Strategies for Stable Merge Sorting

Sam Buss* Alexander Knop*

Abstract

We introduce new stable natural merge sort algorithms, called 2-merge sort and α-merge sort. We prove upper and lower bounds for several merge sort algorithms, including Timsort, Shiver's sort, α-stack sorts, and our new 2-merge and α-merge sorts. The upper and lower bounds have the forms $c \cdot n \log m$ and $c \cdot n \log n$ for inputs of length n comprising m runs. For Timsort, we prove a lower bound of $(1.5-o(1))n \log n$. For 2-merge sort, we prove optimal upper and lower bounds of approximately $(1.089 \pm o(1))n \log m$. We state similar asymptotically matching upper and lower bounds for α-merge sort, when $\varphi < \alpha < 2$, where φ is the golden ratio.

Our bounds are in terms of merge cost; this upper bounds the number of comparisons and accurately models runtime. The merge strategies can be used for any stable merge sort, not just natural merge sorts. The new 2-merge and α-merge sorts have better worst-case merge cost upper bounds and are slightly simpler to implement than the widely-used Timsort; they also perform better in experiments.

1 Introduction

This paper studies stable merge sort algorithms, especially natural merge sorts. We will propose new strategies for the order in which merges are performed, and prove upper and lower bounds on the cost of several merge strategies. The first merge sort algorithm was proposed by von Neumann [16, p.159]: it works by splitting the input list into sorted sublists, initially possibly lists of length one, and then iteratively merging pairs of sorted lists, until the entire input is sorted. A sorting algorithm is *stable* if it preserves the relative order of elements which are not distinguished by the sort order. There are several methods of splitting the input into sorted sublists before starting the merging; a merge sort is called *natural* it finds the sorted sublists by detecting consecutive runs of entries in the input which are already in sorted order. Natural merge sorts were first proposed by Knuth [16, p.160].

Like most sorting algorithms, the merge sort is comparison-based in that it works by comparing the relative order of pairs of entries in the input list. Information-theoretic considerations imply that any comparison-based sorting algorithm must make at least $\log_2(n!) \approx n \log_2 n$ comparisons in the worst case. However, in many practical applications, the input is frequently already partially sorted. There are many *adaptive* sort algorithms which will detect this and run faster on inputs which are already partially sorted. Natural merge sorts are adaptive in this sense: they detect sorted sublists (called "runs") in the input, and thereby reduce the cost of merging sublists. One very popular stable natural merge sort is the eponymous Timsort of Tim Peters [25]. Timsort is extensively used, as it is included in Python, in the Java standard library, in GNU Octave, and in the Android operating system. Timsort has worst-case runtime $O(n \log n)$, but is designed to run substantially faster on inputs which are partially pre-sorted by using intelligent strategies to determine the order in which merges are performed.

There is extensive literature on adaptive sorts: e.g., for theoretical foundations see [21, 9, 26, 23] and for more applied investigations see [6, 12, 5, 31, 27]. The present paper will consider only stable, natural merge sorts. As exemplified by the wide deployment of Timsort, these are certainly an important class of adaptive sorts. We will consider the Timsort algorithm [25, 7], and related sorts due to Shivers [28] and Auger-Nicaud-Pivoteau [2]. We will also introduce new algorithms, the "2-merge sort" and the "α-merge sort" for $\varphi < \alpha < 2$ where φ is the golden ratio.

The main contribution of the present paper is the definition of new stable merge sort algorithms, called 2-merge and α-merge sort. These have better worst-case run times than Timsort, are slightly easier to implement than Timsort, and perform better in our experiments.

We focus on natural merge sorts, since they are so widely used. However, our central contribution is analyzing merge strategies and our results are applicable to any stable sorting algorithm that generates and merges runs, including patience sorting [5], melsort [29, 20], and split sorting [19].

All the merge sorts we consider will use the following framework. (See Algorithm 1.) The input is a list of n elements, which without loss of generality may be assumed to be integers. The first logical stage of the al-

*Department of Mathematics, University of California, San Diego, La Jolla, CA, USA

gorithm (following [25]) identifies maximal length subsequences of consecutive entries which are in sorted order, either ascending or descending. The descending subsequences are reversed, and this partitions the input into "runs" $R_1, \ldots, R_m$ of entries sorted in non-decreasing order. The number of runs is m; the number of elements in a run R is $|R|$. Thus $\sum_i |R_i| = n$. It is easy to see that these runs may be formed in linear time and with linear number of comparisons.

Merge sort algorithms process the runs in left-to-right order starting with R_1. This permits runs to be identified on-the-fly, only when needed. This means there is no need to allocate $\Theta(m)$ additional memory to store the runs. This also may help reduce cache misses. On the other hand, it means that the value m is not known until the final run is formed; thus, the natural sort algorithms do not use m except as a stopping condition.

The runs R_i are called *original runs*. The second logical stage of the natural merge sort algorithms repeatedly merges runs in pairs to give longer and longer runs. (As already alluded to, the first and second logical stages are interleaved in practice.) Two runs A and B can be merged in linear time; indeed with only $|A| + |B| - 1$ many comparisons and $|A| + |B|$ many movements of elements. The merge sort stops when all original runs have been identified and merged into a single run.

Our mathematical model for the run time of a merge sort algorithm is the sum, over all merges of pairs of runs A and B, of $|A| + |B|$. We call the quantity the *merge cost*. In most situations, the run time of a natural sort algorithm can be linearly bounded in terms of its merge cost. Our main theorems are lower and upper bounds on the merge cost of several stable natural merge sorts. Note that if the runs are merged in a balanced fashion, using a binary tree of height $\lceil \log m \rceil$, then the total merge cost $\leq n\lceil \log m \rceil$. (We use log to denote logarithms base 2.) Using a balanced binary tree of merges gives a good worst-case merge cost, but it does not take into account savings that are available when runs have different lengths.[1] The goal is find *adaptive* stable natural merge sorts which can effectively take advantage of different run lengths to reduce the merge cost, but which are guaranteed to never be much worse than the binary tree. Therefore, our preferred upper bounds on merge costs are stated in the form $c \cdot n \log m$ for some constant c, rather than in the form $c \cdot n \log n$.

The merge cost ignores the $O(n)$ cost of forming the original runs R_i: this does not affect the asymptotic limit of the constant c.

Algorithm	Awareness
Timsort (original) [25]	3-aware
Timsort (corrected) [7]	(4,3)-aware
α-stack sort [2]	2-aware
Shivers sort [28]	2-aware
2-merge sort	3-aware
α-merge sort ($\varphi<\alpha<2$)	3-aware

Table 1: Awareness levels for merge sort algorithms

Algorithm 1 shows the framework for all the merge sort algorithms we discuss. This is similar to what Auger et al. [2] call the "generic" algorithm. The input S is a sequence of integers which is partitioned into monotone runs of consecutive members. The decreasing runs are inverted, so S is expressed as a list $\mathcal{R}$ of increasing original runs $R_1, \ldots, R_m$ called "original runs". The algorithm maintains a stack $\mathcal{Q}$ of runs $Q_1, \ldots, Q_\ell$, which have been formed from $R_1, \ldots, R_k$. Each time through the loop, it either pushes the next original run, R_{k+1}, onto the stack $\mathcal{Q}$, or it chooses a pair of adjacent runs Q_i and Q_{i+1} on the stack and merges them. The resulting run replaces Q_i and Q_{i+1} and becomes the new Q_i, and the length of the stack decreases by one. The entries of the runs Q_i are stored in-place, overwriting the elements of the array which held the input S. Therefore, the stack needs to hold only the positions p_i (for $i = 0, \ldots, \ell + 1$) in the input array where the runs Q_i start, and thereby implicitly the lengths $|Q_i|$ of the runs. We have $p_1 = 0$, pointing to the beginning of the input array, and for each i, we have $|Q_i| = p_{i+1} - p_i$. The unprocessed part of S in the input array starts at position $p_{\ell+1} = p_\ell + |Q_\ell|$. If $p_{\ell+1} < n$, then it will be the starting position of the next original run pushed onto $\mathcal{Q}$.

Algorithm 1 is called (k_1, k_2)-*aware* since its choice of what to do is based on just the lengths of the runs in the top k_1 members of the stack $\mathcal{Q}$, and since merges are only applied to runs in the top k_2 members of $\mathcal{Q}$. ([2] used the terminology "degree" instead of "aware".) In all our applications, k_1 and k_2 are small numbers, so it is appropriate to store the runs in a stack. Usually $k_1 = k_2$, and we write "k-aware" instead of "(k, k)-aware". Table 1 shows the awareness values for the algorithms considered in this paper. To improve readability (and following [2]), we use the letters W, X, Y, Z to denote the top four runs on the stack, $Q_{\ell-3}, Q_{\ell-2}, Q_{\ell-1}, Q_\ell$ respectively, (if they exist).[2]

In all the sorting algorithms we consider, the height

[1] [11] gives a different method of achieving merge cost $O(n \log m)$. Like the binary tree method, their method is not adaptive.

[2] [25] used "A, B, C" for "X, Y, Z".

Algorithm 1 The basic framework for our merge sort algorithms.
S is a sequence of integers of length n.
$\mathcal{R}$ is the list of m runs formed from S.
$\mathcal{Q}$ is a stack of ℓ runs, $Q_1, Q_2, \ldots, Q_\ell$.
The top member of the stack $\mathcal{Q}$ is Q_ℓ.
k_1 and k_2 are fixed (small) integers.
The algorithm is (k_1, k_2)-aware.
Upon termination, $\mathcal{Q}$ contains a single run Q_1 which is the sorted version of S.

```
1: procedure MergeSortFramework(S, n)
2:     R ← list of runs forming S
3:     Q ← empty stack
4:     while R ≠ ∅ or Q has > 1 member do
5:         choose to do either (A) or (B_i) for some
6:             ℓ − k_2 < i < ℓ, based on whether R is
7:             empty and on the values |Q_j| for
8:             ℓ − k_1 < j ≤ ℓ
9:             (A) Remove the next run R from R and
10:                push it onto Q.
11:            This increments ℓ = |Q| by 1.
12:            (B_i) Replace Q_i and Q_{i+1} in Q
13:                with Merge(Q_i, Q_{i+1}).
14:            This decrements ℓ = |Q| by 1.
15:        end choices
16:    end while
17:    Return Q_1 as the sorted version of S.
18: end procedure
```

of the stack $\mathcal{Q}$ will be small, namely $\ell = O(\log n)$. Since the stack needs only store the $\ell + 1$ values $p_1, \ldots, p_{\ell+1}$, the memory requirements for the stack are minimal. Another advantage of Algorithm 1 is that runs may be identified on the fly and that merges occur only near the top of the stack: this may help reduce cache misses. (See [18] for other methods for reducing cache misses.)

Merge(A, B) is the run obtained by stably merging A and B. Since it takes only linear time to extract the runs $\mathcal{R}$ from S, the computation time of Algorithm 1 is dominated by the time needed for merging runs. We shall use $|A| + |B|$ as the mathematical model for the runtime of a (stable) merge. The usual algorithm for merging A and B uses an auxiliary buffer to hold the smaller of A and B, and then merging directly into the combined buffer. Timsort [25] uses a variety of techniques to speed up merges, in particular "galloping"; this also still takes time proportional to $|A| + |B|$ in general. It is possible to perform (stable) merges in-place with no additional memory [17, 22, 13, 30, 10]; these algorithms also require time $\Theta(|A| + |B|)$.

More complicated data structures can perform merges in sublinear time in some situations; see for instance [4, 11]. These methods do not seem to be useful in practical applications, and of course still have worst-case run time $\Theta(|A| + |B|)$.

If A and B have very unequal lengths, there are (stable) merge algorithms which use fewer than $\Theta(|A|+|B|)$ comparisons [14, 30, 10]. Namely, if $|A| \le |B|$, then it is possible to merge A and B in time $O(|A|+|B|)$ but using only $O(|A|(1+\log(|B|/|A|)))$ comparisons. Nonetheless, we feel that the cost $|A| + |B|$ is the best way to model the runtime of merge sort algorithms. Indeed, the main strategies for speeding up merge sort algorithms try to merge runs of approximate equal length as much as possible; thus $|A|$ and $|B|$ are very unequal only in special cases. Of course, all our upper bounds on merge cost are also upper bounds on number of comparisons.

Definition 1.1. *The* merge cost *of a merge sort algorithm on an input S is the sum of $|A| + |B|$ taken over all merge operations Merge(A, B) performed. For Q_i a run on the stack $\mathcal{Q}$ during the computation, the* merge cost *of Q_i is the sum of $|A| + |B|$ taken over all merges Merge(A, B) used to combine runs that form Q_i.*

Our definition of merge cost is motivated primarily by analyzing the run time of *sequential* algorithms. Nonetheless, the constructions may be applicable to distributed sorting algorithms, such as in MapReduce [8]. A distributed sorting algorithm typically performs sorts on subsets of the input on multiple processors: each of these can take advantage of a faster sequential merge sort algorithm. Applications to distributed sorting are

beyond the scope of the present paper however.

