Galloping in fast-growth natural merge sorts

Elahe Ghasemi[3,4,5], Vincent Jugé[2,3], Ghazal Khalighinejad[1,5], and Helia Yazdanyar[5]

[1]Duke University, United States of America
[2]IRIF, Université Paris Cité & CNRS, France
[3]LIGM, Université Gustave Eiffel & CNRS, France
[4]Northeastern University, United States of America
[5]Sharif University of Technology, Iran

Abstract

We study the impact of merging routines in merge-based sorting algorithms. More precisely, we focus on the *galloping* routine that TimSort uses to merge monotonic sub-arrays, hereafter called *runs*, and on the impact on the number of element comparisons performed if one uses this routine instead of a naïve merging routine.

This routine was introduced in order to make TimSort more efficient on arrays with few distinct values. Alas, we prove that, although it makes TimSort sort array with two values in linear time, it does not prevent TimSort from requiring up to $\Theta(n \log(n))$ element comparisons to sort arrays of length n with three distinct values. However, we also prove that slightly modifying TimSort's galloping routine results in requiring only $\mathcal{O}(n + n \log(\sigma))$ element comparisons in the worst case, when sorting arrays of length n with σ distinct values.

We do so by focusing on the notion of *dual runs*, which was introduced in the 1990s, and on the associated *dual run-length entropy*. This notion is both related to the number of distinct values and to the number of runs in an array, which came with its own *run-length entropy* that was used to explain TimSort's otherwise "supernatural" efficiency. We also introduce new notions of *fast-* and *middle-growth* for natural merge sorts (i.e., algorithms based on merging runs), which are found in several sorting algorithms similar to TimSort.

We prove that algorithms with the fast- or middle-growth property, provided that they use our variant of TimSort's galloping routine for merging runs, are as efficient as possible at sorting arrays with low run-induced or dual-run-induced complexities.

1 Introduction

In 2002, Tim Peters, a software engineer, created a new sorting algorithm, which was called TimSort [24] and was built on ideas from McIlroy [21]. This algorithm immediately demonstrated its efficiency for sorting actual data, and was adopted as the standard sorting algorithm in core libraries of widespread programming languages such as Python and Java. Hence, the prominence of such a custom-made algorithm over previously preferred worst-case optimal algorithms contributed to the regain of interest in the study of sorting algorithms.

Among the best-identified reasons behind the success of TimSort lies the fact that this algorithm is well adapted to the architecture of computers (e.g., for dealing with cache issues) and to realistic distributions of data. In particular, the very conception of TimSort makes it particularly well-suited to sorting data whose *run decompositions* [3, 10] (see Figure 1) are simple. Such decompositions were already used in 1973 by Knuth's NaturalMergeSort [18, Section 5.2.4], which adapted the traditional MergeSort algorithm as follows: NaturalMergeSort is based on splitting arrays into monotonic (non-decreasing or decreasing) sub-arrays[1], also called *runs*, and on merging these runs together. All algorithms sharing this feature of NaturalMergeSort are also called *natural* merge sorts.

In addition to being a natural merge sort, TimSort includes many optimisations, which were carefully engineered, through extensive testing, to offer the best complexity performances. As a result, the general structure

[1]Early works such as [18] considered increasing sub-arrays only.

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

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

of TimSort can be split into three main components: (i) a variant of an insertion sort, 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 routine for merging these runs, based on a so-called *galloping* strategy.

The second component has been subject to an intense scrutiny these last few years, thereby giving birth to a great variety of TimSort-like algorithms, such as α-StackSort [2], α-MergeSort [8], ShiversSort [27] (which *predated* TimSort), adaptive ShiversSort [16], PeekSort and PowerSort [23], or k-way PowerSort [13], which extends PowerSort by merging runs in batches of k instead of two-by-two. On the contrary, the first and third components, which seem more complicated and whose effect might be harder to quantify, have often been used as black boxes when studying TimSort or designing variants thereof.

In what follows, we focus on the third component and prove that it can be made very efficient: although TimSort may require up to $\Theta(n \log(n))$ comparisons to sort arrays of length n with three distinct values, slight modifications to the galloping routine make TimSort require only $\mathcal{O}(n + n \log(\sigma))$ comparisons to sort arrays of length n with σ distinct values. This is reminiscent of the celebrated complexity result [1] stating that TimSort requires $\mathcal{O}(n + n \log(\rho))$ comparisons to sort arrays of length n that can be decomposed as a concatenation of ρ monotonic sub-arrays.

Complexity measures. Consider an array A of length n, containing data from a linearly pre-ordered set $(\mathcal{S}, \preccurlyeq)$ that we want to sort in a *stable* manner: we want to transform A into the array B whose entries are given by the relation $B[\pi(i)] = A[i]$, where π is the permutation of $\{1, 2, \ldots, n\}$ such that $\pi(i) \leqslant \pi(j)$ if and only if $A[i] \prec A[j]$ or ($A[i] \preccurlyeq A[j] \preccurlyeq A[i]$ and $i \leqslant j$). In a comparison model, all the queries about A that we form amount to checking, for a given pair (i, j) of integers, whether $\pi(i) \leqslant \pi(j)$; hence, two arrays A and A' that lead to the same permutation π (we say that they are *lexicographically equivalent*) must yield the same element moves or swaps, and they should be considered equivalent with each other. Consequently, below, we identify A with the permutation π, and the values $A[1], A[2], \ldots, A[n]$ with the integers from 1 to n.

A natural parameter that captures the complexity of an array is its length. Every comparison-based sorting algorithm requires $n \log_2(n) + \mathcal{O}(n)$ element comparisons in the worst case, a complexity bound achieved by MergeSort. Yet, not all arrays of length n are as difficult to sort as each other: in the extreme case A is already the identity permutation, it suffices to check that A is sorted. This suggests using finer-grained complexity classes, as follows.

A usual measure of presortedness relies on subdividing A into distinct monotonic *runs*, i.e., partitioning the set $\{1, 2, \ldots, n\}$ into intervals $R_1, R_2, \ldots, R_\rho$ on which the function $x \mapsto A[x]$ is monotonic, and in studying the lengths of these runs. Although this partition is *a priori* not unique, there exists a simple greedy algorithm that provides us with a partition for which ρ is minimal: it suffices to construct the intervals $R_1, R_2, \ldots, R_\rho$ from left to right, each time choosing R_i to be as long as possible.

Thus, one may consider only arrays whose run decomposition consists of ρ monotonic runs. On such arrays, the best worst-case time complexity one may hope for is $\mathcal{O}(n + n \log(\rho))$ [20]. We may also consider more restricted classes of input arrays, focusing only on those arrays that consist of ρ runs of lengths $r_1, \ldots, r_\rho$. In that case, every comparison-based sorting algorithm requires at least $n\mathcal{H} + \mathcal{O}(n)$ element comparisons, 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 [3, 16, 21]. The number $\mathcal{H}$ is called the *run-length entropy* of the array.

A dual approach, proposed by McIlroy [21], yields a different parametrisation. After identifying A with the permutation it is lexicographically equivalent to, we partition the set $\{1, 2, \ldots, n\}$ into the monotonic runs $S_1, S_2, \ldots, S_\sigma$ of the inverse permutation A^{-1}. These intervals S_i are already known under the name of *riffle shuffles* [21]; they form a special case of *monotone sequences* described in [19], with the additional condition of

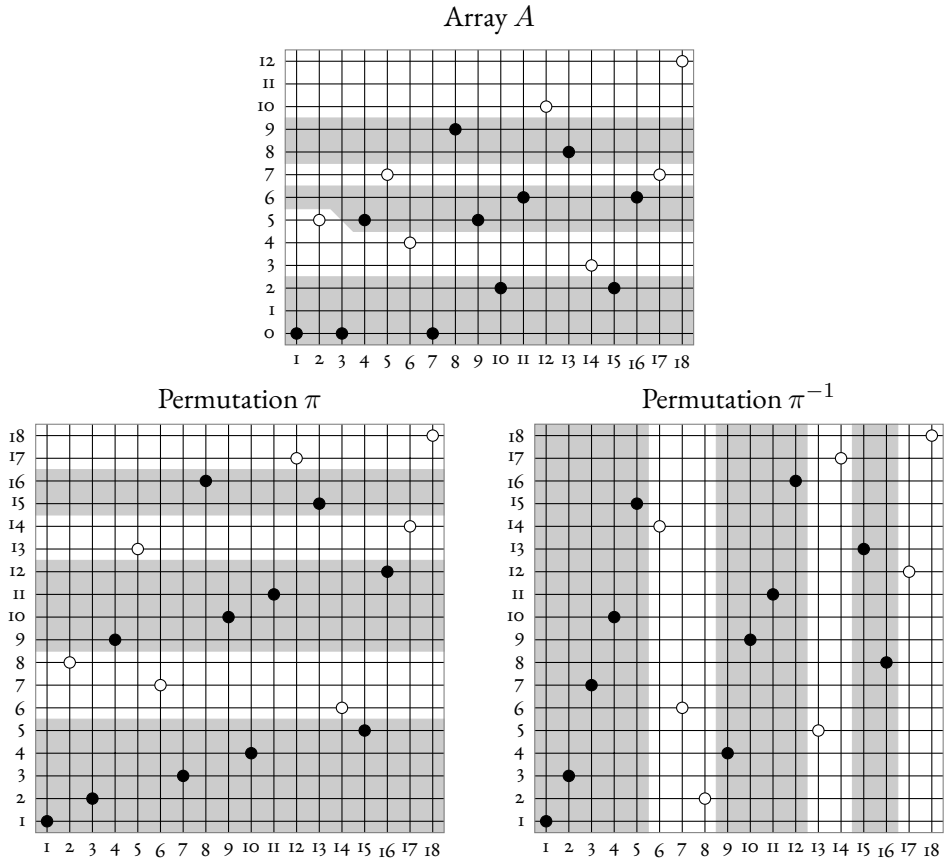

Figure 2: The array A is lexicographically equivalent to the permutation π. Its dual runs, represented with gray and white horizontal stripes, have respective lengths 5, 3, 4, 2, 2 and 2. The mapping $A \mapsto \pi$ identifies them with the dual runs of π, i.e., with the runs of the permutation π^{-1}. Note that, although they are on the same horizontal line in the diagrammatic representation of A, the points with coordinates $(2, 5)$ and $(4, 5)$ belong to distinct dual runs.

being non-overlapping. In order to underline their connection with runs, we say that these intervals are the *dual runs* of A, and their lengths are denoted by s_i. The process of transforming an array into a permutation and then extracting its dual runs is illustrated in Figure 2.

The lower bounds of [3, 16, 21] immediately translate into matching lower bounds: every comparison-based sorting algorithm requires at least $n\mathcal{H}^* + \mathcal{O}(n)$ element comparisons, where $\mathcal{H}^*$ is defined as $\mathcal{H}^* = H(s_1/n, \ldots, s_\sigma/n)$. The number $\mathcal{H}^*$ is called the *dual run-length entropy* of the array.

In particular, if an array has σ values, it cannot have more than σ dual runs. Note, however, that it may have significantly fewer than σ dual runs, as shown by the examples of the monotonic permutations, which have n values but only one dual run. In addition, in general, there is no non-trivial connection between the runs of a permutation and its dual runs. For instance, a permutation with a given number of runs may have arbitrarily many (or few) dual runs, and conversely.

Related work. The success of TimSort has nurtured the interest in the quest for sorting algorithms that would be both excellent all-around and adapted to arrays with few runs. However, its *ad hoc* conception made its complexity analysis harder than what one might have hoped, and it is only in 2015, a decade after TimSort had been largely deployed, that Auger et al. [2] proved that TimSort required $\mathcal{O}(n \log(n))$ comparisons for sorting arrays of length n, which is worst-case optimal in the model of sorting by comparisons.

Since the early 2000s, several natural merge sorts were proposed, all of which were meant to offer easy-to-prove complexity guarantees when sorting arrays of length n, with ρ runs and run-length entropy $\mathcal{H}$: ShiversSort and α-

StackSort, which run in time $\mathcal{O}(n \log(n))$; α-MergeSort, which, like TimSort, runs in time $\mathcal{O}(n + n\mathcal{H})$; adaptive ShiversSort, PeekSort and (k-way) PowerSort, which require only $n\mathcal{H} + \mathcal{O}(n)$ comparisons and element moves. In fact, the latter complexity guaranteed was already implicit in the seminal works of Hu and Tucker [15] and Garsia and Wachs [12] about optimal alphabetic Huffman codes, in the 1970s.

Except TimSort, these algorithms are, in fact, described only as policies for deciding which runs to merge, the actual routine used for merging runs being left implicit: since choosing a naïve merging routine does not harm the worst-case time complexities considered above, all authors identified the cost of merging two runs of lengths m and n with the sum $m+n$, and the complexity of the algorithm with the sum of the costs of the merges performed.

One notable exception is that of Munro and Wild [23]. They compared the running times of TimSort and of TimSort's variant obtained by using a naïve merging routine instead of TimSort's galloping routine. However, and although they mentioned the challenge of finding distributions on arrays that might benefit from galloping, they did not address this challenge, and focused only on arrays with a low entropy $\mathcal{H}$. As a result, they unsurprisingly observed that the galloping routine was slower than the naïve one.

A parallel line of research has been focused on galloping and its impact on merge-based algorithms in general. As early as 1976, Munro and Spira [22] proposed a complexity measure $\mathcal{H}^\circ$ related to the dual run-length entropy $\mathcal{H}^*$, with the property that $\mathcal{H}^* \leqslant \mathcal{H}^\circ \leqslant \log_2(\sigma)$ for arrays with σ values. They also proposed an algorithm for sorting arrays of length n with σ values by using $\mathcal{O}(n + n\mathcal{H}^\circ)$ comparisons. McIlroy [21] then extended their work by using the complexity measure $\mathcal{H}^*$ instead of $\mathcal{H}^\circ$, and proposing a variant of Munro and Spira's algorithm that would use $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons. Similarly, Barbay et al. [4] invented the algorithm QuickSynergySort, which aimed at minimising the number of comparisons, achieving a $\mathcal{O}(n + n\mathcal{H}^*)$ upper bound and further refining the parameters it used, by taking into account the interleaving between runs and dual runs.

In particular, Carlsson et al. [9] proved in 1993 that using a galloping routine, which they called AdaptMerge and is equivalent to the 0-galloping routine introduced below, allowed every natural merge sort to require only $\mathcal{O}(n \log(\rho))$ element comparisons. This upper bound was refined by Schou and Wang [26], who proved that only $\mathcal{O}(n + n\mathcal{H})$ element comparisons were actually required. They also proposed the new algorithm PersiSort, which uses only $\mathcal{O}(n + n\mathcal{H})$ comparisons without computing run lengths or attempting to identify well-adapted run merge strategies. Yet, all of these algorithms require $\omega(n + n\mathcal{H})$ element moves in the worst case.

Furthermore, as a side effect of being rather complicated and lacking a proper analysis, except that of [23] that hinted at its inefficiency, TimSort's galloping routine has been omitted in various mainstream implementations of natural merge sorts, in which it was replaced by its naïve variant. This is the case, for instance, in library TimSort implementations of the programming languages Swift [11] and Rust [25]. On the contrary, TimSort implementations in other languages, such as Java [7], Octave [29] or the V8 JavaScript engine [30], and PowerSort implementation in Python [28] include the galloping routine.

Outline. In this article, we study the impact of integrating galloping routines in natural merge sort algorithms.

In Section 2, we present the galloping routine of TimSort and two of its variants: the $\mathbf{t}$-*galloping* routine, which was already introduced in [14, 21], and the *polylogarithmic* routine, obtained by using the $\mathbf{t}$-galloping routine with $\mathbf{t} = \Theta(\log_2(a+b)^2)$ whenever we merge runs of lengths a and b. The latter routine is an improvement over the logarithmic routine of [14], because it will satisfy Proposition 49, which means, in a nutshell, that integrating the polylogarithmic routine instead of a naïve merging routine in a reasonable merge sort algorithm is risk-free: it requires at most $\mathcal{O}(n)$ additional element comparisons, and no additional element move.

In Section 3, we present three classes of natural merge sort algorithms, already proposed in [14]: algorithms with the *fast-growth*, *middle-growth* and *tight middle-growth* properties, the middle-growth property being strictly weaker than both other properties. Having these properties means that, up to a constant factor, merging runs should increase their sizes exponentially fast (this is the fast-growth property), possibly when restricting oneself to start with a run present in the initial array (middle-growth) or demanding that the growth rate should be maximal, i.e., 2 (tight middle-growth). This interpretation of these properties and the implications between them are represented in Figure 3.

We also prove (Theorem 7) that integrating the $\mathbf{t}$-galloping routine into algorithms with the middle-growth property makes them require only $\mathcal{O}(n + n\mathcal{H}^*)$ element comparisons; this improves the similar statement

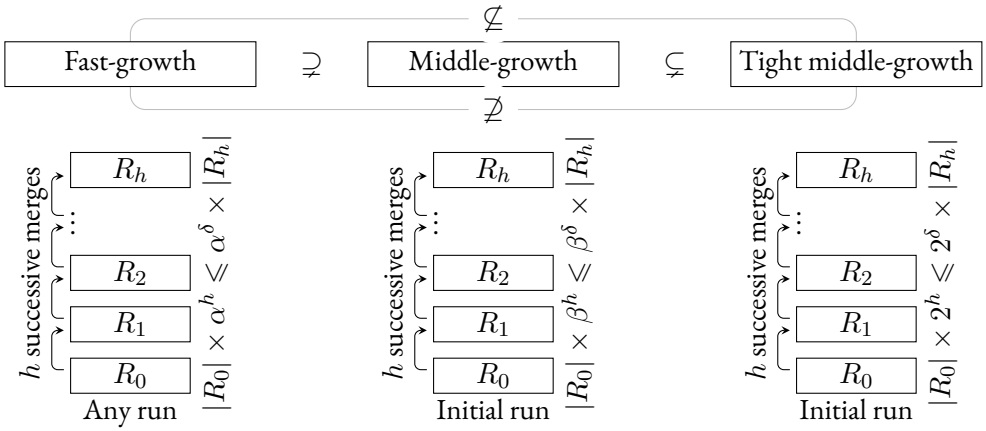

Figure 3: Growth properties: each algorithm with the fast-growth or tight middle-growth property also has the middle-growth property; other implications are invalid. When merging h times a run R_0 gives a run R_h, inequalities of the form $\gamma^h|R_0| \leqslant \gamma^\delta|R_h|$ (for constants $\gamma > 1$ and $\delta \in \mathbb{R}$) must be valid (i) for all runs R_0, when one has the fast-growth property; (ii) for all initial runs R_0, when one has the middle-growth property; (iii) for all initial runs R_0 and $\gamma = 2$, when one has the tight middle-growth property.

of [14, Theorem 8], where we used a different notion of dual run-length property, based on splitting the inverse permutation A^{-1} into increasing runs only, instead of monotonic runs.

In Section 4, we prove that all the algorithms TimSort, α-MergeSort, PowerSort, PeekSort and, adaptive ShiversSort have the fast-growth property (Theorem 8); that NaturalMergeSort, ShiversSort and α-StackSort have the middle-growth property (Theorem 9); and that NaturalMergeSort, ShiversSort and PowerSort have the tight middle-growth property (Theorem 10). This allows recovering complexity results that were proved separately for each of these algorithms.

In Section 5, we prove that TimSort's original galloping routine is flawed, and fails to enjoy the complexity bounds provided in Theorem 7 or the risk-freeness property of Proposition 49: using this routine on arrays with $\sigma = 3$ distinct values may still require $\Omega(n \log(n))$ element comparisons, and using it instead of the naïve routine may require $\Omega(n \log(n))$ additional element comparisons. This legitimates using our variants instead of this original routine.

Finally, in Section 6, we prove that integrating the $\mathbf{t}$-galloping routine (Theorems 46 — already proved in [14] — 53 and 59) in algorithms with the tight middle-growth property, or in Adaptive ShiversSort or PeekSort, yields no more than

$$(1 + 1/(\mathbf{t} + 3))n\mathcal{H}^* + \log_2(\mathbf{t} + 1)n + \mathcal{O}(n)$$

element comparisons, whereas integrating the polylogarithmic routine (Theorems 51, 53 and 60) yields no more than

$$n\mathcal{H}^* + 2\log_2(\mathcal{H}^* + 1)n + \mathcal{O}(n)$$

element comparisons.

These results provide us with the a more complete view on the complexity of using various algorithms for sorting arrays of length n, with ρ runs and σ different values, run-length complexity $\mathcal{H}$ and dual run-length complexity $\mathcal{H}^*$, which is summarised in Table 1. For each algorithm, we indicate the worst-case merge cost, which is approximately equal to the number of element moves, and to the number of element comparisons if a naïve merging routine is used; we also indicate the worst-case number of element comparisons if a polylogarithmic galloping routine is used. Furthermore, our results often suffice to obtain $\mathcal{O}$ variants of already-known results; in those cases, we indicate both the best known bounds and those that our results yield. All results indicated in Table 1 are given up to a $\mathcal{O}(n)$ error term.

