

Adaptive Sorting, also in the BST Model

Ivor van der Hoog 

IT University of Copenhagen, Denmark

Eva Rotenberg 

IT University of Copenhagen, Denmark

Daniel Rutschmann 

Institute of Science and Technology Austria (ISTA), Austria

Abstract

Sorting is one of the most basic algorithmic problems. In adaptive sorting, algorithms perform better on inputs that are easier to sort. Formally, one defines a measure of sortedness that captures features of the input, typically an entropy measure, and tries to design algorithms that adapt to this measure. Over the years, many such measures have been studied, raising two natural questions: How do these measures relate to each other, and are there algorithms that are adaptive to each measure?

We answer these questions by completing a hierarchy of relations between previously studied entropy measures. Our added relations show that this hierarchy has a source, Reg , which is a classical measure from the literature. We then *discover* and *prove* that an existing data structure already yields a sorting algorithm adaptive to Reg . Consequently, this algorithm already achieves the adaptive bounds of all other measures in the hierarchy.

We then make a brief mathematical detour. McIlroy previously noted that Reg has a mathematically inconvenient definition. We argue for studying a slightly weaker, but mathematically much more elegant measure: reg . We show that, if the input contains no duplicates, $O(n + \log \text{reg}) = O(n + \log \text{Reg})$. This makes reg elegant, easier to analyse, and almost as strong.

Finally, we turn to algorithm design. We introduce SPROUTSORT , a reg -adaptive algorithm in the offline binary search tree model (the *BST Model*). In this model, a single binary search tree stores the distinct input keys, which must be accessed in input order. The entire access sequence is known in advance, but each access starts at the root and incurs unit cost for pointer moves and rotations. SPROUTSORT achieves an access cost of $O(n + \log \text{reg})$, making it the most adaptive algorithm in the BST Model to date. Turning our attention back to adaptive sorting in the comparison model, we also present the adaptive sorting algorithm SKELETONSORT . This is a simple randomised pointer-machine algorithm that sorts distinct keys in expected $O(n + \log \text{reg})$ time. Although it does not work if the input has duplicates, SKELETONSORT offers a considerably simpler implementation of an algorithm this high in the adaptive hierarchy.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Design and analysis of algorithms; Theory of computation $\rightarrow$ Sorting and searching

Keywords and phrases Adaptive sorting, entropy, algorithms

Digital Object Identifier 10.4230/LIPIcs...

Funding Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann are grateful to the VILLUM Foundation for supporting this work via Eva Rotenberg’s grant (VIL37507) “Efficient Recomputations for Changeful Problems”. This work was supported by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

Usage of Large Language Models and Artificial Intelligence All lemmas, theorems, and results were obtained without the use of artificial intelligence. In the final writing stages, large language models were used for proofreading and language improvements.



© Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann;
licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

This paper studies comparison-based adaptive sorting of n elements and the related offline binary search tree (BST) access problem. Sorting n distinct elements requires $\Omega(n \log n)$ comparisons in the worst case. However, this lower bound does not take into account the fact that some inputs are easier to solve than others. If the input is already sorted, one can verify this in linear time and, intuitively, an *almost sorted input* can also be sorted faster. A large body of work studies *adaptive sorting algorithms* that exploit input features that reflect notions of almost-sortedness, expressed by entropy functions. Examples include inputs consisting of few sorted contiguous pieces [21, 24], few contiguous blocks [9], or few (possibly interleaved) increasing or monotone subsequences [23]. Whenever a measure of entropy is introduced, a natural question is how it relates to other measures: is one strictly stronger than the other, in the sense that matching one in running time, implies matching the other? In this paper, we extend the hierarchy of entropy measures by establishing the implications marked in orange in Figure 1. As a consequence, the hierarchy has a source Reg.

For each entropy measure in the hierarchy, one can ask whether there exist algorithms that are adaptive to that measure. We **discover** that the unified bound dictionary of Bădoiu, Cole, Demaine, and Iacono [3] already yields a Reg-adaptive algorithm. Thus, [3] yields the most adaptive sorting algorithms to date. We continue by modifying Reg to define a simpler measure reg with much nicer mathematical properties. We show that reg dominates almost all prior measures. Moreover, if the input contains no duplicates then we prove that any reg-adaptive algorithm is also Reg-adaptive. We give a reg-adaptive offline BST access algorithm. Finally, we present a simple reg-adaptive classical algorithm.

Terminology. To explain our results, consider the following abstractions of adaptive sorting bounds: let $\mathcal{I}_n$ denote the set of all size- n inputs. A *notion of presortedness* assigns to every input $X \in \mathcal{I}_n$ some target cost $f(X)$. A sorting algorithm is *comparison-adaptive* with respect to this notion if it sorts every input X in $O(f(X))$ comparisons, and *time-adaptive* if its total running time is $O(f(X))$. For offline BST algorithms, we consider the BST model where one can navigate and rotate nodes of a binary search tree, and we consider the total access cost of accessing all elements of X in-order. The target f is usually defined in two steps: first a number $k(X)$ is defined in a sophisticated way, and then $f(X)$ is a function of n and $k(X)$, e.g. $n(1 + \log k(X))$ or $n(1 + k(X))$. A concrete example is the number of runs [21]: By splitting X between every pair of adjacent items that form a decreasing sequence of length 2, we partition X into maximal contiguous non-decreasing sequences $X_1, X_2, \dots, X_{k=\text{nRuns}}$. Then, [21] puts $f(X) = n(1 + \log(\text{nRuns}))$ and presents a (comparison) adaptive algorithm.

Ranks and Duality. For a list X of elements and $x \in X$ we define the rank of x denoted by $\text{rank}_X(x)$ as its index in the sorted order, *after* removing duplicates. Consider for the sake of analysis, simultaneously replacing each element $X[i]$ by $\text{rank}_X(X[i])$. Then any comparison-based algorithm makes the exact same decisions for the replaced input. Thus, we may assume that X is a list of elements in $[n]$. If X contains no duplicates then X is a permutation. If X is a permutation, it has a unique inverse X^{-1} which we call its dual, and which leads to dual notions of presortedness. For example, dual runs: $\text{nRuns}^*(X) := \text{nRuns}(X^{-1})$, giving rise to $f(X) = n(1 + \log(\text{nRuns}^*(X)))$. In most cases, these dual definitions can be (manually) extended to inputs X with duplicates. E.g., A subsequence $X_i \subseteq X$ is *value-contiguous* if for all $x, y, z \in X$ with $x \leq y \leq z$ and $x, z \in X_i$, we also have $y \in X_i$. We can partition $X = X_1 \sqcup \dots \sqcup X_k$ into maximal non-decreasing value-contiguous sequences X_i and put

89 $\text{nRuns}^*(X) := k$; this agrees with the previous definition when X contains no duplicates. We
 90 *occasionally* define duality for lists with duplicates, by considering the position of the i 'th
 91 element in the sorted order after a stable sort, denoted $\overline{X}[i]$. Note that $\overline{X}$ is a permutation.

92 **Entropy functions.** The Shannon entropy function allows us to define strong notions of
 93 presortedness from a given partition of X . We illustrate three examples: For runs, consider
 94 the partition $X = X_1 \sqcup \dots \sqcup X_k$ into maximal contiguous non-decreasing sequences, put
 95 $\text{HRuns} := \sum_{i=1}^k \frac{|X_i|}{n} \log_2 \frac{n}{|X_i|}$ and let $f_{\text{HRuns}}(X) = n(1 + \text{HRuns}(X))$. For dual runs, consider
 96 the partition $X = X_1 \sqcup \dots \sqcup X_k$ into maximal *value-contiguous* non-decreasing sequences,
 97 put $\text{HRuns}^* := \sum_{i=1}^k \frac{|X_i|}{n} \log_2 \frac{n}{|X_i|}$ and let $f_{\text{HRuns}^*}(X) = n(1 + \text{HRuns}^*(X))$. McIlroy [25]
 98 shows that merge sort with exponential search is both HRuns -adaptive and HRuns^* -adaptive.
 99 For multi-set sorting, assume $X \subseteq [n]^n$ and denote for $i \in [n]$ let m_i be the number of
 100 occurrences of i . Put $\text{HMulti} := \sum_{i=1}^n \frac{m_i}{n} \log \frac{n}{m_i}$ and $f_{\text{HMulti}}(X) := n(1 + \text{HMulti}(X))$. [27]
 101 shows that merge sort, with a minor adaption, is HMulti -adaptive.

102 **A hierarchy.** Measures of presortedness form a hierarchy. We draw an arrow from a measure
 103 M to a measure N , and write $M \preceq N$, if $f_M(X) \in O(f_N(X))$ for all inputs X . In other words,
 104 any M -adaptive algorithm is automatically also N -adaptive. For example, all occurrences
 105 of the same value in X form a value-contiguous subsequence of X , and each dual run is a
 106 union of such sequences. Consequently, $\text{HRuns}^*(X) \leq \text{HMulti}(X)$, so an algorithm being
 107 HRuns^* -adaptive implies adaptivity for multi-set sorting. Figure 1 illustrates the hierarchy
 108 of the entropy measures, and with our added edges its source is Reg . Here, we explain the
 109 definition of Reg (which is complicated). It considers for each element i its rank in $X[1..i]$.

$$110 \quad \text{Reg}(X) := \prod_{i=2}^n \min_{j < i} \{(i-j) \cdot \max\{1, |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j])|\}\}.$$

111 The measure Reg is derived from insertion sort: process X from left to right and insert $X[i]$
 112 into $X[1..(i-1)]$ using a good *finger* $X[j]$. A good finger is close to $X[i]$ both in terms
 113 of index and rank. Crucially, and inconveniently (as acknowledged by McIlroy [25]), these
 114 ranks are taken only among the elements seen so far. For example, for $X = (1, 5, 4, 3, 2)$
 115 consider $i = 2, j = 1$: when $X[2] = 5$ is inserted, only $X[1] = 1$ and $X[2] = 5$ have
 116 been seen, so the rank difference is 1 and the cost is $(2-1)|2-1| = 1$, even though
 117 $|\text{rank}_X(X[2]) - \text{rank}_X(X[1])| = 4$. Thus, Reg considers this insertion to be cheap as 2, 3, 4
 118 have not yet been inserted. Unfortunately, this makes Reg not invariant under reversing X .
 119 Our alternative, weaker, but more natural definition reg simply considers for $X[i]$ and $X[j]$
 120 their ranks in the full input X ; the target cost $n + \log \text{reg}(X)$ is invariant up to constant
 121 factors under reversing the sequence or its value order and (for input without duplicates)
 122 under inversion: $n + \log \text{reg}(X) \in \Theta(n + \log \text{reg}(X^{-1}))$.

$$123 \quad \text{reg}(X) := \prod_{i=2}^n \min_{j < i} \{(i-j) \cdot \max\{1, |\text{rank}_X(X[i]) - \text{rank}_X(X[j])|\}\}.$$

124 **Problem statements.** Our input is a list X of n elements from a totally ordered universe,
 125 which may contain duplicates. The first is comparison-based sorting: compute a non-
 126 decreasing ordering of X using only comparisons to determine the relative order of elements.
 127 The second is offline BST access: given a BST containing exactly the distinct values of X ,
 128 access $X[1], X[2], \dots, X[n]$ in input order (see Section 3 for the formal definition of the BST
 129 model). The goal is to obtain adaptive bounds for both problems.

2 Discovery

In this section, we discover that Reg corresponds to the unified bound for dictionaries [3] and establish the hierarchy in Figure 2. In Appendix C we extend the hierarchy so that Reg dominates all other previous measures in Figure 1. Thus, we *discover* that the dictionary from [3] yields an insertion-sort algorithm that dominates all previously studied adaptive sorting algorithms from [5, 6, 8, 9, 20, 22–27, 31]. We next revisit the observation of McIlroy [25] that Reg is mathematically inelegant. We make the case for a mathematically more convenient and elegant (but slightly weaker) measure reg , and show that this measure also suffices to dominate all measures in the hierarchy except for three, all equally mathematically inconvenient, measures listed in Section 2.3. In Appendix D, we further explore the definitions of presortedness measures and explain how the classical Hist and Loc from Figure 1 correspond to the working set and dynamic finger properties for binary search trees, respectively. For completeness, Appendix E discusses the measure by Barbay, Ochoa and Satti [7].

Discovering a Reg -adaptive algorithm. Badoiu, Cole, Demaine and Iacono [3, Theorem 3] design a *unified bound* dictionary. We show that sorting with their dictionary is Reg -adaptive.

► **Definition 1.** A dictionary maintains a set Y of distinct elements with multiplicities. Inserting x adds it to Y if absent, and otherwise increases its multiplicity. The dictionary can report Y in sorted order in $O(|Y|)$ time. Insertion sort inserts all elements of X , and then reports Y in sorted order, repeating each element according to its multiplicity. Let $\text{age}(y)$ count the distinct elements accessed since the last access to y . The first insertion takes $O(1)$ time. Thereafter, the unified bound demands that inserting x takes amortized $O(\min_{y \in Y} \{\log(2 + |\text{rank}_{Y \cup \{x\}}(x) - \text{rank}_{Y \cup \{x\}}(y)|) + \log(2 + \text{age}(y))\})$ time.

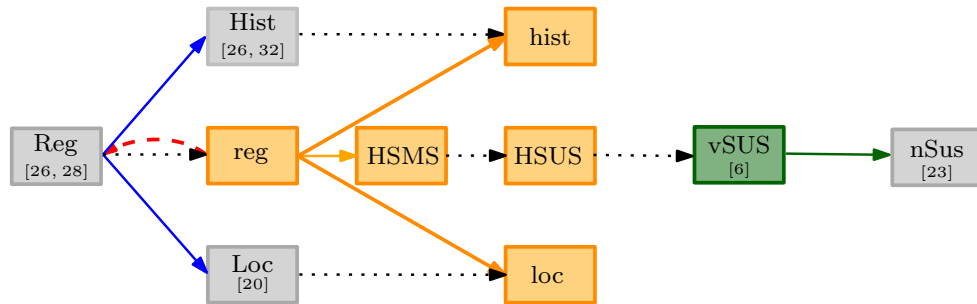
► **Theorem 2.** Insertion sort with a unified bound dictionary sorts a sequence X of n elements in $O(n + \log \text{Reg}(X))$ time and comparisons.

Proof. We insert X into a unified bound dictionary. Processing $X[i]$ consists of inserting it if absent, or searching for it and increasing its multiplicity otherwise.

For every $j < i$, the last access to $X[j]$ occurred at or after step j , so $\text{age}(X[j]) \leq i - j$. Since rank ignores duplicates, and $Y \cup \{X[i]\}$ and $X[1..i]$ contain the same values, we have $\text{rank}_{Y \cup \{X[i]\}}(\cdot) = \text{rank}_{X[1..i]}(\cdot)$. Thus, the unified bound gives a total running time of

$$O(n + \sum_{i=2}^n \min_{j < i} \{\log(2 + |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j])|) + \log(2 + i - j)\}) = O(n + \log \text{Reg}(X)).$$

Finally, we output the distinct elements in sorted order, repeating each according to its multiplicity, in $O(n)$ time. The number of comparisons is bounded by the running time. ◀



■ **Figure 2** The relations established in this section. Orange and red are new, green is [6].

2.1 The Case for reg

The measure Reg has a mathematically inconvenient property: reversing X may significantly change its value. For example, let $n = k^2$ with $k \geq 2$, put $S = (1, k+1, \dots, (k-1)k+1)$, and let $X = (S, \dots, S, 1, 2, \dots, n)$, with k copies of S . Thus, X has length $2n$. In X , consecutive items have prefix-rank distance at most 1, except at the start of each subsequent copy of S and at the start of the final increasing run, where a previous occurrence of the same value is at index distance k . Thus, $\text{Reg}(X) \leq k^k = 2^{o(n)}$.

However, reversing X places $(n, n-1, \dots, 1)$ first, followed by k reversed copies of S . Consider the $n-k$ accesses after the first reversed copy of S . Every finger in the initial descending run has index distance at least k . Among the reversed copies of S , different values have prefix-rank distance at least k , since all values have already appeared, while equal values have index distance at least k . Thus, each of these $n-k$ accesses contributes a factor at least k to $\text{Reg}(\text{Reverse}(X))$. Conversely, choosing the preceding item as a finger bounds each factor by n , and the initial descending run contributes only factors of 1. Hence $\text{Reg}(\text{Reverse}(X)) = k^{\Theta(n)} = 2^{\Theta(n \log n)}$.

