Counting Inversions, Offline Orthogonal Range Counting, and Related Problems

Timothy M. Chan* Mihai Pătraşcu†

July 7, 2009

Abstract

We give an $O(n\sqrt{\lg n})$-time algorithm for counting the number of inversions in a permutation on n elements. This improves a long-standing previous bound of $O(n \lg n / \lg\lg n)$ that followed from Dietz's data structure [WADS'89], and answers a question of Andersson and Petersson [SODA'95]. As Dietz's result is known to be optimal for the related dynamic rank problem, our result demonstrates a significant improvement in the *offline* setting.

Our new technique is quite simple: we perform a "vertical partitioning" of a trie (akin to van Emde Boas trees), and use ideas from external memory. However, the technique finds numerous applications: for example, we obtain

- in d dimensions, an algorithm to answer n offline orthogonal range counting queries in time $O(n \lg^{d-2+1/d} n)$;
- an improved construction time for online data structures for orthogonal range counting;
- an improved update time for the partial sums problem;
- faster Word RAM algorithms for finding the maximum depth in an arrangement of axis-aligned rectangles, and for the slope selection problem.

As a bonus, we also give a simple $(1+\varepsilon)$-approximation algorithm for counting inversions that runs in linear time, improving the previous $O(n \lg\lg n)$ bound by Andersson and Petersson.

1 Introduction

1.1 A Complexity Theoretic View

The standard model of computation enshrined in all widely used programming languages (like C, Java, Python, etc.) allows the following assumptions:

- the memory consists of words, and permits random access.
- machine words can be manipulated in constant time by a standard set of operations (including arithmetic, shift, bitwise, and logical operations).
- the machine word is large enough to represent pointers and indices to the data.

*Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca). Work supported by NSERC.

†IBM Almaden Research Center, San Jose CA, USA (mip@alum.mit.edu).

In theory, this model has been formalized as the *Word RAM*. Understanding the power of this model has been a very active direction after Fredman and Willard's seminal result [FW93] demonstrating that sorting can be done in $o(n \lg n)$ operations.

Concentrating on offline problems, the main flavors of algorithmic improvement achieved in the Word RAM model seem to be:

- "accidental" problems with a linear-time solution, such as the classic algorithm of Rabin for planar closest pair [Rab76] or Thorup's algorithm for undirected single-source shortest paths [Tho99].
- sorting, and the myriad problems reducible to it, which have a current bound of $O(n\sqrt{\lg \lg n})$ due to Han and Thorup [HT02].
- problems related to offline point location, with a current bound of $n \cdot 2^{O(\sqrt{\lg \lg n})}$ [CP07].
- problems with complexity $O(n \lg n / \lg \lg n)$, stemming from the use of data structures with slightly sublogarithmic performance (including the problems addressed in this paper).
- logarithmic improvements to algorithms with a polynomial running time, from the classic "four Russians trick" to computing all-pairs shortest paths [Cha07].

In this paper, we introduce an algorithmic innovation that creates a new "level" in this hardness classification. In particular, we show that counting the number of inversions in a permutation, and, more generally, 2-d offline orthogonal range counting are not limited by the complexity of the associated online data structures. Instead, these problems can be solved in $O(n\sqrt{\lg n})$ time.

Our algorithmic idea (in instantiations of varying technical difficulty) leads to improved results for an array of problems, including static and dynamic data structures. These results are summarized in Section 1.3. Before that, however, we take the opportunity to discuss our approach in the simplest possible setting: counting inversions.

1.2 Counting Inversions

The number of inversions in a permutation π is defined as the number of pairs $i < j$ with $\pi(i) > \pi(j)$. This is a natural and frequently-used measure for the sortedness of the data. The number of inversions between two permutations, called the Kendall tau distance, is a classic distance measure between two orders. In [DKNS01], it is argued that the Kendall tau distance is especially important for rank aggregation in large Internet-related applications.

History. Counting inversions in $O(n \lg n)$ time (e.g., by mergesort) is a textbook problem. For a faster solution, one can reduce to offline dominance counting in two dimensions: given a set of n points, how many other points does each point dominate? A point dominates another if each one of its coordinates is (strictly) larger. The problem in turn reduces to offline orthogonal range counting.

Until now, the fastest way to solve offline dominance counting in d dimensions was to treat one coordinate as "time," turning the problem into *dynamic* dominance counting in $d - 1$ dimensions. In one dimension, dominance counting is more commonly known as "dynamic ranking" or "the partial sums problem." This problem is solved in $O(\lg n / \lg \lg n)$ time for operation by Dietz's well known data structure, dating back to WADS'89 [Die89]. Thus, counting inversions can be done in $O(n \lg n / \lg \lg n)$ time.

Another classic result due to Fredman and Saks [FS89] states that $\Omega(\lg n / \lg \lg n)$ time is needed for dynamic ranking, even in the cell-probe model. Since counting inversions is so inherently tied to this problem, one is tempted to believe that the bound for counting inversions might also be optimal.

In SODA'95, Andersson and Petersson [AP98] attacked the approximate version of the problem and asked whether Dietz's bound can be improved for exact counting. Their algorithm runs in time $O(n \lg \lg n)$ and finds a $(1+\varepsilon)$-approximation for any $\varepsilon = \Omega(1/\sqrt{\lg \lg n})$.

In recent years, the problem has received renewed attention from the streaming community [AJKS02, GZ03]. In 2007, the status of the problem was sufficiently entrenched that, when discussing segment intersection counting, we [CP07] wrote: "Note that this problem is no easier than counting inversions in a permutation, so, in some sense, the best bound one would hope to get is $O(n \frac{\lg n}{\lg \lg n})$."

However, as we show here, counting the exact number of inversions can be done in $O(n\sqrt{\lg n})$ time. In Section 4, we also show how to improve the results of Andersson and Petersson for approximate counting. In particular, we obtain an $O(n)$-time algorithm that find a $(1+\varepsilon)$-approximation to the count, for any constant $\varepsilon > 0$.

The improved exact solution (a quick sketch). The starting point of our solution is the following simple variation of external-memory radix sort. Consider a sequence of n numbers of L bits each. In the external memory model with B words per page, we can count the number of inversions in the sequence using $O(nL/B)$ I/O operations. This is easy to see: We maintain a running count of the number of inversions. We begin by partitioning the elements into those beginning with a zero bit, and those beginning with a one bit. For each zero element, we add to the inversion count the number of preceding elements starting with a one. This partition takes $O(n/B)$ operations. Finally, we recurse into each of the subarrays, with L decremented.

This algorithm can be simulated on a Word RAM, but how do we obtain the ability to manipulate B elements in constant time? By packing multiple elements in a single word. If $B = w/L$ where w denotes the word size, the above algorithm would run in time $O(nL/B) = O(nL^2/w)$. For $w \approx \lg n$, we can simulate word operations in constant time by table lookup. The running time is thus $O(n)$ for $L \approx \sqrt{\lg n}$. We remark that the word packing idea is also a key ingredient in several previous algorithms that solve offline problems faster than their online counterparts (e.g., for sorting [FW93, AHNR98] and point location [CP07]).

How can we use the above subroutine to solve the original problem where the elements can be $\lg n$ bits long? We use a trick similar to the van Emde Boas data structure [vEBKZ77]: for each element, break its word into multiple components. Formally, consider a trie of depth $(\lg n)/L$ over the alphabet $[2^L]$.[1] Each node is associated with the elements of the permutation that fall under the node (they start with the prefix leading to the node). For a given node of the trie, consider the sequence of the first letters after the common prefix of the elements associated with the node (these letters are L-bit numbers). We compute the number of inversions in this sequence by the above subroutine and add to the running count. Finally, we recurse into each child of the node.