Algorithm		Merge cost		Element comparisons	
NaturalMergeSort	[18]	$n\log_2(\rho)$ $n\log_2(n)$	[18] Thms 6 & 10	$n(\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1))$	Thms 10 & 51
TimSort	[24]	$1.5n\mathcal{H}$ $\mathcal{O}(n\mathcal{H})$	[1, 8] Thms 5 & 8	$\mathcal{O}(n\mathcal{H}^*)$	Thms 7 & 8
α-StackSort	[2]	$\mathcal{O}(n\log(n))$	[2] Thms 6 & 9	$\mathcal{O}(n\mathcal{H}^*)$	Thms 7 & 9
α-MergeSort	[8]	$n\mathcal{H}/c_\alpha$ when $\phi < \alpha < 2$ $\mathcal{O}(n\mathcal{H})$	[8] Thms 5 & 8	$\mathcal{O}(n\mathcal{H}^*)$	Thms 7 & 8
ShiversSort	[27]	$n\log_2(n)$	[27] Thms 6 & 10	$n(\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1))$	Thms 10 & 51
adaptive ShiversSort	[16]	$n\mathcal{H}$ $\mathcal{O}(n\mathcal{H})$	[16] Thms 5 & 8	$n(\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1))$	Thm 53
PeekSort	[23]	$n\mathcal{H}$ $\mathcal{O}(n\mathcal{H})$	[23] Thms 5 & 8	$n(\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1))$	Thm 60
PowerSort	[23]	$n\mathcal{H}$ $\mathcal{O}(n\mathcal{H})$ $n\log_2(n)$	[23] Thms 5 & 8 Thms 6 & 10	$n(\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1))$	Thms 10 & 51

Table 1: Upper bounds on the merge costs and number of comparisons required by various natural merge sorts when using the polylogarithmic galloping routine. Here, we set $c_\alpha = \log_2(\alpha+1) - \alpha\log_2(\alpha)/(\alpha+1)$.

2 The galloping routine for merging runs

Here, we describe the galloping routine that TimSort uses to merge adjacent non-decreasing runs. This routine is a blend between a naïve merging algorithm, which requires $a+b-1$ comparisons to merge runs A and B of lengths a and b, and a binary-search-based algorithm, which requires $\mathcal{O}(\log(a+b))$ comparisons in the best case, and $\mathcal{O}(a+b)$ comparisons in the worst case. It depends on a parameter $\mathbf{t}$, and works as follows.

When merging runs A and B into one large run C, we first need to find the least integers k and ℓ such that $B[0] < A[k] \leqslant B[\ell]$: the $k+\ell$ first elements of C are

$$A[0], A[1], \ldots, A[k-1], B[0], B[1], \ldots, B[\ell-1],$$

and the remaining elements of C are obtained by merging the sub-array of A that spans positions k to a and the sub-array of B that spans positions ℓ to b. Computing k and ℓ efficiently is therefore a crucial step towards reducing the number of comparisons required by the merging routine (and, thus, by the sorting algorithm).

This computation is a special case of the following problem: if one wishes to find a secret integer $m \geqslant 1$ by choosing integers $x \geqslant 1$ and testing whether $x \geqslant m$, what is, as a function of m, the least number of tests that one must perform? Bentley and Yao [6] answer this question by providing simple strategies, which they number $\mathsf{B}_0, \mathsf{B}_1, \ldots$:

- B_0: choose $x = 1$, then $x = 2$, and so on, until choosing $x = m$, thereby finding m in m queries.
- B_1: first use B_0 to find $\lceil\log_2(m)\rceil + 1$ in $\lceil\log_2(m)\rceil + 1$ queries, i.e., choose $x = 2^k$ until $x \geqslant m$, then compute the bits of m (from the most significant bit of m to the least significant one) in $\lceil\log_2(m)\rceil - 1$ additional queries. For instance, if $m = 5$, this results in successively choosing $x = 1, 2, 4, 8, 6$ and 5; if $m = 8$, this results in successively choosing $x = 1, 2, 4, 8, 6$ and 7. Bentley and Yao call this strategy a *galloping* (or *exponential search*) technique.
- B_{k+1}: like B_1, except that one finds $\lceil\log_2(m)\rceil + 1$ by using B_k instead of B_0.

Strategy B_0 uses m queries, B_1 uses $2\lceil\log_2(m)\rceil$ queries (except for $m = 1$, where it uses one query), and each strategy B_k with $k \geqslant 2$ uses $\log_2(m) + o(\log(m))$ queries. Thus, if m is known to be arbitrarily large,

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
B_0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
B_1	1	2	**4**	4	**6**	6	6	6	8	8	8	8	8	8	8	8	10

Table 2: Comparisons requested by strategies B_0 and B_1 to find a secret integer $m \geqslant 1$.

one should favour some strategy B_k (with $k \geqslant 1$) over the naïve strategy B_0. However, when merging runs taken from a permutation chosen uniformly at random over the $n!$ permutations of $\{1, 2, \ldots, n\}$, the integer m is frequently small, which makes B_0 suddenly more attractive. In particular, the overhead of using B_1 instead of B_0 is a prohibitive $+20\%$ or $+33\%$ when $m = 5$ or $m = 3$, as illustrated in the black cells of Table 2.

McIlroy [21] addresses this issue by choosing a parameter $\mathbf{t}$ and using a blend between the strategies B_0 and B_1, which consists in two successive steps C_1 and C_2:

- C_1: first follow B_0 for up to $\mathbf{t}$ steps, thereby choosing $x = 1, x = 2, \ldots, x = \mathbf{t}$ (if $m \leqslant \mathbf{t} - 1$, one stops after choosing $x = m$).
- C_2: if $m \geqslant \mathbf{t} + 1$, switch to B_1 (or, more precisely, to a version of B_1 translated by $\mathbf{t}$, since the precondition $m \geqslant 1$ is now $m \geqslant \mathbf{t} + 1$).

Once such a parameter $\mathbf{t}$ is fixed, McIlroy's mixed strategy allows retrieving m in $\mathsf{cost}_{\mathbf{t}}(m)$ queries, where $\mathsf{cost}_{\mathbf{t}}(m) = m$ if $m \leqslant \mathbf{t} + 2$, and $\mathsf{cost}_{\mathbf{t}}(m) = \mathbf{t} + 2\lceil \log_2(m - \mathbf{t}) \rceil$ if $m \geqslant \mathbf{t} + 3$.

In practice, however, the integer we are evaluating is not only positive, but even subject to the double inequality $1 \leqslant k \leqslant a$ or $1 \leqslant \ell \leqslant b$; above, we overlooked those upper bounds. Taking them into account allows us to marginally improve strategy B_1 and McIlroy's resulting mixed strategy, at the expense of providing us with a more complicated cost function; in TimSort's implementation, this improvement is used only when $k \geqslant \max\{\mathbf{t}, a/2\}$ or $\ell \geqslant \max\{\mathbf{t}, b/2\}$.

Therefore, and in order to keep things simple, we will replace the precise cost functions we might have obtained by the following simpler upper bound. By contrast, whenever constructing examples aimed at providing lower bounds, we will make sure that the cases $k \geqslant \max\{\mathbf{t}, a/2\}$ and $\ell \geqslant \max\{\mathbf{t}, b/2\}$ never occur.

Lemma 1. *For all $\mathbf{t} \geqslant 0$ and $m \geqslant 1$, we have $\mathsf{cost}_{\mathbf{t}}(m) \leqslant \mathsf{cost}^*_{\mathbf{t}}(m)$, where*

$$\mathsf{cost}^*_{\mathbf{t}}(m) = \min\{(1 + 1/(\mathbf{t} + 3))m, \mathbf{t} + 2 + 2\log_2(m + 1)\}.$$

Proof. Since the desired inequality is immediate when $m \leqslant \mathbf{t} + 2$, we assume that $m \geqslant \mathbf{t} + 3$. In that case, we already have $\mathsf{cost}_{\mathbf{t}}(m) \leqslant \mathbf{t} + 2(\log_2(m - \mathbf{t}) + 1) \leqslant \mathbf{t} + 2 + 2\log_2(m + 1)$, and we prove now that $\mathsf{cost}_{\mathbf{t}}(m) \leqslant m + 1$.

Let $u = m - \mathbf{t}$ and let $f : x \mapsto x - 1 - 2\log_2(x)$. We first check by hand that $(m+1) - \mathsf{cost}_{\mathbf{t}}(m) = 0, 1, 0, 1$ when $u = 3, 4, 5, 6$. Then, observing that f is positive and increasing on the interval $[7, +\infty)$. proves that $(m + 1) - \mathsf{cost}_{\mathbf{t}}(m) \geqslant f(u) > 0$ when $u \geqslant 7$. It follows, as expected, that $\mathsf{cost}_{\mathbf{t}}(m) \leqslant m + 1 \leqslant (1 + 1/(\mathbf{t} + 3))m$. □

The above discussion immediately provides us with a cost model for the number of comparisons performed when merging two runs.

Proposition 2. *Let π be a permutation with dual runs $S_1, S_2, \ldots, S_\sigma$, and let A and B be two monotonic runs of lengths a and b obtained while sorting π with a natural merge sort. For each integer $i \leqslant \sigma$, let $a_{\to i}$ (respectively, $b_{\to i}$) be the number of elements in A (respectively, in B) whose value belongs to S_i. Using a merging routine based on McIlroy's mixed strategy for a fixed parameter $\mathbf{t}$, we need at most*

$$1 + \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\mathbf{t}}(a_{\to i}) + \mathsf{cost}^*_{\mathbf{t}}(b_{\to i})$$

element comparisons to merge the runs A and B into an increasing run.

Proof. Let C be the increasing run that results from merging A and B. A preliminary step consists in checking whether each of A and B is increasing or decreasing, in which case they are reversed. This only requires comparing the first two elements of A and B.

If S_1 is an increasing run of π^{-1}, the run C starts with those $a_{\to 1}$ elements from A whose value belongs to S_1, and then those $b_{\to 1}$ elements from B whose value belongs to S_1; otherwise, C starts with those $b_{\to 1}$ elements from B whose value belongs to S_1, and only then those $a_{\to 1}$ elements from A whose value belongs to S_1. In both cases, C then consists of $a_{\to 2} + b_{\to 2}$ elements whose value belongs to S_2 (starting with elements of A if S_2 is an increasing run of π^{-1}, and of B otherwise), and so on.

Hence, the elements of A are grouped in blocks of length $a_{\to i}$, some of which will form consecutive blocks of elements of C, thereby being "glued" together. Similarly, the elements of B are grouped in blocks of length $b_{\to i}$, some being glued together. The run C will then consist in an alternation of (possibly, glued) blocks from A and B, and the galloping routine consists in discovering whether the first block comes from A or from B (which takes one query) and then successively computing the lengths of these blocks, except the last one (because once a run has been completely integrated into C, the remaining elements of the other run will necessarily form one unique block).

Since $\mathsf{cost}^*_{\mathbf{t}}$ is sub-additive, i.e., since $\mathsf{cost}^*_{\mathbf{t}}(m) + \mathsf{cost}^*_{\mathbf{t}}(m') \geqslant \mathsf{cost}^*_{\mathbf{t}}(m+m')$ for all $m \geqslant 0$ and $m' \geqslant 0$, discovering the length of a glued block of length $a_{\to i} + a_{\to i+1} + \cdots + a_{\to j}$ requires no more than

$$\mathsf{cost}^*_{\mathbf{t}}(a_{\to i} + a_{\to i+1} + \cdots + a_{\to j}) \leqslant \mathsf{cost}^*_{\mathbf{t}}(a_{\to i}) + \mathsf{cost}^*_{\mathbf{t}}(a_{\to i+1}) + \cdots + \mathsf{cost}^*_{\mathbf{t}}(a_{\to j})$$

element comparisons. Using this upper bound to count comparisons of elements taken from all blocks (glued or not) that belong to either A or B completes the proof. □

We simply call $\mathbf{t}$-*galloping routine* the merging routine based on McIlroy's mixed strategy for a fixed parameter $\mathbf{t}$; when the value of $\mathbf{t}$ is irrelevant, we omit mentioning it. Then, the quantity

$$1 + \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\mathbf{t}}(a_i) + \mathsf{cost}^*_{\mathbf{t}}(b_i)$$

is called the ($\mathbf{t}$-)*galloping cost* of merging A and B. This cost never exceeds $1 + 1/(\mathbf{t}+3)$ times the sum $a+b$, which we call *naïve cost* of merging A and B. Below, we study the impact of using the galloping routine instead of the naïve one, which amounts to replacing naïve merge costs by their galloping variants.

Note that using this new galloping cost measure is relevant only if element comparisons are significantly more expensive than element (or pointer) moves. For example, even if we were lucky enough to observe that each element in B is smaller than each element in A, we would perform only $\mathcal{O}(\log(a+b))$ element comparisons, but as many as $\Theta(a+b)$ element moves.

Updating the parameter $\mathbf{t}$. We assumed above that the parameter $\mathbf{t}$ did not vary while the runs A and B were being merged with each other. This is not how $\mathbf{t}$ behaves in TimSort's implementation of the galloping routine.

Instead, the parameter $\mathbf{t}$ is initially set to a constant ($\mathbf{t} = 7$ in Java), and may change during the algorithm, by proceeding roughly as follows. In step C_2, after using the strategy B_1, and depending on the value of m that we found, one may realise that using B_0 might have been less expensive than using B_1. In that case, the value of $\mathbf{t}$ increases by 1, and otherwise (i.e., if using B_1 was indeed a smart move), it decreases by 1 (with a minimum of 0). As we will see, however, TimSort's actual implementation includes many additional low-level "optimisations" compared to this simplified version, a few of which result in *worse* worst-case complexity bounds.

In this paper, we will study and compare three policies for choosing the value of $\mathbf{t}$:

1. setting $\mathbf{t}$ to a fixed constant, e.g., $\mathbf{t} = 0$ or $\mathbf{t} = 7$;
2. following TimSort's update policy;
3. setting $\mathbf{t} = \tau\lceil\log_2(a+b)\rceil^2$ whenever we merge runs of lengths a and b, for some constant τ;

Since the first policy, which consists in not updating $\mathbf{t}$ at all, is the simplest one, it will be our choice by default. In Section 3, we focus only on generic properties of this "no-update" policy; in Section 4, we prove that these properties are useful in combination with many natural merge sorts. Then, in Section 5, we focus on TimSort's actual update policy for the parameter $\mathbf{t}$, thereby identifying two weaknesses of this policy: using it may cause a non-negligible overhead, and may yet be rather inefficient. Finally, in Section 6, we mostly focus on the third policy presented above, which consists in choosing $\mathbf{t}$ as a function of the lengths of those runs we want to merge; in particular, we shall prove that this policy enjoys the positive aspects of both previous update policies, while avoiding their drawbacks.

3 Fast-growth and (tight) middle-growth properties

In this section, we focus on two novel properties of stable natural merge sorts, which we call *fast-growth* and *middle-growth*, and on a variant of the latter property, which we call *tight middle-growth*. These properties capture all TimSort-like natural merge sorts invented in the last decade, and explain why these sorting algorithms require only $\mathcal{O}(n + n\mathcal{H})$ element moves and $\mathcal{O}(n + n \min\{\mathcal{H}, \mathcal{H}^*\})$ element comparisons. We will prove in Section 4 that many algorithms have these properties.

When applying a stable natural merge sort on an array A, the elements of A are clustered into monotonic subarrays called *runs*, and the algorithm consists in repeatedly merging consecutive runs into one larger run until the array itself contains only one run. Consequently, each element may undergo several successive merge operations. *Merge trees* [3, 16, 23] are a convenient way to represent the succession of runs that ever occur while A is being sorted.

Definition 3. The *merge tree* induced by a stable natural merge sort algorithm on an array A is the binary rooted tree $\mathcal{T}$ defined as follows. The nodes of $\mathcal{T}$ are all the runs that were present in the initial array A or that resulted from merging two runs. The runs of the initial array are the leaves of $\mathcal{T}$, and when two consecutive runs R_1 and R_2 are merged with each other into a new run $\overline{R}$, the run R_1 spanning positions immediately to the left of those of R_2, they form the left and the right children of the node $\overline{R}$, respectively.

Such trees ease the task of referring to several runs that might not have occurred simultaneously. In particular, we will often refer to the i^{th} *ancestor* or a run R, which is just R itself if $i = 0$, or the parent, in the tree $\mathcal{T}$, of the $(i-1)^{\text{th}}$ ancestor of R if $i \geqslant 1$. That ancestor will be denoted by $R^{(i)}$.

Before further manipulating these runs, let us first present some notation about runs and their lengths, which we will frequently use. Each run will commonly be denoted by a upper-case letter, possibly with some index or adornment, and its length will be denoted by the same small-case letter and the same index or adornment. For instance, runs named R, R_i, $R^{(j)}$, Q' and $\overline{S}$ will have respective lengths r, r_i, $r^{(j)}$, q' and $\overline{s}$. Finally, we say that a run R is a *left* run if it is the left child of its parent, and that it is a *right* run if it is a right child. The root of a merge tree is neither left nor right.

Definition 4. We say that a stable natural merge sort algorithm $\mathcal{A}$ has the *fast-growth property* if it satisfies the following statement:

> There exist an integer $\ell \geqslant 1$ and a real number $\alpha > 1$ such that, for every merge tree $\mathcal{T}$ induced by $\mathcal{A}$ and every run R at depth ℓ or more in $\mathcal{T}$, we have $r^{(\ell)} \geqslant \alpha r$.

We also say that $\mathcal{A}$ has the *middle-growth property* if it satisfies the following statement:

> There exists a real number $\beta > 1$ such that, for every merge tree $\mathcal{T}$ induced by $\mathcal{A}$, every integer $h \geqslant 0$ and every run R of height h in $\mathcal{T}$, we have $r \geqslant \beta^h$.

Finally, we say that $\mathcal{A}$ has the *tight middle-growth property* if it satisfies the following statement:

> There exists an integer $\gamma \geqslant 0$ such that, for every merge tree $\mathcal{T}$ induced by $\mathcal{A}$, every integer $h \geqslant 0$ and every run R of height h in $\mathcal{T}$, we have $r \geqslant 2^{h-\gamma}$.

Since every node of height $h \geqslant 1$ in a merge tree is a run of length at least 2, each algorithm with the fast-growth property or with the tight middle-growth property also has the middle-growth property: indeed, it suffices

to choose $\beta = \min\{2, \alpha\}^{1/\ell}$ in the first case, and $\beta = 2^{1/(\gamma+1)}$ in the second one. As a result, the first and third properties are stronger than the second one, and indeed they have stronger consequences.

Theorem 5. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the fast-growth property. If $\mathcal{A}$ uses either the galloping or the naïve routine for merging runs, it requires $\mathcal{O}(n + n\mathcal{H})$ element comparisons and moves to sort arrays of length n and run-length entropy $\mathcal{H}$.*

Proof. Let $\ell \geqslant 1$ and $\alpha > 1$ be the integer and the real number mentioned in the definition of the statement "$\mathcal{A}$ has the fast-growth property". Let A be an array of length n with ρ runs of lengths $r_1, r_2, \ldots, r_\rho$, let $\mathcal{T}$ be the merge tree induced by $\mathcal{A}$ on A, and let d_i be the depth of the run R_i in the tree $\mathcal{T}$.

The algorithm $\mathcal{A}$ uses $\mathcal{O}(n)$ element comparisons and element moves to delimit the runs it will then merge and to make them non-decreasing. Then, both the galloping and the naïve merging routine require $\mathcal{O}(a + b)$ element comparisons and moves to merge two runs A and B of lengths a and b. Therefore, it suffices to prove that $\sum_{R\in\mathcal{T}} r = \mathcal{O}(n + n\mathcal{H})$.

Consider some leaf R_i of the tree $\mathcal{T}$, and let $k = \lfloor d_i/\ell \rfloor$. The run $R_i^{(k\ell)}$ has length $r_i^{(k\ell)} \geqslant \alpha^k r_i$, and thus $n \geqslant r_i^{(k\ell)} \geqslant \alpha^k r_i$. Hence, $d_i + 1 \leqslant \ell(k+1) \leqslant \ell\,(\log_\alpha(n/r_i) + 1)$, and we conclude that

$$\sum_{R\in\mathcal{T}} r = \sum_{i=1}^{\rho}(d_i+1)r_i \leqslant \ell\sum_{i=1}^{\rho}(r_i\log_\alpha(n/r_i) + r_i) = \ell(n\mathcal{H}/\log_2(\alpha) + n) = \mathcal{O}(n + n\mathcal{H}). \qquad \square$$

Similar, weaker results also hold for algorithms with the (tight) middle-growth property.

Theorem 6. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. If $\mathcal{A}$ uses either the galloping or the naïve routine for merging runs, it requires $\mathcal{O}(n\log(n))$ element comparisons and moves to sort arrays of length n. If, furthermore, $\mathcal{A}$ has the tight middle-growth property, it requires at most $n\log_2(n) + \mathcal{O}(n)$ element comparisons and moves to sort arrays of length n.*

Proof. Let us borrow the notations from the previous proof, and let $\beta > 1$ be the real number mentioned in the definition of the statement "$\mathcal{A}$ has the middle-growth property". Like in the proof of Theorem 5, it suffices to show that $\sum_{R\in\mathcal{T}} r = \mathcal{O}(n\log(n))$. The $d_i{}^{\text{th}}$ ancestor of a run R_i is the root of $\mathcal{T}$, and thus $n \geqslant \beta^{d_i}$. Hence, $d_i \leqslant \log_\beta(n)$, which proves that

$$\sum_{R\in\mathcal{T}} r = \sum_{i=1}^{\rho}(d_i+1)r_i \leqslant \sum_{i=1}^{\rho}(\log_\beta(n)+1)r_i = (\log_\beta(n)+1)n = \mathcal{O}(n\log(n)).$$

Similarly, if $\mathcal{A}$ has the tight middle-growth property, let γ be the integer mentioned in the definition of the statement "$\mathcal{A}$ has the tight middle-growth property". This time, $n \geqslant 2^{d_i-\gamma}$, which proves that $d_i \leqslant \log_2(n)+\gamma$ and that

$$\sum_{R\in\mathcal{T}} r = \sum_{i=1}^{\rho}(d_i+1)r_i \leqslant \sum_{i=1}^{\rho}(\log_2(n)+\gamma+1)r_i = (\log_2(n)+\gamma+1)n = n\log_2(n) + \mathcal{O}(n). \qquad \square$$

Theorems 5 and 6 provide us with a simple framework for recovering well-known results on the complexity of many algorithms. By contrast, Theorem 7 consists in new complexity guarantees on the number of element comparisons performed by algorithms with the middle-growth property, provided that they use the galloping routine.

