Galloping in Fast-Growth Natural Merge Sorts

Elahe Ghasemi
Sharif University of Technology, Teheran, Iran
Univ Gustave Eiffel, CNRS, LIGM, F-77454 Marne-la-Vallée, France

Vincent Jugé
Univ Gustave Eiffel, CNRS, LIGM, F-77454 Marne-la-Vallée, France

Ghazal Khalighinejad
Duke University, Durham, NC, USA
Sharif University of Technology, Teheran, Iran

Abstract

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

The efficiency of TimSort and of similar sorting algorithms has often been explained by using the notion of *runs* and the associated *run-length entropy*. Here, we focus on the related notion of *dual runs*, which was introduced in the 1990s, and the associated *dual run-length entropy*. We prove, for this complexity measure, results that are similar to those already known when considering standard run-induced measures: in particular, TimSort requires only $\mathcal{O}(n + n\log(\sigma))$ element comparisons to sort arrays of length n with σ distinct values.

In order to do so, we introduce new notions of *fast-* and *middle-growth* for natural merge sorts (i.e., algorithms based on merging runs). By using these notions, we prove that several merge sorting algorithms, provided that they use TimSort's galloping sub-routine for merging runs, are as efficient as TimSort at sorting arrays with low run-induced or dual-run-induced complexities.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Sorting algorithms, Merge sorting algorithms, Analysis of algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.68

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2012.03996

1 Introduction

In 2002, Tim Peters, a software engineer, created a new sorting algorithm, which was called TimSort [20] and was built on ideas from McIlroy [17]. 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 *optimal* algorithms contributed to the regain of interest in the study of sorting algorithms.

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

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, then the run continues with the maximum number of consecutive elements that preserve its monotonicity.

49th International Colloquium on Automata, Languages, and Programming (ICALP 2022).
Editors: Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff;
Article No. 68; pp. 68:1–68:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Among the best-identified reasons behind the success of TimSort are 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, 9] (see Figure 1) are simple. Such decompositions were already used in 1973 by Knuth's NaturalMergeSort [14, Section 5.2.4], which adapted the traditional MergeSort algorithm as follows: NaturalMergeSort is based on splitting arrays into monotonic subsequences, also called *runs*, and on merging these runs together. Thus, all algorithms sharing this feature of NaturalMergeSort are also called *natural* merge sorts.

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 of TimSort can be split into three main components: (i) a variant of an insertion sort, which is used to deal with *small* runs, e.g., runs of length less than 32, (ii) a simple policy for choosing which *large* runs to merge, (iii) a sub-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 [7], ShiversSort [22] (which *predated* TimSort), adaptive ShiversSort [13], PeekSort and PowerSort [19]. 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 is very efficient: whereas TimSort requires $\mathcal{O}(n + n\log(\rho))$ comparisons to sort arrays of length n that can be decomposed as a concatenation of ρ non-decreasing arrays, this component makes TimSort require only $\mathcal{O}(n + n\log(\sigma))$ comparisons to sort arrays of length n with σ distinct values.

Context and 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.

This is optimal in the model of sorting by comparisons, if the input array can be an arbitrary array of length n. However, taking into account the run decompositions of the input array allows using finer-grained complexity classes, as follows. First, 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 $\mathcal{O}(n + n\log(\rho))$ [16]. Second, we may consider even more restricted classes of input arrays, and focus 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 on average, 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, 13, 17]. The number $\mathcal{H}$ is called the *run-length entropy* of the array.

Since the early 2000s, several natural merge sorts were proposed, all of which were meant to offer easy-to-prove complexity guarantees: ShiversSort, which runs in time $\mathcal{O}(n\log(n))$; α-StackSort, which, like NaturalMergeSort, runs in time $\mathcal{O}(n + n\log(\rho))$; α-MergeSort, which, like TimSort, runs in time $\mathcal{O}(n + n\mathcal{H})$; adaptive ShiversSort, PeekSort and PowerSort, which run in time $n\mathcal{H} + \mathcal{O}(n)$.

Except TimSort, these algorithms are, in fact, described only as policies for deciding which runs to merge, the actual sub-routine used for merging runs being left implicit: since choosing a naive merging sub-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 [19]. They compared the running times of TimSort and of TimSort's variant obtained by using a naive merging routine instead of TimSort's galloping sub-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 sub-routine looked *slower* than the naive one.

Galloping turns out to be very efficient when sorting arrays with few distinct values, a class of arrays that had also been intensively studied. As soon as 1976, Munro and Spira [18] proposed a complexity measure $\mathcal{H}^*$ related to the run-length entropy, with the property that $\mathcal{H}^* \leqslant \log_2(\sigma)$ for arrays with σ values. They also proposed an algorithm for sorting arrays of length n with σ values by using $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons. McIlroy [17] then extended their work to arrays representing a permutation π, identifying $\mathcal{H}^*$ with the run-length entropy of π^{-1} and proposing a variant of Munro and Spira's algorithm that would use $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons in this generalised setting. 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. 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 [19] that hinted at its inefficiency, the galloping sub-routine has been omitted in various mainstream implementations of natural merge sorts, in which it was replaced by its naive variant. This is the case, for instance, in library TimSort implementations of the programming languages Swift [8] and Rust [21]. On the contrary, TimSort's implementation in other languages, such as Java [6], Octave [24] or the V8 JavaScript engine [25], and PowerSort's implementation in Python [23] include the galloping sub-routine.

Contributions

We study the time complexity of various natural merge sort algorithms in a context where arrays are not just parametrised by their lengths. More precisely, we focus on a decomposition of input arrays that is dual to the decomposition of arrays into monotonic runs, and that was proposed by McIlroy [17].

Consider an array A that we want to sort in a *stable* manner, i.e., in which two elements can always considered to be distinct, if only because their positions in A are distinct. Without loss of generality, we identify the values $A[1], A[2], \ldots, A[n]$ with the integers from 1 to n, thereby making A a permutation of the set $\{1, 2, \ldots, n\}$. A common measure of presortedness consists in 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.