Consequently, there are two different, incomparable variants of Reg which we find not particularly elegant. For this reason, we wish to make the case for the simpler measure

$$\text{reg}(X) := \prod_{i=2}^n \min_{j < i} \{(i-j) \cdot \max\{1, |X[i] - X[j]|\}\}.$$

In fact, McIlroy [25] already notes that Reg is needlessly complicated compared to reg (although he incorrectly claims that $\log \text{reg} \leq \log \text{Reg} + O(n)$, see above counterexample). Although reg is weaker than Reg, it appears to capture the essence of both measures. We establish another property that we believe illustrates the mathematical elegance of reg: the bound $n + \log \text{reg}$ is equivalent to the weight of a minimum spanning tree in a natural graph. This characterisation makes $O(n + \log \text{reg})$ invariant up to constant factors under reversing X , reversing its values, or inverting X . We show:

► **Theorem 3.** *Given X of length $n \geq 2$, consider the complete graph $G(X)$ on $[n]$, with edge weights $w(\{i, j\}) = \log(2 + |i - j| + |X[i] - X[j]|)$. Let $\text{unimst}(X)$ be the weight of a minimum spanning tree of $G(X)$. Then $n + \log \text{reg}(X) \in \Theta(\text{unimst}(X))$.*

► **Definition 4.** *A finger function is a map $f : [n] \setminus \{1\} \rightarrow [n]$ with $f(i) < i$. Recall that $\text{reg}(X)$ minimises $\prod_{i=2}^n (i - f(i)) \max\{1, |X[i] - X[f(i)]|\}$ over all finger functions f .*

► **Lemma 5.** *Every forest with m edges contains a spanning subgraph H with at least $m/2$ edges, such that every connected component of H is a star.*

Proof. Root each tree arbitrarily and partition its edges according to whether the child endpoint has even or odd depth. In either part, every vertex is adjacent either only to its parent or only to its children. Thus, every connected component is a star. Since the two parts partition the m edges, one contains at least $m/2$ edges. ◀

Proof of Theorem 3. Every finger function f defines a spanning tree, since $f(i) < i$ for all $i > 1$. Moreover, $w(\{i, f(i)\})$ is at most $2 + \log(i - f(i)) + \log \max\{1, |X[i] - X[f(i)]|\}$. Taking the minimum over all finger functions gives $\text{unimst}(X) \leq 2n + \log \text{reg}(X)$.

Conversely, let T be a minimum spanning tree. We construct a finger function f whose edges have total weight $O(w(T) + n)$.

Let T' be a copy of T . Initially, every vertex forms its own 0-cluster. In rounds $r = 1, 2, \dots$, contract all $(r-1)$ -clusters and consider the forest F of edges whose current weight lies

in $[2r - 2, 2r)$. By Lemma 5, choose a star forest $H \subseteq F$ containing at least half its edges. Merge the clusters of each star into an r -cluster, and increase the weight of every edge in $F - H$ by 2. Clusters not involved in a merge remain unchanged.

We first bound the diameter of the clusters. Put $d(i, j) = 2^{w(i, j)}$. By the triangle inequality, $d(i, j) \leq d(i, k) + d(k, j)$. Let D_r be the maximum d -diameter of an r -cluster. Since every star has diameter realized by at most two edges, each of weight less than $2r$, we have $D_0 = 2$ and $D_r \leq 3D_{r-1} + 2 \cdot 2^{2r}$. Induction gives $D_r \leq 2^{2r+3}$. Thus, any two vertices in the same r -cluster have weight at most $2r + 3$.

The *leader* of a cluster is its smallest-index vertex. Whenever an r -cluster merges k smaller clusters with leaders $i_1 < \dots < i_k$, we set $f(i_j) = i_1$ for every $j > 1$. This is valid since $i_1 < i_j$. Moreover, each vertex receives a finger exactly when it ceases to be a leader. Since eventually all vertices belong to one cluster, whose leader is 1, every vertex except 1 receives exactly one finger. These $k - 1$ finger edges have weight at most $2r + 3$, while the corresponding star contains $k - 1$ edges of weight at least $2r - 2$. Summing over all merges gives $w(f) \leq w(T') + 5(n - 1)$.

Finally, every round uses at least as many edges as it increases in weight. Since each edge is used in exactly one merge, the total weight increase is at most $2(n - 1)$. Hence $w(T') \leq w(T) + 2(n - 1)$, and therefore $w(f) \in O(w(T) + n)$. Since $\log \text{reg}(X) \leq 2w(f)$ and $w(T) \geq n - 1$, we conclude that $n + \log \text{reg}(X) \in O(\text{unimst}(X))$. ◀

2.2 Clean definitions for existing measures: loc, hist, HSMS, HSUS

In addition to Reg, Moffat and Petersson [26] define Hist and Loc (see Table 1), which lie high in the hierarchy of Figure 1. Both measures have been studied through adaptive sorting algorithms [8, 26]. Like Reg, they use prefix ranks and thus share its mathematical inconveniences. McIlroy [25] briefly mentioned more elegant alternatives, analogous to reg:

$$\text{loc}(X) = \prod_{i=2}^n \max\{1, |X[i] - X[i-1]|\} \quad \text{and} \quad \text{hist}(X) = \prod_{i=2}^n |\bar{X}^{-1}[i] - \bar{X}^{-1}[i-1]|.$$

We can define mathematically convenient counterparts for more existing measures: Levopoulos and Petersson [23] study partitions $X = X_1 \sqcup \dots \sqcup X_k$ into non-decreasing subsequences. As the X_i need not be contiguous or value-contiguous, this generalises both runs and dual runs. They define nSUS as the minimum number k of parts, and give a sorting algorithm with adaptive bound $f_{\text{nSUS}}(X) = n(1 + \log \text{nSUS}(X))$. Barbay and Navarro [6] strengthen this bound by considering the entropy of the part sizes rather than their number. They compute a partition with nSUS parts and define $H(\text{vSUS})$ as the entropy of its part sizes. Since any partition into k parts has entropy at most $\log k$, this strengthens the nSUS bound. However, as they observe, their partition need not minimize entropy. We therefore define

$$\text{HSUS}(X) = \min_{X=X_1 \sqcup \dots \sqcup X_k} \sum_{i=1}^k \frac{|X_i|}{n} \log \frac{n}{|X_i|}$$

where the minimum is over all partitions into non-decreasing subsequences. Clearly, $\text{HSUS} \leq H(\text{vSUS})$, giving a stronger measure. We further generalize these partitions by allowing each X_i to be either non-decreasing or non-increasing. Such partitions are called *shuffled monotone sequences* (SMS). Levopoulos and Petersson [23] define nSMS(X) as the minimum number of parts in such a partition. We define its entropy counterpart as

$$\text{HSMS}(X) = \min_{X=X_1 \sqcup \dots \sqcup X_k} \sum_{i=1}^k \frac{|X_i|}{n} \log \frac{n}{|X_i|}$$

where the minimum is over all partitions into monotone subsequences. Every SUS partition is also an SMS partition, so $\text{HSMS} \preceq \text{HSUS}$ and $\text{nSMS} \preceq \text{nSUS}$. Moreover, every partition into nSMS parts has entropy at most $\log \text{nSMS}$, so $\text{HSMS} \preceq \text{nSMS}$. Our entropy-minimizing variants are therefore stronger than their classical counterparts, and independent of the choice of a particular partition. We further study these relations in Section 2.3 and Appendix C. Our main point is that reg dominates these strong, historically motivated measures.

2.3 The strength of reg

In Figure 1, reg dominates all measures except Reg , Hist , and Loc . These three measures all rely on ranks within a prefix of X , rather than within the full input. Replacing them by their full-rank counterparts reg , hist , and loc , leaves reg as the strongest measure. We illustrate this by proving $\text{reg} \preceq \text{loc}$, $\text{reg} \preceq \text{hist}$, and $\text{reg} \preceq \text{HSMS}$.

► **Lemma 6.** $\text{reg} \preceq \text{loc}$.

Proof. Choosing $f(i) = i - 1$ gives $\text{reg}(X) \leq \prod_{i=2}^n \max\{1, |X[i] - X[i-1]|\} = \text{loc}(X)$. ◀

► **Lemma 7.** $\text{reg} \preceq \text{hist}$.

Proof. First suppose that X contains no duplicates. By Lemma 6, $\text{reg}(X^{-1}) \leq \text{loc}(X^{-1}) = \text{hist}(X)$. Moreover, recall that $G(X)$ is the complete graph on $[n]$ with edge weights $w(\{i, j\}) = \log(2 + |i - j| + |X[i] - X[j]|)$. Since X is a permutation, $G(X)$ and $G(X^{-1})$ are isomorphic: mapping each vertex i to $X[i]$ exchanges indices and values, preserving every edge weight. Thus, by Theorem 3, $n + \log \text{reg}(X) \in \Theta(n + \log \text{reg}(X^{-1})) \subseteq O(n + \log \text{hist}(X))$.

If X contains duplicates, recall that $\text{reg}(X) \leq \text{reg}(\bar{X})$ and $\text{hist}(X) = \text{hist}(\bar{X})$. Applying the previous argument to the permutation $\bar{X}$ proves the claim. ◀

► **Lemma 8.** $\text{reg} \preceq \text{HSMS}$.

Proof. Let $X = X_1 \sqcup \dots \sqcup X_k$ be any partition into monotone subsequences. For each $X_i = (X[a_1], \dots, X[a_t])$, where $a_1 < \dots < a_t$, set $f(a_j) = a_{j-1}$ for $j > 1$, and $f(a_1) = 1$ if $a_1 > 1$. This defines a finger function. Recall the edge weights $w(\{i, j\}) = \log(2 + |i - j| + |X[i] - X[j]|)$ from Theorem 3. Since X_i is monotone, and all values are ranks in $[n]$, we have

$$\sum_{j=2}^t ((a_j - a_{j-1}) + |X[a_j] - X[a_{j-1}]|) \leq 2n. \text{ Thus, by concavity of the logarithm, for } t > 1,$$

$$\sum_{j=2}^t w(\{a_j, a_{j-1}\}) \leq (t-1) \log \left(2 + \frac{2n}{t-1} \right) \in O \left(t \left(1 + \log \frac{n}{t} \right) \right).$$

The possible additional edge $\{a_1, 1\}$ has weight $O(\log n)$. Since $\log n \leq t(1 + \log(n/t))$, the total weight contributed by X_i is $O(t(1 + \log(n/t)))$, also when $t = 1$. Summing over all subsequences gives $n + \log \text{reg}(X) \in O \left(n + \sum_{i=1}^k |X_i| \log \frac{n}{|X_i|} \right) = O(n(1 + H(X_1, \dots, X_k)))$. Taking the minimum over all monotone partitions shows $\text{reg} \preceq \text{HSMS}$. ◀

Finally, we show in Appendix C.2 that any reg -adaptive algorithm is also Reg -adaptive, if the input has no duplicates.

► **Theorem 9.** *If X is a list without duplicates, then $\log \text{reg} \in O(n + \log \text{Reg})$.*

3 SproutSort in the BST model

We study the BST model where the input X is a permutation of $[n]$ with n a power of two. Note that we can transform any input X' with distinct keys into X in Reg-adaptive time by sorting X' using [3] and padding. We show a BST-access algorithm that runs in reg-adaptive time. We extend the algorithm to allow X to have duplicate keys in Section 3.3.

The offline BST model. We study the cost of accessing X in the offline BST model [14, 32]. This model maintains a *single* BST containing exactly the distinct values occurring in X . Each value is stored at one node; at every node, all values in the left subtree are smaller and all values in the right subtree are larger. The algorithm knows X in advance and must access $X[1], X[2], \dots, X[n]$ in order. Every access starts with a pointer at the root and must visit the node storing the requested value. At unit cost, the algorithm may move the pointer to its parent or either child, or rotate the current node with its parent. The tree persists between accesses. The total cost is n plus the number of pointer moves and rotations.

BST merge sort. A *BST merge sort* [8] operates on *multiple* BSTs and builds a BST that contains the input values. A size-one input yields a one-node BST. Otherwise, the algorithm picks $k \in [n - 1]$, recursively computes BSTs T_1 and T_2 for $X[1..k]$ and $X[(k + 1)..n]$, and combines them by a *BST merge*: this operation intuitively merges two trees T_1 and T_2 by selecting a connected set of nodes S_1 and S_2 (that each contain the root) and arbitrarily re-arranging their parent and child pointers such that the result is a BST that contains the elements of T_1 and T_2 . The cost of this merge is then $|S_1| + |S_2|$. Formally, for T_1 and T_2 , mark a connected set of nodes S_1 and S_2 containing the root and the *block boundaries*: nodes in T_1 and T_2 whose predecessor or successor in X lies in the other T_i . The merge returns a BST T on exactly the nodes of T_1 and T_2 , minus duplicates, such that $S := S_1 \cup S_2$ is connected and contains the root of T . The total cost is the sum of all merge costs. Blelloch and Dobson [8] show a strong link between merge sort and the offline BST model:

► **Theorem 10** (Theorem 6 in [8]). *A BST merge sort that sorts a permutation X with cost $f(X)$ induces an offline BST algorithm that accesses $X[1], X[2], \dots, X[n]$ with cost $O(f(X))$.*

Their same proof directly applies to input lists with duplicates! Thus, to obtain good access bounds in the BST model, we may focus on designing a BST merge sort algorithm.

3.1 SproutSort as an algorithm

We first introduce SproutSort as a classical algorithm. In Section 3.2 we change it into a BST merge sort. Classical merge sort (Algorithm 1) recursively sorts the two halves of X and merges them in linear time. On X , the merge-sort recursion tree R is then a complete binary tree with n leaves. Our core idea is to store the sorted sublists corresponding to recursive calls as a forest of *sprout trees*, allowing some merges to take sublinear time.

Algorithm 1 Recursive Merge Sort

```

function MERGESORT( $X$ )
  if  $|X| = 1$  then return  $X$ 
   $n \leftarrow |X|$ 
   $X_\ell \leftarrow \text{MERGESORT}(X[1], \dots, X[\frac{n}{2}])$ 
   $X_r \leftarrow \text{MERGESORT}(X[\frac{n}{2} + 1], \dots, X[n])$ 
  return Merge( $X_\ell, X_r$ )

```

Sprout forests. Let T be a complete binary tree with leaves $1, \dots, n$, and associate the element $X[i]$ with the leaf $X[i]$. The *height* of a node $u \in T$ is the path length from u to a leaf in its subtree. For an integer $k \geq 0$ and a subset $Y \subseteq X$, the *sprout forest* $T_k[Y]$ is the subgraph of T induced by all nodes $u \in T$ on the root-to-leaf path of some leaf in Y that have height at most k . In particular, the set of leaves of $T_k[Y]$ is precisely Y . See Figure 3. To represent $T_k[Y]$, for a fixed pair (k, Y) , store a root list with all roots of $T_k[Y]$ in-order. Store each tree in $T_k[Y]$ as a pointer structure where vertices in $T_k[Y]$ are nodes with two pointers to their left and right children. Each node $u \in T_k[Y]$ stores the index of the smallest and largest leaf in its subtree, plus a pointer to the node corresponding to u in T .

Our sprout trees support two operations. $\text{GROW}(T_k[Y])$ for $k \geq 0$ and $Y \subseteq X$ computes $T_{k+1}[Y]$. $\text{MERGE}(T_k[Y_1], T_k[Y_2])$ for $k \geq 0$ and $Y_1, Y_2 \subseteq X$ computes $T_k[Y_1 \cup Y_2]$ (see Figure 4). Both operations consume their input forests. We analyse sequences of these operations with the accounting method [13]. We assign one coin to each node which can pay for $O(1)$ work. The amortized running time of MERGE is $O(m)$ where m is the number of trees in the output.

► **Theorem 11.** *Consider any sequence of GROW and MERGE operations, with $k \leq \log n$, starting with a collection of sprout forests that have n nodes in total. Let m_i be the size of the root-list of the forest produced by operation i . The total running time is $O(n + \sum_i m_i)$. Moreover, MERGE never increases the number of nodes.*

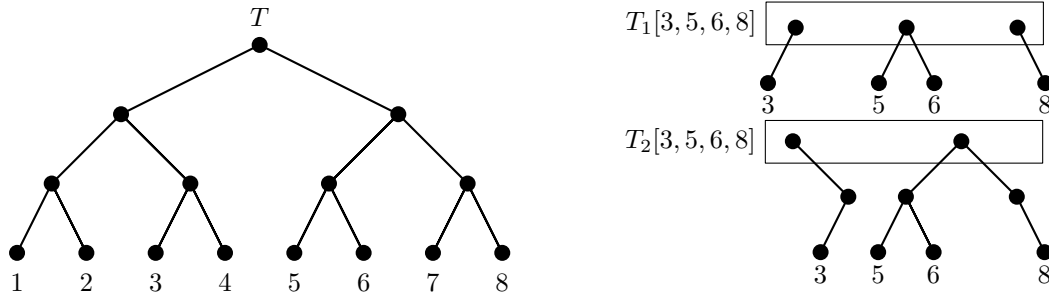
Proof. Initially, we create one coin for each existing node. Whenever we remove a node, we spend its coin to do $O(1)$ work. Additionally, the i -th operation can take $O(m_i)$ time.