Theorem 7. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. If $\mathcal{A}$ uses the galloping routine for merging runs, it requires $\mathcal{O}(n+n\mathcal{H}^*)$ element comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$.*

Proof. Let $\beta > 1$ be the real number mentioned in the definition of the statement "$\mathcal{A}$ has the middle-growth property". Let A be an array of length n with σ dual runs $S_1, S_2, \ldots, S_\sigma$, and let $\mathcal{T}$ be the merge tree induced by $\mathcal{A}$ on A.

The algorithm $\mathcal{A}$ uses $\mathcal{O}(n)$ element comparisons to delimit the runs it will then merge and to make them non-decreasing. We prove now that merging these runs requires only $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons. For every run R in $\mathcal{T}$ and every integer $i \leqslant \sigma$, let $r_{\to i}$ be the number of elements of R that belong to the dual run S_i. Proposition 2 states that merging two runs R and R' requires at most $1+\sum_{i=1}^{\sigma} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i})+\mathsf{cost}^*_{\mathbf{t}}(r'_{\to i})$ element comparisons. Since less than n such merge operations are performed, and since $n = \sum_{i=1}^{\sigma} s_i$ and $n\mathcal{H}^* = \sum_{i=1}^{\sigma} s_i \log(n/s_i)$, it remains to show that

$$\sum_{R\in\mathcal{T}} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) = \mathcal{O}(s_i + s_i \log(n/s_i))$$

for all $i \leqslant \sigma$. Then, since $\mathsf{cost}^*_{\mathbf{t}}(m) \leqslant (\mathbf{t}+1)\mathsf{cost}^*_0(m)$ for all parameter values $\mathbf{t} \geqslant 0$ and all $m \geqslant 0$, we assume without loss of generality that $\mathbf{t} = 0$.

Consider now some integer $h \geqslant 0$, and let $\mathsf{C}_0(h) = \sum_{R\in\mathcal{R}_h} \mathsf{cost}^*_0(r_{\to i})$, where $\mathcal{R}_h$ denotes the set of runs at height h in $\mathcal{T}$. Since no run in $\mathcal{R}_h$ descends from another one, we already have $\mathsf{C}_0(h) \leqslant 2\sum_{R\in\mathcal{R}_h} r_{\to i} \leqslant 2s_i$ and $\sum_{R\in\mathcal{R}_h} r \leqslant n$. Moreover, by definition of β, each run $R \in \mathcal{R}_h$ is of length $r \geqslant \beta^h$, and thus $|\mathcal{R}_h| \leqslant n/\beta^h$.

Let also $f\colon x \mapsto 2 + 2\log_2(x+1)$, $g\colon x \mapsto x\, f(s_i/x)$ and $\lambda = \lceil \log_\beta(n/s_i) \rceil$. Both f and g are positive and concave on the interval $(0, +\infty)$, thereby also being increasing. It follows that, for all $h \geqslant 0$,

$$\begin{aligned}
\mathsf{C}_0(\lambda+h) &\leqslant \sum_{R\in\mathcal{R}_{\lambda+h}} f(r_{\to i}) \leqslant |\mathcal{R}_{\lambda+h}|\, f\big(\textstyle\sum_{R\in\mathcal{R}_{\lambda+h}} r_{\to_i}/|\mathcal{R}_{\lambda+h}|\big) \leqslant g\big(|\mathcal{R}_{\lambda+h}|\big) \\
&\leqslant g\big(n/\beta^{\lambda+h}\big) \leqslant g\big(s_i\beta^{-h}\big) = \big(2 + 2\log_2(\beta^h+1)\big)s_i\beta^{-h} \\
&\leqslant \big(2 + 2\log_2(2\beta^h)\big)s_i\beta^{-h} = \big(4 + 2h\log_2(\beta)\big)s_i\beta^{-h}.
\end{aligned}$$

Inequalities on the first line hold by definition of cost^*_0, because f is concave, and because f is increasing; inequalities on the second line hold because g is increasing and because $|\mathcal{R}_h| \leqslant n/\beta^h$.

We conclude that

$$\begin{aligned}
\sum_{R\in\mathcal{T}} \mathsf{cost}^*_0(r_{\to i}) &= \sum_{h\geqslant 0} \mathsf{C}_0(h) = \sum_{h=0}^{\lambda-1} \mathsf{C}_0(h) + \sum_{h\geqslant 0} \mathsf{C}_0(\lambda+h) \\
&\leqslant 2\lambda s_i + 4s_i \sum_{h\geqslant 0} \beta^{-h} + 2\log_2(\beta) s_i \sum_{h\geqslant 0} h\beta^{-h} \\
&\leqslant \mathcal{O}\big(s_i(1+\log(n/s_i)\big) + \mathcal{O}\big(s_i\big) + \mathcal{O}\big(s_i\big) = \mathcal{O}\big(s_i + s_i\log(n/s_i)\big).
\end{aligned}$$

□

4 Algorithms with the fast- and (tight) middle-growth properties

In this section, we briefly present the algorithms mentioned in Section 1 and prove that each of them enjoys the fast-growth property and/or the (tight) middle-growth property. Before treating these algorithms one by one, we first sum up our results.

Theorem 8. *The algorithms TimSort, α-MergeSort, PowerSort, PeekSort and adaptive ShiversSort have the fast-growth property.*

An immediate consequence of Theorems 5 and 7 is that these algorithms sort arrays of length n and run-length entropy $\mathcal{H}$ in time $\mathcal{O}(n + n\mathcal{H})$, which was already well-known; and that, if used with the galloping merging routine, they only need $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$, which is a new result.

Theorem 9. *The algorithms NaturalMergeSort, ShiversSort and α-StackSort have the middle-growth property.*

Theorem 7 proves that, if these three algorithms are used with the galloping merging routine, they only need $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$. By contrast, observe that they can be implemented by using a stack, following TimSort's own implementation, but where only the two top runs of the stack could be merged. It is proved in [16] that such algorithms may require $\omega(n+n\mathcal{H})$ comparisons to sort arrays of length n and run-length entropy $\mathcal{H}$. Hence, Theorem 5 shows that these three algorithms do *not* have the fast-growth property.

Theorem 10. *The algorithms NaturalMergeSort, ShiversSort and PowerSort have the tight middle-growth property.*

Theorem 6 proves that these algorithms sort arrays of length n in time $n\log_2(n) + \mathcal{O}(n)$, which was already well-known. In Section 6, we will further improve our upper bounds on the number of comparisons that these algorithms require when sorting arrays of length n and dual run-length entropy $\mathcal{H}^*$.

Note that the algorithms α-StackSort, α-MergeSort and TimSort may require more than $n\log_2(n) + \mathcal{O}(n)$ comparisons to sort arrays of length n, which proves that they do not have the tight middle-growth property. In Section 6, we will prove that adaptive ShiversSort and PeekSort also fail to have the this property, although they enjoy still complexity upper bounds similar to those of algorithms with the tight middle-growth property.

4.1 Algorithms with the fast-growth property

4.1.1 PowerSort

The algorithm PowerSort is best defined by introducing the notion of *power* of a run endpoint or of a run, and then characterising the merge trees that PowerSort induces.

Definition 11. Let A be an array of length n, whose run decomposition consists of runs $R_1, R_2, \ldots, R_\rho$, ordered from left to right. For all integers $i \leqslant \rho$, let $e_i = r_1 + \ldots + r_i$. We also abusively set $e_{-1} = -\infty$ and $e_{\rho+1} = n$.

When $0 \leqslant i \leqslant \rho$, let $\mathbf{I}(i)$ denote the half-open interval $(e_{i-1} + e_i, e_i + e_{i+1}]$. The *power* of e_i, denoted by p_i, is then defined as the least integer p such that $\mathbf{I}(i)$ contains an element of the set $\{kn/2^{p-1} : k \in \mathbb{Z}\}$. Thus, we (abusively) have $p_0 = -\infty$ and $p_\rho = 0$, because $\mathbf{I}(0) = (-\infty, r_1]$ and $\mathbf{I}(\rho) = (2n - r_\rho, 2n]$.

Finally, let $R_{i\ldots j}$ be a run obtained by merging consecutive runs $R_i, R_{i+1}, \ldots, R_j$. The *power* of the run $R_{i\ldots j}$ is defined as $\max\{p_{i-1}, p_j\}$.

This definition is borrowed from [23, Definition 3], except that we added *sentinel* powers $p_0 = -\infty$ and $p_\rho = 0$, and work directly with lengths e_i instead of probabilities e_i/n.

Lemma 12. *Each non-empty sub-interval I of the set $\{0, \ldots, \rho\}$ contains exactly one integer i such that $p_i \leqslant p_j$ for all $j \in I$.*

Proof. Assume that the integer i is not unique. Since e_0 is the only endpoint with power $-\infty$, we know that $0 \notin I$. Then, let a and b be elements of I such that $a < b$ and $p_a = p_b \leqslant p_j$ for all $j \in I$, and let $p = p_a = p_b$. By definition of p_a and p_b, there exist odd integers k and ℓ such that $kn/2^{p-1} \in \mathbf{I}(a)$ and $\ell n/2^{p-1} \in \mathbf{I}(b)$. Since $\ell \geqslant k+1$, the fraction $(k+1)n/2^{p-1}$ belongs to some interval $\mathbf{I}(j)$ such that $a \leqslant j \leqslant b$. But since $k+1$ is even, we know that $p_j < p$, which is absurd. This invalidates our initial assumption and completes the proof. □

Corollary 13. *Let $R_1, \ldots, R_\rho$ be the run decomposition of an array A. There is exactly one tree $\mathcal{T}$ that is induced on A and in which every inner node has a smaller power than its children. Furthermore, for every run $R_{i\ldots j}$ in $\mathcal{T}$, we have $\max\{p_{i-1}, p_j\} < \min\{p_i, p_{i+1}, \ldots, p_{j-1}\}$.*

Proof. Given a merge tree $\mathcal{T}$, let us prove that the following statements are equivalent:

- S_1: each inner node of $\mathcal{T}$ has a smaller power than its children;
- S_2: each run $R_{i\ldots j}$ that belongs to $\mathcal{T}$ has a power that is smaller than all of $p_i, \ldots, p_{j-1}$;
- S_3: if a run $R_{i\ldots j}$ is an inner node of $\mathcal{T}$, its children are the two runs $R_{i\ldots k}$ and $R_{k+1\ldots j}$ such that $p_k = \min\{p_i, \ldots, p_{j-1}\}$.

First, if S_1 holds, we prove S_3 by induction on the height h of the run $R_{i\ldots j}$. Indeed, if the restriction of S_3 to runs of height less than h holds, let $R_{i\ldots k}$ and $R_{k+1\ldots j}$ be the children of a run $R_{i\ldots j}$ of height h. If $i < k$, the run $R_{i\ldots k}$ has two children $R_{i\ldots \ell}$ and $R_{\ell+1\ldots k}$ such that $p_\ell = \min\{p_i, \ldots, p_{k-1}\}$, and the powers of these runs, i.e., $\max\{p_{i-1}, p_\ell\}$ and $\max\{p_\ell, p_k\}$, are greater than the power of $R_{i\ldots k}$, i.e., $\max\{p_{i-1}, p_k\}$, which proves that $p_\ell > p_k$. It follows that $p_k = \min\{p_i, \ldots, p_k\}$, and one proves similarly that $p_k = \min\{p_k, \ldots, p_{j-1}\}$, thereby showing that S_3 also holds for runs of height h.

Then, if S_3 holds, we prove S_2 by induction on the depth d of the run $R_{i\dots j}$. Indeed, if the restriction of S_2 to runs of depth less than d holds, let $R_{i\dots k}$ and $R_{k+1\dots j}$ be the children of a run $R_{i\dots j}$ of depth d. Lemma 12 and S_3 prove that p_k is the unique smallest element of $\{p_i, \dots, p_{j-1}\}$, and the induction hypothesis proves that $\max\{p_{i-1}, p_j\} < p_k$. It follows that both powers $\max\{p_{i-1}, p_k\}$ and $\max\{p_k, p_j\}$ are smaller than all of $p_i, \dots, p_{k-1}, p_{k+1}, \dots, p_{j-1}$, thereby showing that S_2 also holds for runs of depth d.

Finally, if S_2 holds, let $R_{i\dots j}$ be an inner node of $\mathcal{T}$, with children $R_{i\dots k}$ and $R_{k+1\dots j}$. Property S_2 ensures that $\max\{p_{i-1}, p_j\} < p_k$, and thus that $\max\{p_{i-1}, p_j\}$ is smaller than both $\max\{p_{i-1}, p_k\}$ and $\max\{p_k, p_j\}$, i.e., that $R_{i\dots j}$ has a smaller power that its children, thereby proving S_1.

In particular, once the array A and its run decomposition $R_1, \dots, R_\rho$ are fixed, S_3 provides us with a deterministic top-down construction of the unique merge tree $\mathcal{T}$ induced on A and that satisfies S_1: the root of $\mathcal{T}$ must be the run $R_{1\dots\rho}$ and, provided that some run $R_{i\dots j}$ belongs to $\mathcal{T}$, where $i < j$, Lemma 12 proves that the integer k mentioned in S_3 is unique, which means that S_3 unambiguously describes the children of $R_{i\dots j}$ in the tree $\mathcal{T}$.

This proves the first claim of Corollary 13, and the second claim of Corollary 13 follows from the equivalence between the statements S_1 and S_2. □

This leads to the following characterisation of the algorithm PowerSort, which is proved in [23, Lemma 4], and which Corollary 13 allows us to consider as an alternative definition of PowerSort.

Definition 14. In every merge tree that PowerSort induces, inner nodes have a smaller power than their children.

Lemma 15. *Let $\mathcal{T}$ be a merge tree induced by PowerSort, let R be a run of $\mathcal{T}$ with power p, and let $R^{(2)}$ be its grandparent. We have $2^{p-2}r < n < 2^p r^{(2)}$.*

Proof. Let $R_{i\dots j}$ be the run R. Without loss of generality, we assume that $p = p_j$, the case $p = p_{i-1}$ being entirely similar. Corollary 13 states that all of $p_i, \dots, p_{j-1}$ are larger than p, and therefore that $p \leqslant \min\{p_i, \dots, p_j\}$. Thus, the union of intervals $\mathbf{I}(i) \cup \dots \cup \mathbf{I}(j) = (e_{i-1} + e_i, e_j + e_{j+1}]$ does not contain any element of the set $\mathcal{S} = \{kn/2^{p-2} \colon k \in \mathbb{Z}\}$. Consequently, the bounds $e_{i-1} + e_i$ and $e_j + e_{j+1}$ are contained between two consecutive elements of $\mathcal{S}$, i.e., there exists an integer ℓ such that

$$\ell n/2^{p-2} \leqslant e_{i-1} + e_i \leqslant e_j + e_{j+1} < (\ell + 1)n/2^{p-2}.$$

We conclude that

$$r = e_j - e_{i-1} \leqslant (e_j + e_{j+1}) - (e_{i-1} + e_i) < n/2^{p-2}.$$

We prove now that $n \leqslant 2^p r^{(2)}$. To that end, we assume that both R and $R^{(1)}$ are left children, the other possible cases being entirely similar. There exist integers u and v such that $R^{(1)} = R_{i\dots u}$ and $R^{(2)} = R_{i\dots v}$. Hence, $\max\{p_{i-1}, p_u\} < \max\{p_{i-1}, p_j\} = p$, which shows that $p_u < p_j = p$. Thus, both intervals $\mathbf{I}(j)$ and $\mathbf{I}(u)$, which are subintervals of $(2e_{i-1}, 2e_v]$, contain elements of the set $\mathcal{S}' = \{kn/2^{p-1} \colon k \in \mathbb{Z}\}$. This means that there exist two integers k and ℓ such that $2e_{i-1} < kn/2^{p-1} < \ell n/2^{p-1} \leqslant 2e_v$, from which we conclude that

$$r^{(2)} = e_v - e_{i-1} > (\ell - k)n/2^p \geqslant n/2^p.$$ □

Theorem 16. *The algorithm PowerSort has the fast-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by PowerSort. Then, let R be a run in $\mathcal{T}$, and let p and $p^{(3)}$ be the respective powers of the runs R and $R^{(3)}$. Definition 14 ensures that $p \geqslant p^{(3)} + 3$, and therefore Lemma 15 proves that

$$2^{p^{(3)}+1} r \leqslant 2^{p-2} r < n < 2^{p^{(3)}} r^{(5)}.$$

This means that $r^{(5)} \geqslant 2r$, and therefore that PowerSort has the fast-growth property. □

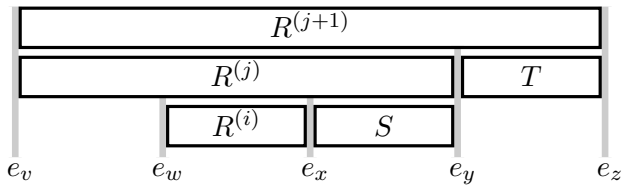

Figure 4: Runs $R^{(i)}$, $R^{(j)}$, $R^{(j+1)}$, their siblings S and T, and endpoints $e_v \leqslant e_w < e_x < e_y < e_z$.

4.1.2 PeekSort

Like its sibling PowerSort, the algorithm PeekSort is best defined by characterizing the merge trees it induces.

Definition 17. Let $\mathcal{T}$ be the merge tree induced by PeekSort on an array A. The children of each internal node $R_{i\cdots j}$ of $\mathcal{T}$ are the runs $R_{i\cdots k}$ and $R_{k+1\cdots j}$ for which the quantity

$$\mathsf{d}(k) = |2e_k - e_j - e_{i-1}|$$

is minimal. In case of equality between two real numbers $\mathsf{d}(k)$, the integer k is chosen to be as small as possible.

Proposition 18. *The algorithm PeekSort has the fast-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by PeekSort, and let R be a run in $\mathcal{T}$. We prove that $r^{(3)} \geqslant 2r$. Indeed, let us assume that a majority of the runs $R = R^{(0)}$, $R^{(1)}$ and $R^{(2)}$ are left runs. The situation is entirely similar if a majority of these runs are right runs.

Let $i < j$ be the two smallest integers such that $R^{(i)}$ and $R^{(j)}$ are left runs, and let S and T be their right siblings, as illustrated in Figure 4. We can write these runs as $R^{(i)} = R_{w+1\cdots x}$, $S = R_{x+1\cdots y}$, $R^{(j)} = R_{v+1\cdots y}$ and $T = R_{y+1\cdots z}$ for some integers $v \leqslant w < x < y < z$.

Definition 17 states that

$$|r^{(j)} - t| = |2e_y - e_z - e_v| \leqslant |2e_{y-1} - e_z - e_v| = |r^{(j)} - t - 2r_y|.$$

Thus, $r^{(j)} - t - 2r_y$ is negative, i.e., $t > r^{(j)} - 2r_y$, and

$$r^{(3)} \geqslant r^{(j+1)} = r^{(j)} + t > 2r^{(j)} - 2r_y = 2(e_{y-1} - e_v) \geqslant 2(e_x - e_w) = 2r^{(i)} \geqslant 2r. \qquad \square$$

4.1.3 Adaptive ShiversSort

The algorithm adaptive ShiversSort is presented in Algorithm 1. It is based on an *ad hoc* tool, which we call *level* of a run : the level of a run R of length r is defined as the number $\ell = \lfloor \log_2(r) \rfloor$. Following our naming conventions, let $\ell^{(i)}$ and ℓ_i denote the respective levels of the runs $R^{(i)}$ and R_i.

Observe that appending a fictitious run of length $2n$ to the array A and stopping our sequence of merges just before merging that fictitious run does not modify the sequence of merges performed by the algorithm, but allows us to assume that every merge was performed in line 4. Therefore, we work below under that assumption.

Our proof is based on the following result, which was stated and proved in [16, Lemma 7].

Lemma 19. *Let $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ be a stack obtained while executing adaptive ShiversSort. We have (i) $\ell_i \geqslant \ell_{i+1} + 1$ whenever $1 \leqslant i \leqslant h-3$ and (ii) $\ell_{h-3} \geqslant \ell_{h-1} + 1$ if $h \geqslant 4$.*

Then, Lemmas 20 and 21 focus on inequalities involving the levels of a given run R belonging to a merge tree induced by adaptive ShiversSort, and of its ancestors. In each case, the stack just before the run R is merged is denoted by $\mathcal{S} = (R_1, R_2, \ldots, R_h)$.

Lemma 20. *If $R^{(1)}$ is a right run, $\ell^{(2)} \geqslant \ell + 1$.*

Input : Array A to sort
Result : The array A is sorted into a single run. That run remains on the stack.
Note : The height of the stack $\mathcal{S}$ is denoted by h; its i^{th} deepest run by R_i; the length of R_i by r_i. Finally, we set $\ell_i = \lfloor \log_2(r_i) \rfloor$. When two consecutive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, with the run resulting from the merge.

