Adaptive Shivers Sort: An Alternative Sorting Algorithm

Vincent Jugé*

Abstract

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 was introduced by Buss and Knop that captures "TimSort-like" algorithms.

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.

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 1940's, 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 [8], QuickSort [10], HeapSort [19], SmoothSort [6], SplaySort [14], . . .

In 2002, Tim Peters, a software engineer, created a new sorting algorithm, which was called TimSort [16]. 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* [7, 3], as illustrated in Figure 1. Such decompositions were already used by Knuth NaturalMergeSort [12], 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 together. Thus, all algorithms sharing this feature of NaturalMergeSort are also called *natural* merge sorts.

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

Figure 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, then it 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

*Université Paris-Est, LIGM (UMR 8049), CNRS, ENPC, ESIEE Paris, UPEM, F77454 Marne-la-Vallée, France 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 $\mathcal{O}(n\log(n))$ 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 [5, 1].

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 [17], as well as Takaoka's MinimalSort [18] (equivalent constructions of this algorithm were also obtained in [3]), Buss and Knop's α-MergeSort [4], and the most recent algorithms PeekSort and PowerSort, due to Munro and Wild [15].

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 that these elements appear in the input. This is very important for merge sorts, because only adjacent runs will be merged, which allows merging 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 $\mathcal{O}(n\log(n))$, and, for all of them except ShiversSort, they even do it in time $\mathcal{O}(n+n\log(\rho))$, where ρ is the number of runs of the array. This is optimal in the model of sorting by comparisons [13], 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 $r_1,\ldots,r_\rho$, these algorithms run in $\mathcal{O}(n+n\mathcal{H})$, where $\mathcal{H}$ is defined as $\mathcal{H}=H(r_1/n,\ldots,r_\rho/n)$ and $H(x_1,\ldots,x_\rho)=-\sum_{i=1}^{\rho}x_i\log_2(x_i)$ 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, five 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, 4, 9, 15]: 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.

In this new model, one can prove that the merge cost of any natural merge sort must be at least $n\mathcal{H}+\mathcal{O}(n)$. This makes MinimalSort, PeekSort and PowerSort the only sorting algorithms with an optimal merge cost, as shown in the last column of Table 1. Since PeekSort and PowerSort are stable, they should be natural options for succeeding TimSort as standard sorting algorithm in Python or Java. Nevertheless, and although they have implementations similar to that of TimSort, their merge policies are arguably more complicated, as illustrated in Section 2.

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

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

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.

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 [4], 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, then 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. Then, 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 4th 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 (ii) 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 (ii) 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 $n\mathcal{H} + \mathcal{O}(n)$ 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 $n\mathcal{H} + \mathcal{O}(n)$, thereby answering a question left open by Buss and Knop in [4]. 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 cost of PowerSort, in every case, is bounded between $n\mathcal{H}$ and $n(\mathcal{H} + 2)$, these bounds being both close and optimal for any stable merge sort. The merge cost of PeekSort is only bounded from above by $n(\mathcal{H} + 3)$; this is slightly worse than PowerSort, and thus we will not discuss this algorithm below. 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 loses the online property, 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 [4], 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.

2 Adaptive ShiversSort and related algorithms

In this section, we describe the run merge policy of the algorithm adaptive ShiversSort and of related algorithms. The merge policy of adaptive ShiversSort is depicted in Algorithm 1. For the ease of subsequent sections, and because it does not make any proof harder, we shall consider adaptive ShiversSort as a parameterised algorithm: in addition to the array to sort, this algorithm also requires a positive integer c as parameter.

In subsequent sections, we will consider most specifically two choices for the parameter c, by either setting $c = 1$ or by setting $c = n + 1$, where n is the length of the array to be sorted. We respectively call 1-adaptive ShiversSort and length-adaptive ShiversSort the algorithms obtained with these two parameter choices. At any moment except when we will specifically refer to length-adaptive ShiversSort, in Section 4, the reader may safely assume that $c = 1$.

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 the conditions of cases #1 to #4 apply. In particular, since these conditions only refer to the values of ℓ_1, ℓ_2 and ℓ_3, and since only the runs R_1, R_2 and R_3

Algorithm 1: 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: We denote the height of the stack $\mathcal{S}$ by $h$,
      and its $i^{\text{th}}$ top-most run (for $1 \leqslant i \leqslant h$) by
      $R_i$. The length of $R_i$ is denoted by $r_i$, and
      we set $\ell_i = \lfloor \log_2(r_i/c) \rfloor$.
      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  $\mathcal{S}$ ← an empty stack
3  while true:                                   ▷ main loop
4     if $h \geqslant 3$ and $\ell_1 \geqslant \ell_3$:                  ▷ case #1
5        merge the runs $R_2$ and $R_3$
6     else if $h \geqslant 3$ and $\ell_2 \geqslant \ell_3$:             ▷ case #2
7        merge the runs $R_2$ and $R_3$
8     else if $h \geqslant 2$ and $\ell_1 \geqslant \ell_2$:             ▷ case #3
9        merge the runs $R_1$ and $R_2$
10    else if runs $\neq \emptyset$:                       ▷ case #4
11       remove a run $R$ from runs and push $R$ onto $\mathcal{S}$
12    else break
13 while $h \geqslant 2$: merge the runs $R_1$ and $R_2$
```

may be merged, this algorithm falls within the class of 3-aware stable sorting algorithms such as described by Buss and Knop [4] as soon as the choice of c is independent of the input. This is the case of 1-adaptive ShiversSort, but not of other variants such as length-adaptive ShiversSort.

Observe here that both cases #1 and #2 result in merging the runs R_2 and R_3. We decided to separate these cases in this presentation of the algorithm in order to simplify the analysis performed in Section 3.

Let us then present briefly some related algorithms. Like adaptive ShiversSort, all of these algorithms rely on discovering and maintaining runs in stack, although their merge policies follow different rules. More precisely, each of these policies is obtained by modifying the *main loop* of adaptive ShiversSort.

The first algorithm we present is TimSort, and is due to Peters [16]. Its main loop is presented in Algorithm 2. As mentioned in the introduction, this algorithm enjoys a $\mathcal{O}(n + n\mathcal{H})$ worst-case merge cost. Two crucial elements in achieving this result consist in showing that some invariant (the fact that $r_i + r_{i+1} \leqslant r_{i+2}$ for all $i \geqslant 3$) is maintained, and in avoiding merging newly pushed runs until their length is comparable with that of the runs stored in the top of the stack. Original versions of TimSort missed the case ♡, which made

Algorithm 2: TimSort main loop

```
3  while true:                                   ▷ main loop
4     if $h \geqslant 3$ and $r_1 > r_3$:                      ▷ case ♢
5        merge the runs $R_2$ and $R_3$
6     else if $h \geqslant 2$ and $r_1 \geqslant r_2$:                ▷ case ♠
7        merge the runs $R_1$ and $R_2$
8     else if $h \geqslant 3$ and $r_1 + r_2 \geqslant r_3$:           ▷ case ♣
9        merge the runs $R_1$ and $R_2$
10    else if $h \geqslant 4$ and $r_2 + r_3 \geqslant r_4$:          ▷ case ♡
11       merge the runs $R_1$ and $R_2$
12    else if runs $\neq \emptyset$:                       ▷ case #4
13       remove a run $R$ from runs and push $R$ onto $\mathcal{S}$
14    else break
```

the invariant invalid and caused several implementation bugs [5, 1]. Yet, even that flawed version of the algorithm managed to enjoy a $\mathcal{O}(n + n\mathcal{H})$ worst-case merge cost. Unfortunately, the constant hidden in the $\mathcal{O}$ is rather high, as outlined by the following result.

Theorem 2.1. *The worst-case merge cost of TimSort on inputs of length n is bounded from above by $3/2\, n\mathcal{H} + \mathcal{O}(n)$ and bounded from below by $3/2\, n \log_2(n) + \mathcal{O}(n)$.*

Proof. The lower bound was proven in [4]. The upper bound is proven 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. With this idea in mind, one hope comes from the algorithm ShiversSort, which was invented by Shivers [17]. It is obtained by omitting the cases #1 and #2 of adaptive ShiversSort, i.e., using the following main loop:

```
3  while true:                                   ▷ main loop
4     if $h \geqslant 2$ and $\ell_1 \geqslant \ell_2$:                 ▷ case #3
5        merge the runs $R_1$ and $R_2$
6     else if runs $\neq \emptyset$:                       ▷ case #4
7        remove a run $R$ from runs and push $R$ onto $\mathcal{S}$
8     else break
```

This sorting algorithm *may* merge a newly pushed run even if its length is much larger than those of the runs stored in the stack, and therefore it is *not* adaptive to the number of runs nor 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 2.2. *The worst-case merge cost of ShiversSort on inputs of length n that decompose into ρ monotonic runs is both bounded from above by $n \log_2(n) + \mathcal{O}(n)$ and bounded from below by $\omega(n \log_2(\rho))$.*

This result was proven in [17, 4]. 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 3.4, is the fact that the sequence $\ell_1, \ell_2, \ldots, \ell_h$ shall always be increasing after some fixed index: that index is 2 in the proof of [4], and it is 3 in Lemma 3.4. In essence, this invariant is similar to that of TimSort, but it allows decreasing the associated constant hidden in the $\mathcal{O}$ notation from 3/2 to 1.

The very idea of integrating features from these two algorithms led Buss and Knop to invent the algorithm augmented ShiversSort [4]. This algorithm is obtained by using the following main loop:

```
3  while true:                                              ▷ main loop
4    if $h \geqslant 3$, $r_1 \geqslant r_3$ and $\ell_1 \geqslant \ell_2$:           ▷ case #1'
5      merge the runs $R_2$ and $R_3$
6    else if $h \geqslant 2$ and $\ell_1 \geqslant \ell_2$:                     ▷ case #3
7      merge the runs $R_1$ and $R_2$
8    else if runs $\neq \emptyset$:                              ▷ case #4
9      remove a run $R$ from runs and push $R$ onto $\mathcal{S}$
10   else 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 2.3. *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 $r_1, \ldots, r_\rho$ be the sequence of run lengths defined by $r_{2i-1} = 6$ and $r_{2i} = 2$ for all $i \leqslant k$. Note that $r_1 + \ldots + r_\rho = n$.

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

- push rhe runs R_1 and R_2 (case #4);
- for all $i \in \{2, \ldots, k\}$, push the run R_{2i-1} (case #4), then merge the runs R_{2i-2} and R_{2i-3} (case #1'), and push the run R_{2i} (case #4 again);
- keep merging the last two runs on the stack (line 13): 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

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

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 $n(n-1)$. ∎

Finally, as mentioned in Section 1, PowerSort enjoys excellent complexity guarantees, with a merge cost bounded from above by $n(\mathcal{H}+2)$. However, it is not a k-aware algorithm for any k, and its merge policy is more complicated than that of TimSort. Indeed, whether two consecutive runs R and R' should be merged does not depend directly on their lengths, but on their *powers*, which are defined as follows.

Let R be a run resulting from the merge of several (original) runs $R_i, \ldots, R_j$, and let pos be the last position contained in the run R. The power of R is defined as the least integer p such that

$$\lfloor 2^p(4\,\mathsf{pos} + 1 - 2r_j)/n \rfloor < \lfloor 2^p(4\,\mathsf{pos} + 1 + 2r_{j+1})/n \rfloor.$$

Then, two consecutive runs R and R' should be merged if $p > p'$. Hence, PowerSort requires additional data structures (e.g., a second stack storing the powers of the runs) that are not needed in TimSort nor in any k-aware algorithm.

In what follows, we focus on designing an algorithm that would successfully integrate the features of both TimSort and ShiversSort, while keeping the simplicity of their merge policies, thereby being k-aware for a small integer k. This new algorithm is adaptive ShiversSort.

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. This is outlined by the result below.

Theorem 3.1. *The merge cost of adaptive ShiversSort is bounded from above by $n(\mathcal{H} + \Delta)$, where $\Delta = 24/5 - \log_2(5) \approx 2.478$.*

The remainder of this section is devoted to proving the following weaker variant of Theorem 3.1, which already provides us with an $n(\mathcal{H}+3)$ upper bound. The proof of this variant is far less technical, and therefore much more insightful.

Theorem 3.2. *For every value of the parameter c, the merge cost of 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.*

In what follows, the value of the parameter c is fixed once and for all. Then, we denote by r the length of a run R, by ℓ the integer $\lfloor \log_2(r/c) \rfloor$ and by λ the real number $\{\log_2(r/c)\} = \log_2(r/c) - \ell$. 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., we denote by r' the length of the run R', by ℓ' the integer $\lfloor \log_2(r'/c) \rfloor$ and by λ' the real number $\{\log_2(r'/c)\}$. In particular, we will commonly denote the stack by $(R_1, \ldots, R_h)$, where R_k is the k^{th} top-most run of the stack. The length of R_k is then denoted by r_k, and we set $\ell_k = \lfloor \log_2(r_k/c) \rfloor$.

With this notation in mind, we first prove two auxiliary results about the levels of the runs manipulated throughout the algorithm.

Lemma 3.3. *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, it comes that

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

and therefore that $\ell'' \leqslant \ell' + 1$. ☐

Lemma 3.4. *At any time during the execution of the algorithm, if the stack of runs is $\mathcal{S} = (R_1, \ldots, R_h)$, we have:*

$$\ell_2 \leqslant \ell_3 < \ell_4 < \ldots < \ell_h. \tag{1}$$

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.

We prove now that, if some stack $\mathcal{S} = (R_1, \ldots, R_h)$ satisfies (1) and is updated into a new stack $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ by merging the runs R_1 and R_2 or R_2 and R_3, or by pushing the run $\overline{R}_1$, then $\overline{\mathcal{S}}$ also staisfies (1). This is done by a case analysis:

- If two runs were just merged, then $\overline{h} = h - 1$, $\overline{R}_i = R_{i+1}$ for all $i \geqslant 3$, and $\overline{R}_2$ is either equal to R_3 (if R_1 and R_2 were merged) or to the result of the merge between the runs R_2 and R_3 (if R_2 and R_3 were merged). Thanks to Lemma 3.3, this means that either $\overline{\ell}_2 = \ell_3$ or $\overline{\ell}_2 \leqslant \max\{\ell_2, \ell_3\} + 1 = \ell_3 + 1$. Moreover, since (1) holds in $\mathcal{S}$, we already have $\ell_3 < \overline{\ell}_3 < \overline{\ell}_4 < \ldots < \overline{\ell}_{\overline{h}}$. It follows that $\overline{\ell}_2 \leqslant \ell_3 + 1 \leqslant \overline{\ell}_3$, which shows that (1) also holds in $\overline{\mathcal{S}}$.
- If the run $\overline{R}_1$ was just pushed (i.e., if case #4 just arose), then $\overline{h} = h + 1$, and $\overline{R}_i = R_{i-1}$ for all $i \geqslant 2$. Since (1) holds in $\mathcal{S}$, we already have $\overline{\ell}_4 < \overline{\ell}_5 < \ldots < \overline{\ell}_{\overline{h}}$. Moreover, since case #4 occurred, none of the conditions for triggering the cases #1 to #3 holds in $\mathcal{S}$. This means that $\ell_1 < \ell_2 < \ell_3$ or, equivalently, that $\overline{\ell}_2 < \overline{\ell}_3 < \overline{\ell}_4$, which shows that (1) holds in $\overline{\mathcal{S}}$. ☐

Roughly speaking, Lemma 3.4 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-most run, whose length we have no control on. As suggested in Section 2, this property was already crucial in the complexity proofs of several algorithms such as ShiversSort, TimSort or α-MergeSort.