In later sections, the notation w_{Q_i} is used to denote the merge cost of the i-th entry Q_i on the stack at a given time. Here "w" stands for "weight". The notation $|Q_i|$ denotes the length of the run Q_i. We use m_{Q_i} to denote the number of original runs which were merged to form Q_i.

All of the optimizations used by Timsort mentioned above can be used equally well with any of the merge sort algorithms discussed in the present paper. In addition, they can be used with other sorting algorithms that generate and merge runs. These other algorithms include patience sort [5], melsort [29] (which is an extension of patience sort), the hybrid quicksort-melsort algorithm of [20], and split sort [19]. The merge cost as defined above applies equally well to all these algorithms. Thus, it gives a runtime measurement which applies to a broad range of sort algorithms that incorporate merges and which is largely independent of which optimizations are used.

Algorithm 1, like all the algorithms we discuss, only merges adjacent elements, Q_i and Q_{i+1}, on the stack. This is necessary for the sort to be stable: If $i < i'' < i'$ and two non-adjacent runs Q_i and $Q_{i'}$ were merged, then we would not know how to order members occurring in both $Q_{i''}$ and $Q_i \cup Q_{i'}$. The patience sort, melsort, and split sort can all readily be modified to be stable, and our results on merge costs can be applied to them.

Our merge strategies do not apply to non-stable merging, but Barbay and Navarro [3] have given an optimal method—based on Huffmann codes—of merging for non-stable merge sorts in which merged runs do not need to be adjacent.

The known worst-case upper and lower bounds on stable natural merge sorts are listed in Table 2. The table expresses bounds in the strongest forms known. Since $m \leq n$, it is generally preferable to have upper bounds in terms of $n \log m$, and lower bounds in terms of $n \log n$.

The main results of the paper are those listed in the final two lines of Table 2. Theorem 5.4 proves that the merge cost of 2-merge sort is at most $(d_2 + c_2 \log m) \cdot n$, where $d_2 \approx 1.911$ and $c_2 \approx 1.089$: these are very tight bounds, and the value for c_2 is optimal by Theorem 5.2. It is also substantially better than the worst-case merge cost for Timsort proved in Theorem 2.2. Similarly for $\varphi < \alpha < 2$, Theorem 5.4 states an upper bound of $(d_\alpha + c_\alpha \log m) \cdot n$. The values for c_α are optimal by Theorem 5.1; however, our values for d_α have unbounded limit $\lim_{\alpha \to \varphi^+} d_\alpha$ and we conjecture this is not optimal.

We only analyze α-merge sorts with $\alpha > \varphi$. For $\varphi < \alpha \leq 2$, the α-merge sorts improve on Timsort, by virtue of having better run time bounds and by being slightly easier to implement. In addition, they perform better in the experiments reported in Section 6. It is an open problem to extend our algorithms to the case of $\alpha < \varphi$; we expect this will require k-aware algorithms with $k > 3$.

The outline of the paper is as follows. Section 2 describes Timsort, and proves the lower bound on its merge cost. Section 3 discusses the α-stack sort algorithms, and gives lower bounds on their merge cost. Section 4 describes the Shivers sort, and gives a simplified proof of the $n \log n$ upper bound of [28]. Section 5 is the core of the paper and describes the new 2-merge sort and α-merge sort. We first prove the lower bounds on their merge cost, and finally prove the corresponding upper bounds for 2-merge sort. For space reasons, the corresponding upper bound for α-merge sort is omitted here, but will be presented in the full version of the paper. Section 6 gives some experimental results on various kinds of randomly generated data. All these sections can be read independently of each other. The paper concludes with discussion of open problems.

We thank the referees for useful comments and suggestions.

2 Timsort lower bound

Algorithm 2 is the Timsort algorithm as defined by [7] improving on [25]. Recall that W, X, Y, Z are the top four elements on the stack $\mathcal{Q}$. A command "Merge Y and Z" creates a single run which replaces both Y and Z in the stack; at the same time, the current third member on the stack, X, becomes the new second member on the stack and is now designated Y. Similarly, the current W becomes the new X, etc. Likewise, the command "Merge X and Y" merges the second and third elements at the top of $\mathcal{Q}$; those two elements are removed from $\mathcal{Q}$ and replaced by the result of the merge.

Timsort was designed so that the stack has size $O(\log n)$, and the total running time is $O(n \log n)$. These bounds were first proved by [2]; simplified proofs were given by [1] who also strengthed the upper bound to $O(n \log m)$.

THEOREM 2.1. ([1, 2]) *The merge cost of Timsort is $O(n \log n)$, and even $O(n \log m)$.*

The proof in Auger et al. [2] did not compute the constant implicit in their proof of the upper bound of Theorem 2.1; but it is approximately equal to $3/\log \varphi \approx 4.321$. The proofs in [1] also do not quantify the con-

[3] When the first draft of the present paper was circulated, the question of an $O(n \log m)$ upper bound for Timsort was still open; this was subsequently resolved by [1].

Algorithm	Upper bound	Lower bound
Timsort[3]	$\left\{ \begin{array}{c} O(n \log n) \text{ [1, 2]} \\ O(n \log m) \text{ [1]} \end{array} \right\}$	$1.5 \cdot n \log n$ [Theorem 2.2]
α-stack sort	$O(n \log n)$ [2]	$\left\{ \begin{array}{ll} c_\alpha \cdot n \log n & \text{[Theorem 3.2]} \\ \omega(n \log m) & \text{[Theorem 3.3]} \end{array} \right.$
Shivers sort	$\left\{ \begin{array}{c} n \log n \text{ [28]} \\ \text{See also Theorem 4.2} \end{array} \right\}$	$\omega(n \log m)$ [Theorem 4.1]
2-merge sort	$c_2 \cdot n \log m$ [Theorem 5.2]	$c_2 \cdot n \log m$ [Theorem 5.4]
α-merge sort	$c_\alpha \cdot n \log n$ [Theorem 5.1]	$c_\alpha \cdot n \log n$ [Theorem 5.3]

Table 2: Upper and lower bounds on the merge cost of various algorithms. For more precise statements, see the theorems. The results hold for $\varphi < \alpha \leq 2$; for these values, c_α is defined by equation (5.2) and satisfies $1.042 < c_\alpha < 1.089$. In particular, $c_2 = 3/\log(27/4) \approx 1.08897$. All bounds are asymptotic; that is, they are correct up to a multiplicative factor of $1 \pm o(1)$. For this reason, the upper and lower bounds in the last two lines of the the table are not exactly matching. The table lists 2-merge sort and α-merge sort on separate lines since the 2-merge sort is slightly simpler than the α-merge sort. In addition, our proof of the upper bound for the 2-merge sort algorithm is substantially simpler than our proof for the α-merge sort.

Algorithm 2 The Timsort algorithm. W, X, Y, Z denote the top four elements of the stack $\mathcal{Q}$. A test involving a stack member that does not exist evaluates as "False". For example, $|X| < |Z|$ evaluates as false when $|\mathcal{Q}| < 3$ and X does not exist.

```
1: procedure TimSort(S, n)
2:     R ← run decomposition of S
3:     Q ← ∅
4:     while R ≠ ∅ do
5:         Remove the next run R from R
6:         and push it onto Q
7:         loop
8:             if |X| < |Z| then
9:                 Merge X and Y
10:            else if |X| ≤ |Y| + |Z| then
11:                Merge Y and Z
12:            else if |W| ≤ |X| + |Y| then
13:                Merge Y and Z
14:            else if |Y| ≤ |Z| then
15:                Merge Y and Z
16:            else
17:                Break out of the loop
18:            end if
19:        end loop
20:    end while
21:    while |Q| ≥ 1 do
22:        Merge Y and Z
23:    end while
24: end procedure
```

stants in the big-O notation, but they are comparable or slightly larger. We prove a corresponding lower bound.

THEOREM 2.2. *The worst-case merge cost of the Timsort algorithm on inputs of length n which decompose into m original runs is $\geq (1.5 - o(1)) \cdot n \log n$. Hence it is also $\geq (1.5 - o(1)) \cdot n \log m$.*

In other words, for any $c < 1.5$, there are inputs to Timsort with arbitrarily large values for n (and m) so that Timsort has merge cost $> c \cdot n \log n$. We conjecture that the results of [1] can be improved to show that Theorem 2.1 is nearly optimal:

CONJECTURE 1. *The merge cost of Timsort is bounded by $(1.5 + o(1)) \cdot n \log m$.*

Proof of Theorem 2.2. We must define inputs that cause Timsort to take time close to $1.5n \log n$. As always, $n \geq 1$ is the length of the input S to be sorted. We define $\mathcal{R}_{\text{tim}}(n)$ to be a sequence of run *lengths* so that $\mathcal{R}_{\text{tim}}(n)$ equals $\langle n_1, n_2 \ldots, n_m \rangle$ where each $n_i > 0$ and $\sum_{i=1}^{m} n_i = n$. Furthermore, we will have $m \leq n \leq 3m$, so that $\log n = \log m + O(1)$. The notation $\mathcal{R}_{\text{tim}}$ is reminiscent of $\mathcal{R}$, but $\mathcal{R}$ is a sequence of runs whereas $\mathcal{R}_{\text{tim}}$ is a sequence of run lengths. Since the merge cost of Timsort depends only on the lengths of the runs, it is more convenient to work directly with the sequence of run lengths.

The sequence $\mathcal{R}_{\text{tim}}(n)$, for $1 \leq n$, is defined as follows.[4] First, for $n \leq 3$, $\mathcal{R}_{\text{tim}}(n)$ is the sequence $\langle n \rangle$,

[4] For purposes of this proof, we allow run lengths to equal 1. Strictly speaking, this cannot occur since all original runs will have length at least 2. This is unimportant for the proof however, as the run lengths $\mathcal{R}_{\text{tim}}(n)$ could be doubled and the asymptotic analysis needed for the proof would be essentially unchanged.

i.e., representing a single run of length n. Let $n' = \lfloor n/2 \rfloor$. For even $n \geq 4$, we have $n = 2n'$ and define $\mathcal{R}_{\text{tim}}(n)$ to be the concatenation of $\mathcal{R}_{\text{tim}}(n')$, $\mathcal{R}_{\text{tim}}(n'-1)$ and $\langle 1 \rangle$. For odd $n \geq 4$, we have $n = 2n'+1$ and define $\mathcal{R}_{\text{tim}}(n)$ to be the concatenation of $\mathcal{R}_{\text{tim}}(n')$, $\mathcal{R}_{\text{tim}}(n'-1)$ and $\langle 2 \rangle$.

We claim that for $n \geq 4$, Timsort operates with run lengths $\mathcal{R}_{\text{tim}}(n)$ as follows: The first phase processes the runs from $\mathcal{R}_{\text{tim}}(n')$ and merges them into a single run of length n' which is the only element of the stack $\mathcal{Q}$. The second phase processes the runs from $\mathcal{R}_{\text{tim}}(n'-1)$ and merges them also into a single run of length $n'-1$; at this point the stack contains two runs, of lengths n' and $n'-1$. Since $n'-1 < n'$, no further merge occurs immediately. Instead, the final run is loaded onto the stack: it has length n'' equal to either 1 or 2. Now $n' \leq n'-1+n''$ and the test $|X| \leq |Y| + |Z|$ on line 10 of Algorithm 2 is triggered, so Timsort merges the top two elements of the stack, and then the test $|Y| \leq |Z|$ causes the merge of the final two elements of the stack.

This claim follows from Claim 1. We say that the stack $\mathcal{Q}$ is *stable* if none of the tests on lines 8,10,12,14, of Algorithm 2 hold.

CLAIM 1. *Suppose that $\mathcal{R}_{\text{tim}}(n)$ is the initial subsequence of a sequence $\mathcal{R}'$ of run lengths, and that Timsort is initially started with run lengths $\mathcal{R}'$ either (a) with the stack $\mathcal{Q}$ empty or (b) with the top element of $\mathcal{Q}$ a run of length $n_0 > n$ and the second element of $\mathcal{Q}$ (if it exists) a run of length $n_1 > n_0 + n$. Then Timsort will start by processing exactly the runs whose lengths are those of $\mathcal{R}_{\text{tim}}(n)$, merging them into a single run which becomes the new top element of $\mathcal{Q}$. Timsort will do this without performing any merge of runs that were initially in $\mathcal{Q}$ and without (yet) processing any of the remaining runs in $\mathcal{R}'$.*

Claim 1 is proved by induction on n. The base case, where $n \leq 3$, is trivial since with $\mathcal{Q}$ stable, Timsort immediately reads in the first run from $\mathcal{R}'$. The case of $n \geq 4$ uses the induction hypothesis twice, since $\mathcal{R}_{\text{tim}}(n)$ starts off with $\mathcal{R}_{\text{tim}}(n')$ followed by $\mathcal{R}_{\text{tim}}(n'-1)$. The induction hypothesis applied to n' implies that the runs of $\mathcal{R}_{\text{tim}}(n')$ are first processed and merged to become the top element of $\mathcal{Q}$. The stack elements X, Y, Z have lengths n_1, n_0, n' (if they exist), so the stack is now stable. Now the induction hypothesis for $n'-1$ applies, so Timsort next loads and merges the runs of $\mathcal{R}_{\text{tim}}(n'-1)$. Now the top stack elements W, X, Y, Z have lengths $n_1, n_0, n', n'-1$ and $\mathcal{Q}$ is again stable. Finally, the single run of length n'' is loaded onto the stack. This triggers the test $|X| \leq |Y| + |Z|$, so the top two elements are merged. Then the test $|Y| \leq |Z|$ is triggered, so the top two elements are again merged. Now the top elements of the stack (those which exist) are runs of length n_1, n_0, n, and Claim 1 is proved.