1 $\mathcal{S} \leftarrow$ an empty stack
2 **while** `true`**:**
3 **if** $h \geqslant 3$ and $\ell_{h-2} \leqslant \max\{\ell_{h-1}, \ell_h\}$**:**
4 merge the runs R_{h-2} and R_{h-1}
5 **else if** the end of the array has not yet been reached**:**
6 find a new monotonic run R, make it non-decreasing, and push it onto $\mathcal{S}$
7 **else:**
8 break
9 **while** $h \geqslant 2$**:**
10 merge the runs R_{h-1} and R_h

Algorithm 1: adaptive ShiversSort

Proof. The run R coincides with either R_{h-2} or R_{h-1}, and $R^{(1)}$ is the parent of the runs R_{h-2} and R_{h-1}. Hence, the run R_{h-3} descends from the left sibling of $R^{(1)}$ and from $R^{(2)}$. Thus, inequalities (i) and (ii) prove that $\ell^{(2)} \geqslant \ell_{h-3} \geqslant \max\{\ell_{h-2}, \ell_{h-1}\} + 1 \geqslant \ell + 1$. □

Lemma 21. *If R is a left run, $\ell^{(2)} \geqslant \ell + 1$.*

Proof. Since R is a left run, it coincides with R_{h-2}, and $\ell_{h-2} \leqslant \max\{\ell_{h-1}, \ell_h\}$. Then, if $R^{(1)}$ is a right run, Lemma 20 already proves that $\ell^{(2)} \geqslant \ell + 1$. Otherwise, $R^{(1)}$ is a left run, and the run R_h descends from $R^{(2)}$, thereby proving that

$$r^{(2)} \geqslant r + \max\{r_{h-1}, r_h\} \geqslant 2^{\ell} + 2^{\max\{\ell_{h-1}, \ell_h\}} \geqslant 2^{\ell+1},$$

and thus that $\ell^{(2)} \geqslant \ell + 1$ too. □

Proposition 22. *The algorithm adaptive ShiversSort has the fast-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by adaptive ShiversSort, and let R be a run in $\mathcal{T}$. Lemma 20 shows that $\ell^{(3)} \geqslant \ell^{(2)} \geqslant \ell + 1$ if $R^{(1)}$ is a right run, and Lemma 21 shows that $\ell^{(3)} \geqslant \ell^{(1)} + 1 \geqslant \ell + 1$ if $R^{(1)}$ is a left run. Thus, the inequality $\ell^{(3)} \geqslant \ell + 1$ is valid in both cases, and we show similarly that $\ell^{(6)} \geqslant \ell^{(3)} + 1$. It follows that $r^{(6)} \geqslant 2^{\ell^{(6)}} \geqslant 2^{\ell+2} \geqslant 2r$. □

4.1.4 TimSort

The algorithm TimSort is presented in Algorithm 2.

We say that a run R is a #1-, a #2-, a #3 or a #4-run if is merged in line 4, 6, 8 or 10, respectively. Like in Section 4.1.3, appending a fictitious run of length $2n$ that we will avoid merging allows us to assume that every run is merged in line 4, 6, 8 or 10, i.e., is a #1-, a #2-, a #3 or a #4-run.

Our proof is then based on the following result, which extends [1, Lemma 5] by adding the inequality (v).

Lemma 23. *Let $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ be a stack obtained while executing TimSort. We have*

(i) $r_i > r_{i+1} + r_{i+2}$ *whenever* $1 \leqslant i \leqslant h - 4$,

(ii) $3r_{h-2} > r_{h-1}$ *if* $h \geqslant 3$,

(iii) $r_{h-3} > r_{h-2}$ *if* $h \geqslant 4$,

(iv) $r_{h-3} + r_{h-2} > r_{h-1}$ *if* $h \geqslant 4$, *and*

(v) $\max\{r_{h-3}/2, 4r_h\} > r_{h-1}$ *if* $h \geqslant 4$.

Input : Array A to sort
Result : The array A is sorted into a single run. That run remains on the stack.
Note : The height of the stack $\mathcal{S}$ is denoted by h; its i^{th} deepest run by R_i; the length of R_i by r_i. When two consecutive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, with the run resulting from the merge.

$\mathcal{S} \leftarrow$ an empty stack
while `true`**:**
  if $h \geqslant 3$ and $r_{h-2} < r_h$**:**
    merge the runs R_{h-2} and R_{h-1} ▷ case #1
  else if $h \geqslant 2$ and $r_{h-1} \leqslant r_h$**:**
    merge the runs R_{h-1} and R_h ▷ case #2
  else if $h \geqslant 3$ and $r_{h-2} \leqslant r_{h-1} + r_h$**:**
    merge the runs R_{h-1} and R_h ▷ case #3
  else if $h \geqslant 4$ and $r_{h-3} \leqslant r_{h-2} + r_{h-1}$**:**
    merge the runs R_{h-1} and R_h ▷ case #4
  else if the end of the array has not yet been reached**:**
    find a new monotonic run R, make it non-decreasing, and push it onto $\mathcal{S}$
  else:
    break
while $h \geqslant 2$**:**
  merge the runs R_{h-1} and R_h

Algorithm 2: TimSort

Proof. Lemma 5 from [1] already proves the inequalities (i) to (iv). Therefore, we prove, by a direct induction on the number of (push or merge) operations performed before obtaining the stack $\mathcal{S}$, that $\mathcal{S}$ also satisfies (v).

When the algorithm starts, we have $h \leqslant 3$, and therefore there is nothing to prove in that case. Then, when a stack $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ obeying the inequalities (i) to (v) is transformed into a stack $\overline{\mathcal{S}} = (\overline{R}_1, \overline{R}_2, \ldots, \overline{R}_{\overline{h}})$

▷ by inserting a run, cases #2 and #3 just failed to occur and $\overline{h} = h + 1$, so that

$$\overline{r}_{\overline{h}-3} = r_{h-2} > r_{h-1} + r_h > 2r_h = 2\overline{r}_{\overline{h}-1};$$

▷ by merging the runs R_{h-1} and R_h, we have $\overline{h} = h - 1$ and

$$\overline{r}_{\overline{h}-3} = r_{h-4} > r_{h-3} + r_{h-2} > 2r_{h-2} = 2\overline{r}_{\overline{h}-1};$$

▷ by merging the runs R_{h-2} and R_{h-1}, case #1 just occurred and $\overline{h} = h - 1$, so that

$$4\overline{r}_{\overline{h}} = 4r_h > 4r_{h-2} > r_{h-2} + r_{h-1} = \overline{r}_{\overline{h}-1}.$$

In each case, $\overline{\mathcal{S}}$ satisfies (v), which completes the induction and the proof. □

Then, Lemmas 24 and 25 focus on inequalities involving the lengths of a given run R belonging to a merge tree induced by TimSort and of its ancestors. In each case, the stack just before the run R is merged is denoted by $\mathcal{S} = (R_1, R_2, \ldots, R_h)$.

Lemma 24. *If $R^{(1)}$ is a right run, $r^{(2)} \geqslant 4r/3$.*

Proof. Let S be the left sibling of the run $R^{(1)}$, and let i be the integer such that $R = R_i$. If $i = h - 2$, (iii) shows that $r_{i-1} > r_i$. If $i = h - 1$, (ii) shows that $r_{i-1} > r_i/3$. In both cases, the run R_{i-1} descends from $R^{(2)}$, and thus $r^{(2)} \geqslant r + r_{i-1} \geqslant 4r/3$.

Finally, if $i = h$, the run R is a #2-, #3- or #4-right run, which means both that $r_{h-2} \geqslant r$ and that R_{h-2} descends from S. It follows that $r^{(2)} \geqslant r + r_{h-2} \geqslant 2r$. □

Lemma 25. *If R is a left run, $r^{(2)} \geqslant 5r/4$.*

Proof. We treat four cases independently, depending on whether R is a #1-, a #2-, a #3- or a #4-left run. In each case, we assume that the run $R^{(1)}$ is a left run, since Lemma 24 already proves that $r^{(2)} \geqslant 4r/3$ when $R^{(1)}$ is a right run.

- ▷ If R is a #1-left run, the run $R = R_{h-2}$ is merged with R_{h-1} and $r_{h-2} < r_h$. Since $R^{(1)}$ is a left run, R_h descends from $R^{(2)}$, and thus $r^{(2)} \geqslant r + r_h \geqslant 2r$.
- ▷ If R is a #2-left run, the run $R = R_{h-1}$ is merged with R_h and $r_{h-1} \leqslant r_h$. It follows, in that case, that $r^{(2)} \geqslant r^{(1)} = r + r_h \geqslant 2r$.
- ▷ If R is a #3-left run, the run $R = R_{h-1}$ is merged with R_h, and $r_{h-2} \leqslant r_{h-1} + r_h = r^{(1)}$. Due to this inequality, our next journey through the loop of lines 2 to 14 must trigger another merge. Since $R^{(1)}$ is a left run, that merge must be a #1-merge, which means that $r_{h-3} < r^{(1)}$. Consequently, (v) proves that $r^{(1)} \geqslant \max\{r_{h-3}, r_{h-1} + r_h\} \geqslant 5r_{h-1}/4 = 5r/4$.
- ▷ We prove that R cannot be a #4-left run. Indeed, if R is a #4-left run, the run $R = R_{h-1}$ is merged with R_h, and we both have $r_{h-2} > r^{(1)}$ and $r_{h-3} \leqslant r_{h-2} + r_{h-1} \leqslant r_{h-2} + r^{(1)}$. Due to the latter inequality, our next journey through the loop of lines 2 to 14 must trigger another merge. Since $r^{(1)} < r_{h-2} < r_{h-3}$, that new merge cannot be a #1-merge, and thus $R^{(1)}$ is a right run, contradicting our initial assumption. □

Proposition 26. *The algorithm TimSort has the fast-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by TimSort, and let R be a run in $\mathcal{T}$. Lemma 24 shows that $r^{(3)} \geqslant r^{(2)} \geqslant 4r/3$ if $R^{(1)}$ is a right run, and Lemma 25 shows that $r^{(3)} \geqslant 5r^{(1)}/4 \geqslant 5r/4$ if $R^{(1)}$ is a left run. Hence, in both cases, $r^{(3)} \geqslant 5r/4$. □

4.1.5 α-MergeSort

The algorithm α-MergeSort is parametrised by a real number $\alpha > 1$ and is presented in Algorithm 3.

Input : Array A to sort, parameter $\alpha > 1$
Result : The array A is sorted into a single run. That run remains on the stack.
Note : The height of the stack $\mathcal{S}$ is denoted by h; its i^{th} deepest run by R_i; the length of R_i by r_i. When two consecutive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, with the run resulting from the merge.

1 $\mathcal{S} \leftarrow$ an empty stack
2 **while** `true`**:**
3 **if** $h \geqslant 3$ and $r_{h-2} < r_h$**:**
4 merge the runs R_{h-2} and R_{h-1} ▷ case #1
5 **else if** $h \geqslant 2$ and $r_{h-1} < \alpha r_h$**:**
6 merge the runs R_{h-1} and R_h ▷ case #2
7 **else if** $h \geqslant 3$ and $r_{h-2} < \alpha r_{h-1}$**:**
8 merge the runs R_{h-1} and R_h ▷ case #3
9 **else if** the end of the array has not yet been reached**:**
10 find a new monotonic run R, make it non-decreasing, and push it onto $\mathcal{S}$
11 **else:**
12 break
13 **while** $h \geqslant 2$**:**
14 merge the runs R_{h-1} and R_h

Algorithm 3: α-MergeSort

Like in Section 4.1.4, we say that a run R is a #1-, a #2- or a #3-run if is merged in line 4, 6 or 8. In addition, still like in Sections 4.1.3 and 4.1.4, we safely assume that each run is a #1-, a #2- or a #3-run.

Our proof is then based on the following result, which is based on the tools used by Buss and Knop to prove [8, Theorem 5.3].

Lemma 27. *Let $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ be a stack obtained while executing α-MergeSort. We have*

(i) $r_i \geqslant \alpha r_{i+1}$ *whenever* $1 \leqslant i \leqslant h-3$,

(ii) $r_{h-2} \geqslant (\alpha-1) r_{h-1}$ *if* $h \geqslant 3$, *and*

(iii) $\max\{r_{h-2}/\alpha, \alpha r_h/(\alpha-1)\} \geqslant r_{h-1}$ *if* $h \geqslant 3$.

Proof. We prove, by a direct induction on the number of (push or merge) operations performed before obtaining the stack $\mathcal{S}$, that $\mathcal{S}$ satisfies (i), (ii) and (iii).

When the algorithm starts, we have $h \leqslant 2$, and therefore there is nothing to prove in that case. Then, when a stack $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ obeying (i), (ii) and (iii) is transformed into a stack $\overline{\mathcal{S}} = (\overline{R}_1, \overline{R}_2, \ldots, \overline{R}_{\overline{h}})$

- ▷ by inserting a run, $\overline{h} = h+1$ and cases #1 and #2 just failed to occur, which means that $\overline{r}_{\overline{h}-3} = r_{h-2} \geqslant \alpha r_{h-1} = \alpha \overline{r}_{\overline{h}-2}$ and $\overline{r}_{\overline{h}-2} = r_{h-1} \geqslant \alpha r_h = \alpha \overline{r}_{\overline{h}-1}$; in addition, $\overline{r}_i = r_i \geqslant \alpha r_{i+1} = \alpha \overline{r}_{i+1}$ whenever $1 \leqslant i \leqslant \overline{h}-3$;
- ▷ by merging the runs R_{h-1} and R_h, we have $\overline{h} = h-1$ and $\overline{r}_i = r_i \geqslant \alpha r_{i+1} = \alpha \overline{r}_{i+1}$ for all $i \leqslant \overline{h}-2 = h-3$;
- ▷ by merging the runs R_{h-2} and R_{h-1}, case #1 just occurred and $\overline{h} = h-1$, so that $\overline{r}_i = r_i \geqslant \alpha r_{i+1} = \alpha \overline{r}_{i+1}$ for all $i \leqslant \overline{h}-3 = h-4$, and $r_h > r_{h-2}$, which means that

$$\min\{\overline{r}_{\overline{h}-2}, \alpha \overline{r}_{\overline{h}}\} = \min\{r_{h-3}, \alpha r_h\} \geqslant \alpha r_{h-2} \geqslant (\alpha-1)(r_{h-2}+r_{h-1}) = (\alpha-1)\overline{r}_{\overline{h}-1}.$$

In each case, $\overline{\mathcal{S}}$ satisfies (i), (ii) and (iii), which completes the induction and the proof. □

Lemmas 28 and 29 focus on inequalities involving the lengths of a given run R belonging to a merge tree induced by α-MergeSort, and of its ancestors. In each case, the stack just before the run R is merged is denoted by $\mathcal{S} = (R_1, R_2, \ldots, R_h)$. In what follows, we also set $\alpha^\star = \min\{\alpha, 1+1/\alpha, 1+(\alpha-1)/\alpha\}$.

Lemma 28. *If $R^{(1)}$ is a right run, $r^{(2)} \geqslant \alpha^\star r$.*

Proof. Let i be the integer such that $R = R_i$. If $i = h-2$, (i) shows that $r_{i-1} \geqslant \alpha r_i$. If $i = h-1$, (ii) shows that $r_{i-1} \geqslant (\alpha-1) r_i$. In both cases, R_{i-1} descends from $R^{(2)}$, and thus $r^{(2)} \geqslant r + r_{i-1} \geqslant \alpha r$. Finally, if $i = h$, the run R is a #2 or #3-right run, which means that $r_{h-2} \geqslant r$ and that R_{h-2} descends from the left sibling of $r^{(1)}$. It follows that $r^{(2)} \geqslant r + r_{h-2} \geqslant 2r \geqslant (1+1/\alpha) r$. □

Lemma 29. *If R is a left run, $r^{(2)} \geqslant \alpha^\star r$.*

Proof. We treat three cases independently, depending on whether R is a #1-, a #2 or a #3-left run. In each case, we assume that $R^{(1)}$ is a left run, since Lemma 28 already proves that $r^{(2)} \geqslant \alpha^\star r$ when $R^{(1)}$ is a right run.

- ▷ If R is a #1-left run, the run $R = R_{h-2}$ is merged with R_{h-1} and $r_{h-2} < r_h$. Since $R^{(1)}$ is a left run, R_h descends from $R^{(2)}$, and thus $r^{(2)} \geqslant r + r_h \geqslant 2r \geqslant (1+1/\alpha) r$.
- ▷ If R is a #2-left run, the run $R = R_{h-1}$ is merged with R_h and $r < \alpha r_h$. It follows, in that case, that $r^{(2)} \geqslant r^{(1)} = r + r_h \geqslant (1+1/\alpha) r$.
- ▷ If R is a #3-left run, the run $R = R_{h-1}$ is merged with R_h and $r_{h-2} < \alpha r_{h-1}$. Hence, (iii) proves that $(\alpha-1) r \leqslant \alpha r_h$, so that $r^{(2)} \geqslant r^{(1)} = r + r_h \geqslant (1+(\alpha-1)/\alpha) r$. □

Proposition 30. *The algorithm α-MergeSort has the fast-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by α-MergeSort, and let R be a run in $\mathcal{T}$. Lemma 28 shows that $r^{(3)} \geqslant r^{(2)} \geqslant \alpha^* r$ if $R^{(1)}$ is a right run, and Lemma 29 shows that $r^{(3)} \geqslant \alpha^* r^{(1)} \geqslant \alpha^* r$ if $R^{(1)}$ is a left run. □

4.2 Algorithms with the tight middle-growth property

4.2.1 PowerSort

Proposition 31. *The algorithm PowerSort has the tight middle-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by PowerSort and let R be a run in $\mathcal{T}$ at depth at least h. We will prove that $r^{(h)} \geqslant 2^{h-4}$.

If $h \leqslant 4$, the desired inequality is immediate. Then, if $h \geqslant 5$, let n be the length of the array on which $\mathcal{T}$ is induced, and let p and $p^{(h-2)}$ be the respective powers of the runs R and $R^{(h-2)}$. Definition 14 and Lemma 15 prove that $2^{p^{(h-2)}+h-4} \leqslant 2^{p-2} \leqslant 2^{p-2}r < n < 2^{p^{(h-2)}}r^{(h)}$. □

4.2.2 NaturalMergeSort

The algorithm NaturalMergeSort consists in a plain binary merge sort, whose unit pieces of data to be merged are runs instead of being single elements. Thus, we identify NaturalMergeSort with the fundamental property that describes those merge trees it induces.

Definition 32. Let $\mathcal{T}$ be a merge tree induced by NaturalMergeSort, and let R and $\overline{R}$ be two runs that are siblings of each other in $\mathcal{T}$. Denoting by n and $\overline{n}$ the respective numbers of leaves of $\mathcal{T}$ that descend from R and from $\overline{R}$, we have $|n - \overline{n}| \leqslant 1$.

Proposition 33. *The algorithm NaturalMergeSort has the tight middle-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by NaturalMergeSort, let R be a run in $\mathcal{T}$, and let h be its height. We will prove by induction on h that, if $h \geqslant 1$, the run R is an ancestor of at least $2^{h-1} + 1$ leaves of $\mathcal{T}$, thereby showing that $r \geqslant 2^{h-1}$.

First, this is the case if $h = 1$. Then, if $h \geqslant 2$, let R_1 and R_2 be the two children of R. One of them, say R_1, has height $h - 1$. Let n, n_1 and n_2 be the numbers of leaves that descend from R, R_1 and R_2, respectively. The induction hypothesis shows that

$$n = n_1 + n_2 \geqslant 2n_1 - 1 \geqslant 2 \times (2^{h-2} + 1) - 1 = 2^{h-1} + 1,$$

which completes the proof. □

4.2.3 ShiversSort

The algorithm ShiversSort is presented in Algorithm 4. Like adaptive ShiversSort, it relies on the notion of *level* of a run.

Input : Array A to sort
Result : The array A is sorted into a single run. That run remains on the stack.
Note : The height of the stack $\mathcal{S}$ is denoted by h; its i^{th} deepest run by R_i; the length of R_i by r_i. Finally, we set $\ell_i = \lfloor \log_2(r_i) \rfloor$. When two consecutive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, with the run resulting from the merge.

```
1  S ← an empty stack
2  while true:
3      if h ⩾ 1 and ℓ_{h−1} ⩽ ℓ_h:
4          merge the runs R_{h−1} and R_h
5      else if the end of the array has not yet been reached:
6          find a new monotonic run R, make it non-decreasing, and push it onto S
7      else:
8          break
9  while h ⩾ 2:
10     merge the runs R_{h−1} and R_h
```

Algorithm 4: ShiversSort

Our proof is based on the following result, which appears in the proof of [8, Theorem 4.2].

Lemma 34. *Let $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ be a stack obtained while executing ShiversSort. We have $\ell_i \geqslant \ell_{i+1} + 1$ whenever $1 \leqslant i \leqslant h-2$.*

Lemmas 35 and 36 focus on inequalities involving the levels of a given run R belonging to a merge tree induced by adaptive ShiversSort, and of its ancestors. In each case, the stack just before the run R is merged is denoted by $\mathcal{S} = (R_1, R_2, \ldots, R_h)$.

However, and unlike Sections 4.1.3 to 4.1.5, we cannot simulate the merge operations that occur in line 10 as if they had occurred in line 4. Instead, we say that a run R is *rightful* if R and its ancestors are all right runs (i.e., if R belongs to the rightmost branch of the merge tree), and that R is *standard* otherwise.