Here, we adopt a dual approach, which consists in partitioning the set $\{1, 2, \ldots, n\}$ into the *increasing* 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 *shuffled up-sequences* [3, 15] or *riffle shuffles* [17]. In order to underline their connection with runs, we say that these intervals are the *dual runs* of A, and we denote their lengths by s_i. The process of transforming an array into a permutation and then extracting its dual runs is illustrated in Figure 2.

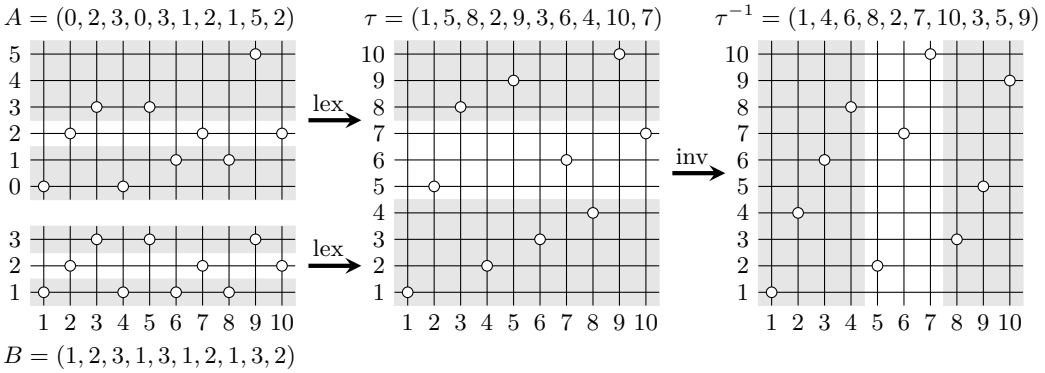

Figure 2 The arrays A and B are lexicographically equivalent to the permutation τ. Their dual runs, represented with gray and white horizontal stripes, have respective lengths 4, 3 and 3. The mappings $A \mapsto \tau$ and $B \mapsto \tau$ identify them with the dual runs of τ, i.e., with the runs of the permutation τ^{-1}. Note that A has only 3 dual runs, although it takes 5 distinct values.

When A is *not* a permutation of $\{1, 2, \ldots, n\}$, the dual runs of A are simply the maximal intervals S_i such that A is non-decreasing on the set of positions $\{j : A[j] \in S_i\} \subseteq \{1, 2, \ldots, n\}$. The length of a dual run is then defined as the cardinality of that set of positions. Thus, two lexicographically equivalent arrays have dual runs of the same lengths: this is the case, for instance, of the arrays A, B and τ in Figure 2.

In particular, we may see τ and B as canonical representatives of the array A: these are the unique permutation of $\{1, 2, \ldots, n\}$ and the unique array with values in $\{1, 2, \ldots, \sigma\}$ that are lexicographically equivalent with A. More generally, an array that contains σ distinct values cannot have more than σ dual runs.

Note that, 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.

In this article, we prove that, by using TimSort's galloping sub-routine, several natural merge sorts require $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons, or even $n\mathcal{H}^* + \mathcal{O}(n)$ comparisons, where $\mathcal{H}^* = H(s_1/n, \ldots, s_\sigma/n) \leqslant \log_2(\sigma)$ is called the *dual run-length entropy* of the array, s_i is the length of the dual run S_i, and H is the general entropy function already mentioned above.

This legitimates using TimSort's arguably complicated galloping sub-routine rather than its naive alternative, in particular when sorting arrays that are constrained to have relatively few distinct values.

This also subsumes results that have been known since the 1970s. For instance, adapting the optimal constructions for alphabetic Huffman codes by Hu and Tucker [12] or Garsia and Wachs [10] to *merge trees* (described in Section 3) already provided sorting algorithms working in time $n\mathcal{H} + \mathcal{O}(n)$.

Our new results rely on notions that we call *fast-* and *middle-growth* properties, and which are found in natural merge sorts like α-MergeSort, α-StackSort, adaptive ShiversSort, ShiversSort, PeekSort, PowerSort or TimSort. More precisely, we prove that merge sorts require $\mathcal{O}(n + n\mathcal{H})$ comparisons *and* element moves when they possess the fast-growth property, thereby encompassing complexity results that were proved separately for each of these algorithms [1, 7, 13, 19], and $\mathcal{O}(n + n\mathcal{H}^*)$ comparisons when they possess the fast- or middle-growth property, which is a completely new result.

Finally, we prove finer complexity bounds on the number of comparisons used by adaptive ShiversSort, ShiversSort, NaturalMergeSort, PeekSort and PowerSort, which require only $n\mathcal{H}^* + \mathcal{O}(n + n\log(\mathcal{H}^* + 1))$ comparisons, nearly matching the $n\mathcal{H}+\mathcal{O}(n)$ (or $n\log_2(n)+\mathcal{O}(n)$ and $n\log_2(\rho) + \mathcal{O}(n)$, in the cases of ShiversSort and NaturalMergeSort) complexity upper bound they already enjoy in terms of comparisons and element moves. These results are summarised in Table 1.

Table 1 Element moves and comparisons needed by various algorithms using appropriate galloping sub-routines to sort arrays of length n with ρ runs, run-length entropy $\mathcal{H}$ and dual run-length entropy $\mathcal{H}^*$.