Let $c(n)$ be the merge cost of the Timsort algorithm on the sequence $\mathcal{R}_{\text{tim}}(n)$ of run lengths. The two merges described at the end of the proof of Claim 1 have merge cost $(n'-1)+n''$ plus $n'+(n'-1)+n'' = n$. Therefore, for $n > 3$, $c(n)$ satisfies

(2.1)
$$c(n) \;=\; \begin{cases} c(n') + c(n'-1) + \frac{3}{2}n & \text{if } n \text{ is even} \\ c(n') + c(n'-1) + \frac{3}{2}n + \frac{1}{2} & \text{if } n \text{ is odd.} \end{cases}$$

Also, $c(1) = c(2) = c(3) = 0$ since no merges are needed. Equation (2.1) can be summarized as

$$c(n) \;=\; c(\lfloor n/2 \rfloor) + c(\lfloor n/2 \rfloor - 1) + \tfrac{3}{2}n + \tfrac{1}{2}(n \bmod 2).$$

The function $n \mapsto \frac{3}{2}n + \frac{1}{2}(n \bmod 2)$ is strictly increasing. So, by induction, $c(n)$ is strictly increasing for $n \geq 3$. Hence $c(\lfloor n/2 \rfloor) > c(\lfloor n/2 \rfloor - 1)$, and thus $c(n) \geq 2c(\lfloor n/2 \rfloor - 1) + \frac{3}{2}n$ for all $n > 3$.

For $x \in \mathbb{R}$, define $b(x) = c(\lfloor x - 3 \rfloor)$. Since $c(n)$ is nondecreasing, so is $b(x)$. Then

$$\begin{aligned} b(x) &= c(\lfloor x-3 \rfloor) \\ &\geq 2c(\lfloor (x-3)/2 \rfloor - 1) + \frac{3}{2}(x-3) \\ &\geq 2c(\lfloor x/2 \rfloor - 3) + \frac{3}{2}(x-3) \\ &= 2b(x/2) + \frac{3}{2}(x-3). \end{aligned}$$

CLAIM 2. *For all $x \geq 3$, $b(x) \geq \frac{3}{2} \cdot [x(\lfloor \log x \rfloor - 2) - x + 3]$.*

We prove the claim by induction, namely by induction on n that it holds for all $x < n$. The base case is when $3 \leq x < 8$ and is trivial since the lower bound is negative and $b(x) \geq 0$. For the induction step, the claim is known to hold for $x/2$. Then, since $\log(x/2) = (\log x) - 1$,

$$\begin{aligned} b(x) &\geq 2 \cdot b(x/2) + \frac{3}{2}(x-3) \\ &\geq 2 \cdot \Big(\frac{3}{2} \cdot \big[(x/2)(\lfloor \log x \rfloor - 3) - x/2 + 3 \big] \Big) \\ &\quad + \frac{3}{2}(x-3) \\ &= \frac{3}{2} \cdot [x(\lfloor \log x \rfloor - 2) - x + 3] \end{aligned}$$

proving the claim.

Claim 2 implies that

$$c(n) = b(n+3) \geq (\frac{3}{2} - o(1)) \cdot n \log n.$$

This proves Theorem 2.2.

3 The α-stack sort

Augur-Nicaud-Pivoteau [2] introduced the α-stack sort as a 2-aware stable merge sort; it was inspired by Timsort and designed to be simpler to implement and to have a simpler analysis. (The algorithm (e2) of [31] is the same as α-stack sort with $\alpha = 2$.) Let $\alpha > 1$ be a constant. The α-stack sort is shown in Algorithm 3. It makes less effort than Timsort to optimize the order of merges: up until the run decomposition is exhausted, its only merge rule is that Y and Z are merged whenever $|Y| \leq \alpha|Z|$. An $O(n \log n)$ upper bound on its runtime is given by the next theorem.

Algorithm 3 The α-stack sort. α is a constant > 1.

```
1:  procedure α-STACK(S, n)
2:      R ← run decomposition of S
3:      Q ← ∅
4:      while R ≠ ∅ do
5:          Remove the next run R from R and push it
              onto Q
6:          while |Y| ≤ α|Z| do
7:              Merge Y and Z
8:          end while
9:      end while
10:     while |Q| ≥ 1 do
11:         Merge Y and Z
12:     end while
13: end procedure
```

THEOREM 3.1. ([2]) *Fix $\alpha > 1$. The merge cost for the α-stack sort is $O(n \log n)$.*

[2] did not explicitly mention the constant implicit in this upper bound, but their proof establishes a constant equal to approximately $(1 + \alpha)/\log \alpha$. For instance, for $\alpha = 2$, the merge cost is bounded by $(3 + o(1))n \log n$. The constant is minimized at $\alpha \approx 3.591$, where is it approximately 2.489.

THEOREM 3.2. *Let $1 < \alpha$. The worst-case merge cost of the α-stack sort on inputs of length n is $\geq (c_\alpha - o(1)) \cdot n \log n$, where c_α equals $\frac{\alpha+1}{(\alpha+1)\log(\alpha+1) - \alpha \log(\alpha)}$.*

The proof of Theorem 3.2 is postponed until Theorem 5.1 proves a stronger lower bound for α-merge sorts; the same construction works to prove both theorems. The value c_α is quite small, e.g., $c_2 \approx 1.089$; this is is discussed more in Section 5.

The lower bound of Theorem 3.2 is not very strong since the constant is close to 1. In fact, since a binary tree of merges gives a merge cost of $n\lceil \log m \rceil$, it is more relevant to give upper bounds in terms of $O(n \log m)$ instead of $O(n \log n)$. The next theorem shows that α-stack sort can be very far from optimal in this respect.

THEOREM 3.3. *Let $1 < \alpha$. The worst-case merge cost of the α-stack sort on inputs of length n which decompose into m original runs is $\omega(n \log m)$.*

In other words, for any $c > 0$, there are inputs with arbitrarily large values for n and m so that α-stack sort has merge cost $> c \cdot n \log m$.

Proof. Let s be the least integer such that $2^s \geq \alpha$. Let $\mathcal{R}_{\alpha\text{st}}(m)$ be the sequence of run lengths

$$\langle 2^{(m-1)\cdot s} - 1,\ 2^{(m-2)\cdot s} - 1,\ \ldots, 2^{3s} - 1,\ 2^{2s} - 1,\ 2^s - 1,\ 2^{m\cdot s} \rangle.$$

$\mathcal{R}_{\alpha\text{st}}(m)$ describes m runs whose lengths sum to $n = 2^{m\cdot s} + \sum_{i=1}^{m-1}(2^{i\cdot s} - 1)$, so $2^{m\cdot s} < n < 2^{m\cdot s+1}$. Since $2^s \geq \alpha$, the test $|Y| \leq \alpha|Z|$ on line 6 of Algorithm 3 is triggered only when the run of length 2^{ms} is loaded onto the stack $\mathcal{Q}$; once this happens the runs are all merged in order from right-to-left. The total cost of the merges is $(m-1) \cdot 2^{ms} + \sum_{i=1}^{m-1} i \cdot (2^{i\cdot s} - 1)$ which is certainly greater than $(m-1) \cdot 2^{ms}$. Indeed, that comes from the fact that the final run in $\mathcal{R}_{\alpha\text{st}}(m)$ is involved in $m-1$ merges. Since $n < 2 \cdot 2^{m\cdot s}$, the total merge cost is greater than $\frac{n}{2}(m-1)$, which is $\omega(n \log m)$.

4 The Shivers merge sort

The 2-aware Shivers sort [28], shown in Algorithm 4, is similar to the 2-stack merge sort, but with a modification that makes a surprising improvement in the bounds on its merge cost. Although never published, this algorithm was presented in 1999.

Algorithm 4 The Shivers sort.

```
1:  procedure SSORT(S, n)
2:      R ← run decomposition of S
3:      Q ← ∅
4:      while R ≠ ∅ do
5:          Remove the next run R from R and push it
              onto Q
6:          while 2^⌊log |Y|⌋ ≤ |Z| do
7:              Merge Y and Z
8:          end while
9:      end while
10:     while |Q| ≥ 1 do
11:         Merge Y and Z
12:     end while
13: end procedure
```

The only difference between the Shivers sort and the 2-stack sort is the test used to decide when to

merge. Namely, line 6 tests $2^{\lfloor \log |Y| \rfloor} \leq |Z|$ instead of $|Y| \leq 2 \cdot |Z|$. Since $2^{\lfloor \log |Y| \rfloor}$ is $|Y|$ rounded down to the nearest power of two, this is somewhat like an α-sort with α varying dynamically in the range $[1, 2)$.

The Shivers sort has the same undesirable lower bound as 2-sort in terms of $\omega(n \log m)$:

THEOREM 4.1. *The worst-case merge cost of the Shivers sort on inputs of length n which decompose into m original runs is $\omega(n \log m)$.*

Proof. This is identical to the proof of Theorem 3.3. We now let $\mathcal{R}_{\text{sh}}(m)$ be the sequence of run lengths

$$\langle\, 2^{m-1} - 1,\, 2^{m-2} - 1,\, \ldots, 7,\, 3,\, 1,\, 2^m \rangle,$$

and argue as before.

THEOREM 4.2. ([28]) *The merge cost of Shivers sort is $(1 + o(1))n \log n$.*

We present a proof which is simpler than that of [28]. The proof of Theorem 4.2 assumes that at a given point in time, the stack $\mathcal{Q}$ has ℓ elements $Q_1, \ldots, Q_\ell$, and uses w_{Q_i} to denote the merge cost of Q_i. We continue to use the convention that W, X, Y, Z denote $Q_{\ell-3}, Q_{\ell-2}, Q_{\ell-1}, Q_\ell$ if they exist.

Proof. Define k_{Q_i} to equal $\lfloor \log |Q_i| \rfloor$. Obviously, $|Q_i| \geq 2^{k_{Q_i}}$. The test on line 6 works to maintain the invariant that each $|Q_{i+1}| < 2^{k_{Q_i}}$ or equivalently $k_{i+1} < k_i$. Thus, for $i < \ell - 1$, we always have $|Q_{i+1}| < 2^{k_{Q_i}}$ and $k_{i+1} < k_i$. This condition can be momentarily violated for $i = \ell - 1$, i.e. if $|Z| \geq 2^{k_Y}$ and $k_Y \leq k_Z$, but then the Shivers sort immediately merges Y and Z.

As a side remark, since each $k_{i+1} < k_i$ for $i \leq \ell - 1$, since $2^{k_1} \leq |Q_1| \leq n$, and since $Q_{\ell-1} \geq 1 = 2^0$, the stack height ℓ is $\leq 2 + \log n$. (In fact, a better analysis shows it is $\leq 1 + \log n$.)

CLAIM 3. *Throughout the execution of the main loop (lines 4-9), the Shivers sort satisfies*

a. $w_{Q_i} \leq k_{Q_i} \cdot |Q_i|$, *for all* $i \leq \ell$,

b. $w_Z \leq k_Y \cdot |Z|$ *i.e.,* $w_{Q_\ell} \leq k_{Q_{\ell-1}} \cdot |Q_\ell|$, *if* $\ell > 1$.

When $i = \ell$, a. says $w_Z \leq k_Z |Z|$. Since k_Z can be less than or greater than k_Y, this neither implies, nor is implied by, b.

The lemma is proved by induction on the number of updates to the stack $\mathcal{Q}$ during the loop. Initially $\mathcal{Q}$ is empty, and a. and b. hold trivially. There are two induction cases to consider. The first case is when an original run is pushed onto $\mathcal{Q}$. Since this run, namely Z, has never been merged, its weight is $w_Z = 0$. So b. certainly holds. For the same reason and using the induction hypothesis, a. holds. The second case is when $2^{k_Y} \leq |Z|$, so $k_Y \leq k_Z$, and Y and Z are merged; here ℓ will decrease by 1. The merge cost w_{YZ} of the combination of Y and Z equals $|Y| + |Z| + w_Y + w_Z$, so we must establish two things:

a′. $|Y| + |Z| + w_Y + w_Z \;\leq\; k_{YZ} \cdot (|Y| + |Z|)$, where $k_{YZ} = \lfloor \log(|Y| + |Z|) \rfloor$.

b′. $|Y| + |Z| + w_Y + w_Z \;\leq\; k_X \cdot (|Y| + |Z|)$, if $\ell > 2$.