Lemma 35. *Let R be a run in $\mathcal{T}$, and let $k \geqslant 1$ be an integer. If each of the k runs $R^{(0)}, \ldots, R^{(k-1)}$ is a right run, $\ell^{(k)} \geqslant k-1$.*

Proof. For all $i \leqslant k$, let $u(i)$ be the least integer such that $R_{u(i)}$ descends from $R^{(i)}$. Since the k runs $R^{(0)}, \ldots, R^{(k-1)}$ are right runs, $u(k) < u(k-1) < \ldots < u(1) \leqslant h-1$. Thus, Lemma 34 proves that

$$\ell^{(k)} \geqslant \ell_{u(k)} \geqslant \ell_{u(1)} + (k-1) \geqslant k-1. \qquad \square$$

Lemma 36. *Let R be a run in $\mathcal{T}$, and let $k \geqslant 1$ be an integer. If R is a left run and the $k-1$ runs $R^{(1)}, \ldots, R^{(k-1)}$ are right runs, we have $\ell^{(k)} \geqslant \ell + k - 1$ if $R^{(1)}$ is a rightful run, and $\ell^{(k)} \geqslant \ell + k$ if $R^{(1)}$ is a standard run.*

Proof. First, assume that $k = 1$. If $R^{(1)}$ is rightful, the desired inequality is immediate. If $R^{(1)}$ is standard, however, the left run $R = R_{h-1}$ was merged with the run R_h because $\ell \leqslant \ell_h$. In that case, it follows that $r^{(1)} = r + r_h \geqslant 2^{\ell} + 2^{\ell_h} \geqslant 2^{\ell} + 2^{\ell} = 2^{\ell+1}$, i.e., that $\ell^{(1)} \geqslant \ell + 1$.

Assume now that $k \geqslant 2$. Note that $R^{(k)}$ is rightful if and only if $R^{(1)}$ is also rightful. Then, for all $i \leqslant k$, let $u(i)$ be the least integer such that the run $R_{u(i)}$ descends from $R^{(i)}$. Since $R^{(1)}, \ldots, R^{(k-1)}$ are right runs, $u(k) < u(k-1) < \ldots < u(1) = h-1$.

In particular, let R' be the left sibling of $R^{(k-1)}$: this is an ancestor of $R_{u(k)}$, and the left child of $R^{(k)}$. Consequently, Lemma 34 and applying our study of the case $k = 1$ to the run R' conjointly prove that

- $\ell^{(k)} \geqslant \ell' \geqslant \ell_{u(k)} \geqslant \ell_{u(1)} + k - 1 = \ell + k - 1$ if $R^{(1)}$ and $R^{(k)}$ are rightful;
- $\ell^{(k)} \geqslant \ell' + 1 \geqslant \ell_{u(k)} + 1 \geqslant \ell_{u(1)} + k = \ell + k$ if $R^{(1)}$ and $R^{(k)}$ are standard. $\square$

Proposition 37. *The algorithm ShiversSort has the tight middle-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by ShiversSort and let R be a run in $\mathcal{T}$ at depth at least h. Let $a_1 < a_2 < \ldots < a_k$ be the non-negative integers smaller than h for which $R^{(a_i)}$ is a left run. We also set $a_{k+1} = h$. Lemma 35 proves that $\ell^{(a_1)} \geqslant a_1 - 1$. Then, for all $i < k$, the run $R^{(a_i+1)}$ is standard, since it descends from the left run $R^{(a_k)}$, and thus Lemma 36 proves that $\ell^{(a_{i+1})} \geqslant \ell^{(a_i)} + a_{i+1} - a_i$. Lemma 36 also proves that $\ell^{(a_{k+1})} \geqslant \ell^{(a_k)} + a_{k+1} - a_k - 1$. It follows that $\ell^{(h)} = \ell^{(a_{k+1})} \geqslant h - 2$, and therefore that $r^{(h)} \geqslant 2^{\ell^{(h)}} \geqslant 2^{h-2}$. $\square$

4.3 Algorithms with the middle-growth property

4.3.1 α-StackSort

The algorithm α-StackSort, which predated and inspired its variant α-MergeSort, is presented in Algorithm 5.

Our proof is based on the following result, which appears in [2, Lemma 2].

Lemma 38. *Let $\mathcal{S} = (R_1, R_2, \ldots, R_h)$ be a stack obtained while executing α-StackSort. We have $r_i > \alpha r_{i+1}$ whenever $1 \leqslant i \leqslant h-2$.*

Lemmas 39 and 40 focus on inequalities involving the lengths of a given run R belonging to a merge tree induced by α-StackSort, and of its ancestors. In each case, the stack just before the run R is merged is denoted by $\mathcal{S} = (R_1, R_2, \ldots, R_h)$. Furthermore, like in Section 4.2.3, we say that a run R is *rightful* if R and its ancestors are all right runs, and that R is *standard* otherwise.

Input : Array A to sort, parameter $\alpha > 1$
Result : The array A is sorted into a single run. That run remains on the stack.
Note : The height of the stack $\mathcal{S}$ is denoted by h; its i^{th} deepest run by R_i; the length of R_i by r_i. When two consecutive runs of $\mathcal{S}$ are merged, they are replaced, in $\mathcal{S}$, with the run resulting from the merge.

$\mathcal{S} \leftarrow$ an empty stack
while `true`**:**
  if $h \geqslant 2$ and $r_{h-1} \leqslant \alpha r_h$**:**
    merge the runs R_{h-1} and R_h
  else if the end of the array has not yet been reached**:**
    find a new monotonic run R, make it non-decreasing, and push it onto $\mathcal{S}$
  else:
    break
while $h \geqslant 2$**:**
  merge the runs R_{h-1} and R_h

Algorithm 5: α-StackSort

Lemma 39. *Let R be a run in $\mathcal{T}$, and let k and m be two integers. If k of the $m+1$ runs $R^{(0)}, R^{(1)}, \ldots, R^{(m)}$ are right runs, $r^{(m+1)} \geqslant \alpha^{k-1}$.*

Proof. Let $a_1 < a_2 < \ldots < a_k$ be the k smallest integers such that $R^{(a_1)}, R^{(a_2)}, \ldots, R^{(a_k)}$ are right runs. For all $i \leqslant k-1$, let $S^{\langle i \rangle}$ be the left sibling of $R^{(a_i)}$, and let $u(i)$ be the least integer such that $R_{u(i)}$ descends from $S^{\langle i \rangle}$. Since $R_{u(i)}$ descends from $R^{(a_{i+1})}$, we know that $u(i+1) < u(i)$, so that $u(k) \leqslant u(1) - (k-1)$. Observing that $u(1) \leqslant h-1$ and using Lemma 38 then proves that $r^{(m+1)} \geqslant r_{u(k)} \geqslant \alpha^{k-1} r_{u(1)} \geqslant \alpha^{k-1}$. □

Lemma 40. *Let R be a run in $\mathcal{T}$. If R is a left run and if $R^{(1)}$ is a standard run, $r^{(1)} \geqslant (1+1/\alpha)r$.*

Proof. The left child of $R^{(1)}$ coincides with $R = R_{h-1}$ and, since $R^{(1)}$ is a standard run, $r_{h-1} \leqslant \alpha r_h$. It follows immediately that $r^{(1)} = r_{h-1} + r_h \geqslant (1+1/\alpha)r$. □

Proposition 41. *The algorithm α-StackSort has the middle-growth property.*

Proof. Let $\mathcal{T}$ be a merge tree induced by ShiversSort, let R be a run in $\mathcal{T}$ at depth at least h, and let $\beta = \min\{\alpha, 1+1/\alpha\}^{1/4}$. We will prove that $r^{(h)} \geqslant \beta^h$.

Indeed, let $k = \lceil h/2 \rceil$. We distinguish two cases, which are not mutually exclusive if h is even.

▷ If at least k of the h runs $R, R^{(1)}, R^{(2)}, \ldots, R^{(h-1)}$ are right runs, Lemma 39 shows that

$$r^{(h)} \geqslant \alpha^{k-1} \geqslant \beta^{4(k-1)}.$$

▷ If at least k of the h runs $R, R^{(1)}, R^{(2)}, \ldots, R^{(h-1)}$ are left runs, let $a_1 < a_2 < \ldots < a_k$ be the k smallest integers such that $R^{(a_1)}, R^{(a_2)}, \ldots, R^{(a_k)}$ are left runs. When $j < k$, the run $R^{(a_j+1)}$ is standard, since it descends from the left run $R^{(a_k)}$. Therefore, due to Lemma 40, an immediate induction on j shows that $r^{(a_j+1)} \geqslant (1+1/\alpha)^j$ for all $j \leqslant k-1$. It follows that

$$r^{(h)} \geqslant r^{(a_{k-1}+1)} \geqslant (1+1/\alpha)^{k-1} \geqslant \beta^{4(k-1)}.$$

Consequently, $r^{(h)} \geqslant \beta^{4(k-1)} \geqslant \beta^{2h-4} \geqslant \beta^h$ when $h \geqslant 4$, whereas $r^{(h)} \geqslant 1 = \beta^h$ when $h = 0$ and $r^{(h)} \geqslant 2 \geqslant 1 + 1/\alpha \geqslant \beta^h$ when $1 \leqslant h \leqslant 3$. □

5 TimSort's update policy

In this Section, we study TimSort's merging routine as it was implemented in [7, 28, 29, 30]. This routine aims at merging consecutive non-decreasing runs A and B. Although similar to the description we gave in Section 2, this implementation does not explicitly rely on $\mathbf{t}$-galloping, but only on 0-galloping, and it includes complicated side-cases or puzzling low-level design choices that we neglected on purpose when proposing a simpler notion of $\mathbf{t}$-galloping.

For instance, one may be surprised to observe that the number of consecutive elements coming from A (or B) needed to trigger the galloping mode depends on whether elements from a previous dual run were discovered via galloping or naïvely. Another peculiar decision is that, when the galloping mode is triggered by a long streak of elements coming *from* B, galloping is first used to count subsequent elements coming *from* A; fortunately, it is then also used to count elements coming from B, which approximately amounts to using 1-galloping instead of 0-galloping. Finally, instances of the galloping mode are launched in pairs, and the algorithm infers that galloping was efficient as soon as one of these two instances was efficient; this makes the update rule easy to fool.

Algorithm 6 consists in a pseudo-code transcription of this merging routine, where we focus only on the number of comparisons performed. Thus, we disregard both the order in which comparisons are performed and element moves. For the sake of simplicity,[2] we also deliberately simplified the behaviour of TimSort's routine, by overlooking some side cases, listed in Section 5.1 below.

Algorithm 6 explicitly relies on the values of the (global) parameter $\mathbf{t}$ and on a (global) variable Status, implicit in the implementations of TimSort [7, 28, 29, 30]. In practice, the parameter $\mathbf{t}$ is often initially set to $\mathbf{t} = 7$, although other initial values might be chosen; it is subsequently updated by each instance of the merging routine, and is *not* reset to $\mathbf{t} = 7$ between two such instances. In what follows, let $\mathbf{t}_{\mathsf{init}}$ denote the initial value of $\mathbf{t}$.

The two main reasons behind introducing this merging routine are as follows: (i) without incurring a super-linear overhead to do so, one wants to (ii) merge arrays with a bounded number of values in linear time. Below, we provide counter-examples to both these assertions.

5.1 Caveat

Providing counter-examples to the goals (i) and (ii) requires paying attention to some low-level details we might wish to skip. The first one is that, as indicated in Section 1, when decomposing an array into runs, TimSort makes sure that these runs are *long enough*, and extends them if they are too short. In practice, given a size ms that is considered large enough in a given implementation, both our counter-examples consist only of runs of length at least ms. Such a size may vary between two implementations (from 32 in Java to 64 in Python), which is why we kept it as a parameter.

The second one is that Algorithm 6 represents a *simplified* version of TimSort's actual merging routine. Here are the differences between Algorithm 6 and that routine:

1. The first element of A in S_2 and the last element of B in $S_{\sigma-1}$ have already been identified in line 1. Thus, TimSort's routine tries to save one comparison per merge by not rediscovering these elements naïvely during the last calls to MergeStep in lines 8 or 13. Nothing is changed if these elements are discovered through galloping.

2. In line 24, TimSort's routine actually uses 1-galloping instead of 0-galloping, because it loses one step trying to discover a streak of 0 elements from A.

3. As mentioned in Section 2, TimSort's galloping routine may actually go faster than forecast when discovering a streak of x elements from a run in which less than $2x$ elements remain to be discovered.

In practice, the first difference is harmless for our study: it concerns only a constant number of comparisons per run merge, i.e., $\mathcal{O}(n)$ comparisons in total when sorting an array of length n. The second difference is also harmless, because it just makes TimSort's routine *more expensive* than what we will claim below. The third difference is tackled in two different ways: in Section 5.2, we simply avoid falling in that case, and in Section 5.3, we will brutally

[2] Counting precisely comparisons performed by TimSort's merging routine required 166 Java code lines [17].

Input : Non-decreasing runs A and B to merge, of lengths a and b
Result : The runs A and B are merged into a single run C.
Note : Let $S_1, S_2, \ldots, S_\sigma$ be the non-decreasing dual runs of the sub-array spanned by A and B. The run A (resp., B) contains a_i (resp., b_i) elements whose value belongs to S_i.

use 0-galloping to discover a_1 and b_σ
Status $\leftarrow$ naïve
if $a - a_1 \leqslant b - b_\sigma$**:** LRMerge ▷ C is discovered from left to right
else: RLMerge ▷ C is discovered from right to left

Function LRMerge**:**
  Naïve$(0, b_1, \mathbf{t}, \mathbf{t}+1)$
  for $i = 2, 3, \ldots, \sigma - 2$**:** MergeStep$(a_i, b_i)$
  if $\sigma \geqslant 3$ and $a_\sigma \geqslant 2$**:** MergeStep$(a_{\sigma-1}, b_{\sigma-1})$
  else if $\sigma \geqslant 3$**:** MergeStep$(a_{\sigma-1}, 0)$

Function RLMerge**:**
  Naïve$(a_\sigma, b_{\sigma-1}, \mathbf{t}+1, \mathbf{t}+1)$
  for $i = 2, 3, \ldots, \sigma - 2$**:** MergeStep$(a_{\sigma+1-i}, b_{\sigma-i})$
  if $\sigma \geqslant 3$ and $a_2 \geqslant 2$**:** MergeStep$(a_2, 0)$

Function MergeStep(a, b)**:**
  if Status $=$ fail**:** $\mathbf{t} \leftarrow \max\{2, \mathbf{t}+1\}$
  else if Status $=$ success**:** $\mathbf{t} \leftarrow \mathbf{t} - 1$
  if Status $=$ naïve**:** Naïve$(a, b, \mathbf{t}, \mathbf{t})$
  else if Status $=$ fail**:** Naïve$(a, b, \mathbf{t}+1, \mathbf{t})$
  else if Status $=$ start**:** Gallop$(a+1, b)$
  else: Gallop(a, b)

Function Naïve(a, b, t_1, t_2)**:**
  use t_1-galloping to discover a
  if $a \geqslant t_1$**:** use 0-galloping to discover b
  else: use t_2-galloping to discover b
  if $a \geqslant t_1 + 7$ or ($a \geqslant t_1$ and $b \geqslant 8$) or $b \geqslant t_2 + 8$**:** Status $\leftarrow$ success
  else if $a \geqslant t_1$ or $b \geqslant t_2 + 1$**:** Status $\leftarrow$ fail
  else if $b = t_2$**:** Status $\leftarrow$ start
    cancel the last comparison
  else: Status $\leftarrow$ naïve

Function Gallop(a, b)**:**
  use 0-galloping to discover a and b
  if $a \geqslant 8$ or $b \geqslant 8$**:** Status $\leftarrow$ success
  else: Status $\leftarrow$ fail

Algorithm 6: TimSort's merging routine. This routine is best modelled by using a variable Status that may take four values: naïve if launching a $\mathbf{t}$-gallop (i.e., starting naïvely) is requested; success if the last gallop was successful (i.e., saved comparisons); fail if the last gallop was unsuccessful (i.e., wasted comparisons); start if a 0-gallop is to be launched. A few side cases, having little influence on the number of comparisons but not on the dynamics of $\mathbf{t}$, are overlooked here and discussed in Section 5.1.

take 0 as an under-estimation of the merge cost of a galloping phase — an under-estimation that remains valid regardless of how the galloping is performed.

Finally, and although we are referring to *TimSort*'s merging routine, we would also like to prove that this routine suffers the same shortcomings with many relevant algorithms. Thus, we made sure that all algorithms would induce the same merge tree on our counterexample, thereby having the same behaviour.

5.2 Super-linear overhead

One desirable safeguard, when using TimSort's routine instead of a naïve merging routine, is that the overhead incurred by this change should be small. In other words, the user should never be heavily penalised for using TimSort's routine.

More precisely, when using an algorithm $\mathcal{A}$ to sort an array A of length n, let U_A be the number of comparisons performed by $\mathcal{A}$ if it relies on a naïve merging routine, and let V_A be the number of comparisons performed by $\mathcal{A}$ if it uses TimSort's routine. Given that the *worst-case* complexity of a reasonable sorting algorithm is $\mathcal{O}(n\log(n))$, one may reasonably expect inequalities of the form $\mathsf{V}_A \leqslant \mathsf{U}_A + o(n\log(n))$. Below, we prove that such inequalities are invalid, meaning that using TimSort's routine may cost a positive fraction of the total merge cost of the algorithm, even in cases where this merge cost is already quite bad.

Proposition 42. *Let $\mathcal{A}$ be one of the algorithms studied in Section 4. Let U_A be the number of comparisons performed by $\mathcal{A}$ to sort an array A when using a naïve merging routine, and let V_A be the number of comparisons performed by $\mathcal{A}$ when using TimSort's routine. There exists an array A, whose length n may be arbitrarily large, for which $\mathsf{V}_A - \mathsf{U}_A \in \Omega(n\log(n))$.*

Proof. What dictates the dynamics of Algorithm 6, when merging runs A and B into a new run C, is not the precise values of those elements of A and B, but only the lengths of the consecutive streaks of elements of C coming either from A or from B. Thus, we only need to control these lengths.

Based on this remark, we use two building bricks. The first brick, represented in Figure 5 (top), aims at changing $\mathbf{t}$, initially equal to $\mathbf{t}_{\mathsf{init}}$, to let it reach the value $\mathbf{t} = 5$; it is used once, as the first merge performed in TimSort. The second brick, represented in Figure 5 (bottom), aims at maximising the number of comparisons performed, without changing the value of $\mathbf{t}$: if the merge started with $\mathbf{t} = 5$, it shall end with $\mathbf{t} = 5$.

Our first brick is built by merging two runs R and R' with a given length $\ell \geqslant 8\mathbf{t}_{\mathsf{init}} + 19$, as indicated in Figure 5 (top). When R and R' are being merged, both lengths r_1 and r'_σ are zero, and the function LRMerge is called. First encountering $\mathbf{t}_{\mathsf{init}} + 8$ consecutive elements from R triggers a call to Gallop; this function is successfully called $\mathbf{t}_{\mathsf{init}}$ times, which brings down $\mathbf{t}$ to 0. Then, the nine following elements from R (and nine elements from R') result in unsuccessfully calling Gallop several times, which gradually raises up $\mathbf{t}$ to 5. The padding area has no influence on $\mathbf{t}$; its only purpose is to ensure that R and R' have the desired length ℓ.

Our second brick is built by merging two runs S and S' with a given length $11m$, as indicated in Figure 5 (bottom). Provided that the parameter $\mathbf{t}$ has been set to 5, encountering 8 consecutive elements coming from S triggers a call to Gallop. This call is successful, because it helps us to discover 8 consecutive elements from S', and it is immediately followed by an unsuccessful call to Gallop, which only helps us to discover streaks of length 3. After these two calls, $\mathbf{t}$ is still equal to 5, which allows us to repeat such sequences of calls. Hence, Algorithm 6 uses 23 comparisons to discover each block $8 + 3 + 8 + 3$ elements — one more comparison than needed. Thus, Algorithm 6 uses a total of $23m - 3$ element comparisons to merge two runs S and S' of length $11m$, which is more than the $22m - 1$ comparisons that a naïve routine would use.

Finally, our array A is built as follows. First, we choose a base length $\ell \geqslant \max\{8\mathbf{t}_{\mathsf{init}} + 19, \mathsf{ms}\}$ divisible by 11, and we set $k = \lfloor \log_2(\ell) \rfloor$. The array will be a permutation of $\{1, 2, \ldots, 2^k\ell\}$, initially subdivided in 2^k runs of length ℓ. This ensures that the merge tree induced on A by each of the algorithms mentioned in Section 2 is perfectly balanced: each run at height h in the tree has length $2^h\ell$, and A has length $n = 2^k\ell$.