Then, the proof of Theorem 3.2 consists 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 merge operations: one part, for a cost of r, is called the *merge of R with R'*, and the other part, for a cost of r', is called the *merge of R' with R*. Together, these operations indeed consist in merging R and R' with each other, and their costs add up to $r + r'$, as expected.

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

Lemma 3.5. *The total cost of expanding 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

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

expanding merges.

Consequently, if the array is initially split into runs of lengths $r_1, \ldots, r_\rho$, the total cost of expanding 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 merges is at most $3n - \Lambda - \rho$. This requires further splitting sequences of merges based on the case that triggered these merges. Hence, when discussing the various updates that may arise, we also call *#k-update* every update triggered by a case $\#k$: this is a merge if $k \leqslant 3$, and a push if $k = 4$ (we may abusively speak of a $\#k$-merge or #4-push). It turns out that these updates may not occur in an arbitrary order.

Lemma 3.6. *No #2-merge is immediately followed by a #1-merge, and no #3-merge is immediately followed by a #1-merge or a #2-merge.*

Proof. Let m be a merge. We denote by $\mathcal{S} = (R_1, \ldots, R_h)$ the stack before the merge and by $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ the stack after the merge, so that $\overline{h} = h-1$.

If m is a #2-merge, then $h \geqslant 3$, $\overline{R}_1 = R_1$ and $\overline{R}_3 = R_4$. We further know that $\ell_3 > \ell_1$, unless what a #1-merge would have been triggered instead of m. It follows from (1) that $\overline{\ell}_3 = \ell_4 > \ell_3 > \ell_1 = \overline{\ell}_1$, which proves that m cannot be followed by a #1-merge.

Similarly, if m is a #3-merge, we know that it cannot be followed by a #1-merge or a #2-merge unless $\overline{h} \geqslant 3$. In that case, we have $h \geqslant 4$, $\overline{R}_2 = R_3$, $\overline{R}_3 = R_4$, and $\overline{R}_1$ is the result of merging R_1 and R_2. We also know that $\ell_3 > \max\{\ell_1, \ell_2\}$, unless what a #1-merge or #2-merge would have been triggered instead of m. Lemma 3.3 then shows that $\overline{\ell}_1 \leqslant \max\{\ell_1, \ell_2\} + 1 \leqslant \ell_3$. Thus, it follows from (1) that

$$\overline{\ell}_3 = \ell_4 > \ell_3 = \overline{\ell}_2 \geqslant \overline{\ell}_1, \tag{2}$$

which proves that m cannot be followed by a #1-merge or a #2-merge. $\square$

A consequence of Lemma 3.6 is the following one. Let R be some run in the array to be sorted, and let $(m_1, \ldots, m_k)$ be the (possibly empty) sequence of merges performed after R has been pushed onto the stack but before any other run is pushed onto the stack (or, if R is the right-most run of the array, before the end of the main loop of adaptive ShiversSort).

This sequence can be subdivided in three (possibly empty) consecutive subsequences and one (possibly non-existent) special merge, as illustrated in Figure 2. First, the *starting sequence* of R contains #1-merges only; then, the *middle sequence* of R, contains #2-merges only; then, we call *critical merge* of R the first #3-merge that comes, if it exists; finally, the *ending sequence* of R contains all subsequent #3-merges.

$$\underset{\substack{\text{run}\\\text{push}}}{\#4}\ ,\ \underbrace{\#1,\#1,\#1}_{\substack{\text{starting}\\\text{sequence}}}\ ,\ \underbrace{\#2,\#2,\#2}_{\substack{\text{middle}\\\text{sequence}}}\ ,\ \underset{\substack{\text{critical}\\\text{merge}}}{\#3}\ ,\ \underbrace{\#3,\#3}_{\substack{\text{ending}\\\text{sequence}}}\ ,\ \underset{\substack{\text{run}\\\text{push}}}{\#4}$$

Figure 2: A run R is pushed, in the left-most #4-update. Then come its starting sequence, its middle sequence, its critical merge and its ending sequence. Finally, a new run R' is pushed, in the right-most #4-update.

Below, we also call *non-expanding cost* of a sequence of merges the total cost of the *non-expanding merges* included in this sequence. Before looking more closely at the non-expanding cost of the starting, middle and ending sequences, we first prove invariants similar to that of Lemma 3.4.

Lemma 3.7. *At any time during an ending sequence, including just before it starts, we have $\ell_1 \leqslant \ell_2$.*

Proof. Let $\mathcal{S} = (R_1, \ldots, R_h)$ be a stack obtained during an ending sequence. The update from which the stack $\mathcal{S}$ resulted was a #3-merge. In this context, the inequality (2) obtained when proving Lemma 3.6 states that $\ell_3 > \ell_2 \geqslant \ell_1$, which completes the proof. $\square$

Corollary 3.8. *Each middle or ending sequence contains intrinsically expanding merges only.*

Proof. Let m be a merge of a middle or ending sequence, and let $\mathcal{S} = (R_1, \ldots, R_h)$ be the stack just before the merge occurs:

- If m belongs to a middle sequence, then it is a #2-merge between the runs R_2 and R_3. Hence, we have $\ell_2 \leqslant \ell_3$ (because of Lemma 3.4) and $\ell_3 \leqslant \ell_2$ (because it is a #2-merge), and thus m is intrinsically expanding.
- If m belongs to an ending sequence, then it is a #3-merge between the runs R_1 and R_2. Hence, we have $\ell_1 \leqslant \ell_2$ (because of Lemma 3.7) and $\ell_2 \leqslant \ell_1$ (because it is a #3-merge), and thus m is also intrinsically expanding. $\square$

Lemma 3.9. *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. $\square$

With the help of these auxiliary results, we may now prove the following statement, whose proof is also the most technical one in this section.

LEMMA 3.10. *The total non-expanding cost of the critical merge (if it exists) and of the starting, middle and ending sequences of a run R is at most* $(2 - \lambda)r - 1$.

Proof. Let $\mathcal{S} = (R, R_2, \ldots, R_h)$ be the stack just after the run R has been pushed, and let k be the largest integer such that $\ell_k \leqslant \ell$. We respectively denote by $\mathcal{M}_{\text{start}}$, $\mathcal{M}_{\text{mid}}$ and $\mathcal{M}_{\text{end}}$ the starting, middle and ending sequences of R, and by m_{cr} its critical merge.

The starting sequence $\mathcal{M}_{\text{start}}$ consists in merging the run R_2 with R_3, then merging the resulting run successively with $R_4, R_5, \ldots, R_k$. Due to Lemma 3.4, the non-expanding merges of $\mathcal{M}_{\text{start}}$ involve at most once each of the runs $R_3, \ldots, R_k$, when these runs are first merged. Let k' be the largest integer, if any, such that the first merge of $R_{k'}$ is non-expanding, and let R^* be the run into which $R_{k'}$ is merged. Then, all runs $R_2, \ldots, R_{k'}$ are merged into that run, and thus the non-expanding cost of $\mathcal{M}_{\text{start}}$ is at most $r_3 + \ldots + r_{k'-1} + r_{k'} = r^* - r_2 \leqslant r^* - 1$. Since $\lfloor \log_2(r^*/c) \rfloor = \ell_{k'} \leqslant \ell$, it even follows that $r^* < 2^{\ell+1}c = 2^{1-\lambda}r$, and therefore the non-expanding cost of $\mathcal{M}_{\text{start}}$ is at most $2^{1-\lambda}r - 1$.