The trie can be built in $O(n)$ time per level, by bucketing. For $L \approx \sqrt{\lg n}$, the subroutine calls cost $O(n)$ per level. Since the depth is $(\lg n)/L$, we obtain an $O(n\sqrt{\lg n})$-time algorithm.

A more precise description of the algorithm, in a more general setting, is provided in Section 2.1.

1.3 Applications

Though our solution idea is quite simple, the end result remains surprising. In fact, more powerful instantiations of this basic idea yield a handful of improved algorithms and data structures.

Offline orthogonal range counting. A slightly trickier version (Section 2.2) of our initial algorithm can solve a more general problem in $O(n\sqrt{\lg n})$ time in 2-d: given n red/blue points, compute the number of red points dominated by each blue point. One can easily observe that this problem is equivalent to 2-d offline

[1] Throughout this paper, $[n]$ denotes $\{0, 1, \ldots, n-1\}$.

orthogonal range counting: given n points and n axis-aligned rectangles, compute the number of points inside each rectangle. The problem has an immediate application to 2-d orthogonal segment intersection counting: given n horizontal/vertical line segments, count the number of their intersections.

The offline orthogonal range counting result can be extended to any constant dimension d, as shown in Section 3, where we get running time $O(n \lg^{d-2+1/d} n)$. Previous known (online) algorithms require at least $O(n(\lg n/\lg\lg n)^{d-1})$ time (even if preprocessing cost is ignored) [JMS04]. Of course, orthogonal range searching is a well-loved topic in computational geometry and countless results have appeared in the literature, e.g., see [AE99, AAL09, ABR00, ABR01, Cha88, Cha90a, Cha90b, Cha97, JMS04, Păt07]. One should not confuse counting problems with their reporting analogs, though, and we have exploited special features of counting problems to get results better than for range searching in a more abstract group or semigroup model.

Offline dynamic ranking and selection. As another immediate corollary, we can solve an offline version of the dynamic ranking problem in 1-d: maintain a set of numbers under insertions and deletions so that we can determine the rank of any query point. Under the assumption that the sequence of updates and queries is known in advance, we obtain a total running time of $O(n\sqrt{\lg n})$.

Selection queries (the "inverse" of ranking) require more effort: given a query value k, we want to find the k-th smallest element in the current set. We show in Section A.1 that an offline sequence of updates and selection queries can be done in $O(n\sqrt{\lg n}\lg^{1/4}\lg n)$ expected time.

Online orthogonal range counting. The space/time tradeoffs possible for 2-d *online* orthogonal range counting are well understood: with space $O(n \operatorname{polylog} n)$, the best possible query time is $\Omega(\lg n/\lg\lg n)$ [Păt08, Păt07]. Furthermore, it is possible to achieve query time $O(\lg n/\lg\lg n)$ with space $O(n)$ [JMS04].

Given this precise understanding, it now makes sense to look at the preprocessing time more carefully. Unfortunately, it is not known how to construct a data structure with optimal query performance in linear time. For instance, the data structure of [JMS04] requires $O(n \lg n)$ preprocessing time; even the earlier $O(n)$-space data structure of Chazelle [Cha88] with $O(\lg n)$ query time, or the (even earlier) standard range tree, requires $O(n \lg n)$ preprocessing time.

In Section 2.3, we offer a 2-d data structure of linear size and optimal $O(\lg n/\lg\lg n)$ query time, which can be constructed faster in $O(n\sqrt{\lg n})$ time.

The generalization of the result in dimension d has $O(n \lg^{d-2+1/d} n)$ preprocessing time, $O(n(\lg n/\lg\lg n)^{d-2})$ space, and $O((\lg n/\lg\lg n)^{d-1})$ query time.

Online partial sums and dynamic ranking and selection. As we have mentioned, Dietz's data structure solved the online partial sum or online dynamic ranking problem in $O(\lg n/\lg\lg n)$ update time and query time. Although the query time matches known lower bounds, we show that curiously the update time can be improved—this is the first improvement since 1989. In Section A.4, we give a new online data structure for partial sums and dynamic ranking/selection with $O(\lg^{0.5+\varepsilon} n)$ update time and $O(\lg n/\lg\lg n)$ query time for an arbitrarily small constant $\varepsilon > 0$. Interestingly this solution exploits further techniques from external memory, namely, buffer trees [Arg03].

Range median. A series of recent papers [KMS05, HPM08, GS09] studied the following problem: preprocess a sequence of numbers $a_1, \ldots, a_n$ so that given any i, j, return the median of the subsequence $a_i, \ldots, a_j$. The best result known, by Gfeller and Sanders [GS09], achieved $O(n \lg n)$ preprocessing time, $O(n)$ space, and $O(\lg n)$ query time. In Section A.3, we obtain $O(n\sqrt{\lg n})$ preprocessing time, $O(n)$ space, and $O(\lg n)$ query time. Our observation is that certain data structures for 2-d orthogonal range counting can already answer range median queries efficiently; only the query algorithm needs to be changed. (The linear-space data

structure from [GS09], for instance, is essentially identical to Chazelle's data structure for 2-d orthogonal range counting [Cha88], and their simple $O(n \log n)$-space data structure is just a standard 2-d range tree in disguise.)

More applications in computational geometry. Many further consequences follow. For example, given n axis-aligned rectangles in 2-d, we can find the maximum depth in the arrangement in $O(n\sqrt{\lg n})$ time, as shown in Section A.2.

There are applications even to nonorthogonal problems. For example, consider the *slope selection* problem: given a set P of n points in 2-d and an integer k, find the k-th smallest slope formed by the $\binom{n}{2}$ lines through P. In the Real RAM model, several $O(n \lg n)$-time algorithms have been proposed for this problem [BC98, CSSS89, KS97]; in particular, in the known randomized algorithms [DMN92, Mat91], the dominant cost lies in counting inversions. It can be checked that with our inversion-counting subroutine, these random-sampling-based algorithms can solve the slope selection problem in $O(n\sqrt{\lg n})$ expected time on the Word RAM for points with integer coordinates.

Remark. All of our results for orthogonal problems on the Word RAM hold not only for integer input but also for floating-point numbers, since the known Word RAM sorting algorithms [HT02] are applicable to floats. The only main assumption is that the word size is at least as large as both $\lg n$ and the maximum size of an input number (i.e., each input number should fit in one word). In fact, in all of our algorithms, if the input numbers have been pre-sorted, we only need RAM operations on $\lg n$-bit words—which are standard and commonly assumed in analysis of algorithms. For results where the claimed time bounds exceed $n \lg n$ (such as our result on offline orthogonal range counting for dimension $d \geq 3$), the pre-sorting assumption is not even needed, since we can afford to use a comparison-based sorting algorithm.

2 2-d Offline Orthogonal Range Counting

2.1 The basic approach

We start by describing our technique for the 2-d offline red/blue dominance counting problem, which includes inversion counting as a special case (and also 2-d offline orthogonal segment intersection counting, by adding and subtracting a constant number of counts).

Theorem 2.1 *Given n red/blue points in the plane, we can count the number of pairs of points (p, q) where the red point p is dominated by the blue point q, in $O(n\sqrt{\lg n})$ time.*

Proof: *Algorithm template.* The basic strategy is a standard divide-and-conquer. Suppose that the input n-point set P is given in sorted x-order, has distinct x-coordinates, and has y-coordinates from $[2^\ell]$. Divide $\mathbf{R} \times [0, 2^\ell)$ into 2^h horizontal strips of height $2^{\ell-h}$ for some parameter h. Let P_i be the subset of points in the i-th strip. Round each point p immediately downward to a point $\widetilde{p}$ on a horizontal dividing line and let $\widetilde{P}$ be the set of rounded points. Recursively solve the subproblems for the P_i's and for $\widetilde{P}$. The overall count for P is just the sum of the counts for P_i's and the count for $\widetilde{P}$. Note that by translation and scaling, we can make the P_i's y-coordinates lie in $[2^{\ell-h}]$ and $\widetilde{P}$'s y-coordinates lie in $[2^h]$.