A $\text{GROW}(T_k(Y))$ operation scans all roots in $T_k(Y)$. For each root r it encounters, it either creates a new root with r as its singular child or a new root with two children, one of which is r . Thus, GROW inspects at most $3m_i$ nodes, and, since the new root list follows the in-order traversal of the old root list, GROW takes $O(m_i)$ time. GROW also creates m_i coins.

For $\text{MERGE}(T_k(Y_1), T_k(Y_2))$, observe that each root in $T_k(Y)$ is at least a root in either $T_k(Y_1)$ or $T_k(Y_2)$, and at most a root in both. Thus, the total number of trees in $T_k(Y_1)$ and $T_k(Y_2)$ is at most $2m_i$, and since root-lists of $T_k(Y_1)$ and $T_k(Y_2)$ are in-order, we can identify these roots in $O(m_i)$ time. Having identified colliding roots, we invoke our TREEMERGE procedure on all trees that share a root (see Algorithm 2). Whenever this recursive merge replaces two nodes by one, we transfer one node's coins to the replacement and spend the other's on the $O(1)$ work needed for this merge. TREEMERGE does not create any coins.

In total, we create $O(n + \sum_i m_i)$ coins and spend $O(m_i)$ time beyond the coin budget. ◀

SproutSort We now design an algorithm SPROUTSORT that uses GROW and MERGE to compute $T_k[X]$ where $|X| = 2^k$. Similar to recursive merge sort, the algorithm splits $X = X_\ell \sqcup X_r$ into equal halves and recursively computes $T_{k-1}[X_\ell]$ and $T_{k-1}[X_r]$. It then returns $T_k[X] = \text{GROW}(\text{MERGE}(T_{k-1}[X_\ell], T_{k-1}[X_r]))$. See Algorithm 3.



■ **Figure 3** A complete binary tree on X and a sprout forest before and after GROW .

■ **Algorithm 2** Recursively merging sprout trees.

```

function TREEMERGE( $T_1, T_2$ )
  if  $T_1$  is nil then return  $T_2$ 
  if  $T_2$  is nil then return  $T_1$ 
  return a new node with children TREEMERGE( $T_1.\ell, T_2.\ell$ ) and TREEMERGE( $T_1.r, T_2.r$ )

```

■ **Algorithm 3** Recursive SproutSort.

```

function SPROUTSORT( $X$ )
  if  $|X| = 1$  then return  $T_0[X]$ 
   $m \leftarrow |X|$ ,  $k \leftarrow \log_2 m$ 
   $X_\ell \leftarrow X[1], \dots, X[m/2]$ ,  $X_r \leftarrow X[m/2 + 1], \dots, X[m]$ 
   $T_{k-1}[X_\ell] \leftarrow \text{SPROUTSORT}(X_\ell)$ 
   $T_{k-1}[X_r] \leftarrow \text{SPROUTSORT}(X_r)$ 
  return GROW(MERGE( $T_{k-1}[X_\ell], T_{k-1}[X_r]$ ))

```

377 ► **Definition 12.** Recall that a finger function is a map $f : [n] \setminus \{1\} \rightarrow [n]$ satisfying $f(i) < i$.
 378 For $i \geq 2$, we define the graph distances:
 379 ■ $\text{dist}_R(i, f(i))$, between leaves i and $f(i)$ in the complete binary tree R on $[n]$.
 380 ■ $\text{dist}_T(i, f(i))$, between leaves $X[i]$ and $X[f(i)]$ in T .

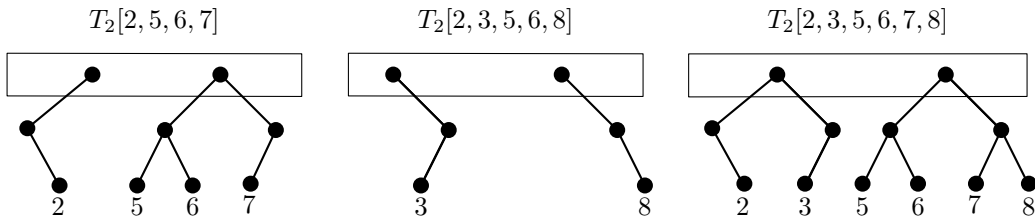
381 For example, for $n \geq 4$, $\text{dist}_R(\frac{n}{2}, \frac{n}{2} + 1) = 2 \log n$ and $\text{dist}_R(\frac{n}{2} + 1, \frac{n}{2} + 2) = 2$.

382 ► **Lemma 13.** Let $n \geq 2$ and let f be any finger function. The running time of SPROUTSORT
 383 is $O(\sum_{i=2}^n (1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$.

384 **Proof.** Let g be the total number of nodes created by GROW. Observe that GROW creates
 385 one node per tree in its output forest, MERGE does not increase the number of nodes, and
 386 both GROW and MERGE reduce the size of the root list by at most a factor 2. Hence, an
 387 invocation of GROW(MERGE(...)) in SPROUTSORT that creates g_i nodes, involves root lists
 388 of size at most $4g_i$. Thus, by Theorem 11, the total running time is $O(n + \sum_i g_i) = O(n + g)$.

389 Whenever GROW creates a node u , we denote by $\text{subtree}(u)$ the resulting sprout tree
 390 rooted at u . We charge the creation of u to the smallest $i \geq 2$ such that $X[i] \in \text{subtree}(u)$
 391 but $X[f(i)] \notin \text{subtree}(u)$. If no such i exists, we charge u to item 1. This happens only if
 392 $X[1] \in \text{subtree}(u)$, which itself happens for at most one node u per level of recursive depth
 393 in our algorithm, for a total charge of $O(\log n)$ to item 1.

394 For all other $i \geq 2$, we bound the charge they receive. At every recursion height
 395 $k \geq \text{dist}_R(i, f(i))/2$, items i and $f(i)$ belong to the same recursive input X' . Thus, they
 396 belong to the same sprout forest $T_k(X')$. If, furthermore $k \geq \text{dist}_T(i, f(i))/2$ then they also
 397 lie in the same sprout tree of $T_k(X')$. Item i participates in only one GROW operation per



■ **Figure 4** Two sprout forests and their merge.

recursive height, and stops being charged once items i and $f(i)$ share a sprout tree. Item i therefore receives at most $O(1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i)))$ charges. Consequently, $g \in O(\log n + \sum_{i=2}^n (1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$. The sum is at least $n - 1$, so it absorbs the $O(n)$ initialization cost and the $O(\log n)$ charges to item 1. ◀

► **Lemma 14.** *Let X be a permutation of $[n]$ and let f be any finger function. Then $\sum_{i=2}^n \text{dist}_R(i, f(i)) \in O(n + \sum_{i=2}^n \log(i - f(i)))$ and $\sum_{i=2}^n \text{dist}_T(i, f(i)) \in O(n + \sum_{i=2}^n \log |X[i] - X[f(i)]|)$.*

Proof. We prove the bound for dist_T , which is the more difficult case because $X[f(i)]$ may lie on either side of $X[i]$. Construct T from the leaves upwards. In round k , the leaves are partitioned into groups of 2^k consecutive values. If two leaves start belonging to the same group in round t , their lowest common ancestor is t edges above them. Thus, $\text{dist}_T(i, f(i))$ is twice the number of rounds in which $X[i]$ and $X[f(i)]$ belong to different groups.

Write $\delta_i = |X[i] - X[f(i)]|$. For a fixed i , there are at most $1 + 2 \log \delta_i$ rounds with $2^k \leq \delta_i^2$. We bound the remaining rounds by counting how often they occur across all indices.

Fix a round k with $2^k > \delta_i^2$, and suppose $X[i]$ and $X[f(i)]$ belong to different groups. Let $[a, b]$ be the group containing $X[i]$. If $X[f(i)] < a$, then $X[i] - a + 1 \leq \delta_i$; if $X[f(i)] > b$, then $b - X[i] + 1 \leq \delta_i$. Hence $X[i]$ is among the first or last $\delta_i < 2^{k/2}$ values of its group. There are $n/2^k$ groups, and X is a permutation, so this happens for at most $2n/2^{k/2}$ indices in round k . Summing over all rounds gives at most $2n \sum_{k=0}^{h-1} 2^{-k/2} = O(n)$ such occurrences.

The first kind of round contributes $O(n + \sum_{i=2}^n \log \delta_i)$ occurrences, and the second contributes $O(n)$. Each occurrence contributes two to the distance, proving the lemma. ◀

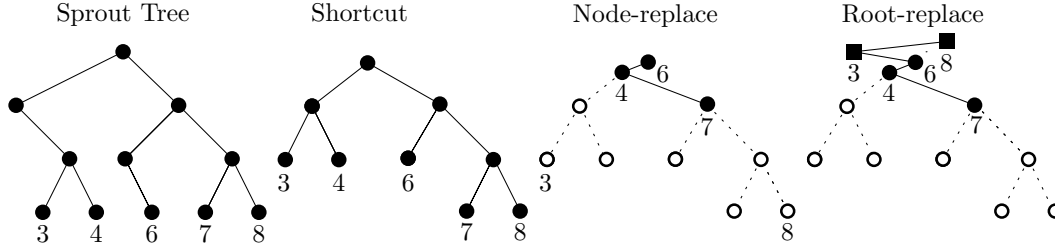
► **Theorem 15.** *If X is a permutation and T is a complete binary tree, then the running time of **SPROUTSORT** is $O(n + \log \text{reg})$.*

Proof. Let f be an arbitrary finger function. By Lemma 13, the running time is $O(\sum_{i=2}^n (1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$. By Lemma 14, the running time is therefore upper-bounded by $O(n + \sum_{i=2}^n \log(i - f(i)) + \sum_{i=2}^n \log |X[i] - X[f(i)]|)$. By the definition of reg , there exists some finger function f such that $O(n + \log \text{reg}) = O(n + \sum_{i=2}^n \log(i - f(i)) + \sum_{i=2}^n \log |X[i] - X[f(i)]|)$ and so the theorem follows. ◀

3.2 Running SproutSort in the BST Model

What remains is to implement **SPROUTSORT** as a BST merge sort algorithm, which then implies an algorithm for BST access. That is, we must implement our recursive algorithm so that the output of each recursive call is a BST whose nodes are exactly the input of the call. The input is a list of singleton BSTs whose left-to-right order is X . Each recursive call combines its two output BSTs using the **MERGE** operator from the beginning of the section.

Consider a recursive call of **SPROUTSORT** at recursion height k on a contiguous sublist X' of X . To represent a sprout tree $T' \in T_k[X']$ as a BST, first contract all its maximal unary paths, yielding a tree T'' . Assign each internal node u of T'' the key of the rightmost leaf in its left subtree, unless this subtree consists of a single leaf. Similarly, assign it the key of the leftmost leaf in its right subtree, unless this subtree consists of a single leaf. Split every node that is assigned two keys, putting the larger key above the smaller. Remove the original leaves and any internal nodes that were not assigned a key, yielding a BST T''' containing all keys except the leftmost and rightmost. Finally, make the rightmost key v a new root, the leftmost key u the left child of v , and T''' the right subtree of u . A one-leaf sprout tree becomes a singleton BST; see Figure 5. To represent a forest $T_k[X']$, simply represent each



■ **Figure 5** Converting a sprout tree to a BST in three steps. White nodes are implicit; each root-to-leaf path has a corresponding BST traversal.

441 sprout tree in this way and connect the roots in a linked list via their right child pointers.
 442 This yields a BST, as the root list of a sprout forest contains the items in increasing order.

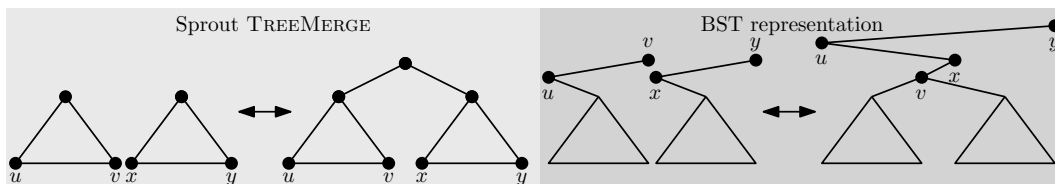
443 **Our operations.** The GROW operation traverses the root list of a sprout forest and may
 444 attach pairs of adjacent trees to a new root. In the BST representation, this corresponds to
 445 traversing the root list through right-child pointers and performing $O(1)$ rotations per pair.
 446 The number of pointer traversals and rotations performed is bounded by the root list size.

447 The MERGE operation, when given two disjoint BSTs, traverses their root lists and calls
 448 TREEMERGE (Algorithm 2) on sprout trees that share the same root. Each recursive call
 449 to TREEMERGE separates the root of each sprout subtree T_i from its children $T_i.l, T_i.r$,
 450 recursively merges the left and right children, and then re-attaches the results to a new root.
 451 In the BST representation, separating its root from its two children involves up to four nodes
 452 near the root. Re-attaching to a new root is the reverse (see Figure 6). For each separation
 453 operation, we mark the roots of the resulting binary search trees, plus their left children
 454 (nodes u, v, x, y in Figure 6). At the start of MERGE, we also mark all nodes on the right
 455 spine of each input BST (i.e., the root list plus their left children). With these markings,
 456 the marked nodes of each input BST form a connected subtree containing its root, and
 457 unmarked subtrees remain unchanged. Moreover, every block boundary between the two input
 458 BSTs is exposed during TREEMERGE and therefore marked. Thus, GROW(MERGE(...)) in
 459 SPROUTSORT is a BST merge. The number of marked nodes is bounded by the number of
 460 root-list entries traversed and nodes processed by TREEMERGE. By Theorem 11 and our
 461 analysis of SPROUTSORT, the total BST merge cost is $O(n + \log \text{reg}(X))$, which shows:

462 ► **Theorem 16.** *SPROUTSORT is a BST merge sort [8]. Thus, it yields an offline BST*
 463 *algorithm for accessing a permutation X in $O(n + \log \text{reg}(X))$ node traversals and rotations.*

464 3.3 Duplicate Items

465 So far, we assumed that X has no duplicates. We now show that SPROUTSORT can be
 466 adapted to the case where X contains duplicate items. In the offline BST model for accessing



■ **Figure 6** The primitives used in Sprout TREE-MERGE (left) and their BST representation (right).

elements, this corresponds to having an access sequence X that contains the same access multiple times, where the BST stores the set Y of distinct keys in X . Although input elements can only be compared, we can transform any input X' into a list $X \in [U]^n$, where U is the number of distinct keys in X (via [3]) in $O(n + \log \text{reg})$ time. Furthermore, we can assume that we know for each $x \in [U]$ how many times x occurs in X .

► **Definition 17.** We denote by T a specific binary tree whose leaves are $1, \dots, U$ in-order and have the same depth. T could have many shapes, as we allow nodes with only one child. We construct T cleverly later, and associate each $X[i]$ with the leaf that stores this value.

Given a specifically-shaped, currently-unspecified T , we run **SPROUTSORT** as before where we use T as the skeleton for the **Sprout Trees**, as opposed to the complete binary tree. Each recursive call on $X[i..j]$ returns a sprout forest of the distinct values in $X[i..j]$. Separate recursive calls store separate copies of their forests, and different forests may contain leaves with the same value. Each leaf with value $x \in [U]$ stores a linked list of the occurrences of x in the corresponding recursive input $X[i..j]$. When **TREEMERGE** encounters two copies of the same leaf, it replaces them by one and concatenates their occurrence lists in $O(1)$ time.

Recreating Theorem 11 and Lemma 13 The amortisation proof of Theorem 11 applies as-is: identifying two copies of a leaf spends the deleted copy's coin on the $O(1)$ work required to identify the duplicate, and the linked list merge. We do have to extend Lemma 13:

► **Lemma 18.** Let T be the tree from Definition 17, f be any finger function and $n \geq 2$. **SPROUTSORT** on input $X \in [U]^n$ runs in time $O(n + \sum_{i=2}^n (\log(i - f(i)) + \text{dist}_T(i, f(i))))$.