Then, Corollary 3.8 ensures that the sequences $\mathcal{M}_{\text{mid}}$ and $\mathcal{M}_{\text{end}}$ contain expanding merges only, hence their non-expanding cost is 0.

Finally, assume that the critical merge m_{cr} exists and is non-expanding, and let also $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ be the stack just before it occurs. Note that $\overline{R}_1 = R$. Then, it must be the case that $\ell = \overline{\ell}_1 > \overline{\ell}_2$. Moreover, if the starting sequence $\mathcal{M}_{\text{start}}$ had a non-zero non-expanding cost, then the above-defined run R^* turns out to be entirely contained in the run $\overline{R}_2$. It follows that $r + r^* \leqslant r + \overline{r}_2 < 2^{\ell+1}c = 2^{1-\lambda}r$.

Let us gather the above results. First, the starting sequence $\mathcal{M}_{\text{start}}$ has a non-expanding cost bounded from above by $r^* - 1 \leqslant 2^{1-\lambda}r - 1$. Then, the non-expanding costs of both sequences $\mathcal{M}_{\text{mid}}$ and $\mathcal{M}_{\text{end}}$ are equal to 0. Finally, if the merge m_{cr} is well-defined and non-expanding, its non-expanding cost is exactly r, and we also have $r + r^* - 1 \leqslant 2^{1-\lambda}r - 1$.

Hence, in all cases, it the total non-expanding cost of the sequences $\mathcal{M}_{\text{start}}$, $\mathcal{M}_{\text{mid}}$ and $\mathcal{M}_{\text{end}}$ and of m_{cr} is at most $2^{1-\lambda}r - 1$. Using Lemma 3.9 completes the proof. ∎

Gathering these results proves Theorem 3.2 itself.

Proof of Theorem 3.2. Lemma 3.5 states that the total cost of expanding merges is at most $n\mathcal{H} + \Lambda$. Then, Lemma 3.10 states that the total non-expanding cost of the critical merge and of the starting, middle and ending sequences of a given run R is at most $(2-\lambda)r-1$. Hence, and taking all runs into account, the total non-expanding cost of these sequences is at most $2n - \Lambda - \rho$.

It remains to take care of the non-expanding cost of the runs merged in line 13 of Algorithm 1. The merges performed in line 13 are the same merges as those that would occur in the *starting sequence* of an additional (fictitious) run of length n that we would have appended to our array. Performing the analysis of Lemma 3.10 in that special case proves that the total non-expanding cost of these merges is at most $n - 1$. This completes the proof of Theorem 3.2. ∎

Now that Theorem 3.2 has been proven, let us present briefly the ideas that lead to Theorem 3.1 itself. Although the main interest of Theorem 3.1 is that the constant Δ it refers to is optimal, as underlined in Proposition 4.3, the tools it is based on are quite more complex, and seeing why these tools fit together is less intuitive. A complete proof is available in the full version of this article [11].

Main ideas of the proof of Theorem 3.1. The cost allocation used for proving Theorem 3.2 is not sufficient to prove the better bound mentioned in Theorem 3.1. Therefore, we must define a notion of *potential* of a sequence of runs, which will fit more closely the merge cost needed to merge this sequence, and which is just a sum of potentials of individual runs.

The potential of a run R is defined as follows. Let $\ell \in \mathbb{Z}$ and $r_\bullet \in [0, 1)$ be defined by the relation $r = 2^\ell(1+r_\bullet)c$, and let $\Phi : [0, 1] \mapsto \mathbb{R}$ be the function defined by $\Phi : x \mapsto \max\{(2 - 5x)/3, 1/2 - x, 0\}$. The potential of R is the real number $\mathbf{Pot}(\mathrm{R}) = 2^\ell\,\Phi(r_\bullet)\,c - \ell\,r$ if R is in the stack, and $\mathbf{Pot}(\mathrm{R}) = 2^{\ell+1}\,c - \ell\,r$ if R has not yet been pushed onto the stack.

The proof itself consists in showing that the merges performed by adaptive ShiversSort can be gathered into groups whose merge cost is bounded from above by the variation of this potential, which necessitates careful disjunctions of cases (e.g., these cases may depend on which of the starting, middle or ending sequences are non-empty, after a run R has been pushed onto the stack). Theorem 3.1 then follows from evaluating the global potential variation during the algorithm. ∎

4 Best-case and worst-case merge costs

In the introduction, we mentioned that, unlike PowerSort, the algorithm adaptive ShiversSort is 3-aware, which suggests that it might be preferred to PowerSort. However, the worst-case merge cost of PowerSort is only of $n(\mathcal{H}+2)$, 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 might be the best option. Furthermore, in other cases than the worst case, we do have little information on the relative costs of adaptive ShiversSort and of PowerSort.

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 is actually optimal among all the stable natural merge-sort algorithms. Finally, we study length-adaptive ShiversSort, which is the special case of adaptive ShiversSort obtained by setting $c = n + 1$, as mentioned in Section 2, and we prove there that length-adaptive ShiversSort also matches the worst-case optimal merge cost of PowerSort.

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 4.1, which is a rephrasing of results from [3, 13, 15], adapted to the context merge costs.

This answer comes from the analysis of *merge trees*, which we describe below, and of the algorithm MinimalSort [3, 18], which is *not* a stable natural merge sort, and whose merge policy is described in Algorithm 3. 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 3: MinimalSort

```
Input: Array to A to sort
Result: The array A is sorted into a single run.
1 runs ← the run decomposition of A
2 while runs contains at least two runs:
3     merge the two shortest runs in runs
```

Given a merge policy and a sequence of runs, we define a binary rooted tree, called *merge tree*, as follows. Every node of the tree is identified with a run, either present in the initial sequence or created by the algorithm. The runs in the sequence are the leaves of the tree, and when two runs R_1 and R_2 are merged together in a run R', then 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.

Any such tree is the tree of a binary prefix encoding on a text on the alphabet $\{1, 2, \ldots, \rho\}$, 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.

Theorem 4.1. *The merge cost of any (stable or not) merge policy on a non-sorted array is minimised by the algorithm MinimalSort, and is at least* $\max\{n, n\mathcal{H}\}$.

Proof. The merge tree of the algorithm MinimalSort is a Huffman tree. This means that MinimalSort is indeed the 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 4.1 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.

We prove below that, unlike 1-adaptive ShiversSort or adaptive ShiversSort for any fixed value of c, the worst-case merge costs of PowerSort and of length-adaptive ShiversSort are optimal.

Proposition 4.2. *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 $n(\alpha\mathcal{H} + \beta)$. Then, we have $\alpha \geqslant 1$ and $\beta \geqslant 2$.*

Proof. First, Theorem 4.1 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, by considering arbitrarily large values of n, it follows that $\beta \geqslant 2$. □

Note that Proposition 4.2 indeed proves that the worst-case merge cost of PowerSort is optimal, since it matches the upper bound of $n(\alpha\mathcal{H} + \beta)$ with $\alpha = 1$ and $\beta = 2$. Addressing the optimality of the worst-case merge cost of adaptive ShiversSort requires considering another example.

