Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs

J. Ian Munro
University of Waterloo, Canada
imunro@uwaterloo.ca
https://orcid.org/0000-0002-7165-7988

Sebastian Wild
University of Waterloo, Canada
wild@uwaterloo.ca
https://orcid.org/0000-0002-6061-9177

Abstract

We present two stable mergesort variants, "peeksort" and "powersort", that exploit existing runs and find nearly-optimal merging orders with negligible overhead. Previous methods either require substantial effort for determining the merging order (Takaoka 2009 [28]; Barbay & Navarro 2013 [3]) or do not have an optimal worst-case guarantee (Peters 2002 [23]; Auger, Nicaud & Pivoteau 2015 [1]; Buss & Knop 2018 [7]). We demonstrate that our methods are competitive in terms of running time with state-of-the-art implementations of stable sorting methods.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases adaptive sorting, nearly-optimal binary search trees, Timsort

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs Programme.

1 Introduction

Sorting is a fundamental building block for numerous tasks and ubiquitous in both the theory and practice of computing. While practical and theoretically (close-to) optimal comparison-based sorting methods are known, *instance-optimal sorting,* i.e., methods that *adapt* to the actual input and exploit specific structural properties if present, is still an area of active research. We survey some recent developments in Section 1.1.

Many different structural properties have been investigated in theory. Two of them have also found wide adoption in practice, e.g., in Oracle's Java runtime library: adapting to the presence of duplicate keys and using existing sorted segments, called *runs*. The former is achieved by a so-called fat-pivot partitioning variant of quicksort [6], which is also used in the GNU implementation of `std::sort` from the C++ STL. It is an *unstable* sorting method, though, i.e., the relative order of elements with equal keys might be destroyed in the process. It is hence used in Java solely for primitive-type arrays.

Making use of existing runs in the input is a well-known option in mergesort; e.g., Knuth [15] discusses a bottom-up mergesort variant that does this. He calls it "natural mergesort" and we will use this as an umbrella term for any mergesort variant that picks up existing runs in the input (instead of starting blindly with runs of size 1). The Java library uses Timsort [23, 13] which is such a natural mergesort originally developed as Python's new library sort.

While fat-pivot quicksort provably adapts to the *entropy of the multiplicities* of keys [32] – it is optimal up to a factor of 1.088 on average with pseudomedian-of-9 (“ninther”) pivots[1] – Timsort is much more heuristic in nature. It picks up existing runs and tries to perform merges in a favorable order (i.e., avoiding merges of runs with very different lengths), but many desirable guarantees are missing: Although it was announced as an $O(n \log n)$ worst-case method with its introduction in Python in 2002 [22], a rigorous proof of this bound was only given in 2015 by Auger, Nicaud, and Pivoteau [1] and required a rather sophisticated amortization argument.[2] The core complication is that – unlike for standard mergesort variants – a given element might participate in more than a logarithmic number of merges. Indeed, Buss and Knop [7] have very recently shown that for some family of inputs, the average number of merges a single element participates in is at least $\big(\frac{3}{2} - o(1)\big) \cdot \lg n$. So in the worst case, Timsort does, e.g., (at least) *1.5 times as many element moves as standard mergesort.*

In terms of adapting to existing order, the only proven guarantee for Timsort’s running time is a trivial $O(nr)$ bound when the input consists of r runs. Proving an informative upper bound like $O(n+n \log r)$ has remained elusive, (although it is conjectured to hold in [1] and [7]). This is in sharp contrast to available alternatives: Takaoka [27, 28] and Barbay and Navarro [3] independently discovered a sorting method that adapts to the *entropy of the distribution of run lengths:* they sort an input consisting of r runs with respective lengths $L_1, \ldots, L_r$ in time $O\big((\mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_r}{n}) + 1)n\big) \subseteq O(n + n \lg r)$, where $\mathcal{H}(p_1, \ldots, p_r) = \sum_{i=1}^{r} p_i \lg(1/p_i)$ is the binary Shannon entropy. Since $\mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_r}{n})n - O(n)$ comparisons are necessary for distinct keys, this is optimal up to linear terms. Their algorithms are also conceptually simple: find runs in a linear scan, determine an optimal merging order using a Huffman tree of the run lengths, and execute those merges bottom-up in the tree. We will refer to this algorithm to determine an optimal merging order as *Huffman-Merge.*

Straight-forward implementations of Huffman-Merge add significant overhead in terms of time and space; (finding the Huffman tree requires storing and sorting the run lengths). This renders these methods uncompetitive to (reasonable implementations of) elementary sorting methods. Moreover, Huffman-Merge leads to an *unstable* sorting method since it merges non-adjacent runs. The main motivation for the invention of Timsort was to find a fast general-purpose sorting method that is *stable* [22], and the Java library even dictates the sorting method used for objects to be stable. We remark that while stability is a much desired feature, practical, stable sorting methods do not try to *exploit* the presence of duplicate elements to speed up sorting, and we will focus on the performance for distinct keys in this article.

It is conceptually simple to modify the idea of Takaoka resp. Barbay-Navarro to sort stably: replace the Huffman tree by an *optimal binary search tree* and otherwise proceed as before (using a stable merging procedure). Since we only have weights at the leaves of the tree, we can compute this tree in $O(n + r \log r)$ time using the Hu-Tucker- or Garsia-Wachs-algorithm, but r can be $\Theta(n)$ and the algorithms are fairly sophisticated, so this idea seems not very appealing for practical use.

[1] The median of three elements is chosen as the pivot, each of which is a median of three other elements. This is a good approximation of the median of 9 elements and often used as pivot selection rule in library implementations.

[2] A further manifestation of the complexity of Timsort was reported by de Gouw et al. [8]: The original rules to maintain the desired invariant for run lengths on the stack was not sufficient in some cases. This (algorithmic!) bug had remained unnoticed until their attempt to formally verify the correctness of the Java implementation of Timsort failed because of it.

In this paper, we present two new natural mergesort variants that have the same optimal asymptotic running time $O\big((\mathcal{H}(\frac{L_1}{n},\ldots,\frac{L_r}{n})+1)n\big)$ as Huffman-merge, but incur much less overhead. For that, we build upon classic algorithms for computing *nearly-optimal binary search trees* [19]; but the vital twist for practical methods is to neither explicitly store the full tree, nor the lengths of all runs at any point in time. In particular – much like Timsort – we only store a *logarithmic* number of runs at any point in time (in fact reducing their number from roughly $\log_\varphi \approx 1.44 \lg n$ in Timsort to $\lg n$), but – much *un*like Timsort – we retain the guarantee of an optimal merging order up to linear terms. Our methods require at most $n \lg n + O(n)$ comparison in the worst case and $\mathcal{H}(\frac{L_1}{n},\ldots,\frac{L_r}{n})n + 3n$ for an input with runs of lengths $L_1,\ldots,L_r$.

We demonstrate in a running-time study that our methods achieve guaranteed (leading-term) optimal adaptive sorting in practice with negligible overhead to compute the merge order: unlike Timsort, our methods are *not* slower than standard mergesort when no existing runs can be exploited. If existing runs are present, mergesort and quicksort are outperformed by far. Finally, we show that Timsort is slower than standard mergesort and our new methods on certain inputs that do have existing runs, but whose lengths pattern hits a weak point of Timsort's heuristic merging-order rule.

Outline: The rest of this paper is organized as follows. In the remainder of this section we survey related work. Section 2 contains notation and known results on optimal binary search trees that our work builds on. The new algorithms and their analytical guarantees are presented in Section 3. Section 4 reports on our running-time study, comparing the the new methods to state-of-the-art sorting methods. Finally, Section 5 summarizes our findings.