By induction hypotheses $w_Y \leq k_Y |Y|$ and $w_Z \leq k_Y |Z|$. Thus the lefthand sides of a′. and b′. are $\leq (k_Y + 1) \cdot (|Y| + |Z|)$. As already discussed, $k_Y < k_X$, therefore condition b. implies that b′. holds. And since $2^{k_Y} \leq |Z|$, $k_Y < k_{YZ}$, so condition a. implies that a′. also holds. This completes the proof of Claim 3.

Claim 3 implies that the total merge cost incurred at the end of the main loop incurred is $\leq \sum_i w_{Q_i} \leq \sum_i k_i |Q_i|$. Since $\sum_i Q_i = n$ and each $k_i \leq \log n$, the total merge cost is $\leq n \log n$.

We now upper bound the total merge cost incurred during the final loop on lines 10-12. When first reaching line 10, we have $k_{i+1} < k_i$ for all $i \leq \ell - 1$ hence $k_i < k_1 + 1 - i$ and $|Q_i| < 2^{k_1 + 2 - i}$ for all $i \leq \ell$. The final loop then performs $\ell - 1$ merges from right to left. Each Q_i for $i < \ell$ participates in i merge operations and Q_ℓ participates in $\ell - 1$ merges. The total merge cost of this is less than $\sum_{i=1}^{\ell} i \cdot |Q_i|$. Note that

$$\begin{aligned} \sum_{i=1}^{\ell} i \cdot |Q_i| \;&<\; 2^{k_1+2} \cdot \sum_{i=1}^{\ell} i \cdot 2^{-i} \\ &<\; 2^{k_1+2} \cdot 2 \;=\; 8 \cdot 2^{k_1} \;\leq\; 8n, \end{aligned}$$

where the last inequality follows by $2^{k_1} \leq |Q_1| \leq n$. Thus, the final loop incurs a merge cost $O(n)$, which is $o(n \log n)$.

Therefore the total merge cost for the Shivers sort is bounded by $n \log n + o(n \log n)$.

5 The 2-merge and α-merge sorts

This section introduces our new merge sorting algorithms, called the "2-merge sort" and the "α-merge sort", where $\varphi < \alpha < 2$ is a fixed parameter. These sorts are 3-aware, and this enables us to get algorithms with merge costs $(c_\alpha \pm o(1))n \log m$. The idea of the α-merge sort is to combine the construction of the 2-stack sort, with the idea from Timsort of merging X and Y instead of Y and Z whenever $|X| < |Z|$. But unlike the Timsort algorithm shown in Algorithm 2, we are able to use a 3-aware algorithm instead of a 4-aware algorithm. In addition, our merging rules are simpler, and our provable upper bounds are tighter. Indeed, our upper bounds for $\varphi < \alpha \leq 2$ are of the form $(c_\alpha + o(1)) \cdot n \log m$ with

$c_\alpha \le c_2 \approx 1.089$, but Theorem 2.2 proves a lower bound $1.5 \cdot n \log m$ for Timsort.

Algorithms 5 and 6 show the 2-merge sort and α-merge sort algorithms. Note that the 2-merge sort is almost, but not quite, the specialization of the α-merge sort to the case $\alpha = 2$. The difference is that line 6 of the 2-merge sort has a simpler **while** test than the corresponding line in the α-merge sort. As will be shown by the proof of Theorem 5.4, the fact that Algorithm 5 uses this simpler **while** test makes no difference to which merge operations are performed; in other words, it would be redundant to test the condition $|X| < 2|Y|$.

The 2-merge sort can also be compared to the α-stack sort shown in Algorithm 3. The main difference is that the merge of Y and Z on line 7 of the α-stack sort has been replaced by the lines lines 7-11 of the 2-merge which conditionally merge Y with either X or Z. For the 2-merge sort (and the α-merge sort), the run Y is never merged with Z if it could instead be merged with a shorter X. The other, perhaps less crucial, difference is that the weak inequality test on line 6 in the α-stack sort has been replaced with a strict inequality test on line 6 in the α-merge sort. We have made this change since it seems to make the 2-merge sort more efficient, for instance when all original runs have the same length.

Algorithm 5 The 2-merge sort.

```
1:  procedure 2-MERGE(S, n)
2:      R ← run decomposition of S
3:      Q ← ∅
4:      while R ≠ ∅ do
5:          Remove the next run R from R and push it
              onto Q
6:          while |Y| < 2|Z| do
7:              if |X| < |Z| then
8:                  Merge X and Y
9:              else
10:                 Merge Y and Z
11:             end if
12:         end while
13:     end while
14:     while |Q| ≥ 1 do
15:         Merge Y and Z
16:     end while
17: end procedure
```

We will concentrate mainly on the cases for $\varphi < \alpha \le 2$ where $\varphi \approx 1.618$ is the golden ratio. Values for $\alpha > 2$ do not seem to give useful merge sorts; our upper bound proof does not work for $\alpha \le \varphi$.

Algorithm 6 The α-merge sort. α is a constant such that $\varphi < \alpha < 2$.

```
1:  procedure α-MERGE(S, n)
2:      R ← run decomposition of S
3:      Q ← ∅
4:      while R ≠ ∅ do
5:          Remove the next run R from R and push it
              onto Q
6:          while |Y| < α|Z| or |X| < α|Y| do
7:              if |X| < |Z| then
8:                  Merge X and Y
9:              else
10:                 Merge Y and Z
11:             end if
12:         end while
13:     end while
14:     while |Q| ≥ 1 do
15:         Merge Y and Z
16:     end while
17: end procedure
```

DEFINITION 5.1. *Let $\alpha \ge 1$, the constant c_α is defined by*

$$c_\alpha = \frac{\alpha + 1}{(\alpha+1)\log(\alpha+1) - \alpha\log(\alpha)}. \tag{5.2}$$

For $\alpha = 2$, $c_2 = 3/\log(27/4) \approx 1.08897$. For $\alpha = \varphi$, $c_\varphi \approx 1.042298$. For $\alpha > 1$, c_α is strictly increasing as a function of α. Thus, $1.042 < c_\alpha < 1.089$ when $\varphi < \alpha \le 2$.

The next four subsections give nearly matching upper and lower bounds for the worst-case running time of the 2-merge sort and the α-merge sort for $\varphi < \alpha < 2$.

5.1 Lower bound for 2-merge sort and α-merge sort.

THEOREM 5.1. *Fix $\alpha > 1$. The worst-case merge cost of the α-merge sort algorithm is $\ge (c_\alpha - o(1))n \log n$.*

The corresponding theorem for $\alpha = 2$ is:

THEOREM 5.2. *The worst-case merge cost of the 2-merge sort algorithm is $\ge (c_2 - o(1))n \log n$, where $c_2 = 3/\log(27/4) \approx 1.08897$.*

The proof of Theorem 5.1 also establishes Theorem 3.2, as the same lower bound construction works for both α-stack sort and α-merge sort. The only difference is that part d. of Claim 5 is used instead of part c. In addition, the proof of Theorem 5.1 also establishes Theorem 5.2; indeed, exactly the same proof applies verbatim, just uniformly replacing "α" with "2".

Proof of Theorem 5.1 Fix $\alpha > 1$. For $n \geq 1$, we define a sequence $\mathcal{R}_{\alpha\mathrm{m}}(n)$ of run lengths that will establish the lower bound. Define N_0 to equal $3 \cdot \lceil \alpha + 1 \rceil$. For $n < N_0$, set $\mathcal{R}_{\alpha\mathrm{m}}$ to be the sequence $\langle n \rangle$, containing a single run of length n. For $n \geq N_0$, define $n''' = \lfloor \frac{n}{\alpha+1} \rfloor + 1$ and $n^* = n - n'''$. Thus n''' is the least integer greater than $n/(\alpha+1)$. Similarly define $n'' = \lfloor \frac{n^*}{\alpha+1} \rfloor + 1$ and $n' = n^* - n''$. These four values can be equivalently uniquely characterized as satisfying

$$n''' = \frac{1}{\alpha+1} n + \epsilon_1 \tag{5.3}$$

$$n^* = \frac{\alpha}{\alpha+1} n - \epsilon_1 \tag{5.4}$$

$$n'' = \frac{\alpha}{(\alpha+1)^2} n - \frac{1}{\alpha+1} \epsilon_1 + \epsilon_2 \tag{5.5}$$

$$n' = \frac{\alpha^2}{(\alpha+1)^2} n - \frac{\alpha}{\alpha+1} \epsilon_1 - \epsilon_2 \tag{5.6}$$

for some $\epsilon_1, \epsilon_2 \in (0, 1]$. The sequence $\mathcal{R}_{\alpha\mathrm{m}}(n)$ of run lengths is inductively defined to be the concatenation of $\mathcal{R}_{\alpha\mathrm{m}}(n')$, $\mathcal{R}_{\alpha\mathrm{m}}(n'')$ and $\mathcal{R}_{\alpha\mathrm{m}}(n''')$.

Claim 4. *Let $n \geq N_0$.*

a. *$n = n' + n'' + n'''$ and $n^* = n' + n''$.*
b. *$\alpha(n''' - 3) \leq n^* < \alpha n'''$.*
c. *$\alpha(n'' - 3) \leq n' < \alpha n''$.*
d. *$n''' \geq 3$.*
e. *$n' \geq 1$ and $n'' \geq 1$.*

Part a. of the claim is immediate from the definitions. Part b. is immediate from the equalities (5.3) and (5.4) since $0 < \epsilon_1 \leq 1$ and $\alpha > 1$. Part c. is similarly immediate from (5.5 and (5.6) since also $0 < \epsilon_2 \leq 1$. Part d. follows from (5.3) and $n \geq N_0 \geq 3(\alpha + 1)$. Part e. follows by (5.5, (5.6), $\alpha > 1$, and $n \geq N_0$.

Claim 5. *Let $\mathcal{R}_{\alpha\mathrm{m}}(n)$ be as defined above.*

a. *The sums of the run lengths in $\mathcal{R}_{\alpha\mathrm{m}}(n)$ is n.*
b. *If $n \geq N_0$, then the final run length in $\mathcal{R}_{\alpha\mathrm{m}}(n)$ is ≥ 3.*
c. *Suppose that $\mathcal{R}_{\alpha\mathrm{m}}(n)$ is the initial subsequence of a sequence $\mathcal{R}'$ of run lengths and that the α-merge sort is initially started with run lengths $\mathcal{R}'$ and (a) with the stack $\mathcal{Q}$ empty or (b) with the top element of $\mathcal{Q}$ a run of length $\geq \alpha(n-3)$. Then the α-merge sort will start by processing exactly the runs whose lengths are those of $\mathcal{R}_{\alpha\mathrm{m}}(n)$, merging them into single run which becomes the new top element of $\mathcal{Q}$. This will be done without merging any runs that were initially in $\mathcal{Q}$ and without (yet) processing any of the remaining runs in $\mathcal{R}'$.*
d. *The property c. also holds for α-stack sort.*

Part a. is immediate from the definitions using induction on n. Part b. is a consequence of Claim 4(d.) and the fact that the final entry of $\mathcal{R}_{\alpha\mathrm{m}}$ is a value $n''' < N_0$ for some n. Part c. is proved by induction on n, similarly to the proof of Claim 1. It is trivial for the base case $n < N_0$. For $n \geq N_0$, $\mathcal{R}_{\alpha\mathrm{m}}(n)$ is the concatenation of $\mathcal{R}_{\alpha\mathrm{m}}(n'), \mathcal{R}_{\alpha\mathrm{m}}(n''), \mathcal{R}_{\alpha\mathrm{m}}(n''')$. Applying the induction hypothesis to $\mathcal{R}_{\alpha\mathrm{m}}(n')$ yields that these runs are initially merged into a single new run of length n' at the top of the stack. Then applying the induction hypothesis to $\mathcal{R}_{\alpha\mathrm{m}}(n'')$ shows that those runs are merged to become the top run on the stack. Since the last member of $\mathcal{R}_{\alpha\mathrm{m}}(n'')$ is a run of length ≥ 3, every intermediate member placed on the stack while merging the runs of $\mathcal{R}_{\alpha\mathrm{m}}(n'')$ has length $\leq n'' - 3$. And, by Claim 4(c.), these cannot cause a merge with the run of length n' already in $\mathcal{Q}$. Next, again by Claim 4(c.), the top two members of the stack are merged to form a run of length $n^* = n' + n''$. Applying the induction hypothesis a third time, and arguing similarly with Claim 4(b.), gives that the runs of $\mathcal{R}_{\alpha\mathrm{m}}(n''')$ are merged into a single run of length n''', and then merged with the run of length n^* to obtain a run of length n. This proves part c. of Claim 5. Part d. is proved exactly like part c.; the fact that α-stack sort is only 2-aware and never merges X and Y makes the argument slightly easier in fact. This completes the proof of Claim 5.