We now describe the choices of input representation and the parameter h, and elaborate on the details and analysis of the various steps. Fix a value L, to be set later.

Case 1: $\ell \leq L$. For each point p, we record its color and its y-coordinate $p.y$ (but not its x-coordinate) using $L + 1$ bits. We store the input point set P as a list of records in x-order, packed into $O(nL/w)$ words, each holding $O(w/L)$ points. We choose $h = 1$, i.e., use a 2-way divide-and-conquer.

We can split the list P into the two sublists P_0 and P_1 in $O(nL/w)$ time, by repeated use of the following word operation: given a word z holding $O(w/L)$ points and the number ℓ, output the word z' (resp. z'') holding the sublist of points p with $\widetilde{p}.y = 0$ (resp. $\widetilde{p}.y = 1$), retaining the same ordering in the sublist.

We can solve the subproblem for $\widetilde{P}$ directly in $O(nL/w)$ time: given the list of words $z_0, z_1, \ldots$, we can compute (i) the number r_i of red points p in z_i with $\widetilde{p}.y = 0$, (ii) the number b_i of blue points q in z_i with $\widetilde{q}.y = 1$, and (iii) the number c_i of red-blue dominating pairs within z_i, by word operations. The count for $\widetilde{P}$ is $\sum_i c_i + \sum_i b_i \sum_{j<i} r_j$, which is computable in time linear in the number of words.

As a result, we get the following recurrence for the running time:

$$T(n, \ell) \;=\; T(n_0, \ell - 1) + T(n_1, \ell - 1) + O(nL/w) \qquad \forall \ell \leq L \tag{1}$$

for some n_0, n_1 with $n_0 + n_1 = n$, where $T(n, 1) = O(nL/w)$ for the base case. This yields $T(n, L) = O(nL^2/w)$.

Case 2: $\ell > L$. Here, we store the given points P naively in $O(n)$ words, but we choose $h = L$ instead. We can split the list P into the sublists $P_0, \ldots, P_{2^L-1}$, while retaining the same ordering in the sublists, in $O(n)$ time by using an array of 2^L buckets. We can generate $\widetilde{P}$ in $O(n)$ time, convert it into the packed-word representation, and compute the count for $\widetilde{P}$ in $O(nL^2/w)$ time by Case 1.

We get the following recurrence:

$$T(n, \ell) \;=\; \sum_{i=0}^{2^L-1} T(n_i, \ell - L) + O(n(1 + L^2/w)), \tag{2}$$

for some n_i's with $\sum_i n_i = n$. This yields $T(n, \ell) = O(n(\ell/L)(1 + L^2/w))$. Setting $L = \sqrt{w}$ gives $T(n, \ell) = O(n\ell/\sqrt{w})$.

Initialization. We can pre-sort both the x- and the y-coordinates by known Word RAM sorting algorithms, for example, in $O(n \lg \lg n)$ deterministic time [Han04]. By normalization, we can make the x- (resp. y-)coordinates distinct and lie in $[n]$, so initially $\ell = \lg n$. To support the above word operations in constant time, we can precompute tables storing the output of the operations on every input combination, in time $2^{O(w)}$. Setting $w = \varepsilon \lg n$ for some constant $\varepsilon > 0$, we get the final time bound of $O((n \lg n)/\sqrt{w}) = O(n\sqrt{\lg n})$. □

Corollary 2.2 *Given a permutation over $[n]$, we can count the number of inversions in $O(n\sqrt{\lg n})$ time.*

Corollary 2.3 *Given n horizontal/vertical line segments in the plane, we can count the number of their intersections in $O(n\sqrt{\lg n})$ time.*

2.2 Individual counts

For the version of the offline dominance counting problem where we want a separate count for each query point, we need to work a little harder. Offline orthogonal range counting reduces to this problem (by adding and subtracting a constant number of counts per query point).

Theorem 2.4 *Given n red/blue points in the plane, we can count the number of red points dominated by each blue point, in $O(n\sqrt{\lg n})$ total time.*

Proof: We follow the same algorithm template as in the proof of Theorem 2.1. Here, for each blue point p, if $p \in P_i$, the count for p in P is just the sum of the count for p in P_i and the count for $\widetilde{p}$ in $\widetilde{P}$.

As output representation becomes an issue, some modifications are needed and one more case arises.

Case $1'$: $\ell \leq L$ and $n \leq 2^L$. The record for each point $p \in P$ now stores an extra field for the output count (initially, this field is nil). Since $n \leq 2^L$, the counts are only $O(L)$-bit long, so the point set P can still be packed into $O(nL/w)$ words. We choose $h = 1$.

Splitting of P into P_1 and P_2 can be done as before. The subproblem for $\widetilde{P}$ can be solved using $O(nL/w)$ number of similarly defined word operations. To combine the output, we first need to "unsplit" P_1 and P_2 to get back P with the count fields filled in. Unsplitting can be done in $O(nL/w)$ time, e.g., by repeated use of the following word operation: given a word z holding $O(w/L)$ points, a number ℓ, and a word z' (resp. z'') holding the sublist of points p with $\widetilde{p}.y = 0$ (resp. $\widetilde{p}.y = 1$), output a modified version of z with the count fields copied from the corresponding counts in z' (resp. z''). We can then add the individual counts computed for $\widetilde{P}$ to the counts for P, again with $O(nL/w)$ word operations.

As a result, we get the same recurrence (1) and obtain $O(nL^2/w)$ running time.

Case $1''$: $\ell \leq L$ and $n > 2^L$. This time, we form the subsets P_i by using vertical dividing lines instead, with each subset containing $n/2^h$ points. We form $\widetilde{P}$ by rounding leftward. We choose $h = \lg n - L$.

We can easily generate the P_i's and $\widetilde{P}$ in $O(n)$ time. We can solve the subproblem for each P_i in $O(nL^2/w)$ time by Case $1'$. We can solve the subproblem for $\widetilde{P}$ directly in $O(n)$ time, since the rounded points lie in a grid $[2^h] \times [2^L]$, which has size $O(n)$ (it is easy to compute all the counts in $\widetilde{P}$ from the multiplicities of all the grid points). Thus, we get a time bound of $O(n(1 + L^2/w))$.

Case 2: $\ell > L$. At this stage, we can proceed exactly as in the proof of Theorem 2.1 and obtain the recurrence (2) and the final time bound of $O(n\sqrt{\lg n})$. □

Corollary 2.5 *Given n points and n axis-aligned rectangles in the plane, we can count the number of points inside each box in $O(n\sqrt{\lg n})$ total time.*

2.3 Adding online queries

Although we cannot improve the query time for the corresponding online problems, we can apply our approach to improve the preprocessing time of online data structures. We will need some extra ideas similar to techniques from succinct data structures (which were also used in Chazelle's previous data structure [Cha88]).

Theorem 2.6 *We can preprocess n points in the plane in $O(n\sqrt{\lg n})$ time, using a data structure with $O(n)$ words of space, so that we can count the number of points dominated by a query point in $O(\lg n/\lg\lg n)$ time.*

Proof: We follow the same algorithm template as in the proof of Theorem 2.1 (but ignoring colors). Here, the count for a query point q in P is just the sum of the count for q in P_i and the count for $\widetilde{q}$ in $\widetilde{P}$. We assume that in the query algorithm, we are given the x-rank of q in P. Let A and H be parameters to be set later.