1.1 Adaptive Sorting

The idea to exploit existing "structure" in the input to speed up sorting dates (at least) back to methods from the 1970s [18] that sort faster when the number of inversions is small. A systematic treatment of this and many further measures of presortedness (e.g., the number of inversions, the number of runs, and the number of shuffled up-sequences), their relation and how to sort *adaptively* w.r.t. these measures are discussed by Estivill-Castro and Wood [10]. While the focus of earlier works is mostly on combinatorial properties of permutations, a more recent trend is to consider more fine-grained statistical quantities. For example, the above mentioned Huffman-Merge adapts to the *entropy* of the vector of run lengths [27, 28, 3]. Other similar measures are the entropy of the lengths of shuffled up-sequences [3] and the entropy of lengths of an LRM-partition [2], a novel measure that lies between runs and shuffled up-sequences.

For multiset sorting, the fine-grained measure, the *entropy of the multiplicities*, has been considered instead of the *number* of unique values already in early work in the field (e.g. [20, 24]). A more recent endeavor has been to find sorting methods that optimally adapt to *both presortedness and repeated values.* Barbay, Ochoa, and Satti refer to this as *synergistic sorting* [4] and present an algorithm based on quicksort that is optimal up to a constant factor. The method's practical performance is unclear.

We remark that (unstable) multiset sorting is the *only* problem from the above list for which a theoretically optimal algorithm has found wide-spread adoption in programming libraries: quicksort is known to almost optimally adapt to the entropy of multiplicities on average [30, 26, 32], when elements equal to the pivot are excluded from recursive calls (fat-pivot partitioning). Supposedly, sorting is so fast to start with that further improvements from exploiting specific input characteristics are only fruitful if they can be realized with minimal additional overhead. Indeed, for algorithms that adapt to the number of inversions, Elmasry and Hammad [9] found that the adaptive methods could only compete with good implementations of elementary sorting algorithms in terms of running time for inputs with extremely few inversions (fewer than 1.5%). Translating the theoretical guarantees of adaptive sorting into practical, efficient methods is an ongoing challenge.

1.2 Lower bound

How much does it help for sorting an array $A[1..n]$ to know that it contains r runs of respective sizes $L_1, \ldots, L_r$, i.e., to know the relative order of $A[1..L_1]$, $A[L_1+1..L_1+L_2]$ etc.? If we assume distinct elements, a simple counting argument shows that there are $\binom{n}{L_1,\ldots,L_r}$ permutations that are compatible with this setup. (the number of ways to partition n keys into r subsets of given sizes.) We thus need $\lg(n!) - \sum_{i=1}^{r} \lg(L_i!) = \mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_r}{n})n - O(n)$ comparisons to sort such an input. A formal argument for this lower bound is given by Barbay and Navarro [3] in the proof of their Theorem 2.

1.3 Results on Timsort and stack-based mergesort

Its good performance in running-time studies especially on partially sorted inputs have lead to the adoption of Timsort in several programming libraries, but as mentioned in the introduction, the complexity of the algorithm has precluded researchers from proving interesting adaptivity guarantees. To make progress towards these, simplified variations of Timsort have been considered [1, 7]. All of those methods work by maintaining a stack of runs yet to be merged and proceed as follows: They find the next run in the input and push it onto the stack. Then they consider the top k elements on the stack (for k a small constant like 3 or 4) and decide based on these if any pair of them is to be merged. If so, the two runs in the stack are replaced with the merged result and the rule is applied repeatedly until the stack satisfies some invariant. The invariant is chosen so as to keep the height of the stack small (logarithmic in n).

The simplest version, “α-stack sort” [1], merges the topmost two runs until the run lengths in the stack grow at least by a factor of α, (e.g., $\alpha = 2$). This method can lead to imbalanced merges (and hence runtime $\omega(n \log r)$ [7]; the authors of [1] also point this out in their conclusion): if the next run is much larger than what is on the stack, a much more balanced merging order results from first merging stack elements until they are at least as big as the new run. This modification is called “α-merge sort”, which achieves a worst-case guarantee of $O(n + n \log r)$, but the constant is provably not optimal [7] (for any $\alpha > 1$).

Timsort is quite similar to α-merge sort for $\alpha = \varphi$ (the golden ratio) by forcing the run lengths to grow at least like Fibonacci numbers. The details of the rule are given in [1] or [7] and are quite intricate – and were indeed wrong in the first (widely-used) version of Timsort (see Footnote 2 on page 2). While it is open if Timsort always runs in $O(n + n \log r)$ time, Buss and Knop gave a family of inputs for which Timsort does asymptotically at least 1.5 times the required effort (in terms of merge costs, see Section 2.2), and hence proved that Timsort – like α-merge sort – is *not* optimally adaptive even to the *number* of runs r, not to speak of the entropy of the run lengths.

2 Preliminaries

We implicitly assume that we are sorting an array $A[1..n]$ of n elements. By $\mathcal{H}$, we denote the binary Shannon entropy, i.e., for $p_1, \ldots, p_m \in [0, 1]$ with $p_1 + \cdots + p_m = 1$ we let $\mathcal{H}(p_1, \ldots, p_m) = \sum p_i \lg(1/p_i)$, where $\lg = \log_2$.

We will always let r denote the *number of runs* in the input and $L_1, \ldots, L_r$ their respective lengths with $L_1 + \cdots + L_r = n$. In the literature, a *run* usually means a maximal (contiguous) weakly increasing[3] region, but we adopt the convention from Timsort in this paper: a run is either a maximal *weakly increasing* region *or* a *strictly decreasing* region. Decreasing runs are immediately reversed; allowing only strict decreasing runs makes their *stable* reversal trivial. The algorithms are not directly affected by different conventions for what a "run" is; they only rely on a unique partition of the input into sorted segments that can be found by sequential scans.

2.1 Nearly-Optimal Binary Search Trees

In the *optimal binary search tree problem,* we are given probabilities $\beta_1, \ldots, \beta_m$ to access the m keys $K_1 < \cdots < K_m$ (internal nodes) and probabilities $\alpha_0, \ldots, \alpha_m$ to access the gaps (leaves) between these keys (setting $K_0 = -\infty$ and $K_{m+1} = +\infty$) and we are interested in the binary search tree that minimizes the expected search cost C, i.e., the expected number of (ternary) comparisons when access follow the given distribution.[4] Nagaraj [21] surveys various versions of the problem. We confine ourselves to *approximation algorithms* here. Moreover, we only need the special case of *alphabetic trees* where all $\beta_j = 0$.

The following methods apply to the general problem, but we present them for the case of *nearly-optimal alphabetic trees.* So in the following let $\alpha_0, \ldots, \alpha_m$ with $\sum_{i=0}^{m} \alpha_i = 1$ be given. If the details are done right, a greedy top-down approach produces provably good search trees [5, 17]: choose the boundary closest to $\frac{1}{2}$ as the bisection at the root (*"weight-balancing heuristic"*). Mehlhorn [19, §III.4.2] discusses two algorithms for nearly-optimal binary search trees that follow this scheme: "Method 1" is the straight-forward recursive application of the above rule, whereas "Method 2" (*"bisection heuristic"*) continues by strictly halving the *original* interval in the recursive calls; see Figure 1.