Proof. Let g be the total number of nodes created by **GROW**. Let g_j be the number of nodes created by **GROW** in the j -th recursive call; then $g := \sum_j g_j$. If **GROW** creates nodes, it creates one per tree in its output forest. Otherwise, its input forest already contains the root of T , and it returns this single tree unchanged. Thus, its output root list has size at most $g_j + 1$. Since **GROW** combines at most two input trees into each output tree, the preceding **MERGE** has an output root list of size at most $2(g_j + 1)$. By Theorem 11, the total running time is $O(n + \sum_j (g_j + 1)) = O(n + g)$, since there are $n - 1$ non-base recursive calls.

Whenever **GROW** creates a node u in a recursive call on $X[a..b]$, let $C_u := \{i \in [a, b] \mid X[i] \in \text{subtree}(u)\}$ be all input keys in the current recursive call that lie below u . We charge the creation of u to the smallest $i \geq 2$ such that $i \in C_u$ but $f(i) \notin C_u$. If no such i exists, the smallest index in C_u must be 1, since $f(i) < i$. We then charge u to item 1. At each recursion height, only one recursive input contains item 1, and only one newly created sprout tree contains its leaf. Thus, item 1 receives $O(\log n)$ charges.

For all other $i \geq 2$, at every recursion height $k \geq \text{dist}_R(i, f(i))/2$, items i and $f(i)$ belong to the same recursive input. If, furthermore, $k \geq \text{dist}_T(i, f(i))/2$, their leaves belong to the same sprout tree. Item i therefore receives at most $O(1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i)))$ charges. This also holds when $X[i] = X[f(i)]$, in which case $\text{dist}_T(i, f(i)) = 0$. Every node created by **GROW** charges exactly one item, so the total number of charges is $\sum_j g_j$. Summing the bounds above, including the $O(\log n)$ charges to item 1, gives $\sum_j g_j \in O(\log n + \sum_{i=2}^n (1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$. Since we could upper bound our running time by $O(n + \sum_j g_j)$, we can now upper bound our running time by $O(n + \sum_{i=2}^n (\text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$. What remains is to replace all dist_R terms. For this, we can directly invoke Lemma 14, as R is completely unaffected by X containing duplicates. ◀

510 **Choosing the tree of Definition 17.** By Lemma 18, what remains is to bound the total
 511 distance between leaves representing $X[i]$ and $X[f(i)]$ in our custom-made tree T . We choose
 512 T using the number of occurrences of each value via some weight function:

513 ► **Lemma 19.** *Let $w : [U] \rightarrow [n]$ be a weight function with $\sum_{x=1}^U w(x) = n$. There is a binary*
 514 *tree T , possibly containing nodes with one child, whose leaves are $1, \dots, U$ in-order and have*
 515 *the same depth, such that, putting $E(x) := \max_{y \in [U]} \{\text{dist}_T(x, y) - 8 \log \max\{1, |x - y|\}\}$,*
 516 *we have $\sum_{x=1}^U w(x)E(x) \in O(n)$. T has $O(U)$ nodes and can be constructed in $O(U)$ time.*

517 **Proof.** For an integer interval $[a, b] \subseteq [U]$, we construct a tree $T'_{a,b}$ whose leaves are $a, \dots, b$.
 518 If $a = b$, then $T'_{a,b}$ consists of a single leaf labelled a . Otherwise, choose an index $r \in$
 519 $\{a, \dots, b-1\}$ by the rule below. Construct $T'_{a,r}$ and $T'_{r+1,b}$ recursively, and make their roots
 520 the left and right children of a new root. The resulting tree is $T'_{a,b}$. Finally, put $T' = T'_{1,U}$.

521 We choose r as follows. Put $J = [a, b]$, $m = b - a + 1$, and $W = \sum_{x \in J} w(x)$. Require
 522 $m/3 \leq r - a + 1 \leq 2m/3$, so both child intervals contain at most $2m/3$ keys. Put $q = \lceil \sqrt{m} \rceil$.
 523 Starting with the smallest permitted index r_0 , consider $r_0, r_0 + 2q, r_0 + 4q, \dots$, stopping
 524 when the index is no longer permitted. For each candidate r , put $B_r = J \cap [r - q + 1, r + q]$.
 525 These are disjoint, and there are $\Theta(\sqrt{m})$ candidates with total weight at most W , so one has
 526 weight $O(W/\sqrt{m})$. Choose an r minimizing this weight, and call the keys in B_r *bad* for $T'_{a,b}$.

527 We now turn T' into T by lengthening its edges. For each node u of T' , let m_u be its
 528 number of descendant leaves and assign it the target height $h(u) = \lfloor 2 \log m_u \rfloor$. Replace
 529 every parent-child edge pu by a path of $h(p) - h(u)$ edges, inserting nodes with one child.
 530 This length is positive because $m_u \leq 2m_p/3$. Along any path from an original node u to a
 531 descendant leaf, the new edge lengths telescope to $h(u)$, since leaves have target height zero.
 532 Thus all leaves have the same depth. What remains is to bound $\sum_x w(x)E(x)$ in T .

533 For distinct leaves x, y , their lowest common ancestor is the same node u in both trees,
 534 and $\text{dist}_T(x, y) = 2h(u)$. Suppose leaves in the subtree of u form an interval of size m ,
 535 partitioned at the chosen index r . The keys x, y lie in different child intervals. If $x \notin B_r$, then
 536 $|x - y| \geq q \geq \sqrt{m}$, so $\text{dist}_T(x, y) = 2h(u) \leq 4 \log m \leq 8 \log |x - y|$. This pair contributes
 537 no positive excess to $E(x)$. If $x \in B_r$, its excess is at most $4 \log m$. Since $y = x$ contributes
 538 zero, $E(x)$ is at most the sum of $4 \log m$, over all nodes at which x is bad.

539 Consider a node whose descendant interval has size m and weight W . Its bad keys have
 540 weight $O(W/\sqrt{m})$. The contribution of these bad keys to the weighted sum is therefore
 541 $O(W \log m / \sqrt{m})$. Charge each key x (good or bad) in this interval $O(w(x) \log m / \sqrt{m})$. The
 542 nodes whose intervals contain x are the ancestors of x in T' , and their interval sizes shrink
 543 by a factor at most $2/3$. For each k , there are $O(1)$ nodes with $2^k \leq m < 2^{k+1}$. Hence the
 544 total charge to x is $O(w(x) \sum_{k \geq 1} (k+1) 2^{-k/2}) = O(w(x))$. Summing over x gives $O(n)$.

545 To get the running time, compute for $t \in [U]$ the prefix sums $P(t)$ where $P(0) = 0$ and
 546 $P(t) = \sum_{x=1}^t w(x)$ in $O(U)$ time. The weight of each candidate's bad interval $[L, R]$ is then
 547 $P(R) - P(L-1)$, which takes $O(1)$ time to compute. Each leaf $T'_{a,a}$ is created in constant
 548 time. At an internal node with m descendant leaves, choosing r takes $O(\sqrt{m})$ time, and
 549 lengthening its outgoing edges inserts $O(\log m)$ nodes. There are $O(U/2^k)$ nodes of T' with
 550 $2^k \leq m < 2^{k+1}$: each has at least 2^k descendant leaves, and each leaf lies below only $O(1)$
 551 such nodes. Summing over k gives $O(U)$ construction time and nodes. ◀

552 ► **Theorem 20.** *Let $X \in [U]^n$, where we know for each element $x \in X$ its multiplicity in X .*
 553 *With T constructed by Lemma 19, SPROUTSORT sorts X in $O(n + \log \text{reg}(X))$ time.*

554 **Proof.** For each $x \in [U]$, let $w(x)$ be its multiplicity in X , and construct T by Lemma 19;
 555 this takes $O(n)$ time. Let f be any finger function and put $\delta_i := \max\{1, |X[i] - X[f(i)]|\}$.
 556 Since all leaves of T have the same depth, $\text{dist}_T(i, f(i)) = 2 \text{height}(\text{LCA}_T(X[i], X[f(i)])) \leq$
 557 $8 \log \delta_i + 2E(X[i])$. This includes equal-value pairs, whose tree distance is zero. Furthermore,
 558 $\sum_{i=2}^n E(X[i]) \leq \sum_{x=1}^U w(x)E(x) \in O(n)$. By Lemma 18, the running time is therefore
 559 $O(n + \sum_{i=2}^n \log(i - f(i)) + \sum_{i=2}^n \log \delta_i)$. By the definition of reg , there is a finger function f
 560 for which these two sums equal $\log \text{reg}(X)$. Traversing the output leaves and their occurrence
 561 lists takes $O(n)$ additional time and yields a sorted ordering of X . ◀

562 Finally, we transform the sorting algorithm into a BST access algorithm in the same
 563 manner as in Section 3.2. We use the BST representation and markings of Section 3.2,
 564 with one node per distinct key in each recursive output. When TREEMERGE identifies two
 565 copies of a leaf, both corresponding BST nodes are marked before they are identified. Thus,
 566 every key present in both input trees is marked in both. Each input BST contains one node
 567 per distinct key. However, the same key may occur in both input trees, so a merge may
 568 encounter two nodes storing that key. We mark both nodes and combine them into a single
 569 node in the output BST. For example, if both input trees contain a node storing 3, the
 570 output tree contains only one node storing 3. Both input nodes count towards the merge
 571 cost. Unmarked subtrees retain their nodes and children. Combining equal-key nodes does
 572 not remove occurrences from the access sequence: the resulting BST still accesses a key once
 573 for each occurrence in X .

574 The extension preserves the access bound of Theorem 10. In Blelloch and Dobson's con-
 575 struction [8], marked nodes with the same key produce identical geometric points. Unmarked
 576 subtrees contain no shared keys, so their structure and relative priorities remain unchanged.
 577 The resulting set is therefore still arborally satisfied, with $O(n)$ points plus $O(1)$ per marked
 578 input node, giving the same asymptotic BST access cost.

579 ▶ **Theorem 21.** *SPROUTSORT yields a deterministic offline BST algorithm for accessing X*
 580 *with cost $O(n + \log \text{reg}(X))$, also when X contains duplicates.*

581 **Proof.** We use the BST representation and markings of Section 3.2, identifying marked
 582 copies of the same key as described above. Each recursive call performs a BST merge with
 583 the duplicate extension described above. The number of marked input nodes is bounded
 584 by the work performed by GROW and MERGE. By Theorem 20, the total merge cost is
 585 therefore $O(n + \log \text{reg}(X))$. The duplicate extension of Theorem 10 then gives an offline
 586 BST algorithm with the claimed access cost. ◀

587 A SkeletonSort with no duplicates

588 Let X be an input list of arbitrary comparable elements that contains no duplicates. We
 589 present a simple, randomised algorithm that sorts X in expected $O(n + \log \text{reg})$ time, without
 590 any prior knowledge of the ranks of elements in X . Our algorithm is based on Quicksort,
 591 which picks a uniformly random pivot p and partitions X according to whether elements
 592 are smaller or larger than p (recursing on each part). We design a tree data structure that
 593 supports both pivot sampling and partitioning, in expected $O(n + \log \text{reg})$ total time. By
 594 Theorem 9, the algorithm is not only reg-adaptive but also Reg-adaptive.

Algorithm 4 Quicksort.

```

function QUICKSORT( $X$ )
  if  $X$  is empty then
    return an empty list
   $p \leftarrow \text{random-item}(X)$ 
   $L_\ell \leftarrow \text{QUICKSORT}(\{x \in X : x < p\})$ 
   $L_r \leftarrow \text{QUICKSORT}(\{x \in X : x > p\})$ 
  return the concatenation of  $L_\ell$ , the singleton list  $[p]$ , and  $L_r$ 

```

595 **Skeleton Trees.** We again assume that $n = 2^k$ is a power of two (we may always pad the
 596 input with distinct increasing sentinel items declared larger than every key in X , without it
 597 asymptotically changing $O(n + \log \text{reg})$). Let R be the complete binary tree whose leaves
 598 are the indices $1, \dots, n$ in-order. For a subset of indices $I \subseteq [n]$, we define the I -skeleton
 599 of R , denoted R_I , as the tree obtained by keeping the leaves in I and their ancestors, and
 600 shortcutting every node with only one child. For $I = \emptyset$, put $R_I = \text{nil}$. We call such trees
 601 *skeleton trees*, and they are the basis of our SKELETONSORT algorithm. Each internal node
 602 of a skeleton tree has exactly two children, and each node u corresponds to an original
 603 node of R ; the *raw height* of u is the number of edges from that original node to any of its
 604 descendant leaves in R . We store each skeleton tree as a pointer structure with child and
 605 parent pointers. A leaf labelled i stores a pointer to the input item $X[i]$. Each node stores
 606 its number of descendant leaves, together with the minimum and maximum of the input
 607 values $X[i]$ represented by those leaves. For each internal node, these fields can be computed
 608 from the two children in constant time.

609 **Random sampling.** To simulate Quicksort, we need for any $I \subset [n]$ to be able to generate a
 610 uniformly random index $i \in I$. We show how to do this given R_I : the root of this tree has
 611 stored $|I|$. Choose a uniformly random integer $j \in [|I|]$ and find the j -th leaf in-order, using

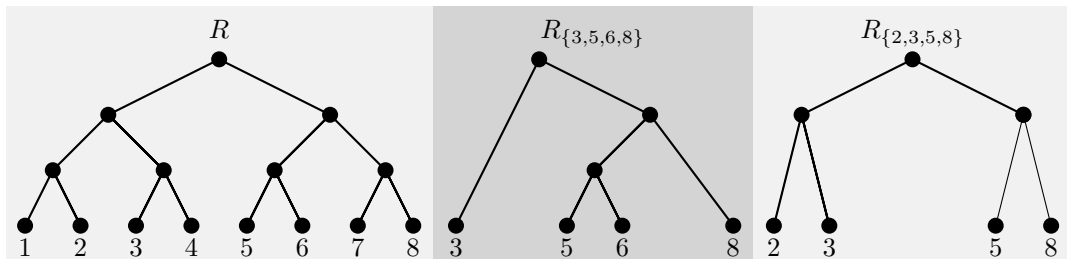


Figure 7 A complete binary tree and two skeleton trees.

the leaf counts to choose the next child. This procedure RANDOM-LEAF returns a pointer to an item sampled uniformly from I and takes time proportional to the number of nodes on its search path. We allow constant-time arithmetic on $O(\log n)$ -bit counters and generation of a uniformly random integer in a specified range of size at most n .

Splitting along a pivot. Given some random index $i \in I$, this yields some pivot value $p = X[i]$. Quicksort then splits the input along this pivot. We simulate this split by defining $I^{<p} = \{i \in I : X[i] < p\}$ and $I^{=p}$ and $I^{>p}$ analogously. The operation SKELETON-SPLIT(R_I, p) (Algorithm 5) consumes its input and returns the three disjoint skeletons $R_{I^{<p}}$, $R_{I^{=p}}$, and $R_{I^{>p}}$. In this code, whenever CREATE-NODE creates a node, it updates the stored fields and the parent pointers of its children in constant time. Every returned non-nil tree has its root's parent pointer reset to nil. When both children are non-nil, the new node corresponds to the same original node of R as the node currently being partitioned.

■ **Algorithm 5** Partitioning skeleton trees.

```

function CREATE-NODE( $S_\ell, S_r$ )
  if both  $S_\ell$  and  $S_r$  are nil then
    return nil
  if exactly one of  $S_\ell, S_r$  is nil then
    return the non-nil tree
   $u \leftarrow$  a new node with children  $S_\ell$  and  $S_r$ 
  Update the stored fields of  $u$  and the parent pointers of its children
  return  $u$ 

function SKELETON-SPLIT( $S, p$ )
  if  $S$  is nil then
    return nil, nil, nil
  if all leaves  $i$  of  $S$  satisfy  $X[i] = p$  then
    return nil,  $S$ , nil
  if all leaves  $i$  of  $S$  satisfy  $X[i] < p$  then
    return  $S$ , nil, nil
  if all leaves  $i$  of  $S$  satisfy  $X[i] > p$  then
    return nil, nil,  $S$ 
   $S_\ell^{<}, S_\ell^{=}, S_\ell^{>} \leftarrow$  SKELETON-SPLIT( $S.\ell, p$ )
   $S_r^{<}, S_r^{=}, S_r^{>} \leftarrow$  SKELETON-SPLIT( $S.r, p$ )
   $S^{<} \leftarrow$  CREATE-NODE( $S_\ell^{<}, S_r^{<}$ )
   $S^{=} \leftarrow$  CREATE-NODE( $S_\ell^{=}, S_r^{=}$ )
   $S^{>} \leftarrow$  CREATE-NODE( $S_\ell^{>}, S_r^{>}$ )
  Discard the old node  $S$ 
  return  $S^{<}, S^{=}, S^{>}$ 

```

624 A.1 SkeletonSort.

Given the above two primitives, we can simulate *Quicksort* without using linear time per recursive call: We construct $R_{[n]} = R$ and its stored fields in $O(n)$ time, then run SKELETONSORT on its root (Algorithm 6). Since the values are distinct, $S^{=}$ consists of the single leaf storing the pivot. Output lists have head and tail pointers, so each concatenation takes $O(1)$ time.