Once this merge tree is fixed, we give a value between 1 and $2^h\ell$ to each element of A, by using the following top-down procedure. When a run $\overline{S}$ of length $2^{h+1}\ell$ results from merging two runs S and S' of length $2^h\ell$, we assign values of $\overline{S}$ to either S or S' according to our second base brick: the first value of $\overline{S}$ comes from S', the second one from S, the third one from S', the next 8 values come from S, and so on. The only exception to this

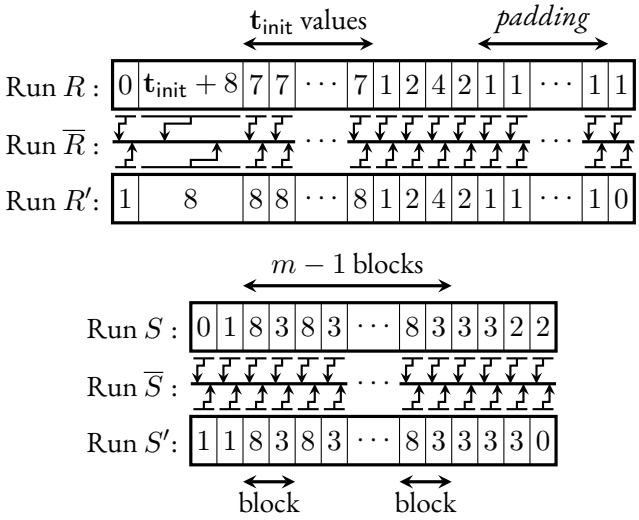

Figure 5: Base bricks of our construction. The diagrams should be read as follows. The run $\overline{R}$ consists in 1 element from R', then $\mathbf{t}_{\mathsf{init}}+8$ elements from R, 8 elements from R', 7 elements from R, 8 elements from R', 7 elements from R, …, and finally 1 element from R. The runs R and R' both have a length $\ell \geqslant 8\mathbf{t}_{\mathsf{init}}+19$, and include a padding area of length $\ell-(8\mathbf{t}_{\mathsf{init}}+18)$. Similarly, $\overline{S}$ consists in 1 element from S', then 1 element from S, 1 element from S', 8 elements from S, 8 elements from S', 3 elements from S, …, and finally 2 elements from S. Both runs S and S' have length $11m$.

rule is the first merge: here, we assign according to our first base brick the values of the run $\overline{R}$ to the two runs R and R' it results from.

Doing so ensures that using TimSort's merging routine will perform at least 0 comparison for its first merge, and $23\times 2^h\ell/11-3$ comparisons for each subsequent merge between runs at height h. There are $2^{k-1}-1$ such merges for $h=0$, and 2^{k-h-1} such merges for each height h such that $1\leqslant h\leqslant k-1$. Hence, a total of

$$\mathsf{V}_A \geqslant (2^{k-1}-1)\left(\frac{23\ell}{11}-3\right)+\sum_{h=1}^{k-1}2^{k-h-1}\left(\frac{23\times 2^h\ell}{11}-3\right)=\frac{23kn}{22}-\frac{23\ell}{11}-3\times 2^k+6$$

comparisons are performed. By contrast, using a naïve merging strategy would lead to performing only $\mathsf{U}_A=kn-2^k+1$ comparisons.

Finally, the inequalities $\ell^2 \geqslant 2^k\ell = n \geqslant 2^{2k} \geqslant 4\ell^2$ prove that both ℓ and 2^k are $\Theta(\sqrt{n})$, whereas $k\sim\log_2(n)/2$. Hence, $\mathsf{U}_A\sim n\log_2(n)/2$ and $\mathsf{V}_A\geqslant 23n\log_2(n)/44+o(n\log(n))$. □

5.3 Sorting arrays with three values in super-linear time

TimSort's merging routine was invented precisely with the goal of decreasing the number of comparisons performed when sorting arrays with few values. In particular, sorting arrays of length n with a constant number of values should require only $\mathcal{O}(n)$ comparisons. Like in Section 5.2, we prove below that this is not the case.

Proposition 43. *Let $\mathcal{A}$ be one of the algorithms studied in Section 4. If $\mathcal{A}$ uses TimSort's routine for merging runs, it may require up to $\Omega(n\log(n))$ element comparisons to sort arrays of length n with $\sigma=3$ distinct values.*

Proof. Like for Proposition 42, we explicitly build the array A by using building blocks that we assemble according to the description of Figure 6. Below, we fix an arbitrarily large parameter p, and we set $\mathsf{L}_p=(4\mathsf{ms}+1)(2^p+p-1+\mathbf{t}_{\mathsf{init}})+12$. We also set $\mathsf{R}(x)=2^{\lfloor\log_2(x)\rfloor}$ for all $x\geqslant 1$, and $t_k=k+s_2(k)+\mathbf{t}_{\mathsf{init}}$ for all $k\geqslant 1$, where $s_2(k)$ is the sum of the binary digits of k.

The array A that we build is divided in 2^p blocks $\mathbf{X}(k)$ of length $2\mathsf{L}_p$. Each block $\mathbf{X}(k)$ is subdivided into three parts $\mathbf{X}_1(k)$, $\mathbf{X}_2(k)$ and $\mathbf{X}_3(k)$, whose lengths $x_1(k)$, $x_2(k)$ and $x_3(k)$ obey the relations $x_1(k)=\mathsf{ms}\,\mathsf{R}(t_k)$

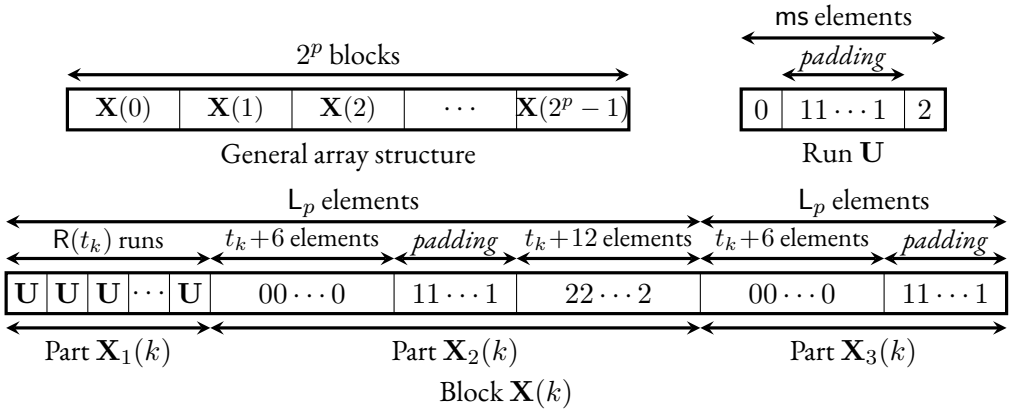

Figure 6: Bad array for TimSort's galloping update policy, containing only values 0, 1 and 2.

and $x_1(k) + x_2(k) = x_3(k) = \mathsf{L}_p$. The part $\mathbf{X}_1(k)$ itself is subdivided into $\mathsf{R}(t_k)$ runs $\mathbf{U}$ of length ms; each of the parts $\mathbf{X}_2(k)$ and $\mathbf{X}_3(k)$ consists of just one run.

By construction, $t_k \leqslant 2^p - 1 + s_2(2^p - 1) + \mathbf{t}_{\mathsf{init}} = 2^p + (p-1) + \mathbf{t}_{\mathsf{init}}$ whenever $0 \leqslant k \leqslant 2^p - 1$, and $x + 1 \leqslant \mathsf{R}(x) \leqslant 2x$ for all integers $x \geqslant 1$. It follows that

$$x_2(k) - x_1(k) = \mathsf{L}_p - 2\mathsf{ms}\,\mathsf{R}(t_k) \geqslant (4\mathsf{ms}+1)t_k + 12 - 4\mathsf{ms}\,t_k = t_k + 12 \geqslant 0.$$

Hence, the algorithm $\mathcal{A}$ proceeds in three phases, interleaved with each other: (i) it sorts each part $\mathbf{X}_1(k)$ by recursively merging the runs $\mathbf{U}$ of which $\mathbf{X}_1(k)$ consists, following a perfectly balanced binary merge tree with $\mathsf{R}(t_k)$ leaves — let $\mathbf{X}_1(k)$ abusively denote the resulting run; (ii) it merges $\mathbf{X}_1(k)$ with $\mathbf{X}_2(k)$, and merges the resulting run (say, $\mathbf{X}_{1+2}(k)$) with $\mathbf{X}_3(k)$; (iii) it merges the sorted blocks $\mathbf{X}(k)$ with each other, again following a perfectly balanced binary merge tree with 2^p leaves. Moreover, merges are performed according to a post-order traversal of the merge tree.

For instance, the merge tree obtained when $p = 2$ and $\mathbf{t}_{\mathsf{init}} = 7$ is presented in Figure 7; its inner nodes are labelled chronologically: the node labelled k results from the k^{th} merge that $\mathcal{A}$ performs.

Once the order in which $\mathcal{A}$ performs merges is known, Algorithm 6 allows us to track the dynamics of the parameter $\mathbf{t}$. For the ease of the explanation, let $\mathbf{X}(2^h, \ell)$ denote the run obtained by merging the blocks $\mathbf{X}(2^h\ell), \mathbf{X}(2^h\ell + 1), \ldots, \mathbf{X}(2^h(\ell+1) - 1)$. The key invariant of our construction is two-fold:

1. the parameter $\mathbf{t}$ is equal to the integer $t_k = k + s_2(k) + \mathbf{t}_{\mathsf{init}}$ just before $\mathcal{A}$ starts sorting a block $\mathbf{X}(k)$, and it is equal to $t_k + 2$ just after $\mathbf{X}(k)$ has been sorted;

2. the parameter $\mathbf{t}$ is equal to the integer $u_{h,\ell} = t_{2^h\ell} + 2^{h+1} + 2$ just before $\mathcal{A}$ starts merging two runs $\mathbf{X}(2^h, \ell)$ and $\mathbf{X}(2^h, \ell+1)$, and it is equal to $u_{h,\ell} - 1$ just after these runs have been merged.

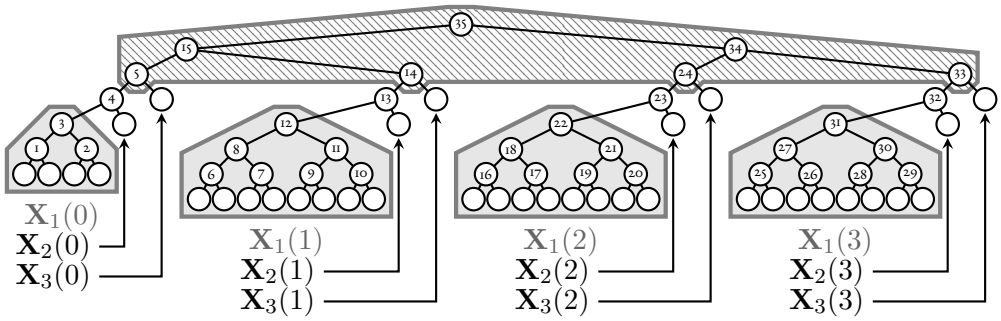

Figure 7: Merge tree induced by the algorithm $\mathcal{A}$ on the array A. The four balanced gray sub-trees result in sorting the parts $\mathbf{X}_1(k)$; the balanced hatched sub-tree results in sorting A itself once each block $\mathbf{X}(k)$ is sorted. Each inner node is labelled k if it results from the k^{th} merge that $\mathcal{A}$ performs.

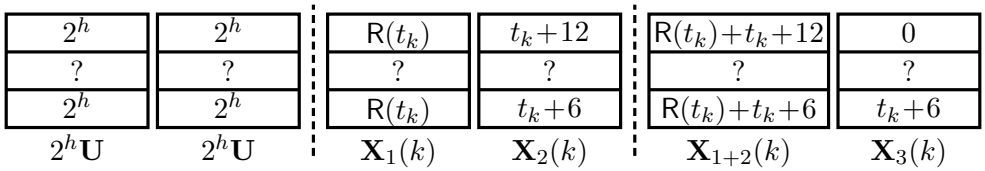

Figure 8: Runs merged while sorting $\mathbf{X}(k)$. Each run is represented by a 3-row array: the integer in the lowest (resp., highest) cell is the number of elements with value 0 (resp., 2) of the run. Elements with value 1 serve as padding runs: counting them exactly is irrelevant. From left to right, we see the merges between (i) two runs that each result from merging 2^h runs $\mathbf{U}$; (ii) the sorted part $\mathbf{X}_1(k)$ with $\mathbf{X}_2(k)$; (iii) $\mathbf{X}_{1+2}(k)$ and $\mathbf{X}_3(k)$.

To prove this invariant, we first show that, if $\mathbf{t} = t_k$ when $\mathcal{A}$ starts sorting $\mathbf{X}(k)$, Algorithm 6 will keep calling the function LRMerge in line 3. Then, the only time at which $\mathbf{t}$ may be changed is just after discovering b_0 by using $\mathbf{t}$-galloping: $\mathbf{t}$ decreases if $b_0 \geqslant \mathbf{t}+9$, it increases if $\mathbf{t}+8 \geqslant b_0 \geqslant \mathbf{t}+2$, and does not vary if $\mathbf{t}+1 \geqslant b_0$.

Now, let us consider each merge performed to sort $\mathbf{X}(k)$; the composition of the intermediate runs obtained while doing so is shown in Figure 8. First, while merging sorted clusters of 2^h contiguous runs $\mathbf{U}$, we have $a - a_2 = b - b_2 = (\mathsf{ms}-1)2^h$ and $b_0 = 2^h \leqslant \mathsf{R}(t_k)/2 \leqslant t_k = \mathbf{t}$. Then, when merging $\mathbf{X}_1(k)$ with $\mathbf{X}_2(k)$, we have $a - a_0 \leqslant a = x_1(k) \leqslant x_2(k) - (t_k+12) = b - b_2$ and $\mathbf{t}+8 \geqslant b_0 = \mathbf{t}+6 \geqslant \mathbf{t}+2$, which leads to increasing $\mathbf{t}$. Finally, when merging $\mathbf{X}_{1+2}(k)$ with $\mathbf{X}_3(k)$, we have $a - a_0 = \mathsf{L}_p - a_0 \leqslant \mathsf{L}_p = b - b_2$ and $\mathbf{t}+8 \geqslant b_0 = \mathbf{t}+5 \geqslant \mathbf{t}+2$, which leads to increasing $\mathbf{t}$ once more. It follows, as promised, that $\mathbf{t} = t_k + 2$ just after $\mathcal{A}$ has sorted $\mathbf{X}(k)$.

The second step towards proving the invariant consists in showing that, if $\mathbf{t} = u_{h,\ell}$ when $\mathcal{A}$ starts merging two runs $\mathbf{X}(2^h,\ell)$ and $\mathbf{X}(2^h,\ell+1)$, it decreases $\mathbf{t}$ once. In other words, we shall prove that either Algorithm 6 calls the function LRMerge in line 3 and $b_0 \geqslant u_{h,\ell}+9$, or it calls RLMerge in line 4 and $a_2 \geqslant u_{h,\ell}+9$. In practice, we will simply prove that $\min\{b_0, a_2\} \geqslant u_{h,\ell+9}$.

Indeed, observe that ℓ is even, and thus that $t_{2^h\ell+i} = t_{2^h\ell} + i + s_2(i)$ whenever $0 \leqslant i < 2^{h+1}$. Thus, when $2^h\ell \leqslant k < 2^h(\ell+2)$, the run $\mathbf{X}(k)$ contains $\mathsf{R}(t_k) + 2t_k + 12 \geqslant 3t_k + 13 \geqslant 2t_{2^h\ell} + 13$ elements with value 0 and $\mathsf{R}(t_k) + t_k + 12 \geqslant 2t_k + 13 \geqslant 2t_{2^h\ell} + 13$ elements with value 2. It follows that $\min\{a_2, b_0\} \geqslant 2^h(t_{2^h\ell}+13) \geqslant (t_{2^h\ell}+11) + 2^h \times 2 = u_{h,\ell}+9$.

We prove now our invariant by induction on the number of merges performed by the algorithm $\mathcal{A}$: we distinguish five types of merges:

- ▷ When sorting the block $\mathbf{X}(0)$, the algorithm starts with a parameter $\mathbf{t} = \mathbf{t}_{\mathsf{init}}$.
- ▷ When sorting a block $\mathbf{X}(k)$, where k is odd, it has just finished merging the block $\mathbf{X}(k-1)$, leaving us with a parameter $\mathbf{t} = t_{k-1} + 2 = t_k$.
- ▷ When sorting a block $\mathbf{X}(2^hk)$, where k is odd and $h \geqslant 1$, it has just finished merging the runs $\mathbf{X}(2^{h-1}, 2k-2)$ and $\mathbf{X}(2^{h-1}, 2k-1)$, leaving us with a parameter

$$\mathbf{t} = u_{h-1,2k-2} - 1 = t_{2^h(k-1)} + 2^h + 1 = 2^h(k-1) + s_2(k-1) + 2^h + 1 = t_{2^hk}.$$

- ▷ When merging two runs $\mathbf{X}(2^h, k)$ and $\mathbf{X}(2^h, k+1)$, where k is even and $h = 0$, it has just finished sorting the block $\mathbf{X}(k+1)$, leaving us with a parameter $\mathbf{t} = t_{k+1} + 2 = t_k + 4 = u_{0,k}$.
- ▷ When merging two runs $\mathbf{X}(2^h, k)$ and $\mathbf{X}(2^h, k+1)$, where k is even and $h \geqslant 1$, it has just finished merging the runs $\mathbf{X}(2^{h-1}, 2k+2)$ and $\mathbf{X}(2^{h-1}, 2k+3)$, leaving us with a parameter

$$\mathbf{t} = u_{h-1,2k+2} - 1 = t_{2^h(k+1)} + 2^h + 1 = 2^h(k+1) + s_2(k+1) + 2^h + 1 = u_{h,k}.$$

Equipped with this invariant, we can finally compute a lower bound on the number of comparisons that $\mathcal{A}$ performs. More precisely, we will count only comparisons performed naïvely while sorting parts $\mathbf{X}_1(k)$. When

sorting such a part, we recursively merge sorted clusters of 2^h runs $\mathbf{U}$. To do so, we call the LRMerge function, and naïvely discover $b_0 = 2^h$ elements with value 0 in the right run we are merging; this is half of all the element with value 0 in our two clusters. Thus, each of the $\mathsf{R}(t_k)$ elements of $\mathbf{X}_1(k)$ with value 0 is merged $\log_2(\mathsf{R}(t_k))$ times with another cluster of runs $\mathbf{U}$, and is naïvely discovered $\log_2(\mathsf{R}(t_k))/2$ times on average, costing one additional comparison each time. This makes a total of $\mathsf{R}(t_k)\log_2(\mathsf{R}(t_k))/2$ such comparisons to sort $\mathbf{X}_1(k)$, and at least

$$\begin{aligned}\mathsf{C}_p = \sum_{k=2^{p-1}}^{2^p-1} \mathsf{R}(t_k)\log_2(\mathsf{R}(t_k))/2 &\geqslant \sum_{k=2^{p-1}}^{2^p-1} \mathsf{R}(k+1)\log_2(\mathsf{R}(k+1))/2 \\ &\geqslant 2^{p-1}\mathsf{R}(2^{p-1}+1)\log_2(\mathsf{R}(2^{p-1}+1))/2 = 2^{2p-2}p\end{aligned}$$

comparisons to sort the array A itself. Observing that A is of length $n = 2^{p+1}\mathsf{L}_p \sim (4\mathsf{ms}+1)2^{2p+1}$ proves that $n\log_2(n) \sim (4\mathsf{ms}+1)2^{2p+2}p \in \mathcal{O}(\mathsf{C}_p)$, which completes the proof. □

In spite of this negative result, we can still prove that TimSort's routine and update strategy is harmless when sorting arrays with only two values, thereby making our above construction somehow *as simple as possible*.

Proposition 44. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. If $\mathcal{A}$ uses TimSort's actual routine (including the heuristics for updating the parameter $\mathbf{t}$), it requires $\mathcal{O}(n)$ element comparisons to sort arrays of length n with two values.*

Proof. When merging two runs containing only $\sigma = 2$ values, Algorithm 6 just uses twice 0-galloping and once $(\mathbf{t}+1)$-galloping, and stops without updating $\mathbf{t}$. Thus, up to wasting a maximum of $\mathbf{t}+1$ comparisons per merge, $\mathcal{A}$ keeps using 0-galloping, and Theorem 7 proves that doing so requires only $\mathcal{O}(n)$ comparisons in total to sort arrays of length n with 2 values. □

6 Refined complexity bounds

One weakness of Theorem 7 is that it cannot help us to distinguish the complexity upper bounds of those algorithms that have the middle-growth property, although the constants hidden in the $\mathcal{O}$ symbol could be dramatically different. Below, we study these constants, and focus on upper bounds of the type $\mathsf{c}n\mathcal{H}^* + \mathcal{O}(n)$ or $\mathsf{c}n(1+o(1))\mathcal{H}^* + \mathcal{O}(n)$.

Since sorting arrays of length n requires at least $\log_2(n!) = n\log_2(n) + \mathcal{O}(n)$ comparisons in general, and since $\mathcal{H}^* \leqslant \log_2(n)$ for all arrays, we already know that $\mathsf{c} \geqslant 1$ for any such constant c. Below, we focus on finding matching upper bounds in two regimes: first using a fixed parameter $\mathbf{t}$, thereby obtaining a constant $\mathsf{c} > 1$, and then letting $\mathbf{t}$ depend on the lengths of those runs that are being merged, in which case we reach the constant $\mathsf{c} = 1$.

Inspired by the success of Theorem 6, which states that algorithms with the tight middle-growth property sort arrays of length n by using only $\mathsf{c}n\log_2(n) + \mathcal{O}(n)$ element comparisons with $\mathsf{c} = 1$, we focus primarily on that property, while not forgetting other algorithms that also enjoyed $n\mathcal{H} + \mathcal{O}(n)$ or $n\log_2(n) + \mathcal{O}(n)$ complexity upper bounds despite not having the tight middle-growth property.