Algorithm	Element moves	Element comparisons
ShiversSort	$n\log_2(n) + \mathcal{O}(n)$	$n\mathcal{H}^* + \mathcal{O}(n + n\log(\mathcal{H}^* + 1))$
α-StackSort	$\mathcal{O}(n + n\log_2(\rho))$	$\mathcal{O}(n + n\min\{\log_2(\rho), \mathcal{H}^*\})$
NaturalMergeSort	$n\log_2(\rho) + \mathcal{O}(n)$	$n\min\{\log_2(\rho), \mathcal{H}^*\} + \mathcal{O}(n + n\log(\mathcal{H}^* + 1))$
α-MergeSort	$\mathcal{O}(n + n\mathcal{H})$	$\mathcal{O}(n + n\min\{\mathcal{H}, \mathcal{H}^*\})$
TimSort	$3n\mathcal{H}/2 + \mathcal{O}(n)$	$\mathcal{O}(n + n\min\{\mathcal{H}, \mathcal{H}^*\})$
adaptive ShiversSort	$n\mathcal{H} + \mathcal{O}(n)$	$n\min\{\mathcal{H}, \mathcal{H}^*\} + \mathcal{O}(n + n\log(\mathcal{H}^* + 1))$
PeekSort	$n\mathcal{H} + \mathcal{O}(n)$	$n\min\{\mathcal{H}, \mathcal{H}^*\} + \mathcal{O}(n + n\log(\mathcal{H}^* + 1))$
PowerSort	$n\mathcal{H} + \mathcal{O}(n)$	$n\min\{\mathcal{H}, \mathcal{H}^*\} + \mathcal{O}(n + n\log(\mathcal{H}^* + 1))$

2 The galloping sub-routine for merging runs

Here, we describe the galloping sub-routine that the algorithm TimSort uses to merge adjacent non-decreasing runs. This sub-routine is a blend between a naive merging algorithm, which requires $a + b - 1$ comparisons to merge runs A and B of lengths a and b, and a dichotomy-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 sub-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 [5] 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 one chooses $x = m$, 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; 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, one should favour some strategy B_k (with $k \geqslant 1$) over the

naive 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.

Table 2 Comparison requests needed by strategies B_0 and B_1 to find a secret integer $m \geqslant 1$.

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

McIlroy [17] 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: one first follows B_0 for up to $\mathbf{t}$ steps, thereby choosing $x = 1$, $x = 2$, …, $x = \mathbf{t}$ (if $m \leqslant \mathbf{t} - 1$, one stops after choosing $x = m$);

C_2: if $m \geqslant \mathbf{t} + 1$, one switches 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, we will replace this cost function by the following simpler upper bound.

▶ **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 $\mathsf{cost}_{\mathbf{t}}(m) \leqslant m + 1$. Indeed, let $u = m - \mathbf{t}$ and let $f \colon x \mapsto x - 1 - 2\log_2(x)$. The function f is positive and increasing on the interval $[7, +\infty)$. Thus, it suffices to check by hand that $(m+1) - \mathsf{cost}_{\mathbf{t}}(m) = 0, 1, 0, 1$ when $u = 3, 4, 5, 6$, and 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 A and B be two non-decreasing runs of lengths a and b, with values in $\{1, 2, \ldots, \sigma\}$. For each integer $i \leqslant \sigma$, let $a_{\to i}$ (respectively, $b_{\to i}$) be the number of elements in A (respectively, in B) with value i. Using a merging sub-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.

Proof. First, assume that $a_{\to i} = 0$ for some $i \geqslant 2$. Replacing every value $j \geqslant i + 1$ with the value $j - 1$ in both arrays A and B does not change the behaviour of the sub-routine and decreases the value of σ. Moreover, the function $\mathsf{cost}^*_{\mathbf{t}}$ is sub-additive, i.e., we have $\mathsf{cost}^*_{\mathbf{t}}(m) + \mathsf{cost}^*_{\mathbf{t}}(m') \geqslant \mathsf{cost}^*_{\mathbf{t}}(m + m')$ for all $m \geqslant 0$ and $m' \geqslant 0$. Hence, without loss of generality, we assume that $a_{\to i} \geqslant 1$ for all $i \geqslant 2$. Similarly, we assume without loss of generality that $b_{\to i} \geqslant 1$ for all $i \leqslant \sigma - 1$.

Under these assumptions, the array C that results from merging A and B consists of $a_{\to 1}$ elements from A, then $b_{\to 1}$ elements from B, $a_{\to 2}$ elements from A, $b_{\to 2}$ elements from B, $\ldots$, $a_{\to \sigma}$ elements from A and $b_{\to \sigma}$ elements from B. Thus, the galloping sub-routine consists in discovering successively the integers $a_{\to 1}, b_{\to 1}, a_{\to 2}, b_{\to 2}, \ldots, a_{\to \sigma}$, each time using McIlroy's strategy based on the two steps C_1 and C_2; checking whether $a_{\to 1} = 0$ requires one more comparison than prescribed by McIlroy's strategy, and the integer $b_{\to \sigma}$ does not need to be discovered once the entire run A has been scanned be the merging sub-routine. ◀

We simply call $\mathbf{t}$-*galloping sub-routine* the merging sub-routine based on McIlroy's mixed strategy for a fixed parameter $\mathbf{t}$; when the value of $\mathbf{t}$ is irrelevant, we simply 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. By construction, this cost never exceeds $1 + 1/(\mathbf{t}+3)$ times the *naive cost* of merging A and B, which is simply defined as $a + b$. Below, we study the impact of using the galloping sub-routine instead of the naive one, which amounts to replacing naive merge costs by their galloping variants.

Note that using this new galloping cost measure is relevant only if the cost of element comparisons is significantly larger than the cost of 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 sub-routine. Instead, the parameter $\mathbf{t}$ is initially set to a constant ($\mathbf{t} = 7$ in Java), and may change during the algorithm 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).

When sorting a random permutation, changing the value of $\mathbf{t}$ in that way decreases the average overhead of sometimes using B_1 instead of B_0 to a constant. More generally, even in the worst case, this overhead is linear in n.