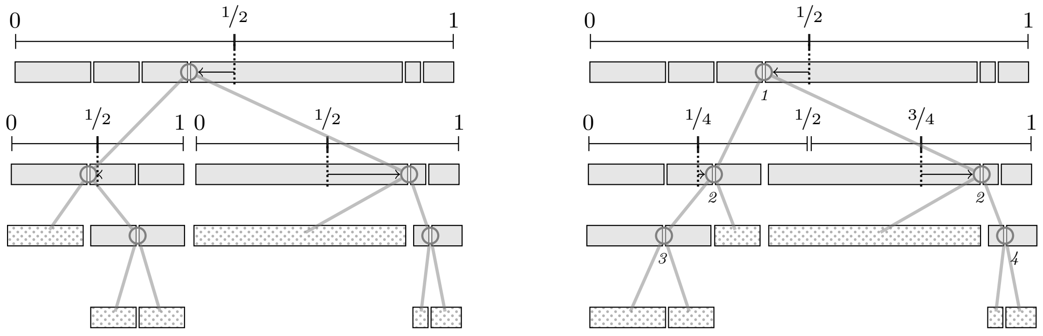


Figure 1 The two versions of weight-balancing for computing nearly-optimal alphabetic trees. The gap probabilities are proportional to 5, 3, 3, 14, 1, 2. **Left:** Mehlhorn's "Method 1" chooses the split closest to the midpoint of the subtree's actual weights (1/2 after renormalizing). **Right:** "Method 2" continues to cut the original interval in half, irrespective of the total weight of the subtrees. The italic numbers are the powers of the nodes (see Definition 3 on page 8).

Method 1 was proposed in [29] and analyzed in [16, 5]; Method 2 is discussed in [17].

[3] We say "weakly increasing" instead of "nondecreasing".
(Isn't it better to say what we mean instead of not saying what we don't mean?)

[4] We deviate from the literature convention and use m to denote the number of keys to avoid confusion with n, the length of the arrays to sort, in the rest of the paper.

While Method 1 is arguably more natural, Method 2 has the advantage to yield splits that are predictable without going through all steps of the recursion. Both methods can be implemented to run in time $O(m)$ and yield very good trees. (Recall that in the case $\beta_j = 0$ the classic information-theoretic argument dictates $C \ge \mathcal{H}$; Bayer [5] gives lower bounds in the general case.)

▶ **Theorem 1** (Nearly-Optimal BSTs). *Let $\alpha_0, \beta_1, \alpha_1, \ldots, \beta_m, \alpha_m \in [0,1]$ with $\sum \alpha_i + \sum \beta_j = 1$ be given and let $\mathcal{H} = \sum_{i=0}^{m} \alpha_i \lg(1/\alpha_i) + \sum_{j=1}^{m} \beta_j \lg(1/\beta_j)$.*

(i) *Method 1 yields a tree with search cost $C \le \mathcal{H} + 2$. [5, Thm 4.8]*

(ii) *If all $\beta_j = 0$, Method 1 yields a tree with search cost $C \le \mathcal{H} + 2 - (m+3)\alpha_{\min}$, where $\alpha_{\min} = \min\{\alpha_0, \ldots, \alpha_m\}$. [14]*

(iii) *Method 2 yields a tree with search cost $C \le \mathcal{H} + 1 + \sum \alpha_i$. [17]*

2.2 Merge Costs

In this paper, we are primarily concerned with finding a good *order* of binary merges for the existing runs in the input. Following [1] and [7], we will define the *merge cost M* for merging two runs of lengths m resp. n as $M = m + n$, i.e., the size of the result. This quantity has been studied earlier by Golin and Sedgewick [12] without giving it a name.

Merge costs abstract away from key comparisons and element moves and simplify computations (see next subsection). Since any merge has to move most elements (except for rare lucky cases), and the average number of comparisons using standard merge routines is $m + n - \bigl(\frac{m}{n+1} + \frac{n}{m+1}\bigr)$, merge costs are a reasonable approximation, in particular when m and n are roughly equal. They always yield an upper bound for both the number of comparisons and moves.

2.3 Merge Trees

Let $L_1, \ldots, L_r$ with $\sum L_i = n$ be the lengths of the runs in the input. Any natural mergesort can be described as a rule to select some of the remaining runs, which are then merged and replaced by the merge result. If we always merge *two* runs this corresponds to a binary tree with the original runs at leaves $\boxed{1}, \ldots, \boxed{r}$. Internal nodes correspond to the result of merging their children. If we assign to internal node $\textcircled{\scriptsize j}$ the size M_j of the (intermediate) merge result it represents, then the overall merge cost is exactly $M = \sum_{\textcircled{\tiny j}} M_j$ (summing over all internal nodes). Figure 1 shows two examples of merge trees; the merge costs are given by adding up all gray areas,[5] (ignoring the dotted leaves).

Let d_i be the *depth* of leaf $\boxed{i}$ (corresponding to the run of length L_i), where depth is the number of edges on the path to the root. Every element in the ith run is counted exactly d_i times in $\sum_{\textcircled{\tiny j}} M_j$, so we have $M = \sum_{i=1}^{r} d_i \cdot L_i$. Dividing by n yields $M/n = \sum_{i=1}^{r} d_i \cdot \alpha_i$ for $\alpha_i = L_i/n$, which happens to be the expected search time C in the merge tree if $\boxed{i}$ is requested with probability α_i for $i = 1, \ldots, r$. So the minimal-cost merge tree for given run lengths $L_1, \ldots, L_r$ is the optimal alphabetic tree for leaf probabilities $\frac{L_1}{n}, \ldots, \frac{L_r}{n}$ and it holds $M \ge \mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n})n$. For distinct keys, the lower bound on comparisons (Section 1.2) coincides up to linear terms with this lower bound.

Combining this fact with the linear-time methods for nearly-optimal binary search trees from Section 2.1 immediately gives a stable sorting method that adapts optimally to existing

[5] The left tree is obviously better here and this is a typical outcome. But there are also inputs where Method 2 yields a better tree than Method 1.

runs up to an $O(n)$ term. We call such a method a *nearly-optimal (natural) mergesort*. A direct implementation of this idea needs $\Theta(r)$ space to store $L_1, \ldots, L_r$ and the merge tree and does an extraneous pass over the data just to determine the run lengths. The purpose of this paper is to show that we can make the overhead for finding a nearly-optimal merging order negligible in time and space.

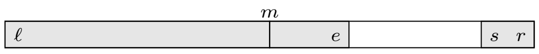
3 Nearly-Optimal Merging Orders

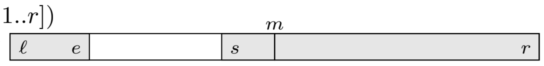
We now describe two sorting methods that simulate nearly-optimal search tree algorithms to compute nearly-optimal merging orders, but do so without ever storing the full merge tree or even the run lengths.

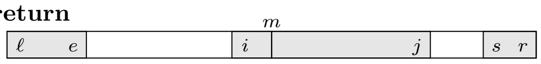
3.1 Peeksort: A Simple Top-Down Method

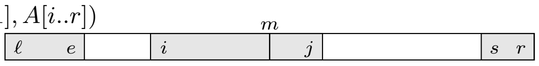
The first method is similar to standard top-down mergesort in that it implicitly constructs a merge tree on the call stack. Instead of blindly cutting the input in half, however, we mimic Mehlhorn's Method 1. For that we need the *run boundary closest to the middle* of the input: this will become the root of the merge tree. Since we want to detect existing runs anyway at some point, we start by finding the run that contains the middle position. The end point closer to the middle determines the top-level cut and we recursively sort the parts left and right of it. A final merge completes the sort.