6.1 Fixed parameter

Lemma 45. *Let $\mathcal{T}$ be a merge tree induced by a stable algorithm on some array A of length n with σ dual runs $S_1, S_2, \ldots, S_\sigma$. Consider a fixed parameter $\mathbf{t} \geqslant 0$, a real number $u > 1$, and some index $i \leqslant \sigma$. Then, for all $h \geqslant 0$, let $\mathcal{T}_h$ be a set of pairwise incomparable nodes of $\mathcal{T}$ (i.e., no node of $\mathcal{T}_h$ descends from another one) such that each run R in $\mathcal{T}_h$ is of length $r \geqslant u^h(\mathbf{t}+1)n/s_i$. We have*

$$\sum_{R\in\mathcal{T}_{\geqslant 0}} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) \leqslant \frac{9u}{u-1}s_i,$$

where $\mathcal{T}_{\geqslant 0}$ denotes the union of the sets $\mathcal{T}_h$.

Proof. For each integer $h \geqslant 0$, let $\mathsf{C}(h) = \sum_{R\in\mathcal{T}_h} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i})$. Let also $f\colon x \mapsto \mathbf{t} + 2 + 2\log_2(x+1)$ and $g\colon x \mapsto x\, f(s_i/x)$. Both functions f and g are concave and increasing on $(0, +\infty)$.

Then, let $v = \mathbf{t} + 1$. Since $\mathcal{T}_h$ consists of pairwise incompatible runs, we have $\sum_{R\in\mathcal{T}_h} r_{\to i} \leqslant s_i$ and $u^h v n|\mathcal{T}_h|/s_i \leqslant \sum_{R\in\mathcal{T}_h} \leqslant n$, i.e., $|\mathcal{T}_h| \leqslant s_i/(u^h v)$. It follows that

$$\begin{aligned}
\mathsf{C}(h) &\leqslant \sum_{R\in\mathcal{T}_h} f(r_{\to i}) \leqslant |\mathcal{T}_h| f\Big(\sum_{R\in\mathcal{T}_h} r_{\to i}/|\mathcal{T}_h|\Big) \leqslant g(|\mathcal{T}_h|) \\
&\leqslant g\Big(\frac{s_i}{u^h v}\Big) = \frac{s_i}{u^h v} f(u^h v) \\
&\leqslant \frac{s_i}{u^h v}(\mathbf{t} + 2 + 2\log_2(2^{v+1}u^h)) = \frac{s_i}{u^h v}(3v + 3 + 2h\log_2(u)) \\
&\leqslant \frac{s_i}{u^h}(6 + 2h\log_2(u)).
\end{aligned}$$

Denoting the latter expression by $\mathsf{C}_+(h)$ and observing that $\log_2(u) \leqslant 3(u-1)/2$ for all $u > 1$, we conclude that

$$\sum_{R\in\mathcal{T}_{\geqslant 0}} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) = \sum_{h\geqslant 0} \mathsf{C}(h) \leqslant \sum_{h\geqslant 0} \mathsf{C}_+(h) = \frac{u s_i}{u-1}\Big(6 + 2\frac{\log_2(u)}{u-1}\Big) \leqslant \frac{9u}{u-1} s_i. \qquad \square$$

Theorem 46. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the tight middle-growth property. For each parameter $\mathbf{t} \geqslant 0$, if $\mathcal{A}$ uses the $\mathbf{t}$-galloping routine for merging runs, it requires at most*

$$(1 + 1/(\mathbf{t}+3))n\mathcal{H}^* + \log_2(\mathbf{t}+1)n + \mathcal{O}(n)$$

element comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$.

Proof. Let us follow a variant of the proof of Theorem 7. Let γ be the integer mentioned in the definition of the statement "$\mathcal{A}$ has the tight middle-growth property", let $\mathcal{T}$ be the merge tree induced by $\mathcal{A}$ on an array A of length n, and let $s_1, s_2, \ldots, s_\sigma$ be the lengths of the dual runs of A. Like in the proof of Theorem 7, we just need to prove that

$$\sum_{R\in\mathcal{T}} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) \leqslant (1 + 1/(\mathbf{t}+3))\log_2(n/s_i)s_i + \log_2(\mathbf{t}+1)s_i + \mathcal{O}(s_i)$$

for all $i \leqslant \sigma$.

Let $\mathcal{R}_h$ be the set of runs at height h in $\mathcal{T}$. By construction, no run in $\mathcal{R}_h$ descends from another one, which, like in Lemma 45, proves that $\sum_{R\in\mathcal{R}_h} r_{\to i} \leqslant s_i$. Thus, if we set

$$\mathsf{C}_{\mathbf{t}}(h) = \sum_{R\in\mathcal{R}_h} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}),$$

it follows that

$$\mathsf{C}_{\mathbf{t}}(h) \leqslant (1 + 1/(\mathbf{t}+3)) \sum_{R\in\mathcal{R}_h} r_{\to i} \leqslant (1 + 1/(\mathbf{t}+3))s_i$$

for all $h \geqslant 0$.

Now, let $\mu = \lceil \log_2((\mathbf{t}+1)n/s_i)\rceil + \gamma$, and let $\mathcal{T}_h = \mathcal{R}_{h+\mu}$. By construction, each run R belonging to the set $\mathcal{T}_h = \mathcal{R}_{h+\mu}$ is of length $r \geqslant 2^{h+\mu-\gamma} \geqslant 2^h(\mathbf{t}+1)n/s_i$. Thus, applying Lemma 45 to $u = 2$ indicates that $\sum_{h\geqslant\mu} \mathsf{C}_{\mathbf{t}}(h) \leqslant 18 s_i = \mathcal{O}(s_i)$, and we conclude as desired that

$$\sum_{h\geqslant 0} \mathsf{C}_{\mathbf{t}}(h) \leqslant (1 + 1/(\mathbf{t}+3))\mu s_i + \mathcal{O}(s_i) = (1 + 1/(\mathbf{t}+3))\log_2(n/s_i)s_i + \log_2(\mathbf{t}+1)s_i + \mathcal{O}(s_i). \quad \square$$

6.2 Polylogarithmic parameter

Letting the parameter $\mathbf{t}$ vary, we minimise the upper bound provided by Theorem 46 by choosing $\mathbf{t} = \Theta(\mathcal{H}^*+1)$, in which case this upper bound simply becomes $n\mathcal{H}^* + \log_2(\mathcal{H}^* + 1)n + \mathcal{O}(n)$. However, computing or approximating $\mathcal{H}^*$ before starting the actual sorting process would be both rather unreasonable and not necessarily worth the effort; indeed, Theorem 51 would make this preliminary step useless unless it requires less than $\log_2(\mathcal{H}^* + 1)n$ element comparisons, which seems extremely difficult. Instead, we update the parameter $\mathbf{t}$ as follows, which will provide us with a slightly larger upper bound.

Definition 47. We call *polylogarithmic* galloping routine the merging routine that, when merging adjacent runs of lengths a and b, performs the same comparisons and element moves as the $\mathbf{t}$-galloping routine for $\mathbf{t} = \lceil \log_2(a + b) \rceil^2$.

We first prove that the overhead of using galloping with this update strategy instead of using a naïve merging routine is at most linear.

Lemma 48. *Let $\mathcal{A}$ be a stable algorithm with the middle-growth property. Let $\mathcal{T}$ be a merge tree induced by $\mathcal{A}$ on some array A of length n with σ dual runs $S_1, S_2, \ldots, S_\sigma$, and let $\mathcal{T}^*$ be the set of internal nodes of $\mathcal{T}$. If $\mathcal{A}$ uses the polylogarithmic routine for merging runs, it requires no more than*

$$\sum_{R \in \mathcal{T}^*} \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\log}(r, r_{\to i}) + \mathcal{O}(n)$$

comparisons to sort $\mathcal{A}$, where we set $\mathsf{cost}^*_{\log}(r, m) = \min\{m, 6\log_2(r + 1)^2 + 6\}$.

Proof. Using a parameter $\mathbf{t} = \lceil \log_2(r) \rceil^2$ to merge runs R' and R'' into one run R requires at most

$$1 + \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\lceil \log_2(r) \rceil^2}(r'_{\to i}) + \mathsf{cost}^*_{\lceil \log_2(r) \rceil^2}(r''_{\to i})$$

element comparisons. Given that

$$\begin{aligned} \mathsf{cost}^*_{\lceil \log_2(r) \rceil^2}(r'_{\to i}) &\leqslant \min\{(1 + 1/\log_2(r)^2)r'_{\to i}, \log_2(r)^2 + 3 + 2\log_2(r'_{\to i} + 1)\} \\ &\leqslant \min\{r'_{\to i}, 3\log_2(r + 1)^2 + 3\} + r'_{\to i}/\log_2(r)^2 \end{aligned}$$

and that $r'_{\to i} + r''_{\to i} = r_{\to i}$, this makes a total of at most

$$1 + \frac{r}{\log_2(r)^2} + \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\log}(r, r_{\to i})$$

element comparisons.

Now, let $\beta > 1$ be the real number mentioned in the definition of the statement "$\mathcal{A}$ has the middle-growth property", and let $\mathcal{R}_h$ be the set of runs at height h in $\mathcal{T}^*$. The lengths of runs in $\mathcal{R}_h$ sum up to n or less, and thus

$$\sum_{R \in \mathcal{T}^*} \frac{r}{\log_2(r)^2} = \sum_{h \geqslant 1} \sum_{R \in \mathcal{R}_h} \frac{r}{\log_2(r)^2} \leqslant \sum_{h \geqslant 1} \sum_{R \in \mathcal{R}_h} \frac{r}{h^2 \log_2(\beta)^2} \leqslant \sum_{h \geqslant 1} \frac{n}{h^2 \log_2(\beta)^2} = \mathcal{O}(n).$$

We conclude by remembering that $\mathcal{A}$ makes $n - 1$ comparisons to identify runs prior to merging them and performs $\rho - 1 \leqslant n - 1$ merges. □

Proposition 49. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. Let U_A be the number of comparisons performed by $\mathcal{A}$ to sort an array A when using a naïve merging routine, and let V_A be the number of comparisons performed by $\mathcal{A}$ when using the polylogarithmic galloping routine. For all arrays A of length n, we have $\mathsf{V}_A \leqslant \mathsf{U}_A + \mathcal{O}(n)$.*

Proof. Lemma 48 proves that

$$\begin{aligned}\mathsf{V}_A = \sum_{R\in\mathcal{T}^*}\sum_{i=1}^{\sigma}\mathsf{cost}^*_{\log}(r, r_{\to i}) + \mathcal{O}(n) \leqslant \sum_{R\in\mathcal{T}^*}\sum_{i=1}^{\sigma} r_{\to i} + \mathcal{O}(n) = \sum_{R\in\mathcal{T}^*} r + \mathcal{O}(n)\\ = \mathsf{U}_A + \mathcal{O}(n).\end{aligned}$$

□

In addition, the polylogarithmic galloping also turns out to be extremely efficient for algorithms with the (tight) middle-growth property.

Lemma 50. *Let $\mathcal{T}$ be a merge tree induced by a stable algorithm on some array A of length n with σ dual runs $S_1, S_2, \ldots, S_\sigma$. Consider a real number $u \in (1, 2]$ and some index $i \leqslant \sigma$. Then, for all $h \geqslant 0$, let $\mathcal{T}_h$ be a set of pairwise incomparable nodes of $\mathcal{T}$ such that each run R in $\mathcal{T}_h$ is of length $r \geqslant u^h \log_2(2n/s_i)^2 n/s_i$. We have*

$$\sum_{R\in\mathcal{T}_{\geqslant 0}} \mathsf{cost}^*_{\log}(r, r_{\to i}) \leqslant \frac{6(10u^2 - 13u + 5)u}{(u-1)^3} s_i,$$

where $\mathcal{T}_{\geqslant 0}$ denotes the union of the sets $\mathcal{T}_h$.

Proof. For each integer $h \geqslant 0$, let $\mathsf{C}(h) = \sum_{R\in\mathcal{T}_h} \mathsf{cost}^*_{\log}(r, r_{\to i})$. Let $f\colon x \mapsto 1 + \log_2(x+1)^2$ and $g\colon x \mapsto x\, f(n/x)$; the function f is concave and increasing on $[2, +\infty)$, and g is increasing on $(0, +\infty)$.

Then, let $z = n/s_i$ and $v = \log_2(2z)^2$. Since $\mathcal{T}_h$ consists of pairwise incompatible runs, we have $u^h vz|\mathcal{T}_h| \leqslant \sum_{R\in\mathcal{T}_h} r \leqslant n$, i.e., $|\mathcal{T}_h| \leqslant s_i/(u^h v)$. It follows that

$$\begin{aligned}\mathsf{C}(h)/6 &\leqslant \sum_{R\in\mathcal{T}_h} f(r) \leqslant |\mathcal{T}_h| f\Big(\sum_{R\in\mathcal{T}_h} r/|\mathcal{T}_h|\Big) \leqslant g(|\mathcal{T}_h|)\\ &\leqslant g(s_i/(u^h v)) = f(u^h vz) s_i/(u^h v)\\ &\leqslant (1 + \log_2(2^{h+3} z^3)^2) s_i/(u^h v)\\ &\leqslant (1 + (h + 3\log_2(2z))^2) s_i/(u^h v)\\ &\leqslant (h^2 + 6h + 10) s_i/u^h.\end{aligned}$$

where the inequality between the second and third lines simply comes from the fact that

$$1 + u^h vz \leqslant 1 + 4u^h z^3 \leqslant 8u^h z^3 \leqslant 2^{h+3} z^3,$$

and the inequality between the last two lines comes from the fact that

$$1 + (h + 3\log_2(2z))^2 \leqslant \log_2(2z)^2 + (h\log_2(2z) + 3\log_2(2z))^2 = (h^2 + 6h + 10)v.$$

Setting $\mathsf{C}_+(h) = (h^2 + 6h + 10) s_i/u^h$, we conclude that

$$\sum_{R\in\mathcal{T}_{\geqslant 0}} \mathsf{cost}^*_{\log}(r, r_{\to i}) \leqslant \sum_{h\geqslant 0} \mathsf{C}(h) \leqslant 6 \sum_{h\geqslant 0} \mathsf{C}_+(h) = 6\frac{(10u^2 - 13u + 5)u}{(u-1)^3} s_i.$$

□

Theorem 51. *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. If $\mathcal{A}$ uses the polylogarithmic galloping routine for merging runs, it requires $\mathcal{O}(n + n\mathcal{H}^*)$ element comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$. If, furthermore, $\mathcal{A}$ has the tight middle-growth property, it requires at most $n\mathcal{H}^* + 2\log_2(\mathcal{H}^* + 1)n + \mathcal{O}(n)$ element comparisons to sort such arrays.*

Proof. Let us refine and adapt the proofs of Theorems 7 and 46. Let $\mathcal{T}$ be the merge tree induced by $\mathcal{A}$ on an array A of length n with σ dual runs of lengths $s_1, s_2, \ldots, s_\sigma$. For all integers $h \geqslant 0$, let $\mathcal{R}_h$ be the set of runs at height h in $\mathcal{T}$, and let $\mathcal{T}^*$ be the set of internal nodes of $\mathcal{T}$. Let $\beta \in (1, 2]$ and $\gamma \geqslant 0$ be a real number and an integer such that $r \geqslant \beta^{h-\gamma}$ for all runs R of height h in $\mathcal{T}$; if $\mathcal{A}$ has the tight middle-growth property, we choose $\beta = 2$.

Thanks to Lemma 48, we shall just prove that

$$\sum_{R\in\mathcal{T}^*} \mathsf{cost}^*_{\log}(r, r_{\to i}) \leqslant \log_\beta(n/s_i)s_i + 2\log_\beta(\log_2(n/s_i)+1)s_i + \mathcal{O}(s_i)$$

for all $i \leqslant \sigma$. Indeed, the algorithm $\mathcal{A}$ will then perform no more than

$$\begin{aligned}\sum_{R\in\mathcal{T}^*}\sum_{i=1}^{\sigma} \mathsf{cost}^*_{\log}(r, r_{\to i}) + \mathcal{O}(n) &\leqslant \sum_{i=1}^{\sigma} \log_\beta(n/s_i)s_i + 2\log_\beta(\log_2(n/s_i)+1)s_i + \mathcal{O}(n)\\ &\leqslant \log_\beta(2)n\mathcal{H}^* + 2\log_\beta(\mathcal{H}^*+1)n + \mathcal{O}(n)\end{aligned}$$

comparisons, the latter inequality being due to the concavity of the function $x \mapsto \log_\beta(x+1)$.

Then, let $\mathcal{R}_h$ be the set of runs at height h in $\mathcal{T}$, and let

$$\mathsf{C}_{\log}(h) = \sum_{R\in\mathcal{R}_h} \mathsf{cost}^*_{\log}(r, r_{\to i}).$$

Once again, $\mathsf{C}_{\log}(h) \leqslant \sum_{R\in\mathcal{R}_h} r_{\to i} \leqslant \sum_{R\in\mathcal{R}_h} r_{\to i} = s_i$ for all $h \geqslant 0$, because $\mathcal{R}_h$ consists of pairwise incomparable runs R.

Now, let $\nu = \lceil \log_\beta(\log_2(2n/s_i)^2 n/s_i)\rceil + \gamma$. By construction, each run R in $\mathcal{T}_h = \mathcal{R}_{h+\nu}$ is of length

$$r \geqslant \beta^{h+\nu-\gamma} \geqslant \beta^h \log_2(2n/s_i)^2 n/s_i.$$

Thus, applying Lemma 50 to $u = \beta$ indicates that $\sum_{h\geqslant\nu} \mathsf{C}_{\mathbf{t}}(h) = \mathcal{O}(s_i)$, and we conclude that

$$\sum_{h\geqslant 0} \mathsf{C}_{\mathbf{t}}(h) \leqslant \nu s_i + \mathcal{O}(s_i) = \log_\beta(n/s_i)s_i + 2\log_\beta(\log_2(n/s_i)+1)s_i + \mathcal{O}(s_i). \qquad \square$$

Finally, in practice, we could improve the $n\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1)n + \mathcal{O}(n)$ upper bound by adapting our update policy. For instance, choosing $\mathbf{t} = \lceil\log_2(a+b)\rceil \times \lceil\log_2(\log_2(a+b))\rceil^2$ would slightly damage the constant hidden in the $\mathcal{O}(n)$ side of the inequality $\mathsf{V}_A \leqslant \mathsf{U}_A + \mathcal{O}(n)$, but would also reduce the number of comparisons required by $\mathcal{A}$ to

$$n\mathcal{H}^* + \log_2(\mathcal{H}^*+1)n + \mathcal{O}(\log(\log(\mathcal{H}^*+1)+1)n).$$

However, such improvements may soon become negligible in comparison with the overhead of having to compute the value of $\mathbf{t}$.

6.3 Refined upper bounds for adaptive ShiversSort

Proposition 52. *The algorithm adaptive ShiversSort does not have the tight middle-growth property.*

Proof. Let A_k be an array whose run decomposition consists in runs of lengths $1, 2, 1, 4, 1, 8, 1, 16, 1, \ldots, 1, 2^k, 1$ for some integer $k \geqslant 0$. When sorting the array A_k, the algorithm adaptive ShiversSort keeps merging the two leftmost runs at its disposal. The resulting tree, represented in Figure 9, has height $h = 2k$ and its root is a run of length $r = 2^{k+1} + k - 1 = o(2^h)$. $\square$

Theorem 53. *Theorems 46 and 51 remain valid if we consider the algorithm adaptive ShiversSort instead of an algorithm with the tight middle-growth property.*

Proof. Our proof is similar to the proofs of Theorems 46 and 51. Let $\mathcal{T}$ be the merge tree induced by adaptive ShiversSort on an array A of length n and dual runs $S_1, S_2, \ldots, S_\sigma$. Following the terminology of [16], we say that a run R in $\mathcal{T}$ is *non-expanding* if it has the same level as its parent $R^{(1)}$, i.e., if $\ell = \ell^{(1)}$. It is shown, in the proof of [16, Proposition 4], that the lengths of the non-expanding runs sum up to an integer smaller than $3n$. Hence, we partition $\mathcal{T}$ as follows.

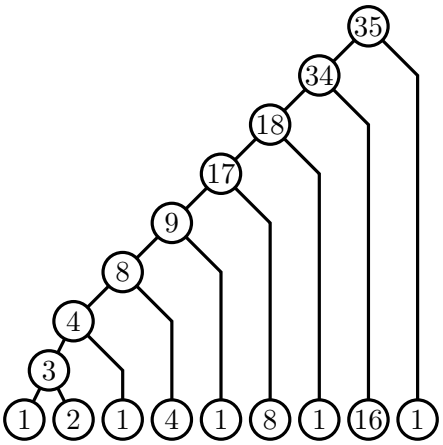

Figure 9: Merge tree induced by adaptive ShiversSort on A_4. Each run is labelled by its length.

We place all the non-expanding runs into one set $\mathcal{R}_\perp$. Then, for each $\ell \geqslant 0$, we define $\mathcal{R}_\ell$ as the set of expanding runs in $\mathcal{T}^*$ with level ℓ, where $\mathcal{T}^*$ is the set of all internal nodes of $\mathcal{T}$. By construction, each run R in $\mathcal{R}_\ell$ has a length $r \geqslant 2^\ell$, and the elements of $\mathcal{R}_\ell$ are pairwise incomparable.