■ **Algorithm 6** SkeletonSort.

```

function SKELETONSORT( $S$ )
  if  $S$  is nil then
    return an empty list
   $p \leftarrow$  the item returned by RANDOM-LEAF( $S$ )
   $S^<, S^=, S^> \leftarrow$  SKELETON-SPLIT( $S, p$ )
   $L_\ell \leftarrow$  SKELETONSORT( $S^<$ )
   $L_r \leftarrow$  SKELETONSORT( $S^>$ )
  return the concatenation of  $L_\ell$ , the singleton list  $[p]$ , and  $L_r$ 

```

630 To analyse this algorithm, fix a random seed and form a tree T_0 as follows. Create one
 631 node for every non-empty recursive invocation of SKELETONSORT. Its children are the nodes
 632 for its non-empty recursive invocations, and the node is labeled by its pivot. Then, T_0 is a
 633 binary search tree that contains the values of X in sorted order.

634 For each node v of T_0 , let $h(v)$ be the height of the subtree of v , i.e. the maximum
 635 number of edges from v to a descendant leaf. The node v is labeled by some pivot x . Remove
 636 this label and instead attach a path of length $h(v)$ to v whose leaf is labeled by x . Finally,
 637 replace every parent-child edge vu by a path of $h(v) - h(u)$ edges, inserting nodes with one
 638 child, and call the resulting tree T . Every path from an original node v to a descendant leaf
 639 now has length $h(v)$, so all leaves of T have the same depth. The resulting tree T is used
 640 only in the analysis; the algorithm does not construct it.

We again use graph distances and slightly overload our notation from previous sections.
 Here, $\text{dist}_R(i, j)$ is the distance between the index leaves i, j in R which is the complete
 binary tree, and $\text{dist}_T(i, j)$ is the distance between the value leaves $X[i], X[j]$ in T .
 Recall that a finger function is a map $f : [n] \setminus \{1\} \rightarrow [n]$ with $f(i) < i$.

641
 642 ► **Lemma 22.** Fix a random seed and let R and T be as above. For any finger function f ,
 643 the running time of SKELETONSORT is $O(n + \sum_{i=2}^n (\text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$.

644 **Proof.** Consider a non-empty invocation of SKELETONSORT on a skeleton tree S , where
 645 RANDOM-LEAF selects leaf i and pivot $p = X[i]$. The subsequent SKELETON-SPLIT must
 646 traverse the entire path from the root of S to i , since every subtree along this path contains p .
 647 Thus, the time spent on RANDOM-LEAF is bounded by the time spent on SKELETON-SPLIT.

648 What remains is to bound that time. A node *gets split* if none of the three early stopping
 649 conditions apply (in those cases, the call takes constant time and does not recurse). If s
 650 nodes get split, the partitioning procedure visits $2s + 1$ nodes and takes $O(1 + s)$ time.

651 Singleton invocations take $O(1)$ time each, so we consider only invocations with at least
 652 two items. Recall that, for our fixed seed, the recursion tree T_0 represents the execution of
 653 Quicksort that SKELETONSORT simulates. Let r be the node of T_0 corresponding to the
 654 current invocation SKELETONSORT(S), and let $h(r)$ denote its height in T_0 . Color a node
 655 of S red if its raw height is at least $h(r)$, and blue otherwise. Since $h(r) \geq 1$, all leaves
 656 are blue. If present, the red nodes form a connected subtree containing the root of S , with
 657 maximal blue subtrees hanging below it. If there are no red nodes, the entire tree S is the
 658 sole maximal blue subtree. If there are b maximal blue subtrees, there are $b - 1$ red nodes,
 659 as every red node has two children.

660 The following analysis is very similar to running SPROUTSORT, and specifically, The-
 661 orem 11, in reverse. It might be useful to think of the red nodes as the root list of a sprout

forest, and each maximal blue subtree as a sprout tree. (This analogy only works for the analysis. We cannot run SKELETONSORT with sprout trees as, due to the random pivot selection, we do not know the height $h(r)$ of the current invocation during the algorithm.)

We charge the work of SKELETON-SPLIT to leaf indices of S . Define the *leader* of a subtree of S to be its leftmost leaf. For each of the b maximal blue subtrees, charge its leader once. These b charges cover the constant overhead and the cost of splitting all red nodes. Next, consider a blue node u that gets split by SKELETON-SPLIT. At least two of its three output trees are nonempty. The leader of u remains the leader of one of these output trees. We charge the work at u to the leader of another nonempty output tree. Thus, every blue split is charged to an index that becomes a new leader.

We first consider charges to index 1. It receives at most one red charge per non-empty invocation of SKELETONSORT, and therefore $O(n)$ red charges in total. It receives no blue charges, since whenever its leaf is present, it is already the leader of its subtree.

To bound the remaining charges, fix $i \geq 2$ and consider red charges to i . If the current height $h(r)$ exceeds both $\text{dist}_T(i, f(i))$ and $\text{dist}_R(i, f(i))/2$, then the current input S contains both indices i and $f(i)$, and their common ancestor in R is blue. They therefore lie in the same maximal blue subtree. Since $f(i) < i$, index i cannot be its leader and receives no red charge. The heights decrease strictly along the recursive path containing i , and i receives at most one red charge per invocation. Hence i receives $O(1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i)))$ red charges.

Next consider a blue split charged to i , and let u be the corresponding original node in R , of raw height k . If the leaf $f(i)$ is not a descendant of u in R , then the lowest common ancestor of i and $f(i)$ is a strict ancestor of u , so $k < \text{dist}_R(i, f(i))/2$. Otherwise, consider the output subtree whose new leader is i . Since $f(i) < i$, this subtree cannot contain the leaf $f(i)$. Thus, the current or an earlier Quicksort pivot must have separated $X[i]$ and $X[f(i)]$. Their lowest common ancestor in T_0 is therefore the current invocation or one of its ancestors, so $h(r) \leq \text{dist}_T(i, f(i))$. Since the charged node is blue, $k < h(r)$. In either case, $k < \max\{\text{dist}_R(i, f(i))/2, \text{dist}_T(i, f(i))\}$.

Once a blue split at an original node u is charged to i , the leaf i becomes the leftmost surviving leaf below u in its output tree. Every subsequent invocation containing i operates on a subset of these leaves, so i remains the leftmost surviving leaf below u . In other words, i is only charged when it becomes the leader of a node u , after which i remains the leader of that node as long as u remains blue. Thus, i receives at most one blue charge at each original node $u \in R$. Since there is at most one ancestor of i at each raw height, and all charged nodes have raw height $k < \max\{\text{dist}_R(i, f(i))/2, \text{dist}_T(i, f(i))\}$, index i receives $O(1 + \text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i)))$ blue charges in total.

Summing the red and blue charges over all indices, and adding the $O(n)$ initialization and output costs, proves the lemma. ◀

► **Lemma 23.** *Let X be a list of n distinct elements and let T be as above. For any distinct indices $i, j \in [n]$, we have $\mathbb{E}[\text{dist}_T(i, j)] \in O(1 + \log |\text{rank}_X(X[i]) - \text{rank}_X(X[j])|)$.*

Proof. We use the model of random binary search trees [30] where each rank $x \in [n]$ receives an independent uniform priority $p_x \in [0, 1]$, and the root of every subtree is the rank with the largest priority. This has the same distribution as the Quicksort recursion tree T_0 .

Let $a < b$ be the ranks of $X[i]$ and $X[j]$ in sorted order, and put $d = b - a$. Note that $d = |\text{rank}_X(X[i]) - \text{rank}_X(X[j])|$. Let k be the rank with the largest priority among $[a, b]$. Then the invocation with pivot k is the lowest common ancestor of $X[i]$ and $X[j]$ in T_0 .

Let B be the first rank greater than b whose priority exceeds p_k (or $n + 1$ if no such rank exists). Since k has the largest priority in $[a, b]$, the right subtree of k contains exactly the elements with rank $k + 1, \dots, B - 1$. Conditioned on k and B , this is a random binary search tree. The expected height of a random binary search tree with s keys is $\leq 5 \log(2 + s)$, see [13, Chapter 12.4]. Thus, the expected height of the right subtree, conditioned on k and B , is $\leq 5 \log(2 + B - a)$. For any $t \in [b, n]$, the event $B > t$ occurs precisely when the largest priority among $[a, t]$ belongs to $[a, b]$. Thus, $\Pr(B > t) = (d + 1)/(t - a + 1)$. In particular, for every integer $q \geq 0$, $\Pr(B - a > 2^q(d + 1)) \leq 2^{-q}$. Summing over these exponentially increasing ranges gives $\mathbb{E}[\log(2 + B - a)] \in O(1 + \log d)$. Hence the expected height of the right subtree is $O(1 + \log d)$. By a symmetric argument, the expected height of the left subtree is also $O(1 + \log d)$. The height of k is one larger than the maximum height of the two subtrees, thus $\mathbb{E}[h(k)] \in O(1 + \log d)$.

Recall that T is obtained from T_0 by extending edges until all leaves have the same depth. Thus, $\text{dist}_T(i, j) = 2h(k)$. Therefore, $\mathbb{E}[\text{dist}_T(i, j)] \in O(1 + \log d)$, which proves the lemma. $\blacktriangleleft$

► **Lemma 24.** *Let X be a list of n distinct elements and let f be any finger function. Then*

$$\sum_{i=2}^n \text{dist}_R(i, f(i)) \in O\left(n + \sum_{i=2}^n \log(i - f(i))\right),$$

$$\mathbb{E}\left[\sum_{i=2}^n \text{dist}_T(i, f(i))\right] \in O\left(n + \sum_{i=2}^n \log \max\{1, |\text{rank}_X(X[i]) - \text{rank}_X(X[f(i)])|\}\right).$$

Proof. The first bound follows directly from Lemma 14, since R is the complete binary tree whose leaves are the indices $1, \dots, n$ in-order. For the second bound, we apply linearity of expectation and Lemma 23. $\blacktriangleleft$

► **Theorem 25.** *For any list X of n distinct elements, SKELETONSORT sorts X in expected $O(n + \log \text{reg}(X))$ time.*

Proof. Let f be an arbitrary finger function. By Lemma 22, for every fixed random seed, the running time of SKELETONSORT is $O(n + \sum_{i=2}^n (\text{dist}_R(i, f(i)) + \text{dist}_T(i, f(i))))$.

Taking expectations and applying Lemma 24, the expected running time is

$$O\left(n + \sum_{i=2}^n \log(i - f(i)) + \sum_{i=2}^n \log \max\{1, |\text{rank}_X(X[i]) - \text{rank}_X(X[f(i)])|\}\right).$$

By the definition of reg , there exists a finger function f for which these two sums equal $\log \text{reg}(X)$. The theorem follows. $\blacktriangleleft$

B

 Notions of presortedness

■ **Table 1** Measures of presortedness and their target adaptive bounds. Throughout, $X : [n] \rightarrow [n]$ is surjective onto $[\max(X)]$, $\bar{X}$ is its stable rank permutation, and logarithms are base two. Zero-frequency entropy terms contribute zero. The notation $X = X_1 \sqcup \dots \sqcup X_k$ denotes a partition into disjoint, possibly interleaved subsequences. For brevity, increasing means non-decreasing, and decreasing means non-increasing.

Name (aliases)	Adaptive bound	Mathematical definition + intuition
Ham (Pos) [24]	$O(n + \text{Ham} \log(\text{Ham} + 1))$	$\text{Ham}(X) = \{i \in [n] \mid \bar{X}[i] \neq i\} $. <i>Number of elements outside their final positions.</i>
Exc [24]	$O(n + \text{Exc} \log(\text{Exc} + 1))$	$\text{Exc}(X) = n - c(X)$, where $c(X)$ is the number of cycles in $i \mapsto \bar{X}[i]$. <i>Minimum number of swaps to reach the stable sorted order; for distinct keys, the minimum number of swaps to sort X.</i>
Rem (Las) [12, 24]	$O(n + \text{Rem} \log(\text{Rem} + 1))$	$\text{Rem}(X) = \min\{ Y : Y \subseteq X \text{ and } X - Y \text{ is increasing}\} = n - \text{Las}(X)$. <i>Minimum number of deletions needed to leave X sorted.</i>
Max [24]	$O(n(1 + \log(\text{Max} + 1)))$	$\text{Max}(X) = \max_{i \in [n]} \bar{X}[i] - i $. <i>Maximum distance from an element's final position.</i>
Par (Dis, Radius) [2, 18]	$O(n(1 + \log(\text{Par} + 1)))$	$\text{Par}(X) = \max\{j - i : 1 \leq i < j \leq n, X[i] > X[j]\}$, with $\max \emptyset = 0$. <i>Maximum positional distance between an inverted pair.</i>
Inv [24]	$O(n + n \log(1 + \frac{\text{Inv}}{n}))$	$\text{Inv}(X) = \{(i, j) : 1 \leq i < j \leq n, X[i] > X[j]\} $. <i>Number of inverted pairs.</i>
DS (Spear) [17]	$O(n + n \log(1 + \frac{\text{DS}}{n}))$	$\text{DS}(X) = \sum_{i=1}^n \bar{X}[i] - i $. <i>Total distance from elements' final positions.</i>
HDS (HD) [4]	$O(n \text{HDS})$	$\text{HDS}(X) = \frac{1}{n} \sum_{i=1}^n \log(\bar{X}[i] - i + 2)$. <i>Average logarithmic distance from elements' final positions.</i>
nMulti [27]	$O(n \log(1 + \text{nMulti}))$	$\text{nMulti} = \max(X) = \{x \in X\} $. <i>Number of distinct values.</i>
HMulti [27]	$O(n(1 + \text{HMulti}))$	$\text{HMulti} = \sum_{i=1}^n \frac{m_i}{n} \log \frac{n}{m_i}$, where m_i is the number of occurrences of i in X . <i>Entropy of the value multiplicities.</i>
Block [9]	$O(n + \text{Block} \log \text{Block})$	$\text{Block}(X) = 1 + \{i \in [n-1] : \bar{X}[i+1] \neq \bar{X}[i] + 1\} $. <i>Number of maximal sorted blocks contiguous in both position and value. The measure nSRuns uses the same count but the bound $O(n(1 + \log \text{nSRuns}))$.</i>
HSRuns (HBlock) (H(vSRuns)) [6]	$O(n(1 + \text{HSRuns}))$	$\text{HSRuns}(X) = \text{HBlock}(X) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$, where $b_1, \dots, b_k$ are the lengths of the maximal increasing blocks that are contiguous in both index and value. <i>Entropy of lengths of maximal sorted blocks contiguous in both position and value.</i>
nRuns [21, 24]	$O(n(1 + \log \text{nRuns}))$	$\text{nRuns}(X) = 1 + \{i \in [n-1] : X[i+1] < X[i]\} $. <i>Number of maximal sorted contiguous pieces.</i>
HRuns (H(vRuns), runlength) [6, 25]	$O(n(1 + \text{HRuns}))$	$\text{HRuns}(X) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$, where $b_1, \dots, b_k$ are the lengths of the maximal increasing index-contiguous blocks. <i>Entropy of lengths of maximal sorted contiguous pieces.</i>

