrm{b}_{\ell_i}(r_i) = \mathrm{b}_{\ell_i}(\textbf{not}\ x) = 1$, and therefore both $\mathrm{b}_{\ell_i}(x') = 1$ too. This proves that $\ell' \geqslant \ell_i > \ell$, and it follows that $x' > x$. □

Consequently, and as promised, switching from TimSort to adaptive ShiversSort would be extremely easy in practice. For instance, in Python, it would suffice to replace the lines 1940 to 1951 of the implementation of TimSort [22] by the following 8 lines:

```
1940  Py_ssize_t n = ms->n - 3;
1941  if (n >= 0) {
1942    Py_ssize_t x = p[n+1].len | p[n+2].len;
1943    if (x > (p[n].len &~x)) {
1944      if (merge_at(ms, n) < 0)
1945        return -1;
1946    }
1947  }
```

In Java, it would also suffice to replace the lines 405 to 412 of the implementation of TimSort [4] by the following 4 lines:

```
405  int n = stackSize - 3;
406  int x = runLen[n+1] | runLen[n+2];
407  if (n < 0 || x <= (runLen[n] &~x))
408    break;
```

Similarly, when $c = n + 1$, where n is the length of the array to be sorted, checking whether $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$ is the object of the following five-line algorithm.

Algorithm 12: Checking whether $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, when $c = n + 1$

Input : Integers r_i, r_{i+1}, r_{i+2} and n
Result: **true** if $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, and **false** otherwise.

```
1 m ← n
2 while m ⩾ r_i:
3     m ← ⌊m/2⌋
4 return (m < r_{i+1}) or (m < r_{i+2})
```

Proposition 64. *When given positive integers r_i, r_{i+1} and r_{i+2} as input, and provided that $c = n + 1$, Algorithm 12 returns **true** if $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, and **false** otherwise.*

Proof. When one checks whether $m \geqslant r_i$ (in line 2) for the k^{th} time, the variable m is equal to $\lfloor n/2^{k-1} \rfloor$. Since $n \geqslant r_i$, this test is positive when $k = 1$. More generally, it is positive if and only if $n \geqslant 2^{k-1} r_i$. Since $k \geqslant 1$ and r_i is an integer, the latter inequality holds if and only if $n+1 > 2^{k-1} r_i$ or, equivalently, if $1 - k > \ell_i$. Thus, we keep going into the loop of lines 2 and 3 until $k = 1 - \ell_i$.

At this point, we go to line 4 and, for the same reasons as before, we have $m \geqslant r_{i+1}$ if and only if $1 - k > \ell_{i+1}$. It follows that $m < r_{i+1}$ if and only if $\ell_i = 1 - k \leqslant \ell_{i+1}$ and, similarly, that $m < r_{i+2}$ if and only if $\ell_i \leqslant \ell_{i+2}$. □

Thus, switching from TimSort to length-adaptive ShiversSort would also be very easy, although it requires modifications in more than one place. For instance, in Python, it would suffice to replace the lines 1934, 1940 to 1951, and 2041 of the implementation of TimSort [22] by the following 11 lines:

```
1934  merge_collapse(MergeState *ms, Py_ssize_t m)
...
1940  Py_ssize_t n = ms->n - 3;
1941  if (n >= 0) {
1942    while (m >= p[n].len)
1943      m >>= 1;
1944    if (m < p[n+1].len || m < p[n+2].len) {
1945      if (merge_at(ms, n) < 0)
1946        return -1;
1947    }
1948  }
...
2401  if (merge_collapse(&ms, saved_ob_size) < 0)
```

Similarly, in Java, it would suffice to replace the lines 213 and 403 to 412 of the implementation of TimSort [4] by the following 8 lines:

```
213  ts.mergeCollapse(hi - lo);
...
403  private void mergeCollapse(int len) {
404    while (stackSize > 2) {
405      int n = stackSize - 3;
406      while (len >= runLen[n])
407        len >>= 1;
408      if (len >= runLen[n+1] && len >= runLen[n+2])
409        break;
```

REFERENCES

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of Timsort. In *26th Annual European Symposium on Algorithms (ESA 2018)*, 4:1–4:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.
Extended version available at: https://arxiv.org/abs/1805.08612.

[2] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: From Merge sort to Timsort. Research report hal-01212839, HAL, 2015.

[3] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[4] Josh Bloch et al. Timsort implementation in Java, retrieved on 01/09/2021.
https://github.com/openjdk/jdk/blob/3afeb2cb4861f95fd20c3c04f04be93b435527c0/src/java.base/share/classes/java/util/ComparableTimSort.java.

[5] Sam Buss and Alexander Knop. Strategies for stable merge sorting. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1272–1290. Society for Industrial and Applied Mathematics, 2019.

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

[7] Edsger Dijkstra. Smoothsort, an alternative for sorting in situ. In *Theoretical Foundations of Programming Methodology*, 3–17. Springer, 1982.

[8] Vladmir Estivill-Castro and Derick Wood. *A survey of adaptive sorting algorithms.* ACM Computing Surveys, 24(4):441–476, 1992.

[9] Adriano Garsia and Michelle Wachs. A new algorithm for minimal binary search trees. In *SIAM Journal on Computing*, 6(4):622–642, 1977.

[10] Herman Goldstine and John von Neumann. *Planning and coding of problems for an electronic computing instrument.* Research report, Institute for Advanced Study, 1947.

[11] Mordecai Golin and Robert Sedgewick. Queue-mergesort. *Information Processing Letters*, 48(5):253–259, 1993.

[12] Tony Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[13] Te Hu and Alan Tucker. Optimal computer search trees and variable-length alphabetical codes. In *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[14] Vincent Jugé. Adaptive Shivers sort: An alternative sorting algorithm. In *2020 ACM-SIAM Symposium on Discrete Algorithms*, 1639–1654, 2020.

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

[16] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 34(4):318–325, 1985.

[17] Alistair Moffat, Gary Eddy, and Ola Petersson. Splaysort: Fast, versatile, practical. *Software: Practice and Experience*, 26(7):781–797, 1996.

[18] J. Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In *26th Annual European Symposium on Algorithms (ESA 2018)*, 63:1–63:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

[19] Tim Peters. Timsort description, retrieved on 01/09/2021.
https://github.com/python/cpython/blob/e5c8ddb1714fb51ab1defa24352c98e0f01205dc/Objects/listsort.txt.

[20] Olin Shivers. A simple and efficient natural merge sort. Technical report, Georgia Institute of Technology, 2002.

[21] Tadao Takaoka. Partial solution and entropy. In *Mathematical Foundations of Computer Science 2009*, 700–711. Springer, 2009.

[22] Guido van Rossum and 69 other contributors. Timsort implementation in CPython, retrieved on 01/09/2021. https://github.com/python/cpython/blob/e5c8ddb1714fb51ab1defa24352c98e0f01205dc/Objects/listobject.c.

[23] John Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.