Claim 6. *Let $c(x)$ equal the merge cost of the α-merge sort on an input sequence with run lengths given by $\mathcal{R}_{\alpha\mathrm{m}}(n)$. Then $c(n) = 0$ for $n < N_0$. For $n \geq N_0$,*

$$\begin{aligned} c(n) &= c(n') + c(n'') + c(n''') + 2n' + 2n'' + n''' \\ &= c(n') + c(n'') + c(n''') + n + n' + n''. \end{aligned} \tag{5.7}$$

For $n \geq N_0$, $c(n)$ is strictly increasing as a function of n.

The first equality of Equation (5.7) is an immediate consequence of the proof of part c. of Claim 5; the second follows from $n = n' + n'' + n'''$. To see that $c(n)$ is increasing for $n \geq N_0$, let $(n+1)', (n+1)'', (n+1)'''$ indicate the three values such that $\mathcal{R}_{\alpha\mathrm{m}}(n+1)$ is the concatenation of $\mathcal{R}_{\alpha\mathrm{m}}((n+1)')$, $\mathcal{R}_{\alpha\mathrm{m}}((n+1)'')$ and $\mathcal{R}_{\alpha\mathrm{m}}((n+1)''')$. Note that $(n+1)' \geq n'$, $(n+1)'' \geq n''$, and $(n+1)''' \geq n'''$. An easy proof by induction now shows that $c(n+1) > c(n)$ for $n \geq N_0$, and Claim 6 is proved.

Let $\delta = \lceil 2(\alpha+1)^2/(2\alpha+1)\rceil$. (For $1 < \alpha \le 2$, we have $\delta \le 4$.) We have $\delta \le N_0 - 1$ for all $\alpha > 1$. For real $x \ge N_0$, define $b(x) = c(\lfloor x \rfloor - \delta)$. Since $c(n)$ is increasing, $b(x)$ is nondecreasing.

CLAIM 7. a. $\frac{1}{\alpha+1}n - \delta \le (n-\delta)'''$.

b. $\frac{\alpha}{(\alpha+1)^2}n - \delta \le (n-\delta)''$.

c. $\frac{\alpha^2}{(\alpha+1)^2}n - \delta \le (n-\delta)'$.

d. *If $x \ge N_0 + \delta$, then*

$$b(x) \ge b(\tfrac{\alpha^2}{(\alpha+1)^2}x) + b(\tfrac{\alpha}{(\alpha+1)^2}x) + b(\tfrac{1}{\alpha+1}x) + \tfrac{2\alpha+1}{\alpha+1}(x-\delta-1) - 1.$$

For a., (5.3) implies that $(n-\delta)''' \ge \frac{n-\delta}{\alpha+1}$, so a. follows from $-\delta \le -\delta/(\alpha+1)$. This holds as $\delta > 0$ and $\alpha > 1$. For b., (5.5) implies that $(n-\delta)'' \ge \frac{\alpha}{(\alpha+1)^2}(n-\delta) - \frac{1}{\alpha+1}$, so after simplification, b. follows from $\delta \ge (\alpha+1)/(\alpha^2+\alpha+1)$; it is easy to verify that this holds by choice of δ. For c., (5.6) also implies that $(n-\delta)' \ge \frac{\alpha^2}{(\alpha+1)^2}(n-\delta) - 2$, so after simplification, c. follows from $\delta \ge 2(\alpha+1)^2/(2\alpha+1)$.

To prove part d., letting $n = \lfloor x \rfloor$ and using parts a., b. and c., equations (5.3)-(5.6), and the fact that $b(x)$ and $c(n)$ are nondecreasing, we have

$$\begin{aligned}
b(x) &= c(n-\delta) \\
&= c((n-\delta)') + c((n-\delta)'') \\
&\quad + c((n-\delta)''') + (n-\delta) + (n-\delta)' + (n-\delta)'' \\
&\ge c(\lfloor \tfrac{\alpha^2}{(\alpha+1)^2}n \rfloor - \delta) \\
&\quad + c(\lfloor \frac{\alpha}{(\alpha+1)^2}n \rfloor - \delta) + c(\lfloor \frac{1}{\alpha+1}n \rfloor - \delta) \\
&\quad + \big(1 + \tfrac{\alpha^2}{(\alpha+1)^2} + \tfrac{\alpha}{(\alpha+1)^2}\big)\cdot(n-\delta) \\
&\quad - \tfrac{\alpha}{\alpha+1}\epsilon_1 - \epsilon_2 - \tfrac{1}{\alpha+1}\epsilon_1 + \epsilon_2 \\
&\ge b(\tfrac{\alpha^2}{(\alpha+1)^2}x) + b(\tfrac{\alpha}{(\alpha+1)^2}x) \\
&\quad + b(\tfrac{1}{\alpha+1}x) + \frac{2\alpha+1}{\alpha+1}(x-\delta-1) - 1.
\end{aligned}$$

Claim 7(d.) gives us the basic recurrence needed to lower bound $b(x)$ and hence $c(n)$.

CLAIM 8. *For all $x \ge \delta + 1$,*

$$b(x) \ge c_\alpha \cdot x\log x - Bx + A, \tag{5.8}$$

where

$$A = \tfrac{2\alpha+1}{2\alpha+2}(\delta+1) + \tfrac{1}{2}$$

and

$$B = \tfrac{A}{\delta+1} + c_\alpha \log(\max\{N_0+\delta+1, \left\lceil \tfrac{(\delta+1)(\alpha+1)^2}{\alpha} \right\rceil\}).$$

The claim is proved by induction, namely we prove by induction on n that (5.8) holds for all $x < n$. The base case is for $x < n = \max\{N_0+\delta+1, \left\lceil \frac{(\delta+1)(\alpha+1)^2}{\alpha} \right\rceil\}$. In this case $b(x)$ is non-negative, and the righthand side of (5.8) is ≤ 0 by choice of B. Thus (5.8) holds trivially.

For the induction step, we may assume $n-1 \le x < n$ and have

$$\delta + 1 \le \tfrac{\alpha}{(\alpha+1)^2}x < \tfrac{\alpha^2}{(\alpha+1)^2}x < \tfrac{1}{\alpha+1}x < n-1.$$

The first of the above inequalities follows from $x \ge \frac{(\delta+1)(\alpha+1)^2}{\alpha}$; the remaining follow from $1 < \alpha < 2$ and $x < n$ and $n \ge N_0 \ge 6$. Therefore, the induction hypothesis implies that the bound (5.8) holds for $b(\frac{\alpha^2}{(\alpha+1)^2}x)$, $b(\frac{\alpha}{(\alpha+1)^2}x)$, and $b(\frac{1}{\alpha+1}x)$. So by Claim 7(d.),

$$\begin{aligned}
b(x) &\ge c_\alpha \tfrac{\alpha^2}{(\alpha+1)^2}x\log\tfrac{\alpha^2 x}{(\alpha+1)^2} \\
&\quad - B\tfrac{\alpha^2}{(\alpha+1)^2}x + \tfrac{2\alpha+1}{2\alpha+2}(\delta+1) + \tfrac{1}{2} \\
&\quad + c_\alpha \tfrac{\alpha}{(\alpha+1)^2}x\log\tfrac{\alpha x}{(\alpha+1)^2} \\
&\quad - B\tfrac{\alpha}{(\alpha+1)^2}x + \tfrac{2\alpha+1}{2\alpha+2}(\delta+1) + \tfrac{1}{2} \\
&\quad + c_\alpha \tfrac{1}{\alpha+1}x\log\tfrac{x}{\alpha+1} \\
&\quad - B\tfrac{1}{\alpha+1}x + \tfrac{2\alpha+1}{2\alpha+2}(\delta+1) + \tfrac{1}{2} \\
&\quad + \tfrac{2\alpha+1}{\alpha+1}x - \tfrac{2\alpha+1}{\alpha+1}(\delta+1) - 1 \\
&= c_\alpha x\log x - Bx + A \\
&\quad + c_\alpha x\Big[\tfrac{\alpha^2}{(\alpha+1)^2}\log\tfrac{\alpha^2}{(\alpha+1)^2} \\
&\quad + \tfrac{\alpha}{(\alpha+1)^2}\log\tfrac{\alpha}{(\alpha+1)^2} + \tfrac{1}{\alpha+1}\log\tfrac{1}{\alpha+1}\Big] \\
&\quad + \tfrac{2\alpha+1}{\alpha+1}x.
\end{aligned}$$

The quantity in square brackets is equal to

$$\begin{aligned}
&\Big(\tfrac{2\alpha^2}{(\alpha+1)^2} + \tfrac{\alpha}{(\alpha+1)^2}\Big)\log\alpha \\
&\qquad - \Big(\tfrac{2\alpha^2}{(\alpha+1)^2} + \tfrac{2\alpha}{(\alpha+1)^2} + \tfrac{1}{\alpha+1}\Big)\log(\alpha+1) \\
&= \tfrac{\alpha(2\alpha+1)}{(\alpha+1)^2}\log\alpha - \tfrac{2\alpha^2+3\alpha+1}{(\alpha+1)^2}\log(\alpha+1) \\
&= \tfrac{\alpha(2\alpha+1)}{(\alpha+1)^2}\log\alpha - \tfrac{2\alpha+1}{\alpha+1}\log(\alpha+1) \\
&= \tfrac{\alpha\log\alpha - (\alpha+1)\log(\alpha+1)}{\alpha+1}\cdot\tfrac{2\alpha+1}{\alpha+1}.
\end{aligned}$$

Since $c_\alpha = (\alpha+1)/((\alpha+1)\log(\alpha+1) - \alpha\log\alpha)$, we get that $b(x) \ge c_\alpha x\log x - Bx + A$. This completes the induction step and proves Claim 8.

Since A and B are constants, this gives $b(x) \ge (c_\alpha - o(1))x\log x$. Therefore $c(n) = b(n+\delta) \ge (c_\alpha - o(1))n\log n$. This completes the proofs of Theorems 3.2, 5.1 and 5.2.

5.2 Upper bound for 2-merge sort and α-merge sort — preliminaries. We next state our precise upper bounds on the worst-case runtime of the 2-merge sort and the α-merge sort for $\varphi < \alpha < 2$. The upper bounds will have the form $n \cdot (d_\alpha + c_\alpha \log n)$, with no hidden or missing constants. c_α was already defined in (5.2). For $\alpha = 2$, $c_\alpha \approx 1.08897$ and the constant d_α is

$$(5.9)\quad \begin{aligned} d_2 &= 6 - c_2 \cdot (3\log 6 - 2\log 4) \\ &= 6 - c_2 \cdot ((3\log 3) - 1) \;\approx\; 1.91104. \end{aligned}$$

For $\varphi < \alpha < 2$, first define

$$(5.10)\qquad k_0(\alpha) \;=\; \min\{\ell \in N \;:\; \tfrac{\alpha^2-\alpha-1}{\alpha-1} \geq \tfrac{1}{\alpha^\ell}\}.$$

Note $k_0(\alpha) \geq 1$. Then set, for $\varphi < \alpha < 2$,

$$(5.11)\qquad d_\alpha \;=\; \frac{2^{k_0(\alpha)+1} \cdot \max\{(k_0(\alpha)+1), 3\} \cdot (2\alpha - 1)}{\alpha - 1} + 1.$$

Our proof for $\alpha = 2$ is substantially simpler than the proof for general α: it also gives the better constant d_2. The limits $\lim_{\alpha\to\varphi^+} k(\alpha)$ and $\lim_{\alpha\to\varphi^+} d_\alpha$ are both equal to ∞; we suspect this is not optimal.[5] However, by Theorems 5.1 and 5.2, the constant c_α is optimal.

THEOREM 5.3. *Let $\varphi < \alpha < 2$. The merge cost of the α-merge sort on inputs of length n composed of m runs is*

$$\leq n \cdot (d_\alpha + c_\alpha \log m).$$

The corresponding upper bound for $\alpha = 2$ is:

THEOREM 5.4. *The merge cost of the 2-merge sort on inputs of length n composed of m runs is $\leq n \cdot (d_2 + c_2 \log m) \approx n \cdot (1.91104 + 1.08897 \log m)$.*

The proofs of these theorems will allow us to easily prove the following upper bound on the size of the stack:

THEOREM 5.5. *Let $\varphi < \alpha < 2$. For α-merge sort on inputs of length n, the size $|\mathcal{Q}|$ of the stack is always $\leq 1 + \lceil \log_\alpha n \rceil = 1 + \lceil (\log n)/(\log \alpha) \rceil$. For 2-merge sort: the size of the stack is always $\leq 1 + \lceil \log n \rceil$.*

This extended abstract omits the proof of Theorem 5.3 due to space reasons. Thus, the rest of the discussion is specialized towards what is needed for Theorem 5.4 with $\alpha = 2$, and does not handle everything needed for the case of $\alpha < 2$.

[5] Already the term $2^{k_0(\alpha)+1}$ is not optimal as the proof of Theorem 5.3 shows that the base 2 could be replaced by $\sqrt{\alpha+1}$; we conjecture however, that in fact it is not necessary for the limit of d_α to be infinite.