Continued on next page

XX:22 Adaptive sorting, also in the BST model

Name (aliases)	Adaptive bound	Mathematical definition + intuition
nRuns* (dual runs, riffles) [25]	$O(n(1 + \log \text{nRuns}^*))$	$\text{nRuns}^*(X) = 1 + \{v \in [n-1] : \overline{X}^{-1}[v+1] < \overline{X}^{-1}[v]\} $. For permutations, this equals $\text{nRuns}(X^{-1})$. <i>Number of maximal consecutive value ranges appearing in sorted order.</i>
HRuns* (dual runs, riffles) [19, 25]	$O(n(1 + \text{HRuns}^*))$	$\text{HRuns}^*(X) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$, where $b_1, \dots, b_k$ are the lengths of the maximal increasing value-contiguous subsequences. <i>Entropy of lengths of maximal consecutive value ranges appearing in sorted order.</i>
nSUS (SUS, Lds) [23]	$O(n(1 + \log(\text{nSUS})))$	$\text{nSUS}(X) = \min\{k : X = X_1 \sqcup \dots \sqcup X_k, X_r \text{ increasing}\}$. <i>Minimum number of increasing subsequences partitioning X.</i>
$H(\text{vSUS})$ (greedy SUS) [6]	$O(n(1 + H(\text{vSUS})))$	$H(\text{vSUS}(X)) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$, where $b_1, \dots, b_k$ are the lengths of the increasing subsequences obtained by greedily scanning X left to right and appending each element to the subsequence with the largest last element at most the new element, starting a new subsequence if none exists. <i>Entropy of subsequence lengths in the greedy increasing partition.</i>
$H(\text{vLRM})$ [5]	$O(n(1 + H(\text{vLRM})))$	$H(\text{vLRM}(X)) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$, where $b_1, \dots, b_k$ are the lengths of the subsequences obtained by repeatedly removing a longest root-to-leaf path from the remaining LRM forest, excluding the dummy root. <i>Entropy of subsequence lengths obtained by joining runs along the LRM tree.</i>
HSUS (chromatic entropy) [1, 6]	$O(n(1 + \text{HSUS}))$	$\text{HSUS}(X) = \min_{X=X_1 \sqcup \dots \sqcup X_k} \sum_{i=1}^k \frac{ X_i }{n} \log \frac{n}{ X_i }$, where each X_i is increasing. <i>Minimum entropy of subsequence lengths over increasing partitions.</i>
nSMS (SMS, Lads) [23, 33]	$O(n(1 + \log(\text{nSMS})))$	$\text{nSMS}(X) = \min\{k : X = X_1 \sqcup \dots \sqcup X_k, X_r \text{ increasing or decreasing}\}$. <i>Minimum number of monotone subsequences partitioning X.</i>
$H(\text{vSMS})$ [6]	$O(n(1 + H(\text{vSMS})))$, given the partition	$H(\text{vSMS}(X)) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$, where $b_i = X_i $ for a chosen partition $X = X_1 \sqcup \dots \sqcup X_k$ into the minimum number of monotone subsequences. <i>Entropy of subsequence lengths in a specific monotone partition.</i>
HSMS [6]	$O(n(1 + \text{HSMS}))$	$\text{HSMS}(X) = \min_{X=X_1 \sqcup \dots \sqcup X_k} \sum_{i=1}^k \frac{ X_i }{n} \log \frac{n}{ X_i }$, with each X_i increasing or decreasing. <i>Minimum entropy of subsequence lengths over monotone partitions.</i>
enc [31]	$O(n(1 + \log(\text{enc})))$	$\text{enc}(X) = k$, where k is the number of sorted lists built by scanning X left to right and adding each element at the beginning or end of the earliest-created list for which this preserves increasing order, creating a new list if none exists. <i>Number of sorted lists built greedily by insertion at either end.</i>
osc [22]	$O(n(1 + \log(1 + \frac{\text{osc}}{n})))$	$\text{osc}(X) = \sum_{i=1}^{n-1} (\max\{0, X[i+1] - X[i] - 1\})$. <i>Total number of distinct ranks strictly between consecutive values; the formula extends the permutation measure to repeated keys.</i>
Loc [20] (dynamic finger bound [10])	$O(n + \log(\text{Loc}))$	$\text{Loc}(X) = \prod_{i=2}^n \max\{1, \text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[i-1]) \}$. <i>Consecutive elements should be close in rank among elements seen so far.</i>

Continued on next page

Name (aliases)	Adaptive bound	Mathematical definition + intuition
Hist [26] (working-set property [32])	$O(n + \log(\text{Hist}))$	$\text{Hist}(X) = \prod_{i=2}^n \min \{i - j : j < i \wedge \text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j]) \leq 1\}.$ <i>Elements should have a recently inserted predecessor or successor.</i>
Reg [26, 28] (unified bound [3, 16])	$O(n + \log(\text{Reg}))$	$\text{Reg}(X) = \prod_{i=2}^n \min_{j < i} \{(i - j) \cdot \max \{1, \text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j]) \}\}.$ <i>Elements should be close to an earlier element in position and current rank.</i>
loc [25]	$O(n + \log(\text{loc}))$	$\text{loc}(X) = \prod_{i=2}^n \max \{1, X[i] - X[i-1] \}.$ <i>Consecutive elements should be close in final rank.</i>
hist [25]	$O(n + \log(\text{hist}))$	$\text{hist}(X) = \text{loc}(\overline{X}^{-1}) = \prod_{i=2}^n \overline{X}^{-1}[i] - \overline{X}^{-1}[i-1] ,$ <i>Elements of consecutive final ranks should be close in position.</i>
reg (this paper)	$O(n + \log(\text{reg}))$	$\text{reg}(X) = \prod_{i=2}^n \min_{j < i} \{(i - j) \max \{1, X[i] - X[j] \}\}.$ <i>Each new element should be close to an earlier element in position and final rank.</i>
unimst (this paper)	$O(n + \text{unimst})$	$\text{unimst}(X) = \text{weight of an MST with } w(i, j) = \log(2 + i - j + X[i] - X[j]).$ <i>Minimum spanning-tree weight for logarithmic distance in the index-rank plane.</i>

738

C The Hierarchy of Adaptive Measures

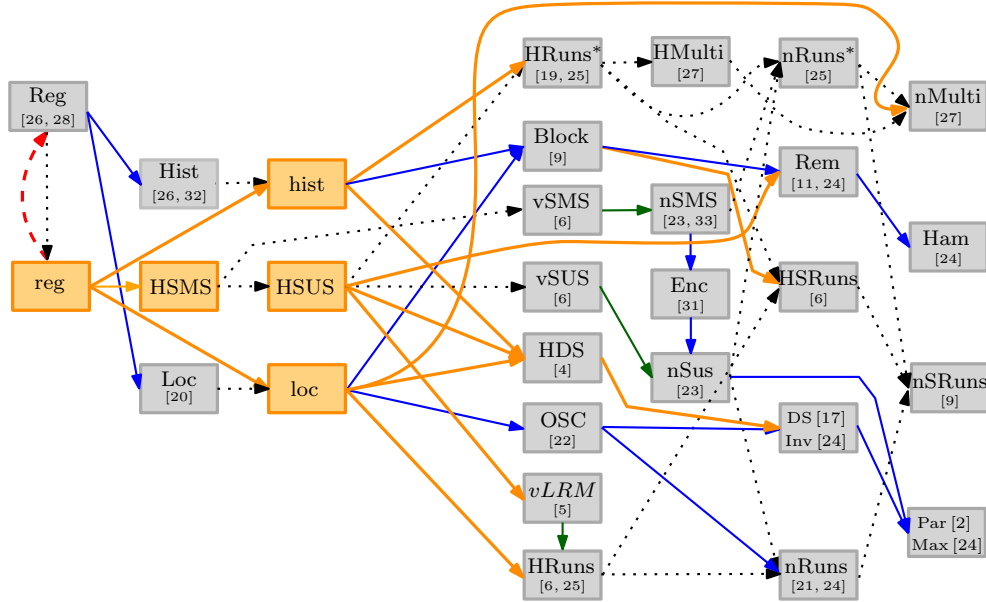


Figure 8 We recall Figure 1. Orange and red items are new. Red only holds if X has no duplicates. Blue arrows are attributed to [28] and green edges to [6].

739

740

741

In Section 2.3, we show that $\text{reg} \preceq \text{hist}$, $\text{reg} \preceq \text{loc}$ and $\text{reg} \preceq \text{HSMS}$. In this section, we show the remaining relations in Figure 1, as depicted in Figure 8. Some of these relations were already shown by Moffat and Petersson [26], or Belloch and Dobson [8]. While Moffat

and Petersson use Hist and Loc, their proofs also work for hist and loc.

Many relations in Figure 8 are straightforward from the definition. For example, nSUS considers partitions that are less restrictive than those of nRuns, which in turn is less restrictive than nSRuns; and all of these compute the same adaptive bound from a given permutation. Therefore $\text{nSUS} \preceq \text{nRuns} \preceq \text{nSRuns}$. Another example: HSUS minimises the entropy over a family of partitions that includes the one partition considered by $H(\text{vSUS})$. Therefore $\text{HSUS} \preceq H(\text{vSUS})$. Furthermore, as explained in the introduction, entropy-based measures are stronger than their counting variant, e.g. $\text{HRuns} \preceq \text{nRuns}$. (See also Lemma 26 (1) below.) We explicitly prove the remaining relations in Figure 8 in this section.

Inaccuracies in Barbay-Navarro Figure 1 in [6] claims the following relations, which seem to not hold: First they claim that $\text{Block} \cong \text{nSRuns}$. However, these notions use different target running times: $O(n + \text{Block} \log(1 + \text{Block}))$ vs $O(n \log(1 + \text{nSRuns}))$. Second, they claim that $H(\text{vSUS}) \preceq H(\text{vLRM})$, but Section 4 and Lemma 7 in [5] show that these are incomparable. Third, they claim that $H(\text{vSMS}) \preceq H(\text{vSUS})$. This holds if $H(\text{vSMS})$ denotes the minimum entropy over all SMS partitions, but need not hold if vSMS is an unspecified minimum-cardinality partition, which may have much larger entropy than their greedy vSUS partition. What is true, however, is $\text{HSMS} \preceq \text{HSUS}$.

C.1 Proving the remaining relations

We first prove a helper lemma on entropy functions.

► **Lemma 26.** *Let $b_1, \dots, b_k$ be positive integers with $b_1 + \dots + b_k = n$. Let $H(b_1, \dots, b_k) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i}$ be the entropy function. Then*

$$nH(b_1, \dots, b_k) \leq n \log k \quad (1)$$

$$k \log n \leq 2n + k \log k \in O(n(1 + H(b_1, \dots, b_k))) \quad (2)$$

$$nH(b_1, \dots, b_k) \leq n + 2 \sum_{i=1}^k b_i \log i \quad (3)$$

Proof. By concavity of log, we have

$$H(b_1, \dots, b_k) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i} \leq \log \left(\sum_{i=1}^k \frac{b_i}{n} \cdot \frac{n}{b_i} \right) = \log k.$$

This shows (1). Next, we show (2). By re-indexing, we may assume that $b_1 \geq b_2 \geq \dots \geq b_k$. Then $\frac{n}{b_i} \geq i$ since $n \geq b_1 + \dots + b_i \geq i \cdot b_i$. Also, $b_i \geq 1$ since b_i is positive. Thus,

$$nH(b_1, \dots, b_k) = \sum_{i=1}^k b_i \log \frac{n}{b_i} \geq \sum_{i=1}^k 1 \cdot \log i = \log \prod_{i=1}^k i = \log k! \in \Omega(k \log k).$$

In particular, $n + k \log k \in O(n(1 + H(b_1, \dots, b_k)))$. Finally, $\log \frac{n}{k} \leq \log e \cdot \frac{n}{k}$ and thus

$$k \log n = (k \log k + k \log \frac{n}{k}) \leq k \log k + k \log e \cdot \frac{n}{k} \leq k \log k + 2n.$$

This shows (2). Next, we show (3). The log-sum inequality for sequences $y_1, \dots, y_k > 0$ and $z_1, \dots, z_k > 0$ with $Y = y_1 + \dots + y_k$ and $Z = z_1 + \dots + z_k$ states that

$$\sum_{i=1}^k y_i \log \frac{1}{y_i} \leq \sum_{i=1}^k y_i \log \frac{1}{z_i} + Y \log \frac{Z}{Y}.$$

776 We apply this with $y_i := \frac{b_i}{n}$ and $z_i := \frac{1}{i^2}$, which yields $Y = 1$, $Z \leq \frac{\pi^2}{6}$, and

$$777 \quad H(b_1, \dots, b_k) = \sum_{i=1}^k \frac{b_i}{n} \log \frac{n}{b_i} \leq \sum_{i=1}^k \frac{b_i}{n} \log(i^2) + \log \frac{\pi^2}{6} \leq 2 \sum_{i=1}^k \frac{b_i}{n} \log i + 1.$$

778 This shows (3). ◀

779 Lemma 26 (1) immediately implies the “entropy” arrows in Figure 8.

780 ▶ **Lemma 27.** $\text{Block} \preceq \text{HSRuns}$.

781 **Proof.** Let $X = X_1 \sqcup \dots \sqcup X_k$ be the partition of X into maximal increasing sequences that
782 are both contiguous in index and value. By Lemma 26 (2),

$$783 \quad n + \text{Block} \log(1 + \text{Block}) = n + k \log(k+1) \in O(n(1 + H(|X_1|, \dots, |X_k|))) = O(n(1 + \text{HSRuns})).$$

784 This shows $\text{Block} \preceq \text{HSRuns}$. ◀

785 ▶ **Lemma 28.** $\text{HDS} \preceq \text{DS}$.

786 **Proof.** By concavity of $x \mapsto \log(2 + x)$, we have

$$787 \quad \text{HDS} = \sum_{i=1}^n \frac{\log(2 + |\bar{X}[i] - i|)}{n} \leq \log(2 + \frac{\sum_{i=1}^n |\bar{X}[i] - i|}{n}) = \log(2 + \frac{\text{DS}}{n}).$$

788 In particular, $n \text{HDS} \in O(n + n \log(1 + \frac{\text{DS}}{n}))$. This shows $\text{HDS} \preceq \text{DS}$. ◀

789 ▶ **Lemma 29.** $\text{loc} \preceq \text{HRuns}$ and $\text{hist} \preceq \text{HRuns}^*$.

790 **Proof.** Let $X = X_1 \sqcup \dots \sqcup X_k$ be the partition of X into maximal increasing sequences
791 that are contiguous in index. Put $s_i = |X_i|$. A singleton run contributes zero. For $s_i > 1$,
792 monotonicity gives $\sum_{j=2}^{s_i} \max\{1, X_i[j] - X_i[j-1]\} \leq s_i - 1 + n$. By concavity of $\log$,

$$793 \quad \sum_{j=2}^{s_i} \log \max\{1, |X_i[j] - X_i[j-1]|\} \leq (s_i - 1) \log \left(1 + \frac{n}{s_i - 1} \right) \leq s_i \log \frac{2n}{s_i}.$$

794 Due to the boundaries between the X_i , the sum $\log \text{loc}$ contains $k - 1$ additional terms that
795 are each at most $\log n$. Hence, by Lemma 26 (2), we have

$$796 \quad \log \text{loc} \leq k \log n + \sum_{i=1}^k s_i \log \frac{2n}{s_i} = k \log n + n + \sum_{i=1}^k s_i \log \frac{n}{s_i} \in O(n(1 + H(s_1, \dots, s_k))).$$

797 This shows that $\text{loc} \preceq \text{HRuns}$. For hist , observe that $\text{hist}(X) = \text{loc}(\bar{X}^{-1})$. On the other hand,
798 $\text{HRuns}^*(\bar{X}) \leq \text{HRuns}^*(X)$. Indeed, any value-contiguous subsequence of X contains, for
799 each value, either all copies or no copies of that value. Thus, a value-contiguous subsequence
800 of X is also a value-contiguous subsequence of $\bar{X}$. The partition of $\bar{X}$ into maximal such
801 subsequences therefore only merges parts, and entropy does not increase under merging.
802 Applying $\text{loc} \preceq \text{HRuns}$ to $\bar{X}^{-1}$ now shows that $\text{hist} \preceq \text{HRuns}^*$. ◀

803 ▶ **Lemma 30.** $\text{loc} \preceq \text{HDS}$ and $\text{hist} \preceq \text{HDS}$.

XX:26 Adaptive sorting, also in the BST model

Proof. For positive integers a, b, c , by the triangle inequality, we have

$$2 + |c - a| \leq 3 + |a - b| + |c - (b + 1)| \leq (2 + |a - b|) \cdot (2 + |c - (b + 1)|).$$

We use this with $a = \bar{X}[i - 1]$, $b = i - 1$, $c = \bar{X}[i]$, and use the fact that $|X[i] - X[i - 1]| \leq |\bar{X}[i] - \bar{X}[i - 1]|$ to get

$$\text{loc} \leq \prod_{i=2}^n (2 + |\bar{X}[i] - \bar{X}[i - 1]|) \leq \prod_{i=2}^n \left((2 + |\bar{X}[i - 1] - (i - 1)|) \cdot (2 + |\bar{X}[i] - i|) \right) \leq \prod_{i=1}^n (2 + |\bar{X}[i] - i|)^2.$$

Taking logarithms yields $\log \text{loc} \leq 2n \text{HDS}$, which shows $\text{loc} \preceq \text{HDS}$. Since $\text{hist}(X) = \text{loc}(\bar{X}^{-1})$ and $\text{HDS}(X) = \text{HDS}(\bar{X}) = \text{HDS}(\bar{X}^{-1})$, this also shows $\text{hist} \preceq \text{HDS}$. $\blacktriangleleft$

Lemma 31. $\text{HSUS} \preceq \text{HDS}$.