▶ **Proposition 3.** *Let $\mathcal{A}$ be a stable natural merge sort algorithm, and let A be an array of length n. Let c_1 be the number of comparisons that $\mathcal{A}$ requires to sort A when it uses the naive sub-routine, and let c_2 be the number of comparisons that $\mathcal{A}$ requires to sort A when it uses the galloping sub-routine with TimSort's update policy for the parameter $\mathbf{t}$. We have $\mathsf{c}_2 \leqslant \mathsf{c}_1 + \mathcal{O}(n)$.*

Proof. Below, we group the comparisons that $\mathcal{A}$ performs while sorting A into *steps*, which we will consider as individual units. Steps are formed as follows. Let R and R' be consecutive runs that $\mathcal{A}$ is about to merge, and let us subdivide their concatenation $R \cdot R'$ into σ dual runs $S_1, S_2, \ldots, S_\sigma$ (note that these are the dual runs of $R \cdot R'$ and not the dual runs of A, i.e., some elements of R may belong to a given dual run S_i of $R \cdot R'$ while belonging to distinct dual runs or A).

Each step consists of those comparisons used to discover the elements of R (resp., R') that belong to a given dual run S_i. Thus, the comparisons used to merge R and R' are partitioned into $2\sigma - 1$ steps: in the first step, we discover those elements of R that belong to S_1; in the second step, those elements of R' that belong to S_1; then, those elements of R that belong to S_2 ; ...; and, finally, those elements of R that belong to S_σ.

Let $s_1, s_2, \ldots, s_\ell$ be the steps into which the comparisons performed by $\mathcal{A}$ are grouped. By construction, each step s_i consists in finding an integer $m_i \geqslant 1$ (or, possibly, $m_i = 0$ if s_i is the first step of a merge between two consecutive runs). If $\mathcal{A}$ uses TimSort's update policy, the step s_i consists in using McIlroy's strategy for a given parameter $\mathbf{t}(s_i)$ that depends on s_i. We also denote by $\mathbf{t}(s_{\ell+1})$ the parameter value obtained after $\mathcal{A}$ has finished sorting the array A.

Strategy B_0 requires m_i comparisons to find that integer m_i, and McIlroy's strategy requires m_i comparisons if $m_i \leqslant \mathbf{t}(s_i)$, up to $m_i + 1$ comparisons if $\mathbf{t}(s_i) + 1 \leqslant m_i \leqslant \mathbf{t}(s_i) + 6$, and up to $m_i - 1$ comparisons if $m_i \geqslant \mathbf{t}(s_i) + 7$. Since $\mathbf{t}(s_{i+1}) = \mathbf{t}(s_i)$ in the first case, $\mathbf{t}(s_{i+1}) = \mathbf{t}(s_i) + 1$ in the second case and $\mathbf{t}(s_{i+1}) = \max\{0, \mathbf{t}(s_i) - 1\}$ in the third case, McIlroy's strategy never uses more than $m_i + \mathbf{t}(s_{i+1}) - \mathbf{t}(s_i)$ comparisons. Consequently, the overhead of using TimSort's update policy instead of a naive merging sub-routine is at most $\mathbf{t}(s_{\ell+1}) - \mathbf{t}(s_1)$.

Moreover, let $\mu_i = m_1 + m_2 + \ldots + m_i$ for all $i \leqslant \ell$. We show by induction on τ that, whenever $\mathbf{t}(s_i) = \mathbf{t}(s_1) + \tau$, we have $2\mu_i \geqslant \tau^2$. Indeed, if $\tau \geqslant 1$ and if s_i is the first step for which $\mathbf{t}(s_i) = \mathbf{t}(s_1) + \tau$, we have $\mathbf{t}(s_{i-1}) = \mathbf{t}(s_1) + \tau - 1$. It follows that $2\mu_{i-1} \geqslant (\tau - 1)^2$ and $m_i \geqslant \mathbf{t}(s_1) + \tau \geqslant \tau$, which proves that $2\mu_i \geqslant (\tau - 1)^2 + 2\tau = \tau^2 + 1$. Thus, we conclude that $\mathbf{t}(s_\ell + 1) - \mathbf{t}(s_1) \leqslant \sqrt{2\mu_\ell}$.

Finally, μ_ℓ is equal to the number of element comparisons that $\mathcal{A}$ would perform if it used the naive merging strategy, i.e., $\mu_\ell = \mathsf{c}_1$. No merge sort requires more than $\mathcal{O}(n^2)$ comparisons, and therefore $\mu_\ell = \mathcal{O}(n^2)$, which is why the overhead of using TimSort's update policy is at most $\mathbf{t}(s_{\ell+1}) - \mathbf{t}(s_1) \leqslant \sqrt{2\mu_\ell} = \mathcal{O}(n)$. ◀

Deciding whether our results remain valid when $\mathbf{t}$ is updated like in TimSort remains an open question. However, in Section 5.3, we propose and study the following alternative update policy: when merging runs of lengths a and b, we set $\mathbf{t} = \lceil \log_2(a + b) \rceil$.

3 A fast-growth property and its consequences

In this section, we focus on two novel properties of stable natural merge sorts, which we call *fast-growth* and *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 subsequent sections 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 sub-arrays 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, 13, 19] are a convenient way to represent the succession of runs that ever occur while A is being sorted.

▶ **Definition 4.** *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. We will commonly denote runs with capital letters, possibly with some index or adornment, and we will then denote the length of such a run with the same small-case letter and the same index or adornment. For instance, runs named R, R_i, Q' and $\overline{S}$ will have respective lengths r, r_i, q' and $\overline{s}$.

▶ **Definition 5.** *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 $\theta > 1$ such that, for every merge tree $\mathcal{T}$ induced by $\mathcal{A}$ and every run at depth ℓ or more in $\mathcal{T}$, we have $r^{(\ell)} \geqslant \theta r$.*

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

> *There exists a real number $\kappa > 1$ 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 $r \geqslant \kappa^h$.*

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 also has the middle-growth property: indeed, it suffices to choose $\kappa = \min\{2, \theta\}^{1/\ell}$. As a result, the former property is stronger than the latter one, and indeed it has stronger consequences.