To avoid redundant scans, we pass on the information about already detected runs. In the general case, we are sorting a range $A[\ell..r]$ whose prefix $A[\ell..e]$ and suffix $A[s..r]$ are runs. Depending on whether the middle is contained in one of those runs, we have one of four different cases; apart from that the overall procedure (Algorithm 1) is quite straight-forward.

```
PeekSort($A[\ell..r], e, s$)
if $e == r$ or $s == \ell$ then return
$m := \ell + \lfloor \frac{r-\ell}{2} \rfloor$
if $m \le e$                //
```


```
    PeekSort($A[e+1..r], e+1, s$)
    Merge($A[\ell..e], A[e+1..r]$)
else if $m \ge s$           //
```


```
    PeekSort($A[\ell..s-1], e, s-1$)
    Merge($A[\ell..s-1], A[s..r]$)
else
    // Find existing run $A[i..j]$ containing position $m$
    $i :=$ ExtendRunLeft($A[m], \ell$);    $j :=$ ExtendRunRight($A[m], r$)
    if $i == \ell$ and $j == r$ return
    if $m - i < j - m$      //
```


```
        PeekSort($A[\ell..i-1], e, i-1$)
        PeekSort($A[i..r], j, s$)
        Merge($A[\ell..i-1], A[i..r]$)
    else                    //
```


```
        PeekSort($A[\ell..j], e, i$)
        PeekSort($A[j+1..r], j+1, s$)
        Merge($A[\ell..j], A[j+1..r]$)
```

Algorithm 1 Peeksort: A simple top-down version of nearly-optimal natural mergesort. The initial call is PeekSort($A[1..n], 1, n$). Procedures ExtendRunLeft (-Right) scan left (right) starting at $A[m]$ as long as the run continues (and we did not cross the second parameter).

The following theorem shows that PeekSort is indeed a nearly-optimal mergesort. Unlike previous such methods, its code has very little overhead (in terms of both time and space) in comparison with a standard top-down mergesort, so it is a promising method for a practical nearly-optimal mergesort.

▶ **Theorem 2.** *The merge cost of* PeekSort *on an input consisting of r runs with respective lengths* $L_1, \ldots, L_r$ *is at most* $\mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n})n + 2n - (r+2)$*, the number of comparisons is at most* $\mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n}) + 3n - (2r+3)$*. Both is optimal up to* $O(n)$ *terms (in the worst case).*

Proof. The recursive calls of Algorithm 1 produce the same tree as Mehlhorn's Method 1 with input $(\alpha_0, \ldots, \alpha_m) = (\frac{L_1}{n}, \ldots, \frac{L_r}{n})$ (i.e., $m = r - 1$) and $\beta_j = 0$. By Theorem 1–(ii), the search costs in this tree are $C \le \mathcal{H} + 2 - (m+3)\alpha_{\min}$ with $\mathcal{H} = \mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n})$. Since $L_j \ge 1$, we have $\alpha_{\min} \ge \frac{1}{n}$. As argued in Section 2.3, the overall merge costs are then given by $M = Cn \le \mathcal{H}n + 2n - (r+2)$, which is within $O(n)$ of the lower bound for M.

We save at least one comparison per merge since merging runs of lengths m and n requires at most $n + m - 1$ comparisons. In total, we do exactly $r - 1$ merge operations. Apart from merging, we need a total of $n - 1$ additional comparisons to detect the existing runs in the input. Barbay and Navarro [3, Thm. 2] argued that $\mathcal{H}n - O(n)$ comparisons are necessary if the elements in the input are all distinct. ◀

3.2 Powersort: A Single-Pass Stack-Based Method

One little blemish remains in PeekSort: we have to use "random accesses" into the middle of the array to decide how to proceed. Even though we only use cache-friendly sequential scans, the I/O operations to load the middle run are effectively wasted, since it will often be merged only much later (after further recursive calls). Timsort and the other stack-based variants from [1, 7] proceed in one left-to-right scan over the input and merge the top runs on their stack. This increases the likelihood to still have (parts of) the most recently detected run in cache when it is merged subsequently.

3.2.1 The power of top-down in a bottom-up method

Method 2 to construct nearly-optimal search trees suggests the following definition:

▶ **Definition 3** (Node Power). Let $\alpha_0, \ldots, \alpha_m$, $\sum \alpha_j = 1$ be leaf probabilities. For $1 \le j \le m$, let $\textcircled{j}$ be the internal node separating the $(j-1)$st and jth leaf. The *power* of (the split at) node $\textcircled{j}$ is

$$P_j \;=\; \min\left\{ \ell \in \mathbb{N} : \left\lfloor \frac{a}{2^{-\ell}} \right\rfloor < \left\lfloor \frac{b}{2^{-\ell}} \right\rfloor \right\}, \text{ where } a \;=\; \sum_{i=0}^{j-1} \alpha_i - \tfrac{1}{2}\alpha_{j-1},\; b \;=\; \sum_{i=0}^{j-1} \alpha_i + \tfrac{1}{2}\alpha_j.$$

(P_j is the index of the first bit where the (binary) fractional parts of a and b differ.)

Intuitively, P_j is the "intended" depth of $\textcircled{j}$, but nodes occasionally end up higher in the tree if some leaf has a large weight relative to the current subtree, (see the rightmost branch in Figure 1). Mehlhorn's [17, 19] original implementation of Method 2, procedure *construct-tree*, does not single out the case that the next desired cut point lies *outside* the range of a subtree. This reduces the number of cases, but for our application, it is more convenient to explicitly check for this out-of-range case, and if it occurs to directly proceed to the next cut point. We refer to the modified algorithm as *Method 2′*; Appendix A gives the details and shows that the changes do not affect the guarantee for Method 2 in Theorem 1. In fact Method 2′ seems to typically yield slightly *better* trees than Method 2, but there are also counterexamples.

The core benefit of Method 2′, however, is that the resulting tree is characterized by local information, namely the node powers, without requiring any coordination of a top-down recursive procedure.

▶ **Lemma 4** (Path power monotonicity). *Consider the tree constructed by Method 2′. The powers of internal nodes along any root-to-leaf path is strictly increasing.*

The proof is given in Appendix A.

▶ **Corollary 5.** *The tree constructed by Method 2′ for leaf probabilities $\alpha_0, \ldots, \alpha_m$ is the (min-oriented) Cartesian tree for the sequence of node powers $P_1, \ldots, P_m$. It can thus be constructed iteratively (left to right) by the algorithm of Gabow, Bentley, and Tarjan [11].*

3.2.2 Merging on-the-fly

We implicitly use the observation from Corollary 5 in our algorithm "powersort" to construct the tree from left to right. Whenever the next internal node has a smaller power than the preceding one, we are following an edge from a left child up to its parent. That means that this subtree does not change anymore and we can execute any pending merges in it before continuing. If we are instead following an edge down to a right child of a node, that subtree is still "open" and we push the corresponding run on the stack. Algorithm 2 shows the detailed code.

```
PowerSort($A[1..n]$)
 1  $X$ := stack of runs              (capacity $\lfloor \lg(n) \rfloor + 1$)
 2  $P$ := stack of powers            (capacity $\lfloor \lg(n) \rfloor + 1$)
 3  $s_1 := 1$;   $e_1 = $ ExtendRunRight($A[1], n$)     // $A[s_1..e_1]$ is leftmost run
 4  while $e_1 < n$
 5      $s_2 := e_1 + 1$;   $e_2 :=$ ExtendRunRight($A[s_2], n$)     // $A[s_2..e_2]$ next run
 6      $p :=$ NodePower($s_1, e_1, s_2, e_2, n$)     // $P_j$ for node $\textcircled{j}$ between $A[s_1..e_1]$ and $A[s_2..e_2]$
 7      while $P.top() > p$       // previous merge deeper in tree than current
 8          $P.pop()$             // $\rightsquigarrow$ merge and replace run $A[s_1..e_1]$ by result
 9          $(s_1, e_1) :=$ Merge($X.pop()$, $A[s_1..e_1]$)
10      $X.push(A[s_1, e_1])$;   $P.push(p)$
11      $s_1 := s_2$;   $e_1 := e_2$
12  end while // Now $A[s_1..e_1]$ is the rightmost run
13  while $\neg X.empty()$
14      $(s_1, e_1) :=$ Merge($X.pop()$, $A[s_1..e_1]$)
```

```
NodePower($s_1, e_1, s_2, e_2, n$)
1  $n_1 := e_1 - s_1 + 1$;   $n_2 := e_2 - s_2 + 1$;   $\ell := 0$
2  $a := (s_1 + n_1/2 - 1)/n$;   $b := (s_2 + n_2/2 - 1)/n$
3  while $\lfloor a \cdot 2^\ell \rfloor == \lfloor b \cdot 2^\ell \rfloor$ do $\ell := \ell + 1$ end while
4  return $(\ell)$
```

Algorithm 2 Powersort: A one-pass stack-based nearly-optimal natural mergesort. Procedure ExtendRunRight scans right as long as the run continues.