Proof. Apply the following construction to the permutation $P := \bar{X}$. Any increasing subsequence of P corresponds to a non-decreasing subsequence of X , and $\text{HDS}(P) = \text{HDS}(X)$. For each integer d , let Y_d be the subsequence consisting of entries $P[i]$ with $P[i] - i = d$. Each Y_d is increasing. Order these subsequences as $Y_0, Y_1, Y_{-1}, Y_2, Y_{-2}, \dots$ and discard empty parts, obtaining $X_1, \dots, X_k$. An entry $P[i] \in X_j$ then satisfies $j \leq 2(2 + |P[i] - i|)$. By Lemma 26 (3),

$$nH(|X_1|, \dots, |X_k|) \leq n + 2 \sum_{j=1}^k |X_j| \log j \leq n + 2 \sum_{i=1}^n \log(2(2 + |P[i] - i|)) \in O(n(1 + \text{HDS}(X))).$$

Transferring this partition to X proves $\text{HSUS} \preceq \text{HDS}$. $\blacktriangleleft$

Lemma 32. $\text{HSUS} \preceq \text{Rem}$.

Proof. Let $k = \text{Rem} + 1$. We partition $X = X_1 \sqcup \dots \sqcup X_k$ where $X_2, \dots, X_k$ are singletons whose removal leaves the sorted sequence X_1 . By Lemma 26 (2),

$$nH(|X_1|, \dots, |X_k|) = |X_1| \log \frac{n}{|X_1|} + \sum_{i=2}^k \log n \leq 2n + k \log n \leq 4n + k \log k.$$

Thus, $n(1 + \text{HSUS}) \in O(n + \text{Rem} \log(1 + \text{Rem}))$, which shows $\text{HSUS} \preceq \text{Rem}$. $\blacktriangleleft$

Lemma 33. $\text{Loc} \preceq \text{loc}$ and $\text{Reg} \preceq \text{reg}$.

Proof. We have $|\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[i - 1])| \leq |X[i] - X[i - 1]|$, and thus

$$\begin{aligned} \text{Loc}(X) &= \prod_{i=2}^n \max\{1, |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[i - 1])|\} \\ &\leq \prod_{i=2}^n \max\{1, |X[i] - X[i - 1]|\} = \text{loc}(X). \end{aligned}$$

This shows $\text{Loc} \preceq \text{loc}$. The same rank-distance argument immediately gives $\text{Reg} \preceq \text{reg}$. $\blacktriangleleft$

Lemma 34. $\text{loc} \preceq n\text{Multi}$.

Proof. We have

$$\text{loc}(X) = \prod_{i=2}^n \max\{1, |X[i] - X[i - 1]|\} \leq \max(X)^{n-1} \leq \max(X)^n.$$

Taking logarithms yields $n + \log \text{loc} \in O(n \log(1 + n\text{Multi}))$, which shows $\text{loc} \preceq n\text{Multi}$. $\blacktriangleleft$

834 ► **Lemma 35.** $\text{Hist} \preceq \text{hist}$.

835 **Proof.** If we replace X by $\overline{X}$, then Hist only increases, while hist is unaffected. Thus, it
 836 suffices to consider the cases where X is a permutation.

837 For $i \in [n]$, let $S_i := \{j \in [n] : j < i \wedge |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j])| \leq 1\}$. Our
 838 goal is to define a function $f : [n] \setminus \{1\} \rightarrow [n]$ with $f(i) \in S_i$ for all i and $\sum_{i=2}^n \log(i - f(i)) \in$
 839 $O(n + \log \text{hist}(X))$. Given such a function, picking $j = f(i)$ in the definition of Hist shows
 840 $\text{Hist} \preceq \text{hist}$.

841 For $t \in [n]$, the t -front consists of all $i \in [n]$ with $X[i] \leq t$ such that there exists no $j < i$
 842 with $X[j] \leq X[i] \leq t$.

843 ► **Claim 36.** Let $t \in [n]$ be arbitrary. Put $I_t = X^{-1}([1..t])$ and $r_t = \min(I_t)$. There exists a
 844 function $f : I_t \setminus \{r_t\} \rightarrow I_t$ with $f(i) \in S_i$ for all i with the sum property

$$845 \quad \sum_{i \in I_t \setminus \{r_t\}} \log(i - f(i)) \leq \sum_{y=2}^t \log |X^{-1}[y] - X^{-1}[y-1]|,$$

846 and the following additional property: If $i_1 < \dots < i_k$ are the indices of the t -front, then
 847 $f(i_j) = i_{j-1}$ for all $j \geq 2$.

848 **Proof of the Claim.** We do induction over t . The base case $t = 1$ is straightforward as
 849 f is the trivial function and both sums in the sum property are empty. Let $t \geq 2$ be
 850 arbitrary. Suppose the claim holds for $t-1$ and yields a function f_{t-1} . We construct
 851 f_t by slightly modifying f_{t-1} , keeping all choices unchanged unless stated otherwise. Let
 852 $i_1, \dots, i_k$ be the $(t-1)$ -front. Put $i = X^{-1}[t]$. If $i_k < i$, then the t -front is $i_1, \dots, i_k, i$. Put
 853 $f_t(i) = i_k$. The sum property is maintained since $i_k = X^{-1}[t-1]$ and $i = X^{-1}[t]$, meaning
 854 $\log(i - f_t(i)) = \log |X^{-1}[t] - X^{-1}[t-1]|$.

855 Otherwise, suppose that $i_j < i < i_{j+1}$ for some index j . We put $f_t(i) = i_j$ and $f_t(i_{j+1}) = i$,
 856 and keep $f_t(i') = f_{t-1}(i')$ for all other arguments i' . The t -front is $i_1, \dots, i_j, i$, so f_t satisfies
 857 the additional property. We have

$$858 \quad i - f_t(i) = i - i_j \leq i_{j+1} - i_j = i_{j+1} - f_{t-1}(i_{j+1})$$

859 and

$$860 \quad i_{j+1} - f_t(i_{j+1}) = i_{j+1} - i \leq i_k - i = X^{-1}[t-1] - X^{-1}[t].$$

861 Therefore, the sum property for $(t-1)$ implies the sum-property for t :

$$\begin{aligned} 862 \quad & \sum_{i' \in I_t \setminus \{r_t\}} \log(i' - f_t(i')) \\ 863 \quad &= \log(i - f_t(i)) + \log(i_{j+1} - f_t(i_{j+1})) - \log(i_{j+1} - f_{t-1}(i_{j+1})) + \sum_{i' \in I_{t-1} \setminus \{r_{t-1}\}} \log(i' - f_{t-1}(i')) \\ 864 \quad &\leq \log |X^{-1}[t-1] - X^{-1}[t]| + \sum_{y=2}^{t-1} \log |X^{-1}[y] - X^{-1}[y-1]| \\ 865 \quad &= \sum_{y=2}^t \log |X^{-1}[y] - X^{-1}[y-1]|. \end{aligned}$$

866 Finally, if $i < i_1$, then $i_1 = r_{t-1}$ and $i = r_t$. We put $f_t(i_1) = i$. The sum property follows
 867 since

$$868 \quad \log(i_1 - i) \leq \log(i_k - i) = \log |X^{-1}[t-1] - X^{-1}[t]|.$$

869 In all three cases, the choices belong to the corresponding sets S_i : the elements of the
870 $(t-1)$ -front are prefix maxima among the values $1, \dots, t-1$, and t is larger than all these
871 values. ◀

872 The claim with $t = n$ immediately yields the desired function f . This shows $\text{Hist} \preceq \text{hist}$. ◀

873 C.2 Reg equals reg for permutations

874 Lemma 33 shows $\text{Reg} \preceq \text{reg}$. In Section 2.1 we show that, if X contains duplicates, then
875 $\text{reg} \not\preceq \text{Reg}$. However, if X is a permutation, then we can show that $\text{reg} \preceq \text{Reg}$.

876 ▶ **Theorem 9.** *If X is a list without duplicates, then $\log \text{reg} \in O(n + \log \text{Reg})$.*

877 ▶ **Lemma 37.** *Let X be a permutation. For $i \geq 2$, put*

$$\begin{aligned} 878 \quad d(i, j) &= |X[i] - X[j]| \\ 879 \quad D(i, j) &= |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..j]}(X[j])| \\ 880 \quad r(i) &:= \min_{j < i} \max\{(i-j), d(i, j)\} \\ 881 \quad R(i) &:= \min_{j < i} \max\{(i-j), D(i, j)\}. \end{aligned}$$

882 For every integer $t \geq 1$, we have

$$883 \quad |\{i : r(i) \geq t^3\}| \leq 5|\{i : R(i) \geq t\}| + O\left(\frac{n}{t}\right).$$

884 **Proof of Theorem 9 from Lemma 37.** We apply Lemma 37 for $t = 2^j$ where $0 \leq j \leq$
885 $\lfloor \log n \rfloor$. This yields

$$886 \quad \underbrace{\sum_{j=0}^{\lfloor \log n \rfloor} |\{i : r(i) \geq 2^{3j}\}|}_{=:S} \leq 5 \underbrace{\sum_{j=0}^{\lfloor \log n \rfloor} |\{i : R(i) \geq 2^j\}|}_{=:T} + O(n).$$

887 The contribution of an index i to S is at least $\log(r(i))/3 - 1$, and the contribution to T is
888 at most $\log(R(i)) + 1$. Therefore, this shows

$$889 \quad \sum_{i=2}^n \log r(i) \leq 3S + O(n) \leq 15T + O(n) \leq 30 \sum_{i=2}^n \log R(i) + O(n).$$

890 We have $\log \text{reg} \leq 2 \sum_{i=2}^n \log r(i)$. We also have $\log \text{Reg} \geq \sum_{i=2}^n \log R(i)$. Indeed, for $j < i$,
891 put $a = i - j$ and $b = |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..j]}(X[j])|$. Since X is a permutation, $b \geq 1$.
892 The rank of $X[j]$ can increase by at most a between steps j and i . If $X[i] > X[j]$, at most
893 $a - 1$ of these elements are smaller than $X[j]$, so $D(i, j) \leq b + a - 1$. If $X[i] < X[j]$, the rank
894 of $X[j]$ increases by at least one, and again $D(i, j) \leq b + a - 1$. Thus, $D(i, j) \leq a + b - 1 \leq ab$.
895 In particular, $R(i) \leq ab$ for every $j < i$, proving the claimed inequality. Therefore, $\log \text{reg} \leq$
896 $60 \log \text{Reg} + O(n)$. ◀

897 **Proof of Lemma 37.** The claim is immediate for $t = 1$ or $t^3 > n$, so assume $2 \leq t \leq n^{1/3}$.
898 Put $F(i) := \arg \min_{j < i} \max\{(i-j), D(i, j)\}$. We partition the value space $[n]$ into *buckets* of
899 size t^3 . The last bucket may be smaller. The item $X[i]$ goes into the bucket $\lfloor (X[i] - 1)/t^3 \rfloor$.
900 We say an index i is *early* if $X[i]$ is among the first $4t^2$ items inserted in its bucket. Observe
901 that the total number of early items is $O(\frac{n}{t^3} \cdot 4t^2) = O(\frac{n}{t})$.

Additionally, we partition the index space into *epochs* of size t^2 . The last epoch may be smaller. The item $X[i]$ goes into the epoch $\lfloor (i-1)/t^2 \rfloor$. We say a bucket is *heavy* at the start of an epoch if it contains at least $2t^2$ previously inserted items, and *light* otherwise.

Our proof is a charging argument where we assign one charge for every item i with $r(i) \geq t^3$. We are allowed to assign these charges as follows: At most 5 charges to every item i with $R(i) \geq t$, at most 5 charges to every early item, and at most 5 charges to every bucket-epoch intersection that contains at least $t/3$ items. This budget guarantees that the total number of charges is at most $5|\{i : R(i) \geq t\}| + O(\frac{n}{t})$, which shows the lemma.

Let $i \in [n]$ be an index with $r(i) \geq t^3$. If i is early, then we charge i to itself. Otherwise, let E be the epoch of i and $E-1$ the previous epoch. Since i is not early, at least $4t^2$ items were inserted into its bucket B before i . At most $2t^2$ of these were inserted during epochs $E-1$ and E , so B is $(E-1)$ -heavy. If $R(i) \geq t$, then we charge i to itself. Otherwise, consider the *finger path* $j_1 = F(i), j_2 = F(j_1), \dots$ until we hit the first item j_a that is early, has $R(j_a) \geq t$, or until we reach j_a for $a = t$. This finger path is very well-behaved: We have $i > j_1 > \dots > j_a$, and also $j_a > i - t^2$ since $R(i), R(j_1), \dots, R(j_{a-1}) < t$, and $a \leq t$. In particular, the finger path lies within the epochs E and $(E-1)$, and, since $r(i) \geq t^3$, it is disjoint from bucket B .

Also, the finger path can never cross over an $(E-1)$ -heavy bucket. To see this, put $\rho(j) := \text{rank}_{X[1..j]}(X[j])$. For any two vertices p, q on the finger path, including i , we have

$$|p - q| < t^2 \quad \text{and} \quad |\rho(p) - \rho(q)| < t^2.$$

Suppose that $X[p]$ and $X[q]$ lie on opposite sides of an $(E-1)$ -heavy bucket C . At least $2t^2$ elements of C were inserted before epoch $E-1$. Since the insertion ranks can change by at most $|p - q|$ between steps p and q , we have $|\rho(p) - \rho(q)| \geq 2t^2 - |p - q| \geq t^2$, a contradiction.

Since $j_1, \dots, j_{a-1}$ are not early, their buckets are all $(E-1)$ -heavy. Thus, if $a > 1$, the barrier argument shows that $j_1, \dots, j_{a-1}$ lie in the same $(E-1)$ -heavy bucket B' adjacent to B among the $(E-1)$ -heavy buckets, and j_a lies in B' or in an $(E-1)$ -light bucket between B and B' . If $a = 1$, then j_a lies in a neighbouring $(E-1)$ -heavy bucket or in a light bucket between B and the next heavy bucket.

If $a < t$, then we charge i to j_a . Otherwise, one of the pairs (B', E) and $(B', E-1)$ contains at least half of $j_1, \dots, j_{t-1}$, and thus at least $t/3$ items, so we charge i to that pair.

To analyse the number of charges assigned to each item or pair, observe that every bucket-epoch pair contains at most one index $i \in [n]$ with $r(i) \geq t^3$. Indeed, two items in the same bucket and epoch have both index distance less than t^2 and value distance less than t^3 .

Consider an item j that receives a charge as the endpoint of a finger path. If j lies in epoch K , the source item i lies in epoch K or $K+1$. For each of these two possibilities, the source bucket B must be one of the two $(E-1)$ -heavy buckets neighbouring the bucket of j , or the light interval containing it. Since each bucket-epoch pair contains at most one source item, j receives at most four such charges, plus possibly one charge from itself.

Similarly, consider a bucket-epoch pair (B', K) that receives a charge. The source item lies in epoch K or $K+1$, and its bucket is one of the two neighbouring heavy buckets of B' at the start of epoch $E-1$. Thus, this pair receives at most four charges.

Finally, there are $O(n/t)$ early items and $O(n/t)$ bucket-epoch pairs containing at least $t/3$ items. Therefore, the total number of charges is at most $5|\{i : R(i) \geq t\}| + O(n/t)$, proving the lemma. $\blacktriangleleft$

D

 Loc is the Dynamic Finger Property, and Hist is the Working Set

In Section 2, we showed that insertion sort using a dictionary with the unified bound is Reg-adaptive. In this section, we similarly relate Loc and Hist to the dynamic-finger and working-set properties of binary search trees [10, 32].