Proposition 4.3. *Let c be a fixed parameter value, and 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. Then, we have $\beta \geqslant \Delta$.*

Proof. Let $k \geqslant 3$ some integer, and let $m = 2^k c$. Since c is an integer, we have $m \geqslant 8$. Then, consider the sequence of run lengths

$$\mathbf{r} = (2, 2m-10, 2, m+1, 2, 2m+2, 1).$$

For each sequence $s \in \mathcal{S}_{\mathbf{r}}$, we have $n = 5m = 5 \cdot 2^k c$ and

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

On the other hand, the merge cost of adaptive ShiversSort is equal to $\mathsf{mc} = 20m - 23 = n\,(4 + o(1))$. Since $\mathsf{mc} \leqslant n(\mathcal{H}+\beta)$, it follows that

$$\beta \geqslant 4 - (\log_2(5) - 4/5) = 24/5 - \log_2(5). \qquad \square$$

Hence, when c is fixed a priori, and in particular when we set $c = 1$, the worst-case merge cost of adaptive ShiversSort is *not* optimal. This is not very surprising since, unlike PowerSort, the algorithm adaptive ShiversSort cannot take into account the total length of the input until that end is indeed reached.

Indeed, optimal worst-case merge costs can be recovered by taking into account the length of the input, when setting the value of the parameter c. This is how we designed length-adaptive ShiversSort, which is nothing but the algorithm $(n+1)$-adaptive ShiversSort when sorting arrays of length n.

Proposition 4.4. *The merge cost of length-adaptive ShiversSort is bounded from above by $n(\mathcal{H}+2)$.*

Proof. Theorem 3.2 states that length-adaptive ShiversSort has a merge cost

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

Since $\{\log_2(n/(n+1))\} \geqslant 1 - 2/n$, it follows that $\mathsf{mc} \leqslant n(\mathcal{H}+2) - \rho + 1 \leqslant n(\mathcal{H}+2)$. $\square$

In general, one might expect PowerSort and length-adaptive ShiversSort to share more than just the above upper bound on their merge costs. In fact, the most natural interpretation of PowerSort relies on a top-down point of view on its merge tree. Indeed, in [15], the authors of PowerSort give the following intuition about those runs that lie at depth d in the tree: the positions they span should be the best approximation of intervals of the form $[kn/2^d, (k+1)n/2^d]$. By contrast, the construction of length-adaptive ShiversSort is entirely bottom-up, the idea being that those runs of length $n/2^d$ that happen to be created during an execution of the algorithm should lie at depth d.

Nevertheless, the subdivision of the array into runs spanning intervals of length $n/2^d$ is a crucial element of both algorithms. In particular, this makes PowerSort and length-adaptive ShiversSort scale-invariant algorithms: if the length of every run is multiplied by a constant factor κ, then their merge costs are also multiplied by κ.

This is not the case of other algorithms such as adaptive ShiversSort, as shown by the following example: when sorting an array with three runs of lengths $r_1 = 3$ and $r_2 = r_3 = 2$, the merge cost of adaptive ShiversSort is 12, but if these lengths are rescaled to $r_1 = 9$ and $r_2 = r_3 = 6$, then the merge cost is only of 33 instead of $3 \times 12 = 36$.

5 Approximately optimal sorting algorithms

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, on arrays with three runs of lengths $r_1 = r_2 = n-1$ and $r_3 = 2$ (with $n \geqslant 4$), the merge cost of PowerSort is $3n+1$ (which is optimal) and the cost adaptive ShiversSort is $4n-2$ (which is 33% worse). In the opposite direction, on arrays with four runs of length $r_1 = n-2$, $r_2 = r_3 = 2$ and $r_4 = n-2$ (with $n \geqslant 5$), the merge cost of adaptive ShiversSort is $3n+6$ (which is optimal) and that of PowerSort is $4n$ (which is also 33% worse).

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 $\mathbf{r} = (2, 2, \ldots, 2, k^2)$ whose 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 (besides the fact that $2^{\mathcal{H}} \leqslant \rho \leqslant n$), we aim for the uniform approximation result captured by the following definition.

Definition 5.1. *Let $\mathcal{A}$ be a stable natural merge sort, and let $\varepsilon \geqslant 0$ be a real number. We say that $\mathcal{A}$ is* asymptotically ε-optimal *if there exists an integer $\rho \geqslant 1$ such that, for every stable natural merge sort $\mathcal{B}$ and every array to be sorted and consisting of at least*

ρ monotonic runs, the respective merge costs $\mathbf{m}_a$ and $\mathbf{m}_b$ of $\mathcal{A}$ and $\mathcal{B}$ are such that $\mathbf{m}_a \leqslant (1+\varepsilon)\mathbf{m}_b$. If, furthermore, we can set $\rho = 1$, then we simply say that $\mathcal{A}$ is ε-optimal.

Below, we present two results. The first one is negative, and states that the algorithms TimSort and adaptive ShiversSort (or even PowerSort and length-adaptive ShiversSort) fall into a family of algorithms, called k-aware algorithms, that cannot contain any ε-optimal when ε is sufficiently small. The second one, on the other hand, is positive, and consists in building up, for every $\varepsilon > 0$, a variant of adaptive ShiversSort that is both ε-optimal and k-aware for some constant k.

First, we define more precisely 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 [4]. 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. 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 (k,ℓ)-awareness such as defined in [4].

Definition 5.2. *Let k and ℓ be elements of the set $\{0,1,2,\ldots\} \cup \{\infty\}$, 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 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.*

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 supported by Propositions 5.3 and 5.5, and the other direction is supported by Theorem 5.6.

Proposition 5.3. *Let $k \geqslant 2$ be an integer, and let*

$$\varepsilon_k = 1/(5(k+2)2^{k+2}).$$

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

Proof. Let $\rho = 2^k$, and let $n \geqslant 2$ be some positive integer. We design two arrays $\mathbf{a}^1$ and $\mathbf{a}^{-1}$ with $\rho + 2^n$ runs and with the same length, such that no length-(∞,k)-aware algorithm can approach the merge cost of an optimal stable merge algorithm by a factor $1+\varepsilon_k$ on both arrays $\mathbf{a}^1$ and $\mathbf{a}^{-1}$.

These arrays are constructed as follows. Let $r_1 = r_{\rho-1} = 3 \times 2^{2n+1}$, $r_i = 2^{2n+2}$ when $2 \leqslant i \leqslant \rho-2$, $r_\rho = (k+2)2^{2n+k+3}$, and $r_i = 2$ when $\rho+1 \leqslant i \leqslant \rho+2^n$. For $\theta = \pm 1$, let us denote by $\mathbf{r}^\theta$ the sequence

$$(r_1,\ldots,r_{\rho-2},r_{\rho-1}+\theta 2^{2n},r_\rho-\theta 2^{2n},r_{\rho+1},\ldots,r_{\rho+2^n}).$$

Then, $\mathbf{a}^\theta$ is an array of data that belongs to $\mathcal{S}_{\mathbf{r}^\theta}$, i.e., that splits into monotonic runs $R_1,\ldots,R_{\rho+2^n}$ of lengths $r_1,\ldots,r_{\rho-2},r_{\rho-1}+\theta 2^{2n},r_\rho-\theta 2^{2n},r_{\rho+1},\ldots,r_{\rho+2^n}$. Note that $\mathbf{a}^\theta$ has more than 2^n runs, and that its length does not depend on θ.

Let us split the array $\mathbf{a}^\theta$ into the following three *blocks*: the *left block* consists in the runs $R_1,\ldots,R_{\rho-1}$, the *middle block* consists of the run R_ρ, and the *right block* consists of the runs $R_{\rho+1},\ldots,R_{\rho+2^n}$.