The runs on the stack correspond to nodes with strictly increasing powers, so we can bound the stack height by the maximal P_j. Since our leaf probabilities here are $\alpha_j = \frac{L_j}{n} \geq \frac{1}{n}$, we have $P_j \leq \lfloor \lg n \rfloor + 1$.

▶ **Theorem 6.** *The merge cost of* PowerSort *is at most $\mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n}) + 2n$, the number of comparisons is at most $\mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n}) + 3n - r$. Moreover, (apart from buffers for merging) only $O(\log n)$ words of extra space are required.*

Proof. The merge tree of PowerSort is exactly the search tree constructed by Method 2′ on leaf probabilities $(\alpha_0, \ldots, \alpha_m) = (\frac{L_1}{n}, \ldots, \frac{L_r}{n})$ and $\beta_j = 0$. By Theorem 1–(iii), the search costs are $C \le \mathcal{H} + 2$ with $\mathcal{H} = \mathcal{H}(\frac{L_1}{n}, \ldots, \frac{L_m}{n})$, so the overall merge costs are $M = Cn \le \mathcal{H}n + 2n$, which is within $O(n)$ of the lower bound for M. The comparisons follow as for Theorem 2. ◀

4 Running-Time Study

We conducted a running-time study comparing the two new nearly-optimal mergesorts with current state-of-the-art implementations and elementary mergesort variants. The code is available on github [31].

The goal of this study is to show that

1. peeksort and powersort have very little overhead compared to standard (non-natural) mergesort variants (i.e., they are never (much) slower), and at the same time
2. peeksort and powersort outperform other mergesort variants on partially presorted inputs.

Timsort is arguably the most used adaptive sorting method; even though analytical guarantees are still to be found, its efficiency in particular for partially sorted inputs has been demonstrated empirically [23]. A secondary goal is hence to

3. investigate the typical merge costs of Timsort on different inputs.

4.1 Setup

Oracle's Java runtime library includes a highly tuned Timsort implementation; to be able to directly compare with it, we chose to implement our algorithms in Java. The Timsort implementation is used for `Object[]`, i.e., arrays of *references* to objects; since the location of objects on the heap is hard to control and likely to produce more or less cache misses from run to run, we chose to sort `int[]`s instead to obtain more reproducible results. We thus modified the library implementation of Timsort accordingly. This scenario makes key comparisons and element moves relatively cheap and thereby emphasizes the remaining overhead of the methods, which is in line with our primary goal 1) to study the impact of the additional bookkeeping required by the adaptive methods.

We compare our methods with simple top-down and bottom-up mergesort implementations. We use the code given by Sedgewick [25, Programs 8.3 and 8.5] with his simple merge method (Program 8.2) as its basis; in both cases, we add a check before calling merge to detect if the runs happen to be already in sorted order, and we use insertionsort for base cases of size $n \le w = 24$. (For bottom-up mergesort, we start by sorting chunks of size $w = 24$.) Our Java implementations of peeksort and powersort are described in more detail in Appendix B. Apart from a mildly optimized version of the pseudocode, we added the same cutoff / minimal run length ($w = 24$) as above.

All our methods call the same merge procedure, whereas the library Timsort contains a modified merge method that tries to save key comparisons: when only elements from the same run are selected for the output repeatedly, Timsort enters a *"galloping mode"* and uses exponential searches (instead of the conventional sequential search) to find the insertion position of the next element. Details are described by Peters [23]. Since saving comparisons is not of utmost importance in our scenario of sorting `int`s, we also added a version of Timsort, called "trotsort", that uses our plain merge method instead of galloping, but is otherwise identical to the library Timsort.

Our hard- and software setup is listed in Appendix C. We use the following inputs types:

- *random permutations* are a case where no long runs are present to exploit;

- *"random-runs" inputs* are constructed from a random permutation by sorting segments of random lengths, where the lengths are chosen independently according to a geometric distribution with a given mean ℓ; since the geometric distribution has large variance, these inputs tend to have runs whose sizes vary a lot;
- *"Timsort-drag" inputs* are special instances of random-runs inputs where the run lengths are chosen as $\mathcal{R}_{\mathrm{tim}}$, the bad-case example for Timsort from [7, Thm. 3].

4.2 Overhead of Nearly-Optimal Merge Order

We first consider random permutations as inputs. Since random permutations contain (with high probability) no long runs that can be exploited, the adaptive methods will not find anything that would compensate for their additional efforts to identify runs. (This is confirmed by the fact that the total merge costs of all methods, including Timsort, are within 1.5% of each other in this scenario.) Figure 2 shows average running times for inputs sizes from 100 000 to 100 million ints. (Measurements for $n = 10\,000$ were too noisy to draw meaningful conclusions.)

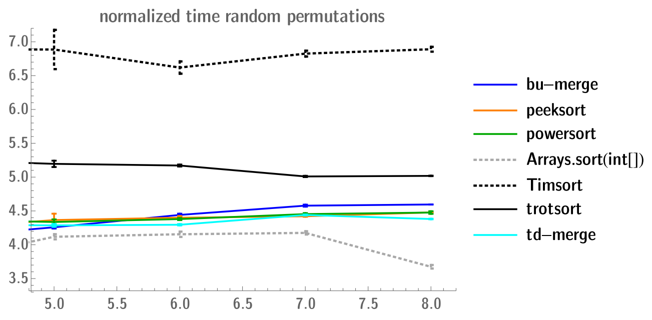


Figure 2 Normalized running times on random permutations. The x-axis shows $\log_{10}(n)$, the y-axis show average and standard deviation (as error bars) of $t/(n \lg n)$ where t is the running time in ms. We plot averages over 1000 repetitions (resp. 200 and 20 for the largest sizes).

The relative ranking is clearly stable across different input sizes. `Arrays.sort(int[])` (dual-pivot quicksort) is the fastest method, but is not a stable sort and only serves as a baseline. The timings of top-down and bottom-up mergesort, peeksort and powersort are 20–30% slower than dual-pivot quicksort. Comparing the four to each other, no large differences are visible; if anything, bottom-up mergesort was a bit slower for large inputs. Since the sorting cutoff / minimal run length $w = 24$ exceeded the length of *all* runs in all inputs, we are effectively presented with a case of all-equal run lengths. Merging them blindly from left to right (as in bottom-up mergesort) then performs just fine, and top-down mergesort finds a close-to-optimal merging order in this case. That peek- and powersort perform essentially *as good as* elementary mergesorts on random permutations thus clearly indicates that their overhead for determining a nearly-optimal merging order is negligible.

The library Timsort performs surprisingly bad on `int[]`s, probably due to the relatively cheap comparisons. Replacing the galloping merge with the ordinary merge alleviates this (see "trotsort"), but Timsort remains inferior on random permutations by a fair margin (10–20%).