▶ **Theorem 6.** *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the fast-growth property. If $\mathcal{A}$ uses either the galloping or the naive sub-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 $\theta > 1$ 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 naive merging sub-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 $(k\ell)^{\text{th}}$ ancestor of R_i is a node R of size $r \geqslant \theta^k r_i$, and thus $n \geqslant r \geqslant \theta^k r_i$. Hence, $d_i + 1 \leqslant \ell(k+1) \leqslant \ell\,(\log_\theta(n/r_i) + 1)$, and we conclude that

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

A similar, weaker result also holds for algorithms with the middle-growth property.

▶ **Theorem 7.** *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. If $\mathcal{A}$ uses either the galloping or the naive sub-routine for merging runs, it requires $\mathcal{O}(n \log(n))$ element comparisons and moves to sort arrays of length n.*

Proof. Let us borrow the notations used when proving Theorem 6, and let $\kappa > 1$ be the real number mentioned in the definition of the statement "$\mathcal{A}$ has the middle-growth property". Like in the proof of Theorem 6, 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 $n \geqslant \kappa^{d_i}$. Hence, $d_i \leqslant \log_\kappa(n)$, and we conclude that

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

Theorems 6 and 7 provide us with a simple framework for recovering well-known results on the complexity of many algorithms. By contrast, Theorem 8 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 sub-routine.

▶ **Theorem 8.** *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the middle-growth property. If $\mathcal{A}$ uses the galloping sub-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. All comparisons performed by the galloping sub-routine are of the form $A[i] \leqslant^? A[j]$, where i and j are positions such that $i < j$. Thus, the behaviour of $\mathcal{A}$, i.e., the element comparisons and element moves it performs, is invariant under lexicographic equivalence, as illustrated in Figure 2. Consequently, starting from an array A of length n with σ dual runs $S_1, S_2, \ldots, S_\sigma$, we create a new array B of length n with σ distinct values, setting $B[j] \stackrel{\text{def}}{=\!=} i$ whenever $A[j]$ belongs to the dual run S_i, and we may now assume that A coincides with B. This assumption allows us to directly use Proposition 2, whose presentation would have been more complicated if we had referred to dual runs of an underlying array instead of referring directly to distinct values.

Now, let $\kappa > 1$ 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 and whose values are integers from 1 to σ, let $s_1, s_2, \ldots, s_\sigma$ be the lengths of its dual runs, 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 with value i. In the galloping cost model, 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 $\mathbf{t} = 0$.

Now, consider some integer $h \geqslant 0$, let $\mathcal{R}_h$ be the set of runs at height h in $\mathcal{T}$, and let $\mathsf{C}_0(h) = \sum_{R\in\mathcal{R}_h} \mathsf{cost}^*_0(r_{\to i})$. Since no run in $\mathcal{R}_h$ descends from another one, we already have

$$\mathsf{C}_0(h) \leqslant 2\sum_{R\in\mathcal{R}_h} r_{\to i} \leqslant 2s_i \text{ and } \sum_{R\in\mathcal{R}_h} r \leqslant n.$$

Moreover, by definition of κ, each run $R \in \mathcal{R}_h$ is of length $r \geqslant \kappa^h$, and thus $|\mathcal{R}_h| \leqslant n/\kappa^h$.

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

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

The inequalities on the first line respectively hold by definition of cost_1^*, because f is concave, and because f is increasing and $\sum_{R\in\mathcal{R}_h} r_{\to_i} \leqslant s_i$; the inequalities on the second line hold because g is increasing and $|\mathcal{R}_h| \leqslant n/\kappa^h$.

We conclude that

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

◀

4 PowerSort has the fast-growth property

In this section, we prove that PowerSort and many TimSort-like algorithms enjoy the fast- or middle-growth properties. To that aim, we first define the run merge policy of PowerSort, by introducing the notion of *power* of a run endpoint or of a run, and then characterising the merge trees that PowerSort induces.

▶ **Definition 9.** *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 $0 \leqslant i \leqslant \rho$, we denote by $\mathbf{I}(i)$ the half-open interval $(e_{i-1}+e_i, e_i+e_{i+1}]$. The power *of e_i, which we denote 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 $p_\rho = 0$.*

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 is defined as* $\max\{p_{i-1}, p_j\}$.

The notion of power quickly comes with nice properties, two of which we mention now.

▶ **Lemma 10.** *For each non-empty sub-interval I of the set $\{0, \ldots, \rho\}$, there exists a unique integer $i \in I$ 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. Thus, our initial assumption is invalid, which completes the proof. ◀

▶ **Lemma 11.** *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 $R_{i\dots j}$ that belongs to $\mathcal{T}$ has a power that is smaller than all of $p_i, \dots, p_{j-1}$;
S_3: if a run $R_{i\dots j}$ is an inner node of $\mathcal{T}$, its children are the two runs $R_{i\dots k}$ and $R_{k+1\dots j}$ such that $p_k = \min\{p_i, \dots, p_{j-1}\}$.

First, if S_1 holds, we prove S_3 by induction on the height h of the run $R_{i\dots j}$. Indeed, if the restriction of S_3 to runs of height less than h holds, let $R_{i\dots k}$ and $R_{k+1\dots j}$ be the children of a run $R_{i\dots j}$ of height h. If $i < k$, the run $R_{i\dots k}$ has two children $R_{i\dots \ell}$ and $R_{\ell+1\dots k}$ such that $p_\ell = \min\{p_i, \dots, p_{k-1}\}$, 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 $R_{i\dots k}$, i.e., $\max\{p_{i-1}, p_k\}$, which proves that $p_\ell > p_k$. It follows that $p_k = \min\{p_i, \dots, p_k\}$, and one proves similarly that $p_k = \min\{p_k, \dots, p_{j-1}\}$, 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 10 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 10 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 Lemma 11, and the second claim of Lemma 11 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 [19, Lemma 4] and which we consider as an alternative definition of PowerSort.