Then, let $\mathcal{B}$ be the algorithm that operates on $\mathbf{a}^\theta$ as follows. First, $\mathcal{B}$ merges every pair of runs R_{2i} and $R_{2i-\theta}$, for $i \leqslant 2^{k-1}-1$. After that step has been carried, the number of runs in each block is a power of two: $\mathcal{B}$ sorts each block by acting like NaturalMergeSort, i.e., it keeps merging every pair of adjacent runs, thereby reducing by half the number of runs in the block, until every block contains only one run. Finally, $\mathcal{B}$ merges the middle and right block, then merges the resulting block with the left block.

Each element of the left (respectively, middle and right) block is merged at most $k+1$ times (respectively, twice and $n+2$ times). Hence, both merge costs of $\mathcal{B}$ on $\mathbf{a}^1$ and $\mathbf{a}^{-1}$ are smaller than $\mathbf{N}$, where

$$\mathbf{N} = 2r_\rho + (k+1)2^{2n+k+2} + (n+2)2^{n+1} + (k+1)2^{2n}.$$

Thus, using the fact that $x \leqslant 2^x$ for all integers $x \geqslant 0$, these costs are also smaller than $\mathbf{M}$, where

$$\mathbf{M} = 2r_\rho + (k+2)2^{2n+k+2} = 5r_\rho/2 = 2^{2n}/\varepsilon_k.$$

Then, let $\mathcal{A}$ be some stable merge sort algorithm. Let us assume that, denoting by $\mathbf{m}_a^\theta$ and by $\mathbf{m}_b^\theta$ the respective merge costs of $\mathcal{A}$ and $\mathcal{B}$ on the array $\mathbf{a}^\theta$, we have both $\mathbf{m}_a^1 \leqslant (1+\varepsilon_k)\mathbf{m}_b^1$ and $\mathbf{m}_a^{-1} \leqslant (1+\varepsilon_k)\mathbf{m}_b^{-1}$. We show below that, under this assumption, $\mathcal{A}$ cannot be length-(∞, k)-aware, thereby proving Proposition 5.3.

Let us first focus on the array $\mathbf{a}^1$. If, when sorting this array, the algorithm $\mathcal{A}$ includes the run R_ρ in three merge operations or more, its merge cost will be

$$\mathbf{m}_a^1 \geqslant 3r_\rho = 6\mathbf{M}/5 \geqslant 6\mathbf{m}_b^1/5 > (1+\varepsilon_k)\mathbf{m}_b^1.$$

Thus, $\mathcal{A}$ can merge R_ρ at most twice, and therefore $\mathcal{A}$ must first sort independently the left and right blocks. Note that $\mathcal{B}$ also starts by sorting these blocks, and the merges it performs are those of a Huffman tree. Hence, the merge cost of $\mathcal{B}$ on both blocks is optimal among all the (possibly non stable) merge sorts.

Then, $\mathcal{A}$ must merge R_ρ once with the entire left block, and once with the entire right block. Since the left block is longer than the right block, the total cost of these two merges is minimised if $\mathcal{A}$ merges R_ρ with the right block first, and then merges the resulting run with the left block. Since this is already what $\mathcal{B}$ does, it follows that $\mathcal{B}$ is in fact the stable algorithm whose merge cost, when sorting $\mathbf{a}^1$, is minimal.

Now, and since both $\mathcal{A}$ and $\mathcal{B}$ merge independently the left and right blocks, it makes sense to consider their merge costs on the left block only: we denote these costs by a and b. If $a > b$, and since both a and b are multiples of 2^{2n}, it comes that

$$\mathbf{m}_a^1 - \mathbf{m}_b^1 \geqslant a - b \geqslant 2^{2n} = \varepsilon_k \mathbf{M} > \varepsilon_k \mathbf{m}_b^1,$$

which means that $\mathbf{m}_a^1 > (1+\varepsilon_k)\mathbf{m}_b^1$.

Hence, we know that $a = b$. This means that the merge cost of $\mathcal{A}$ on the left block is optimal among all the (possibly non stable) merge sorts. Denoting by d_i the number of merges involving the run R_i while $\mathcal{A}$ sorts the left block, we have $a = \sum_i r_i d_i$. Since a is minimal, the rearrangement inequality states that larger runs R_i must be associated with smaller integers d_i: in other words, for any two runs R_i and R_j such that $r_i > r_j$, we must have $d_i \leqslant d_j$.

Then, let $\mathbf{D}$ be the largest of the integers d_i, and let $\mathbf{I}$ be the number of indices i such that $d_i = \mathbf{D}$. It comes that $\mathbf{D} \geqslant d_i \geqslant d_1 \geqslant d_{\rho-1}$ for all $i \in \{2, \ldots, \rho-2\}$. Then, if $d_{\rho-1} \neq \mathbf{D}$, let R_i and R_{i+1} two runs that $\mathcal{A}$ merges together, with $d_i = d_{i+1} = \mathbf{D}$. The resulting run is longer than $R_{\rho-1}$, and using the rearrangement inequality again proves that $d_{\rho-1} \geqslant \mathbf{D} - 1$.

Finally, Kraft inequality states that

$$1 = \sum_{i=1}^{\rho-1} 2^{-d_i} = 2^{-\mathbf{D}}\mathbf{I} + 2^{1-\mathbf{D}}(\rho - 1 - \mathbf{I}).$$

It comes that $2\rho = 2^{\mathbf{D}} + 2 + \mathbf{I}$, with $1 \leqslant \mathbf{I} \leqslant \rho - 1$, and thus that $2\rho > 2^{\mathbf{D}} \geqslant \rho - 1$. Since $\rho = 2^k$, it follows that $\mathbf{D} = k$, and then that $\mathbf{I} = \rho - 2$. This means that $d_1 = \ldots = d_{\rho-2} = k$ and that $d_{\rho-1} = k - 1$. Thus, since $\mathcal{A}$ can merge adjacent runs only, it must perform the same merges as $\mathcal{B}$ on the left block of $\mathbf{a}^1$, although these merges might be performed in a different order. Similarly, when sorting the array $\mathbf{a}^{-1}$, $\mathcal{A}$ must perform the same merges as $\mathcal{B}$ on the left block.

Let us now further assume that $\mathcal{A}$ is length-(∞, k)-aware. Then, $\mathcal{A}$ canot distinguish between $\mathbf{a}^1$ from $\mathbf{a}^{-1}$ before it discovers the last run of the left block. Moreover, note that no two adjacent runs R_i and R_{i+1} of the left block would be merged by $\mathcal{B}$ in both arrays $\mathbf{a}^1$ and $\mathbf{a}^{-1}$. Thus, $\mathcal{A}$ must wait until discovering the entire left block before it can perform a single merge on that block.

However, we have shown above that, when sorting the array $\mathbf{a}^1$, $\mathcal{A}$ must merge the run R_1 a total of $d_1 = k$ times. Thus, when R_1 is merged for the first time, each run R of the left block with which R_1 will be merged with must already be stored in the stack, possibly as a sequence of contiguous runs that will be merged into R itself before being merged with R_1. Since there are k such runs R, which must all lie above R_1 in the stack, the algorithm $\mathcal{A}$ 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 asymptotically ε-optimal for any ε.

Lemma 5.4. *Consider the following dynamic system. Starting with one empty stack $\mathcal{S}$, we successively perform operations of the following type: either (i) if $\mathcal{S}$ has at least two elements, we 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 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 (ii).

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 each integer k is becomes the top element after less than $f(k)$ operations of type (ii).