Proving Theorems 5.3 and 5.4 requires handling three situations: First, the algorithm may have top stack element Z which is not too much larger than Y (so $|Z| \leq \alpha|Y|$): in this situation either Y and Z are merged or Z is small compared to Y and no merge occurs. This first situation will be handled by case (A) of the proofs. Second, the top stack element Z may be much larger than Y (so $|Z| > \alpha|Y|$): in this situation, the algorithm will repeatedly merge X and Y until $|Z| \leq |X|$. This is the most complicated part of the argument, and is handled by case (C) of the proof of Theorem 5.4. In the proof of Theorem 5.3, this is handled by two cases, cases (C) and (D). (However, recall that for space reasons, the proof of Theorem 5.3 is not included in this extended abstract, and instead will appear in the full version of the paper.) In the third situation, the original runs in $\mathcal{R}$ have been exhausted and the final loop on lines 14-16 repeatedly merges Y and Z. The third situation is handled by case (B) of the proofs.

The next three technical lemmas are key for the proof of Theorem 5.4. Lemma 5.1 is used for case (A) of the proof; the constant c_2 is exactly what is needed to make this hold. Lemma 5.2 is used for case (B). Lemma 5.3 is used for case (C).

LEMMA 5.1. *Let $\alpha > 1$. Let A, B, a, b be positive integers such that $A \leq \alpha B$ and $B \leq \alpha A$. Then*

$$(5.12)\quad \begin{aligned} &A \cdot c_\alpha \log a + B \cdot c_\alpha \log b + A + B \\ &\qquad\leq\; (A+B) \cdot c_\alpha \log(a+b). \end{aligned}$$

LEMMA 5.2. *Let $\alpha > 1$. Let A, B, a, b be positive integers such that $(\alpha - 1)B \leq A$. Then*

$$(5.13)\quad \begin{aligned} &A \cdot c_\alpha \log a + B \cdot (1 + c_\alpha \log b) + A + B \\ &\qquad\leq\; (A+B) \cdot (1 + c_\alpha \log(a+b)). \end{aligned}$$

LEMMA 5.3. *(For $\alpha = 2$.) Let A, B, a, b be positive integers. If $a \geq 2$ and $A \leq 2B$, then*

$$\begin{aligned} &A \cdot (d_2 + c_2 \log(a-1)) + A + B \\ &\qquad\leq (A+B) \cdot (d_2 - 1 + c_2 \log(a+b)). \end{aligned}$$

Proof of Lemma 5.1 The inequality (5.12) is equivalent to

$$\begin{aligned} &A \cdot (c_\alpha \log a - c_\alpha \log(a+b) + 1) \\ &\qquad\leq\; B \cdot (c_\alpha \log(a+b) - c_\alpha \log b - 1) \end{aligned}$$

and hence to

$$A \cdot \Big(1 + c_\alpha \log \frac{a}{a+b}\Big) \;\leq\; B \cdot \Big(-1 - c_\alpha \log \frac{b}{a+b}\Big).$$

Setting $t = b/(a+b)$, this is the same as

$$A \cdot (1 + c_\alpha \log(1-t)) \;\le\; B \cdot (-1 - c_\alpha \log t). \tag{5.14}$$

Let $t_0 = 1 - 2^{-1/c_\alpha}$. Since $c_\alpha > 1$, we have $t_0 < 1/2$, so $t_0 < 1 - t_0$. The lefthand side of (5.14) is positive iff $t < t_0$. Likewise, the righthand side is positive iff $t < 1 - t_0$. Thus (5.14) certainly holds when $t_0 \le t \le 1 - t_0$ where the lefthand side is ≤ 0 and the righthand side is ≥ 0.

Suppose $0 < t < t_0$, so $1 + c_\alpha \log(1-t)$ and $-1 - c_\alpha \log t$ are both positive. Since $A \le \alpha B$, to prove (5.14) it will suffice to prove $\alpha(1 + c_\alpha \log(1-t)) \le -1 - c_\alpha \log t$, or equivalently that $-1 - \alpha \ge c_\alpha \log(t(1-t)^\alpha)$. The derivative of $\log(t(1-t)^\alpha)$ is $(1-(1+\alpha)t)/(t(1-t))$; so $\log(t(1-t)^\alpha)$ is maximized at $t = 1/(1+\alpha)$ with value $\alpha \log \alpha - (\alpha+1)\log(\alpha+1)$. Thus the desired inequality holds by the definition of c_α, so (5.14) holds in this case.

Now suppose $1 - t_0 < t < 1$, so $1 + c_\alpha \log(1-t)$ and $-1 - c_\alpha \log t$ are both negative. Since $B \le \alpha A$, it suffices to prove $1 + c_\alpha \log(1-t) \le \alpha(-1 - c_\alpha \log t)$ or equivalently that $-1 - \alpha \ge c_\alpha \log(t^\alpha(1-t))$. This is identical to the situation of the previous paragraph, but with t replaced by $1-t$, so (5.14) holds in this case also.

Proof of Lemma 5.2. The inequality (5.13) is equivalent to

$$B \cdot (1 + c_\alpha \log b - c_\alpha \log(a+b)) \le A \cdot (c_\alpha \log(a+b) - c_\alpha \log a).$$

Since $\log(a+b) - \log a > 0$ and $(\alpha-1)B \le A$, it suffices to prove

$$1 + c_\alpha \log b - c_\alpha \log(a+b) \le (\alpha-1) \cdot (c_\alpha \log(a+b) - c_\alpha \log a).$$

This is equivalent to

$$c_\alpha \log b + (\alpha-1) \cdot c_\alpha \log(a) - \alpha \cdot c_\alpha \log(a+b) \;\le\; -1.$$

Letting $t = b/(a+b)$, we must show $-1 \ge c_\alpha \log(t(1-t)^{\alpha-1})$. Similarly to the previous proof, taking the first derivative shows that the righthand side is maximized with $t = 1/\alpha$, so it will suffice to show that $-1 \ge c_\alpha \log((\alpha-1)^{\alpha-1}/\alpha^\alpha)$, i.e., that

$$c_\alpha \cdot (\alpha \log \alpha - (\alpha-1)\log(\alpha-1)) \ge 1.$$

Numerical examination shows that this is true for $\alpha > 1.29$.

Proof of Lemma 5.3 We assume w.l.o.g. that $b = 1$. The inequality of Lemma 5.3 is equivalent to

$$A \cdot (2 + c_2 \log(a-1) - c_2 \log(a+1)) \le B \cdot (d_2 - 2 + c_2 \log(a+1)).$$

Since the righthand side is positive and $A \le 2B$, we need to prove

$$2 \cdot (2 + c_2 \log(a-1) - c_2 \log(a+1)) \;\le\; d_2 - 2 + c_2 \log(a+1).$$

This is the same as $6 - d_2 \le c_2 \log((a+1)^3/(a-1)^2)$. With $a > 1$ an integer, the quantity $(a+1)^3/(a-1)^2$ is minimized when $a = 5$. Thus, we must show that

$$d_2 \;\ge\; 6 - c_2 \log(6^3/4^2) \;=\; 6 - c_2(3\log 6 - 2\log 4).$$

In fact, d_2 was defined so that equality holds. Thus Lemma 5.3 holds.

5.3 Upper bound proof for 2-merge sort This section gives the proof of Theorem 5.4. The proof uses two functions G_2 and H_2 to bound the merge cost of runs stored on the stack.

Definition 5.2. *(For $\alpha = 2$.) Define*

$$G_2(n,m) \;=\; n \cdot (d_2 - 1 + c_2 \log m)$$

$$H_2(n,m) \;=\; \begin{cases} n \cdot (d_2 + c_2 \log(m-1)) & \text{if } m \ge 2 \\ 0 & \text{if } m = 1. \end{cases}$$

Recall that m_X is the number of original runs merged to form a run X. For the proof of Theorem 5.4, upper bounding the merge cost of the 2-merge sort, the idea is that for most runs X on the stack $\mathcal{Q}$, the merge cost of X will be bounded by $G_2(|X|, m_X)$. However, many of the runs formed by merges in cases (B) and (C) will instead have merge cost bounded by $H_2(|X|, m_X)$.

The next lemma, basically just a restatement of Lemma 5.1 for $\alpha = 2$, is the crucial property of G_2 that is needed for Theorem 5.4. The lemma is used to bound the merge costs incurred when merging two runs which differ in size by at most a factor 2. The constant c_2 is exactly what is needed to make this lemma hold.

Lemma 5.4. *Suppose n_1, n_2, m_1, m_2 are positive integers. Also suppose $n_1 \le 2n_2$ and $n_2 \le 2n_1$. Then,*

$$G_2(n_1,m_1) + G_2(n_2,m_2) + n_1 + n_2 \le G_2(n_1+n_2, m_1+m_2).$$

Proof. The inequality expresses that

$$n_1 \cdot (d_2 - 1 + c_2 \log m_1) + n_2 \cdot (d_2 - 1 + c_2 \log m_2) + n_1 + n_2 \le (n_1+n_2) \cdot (d_2 - 1 + c_2 \log(m_1+m_2)).$$

This is an immediate consequence of Lemma 5.1 with A, B, a, b replaced with n_1, n_2, m_1, m_2.

Lemma 5.5 states some further properties of G_2 and H_2 which follow from Lemmas 5.2 and 5.3.

LEMMA 5.5. *Suppose n_1, n_2, m_1, m_2 are positive integers.*

(a) *If $n_2 \le n_1$, then*

$$G_2(n_1, m_1) + H_2(n_2, m_2) + n_1 + n_2 \le H_2(n_1 + n_2, m_1 + m_2).$$

(b) *If $n_1 \le 2n_2$, then*

$$H_2(n_1, m_1) + n_1 + n_2 \le G_2(n_1 + n_2, m_1 + m_2).$$

Proof. If $m_2 \ge 2$, part (a) states that

$$n_1 \cdot (d_2 - 1 + c_2 \log m_1) + n_2 \cdot (d_2 + c_2 \log(m_2 - 1)) + n_1 + n_2 \le (n_1 + n_2) \cdot (d_2 + c_2 \log(m_2 + m_2 - 1)).$$

This is an immediate consequence of Lemma 5.2. If $m_2 = 1$, then part (a) states

$$n_1 \cdot (d_2 - 1 + c_2 \log m_1) + n_1 + n_2 \le (n_1 + n_2) \cdot (d_2 + c_2 \log m_1).$$

This holds since $d_2 \ge 1$ and $n_2 > 0$ and $m_1 \ge 1$.

When $m_1 \ge 2$, part (b) states that

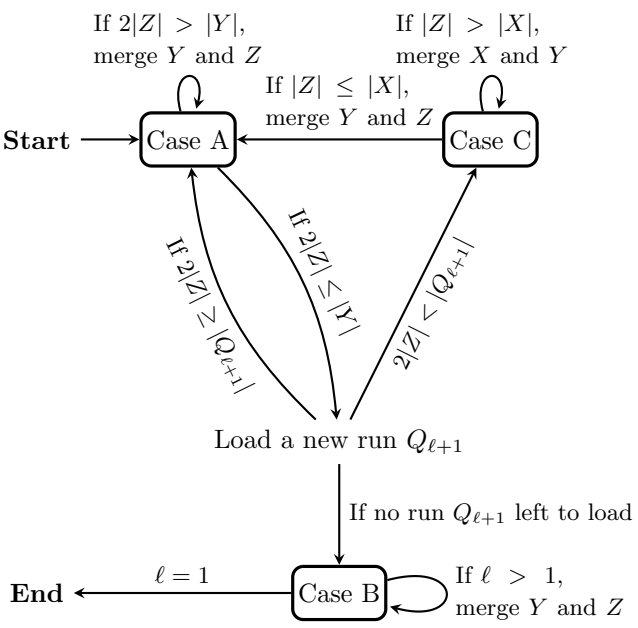
$$n_1 \cdot (d_2 + c_2 \log(m_1 - 1)) + n_1 + n_2 \le (n_1 + n_2) \cdot (d_2 - 1 + c_2 \log(m_1 + m_2));$$

this is exactly Lemma 5.3. When $m_1 = 1$, (b) states $n_1 + n_2 \le (n_1 + n_2)(d_2 - 1 + c_2 \log(m_2 + 1))$, and this is trivial since $c_2 + d_2 \ge 2$ and $m_2 \ge 1$.

We next prove Theorem 5.4. Recall the convention that the 2-merge sort maintains a stack $\mathcal{Q}$ containing runs $Q_1, Q_2, \ldots, Q_\ell$. The last four runs are denoted W, X, Y, Z. Each Q_i is a run of $|Q_i|$ many elements. Recall also that m_{Q_i} and w_{Q_i} denote the number of original runs that were merged to form Q_i and the merge cost of Q_i (respectively). If Q_i is an original run, then $m_{Q_i} = 1$ and $w_{Q_i} = 0$. If $m_{Q_i} = 2$, then Q_i was formed by a single merge, so $w_{Q_i} = |Q_i|$. To avoid handling the special cases for $\ell \le 2$, we adopt the convention that there is a virtual initial run Q_0 with infinite length, so $|Q_0| = \infty$.