▶ **Definition 12.** *In every merge tree that PowerSort induces, inner nodes have a smaller power than their children.*

▶ **Lemma 13.** *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 symmetric. Lemma 11 states that all of $p_i, \dots, p_{j-1}$ are larger than p. 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 $\mathcal{S} = \{kn/2^{p-2} : k \in \mathbb{Z}\}$. The bounds $e_{i-1} + e_i$ and $e_j + e_{j+1}$ are therefore contained between two consecutive elements of $\mathcal{S}$, i.e., there exists an integer ℓ such that

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

and 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 its parent are left children, the other possible cases being symmetric. There exist integers u and v such that the parent of R is $R_{i\ldots u}$, and its grandparent is $R_{i\ldots v}$. 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 $\mathcal{S}' = \{kn/2^{p-1} : k \in \mathbb{Z}\}$. 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

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

▶ **Theorem 14.** *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 12 ensures that $p \geqslant p^{(3)} + 3$, and therefore Lemma 13 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. ◀

For the sake of conciseness, we only list which algorithms were found to have the fast- or middle-growth property. Proofs that they do can be found in the complete version of this article [11].

▶ **Theorem 15.** *The algorithms TimSort, α-MergeSort, PeekSort and adaptive ShiversSort have the fast-growth property.*

An immediate consequence of Theorems 6 and 8 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 sub-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 16.** *The algorithms NaturalMergeSort, ShiversSort and α-StackSort have the middle-growth property.*

Theorem 8 proves that, if these three algorithms are used with the galloping merging sub-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 [13] 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 6 shows that these three algorithms do *not* have the fast-growth property.

5 Refined complexity bounds for PowerSort

One weakness of Theorem 8 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. In this section, we study these constants, thereby focusing 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, in general, requires at least $\log_2(n!) = n\log_2(n) + \mathcal{O}(n)$ comparisons, 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 $\mathsf{c} = 1$.

5.1 A tight middle-growth property

Below, we aim at computing the least constant that might lie hidden in the $\mathcal{O}(n + n\mathcal{H}^*)$ upper bound of Theorem 8. If we were to simply extract that constant from the proof we gave, this constant would depend directly on the real number κ mentioned in the definition of the statement "$\mathcal{A}$ has the middle-growth property". To that aim, we make that statement more precise.

▶ **Definition 17.** *We say that a stable natural merge sort algorithm $\mathcal{A}$ has the* tight *middle-growth property if it satisfies the following statement:*

> *There exists an integer $\theta \geqslant 0$ 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 $r \geqslant 2^{h-\theta}$.*

Since every node of height $h \geqslant 1$ in a merge tree is a run of length at least 2, each algorithm with the tight middle-growth property also has the middle-growth property: indeed, it suffices to choose $\kappa = 2^{1/(\theta+1)}$. The tight middle-growth property is incomparable with the fast-growth property since, for instance, adaptive ShiversSort and PeekSort fail to have the tight middle-growth property. In practice, we might also introduce a related notion of tight fast-growth property, which would be useful when evaluating the constant hidden in the $\mathcal{O}(n + n\mathcal{H})$ upper bound on the complexity of sorting algorithms.

▶ **Theorem 18.** *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. Let also p and $p^{(h-2)}$ be the respective powers of the runs R and $R^{(h-2)}$. Definition 12 and Lemma 13 prove that $2^{p^{(h-2)}+h-4} \leqslant 2^{p-2} \leqslant 2^{p-2}r < n < 2^{p^{(h-2)}}r^{(h)}$. ◀

5.2 Using a fixed parameter $\mathbf{t}$

Following the structure of Section 3, we prove now that each algorithm with the tight middle-growth property, such as PowerSort, enjoys excellent upper bounds on the number of element comparisons it requires.

▶ **Theorem 19.** *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 sub-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 8. Let θ 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 8, we just need to prove that

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

for all $i \leqslant \sigma$.

Then, 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 proves that

$$\sum_{R \in \mathcal{R}_h} r_{\to i} \leqslant s_i \text{ and that } \sum_{R \in \mathcal{R}_h} r \leqslant n.$$

Since each run $R \in \mathcal{R}_h$ is of length $r \geqslant 2^{h-\theta}$, it follows that $|\mathcal{R}_h| \leqslant n/2^{h-\theta}$.

Then, consider the function

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

We noted above 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 $h \geqslant 0$.

Let also $f\colon x \mapsto \mathbf{t} + 2 + 2\log_2(x+1)$, $g\colon x \mapsto x\, f(s_i/x)$ and $\mu = \lceil \log_2((\mathbf{t}+1)n/s_i) \rceil$. Both functions f and g are positive, concave and increasing on $(0, +\infty)$, which shows that

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

We conclude that

$$\begin{aligned}
\sum_{R \in \mathcal{T}} \mathsf{cost}^*_{\mathbf{t}}(r_{\to i}) = \sum_{h \geqslant 0} \mathsf{C}_{\mathbf{t}}(h) &= \sum_{h=0}^{\mu+\theta-1} \mathsf{C}_{\mathbf{t}}(h) + \sum_{h \geqslant 0} \mathsf{C}_{\mathbf{t}}(\mu+\theta+h) \\
&\leqslant (1 + 1/(\mathbf{t}+3))(\mu+\theta)s_i + 4s_i \sum_{h \geqslant 0} 2^{-h} + 2s_i \sum_{h \geqslant 0} h 2^{-h} \\
&\leqslant (1 + 1/(\mathbf{t}+3))(\log_2(n/s_i) + \log_2(\mathbf{t}+1))s_i + \mathcal{O}(s_i) \\
&\leqslant (1 + 1/(\mathbf{t}+3))\log_2(n/s_i)s_i + \log_2(\mathbf{t}+1)s_i + \mathcal{O}(s_i).
\end{aligned}$$

◀

5.3 Using a parameter $\mathbf{t}$ with logarithmic growth

The upper bound provided by Theorem 19 is minimal when $\mathbf{t} = \Theta(\mathcal{H}^*)$, in which case it simply becomes $n\mathcal{H}^* + \log_2(\mathcal{H}^*+1)n + \mathcal{O}(n)$. However, computing $\mathcal{H}^*$ before starting the actual sorting process is not reasonable. Instead, we update the parameter $\mathbf{t}$ as follows, which will provide us with a slightly larger upper bound.