Now, we first observe that

$$\sum_{R\in\mathcal{R}_\perp}\sum_{i=1}^{\sigma}\mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) \leqslant \sum_{R\in\mathcal{R}_\perp}\sum_{i=1}^{\sigma}2r_{\to i} = \sum_{R\in\mathcal{R}_\perp}2r \leqslant 6n \text{ and}$$
$$\sum_{R\in\mathcal{R}_\perp}\sum_{i=1}^{\sigma}\mathsf{cost}^*_{\log}(r, r_{\to i}) \leqslant \sum_{R\in\mathcal{R}_\perp}\sum_{i=1}^{\sigma}r_{\to i} = \sum_{R\in\mathcal{R}_\perp}r \leqslant 3n.$$

Consequently, like in the proofs of Theorems 46 and 51, we just need to show for all $i \leqslant \sigma$ that

$$\sum_{\ell\geqslant 0}\mathsf{C}_{\mathbf{t}}(\ell) \leqslant (1+1/(\mathbf{t}+3))\log_2(n/s_i)s_i + \log_2(\mathbf{t}+1)s_i + \mathcal{O}(s_i) \text{ and}$$
$$\sum_{\ell\geqslant 0}\mathsf{C}_{\log}(\ell) \leqslant \log_2(n/s_i)s_i + 2\log_2(\log_2(n/s_i)+1)s_i + \mathcal{O}(s_i),$$

where $\mathsf{C}_{\mathbf{t}}(\ell) = \sum_{R\in\mathcal{R}_\ell}\mathsf{cost}^*_{\mathbf{t}}(r_{\to i})$ and $\mathsf{C}_{\log}(\ell) = \sum_{R\in\mathcal{R}_\ell}\mathsf{cost}^*_{\log}(r, r_{\to i})$.

Note, however, that

$$\mathsf{C}_{\mathbf{t}}(\ell) \leqslant (1+1/(\mathbf{t}+3))\sum_{R\in\mathcal{R}_\ell}r_{\to i} \leqslant (1+1/(\mathbf{t}+3))s_i \text{ and}$$
$$\mathsf{C}_{\log}(\ell) \leqslant \sum_{R\in\mathcal{R}_\ell}r_{\to i} \leqslant s_i$$

for all $\ell \geqslant 0$. Then, if we set $\mu = \lceil\log_2((\mathbf{t}+1)n/s_i)\rceil$ and $\nu = \lceil\log_2(\log_2(2n/s_i)^2 n/s_i)\rceil$, we observe that $r \geqslant 2^{\mu+h} \geqslant 2^h(\mathbf{t}+1)n/s_i$ for all $r \in \mathcal{R}_{\mu+h}$, and that $r \geqslant 2^{\nu+h} \geqslant 2^h\log_2(2n/s_i)^2 n/s_i$ for all $r \in \mathcal{R}_{\nu+h}$. Hence, applying Lemmas 45 and 50 proves, as desired, that

$$\begin{aligned}\sum_{\ell=0}^{\mu-1}\mathsf{C}_{\mathbf{t}}(\ell) + \sum_{\ell\geqslant 0}\mathsf{C}_{\mathbf{t}}(\ell+\mu) &\leqslant (1+1/(\mathbf{t}+3))\mu s_i + \mathcal{O}(s_i)\\ &\leqslant (1+1/(\mathbf{t}+3))\log_2(n/s_i)s_i + \log_2(\mathbf{t}+1)s_i + \mathcal{O}(s_i) \text{ and}\\ \sum_{\ell=0}^{\nu-1}\mathsf{C}_{\log}(\ell) + \sum_{\ell\geqslant 0}\mathsf{C}_{\log}(\ell+\nu) &\leqslant \nu s_i + \mathcal{O}(s_i)\\ &\leqslant \log_2(n/s_i)s_i + 2\log_2(\log_2(n/s_i)+1)s_i + \mathcal{O}(s_i).\end{aligned}$$

□

6.4 Refined upper bounds for PeekSort

Proposition 54. *The algorithm PeekSort does not have the tight middle-growth property.*

Proof. Let B_k be an array whose run decomposition consists in runs of lengths $1, 1, 3, 3, 9, 9, \ldots, 3^k, 3^k$ for some integer $k \geqslant 0$. Its length is the integer $b_k = 3^{k+1} - 1$ and, when $k \geqslant 1$, it consists in a copy of B_{k-1} and then two runs of lengths 3^k. Thus, when sorting B_k, PeekSort shall recursively sort B_{k-1}, then merge it with its neighbouring run of length 3^k, and merge the resulting run with the rightmost run of length 3^k. The resulting tree, represented in Figure 10, has height $h = 2k + 1$ and its root is a run of length $b_k = 3^{k+1} - 1 = o(2^h)$. □

The method of casting aside runs with a limited total length, which we used while adapting the proofs Theorems 46 and 51 to adaptive ShiversSort, does not work with the algorithm PeekSort. Instead, we rely on the approach employed by Bayer [5] to prove the $n\mathcal{H} + 2n$ upper bound on the number of element comparisons and moves when PeekSort uses the naïve merging routine. This approach is based on the following result, which relies on the notions of *split runs* and *growth rate* of a run.

In what follows, let us recall some notations: given an array A of length n whose run decomposition consists of runs $R_1, R_2, \ldots, R_\rho$, we set $e_i = r_1 + \ldots + r_i$ for all integers $i \leqslant \rho$, and the run that results from merging consecutive runs $R_i, R_{i+1}, \ldots, R_j$ is denoted by $R_{i\ldots j}$.

Definition 55. Let $\mathcal{T}$ be a merge tree, and let R and R' be the children of a run $\overline{R}$. The *split run* of $\overline{R}$ is defined as the rightmost leaf descending from R if $r \geqslant r'$, and as the leftmost leaf descending from R' otherwise. The length of that split run is then called the *split length* of $\overline{R}$, and is denoted by $\mathsf{sl}(\overline{R})$. Finally, the quantity $\log_2(\overline{r}) - \max\{\log_2(r), \log_2(r')\}$ is called *growth rate* of the run $\overline{R}$, and is denoted by $\mathsf{gr}(\overline{R})$.

Lemma 56. *Let $\mathcal{T}$ be a merge tree induced by PeekSort. We have $\mathsf{gr}(\overline{R})\overline{r} + 2\mathsf{sl}(\overline{R}) \geqslant \overline{r}$ for each internal node $\overline{R}$ of $\mathcal{T}$.*

Proof. Let $R = R_{i\ldots k}$ and $R' = R_{k+1\ldots j}$ be the children of the run $\overline{R}$. We assume, without loss of generality, that $r \geqslant r'$. The case $r < r'$ is entirely symmetric. Definition 17 then states that

$$|r - r'| = |2e_k - e_j - e_{i-1}| \leqslant |2e_{k-1} - e_j - e_{i-1}| = |r - r' - 2r_k|,$$

which means that $r - r' - 2r_k$ is negative and that $r - r' \leqslant 2r_k + r' - r$.

Then, observe that the function $f : t \mapsto 4t - 3 - \log_2(t)$ is non-negative on the interval $[1/2, +\infty)$. Finally, let $z = r/\overline{r}$, so that $z \geqslant 1/2$, $\mathsf{gr}(\overline{R}) = -\log_2(z)$ and $r' = (1 - z)\overline{r}$:

$$\mathsf{gr}(\overline{R})\overline{r} + 2\mathsf{sl}(\overline{R}) = 2r_k - \log_2(z)\overline{r} \geqslant 2(r - r') - \log_2(z)\overline{r} = (f(z) + 1)\overline{r} \geqslant \overline{r}. \qquad \square$$

Lemma 57. *Let $\mathcal{T}$ be a merge tree induced by PeekSort on an array of length n, and let $\mathcal{T}^*$ be the set of internal nodes of $\mathcal{T}$. We have $\sum_{R \in \mathcal{T}^*} \mathsf{sl}(R) \leqslant 2n$.*

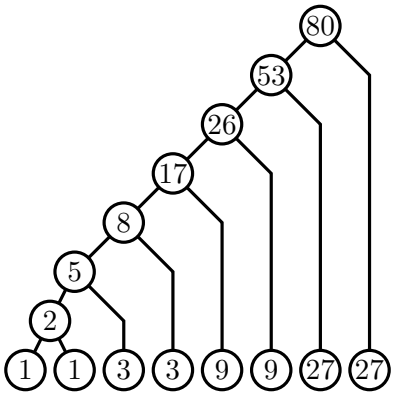

Figure 10: Merge tree induced by PeekSort on B_3. Each run is labelled by its length.

Proof. Let $\mathcal{R}$ be the set of leaves of $\mathcal{T}$. Each run $R \in \mathcal{R}$ is a split run of at most two nodes of $\mathcal{T}$: these are the parents of the least ancestor of R that is a left run (this least ancestor can be R itself) and of the least ancestor of R that is a right run. It follows that

$$\sum_{R \in \mathcal{T}^*} \mathsf{sl}(R) \leqslant 2 \sum_{R \in \mathcal{R}} r = 2n. \qquad \square$$

Lemma 58. *Let $\mathcal{T}$ be a merge tree induced by PeekSort on an array with σ dual runs $S_1, S_2, \ldots, S_\sigma$. For all $i \leqslant \sigma$ and all sets $\mathcal{X}$ of inner nodes of $\mathcal{T}$, we have*

$$\sum_{R \in \mathcal{X}} \mathsf{gr}(R) r_{\to i} \leqslant \log_2(x) s_i,$$

where x is the largest length of a run $R \in \mathcal{X}$.

Proof. Let $\mathcal{R}$ be the set of leaves of $\mathcal{T}$. Without loss of generality, we assume that $\mathcal{X}$ contains each node $\overline{R}$ of $\mathcal{T}$ with length $\overline{r} \leqslant x$. Now, for all runs $R \in \mathcal{R}$, let $R^\uparrow$ be the set of those strict ancestors of R that belong to $\mathcal{X}$. Since every child of a node in $\mathcal{X}$ is either a leaf or a node in $\mathcal{X}$, the set $R^\uparrow$ consists of those runs $R^{(k)}$ such that $1 \leqslant k \leqslant |R^\uparrow|$. It follows that

$$\begin{aligned}
\sum_{\overline{R} \in \mathcal{X}} \mathsf{gr}(\overline{R}) \overline{r}_{\to i} &= \sum_{R \in \mathcal{R}} \sum_{\overline{R} \in R^\uparrow} \mathsf{gr}(\overline{R}) r_{\to i} = \sum_{R \in \mathcal{R}} \sum_{k=1}^{|R^\uparrow|} \mathsf{gr}(R^{(k)}) r_{\to i} \\
&\leqslant \sum_{R \in \mathcal{R}} \sum_{k=1}^{|R^\uparrow|} \log_2(r^{(k)}/r^{(k-1)}) r_{\to i} = \sum_{R \in \mathcal{R}} \log_2(r^{(|R^\uparrow|)}/r) r_{\to i} \\
&\leqslant \sum_{R \in \mathcal{R}} \log_2(x) r_{\to i} = \log_2(x) s_i. \qquad \square
\end{aligned}$$

Theorem 59. *Theorem 46 remains valid if we consider the algorithm PeekSort instead of an algorithm with the tight middle-growth property.*

Proof. Let $\mathcal{T}$ be the merge tree induced by PeekSort on an array A of length n and dual runs $S_1, S_2, \ldots, S_\sigma$, and let $\mathcal{T}^*$ be the set of internal nodes of $\mathcal{T}$. The algorithm PeekSort requires no more than

$$\mathsf{C} = \sum_{R \in \mathcal{T}^*} \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) + \mathcal{O}(n)$$

comparisons and, for a given run $R \in \mathcal{T}^*$, we have

$$\begin{aligned}
\sum_{i=1}^{\sigma} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) - 4\mathsf{sl}(R) &\leqslant \sum_{i=1}^{\sigma} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) - 2(1 + 1/(\mathbf{t}+3))\mathsf{sl}(R) \\
&\leqslant \sum_{i=1}^{\sigma} \min\{(1 + 1/(\mathbf{t}+3)) r_{\to i}, \mathbf{t} + 2 + 2\log_2(r_{\to i} + 1)\} + \\
&\qquad (1 + 1/(\mathbf{t}+3))(\mathsf{gr}(R) - 1) r_{\to i} \\
&\leqslant \sum_{i=1}^{\sigma} \mathsf{cost}^{**}_{\mathbf{t}}(\mathsf{gr}(R), r_{\to i}),
\end{aligned}$$

where we set $\mathsf{cost}^{**}_{\mathbf{t}}(\gamma, m) = \min\{(1 + 1/(\mathbf{t}+3))\gamma m, \mathbf{t} + 2 + 2\log_2(m+1)\}$.

Since PeekSort has the fast-growth property, there exist a real number $\alpha > 1$ and an integer $\delta \geqslant 0$ such that $r^{(k)} \geqslant \alpha^{k-\delta} r$ for all runs R of depth at least k in $\mathcal{T}$. Now, given an index $i \leqslant \sigma$, let $\mathcal{X}$ be the set of runs R of length $r < \alpha^\delta(\mathbf{t}+1)n/s_i$, and let $\mathcal{Y}$ be the sub-tree of $\mathcal{T}$ consisting of runs of length $r \geqslant \alpha^\delta(\mathbf{t}+1)n/s_i$. For all $h \geqslant 0$, let $\mathcal{Y}_h$ denote the set of runs of height h in the tree $\mathcal{Y}$; by construction, $\mathcal{Y}_h$ contains only runs of length $r \geqslant \alpha^h(\mathbf{t}+1)n/s_i$.

Consequently, Lemma 58 states that

$$\sum_{R\in\mathcal{X}} \mathsf{cost}^{**}_{\mathbf{t}}(\mathsf{gr}(R), r_{\to i}) \leqslant (1+1/(\mathbf{t}+3)) \sum_{R\in\mathcal{X}} \mathsf{gr}(R) r_{\to i} \leqslant (1+1/(\mathbf{t}+3)) \log_2(\alpha^\delta(\mathbf{t}+1)n/s_i)s_i,$$

whereas Lemma 45 states that

$$\sum_{R\in\mathcal{Y}} \mathsf{cost}^{**}_{\mathbf{t}}(\mathsf{gr}(R), r_{\to i}) \leqslant \sum_{R\in\mathcal{Y}} \mathsf{cost}^{*}_{\mathbf{t}}(r_{\to i}) \leqslant 9\alpha s_i/(\alpha-1).$$

It follows that

$$\begin{aligned}
\mathsf{C} &\leqslant \sum_{R\in\mathcal{T}^*} 4\mathsf{sl}(R) + \sum_{i=1}^{\sigma} \mathsf{cost}^{**}_{\mathbf{t}}(\mathsf{gr}(R), r_{\to i}) + \mathcal{O}(n) \\
&\leqslant \sum_{i=1}^{\sigma} \Big((1+1/(\mathbf{t}+3)) \log_2(\alpha^\delta(\mathbf{t}+1)n/s_i)s_i + \mathcal{O}(s_i) \Big) + \mathcal{O}(n) \\
&\leqslant (1+1/(\mathbf{t}+3)) n\mathcal{H}^* + n\log_2(\mathbf{t}+1) + \mathcal{O}(n).
\end{aligned}$$

□

Theorem 60. *Theorem 51 remains valid if we consider the algorithm PeekSort instead of an algorithm with the tight middle-growth property.*

Proof. Let us reuse the notations and the main lines of the proof of Theorem 59. With these notations, PeekSort requires no more than

$$\mathsf{C} = \sum_{R\in\mathcal{T}^*} \sum_{i=1}^{\sigma} \mathsf{cost}^{*}_{\log}(r, r_{\to i}) + \mathcal{O}(n)$$

element comparisons and, for a given run $R \in \mathcal{T}^*$, we have

$$\begin{aligned}
\sum_{i=1}^{\sigma} \mathsf{cost}^{*}_{\log}(r, r_{\to i}) - 2\mathsf{sl}(R) &\leqslant \sum_{i=1}^{\sigma} \min\{r_{\to i}, 6\log_2(r+1)^2+6\} + (1-\mathsf{gr}(R))r_{\to i} \\
&\leqslant \sum_{i=1}^{\sigma} \mathsf{cost}^{**}_{\log}(r, \mathsf{gr}(R)r_{\to i}),
\end{aligned}$$

where we set $\mathsf{cost}^{**}_{\log}(r,m) = \min\{m, 6\log_2(r+1)^2+6\}$.

Let $\alpha > 1$ and $\delta \geqslant 0$ be a real number and an integer such that $r^{(k)} \geqslant \alpha^{k-\delta} r$ for all runs R of depth at least k in $\mathcal{T}$. Then, given an integer $i \leqslant \sigma$, let $\mathcal{X}$ be the set of runs R of length $r < \alpha^\delta \log_2(2n/s_i)^2 n/s_i$, and let $\mathcal{Y}$ be the sub-tree of $\mathcal{T}$ consisting of runs of length $r \geqslant \alpha^\delta \log_2(2n/s_i)^2 n/s_i$. For all $h \geqslant 0$, let $\mathcal{Y}_h$ denote the set of runs of height h in the tree $\mathcal{Y}$; by construction, $\mathcal{Y}_h$ contains only runs of length $r \geqslant \alpha^h \log_2(2n/s_i)^2 n/s_i$.

Consequently, Lemma 58 states that

$$\sum_{R\in\mathcal{X}} \mathsf{cost}^{**}_{\log}(r, \mathsf{gr}(R)r_{\to i}) \leqslant \sum_{R\in\mathcal{X}} \mathsf{gr}(R) r_{\to i} \leqslant \log_2(\alpha^\delta \log_2(2n/s_i)^2 n/s_i)s_i,$$

whereas Lemma 50 states that

$$\sum_{R\in\mathcal{Y}} \mathsf{cost}^{**}_{\log}(r, \mathsf{gr}(R)r_{\to i}) \leqslant \sum_{R\in\mathcal{Y}} \mathsf{cost}^{*}_{\log}(r, r_{\to i}) = \mathcal{O}(s_i).$$

It follows that

$$\begin{aligned}
\mathsf{C} &\leqslant \sum_{R\in\mathcal{T}^*} 2\mathsf{sl}(R) + \sum_{i=1}^{\sigma} \mathsf{cost}^{**}_{\log}(r, \mathsf{gr}(R)r_{\to i}) + \mathcal{O}(n) \\
&\leqslant \sum_{i=1}^{\sigma} \Big(\log_2(\alpha^\delta \log_2(2n/s_i)^2 n/s_i)s_i + \mathcal{O}(s_i) \Big) + \mathcal{O}(n) \\
&\leqslant n\mathcal{H}^* + 2n\log_2(\mathcal{H}^*+1) + \mathcal{O}(n).
\end{aligned}$$

□

References

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

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

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

[4] Jérémy Barbay, Carlos Ochoa and Srinivasa Rao Satti. Synergistic solutions on multisets. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 31:2–14, 2017.

[5] Paul Bayer. Improved bounds on the cost of optimal and balanced binary search trees. M.Sc. Thesis, MIT, Cambridge, 1975.

[6] Jon Bentley and Andrew Yao. An almost optimal algorithm for unbounded searching. In *Information Processing Letters*, pages 5(3):82–87, 1976.

[7] Josh Bloch. Timsort implementation in Java 13, retrieved 01/01/2024. `github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/ComparableTimSort.java`

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

[9] Svante Carlsson, Christos Levcopoulos and Ola Petersson. Sublinear merging and natural mergesort. In *Algorithmica*, pages 9:629–648, 1993.

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

[11] Ben Cohen. Timsort implementation in Swift, retrieved 01/01/2024. `github.com/apple/swift/blob/master/stdlib/public/core/Sort.swift`

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

[13] William Gelling, Markus Nebel, Benjamin Smith and Sebastian Wild. Multiway Powersort. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 190–200, 2023.

[14] Elahe Ghasemi, Vincent Jugé and Ghazal Khalighinejad. Galloping in fast-growth natural merge sorts. In *49th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 68:1–68:19, 2022.

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

[16] Vincent Jugé. Adaptive Shivers sort: An alternative sorting algorithm. In *ACM Transactions on Algorithms*, pages 20(4):1–55, 2024.

[17] Vincent Jugé. Counting comparisons performed by Timsort to merge two non-decreasing runs. `github.com/VincentJuge1987/timsort`

[18] Donald Knuth. *The Art of computer programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publish. Co., 1998.

[19] Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences.
In *Scandinavian Workshop on Algorithm Theory*, pages 181–191, 1990.

[20] Heikki Mannila. Measures of presortedness and optimal sorting algorithms.
In *IEEE Trans. Computers*, pages 34(4):318–325, 1985.

[21] Peter McIlroy. Optimistic sorting and information theoretic complexity.
In *Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 467–474, 1993.

[22] Ian Munro and Philip Spira. Sorting and searching in multisets.
In *SIAM Journal on Computing*, pages 5(1):1–8, 1976.

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

[24] Tim Peters. Timsort description, retrieved 01/09/2021.
`github.com/python/cpython/blob/master/Objects/listsort.txt`

[25] Clément Renault *et al*. Timsort implementation in Rust, retrieved 01/01/2024.
`github.com/rust-lang/rust/blob/master/library/alloc/src/slice.rs`

[26] Jens Schou and Bei Wang. PersiSort: A new perspective on adaptive sorting based on persistence.
In *36th Canadian Conference on Computational Geometry (CCCG)*, pages 283–297, 2024.

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

[28] Guido van Rossum *et al*. Powersort implementation in CPython, retrieved 01/01/2024.
`github.com/python/cpython/blob/master/Objects/listobject.c`

[29] Jeff Weaton and Markus Mützel. Timsort implementation in Octave, retrieved 01/01/2024.
`github.com/gnu-octave/octave/blob/master/liboctave/util/oct-sort.cc`

[30] Simon Zünd *et al.* Timsort implementation in V8, retrieved 01/01/2024.
`github.com/v8/v8/blob/master/third_party/v8/builtins/array-sort.tq`