Case $0'$: $\ell \leq H$ and $n \leq 2^A$. Here, we solve the problem directly without recursion. We store the given point set P as a sequence of $O(nH/w)$ words $z_0, z_1, \ldots$, each holding $g = O(w/H)$ points of P, in x-order. For each i and each $j \in [2^H]$, let c_{ij} be the number of points p in $z_0, \ldots, z_{i-1}$ with $p.y < j$. These $O(nH/w \cdot 2^H)$ numbers are A bits long and can thus be packed in $O(nH/w \cdot 2^H \cdot A/w)$ words. We can bound the preprocessing time by $O(nH/w \cdot 2^H)$ and space by $O(nH/w + nH/w \cdot 2^H \cdot A/w)$.

Given a query point q with x-rank r, we can compute the count for q in P as follows: letting $i = \lfloor r/g \rfloor$ and $j = q.y$, we compute the number of points p in z_i that are dominated by q by a word operation, and add c_{ij} to the count. The query time is $O(1)$.

Case $0''$: $\ell \le H$ and $n > 2^A$. We form the subsets P_i by using vertical dividing lines instead, with each subset containing $n/2^h$ points. We form $\widetilde{P}$ by rounding leftward. We choose $h = \lg n - A$. For $\widetilde{P}$, we explicitly build a table containing all answers, in $O(n/2^A \cdot 2^H)$ time and space. The total preprocessing time $O(2^H nH/w + n/2^{A-H})$, and space is $O(nH/w + 2^H n(A/w)(H/w) + n/2^{A-H})$.

Given a query point q with x-rank r, we can compute the count for q in P as follows: letting $i = \lfloor r/g \rfloor$, we recursively compute the add for q in P_i, look up the count for $\widetilde{q}$ in $\widetilde{P}$ from the table, and return the sum. The query time is $O(1)$.

Case 1: $H < \ell \le L$. We represent the given point set P as a sequence of $O(nL/w)$. We return to dividing by horizontal lines. Previously we chose $h = 1$ for this case, but we now choose $h = H$.

Recall that we can split P into two sublists in $O(nL/w)$ time. By applying this procedure recursively, we can split the list P into the 2^H sublists $P_0, \ldots, P_{2^H-1}$ in $O(nHL/w)$ time.

We build the data structure for $\widetilde{P}$ by Cases $0'$ and $0''$. We then recursively build the data structure for the P_i's.

Given a query point q with x-rank r, we can compute the count for $\widetilde{q}$ in $\widetilde{P}$ by Cases $0'$ and $0''$, and recursively compute the count for q in P_i with $i = \widetilde{q}.y$. Before the recursion, we need to determine the x-rank of q in P_i, which can be done in two queries on $\widetilde{P}$: just take the number of points dominated by $(\widetilde{q}.x, \widetilde{q}.y + 1)$ minus the number of points dominated by $(\widetilde{q}.x, \widetilde{q}.y)$ in $\widetilde{P}$.

As a result, we get the following recurrences for the preprocessing time, space, and query time:

$$\begin{aligned}
P(n,\ell) &= \sum_i P(n_i, \ell - H) + O(nHL/w + 2^H nH/w + n/2^{A-H}) \\
S(n,\ell) &= \sum_i S(n_i, \ell - H) + O(nH/w + 2^H n(A/w)(H/w) + n/2^{A-H}) \\
Q(n,\ell) &= \max_i Q(n_i, \ell - H) + O(1) \qquad\qquad \forall \ell \le L
\end{aligned}$$

for some n_i's with $\sum_i n_i = n$. This yields $P(n, L) = O(nL^2/w + 2^H nL/w + (n/2^{A-H})(L/H))$, $S(n, L) = O(nL/w + 2^H n(A/w)(L/w) + (n/2^{A-H})(L/H))$, and $Q(n, L) = O(L/H)$.

Case 2: $\ell > L$. Here, we store the given points P naively in $O(n)$ words, but we choose $h = L$ instead. We build the data structure for $\widetilde{P}$ by Case 1 and build the data structure for the P_i's recursively. Given a query point q with x-rank r, we can compute the count for $\widetilde{q}$ in $\widetilde{P}$ by Case 1, and recursively compute the count for q in P_i with $i = \widetilde{q}.y$. Before recursion, we determine the x-rank of q in P_i, which can be done in $O(1)$ time by two queries on $\widetilde{P}$ as before.

We get the following recurrences:

$$\begin{aligned}
P(n,\ell) &= \sum_i P(n_i, \ell - L) + O(n + nL^2/w + 2^H nL/w + (n/2^{A-H})(L/H)) \\
S(n,\ell) &= \sum_i S(n_i, \ell - L) + O(nL/w + n(A/w)(L/w) + (n/2^{A-H})(L/H)) \\
Q(n,\ell) &= \max_i Q(n_i, \ell - L) + O(L/H).
\end{aligned}$$

for some n_i's with $\sum_i n_i = n$. This yields $P(n,\ell) = O((\ell/L)[n + nL^2/w + 2^H nL/w + (n/2^{A-H})(L/H)])$, $S(n,\ell) = O(\ell/L)[nL/w + 2^H n(A/w)(L/w) + (n/2^{A-H})(L/H)])$, and $Q(n,\ell) = O(\ell/H)$. Setting $L = \sqrt{w}$, $w = \varepsilon \lg n$, $A = (1+\varepsilon)\lg w$, and $H = \varepsilon \lg w$ gives $P(n, \lg n) = O(n\sqrt{\lg n})$, $S(n, \lg n) = O(n)$, and $Q(n, \lg n) = O(\lg n / \lg \lg n)$.

Note that initially we can determine the x-rank of the query point by predecessor search, e.g., in $O(\sqrt{\lg n})$ time [FW93]. □

3 Higher Dimensions

We can generalize the results in Section 2 to higher dimensions. In fact, the improvement gets closer to a full logarithmic factor as the dimension increases.

Theorem 3.1 *Given n red/blue points in a constant dimension $d \geq 3$, we can count the number of red points dominated by each blue point, in $O(n \lg^{d-2+1/d} n)$ total time.*

Proof: We generalize the algorithm template in the proof of Theorems 2.1 and 2.4 in the obvious way. Suppose that the input point set P is given in sorted order with respect to the last coordinate, has distinct last coordinates, and has j-th-coordinates from $[2^{\ell_j}]$ for $j = 1, \ldots, d-1$.

In Case 1′, we assume that $\ell_1, \ldots, \ell_{d-1} \leq L$ and $n \leq 2^{(d-1)L}$. The point set can still be packed into $O(nL/w)$ words. By dividing with respect to the j-th coordinate, we get the following analog of recurrence (1):

$$\begin{aligned} T(n, \ell_1, \ldots, \ell_{d-1}) &= \sum_{i=0}^{1} T(n_i, \ell_1, \ldots, \ell_{j-1}, \ell_j - 1, \ell_{j+1}, \ldots, \ell_{d-1}) \\ &\quad + T(n_i, \ell_1, \ldots, \ell_{j-1}, 1, \ell_{j+1}, \ldots, \ell_{d-1}) \\ &\quad + O(nL/w) \qquad \forall \ell_1, \ldots, \ell_{d-1} \leq L,\ \forall j \in \{1, \ldots, d-1\} \end{aligned}$$

for some n_1, n_2 with $n_1 + n_2 = n$, where $T(n, 1, \ldots, 1) = O(nL/w)$. This yields $T(n, \ell_1, \ldots, \ell_{d-1}) = O(n\ell_1 \cdots \ell_{d-1} L/w) = O(nL^d/w)$.