First, we show 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 5.5. *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 asymptotically ε-optimal for any real number ε.*

Proof. Let us assume in the entire proof, for the sake of contradiction, 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 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

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

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 it is executed on some array $\mathbf{a}_{k,\ell}$, the algorithm $\mathcal{A}$ cannot distinguish between the arrays $\mathbf{a}_{0,\ell}, \ldots, \mathbf{a}_{\ell,\ell}$ until it discovers their last run. 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 some run of length $2^{\ell-j}$ takes part in $7\mathbf{z}2^j$ merges, then of course these merges have a total cost of $7\mathbf{z}2^\ell$ at least. Hence, every run of length $2^{\ell-j}$ must take part in at most $f(j)$ merges, where $f(j) = 7\mathbf{z}2^j$. Furthermore, our stack follows exactly the dynamics described in Lemma 5.4, where choosing the option (ii) 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 run whose original length was $2^{\ell-j}$. It follows from Lemma 5.4 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.

Furthermore, since no algorithm can require a merge cost greater than ρn on arrays of length n and with ρ runs, every array falsifying the $(\mathbf{z}/2-1)$-optimality of $\mathcal{A}$ must have at least $\mathbf{z}/2-2$ runs. Hence, for every $\varepsilon > 0$ and every $\rho \geqslant 0$, by choosing $\mathbf{z} \geqslant \max\{\varepsilon, 2\rho+4\}$, one checks that $\mathcal{A}$ cannot be asymptotically ε-optimal either. ∎

Theorem 5.6. *Let $k \geqslant 8$ be an integer, and let*

$$\eta_k = 11/\log_2((k-3)/4).$$

There exists a k-aware sorting algorithm that is η_k-optimal.

Note that, if $k \geqslant 3$, then Theorems 3.1 and 4.1 already prove that 1-adaptive ShiversSort, which is a 3-aware (and thus a k-aware) algorithm, is also $(\Delta+1)$-optimal. In particular, numerical computations show that $\Delta + 1 \leqslant \eta_k$ when $k \leqslant 22$, which already proves Theorem 5.6 in that case.

Furthermore, and although both constants ε_k and η_k become arbitrarily small when $k \to \infty$, there is a double exponential gap between those constants. This mismatch might not be too surprising, given that Proposition 5.3 focuses on all length-(∞, k)-aware algorithms, while Theorem 5.6 deals with the much more restricted class of k-aware algorithms.

A complete proof of Theorem 5.6 is available in [11]. However, this proof is rather long and technical, and is based on tools that would require changing the notations used in previous sections. Thus, in the present paper, we only sketch the main ideas of the proof of Theorem 5.6, sometimes sacrificing slightly the accuracy of the presentation for its clarity and succinctness.

The proof relies on designing explicitly, for every integer $\kappa \geqslant 3$, an algorithm that is both $(2\kappa+2)$-aware and $\eta_{2\kappa+3}$-optimal. This algorithm is presented in Algorithm 4. It is a parameterised variant of 1-adaptive ShiversSort (i.e., of adaptive ShiversSort for the parameter $c = 1$) whose parameter is the integer κ, and therefore this algorithm is called κ-ShiversSort.

Algorithm 4: κ-ShiversSort

```
Input: Array $A$ to sort, integer parameter $\kappa$
Result: The array $A$ is sorted into a single run.
        That run remains on the stack.
Note: We denote the height of the stack $\mathcal{S}$ by $h$,
      and its $i^{\text{th}}$ top-most run (for $1 \leqslant i \leqslant h$) by
      $R_i$. The length of $R_i$ is denoted by $r_i$, and
      we set $\ell_i = \lfloor \log_2(r_i) \rfloor$.

runs ← the run decomposition of $A$                ▷ phase 1
$\mathcal{S}$ ← an empty stack
while true:
    $b \leftarrow$ true
    while $h \geqslant 2\kappa + 2$ and $b$:            ▷ high-stack mode
        $b \leftarrow$ TestMerge$(1, \infty)$
    if $h = 2\kappa + 1$: LowStack                 ▷ low-stack mode
    if runs $\neq \emptyset$:
        remove a run $R$ from runs and push $R$ onto $\mathcal{S}$
    else break
while $h \geqslant 2\kappa + 2$:                      ▷ high-stack mode
    merge the runs $R_{h-1}$ and $R_h$
apply the optimal merge policy on the remaining $h$
  runs (with $h \leqslant 2\kappa + 1$)                     ▷ phase 2

Function LowStack:              ▷ called with $h = 2\kappa + 1$
    $r_{\mathsf{large}} \leftarrow \lfloor (r_2 + r_3 + \ldots + r_h)/\kappa \rfloor$
    $\ell_{\mathsf{large}} \leftarrow \lfloor \log_2(r_{\mathsf{large}}) \rfloor$
    $i \leftarrow h - 2$
    $b \leftarrow$ true
    while $i \geqslant 1$ and $b$:
        $b \leftarrow \neg$ TestMerge$(i, \ell_{\mathsf{large}})$
        $i \leftarrow i - 1$

Function TestMerge$(i, \ell_\star)$:   ▷ called with $i \leqslant h - 2$
    if $\max\{\ell_i, \ell_{i+1}\} \geqslant \ell_{i+2}$ and $\ell_\star \geqslant \max\{\ell_{i+1}, \ell_{i+2}\}$:
        merge the runs $R_{i+1}$ and $R_{i+2}$
        return true
    else return false
```

Consider some array $\mathbf{a}$ of length n and entropy $\mathcal{H}$, which is to be sorted. First, since every algorithm sorting $\mathbf{a}$ has a merge cost $m \geqslant n\mathcal{H}$, and the overhead of adaptive ShiversSort is at most Δn (i.e., the merge cost of adaptive ShiversSort is at most $m + \Delta n$), this overhead is at most $\Delta/\mathcal{H}$ times the optimal merge cost. Yet, if $\mathcal{H}$ is low, the overhead of adaptive ShiversSort will be too large for our needs.

Thus, a naive idea is to split the dynamics of κ-ShiversSort in two phases. In a first phase (which spans the lines 1 to 12 in Algorithm 4), we would perform only some of the merges prescribed by adaptive ShiversSort, incurring only a fraction of the overhead of adaptive ShiversSort, and thereby obtaining a partially sorted array $\mathbf{a}'$; in a second phase, we would then apply brutally an optimal merge policy on $\mathbf{a}'$. Surprisingly, and provided we choose carefully which merges are performed in the first phase, this idea is in fact sufficient to prove Theorem 5.6.

In particular, dividing the dynamics of our new algorithm κ-ShiversSort in such a way, the overhead of this algorithm is naturally split into two parts: (i) one part due to the fact that, in order to transform $\mathbf{a}$ into $\mathbf{a}'$, we used merges prescribed by adaptive ShiversSort instead of a merge policy with the lowest possible merge cost; (ii) the other part is the due to the fact that the array $\mathbf{a}'$ itself might be suboptimal, and quite different from partially sorted arrays obtained while applying any optimal merge policy on the initial array $\mathbf{a}$.

Part (i) of this overhead can be taken care of with arguments presented in Section 3. Indeed, we can roughly identify the overhead of adaptive ShiversSort with the non-expanding cost of those merges it performs. Thus, it follows from Lemma 3.10 that, if pushing a run R results in merging runs $R_1, \ldots, R_k$, the overhead incurred by these merges is dominated by $\mathcal{O}(\min\{r, r_1 + \ldots, r_k\})$. In particular, if only n^* entries of $\mathbf{a}$ belong to some run that is merged during the first phase of κ-ShiversSort, we can prove that part (i) is dominated by $\mathcal{O}(n^*)$.

The arguments used for dealing with part (ii) of the overhead are quite different, as they rely on structural properties of the lengths of the runs in $\mathbf{a}'$. Roughly said, those merges performed in phase 1 of κ-ShiversSort are those merges prescribed by adaptive ShiversSort and that did not involve any run of size $r \geqslant n/\kappa$. Managing to do this is a difficult task, on which we will come back later.