LEMMA 5.6. *Suppose Q_i is a run in $\mathcal{Q}$ and that $w_{Q_i} \le G_2(|Q_i|, m_{Q_i})$. Then $w_{Q_i} \le H_2(|Q_i|, m_{Q_i})$.*

Proof. If $m_{Q_i} = 1$, the lemma holds since $w_X = 0$. If $m_{Q_i} = 2$, then it holds since $w_{Q_i} = |Q_i|$. If $m_{Q_i} > 2$, then it holds since $c_2 \log(m_{Q_i}/(m_{Q_i} - 1)) < 1$ and hence $G_2(|Q_i|, m_{Q_i}) < H_2(|Q_i|, m_{Q_i})$.



Proof of Theorem 5.4. We describe the 2-merge algorithm by using three invariants (A), (B), (C) for the stack; and analyzing what action is taken in each situation. Initially, the stack contains a single original Q_1, so $\ell = 1$ and $m_{Q_1} = 1$ and $w_{Q_1} = 0$, and case (A) applies.

(A): Normal mode. The stack satisfies

(A-1) $|Q_i| \ge 2 \cdot |Q_{i+1}|$ for all $i < \ell - 1$.
This includes $|X| \ge 2|Y|$ if $\ell \ge 2$.

(A-2) $2 \cdot |Y| \ge |Z|$; i.e. $2|Q_{\ell-1}| \ge |Q_\ell|$.

(A-3) $w_{Q_i} \le G_2(|Q_i|, m_{Q_i})$ for all $i \le \ell$.

If $\ell \ge 2$, (A-1) and (A-2) imply $|X| \ge |Z|$, i.e. $|Q_{\ell-2}| \ge |Q_\ell|$. The α-merge algorithm does one of the following:

- If $2|Z| \le |Y|$ and there are no more original runs to load, then it goes to case (B). We claim the four conditions of (B) hold (see below). The condition (B-2) holds by $|Y| \ge 2|Z|$, and (B-1) and (B-3) hold by (A-1) and (A-3). Condition (B-4) holds by (A-3) and Lemma 5.6.
- If $2|Z| \le |Y|$ and there is another original run to load, then the algorithm loads the next run as $Q_{\ell+1}$.
  - If $|Q_{\ell+1}| \le 2|Q_\ell|$, then we claim that case (A) still holds with ℓ incremented by one. In particular, (A-1) will hold since $|Y| \ge 2|Z|$ is the same as $2|Q_\ell| \le |Q_{\ell-1}|$. Condition (A-2) will hold by the assumed bound on $|Q_{\ell+1}|$. Condition (A-3) will still hold since $|Q_{\ell+1}|$ is an original run so $m_{Q_{\ell+1}} = 1$ and $w_{Q_{\ell+1}} = 0$.

- Otherwise $|Q_{\ell+1}| > 2|Q_\ell|$, and we claim that case (C) below holds with ℓ incremented by one. (C-1) and (C-4) will hold by (A-1) and (A-3). For (C-2), we need $|Q_{\ell-1}| \geq |Q_\ell|$; i.e. $|Y| \geq |Z|$: this follows trivially from $|Y| \geq 2|Z|$. (C-5) holds by (A-3) and Lemma 5.6. (C-3) holds since $2|Q_\ell| < |Q_{\ell+1}|$. (C-6) holds since $Q_{\ell+1}$ is an original run.

- If $2|Z| > |Y|$, then the algorithm merges the two runs Y and Z. We claim the resulting stack satisfies condition (A) with ℓ decremented by one. (A-1) clearly will still hold. For (A-2) to still hold, we need $2|X| \geq |Y| + |Z|$: this follows from $2|Y| \leq |X|$ and $|Z| \leq |X|$. (A-3) will clearly still hold for all $i < \ell - 1$. For $i = \ell - 1$, since merging Y and Z added $|Y| + |Z|$ to the merge cost, (A-3) implies that the new top stack element will have merge cost at most

$$G_2(|Y|, m_Y) + G_2(|Z|, m_Z) + |Y| + |Z|.$$

By (A-2) and Lemma 5.4, this is $\leq G_2(|Y| + |Z|, m_Y + m_Z)$, so (A-3) holds.

(B): Wrapup mode, lines 15-18 of Algorithm 6. There are no more original runs to process. The entire input has been combined into the runs $Q_1, \ldots, Q_\ell$ and they satisfy:

(B-1) $|Q_i| \geq 2 \cdot |Q_{i+1}|$ for all $i < \ell - 1$.
This includes $|X| \geq 2|Y|$ if $\ell \geq 2$.

(B-2) $|Y| \geq |Z|$; i.e., $|Q_{\ell-1}| \geq |Q_\ell|$.

(B-3) $w_{Q_i} \leq G_2(|Q_i|, m_{Q_i})$ for all $i \leq \ell - 1$,

(B-4) $w_Z \leq H_2(|Z|, m_Z)$.

If $\ell = 1$, the run $Z = Q_1$ contains the entire input in sorted order and the algorithm terminates. The total merge cost is $\leq H_2(|Z|, m_Z)$. This is $< n \cdot (d_2 + c_2 \log m)$ as needed for Theorem 5.4.

Otherwise $\ell > 1$, and Y and Z are merged.[6] We claim the resulting stack of runs satisfies case (B), now with ℓ decremented by one. It is obvious that (B-1) and (B-3) still hold. (B-4) will still hold since by (B-3) and (B-4) the merge cost of the run formed by merging Y and Z is at most $|Y| + |Z| + G_2(|Y|, m_Y) + H_2(|Z|, m_Z)$, and this is $\leq H_2(|Y| + |Z|, m_Y + m_Z)$ by Lemma 5.5(a). To show (B-2) still holds, we must show that $|X| \geq |Y| + |Z|$. To prove this, note that $\frac{1}{2}|X| \geq |Y|$ by (B-1); thus from (B-2), also $\frac{1}{2}|X| \geq |Z|$. Hence $|X| \geq |Y| + |Z|$.

[6] Note that in this case, if $\ell \geq 2$, $|Z| < |X|$, since (B-1) and (B-2) imply that $|X| \geq 2|Y| \geq 2|Z| > |Z|$. This is the reason why Algorithm 5 does not check for the condition $|X| < |Z|$ in lines 14-16 (unlike what is done on line 7).

(C): Encountered long run Z. When case (C) is first entered, the final run $|Z|$ is long relative to $|Y|$. The algorithm will repeatedly merge X and Y until $|Z| \leq |X|$, at which point it merges Y and Z and returns to case (A). (The merge of Y and Z must eventually occur by the convention that Q_0 has infinite length.) The following conditions hold with $\ell \geq 2$:

(C-1) $|Q_i| \geq 2|Q_{i+1}|$ for all $i < \ell - 2$.
If $\ell \geq 3$, this includes $|W| \geq 2|X|$.

(C-2) $|X| \geq |Y|$; i.e., $|Q_{\ell-2}| \geq |Q_{\ell-1}|$.

(C-3) $|Y| < 2|Z|$; i.e., $|Q_{\ell-1}| < 2|Q_\ell|$.

(C-4) $w_{Q_i} \leq G_2(|Q_i|, m_{Q_i})$ for all $i \leq \ell - 2$.

(C-5) $w_Y \leq H_2(|Y|, m_Y)$.

(C-6) $m_Z = 1$ and $w_Z = 0$, because Z is an original run and has not undergone a merge.

By (C-3), the test on line 6 of Algorithm 5 will now trigger a merge, either of Y and Z or of X and Y, depending on the relative sizes of X and Z. We handle separately the cases $|Z| > |X|$ and $|Z| \leq |X|$.

- Suppose $|Z| > |X|$. Then $\ell \geq 3$ and the algorithm merges X and Y. We claim that case (C) still holds, now with ℓ decremented by 1. It is obvious that (C-1), (C-4) and (C-6) still hold. (C-5) will still hold, since by (C-4) and (C-5), the merge cost of the run obtained by merging X and Y is at most $|X| + |Y| + G_2(|X|, m_X) + H_2(|Y|, m_Y)$, and this is $\leq H_2(|X| + |Y|, m_X + m_Y)$ by Lemma 5.5(a) since $|X| \geq |Y|$. To see that (C-2) still holds, we argue exactly as in case (B) to show that $|W| \geq |X| + |Y|$. To prove this, note that $\frac{1}{2}|W| \geq |X|$ by (C-1); thus from (C-2), $\frac{1}{2}W \geq |Y|$. Hence $|W| \geq |X| + |Y|$. To establish that (C-3) still holds, we must prove that $|X| + |Y| < 2|Z|$. By the assumption that $|Z| > |X|$, this follows from $|Y| \leq |X|$, which holds by (C-2).

- Otherwise, $|Z| \leq |X|$ and Y and Z are merged. We claim that now case (A) will hold. (A-1) will hold by (C-1). To show (A-2) will hold, we need $|Y| + |Z| \leq 2|X|$: this holds by (C-2) and $|Z| \leq |X|$. (A-3) will hold for $i < \ell - 1$ by (C-4). For $i = \ell - 1$, the merge cost w_{YZ} of the run obtained by merging Y and Z is $\leq H_2(|Y|, m_Y) + |Y| + |Z|$ by (C-5) and (C-6). By (C-3) and Lemma 5.5(b) this is $\leq G_2(|Y| + |Z|, m_Y + m_Z)$. Hence (A-3) will hold with $i = \ell - 1$.

That completes the proof of Theorem 5.4.

Examination of the above proof shows why the 2-merge Algorithm 5 does not need to test the condition

$|X| < 2|Y|$ on line 6; in contrast to what the α-merge Algorithm 6 does. In cases (A) and (B), the test will fail by conditions (A-1) and (B-1). In case (C), condition (C-3) gives $|Y| < 2|Z|$, so an additional test would be redundant.

The proof of Theorem 5.3 is omitted due to space reasons: its proof will be presented in the full version of this paper.

5.4 Proof of Theorem 5.5. Given Theorems 5.3 and 5.4, we can readily prove Theorem 5.5.

Proof of Theorem 5.5. Examination of the conditions (A-1), (B-1), (C-1) and (D-1) for the proofs of Theorems 5.3 and 5.4 shows that the stack $\mathcal{Q}$ satisfies the following properties, for some $\ell' \geq 0$: (a) $|Q_i| \geq \alpha|Q_{i+1}|$ for all $i < \ell'$; and (b) The size ℓ of the stack $\mathcal{Q}$ equals either $\ell'+1$ or $\ell'+2$. The total of the run lengths of $Q_1, \ldots, Q_{\ell'}$ is

$$\sum_{i=1}^{\ell'} |Q_i| \;\geq\; |Q_{\ell'}| \cdot \sum_{i=0}^{\ell'-1} \alpha^i \;=\; \frac{\alpha^{\ell'}-1}{\alpha-1} \cdot |Q'_\ell|$$

Therefore, the $\ell'+1$ runs $Q_1, \ldots, Q_{\ell'+1}$ have total length at least $\frac{\alpha^{\ell'}-1}{\alpha-1}+1$. This must be $\leq n$ if $\ell = \ell'+1$, and must be $\leq n-1$ if $\ell = \ell'+2$.

First suppose $\alpha < 2$. Since $\frac{\alpha^{\ell'}-1}{\alpha-1}+1 \leq n$, we have $\alpha^{\ell'} < n$. Thus $\ell' < \log_\alpha n$, and $\ell \leq \ell+2 < 1+\lceil \log_\alpha n \rceil$.

Now suppose $\alpha = 2$. If $\ell = \ell'+1$, then from $\frac{2^{\ell'}-1}{2-1}+1 = 2^{\ell'} \leq n$, we obtain $\ell' \leq \log n$, so $\ell \leq 1+\log n$. If $\ell = \ell'+2$, then $2^{\ell'} < n$ so $\ell = \ell'+2 < 2+\log n$. So in either case, $\ell \leq 1+\lceil \log n \rceil$.

6 Experimental results

This section reports some computer experiments comparing the α-stack sorts, the α-merge sorts, Timsort, and the Shivers sort. The test sequences use the following model. We only measure merge costs, so the inputs to the sorts are sequences of run lengths (not arrays to be sorted). Let μ be a distribution over integers. A sequence of m run lengths is chosen by choosing each of the m lengths independently according to the distribution μ. We consider two types of distributions:

1. The uniform distribution over numbers between 1 and 100,
2. A mixture of the uniform distribution over integers between 1 and 100 and the uniform distribution over integers between 10000 and 100000, with mixture weights 0.95 and 0.05. This distribution was specially tailored to work better with 3-aware algorithms.

We also experimented with power law distributions. However, they gave very similar results to the uniform distribution so we do not report these results here.