In Case 1″, we remove the assumption about n. After resetting h to $\lg n - (d-1)L$, we get the time bound $O(n(1 + L^d/w))$.

In Case 2, the recurrence (2) modifies to

$$\begin{aligned} T(n, \ell_1, \ldots, \ell_{d-1}) &= \sum_{i=0}^{2^L - 1} T(n_i, \ell_1, \ldots, \ell_{j-1}, \ell_j - L, \ell_{j+1}, \ldots, \ell_{d-1}) \\ &\quad + T(n_i, \ell_1, \ldots, \ell_{j-1}, L, \ell_{j+1}, \ldots, \ell_{d-1}) \\ &\quad + O(nL/w) \qquad \forall \ell_1, \ldots, \ell_{d-1} \leq L,\ \forall j \in \{1, \ldots, d-1\} \end{aligned}$$

for some n_i's with $\sum_i n_i = n$, where $T(n, L, \ldots, L) = O(n(1 + L^d/w))$. This yields $T(n, \ell_1, \ldots, \ell_{d-1}) = O(n(\ell_1/L) \cdots (\ell_{d-1}/L)(1 + L^d/w))$. Setting $L = w^{1/d}$ and $w = \varepsilon \lg n$ gives $T(n, \lg n, \ldots, \lg n) = O(n[(\lg n)/w^{1/d}]^{d-1}) = O(n[(\lg n)^{1-1/d}]^{d-1}) = O(n \lg^{d-2+1/d} n)$.

In initialization, we can afford to use any standard $O(n \lg n)$ sorting algorithm in place of known Word RAM results. □

Corollary 3.2 *Given n points and n axis-aligned boxes in a constant dimension $d \geq 3$, we can count the number of points inside each box, in $O(n \lg^{d-2+1/d} n)$ total time.*

The above result leads to some interesting new time bounds for a few applications. For example:

- The 2-d offline dynamic orthogonal range counting problem reduces to 3-d offline (static) orthogonal range counting (by adding time as the third coordinate and subtracting counts for points deleted from counts for points inserted). It can thus be solved in $O(n \lg^{4/3} n)$ total time, i.e., $O(\lg^{4/3} n)$ amortized update/query time. In contrast, the best 2-d online dynamic orthogonal range counting result with $O(\text{polylog}\, n)$ update time has $O((\lg n / \lg \lg n)^2)$ query time [Nek09].

- The following rectangle enclosure problem has been studied by several researchers [PS85, LP82, GJSD97]: given n axis-aligned rectangles in the plane, report all pairs of rectangles (r_1, r_2) where r_1 is strictly contained in r_2. The counting version of the problem reduces to 4-d offline orthogonal range counting (by mapping each rectangle to a 4-d point) and can thus be solved in $O(n \lg^{2.25} n)$ time.

We can also obtain a generalization of the online data structure in Theorem 2.6 (the proof is similar enough and hence is omitted):

Theorem 3.3 *We can preprocess n points in a constant dimension $d \geq 3$ in $O(n \lg^{d-2+1/d} n)$ time, using a data structure with $O(n(\lg n/\lg\lg n)^{d-2})$ words of space, so that we can count the number of points dominated by a query point in $O((\lg n/\lg\lg n)^{d-1})$ time.*

4 Approximation

In this final section, we turn to approximation results, specifically, for our initial problem: counting inversions. We present a nice, simple linear-time algorithm that improves over known $(1+\varepsilon)$-approximation algorithms. We first solve a modified version of the approximation problem in which we are given a "threshold" parameter K. We will describe our solution in setting of 2-d offline dominance counting:

Theorem 4.1 *Given m red and n blue points in $[m+n]^2$, let K^* denote the (unknown) number of pairs (p,q) where the red point p is dominated by the blue point q. Given K, we can either report $K^* > K$ or approximate K^* to within an additive error of εK, in $O(m+n)$ time.*

Proof: *The algorithm.* Let $k = 2K/n$. We sweep the points from left to right. Let S be the (dynamic) set of all red points to the left of the sweep line, ordered by y-coordinates. Let $\{p_1, \ldots, p_b\}$ be a set of "approximate quantiles", with $b = \lceil 20/\varepsilon \rceil$, satisfying the property that p_i has rank in $[ik/b, ik/b + 0.05\varepsilon k)$ in S at all times. Such approximate quantiles can be maintained easily in $O(1)$ amortized time as follows: initialize T to $\emptyset$; whenever the sweep line passes through $\lfloor 0.05\varepsilon k \rfloor$ red points, insert these points to T, eliminate all points of y-rank exceeding k from T, and reset each p_i to be the (ik/b)-th lowest point in T, computable in $O(k)$ time by a linear-time selection algorithm.

Whenever the sweep line passes through a blue point q, we do the following: If $q.y > p_b$, then we mark q as bad. Otherwise, if $p_i.y \leq q.y < p_{i+1}.y$, then the number of red points dominated by q lies in $[ik/b, (i+1)k/b + 0.05\varepsilon k) \subseteq [ik/b, ik/b + 0.1\varepsilon k)$, and we add ik/b to the total count.

Observe that each bad red point dominates at least k blue points. Thus, if the number of bad red points exceed $n/2$, then $K^* > nk/2 = K$ and we can quit. Otherwise, we recurse over all bad blue points and all red points, with $\varepsilon \leftarrow 0.8\varepsilon$ and the colors swapped.

Analysis. The additive error incurred by the non-bad blue points is at most $0.1\varepsilon kn \leq 0.2\varepsilon K$. The additive error from the recursive call is at most $0.8\varepsilon K$ by hypothesis. So, the total error is indeed at most εK as claimed.

The running time satisfies the recurrence

$$\begin{aligned} T(m,n,\varepsilon) &\leq T(n/2, m, 0.8\varepsilon) + O((m+n)/\varepsilon) \\ &\leq T(m/2, n/2, 0.64\varepsilon) + O((m+n)/\varepsilon), \end{aligned}$$

which expands to a decreasing geometric series, yielding $T(m,n,\varepsilon) = O((m+n)/\varepsilon)$. □

Corollary 4.2 *Given a permutation π over $[n]$ and any constant $\varepsilon > 0$, we can compute a factor-$(1+\varepsilon)$ approximation to the number of inversions in $O(n)$ time.*

Proof: It is known that the quantity $\sum_i |\pi(i) - i|$ gives a factor-2 approximation on the number of inversions [DG77], and can be found in linear time. Thus, we can find a K such that $K^* \leq K < 2K^*$. We can then apply Theorem 4.1 to get a factor-$(1 + O(\varepsilon))$ approximation. □

We have not optimized the ε-dependencies in the running time, to keep the algorithm simple. It is not difficult to get $O(n \lg(1/\varepsilon))$ time by using extra data structures to maintain the approximate quantiles (we can even get $O(n)$ time for $\varepsilon = \Omega(1/\mathrm{polylog}\, n)$ with more effort, by using fusion trees). We can also obtain randomized $(1+\varepsilon)$-approximation results for other related problems, for example, dynamic approximate rank queries. Due to lack of space, these results are deferred to the full version of the paper.

References

[AAL09] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *Proc. 50th Annual Symposium on Foundations of Computer Science (FOCS)*, 2009. To appear.

[ABR00] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

[ABR01] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.

[AE99] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.

[AHNR98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998. See also STOC'95.