▶ **Definition 20.** *We call* logarithmic *galloping sub-routine the merging sub-routine that, when merging adjacent runs of lengths a and b, performs the same comparisons and element moves as the $\mathbf{t}$-galloping sub-routine for $\mathbf{t} = \lceil \log_2(a+b) \rceil$.*

▶ **Theorem 21.** *Let $\mathcal{A}$ be a stable natural merge sort algorithm with the tight middle-growth property. If $\mathcal{A}$ uses the logarithmic galloping sub-routine for merging runs, it requires at most $n\mathcal{H}^* + 2\log_2(\mathcal{H}^*+1)n + \mathcal{O}(n)$ element comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$.*

Proof. Let us refine and adapt the proofs of Theorems 8 and 19. Let θ 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.

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

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

element comparisons. Given that

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

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

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

element comparisons, where $\mathsf{cost}^*_{\log}(r, m) = \min\{(1 + 1/\log_2(r))m, 6\log_2(r+1) + 6\}$.

Then, let $\mathcal{T}^*$ denote the tree obtained after removing the leaves of $\mathcal{T}$. We focus on proving that

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

for all $i \leqslant \sigma$. Indeed, finding the run decomposition $R_1, R_2, \ldots, R_\rho$ of A requires $n-1$ comparisons, and $\rho - 1 \leqslant n - 1$ merges are then performed, which will make a total of up to

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

comparisons, the latter inequality being due to the concavity of the function $x \mapsto \log_2(x+1)$.

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

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

No run in $\mathcal{R}_h$ descends from another one, and each run $R \in \mathcal{R}_h$ has length $r \geqslant 2^{\max\{1, h-\theta\}}$, which proves that $|\mathcal{R}_h| \leqslant n/2^{h-\theta}$ and that

$$\mathsf{C}_{\log}(h) \leqslant \sum_{R \in \mathcal{R}_h} (1+1/\log_2(r))r_{\to i} \leqslant \sum_{R \in \mathcal{R}_h} (1+1/\max\{1, h-\theta\})r_{\to i} \leqslant (1+1/\max\{1, h-\theta\})s_i.$$

Finally, let $z = n/s_i \geqslant 1$, and consider the constant $\nu = \lceil \log_2(z\log_2(2z)) \rceil + \theta$ and the functions $f\colon x \mapsto 1 + \log_2(x+1)$ and $g\colon x \mapsto x\, f(n/x)$. Both functions f and g are concave, positive and increasing on $(0, +\infty)$, which proves that

$$\begin{aligned}\mathsf{C}_{\log}(\nu + h)/6 &\leqslant \sum_{R \in \mathcal{R}_{\nu+h}} f(r) \leqslant |\mathcal{R}_{\nu+h}|\, f\big(\textstyle\sum_{R \in \mathcal{R}_{\nu+h}} r/|\mathcal{R}_{\nu+h}|\big) \leqslant g(|\mathcal{R}_{\nu+h}|) \leqslant g(n/2^{\nu-\theta+h}) \\ &\leqslant g\big(2^{-h}s_i/\log_2(2z)\big) = 2^{-h}s_i f(2^h z\log_2(2z))/\log_2(2z) \\ &\leqslant 2^{-h}s_i\big(1 + \log_2(2^h(2z)^2)\big)/\log_2(2z) = 2^{-h}(h+1)s_i/\log_2(2z) + 2^{1-h}s_i \\ &\leqslant (h+3)2^{-h}s_i,\end{aligned}$$

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

$$1 + 2^h z \log_2(2z) \leqslant 1 + 2^{h+1} z^2 \leqslant 2^h (1 + 2z^2) \leqslant 2^h \times (2z)^2$$

whenever $h \geqslant 0$ and $z \geqslant 1$.

It follows that

$$\sum_{h=1}^{\theta} \mathsf{C}_{\log}(h) + \sum_{h \geqslant 0} \mathsf{C}_{\log}(\nu + h) \leqslant 2\theta s_i + 6 \sum_{h \geqslant 0} (h+3) 2^{-h} s_i = \mathcal{O}(s_i),$$

whereas

$$\sum_{h=1}^{\nu-\theta-1} \mathsf{C}_{\log}(\theta + h) \leqslant \sum_{h=1}^{\nu-1} (1 + 1/h) s_i \leqslant ((\nu - 1) + 1 + \ln(\nu - 1)) s_i = (\nu + \ln(\nu)) s_i.$$

Thus, we conclude that

$$\begin{aligned} \sum_{R \in \mathcal{T}^*} \mathsf{cost}^*_{\log}(r, r_{\to i}) = \sum_{h \geqslant 1} \mathsf{C}_{\log}(h) &\leqslant s_i(\nu + \ln(\nu)) + \mathcal{O}(s_i) \\ &\leqslant s_i \log_2(z) + s_i \log_2(\log_2(2z)) + s_i \log_2(\log_2(2z^2)) + \mathcal{O}(s_i) \\ &\leqslant s_i \log_2(z) + 2 s_i \log_2(\log_2(2z)) + \mathcal{O}(s_i). \end{aligned}$$

◀

It is of course possible to marginally improve our update policy in order to approach the $n\mathcal{H}^* + \log_2(\mathcal{H}^* + 1)n + \mathcal{O}(n)$ upper bound. For instance, choosing $\mathbf{t} = \tau\lceil \log_2(a+b) \rceil$ for a given constant $\tau \geqslant 1$ provides us with an $n\mathcal{H}^* + (1 + 1/\tau) \log_2(\mathcal{H}^* + 1)n + \log_2(\tau) + \mathcal{O}(n)$ upper bound, and choosing $\mathbf{t} = \lceil \log_2(a+b) \rceil \times \lceil \log_2(\log_2(a+b)) \rceil$ further improves that upper bound. However, such improvements soon become negligible in comparison with the overhead of having to compute the value of $\mathbf{t}$.