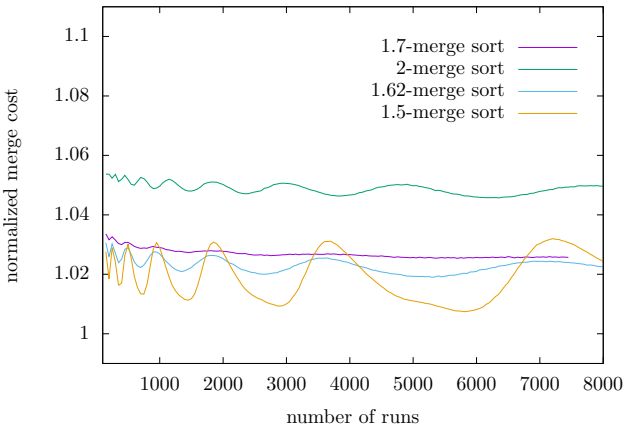


Figure 1: Comparison between α-merge sorts for different α on uniform distribution over integers between 1 and 100.

In Figures 1 and Figure 2, we estimate the "best" α values for the α-merge sort and the α-stack sort under the uniform distribution. The experiments show that the best value for α for both types of algorithms is around the golden ratio, or even slightly lower. For α at or below φ, the results start to show more and more oscillation. We do not know the reason for this oscillation.

Next we compared all the stable merge sorts discussed in the paper. Figure 3 reports on comparisons using the uniform distribution. It shows that the 1.62-merge sort performs slightly better than the 1.62-stack sort; and they perform better than Timsort, the 2-stack sort and the 2-merge sort. The Shivers sort performed comparably to the 1.62-stack sort and the 1.62-merge sort, but exhibited a great deal of oscillation in performance, presumably due to its use of rounding to powers of two.

Figure 4 considers the mixed distribution. Here the 1.62-merge sort performed best, slightly better than the 2-merge sort and Timsort. All three performed substantially better the 1.62-stack sort, the 2-stack sort, and the Shivers sort.

The figures show that in many cases the merge cost fluctuates periodically with the number of runs; this was true in many other test cases not reported here. We do not have a good explanation for this phenomenon.

7 Conclusion and open questions

Theorem 5.3 analyzed the α-merge sort only for $\alpha > \varphi$. This leaves several open questions:

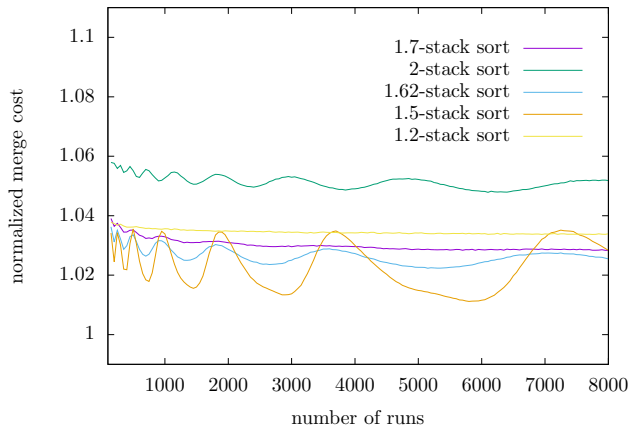


Figure 2: Comparison between α-stack sorts for different α on uniform distribution over integers between 1 and 100.

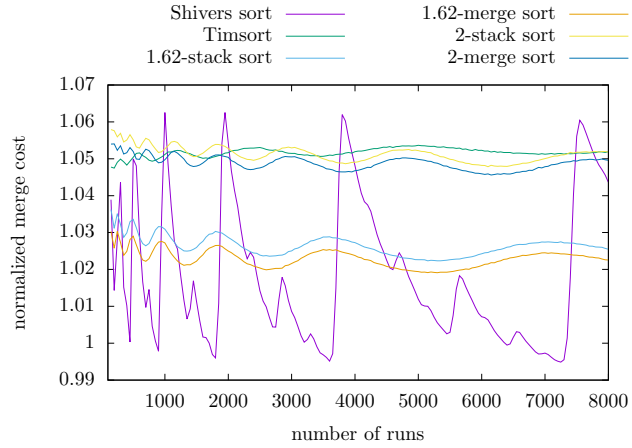


Figure 3: Comparison between sorting algorithms using the uniform distribution over integers between 1 and 100

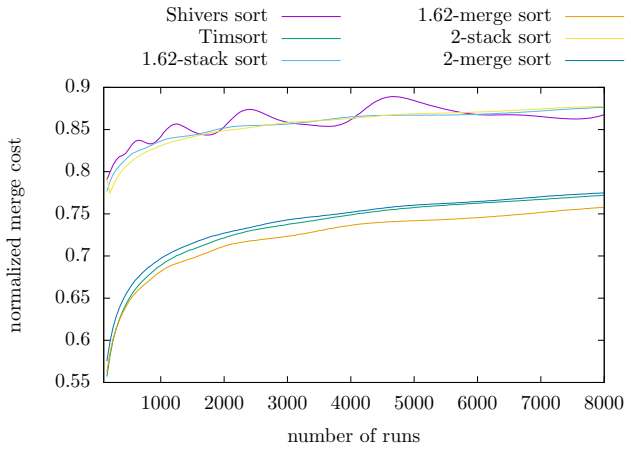


Figure 4: Comparison between sorting algorithms using a mixture of the uniform distribution over integers between 1 and 100 and the uniform distribution over integers between 10000 and 100000, with mixture weights 0.95 and 0.05

QUESTION 1. *For $\alpha \leq \varphi$, does the α-merge sort run in time $c_\alpha(1+\omega_m(1))n\log m$?*

It is likely that when $\alpha < \varphi$, the α-merge sort could be improved by making it 4-aware, or more generally, as $\alpha \to 1$, making it k-aware for even larger k's.

QUESTION 2. *Is it necessary that the constants $d_\alpha \to \infty$ as α approaches φ?*

An *augmented Shrivers sort* can defined by replacing the inner while loop on lines 6-8 with the code:

while $2^{\lfloor \log |Y| \rfloor} \leq |Z|$ **do**
  if $|Z| \leq |X|$ **then**
    Merge Y and Z
  else
    Merge X and Y
  end if
end while

The idea is to incorporate the 3-aware features of the 2-merge sort into the Shrivers sort method. The hope is that this might give an improved worst-case upper bound:

QUESTION 3. *Does the augmented Shrivers sort run in time $O(n\log m)$? Does it run in time $(1+o_m(1))n\log m$?*[7]

The notation $o_m(1)$ is intended to denote a value that tends to zero as $m \to \infty$.

[7] Jugé [15] has very recently answered this question in the affirmative for a different modification of the Shivers sort; see the next footnote.

We may define the "optimal stable merge cost" of an array A, denoted $\mathrm{Opt}(A)$, as the minimal merge cost of the array A over all possible stable merge sorts. It is not hard to see that there is a quadratic time m-aware optimal stable merge sort, which first determines all m run lengths and then works somewhat like dynamic programming. But for awareness less than m we know nothing.

QUESTION 4. *Is there a k-aware algorithm for $k = O(1)$ or even $k = O(\log m)$ such that for any array A this algorithm has merge cost $(1 + o_m(1))\mathrm{Opt}(A)$?*[8]

Our experiments used random distributions that probably do not do a very good job of modeling "real-world data." Is it possible to create better models for real-world data? Finally, it might be beneficial to run experiments with actual real-world data, e.g., by modifying deployed sorting algorithms to calculate statistics based on run lengths that arise in actual practice.

References

[1] N. AUGER, V. JUGÉ, C. NICAUD, AND C. PIVOTEAU, *On the worst-case complexity of TimSort*. arXiv:1805.08612, 2018.

[2] N. AUGER, C. NICAUD, AND C. PIVOTEAU, *Merge strategies: from merge sort to Timsort*. HAL technical report: hal-01212839, 2015.

[3] BARBAY AND NAVARRO, *On compressing permutations and adaptive sorting*, Theoretical Computer Science, 513 (2013), pp. 109–123.

[4] S. CARLSSON, C. LEVCOPOULOS, AND O. PETERSSON, *Sublinear merging and natural mergesort*, Algorithmica, 9 (1993), pp. 629–648.

[5] B. CHANDRAMOULI AND J. GOLDSTEIN, *Patience is a virtue: Revisiting merge and sort on modern processors*, in Proc. ACM SIGMOD Conference on Management of Data, 2014, pp. 731–742.

[6] C. R. COOK AND D. J. KIM, *Best sorting algorithm for nearly sorted lists*, Communications of the ACM, 23 (1980), pp. 620–624.

[7] S. DE GOUW, J. ROT, F. S. D. BOER, R. BUBEL, AND R. HÄHNLE, *OpenJDK's java.util.Collection.sort() is broken: The good, the bad, and the worst case*, in Proc. International Conference on Computer Aided Verification (CAV), Lecture Notes in Computer Science 9206, Springer, 2015, pp. 273–289.

[8] J. DEAN AND S. GHEMAWAT, *MapReduce: Simplified data processing on large clusters*, Communications of the ACM, 51 (2008), pp. 107–113.

[9] V. ESTIVILL-CASTRO AND D. WOOD, *A survey of adaptive sorting algorithms*, ACM Computing Surveys, 24 (1992), pp. 441–476.

[10] V. GEFFERT, J. KATAJAINEN, AND T. PASANEN, *Asyptoptotically efficient in-place merging*, Theoretical Computer Science, 237 (2000), pp. 159–181.

[11] H. HAMAMURA, J. MIYAO, AND S. WAKABAYASHI, *An optimal sorting algorithm for presorted sequences*, RIMS Kokyuroku, 754 (1991), pp. 247–256.

[12] J. D. HARRIS, *Sorting unsorted and partially sorted lists using the natural merge sort*, Software — Practice and Experience, 11 (1981), pp. 1339–1340.

[13] B.-C. HUANG AND M. A. LANGSTON, *Practical in-place merging*, Communications of the ACM, 31 (1988), pp. 348–352.

[14] F. K. HWANG AND S. LIN, *A simple algorithm for merging two disjoint linearly ordered sets*, SIAM J. on Computing, 1 (1972), pp. 31–39.

[15] V. JUGÉ, *Adaptive Shrivers sort: An alternative sorting algorithm*. Preprint, arXiv:1809.08411 [cs.DS], September 2018.

[16] D. E. KNUTH, *The Art of Computer Programming, Volume III: Sorting and Searching*, Addison-Wesley, second ed., 1998.

[17] M. A. KRONROD, *Optimal ordering algorithm without a field of operation*, Soviet Math. Dokl., 10 (1969), pp. 744–746. Russian original in *Dokl. Akad. Nauk SSSR* 186, 6 (1969) pp. 1256-1258.

[18] A. LAMARCA AND R. E. LADNER, *The influence of caches on the performance of sorting*, Journal of Algorithms, 31 (1999), pp. 66–104.

[19] C. LEVCOPOULOS AND O. PETERSSON, *Splitsort—an adaptive sorting algorithm*, Information Processing Letters, 39 (1991), pp. 205–211.

[20] ———, *Sorting shuffled monotone sequences*, Information and Computation, 112 (1994), pp. 37–50.

[21] H. MANNILA, *Measures of presorteness and optimal sorting algorithms*, IEEE Transactions on Computers, C-34 (1985), pp. 318–325.

[22] H. MANNILA AND E. UKKONEN, *A simple linear-time algorithm for in situ merging*, Information Processing Letters, 18 (1984), pp. 203–208.

[23] P. MCILROY, *Optimistic sorting and information theoretic complexity*, in Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1993, pp. 467–474.

[24] J. I. MUNRO AND S. WILD, *Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs*. arXiv:1805.04154, 2018.

[25] T. PETERS, *[Python-Dev] Sorting*. Python Developers Mailinglist. Retrieved from https://mail.python.org/pipermail/python-dev/2002-July/026837.html on July 5, 2017, July 2002.

[8] After the circulation of the first draft of this paper, Munro and Wild [24] gave an online 3-aware algorithm with asymptotically optimal stable merge cost. Their algorithm also uses the sum of run lengths processed, so it does not quite fit the framework of our question. Very recently, Jugé [15] gave a truly 3-aware algorithm also with asymptotically optimal stable merge cost. Surprisingly the optimal stable merge cost is the same as the entropy-based lower bounds of Barbay and Navarro [3] up to $O(n)$ additive component.

[26] O. Petersson and A. Moffat, *An overview of adaptive sorting*, Australian Computer Journal, 24 (1992), pp. 70–77.
[27] M. L. Rocca and D. Cantone, *NeatSort – a practical adaptive algorithm*. arXiv:1407:6183, July 2014.
[28] O. Shivers, *A simple and efficient natural merge sort*. Unpublished, presented at IFIP WG 2.8 Workshop on Function Programming, St. Malo, France. Retrieved from https://www.cs.tufts.edu/∼nr/cs257/archive/olin-shivers/msort.ps on July 25, 2017, 1999.
[29] S. S. Skiena, *Encroaching lists as a measure of presortedness*, BIT Numerical Mathematics, 28 (1988), pp. 755–784.
[30] A. Symvonis, *Optimal stable matching*, The Computer Journal, 38 (1995), pp. 681–690.
[31] H. Zhang, B. Meng, and Y. Liang, *Sort race*. arXiv:1609.04471, September 2015.