[AJKS02] Miklós Ajtai, T. S. Jayram, Ravi Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *Proc. 34th ACM Symposium on Theory of Computing (STOC)*, pages 370–379, 2002.

[AP98] Arne Andersson and Ola Petersson. Approximate indexed lists. *Journal of Algorithms*, 29(2):256–276, 1998. See also SODA'95.

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. See also WADS'95.

[BC98] Hervé Brönnimann and Bernard Chazelle. Optimal slope selection via cuttings. *Computational Geometry: Theory and Applications*, 10(1):23–29, 1998.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[Cha88] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17:427–462, 1988. See also FOCS'85.

[Cha90a] Bernard Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *Journal of the ACM*, 37:200–212, 1990. See also FOCS'86.

[Cha90b] Bernard Chazelle. Lower bounds for orthogonal range searching: II. The arithmetic model. *Journal of the ACM*, 37:439–463, 1990. See also FOCS'86.

[Cha97] Bernard Chazelle. Lower bounds for off-line range searching. *Discrete and Computational Geometry*, 17(1):53–66, 1997. See also STOC'95.

[Cha99] Timothy M. Chan. Geometric applications of a randomized optimization technique. *Discrete and Computational Geometry*, 22(4):547–567, 1999.

[Cha07] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. 39th ACM Symposium on Theory of Computing (STOC)*, pages 590–598, 2007.

[CP07] Timothy M. Chan and Mihai Pătraşcu. Voronoi diagrams in $n \cdot 2^{O(\sqrt{\lg \lg n})}$ time. In *Proc. 39th ACM Symposium on Theory of Computing (STOC)*, pages 31–39, 2007.

[CSSS89] Richard Cole, Jeffrey S. Salowe, William L. Steiger, and Endre Szemerédi. An optimal-time algorithm for slope selection. *SIAM Journal on Computing*, 18(4):792–810, 1989. See also ICALP'88.

[Die89] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st Workshop on Algorithms and Data Structures (WADS)*, pages 39–46, 1989.

[DG77] Persi Diaconis and Ronald L. Graham. Spearman's footrule as a measure of disarray. *J. Royal Statistical Society Series B*, 39:262–268, 1977.

[DKNS01] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Ranking aggregation methods for the web. In *Proc. 10th WWW*, pages 613–622, 2001.

[DMN92] Michael B. Dillencourt, David M. Mount, and Nathan S. Netanyahu. A randomized algorithm for slope selection. *International Journal of Computational Geometry and Applications*, 2:1–27, 1992.

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[EE94] David Eppstein and Jeff Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete and Computational Geometry*, 11:321–350, 1994.

[FS89] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. See also STOC'90.

[GJSD97] Prosenjit Gupta, Ravi Janardan, Michiel H. M. Smid, and Bhaskar DasGupta. The rectangle enclosure and point-dominance problems revisited. *International Journal of Computational Geometry and Applications*, 7(5):437–455, 1997.

[GS09] Beat Gfeller and Peter Sanders. Towards optimal range medians. CoRR abs/0901.1761, 2009. See also ICALP'09.

[GZ03] Anupam Gupta and Francis Zane. Counting inversions in lists. In *Proc. 14th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 253–254, 2003.

[Han04] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004. See also STOC'02.

[HPM08] Sariel Har-Peled and S. Muthukrishnan. Range medians. In *Proc. 16th European Symposium on Algorithms (ESA)*, pages 503–514, 2008.

[HT02] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proc. 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002.

[JMS04] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proc. 15th International Symposium on Algorithms and Computation (ISAAC)*, pages 558–568, 2004.

[KMS05] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1–17, 2005. See also ISAAC'03.

[KS97] Matthew J. Katz and Micha Sharir. An expander-based approach to geometric optimization. *SIAM Journal on Computing*, 26:1384–1408, 1997.

[LP82] D. T. Lee and Franco P. Preparata. An improved algorithm for the rectangle enclosure problem. *Journal of Algorithms*, 3(3):218–224, 1982.

[Mat91] Jiří Matoušek. Randomized optimal algorithm for slope selection. *Information Processing Letters*, 39:183–187, 1991.

[Nek09] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry: Theory and Applications*, 42(4):342–351, 2009. See also WADS'07.

[Păt07] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proc. 39th ACM Symposium on Theory of Computing (STOC)*, pages 40–46, 2007.

[Păt08] Mihai Pătraşcu. (Data) STRUCTURES. In *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[PD06] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. See also SODA'04 and STOC'04.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[Rab76] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–30. Academic Press, 1976.

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999. See also FOCS'97.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. Conference version by van Emde Boas alone in FOCS'75.

A Appendix

A.1 1-d offline dynamic ranking and selection

In this section, we explore variants of our algorithms to solve several related problems. We first consider the 1-d dynamic offline ranking/selection problem. Here, we want to maintain a set P of numbers, under the following operations: P.insert(p) inserts a new element p to P; P.delete(p) removes p from P; P.rank(p) returns the rank of p; and P.select(k) returns the k-th element of P. We allow duplicate elements (i.e., P is a multiset). In the offline setting, the sequence of all update and query operations is given in advance.

It is easy to reduce offline dynamic ranking queries directly to 2-d offline red/blue dominance counting. For selection queries, we present a modification of our algorithm:

Theorem A.1 *We can perform an offline sequence of n updates and selection queries over a (multi)set of integers in $O(n\sqrt{\lg n}\,\lg^{1/4}\lg n)$ expected total time.*

Proof: *Algorithm template.* Suppose we are given an offline sequence S of n operations over a multiset $P \subseteq [2^\ell]$, and an initial multiset for P (we allow P to be initially nonempty).

For each $i \in [2^h]$, let P_i be the multiset $\{p \in P : \left\lfloor p/2^{\ell-h} \right\rfloor = i\}$. Let $\widetilde{P}$ be the multiset $\{\left\lfloor p/2^{\ell-h} \right\rfloor : p \in P\}$. To handle each operation P.insert(p) (resp. P.delete(p)), we compute $i = \left\lfloor p/2^{\ell-h} \right\rfloor$ and invoke the operations P_i.insert(p) and $\widetilde{P}$.insert(i) (resp. P_i.delete(p) and $\widetilde{P}$.delete(i)). To handle the operation P.select(k), we first invoke $\widetilde{P}$.select(k) to get an intermediate output i, then invoke $\widetilde{P}$.rank(i) to get another intermediate output k', and then invoke P_i.select$(k - k' + 1)$ to get the final output. Note that all

the invoked operations on $\widetilde{P}$ are to be executed collectively first, before the operations on the P_i's. Also note that by translation, we can make $P_i \subseteq [2^{\ell-h}]$, and we know $\widetilde{P} \subseteq [2^h]$.

Case $1'$: $\ell \leq L$, $n \leq 2^L$, and the maximum multiplicity is at most $O(2^L)$. For each operation in S, we create a record storing its type, its input parameter (an element p or an index k), and its (intermediate or final) output. Since each index k can be at most $2^L \cdot 2^L$ by the multiplicity assumption, we can store the given sequence S, together with the initial multiset, in $O(nL/w)$ words. We choose $h = 1$.

We first gather the sequence of invoked insert/delete/select operations on $\widetilde{P}$ and execute them directly. This can be done in $O(nL/w)$ time by appropriate word operations. We then perform the sequence of invoked rank operations on $\widetilde{P}$ directly, also in $O(nL/w)$ time (recall that offline dynamic rank queries reduce to red-blue dominance counting). We then split S (based on the intermediate output for each select operation) to get the sequence of invoked operations on P_0 and the invoked operations on P_1. This again takes $O(nL/w)$ time. After recursively handling these two sequences, we can unsplit to get back S with the final output filled in.