However, once this is done, and due to properties of adaptive ShiversSort, it turns out that $\mathbf{a}'$ cannot contain any two consecutive runs of length smaller than n/κ. At the same time, no merge can result in a run of length more than $2n/\kappa$. Consequently, we split the runs of $\mathbf{a}'$ into three families: *very large* runs, with a length $r \geqslant 2n/\kappa$, and which must be original runs of the array $\mathbf{a}$; *large* runs, with a length r such that $n/\kappa \leqslant r < 2n/\kappa$; and *small* runs, with a length $r < n/\kappa$, which cannot be adjacent with another small run.

A first, immediate consequence is that, since $\mathbf{a}'$ cannot contain more than κ large or very large runs, it cannot contain more than $2\kappa + 1$ runs. That is why it is then legitimate to sort $\mathbf{a}'$ by using an optimal merge policy, which can be implemented by using a (k, k)-aware algorithm for $k = 2\kappa + 1$.

Another consequence is that, denoting by $n^\bullet$ the total length of those runs that are either small or large (but not very large), we must have $n^* \leqslant n^\bullet$. Then, studying the dynamics of any optimal merge policy for sorting the array $\mathbf{a}$, we can show that such a policy might also have involved constructing, as an intermediate step, a partially sorted array $\mathbf{a}''$ with similar structural properties as $\mathbf{a}'$; the only difference being that, while $\mathbf{a}'$ could not contain two consecutive small runs, the array $\mathbf{a}''$ might contain up to 7 consecutive small runs. As a result, it can be shown that part (ii) of the overhead of κ-ShiversSort is also dominated by $\mathcal{O}(n^\bullet)$.

Thus, the overhead of κ-ShiversSort itself is dominated by $\mathbf{K}n^\bullet$ for some constant $\mathbf{K}$. Moreover, since each of the $n^\bullet$ entries of $\mathbf{a}'$ belonging to a small or a large run already belonged to a small or large run of $\mathbf{a}$, we know that $\mathbf{a}$ had an entropy $\mathcal{H} \geqslant n^\bullet \log_2(\kappa/2)/n$. Consequently, and recalling that any sorting algorithm has a merge cost $m \geqslant n\mathcal{H}$ when sorting $\mathbf{a}$, it follows that $n^\bullet \leqslant m/\log_2(\kappa/2)$, thereby proving that κ-ShiversSort is $(\mathbf{K}/\log_2(\kappa/2))$-optimal in the sense of Definition 5.1.

Finally, let us come back to the unproven assertion that those merges performed in phase 1 of κ-ShiversSort are merges that would be prescribed by adaptive ShiversSort and that involve only small runs. An obvious hurdle is that, until all the runs of the array have been discovered, the value of n remains unknown, and therefore we cannot know which runs are small. Hence, and since we cannot afford to merge any run that might be (very) large, a natural idea is that κ-ShiversSort should maintain an estimation, let us say $\hat{n}$, of the length n of the array that is being sorted. Then, we must postpone merging any run of length $r \geqslant \hat{n}/\kappa$ until new runs are uncovered, thereby increasing the value of $\hat{n}$, or until the end of phase 1.

A difficulty in doing so is that (k,k)-aware algorithms are not allowed to maintain any counter that would come in addition to their stacks of runs. Therefore, the only times at which we can evaluate $\hat{n}$ are those times where the stack contains no more than k runs. Hence, there are two obvious things to do whenever possible: (i) if the stack is of height $h \leqslant k-1$, just keep discovering and pushing runs onto it, since it will always be possible to perform those merges that are due according to the policy of adaptive ShiversSort; (ii) if the stack is of height $h = k$, we evaluate $\hat{n}$ as the sum of the lengths of the runs in the stack; then, we check whether some merge is due, and if yes, we do it. These two types of actions constitute the *low-stack mode* of line 7 of the algorithm.

However, if there is still no merge due except those that might involve (very) large runs, we must push an additional run, which will prevent evaluating $\hat{n}$ until the stack's height decreases again: we enter the *high-stack mode* of lines 5 and 11. In that mode, if adaptive ShiversSort ever dictates to merge two runs, we face a dilemma: should we obey and merge them, or should we wait, by fear one of the runs be large? Fortunately, and using structural properties of the sequence of runs maintained in the stack (which are, roughly, a blend between the inequality (1) and the fact that, among the lowest k runs, no two consecutive runs can be of length $r < \hat{n}/\kappa$), we can prove that both runs we wish to merge must be small. This solves the dilemma, as it suffices now to perform the merge prescribed by adaptive ShiversSort.

Finally, it remains to explain how we detect which merges are prescribed by adaptive ShiversSort. This is obvious in the high-stack mode, but much less so in the low-stack mode. Indeed, in the latter case, it may occur that we had once chosen to postpone merging two runs R and R' because one of them might have been large, but that our estimate $\hat{n}$ has increased since then, ensuring that neither R nor R' was large. The strategy used in Algorithm 4 for solving that problem is simple: we start looking at the stack from bottom to top, as if we were discovering its runs one by one during an execution of adaptive ShiversSort, and merging them if adaptive ShiversSort dictates so and if both runs involved are definitely small runs.

It turns out that, due to subtle properties of the dynamics of adaptive ShiversSort, which we will omit here, this strategy is a correct one. In particular, this means that the merges performed in the phase 1 of κ-ShiversSort are indeed merges that would have been performed by adaptive ShiversSort, although, in general, they will not be performed in the same order.

References

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. *On the worst-case complexity of Timsort.* 26th Annual European Symposium on Algorithms (ESA 2018), LIPIcs vol. 112, pages 4:1–13, 2018. Full version available at: `https://arxiv.org/abs/1805.08612`.

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

[3] Jérémy Barbay and Gonzalo Navarro. *On compressing permutations and adaptive sorting.* Theor. Comput. Sci., pages 513:109–123, 2013.

[4] Sam Buss and Alexander Knop. *Strategies for stable merge sorting.* 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 1272–1290. SIAM, 2019.

[5] 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.* 27th International Conference on Computer Aided Verification (CAV), pages 273–289. Springer, 2015.

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

[7] Vladmir Estivill-Castro and and Derick Wood. *A survey of adaptive sorting algorithms.* ACM Computing Surveys, vol. 24, issue 4, pages 441-476, 1992.

[8] Herman Goldstine and John von Neumann. *Planning and coding of problems for an electronic computing instrument.* 1947.

[9] Mordecai Golin and Robert Sedgewick. *Queue-mergesort.* Information Processing Letters, vol. 48, issue 5, pages 253–259, 1993.

[10] Tony Hoare. *Algorithm 64: Quicksort.* Communications of the ACM, vol. 4, issue 7, page 321, 1961.

[11] Vincent Jugé. *Adaptive Shivers sort: an alternative sorting algorithm.* 31th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). SIAM, 2019. Full version available at: `https://arxiv.org/abs/1809.08411`.

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

[13] Heikki Mannila. *Measures of presortedness and optimal sorting algorithms.* IEEE Transactions on Computers, vol. 34, issue 4, pages 318–325, 1985.

[14] Alistair Moffat, Gary Eddy, and Ola Petersson. *Splaysort: Fast, versatile, practical.* Software: Practice and Experience, vol. 26, issue 7, pages 781–797, 1996.

[15] J. Ian Munro and Sebastian Wild. *Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs.* 26th Annual European Symposium on Algorithms (ESA 2018), LIPIcs vol. 112, pages 63:1–15, 2018.

[16] Tim Peters. *Timsort description,* accessed june 2015. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`.

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

[18] Tadao Takaoka. *Partial solution and entropy.* 34th International Symposium on Mathematical Foundations of Computer Science (MFCS), pages 700–711. Springer Berlin Heidelberg, 2009.

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