4.3 Practical speedups by adaptivity

After demonstrating that we do not lose much by using our adaptive methods when there is nothing to adapt to, we next investigate how much can be gained if there is. We consider random-runs inputs as described above. This input model instills a good dose of presortedness, but not in a way that gives any of the tested methods an obvious advantage or disadvantage

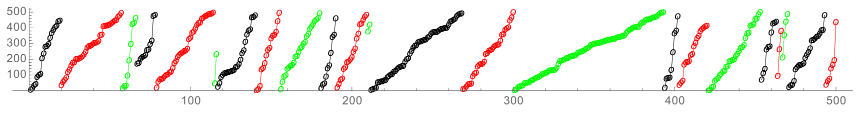


Figure 3 A random-runs input with $n = 500$ and $\ell = 20 \approx \sqrt{n}$. The x-axis is the index in the array, the y-axis the value. Runs are emphasized by color.

over the others. We choose a representative size of $n = 10^7$ and an expected run length $\ell = 3\,000$, so that we expect roughly $\sqrt{n}$ runs of length $\sqrt{n}$.

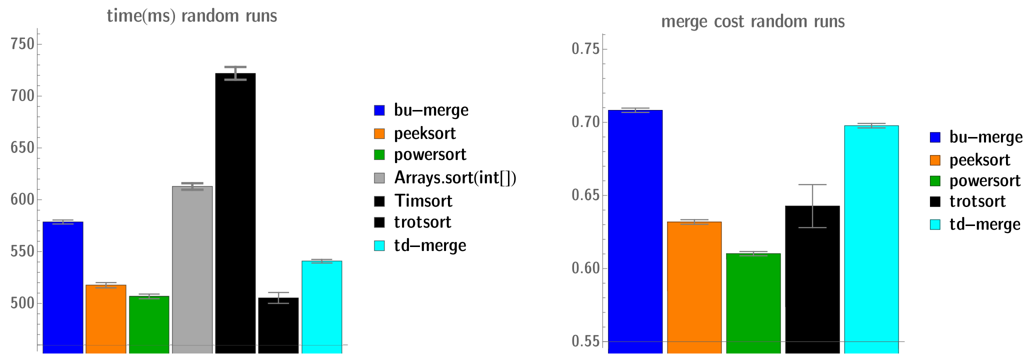


Figure 4 Average running times (left) and normalized merge cost (right) on random-runs inputs with $n = 10^7$ and $\ell = 3\,000 \approx \sqrt{n}$. The merge costs have been divided by $n \lg(n/w) \approx 1.87 \cdot 10^8$, which is the merge cost a (hypothetical) optimal mergesort that does not pick up existing runs, but starts merging with runs of length $w = 24$.

If this was a random permutation, we would expect merge costs of roughly $n \lg(n/w) \approx 1.87 \cdot 10^8$ (indeed a bit above this). The right chart in Figure 4 shows that the adaptive methods can bring the merge cost down to a little over 60% of this number. Note the run lengths vary considerably – to give some intuitive feel for this volatility, Figure 3 shows a stereotypical (but smaller) random-runs input. Powersort achieved average merge costs of $1.14 \cdot 10^8 < n \lg r \approx 1.17 \cdot 10^8$, i.e., less than a method would that only adapts to the *number* of runs r.

In terms of running time, powersort is again among the fastest stable methods, and indeed 20% *faster* than `Arrays.sort(int[])`. The best adaptive methods are also 10% faster than top-down mergesort. (The latter is "slightly adaptive", by omitting merges if the runs happen to already be in order.) This supports the statement that significant speedups can be realized by adaptive sorting on inputs with existing order, and $\sqrt{n}$ runs suffice for that. If we increase ℓ to $100\,000$, so that we expect only roughly 100 long runs, the library quicksort becomes twice as slow as powersort and Timsort (trotsort).

Timsort's running time is a bit anomalous again. Even though it occasionally incurs 10% more merge costs on a given input than powersort, the running times were within 1% of each other (considering the trotsort variant; the original galloping version was again uncompetitive).

4.4 Non-optimality of Timsort

Finally, we consider "Timsort-drag" inputs, a sequence of run lengths $\mathcal{R}_{\text{tim}}(n)$ specifically crafted by Buss and Knop [7] to generate unbalanced merges (and hence large merge cost) in Timsort. Since actual Timsort implementations employ minimal run lengths of up to 32 elements we multiplied the run lengths by 32. Figure 5 shows running time and merge cost for all methods on a characteristic Timsort-drag input of length $2^{24} \approx 1.6 \cdot 10^7$.

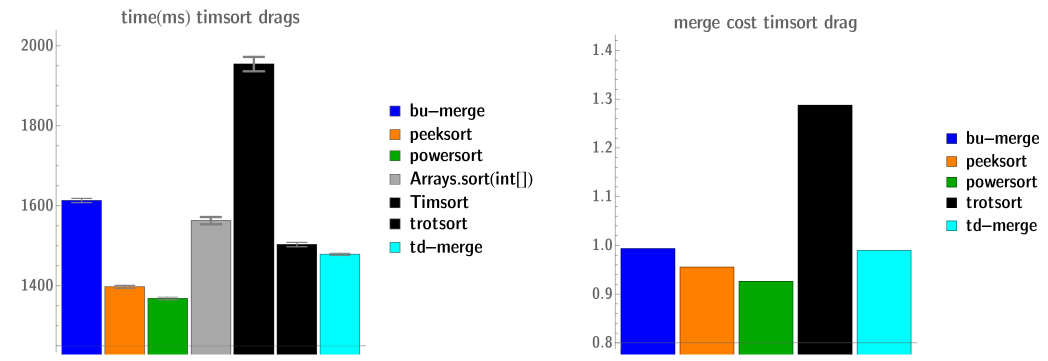


Figure 5 Average running times (left) and normalized merge cost (right) on "Timsort-drag" inputs with $n = 2^{24}$ and run lengths $\mathcal{R}_{\text{tim}}(2^{24}/32)$ multiplied by 32. Merge costs have been divided by $n\lg(n/w) \approx 3.26 \cdot 10^8$.

In terms of merge costs, Timsort now pays 30% more than even a simple non-adaptive mergesort, whereas peeksort and powersort obviously retain their proven nearly-optimal behavior. Also in terms of running time, Timsort is a bit slower than top-down mergesort, and 10% slower than powersort on these inputs. It is remarkable that the dramatically larger merge cost does not lead to a similarly drastic slow down in practice. Nevertheless, it must be noted that Timsort's merging strategy has weaknesses, and it is unclear if more dramatic examples are yet to be found.

5 Conclusion

In this paper, we have demonstrated that provably good merging orders for natural mergesort can be found with negligible overhead. The proposed algorithms, peeksort and powersort offer more reliable performance than the widely used Timsort, and at the same time, are arguably simpler.

Powersort builds on a modified bisection heuristic for computing nearly-optimal binary search trees that might be independent interest. It has the same quality guarantees as Mehlhorn's original formulation, but allows the tree to be built "bottom-up" as a Cartesian tree over a certain sequence, the "node powers". It is the only such method for nearly-optimal search trees to our knowledge.

Buss and Knop conclude with the question whether there exists a k-aware algorithm (a stack-based natural mergesort that only considers the top k runs in the stack) with merge cost $(1 + o_r(1))$ times the optimal merge cost [7, Question 37]. Powersort effectively answers this question in the affirmative with $k = 3$.[6]

5.1 Extensions and future work

Timsort's "galloping merge" procedure saves comparisons when we consistently consume elements from one run, but in "well-mixed" merges, it does not help (much). It would be

[6] Strictly speaking, powersort needs a relaxation of the model of Buss and Knop. They require decisions to be made solely based on the *lengths* of runs, whereas node power takes the location of the runs within the array into account. Since the location of a run must be stored anyway, this appears reasonable to us.

interesting to compare this method with other comparison-efficient merge methods.

Another line of future research is to explore ways to profit from duplicate keys in the input. The ultimate goal would be a “synergistic” sorting method (in the terminology of [4]) that has practically no overhead for detecting existing runs and equals and yet exploits their *combined* presence optimally.