As a result, we get the same recurrence (1) and obtain $O(nL^2/w)$ running time.

Case $1''$: $\ell \leq L$, without further restrictions. We divide the sequence S into $O(n/2^L)$ blocks, each containing $2^L - 1$ operations. Fix one block. At the beginning of the block, identify the "problematic" elements $p_0, p_1, \ldots$ that have initial multiplicity greater than 2^{L+1}. Let μ_i and r_i denote the initial multiplicity and rank of p_i respectively, and let $d_i = \sum_{j \leq i}(\mu_j - 2^L)$. We can easily compute all these values in $O(2^L)$ time per block. Let P' be equal to P, except that the multiplicity of p_i in P' is decreased by $\mu_i - 2^L$. Note that the multiplicities in P' lie in the range $(0, 2^{L+1})$ at all times, since the block has fewer than 2^L operations.

Updates to P are applied to P'. To find the k-th smallest element in P, we first find i with $r_i + 2^L \leq k < r_{i+1} + 2^L$ by a predecessor search. Observe that at all times, the rank of p_i in P is $< r_i + 2^L \leq k$, and the rank of $p_{i+1} + 1$ in P is $> r_{i+1} + 2^{L+1} - 2^L \geq k$. So, the answer must lie between p_i and $p_{i+1} + 1$. On the other hand, observe that for each $p = p_i + 1, \ldots, p_{i+1}$, the rank of p in P' exactly equals the rank of p in P minus d_i. So, compute the $(k - d_i)$-th smallest element q in P'. If $q \leq p_i$, return p_i; if $q \geq p_{i+1} + 1$, return $p_{i+1} + 1$; otherwise, return q.

Note that for each block, the $O(2^L)$ predecessor queries are given offline and are done to a static set of size $O(2^L)$. Thus, they can be performed collectively by sorting, which takes $O(2^L\sqrt{\lg\lg 2^L})$ expected time by the fastest known Word RAM algorithm [HT02]. The selection queries on P' take $O(2^L L^2/w)$ time by Case $1'$. Summing over all $O(n/2^L)$ blocks, we get a total expected running time of $O(n(\sqrt{\lg L} + L^2/w))$.

Case 2: $\ell > L$. Here, we store the given sequence S naively in $O(n)$ words, but we choose $h = L$ instead. We handle the invoked insert/delete/select operations on $\widetilde{P}$ by Case 1, and the rank operations on $\widetilde{P}$ by dominance counting, and then recursively perform the invoked operations on the P_i's. We get almost the same recurrence as (2), except that the overhead term is $O(n(\sqrt{\lg L} + L^2/w))$. This yields $T(n, \ell) = O(n(\ell/L)(\sqrt{\lg L} + L^2/w))$. Setting $L = \sqrt{w}\lg^{1/4} w$ gives $T(n, \lg n) = O((n/\sqrt{w})\lg n \lg^{1/4} w) = O(n\sqrt{\lg n}\lg^{1/4}\lg n)$.

In initialization, we can again use a Word RAM sorting algorithm to convert the integers to lie in $[n]$. □

A.2 1-d offline dynamic depth

Next, we turn to an offline version of the following problem: maintain the maximum depth in a set of intervals in 1-d, under insertions and deletions of intervals. Here, the depth of a point p with respect to a collection of objects (e.g., intervals) is defined as the number of objects containing p, and the maximum depth refers to the maximum depth over all possible points.

We generalize the problem slightly and consider the maintenance of a set P of weighted integer points in 1-d to support the following operation:

P.prefix-update(p, δ), where $\delta \in \{-1, +1\}$: add δ to the weight of every element $q \leq p$ in P, and output the change in the maximum weight in P (the change is either 0 or δ).

Initially, we let P contain all the intervals' endpoints (which can be normalized to lie in $[2n]$ after calling a Word RAM sorting algorithm), and we set all the weights to 0. We can simulate an insertion (resp. deletion) of an interval $[p, q]$ by invoking P.prefix-update$(q, +1)$ and P.prefix-update$(p - 1, -1)$ (resp. P.prefix-update$(q, -1)$ and P.prefix-update$(p - 1, +1)$); then the maximum depth corresponds to the maximum weight.

Theorem A.2 *We can perform an offline sequence of n* prefix-update() *operations in $O(n\sqrt{\lg n})$ total time.*

Proof: *Algorithm template.* Suppose we are given the offline sequence S of n operations over a set $P \subseteq [2^\ell]$. Divide P into 2^h subsets $P_i \subseteq [2^{\ell-h} i, 2^{\ell-h}(i+1))$. Create an extra set $\widetilde{P}$, maintaining the invariant that the weight of i in $\widetilde{P}$ equals the maximum weight of p in P over all $p \in P_i$. To handle the operation P.prefix-update(p, δ), we compute $i = \left\lfloor p/2^{\ell-h} \right\rfloor$ and invoke P_i.prefix-update(p, δ); if the output is 0, then we invoke $\widetilde{P}$.prefix-update$(\left\lfloor p/2^{\ell-h} \right\rfloor - 1, \delta)$, else $\widetilde{P}$.prefix-update$(\left\lfloor p/2^{\ell-h} \right\rfloor, \delta)$, to get the final output. Note that by translation, we can make $P_i \subseteq [2^{\ell-h}]$, and we know $\widetilde{P} \subseteq [2^h]$.

Case 1: $\ell \leq L$. For each operation in S, we create a record storing its input parameters and a bit for (intermediate or final) output. The given sequence S can be stored in $O(nL/w)$ words. We choose $h = 1$.

We first split S to get the sequence of invoked operations on P_0 and the sequence of invoked operations on P_1, using $O(nL/w)$ word operations. We execute these two sequences recursively and then unsplit. We next generate the sequence of invoked operations on $\widetilde{P}$, again in $O(nL/w)$ time using appropriate word operations. We can solve the subproblem for $\widetilde{P}$ directly in $O(nL/w)$ time.

As a result, we get the same recurrence (1) and obtain $O(nL^2/w)$ running time.

Case 2: $\ell > L$. Here, we store the sequence S naively in $O(n)$ words, but we choose $h = L$ instead. As in the proof of Theorem 2.1, we get the same recurrence (2) and obtain the final time bound of $O(n\sqrt{\lg n})$. □

Corollary A.3 *We can maintain the maximum depth over a set of intervals under an offline sequence of n insertions and deletions in $O(n\sqrt{\lg n})$ total time.*

We mention two immediate consequences:

- We can compute the maximum depth in a 2-d arrangement of n axis-aligned rectangles in $O(n\sqrt{\lg n})$ time. This follows by applying a standard plane-sweep algorithm.
- Given n points in the plane and an integer k, we can find the smallest axis-aligned square enclosing k points in $O(n\sqrt{\lg n})$ expected time. This follows from the preceding item by applying a known randomized optimization technique [Cha99]. (Alternatively, by parametric or matrix searching techniques, we get an $O(n \lg^{3/2} n)$-time deterministic algorithm for this k-enclosing square problem, which improves over the previous best deterministic algorithm running in $O(n \lg^2 n)$ time [EE94].)

A.3 Online range median

Next, we consider range median queries. This problem can be viewed geometrically: for a set of n points 2-d, we want to find the point with the median y-coordinate among the points inside a query vertical slab. A decision version of the problem (deciding whether the median is less than a given value) is easily reducible to orthogonal range or dominance counting. We show that a simpler version of our online data structure for dominance counting queries with $O(\lg n)$ query time can answer range median queries in the same amount of time.