Recall that a binary search tree stores a sorted set Y subject to an $\text{ACCESS}(y)$ operation that forces it to navigate from the root to some item $y \in Y$. A BST has the *working-set property* if $\text{ACCESS}(y)$ takes amortised $O(\log(2 + \text{age}(y)))$ time, where $\text{age}(y)$ counts the distinct keys accessed since the previous access to y . For a key not previously accessed, use $\text{age}(y) = |Y|$. Counting all intervening access operations gives an upper bound on this age, which suffices in the proofs below. Splay trees have the working-set property [32]. A BST has the *dynamic-finger property* [10] if $\text{ACCESS}(y)$ takes amortised $O(\log(2 + d(y', y)))$ time, where y' is the previously accessed item and $d(y', y)$ is the number of items in Y that lie between y and y' . Cole [10], building on joint work with Mishra, Schmidt, and Siegel [11], proves the dynamic-finger property for splay trees.

There are different algorithms for inserting an item y' into a binary search tree. For this section, we allow any insertion algorithm that accesses both the predecessor and successor of y' in Y , adds a new node y' , and then accesses y' . To perform insertion sort, we start with a binary search tree that contains two dummy elements $-\infty, \infty$, and then insert $X[1], \dots, X[n]$ in this order.

► **Theorem 38.** *Insertion sort with a binary search tree that has the dynamic finger property is Loc-adaptive.*

Proof. Inserting $X[1]$ takes constant time. For $i \geq 2$, when inserting $y = X[i]$, the last accessed item was $y' = X[i-1]$. Let z be the predecessor or successor of y accessed first. Then $d(y', z) \leq |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[i-1])| + 1$, while the remaining accesses during insertion take constant amortised time. Thus, the total running time is

$$\begin{aligned} & O\left(\sum_{i=2}^n (2 + \log(3 + |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[i-1])|))\right) \\ &= O\left(n + \sum_{i=2}^n \log(\max\{1, |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[i-1])|\})\right) = O(n + \log \text{Loc}). \blacktriangleleft \end{aligned}$$

► **Theorem 39.** *Insertion sort with a binary search tree that has the working set property is Hist-adaptive.*

Proof. Inserting $X[1]$ takes constant time. For $i \geq 2$, when inserting $y = X[i]$ into Y , let $j = \arg \min \{i - j : j < i \wedge |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j])| \leq 1\}$. Then $y' := X[j]$ is the predecessor or successor of $X[i]$ in Y , and the other predecessor or successor of $X[i]$ is an element y'' that was inserted before $X[j]$. Hence, the insertion of y' accessed both y' and y'' . Since every insertion accesses three nodes, this shows $\text{age}(y'), \text{age}(y'') \leq 3(i - j)$, so that accessing y'' and y' during the insertion of y takes $O(1 + \log(i - j))$ time. We assume that accessing the node $X[i]$ that was just created is free. Then, the total running time is

$$O\left(n + \sum_{i=2}^n \log \min\{i - j : j < i \wedge |\text{rank}_{X[1..i]}(X[i]) - \text{rank}_{X[1..i]}(X[j])| \leq 1\}\right) = O(n + \log \text{Hist}(X)). \blacktriangleleft$$

E Synergistic Sorting

Barbay, Ochoa and Satti [7] introduce two sorting algorithms DLMSORT and QUICKSYNERGY SORT that combine many adaptive properties. We show that their notion of adaptivity is stronger than nRuns, Block, and HMulti, but weaker than loc if the input has no duplicates. Moreover, we show that their notion is incomparable to Reg, as it is weaker than loc when the input has no duplicates, but there does exist families of inputs on which their algorithm is better. To this end, we define the measure Syn that describes their algorithm.

► **Definition 40.** Let $X_1, \dots, X_k$ be the partition of X into maximal contiguous non-decreasing sequences. We say a value $x \in [n]$ is free if, for every index j with $X[j] = x$, we have $\overline{X}[j] = \overline{X}[j-1] + 1$ and $\overline{X}[j] = \overline{X}[j+1] - 1$. Note that, if x is free, then x occurs in at most one X_i . For every non-free value x , let m_x be the number of X_i that contain at least one copy of x . Let $\text{Syn} = \prod_{x: \text{non-free}} \binom{k}{m_x}$ and $f_{\text{Syn}}(X) = n + \log \text{Syn}(X)$.

► **Theorem 41** (Corollary 1 and Theorem 5 in [7]). DLMSORT and QUICKSYNERGY SORT are Syn-adaptive.

► **Lemma 42.** $\text{Syn} \preceq \text{nRuns}$, $\text{Syn} \preceq \text{HMulti}$ and $\text{Syn} \preceq \text{Block}$.

Proof. We use the notation of Definition 40. Since $\binom{k}{i} \leq (ek/i)^i$, we have

$$\log \text{Syn} = \sum_{x: \text{non-free}} \log \binom{k}{m_x} \leq \sum_{x: \text{non-free}} m_x \log \frac{ek}{m_x} \leq \sum_{x: \text{non-free}} m_x \log(ek) \leq n \log(ek).$$

Since $k = \text{nRuns}$, this shows $n + \log \text{Syn}(X) \in O(n + n \log \text{nRuns})$, i.e. $\text{Syn} \preceq \text{nRuns}$.

For every value $x \in [n]$, let m'_x denote the number of occurrences of x in X . Since $y \mapsto y \log \frac{en}{y}$ is an increasing function for $y \in [0, n]$, we have

$$\log \text{Syn} \leq \sum_{x: \text{non-free}} m_x \log \frac{ek}{m_x} \leq \sum_{x: \text{non-free}} m_x \log \frac{en}{m_x} \leq \sum_{x \in [n]} m'_x \log \frac{en}{m'_x} \leq n \log e + n \text{HMulti},$$

with the convention “ $0 \log \frac{en}{0} = 0$ ”. This shows $\text{Syn} \preceq \text{HMulti}$.

Finally, observe that $\sum_{x: \text{non-free}} m_x \leq 2 \text{Block}$ and $k \leq \text{Block}$. Hence

$$\log \text{Syn} \leq \sum_{x: \text{non-free}} m_x \log \frac{ek}{m_x} \leq \sum_{x: \text{non-free}} m_x \log(ek) \leq n \log e + 2 \text{Block} \log \text{Block},$$

which shows $\text{Syn} \preceq \text{Block}$. ◀

► **Lemma 43.** If X contains no duplicates, then $\text{loc} \preceq \text{Syn}$

If X contains duplicates, then $\text{loc} \not\preceq \text{Syn}$ since $\text{Syn} \preceq \text{HMulti}$ but $\text{loc} \not\preceq \text{HMulti}$.

Proof. We use the notation of Definition 40. If X contains no duplicates, then $m_x = 1$ for every non-free value x . Thus, if n' denotes the number of non-free values, then

$$\log \text{Syn} = \sum_{x: \text{non-free}} \log \binom{k}{m_x} = n' \log k.$$

Consider a single maximal contiguous non-decreasing subsequence X_i and suppose it is of the form $X[a_i], X[a_i + 1], \dots, X[b_i]$. Let n'_i denote the number of non-free values in X_i . For

XX:32 Adaptive sorting, also in the BST model

1015 all $x, y > 0$, we have $\log(x) = \log \frac{x}{y} + \log(y) \leq \log \frac{x}{y} + y$. Therefore,

$$1016 \quad \sum_{j=a_i+1}^{b_i} \log(X[j] - X[j-1]) = \sum_{\substack{j=a_i+1 \\ X[j] \text{ non-free}}}^{b_i} \log(X[j] - X[j-1]) \leq \sum_{\substack{j=a_i+1 \\ X[j] \text{ non-free}}}^{b_i} (\log k + \frac{X[j] - X[j-1]}{k}) \leq n'_i \log k + \frac{n}{k}.$$

1017 Also, $\log |X[a_i] - X[a_i - 1]| \leq \log n$. Therefore, summing over all X_i yields

$$1018 \quad \log \text{loc} = \sum_{j=2}^n \log |X[j] - X[j-1]| \leq k \log n + \sum_{i=1}^k (n'_i \log k + \frac{n}{k}) = k \log n + n' \log k + n.$$

1019 We have $k \leq n'$ since $X[a_i]$ is non-free for every i . By Lemma 26 (2), $k \log n \leq 2n + k \log k$.
1020 Therefore

$$1021 \quad \log \text{loc} \leq k \log n + n' \log k + n \leq 3n + k \log k + n' \log k \leq 3n + 2n' \log k \in O(n + \log \text{Syn}).$$

1022 This shows $\text{loc} \preceq \text{Syn}$. ◀

1023 ▶ **Lemma 44.** *For inputs with duplicates, $\text{Reg} \not\preceq \text{Syn}$.*

1024 **Proof.** Let $k \geq 3$, put $S = (1, k+1, 2k+1, \dots, (k-1)k+1)$, and consider $X = (1, 2, \dots, k^2, \underbrace{S, \dots, S}_{k \text{ copies}})$.

1025 The input has length $N = 2k^2$ and exactly $k+1$ maximal contiguous non-decreasing runs.

1026 Every value in S occurs in all $k+1$ runs, so its contribution to Syn is $\binom{k+1}{k+1} = 1$. Every
1027 other value occurs exactly once, in the initial run. Such a value is free unless it immediately
1028 follows a value of S , or is the final value k^2 of the initial run. Indeed, all other singleton values
1029 have consecutive stable ranks with both neighbours in X . Thus, at most $k+1$ singleton
1030 values are non-free, each contributing $\binom{k+1}{1} = k+1$. Consequently,

$$1031 \quad \log \text{Syn}(X) \leq (k+1) \log(k+1), \quad N + \log \text{Syn}(X) \in \Theta(N).$$

1032 Now consider an occurrence $X[i]$ in one of the last $k-1$ copies of S , and any earlier
1033 finger $X[j]$. If $X[j]$ belongs to the initial run, then $i-j \geq k$. Otherwise, both values belong
1034 to S . If they are equal, their occurrences are at least k positions apart. If they differ, their
1035 prefix-rank distance is at least k , since the initial run already contains every value in $[k^2]$.
1036 Therefore every finger choice contributes a factor at least k to the corresponding factor of
1037 $\text{Reg}(X)$. There are $k(k-1)$ such occurrences, so

$$1038 \quad \log \text{Reg}(X) \geq k(k-1) \log k \in \Omega(N \log N).$$

1039 Hence $N + \log \text{Reg}(X)$ is not $O(N + \log \text{Syn}(X))$, proving $\text{Reg} \not\preceq \text{Syn}$. ◀

1040 F Open Problems

1041 We have not fully considered the adaptive heap properties from [29], and conjecture that they
1042 realise as a sorting algorithm an equally strong, dual variant of Reg . Since our paper has
1043 not developed the tools for dualisation under inputs with duplicates, and $O(n + \log \text{reg}) =$
1044 $O(n + \log \text{Reg})$ when there are no duplicates, we left this question open.

HSMS and HSUS-adaptive sorting

In this paper, we have described adaptive algorithms for most of the strongest measures in Figure 8: insertion sort variants for Reg, Hist, and Loc, and `SkeletonSort` for reg. While a reg-adaptive sorting algorithm is automatically HSMS-adaptive and HSUS-adaptive, we are not aware of any HSMS-adaptive or HSUS-adaptive sorting algorithm that does not go via reg (or Reg). Since HSMS, HSUS are considerably weaker notions than reg, it should be easier to design an adaptive algorithm for them, but we were unable to do so.

References

- 1 Noga Alon and Alon Orlitsky. Source coding and graph entropies. *IEEE Transactions on Information Theory*, 42(5):1329–1339, 1996. doi:10.1109/18.532875.
- 2 Tom Altman and Yoshihide Igarashi. Roughly sorting: Sequential and parallel approach. *Journal of Information Processing*, 12(2):154–158, 1989.
- 3 Mihai Bădoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007. doi:10.1016/j.tcs.2007.03.002.
- 4 Xingjian Bai and Christian Coester. Sorting with predictions. In *Advances in Neural Information Processing Systems*, volume 36, pages 26563–26584, 2023. URL: https://proceedings.neurips.cc/paper_files/paper/2023/hash/544696ef4847c903376ed6ec58f3a703-Abstract-Conference.html.
- 5 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.
- 6 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013. doi:10.1016/j.tcs.2013.10.019.
- 7 Jérémy Barbay, Carlos Ochoa, and Srinivasa Rao Satti. Synergistic solutions on multisets. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics*, pages 31:1–31:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.31.
- 8 Guy E. Blelloch and Magdalen Dobson. The geometry of tree-based sorting. In *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, volume 261 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ICALP.2023.26.
- 9 Svante Carlsson, Christos Levkopoulou, and Ola Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, 1993. Preliminary version appeared in ISAAC 1990. doi:10.1007/BF01190160.
- 10 Richard Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000. doi:10.1137/S009753979732699X.
- 11 Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting log n -block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000. doi:10.1137/S0097539797326988.
- 12 Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, 1980. doi:10.1145/359024.359026.
- 13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.
- 14 Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătraşcu. The geometry of binary search trees. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 496–505, 2009. doi:10.1137/1.9781611973068.55.
- 15 Erik D. Demaine, John Iacono, Stefan Langerman, and Özgür Özkan. Combining binary search trees. In *Automata, Languages, and Programming (ICALP 2013)*, pages 388–399, 2013. doi:10.1007/978-3-642-39206-1_33.

- 1095 16 Jonathan C. Derryberry and Daniel D. Sleator. Skip-splay: Toward achieving the unified bound
1096 in the bst model. In *Algorithms and Data Structures (WADS 2009)*, volume 5664 of *Lecture*
1097 *Notes in Computer Science*, pages 194–205, 2009. doi:10.1007/978-3-642-03367-4_18.
- 1098 17 Persi Diaconis and Ronald L. Graham. Spearman’s footrule as a measure of disarray. *Journal*
1099 *of the Royal Statistical Society. Series B (Methodological)*, 39(2):262–268, 1977. doi:10.1111/
1100 j.2517-6161.1977.tb01624.x.
- 1101 18 Vladimir Estivill-Castro and Derick Wood. A new measure of presortedness. *Information and*
1102 *Computation*, 83(1):111–119, 1989. doi:10.1016/0890-5401(89)90050-3.
- 1103 19 Elahe Ghasemi, Vincent Jugé, and Ghazal Khalighinejad. Galloping in fast-growth natural
1104 merge sorts. In *49th International Colloquium on Automata, Languages, and Programming*
1105 *(ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages
1106 68:1–68:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.
1107 ICALP.2022.68.
- 1108 20 Jyrki Katajainen, Christos Levkopoulos, and Ola Petersson. Local insertion sort revisited.
1109 In *Optimal Algorithms*, volume 401 of *Lecture Notes in Computer Science*, pages 239–253.
1110 Springer, 1989. doi:10.1007/3-540-51859-2_19.
- 1111 21 Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*.
1112 Addison-Wesley, Reading, MA, 2 edition, 1998.
- 1113 22 Christos Levkopoulos and Ola Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3):395–
1114 413, 1993. doi:10.1006/jagm.1993.1021.
- 1115 23 Christos Levkopoulos and Ola Petersson. Sorting shuffled monotone sequences. *Information*
1116 *and Computation*, 112(1):37–50, 1994. doi:10.1006/inco.1994.1050.
- 1117 24 Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions*
1118 *on Computers*, C-34(4):318–325, 1985. doi:10.1109/TC.1985.5009382.
- 1119 25 Peter McIlroy. Optimistic sorting and information theoretic complexity. In *Proceedings of*
1120 *the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 467–474.
1121 Society for Industrial and Applied Mathematics, 1993. URL: [https://dl.acm.org/doi/10.](https://dl.acm.org/doi/10.5555/313559.313859)
1122 [5555/313559.313859](https://dl.acm.org/doi/10.5555/313559.313859).
- 1123 26 Alistair Moffat and Ola Petersson. Historical searching and sorting. In *Algorithms: Second*
1124 *International Symposium, ISA ’91*, volume 557 of *Lecture Notes in Computer Science*, pages
1125 263–272. Springer, 1991. doi:10.1007/3-540-54945-5_70.
- 1126 27 J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM Journal on*
1127 *Computing*, 5(1):1–8, 1976. doi:10.1137/0205001.
- 1128 28 Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied*
1129 *Mathematics*, 59(2):153–179, 1995. doi:10.1016/0166-218X(93)E0160-Z.
- 1130 29 Daniel Rutschmann. Sorting under partial information with optimal preprocessing time via
1131 unified bound heaps. In *Proceedings of the 67th Annual IEEE Symposium on Foundations of*
1132 *Computer Science (FOCS)*, 2026. To appear.
- 1133 30 Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497,
1134 1996.
- 1135 31 Steven S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28(4):775–784, 1988.
1136 doi:10.1007/BF01954897.
- 1137 32 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal*
1138 *of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.
- 1139 33 Klaus W. Wagner. Monotonic coverings of finite sets. *Elektronische Informationsverarbeitung*
1140 *und Kybernetik*, 20(12):633–639, 1984.