A Method 2′ for nearly-optimal search trees

This section uses Mehlhorn's [17, 19] original notation, in particular, it handles the general optimal binary search tree setting. We add the following two "out-of-range" cases to Mehlhorn's procedure *construct-tree*(i, j, cut, ℓ), at the very beginning:

- (Case L) If $s_j < cut + 2^{-\ell}$, return *construct-tree*$(i, j, cut, \ell + 1)$
- (Case R) If $cut + 2^{-\ell} < s_i$, return *construct-tree*$(i, j, cut + 2^{-\ell}, \ell + 1)$

The original procedure would end up in case C instead of L resp. case D instead of R and choose a right- resp. leftmost key as the root node. But if the desired cut point $cut + 2^{-\ell}$ lies completely outside the range of bisection $[s_i, s_j]$, this produces an unnecessarily unbalanced split. This case can only happen if the neighboring leaf has a probability larger than $\frac{1}{2}$ relative to the previous subtree, so that the current split point still lies within the range corresponding to the previously chosen root. Our cases L and R thus "skip" this void cut point $cut + 2^{-\ell}$ and increment ℓ without generating a node.

Given that the invariants (1)–(4) are fulfilled for the current parameters of *construct-tree*, they will also be fulfilled in the recursive calls issued in cases L and R. Therefore, Mehlhorn's analysis remains valid for our modified procedure: In his Fact 3, we have $b_h + 1 \le \ell$ and $a_h \le \ell$ (instead of equality), but this is all that is needed to establish Fact 4 and hence the bound on the search costs.

A.1 Proof of Lemma 4

Claim: In the tree constructed by Method 2′, the powers of internal nodes along any root-to-leaf path are strictly increasing.

Proof. Consider the recursive procedure *construct-tree* as described by Mehlhorn [17, 19], but with the additional cases from above. We prove that whenever a recursive call *construct-tree*(i, j, cut, ℓ) creates an internal node $\textcircled{k}$, we have $P_k = \ell$. Since ℓ is incremented in all recursive calls, the claim follows.

Only cases A, B, C or D create new nodes, so assume we are not in case L or R. Then we actually have the stronger version of invariant (4): $cut \le s_i \le cut + 2^{-\ell} \le s_j \le cut + 2^{-\ell+1}$ and hence will always find a k with $i < k \le j$ and

$$cut \;\le\; s_{k-1} \;<\; cut + 2^{-\ell} \;\le\; s_k \;\le\; cut + 2^{-(\ell-1)}$$

for which we now create the node $\textcircled{k}$. Dividing by $2^{-\ell}$ shows that $P_k = \ell$ (we have $a = s_{k-1}$ and $b = s_k$). ◀

B Java Code

In this appendix, we give the core methods of a Java implementation of Algorithm 1 and Algorithm 2 that form the basis of our running-time study. The goal is to document (and comment on) a few design decisions and optimizations that have been included in the running time study. The full code, including the main routines to reproduce the running time studies exactly as used for this paper, is available on github [31].

B.1 Merging

All our mergesort variants (except for the original library implementation of Timsort) use the following straight-forward “bitonic” merge procedure that merges two adjacent runs. It is taken from [25, Program 8.2]

```java
/** merges A[l..m-1] and A[m..r] */
static void mergeRuns(int[] A, int l, int m, int r, int[] aux) {
   --m; // accounts for different convention in Sedgewick's book
   int i, j;
   for (i = m+1; i > l; --i) aux[i-1] = A[i-1];
   for (j = m; j < r; ++j) aux[r+m-j] = A[j+1];
   for (int k = l; k <= r; ++k)
      A[k] = aux[j] < aux[i] ? aux[j--] : aux[i++];
}
```

Merging offers some potential for improvements in particular w.r.t. the number of used key comparisons. Since we operate on integers, comparisons are cheap and more sophisticated merging strategies will not be needed here.

B.2 Peeksort

The pseudocode for Peeksort can be translated to Java almost verbatim. Since the recursion is costly for small input sizes, we switch to insertionsort when the subproblem size is small. Values for `INSERTION_SORT_THRESHOLD` of 10–32 yielded good results in the experiments; we ultimately set it to 24 to approximate the choice in the library implementation of Timsort. (The latter chooses the minimal run length between 16 and 32 depending on n by a heuristic that tries to avoid imbalanced merges. We did not use this adaptive choice in our methods.)

```java
static void peeksort(int[] A, int left, int right,
                     int leftRunEnd, int rightRunStart, int[] B) {
   if (leftRunEnd == right || rightRunStart == left) return;
   if (right - left + 1 <= INSERTION_SORT_THRESHOLD) {
      insertionsort(A, left, right, leftRunEnd - left + 1); return;
   }
   int mid = left + ((right - left) >> 1);
   if (mid <= leftRunEnd) {            // |XXXXXXXX|XX     X|
      peeksort(A, leftRunEnd+1, right, leftRunEnd+1,rightRunStart, B);
      mergeRuns(A, left, leftRunEnd+1, right, B);
   } else if (mid >= rightRunStart) {  // |XX     X|XXXXXXXX|
      peeksort(A, left, rightRunStart-1, leftRunEnd, rightRunStart-1, B);
      mergeRuns(A, left, rightRunStart, right, B);
   } else { // find middle run
      int i, j;
      if (A[mid] <= A[mid+1]) {
         i = extendIncreasingRunLeft(A, mid, leftRunEnd + 1);
         j = mid+1 == rightRunStart ? mid :
            extendIncreasingRunRight(A, mid+1, rightRunStart - 1);
      } else {
         i = extendDecreasingRunLeft(A, mid, leftRunEnd + 1);
         j = mid+1 == rightRunStart ? mid :
            extendStrictlyDecreasingRunRight(A, mid+1,rightRunStart - 1);
         reverseRange(A, i, j);
      }
      if (i == left && j == right) return;
      if (mid - i < j - mid) {         // |XX     x|xxxx   X|
         peeksort(A, left, i-1, leftRunEnd, i-1, B);
         peeksort(A, i, right, j, rightRunStart, B);
         mergeRuns(A,left, i, right, B);
      } else {                         // |XX   xxx|x      X|
         peeksort(A, left, j, leftRunEnd, i, B);
         peeksort(A, j+1, right, j+1, rightRunStart, B);
```

```
            mergeRuns(A,left, j+1, right, B);
        }
    }
}
```

B.3 Insertionsort to extend runs

Here (and in the following code), we use a straight insertionsort variant that accepts the length of a sorted prefix as an additional parameter. A similar method is also used in Timsort to extend runs to a forced minimal length. The library Timsort uses binary insertionsort instead, but unless comparisons are expensive, a straight sequential-search variant is sufficient.

```
static void insertionsort(int[] A, int left, int right, int nPresorted) {
    assert right >= left;
    assert right - left + 1 >= nPresorted;
    for (int i = left + nPresorted; i <= right ; ++i) {
        int j = i - 1,  v = A[i];
        while (v < A[j]) {
            A[j+1] = A[j];
            --j;
            if (j < left) break;
        }
        A[j+1] = v;
    }
}
```