Theorem A.4 *We can preprocess n points in the plane in $O(n\sqrt{\lg n})$ time, using a data structure with $O(n)$ words of space, so that we can select the k-th lowest point among the points inside a query vertical slab for any given k in $O(\lg n)$ time.*

Proof: We use the same data structure as in the proof of Theorem 2.6 but with a different setting of parameters: $H = 1$ and $A = \lg n$. Case $0''$ disappears, and Case $0'$ becomes trivial. Thus, in Case 1, we revert to a simpler 2-way divide-and-conquer. The preprocessing time is $O(n\sqrt{\lg n})$ and space is $O(n)$.

Assume that we are a given the two x-ranks of the coordinates of the query slab σ. To select the k-th lowest point of $P \cap \sigma$, we (i) select the k-th lowest point $\tilde{q}$ of $\tilde{P} \cap \sigma$, (ii) compute the y-rank k' of $\tilde{q}$ in $\tilde{P} \cap \sigma$, and (iii) select the $(k - k' + 1)$-th lowest point of $P_i \cap \sigma$ with $i = \tilde{q}.y$.

In Case 1, step (i) takes $O(1)$ time by computing the y-rank of only two y-values (since $h = 1$) in $\tilde{P} \cap \sigma$, which reduces to dominance counting in $\tilde{P}$. Step (ii) takes $O(1)$ time, again by dominance counting in $\tilde{P}$. Step (iii) is done recursively. We thus get the same recurrence for the query time.

In Case 2, step (i) reduces to Case 1, step (ii) reduces to dominance counting in $\tilde{P}$, and step (iii) is done recursively. Again we get the same recurrence for the query time.

Note that initially the x-ranks can be determined more simply by binary search in $O(\lg n)$ time. □

An intriguing question we leave open is whether sublogarithmic (e.g., $O(\lg n/\lg\lg n)$) query time is possible for range median with $O(n \operatorname{polylog} n)$ space.

A.4 1-d online dynamic ranking and selection

In this subsection, we give a data structure for online dynamic ranking and selection with an improved update time. Roughly speaking, we need to rework the offline algorithms from Section 2 or A.1 to handle an online sequence of updates. For convenience, we will deviate somewhat from notation of the earlier sections, however.

We begin by designing a better data structure for maintaining partial sums. In this problem, we are to maintain an array $A[1\,..\,n]$ of positive integers under: incrementing or decrementing a location $A[i]$; asking for a partial sum: $\sum_{i=1}^{k} A[i]$; and selection query: asking for the predecessor of some x among the partial sums. That is, find k such that $\sum_{i=1}^{k} A[i] \le x < \sum_{i=1}^{k+1} A[i]$.

We can achieve an optimal $O(\lg n/\lg\lg n)$ query time, while improving the update time almost quadratically:

Theorem A.5 *We can maintain partial sums with $O(\lg n/\lg\lg n)$ time per query, and $O(\lg^{0.5+\varepsilon} n)$ time per update.*

Proof: Let $H = \varepsilon \lg w$. If we can maintain partial sums with $O(1)$ update and query time in an array of size 2^H, we can obtain $O((\lg n)/H)$ time per operation overall. Indeed, consider a balanced tree with degree 2^H spanning the array A. To each node, we associate the sum of all leaves in its subtree. A node stores a small partial sums data structure for the values associated to its children. An update translates into $O((\lg n)/H)$ updates on a root-to-leaf path, and a query into $O((\lg n)/H)$ queries.

Let us ignore selection queries for now. To speed up the updates, we will maintain the partial sums data structure in a node with $O(1)$ query time and subconstant amortized update time. Imagine first that we are working in external memory, and B words (of w bits each) can be manipulated in constant time. Then we can use buffer trees [Arg03] to delay the updates. Each node will hold a buffer of B recent updates that concern leaves in its subtree. In addition, it will maintain the partial sums of its children before these recent updates.

As long as the buffer of the root is not full, we can simply insert a new update there. When it gets filled, we recompute the partial sums array (incorporating the B updates), and distribute the corresponding

updates down to the 2^H children of the root. Recurse if any child's buffer gets filled. Emptying a buffer takes $O(2^H)$ time, and each of B updates is pushed down one level. Suppose $n \leq 2^L$. Overall, an update is pushed down at most $O(L/H)$ times, so its amortized cost is $O(1 + (1/B) \cdot 2^H L/H)$. The queries are only slowed down by a constant factor, since the buffer of recent updates can be examined in constant time at each level. For $B = w/L$, we can indeed pack B elements into a word. Thus, we get amortized update cost $O(1 + 2^H L/H \cdot L/w)$, which is at most $O(1 + 2^H L^2/w)$, and query cost $O(L/H)$.

To handle the case $n > 2^L$, we solve the partial sums problem using a tree of depth $(\lg n)/L$, and branching factor 2^L. In each node, we are facing a small partial sums problem, which can be solved above. Overall, we get amortized update cost $O((\lg n)/L \cdot [1 + 2^H L^2/w])$, and query cost $O((\lg n)/L \cdot L/H) = O((\lg n)/H)$. Setting $L = \sqrt{w}$, $w = \varepsilon \log n$, and $H = \varepsilon \log w$ gives update time $O(\lg^{0.5+\varepsilon} n)$ and query time $O(\lg n / \lg \lg n)$.

Finally, we discuss the selection query. In each node of the 2^H-ary tree, we maintain the partial sums array (ignoring recent updates) as a fusion data structure [FW93]. This allows for constant-time predecessor search. The reconstruction time is slowed down to $2^{O(H)}$, but this is inconsequential (except in the dependence on the constant ε).

It is not immediately possible to answer the selection query based on the predecessor and the recent updates, but there is a standard trick to solve this issue [PD06]. In a packed word, we also remember the value of each child, capped to at most $B + 1$. The selection query can be answered from the predecessor and the capped values, since we are only considering B recent updates (see [PD06]). The capped values fit in a word, since the only require $O(2^H \lg B) = O(\lg^{0.5+2\varepsilon} n)$ bits. □

We now turn our attention to dynamic algorithms for the rank problem. The goal is to support a list of n elements under: querying the rank of a given element; querying for the k-th element (selection queries); deleting a given element; and inserting an element after a given element.

Theorem A.6 *There exists a data structure for dynamic rank supporting updates in $O(\lg^{0.5+\varepsilon} n)$ time and queries in $O(\lg n / \lg \lg n)$ time.*

Proof: This proceeds by reduction to partial sums. We recall the ordered file data structure of [DS87, BCD+02]. This data structure can maintain n objects inside an array of size $O(n)$ under insertions and deletions, always preserving a strict ordering of the elements. At insertion time, an amortized $O(\lg^2 n)$ elements are moved to create space for the inserted item.

We will group our list elements into buckets of $\Theta(\lg^2 n)$ consecutive elements, each of which is maintained as a balanced binary search tree. Each bucket is stored as an element in an ordered file data structure. On top of the $O(n)$ array of the ordered file data structure, we store a partial sums data structure (the base elements are the exact size of each bucket).

To insert or delete an element, we first insert/delete in the binary search tree of the bucket. Then, the count of the bucket size is updated in the partial sums structure. If the bucket has grown or shrunk by a constant factor, we break it or merge two consective buckets. An element must be inserted or removed into the ordered file structure. However, one such update is done for $O(\lg^2 n)$ original updates, so the amortized cost is constant. Overall, the time is equal to the update in the partial sums structure, plus $O(\lg \lg n)$.

To query the rank of the element, we first run a partial sums query up to its bucket, and add the rank within the bucket. To select an element, we select in the partial sums data structure, and finish off in the corresponding bucket. In both cases, the query time is the partial sums query, plus $O(\lg \lg n)$. □