Finally, and like in Section 4, we list a few algorithms that enjoy similar complexity upper bounds. Proofs that they do can be found in the complete version of this article [11].

▶ **Theorem 22.** *The algorithms* NaturalMergeSort *and* ShiversSort *have the tight middle-growth property.*

▶ **Theorem 23.** *Theorems 19 and 21 remain valid if we consider the algorithms* PeekSort *and* adaptive ShiversSort *instead of an algorithm with the tight middle-growth property.*

By contrast, we conjecture that no choice policy for the parameters $\mathbf{t}$ would provide us with $(1 + o(1))n\mathcal{H}^* + \mathcal{O}(n)$ upper bounds on the number of element comparisons performed by TimSort, α-StackSort or α-MergeSort. However, finding precise characterisations of the best constants c that could be achieved for these algorithms is a wide open question.

6 Conclusion: An idealistic galloping cost model

In the above sections, we observed the impact of using a galloping sub-routine for a fixed or a variable parameter $\mathbf{t}$. Although choosing a constant value of $\mathbf{t}$ (e.g., $\mathbf{t} = 7$, as advocated in [17]) already leads to very good results, letting $\mathbf{t}$ vary, for instance by using the logarithmic variant of the sub-routine, provides us with even better complexity guarantees, with an often negligible overhead of $\mathcal{O}(n \log(\mathcal{H}^* + 1) + n)$ element comparisons: up to a small error, this provides us with the following *idealistic* cost model for run merges, allowing us to simultaneously identify the parameter $\mathbf{t}$ with $+\infty$ and with a constant.

▶ **Definition 24.** *Let A and B be two non-decreasing runs with $a_{\to i}$ (respectively, $b_{\to i}$) elements of value i for all $i \in \{1, 2, \ldots, \sigma\}$. The* idealistic galloping cost *of merging A and B is defined as the quantity*

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

where $\mathsf{cost}^*_{\mathsf{ideal}}(m) = \min\{m, \log_2(m+1) + \mathcal{O}(1)\}$.

Indeed, abusively identifying the real number of element comparisons performed while runs A and B with this idealistic galloping cost, we may prove that every stable natural merge sort with the tight middle-growth property (but also related algorithms such as PeekSort and adaptive ShiversSort) requires at most $n\mathcal{H}^* + \mathcal{O}(n)$ element comparisons to sort arrays of length n and dual run-length entropy $\mathcal{H}^*$.

We think that this idealistic cost model is both simple and precise enough to allow studying the complexity of natural merge sorts in general, provided that they use the galloping sub-routine. Thus, it would be interesting to use that cost model in order to study, for instance, the least constant c for which various algorithms such as TimSort or α-MergeSort require up to $\mathsf{c}n(1 + o(1))\mathcal{H}^* + \mathcal{O}(n)$ element comparisons.

We also hope that this simpler framework will foster the interest for the galloping merging sub-routine of TimSort, and possibly lead to amending Swift and Rust implementations of TimSort to include that sub-routine, which we believe is too efficient in relevant cases to be omitted.

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: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*, 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. Jon Bentley and Andrew Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.
6. Josh Bloch. Timsort implementation in java 13, retrieved 01/01/2022. URL: `https://github.com/openjdk/jdk/blob/3afeb2cb4861f95fd20c3c04f04be93b435527c0/src/java.base/share/classes/java/util/ComparableTimSort.java`.
7. Sam Buss and Alexander Knop. Strategies for stable merge sorting. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1272–1290, 2019.
8. Ben Cohen. Timsort implementation in java 13, retrieved 01/01/2022. URL: `https://github.com/apple/swift/blob/4c1d46bc0980d84bf3178bc42295522c242fec86/ stdlib/public/core/Sort.swift`.
9. Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.
10. Adriano Garsia and Michelle Wachs. A new algorithm for minimal binary search trees. *SIAM Journal on Computing*, 6(4):622–642, 1977.
11. Elahe Ghasemi, Vincent Jugé, and Ghazal Khalighinejad. Galloping in natural merge sorts. In *49th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 61:1–61:19, 2022. Extended version available at: `arrXiv:2012.03996`.

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

13 Vincent Jugé. Adaptive shivers sort: an alternative sorting algorithm. In *31th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1639–1654, 2020.

14 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publish. Co., 1998.

15 Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences. *Information and Computation*, 112(1):37–50, 1994.

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

17 Peter McIlroy. Optimistic sorting and information theoretic complexity. In *4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 467–474, 1993.

18 Ian Munro and Philip Spira. Sorting and searching in multisets. *SIAM journal on Computing*, 5(1):1–8, 1976.

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

20 Tim Peters. Timsort description, retrieved 01/09/2021. URL: `https://github.com/python/cpython/blob/24e5ad4689de9adc8e4a7d8c08fe400dcea668e6/Objects/listsort.txt`.

21 Clément Renault et al. Timsort implementation in rust, retrieved 01/01/2022. URL: `https://github.com/rust-lang/rust/blob/a5a91c8e0732753de7c028182cbb02901fe1b608/library/alloc/src/slice.rs`.

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

23 Guido van Rossum et al. Powersort implementation in cpython, retrieved 01/01/2022. URL: `https://github.com/python/cpython/blob/c8749b578324ad4089c8d014d9136bc42b065343/Objects/listobject.c`.

24 Jeff Weaton and Markus Mützel. Timsort implementation in octave, retrieved 01/01/2022. URL: `https://github.com/gnu-octave/octave/blob/7e6da6d504ad962a8d33622b6955b0210ff62365/liboctave/util/oct-sort.cc`.

25 Simon Zünd et al. Timsort implementation in v8, retrieved 01/01/2022. URL: `https://github.com/v8/v8/blob/25f0e32915930df1d53722b91177b1dee5202499/third_party/v8/builtins/array-sort.tq`.