B.4 Powersort

For powersort, we implement the stack as an array that is indexed by the node power. Thereby, we avoid explicit stack operations and to store powers explicitly. On the other hand, we have to check for empty entries since powers are not always consecutive.

```
static void powersort(int[] A, int left, int right) {
    int n = right - left + 1;
    int lgnPlus2 = log2(n) + 2;
    int[] leftRunStart = new int[lgnPlus2], leftRunEnd = new int[lgnPlus2];
    Arrays.fill(leftRunStart, -1);
    int top = 0;
    int[] buffer = new int[n >> 1];

    int startA = left, endA = extendRunRight(A, startA, right);
    int lenA = endA - startA + 1;
    if (lenA < minRunLen) { // extend to minRunLen
        endA = Math.min(right, startA + minRunLen-1);
        insertionsort(A, startA, endA, lenA);
    }
    while (endA < right) {
        int startB = endA + 1, endB = extendRunRight(A, startB, right);
        int lenB = endB - startB + 1;
        if (lenB < minRunLen) { // extend to minRunLen
            endB = Math.min(right, startB + minRunLen-1);
            insertionsort(A, startB, endB, lenB);
        }
        int k = nodePower(left, right, startA, startB, endB);
        assert k != top;
        for (int l = top; l > k; --l) { // clear left subtree bottom-up
            if (leftRunStart[l] == -1) continue;
            mergeRuns(A, leftRunStart[l], leftRunEnd[l]+1, endA, buffer);
            startA = leftRunStart[l];
            leftRunStart[l] = -1;
        }
```

```
      // store left half of merge between A and B on stack
      leftRunStart[k] = startA; leftRunEnd[k] = endA;
      top = k;
      startA = startB; endA = endB;
   }
   assert endA == right;
   for (int l = top; l > 0; --l) {
      if (leftRunStart[l] == -1) continue;
      mergeRuns(A, leftRunStart[l], leftRunEnd[l]+1, right, buffer);
   }
}
```

The computation of the node powers can be done in many different ways and offers a lot of potential for low-level bitwise optimizations; some care is needed to prevent overflows. In our experiments, the following loop-less version was a tiny bit faster than other tried alternatives.

```
static int nodePower(int left, int right, int startA, int startB, int endB) {
   int twoN = (right - left + 1) << 1; // 2*n
   long l = startA + startB - (left << 1);
   long r = startB + endB + 1 - (left << 1);
   int a = (int) ((l << 31) / twoN);
   int b = (int) ((r << 31) / twoN);
   return Integer.numberOfLeadingZeros(a ^ b);
}
```

C Experimental Setup

All experiments were run on a Lenovo Thinkpad X230 Tablet running Ubuntu 16.04.01 with Linux kernel 4.13.0-38-generic. The CPU is an Intel Core i7-3520M CPU with 2.90GHz, the system has 8GB of main memory.

The Java compiler was from the Oracle Java JDK version 1.8.0_161, the JVM is Java HotSpot 64-Bit Server VM (build 25.161-b12, mixed mode). All experiments were run with disabled X server from the TTY, the java process was bound to one core (with the `taskset` utility). They started with a warmup phase to trigger just-in-time (JIT) compilation before measuring individual sorting operations. The inputs were generated outside the timing window and reused the same array for all repetitions. The following flags were used for the JVM: `-XX:+UnlockDiagnosticVMOptions`, `-XX:-TieredCompilation` and `-XX:+PrintCompilation`. Tiered compilation was disabled to avoid multiple passes of just-in-time compilation to occur during the timed experiments; the print-compilation flag was used to monitor whether relevant methods are subjected to recompilation or deoptimization during the experiments.

References

1 Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: from merge sort to TimSort. December 2015. URL: `https://hal-upec-upem.archives-ouvertes.fr/hal-01212839`.

2 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. Lrm-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theoretical Computer Science*, 459:26–41, 2012. `doi:10.1016/j.tcs.2012.08.010`.

3 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, November 2013. `doi:10.1016/j.tcs.2013.10.019`.

4 Jérémy Barbay, Carlos Ochoa, and Srinivasa Rao Satti. Synergistic solutions on multisets. *DROPS-IDN/7344*, 78, 2017. URL: `http://drops.dagstuhl.de/opus/volltexte/2017/7344/`, `doi:10.4230/LIPICS.CPM.2017.31`.

5 Paul J. Bayer. *Improved Bounds on the Cost of Optimal and Balanced Binary Search Trees*. Master's thesis, Massachusetts Institute of Technology, 1975.

6 Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993. `doi:10.1002/spe.4380231105`.

7 Sam Buss and Alexander Knop. Strategies for stable merge sorting. January 2018. URL: `http://arxiv.org/abs/1801.04641`, `arXiv:1801.04641`.

8 Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying OpenJDK's sort method for generic collections. *Journal of Automated Reasoning*, August 2017. `doi:10.1007/s10817-017-9426-4`.

9 Amr Elmasry and Abdelrahman Hammad. Inversion-sensitive sorting algorithms in practice. *Journal of Experimental Algorithmics*, 13:11:1–11:18, 2009. `doi:10.1145/1412228.1455267`.

10 Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, December 1992. `doi:10.1145/146370.146381`.

11 Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC '84*, pages 135–143, New York, New York, USA, 1984. ACM Press. `doi:10.1145/800057.808675`.

12 Mordecai J. Golin and Robert Sedgewick. Queue-mergesort. *Information Processing Letters*, 48(5):253–259, dec 1993. `doi:10.1016/0020-0190(93)90088-q`.

13 Chris Hegarty. Replace "modified mergesort" in java.util.Arrays.sort with timsort, 2009. URL: `https://bugs.openjdk.java.net/browse/JDK-6804124`.

14 Yasuichi Horibe. An improved bound for weight-balanced tree. *Information and Control*, 34(2):148–151, June 1977. `doi:10.1016/S0019-9958(77)80011-9`.

15 Donald E. Knuth. *The Art Of Computer Programming: Searching and Sorting*. Addison Wesley, 2nd edition, 1998.

16 Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5(4), 1975. `doi:10.1007/BF00264563`.

17 Kurt Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6(2):235–239, June 1977. `doi:10.1137/0206017`.

18 Kurt Mehlhorn. Sorting presorted files. In *Theoretical Computer Science 4th GI Conference*, pages 199–212. 1979. `doi:10.1007/3-540-09118-1_22`.

19 Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer, 1984.

20 Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing*, 5(1):1–8, March 1976. `doi:10.1137/0205001`.

21 S.V. Nagaraj. Optimal binary search trees. *Theoretical Computer Science*, 188(1-2):1–44, November 1997. `doi:10.1016/S0304-3975(96)00320-9`.

22 Tim Peters. [Python-Dev] Sorting, 2002. URL: `https://mail.python.org/pipermail/python-dev/2002-July/026837.html`.

23 Tim Peters. Timsort, 2002. URL: `http://hg.python.org/cpython/file/tip/Objects/listsort.txt`.

24 Robert Sedgewick. Quicksort with equal keys. *SIAM Journal on Computing*, 6(2):240–267, 1977.

25 Robert Sedgewick. *Algorithms in Java*. Addison-Wesley, 2003.

26 Robert Sedgewick and Jon Bentley. Quicksort is optimal (talk slides), 2002. URL: `http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf`.

27 Tadao Takaoka. A new measure of disorder in sorting - entropy. In *Proceedings of Computing: The Fourth Australasian Theory Symposium (CATS'98), Perth, WA, Australia, February 2-3, 1998*, pages 77–86, 1998.

28 Tadao Takaoka. Partial solution and entropy. In *MFCS 2009*, pages 700–711, 2009. doi:10.1007/978-3-642-03816-7_59.

29 W.A. Walker and C.C. Gotlieb. A top-down algorithm for constructing nearly optimal lexicographic trees. In *Graph Theory and Computing*, pages 303–323. Elsevier, 1972. doi:10.1016/B978-1-4832-3187-7.50023-4.

30 Lutz M. Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, C-34(4):362–367, April 1985. doi:10.1109/TC.1985.5009387.

31 Sebastian Wild. Accompanying code for running time study, May 2018. URL: https://github.com/sebawild/nearly-optimal-mergesort-code/releases/tag/paper, doi:10.5281/zenodo.1241162.

32 Sebastian Wild. Quicksort is optimal for many equal keys. In *ANALCO 2018*, pages 8–22. SIAM, January 2018. doi:10.1137/1.9781611975062.2.